

27th International Conference on DNA Computing and Molecular Programming

DNA 27, September 13–16, 2021, Oxford, UK
(Virtual Conference)

Edited by

Matthew R. Lakin

Petr Šulc



Editors

Matthew R. Lakin 

Department of Computer Science, Department of Chemical & Biological Engineering,
Center for Biomedical Engineering, University of New Mexico, Albuquerque, NM, USA
mlakin@cs.unm.edu

Petr Šulc 

School of Molecular Sciences, Arizona State University, Tempe, AZ, USA
psulc@asu.edu

ACM Classification 2012

Theory of computation → Models of computation; Applied computing → Molecular structural biology;
Applied computing → Biological networks; Information systems → Information storage systems

ISBN 978-3-95977-205-1

Published online and open access by

Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern,
Germany. Online available at <https://www.dagstuhl.de/dagpub/978-3-95977-205-1>.

Publication date

September, 2021

Bibliographic information published by the Deutsche Nationalbibliothek

The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed
bibliographic data are available in the Internet at <https://portal.dnb.de>.

License

This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0):
<https://creativecommons.org/licenses/by/4.0/legalcode>.



In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work
under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.DNA.27.0

ISBN 978-3-95977-205-1

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

LIPICs – Leibniz International Proceedings in Informatics

LIPICs is a series of high-quality conference proceedings across all fields in informatics. LIPICs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Luca Aceto (*Chair*, Reykjavik University, IS and Gran Sasso Science Institute, IT)
- Christel Baier (TU Dresden, DE)
- Mikolaj Bojanczyk (University of Warsaw, PL)
- Roberto Di Cosmo (Inria and Université de Paris, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Daniel Král' (Masaryk University - Brno, CZ)
- Meena Mahajan (Institute of Mathematical Sciences, Chennai, IN)
- Anca Muscholl (University of Bordeaux, FR)
- Chih-Hao Luke Ong (University of Oxford, GB)
- Phillip Rogaway (University of California, Davis, US)
- Eva Rotenberg (Technical University of Denmark, Lyngby, DK)
- Raimund Seidel (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

■ Contents

Preface	
<i>Matthew R. Lakin and Petr Šulc</i>	0:vii

Organization

Steering Committee	0:ix
Program Committee	0:x
Additional Reviewers for Tracks A and B	0:xi
Organizing Committee for DNA 27	0:xii
Sponsors	0:xiii

Regular Papers

Robust Digital Molecular Design of Binarized Neural Networks <i>Johannes Linder, Yuan-Jyue Chen, David Wong, Georg Seelig, Luis Ceze, and Karin Strauss</i>	1:1–1:20
Computing Properties of Thermodynamic Binding Networks: An Integer Programming Approach <i>David Haley and David Doty</i>	2:1–2:16
Self-Replication via Tile Self-Assembly (Extended Abstract) <i>Andrew Alseth, Daniel Hader, and Matthew J. Patitz</i>	3:1–3:22
Improved Lower and Upper Bounds on the Tile Complexity of Uniquely Self-Assembling a Thin Rectangle Non-Cooperatively in 3D <i>David Furcy, Scott M. Summers, and Logan Withers</i>	4:1–4:18
ENSnano: A 3D Modeling Software for DNA Nanostructures <i>Nicolas Levy and Nicolas Schabanel</i>	5:1–5:23
Directed Non-Cooperative Tile Assembly Is Decidable <i>Pierre-Étienne Meunier and Damien Regnault</i>	6:1–6:21
Molecular Machines from Topological Linkages <i>Keenan Breik, Austin Luchsinger, and David Soloveichik</i>	7:1–7:20
Small Tile Sets That Compute While Solving Mazes <i>Matthew Cook, Tristan Stérin, and Damien Woods</i>	8:1–8:20



Predicting Minimum Free Energy Structures of Multi-Stranded Nucleic Acid Complexes Is APX-Hard <i>Anne Condon, Monir Hajiaghayi, and Chris Thachuk</i>	9:1–9:21
Reactamole: Functional Reactive Molecular Programming <i>Titus H. Klinge, James I. Lathrop, Peter-Michael Osera, and Allison Rogers</i>	10:1–10:20
Parallel Pairwise Operations on Data Stored in DNA: Sorting, Shifting, and Searching <i>Tonglin Chen, Arnav Solanki, and Marc Riedel</i>	11:1–11:21

■ Preface

This volume contains the papers presented at DNA 27: the 27th International Conference on DNA Computing and Molecular Programming. The conference was originally scheduled to be held at the University of Oxford, but due to the ongoing COVID-19 pandemic it was changed to an online format. The virtual conference was held during September 13–16, 2021, and was organized under the auspices of the International Society for Nanoscale Science, Computation, and Engineering (ISNSCE). The DNA conference series aims to draw together researchers from the fields of mathematics, computer science, physics, chemistry, biology, and nanotechnology to address the analysis, design, and synthesis of information-based molecular systems.

Papers and presentations were sought in all areas that relate to biomolecular computing, including, but not restricted to: algorithms and models for computation on biomolecular systems; computational processes *in vitro* and *in vivo*; molecular switches, gates, devices, and circuits; molecular folding and self-assembly of nanostructures; analysis and theoretical models of laboratory techniques; molecular motors and molecular robotics; information storage; studies of fault-tolerance and error correction; software tools for analysis, simulation, and design; synthetic biology and *in vitro* evolution; and applications in engineering, physics, chemistry, biology, and medicine.

Authors who wished to orally present their work were asked to select one of two submission tracks: Track A (full paper) or Track B (one-page abstract with supplementary document). Track B is primarily for authors submitting experimental or theoretical results who plan to submit to a journal rather than publish in the conference proceedings. We received 33 submissions for oral presentations: 17 submissions to Track A and 16 submissions to Track B. Each submission was reviewed by at least three reviewers, with most reviewed by four or more reviewers. The Program Committee accepted 11 papers for Track A (65%) and 11 papers for Track B (69%). We also received 29 submissions for Track C (poster), of which five were selected as additional oral presentations by the Program Committee. This volume contains the papers accepted for Track A.

We express our sincere appreciation to our invited speakers: Michael Brenner, Luca Cardelli, Chengde Mao, Petra Schwill, Friedrich Simmel, and Reidun Twarock. We thank all of the authors who contributed papers to these proceedings at a difficult time, and who presented papers and posters during the conference. Last, but by no means least, the editors are especially grateful to the members of the Program Committee and the additional invited reviewers for their hard work in reviewing the papers on a tight deadline and for providing insightful and constructive comments to the authors.

Matthew Lakin
Petr Šulc

September 2021



■ Organization

Steering Committee

Anne Condon (Chair)	University of British Columbia, Canada
Luca Cardelli	University of Oxford, UK
Masami Hagiya	University of Tokyo, Japan
Natasha Jonoska	University of South Florida, USA
Chengde Mao	Purdue University, USA
Satoshi Murata	Tohoku University, Japan
John H. Reif	Duke University, USA
Grzegorz Rozenberg	University of Leiden, The Netherlands
Rebecca Schulman	Johns Hopkins University, USA
Nadrian C. Seeman	New York University, USA
Friedrich Simmel	Technical University Munich, Germany
David Soloveichik	University of Texas at Austin, USA
Andrew J. Turberfield	University of Oxford, UK
Erik Winfree	California Institute of Technology, USA
Damien Woods	Maynooth University, Ireland
Hao Yan	Arizona State University, USA



Program Committee

Matthew Lakin (Co-chair)	University of New Mexico, USA
Petr Šulc (Co-chair)	Arizona State University, USA
Stefan Badelt	University of Vienna, Austria
Jonathan Bath	University of Oxford, UK
Luca Cardelli	University of Oxford, UK
Ho-Lin Chen	National Taiwan University, Taiwan (R.O.C.)
Yuan-Jyue Chen	Microsoft Research, Redmond, USA
Anne Condon	University of British Columbia, Canada
David Doty	University of California, Davis, USA
Jonathan Doye	University of Oxford, UK
Constantine Evans	Evans Foundation and Maynooth University, Ireland
Elisa Franco	University of California, Los Angeles, USA
Cody Geary	Aarhus University, Denmark
Manoj Gopalkrishnan	Indian Institute of Technology, Bombay, India
Elton Graunard	Boise State University, USA
Masami Hagiya	University of Tokyo, Japan
Lila Kari	University of Waterloo, Canada
Titus Klinge	Drake University, USA
Satoshi Kobayashi	University of Electro-Communications, Tokyo, Japan
James Lathrop	Iowa State University, USA
Chenxiang Lin	Yale University, USA
Satoshi Murata	Tohoku University, Japan
Eyal Nir	Ben Gurion University, Israel
Pekka Orponen	Aalto University, Finland
Matthew Patitz	University of Arkansas, USA
Lulu Qian	California Institute of Technology, USA
John H. Reif	Duke University, USA
Flavio Romano	Ca Foscari University of Venice, Italy
Lorenzo Rovigatti	Sapienza University of Rome, Italy
Dominic Scalise	California Institute of Technology, USA
Nicolas Schabanel	CNRS and École Normale Supérieure de Lyon, France
Joseph Schaeffer	Google Health, USA
Robert Schweller	University of Texas Rio Grande Valley, USA
Shalin Shah	Bloomberg, USA
William Shih	Harvard University, USA
David Soloveichik	University of Texas at Austin, USA
Darko Stefanovic	University of New Mexico, USA
Jaimie Stewart	California Institute of Technology, USA
Chris Thachuk	University of Washington, USA
Grigory Tikhomirov	University of California Berkeley, USA
Andrew Turberfield	University of Oxford, UK
Shelley Wickham	University of Sydney, Australia
Damien Woods	Maynooth University, Ireland
Fei Zhang	Rutgers University, USA

Additional Reviewers for Tracks A and B

Andrew Alseth
David Caballero
Christian Cuba Samaniego
Timothy Gomez
Leopold Green

Daniel Hader
Jacob Hendricks
Trent Rogers
Scott Summers
Xun Tang

Organizing Committee for DNA 27

Andrew Phillips (Co-chair)	Microsoft Research, Cambridge, UK
Andrew Turberfield (Co-chair)	University of Oxford, UK
Claire Garland	Institute of Physics, UK

Sponsors

International Society for Nanoscale Science, Computation, and Engineering
Biological Physics Group, Institute of Physics
Department of Physics, University of Oxford
Microsoft Research

Robust Digital Molecular Design of Binarized Neural Networks

Johannes Linder ✉

University of Washington, Paul G. Allen School of Computer Science and Engineering,
Seattle, WA, USA

Yuan-Jyue Chen

Microsoft Research, Redmond, WA, USA

David Wong

University of Washington, Department of Bioengineering, Seattle, WA, USA

Georg Seelig

University of Washington, Paul G. Allen School of Computer Science and Engineering,
Seattle, WA, USA

University of Washington, Department of Electrical and Computer Engineering,
Seattle, WA, USA

Luis Ceze

University of Washington, Paul G. Allen School of Computer Science and Engineering,
Seattle, WA, USA

Karin Strauss

Microsoft Research, Redmond, WA, USA

University of Washington, Paul G. Allen School of Computer Science and Engineering,
Seattle, WA, USA

Abstract

Molecular programming – a paradigm wherein molecules are engineered to perform computation – shows great potential for applications in nanotechnology, disease diagnostics and smart therapeutics. A key challenge is to identify systematic approaches for compiling abstract models of computation to molecules. Due to their wide applicability, one of the most useful abstractions to realize is neural networks. In prior work, real-valued weights were achieved by individually controlling the concentrations of the corresponding “weight” molecules. However, large-scale preparation of reactants with precise concentrations quickly becomes intractable. Here, we propose to bypass this fundamental problem using Binarized Neural Networks (BNNs), a model that is highly scalable in a molecular setting due to the small number of distinct weight values. We devise a noise-tolerant digital molecular circuit that compactly implements a majority voting operation on binary-valued inputs to compute the neuron output. The network is also rate-independent, meaning the speed at which individual reactions occur does not affect the computation, further increasing robustness to noise. We first demonstrate our design on the MNIST classification task by simulating the system as idealized chemical reactions. Next, we map the reactions to DNA strand displacement cascades, providing simulation results that demonstrate the practical feasibility of our approach. We perform extensive noise tolerance simulations, showing that digital molecular neurons are notably more robust to noise in the concentrations of chemical reactants compared to their analog counterparts. Finally, we provide initial experimental results of a single binarized neuron. Our work suggests a solid framework for building even more complex neural network computation.

2012 ACM Subject Classification Theory of computation → Models of computation; Applied computing

Keywords and phrases Molecular Computing, Neural Network, Binarized Neural Network, Digital Logic, DNA, Strand Displacement

Digital Object Identifier 10.4230/LIPIcs.DNA.27.1



© Johannes Linder, Yuan-Jyue Chen, David Wong, Georg Seelig, Luis Ceze, and Karin Strauss;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on DNA Computing and Molecular Programming (DNA 27).

Editors: Matthew R. Lakin and Petr Šulc; Article No. 1; pp. 1:1–1:20

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Computing in molecules is a prerequisite for a wide range of potentially revolutionizing nanotechnologies, such as molecular nanorobots and smart therapeutics. For example, molecular “programs” delivered to cells could collect sensory input from gene expression to determine the prevalence of disease and conditionally release a drug. Molecular disease classifiers have already been demonstrated outside of cells [1, 18]. Moreover, as researchers look toward synthetic DNA for storing data [9, 20, 4], it becomes increasingly important to develop operations that can be executed on data in molecular form.

Bringing such applications closer to reality requires abstractions and programming languages that can capture the desired molecular behaviors. A methodical way to reason about chemical computing is through the formalism of chemical reaction networks (CRNs). A CRN is a collection of coupled chemical reactions, each consuming a set of reactants to create a set of products. CRNs are Turing universal [27, 12] and any CRN can in principle be implemented using synthetic DNA molecules, specifically DNA strand displacement [28, 7].

Molecular computing has traditionally been approached from two different design philosophies; in *analog* computing, the numerical value of each variable is encoded in the concentration of its corresponding molecular species [24, 5, 33]. Conversely, in *digital* molecular computing, concentrations of species representing logical values are restricted to “high” or “low” ranges similar to voltages in digital electronics [13, 19]. Feed-forward neural networks have previously been demonstrated as analog molecular circuits [34, 23, 6, 8, 32]. While compact, analog molecular circuits are inherently sensitive to concentration noise, as such perturbation directly impacts the correctness of the system.

Here, we develop a digital molecular implementation of binarized neural networks, a class of models where the inputs, weights and activations are constrained to take the values $\{+1, -1\}$ [14]. We devise an efficient molecular circuit for computing binary n -input majority using $\mathcal{O}(n^2)$ gates and $\mathcal{O}(\log n)$ (optimal) depth. Each neuron uses this circuit for their weighted sum and threshold operation. This design offers a uniquely scalable molecular implementation: while an analog monotonic network requires exponentially large concentrations of molecular substrates as a function of layer depth to avoid saturation of outputs (we develop this argument in Section 3), our design uses a constant concentration across all network layers. We demonstrate our design on the MNIST task, both as idealized CRNs and as DNA strand displacement cascades. Finally, we compare the digital neuron to an analog, rate-independent HardTanh-activated neuron and present improved robustness to concentration noise and leak. We also present initial experimental results of a simple 2-input binarized neuron.

2 Background

2.1 Binarized Neural Networks

In this paper we focus on deterministic binarized neural networks (BNNs) as described by [14]. If we first consider a single neuron, its inputs x_i , weights w_i , bias term b and activation y are binary-valued and constrained to $\{+1, -1\}$. We compute neuron activation y as:

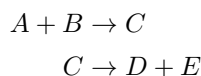
$$y = \begin{cases} +1 & \text{if } b + \sum_{i=1}^N w_i \cdot x_i > 0 \\ -1 & \text{else} \end{cases} \quad (1)$$

Binarized neurons are assembled into fully connected neural networks by treating output y of a neuron i in layer k as the input x_i to neuron j of layer $k + 1$, connected through weight $w_{ij}^{(k)} \in \{+1, -1\}$. BNNs can be efficiently trained by gradient descent following the

scheme of the original authors, where a straight-through estimator was used to propagate gradients through binarized non-linearities. While BNNs usually are less accurate than their full-precision counterparts, they offer substantial improvements in execution time and storage space [26]. Since BNN computation can be executed with bitwise operations, they are also amenable to efficient implementation on FPGAs and ASICs. Recent BNN architectures show relatively good performance on more complex tasks such as ImageNet classification [10].

2.2 Chemical Reaction Networks

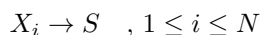
Chemical Reaction Networks (CRNs) are a mathematical formalism traditionally used by chemists to describe how the concentration or counts of chemical species evolve over time. Two example reactions are shown below:



Reaction 1 dictates that one molecule each of A and B react to produce C, until A or B are fully consumed. Reaction 2 consumes one molecule of C to produce one molecule each of D and E. Commonly, a rate constant that captures how fast an instance of a reaction occurs is also associated with each reaction. In this paper, we consider reactions with at most two reactants or two products. When modeling the time evolution of CRNs, we often reason about concentrations. We denote the concentration of A at time t as $a(t)$. Assuming mass action kinetics, $a(t)$ can be modeled as an ODE [11].

Analog and Digital CRNs

Molecular programming with CRNs can be approached from different computational models. Two of the most widely used models are *analog* CRNs and *digital* CRNs. In analog CRNs, any non-negative real-valued variable $x \in \mathbb{R}^+$ is represented by a corresponding molecular species X . The value of x is encoded in the concentration $x(t)$ [24]. For example, $x = 3.2$ is represented as $x(t) = 3.2\text{nM}$. This convention makes summation trivial to implement; to compute $s = \sum_{i=1}^N x_i$, we add N reactions translating species X_i to the same species S :



As time progresses, each input X_i accumulates in species S such that $t \rightarrow \infty s(t) = \sum_{i=1}^N x_i(0)$.

In a digital computing paradigm, we restrict variables to be discrete, such that they can only take on a finite number of states. Each discrete variable and state is encoded by its own molecular species (e.g. Boolean variable x is encoded by two species, $X^{(\text{on})}$ and $X^{(\text{off})}$). Chemical concentrations represent only high or low signals indicating which state is active. For example, Boolean variable x is modeled by the following CRN convention (Here $T^{(\text{high})}$ is a constant representing the signal filter cutoff and the third case represents incorrect states):

$$x = \begin{cases} \text{on} & \text{if } x^{(\text{on})}(t) \geq T^{(\text{high})} \text{ and } x^{(\text{off})}(t) < T^{(\text{high})} \\ \text{off} & \text{if } x^{(\text{on})}(t) < T^{(\text{high})} \text{ and } x^{(\text{off})}(t) \geq T^{(\text{high})} \\ \text{(undefined)} & \text{else} \end{cases}$$

Note that N -input summation is more complicated under this paradigm, as inputs can no longer simply be translated to the same output species to encode logical sum. Instead, proper digital circuits such as carry adders have been implemented as CRNs [16, 22].

Piecewise Linear CRNs and HardTanh

In analog CRNs, a real-valued variable $x \in \mathbb{R}$ that can take on negative values is often represented in dual-rail form by two species X^- and X^+ . The value of x is encoded as the difference in their concentration, $x = \lim_{t \rightarrow \infty} x^+(t) - x^-(t)$. Using this convention, the function $h = \max(x, -k)$ can be implemented by two reactions [5] (the initial concentration of K is set to $k(0) = k$ and $x^+(0), x^-(0)$ are initialized such that $x^+(0) - x^-(0) = x$):



Similarly, function $y = \min(h, m)$ can be implemented as (here $m(0) = m$ etc.):



Note that $k, m \in \mathbb{R}^+$, so dual-rail is not needed to express their values. By stacking these two sets of reactions and setting $k(0) = 1$ and $m(0) = 1$, we can implement a rate-independent, analog HardTanh function, $y = \min(\max(x, -1), 1)$. We use this construction below when comparing the digital neuron developed in this paper to a HardTanh-activated neuron based on analog CRNs, which is similar to the ReLU (Rectified Linear Unit) network by [32].

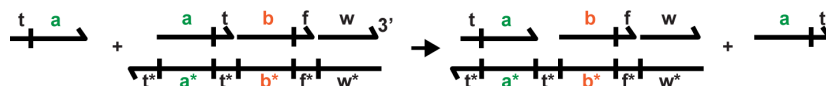
2.3 DNA Strand Displacement

Toehold-mediated DNA strand displacement (DSD) is a framework capable of synthesizing any CRN [28, 7, 30]. Molecular species are compiled into signal strands which react through synthetic DNA gates as specified by the CRN reactions. In this paper we use the *two-domain* architecture from [3], which supports all of the functionality needed to implement our digital BNN design; the gates can be prepared at large scale from double-stranded DNA by enzymatic processing and the architecture supports AND-logic, catalytic amplification and fan-out operations.

Two-domain gates work by exposing a toehold which, when hybridized by an input strand, triggers a sequence of displacements. These displacements ultimately release the output strand bound to the gate. As an example, consider the simple CRN reaction $A \rightarrow B$. The corresponding two-domain DSD system is shown in Figure 1. Species A is represented by the signal strand to the left (domains t and a – abbreviated *ta*) and B is represented by the strand with domains b and f (*bf*). A sequence of displacements mediated by toehold t eventually releases the output strand from the gate. Specifically, the sequence of displacements are:

1. Input strand *ta* binds to gate *G*, displacing strand *at* and exposing inner toehold *t*. (Reversible reaction)
2. Helper strand *tb* binds to the newly opened toehold, displacing output strand *bf* and exposing inner toehold *f*. (Reversible reaction)
3. Helper strand *fw* binds to the newly opened toehold, displacing strand *w* and closing gate *G*. (Irreversible reaction)

A promising alternative for synthesizing CRNs is Polymerase-mediated strand displacement (PSD), where polymerase enzymes trigger initially single-stranded DNA gates by extending hybridized input strands and consequently displacing any output strand [29, 25]. In this paper, we focus solely on implementing the neural network with enzyme-free toehold-mediated DSD, but we discuss implementation with PSD towards the end.



■ **Figure 1** Input strand ta releases strand at and exposes new toehold t . Additional steps of strand displacement (not shown) ultimately release output strand bf . Additional helper strands tb and fw (not shown) help carry out the sequence of displacement reactions.

3 Related Work

Neural networks with real-valued weights based on analog computation have previously been built with DNA strand displacement cascades [23, 8, 6]. Non-linear activation was achieved by introducing threshold gates with much faster reaction rates than gates producing an output signal. A recent paper by [32] proposed a rate-independent analog CRN to implement a neural network with binary-valued weights and ReLU activations.

Neural networks based on monotonic analog CRNs may require exponentially large concentrations of molecular substrates at deeper network layers to avoid saturation. To see why, assume a binary-weighted, ReLU-activated network consisting of K layers and N neurons per layer. If we set all weights w_{ij}^k to $+1$ and all inputs x_i^0 to $+1$, then each activation in the first layer becomes:

$$x_j^1 = \max \left(\sum_{i=1}^N w_{ij}^0 \cdot x_i^0, 0 \right) = N.$$

To physically implement this computation, we require a DSD gate or other substrate that can produce the molecular species for x_j^1 at a concentration N times larger than x_i^0 . Inductively, at the final layer, each activation x_j^K can be as large as N^K , requiring gates with concentrations proportional to N^K to avoid saturation. We can similarly construct a worst-case example for the HardTanh function from the previous section; by setting half the weights w_{ij}^k to $+1$ and half the weights to -1 , the concentration of output species Y^+ / Y^- of Reaction Set 3 will be $N/2$. With the same inductive argument as before, we need gate concentrations proportional to $(N/2)^K$ to avoid saturation.

Our design differs from that of [23] and [32] in that the CRN computation is carried out with digital logic. Since concentrations are used only to represent high or low signals, the computation remains correct under perturbation, similar in concept to how electronic digital circuits offer more robust behavior than analog electronic circuits. The digital design also makes the system rate-independent and allows for a uniform gate concentration across all network layers.

4 A Digital Molecular Implementation of Binarized Neurons

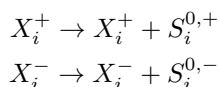
A binarized neuron with inputs $x_i \in \{+1, -1\}$, weights $w_i \in \{+1, -1\}$ and bias $b \in \{+1, -1\}$ is illustrated in Figure 2A. To simplify the implementation, we assume the number of inputs, N , is a power of 2. The neuron consists of a sequence of three operations:

1. Weight operations $s_i^0 = w_i \cdot x_i$, where s_i^0 denotes weighted input i before summation.
2. A sum operation $s = \sum_{i=1}^N s_i^0$.
3. A sign operation $y = \text{sign}(s + b)$, where b breaks ties in the case when $s = 0$.

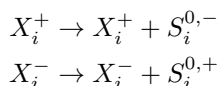
Alternatively, steps 2 and 3 may be considered a majority voting operation between positive and negative inputs. The following sections describe how we implement the computational graph shown in Figure 2B of a digital neuron with chemical reactions. We only discuss a single neuron here to keep the notation light, but a generalization to an arbitrarily sized BNN is described in Appendix A.

4.1 Weight

Each input x_i is represented by two CRN species, X_i^+ and X_i^- , corresponding to states $x_i = +1$ and $x_i = -1$ respectively. We implement the weight operation $s_i^0 = w_i \cdot x_i$ as follows (Figure 2C): If $w_i = +1$, we add catalytic reactions translating the **positive-state** input species X_i^+ to the **positive-state** weighted species $S_i^{0,+}$, and we similarly translate X_i^- to $S_i^{0,-}$.



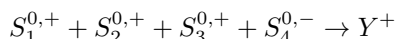
If $w_i = -1$, we instead translate X_i^+ to $S_i^{0,-}$, and vice versa for the negative-state input species.



In practice, the catalytic reactions do not replenish the input species indefinitely, but rather transform some gate substrate G into waste W (e.g. $X + G \rightarrow X + W + Y$ for input species X and output Y), but this does not matter for our theoretical analysis.

4.2 Majority Vote

If there are more weighted species $s_i^0 = w_i \cdot x_i$ in a positive state than a negative state, the circuit should output a positive state, and vice versa. We can easily enumerate every such rule as a chemical reaction of N -ary AND-clauses, given the CRN species $S_i^{0,+}$ and $S_i^{0,-}$ of each weighted input. For example, to handle the case $\mathbf{s}^0 = \mathbf{w} * \mathbf{x} = (+1, +1, +1, -1)$, we would add the reaction:



This CRN is problematic for two reasons: First, the number of reactions grows exponentially with N . Second, the CRN requires arbitrary-length AND-clauses, which is not feasible when implemented as DNA strand displacement gates. Instead, we here devise an efficient digital majority voting CRN that requires only a quadratic number of gates and a logarithmic (optimal) depth. We will compute the sum $s = \sum_{i=1}^N s_i^0$ as a balanced tree of binary additions using a recursive definition (Figure 2B):

$$s_h^k = s_{2h-1}^{k-1} + s_{2h}^{k-1}$$

Here $k = 1, \dots, \log(N)$ and $h = 1, \dots, N/2^k$. At level $k = \log(N)$, the full sum s is stored in $s_1^{\log(N)}$. For each binary addition performed during the recursion, we will add chemical reactions translating every possible combination of discrete values of s_{2h-1}^{k-1} and s_{2h}^{k-1} into the species of s_h^k corresponding to their sum. Suppose the summands can take on m discrete

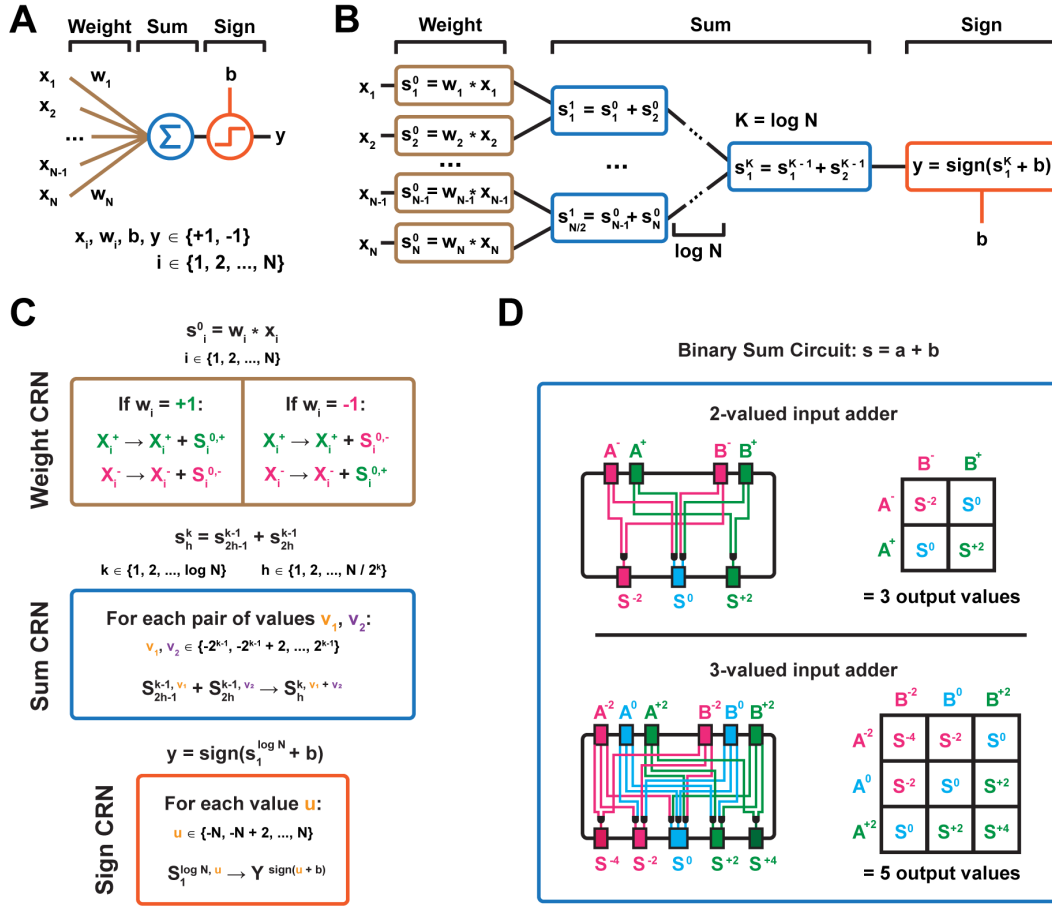
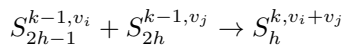


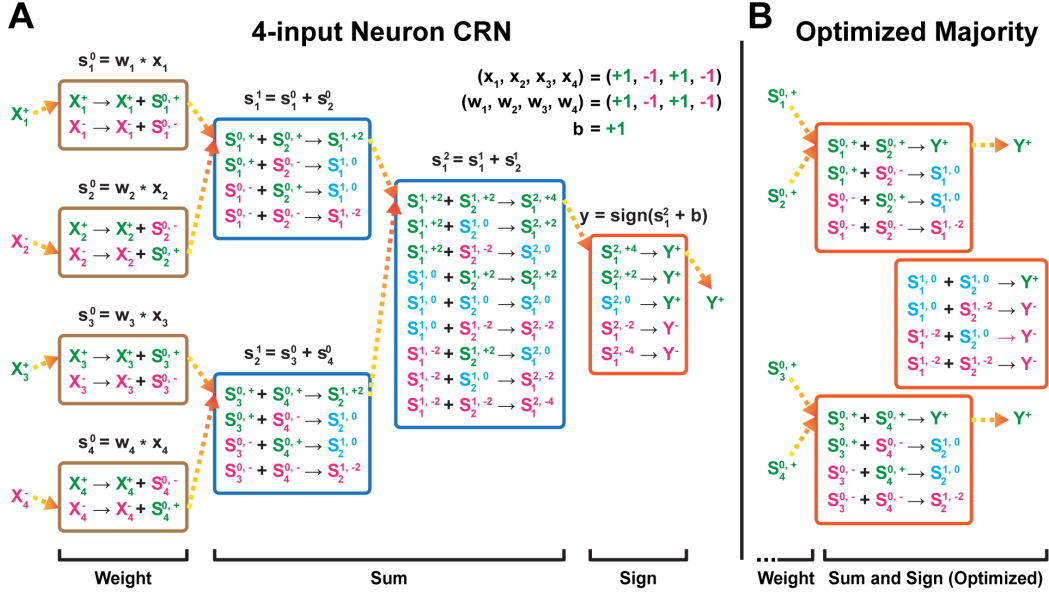
Figure 2 **A** Illustration of a binarized neuron. **B** Computational graph of a binarized neuron. The binary summation tree (blue) has a depth of $\log(N)$. **C** Chemical reactions which implement the modules of the computational graph. **D** Example illustration of the binary addition circuit. A set of AND rules encode which input states (summands) correspond to what output state (sum).

values, $s_{2h-1}^{k-1}, s_{2h}^{k-1} \in \{v_1, \dots, v_m\}$. We represent these states with species $S_{2h-1}^{k-1, v_1}, \dots, S_{2h-1}^{k-1, v_m}$ and $S_{2h}^{k-1, v_1}, \dots, S_{2h}^{k-1, v_m}$. For every combination of values v_i, v_j , add the following reaction (Figure 2C):



Example logical circuits are illustrated in Figure 2D for $m = 2$ and $m = 3$. In general, if the input variables have cardinality m , we require m^2 reactions and the cardinality of the output variable becomes $2m - 1$. Note that this circuit is more similar to a demultiplexer than a carry adder.

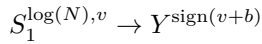
Since each binary addition has depth 1, and there are $\log(N)$ levels of additions, the total depth is $\log(N)$. To calculate the total number of reactions, we first note that, at level k , there are $N/2^k$ additions. Each summand has cardinality $m = 2^{k-1} + 1$ (assuming binary-valued initial inputs). Hence, the total number of reactions across all $\log(N)$ levels can be calculated as a geometric series:



■ **Figure 3** **A** Example CRN execution of a 4-input neuron. $(x_1, x_2, x_3, x_4) = (+1, -1, +1, -1)$ and $(w_1, w_2, w_3, w_4) = (+1, -1, +1, -1)$. **B** Optimized 4-input neuron (weight reactions are omitted).

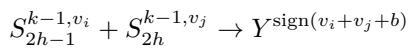
$$\begin{aligned}
 \sum_{k=1}^{\log(N)} N \cdot \frac{(2^{k-1} + 1)^2}{2^k} &= N \cdot \sum_{k=0}^{\log(N)-1} \frac{2^{2k} + 2^{k+1} + 1}{2^k} \\
 &= \frac{N^2}{2} + N \cdot \log(N) + \frac{N}{2} - 1 \\
 &= \mathcal{O}(N^2)
 \end{aligned}$$

We have thus shown that we can construct a digital sum circuit as a balanced binary tree of chemical reactions with $\mathcal{O}(\log(N))$ depth and $\mathcal{O}(N^2)$ reactions. We finalize the majority voting circuit by adding uni-molecular reactions translating every possible state species $S_1^{\log(N), -N}, \dots, S_1^{\log(N), +N}$ of the final sum $s_1^{\log(N)}$ into the correct signed output species Y^+ or Y^- (N reactions):



4.3 An Illustrative Example: A 4-Input Binarized Neuron

We show the digital CRN of a 4-input binarized neuron in Figure 3A, which carries out the full sum before applying the sign reaction. However, we are ultimately not interested in computing the sum of input species, only their majority vote. If the absolute value of the partial sum in any of the sub trees is greater than $N/2$, we can stop computing the sum since the majority is already determined. That is, if $|v_i + v_j + b| > N/2$, alter the reaction to immediately produce the output species Y^+ or Y^- :



■ **Table 1** Number of CRN reactions required to compute N -input digital majority.

N	2	4	8	16	32
# Reactions	4	12	50	182	654

Similarly, it is unnecessary to translate the final sums into their corresponding signs with separate uni-molecular reactions; we can immediately produce the output species Y^+ , Y^- from the last level of sums. An optimized 4-input binarized neuron is illustrated in Figure 3B. The optimization reduces the input cardinality of the two summands at the final level by 1. Using the geometric series defined in Section 4.2, we can calculate the number of removed reactions at level $k = \log(N)$:

$$N \cdot \frac{(2^{\log(N)-1} + 1)^2}{2^{\log(N)}} - N \cdot \frac{(2^{\log(N)-1})^2}{2^{\log(N)}} = N + 1$$

The optimized circuit thus requires $\frac{N^2}{2} + N \cdot \log(N) - \frac{N}{2} - 2$ reactions to compute digital majority. Table 1 lists the number of required reactions up to $N = 32$ inputs.

4.4 DNA Strand Displacement Design

Here we present the DNA strand displacement (DSD) implementation of the digital BNN CRN. The implementation is based on the *two-domain* design of [3]. The complete DSD schematic is shown in Figure 4. The implementation is described in detail below.

Each activation x_i^l in layer l is represented by two input strands, $X_i^{l,+}$ and $X_i^{l,-}$. For each weighted connection $s_{i,j}^{l,0} = w_{i,j}^l \cdot x_i^{l-1}$, we add four gates. If $w_{i,j}^l = +1$, we add the following gates:

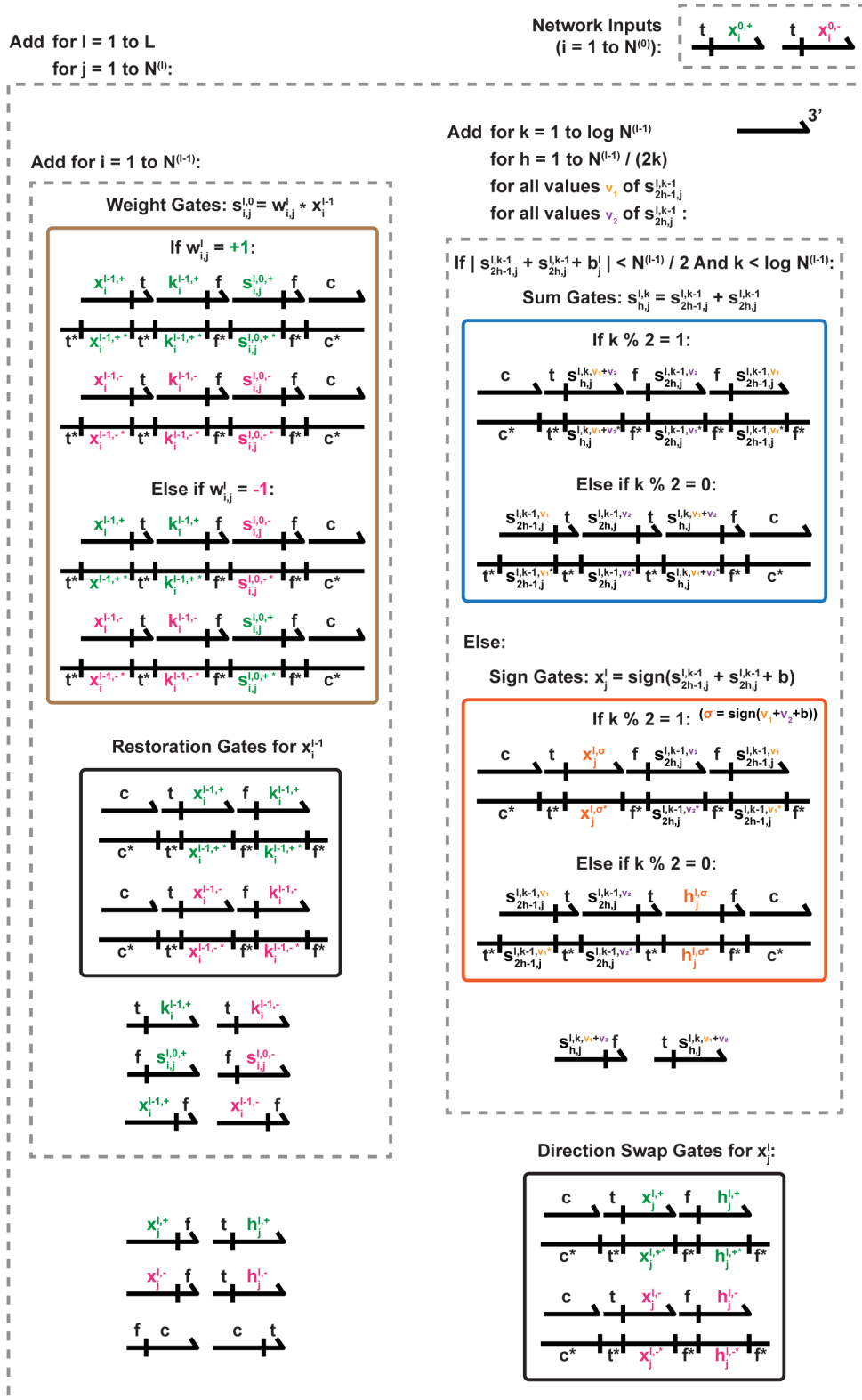
1. Gate $G_{\text{Weight},i,j}^{l,+}$, which outputs $K_i^{l-1,+}$ and $S_{i,j}^{l,0,+}$ given the strand $X_i^{l-1,+}$ as input.
2. Gate $G_{\text{Restore},i}^{l-1,+}$, which translates $K_i^{l-1,+}$ back to $X_i^{l-1,+}$.
3. Gate $G_{\text{Weight},i,j}^{l,-}$, which outputs $K_i^{l-1,-}$ and $S_{i,j}^{l,0,-}$ given the strand $X_i^{l-1,-}$ as input.
4. Gate $G_{\text{Restore},i}^{l-1,-}$, which translates $K_i^{l-1,-}$ back to $X_i^{l-1,-}$.

If $w_{i,j}^l = -1$, we swap the output strands $S_{i,j}^{l,0,+}$ and $S_{i,j}^{l,0,-}$ such that they are released by gates $G_{\text{Weight},i,j}^{l,-}$ and $G_{\text{Weight},i,j}^{l,+}$ respectively. Next, we add a cascade of AND gates to implement $s_j^l = \sum_{i=1}^{N^{l-1}} s_i^{l,0}$. For each binary addition $s_{h,j}^{l,k} = s_{2h-1,j}^{l,k-1} + s_{2h,j}^{l,k-1}$, for all M^2 combinations of summand values v_{m_1}, v_{m_2} , we add:

1. Gate $G_{\text{Sum},h,j}^{l,k,m_1,m_2}$, which outputs $S_{h,j}^{l,k,v_{m_1}+v_{m_2}}$ given $S_{2h-1,j}^{l,k-1,v_{m_1}}$ and $S_{2h,j}^{l,k-1,v_{m_2}}$ as input.

However, if $k = \log(N^{l-1})$ (the final tree level), or if $|s_{2h-1,j}^{l,k-1} + s_{2h,j}^{l,k-1} + b_j^l| > N^{l-1}/2$, we let $G_{\text{Sum},h,j}^{l,k,m_1,m_2}$ produce the neuron majority species $X_j^{l,+}$ or $X_j^{l,-}$.

Note that, since the output strand of each two-domain gate reverses orientation as compared to its input strand(s) (the toehold moves to the opposite side of the recognition domain), we have to alternate the orientation of gates at each level in the summation tree. Also, since the summation can end at either an odd or- even numbered level, we have to add translator gates which swap the orientation of the final activation species $X_j^{l,+}$ and $X_j^{l,-}$. This guarantees that the activation output strands are always in the correct orientation with respect to the input weight gates of the next layer.



Weight Gates: $s_{ij}^{l,0} = w_{ij}^l * x_i^{l-1}$

If $w_{ij}^l = +1$:



Else if $w_{ij}^l = -1$:



Restoration Gates for x_i^{l-1}






■ **Figure 4** DSD schematic of the binarized neural network implementation, based on the two-domain architecture of [3].

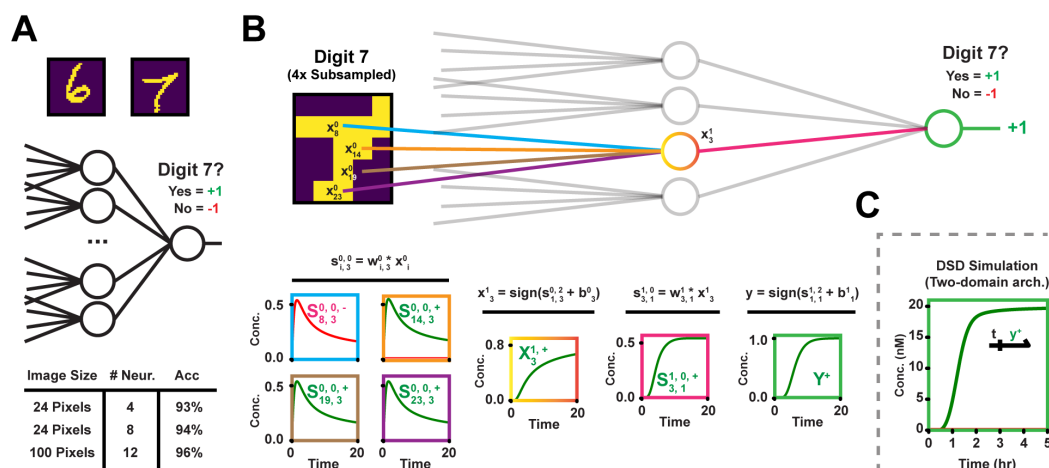


Figure 5 **A** A single-hidden layer network with 4, 8 or 12 neurons was trained to classify MNIST digits 6 vs. 7. **B** Example CRN simulation, as a system of ODEs with unit concentrations and reaction rates. Graph color corresponds to network component. **C** The network was simulated in Microsoft’s DSD tool. Signal concentration = 20nM. Shown is the final output strand trajectory.

5 Experiments

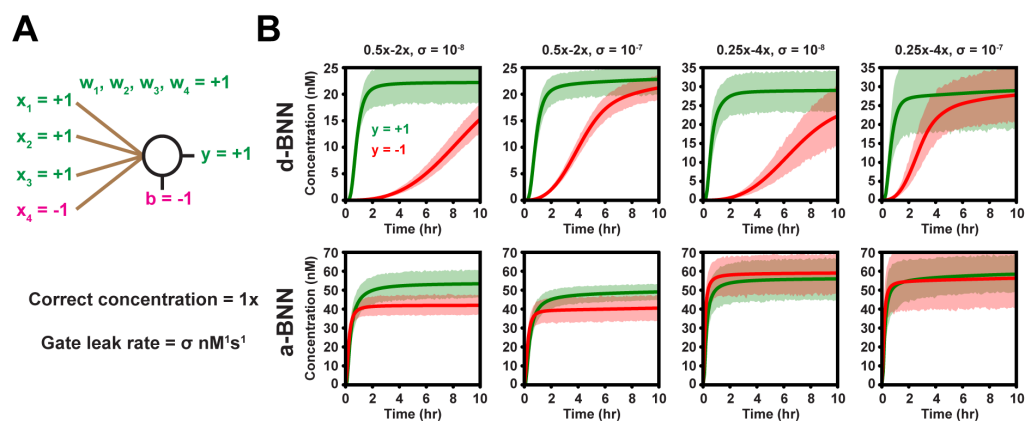
5.1 MNIST Simulations

We demonstrate our binarized neural network design on the MNIST digit classification task. Similar to one of the analyses in [8], we tested the model’s ability to distinguish between digits 6 and 7. We trained three versions of a single-hidden layer network (Figure 5A), with 4, 8 and 12 hidden neurons respectively. For the 4 and- 8 hidden neuron networks, we downsampled the input images to 5×5 pixels. For the 12 neuron version, the images were downsampled to 10×10 pixels. The image pixel values were binarized by subtracting the mean pixel intensity and thresholding at 0. We settled on a sparse connectivity structure, where neurons were connected to 4 randomly chosen inputs. The networks were trained following the procedure of [14], using PyTorch [21] and the Adam optimizer [15].

The network with only 4 hidden neurons correctly classified as many as 93% of test images (Figure 5A, bottom table). Test accuracy increased marginally up to 96% for the largest network. We translated the 4-hidden neuron network into our digital CRN design, totalling 192 molecular species and 102 reactions, and simulated the entire system of ODEs for an example input image (Figure 5B). Finally, we mapped the 4-hidden neuron network CRN to a DNA strand displacement (DSD) cascade, using the architecture presented in Section 4.4. The DSD specification was compiled into a system of ODEs using Microsoft’s DSD tool with default toehold binding rates and “infinite” compilation mode [17]. The ODE was simulated by Python SciPy’s *odeint* (Figure 5B).

5.2 Noise Tolerance Simulations

Next, we compared our digital design (d-BNN) to an analog rate-independent design (a-BNN) with the HardTanh activation function defined in Section 2.2. The designs were compiled into CRNs using Microsoft’s DSD tool. Each CRN was copied to Python, compiled into ODEs and simulated by SciPy’s *odeint*. Keeping the designs as CRNs in Python allows us to easily add leak pathways. We provide the schematic for the analog HardTanh network as idealized CRNs in Appendix B and as DNA strand displacement cascades in Appendix C.



■ **Figure 6** **A** A single 4-input neuron was compiled into DSD (both as a digital circuit – d-BNN, and as an analog circuit – a-BNN). Shown is the tested input pattern. **B** Noise- and leak tolerance simulations. Concentrations varied uniformly between $0.5x-2x$ or $0.25x-4x$. Signal concentration ($1x$) = 20nM. Leak reactions were added to DSD gates with rates of 10^{-8} or 10^{-7} $\text{nM}^{-1}\text{s}^{-1}$. Each simulation was run 10 times. 95% confidence intervals estimated from 1000-fold bootstrapping.

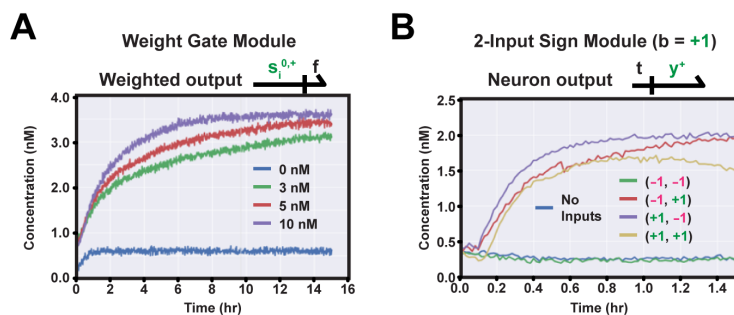
We compared the effects of concentration noise and gate leak on each respective design for a single 4-input neuron (Figure 6A). Specifically, we multiplied input strand and gate concentrations with a uniform random value and added leak reactions to all gates. Four different conditions were tested, and each condition was simulated 10 times. The results indicate that the digital binarized neuron is more robust than its analog counterpart (Figure 6B); in all four conditions, the correct “turned-on” output trajectory is separable from the “turned-off” (leaked) trajectory up to 3 hours for the digital neuron, whereas the analog neuron loses separability of the output trajectories almost immediately.

5.3 Physical Experimental Results

We performed wet lab experiments to validate the function of the basic DSD components used in the BNN. In Figure 7A, we tested a single catalytic two-domain weight gate, which is used to restore the input signal to the operating concentration (4nM) given different concentrations of input strand. As can be seen, the gate restores the signal with low levels of leak, and all conditions reach 75% of the target concentration within 12 hours. In Figure 7B, we tested the function of a simple 2-input majority voter where the bias term b is set to +1. The experiment suggests that the system functions correctly with low levels of leak. Note that there is no catalytic amplification of the majority voting output signal in Figure 7B, which is why the concentration is not restored to 4nM.

6 Discussion

When comparing digital BNNs to other molecular neural network implementations, our design offers both advantages and disadvantages. In terms of complexity, our design is less efficient (compact) than both rate-dependent and rate-independent analog designs [23, 8, 32], which require $\mathcal{O}(N)$ bi-molecular reactions for weighting and only $\mathcal{O}(1)$ reactions for majority voting. However, our simulations indicate that the digital design is more robust to concentration noise and gate leak compared to analog implementations. Furthermore, since rate-independent analog CRNs operate on monotonic dual-rail species, any physical implementation of a



■ **Figure 7** **A** An amplifier is used to restore the operating concentration (4nM) of the input strand given different input strand concentrations. **B** A 2-input majority voting circuit with +1 bias is used to trigger the output strand only if the positive inputs are in majority. Input concentration is 4nM.

deep, fully connected neural network would require an exponentially large concentration of molecular substrates at the final network layers (exponential in the network layers). The digital design, however, allows a constant substrate concentration across all layers.

The DSD architecture of the digital BNN can potentially support even large networks, since the DNA gates can be enzymatically prepared from a pool of fully double-stranded DNA [7]. However, it is often difficult in practice to scale up the number of two-domain gates in a single reaction vessel due to the many possible leak pathways, in particular for catalytic gates. To reduce noise, we might consider isolating each neuron computation with either localized reactions [2] or physical separation by microfluidic droplets [31]. Alternatively, strand-displacing polymerase (PSD) may be a promising option, which leak minimally [29, 25]. Furthermore, all reactions can be implemented with single-stranded PSD AND-gates, allowing for simple large-scale synthesis. The main caveat is that PSD currently does not support catalytic reactions. However, we can forego the catalytic reactions and instead start with exponentially large input concentrations, which may be feasible for 1–2 hidden layers of computation.

Finally, for future work we wonder whether the gate complexity of $\mathcal{O}(N^2)$ for digital majority voting can be reduced by acknowledging that neural networks often behave well with small errors. We thus ask if we could design an “approximate” majority voter with $\mathcal{O}(N)$ gates. For example, instead of computing exact partial sums on groups of inputs, we might get approximately correct output using only the signs of the inputs at each level of the tree.

7 Conclusion

In this paper, we present a digital molecular design of binarized neural networks. We devise a depth-optimal majority voting circuit that uses $\mathcal{O}(N^2)$ bi-molecular chemical reactions in a cascade of depth $\mathcal{O}(\log(N))$ to compute N -input majority. Each neuron uses this circuit to compute its activation function. We demonstrated our molecular implementation on the MNIST digit classification task, by simulating the ODE of a network with 4 hidden neurons as a DNA strand displacement cascade. We further demonstrated improved tolerance to concentration noise compared to analog BNN implementations in simulations.

We hope this paper sparks future research in molecular implementations of machine learning models. The intersection of digital circuit design, ML techniques and chemical reaction networks can enable other computational models implemented as molecular circuits and lead to whole new applications in molecular computing.

References

- 1 Y. Benenson, B. Gil, U. Ben-Dor, R. Adar, and E. Shapiro. An autonomous molecular computer for logical control of gene expression. *Nature*, 429(6990):423–429, 2004.
- 2 H. Bui, S. Shah, R. Mokhtar, T. Song, S. Garg, and J. Reif. Localized DNA hybridization chain reactions on DNA origami. *ACS nano*, 12(2):1146–1155, 2018.
- 3 L. Cardelli. Two-domain DNA strand displacement. *Mathematical Structures in Computer Science*, 23(2):247–271, 2013.
- 4 L. Ceze, J. Nivala, and K. Strauss. Molecular digital data storage using dna. *Nature Reviews Genetics*, 20(8):456–466, 2019.
- 5 H.L. Chen, D. Doty, and D. Soloveichik. Rate-independent computation in continuous chemical reaction networks. In *Proceedings of the 5th Conference on Innovations in Theoretical Computer Science*, pages 313–326, 2014 January.
- 6 S.X. Chen and G. Seelig. A DNA neural network constructed from molecular variable gain amplifiers. In *International Conference on DNA-Based Computers*, pages 110–121, 2017 September.
- 7 Y.J. Chen, N. Dalchau, N. Srinivas, A. Phillips, L. Cardelli, D. Soloveichik, and G. Seelig. Programmable chemical controllers made from DNA. *Nature nanotechnology*, 8(10):755–762, 2013.
- 8 K.M. Cherry and L. Qian. Scaling up molecular pattern recognition with DNA-based winner-take-all neural networks. *Nature*, 559(7714):370–376, 2018.
- 9 G.M. Church, Y. Gao, and S. Kosuri. Next-generation digital information storage in DNA. *Science*, 337(6102):1628–1628, 2012.
- 10 S. Darabi, M. Belbahri, M. Courbariaux, and V.P. Nia. BNN+: Improved binary network training. *OpenReview*, 2018.
- 11 I.R. Epstein and J.A. Pojman. An introduction to nonlinear chemical dynamics: Oscillations, waves, patterns, and chaos. *Oxford Univ Press London*, page London, 1998.
- 12 F. Fages, G. Le Gulse, O. Bournez, and A. Pouly. Strong turing completeness of continuous chemical reaction networks and compilation of mixed analog-digital programs. In *International Conference on Computational Methods in Systems Biology*, pages 108–127, 2017 September.
- 13 A. Hjelmfelt, E.D. Weinberger, and J. Ross. Chemical implementation of neural networks and turing machines. *Proceedings of the National Academy of Sciences*, 88(24):10983–10987, 1991.
- 14 I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks. In *Advances in Neural Information Processing Systems*, pages 4107–4115, 2016.
- 15 D.P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv*, 2014. [arXiv: 1412.6980](https://arxiv.org/abs/1412.6980).
- 16 M.R. Lakin and A. Phillips. Modelling, simulating and verifying turing-powerful strand displacement systems. In *International Workshop on DNA-Based Computers*, pages 130–144, 2011 September.
- 17 M.R. Lakin, S. Youssef, F. Polo, S. Emmott, and A. Phillips. Visual DSD: a design and analysis tool for DNA strand displacement systems. *Bioinformatics*, 27(22):3211–3213, 2011.
- 18 R. Lopez, R. Wang, and G. Seelig. A molecular multi-gene classifier for disease diagnostics. *Nature chemistry*, 10(7):746–754, 2018.
- 19 M.O. Magnasco. Chemical kinetics is turing universal. *Physical Review Letters*, 78(6):1190, 1997.
- 20 L. Organick, S.D. Ang, Y.J. Chen, R. Lopez, S. Yekhanin, K. Makarychev, M.Z. Racz, G. Kamath, P. Gopalan, B. Nguyen, and C.N.etal. Takahashi. Random access in large-scale DNA data storage. *Nature biotechnology*, 36(3):242, 2018.
- 21 A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch, 2017.
- 22 L. Qian and E. Winfree. Scaling up digital circuit computation with DNA strand displacement cascades. *Science*, 332(6034):1196–1201, 2011.

- 23 L. Qian, E. Winfree, and J. Bruck. Neural network computation with DNA strand displacement cascades. *Nature*, 475(7356):368–372, 2011.
- 24 P. Senum and M. Riedel. Rate-independent constructs for chemical computation. *PLoS one*, 6(6):e21414, 2011.
- 25 S. Shah, J. Wee, T. Song, L. Ceze, K. Strauss, Y.J. Chen, and J. Reif. Using strand displacing polymerase to program chemical reaction networks. *Journal of the American Chemical Society*, 142(21):9587–9593, 2020.
- 26 T. Simons and D.J. Lee. A review of binarized neural networks. *Electronics*, 8(6):661, 2019.
- 27 D. Soloveichik, M. Cook, E. Winfree, and J. Bruck. Computation with finite stochastic chemical reaction networks. *Natural Computing*, 7(4):615–633, 2008.
- 28 D. Soloveichik, G. Seelig, and E. Winfree. DNA as a universal substrate for chemical kinetics. *Proceedings of the National Academy of Sciences*, 107(12):5393–5398, 2010.
- 29 T. Song, A. Eshra, S. Shah, H. Bui, D. Fu, M. Yang, R. Mokhtar, and J. Reif. Fast and compact DNA logic circuits based on single-stranded gates using strand-displacing polymerase. *Nature nanotechnology*, 14(11):1075–1081, 2019.
- 30 N. Srinivas, J. Parkin, G. Seelig, E. Winfree, and D. Soloveichik. Enzyme-free nucleic acid dynamical systems. *Science*, 358(6369), 2017.
- 31 A. Stephenson, M. Willsey, J. McBride, S. Newman, B. Nguyen, C. Takahashi, K. Strauss, and L. Ceze. PurpleDrop: A digital microfluidics-based platform for hybrid molecular-electronics applications. *IEEE Micro*, 40(5):76–86, 2020.
- 32 M. Vasic, C. Chalk, S. Khurshid, and D. Soloveichik. Deep molecular programming: A natural implementation of binary-weight ReLU neural networks. *arXiv*, 2020. [arXiv:2003.13720](https://arxiv.org/abs/2003.13720).
- 33 M. Vasic, D. Soloveichik, and S. Khurshid. CRN++: Molecular programming language. In *International Conference on DNA Computing and Molecular Programming*, pages 1–18, 2018 October.
- 34 D.Y. Zhang and G. Seelig. DNA-based fixed gain amplifiers and linear classifier circuits. In *International Workshop on DNA-Based Computers*, pages 176–186, 2010 June.

A Generalized CRN Definition for Multi-layered Digital BNN

In this appendix, we extend the CRN formalism defined in Section 4 from a single binarized neuron to an arbitrarily sized network consisting of multiple neurons across many layers. Let us first extend the notation of the in-silico computational model, which, to remind the reader, is based on deterministic binarized neural networks (BNNs) as described by [14] with the added constraint that the number of neurons in any layer is a power of 2.

Let L be the number of network layers and let N^l be the number of neurons in layer l . Define $x_i^{(l)} \in \{+1, -1\}$ as the binary-valued activation of neuron i in layer l or, if $l = 0$, let $x_i^{(0)}$ be the i :th input to the network. Neuron i in layer $l - 1$ is connected to neuron j of layer l through the binary-valued weight $w_{i,j}^{(l)} \in \{+1, -1\}$. Additionally, Neuron j of layer l has an associated bias term (intercept) $b_j^{(l)} \in \{+1, -1\}$. We define activation $x_j^{(l)}$ of neuron j recursively as:

$$x_j^{(l)} = \begin{cases} +1 & \text{if } b_j^{(l)} + \sum_{i=1}^{N^{l-1}} w_{i,j}^{(l)} \cdot x_i^{(l-1)} > 0 \\ -1 & \text{else} \end{cases}$$

CRN Definition

Each neuron activation x_i^l is represented by two CRN species $X_i^{l,+}$ and $X_i^{l,-}$, corresponding to states $x_i^l = +1$ and $x_i^l = -1$ respectively. For each weight operation $s_{i,j}^{l,0} = w_{i,j}^l \cdot x_i^{l-1}$, add either of the following sets of reactions based on the sign of $w_{i,j}^l$:

If $w_{i,j}^l = +1$:

$$\begin{aligned} X_i^{l-1,+} &\rightarrow X_i^{l-1,+} + S_{i,j}^{l,0,+} \\ X_i^{l-1,-} &\rightarrow X_i^{l-1,-} + S_{i,j}^{l,0,-} \end{aligned}$$

Else if $w_{i,j}^l = -1$:

$$\begin{aligned} X_i^{l-1,+} &\rightarrow X_i^{l-1,+} + S_{i,j}^{l,0,-} \\ X_i^{l-1,-} &\rightarrow X_i^{l-1,-} + S_{i,j}^{l,0,+} \end{aligned}$$

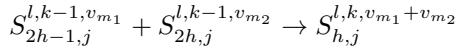
The sum operation $s_j^l = \sum_{i=1}^{N^{l-1}} s_i^{l,0}$ is calculated as a balanced tree of binary additions using the following recursive definition:

$$s_{h,j}^{l,k} = s_{2h-1,j}^{l,k-1} + s_{2h,j}^{l,k-1}$$

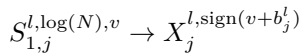
Here $k = 1, \dots, \log(N^{l-1})$ denotes the current depth in the tree and $h = 1, \dots, N^{l-1}/2^k$ denotes the tree node. Assume each summand can take on M discrete values, $s_{2h-1,j}^{l,k-1}, s_{2h,j}^{l,k-1} \in \{v_1, \dots, v_M\}$. The resulting sum can take on $2M-1$ values, $s_{h,j}^{l,k} \in \{v_1+v_1, v_1+v_2, \dots, v_M+v_M\}$. We represent each discrete state of each variable with a distinct molecular species:

State $s_{h,j}^{l,k} = v_m$ is encoded by species $S_{h,j}^{l,k,v_m}$

For each of the M^2 combinations of summand values $s_{2h-1,j}^{l,k-1} = v_{m_1}, s_{2h,j}^{l,k-1} = v_{m_2}$, add the reaction:



At depth $\log(N^{l-1})$ in the tree, the final weighted sum will be stored in variable $s_{1,j}^{l,\log(N^{l-1})}$. To compute the binary threshold activation function of neuron j in layer l , which we represent with species $X_j^{l,+}$ and $X_j^{l,-}$, add the following reaction for each of the $N+1$ possible sum output species $S_{1,j}^{l,\log(N),-N}, \dots, S_{1,j}^{l,\log(N),+N}$ (recalling that b_j^l is the bias term for neuron j in layer l):



B Analog Rate-Independent HardTanh Network CRN

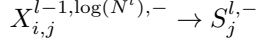
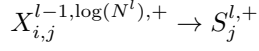
In the main paper, we compare the digital CRN of a binary-threshold neuron to an analog CRN implementation of a HardTanh-activated neuron. Here, we describe the analog design, including idealized CRN reactions and the two-domain DSD schematic. The implementation is based on the rate-independent neural network CRN that was recently proposed by [32], but with a HardTanh activation function $\min(\max(x, -1), 1)$ instead of the ReLU function $\max(x, 0)$.

In analog CRN computing, the weight and- sum operations are performed simultaneously. We implement the weighted sum of neuron j ,

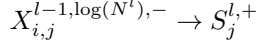
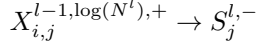
$$s_j^l = \sum_{i=1}^{N^{l-1}} w_{i,j}^l \cdot x_i^l$$

by adding either of the following two sets of reactions for each input i :

If $w_{i,j}^l = +1$:



Else if $w_{i,j}^l = -1$:

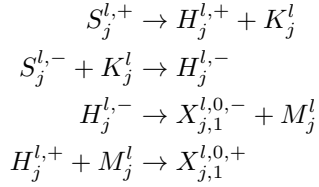


Note that the molecular species for input x_i^{l-1} are indexed by j . We cannot use catalytic reactions for rate-independent monotonic CRNs, which means we have to make individual copies of x_i^{l-1} for each outgoing neuron j . Further down in this text, we will add fan-out reactions which create as many copies of x_j^l as needed by the next layer. This copy operation, which is implemented as a binary tree, is the reason for having the hard-coded superscript $\log(N^l)$ in the species notation. Also note that the bias term b_j^l is implemented by setting the initial concentrations of $S_j^{l,+}$ and $S_j^{l,-}$ appropriately; if $b_j^l = +1$, start with $s_j^{l,+}(0) = 1$, or if $b_j^l = -1$, start with $s_j^{l,-}(0) = 1$.

Next, we implement the HardTanh activation function,

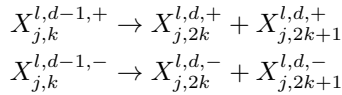
$$x_j^l = \min(\max(s_j^l, -1), 1)$$

by stacking the monotonic, dual-rail reaction set of the two functions $h_j^l = \max(s_j^l, -k)$ and $x_j^l = \min(h_j^l, m)$ as described by [5]:



In order for these reactions to implement clipping of s_j^l at $[-1, 1]$, we have to start with initial concentrations $k_j^l(0) = 1$ and $m_j^l(0) = 1$.

Finally, we fan out activation x_j^l to the outgoing N^{l+1} neurons of the next layer, by implementing copy operations $x_{j,k}^l = x_j^l$, $1 \leq k \leq N^{l+1}$. Since we only allow reactions with at most 2 products, we have to perform the copy in a balanced binary tree of depth of $\log(N^{l+1})$. Specifically, for $d = 1$ to $\log(N^{l+1})$ and $k = 1$ to 2^{d-1} , add the following two reactions:



To demonstrate the operation of the analog HardTanh CRN, we compiled the same 4-neuron network that was used in Figure 5 and simulated the resulting system of ODEs when classifying MNIST digit 7 from 6 (Figure 8A). Here, the monotonic dual-rail computation of the HardTanh function will make it so that the steady state concentration of the correct network output species (in this case $X_{1,1}^{2,0,+}$) is exactly 1 unit larger than the concentration of the minority species ($X_{1,1}^{2,0,-}$).

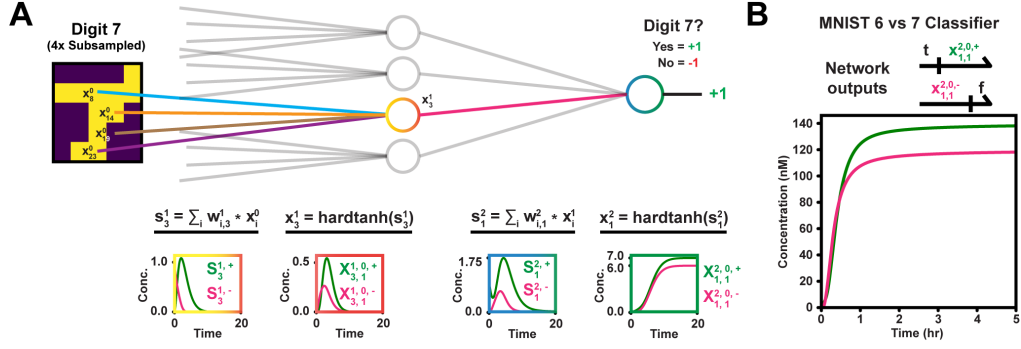


Figure 8 **A** Example CRN simulation, as a system of ODEs with unit concentrations and reaction rates. Graph color corresponds to network component. **B** The network was compiled into DSD gates and simulated in Microsoft's DSD tool. The graph shows the trajectory of the final output strand.

C Analog HardTanh BNN DSD Schematic

The DSD schematic for the analog HardTanh network is shown in Figure 9. Each activation x_i^l (before fanning out) is represented by two input strands, $X_{i,1}^{l,0,+}$ and $X_{i,1}^{l,0,-}$. Immediately following the network input strands $X_{i,1}^{0,0,+}$ and $X_{i,1}^{0,0,-}$, we add a cascade of gates which create N^l copies of $X_{i,1}^{0,0,+}$ and $X_{i,1}^{0,0,-}$. For $d = 1$ to $\log(N^l)$ and $h = 1$ to 2^{d-1} , we add:

1. Gate $G_{\text{Fanout},i,h}^{l,d,+}$, which translates $X_{i,h}^{l,d-1,+}$ to $X_{i,2h}^{l,d,+}$ and $X_{i,2h+1}^{l,d,+}$.

The N^l copies are now stored in the signal strands $X_{i,j}^{0,\log(N^l),+}$ and $X_{i,j}^{0,\log(N^l),-}$, $1 \leq j \leq N^l$. Note that we require different orientations for the negative and- positive signal strands; this is needed to make the Fork and- Join gates of the HardTanh circuit compatible without extra translators. Next, for each input i in the weighted sum $s_j^l = \sum_{i=1}^{N^{l-1}} w_{i,j}^l \cdot x_i^l$, we add either of the following two sets of gates depending on the sign of $w_{i,j}^l$:

If $w_{i,j}^l = -1$, we add:

1. Gate $G_{\text{Weight},i,j}^{l,+}$ which outputs signal strand $S_j^{l,-}$.
2. Gate $G_{\text{Weight},i,j}^{l,-}$ which outputs signal strand $S_j^{l,+}$.

If $w_{i,j}^l = +1$, we must add an extra translation step in order to maintain the orientation of the output strands. We thus add:

1. Gate $G_{\text{Weight},i,j}^{l,+}$ which outputs signal strand $D_j^{l,+}$.
2. Gate $G_{\text{Weight},i,j}^{l,-}$ which outputs signal strand $D_j^{l,-}$.
3. Gate $G_{\text{Sum-Swap},j}^{l,+}$ which outputs $S_j^{l,+}$ given $D_j^{l,+}$ as input.
4. Gate $G_{\text{Sum-Swap},j}^{l,-}$ which outputs $S_j^{l,-}$ given $D_j^{l,-}$ as input.

The HardTanh circuit is implemented by adding a sequence of 4 gates:

1. Gate $G_{\text{HardTanh}_1,j}^l$, which outputs K_j^l and $H_j^{l,+}$ given $S_j^{l,+}$ as input.
2. Gate $G_{\text{HardTanh}_2,j}^l$, which outputs $H_j^{l,-}$ given $S_j^{l,-}$ and K_j^l as input.
3. Gate $G_{\text{HardTanh}_3,j}^l$, which outputs M_j^l and $X_{j,1}^{l,0,+}$ given $H_j^{l,-}$ as input.
4. Gate $G_{\text{HardTanh}_4,j}^l$, which outputs $X_{j,1}^{l,0,+}$ given $H_j^{l,+}$ and M_j^l and K_j^l as input.

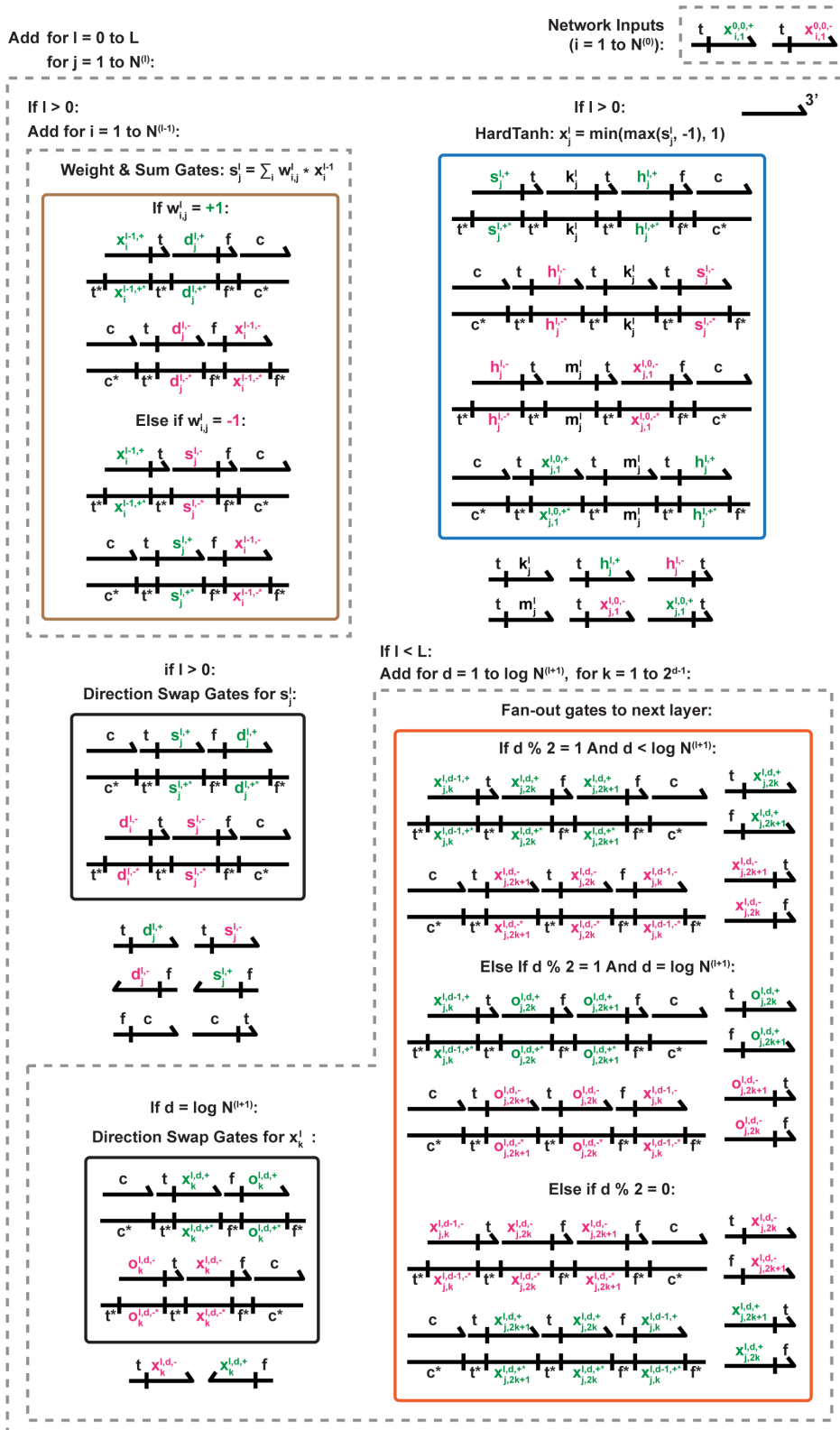


Figure 9 DSD schematic of the analog HardTanh neural network implementation, based on the two-domain architecture of [3].

1:20 Robust Digital Molecular Neural Networks

Finally, we add a cascade of fan-out gates which multiplex the neuron activation strands $X_{j,1}^{l,0,+}$ and $X_{j,1}^{l,0,-}$ to N^{l+1} copies, $X_{j,k}^{l,0,+}$ and $X_{j,k}^{l,0,-}$ (same set of gates $G_{\text{Fanout},j,k}^{l,d,+}$ and $G_{\text{Fanout},j,k}^{l,d,-}$ as previously described). We mind the orientation of positive and- negative signal strands by alternating gate orientation and add direction swap gates at the final layer in case the fan-out depth is odd-numbered.

We replicated the DSD simulation of the 4-neuron MNIST classifier of Figure 5C using the HardTanh circuit. The trajectories of the final output species $X_{1,1}^{2,0,+}$ and $X_{1,1}^{2,0,-}$ are shown in Figure 8B. We set the signal unit to 20 nM (same as the digital BNN in Figure 5C), which means that the final steady state concentrations become 140 nM and 120 nM (Compare to the unit-less steady-state concentrations of Figure 8A, which were 7 and 6 respectively).

Computing Properties of Thermodynamic Binding Networks: An Integer Programming Approach

David Haley ✉

University of California, Davis, CA, USA

David Doty ✉

University of California, Davis, CA, USA

Abstract

The thermodynamic binding networks (TBN) model [9] is a tool for studying engineered molecular systems. The TBN model allows one to reason about their behavior through a simplified abstraction that ignores details about molecular composition, focusing on two key determinants of a system’s energetics common to *any* chemical substrate: how many molecular bonds are formed, and how many separate complexes exist in the system. We formulate as an integer program the NP-hard problem of computing *stable* (a.k.a., minimum energy) configurations of a TBN: those configurations that maximize the number of bonds and complexes. We provide open-source software [13] solving this integer program. We give empirical evidence that this approach enables dramatically faster computation of TBN stable configurations than previous approaches based on SAT solvers [3]. Furthermore, unlike SAT-based approaches, our integer programming formulation can reason about TBNs in which some molecules have unbounded counts. These improvements in turn allow us to efficiently automate verification of desired properties of practical TBNs. Finally, we show that the TBN has a natural representation with a unique Hilbert basis describing the “fundamental components” out of which *locally* minimal energy configurations are composed. This characterization helps verify correctness of not only stable configurations, but entire “kinetic pathways” in a TBN.

2012 ACM Subject Classification Theory of computation → Theory and algorithms for application domains

Keywords and phrases thermodynamic binding networks, integer programming, constraint programming

Digital Object Identifier 10.4230/LIPIcs.DNA.27.2

Supplementary Material *Software (Source Code)*: https://github.com/drhaley/stable_tbn

Funding Supported by NSF award 1900931 and CAREER award 1844976.

1 Introduction

Recent experimental breakthroughs in DNA nanotechnology [5] have enabled the construction of intricate molecular machinery whose complexity rivals that of biological macromolecules, even executing general-purpose algorithms [23]. A major challenge in creating synthetic DNA molecules that undergo desired chemical reactions is the occurrence of erroneous “leak” reactions [16], driven by the fact that the products of the leak reactions are more energetically favorable. A promising design principle to mitigate such errors is to build “thermodynamic robustness” into the system, ensuring that leak reactions incur an energetic cost [14, 20, 22] by logically forcing one of two unfavorable events: either many molecular bonds must break – an “enthalpic” cost – or many separate molecular complexes (called *polymers* in this paper) must simultaneously come together – an “entropic” cost.

The model of *thermodynamic binding networks* (TBNs) [9] was defined as a combinatorial abstraction of such molecules, deliberately simplifying substrate-dependent details of DNA in order to isolate the foundational energetic contributions of forming bonds and separating polymers. A TBN consists of *monomers* containing specific *binding sites*, where binding



© David Haley and David Doty;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on DNA Computing and Molecular Programming (DNA 27).

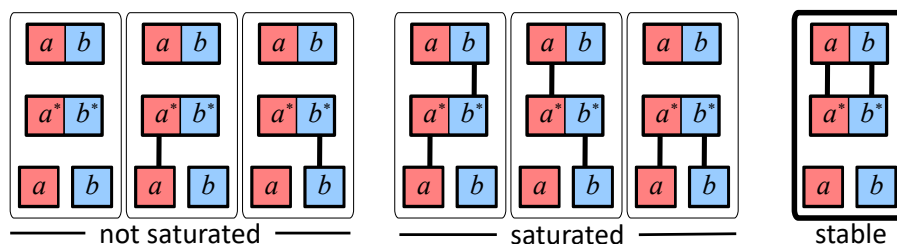
Editors: Matthew R. Lakin and Petr Šulc; Article No. 2; pp. 2:1–2:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

site a can bind only to its complement a^* . A key aspect of the TBN model is the lack of geometry: a monomer is an *unordered* collection of binding sites such as $\{a, a, b^*, c\}$. A *configuration* of a TBN describes which monomers are grouped into *polymers*; bonds can only form within a polymer.¹ One can formalize the “correctness” of a TBN by requiring that its desired configuration(s) be *stable*: the configuration maximizes the number of bonds formed, a.k.a., it is *saturated*, and, among all saturated configurations, it maximizes the number of separate polymers.² See Figure 1 for an example. Stable configurations are meant to capture the *minimum free energy structures* of the TBN. Unfortunately, answering basic questions such as “*Is a particular TBN configuration stable?*” turn out to be NP-hard [3].



■ **Figure 1** Example of a simple thermodynamic binding network (TBN). There are four monomers: $\mathbf{m}_1 = \{a^*, b^*\}$, $\mathbf{m}_2 = \{a, b\}$, $\mathbf{m}_3 = \{a\}$, $\mathbf{m}_4 = \{b\}$, with seven configurations shown: four of these configurations are saturated because they have the maximum of 2 bonds. Of these, three have 2 polymers and one has 3 polymers, making the latter the only stable configuration. Despite the suggestive lines between binding sites, the model of this paper ignores individual bonds, defining a configuration solely by how it partitions the set of monomers into polymers, assuming that a maximum number of bonds will form within each polymer. (Thus other configurations exist besides those shown, which would merge polymers shown without allowing new bonds to form.)

1.1 Our contribution

Our main contribution is a reduction that formulates the problem of finding stable configurations of a TBN as an integer program (IP). Of course, the problem, appropriately formalized, is “clearly” an NP search problem, so the mere existence of such a reduction is not particularly novel. However, our formulation is notable in three respects: 1) We carefully avoid certain symmetries (particularly those present in the existing SAT-based formulation of Breik et al. [3]), which dramatically increases the search efficiency in practice. 2) We use the optimization ability of IP solvers as a natural way to maximize the number of polymers in any saturated configuration. 3) Our formulation leads to a natural interpretation of the *Hilbert basis* [7] of a TBN as its minimal saturated polymers, which intuitively are the polymers existing in any *local* energy minimum configuration. Since highly optimized software exists for calculating Hilbert bases [1], this expands the range of TBN behaviors that can be automatically reasoned about.

¹ A primary goal of the TBN model is to establish that certain undesired configurations impose an energetic cost, i.e., to demonstrate *impossibility* results. From this perspective, the lack of geometric constraints means that such impossibility results give stronger guarantees than if geometric constraints, such as unpsuedoknottedness of DNA secondary structure, were imposed on the allowable configurations. For example, “leakless” DNA strand displacement systems [20, 22] retain their leaklessness even if pseudoknotted structures are permitted.

² This definition captures the limiting case (often approximated in practice in DNA nanotechnology) corresponding to increasing the strength of bonds, while diluting (increasing volume), such that the ratio of binding to unbinding rate goes to infinity.

This formulation allows us to automate portions of formal reasoning about TBNs, helping verify their correctness. The TBN model abstracts away the continuous nature of real free energy into discrete integer-valued steps. In the limit of dilute solutions (bringing together polymers incurs a large energy cost) and very strong bonds (breaking a bond incurs a huge energy cost), even one integer step of energy difference is considered significant. Furthermore, in a catalytic system, the presence (or absence) of even a single polymer may have a very large impact on the system’s trajectory, an indication that system analyses are very numerically sensitive, and thus their behavior is not well-described by approximation methods. For these reasons it is crucial when verifying such systems that we identify the *exact* solution to the optimization problem, rather than settling for more efficiently computable approximations (e.g., via continuous relaxation [6] or local search [18]).

1.2 Related work

Breik, Thachuk, Heule, and Soloveichik [3] characterize the computational complexity of several natural problems related to TBNs. For instance, it is NP-complete to decide whether a saturated configuration exists with a specified number of polymers, and even NP-hard to approximate within any constant factor the number of polymers in a stable configuration (i.e., the maximum in any saturated configuration).

Breik et al. also developed software using a SAT solver to produce stable configurations of a TBN. This formulation requires “labelled” monomers (where two different instances of the same monomer type are represented by separate Boolean variables), which become nodes in a graph, and polymers are realized as connected components within the graph. By labelling the monomers they become unique copies of the same monomer type; n copies of a monomer type increases the size of the search space by factor $n!$ by considering these symmetric configurations separately. Furthermore, the software explicitly explores all possible symmetries of bonding arrangements within a polymer. For instance, monomers $\{a^*, a^*\}$ and $\{a, a\}$ can bind in two different ways (the first a^* can bind either the first or second a), even though both have the same number of bonds and polymers. This over-counting of symmetric configurations prevents the software from scaling to efficiently analyze certain TBNs with large counts of monomers. Our IP formulation avoids both types of symmetry.

2 Preliminaries

2.1 Definitions

A *multiset* is an unordered collection of objects allowing duplicates (including countably infinite multiplicities), e.g., $\mathbf{v} = \{2 \cdot a, b, \infty \cdot d\}$. Equivalently, a multiset with elements from a finite set U is a vector $\mathbf{v} \in (\mathbb{N} \cup \{\infty\})^U$ describing the counts, indexed by U ; in the example above, if $U = \{a, b, c, d\}$, then $\mathbf{v}(a) = 2$, $\mathbf{v}(b) = 1$, $\mathbf{v}(c) = 0$, and $\mathbf{v}(d) = \infty$. The *cardinality* of a multiset $\mathbf{v} \in \mathbb{N}^U$ is $|\mathbf{v}| = \sum_{u \in U} \mathbf{v}(u)$; a *finite multiset* \mathbf{v} obeys $|\mathbf{v}| < \infty$. A *site type* is a formal symbol, such as a , representing a specific binding site on a molecule; in Figure 1 the site types are a, a^*, b, b^* . Each site type has a corresponding *complement type* denoted by a star, e.g. a^* . Complementarity is an involution: $(a^*)^* = a$. A site and its complement can form an attachment called a *bond*. We follow the convention that for any complementary pair of sites a, a^* , the total count of a^* across the whole TBN is at most that of a , i.e., the starred sites are *limiting*. A *monomer type* is a finite multiset of site types.³

³ Concretely, in a DNA nanotech design, a monomer corresponds to a strand of DNA, whose sequence is logically partitioned into binding sites corresponding to long (5-20 base) regions, e.g., $5'$ -AAAGG- $3'$, intended to bind to the complementary sequence $3'$ -TTTCC- $5'$ that is part of another strand.

A *thermodynamic binding network (TBN)* is a multiset of monomer types; equivalently, a vector in $(\mathbb{N} \cup \{\infty\})^m$, if m is the number of monomer types and we have fixed some standardized ordering of them. We allow some monomer counts to be infinite in order to capture the case where some monomers are added in “large excess” over others, a common experimental approach [16,17]. A *polymer* \mathbf{P} is a finite multiset of monomer types, equivalently, a vector in \mathbb{N}^m , where m is the number of monomer types in a standardized ordering;⁴ thus $|\mathbf{P}|$ represents the number of monomers in \mathbf{P} . Note that despite the suggestive lines representing bonds in Figure 1, this definition does not track which pairs of complementary sites are bound within a polymer. Indeed, it is allowable in the model for a polymer to consist of monomers that cannot be connected by bonds, e.g., the polymer $\{\{a\}, \{b\}\}$ using monomers from Figure 1. In general we study configurations that *minimize* the number of merges necessary to reach them, and merges that create such polymers in general would not be useful in such a setting.

Given a TBN \mathcal{T} , let $S_{\mathcal{T}}$ (respectively, $S_{\mathcal{T}}^*$) be the set of unstarred (resp., starred) site types of \mathcal{T} . For a monomer \mathbf{m} and site type $s \in S_{\mathcal{T}}$, let $\mathbf{m}(s)$ denote the count of s minus the count of s^* in \mathbf{m} (intuitively, $\mathbf{m}(s)$ is the “net count” of s in \mathbf{m} , negative if there are more s^* .) For $s^* \in S_{\mathcal{T}}^*$ let $\mathbf{m}(s^*) = -\mathbf{m}(s)$. The *exposed sites* of a polymer \mathbf{P} are a finite multiset of site types that results from removing as many (site, complement) pairs from a polymer as possible, described by the net count of sites when summed across all monomers in the polymer. For example, in the polymer $\{\{a^*, b^*, c^*\}, \{a, c\}, \{a, b, c\}, \{c, d^*\}\}$, the exposed sites are $\{a, 2 \cdot c, d^*\}$.

A *configuration* of a TBN is a partition of the monomers of the TBN into polymers; we write $|\gamma|$ to denote the number of polymers in configuration γ . A polymer is *self-saturated* if it has no exposed starred sites. A configuration is *saturated* if all of its polymers are self-saturated. Since we assume that, across the entire configuration, starred sites are limiting, this is equivalent to stipulating that the maximum possible number of bonds are formed. Write $\Gamma_{\mathcal{T}}$ to denote the set of all saturated configurations of the TBN \mathcal{T} . A configuration is *stable* if it is saturated and has the maximum number of non-singleton polymers among all saturated configurations.

Since the number of polymers may be infinite, we will use the equivalent notion that stable configurations are those that can be constructed by starting with the “melted” configuration whose polymers are all singletons containing only one monomer, performing the minimum number of polymer merges necessary to reach a saturated configuration. In general this quantity could be infinite, but we consider only TBNs in which a saturated configuration is reachable via a finite number of merges. For example, consider the TBN consisting of monomer types $\mathbf{t} = \{a\}$, $\mathbf{b} = \{a^*\}$, with counts $\infty \cdot \mathbf{t}$ and $2 \cdot \mathbf{b}$. The unique stable configuration has polymers $\{2 \cdot \{\mathbf{b}, \mathbf{t}\}, \infty \cdot \mathbf{t}\}$, since two merges of a \mathbf{b} and a \mathbf{t} are necessary and sufficient to create this configuration from the individual monomers.

2.2 Solvers

The problems addressed in this paper are NP-hard. To tackle this difficulty, we cast the problems as integer programs and use the publicly available IP solver SCIP [11].

⁴ The term “polymer” is chosen to convey the concept of combining many atomic objects into a complex, but it is not necessarily a linear chain of repeated units.

We also use the open-source software OR-tools [15], which is a common front-end for SCIP [11], Gurobi [12], and a bundled constraint programming solver CP-SAT. Though we model our problems as IPs, we would also like to be able to solve for all feasible/optimal solutions rather than just one, which CP-SAT can do. This flexible front-end lets us switch seamlessly between the two types of solvers without significant alterations to the model.

We use the software package 4ti2 [1] to calculate Hilbert Bases as described in Section 4.

3 Computing stable configurations of TBNs

Section 3.1 formally defines the stable configurations problem. Section 3.2 explains our IP formulation of the problem. Section 3.3 shows empirical runtime benchmarks.

3.1 Finding stable configurations of TBNs

We consider the problem of finding the stable configurations of a TBN. Given a TBN \mathcal{T} , let $\Gamma_{\mathcal{T}}$ denote the set of all saturated configurations of \mathcal{T} .

Recall that a configuration $\gamma \in \Gamma_{\mathcal{T}}$ is defined as a partition of the monomers of \mathcal{T} into polymers, so its elements $\mathbf{P} \in \gamma$ are polymers, i.e., multisets of monomers. For any $\gamma \in \Gamma_{\mathcal{T}}$, we define the corresponding *partial configuration* $\bar{\gamma} = \{\mathbf{P} \in \gamma : |\mathbf{P}| > 1\}$ that excludes polymers consisting of only a single monomer. Note that in the context of \mathcal{T} , the mapping $\gamma \mapsto \bar{\gamma}$ is one-to-one. We consider only partial configurations with finite-sized polymers. The notion of partial configuration will be useful in reasoning about TBNs with infinite monomer counts but finite size polymers, since all but finitely many monomers will be excluded from the partial configurations we consider.

Now we define the number of elementary merge operations required to reach a saturated configuration γ from the configuration of all singletons. We can calculate this directly as the difference in the counts of the monomers and polymers in the partial configuration, since each merge reduces the number of polymers by one:

$$m(\gamma) = \left(\sum_{\mathbf{P} \in \bar{\gamma}} |\mathbf{P}| \right) - |\bar{\gamma}| \quad (1)$$

We can then define the stable configurations as those saturated configurations that minimize the number of merges required to reach them from the all-singletons configuration.

$$\text{STABLECONFIGS}(\mathcal{T}) = \{\gamma \in \Gamma_{\mathcal{T}} : (\forall \gamma' \in \Gamma_{\mathcal{T}}) m(\gamma) \leq m(\gamma')\}$$

Note that $m(\gamma) = m(\bar{\gamma})$. Thus the STABLECONFIGS problem may be equivalently posed as finding the set of saturated partial configurations $\bar{\gamma}$ that minimize $m(\bar{\gamma})$.

We now describe how to handle infinite counts. A configuration is saturated if and only if none of its starred sites (elements of $S_{\mathcal{T}}^*$) are exposed. Thus we focus on the subset of monomers that contain starred sites: the *limiting monomers* $\mathcal{T}_L = \{\mathbf{m} \in \mathcal{T} : \mathbf{m} \cap S_{\mathcal{T}}^* \neq \emptyset\}$. Limiting monomers are required to have finite count, whereas nonlimiting monomers (those with all unstarred sites) are allowed to be finite or infinite count. Our IP representation of a configuration explicitly accounts for *all* the limiting monomers, but only the nonlimiting monomers (in $\mathcal{T} \setminus \mathcal{T}_L$) in a polymer with some limiting monomer; implicitly every other nonlimiting monomer is unbound (i.e., in its own singleton polymer). This allows us to describe infinite configurations where all but finitely many of the infinite count monomers are unbound, guaranteeing that the number of merges counted in Equation (1) is finite.

3.2 Casting StableConfigs as an IP

3.2.1 Finding a single stable configuration

We first describe how to find a single element from $\text{STABLECONFIGS}(\mathcal{T})$ by identifying its partial configuration in \mathcal{T} . We begin by fixing an upper bound B on the number of non-singleton polymers in any partial configuration. If no *a priori* bound for B is available, conservatively take $B = |\mathcal{T}_L|$, the total number of limiting monomers.

3.2.1.1 Nonnegative integer variables

Assume an arbitrary ordering of the m monomer types $\mathbf{m}_1, \mathbf{m}_2, \dots$. Our IP formulation uses $B \cdot m + B$ nonnegative integer variables describing the solution via its partial configuration:

- $\text{Count}(\mathbf{m}, j)$: count of monomer type $\mathbf{m} \in \mathcal{T}$ in polymer \mathbf{P}_j where $j \in \{1, 2, \dots, B\}$
- $\text{Exists}(j)$: false (0) if polymer \mathbf{P}_j is empty, possibly true (1) otherwise, $j \in \{1, 2, \dots, B\}$

► **Remark 1.** The constraints described below allow $\text{Exists}(j) = 0$ even if polymer j is nonempty, even though the variables ultimately aim to count exactly the number of nonempty polymers (as $\sum_{j=1}^m \text{Exists}(j)$). A false negative undercounts the number of polymers, overcounting the number of merges in Equation (1). However, the number of merges is being *minimized* by the IP. For a given setting of Count variables, the minimum is achieved (subject to the constraints) by setting each $\text{Exists}(j) = 1$ *if and only if* polymer j is nonempty.

► **Example 2.** Recall the TBN of Figure 1. Suppose the TBN has 1 each of $\mathbf{m}_1 = \{a^*, b^*\}$, $\mathbf{m}_2 = \{a, b\}$, $\mathbf{m}_3 = \{a\}$, $\mathbf{m}_4 = \{b\}$, with upper bound $B = 2$ on the number of non-singleton polymers. $\mathcal{T}_L = \{\mathbf{m}_1\}$ since \mathbf{m}_1 is the only monomer with starred sites. The linear constraints (see below for details) do not require all copies of $\mathbf{m}_2, \mathbf{m}_3, \mathbf{m}_4 \in \mathcal{T} \setminus \mathcal{T}_L$ to be included in a polymer. The stable configuration on the right of Figure 1 (partition $\{\mathbf{m}_1, \mathbf{m}_2\}, \{\mathbf{m}_3\}, \{\mathbf{m}_4\}$) is represented in the IP by setting $\text{Count}(\mathbf{m}_1, 1) = \text{Count}(\mathbf{m}_2, 1) = 1$, $\text{Count}(\mathbf{m}_3, 1) = \text{Count}(\mathbf{m}_4, 1) = 0$ (monomers 1 and 2 are in polymer 1, but monomers 3 and 4 are not), and setting $\text{Count}(\mathbf{m}_i, 2) = 0$ for $i = 1, 2, 3, 4$ (no monomers are in polymer 2), $\text{Exists}(1) = 1$ (polymer 1 is non-empty), and $\text{Exists}(2) = 0$ (polymer 2 is empty).

► **Example 3.** Suppose a TBN with the same monomer types as Example 2 has 3 of \mathbf{m}_1 and infinitely many of the remaining monomers (allowed since they are not limiting), with $B = 4$. The partial configuration where two copies of \mathbf{m}_1 are each bound to a single \mathbf{m}_2 (forming two polymers with two monomers each, as in the stable configuration of Figure 1), and the third \mathbf{m}_1 is bound to an \mathbf{m}_3 and an \mathbf{m}_4 (forming one polymer with three monomers, as in the rightmost non-stable saturated configuration of Figure 1) is represented in the IP by setting $\text{Exists}(1) = \text{Exists}(2) = \text{Exists}(3) = 1$ and $\text{Exists}(4) = 0$, and

$$\begin{array}{llll} \text{Count}(\mathbf{m}_1, 1) = 1, & \text{Count}(\mathbf{m}_2, 1) = 1, & \text{Count}(\mathbf{m}_3, 1) = 0, & \text{Count}(\mathbf{m}_4, 1) = 0, \\ \text{Count}(\mathbf{m}_1, 2) = 1, & \text{Count}(\mathbf{m}_2, 2) = 1, & \text{Count}(\mathbf{m}_3, 2) = 0, & \text{Count}(\mathbf{m}_4, 2) = 0, \\ \text{Count}(\mathbf{m}_1, 3) = 1, & \text{Count}(\mathbf{m}_2, 3) = 0, & \text{Count}(\mathbf{m}_3, 3) = 1, & \text{Count}(\mathbf{m}_4, 3) = 1, \\ \text{Count}(\mathbf{m}_1, 4) = 0, & \text{Count}(\mathbf{m}_2, 4) = 0, & \text{Count}(\mathbf{m}_3, 4) = 0, & \text{Count}(\mathbf{m}_4, 4) = 0. \end{array}$$

3.2.1.2 Linear constraints

Let $\mathcal{T}(\mathbf{m})$ denote the number of monomers of type \mathbf{m} in the TBN \mathcal{T} . Recall that $\mathbf{m}(s)$ is the net count of site type $s \in S_{\mathcal{T}}$ in monomer type \mathbf{m} (negative if \mathbf{m} has more s^* than s).

$$\text{Exists}(j) \leq 1 \quad \forall j \in \{1, 2, \dots, B\} \quad (2)$$

$$\sum_{j=1}^B \text{Count}(\mathbf{m}, j) = \mathcal{T}(\mathbf{m}) \quad \forall \mathbf{m} \in \mathcal{T}_L \quad (3)$$

$$\sum_{j=1}^B \text{Count}(\mathbf{m}, j) \leq \mathcal{T}(\mathbf{m}) \quad \forall \mathbf{m} \in \mathcal{T} \setminus \mathcal{T}_L \quad (4)$$

$$\sum_{\mathbf{m} \in \mathcal{T}} \text{Count}(\mathbf{m}, j) \cdot \mathbf{m}(s) \geq 0 \quad \forall j \in \{1, 2, \dots, B\}, \forall s \in S_{\mathcal{T}} \quad (5)$$

$$\sum_{\mathbf{m} \in \mathcal{T}_L} \text{Count}(\mathbf{m}, j) \geq \text{Exists}(j) \quad \forall j \in \{1, 2, \dots, B\} \quad (6)$$

Constraint (2) enforces that `Exists` variables are Boolean. Constraints (3) and (4) intuitively establish “monomer conservation” in the partial configuration. Constraint (3) enforces that we account for every limiting monomer in \mathcal{T} . Constraint (4) establishes that for *non*-limiting monomers, we cannot exceed their supply (trivially satisfied for any infinite-count monomer); any leftovers are assumed to be in singleton polymers in the full configuration, but are not explicitly described by `Count` variables. Constraint (5) ensures that all polymers are self-saturated. Specifically, the count of site $s \in S_{\mathcal{T}}$ within polymer j must meet or exceed that of s^* . Lastly, Constraint (6) enforces that nonempty polymers contain at least one limiting monomer. Ideally, this constraint should enforce that if a polymer contains no monomers at all, then it cannot be part of the nonempty polymer tally; however, if the constraint were modeled in this way, the formulation would admit invalid partial configurations that include explicit singleton polymers.

3.2.1.3 Linear objective function

Subject to the above constraints, we minimize the number of merges needed to go from a configuration where all monomers are separate to a saturated configuration. For finite count TBNs, this is the number of monomers minus the number of polymers in the partial configuration. Equivalently (and applying to infinite TBNs), this is the sum over all nonempty polymers of its number of monomers minus 1. Formally, the IP minimizes (7):

$$\sum_{j=1}^B \left[\left(\sum_{\mathbf{m} \in \mathcal{T}} \text{Count}(\mathbf{m}, j) \right) - \text{Exists}(j) \right] \quad (7)$$

If polymer j is empty ($\sum_{\mathbf{m} \in \mathcal{T}} \text{Count}(\mathbf{m}, j) = 0$), then constraint (6) forces $\text{Exists}(j) = 0$; otherwise $\text{Exists}(j) = 1$ minimizes (7). Thus the outer sum is over the nonempty polymers.

3.2.2 Finding all stable configurations

While an IP formulation for finding a single stable configuration is well-defined above, without modification it is ill-suited as a formulation to find *all* stable configurations. In addition, tightening the available constraints (e.g., enforcing $\text{Exists}(j) \iff$ polymer j is nonempty, described below) provides a more robust framework to which to add custom constraints (e.g. specifying a fixed number of polymers).

3.2.2.1 IP to find optimal objective value, CP to enumerate optimal solutions

One straightforward improvement is to solve for the optimal value of the objective function using a dedicated IP solver such as SCIP, whose primal-dual methods exploit the underlying real-valued geometry of the search space to find an objective value more efficiently than Constraint Programming (CP) solvers such as CP-SAT. Then, use this optimal value to bootstrap the CP formulation, which is better suited to enumerating all solutions with a given objective value. This works particularly well in our experiments: use SCIP to solve the optimization problem (but SCIP has no built-in ability to enumerate all feasible solutions), then use CP-SAT (which takes longer than SCIP to find the objective value) to locate all feasible solutions to the IP obtaining the objective value found by SCIP.

3.2.2.2 Enforcing that Exists variables exactly describe nonempty polymers

Constraint (5) enforces that $\text{Exists}(j) = 0$ if polymer \mathbf{P}_j is empty, but it does not enforce the converse. However, when using CP-SAT with a *fixed* objective value, we can no longer rely on the minimization of Equation (7) to enforce that $\text{Exists}(j) = 1 \iff \mathbf{P}_j$ is nonempty.

We add a new constraint to handle this. Let

$$C = 1 + \sum_{s \in S_{\mathcal{T}}} \sum_{\mathbf{m} \in \mathcal{T}} \mathcal{T}(\mathbf{m}) \cdot \mathbf{m}(s^*). \quad (8)$$

C is an upper bound on the largest number of monomers in a polymer in any valid partial configuration of \mathcal{T} minimizing Equation (7). This corresponds to the worst case in which a single polymer contains *every* limiting monomer, and each starred site is bound to its own unique monomer. The following constraint enforces that if $\text{Exists}(j) = 0$, then polymer \mathbf{P}_j contains no monomers:

$$\sum_{\mathbf{m} \in \mathcal{T}_L} \text{Count}(\mathbf{m}, j) \leq C \cdot \text{Exists}(j) \quad \forall j \in \{1, 2, \dots, B\} \quad (9)$$

3.2.2.3 Eliminating symmetries due to polymer ordering

In the formulation of Section 3.2, many isomorphic solutions exist in the feasible region. For instance, one could obtain a “new” solution by swapping the compositions of polymers \mathbf{P}_1 and \mathbf{P}_2 . The number of isomorphic partial configurations grows factorially with the number of polymers. Before asking the solver to enumerate all solutions, we must add constraints that eliminate isomorphic solutions. We achieve this by using the (arbitrary) ordering of the monomer types to induce a lexicographical ordering on the polymers, then add constraints ensuring that any valid solution contains the polymers in sorted order.

Sorting non-binary vectors in an IP is generally a difficult task (for instance, see [21]). The primary reason for this difficulty is that encoding the sorting constraints involves logical implications ($p \implies q$), which, being a type of disjunction ($\neg p \text{ OR } q$), are difficult to encode into a convex formulation described as a conjunction (AND) of several constraints. However, we do have an upper bound C on the values that the Count variables can take, making certain “large-number” techniques possible.

Intuitively, when comparing two lists of scalars (i.e., vectors) to verify that they are correctly sorted, one must proceed down the list of entries until one of the entries is larger than its corresponding entry in the other list. For as long as the numbers are the same, they are considered “tied”. When one entry exceeds the corresponding other, the tie is considered “broken”, after which no further comparisons need be conducted between the two vectors.

We introduce $B \cdot m$ new Boolean (0/1-valued) variables ($\text{Tied}(\mathbf{m}_i, j)$ for each $i = 1, \dots, m$ and $j = 1, \dots, B$), that reason about consecutive pairs of polymers $\mathbf{P}_{j-1}, \mathbf{P}_j$. We describe constraints enforcing that for each $h \leq i$, $\text{Tied}(\mathbf{m}_i, j) = 1 \iff \text{Count}(\mathbf{m}_h, j - 1) = \text{Count}(\mathbf{m}_h, j)$.

Let C be defined as in (8). For simplicity of notation below, define the constants $\text{Tied}(\mathbf{m}_0, j) = 1$ for all $j = 1, \dots, B$. The meaning of the sorting variables is then enforced by the following constraints, which we define for $i \in \{1, 2, \dots, m\}$ and $j \in \{2, 3, \dots, B\}$:

$$\text{Tied}(\mathbf{m}_i, j) \leq \text{Tied}(\mathbf{m}_{i-1}, j) \quad (10)$$

$$\text{Count}(\mathbf{m}_i, j - 1) - \text{Count}(\mathbf{m}_i, j) \leq C \cdot (1 - \text{Tied}(\mathbf{m}_i, j)) \quad (11)$$

$$\text{Count}(\mathbf{m}_i, j - 1) - \text{Count}(\mathbf{m}_i, j) \geq -C \cdot (1 - \text{Tied}(\mathbf{m}_i, j)) \quad (12)$$

$$\text{Count}(\mathbf{m}_i, j - 1) - \text{Count}(\mathbf{m}_i, j) \geq 1 - C \cdot (1 + \text{Tied}(\mathbf{m}_i, j) - \text{Tied}(\mathbf{m}_{i-1}, j)) \quad (13)$$

Intuitively, (10) enforces $\text{Tied}(\mathbf{m}_i, j) \implies \text{Tied}(\mathbf{m}_{i-1}, j)$: a tie in the current entry is only relevant if the tie was not resolved before. (11) and (12) together enforce $\text{Tied}(\mathbf{m}_i, j) \implies (\text{Count}(\mathbf{m}_i, j - 1) = \text{Count}(\mathbf{m}_i, j))$: ties can only continue for as long as the corresponding entries are equal.

(13) enforces $\text{Tied}(\mathbf{m}_{i-1}, j) \wedge \neg \text{Tied}(\mathbf{m}_i, j) \implies (\text{Count}(\mathbf{m}_i, j - 1) > \text{Count}(\mathbf{m}_i, j))$: ties can only be broken if the tie was not broken previously and the current entries are ordered correctly. Thus any solution satisfying these constraints must sort the polymers.

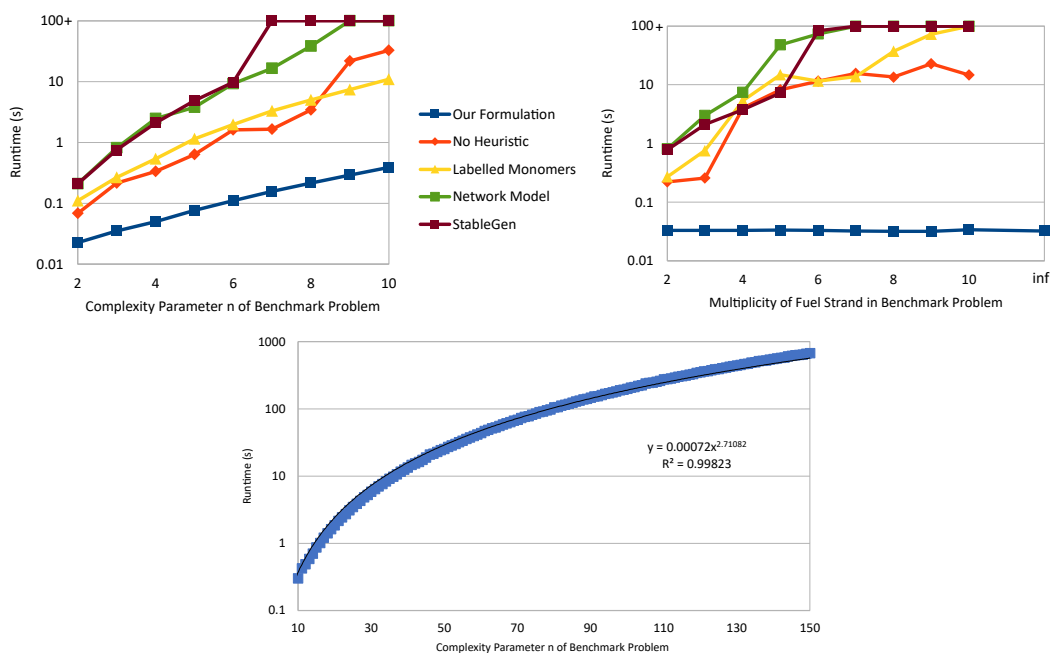
3.3 Empirical running time measurements

For our empirical tests we use as a benchmark the autocatalytic TBN described in [2, Section 4.2.2 and Fig. 6]. This TBN features two large monomers of size n^2 in which n is a parameter in the design, as well as a variable number of additional monomers (“fuels”) intended to be present in large excess quantities.

In addition to the formulation we give in this paper, we also tested a number of formulation variants, including the StableGen algorithm originally posed in [3] for solving the STABLECONFIGS problem, justifying some of our design choices. “No Heuristic” performs a thorough accounting of all monomers (not just those needed to achieve saturation against the limiting monomers). “Labelled Monomers” assumes that the monomers are provided as a set, rather than a multiset. “Network Model” is a modification of StableGen with an alternate saturation constraint which does not require the explicit invocation of site-level bonds.

Each data point represents the average of three runs, and the solver was allowed to run for up to 100 seconds before a timeout was forced. Figure 2 (left) shows the runtimes as they increase with the parameter n , holding the count of each fuel at 2. Figure 2 (right) fixes $n = 3$ and shows the runtimes as they increase with the multiplicity of the fuel monomers. Note that our formulation can solve the case when fuels are in unbounded excess, while the variant formulations require bounded counts of all monomers.

Figure 2 (bottom) shows the runtime of our formulation in the case of unbounded fuels as it grows with increasing size parameter n . The rote specification of the benchmark problem is quadratic in n , and the empirical runtime growth appears polynomial with exponent close to this lower bound (2.71). However, we note that this is just a single family of problem instances, and we expect the runtime to be exponential in the worst case, since the problem is NP-hard.



■ **Figure 2** Empirical tests solving STABLECONFIGS for our benchmark problem based upon its complexity parameter n (left and bottom), and the multiplicity of the unstarred “fuel” strands (right). Our formulation is tested against several variations on the approach (which are described in the text) and the StableGen algorithm from [3]. The TBN is parameterized by n and contains the monomers $G_n = \{x_{ij}^* : 1 \leq i, j \leq n\}$, $H_i = \{x_{ij} : 1 \leq j \leq n\}$ for all $1 \leq i \leq n$, and $V_j = \{x_{ij} : 1 \leq i \leq n\} \cup \{x_{ij} : j \leq i \leq n\}$ for all $1 \leq j \leq n$. See Fig. 6 from [2] for a detailed explanation of this TBN and its operation. The vertical axis is log scale. Points at the top of the scale timed out after 100 seconds. The alternate formulations cannot solve the instance in the case of infinite fuel strands.

4 Computing bases of locally stable configurations of TBNs

We now shift attention to *locally* stable configurations: those in which no polymer can be split without breaking a bond. Such a configuration may not be stable, but the only paths to create more polymers, without breaking any bonds, require first merging existing polymers (i.e., going uphill in energy before going down). The saturated configurations are precisely those obtained by merging polymers starting from some locally stable configuration. In this section we describe a technique for computing what we call the *polymer basis*: the (finite) set of polymers that can exist in locally stable configurations. In Section 4.1, we show that an algebraic concept called the *Hilbert basis* [7] characterizes the polymer basis. In Sections 4.2 and 4.3 we show how the polymer basis can be used to reason about TBN behavior.

4.1 Equivalence of polymer bases and Hilbert bases

We note that the connection between Hilbert bases and polymer bases is not particularly deep and does not require clever techniques to prove. Once the definitions are appropriately set up, the equivalence follows almost immediately. (Though we provide a self-contained proof.) The primary insight of this section is that casting TBNs in our IP formulation sets up the connection with Hilbert bases. Since highly optimized software exists for computing Hilbert bases [1], this software can be deployed to automate reasoning about TBNs.

Let M be a set of monomer types with $m = |M|$. Let \mathcal{S}_M denote the *TBN schema* of M , the set of all TBNs containing only monomers from M , such that starred sites are limiting (i.e., such that saturated configurations have all starred sites bound). Let A_M be the *matrix representation* of the monomer types in \mathcal{S}_M , describing the contents of each monomer type: formally, the row- i , column- j entry of A_M is $\mathbf{m}_j(s_i)$, the net count of site type s_i in monomer type \mathbf{m}_j (as an example, $\{a^*, b, a, a, a, c, c^*, c^*\}$ has net count 2 of a , 1 of b , and -1 of c). Formally, a TBN $\mathcal{T} \in \mathcal{S}_M$ if and only if $A_M \mathcal{T} \geq \mathbf{0}$.

Recall that $\Gamma_{\mathcal{T}}$ is the set of saturated configurations of the TBN \mathcal{T} , and that a polymer \mathbf{P} is *self-saturated* if it has no exposed starred sites, i.e., $A_M \mathbf{P} \geq \mathbf{0}$. Define the *polymer basis* $\mathcal{B}_{\mathcal{S}_M}$ to be the set of all polymers \mathbf{P} with the following properties:

- $(\exists \mathcal{T} \in \mathcal{S}_M)(\exists \alpha \in \Gamma_{\mathcal{T}}) \mathbf{P} \in \alpha$ (i.e., \mathbf{P} appears in some saturated configuration of a TBN using only the monomer types from M .)
- There is no partition of \mathbf{P} into two (or more) self-saturated polymers.

For example, consider the monomers $G = \{a^*, b^*, c^*, d^*\}$, $H_1 = \{a, b\}$, $H_2 = \{c, d\}$, $V_1 = \{a, c\}$, $V_2 = \{b, d\}$ and let $M = \{G, H_1, H_2, V_1, V_2\}$. The polymer basis $\mathcal{B}_{\mathcal{S}_M}$ is $\{\{G, H_1, H_2\}, \{G, V_1, V_2\}, \{H_1\}, \{H_2\}, \{V_1\}, \{V_2\}\}$. All other self-saturated polymers are unions of these.

To show that polymer bases can be characterized by Hilbert bases, we must first define some additional terms. A *conical combination* of a set of vectors is a linear combination of the vectors using only nonnegative coefficients. An *integer conical combination* of a set of vectors is a conical combination of the vectors using only integer coefficients. A (*polyhedral*) *convex cone* $C = \{\lambda_1 \mathbf{a}_1 + \dots + \lambda_n \mathbf{a}_n : \lambda_1, \dots, \lambda_n \geq 0\}$ is the space of all conical combinations of a finite set of vectors $\{\mathbf{a}_1, \dots, \mathbf{a}_n\}$ (and is said to be *generated* by $\{\mathbf{a}_1, \dots, \mathbf{a}_n\}$). C is *pointed* if $C \cap (-C) = \{\mathbf{0}\}$. A set of the form $\{\mathbf{x} \in \mathbb{R}^m : A\mathbf{x} \geq \mathbf{0} \text{ and } \mathbf{x} \geq \mathbf{0}\}$ is always a pointed convex cone [7].

A set is *inclusion-minimal* with respect to a property if it has no proper subset that satisfies the same property. The *Hilbert basis* of a pointed convex cone C is the unique inclusion-minimal set of integer vectors such that every integer vector in C is an integer conical combination of the vectors in the Hilbert basis. For example, the Hilbert basis of the convex cone generated (with nonnegative real coefficients) by $(1, 3)$ and $(2, 1)$ is $\{(1, 1), (1, 2), (1, 3), (2, 1)\}$; note that $\frac{2}{5} \cdot (1, 3) + \frac{4}{5} \cdot (2, 1) = (2, 2)$, which is not an integer combination of $(1, 3)$ and $(2, 1)$, but $2 \cdot (1, 1) = (2, 2)$.

Recall that the matrix-vector product $A_M \mathbf{P}$ gives the number of exposed sites of each type in the polymer, so that $A_M \mathbf{P} \geq \mathbf{0}$ iff the polymer is self-saturated (i.e. none of the starred sites are exposed).

We are then interested in vectors contained in $\{\mathbf{P} \in \mathbb{N}^m : A_M \mathbf{P} \geq \mathbf{0}\}$. Noting that $\mathbb{N}^m = \{\mathbf{P} \in \mathbb{R}^m : \mathbf{P} \geq \mathbf{0}\} \cap \mathbb{Z}^m$, we can equivalently state that we are interested in all integer vectors contained in the pointed convex cone $\{\mathbf{P} \in \mathbb{R}^m : A_M \mathbf{P} \geq \mathbf{0} \text{ and } \mathbf{P} \geq \mathbf{0}\}$.

► **Theorem 4.** *Let \mathcal{S}_M be a TBN schema and let A_M be the matrix representation of its monomer types. Then the polymer basis $\mathcal{B}_{\mathcal{S}_M}$ of \mathcal{S}_M is the Hilbert basis of $\{\mathbf{P} \in \mathbb{R}^m : A_M \mathbf{P} \geq \mathbf{0} \text{ and } \mathbf{P} \geq \mathbf{0}\}$.*

Proof. Note that the integer vectors in $\{\mathbf{P} \in \mathbb{R}^m : A_M \mathbf{P} \geq \mathbf{0} \text{ and } \mathbf{P} \geq \mathbf{0}\}$ are precisely the polymers that appear in saturated configurations of TBNs in \mathcal{S}_M , since $A_M \mathbf{P} \geq \mathbf{0} \iff$ polymer \mathbf{P} is self-saturated, and \mathcal{S}_M is defined to have starred sites limiting, so that a configuration is saturated if and only if each of its polymers is self-saturated.

We must show two properties to establish that $\mathcal{B}_{\mathcal{S}_M}$ is the Hilbert basis. First we must show that every polymer in saturated configurations of TBNs in \mathcal{S}_M is a nonnegative integer combination of polymers in $\mathcal{B}_{\mathcal{S}_M}$. Next, to establish inclusion-minimality, we must show that no polymer can be removed from $\mathcal{B}_{\mathcal{S}_M}$ while satisfying the first property.

To see the first property, consider a polymer \mathbf{P} in a saturated configuration of some TBN in \mathcal{S}_M . If it cannot be split into multiple self-saturated polymers, then we are done since it is in $\mathcal{B}_{\mathcal{S}_M}$ (it is the integer combination consisting of one copy of itself). Otherwise, we can iteratively split \mathbf{P} into polymers $\mathbf{P}_1, \dots, \mathbf{P}_k$ that themselves cannot be split into self-saturated polymers. Then $\mathbf{P} = \mathbf{P}_1 + \dots + \mathbf{P}_k$.

To see the second property, suppose for the sake of contradiction that a polymer $\mathbf{P} \in \mathcal{B}_{\mathcal{S}_M}$ can be removed while maintaining the first property. By the definition of polymer basis, all polymers in $\mathcal{B}_{\mathcal{S}_M}$ are self-saturated, so \mathbf{P} is self-saturated. If \mathbf{P} can be removed from $\mathcal{B}_{\mathcal{S}_M}$ while maintaining the first property, then \mathbf{P} is the nonnegative integer sum of some polymers remaining in $\mathcal{B}_{\mathcal{S}_M} \setminus \{\mathbf{P}\}$, and these polymers must also be self-saturated by virtue of being in $\mathcal{B}_{\mathcal{S}_M}$. This means that \mathbf{P} can be partitioned into multiple self-saturated polymers, but then \mathbf{P} does not satisfy the second constraint required to be in $\mathcal{B}_{\mathcal{S}_M}$ to begin with. ◀

4.2 Using the polymer basis to reason about TBN behavior

The complexity of computing the polymer basis in general can be very large; however, once it is calculated, reasoning about the stable configurations becomes a simpler task. For instance, in a previous example we had $\mathcal{B}_{\mathcal{S}_M} = \{ \{G, H_1, H_2\}, \{G, V_1, V_2\}, \{H_1\}, \{H_2\}, \{V_1\}, \{V_2\} \}$. We can see from the above basis that in saturated configurations, G can only be present one of two unsplitable polymer types: $\{G, H_1, H_2\}$ or $\{G, V_1, V_2\}$, and we can optimize the number of polymers in a configuration by taking the other monomers as singletons (which is allowed, as these singletons are in the polymer basis). More generally, reasoning about stable configurations amounts to determining the number of each polymer type to use from the polymer basis so that the union of all polymers is the TBN, while using the maximum number of polymers possible. Our software can also solve for stable configurations in this way; specifically, for a TBN \mathcal{T} , it can calculate the polymer basis (abbreviated here as \mathcal{B}) and then solve for the stable configurations using the following IP:

$$\max_{\mathbf{c} \in \mathbb{N}^{|\mathcal{B}|}} \|\mathbf{c}\|_1 \text{ subject to } \sum_{i=1}^{|\mathcal{B}|} c_i \mathcal{B}_i = \mathcal{T}$$

Alternately, one can solve for the stable systems via an augmentation approach (see [7]).

If the goal is simply to solve the STABLECONFIGS problem, we do not expect that solving for the stable configurations in this way will be more efficient than the previous formulation, as the time spent computing the Hilbert basis alone can require a great deal longer than solving via the formulation of the previous section. Instead, the true value of the basis is in its ability to describe *all* saturated configurations of a TBN.

For instance, in [2], the authors define an augmented TBN model in which a system can move between saturated configurations by two atomic operations: polymers can be pairwise merged (with an energetic penalty, i.e., higher energy) or they can be split into two so long as no bonds are broken (with an energetic benefit, i.e., lower energy; for instance $\{a, b\}, \{a^*, b^*\}, \{a\}, \{a^*\}$ can be split into $\{a, b\}, \{a^*, b^*\}$ and $\{a\}, \{a^*\}$, whereas $\{a\}, \{a^*\}$ cannot be split). Any saturated polymer not in the basis can split into its basis components without breaking any bonds. Thus the polymer basis contains all polymers that can form in a *local* minimum energy configuration, i.e., one where no polymer can split.

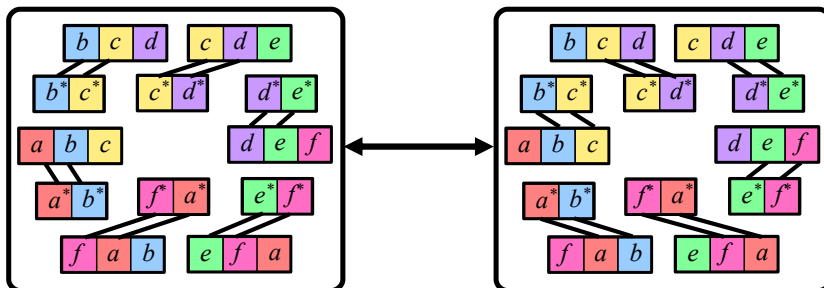
When designing a TBN, the designer will typically have a sense for which polymers are to be “allowed” in local energy minima. Proving that the system observes this behavior was not previously straightforward, but we can now observe that the TBN will behave ideally when its expected behavior matches its polymer basis.

4.3 A case example: Circular Translator Cascade

We now discuss an example of using the polymer basis to reason about a TBN’s kinetic behavior, studying a TBN known as a *circular translator cascade*, first defined in [2]:

$$\begin{aligned} & \{\{a, b, c\}, \{b, c, d\}, \{c, d, e\}, \{d, e, f\}, \{e, f, a\}, \{f, a, b\}, \\ & \{a^*, b^*\}, \{b^*, c^*\}, \{c^*, d^*\}, \{d^*, e^*\}, \{e^*, f^*\}, \{f^*, a^*\}\} \end{aligned}$$

There are two stable configurations of this TBN, shown in Figure 3.



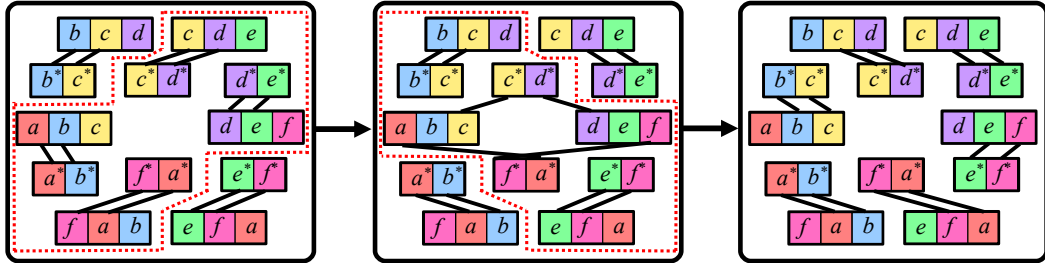
■ **Figure 3** The two stable configurations of a variant of the circular translator cascade described in [2]. In the left configuration, the unstarred monomers are bound to their “left-side” companions (e.g. $\{a, b, c\}$ is bound to $\{a^*, b^*\}$), and in the right configuration, the unstarred monomers are bound to their “right-side” companions (e.g. $\{a, b, c\}$ is bound to $\{b^*, c^*\}$).

We consider now the “pathways” by which one of the stable configurations can “transition” to another. This process is described formally in [2]; here we give an intuitive description. Informally, we admit as atomic operations the ability for two polymers to merge or for one polymer to split into two polymers, so long as the resulting configuration remains saturated. In essence, these operations are modelling the physical phenomenon of solutes colocalizing in solution before reactions occur, specifically in dilute solutions in which enthalpic bond rearrangements occur on a timescale much faster than the timescale for entropic colocalization. If many polymers must be merged in some intermediate configuration to transition between stable configurations, then since each merge is individually unlikely, the successive merges required are exponentially unlikely i.e., a large *energy barrier* exists to transition between the configurations.

The design intention of this TBN is to have two stable configurations with a large energy barrier to transition between them. For the largest possible energy barrier, the transition should require the simultaneous merging of *all* of the polymers into a single polymer as an intermediate step. However, this is not the case for the TBN of Figure 3; the polymer basis gives insight into why. See [2, Section A.2] for an argument why more domain types and monomer types are required. We interpret this as a design error. We now explain how the error can be detected by reasoning about the polymer basis of the system, justifying that the automated computation of the polymer basis by our software enables one to automate some reasoning about the correct behavior of TBNs.

If it were true that the polymer basis contained only the 12 polymer types that are present in the two stable configurations of Figure 3, then that would be sufficient to prove the high energy barrier. To see why this is true, suppose there were a locally stable intermediate configuration that is part of a lower barrier transition. Since the configuration is locally stable, it is saturated, and none of its polymers can be partitioned into self-saturated polymers. By definition, the polymer basis should then contain all of the polymers present in this

intermediate configuration. However, all of the polymers in the basis have exactly two monomers, and so there must be 6 polymers in the intermediate configuration. The stable configurations also have 6 polymers, and so the intermediate configuration is also stable, but this contradicts that there are only two stable configurations.



■ **Figure 4** A counterexample to the claim that transitioning between stable configurations of this TBN requires the simultaneous merger of all monomers into a single polymer. Starting from the stable configuration on the left, by merging the four polymers within the red dotted outline, it is possible to re-arrange bonds and then split to the middle configuration. Then from the middle configuration, by merging the three polymers in the red dotted outline, it is possible to re-arrange bonds and then split to the stable configuration on the right. Such intermediate configurations are evident by examining the elements of the polymer basis.

In fact, the polymer basis for this TBN has 57 entries (determined via our software), not 12, and we can use this basis to disprove the high energy barrier, i.e., to show that there is a sequence of merges and splits that transitions between the two stable configurations, without all monomers ever being merged into a single polymer. To discover a pathway that demonstrates the lower energy barrier, consider one unexpected entry in the polymer basis: $\mathbf{P} = \{\{a, b, c\}, \{d, e, f\}, \{c^*, d^*\}, \{f^*, a^*\}\}$. Its existence in the polymer basis tells us that there must be some saturated configuration that contains it. If we examine where these monomers were in one of the original stable configurations (Figure 3, left), we see that these were originally in polymers

$$\{\{a, b, c\}, \{a^*, b^*\}\}, \quad \{\{c, d, e\}, \{c^*, d^*\}\}, \quad \{\{d, e, f\}, \{d^*, e^*\}\}, \quad \{\{f, a, b\}, \{f^*, a^*\}\}.$$

From the starting configuration, if only these four polymers were merged, then they could then iteratively split into \mathbf{P} , $\{\{c, d, e\}, \{d^*, e^*\}\}$, and $\{\{f, a, b\}, \{a^*, b^*\}\}$. Since the latter two polymers are part of the target configuration, one could now greedily merge all polymers except for these latter two and then split into the target configuration. At no point in the interim were all polymers merged together into a single polymer. The resulting pathway is illustrated in Figure 4.

The difference between intended and actual barrier in this design becomes more pronounced if it is scaled up to include more site types and monomers. In [2] it is shown that by modifying the design, it is possible to achieve a linear energy barrier by using a quadratic number of site types.

5 Conclusion

In our investigation we observed that it was generally more efficient to solve SATURATED-CONFIGS by finding the optimal objective value using an IP solver as a first step, followed by using a CP solver on the same formulation with the objective value now constrained to the value found by the IP solver. Are further computational speedups possible by using IP as a callback during the CP search, instead of only in the beginning? How would one formulate the subproblems that would need to be solved in these callbacks?

In this paper we also note the value of polymer bases that are derived from the matrix containing the monomer descriptions. Such polymer bases can be used to describe all saturated configurations of a TBN, and so provide a valuable tool for analyzing potential behavior of a TBN when the model is augmented with rules that allow for dynamics. In practice, rather than discover unexpected behavior by calculating the polymer basis, a designer would instead like to begin with a set of behaviors and then create a TBN that respects them. Can we begin from verifiable polymer/Hilbert bases, encoding desired behavior, and transform them into TBN/DNA designs?

The full TBN model [2] can also be used to describe the regime in which there is a more modest tradeoff between the two energetic penalties of breaking bonds and merging complexes (i.e., saturation is not guaranteed). For example toehold-length binding sites in DNA strand displacement systems [4, 16, 19, 24, 25] are intended to dissociate over timescales comparable to association. Indeed, our software [13] includes an implementation of the STABLECONFIGS formulation in which this relative weighting factor is included in the objective function. Under what conditions can a comparable polymer basis for such a system be found? Within the context of integer programming, it is known that by adding constraints to the design, one can reduce the complexity of finding/verifying Hilbert bases (and related *Graver bases*) [10], but it is not clear how to interpret these numeric constraints within the context of TBNs.

Satisfiability Modulo Theory (SMT) formulations have the ability to express TBN concepts (such as reachability along kinetic paths) without converting to a linear algebra framework, and existing solvers can solve SMT instances with surprising efficiency (for example, Z3 [8]). Can such solvers reason about TBNs directly within a reasonable time frame? Can they efficiently extract information beyond what is contained in the polymer basis?

References

- 1 4ti2 team. 4ti2 – A software package for algebraic, geometric and combinatorial problems on linear spaces. <https://4ti2.github.io/hilbert.html>. URL: <https://4ti2.github.io>.
- 2 Keenan Breik, Cameron Chalk, David Haley, David Doty, and David Soloveichik. Programming substrate-independent kinetic barriers with thermodynamic binding networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 18(1):283–295, 2021. Special issue of invited papers from CMSB 2018.
- 3 Keenan Breik, Chris Thachuk, Marijn Heule, and David Soloveichik. Computing properties of stable configurations of thermodynamic binding networks. *Theoretical Computer Science*, 785:17–29, 2019.
- 4 Yuan-Jyue Chen, Neil Dalchau, Niranjan Srinivas, Andrew Phillips, Luca Cardelli, David Soloveichik, and Georg Seelig. Programmable chemical controllers made from DNA. *Nature Nanotechnology*, 8(10):755–762, 2013.
- 5 Yuan-Jyue Chen, Benjamin Groves, Richard A. Muscat, and Georg Seelig. DNA nanotechnology from the test tube to the cell. *Nature Nanotechnology*, 10:748–760, 2015.
- 6 Michele Conforti, Gérard Cornuéjols, Giacomo Zambelli, et al. *Integer programming*, volume 271. Springer, 2014.
- 7 Jesús A De Loera, Raymond Hemmecke, and Matthias Köppe. *Algebraic and geometric ideas in the theory of discrete optimization*. SIAM, 2012.
- 8 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- 9 David Doty, Trent A Rogers, David Soloveichik, Chris Thachuk, and Damien Woods. Thermodynamic binding networks. In *DNA 2017: Proceedings of the 23rd International Meeting on DNA Computing and Molecular Programming*, pages 249–266. Springer, 2017.

- 10 Friedrich Eisenbrand, Christoph Hunkenschroder, Kim-Manuel Klein, Martin Koutecký, Asaf Levin, and Shmuel Onn. An algorithmic theory of integer programming. *arXiv preprint*, 2019. [arXiv:1904.01361](https://arxiv.org/abs/1904.01361).
- 11 Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Wei-Kun Chen, Leon Eifler, Maxime Gasse, Patrick Gemander, Ambros Gleixner, Leona Gottwald, Katrin Halbig, Gregor Hendel, Christopher Hojny, Thorsten Koch, Pierre Le Bodic, Stephen J. Maher, Frederic Matter, Matthias Miltenberger, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Christine Tawfik, Stefan Vigerske, Fabian Wegscheider, Dieter Weninger, and Jakob Witzig. The SCIP Optimization Suite 7.0. Technical report, Optimization Online, March 2020. URL: http://www.optimization-online.org/DB_HTML/2020/03/7705.html.
- 12 LLC Gurobi Optimization. Gurobi optimizer reference manual, 2020. URL: <http://www.gurobi.com>.
- 13 David Haley. Stable-TBN – a software package for computing the stable configurations of thermodynamic binding networks. URL: https://github.com/drhaley/stable_tbn.
- 14 Dionis Mineev, Christopher M Wintersinger, Anastasia Ershova, and William M Shih. Robust nucleation control via crisscross polymerization of highly coordinated DNA slats. *Nature Communications*, 12(1):1–9, 2021.
- 15 Laurent Perron and Vincent Furnon. OR-tools. URL: <https://developers.google.com/optimization>.
- 16 Lulu Qian and Erik Winfree. Scaling up digital circuit computation with DNA strand displacement cascades. *Science*, 332(6034):1196–1201, 2011.
- 17 Paul W. K. Rothemund. Folding DNA to create nanoscale shapes and patterns. *Nature*, 440(7082):297–302, 2006.
- 18 Paul Shaw, Vincent Furnon, and Bruno De Backer. A constraint programming toolkit for local search. In *Optimization Software Class Libraries*, pages 219–261. Springer, 2003.
- 19 Niranjana Srinivas, James Parkin, Georg Seelig, Erik Winfree, and David Soloveichik. Enzyme-free nucleic acid dynamical systems. *Science*, 358(6369):eaal2052, 2017.
- 20 Chris Thachuk, Erik Winfree, and David Soloveichik. Leakless DNA strand displacement systems. In *DNA 2015: Proceedings of the 21st International Meeting on DNA Computing and Molecular Programming*, pages 133–153. Springer, 2015.
- 21 Andrew C Trapp and Oleg A Prokopyev. Solving the order-preserving submatrix problem via integer programming. *INFORMS Journal on Computing*, 22(3):387–400, 2010.
- 22 Boya Wang, Chris Thachuk, Andrew D Ellington, Erik Winfree, and David Soloveichik. Effective design principles for leakless strand displacement systems. *Proceedings of the National Academy of Sciences*, 115(52):E12182–E12191, 2018.
- 23 Damien Woods[†], David Doty[†], Cameron Myhrvold, Joy Hui, Felix Zhou, Peng Yin, and Erik Winfree. Diverse and robust molecular algorithms using reprogrammable DNA self-assembly. *Nature*, 567(7748):366–372, 2019. [†]joint first authors. [doi:10.1038/s41586-019-1014-9](https://doi.org/10.1038/s41586-019-1014-9).
- 24 David Yu Zhang and Georg Seelig. Dynamic DNA nanotechnology using strand-displacement reactions. *Nature chemistry*, 3(2):103–113, 2011.
- 25 David Yu Zhang and Erik Winfree. Control of DNA strand displacement kinetics using toehold exchange. *Journal of the American Chemical Society*, 131(47):17303–17314, 2009.

Self-Replication via Tile Self-Assembly (Extended Abstract)

Andrew Alseth ✉ 

University of Arkansas, Fayetteville, AR, USA

Daniel Hader ✉

University of Arkansas, Fayetteville, AR, USA

Matthew J. Patitz ✉ 

University of Arkansas, Fayetteville, AR, USA

Abstract

In this paper we present a model containing modifications to the Signal-passing Tile Assembly Model (STAM), a tile-based self-assembly model whose tiles are capable of activating and deactivating glues based on the binding of other glues. These modifications consist of an extension to 3D, the ability of tiles to form “flexible” bonds that allow bound tiles to rotate relative to each other, and allowing tiles of multiple shapes within the same system. We call this new model the STAM*, and we present a series of constructions within it that are capable of self-replicating behavior. Namely, the input seed assemblies to our STAM* systems can encode either “genomes” specifying the instructions for building a target shape, or can be copies of the target shape with instructions built in. A universal tile set exists for any target shape (at scale factor 2), and from a genome assembly creates infinite copies of the genome as well as the target shape. An input target structure, on the other hand, can be “deconstructed” by the universal tile set to form a genome encoding it, which will then replicate and also initiate the growth of copies of assemblies of the target shape. Since the lengths of the genomes for these constructions are proportional to the number of points in the target shape, we also present a replicator which utilizes hierarchical self-assembly to greatly reduce the size of the genomes required. The main goals of this work are to examine minimal requirements of self-assembling systems capable of self-replicating behavior, with the aim of better understanding self-replication in nature as well as understanding the complexity of mimicking it.

2012 ACM Subject Classification Theory of computation → Models of computation; General and reference → General conference proceedings

Keywords and phrases Algorithmic self-assembly, tile assembly model, self-replication

Digital Object Identifier 10.4230/LIPIcs.DNA.27.3

Related Version *Full Version*: <https://arxiv.org/abs/2105.02914> [2]

Funding *Andrew Alseth*: This author’s work was supported in part by NSF grant CAREER-1553166.

Daniel Hader: This author’s work was supported in part by NSF grant CAREER-1553166.

Matthew J. Patitz: This author’s work was supported in part by NSF grant CAREER-1553166.

1 Introduction

1.1 Background and motivation

Research in tile based self-assembly is typically focused on modeling the computational and shape-building capabilities of biological nano-materials whose dynamics are rich enough to allow for interesting algorithmic behavior. Polymers such as DNA, RNA, and poly-peptide chains are of particular interest because of the complex ways in which they can fold and bind with both themselves and others. Even when only taking advantage of a small subset of the dynamics of these materials, with properties like binding and folding generally being restricted to very manageable cases, tile assembly models have been extremely successful in exhibiting vast arrays of interesting behavior [45, 48, 17, 11, 13, 38, 50, 16, 5, 8, 15]. Among



© Andrew Alseth, Daniel Hader, and Matthew J. Patitz;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on DNA Computing and Molecular Programming (DNA 27).

Editors: Matthew R. Lakin and Petr Šulc; Article No. 3; pp. 3:1–3:22

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

other things, a typical question in the realm of algorithmic tile assembly asks what the minimal set of requirements is to achieve some desired property. Such questions can range from very concrete, such as “how many distinct tile types are necessary to construct specific shapes?”, to more abstract such as “under what conditions is the construction of self-similar fractal-like structures possible?”. Since the molecules inspiring many tile assembly models are used in nature largely for the purpose of self-replication of living organisms, a natural tile assembly question is thus whether or not such behavior is possible to model algorithmically.

In this paper we show that we can define a model of tile assembly in which the complexities of self-replication type behavior can be captured, and provide constructions in which such behavior occurs. We define our model with the intention of it (1) being hopefully physically implementable in the (near) future, and (2) using as few assumptions and constraints as possible. Our constructions therefore provide insight into understanding the basic rules under which the complex dynamics of life, particularly self-replication, may occur.

We chose to use the Signal-passing Tile Assembly Model (STAM) as a basis for our model, which we call the STAM*, because (1) there has been success in physically realizing such systems [41] and potential exists for further, more complex, implementations using well-established technologies like DNA origami [44, 39, 52, 3, 4] and DNA strand displacement [43, 51, 47, 54, 53, 7], and (2) the STAM allows for behavior such as cooperative tile attachment as well as detachment of subassemblies. We modify the STAM by bringing it into 3 dimensions and making a few simplifying assumptions, such as allowing multiple tile shapes and tile rotation around flexible glues and removing the restriction that tiles have to remain on a fixed grid. Allowing flexibility of structures and multiple tile shapes provides powerful new dynamics that can mimic several aspects of biological systems and suffice to allow our constructions to model self-replicating behavior. Prior work, theoretical [36] and experimental [46], has focused on the replication of patterns of bits/letters on 2D surfaces, as well as the replication of 2D shapes in a model using staged assembly [1], or in the STAM [27]. However, all of these are fundamentally 2D results and our 3D results, while strictly theoretical, are a superset with constructions capable of replicating all finite 2D and 3D patterns and shapes.

Biological self-replication requires three main categories of components: (1) instructions, (2) building blocks, and (3) molecular machinery to read the instructions and combine building blocks in the manner specified by the instructions. We can see the embodiment of these components as follows: (1) DNA/RNA sequences, (2) amino acids, and (3) RNA polymerase, transfer RNA, and ribosomes, among other things. With our intention to study the simplest systems capable of replication, we started by developing what we envisioned to be the simplest model that would provide the necessary dynamics, the STAM*, and then designed modular systems within the STAM* which each demonstrated one or more important behaviors related to replication. Quite interestingly, and unintentionally, our constructions resulted in components with strong similarities to biological counterparts. As our base encoding of the instructions for a target shape, we make use of a linear assembly which has some functional similarity to DNA. Similar to DNA, this structure also is capable of being replicated to form additional copies of the “genome”. In our main construction, it is necessary for this linear sequence of instructions to be “transcribed” into a new assembly which also encodes the instructions but which is also functionally able to facilitate translation of those instructions into the target shape. Since this sequence is also degraded during the growth of the target structure, it shares some similarity with RNA and its role in replication. Our constructions don’t have an analog to the molecular machinery of the ribosome, and can therefore “bootstrap” with only singleton copies of tiles from our universal set of tiles in solution. However, to balance the fact that we don’t need preexisting machinery, our building blocks are more complicated than amino acids, instead being tiles capable of a constant number of signal operations each (turning glues on or off due to the binding of other glues).

1.2 Our results

Beyond the definition of the STAM* as a new model, we present a series of STAM* constructions. They are designed and presented in a modular fashion, and we discuss the ways in which they can be combined to create various (self-)replicating systems.

1.2.1 Genome-based replicator

We first develop an STAM* tileset which functions as a simple self-replicator (in Section 3) that begins from a seed assembly encoding information about a target structure, a.k.a. a *genome*, and grows arbitrarily many copies of the genome and target structure, a.k.a. the *phenotype*. This tileset is universal for all 3D shapes comprised of $1 \times 1 \times 1$ cubes when they are inflated to scale factor 2 (i.e. each $1 \times 1 \times 1$ block in the shape is represented by a cube of $2 \times 2 \times 2$ tiles). This construction requires a genome whose length is proportional to the number of cube tiles in the phenotype; for non-trivial shapes the genome is a constant factor longer in order to follow a Hamiltonian path through an arbitrary 3D shape at scale factor 2. This is compared to the Soloveichik and Winfree universal (2D) constructor [49] where a “genome” is optimally shortened, but the scale factor of blocks is much larger.

The process by which this occurs contains analogs to natural systems. We progress from a genome sequence (acting like DNA), which is translated into a messenger sequence (somewhat analogous to RNA), that is modified and consumed in the production of tertiary structures (analogous to proteins). We have a number of helper structures that fuel both the replication of the genome and the translation of the messenger sequence.

1.2.2 Deconstructive self-replicator

In Section 4, we construct an STAM* tileset that can be used in systems in which an arbitrarily shaped seed structure, or phenotype, is disassembled while simultaneously forming a genome that describes its structure. This genome can then be converted into a linear genome (of the form used for the first construction) to be replicated arbitrarily and can be used to construct a copy of the phenotype. We show that this can be done for any 3D shape at scale factor 2 which is sufficient, and in some cases necessary, to allow for a Hamiltonian path to pass through each point in the shape. This Hamiltonian path, among other information necessary for the disassembly and, later, reassembly processes, is encoded in the glues and signals of the tiles making up the phenotype. We then show how, using simple signal tile dynamics, the phenotype can be disassembled tile by tile to create a genome encoding that same information. Additionally, a reverse process exists so that once the genome has been constructed from a phenotype, a very similar process can be used to reconstruct the phenotype while disassembling the genome.

In sticking with the DNA, RNA, protein analogy, this disassembly process doesn't have a particular biological analog; however, this result is important because it shows that we can make our system robust to starting conditions. That is, we can begin the self-replication process at any stage be it from the linear genome, “kinky genome” (the messenger sequence from the first construction), or phenotype. Finally, since this construction requires the phenotype to encode information in its glues and signals, we show that this can be computed efficiently using a polynomial time algorithm given the target shape. This not only shows that the STAM* systems can be described efficiently for any target shape via a single universal tile set, but that results from intractable computations aren't built into our phenotype (i.e. we're not “cheating” by doing complex pre-computations that couldn't be done efficiently by a typical computationally universal system). Due to space constraints we only include a result about the necessity for deconstruction in a universal replicator in the online version [2].

1.2.3 Hierarchical assembly-based replicator

For our final construction, in Section 5, our aims were twofold. First, we wanted to compress the genome so that its total length is much shorter than the number of tiles in the target shape. Second, we wanted to more closely mimic the biological process in which individual proteins are constructed via the molecular machinery, and then they are released to engage in a hierarchical self-assembly process in which proteins combine to form larger structures.

Biological genomes are many orders of magnitude smaller than the organisms which they encode, but for our previous constructions the genomes are essentially equivalent in size to the target structures. Our final construction is presented in a “simple” form in which the general scaling approximately results in a genome which is length $n^{\frac{1}{3}}$ for a target structure of size n . However, we discuss relatively simple modifications which could, for some target shapes, result in genome sizes of approximately $\log n$, and finally we discuss a more complicated extension (which also consumes a large amount of “fuel”, as opposed to the base constructions which consume almost no fuel) that can achieve asymptotically optimal encoding.

1.2.4 Combinations and permutations of constructions

Due to length restrictions for this version of the paper, and our desire to present what we found to be the “simplest” systems capable of combining to perform self-replication, there are several additions to our results which we only briefly mention. For instance, to make our first construction (in Section 3) into a standalone self-replicator, and one which functions slightly more like biological systems, the input to the system, i.e. the seed assembly, could instead be a copy of the target structure with a genome “tail” attached to it. The system could function very similarly to the construction of Section 3 but instead of genome replication and structure building being separated, the genome could be replicated and then initiate the growth of a connected messenger structure so that once the target structure is completed, the genome is attached. Thus, the input assembly would be completely replicated, and be a self-replicator more closely mirroring biology where the DNA along with the structure cause the DNA to replicate itself and the structure. Attaching the genome to the structure is a technicality that could satisfy the need to have a single seed assembly type, but clearly it doesn’t meaningfully change the behavior. At the end of Section 5 we discuss how that construction could be combined with those from Sections 3 and 4, as well as further optimized. This version of the paper contains high-level overviews of the definition of the STAM* as well as of the results. Full technical details for each section can be found in the full version online [2] in the corresponding sections of the technical appendix.

2 Preliminaries

In this section we define the notation and models used throughout the paper.

We define a *3D shape* $S \subset \mathbb{Z}^3$ as a finite connected set of $1 \times 1 \times 1$ cubes (a.k.a. unit cubes) which define an arbitrary polycube, i.e. a shape composed of unit cubes connected face to face where each cube represents a voxel (3-D pixel) of S . For each shape S , we assume a canonical translation and rotation of S so that, without loss of generality, we can reference the coordinates of each of its voxels and directions of its surfaces, or faces. We say a unit cube is *scaled by factor* c if it is replaced by a $c \times c \times c$ cube composed of c^3 unit cubes. Given an arbitrary 3D shape S , we say S is *scaled by factor* c if every unit cube of S is scaled by factor c and those scaled cubes are arranged in the shape of S . We denote a shape S scaled by factor c as S^c .

2.1 Definition of the STAM*

The 3D Signal-passing Tile Assembly Model* (3D-STAM*, or simply STAM*) is a generalization of the STAM [40, 20, 26, 37] (that is similar to the model in [30, 31]) in which (1) the natural extension from 2D to 3D is made (i.e. tiles become 3-dimensional shapes rather than 2-dimensional squares), (2) multiple tile shapes are allowed, (3) tiles are allowed to flip and rotate [11, 28], and (4) glues are allowed to be rigid (as in the aTAM, 2HAM, STAM, etc., meaning that when two adjacent tiles bind to each other via a rigid glue, their relative orientations are fixed by that glue) or *flexible* (as in [18]) so that even after being bound tiles and subassemblies are free rotate with respect to tiles and subassemblies to which they are bound by bending or twisting around a “joint” in the glue. (This would be analogous to rigid glues forming as DNA strands combine to form helices with no single-stranded gaps, while flexible glues would have one or more unpaired nucleotides leaving a portion of single-stranded DNA joining the two tiles, which would be flexible and rotatable.) See Figure 1a for a simple example. These extensions make the STAM* a hybrid model of those in previous studies of hierarchical assembly [8, 12, 14, 42, 29], 3D tile-based self-assembly [10, 22, 6, 24], systems allowing various non-square/non-cubic tile types [19, 23, 11, 21, 25, 35], and systems in which tiles can fold and rearrange [18, 34, 32, 33].

Due to space constraints, we now provide a high-level overview of several aspects of the STAM* model, and full definitions can be found in the online version [2].

The basic components of the model are *tiles*. Tiles bind to each other via *glues*. Each glue has a *glue type* that specifies its domain (which is the string label of the glue), integer strength, *flexibility* (a boolean value with **true** meaning *flexible* and **false** meaning *rigid*), and length (representing the length of the physical glue component). A glue is an instance of a glue type and may be in one of three states at any given time, $\{\text{latent}, \text{on}, \text{off}\}$. A pair of adjacent glues are able to bind to each other if they have complementary domains and are both in the **on** state, and do so with strength equal to their shared strength values (which must be the same for all glues with the same label l or the complementary label l^*).

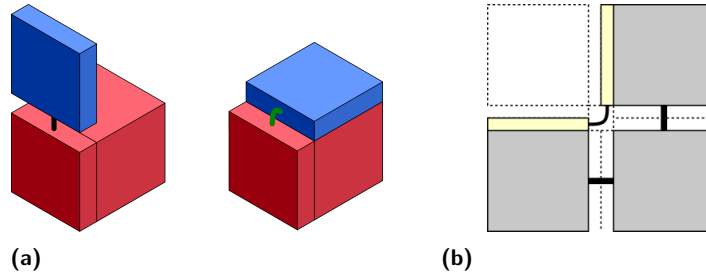
A *tile type* is defined by its 3D shape (and although arbitrary rotation and translation in \mathbb{R}^3 are allowed, each is assigned a canonical orientation for reference), its set of glues, and its set of *signals*. Its set of glues specify the types, locations, and initial states of its glues. Each signal in its set of signals is a triple (g_1, g_2, δ) where g_1 and g_2 specify the *source* and *target* glues (from the set of the tile type’s glues) and $\delta \in \{\text{activate}, \text{deactivate}\}$. Such a signal denotes that when glue g_1 forms a bond, an action is initiated to turn glue g_2 either **on** (if $\delta == \text{activate}$) or **off** (otherwise). A *tile* is an instance of a tile type represented by its type, location, rotation, set of glue states (i.e. **latent**, **on** or **off** for each), and set of *signal states*. Each signal can be in one of the signal states $\{\text{pre}, \text{firing}, \text{post}\}$. A signal which has never been activated (by its source glue forming a bond) is in the **pre** state. A signal which has activated but whose action has not yet completed is in the **firing** state, and if that action has completed it is in the **post** state. Each signal can “fire” only one time, and each glue which is the target of one or more signals is only allowed to make the following state transitions: (1) **latent** \rightarrow **on**, (2) **on** \rightarrow **off**, or (3) **latent** \rightarrow **off**.

We use the terms *assembly* and *supertile*, interchangeably, to refer to the full set of rotations and translations of either a single tile (the base case) or a collection of tiles which are bound together by glues. A supertile is defined by the tiles it contains (which includes their glue and signal states) and the glue bonds between them. A supertile may be flexible (due to the existence of a cut consisting entirely of flexible glues that are co-linear and there being an unobstructed path for one subassembly to rotate relative to the other), and we call each valid positioning of its sets of subassemblies a *configuration* of the supertile. A supertile may also be translated and rotated while in any valid configuration. We call a supertile in a particular configuration, rotation, and translation a *positioned supertile*.

3:6 Self-Replication

Each supertile induces a *binding graph*, a multigraph whose vertices are tiles, with an edge between two tiles for each glue which is bound between them. The supertile is τ -stable if every cut of its binding graph has strength at least τ , where the weight of an edge is the strength of the glue it represents. That is, the supertile is τ -stable if cutting bonds of at least summed strength of τ is required to separate the supertile into two parts.

For a supertile α , we use the notation $|\alpha|$ to represent the number of tiles contained in α . The *domain* of a positioned supertile α , written $\text{dom } \alpha$, is the union of the points in \mathbb{R}^3 contained within the tiles composing α . Let α be a positioned supertile. Then, for $\vec{v} \in \mathbb{R}^3$, we define the partial function $\alpha(\vec{v}) = t$ where t is the tile containing \vec{v} if $\vec{v} \in \text{dom } \alpha$, otherwise it is undefined. Given two positioned supertiles, α and β , we say that they are *equivalent*, and we write $\alpha \approx \beta$, if for all $\vec{v} \in \mathbb{R}^3$ $\alpha(\vec{v})$ and $\beta(\vec{v})$ both either return tiles of the same type, or are undefined. We say they're *equal*, and write $\alpha \equiv \beta$, if for all $\vec{v} \in \mathbb{R}^3$ $\alpha(\vec{v})$ and $\beta(\vec{v})$ either both return tiles of the same type having the same glue and signal states, or are undefined.



■ **Figure 1** (a) Example showing flat and cubic tiles, and possible behavior of a flexible glue allowing the blue tile to fold upward, away from the red cubic tile, or down against it. (b) The glue lengths in our constructions: (1) length 2ϵ rigid bonds between cubic tiles, (2) length 0 rigid bonds between flat and cubic tiles (as though one tile's glue strand binds into a cavity), and (3) length $3\sqrt{2} \epsilon/2$ flexible glues between flat tiles.

An STAM* tile assembly system, or TAS, is defined as $\mathcal{T} = (T, C, \tau)$ where T is a finite set of tile types, C is an initial configuration, and $\tau \in \mathbb{N}$ is the minimum binding threshold (a.k.a. temperature) specifying the minimum binding strength that must exist over the sum of binding glues between two supertiles in order for them to attach to each other. The initial configuration $C = \{(S, n) \mid S \text{ is a supertile over the tiles in } T \text{ and } n \in \mathbb{N} \cup \infty \text{ is the number of copies of } S\}$. Note that for each $s \in S$, each tile $\alpha = (t, \vec{l}, S, \gamma) \in s$ has a set of glue states S and signal states γ . By default, it is assumed that every tile in every supertile of an initial configuration begins with all glues in the initial states for its tile type, and with all signal states as **pre**, unless otherwise specified. The initial configuration C of a system \mathcal{T} is often simply given as a set of supertiles, which are also called *seed* supertiles, and it is assumed that there are infinite counts of each seed supertile as well as of all singleton tile types in T . If there is only one seed supertile σ , we will we often just use σ rather than C .

2.1.1 Overview of STAM* dynamics

An STAM* system $\mathcal{T} = (T, C, \tau)$ evolves nondeterministically in a series of (a possibly infinite number of) steps. Each step consists of randomly executing one of the following actions: (1) selecting two existing supertiles which have configurations allowing them to combine via a set of neighboring glues in the **on** state whose strengths sum to strength $\geq \tau$ and combining them via a random subset of those glues whose strengths sum to $\geq \tau$ (and changing any signals with those glues as sources to the state **firing** if they are in state **pre**),

or (2) randomly select two adjacent unbound glues of a supertile which are able to bind, bind them and change attached signals in state **pre** to **firing**, or (3) randomly select a supertile which has a cut $< \tau$ (due to glue deactivations) and cause it to *break* into 2 supertiles along that cut, or (4) randomly select a signal on some tile of some supertile where that signal is in the **firing** state and change that signal's state to **post**, and as long as its action (**activate** or **deactivate**) is currently valid for the signal's target glue, change the target glue's state appropriately.¹ Although at each step the next choice is random, it must be the case that no possible selection is ever ignored infinitely often.

Given an STAM* TAS $\mathcal{T} = (T, C, \tau)$, a supertile is *producible*, written as $\alpha \in \mathcal{A}[\mathcal{T}]$, if either it is a single tile from T , or it is the result of a (possibly infinite) series of combinations of pairs of finite producible assemblies (which have each been positioned so that they do not overlap and can be τ -stably bonded), and/or breaks of producible assemblies. A supertile α is *terminal*, written as $\alpha \in \mathcal{A}_{\square}[\mathcal{T}]$, if (1) for every $\beta \in \mathcal{A}[\mathcal{T}]$, α and β cannot be τ -stably attached, (2) there is no configuration of α in which a pair of unbound complementary glues in the **on** state are able to bind, and (3) no signals of any tile in α are in the **firing** state.

In this paper, we define a shape as a connected subset of \mathbb{Z}^3 to both simplify the definition of a shape and to capture the notion that to build an arbitrary shape out of a set of tiles we will actually approximate it by “pixelating” it. Therefore, given a shape S , we say that assembly α has shape S if α has only one valid configuration (i.e. it is *rigid*) and there exist (1) a rotation of α and (2) a scaling of S , S' , such that the rotated α and S' can be translated to overlap where there is a one-to-one and onto correspondence between the tiles of α and cubes of S' (i.e. there is exactly 1 tile of α in each cube of S' , and none outside of S').²

► **Definition 1.** We say a shape X self-assembles in \mathcal{T} with waste size c , for $c \in \mathbb{N}$, if there exists terminal assembly $\alpha \in \mathcal{A}_{\square}[\mathcal{T}]$ such that α has shape X , and for every $\alpha \in \mathcal{A}_{\square}[\mathcal{T}]$, either α has shape X , or $|\alpha| \leq c$. If $c = 1$, we simply say X self-assembles in \mathcal{T} .

► **Definition 2.** We call an STAM* system $\mathcal{R} = (T, C, \tau)$ a shape self-replicator for shape S if C consists exactly of infinite copies of each tile from T as well as of a single supertile σ of shape S , there exists $c \in \mathbb{N}$ such that S self-assembles in \mathcal{R} with waste size c , and the count of assemblies of shape S increases infinitely.

► **Definition 3.** We call an STAM* system $\mathcal{R} = (T, C, \tau)$ a self-replicator for σ with waste size c if C consists exactly of infinite copies of each tile from T as well as of a single supertile σ , there exists $c \in \mathbb{N}$ such that for every terminal assembly $\alpha \in \mathcal{A}_{\square}[\mathcal{T}]$ either (1) $\alpha \approx \sigma$, or (2) $|\alpha| \leq c$, and the count of assemblies $\approx \sigma$ increases infinitely.³ If $c = 1$, we simply say \mathcal{R} is a self-replicator for σ .

The multiple aspects of STAM* tiles and systems give rise to a variety of metrics with which to characterize and measure the complexity of STAM* systems, beyond metrics seen for models such as the aTAM or even STAM. For a brief discussion, please see the online version [2].

¹ The asynchronous nature of signal firing and execution is intended to model a signalling process which can be arbitrarily slow or fast. Please see the online version [2] for more details.

² In this paper we only consider completely rigid assemblies for target shapes, since the target shapes are static. We could also target “reconfigurable shapes, i.e. sets of shapes, but don't do so in this paper. Also, it could be reasonable to allow multiple tiles in each pixel location as long as the correct overall shape is maintained, but we don't require that.

³ We use \approx rather than \equiv since otherwise either both the seed assemblies and produced assemblies are terminal, meaning nothing can attach to a seed assembly and the system can't evolve, or neither are terminal and it becomes difficult to define the product of a system. However, our construction in Section 4 can be modified to produce assemblies satisfying either the \approx or \equiv relation with the seed assemblies.

2.1.2 STAM* conventions used in this paper

Although the STAM* is a highly generalized model allowing for variety in tile shapes, glue lengths, etc., throughout this paper all constructions are restricted to the following conventions.

1. All tile types have one of two shapes (shown in Figure 1a):
 - a. A *cubic* tile is a tile whose shape is a $1 \times 1 \times 1$ cube.
 - b. A *flat* tile is a tile whose shape is a $1 \times 1 \times \epsilon$ rectangular prism, where $\epsilon < 1$ is a small constant.
 - c. We call a 1×1 face of a tile a *full* face, and a $1 \times \epsilon$ face is called a *thin* face.
2. Glue lengths are the following (and are shown in Figure 1b):
 - a. All rigid glues between cubic tiles, as well as between thin faces of flat tiles, are length 2ϵ .
 - b. All rigid glues between cubic and flat tiles are length 0. (Note that this could be implemented via the glue strand of one tile extending into the tile body of the other tile in order to bind, thus allowing the tile surfaces to be adjacent without spacing between the faces.)
 - c. All flexible glues are length $\frac{3}{2}\sqrt{2}\epsilon$.⁴

Given that rigidly bound cubic tiles cannot rotate relative to each other, for convenience we often refer to rigidly bound tiles as though they were on a fixed lattice. This is easily done by first choosing a rigidly bound cubic tile as our origin, then using the location \vec{l} , orientation matrix R , and rigid glue length g , put in one-to-one correspondence with each vector \vec{v} in \mathbb{Z}^3 , the vector $\vec{l} + gR\vec{v}$. Once we define an absolute coordinate system in this way, we refer to the directions in 3-dimensional space as North ($+y$), East ($+x$), South ($-y$), West ($-x$), Up ($+z$), and Down ($-z$), abbreviating them as N, E, S, W, U , and D , respectively.

3 A Genome Based Replicator

We now present our first construction in the STAM*, in which a “universal” set of tiles will cause a pre-formed seed assembly encoding a Hamiltonian path through a target structure, which we call the *genome*, to replicate infinitely many copies of itself as well as build infinitely many copies of the target structure at temperature 2. We consider 4 unique structures which are generated/utilized as part of the self-replication process: σ, μ, μ' , and π . The seed assembly, σ , is composed of a connected set of flat tiles considered to be the *genome*. Let π represent an assembly of the target shape encoded by σ . μ is an intermediate “messenger” structure directly copied from σ , which is modified into μ' to assemble π . We split T into subsets of tiles, $T = \{T_\sigma \cup T_\mu \cup T_\varphi \cup T_\pi\}$. T_σ are the tiles used to replicate the genome, T_μ are the tiles used to create the messenger structure, T_π are the cubic tiles which comprise the phenotype π , and T_φ are the set of tiles which combine to make fuel structures used in both the genome replication process and conversion of μ to μ' . We denote this universal self-assembling system as $\mathcal{R} = \{T, \sigma, 2\}$

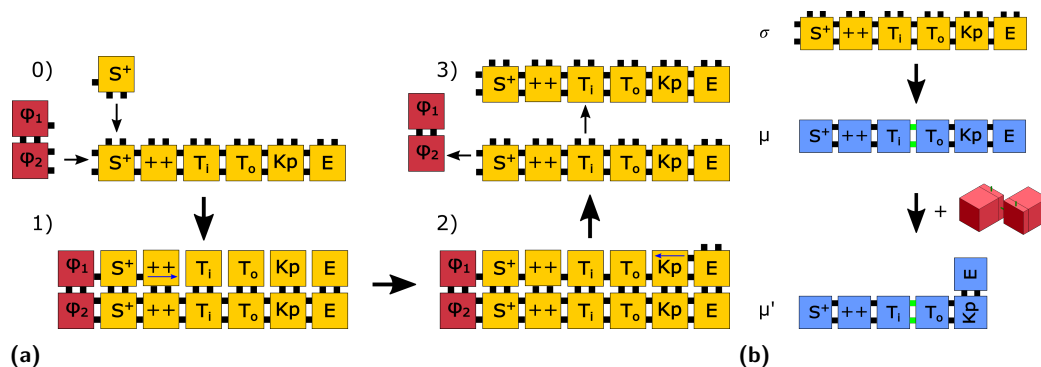
The tile types which make up this replicator are carefully designed to prevent spurious structures and enforce two key properties for the self-replication process. First, a genome is never consumed during replication, allowing for exponential growth in the number of

⁴ These glue lengths were chosen so that (1) rigidly bound cubic tiles could each have a flat tile bound to each of their sides if needed and (2) so that two flat tiles attached to diagonally adjacent rigid tiles could be attached via a flexible glue.

completed genome copies. Second, the replication process from messenger to phenotype strictly follows $\mu \rightarrow \mu' \rightarrow \pi$; each step in the assembly process occurs only after the prior structure is in its completed form. This prevents unexpected geometric hindrances which could block progression of any further step. Complete details of T are located in [2].

3.1 Replication of the genome

The minimal requirements to generate copies of σ in \mathcal{R} are the following: (1) for all individual tile types $s \in \sigma, s \in T_\sigma$, (2) the last tile is the end tile E , and (3) the first tile in σ is a start tile in the set (S^+, S^-) . However, for the shape-self replication of S one additional property must hold: (4) σ encodes a Hamiltonian path which ends on an exterior cubic tile. We define the genome to be “read” from left to right; given requirements (2) and (3), the leftmost tile in a genome is a start tile and the rightmost is an end tile. (4) can be guaranteed by scaling S up to S^2 and utilizing the algorithm in Section 4.3, selecting a cubic tile on the exterior as a start for the Hamiltonian path and then reversing the result. This requirement ensures the possibility of cubic tile diffusion into necessary locations at all stages of assembly.



■ **Figure 2** (a) In step 0 (before replication begins) both fuel and tiles from T_σ bind to σ . Step 1 indicates the fuel tile binding with the leftmost S^+ tile in σ' , propagating the binding of tiles from west to east indicated by blue arrow on the $++$ tile. Step 2 begins after all σ' glues are bound by strength-1, leading to the propagation of a second glue binding σ' from east to west. Additionally, glues on the north face of σ' tiles are activated and glues on the south face binding to σ are deactivated once they have a strength-2 connection to. Step 3 demonstrates the detachment – once the second glue binds to the fuel duple (φ_1, φ_2) signals propagate to detach from σ and σ' . (b) Process of translation: the information encoded in σ is copied to μ by a mapping of tiles via glue domains. Green glues on μ and μ' are flexible. One kink-ase (red) is used to convert μ to μ' .

Figure 9a (located in A.1) is a template for the tile set required for the replication of an arbitrary genome. The process of replicating a genome σ into a new copy σ' demonstrated in Figure 2a is carried out left to right, initiated by a fuel assembly which is jettisoned after all tiles in σ' are connected with strength 2. This allows for the genome σ to be copied without itself being used up or firing signals, leading to exponential growth. Full detail is available in the online version [2].

3.2 Translation of σ to μ

Translation is defined as the process by which the Hamiltonian path encoded in σ is built into a new messenger assembly μ . Since the signals to attach and detach μ from σ are fully contained in the tiles of T_μ , translation continues as long as T_μ tiles remain in the system. We note that the translation process can occur at the same time as σ is replicating. This causes no unwanted geometric hindrances as demonstrated in Figure 9b.

3.2.1 Placement of μ tiles

Messenger tiles from the set T_μ attach to σ as soon as complementary glues on the back flat face of σ are activated after the binding of the fuel duple φ to σ' . The process of building μ does not require a fuel structure to continue, as the messenger tiles have built-in signals to deactivate the glues on μ which attach μ to σ . This allows for a genome to replicate the messenger structure without itself being consumed in any manner. Once a flat tile in μ is bound to its eastern neighbor, signals are fired from the eastern glues to deactivate the glue connecting μ to σ . This leaves μ as its own separate assembly when every tile has attached to its neighbor(s). The example of translation shown in Figure 2b illustrates that the same information (i.e., sequence of tiles representing a Hamiltonian path) remains encoded in μ , but allows for new structural functionality that would otherwise not be possible by σ .

3.2.2 Modification of μ to μ'

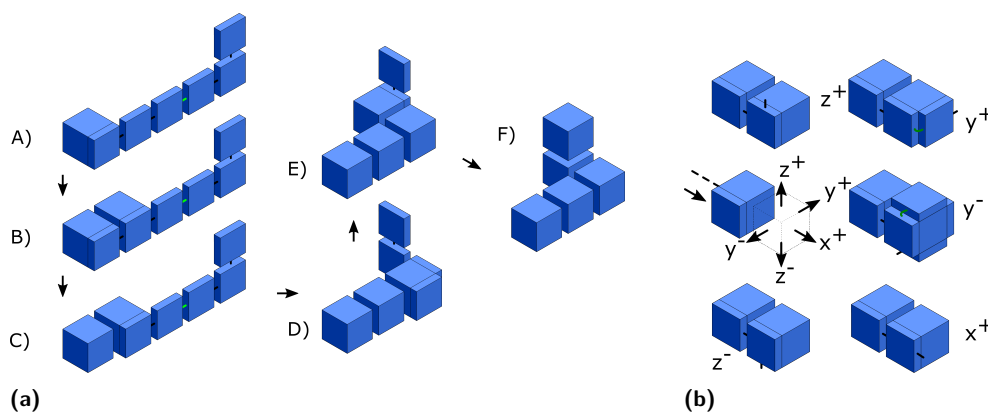
The current shape of μ is such that it could only replicate a trivial 2D structure; μ must be modified to follow a Hamiltonian path in 3 dimensions as made possible by a set of *turning* tiles. Additionally, in the current state of μ no cubic tiles can be placed as all the glues which are complementary to cubic tiles are currently in the **latent** state. Once a glue of type “p” is bound on the start tile, we then consider μ to have completed its modification into μ' . The “p” glue on turning tiles can only be bound once they have been turned, and as such the turning tiles present in μ' must be turned before assembly of π begins.

Turning tiles modify the shape of μ by adding “kinks” into the otherwise linear structure by the use of a fuel-like structure called a *kink-ase*. The kink-ase structure is generated from a set of 2 flat tiles and 2 cube tiles. The unique form of kink-ase allows for the orientation of two adjacent tiles to be modified without separating μ , shown by Figure 10 in A.2. The turning tiles are physically rotated such that the connection between a turning tile and its predecessor along the west thin edge of the turning tile is broken, and then reattached along either the up or down thin edge of the turning tile. Each turning tile requires the use of a single kink-ase, which turns into a junk assembly. Additional detail on this turning process is found in A.2.

3.2.3 Assembly of π

At the end of translation, the tiles of μ' have two strength-1 glues exposed which map to a specific cubic tile in T_π . The only tile in the the set T_π which starts with two complementary glues on is the start cubic tile. Once this cubic tile is bound to the start tile, a strength-1 glue is activated on the cube face adjacent to the next cubic tile in the Hamiltonian path, allowing for the cooperative binding to the superstructure of both μ' and the first tile of π .

After this process continues and a cubic tile is bound to its neighbor(s) with strength 2, the flat tile receives a signal to jettison itself from the remaining tiles of μ' by deactivating all active glues, becoming a junk tile. Due to the asynchronous nature of signals, there may be instances which the addition of cubic tiles of π are temporarily blocked. These will be eventually resolved, allowing assembly to continue. This process is repeated, adding cube by cube until the end tile in μ' is reached – see Figure 3a for a simple example. Once the end cube has been added to π , it has placed cubic tiles in all locations encoded by σ and μ' has been disassembled into junk tiles.



■ **Figure 3** (a) Building π from μ' (same as in Figure 2b). After the start cube binds to μ' in step A), the process of assembling π successively adds cubic tiles then detaches flat tiles from μ' . Step F) is phenotype π originally encoded by σ . (b) The inductive steps required in the creation of π which follows a Hamiltonian path given by a σ . The arrow going into the flat tile is the direction taken by the Hamiltonian path in the prior tile addition step. The five arrows indicate possible directions for the direction of the Hamiltonian path after the placement of the transparent cubic tile.

3.3 Analysis of \mathcal{R} and its correctness

► **Theorem 4.** *There exists an STAM* tile set T such that, given an arbitrary shape S , there exists STAM* system $\mathcal{R} = (T, \sigma, 2)$ and S^2 self-assembles in \mathcal{R} with waste size 4.*

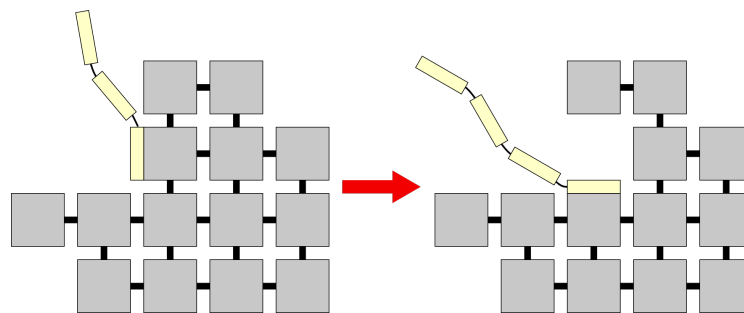
We provide the main idea of the correctness proof, further described in [2]. We demonstrate inductively that the construction process of an assembly π correctly generates a structure of shape S^2 , as shown in Figure 3b. The intuition is that at each step in the Hamiltonian path, there exists some combination of flat tiles which can correctly orient the placement of every cubic tile in the Hamiltonian path. This overall set of tiles are encoded in σ , demonstrating the ability of \mathcal{R} to replicate arbitrarily many copies of S^2 .

4 A Self-Replicator that Generates its own Genome

In this section we outline our main result: a system which, given an arbitrary input shape, is capable of disassembling an assembly of that shape block-by-block to build a genome which encodes it. We describe the process by which this disassembly occurs and then show how, from our genome, we can reconstruct the original assembly. Here we describe the construction at a high level. The technical details for this construction can be found in [2]. We prove the following theorem by implicitly defining the system \mathcal{R} , describing the process by which an input assembly is disassembled to form a “kinky” genome which is then used to make a copy of a linear genome (which replicates itself) and of the original input assembly.

► **Theorem 5.** *There exists a universal tile set T such that for every shape S , there exists an STAM* system $\mathcal{R} = (T, \sigma_{S^2}, 2)$ where σ_{S^2} has shape S^2 and \mathcal{R} is a self-replicator for σ_{S^2} with waste size 2.*

In this construction, there are two main components which here we call the *phenotype* and the *kinky genome*. The phenotype, which is the seed of our STAM* system, is a scale 2 version of our target shape made entirely out of cubic tiles. These tiles are connected to one another so that the assembly is τ -stable at temperature 2. We require the phenotype



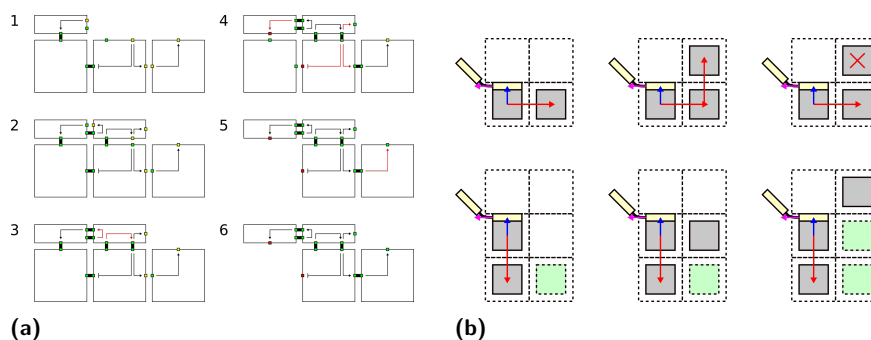
■ **Figure 4** During disassembly, the genome will be dangling off of a single structural tile in the phenotype. In each iteration, a new genome tile will attach and the old structural tile will detach along the Hamiltonian path embedded in the phenotype.

to be a 2-scaled version of S since the disassembly process requires a Hamiltonian path to pass through each of the tiles. This path describes the order in which the disassembly process will occur. Generally it is often either impossible or intractable to find a Hamiltonian path through an arbitrarily connected graph; however, using a 2-scaled shape we show that it's always possible efficiently. Additionally, the tiles in the phenotype contain glues and signals that will allow the various attachments and detachments to occur in the disassembly process. The genome is a sequence of flat tiles connected one to the next, whose glues encode the construction of the phenotype. In our system, the genome will be constructed as the phenotype is deconstructed and then will be duplicated or used to make copies of the original phenotype. Throughout this section, we refer to the cubic tiles that make up the phenotype as structural tiles and the flat tiles that make up the genome as genome tiles. Additionally, the tiles used in this construction are part of a finite tile set T , making T a universal tile set.

4.1 Disassembly

Given a phenotype P with encoded Hamiltonian path H , the disassembly process occurs iteratively by the detachment of at most 2 of tiles at a time. The process begins by the attachment of a special genome tile to the start of the Hamiltonian path. In each iteration, depending on the relative structure of the upcoming tiles in the Hamiltonian path, new genome tiles will attach to the existing genome encoding the local structure of H and, using signals from these newly attached genome tiles, a fixed number of structural tiles belonging to nearby points in the Hamiltonian path will detach from P . The order in which these detachments happen follow the path H and they will also cause all but the most recently attached genome tile to detach from the structure causing them to dangle, hanging on to the most recently attached genome tile as illustrated in Figure 5.

To show that the disassembly process happens correctly, we break down each iteration into one of 6 cases based on the tiles nearby the next in the Hamiltonian path. We show that these cases are complete and describe the process of disassembly for each one in [2]. Figure 5 illustrates the process and many of the important signals necessary for the most basic case. In it, a single genome tile attaches causing the previous one to dangle and the previous structural tile to detach. This new genome tile encodes this detachment so that reassembly can occur later and the process continues from there in the next iteration.



■ **Figure 5** (a) A side view of some of example glues and signals firing during disassembly. (b) A side view of the local structure of nearby tiles for all 6 different cases in the disassembly process.

4.2 Reassembly

Once the genome is built, we show that the original shape can be reconstructed. This occurs when a special structural tile attaches to the genome. This tile is identical to the last tile in the Hamiltonian path of the original phenotype and initiates the reassembly process. The online version [2] contains more details of the reassembly process, but essentially that reassembly occurs very similarly to disassembly in reverse – still using the same 6 cases as above and instead of having a new genome tile attach and the old structural tiles detach, the opposite occurs.

4.3 Generating a Hamiltonian Path

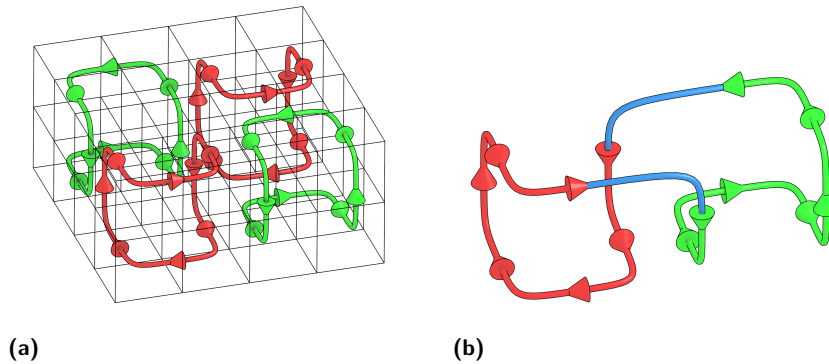
► **Lemma 6.** *Any scale factor 2 shape S^2 admits a Hamiltonian path and generating this path given a graph representing S^2 can be done in polynomial time.*

The algorithm for generating this Hamiltonian path is described in detail in [9] and was inspired by [50]. At a high level, the process proceeds as follows. First we generate a spanning tree through the shape S . We then scale the shape by a factor of two, assigning to each $2 \times 2 \times 2$ block of tiles one of two orientation graphs as illustrated in Figure 6. These orientation graphs make a path through the 8 tiles making up a tile block. For each edge in the spanning tree, we connect the corresponding orientation graphs, combining them to form a single orientation graph. Doing this for all edges will leave us with a Hamiltonian path through S^2 . In fact, we actually define a Hamiltonian circuit which guarantees that during disassembly, the remaining phenotype will always remain connected.

The resulting Hamiltonian path, which we will call H , passes through each tile in the 2-scaled version of our shape and only take a polynomial amount of time to compute since spanning trees can be found efficiently and only contain a polynomial number of edges. Additionally, it should be noted that once we generate a Hamiltonian path, an algorithm can easily iterate over the path simulating which tiles would still be attached during each stage of the disassembly process. This means such an algorithm can also easily determine the glues and signals necessary for each tile in the path by considering the appropriate iteration case.

5 Shape Building via Hierarchical Assembly

In this section we present details of a shape building construction which makes use of hierarchical self-assembly. The main goals of this construction are to (1) provide more compact genomes than the previous constructions, and (2) to more closely mimic the fact



■ **Figure 6** (a) Each $2 \times 2 \times 2$ block of space is assigned an orientation graph which will be used to help generate the Hamiltonian path through our shape. Adjacent blocks are assigned opposite orientation graphs, the edges of which will help guide the Hamiltonian path around the shape. (b) Orientation graphs of adjacent blocks are joined to form a continuous path.

that in the replication of biological systems, individual proteins are independently constructed and then they combine with other proteins to form cellular structures. First, we define a class of shapes for which our base construction works, then we formally state our result.

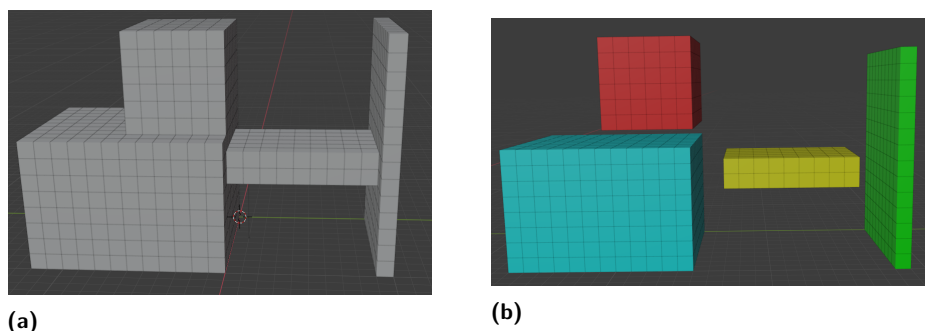
Let a *block-diffusable* shape be a shape S which can be divided into a set of rectangular prism shaped blocks⁵ whose union is S (following the algorithm in the online version [2]) such that a connectivity tree T can be constructed through those blocks and if any prism is removed but T remains connected, that prism can be placed arbitrarily far away and move in an obstacle-free path back into its location in S .

► **Theorem 7.** *There exists a tile set U such that, for any block-diffusable shape S , there exists a scale factor $c \geq 1$ and STAM* system $\mathcal{T}_S = (U, \sigma_{S^c}, 2)$ such that S^c self-assembles in \mathcal{T}_S with waste size 1. Furthermore, $|\sigma_S|$ is approximately $O(|S|^{1/3})$.*

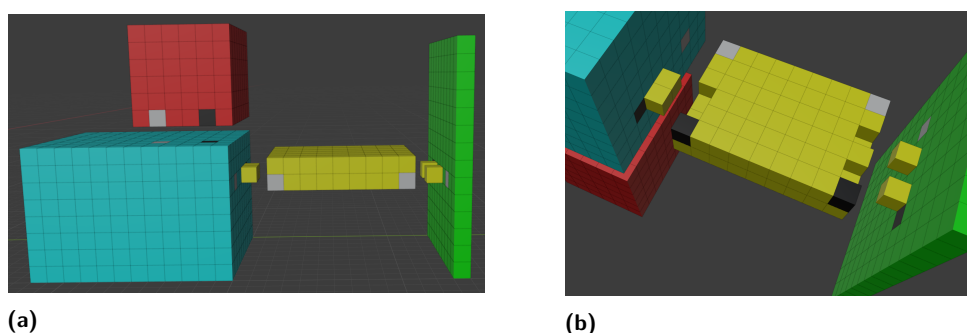
To prove Theorem 7, we present the algorithm which computes the encoding of S into seed assembly σ_S as well as the value of the scale factor c (which may simply be 1), and then explain the tiles that make up U so that \mathcal{T}_S will produce components that hierarchically self-assemble to form a terminal assembly of shape S . At a high level, in this construction the seed assembly is the **genome**, which is a compressed linear encoding of the target shape that is logically divided into separate regions (called **genes**), and each **gene** independently initiates the growth a (potentially large) portion of the target shape called a **block**. Once sufficiently grown, each **block** detaches from the **genome**, completes its growth, and freely diffuses until binding with the other **blocks**, along carefully defined binding surfaces called **interfaces**, to form the target shape.

It is important to note that there are many potential refinements to the construction we present which could serve to further optimize various aspects such as **genome** length, scale factor, tile complexity, etc., especially for specific categories of target shapes. For ease of understanding, we will present a relatively simple version of the construction, and in several places we will point out where such optimizations and/or tradeoffs could be made. Throughout this section, S is the target shape of our system. For some shapes, it may be the case that a scale factor is required (and the details of how that is computed are provided

⁵ A rectangular prism is simply a 3D shape that has 6 faces, all of which are rectangles.



■ **Figure 7** (a) An example 3D shape S . (b) S split into 4 **blocks**, each of which can be grown from its own **genome**. Note that the surfaces which will be adjacent when the **blocks** combine will also be assigned **interfaces** to ensure correct assembly of S .



■ **Figure 8** (a) The **blocks** for the example shape S from Figure 7 with example **interfaces** included. (b) View from underneath showing more of the **interfaces** between **blocks**. Note that the actual **interfaces** created by the algorithm would be shorter, but to make the example more interesting their sizes have been increased.

in [2]). We will first describe how the shape S can be broken into a set of constituent **blocks**, then how the **interfaces** between **blocks** are designed, then how individual **blocks** self-assemble before being freed to autonomously combine into an assembly of shape S .

5.1 Decomposition into blocks

Since S is a shape in \mathbb{Z}^3 , it is possible to split it into a set of rectangular prisms whose union is S . We do so using a simple greedy algorithm which seeks to maximize the size of each rectangular prism, which we call a **block**, and we call the full set of **blocks** B .

After the application of a greedy algorithm to compute an initial set B , we refine it by splitting some of the **blocks** as needed to form a binding graph in the form of a tree T such that every **block** is connected to at least one adjacent **block**, but also so that each **block** has no more than one connected neighbor in each direction in T . This results in the final set of **blocks** that combine to define S , can join along the edges defined by T , and each **block** has at most 6 neighbors to which it combines. (Figure 7 shows a simple example.)

5.2 Interface design

The **blocks** self-assemble individually, then separate from the **genome** to freely diffuse until they combine together via **interfaces** along the surfaces between which there were edges in the binding tree T . Each **interface** is assigned a unique length and number. The two

blocks that join along a given **interface** are assigned complementary patterns of “bumps” and “dents” and a pair of complementary glues on either side of those patterns (to provide the necessary binding strength between the blocks). The number assigned to each **interface** is represented in binary and the **block** on one side of an **interface** has a protruding tile “bump” in the location of each 1 bit but not in locations of 0 bits, and for the **block** on the other side of the **interface** 1 bits have single tile “dents” where a tile is missing. The length of each **interface** dictates which other **interfaces** have glues at the correct spacing to allow binding, and the binary pattern of “bumps” and “dents” guarantees that only the single, correct complementary half can combine with it.

Depending on the shape S and how it is split into **blocks**, it is possible that there are too many **interfaces** of a given length ($> 2^{(n-2)/2}$ for an **interface** of length n) to be able to assign a unique number to each. Our algorithm will attempt to assign a unique length and number to an **interface** for all lengths 2 to $n/2$ (2 being the minimum since there must be room for the two glues), but since n is the full length of the surface between a pair of **blocks** and each bit of the assigned number is represented by a pair of bits, a greater length can't be encoded in the tiles along it. Therefore, if there are too many **interfaces** for a unique assignment, the shape S is scaled upward. This is repeated until there can be unique assignments. (Note that there are many ways in which the algorithm could be optimized to reduce the number of shapes for which scaling is necessary, and/or the amount of scaling, especially for particular categories of shapes.) More technical details can be found in [2], and an example of a few **interfaces** can be seen in Figure 8.

5.3 Block growth

The growth of each **block** is initiated by a portion of the linear **genome** called a **gene**, which is merely a line of tiles with glues exposed in one direction that encode all of the information required for the **block** to self-assemble to the correct dimensions and with the necessary **interfaces**. The techniques used to encode the information and allow the **blocks** to grow are very standard tile assembly techniques involving binary counters, zig-zag growth patterns, and rotation of patterns of information. The information to seed the counters and encode the **interfaces** is encoded in the outward facing glues of the **gene** and can be done so with the universal tile set U since only a constant amount of information needs to be encoded in any particular **gene** glue, due to the design of **blocks** and the fact that each has at most a single **interface** on each side which is no longer than that side. Signals are used for detecting completed growth of **blocks**, controlling growth of **interfaces** so “bump” **interfaces** can't complete before all “bumps” are in place, and “dent” **interfaces** can grow beyond “dent” locations and then those tiles can fall out, and also so **blocks** can dissociate from **genes**.

5.4 Overview of the hierarchical construction

Once a **block** is freely diffusing and complete, it can combine along its **interfaces** with the **blocks** that have complementary **interfaces** since, due to the fact that S is a block-diffusable shape, free **blocks** can always diffuse into the proper locations to form the complete shape. We've described a tile set U that can be used to (1) form the linear seed assembly σ_S , and (2) to self-assemble the **blocks** which correctly combine to form the target assembly. The STAM* system $\mathcal{T}_S = (U, \sigma_S, 2)$ will produce an infinite number of copies of terminal assemblies of shape S (properly scaled if necessary). The only fuel (a.k.a. consumed, junk assemblies) will be singleton Dent tiles that attached during **block** growth then detached. Note that this construction can be combined with the previous constructions as well, to create a version of a shape self-replicator. Full technical details of the construction, as well as a discussion of possible enhancements, can be found in [2].

References

- 1 Zachary Abel, Nadia Benbernou, Mirela Damian, Erik Demaine, Martin Demaine, Robin Flatland, Scott Kominers, and Robert Schweller. Shape replication through self-assembly and RNase enzymes. In *SODA 2010: Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, Austin, Texas, 2010. Society for Industrial and Applied Mathematics.
- 2 Andrew Alseth, Daniel Hader, and Matthew J. Patitz. Self-replication via tile self-assembly (extended abstract). Technical Report 2105.02914, Computing Research Repository, 2021. [arXiv:2105.02914](https://arxiv.org/abs/2105.02914).
- 3 Ebbe S. Andersen, Mingdong Dong, Morten M. Nielsen, Kasper Jahn, Ramesh Subramani, Wael Mamdouh, Monika M. Golas, Bjoern Sander, Holger Stark, Cristiano L. P. Oliveira, Jan S. Pedersen, Victoria Birkedal, Flemming Besenbacher, Kurt V. Gothelf, and Jorgen Kjems. Self-assembly of a nanoscale dna box with a controllable lid. *Nature*, 459(7243):73–76, May 2009. doi:10.1038/nature07971.
- 4 Robert D. Barish, Rebecca Schulman, Paul W. K. Rothmund, and Erik Winfree. An information-bearing seed for nucleating algorithmic self-assembly. *Proceedings of the National Academy of Sciences*, 106(15):6054–6059, April 2009. doi:10.1073/pnas.0808736106.
- 5 Florent Becker, Ivan Rapaport, and Eric Rémila. Self-assembling classes of shapes with a minimum number of tiles, and in optimal time. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 45–56, 2006. doi:10.1007/11944836_7.
- 6 Florent Becker, Eric Rémila, and Nicolas Schabanel. Time optimal self-assembly for 2d and 3d shapes: The case of squares and cubes. In Ashish Goel, Friedrich C. Simmel, and Petr Sosík, editors, *DNA*, volume 5347 of *Lecture Notes in Computer Science*, pages 144–155. Springer, 2008. doi:10.1007/978-3-642-03076-5_12.
- 7 Hieu Bui, Shalin Shah, Reem Mokhtar, Tianqi Song, Sudhanshu Garg, and John Reif. Localized dna hybridization chain reactions on dna origami. *ACS nano*, 12(2):1146–1155, 2018.
- 8 Qi Cheng, Gagan Aggarwal, Michael H. Goldwasser, Ming-Yang Kao, Robert T. Schweller, and Pablo Moisset de Espanés. Complexities for generalized models of self-assembly. *SIAM Journal on Computing*, 34:1493–1515, 2005.
- 9 Kenneth C Cheung, Erik D Demaine, Jonathan R Bachrach, and Saul Griffith. Programmable assembly with universally foldable strings (moteins). *IEEE Transactions on Robotics*, 27(4):718–729, 2011.
- 10 Matthew Cook, Yunhui Fu, and Robert T. Schweller. Temperature 1 self-assembly: Deterministic assembly in 3D and probabilistic assembly in 2D. In *SODA 2011: Proceedings of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2011.
- 11 E. D. Demaine, M. L. Demaine, S. P. Fekete, M. J. Patitz, R. T. Schweller, A. Winslow, and D. Woods. One tile to rule them all: Simulating any tile assembly system with a single universal tile. In *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP 2014)*, IT University of Copenhagen, Denmark, July 8-11, 2014, volume 8572 of *LNCS*, pages 368–379, 2014.
- 12 Erik D. Demaine, Martin L. Demaine, Sándor P. Fekete, Mashhood Ishaque, Eynat Rafalin, Robert T. Schweller, and Diane L. Souvaine. Staged self-assembly: nanomanufacture of arbitrary shapes with $O(1)$ glues. *Natural Computing*, 7(3):347–370, 2008. doi:10.1007/s11047-008-9073-0.
- 13 Erik D. Demaine, Matthew J. Patitz, Trent A. Rogers, Robert T. Schweller, Scott M. Summers, and Damien Woods. The two-handed assembly model is not intrinsically universal. In *40th International Colloquium on Automata, Languages and Programming, ICALP 2013, Riga, Latvia, July 8-12, 2013*, Lecture Notes in Computer Science. Springer, 2013.
- 14 Erik D. Demaine, Matthew J. Patitz, Trent A. Rogers, Robert T. Schweller, Scott M. Summers, and Damien Woods. The two-handed tile assembly model is not intrinsically universal. *Algorithmica*, 74(2):812–850, February 2016. doi:10.1007/s00453-015-9976-y.

- 15 David Doty. Randomized self-assembly for exact shapes. In *Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 85–94. IEEE, 2009.
- 16 David Doty, Lila Kari, and Benoît Masson. Negative interactions in irreversible self-assembly. *Algorithmica*, 66(1):153–172, 2013. doi:10.1007/s00453-012-9631-9.
- 17 David Doty, Jack H. Lutz, Matthew J. Patitz, Robert T. Schweller, Scott M. Summers, and Damien Woods. The tile assembly model is intrinsically universal. In *Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012*, pages 302–310, 2012.
- 18 Jérôme Durand-Lose, Jacob Hendricks, Matthew J. Patitz, Ian Perkins, and Michael Sharp. Self-assembly of 3-D structures using 2-D folding tiles. In *Proceedings of the 24th International Conference on DNA Computing and Molecular Programming (DNA 24)*, Shandong Normal University, Jinan, China October 8-12, pages 105–121, 2018.
- 19 Sándor P. Fekete, Jacob Hendricks, Matthew J. Patitz, Trent A. Rogers, and Robert T. Schweller. Universal computation with arbitrary polyomino tiles in non-cooperative self-assembly. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2015)*, San Diego, CA, USA January 4-6, 2015, pages 148–167, 2015. doi:10.1137/1.9781611973730.12.
- 20 Tyler Fochtman, Jacob Hendricks, Jennifer E. Padilla, Matthew J. Patitz, and Trent A. Rogers. Signal transmission across tile assemblies: 3d static tiles simulate active self-assembly by 2d signal-passing tiles. *Natural Computing*, 14(2):251–264, 2015.
- 21 Bin Fu, Matthew J. Patitz, Robert T. Schweller, and Robert Sheline. Self-assembly with geometric tiles. In *Proceedings of the 39th International Colloquium on Automata, Languages and Programming, ICALP*, pages 714–725, 2012.
- 22 David Furcy, Samuel Micka, and Scott M. Summers. Optimal program-size complexity for self-assembly at temperature 1 in 3D. In *DNA Computing and Molecular Programming - 21st International Conference, DNA 21, Boston and Cambridge, MA, USA, August 17-21, 2015. Proceedings*, pages 71–86, 2015. doi:10.1007/978-3-319-21999-8_5.
- 23 Oscar Gilbert, Jacob Hendricks, Matthew J. Patitz, and Trent A. Rogers. Computing in continuous space with self-assembling polygonal tiles. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2016)*, Arlington, VA, USA January 10-12, 2016, pages 937–956, 2016.
- 24 Daniel Hader, Aaron Koch, Matthew J. Patitz, and Michael Sharp. The impacts of dimensionality, diffusion, and directedness on intrinsic universality in the abstract tile assembly model. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 2607–2624. SIAM, 2020.
- 25 Daniel Hader and Matthew J. Patitz. Geometric tiles and powers and limitations of geometric hindrance in self-assembly. In *Proceedings of the 18th Annual Conference on Unconventional Computation and Natural Computation (UCNC 2019)*, Tokyo, Japan June 3–7, 2019, pages 191–204, 2019.
- 26 Jacob Hendricks, Meagan Olsen, Matthew J. Patitz, Trent A. Rogers, and Hadley Thomas. Hierarchical self-assembly of fractals with signal-passing tiles. Submit to *Natural Computing*.
- 27 Jacob Hendricks, Matthew J. Patitz, and Trent A. Rogers. Replication of arbitrary hole-free shapes via self-assembly with signal-passing tiles. In Cristian S. Calude and Michael J. Dinneen, editors, *Unconventional Computation and Natural Computation - 14th International Conference, UCNC 2015, Auckland, New Zealand, August 30 - September 3, 2015, Proceedings*, volume 9252 of *Lecture Notes in Computer Science*, pages 202–214. Springer, 2015. doi:10.1007/978-3-319-21819-9_15.
- 28 Jacob Hendricks, Matthew J. Patitz, and Trent A. Rogers. Reflections on tiles (in self-assembly). *Natural Computing*, 16(2):295–316, 2017. doi:10.1007/s11047-017-9617-2.

- 29 Jacob Hendricks, Matthew J. Patitz, and Trent A. Rogers. The simulation powers and limitations of higher temperature hierarchical self-assembly systems. *Fundam. Inform.*, 155(1-2):131–162, 2017. doi:10.3233/FI-2017-1579.
- 30 Nataša Jonoska and Daria Karpenko. Active tile self-assembly, part 1: Universality at temperature 1. *International Journal of Foundations of Computer Science*, 25(02):141–163, 2014. doi:10.1142/S0129054114500087.
- 31 Nataša Jonoska and Daria Karpenko. Active tile self-assembly, part 2: Self-similar structures and structural recursion. *International Journal of Foundations of Computer Science*, 25(02):165–194, 2014. doi:10.1142/S0129054114500099.
- 32 Nataša Jonoska and Gregory L. McColm. A computational model for self-assembling flexible tiles. In *Proceedings of the 4th international conference on Unconventional Computation, UC’05*, pages 142–156, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/11560319_14.
- 33 Nataša Jonoska and Gregory L. McColm. Complexity classes for self-assembling flexible tiles. *Theor. Comput. Sci.*, 410(4-5):332–346, 2009. doi:10.1016/j.tcs.2008.09.054.
- 34 Nataša Jonoska and Gregory L. McColm. Flexible versus rigid tile assembly. In Cristian S. Calude, Michael J. Dinneen, Gheorghe Păun, Grzegorz Rozenberg, and Susan Stepney, editors, *Unconventional Computation*, volume 4135 of *Lecture Notes in Computer Science*, pages 139–151. Springer Berlin Heidelberg, 2006. doi:10.1007/11839132_12.
- 35 Lila Kari, Shinnosuke Seki, and Zhi Xu. Triangular and hexagonal tile self-assembly systems. In *Proceedings of the 2012 international conference on Theoretical Computer Science: computation, physics and beyond, WTCS’12*, pages 357–375, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-27654-5_28.
- 36 Alexandra Keenan, Robert Schweller, and Xingsi Zhong. Exponential replication of patterns in the signal tile assembly model. *Natural Computing*, 14(2):265–278, 2014.
- 37 Alexandra Keenan, Robert T. Schweller, and Xingsi Zhong. Exponential replication of patterns in the signal tile assembly model. In David Soloveichik and Bernard Yurke, editors, *DNA*, volume 8141 of *Lecture Notes in Computer Science*, pages 118–132. Springer, 2013. doi:10.1007/978-3-319-01928-4_9.
- 38 James I. Lathrop, Jack H. Lutz, Matthew J. Patitz, and Scott M. Summers. Computability and complexity in self-assembly. *Theory Comput. Syst.*, 48(3):617–647, 2011. doi:10.1007/s00224-010-9252-0.
- 39 Wenyan Liu, Hong Zhong, Risheng Wang, and Nadrian C. Seeman. Crystalline two-dimensional dna-origami arrays. *Angewandte Chemie International Edition*, 50(1):264–267, 2011. doi:10.1002/anie.201005911.
- 40 Jennifer E. Padilla, Matthew J. Patitz, Robert T. Schweller, Nadrian C. Seeman, Scott M. Summers, and Xingsi Zhong. Asynchronous signal passing for tile self-assembly: Fuel efficient computation and efficient assembly of shapes. *International Journal of Foundations of Computer Science*, 25(4):459–488, 2014.
- 41 Jennifer E. Padilla, Ruojie Sha, Martin Kristiansen, Junghuei Chen, Natasha Jonoska, and Nadrian C. Seeman. A signal-passing DNA-strand-exchange mechanism for active self-assembly of DNA nanostructures. *Angewandte Chemie International Edition*, 54(20):5939–5942, March 2015. doi:10.1002/anie.201500252.
- 42 Matthew J. Patitz, Trent A. Rogers, Robert T. Schweller, Scott M. Summers, and Andrew Winslow. Resiliency to multiple nucleation in temperature-1 self-assembly. In *Proceedings of the 22nd International Conference on DNA Computing and Molecular Programming (DNA 22)*, Ludwig-Maximilians-Universität, Munich, Germany September 4-8, 2016, pages 98–113, 2016.
- 43 Lulu Qian and Erik Winfree. Scaling up digital circuit computation with dna strand displacement cascades. *Science*, 332(6034):1196–1201, 2011.
- 44 P. W. K. Rothmund. Design of dna origami. In *ICCAD ’05: Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pages 471–478, Washington, DC, USA, 2005. IEEE Computer Society.

- 45 Paul W. K. Rothemund and Erik Winfree. The program-size complexity of self-assembled squares (extended abstract). In *STOC '00: Proceedings of the thirty-second annual ACM Symposium on Theory of Computing*, pages 459–468, Portland, Oregon, United States, 2000. ACM. doi:10.1145/335305.335358.
- 46 Rebecca Schulman, Bernard Yurke, and Erik Winfree. Robust self-replication of combinatorial information via crystal growth and scission. *Proc Natl Acad Sci USA*, 109(17):6405–10, 2012. URL: <http://www.biomedsearch.com/nih/Robust-self-replication-combinatorial-information/22493232.html>.
- 47 Friedrich C. Simmel, Bernard Yurke, and Hari R. Singh. Principles and applications of nucleic acid strand displacement reactions. *Chemical Reviews*, 119(10):6326–6369, 2019.
- 48 David Soloveichik and Erik Winfree. Complexity of compact proofreading for self-assembled patterns. In *The eleventh International Meeting on DNA Computing*, 2005.
- 49 David Soloveichik and Erik Winfree. Complexity of self-assembled shapes. *SIAM Journal on Computing*, 36(6):1544–1569, 2007. doi:10.1137/S0097539704446712.
- 50 Scott M. Summers. Reducing tile complexity for the self-assembly of scaled shapes through temperature programming. *Algorithmica*, 63(1-2):117–136, June 2012. doi:10.1007/s00453-011-9522-5.
- 51 Boya Wang, Chris Thachuk, Andrew D. Ellington, Erik Winfree, and David Soloveichik. Effective design principles for leakless strand displacement systems. *Proceedings of the National Academy of Sciences*, 115(52):E12182–E12191, 2018.
- 52 Bryan Wei, Mingjie Dai, and Peng Yin. Complex shapes self-assembled from single-stranded dna tiles. *Nature*, 485(7400):623–626, 2012.
- 53 David Yu Zhang and Rizal F. Hariadi. Integrating dna strand-displacement circuitry with dna tile self-assembly. *Nature Communications*, 4(6):Art. No. 1965, June 2013.
- 54 David Yu Zhang and Georg Seelig. Dynamic dna nanotechnology using strand-displacement reactions. *Nature chemistry*, 3(2):103–113, 2011.

A Genome Based Replicator

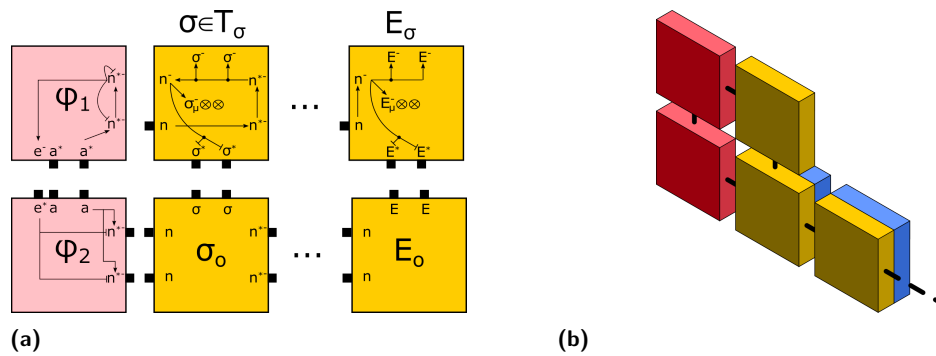
A.1 Replication and translation details

The replication process of σ begins with the attachment of tiles from the set T_σ to σ due to the two strength-1 glues on the north face of individual tiles comprising σ . We denote the incomplete copy of σ as σ' . Asynchronously, a fuel tile assembly φ comprised of two subtiles $\varphi_1, \varphi_2 \in T_\varphi$ binds to the leftmost tile of σ . Upon the binding of a start tile to the north thin face of the start tile of σ' , the signal provided by φ begins a chain reaction starting with the binding to the the active “n” glue on the west thin face of the newly attached tile and the signal propagates through the chain of connected σ' tiles. Once the end tile E_σ is bound to the remainder of σ' by the active “n” glue, it returns a signal through its newly activated west glue to fully connect it to the prior tile and then detach from the genome to the south. This signal cascades back through the remaining tiles of σ' until reaching φ , at which point φ deactivates its glues. allowing the newly replicated copy of σ to separate and begin the process of replicating itself and translating copies of μ .

A.2 Turning Tile and Kink-ase

This section describes in detail how μ is converted to μ' utilizing the kink-ase structure, and an example is shown in Figure 10.

- A) Kink-ase attaches to a turning tile and the predecessor which will be re-oriented in μ . Simultaneously, glues are activated on the kink-ase cube structure attached to the turning tile to bind the turning tile face and to the kink-ase cube structure attached to

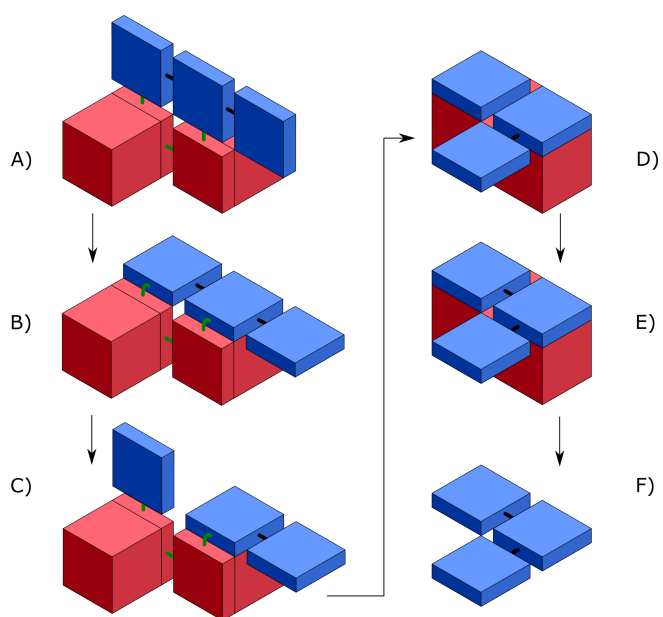


■ **Figure 9** (a) Initial genome replicator tiles. Note that $\otimes\otimes$ represents two strength 1 glues which are on the full face of the seed tiles opposite from the reader (b) Illustration of an arbitrary translation process occurring at the same time as genome replication. Red tiles are representative of φ , gold tiles are representative of σ and σ' , and blue tiles are representative of μ .

the predecessor tile to enable the folding of the cube structure in step D). Note – glues connecting tiles in μ may be either rigid or flexible depending upon the Hamiltonian path generated for π . This does not effect any intermediate steps presented.

- B) The turning tile’s rear face binds to the kink-ase due to random movement allowed by the flexible glues which attach the kink-ase to the turning and predecessor tiles, i.e. the flexible bond allows the tile to rotate and randomly assume various relative positions. When it enters the correct configuration, the glues bind to “lock it in”.
- C) Upon connection of the turning tile face to the kink-ase cube, a signal deactivates the rigid glue attaching the predecessor tile to the turning tile. A signal activates glues on the exposed face of the kink-ase tile attached to cube and turning tile structure. The flexible connection between the predecessor tile and kink-ase ensures μ does not split into two pieces.
- D) The kink-ase cube and kink-ase tile with activated glue bind on faces when they rotate into the correct configuration, bringing the turning tile into correct alignment with the predecessor tile. The kink-ase cube face adjacent to the predecessor tile activates its glue, allowing for binding with the face of the two. The flexible glue allows for random movement for the complementary glues to attach and bind. Concurrently, the flexible glue on the turning tile is deactivated and a rigid glue of similar type to the turning tile glue deactivated in step C) is activated.
- E) A rigid glue between the turning tile and predecessor tile binds, leading to re-connection between both prior detached portions of μ . Activation of the final glue leads to the turning tile signaling to kink-ase to detach from μ .
- F) This structure represents μ after one turning tile has been resolved. A completion signal is passed through glues attaching the turning tile and predecessor tile. This process continues for all turning tiles serially, working backwards from the termination tile. This is to prevent any interference between structures incurred by multiple adjacent turning tiles.

3:22 Self-Replication



■ **Figure 10** Conversion of one turning tile. Blue tiles indicate μ , whereas the red indicate the kink-ase.

Improved Lower and Upper Bounds on the Tile Complexity of Uniquely Self-Assembling a Thin Rectangle Non-Cooperatively in 3D

David Furcy ✉

Computer Science Department, University of Wisconsin Oshkosh, WI, USA

Scott M. Summers ✉

Computer Science Department, University of Wisconsin Oshkosh, WI, USA

Logan Withers ✉

Computer Science Department, University of Wisconsin Oshkosh, WI, USA

Abstract

We investigate a fundamental question regarding a benchmark class of shapes in one of the simplest, yet most widely utilized abstract models of algorithmic tile self-assembly. More specifically, we study the directed tile complexity of a $k \times N$ thin rectangle in Winfree’s ubiquitous abstract Tile Assembly Model, assuming that cooperative binding cannot be enforced (temperature-1 self-assembly) and that tiles are allowed to be placed at most one step into the third dimension (just-barely 3D). While the directed tile complexities of a square and a scaled-up version of any algorithmically specified shape at temperature 1 in just-barely 3D are both asymptotically the same as they are (respectively) at temperature 2 in 2D, the (nearly tight) bounds on the directed tile complexity of a thin rectangle at temperature 2 in 2D are not currently known to hold at temperature 1 in just-barely 3D. Motivated by this discrepancy, we establish new lower and upper bounds on the directed tile complexity of a thin rectangle at temperature 1 in just-barely 3D. The proof of our upper bound is based on the construction of a novel, just-barely 3D temperature-1 self-assembling counter. Each value of the counter is comprised of $k - 2$ digits, represented in a geometrically staggered fashion within k rows. This nearly optimal digit density, along with the base of the counter, which is proportional to $N^{\frac{1}{k-1}}$, results in an upper bound of $O\left(N^{\frac{1}{k-1}} + \log N\right)$, and is an asymptotic improvement over the previous state-of-the-art upper bound. On our way to proving our lower bound, we develop a new, more powerful type of specialized Window Movie Lemma that lets us bound the number of “sufficiently similar” ways to assign glues to a set (rather than a sequence) of fixed locations. Consequently, our lower bound, $\Omega\left(N^{\frac{1}{k}}\right)$, is also an asymptotic improvement over the previous state-of-the-art lower bound.

2012 ACM Subject Classification Theory of computation → Models of computation

Keywords and phrases Self-assembly, algorithmic self-assembly, tile self-assembly

Digital Object Identifier 10.4230/LIPIcs.DNA.27.4

1 Introduction

A key objective in algorithmic self-assembly is to characterize the extent to which an algorithm can be converted to an efficient self-assembling system comprised of discrete, distributed and disorganized units that, through random encounters with, and locally-defined reactions to each other, coalesce into a terminal assembly having a desirable form or function. In this paper, we study a fundamental theoretical question regarding a benchmark class of shapes in one of the simplest yet most popular abstract models of algorithmic self-assembly.

Ubiquitous throughout the theory of tile self-assembly, Erik Winfree’s abstract Tile Assembly Model (aTAM) [26] is a discrete mathematical model of DNA tile self-assembly [23] that augments classical Wang tiling [25] with a mechanism for automatic growth. In the



© David Furcy, Scott M. Summers, and Logan Withers;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on DNA Computing and Molecular Programming (DNA 27).

Editors: Matthew R. Lakin and Petr Šulc; Article No. 4; pp. 4:1–4:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

aTAM, a DNA tile is represented by a unit square (or cube) tile type that may neither rotate, reflect, nor fold. Each side of a tile type is decorated with a glue consisting of both a non-negative integer strength and a string label, the symbols of which are drawn from some fixed alphabet. A tile set is a finite set of tile types, from which infinitely many tiles of each type may be instantiated. If one tile is positioned at an unoccupied location Manhattan distance 1 away from another tile and their opposing glues are equal, then the two tiles bind with the strength of the adjacent glues. A special seed tile type is designated and a seed tile, which defines the seed-containing assembly, is placed at some fixed location. During the process of self-assembly, a sequence of tiles bind to and never detach from the seed-containing assembly, provided that each one, in a non-overlapping fashion, binds to one or more tiles in the seed-containing assembly with total strength at least a certain positive integer value called the temperature. If the temperature is greater than or equal to 2, then it is possible to enforce cooperative binding, where a tile may be prevented from binding at a certain location until at least two adjacent locations become occupied by tiles. Otherwise, only non-cooperative binding is allowed (temperature-1 self-assembly). A fundamental theoretical question in tile self-assembly is determining the effect of the value of the temperature on the computational and geometric expressiveness of tile self-assembly.

To that end, temperature-1 self-assembly has been shown to hinder the efficient self-assembly of shapes when tile assemblies are required to be fully connected [22] or contain no glue mismatches [16]. Temperature-1 self-assembly is also neither intrinsically universal [17,19], nor capable of bounded Turing computation [19]. Recently and quite remarkably, Meunier, Regnault and Woods [18] established a general pumping lemma for temperature-1 self-assembly, nearly proving a conjecture by Doty, Patitz and Summers [6] on the computational weakness of temperature-1 self-assembly.

Interestingly, temperature-1 self-assembly does not limit the computational or geometric expressiveness of generalizations of the aTAM [4, 5, 7, 8, 12]. This is also true even when the generalization only adds a small number of additional features to the model, like a single negative glue [20], duple tiles [13], or another plane in which (cubic) 3D tiles are allowed to be placed [3, 9–11]. The latter variant is colloquially known as “just-barely” 3D self-assembly. In this paper, we study the limitations of temperature-1 self-assembly for unique shape-building in the just-barely 3D aTAM. We are specifically interested in studying the *directed tile complexity* of a given target shape, or the size of the smallest tile set that, regardless of the order in which tiles bind to the seed-containing assembly, always self-assembles into a unique terminal assembly of tiles that are placed on and only on points of a given target shape.

Although temperature-1 self-assembly cannot enforce cooperative binding, there is a striking resemblance of its computational and geometric expressiveness in just-barely 3D, to that of temperature-2 self-assembly in 2D, with respect to the directed tile complexity of two benchmark shapes: a square and a scaled-up version of any algorithmically specified shape. Adleman, Cheng, Goel and Huang [1] proved, using optimal base conversion, that the directed tile complexity of an $N \times N$ square at temperature 2 in 2D is $O\left(\frac{\log N}{\log \log N}\right)$, matching a corresponding lower bound for all Kolmogorov-random N and all positive temperature values, set by Rothmund and Winfree [22]. The lower bound also holds in just-barely 3D. An $O\left(\frac{\log N}{\log \log N}\right)$ upper bound for the directed tile complexity of an $N \times N$ square at temperature 1 in just-barely 3D was established by Furcy, Micka and Summers [9] via a just-barely 3D, optimal encoding construction at temperature 1. Just-barely 3D, optimal encoding at temperature 1 was inspired by, achieves the same result as, but is drastically different from the 2D optimal encoding at temperature 2 developed by Soloveichik and Winfree [24], who proved that the directed tile complexity of a scaled-up version of any

algorithmically specified shape X at temperature 2 is $\Theta\left(\frac{K(X)}{\log K(X)}\right)$, where $K(X)$ is the size of the smallest Turing machine that outputs the list of points in X . This tight bound for temperature-2 self-assembly in 2D was shown to hold for temperature-1 self-assembly in just-barely 3D by Furcy and Summers [10].

Another benchmark shape, for positive integers k, N , is the $k \times N$ rectangle, where $k < \frac{\log N}{\log \log N - \log \log \log N}$, making it “thin”. A thin rectangle is an interesting testbed because its restricted height creates a limited channel through which tiles may propagate information, for example, the current value of a self-assembling counter. In fact, Aggarwal, Cheng, Goldwasser, Kao, Moisset de Espanés and Schweller [2] used an optimal, base- $\lceil N^{\frac{1}{k}} \rceil$ counter that uniquely self-assembles within the restricted confines of a thin rectangle to derive an upper bound of $O\left(N^{\frac{1}{k}} + k\right)$ on the directed tile complexity of a $k \times N$ thin rectangle at temperature 2 in 2D. They then leveraged the limited bandwidth of a thin rectangle in a counting argument for a corresponding lower bound of $\Omega\left(\frac{N^{\frac{1}{k}}}{k}\right)$.

The previous theory for a square and an algorithmically specified shape would suggest that these thin rectangle bounds should hold at temperature 1 in just-barely 3D. Yet, we currently do not know if this is the case. Thus, the power of temperature-1 self-assembly in just-barely 3D resembles that of temperature-2 self-assembly in 2D, with respect to the directed tile complexities of a square and a scaled-up version of any algorithmically specified shape, but not a thin rectangle.

Motivated by this theoretical discrepancy, we prove new lower and upper bounds on the directed tile complexity of a thin rectangle at temperature 1 in just-barely 3D, where $R_{k,N}^3$ is a just-barely 3D $k \times N$ rectangle if it satisfies $\{0, 1, \dots, N - 1\} \times \{0, 1, \dots, k - 1\} \times \{0\} \subseteq R_{k,N}^3 \subseteq \{0, 1, \dots, N - 1\} \times \{0, 1, \dots, k - 1\} \times \{0, 1\}$. See Tables 1 and 2 for a quick summary of our results and how they compare with previous state-of-the-art results.

■ **Table 1** State-of-the-art directed tile complexity for benchmark shapes in the aTAM, where $K(X)$ is the size of the smallest Turing machine that outputs the list of points in X .

	2D Temperature 2		Just-barely 3D Temperature 1	
	Lower bound	Upper bound	Lower bound	Upper bound
$N \times N$ Square	$\Theta\left(\frac{\log N}{\log \log N}\right)$		Same as 2D Temperature 2	
Algorithmically-defined shape X	$\Theta\left(\frac{K(X)}{\log K(X)}\right)$		Same as 2D Temperature 2	
$k \times N$ rectangle	$\Omega\left(\frac{N^{\frac{1}{k}}}{k}\right)$	$O\left(N^{\frac{1}{k}} + k\right)$	$\Omega\left(\frac{N^{\frac{1}{2k}}}{k}\right)$	$O\left(N^{\lceil \frac{1}{2} \rceil} + \log N\right)$

■ **Table 2** In this table, we highlight our improved lower and upper bounds on the directed tile complexity of rectangles, the two main contributions of this paper, and compare them with corresponding bounds in 2D at temperature 2. Note that, for thin rectangles, the additive logarithmic term disappears, since, for a thin rectangle, $k < \frac{\log N}{\log \log N - \log \log \log N}$, which implies that $\log N < N^{\frac{1}{k}} < N^{\frac{1}{k-1}}$, for sufficiently large k and N .

	2D Temperature 2		Just-barely 3D Temperature 1	
	Lower bound	Upper bound	Lower bound	Upper bound
$k \times N$ rectangle	$\Omega\left(\frac{N^{\frac{1}{k}}}{k}\right)$	$O\left(N^{\frac{1}{k}} + k\right)$	$\Omega\left(N^{\frac{1}{k}}\right)$	$O\left(N^{\frac{1}{k-1}} + \log N\right)$

4:4 Tile Complexity of Uniquely Self-Assembling Thin Rectangles

First, we have our upper bound:

► **Theorem 1.** *The directed tile complexity of a just-barely 3D $k \times N$ rectangle at temperature 1 is $O\left(N^{\frac{1}{k-1}} + \log N\right)$.*

Theorem 1 is an asymptotic improvement over the previous state-of-the-art upper bound: $O\left(N^{\lfloor \frac{1}{3} \rfloor} + \log N\right)$ [11]. The latter bound is based on the self-assembly of a just-barely 3D counter that uniquely self-assembles at temperature 1, but whose base M depends on the dimensions of the target rectangle, where each digit is represented geometrically and in binary within a just-barely 3D region of space comprised of $\Theta(\log N)$ columns and 3 rows. In a construction like this, the number of rows used to represent each digit affects the base of the counter, which, for a thin rectangle, turns out to be the asymptotically-dominating term in the tile complexity. For example, in the Furcy, Summers and Wendlandt construction, the number of rows per digit is 3, so the base is set to $\Theta\left(N^{\lfloor \frac{1}{3} \rfloor}\right)$. Intuitively, “squeezing” more digits into the counter for the same rectangle of height k will result in a decrease in the base and therefore the tile complexity.

Our construction for Theorem 1 is based on the self-assembly of a just-barely 3D counter similar to the Furcy, Summers and Wendlandt construction, but the geometric structure of our counter is organized according to digit regions, or just-barely 3D regions of space comprised of k rows and $\Theta\left(N^{\frac{1}{k-1}}\right)$ columns, in which $k - 2$ base- $\Theta\left(N^{\frac{1}{k-1}}\right)$ digits are represented in a staggered fashion. This increase in digit density is the main reason why the “ $\lfloor \frac{k}{3} \rfloor$ ” term from the Furcy, Summers and Wendlandt upper bound is replaced by a “ $k - 1$ ” term in Theorem 1. Finally, we have our lower bound:

► **Theorem 2.** *The directed tile complexity of a just-barely 3D $k \times N$ rectangle at temperature 1 is $\Omega\left(N^{\frac{1}{k}}\right)$.*

Theorem 2 is an asymptotic improvement over the previous state-of-the-art lower bound: $\Omega\left(\frac{N^{\frac{1}{2k}}}{k}\right)$. Technically, the latter bound is not explicitly proved (or even stated) and therefore cannot be referenced, but it can be derived via a straightforward application of the standard Window Movie Lemma introduced in [17]. On our way to proving Theorem 2, we prove Lemma 5, which is essentially a new, more powerful type of Window Movie Lemma technique, specifically designed for temperature-1 self-assembly within a just-barely 3D, rectangular region of space. We conjecture that the conclusion of Lemma 5 can be generalized to give a powerful tool for proving even better lower bounds on the directed tile complexity of 3D non-rectangular shapes at temperature-1 than what would otherwise be possible with the standard Window Movie Lemma. Lemma 5 lets us develop a more refined counting argument based on upper bounding the number of “sufficiently similar” ways for an assembly sequence to assign glues to a fixed set of locations abutting a plane. Intuitively, two assignments are sufficiently similar if, up to translation, they respectively agree on: the set of locations to which glues are assigned, the local order in which certain consecutive pairs of glues appear, and the glues that are assigned to a certain set (of roughly half) of the locations.

2 Formal definition of the abstract Tile Assembly Model

In this section, we briefly sketch a strictly 3D version of Winfree's abstract Tile Assembly Model (see also [14, 21, 22]).

All logarithms in this paper are base-2. A *grid graph* is an undirected graph $G = (V, E)$, where $V \subset \mathbb{Z}^3$, such that, for all $\{\vec{a}, \vec{b}\} \in E$, $\vec{a} - \vec{b}$ is a 3-dimensional unit vector. The *full grid graph* of V is the undirected graph $G_V^f = (V, E)$, such that, for all $\vec{x}, \vec{y} \in V$, $\{\vec{x}, \vec{y}\} \in E \iff \|\vec{x} - \vec{y}\| = 1$, i.e., if and only if \vec{x} and \vec{y} are adjacent in the 3-dimensional integer Cartesian space.

A 3-dimensional *tile type* is a tuple $t \in (\Sigma^* \times \mathbb{N})^6$, e.g., a unit cube, with six sides, listed in some standardized order, and each side having a *glue* $g \in \Sigma^* \times \mathbb{N}$ consisting of a finite string *label* and a nonnegative integer *strength*. We assume a finite set of tile types, but an infinite number of copies of each tile type, each copy referred to as a *tile*. A *tile set* is a set of tile types and is usually denoted as T .

A *configuration* is a (possibly empty) arrangement of tiles on the integer lattice \mathbb{Z}^3 , i.e., a partial function $\alpha : \mathbb{Z}^3 \dashrightarrow T$. Two adjacent tiles in a configuration *bind*, *interact*, or are *attached*, if the glues on their abutting sides are equal (in both label and strength) and have positive strength. Each configuration α induces a *binding graph* G_α^b , a grid graph whose vertices are positions occupied by tiles, according to α , with an edge between two vertices if the tiles at those vertices bind. An *assembly* is a connected, non-empty configuration, i.e., a partial function $\alpha : \mathbb{Z}^3 \dashrightarrow T$ such that $G_{\text{dom } \alpha}^f$ is connected and $\text{dom } \alpha \neq \emptyset$. Given $\tau \in \mathbb{Z}^+$, α is τ -*stable* if every cut-set of G_α^b has weight at least τ , where the weight of an edge is the strength of the glue it represents. When τ is clear from context, we say α is *stable*. Given two assemblies α, β , we say α is a *subassembly* of β , and we write $\alpha \sqsubseteq \beta$, if $\text{dom } \alpha \subseteq \text{dom } \beta$ and, for all points $\vec{p} \in \text{dom } \alpha$, $\alpha(\vec{p}) = \beta(\vec{p})$.

A 3-dimensional *tile assembly system* (TAS) is a triple $\mathcal{T} = (T, \sigma, \tau)$, where T is a tile set, $\sigma : \mathbb{Z}^3 \dashrightarrow T$ satisfying $|\text{dom } \sigma| = 1$ is the *seed assembly* (trivially τ -stable), and $\tau \in \mathbb{Z}^+$ is the *temperature*. Given two τ -stable assemblies α, β , we write $\alpha \rightarrow_1^\mathcal{T} \beta$ if $\alpha \sqsubseteq \beta$ and $|\text{dom } \beta \setminus \text{dom } \alpha| = 1$. In this case we say α \mathcal{T} -*produces* β *in one step*. If $\alpha \rightarrow_1^\mathcal{T} \beta$, $\text{dom } \beta \setminus \text{dom } \alpha = \{\vec{p}\}$, and $t = \beta(\vec{p})$, we write $\beta = \alpha + (\vec{p} \mapsto t)$. The \mathcal{T} -*frontier* of α is the set $\partial^\mathcal{T} \alpha = \bigcup_{\alpha \rightarrow_1^\mathcal{T} \beta} (\text{dom } \beta \setminus \text{dom } \alpha)$, i.e., the set of empty locations at which a tile could stably attach to α . The t -*frontier* of α , denoted $\partial_t^\mathcal{T} \alpha$, is the subset of $\partial^\mathcal{T} \alpha$ defined as $\{\vec{p} \in \partial^\mathcal{T} \alpha \mid \alpha \rightarrow_1^\mathcal{T} \beta \text{ and } \beta(\vec{p}) = t\}$.

Let \mathcal{A}^T denote the set of all assemblies of tiles from T , and let $\mathcal{A}_{<\infty}^T$ denote the set of finite assemblies of tiles from T . A sequence of $k \in \mathbb{Z}^+ \cup \{\infty\}$ assemblies $\vec{\alpha} = (\alpha_0, \alpha_1, \dots)$ over \mathcal{A}^T is a \mathcal{T} -*assembly sequence* if, for all $1 \leq i < k$, $\alpha_{i-1} \rightarrow_1^\mathcal{T} \alpha_i$. The *result* of an assembly sequence $\vec{\alpha}$, denoted as $\text{res}(\vec{\alpha})$, is the unique limiting assembly (for a finite sequence, this is the final assembly in the sequence). We write $\alpha \rightarrow^\mathcal{T} \beta$, and we say α \mathcal{T} -*produces* β (in 0 or more steps), if there is a \mathcal{T} -assembly sequence $\alpha_0, \alpha_1, \dots$ of length $k = |\text{dom } \beta \setminus \text{dom } \alpha| + 1$ such that (1) $\alpha = \alpha_0$, (2) $\text{dom } \beta = \bigcup_{0 \leq i < k} \text{dom } \alpha_i$, and (3) for all $0 \leq i < k$, $\alpha_i \sqsubseteq \beta$. We say α is \mathcal{T} -*producible* if $\sigma \rightarrow^\mathcal{T} \alpha$, and we write $\mathcal{A}[\mathcal{T}]$ to denote the set of \mathcal{T} -producible assemblies. An assembly α is \mathcal{T} -*terminal* if α is τ -stable and $\partial^\mathcal{T} \alpha = \emptyset$. We write $\mathcal{A}_\square[\mathcal{T}] \subseteq \mathcal{A}[\mathcal{T}]$ to denote the set of \mathcal{T} -producible, \mathcal{T} -terminal assemblies. If $|\mathcal{A}_\square[\mathcal{T}]| = 1$ then \mathcal{T} is said to be *directed*. We say that a TAS \mathcal{T} *uniquely self-assembles* a shape $X \subseteq \mathbb{Z}^3$ if $\mathcal{A}_\square[\mathcal{T}] = \{\alpha\}$ and $\text{dom } \alpha = X$.

The *directed tile complexity* of a shape X at temperature τ is the minimum number of distinct tile types of any TAS that uniquely self-assembles (USA) X , denoted by $K_{USA}^\tau(X) = \min \{n \mid \mathcal{T} = (T, \sigma, \tau), |T| = n \text{ and } \mathcal{T} \text{ uniquely self-assembles } X\}$.

3 The upper bound

In this section, we prove Theorem 1, our upper bound, namely that $K_{USA}^1(R_{k,N}^3) = O(N^{\frac{1}{k-1}} + \log N)$. In order to do so, we construct a TAS that uniquely self-assembles a sufficiently large rectangle (of any height $k \geq 3$) $R_{k,N}^3$. Specifically, we construct a TAS $\mathcal{T} = (T, \sigma, 1)$ so that it simulates a base $B = \lceil N^{\frac{1}{k-1}} \rceil$, $W = k - 2$ digit counter, henceforth referred to as *the counter*, that starts counting at a specified starting value and stops after the maximum value is incremented, before rolling over to 0. In the remainder of this section, we will describe the self-assembly of the counter.

Each W -digit, base- B value of the counter is represented in a corresponding just-barely 3D rectangular region of space called a digit region. There are two types of digit regions, one for each type of counter step: *copy* and *increment*. The former duplicates the value from the previous increment region and the latter increments the value from the previous copy region. The counter alternates between increment and copy steps.

Counter digits are represented geometrically and in binary, using the *bit bump* technique by Cook, Fu and Schweller [3]. Each digit is comprised of $b + 2 = \lceil \log B \rceil + 2$ bit bumps that protrude from a row of tiles. Each bit bump geometrically encodes one bit. The two most significant (westernmost) bits of a digit are its *indicator bits*: 10 – most significant, 01 – least significant, 00 – neither, 11 – both. The rest of the bits represent a base- B value. Bumps of a digit in increment (copy) regions protrude to the south (north). If the bump is in the $z = 0$ ($z = 1$) plane, then it represents a 0 (1). The W digits in a digit region are staggered like the steps of a staircase, descending (ascending) in a copy (increment) region. Figure 1 shows the layout of the two types of digit regions, positioned consecutively as they would be in the counter, which is self-assembling to the east.

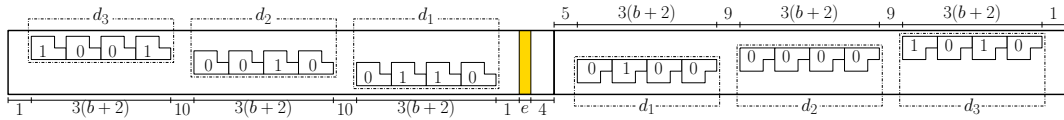
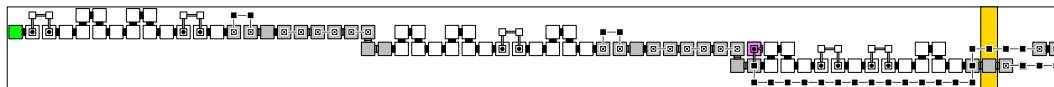


Figure 1 A copy region (west) and an increment region (east). The numeric quantities below and above each region indicate a number of columns. Note that $W = 3$, so $k = 5$ (not drawn to scale).

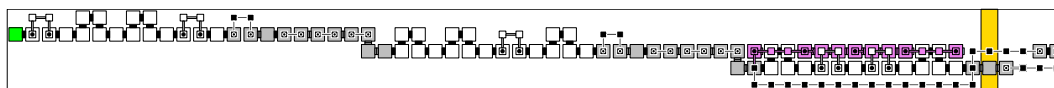
Digit regions (Figure 1): We assume that the most significant bit of a digit is represented by the westernmost bit bump (to the east of the indicator bits), which means that we have $d_1 = 2$ (least significant digit), $d_2 = 2$ and $d_W = d_3 = 1$ (most significant digit). Thus, the base-3 value 122 in the copy region is incremented to 200 in the increment region. The orange column represents a variable number of $e = B - (2W(3(b+2)) + 50)$ extra columns. The extra columns ensure that the combined width of two consecutive digit regions is B . The idea is that the terminal assembly of \mathcal{T} is a sequence of copy-increment digit region pairs with one pair per value of the counter. Since each value of the counter corresponds to a pair of consecutive digit regions and e is such that the width of two consecutive digit regions is B columns, assuming the counter starts counting at 0, the width of the k -row terminal assembly that \mathcal{T} will produce is $B \cdot B^W = \lceil N^{\frac{1}{k-1}} \rceil \cdot \lceil N^{\frac{1}{k-1}} \rceil^{k-2} \geq N^{\frac{1}{k-1}} \cdot N^{\frac{k-2}{k-1}} \geq N$. Then, \mathcal{T} can be modified to produce a unique terminal assembly of height k and width N , by using a positive starting value and $O(N \bmod B)$ additional tile types.

Figures 2 through 10 illustrate the self-assembly of an increment step for an artificial example with $B = 3$, $k = 5$ and starting value 122.



■ **Figure 2** The initial value assembly.

The initial value (Figure 2): The initial value is represented inside a copy region. The green tile is the seed tile. We use big (small) squares to represent tiles placed in the $z = 0$ ($z = 1$) plane. A glue between a $z = 0$ tile and $z = 1$ tile is denoted as a small disk. Glues between $z = 1$ ($z = 0$) tiles are denoted as thin (thick) lines. Each three-tile-wide bit bump geometrically encodes one bit. The bits of a digit are comprised of white tiles. In every copy region, the bump in the $z = 1$ plane immediately east of digit d_i for $i > 1$ does not represent a bit, but rather a portion of the assembly that will eventually block the self-assembly of a subsequent path of repeating tiles (for example, the path of red tiles in Figure 6). The two easternmost tiles in the $z = 0$ plane will also block the self-assembly of a subsequent path of repeating tiles (for example, the path of blue tiles in Figure 4). The tiles that traverse the orange column represent paths of e tiles. In general, we can *hard-code* a path of tiles that uniquely self-assembles a corresponding initial value assembly, from the seed to the $z = 0$ purple tile, where the glues of each tile type along the path encode the relative location of the tile in the path. In general, such a path contributes $O(e + kb) = O(B + \log N)$ tile types to T . After the initial value self-assembles, the counter executes an increment step. A bit indicating the presence of an arithmetic carry, *the carry bit*, initially set to 1, is introduced in the east-facing glues of both purple tiles that specifically start reading d_1 for an increment step. The purple tiles that start reading d_i for $i > 1$ propagate the carry bit.

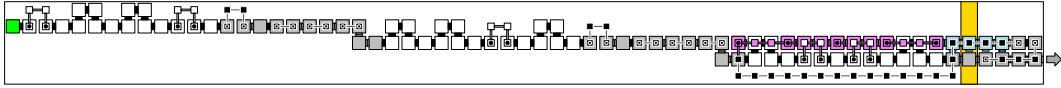


■ **Figure 3** The purple tiles are reading d_1 from most to least significant bit (west to east).

Read digit (Figure 3): The purple tiles are reading the four bits of d_1 in the copy region, starting with its most significant (westernmost) bit. In general, the counter reads the value of a digit as follows. Below the westernmost purple tile in the $z = 1$ plane, another purple tile attaches in the $z = 0$ plane (see also the purple tiles in Figure 2) and they both have east-facing glues. The east-facing glue on the $z = 1$ ($z = 0$) tile encodes 0 (1), but, due to the presence of the westernmost bit bump, only one is exposed. For each $1 \leq i < b + 2$ and $x \in \{0, 1\}^{i-1}$, there corresponds a *reader gadget* that reads a 0 in bit position i (for notational convenience, $i = 1$ is the position of the most significant bit of a digit). Such a reader gadget self-assembles a horizontal path of three tiles in the $z = 1$ plane and a tile below the easternmost $z = 1$ tile (in the $z = 0$ plane), such that, the west-facing *input* glue of its westernmost tile encodes $x0$, the east-facing *output* glue of its easternmost $z = 0$ tile encodes $x01$ and the east-facing output glue of its easternmost $z = 1$ tile encodes $x00$. A reader gadget that reads a 1 is defined similarly. All reader gadgets also propagate the carry bit. A reader gadget that reads the bit in position $b + 2$ is similar to previous reader gadgets but has only one output glue, which is on a tile in the $z = 0$ plane and encodes the value of its input glue. It also initiates the self-assembly of a path of repeating tiles, along which

4:8 Tile Complexity of Uniquely Self-Assembling Thin Rectangles

the value of the digit that was just read, the carry bit and a bit indicating whether d_1 was just read are propagated. The presence of all $b + 2$ bit bumps ensures a unique assembly sequence of $b + 2$ corresponding reader gadgets. In general, for each $1 \leq i \leq b + 2$, we need $O(2^i)$ reader gadgets that read a bit in position i . Thus, in general, all the reader gadgets contribute $O(2^b) = O(B)$ tile types to T . Our reader gadgets are inspired by the simulation macro tiles depicted in Figures 3 and 6 of [3].



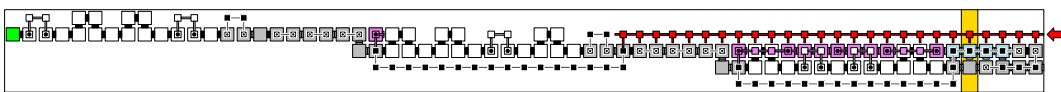
■ **Figure 4** The path of repeating blue tiles propagates d_1 and the carry bit to the increment region.

Propagate digit (Figure 4): After the bits of any digit in a copy region are read, a blue tile type with equal west and east glues self-assembles in a path of repeating tiles to the east, propagating the value of the digit that was just read and the carry bit. A previous portion of the assembly blocks the self-assembly of this path, at which point the south-facing glue of the easternmost blue tile in the path is exposed, from which a fixed size, hard-coded path of gray tiles self-assembles to the location immediately west of the most significant bit of d_1 in the increment region. The path of repeating blue tiles, all of the same type, propagate the value of a base- B digit, the carry bit and a bit indicating whether d_1 was just read, thus, in general, contributing $O(B)$ tile types to T . Similarly, the hard-coded path, in general, contributes $O(B)$ tile types to T . We use two different types of hard-coded paths: one for d_1 and another for d_i for $i > 1$. The latter self-assembles a bump in the $z = 1$ plane that eventually blocks a subsequent path of repeating tiles.



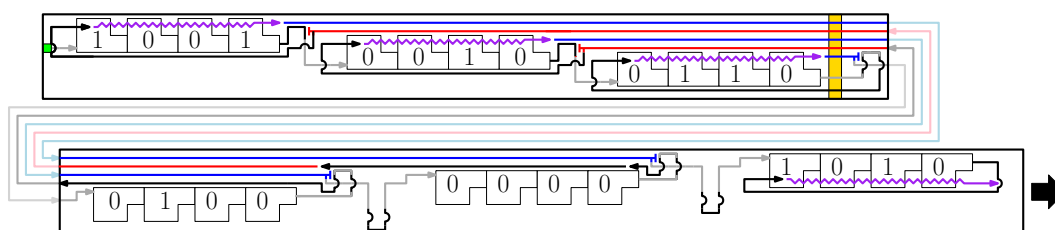
■ **Figure 5** The bits of d_1 , read in Figure 3, are written.

Write digit (Figure 5): In this example, the value of d_1 is incremented from 2 and rolls over to 0, resulting in a carry out. The bits are written using fixed size *writer* gadgets, that work in a fashion similar to the reader gadgets, where we have a corresponding writer gadget for each bit position and each one propagates the carry bit (the first writer gadget receives the carry bit, updates it accordingly and propagates it). Thus, just like the reader gadgets, the writer gadgets contribute $O(B)$ tile types to T . The two easternmost $z = 0$ tiles will eventually block the self-assembly of a subsequent path of repeating blue tiles (for example, the blue tiles in Figure 4 but after reading a different digit). The path of tiles, starting with the gray tile immediately east of the least significant bit and ending at the westernmost $z = 1$ black tile is hard-coded and propagates the carry bit, thus, in general, contributing $O(b) = O(\log N)$ tile types to T . Note that, in general, the same tile types are used for the self-assembly of similar paths that self-assemble after writing every digit except d_W in an increment region.



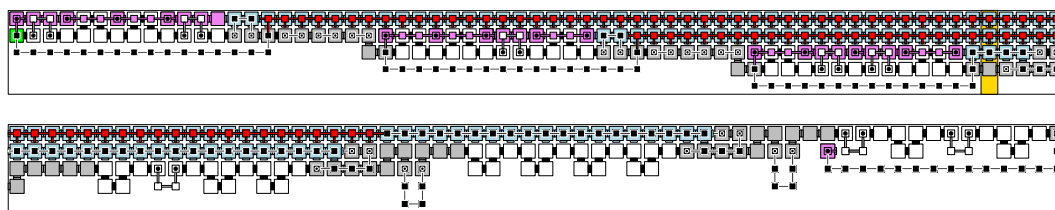
■ **Figure 6** Return to the copy region to read the next digit, d_2 .

Return to read another digit (Figure 6): We use a red tile type with equal west and east glues to self-assemble a path of repeating tiles to the west, starting six tiles to the west of the most significant bit of the digit that was just written in the increment region. A previous portion of the assembly, namely the bump in the $z = 1$ plane that is east of the least significant bit of each digit d_i for $i > 1$ in a copy region blocks the self-assembly of the path of repeating red tiles, at which point the south-facing glue of the westernmost red tile in the path is exposed, from which a hard-coded path of tiles self-assembles, ending at the $z = 0$ purple tile. The red tile types propagate the carry bit, thus, in general, contributing $O(1)$ tile types to T . The hard-coded path also propagates the carry bit, thus, in general, contributing $O(b) = O(\log N)$ tile types to T . Note that the same tile types are used for the self-assembly of similar paths that self-assemble after writing every digit except d_W in an increment region.



■ **Figure 7** A conceptual depiction of the complete self-assembly of an increment region (bottom), given the initial value copy region (top).

Increment step high-level assembly sequence and corresponding full assembly (Figures 7 and 8): The assembly sequence begins at the location of the green seed tile in the copy region. The gray (black) lines are hard-coded paths of tiles in the $z = 0$ ($z = 1$) plane. The blue (red) lines are paths of repeating tiles in the $z = 0$ ($z = 1$) plane. The purple zig-zag lines represent the reader gadgets. A corresponding full assembly is shown in Figure 8. The counter concludes an increment step after it writes d_W in the increment region. The purple zig-zag line through d_W in the increment region represents the first sequence of tile placements for the next copy step. Note that if the carry bit is 1 after writing d_W in the increment region, then the counter can stop counting.

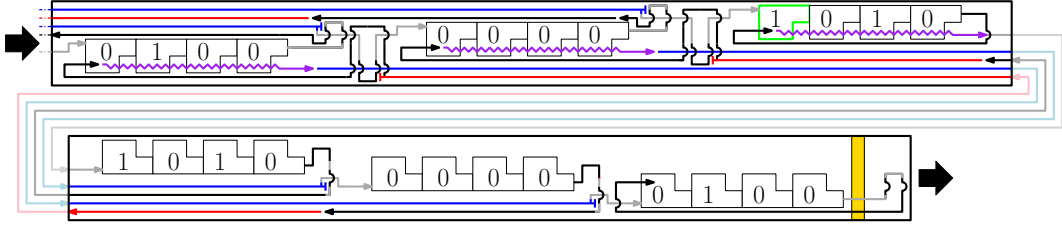


■ **Figure 8** The full assembly of the initial value copy region is on the top and the next increment region is on the bottom. A producible, non-terminal assembly results when the latter is translated so that its westernmost column is immediately east of the easternmost column of the former.

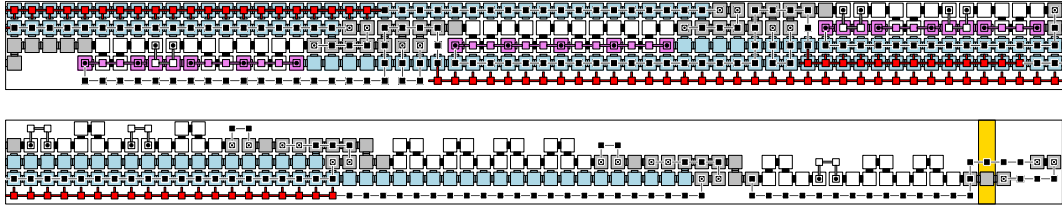
Copy step high-level assembly sequence: (Figures 9 and 10): After an increment step concludes, a copy step is executed. A copy step is carried out in a fashion similar to an increment step, using a specific set of tile types. In a copy step, the digits are read from the previous increment region and written in the next copy region in the order d_W to d_1 (reverse

4:10 Tile Complexity of Uniquely Self-Assembling Thin Rectangles

of an increment step). Regardless, the tile types for a copy step, in general, contribute $O(B + \log N)$ tile types to T . Moreover, all the tile types used for an increment step are defined to be disjoint from those for a copy step, which has no effect on the asymptotic size of T .



■ **Figure 9** A conceptual depiction of the self-assembly of a copy region (bottom) from a given increment region (top), which begins at the most significant bit of d_W in the increment region (outlined in green). A copy step concludes after the self-assembly of the hard-coded path of tiles, starting at the least and ending at the most significant bit of d_1 (copy region). This hard-coded path contributes $O(e)$ tile types to T .



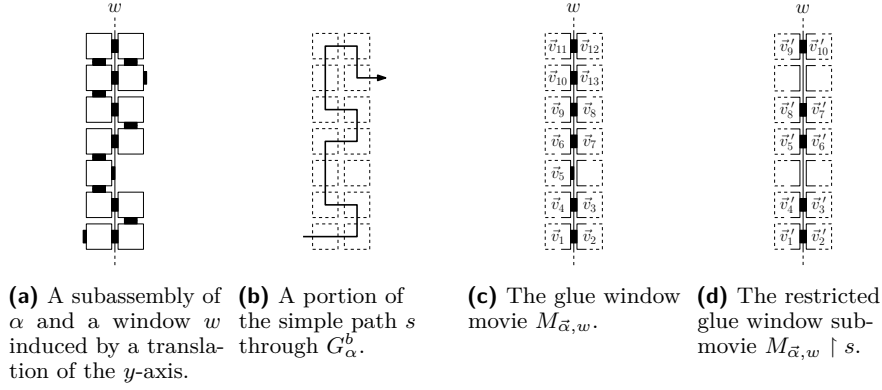
■ **Figure 10** The full assembly of the completed increment region is on the top (see also the bottom assembly of Figure 8) and the next copy region is on the bottom. A producible, non-terminal assembly results when first the latter is translated so that its westernmost column is immediately east of the easternmost column of the former and then this assembly is translated so its westernmost column is immediately east of the easternmost column of the top assembly shown in Figure 8.

Tile complexity and correctness: From the preceding discussion, generalized to arbitrary k and N , we have $|T| = O(B + \log N) = O\left(N^{\frac{1}{k-1}} + \log N\right)$, from which the bound for Theorem 1 follows. Formal correctness can be shown using the method of Conditional Determinism by Shutters and Lutz [15].

4 The lower bound

In this section, we prove Theorem 2, our lower bound, namely that $K_{USA}^1\left(R_{k,N}^3\right) = \Omega\left(N^{\frac{1}{k}}\right)$. We first give some notation that will be used throughout the remainder of this section. Let $\mathcal{T} = (T, \sigma, \tau = 1)$ be a 3D TAS with $\alpha \in \mathcal{A}_{\square}[\mathcal{T}]$. Assume $s = (\vec{x}_0, \vec{x}_1, \dots, \vec{x}_m)$ is a simple path in G_{α}^b , such that, $\vec{x}_0 = \text{dom } \sigma$. We say that $\vec{\alpha}$ follows s if there is a \mathcal{T} -assembly sequence $\vec{\alpha} = (\alpha_i \mid 0 \leq i < m + 1)$ that satisfies the next two conditions: $\alpha_0 = \sigma$, and for all $0 < i \leq m$, $\text{dom } \alpha_i \setminus \text{dom } \alpha_{i-1} = \{\vec{x}_i\}$.

This paragraph contains definitions that were taken directly from [17]. A *window* w is a set of edges forming a cut-set of the full grid graph of \mathbb{Z}^3 . Given a window w and an assembly α , a window that *intersects* α is a partitioning of α into two configurations (i.e., after being split into two parts, each part may or may not be disconnected). In this case



■ **Figure 11** An assembly, a simple path, and two types of glue window movies in 2D. Here, we have $M_{\vec{\alpha},w} = (\vec{v}_1, g_1), (\vec{v}_2, g_2), (\vec{v}_3, g_3), (\vec{v}_4, g_4), (\vec{v}_5, g_5), (\vec{v}_6, g_6), (\vec{v}_7, g_7), (\vec{v}_8, g_8), (\vec{v}_9, g_9), (\vec{v}_{10}, g_{10}), (\vec{v}_{11}, g_{11}), (\vec{v}_{12}, g_{12}), (\vec{v}_{13}, g_{13})$, where $g_1 = g_2, g_3 = g_4, g_6 = g_7, g_8 = g_9, g_{11} = g_{12}$ and $g_{13} = g_{10}$. Note that $M_{\vec{\alpha},w} \upharpoonright s$ only includes the location-glue pairs where the glues actually form bonds between locations in s . For example, \vec{v}_{10} and \vec{v}_{13} are excluded from $M_{\vec{\alpha},w} \upharpoonright s$ because the glues that connect them are not part of the path of glue that follow s .

we say that the window w cuts the assembly α into two non-overlapping configurations α_L and α_R , satisfying, for all $\vec{x} \in \text{dom } \alpha_L$, $\alpha(\vec{x}) = \alpha_L(\vec{x})$, for all $\vec{x} \in \text{dom } \alpha_R$, $\alpha(\vec{x}) = \alpha_R(\vec{x})$, and $\alpha(\vec{x})$ is undefined at any point $\vec{x} \in \mathbb{Z}^3 \setminus (\text{dom } \alpha_L \cup \text{dom } \alpha_R)$. Given a window w , its translation by a vector $\vec{\Delta}$, written $w + \vec{\Delta}$ is simply the translation of each one of w 's elements (edges) by $\vec{\Delta}$. All windows in this paper are assumed to be induced by some translation of the yz -plane. Each window is thus uniquely identified by its x coordinate. For a window w and an assembly sequence $\vec{\alpha}$, we define a *glue window movie* M to be the order of placement, position and glue type for each glue that appears along the window w in $\vec{\alpha}$, regardless of whether the glue (eventually) forms a bond. Given an assembly sequence $\vec{\alpha}$ and a window w , the associated glue window movie is the maximal sequence $M_{\vec{\alpha},w} = (\vec{v}_1, g_1), (\vec{v}_2, g_2), \dots$ of pairs of grid graph vertices \vec{v}_i and glues g_i , given by the order of appearance of the glues along window w in the assembly sequence $\vec{\alpha}$. We write $M_{\vec{\alpha},w} + \vec{\Delta}$ to denote the translation by $\vec{\Delta}$ of $M_{\vec{\alpha},w}$, yielding $(\vec{v}_1 + \vec{\Delta}, g_1), (\vec{v}_2 + \vec{\Delta}, g_2), \dots$

If $\vec{\alpha}$ follows s , then the notation $M_{\vec{\alpha},w} \upharpoonright s$ denotes the *restricted* glue window submovie (*restricted to* s), which consists of only those steps of $M_{\vec{\alpha},w}$ that place glues that immediately form positive-strength bonds that cross w at locations belonging to the simple path s . Let \vec{v} denote the location of the starting point of s (i.e., the location of σ). Let \vec{v}_i and \vec{v}_{i+1} denote two consecutive locations in $M_{\vec{\alpha},w} \upharpoonright s$ that are located across w from each other. We say that these two locations define a *crossing* of w , where a crossing has exactly one direction. We say that this crossing is *away from* \vec{v} (or *away from* σ) if the x coordinates of \vec{v} and \vec{v}_i are equal or the x coordinate of \vec{v}_i is between the x coordinates of \vec{v} and \vec{v}_{i+1} . In contrast, we say that this crossing is *toward* \vec{v} (or *toward* σ) if the x coordinates of \vec{v} and \vec{v}_{i+1} are equal or the x coordinate of \vec{v}_{i+1} is between the x coordinates of \vec{v} and \vec{v}_i . See Figure 11 for 2D examples of $M_{\vec{\alpha},w}$ and $M_{\vec{\alpha},w} \upharpoonright s$, where σ is located west of w and the locations \vec{v}_1 and \vec{v}_2 form an away crossing, whereas the locations \vec{v}_3 and \vec{v}_4 form a crossing toward σ .

We say that two restricted glue window submovies are “sufficiently similar” if they have the same (odd) number of crossings, the same set of crossing locations (up to horizontal translation), the same crossing directions at corresponding crossing locations, and the same glues in corresponding “away crossing” locations.

► **Definition 3.** Assume: $\mathcal{T} = (T, \sigma, 1)$ is a 3D TAS, $\alpha \in \mathcal{A}[\mathcal{T}]$, s is a simple path in G_α^b starting from the location of σ , $\vec{\alpha}$ is a sequence of \mathcal{T} -producible assemblies that follows s , w and w' are windows, σ is not located between w and w' , $\vec{\Delta} \neq \vec{0}$ is a vector satisfying $w' = w + \vec{\Delta}$, e and e' are two odd numbers, and $M = M_{\vec{\alpha}, w} \upharpoonright s = (\vec{v}_1, g_1), \dots, (\vec{v}_{2e}, g_{2e})$ and $M' = M_{\vec{\alpha}, w'} \upharpoonright s = (\vec{v}'_1, g'_1), \dots, (\vec{v}'_{2e'}, g'_{2e'})$ are both non-empty restricted glue window submovies. We say that M and M' are sufficiently similar if the following are satisfied:

1. same number of crossings: $e = e'$,
2. same set of crossing locations (up to translation):

$$\left\{ \vec{v}_i + \vec{\Delta} \mid 1 \leq i \leq 2e \right\} = \left\{ \vec{v}'_j \mid 1 \leq j \leq 2e' \right\},$$
3. same crossing directions at corresponding crossing locations:

$$\left\{ \vec{v}_{4i-2} + \vec{\Delta} \mid 1 \leq i \leq \frac{e+1}{2} \right\} = \left\{ \vec{v}'_{4j-2} \mid 1 \leq j \leq \frac{e'+1}{2} \right\},$$
 and
4. same glues in corresponding “away crossing” locations:
 for all $1 \leq i, j \leq \frac{e+1}{2}$, if $\vec{v}'_{4j-2} = \vec{v}_{4i-2} + \vec{\Delta}$, then $g'_{4j-2} = g_{4i-3}$.

See Figure 12 for an example of two restricted glue window submovies that are sufficiently similar. The following result basically says that we must examine only a “small” number of distinct restricted glue window submovies in order to find two different ones that are sufficiently similar.

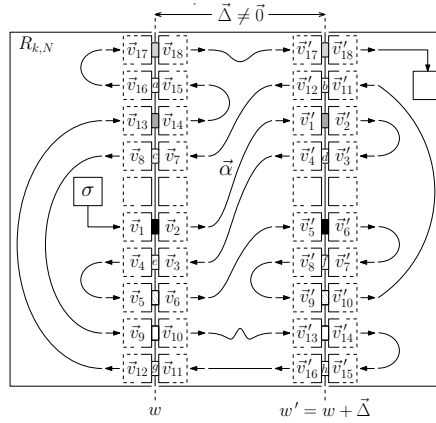
► **Lemma 4.** Assume: $\mathcal{T} = (T, \sigma, 1)$ is a 3D TAS, G is the set of all glues in T , $k, N \in \mathbb{Z}^+$, s is a simple path starting from the location of σ such that $s \subseteq R_{k,N}^3$, $\vec{\alpha}$ is a sequence of \mathcal{T} -producible assemblies that follows s , $m \in \mathbb{Z}^+$, for all $1 \leq l \leq m$, w_l is a window, for all $1 \leq l < l' \leq m$, $\vec{\Delta}_{l,l'} \neq \vec{0}$ satisfies $w_{l'} = w_l + \vec{\Delta}_{l,l'}$, and for all $1 \leq l \leq m$, there is an odd $1 \leq e_l < 2k$ such that $M_{\vec{\alpha}, w_l} \upharpoonright s$ is a non-empty restricted glue window submovie of length $2e_l$. If $m > |G|^k \cdot k \cdot 16^k$, then there exist $1 \leq l < l' \leq m$ such that $e_l = e_{l'} = e$ and $M_{\vec{\alpha}, w_l} \upharpoonright s = (\vec{v}_1, g_1), \dots, (\vec{v}_{2e}, g_{2e})$ and $M_{\vec{\alpha}, w_{l'}} \upharpoonright s = (\vec{v}'_1, g'_1), \dots, (\vec{v}'_{2e}, g'_{2e})$ are sufficiently similar non-empty restricted glue window submovies.

To prove Lemma 4, we first count the number of ways to choose the set $\{\vec{v}_1, \dots, \vec{v}_{2e}\}$. Then, we count the number of ways to choose the set $\{\vec{v}_{4i-2} \mid 1 \leq i \leq \frac{e+1}{2}\}$. Finally, we count the number of ways to choose the sequence $(g_{\vec{x}_i} \mid i = 1, \dots, \frac{e+1}{2})$. After summing over all odd e , we get the indicated lower bound on m that notably neither contains a “factorial” term nor a coefficient on the “ k ” in the exponent of “ $|G|$ ”. The full proof of Lemma 4 is omitted from this version of the paper.

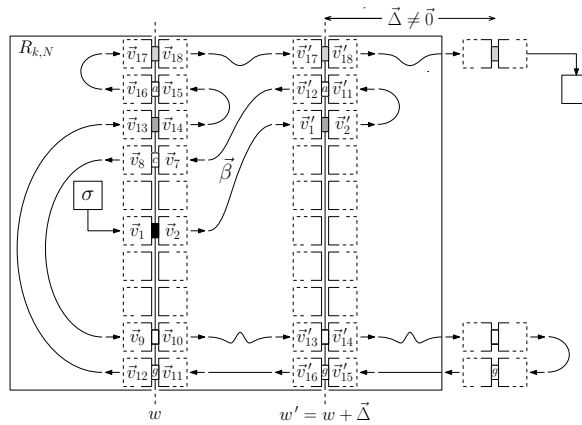
The following result is the cornerstone of our lower bound machinery. It basically says that if, for some directed TAS \mathcal{T} , two distinct restricted glue window submovies are sufficiently similar, then \mathcal{T} does not self-assemble $R_{k,N}^3$.

► **Lemma 5.** Assume: \mathcal{T} is a directed, 3D TAS, $k, N \in \mathbb{Z}^+$, $s \subseteq R_{k,N}^3$ is a simple path, in the full grid graph of $R_{k,N}^3$, from the location of the seed of \mathcal{T} to some location in the furthest extreme column of $R_{k,N}^3$, $\vec{\alpha}$ is a \mathcal{T} -assembly sequence that follows s , w and w' are windows, such that, $\vec{\Delta} \neq \vec{0}$ is a vector satisfying $w' = w + \vec{\Delta}$, and e is an odd number satisfying $1 \leq e < 2k$. If $M = M_{\vec{\alpha}, w} \upharpoonright s = (\vec{v}_1, g_1), \dots, (\vec{v}_{2e}, g_{2e})$ and $M' = M_{\vec{\alpha}, w'} \upharpoonright s = (\vec{v}'_1, g'_1), \dots, (\vec{v}'_{2e}, g'_{2e})$ are sufficiently similar non-empty restricted glue window submovies, then \mathcal{T} does not self-assemble $R_{k,N}^3$.

See Figures 12 and 13 for a 2D example of Lemma 5. We now give some notation that will be useful for proving Lemma 5. The definitions and notation in the following paragraph are inspired by notation that first appeared in [17].



■ **Figure 12** A 2D example of the hypothesis of Lemma 5 for $k = 10$ and $e = 9$. Since the example is 2D, we use $R_{k,N} = \{0, 1, \dots, N - 1\} \times \{0, 1, \dots, k - 1\}$, rather than $R_{k,N}^3$. Note that $\vec{\alpha}$ follows a simple path s from the location of σ to a location in the furthest extreme column. The restricted glue window movies are sufficiently similar because their glues are at the same locations (up to translation), oriented in the same direction (away or toward σ), and each pair of glues that are placed by $\vec{\alpha}$ at an “away crossing” of one of the windows is equal to its translated counterpart in the other window, e.g., the two topmost glues that touch w and w' are both light gray. The same constraint holds for all glue pairs shown with a solid shade of gray or a striped pattern. On the other hand, the glues adjacent to w' that are placed by $\vec{\alpha}$ at a “toward crossing”, for example g'_{11} and g'_{12} , are decorated with a letter in order to represent the fact that we do not assume that these glues are equal to their translated counterparts that touch w (i.e., g_{15} and g_{16}).



■ **Figure 13** A 2D example of the conclusion of Lemma 5, corresponding to example of the hypothesis from Figure 12. Given the fact that \mathcal{T} is directed and the way $\vec{\beta}$ is defined, every pair of glues that touch w must be equal to the corresponding pair of glues that touch w' (if any). Thus, e.g., the glue pairs labelled b and h in Figure 12 must really be equal to the glue pairs a and g , respectively. After $\vec{\beta}$ places a tile at location \vec{v}'_{17} , it will mimic how $\vec{\alpha}$ got from \vec{v}_{18} to the tile in the extreme column of $R_{k,N}$, as depicted in Figure 12. Since $\vec{\Delta} \neq \vec{0}$, this always results in at least one tile placement outside of $R_{k,N}$. In this example, β also happens to exit $R_{k,N}$ earlier in its assembly sequence, i.e., in the sub-path from \vec{v}'_{14} to \vec{v}'_{15} .

4:14 Tile Complexity of Uniquely Self-Assembling Thin Rectangles

For a \mathcal{T} -assembly sequence $\vec{\alpha} = (\alpha_i \mid 0 \leq i < l)$, we write $|\vec{\alpha}| = l$. We write $\vec{\alpha}[i]$ to denote $\vec{x} \mapsto t$, where \vec{x} and t are such that $\alpha_{i+1} = \alpha_i + (\vec{x} \mapsto t)$. We write $\vec{\alpha}[i] + \vec{\Delta}$, for some vector $\vec{\Delta}$, to denote $(\vec{x} + \vec{\Delta}) \mapsto t$. If $\alpha_{i+1} = \alpha_i + (\vec{x} \mapsto t)$, then we write $Pos(\vec{\alpha}[i]) = \vec{x}$ and $Tile(\vec{\alpha}[i]) = t$. Assuming $|\vec{\alpha}| > 0$, the notation $\vec{\alpha} = \vec{\alpha} + (\vec{x} \mapsto t)$ denotes a *tile placement step*, namely the sequence of configurations $(\alpha_i \mid 0 \leq i < l + 1)$, where α_l is the configuration satisfying, $\alpha_l(\vec{x}) = t$ and for all $\vec{y} \neq \vec{x}$, $\alpha_l(\vec{y}) = \alpha_{l-1}(\vec{y})$. Note that the “+” in a tile placement step is different from the “+” used in the notation “ $\beta = \alpha + (\vec{p} \mapsto t)$ ”. However, since the former operates on an assembly sequence, it should be clear from the context which operator is being invoked. The definition of a tile placement step does not require that the sequence of configurations be a \mathcal{T} -assembly sequence. After all, the tile placement step $\vec{\alpha} = \vec{\alpha} + (\vec{x} \mapsto t)$ could be attempting to place a tile at a location that is not even adjacent to (a location in the domain of) α_{l-1} . Or, it could be attempting to place a tile at a location that is in the domain of α_{l-1} , which means a tile has already been placed at \vec{x} . So we say that such a tile placement step is *correct* if $(\alpha_i \mid 0 \leq i < l + 1)$ is a \mathcal{T} -assembly sequence. If $|\vec{\alpha}| = 0$, then $\vec{\alpha} = \vec{\alpha} + (\vec{x} \mapsto t)$ results in the \mathcal{T} -assembly sequence (α_0) , where α_0 is the assembly such that $\alpha_0(\vec{x}) = t$ and $\alpha_0(\vec{y})$ is undefined at all other locations $\vec{y} \neq \vec{x}$.

■ **Algorithm 1** The algorithm for $\vec{\beta}$.

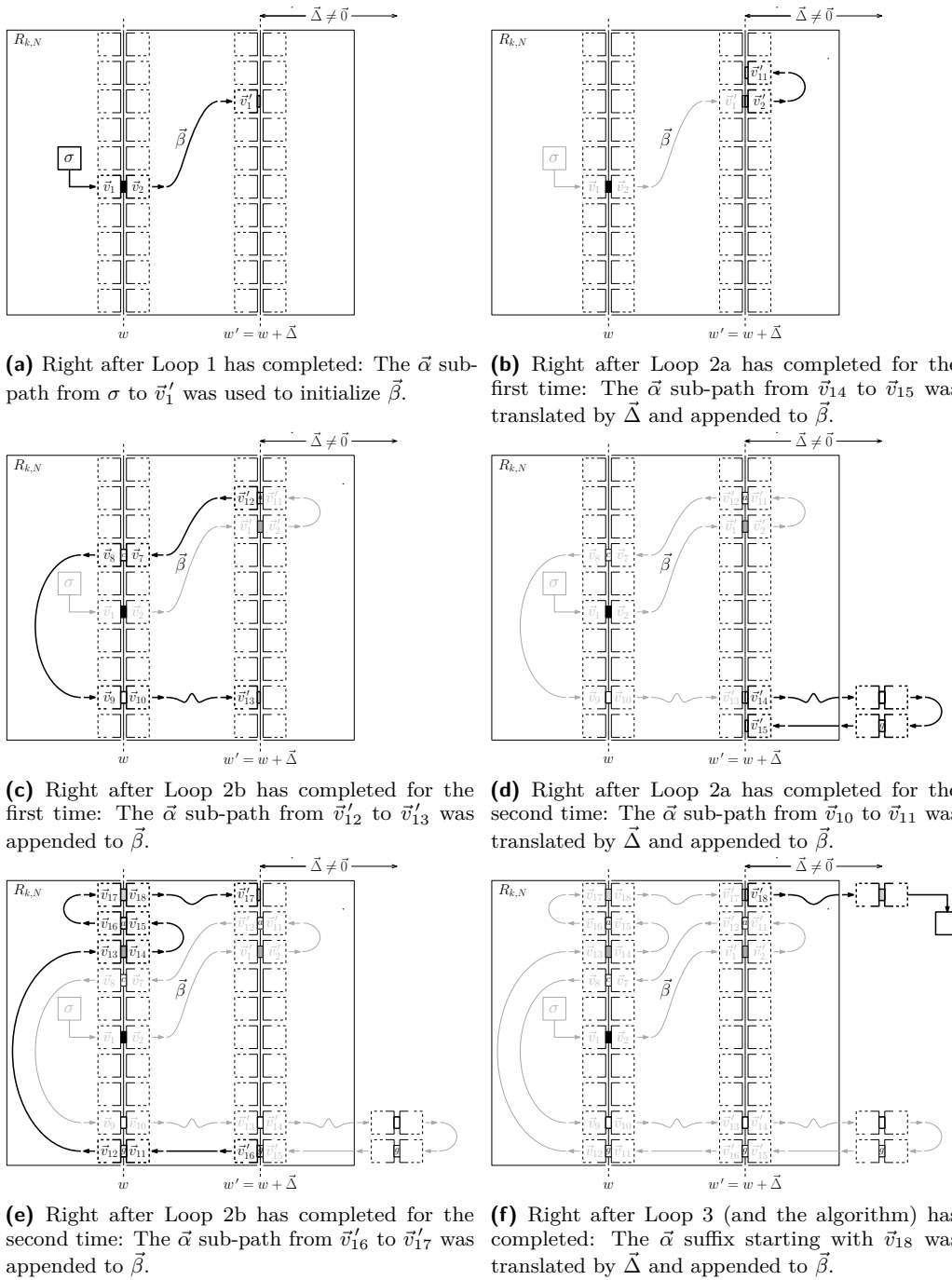
```

1 Initialize  $j = 1$ ,  $n = 0$  and  $\vec{\beta} = ()$ 
2 while  $Pos(\vec{\alpha}[n]) \neq \vec{v}'_{4j-2}$  do /* Loop 1 */
3    $\vec{\beta} = \vec{\beta} + \vec{\alpha}[n]$ 
4    $n = n + 1$ 
5 while  $\vec{v}'_{4j-2} \neq \vec{v}_{2e} + \vec{\Delta}$  do /* Loop 2 */
6   Let  $i$  be such that  $4i - 2$  is the index of  $\vec{v}'_{4j-2} - \vec{\Delta}$  in  $M$ 
7   Let  $n$  be such that  $Pos(\vec{\alpha}[n]) = \vec{v}_{4i-2}$ 
8   while  $Pos(\vec{\alpha}[n]) \neq \vec{v}_{4i}$  do /* Loop 2a */
9      $\vec{\beta} = \vec{\beta} + (\vec{\alpha}[n] + \vec{\Delta})$ 
10     $n = n + 1$ 
11   Let  $j'$  be such that  $4j'$  is the index of  $\vec{v}_{4i} + \vec{\Delta}$  in  $M'$ 
12   Let  $n$  be such that  $Pos(\vec{\alpha}[n]) = \vec{v}'_{4j'}$ 
13   while  $Pos(\vec{\alpha}[n]) \neq \vec{v}'_{4j'+2}$  do /* Loop 2b */
14      $\vec{\beta} = \vec{\beta} + \vec{\alpha}[n]$ 
15      $n = n + 1$ 
16    $j = j' + 1$ 
17 Let  $n$  be such that  $Pos(\vec{\alpha}[n]) = \vec{v}_{2e}$ 
18 while  $n < |\vec{\alpha}|$  do /* Loop 3 */
19    $\vec{\beta} = \vec{\beta} + (\vec{\alpha}[n] + \vec{\Delta})$ 
20    $n = n + 1$ 
21 return  $\vec{\beta}$ 

```

The proof of Lemma 5 relies on Algorithm 1 that uses $\vec{\alpha}$ to construct a new assembly sequence $\vec{\beta}$ such that the tile placement steps by $\vec{\beta}$ on the far side of w' from the seed mimic a (possibly strict) subset of the tile placements by $\vec{\alpha}$ on the far side of w from the seed.

When $\vec{\beta}$ is on the near side of w' to the seed, it mimics $\vec{\alpha}$, although $\vec{\beta}$ does not necessarily mimic every tile placement by $\vec{\alpha}$ on the near side of w' to the seed. When $\vec{\beta}$ crosses w' , going away from the seed, by placing tiles at \vec{v}'_{4j-3} and \vec{v}'_{4j-2} in this order, then the tile it places



■ **Figure 14** The trace of Algorithm 1 when applied to the assembly sequence $\vec{\alpha}$ shown in Figure 12. In each sub-figure, the new sub-path is bolded and is a continuation of the sub-path in the previous one. The last sub-figure above shows the same assembly sequence $\vec{\beta}$ depicted in Figure 13.

at \vec{v}'_{4j-2} is of the same type as the tile that $\vec{\alpha}$ places at $\vec{v}_{4i-2} = \vec{v}'_{4j-2} - \vec{\Delta}$. After $\vec{\beta}$ crosses w' by placing a tile at \vec{v}'_{4j-2} , $\vec{\beta}$ places tiles that $\vec{\alpha}$ places along s from \vec{v}_{4i-2} to \vec{v}_{4i-1} , but the tiles $\vec{\beta}$ places are translated to the far side of w' from the seed. When $\vec{\beta}$ is about to cross w' , going toward the seed, by placing a tile at \vec{v}'_{4j-1} , then, since \mathcal{T} is directed, the type of tile that it places at this location is equal to the type of tile that $\vec{\alpha}$ places at \vec{v}'_{4j-1} . This means that $\vec{\beta}$ may continue to follow s but starting from \vec{v}'_{4j} . Eventually, $\vec{\beta}$ will finish crossing w' going away from the seed for the last time by placing a tile at $\vec{v}_{2e} + \vec{\Delta}$. Then, $\vec{\beta}$ places tiles that $\vec{\alpha}$ places along s , starting from \vec{v}_{2e} , but the tiles that $\vec{\beta}$ places are translated to the far side of w' from the seed. Since $\vec{\Delta} \neq \vec{0}$, $\vec{\beta}$ will ultimately place a tile that is not in $R_{k,N}^3$, which means \mathcal{T} does not self-assemble $R_{k,N}^3$.

We illustrate the behavior of this algorithm in Figure 14, where we apply it to the assembly sequence $\vec{\alpha}$ shown in Figure 12. The full proof of Lemma 5 is omitted from this version of the paper. The following result combines Lemmas 4 and 5 and we will use its contrapositive to prove our main lower bound.

► **Lemma 6.** *Assume: $\mathcal{T} = (T, \sigma, 1)$ is a 3D TAS, G is the set of all glues in T , $k, N \in \mathbb{Z}^+$, $s \subseteq R_{k,N}^3$ is a simple path, in the full grid graph of $R_{k,N}^3$, from the location of σ to some location in the furthest extreme column of $R_{k,N}^3$, $\vec{\alpha}$ is a \mathcal{T} -assembly sequence that follows s , $m \in \mathbb{Z}^+$, for all $1 \leq l \leq m$, w_l is a window, for all $1 \leq l < l' \leq m$, $\vec{\Delta}_{l,l'} \neq \vec{0}$ satisfies $w_{l'} = w_l + \vec{\Delta}_{l,l'}$, and for all $1 \leq l \leq m$, there is an odd $1 \leq e_l < 2k$ such that $M_{\vec{\alpha}, w_l} \upharpoonright s$ is a non-empty restricted glue window submovie of length $2e_l$. If $m > |G|^k \cdot k \cdot 16^k$, then \mathcal{T} does not self-assemble $R_{k,N}^3$.*

The proof of Lemma 6 is omitted from this version of the paper. Here is our main lower bound:

► **Theorem 2.** $K_{USA}^1(R_{k,N}^3) = \Omega(N^{\frac{1}{k}})$.

Proof. Assume $\mathcal{T} = (T, \sigma, \tau = 1)$ is a directed, 3D TAS that self-assembles $R_{k,N}^3$. Assume $\alpha \in \mathcal{A}_{\square}[\mathcal{T}]$ with $\text{dom } \alpha = R_{k,N}^3$. Let $s = (\vec{x}_0, \vec{x}_1, \dots, \vec{x}_m)$ be a simple path in G_{α}^b , such that, $\vec{x}_0 = \text{dom } \sigma$ and \vec{x}_m is in the furthest extreme (westernmost or easternmost) column of $R_{k,N}^3$ from the location of σ , in either z plane. Since $\tau = 1$, there is a \mathcal{T} -assembly sequence $\vec{\alpha}$ that follows s . Assume $N \geq 3$. Since s is a simple path from the location of the seed to some location in the furthest extreme column of $R_{k,N}^3$, in either z plane, there is some positive integer $m \geq \lfloor \frac{N}{2} \rfloor \geq \frac{N}{3}$ such that, for all $1 \leq l \leq m$, w_l is a window that cuts $R_{k,N}^3$, for all $1 \leq l < l' \leq m$, there exists $\vec{\Delta}_{l,l'} \neq \vec{0}$ satisfying $w_{l'} = w_l + \vec{\Delta}_{l,l'}$, and for each $1 \leq l \leq m$, there exists a corresponding odd number $1 \leq e_l < 2k$ such that $M_{\vec{\alpha}, w_l} \upharpoonright s$ is a non-empty restricted glue window submovie of length $2e_l$. Since \mathcal{T} self-assembles $R_{k,N}^3$, (the contrapositive of) Lemma 6 says that $m \leq |G|^k \cdot k \cdot 16^k$. We also know that $\frac{N}{3} \leq m$, which means that $\frac{N}{3} \leq |G|^k \cdot k \cdot 16^k$. Thus, we have $N \leq 3 \cdot |G|^k \cdot k \cdot 16^k$ and it follows that $|T| \geq \frac{|G|}{6} \geq \frac{1}{6} \frac{N^{\frac{1}{k}}}{(3 \cdot k \cdot 16^k)^{\frac{1}{k}}} \geq \frac{1}{6} \frac{N^{\frac{1}{k}}}{(3^k \cdot 2^k \cdot 16^k)^{\frac{1}{k}}} = \frac{1}{6} \frac{N^{\frac{1}{k}}}{96} = \Omega(N^{\frac{1}{k}})$. ◀

Lemma 5, upon which our proof of Theorem 2 crucially depends (via Lemma 6), assumes that \mathcal{T} is directed. If \mathcal{T} is not assumed to be directed, then it is possible to construct an undirected 3D TAS \mathcal{T} that satisfies all the other conditions of the hypothesis of Lemma 5, but \mathcal{T} self-assembles $R_{k,N}^3$. The full construction of such an undirected \mathcal{T} is omitted from this version of the paper.

5 Conclusion

In this paper, we gave improved lower and upper bounds on $K_{USA}^1(R_{k,N}^3)$, namely $\Omega\left(N^{\frac{1}{k}}\right)$ and $O\left(N^{\frac{1}{k-1}} + \log N\right)$. We leave open the question of determining tight bounds for $K_{USA}^1(R_{k,N}^3)$ as well as for $K_{SA}^1(R_{k,N}^3)$.

References

- 1 Leonard M. Adleman, Qi Cheng, Ashish Goel, and Ming-Deh A. Huang. Running time and program size for self-assembled squares. In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing (STOC)*, pages 740–748, 2001.
- 2 Gagan Aggarwal, Qi Cheng, Michael H. Goldwasser, Ming-Yang Kao, Pablo Moisset de Espanés, and Robert T. Schweller. Complexities for generalized models of self-assembly. *SIAM Journal on Computing (SICOMP)*, 34:1493–1515, 2005.
- 3 Matthew Cook, Yunhui Fu, and Robert T. Schweller. Temperature 1 self-assembly: Deterministic assembly in 3D and probabilistic assembly in 2D. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 570–589, 2011.
- 4 Erik D. Demaine, Martin L. Demaine, Sándor P. Fekete, Mashhood Ishaque, Eynat Rafalin, Robert T. Schweller, and Diane L. Souvaine. Staged self-assembly: nanomanufacture of arbitrary shapes with $O(1)$ glues. *Natural Computing*, 7(3):347–370, 2008. doi:10.1007/s11047-008-9073-0.
- 5 David Doty, Matthew J. Patitz, Dustin Reishus, Robert T. Schweller, and Scott M. Summers. Strong fault-tolerance for self-assembly with fuzzy temperature. In *Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science (FOCS 2010)*, pages 417–426, 2010.
- 6 David Doty, Matthew J. Patitz, and Scott M. Summers. Limitations of self-assembly at temperature 1. *Theoretical Computer Science*, 412:145–158, 2011.
- 7 Sándor P. Fekete, Jacob Hendricks, Matthew J. Patitz, Trent A. Rogers, and Robert T. Schweller. Universal computation with arbitrary polyomino tiles in non-cooperative self-assembly. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 148–167, 2015.
- 8 Bin Fu, Matthew J. Patitz, Robert T. Schweller, and Robert Sheline. Self-assembly with geometric tiles. In *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part I*, pages 714–725, 2012.
- 9 David Furcy, Samuel Micka, and Scott M. Summers. Optimal program-size complexity for self-assembled squares at temperature 1 in 3D. *Algorithmica*, 77(4):1240–1282, 2017.
- 10 David Furcy and Scott M. Summers. Optimal self-assembly of finite shapes at temperature 1 in 3D. *Algorithmica*, 80(6):1909–1963, 2018.
- 11 David Furcy, Scott M. Summers, and Christian Wendlandt. Self-assembly of and optimal encoding within thin rectangles at temperature-1 in 3D. *Theoretical Computer Science*, 872:55–78, 2021.
- 12 Oscar Gilbert, Jacob Hendricks, Matthew J. Patitz, and Trent A. Rogers. Computing in continuous space with self-assembling polygonal tiles (extended abstract). In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 937–956, 2016.
- 13 Jacob Hendricks, Matthew J. Patitz, Trent A. Rogers, and Scott M. Summers. The power of duples (in self-assembly): It’s not so hip to be square. *Theoretical Computer Science*, 743:148–166, 2018.
- 14 James I. Lathrop, Jack H. Lutz, and Scott M. Summers. Strict self-assembly of discrete Sierpinski triangles. *Theoretical Computer Science*, 410:384–405, 2009.

- 15 Jack H. Lutz and Brad Shuttters. Approximate self-assembly of the sierpinski triangle. *Theory Comput. Syst.*, 51(3):372–400, 2012.
- 16 Ján Manuch, Ladislav Stacho, and Christine Stoll. Two lower bounds for self-assemblies at temperature 1. *Journal of Computational Biology*, 17(6):841–852, 2010.
- 17 P.-E. Meunier, M. J. Patitz, S. M. Summers, G. Theyssier, A. Winslow, and D. Woods. Intrinsic universality in tile self-assembly requires cooperation. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 752–771, 2014.
- 18 Pierre-Étienne Meunier, Damien Regnault, and Damien Woods. The program-size complexity of self-assembled paths. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 727–737, 2020.
- 19 Pierre-Étienne Meunier and Damien Woods. The non-cooperative tile assembly model is not intrinsically universal or capable of bounded turing machine simulation. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 328–341, 2017.
- 20 Matthew J. Patitz, Robert T. Schweller, and Scott M. Summers. Exact shapes and Turing universality at temperature 1 with a single negative glue. In *Proceedings of the 17th international conference on DNA computing and molecular programming, DNA’11*, pages 175–189, Berlin, Heidelberg, 2011. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=2042033.2042050>.
- 21 Paul W. K. Rothmund. *Theory and Experiments in Algorithmic Self-Assembly*. PhD thesis, University of Southern California, December 2001.
- 22 Paul W. K. Rothmund and Erik Winfree. The program-size complexity of self-assembled squares (extended abstract). In *The Thirty-Second Annual ACM Symposium on Theory of Computing (STOC)*, pages 459–468, 2000.
- 23 Nadrian C. Seeman. Nucleic-acid junctions and lattices. *Journal of Theoretical Biology*, 99:237–247, 1982.
- 24 David Soloveichik and Erik Winfree. Complexity of self-assembled shapes. *SIAM Journal on Computing (SICOMP)*, 36(6):1544–1569, 2007.
- 25 Hao Wang. Proving theorems by pattern recognition – II. *The Bell System Technical Journal*, XL(1):1–41, 1961.
- 26 Erik Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, California Institute of Technology, June 1998.

ENSnano: A 3D Modeling Software for DNA Nanostructures

Nicolas Levy  

École Normale Supérieure de Lyon, LIP (UMR 5668, Équipe MC2), France

Nicolas Schabanel  

CNRS, École Normale Supérieure de Lyon, LIP (UMR 5668, Équipe MC2) and IXXI, France

Abstract

Since the 1990s, increasingly complex nanostructures have been reliably obtained out of self-assembled DNA strands: from “simple” 2D shapes to 3D gears and articulated nano-objects, and even computing structures. The success of the assembly of these structures relies on a fine tuning of their structure to match the peculiar geometry of DNA helices. Various softwares have been developed to help the designer. These softwares provide essentially four kind of tools: an abstract representation of DNA helices (e.g. *cadnano*, *scadnano*, *DNApen*, *3DNA*, *Hex-tiles*); a 3D view of the design (e.g., *vHelix*, *Adenita*, *oxDNAviewer*); fully automated design (e.g., *BScOR*, *Daedalus*, *Perdix*, *Talos*, *Athena*), generally dedicated to a specific kind of design, such as wireframe origami; and coarse grain or thermodynamical physics simulations (e.g., *oxDNA*, *MrDNA*, *SNUPI*, *Nupack*, *ViennaRNA*,...). *MagicDNA* combines some of these approaches to ease the design of configurable DNA origamis.

We present our first step in the direction of conciliating all these different approaches and purposes into one single reliable GUI solution: the first fully usable version (design from scratch to export) of our general purpose 3D DNA nanostructure design software **ENSnano**. We believe that its intuitive, swift and yet powerful graphical interface, combining 2D and 3D editable views, allows fast and precise editing of DNA nanostructures. It also handles editing of large 2D/3D structures smoothly, and imports from the most common solutions. Our software extends the concept of *grids* introduced in *cadnano*. Grids allow to abstract and articulated the different parts of a design. **ENSnano** also provides new design tools which speeds up considerably the design of complex large 3D structures, most notably: a *2D split view*, which allows to edit intricate 3D structures which cannot easily be mapped in a 2D view, and a *copy, paste & repeat* functionality, which takes advantage of the grids to design swiftly large repetitive chunks of a structure. **ENSnano** has been validated experimentally, as proven by the AFM images of a DNA origami entirely designed in **ENSnano**.

ENSnano is a *light-weight ready-to-run independent single-file* app, running seamlessly in most of the operating systems (Windows 10, MacOS 10.13+ and Linux). Precompiled versions for Windows and MacOS are ready to download on **ENSnano** website. As of writing this paper, our software is being actively developed to extend its capacities in various directions discussed in this article. Still, its 3D and 2D editing interface is already meeting our usability goals. Because of its stability and ease of use, we believe that **ENSnano** could already be integrated in anyone’s design chain, when precise editing of a larger nanostructure is needed.

2012 ACM Subject Classification Computer systems organization → Molecular computing; Computing methodologies → Molecular simulation; Applied computing → Molecular structural biology

Keywords and phrases Software, DNA nanostructure, Molecular design, molecular self-assembly

Digital Object Identifier 10.4230/LIPIcs.DNA.27.5

Supplementary Material *Software (Source Code)*: <https://github.com/thenlevy/ensnano>

archived at `swh:1:dir:0569f133306e7972335c4600d2be5794f467c0ba`

Funding *Nicolas Schabanel*: this work was supported in part by ENSL emergence “Algorithmes en ADN”, CNRS MITI “NoPrExProgMol”, “AMARP” and “Scalable DNA algorithms”, and CNRS INS2I “Algadène” grants.

Acknowledgements We want to thank Damien Woods, Pierre-Étienne Meunier, Pierre Marcus, Octave Hazard, Constantine Evans, Trent Rogers, and Dave Doty for fruitful discussions about this project. We would also like to thanks the students who followed the lecture CR11 on DNA computing at the ÉNS de Lyon in 2020 for their contribution to the rocket design in **ENSnano**.



© Nicolas Levy and Nicolas Schabanel;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on DNA Computing and Molecular Programming (DNA 27).

Editors: Matthew R. Lakin and Petr Šulc; Article No. 5; pp. 5:1–5:23

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

DNA nanostructures design. Since the 1990s, increasingly complex nanostructures have been reliably obtained out of self-assembled DNA strands: from “simple” 2D shapes [2, 26] to 3D gears and articulated nano-objects (e.g. [38, 8]) and even computing structures [27, 28, 35]. The success of the assembly of these structures relies on a fine tuning of their structure to match the peculiar geometry of DNA helices. Various softwares have been developed to help the designer.

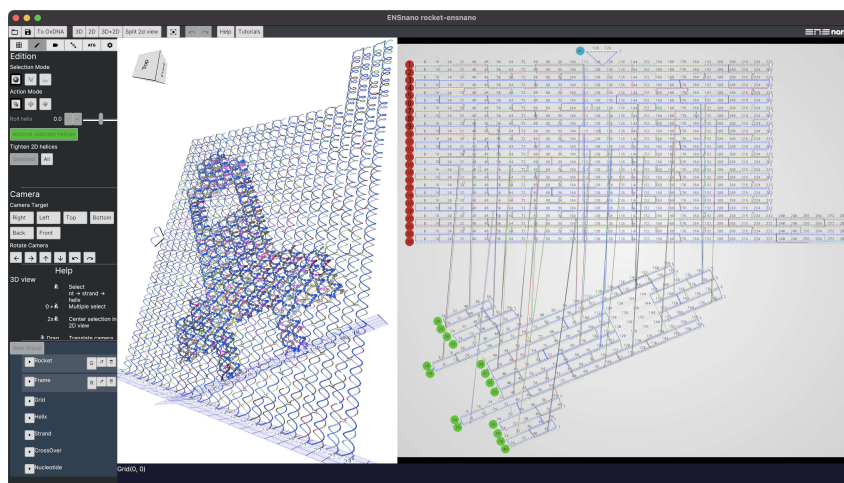
A tour of existing softwares. These softwares provides essentially four kind of tools:

- an *abstract representation* of DNA helices, e.g.: cadnano [5], scadnano [4], DNAPen [10], 3DNA [11], Hex-tiles [23];
- a *3D representation* of the design, e.g.: vHelix [1], Tiamat [32], Adenita [3], oxDNAviewer [25];
- a *fully automated design*, e.g.: BScOR [1], Daedalus [30], Perdix, Talos, Athena [16, 14, 15]. These are generally dedicated to a specific kind of design, such as wireframe origami;
- *coarse grain or thermodynamic physics simulation*, e.g.: oxDNA [6], MrDNA [21], Snupi [18], Nupack [37], ViennaRNA [20],...

Recently, the software MagicDNA [12] proposed to combine some of these approaches to ease the design of configurable DNA origamis. The recent article [9] presents a very detailed survey of all these solutions with their pros and cons. Here is a list of the features of interest found in these softwares:

- *High-level description:* DNA geometry is very peculiar and hard to grasp. The seminal cadnano interface allowed to abstract from it and focus on the relevant aspects of DNA helices: it provides a framework to place DNA helices next to each other, as well as guides to connect them using crossovers, together with an easily-understandable and printable blueprint of the resulting design. In spite of its imperfection in terms of DNA geometry (incorrect turn/bp [34] and absence of roll of helices), cadnano is still the reference for fine tuning a design.
- *Live (editable) 3D view:* With the raise of 3D complex designs involving DNA helices pointing in various directions or even making curves, various softwares proposed 3D rendering of a design. A notable software in this category is Tiamat [32] which introduced a 3D interface capable of fine tuning and editing of the design. Unfortunately, 3D views are not always the best solution to edit a design beyond placing properly DNA helices in space. Indeed, 3D views are often jammed, even for simple design, and the 3D→2D parallax effect makes it hard to evaluate the length of a crossover, especially because its direction, and thus its apparent length, changes when it is moved around.
- *Fine tuning and editing:* As mentioned above, 3D views are not well adapted for editing or fine tuning [9], especially when the 3D view is provided by a “mother” software (Matlab, Maya, or Samson) which was not designed for the specific needs of DNA nanostructures. As it turns out, almost all the softwares, including the all-automated ones, e.g. [12], recommend to export to (s)cadnano for checking and fine tuning their designs. As a matter of fact, the 2D representation of a helix as a double array, initially proposed in cadnano, remains the most practical for editing. Rotating the array representations as in [38, 4] allows an even more convenient rendering of the design. 3D complex designs however cannot be faithfully mapped to 2D, and, no matter what, some helices that are close to each other in 3D, will be mapped at distant locations in a 2D representation. As a result, many crossovers overlay the design and make it confusing to read and edit.

- *Design versatility:* Most softwares focus on a specific class of design: DNA origami, wireframe origami or DNA tiles (SST). Only a few provide an abstraction for dealing with all of them, e.g.: *MagicDNA* [12] proposes a common framework for classic and wireframe DNA origami. The most versatile remain (s)cadnano (for 2D editing) and Tiamat (for 3D editing). (s)cadnano make it possible to work with any class of designs, but provides almost no specific automation. It should however be noted that Tiamat comes with an integrated sequence generator.
- *Automation:* When designing a DNA origami, many tasks are repetitive and dull, such as “stapling” a rectangle. On the opposite, some tasks require a high technical level, such as routing a scaffold in a wireframe design, or designing 3D staples in a 3D bulk. Both of these types of tasks benefit a lot from automation. Algorithms have been designed to complete these tasks and designers can save a lot of time and avoid mistakes by using them, e.g. [1, 30, 14, 12].
- *Programmable designs:* Many designs gain to be specified as the output of a program: either because they are very big and repetitive (e.g., SST designs); or because they require specific angles and lengths (e.g. quasi-crystals in [38]); or because they sometimes require many trials-and-errors to be properly configured, for instance to match the length of the holy M13mp18 scaffold strand. To achieve this goal, scadnano [4] and the prototype codenano [7] define a programming framework consisting of various function calls (in python or rust) to describe helices and strands positions. *MagicDNA* [12] proposes an interesting, user-friendlier, alternative approach, generalizing the wireframe approach, which consists in describing the design as graph embedded in space, whose edges are replaced by configurable chunks of parallel helices, along which the routing of the scaffold and its staples is algorithmically computed.
- *Simulation:* 3D views of a design are only a wishful representation of the design, as strands may not self-assemble as foreseen by the designer. If 3D views are essential to choose the right crossovers to bind strands together in order to achieve the desired assembly, they offer no guarantee in the resulting shape. Coarse grained physics simulation [6, 21, 18] and thermodynamic binding estimation [37, 20] softwares allow a much more precise feedback on the feasibility and stability of a design. Their predictions are now considered good enough to demonstrate the validity of a design, e.g. [12]. However, they are computer-intensive and furthermore require a high level of competency to interact with. They can hardly produce a fast-enough feedback to be use during the design process. They are thus usually used at the end of the design chain. Design softwares usually offer to export the design into the sophisticated file format of these simulation softwares for validation.
- *Intuitive fast-responding interface:* Various directions have been explored for designing a adequate graphic user interface for designing DNA nanostructure. As mentioned earlier, 3D views did not improve, and in many cases, arguably deteriorated the usability of the interface. This is particularly true when the software is embedded in a mother software in charge of the 3D rendering, which offers, most of the time, only slow or little-to-no editing capacities. As a consequence, (s)cadnano remains the most practical interface so far and this is no wonder it is still intensively used, in spite of its limited ergonomics.
- *Reliability:* Designing complex DNA nanostructures requires focused attention and unfortunate failures or slowness in softwares add considerable stress to the designer. Software reliability can only be ensured by developing them in a *safe* programming language, that is, whose compiler imposes rigorous safeguards to the programmers and checks their code in depth to avoid as much as possible bugs at runtime. Untyped programming languages should thus be ruled out to develop reliable design softwares.



■ **Figure 1 ENSnano interface:** the 3D and 2D views of a design.

- *Distribution:* The most commonly cited softwares, namely `cadnano`, `oxDNA`, `oxDNAviewer`, `MrDNA`, `Nupack` and `ViennaRNA`, are the easiest to install (they are distributed either as an independent, python packages or web-based app), and are available for the most commonly used operating systems (Windows, MacOS and Linux). We have had varying experience with installing and using softwares embedded into a generic-purpose mother application. Cross-platform distribution, either as an independent or a web-based app, seems thus to be an important usability criteria.

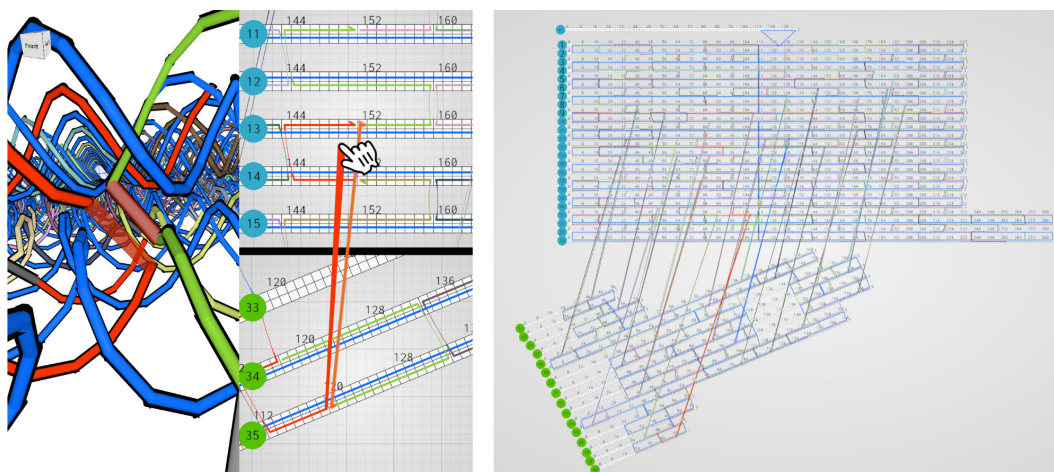
Our contribution. In this paper, we present our software ENSnano which introduces a 3D editable view *working together* with a (s)cadnano-like 2D view. As in `cadnano`, our helices are attached to *grids* organizing the helices, next to each other, in parallel subsets. We extend this grid concept in three ways:

- first, the grids are now freely and precisely placed and oriented in space in the 3D view;
- second, grids can be any 1D or 2D lattice, for now: the classic square and hexagonal grids, but also circular lattices to design nanotubes;
- third, we take advantage of this lattice structure to introduce a *geometry-aware copy-paste- \mathcal{E} -repeat* process.

► **2D and 3D live views working together.** 2D and 3D views are illustrated in Fig. 1. The main purpose of the 3D view is to position the grids and helices, from which the coordinates of each nucleotide is deduced. These coordinates are used to suggest crossover positions (see Sec. 3.5). Crossovers can be created indifferently in the 2D or the 3D view. The 3D view provides also indications on the length of the crossovers: crossovers of excessive length are highlighted in black. The 2D view is pretty similar to the standard (s)cadnano view with three important differences:

- the design of the strands in the 2D view has been drastically simplified;
- the 2D view can be freely organized with few mouse clicks (see Sec. 3.3);
- the 2D view can be split into two *interacting* views focusing on different parts of the 2D map of the design, as explained next.

The 2D and 3D views are synchronized: modifications of the designs made in one interface is immediately visible in the other one. Moreover hovering a design element (strand, helix or nucleotide) with the mouse cursor in one interface will highlight it in both views, which makes the correspondence between the two representations easier to grasp.



■ **Figure 2 Editing with the 2D split view.** (left) Building a crossover between “2D-distant” helices by dragging the mouse from one split view to the other. (right) In a single 2D view, the zoom factor required to see both ends of the crossover (highlighted in red) would be so small that precise editing would be impractical.

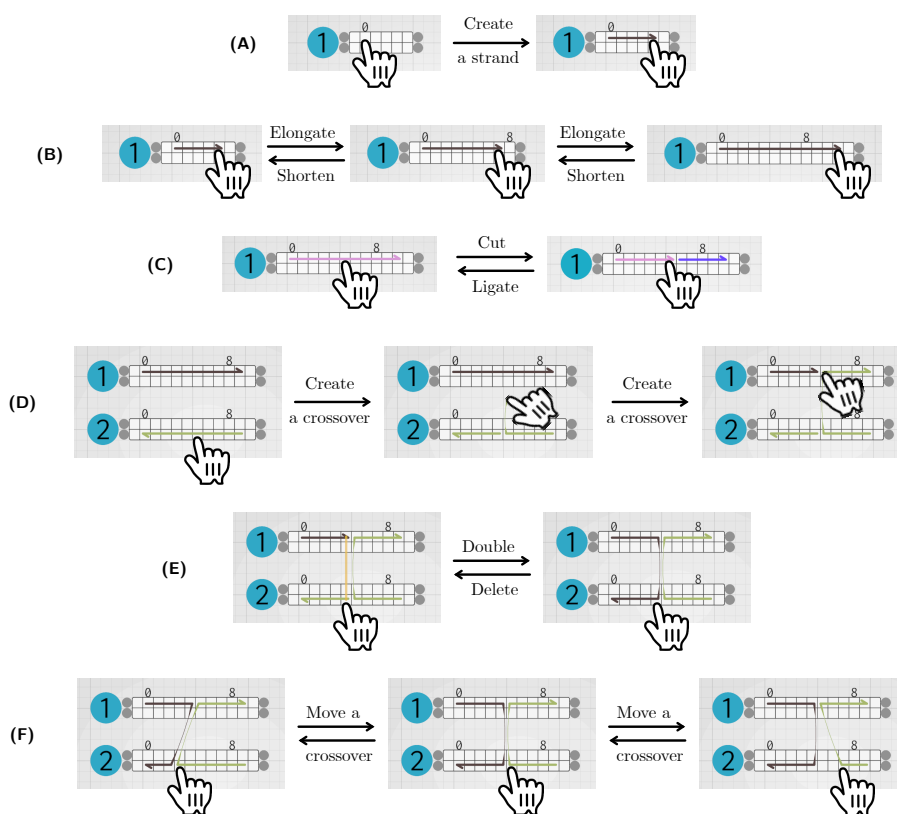
► **The 2D split view.** As mentioned above, 3D complex structures cannot be faithfully mapped into 2D: some parts of the design that are close in 3D, will, no matter what, be mapped far apart from each other in the 2D view. The 2D split view of ENSnano allows to edit parts of the design that are mapped far apart from each other in the 2D view, as if they were next to each other, see Fig. 2. This is particularly useful for creating, moving or deleting a crossover between parts that are mapped across the 2D view. Furthermore, the 3D view interacts naturally with this feature by switching automatically to the 2D split view, when we double-left-click on a crossover that binds nucleotides that cannot be shown together in 2D view at its present zoom factor. Double-clicking on an element allows to switch easily back and forth between its locations in the 3D view and 2D view (or the 2D split view, when needed), see Sec. 3.3.

► **Natural swift point-and-click editing in the 2D view.** As shown in Fig. 3, the interface of ENSnano is organized so that most of the editing in the design can be made by clicking directly at the wanted location: the action applied is *deduced* from the *natural mouse movement* associated to the desired action together with the local configuration at the click location (see Sec. 3.3).

► **Geometry aware copy/pasting.** ENSnano introduces a *geometry-aware copy/pasting* of strands and crossovers across the 3D structure, regardless of the actual numbering of the helices and of their arrangement in the 2D view. This allows to build in the blink of an eye a complete set of staples by duplicating repetitively the selected pattern (a subset of strands or crossovers) according to the initial translation, along the helices *and* across the grid lattice, of the firstly pasted copy (see Fig 9 in Section 3.4).

► **File format, import and export.** ENSnano file format is pretty similar to scadnano and codenano. It consists in a human readable and editable json file (see Sec. B). Currently, ENSnano imports files from scadnano and cadnano (with similar limitations as scadnano). ENSnano also exports designs to oxDNA for precise physics simulation.

We do not provide yet any python framework to generate ENSnano designs. However, ENSnano files can be generated by a python program with moderate efforts. This issue will be resolved in an upcoming version of ENSnano.



■ **Figure 3 Swift strand editing in the 2D view.** (A) **Create a strand:** Left click on an empty position and drag. (B) **Extend a domain:** Left click on an end of a domain and drag. Note that the helix automatically extends if needed. (C) **Cut & ligate a strand:** A left click in the middle of strand cuts the strands. A left click on the end of a strand next to another ligates them. (D) **Create a crossover:** A left click and drag up-/down-wards on a strand initiates the creation of a crossover that will bind the initial click position to the position where the mouse is dragged to. (E) **Double/Delete a crossover:** A left click on an unconnected end of a strand next to a crossover will double the crossover. A left click on one end of a crossover will break it. (F) **Move a crossover:** A left click-and-drag at one end of a crossover will move this crossover. Note that the possibly neighboring crossover will be pushed and pulled back during the dragging until the final position is set.

► **Sequences export.** ENSnano is presently fully functional to design and edit precisely complex DNA origami and to export its staple sequences for ordering.

► **DNA parameters.** ENSnano uses the DNA helix physical data collected from [34, 29]. They are presented in appendix in Sec. B.2. They can easily be edited to fit specific needs, directly in ENSnano file format (a `json` text file) as explained in Sec. B.

Distribution and Installation. ENSnano is developed in the high performance safe programming language Rust [22]. We rely on the cross platform libraries `winit`, `wgpu` and `iced` [33, 31, 13] to produce an *identical and highly responsive* interface across all commonly used operating systems. ENSnano is distributed as a *single-file app* for Windows and MacOS, as well as an open source repository, ready to be cloned and compiled, for Linux and the brave ones. It can be downloaded directly from ENSnano website [19].

2 ENSnano concepts

2D and 3D live editable views. ENSnano combines two graphical interfaces to visualize and edit DNA nanostructures, each of them serving different purposes:

- *the 3D view* helps to visualize and arrange precisely the different elements composing the design in space. It enables to navigate inside the nanostructure, to ensure, for instance, that the crossovers that bind it together, are not too short nor too long. It also allows to arrange complex structures where helices are not parallel. It also provides new 3D tools that helps, for instance, to find suitable positions for crossovers between any kind of helices.
- *the 2D view* presents the blueprint of the design in a streamlined manner, similar to the one initially proposed in *cadnano*. This linear representation of the helices is easy to read and arguably more ergonomic for most editing task.

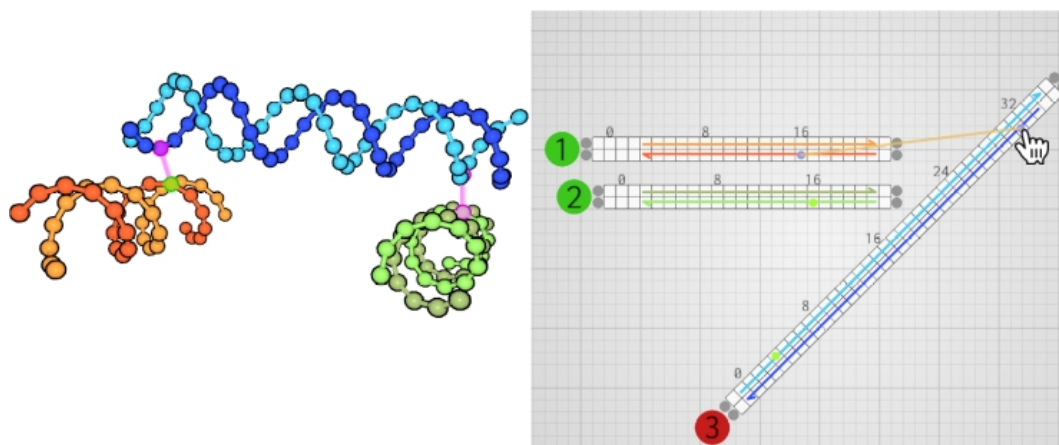
As we will see later, both views work together to facilitate the work of the designer. A particular care was given to provide a homogeneous visual experience for both views, with matching color codes between the views, for instance.

Grids. In ENSnano, like in several other nanostructure design softwares, the strands composing a nanostructure are positioned on helices that are assumed to be rigid shapes – currently, rigid cylinders. The 3D shape of the design is greatly influenced by the relative positions and orientations of these helices. In *cadnano*, the helices are all parallel and their positions are given by a point on a grid. In ENSnano helices are also positioned on a grid, but several grids can coexist in a design, and these grids can have different orientations, which makes it possible to create designs in which all helices point in different directions.

By grouping helices together on a grid, one can organize their design into bulk components made of parallel helices. Each of this component can be thought as a separated *cadnano* design. As pointed out in the introduction, the 2D view is typically more ergonomic to edit those components. Connecting two non-parallel components in a 2D interface is however a challenging task. This is where the 3D editing of crossovers and the crossover suggestions based on the 3D positions of the nucleotides are useful.

At the moment, ENSnano offers squared and hexagonal grids, as well as nanotubes made of 5 to 60 helices. Grids are internally implemented as a mapping $\mathbb{Z}^2 \mapsto \mathbb{R}^2$. This flexible representation makes it easy to add new grid types in the future, and could be easily extended to arbitrary Cayley graphs.

Group-based organization of the design. Elements of the designs can be grouped in named groups. These groups can be used to quickly select one part of the design, or to adjust altogether the properties of the elements in the set. Groups can for instance be used to quickly set the color of all the helices of a groups for crossover suggestion. They can also be used to hide or show temporarily parts of the design. Groups will have extended capabilities in upcoming versions of our software. Note that the group structure in ENSnano does not requires the groups to be disjoint. This is typically useful in the case where one wants to visualize the interface between two components linked together by a set of crossovers. One can create two groups, each of them containing one of the components and the set of crossovers. Using these groups one can chose to hide everything in the design but one of the components and the crossovers at the interface, as illustrated in Fig. A.1.



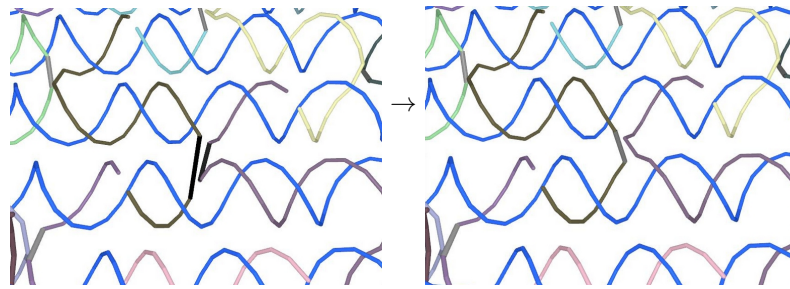
■ **Figure 4 Crossover suggestions.** As none of the three helices are parallel, finding crossover positions may be difficult. We thus assign the helices 1 and 2 (the orange and green stranded) to the green family and helix 3 (the blue stranded) to the red (note that the color family is displayed as the background color of their identifier disc in the 2D view). In the 3D view, the suggested crossovers are indicated by a translucent purple connection between two nucleotides. In the 2d view, the nucleotides that could be bound by a crossover are indicated by pairs of dots of matching color.

Design helpers. When designing a DNA nanostructure, special care must be given when choosing the positions of the crossovers. Indeed, the length and orientation of a crossover between two points of two neighboring helices can vary greatly, because of the spiraling nature of DNA helices. For this reason, DNA nanostructure design softwares often offer a feature that suggests positions at which crossovers between two helices are possible.

► **Crossover suggestions.** In ENSnano, one can assign to each helix a *color family*: none, red or green. Crossover suggestions will be made between helices of color families red and green: purple translucent clues will show up in the 3D view to indicate possible crossover positions, based on the distance between the nucleotides. Dots of matching colors indicates these same locations in the 2D view. Fig. 4 illustrates this feature.

► **Crossover length color shading.** In addition to the suggestion for creating new crossovers, ENSnano also offers visual clues for assessing the length of the existing crossovers. In the 3D interface the crossovers are displayed as cylinders, whose color depends on the distance between their extremities. Short crossovers have the same color as the strand they belong to, while crossover of excessive length are shaded from light grey to dark, where a darker color indicates a longer crossover, see Fig. 5.

► **Helices (auto)roll.** Rolling a helix around its axis shifts its strands forward or backward and thus has a huge impact on the position of its possible crossovers with neighboring helices. Reciprocally, placing some crossovers will have an impact on its optimal roll which will in turn impact all the crossovers around it. Choosing the roll of the helices gives more freedom in a design, and improves its feasibility. *cadnano* for instance ignores this factor. ENSnano enables to either input the value of the helix roll or to run a simple physics simulator which will auto-roll the selected helices according to the torsion forces applied by their crossovers, where each crossover is considered as a spring with free length 0.7nm, the expected length of a crossover. This allows a simple and almost instantaneous sanity check of a design as it is being built. Typical use of the auto-roll feature is when we want a specific crossover between a newly created helix and an other one: we first add this crossover regardless of its length



■ **Figure 5 Length color shading of the crossover in the 3D view. (left)** In this example, several crossovers of various lengths are visible. Crossovers that are short and don't require the designer's attention are displayed in the same color as their strand. Some crossovers are displayed in light-grey indicating that their moderately excessive length may only represents a minor problem in the design. However, one pair of crossovers is displayed in black. This should catch the designer's eyes and indicates that this pair of crossovers should be rearranged. **(right)** After correcting the two faulty crossovers, they are now shorter and appear in a lighter shade. This indicates that the design on the right panel is more likely to be feasible.

and then we auto-roll the newly created helix so that it adjusts its roll to satisfy this desired crossover; the crossover suggestion feature will then adapt and indicate where to place the other crossovers according to the initial one.

Basic stability testing. Dependable physics engines such as oxDNA and MrDNA are computer intensive and cannot provide a fast enough feedback for a user in doubt while building up a large design. As pointed out in [9], these softwares tend to be used at the end of the design chain, to validate a complete design, before ordering the corresponding strands. *In ENSnano, we propose to give up on dependable physics simulations of the resulting shape of a design*, but just to focus on its *immediate stability*. Indeed, the commonly accepted *motto* of DNA nanostructures design is that DNA helices can be abstracted as rigid shapes (cylinder or curves) if correctly tied up together. Dependable physics engines are used to check the correctness of this assumption. But, during the whole design process, we take it for granted, because its correctness relies on it. What we need is thus a fast regular feedback on whether the current (assumed to be) rigid helices are “tied up” satisfyingly to keep the desired shape. For this purpose, ENSnano includes a *very basic* rigid body physics simulator in which:

- each crossover is modelled as a spring with free length 0.7nm, the expected length of a covalent bond;
- each contiguous double-stranded part of a helix is modelled as an independent rigid cylinder;
- each single stranded part of an helix is considered as a chain of isolated points connected by crossovers (modelled as spring as above);
- Brownian motion is emulated by jiggling the isolated points (or not), with an adjustable intensity;
- the stiffness of the spring, ambient friction and mass of the nucleotide can be configured live as the simulation runs; we usually recommend to start with a high friction and to lower it slowly as the simulation progresses.


This very basic model has no other ambition than to provide a quick feedback on the stability of the currently developed design and may not always produce correct results. In particular, we need to review in depth the volume exclusion procedure. Some perfectly valid design may converge to a spaghetti mess.

To get a definitive assessment of the physical viability of their design, users are invited to export it to oxDNA.

3 Graphical User Interface and tools







3.1 Getting started



The interface of ENSnano is designed so that every action is only one click away: editing either grids, helices, strands, crossovers, 5' or 3' ends of domains,... do not require any edition mode switching; the intended action is naturally guessed by the program. The interface is divided in four area (see Fig. 1):








- the top bar consists of the Open/Save/Export buttons, the view selector (3D and/or (split) 2D), the zoom fitting button , the undo/redo buttons, and the help & tutorials buttons;
- the left bar consists of four panels, from top to bottom: the tool panel, the camera shortcuts, the contextual panel, and the organizer;
- the main view(s) represent(s) the design in 2D and/or 3D;
- and the status bar at the bottom is currently essentially unused.




The camera panel gathers buttons that set the camera in standard positions, which is useful to align the design with the axis. The contextual panel displays and allows to edit information on the current selection. It also displays the help when nothing is selected. The organizer allows to group the design into (non-disjoint) sets as already discussed in Sec. 2.

The tool panel is composed of six tabs:

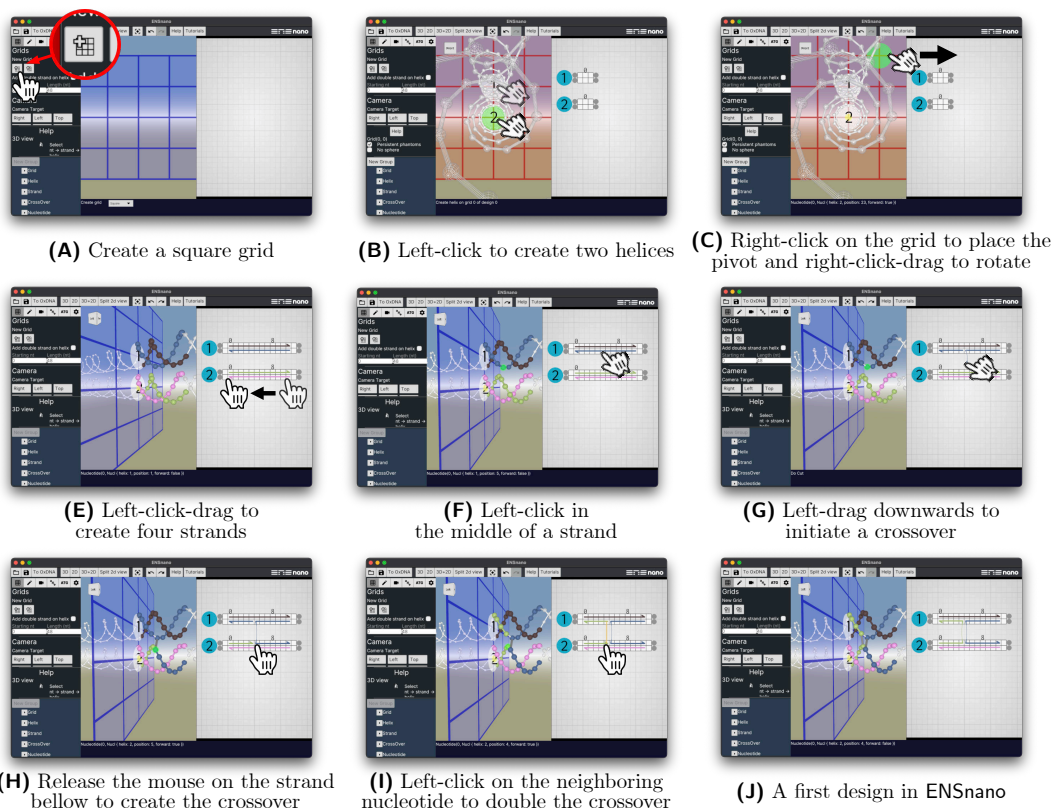
- the grid tab  gathers the tools to create grids and add helices to them;
- the edit tab  gathers the tools to edit nucleotides, strands and helices;
- the camera tab  presents the visualization parameters;
- the rigid body engine tab  presents the (very basic) available physics simulation tools;
- the sequence tab  allows to set and export the sequences of the strands;
- and the parameters tab  allows to change the font size and the scrolling sensitivity.

What is selected or edited when clicking in the views is determined by the selection mode and by the action mode in the two editing tabs  and .

- the selection modes are:  for nucleotides and crossovers,  for strands,  for helices;
- the action modes are:  to edit objects,  to translate objects,  to rotate objects, and  to add helices to a grid.

Grids are added to the design using the buttons:  for square grid,  for hexagonal grid, and  for nanotubes. Helices are added to a grid by clicking on the desired position on the grid. One can chose to equip a helix with a double strand at its creation: just set the starting position and length of the strands in the text fields bellow. By default a phantom helix is displayed when an helix is created, this can be switched off in contextual panel after selecting the grid.


Building a first design. Fig. 6 presents step-by-step how to build a very simple design in ENSnano.








■ **Figure 6** Step-by-step construction of a first design in ENSnano.

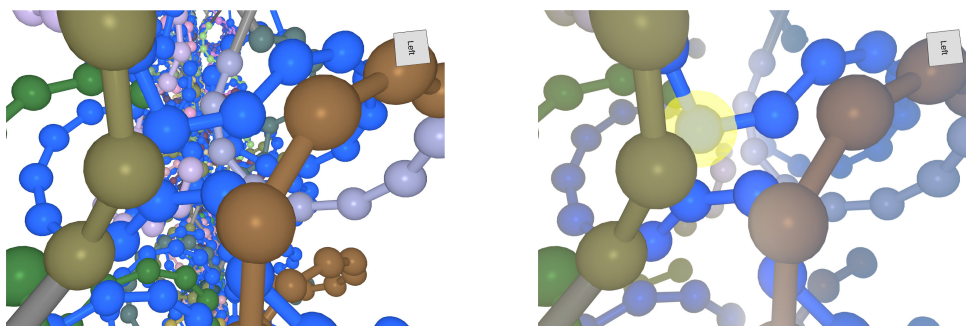
3.2 The 3D view

The main purpose of the 3D view is to visualize and organize the components in space.

The camera. The camera can be: translated (using the mouse left- or middle-click, or the keyboard arrows), rotated freely (using the mouse right-click or Ctrl/⌘+middle-click), and zoomed in and out (using the mouse wheel). Rotations and zooms are performed around the *pivot point* which is highlighted by a yellow ball. The pivot point is set by right-clicking on an element of the design. The button  in the top bar allows to auto-adjust the zoom to fit the whole design in the views. *Right-double-clicking* on an element in the 2D view centers this element in the 3D view. *Left-double-clicking* on an element in the 3D view centers this element in the 2D view.

3D arrangement. The grids and helices can be rearranged by selecting them and choosing the action mode  or . Handles appear to be pulled in the three possible directions. Handles are either aligned with the current orientation of the object, or with canonical axes of the design, see Fig. A.2. The latter choice is preferred to align different components precisely. Clicking again on the action mode  or  switches from one handles alignment to the other.

Fog. Sometimes, the 3D view can be confusing. The *fog* feature in the tab  allows to display only the part of the design within a given radius around the pivot or the camera, fading the rest progressively to invisible, see Fig. 7.



■ **Figure 7 The fog feature.** (left) **no fog:** the background is jammed with strands, and it is hard to focus on the crossovers between the two layers; (right) **with fog:** the background is cleared and the crossover is now clearly visible, ready to be adjusted if needed.

Rendering style. By default, the background of the 3D view is a landscape gradient, which is convenient to keep track of the current camera orientation. It can be replaced by a white background (e.g., for publication) in the rendering section of the tab ■. One can also opt there for a cartoon rendering, where each object is outlined in black (e.g., for printing).

Editing in the 3D view. Sometimes, it is easier to edit directly in the 3D view. Left-click-and-drag on an end of a strand or a crossover allows to translate it along the helix.¹ One can build a crossover by a long-press left-click on a nucleotide (longer than 250ms): a blue ball appears, and dragging to another nucleotide creates the crossover.

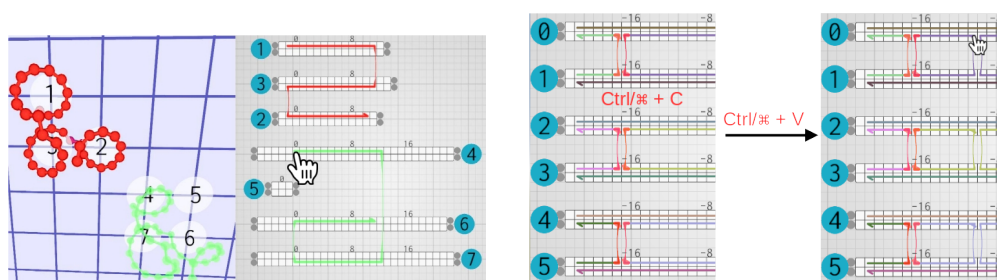
3.3 The 2D view

The 2D view is the blueprint of the design. It follows and extends the streamlined interface of *cadnano* and *scadnano* by adding a list of features that improves its ergonomics:

- Creation and edition of strands, creation and translation of crossovers, cutting and ligating strands are all done in the same edition mode, using only the mouse *left-click*, as shown in Fig. 3. Note that the *selection in the 2D view* is accomplished by *right-clicking*.
- The 2D and 3D views work together: a translucent green ball indicates in the 3D view which nucleotide is hovered by the mouse in the 2D view (e.g., see Fig. 3F and 3H). Also, double-right-clicking a nucleotide in the 2D view centers it in the 3D view.
- The helix representations automatically extend when needed, e.g. when elongating a strand. The helix representations can be tighten back using the buttons “All/Selected” under “Tighten 2D helices” in the edition tab ✏, or simply by clicking on their handles.
- Any helix representation can be translated and rotated arbitrarily in the 2D view, to match as closely as possible the 3D arrangement of design, or serve any other purposes, e.g. Fig. A.3. Left-clicking on their number will translate the selected helices, while right-clicking will rotate them.
- Moving in the 2D view is done by middle- or Alt/⌘+left-click-and-drag. Zooming in and out is done by scrolling the mouse wheel.

The 2D split view. As mentioned earlier, 3D complex structures cannot be faithfully mapped into 2D, and some parts that are next to each other in space, will inevitably be mapped far apart in any 2D view. This usually makes 2D representation of 3D DNA nanostructures complex to read and even more to edit. For instance, very long crossovers

¹ Cutting and ligating a strand by left-click are disabled in the 3D view because it is too error-prone.



■ **Figure 8 Grid geometry-aware copy and paste of strands and crossovers. (left) Duplication of a strand:** the red strand gets duplicated on other helices of the grid. One can check in the 3D interface that the path of the strand is correctly being copied, even if the 2D view could be reorganized to present a clearer representation of the strand. **(right) Duplication of crossovers:** four crossovers are being copied at once on existing strands.

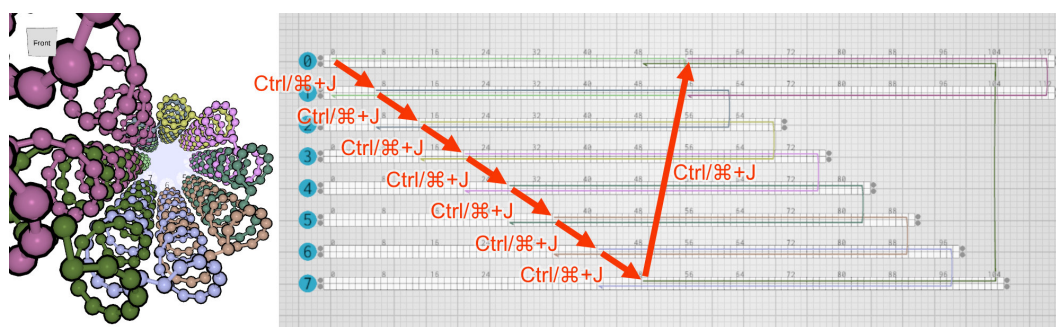
cross each other in every possible direction in the *cadnano* representation of the double-layer origami in [29], and make it almost impossible to edit. *ENSnano*'s 2D split view solves this issue elegantly by allowing to zoom and to *act seamlessly on two distant parts* of the 2D view, as if they were next to each other. Indeed, one can build a crossover from one split view to the other just as if it was one single 2D view, see Fig. 2. Moreover, crossovers whose ends are in opposite sides of the split view, *are drawn across the views*, making them easy to read and edit.

Note that left double-clicking on a crossover in the 3D view, splits the 2D view as soon as both of its ends do not fit in the 2D view. We thus recommend to 1) create the desired crossover approximately in the 3D view and then 2) to double-click on it, so that it gets focused and drawn across the split view, where the user can then edit it comfortably.

3.4 Grid-aware copy, paste & repeat

Grid geometry-aware copy & paste. Many DNA nanostructure designs consist of repeating the same pattern over and over, e.g.: SST nanotubes [36, 35], rectangular DNA origami [34], or SST 3D assemblies [17, 24]... and many more contains many repeating crossover patterns. A designer can thus save a lot of time by copying and pasting patterns, as proposed for instance in *scadnano*. *ENSnano* extends with capacity by using the grid structure to compute the 3D path followed by the pattern copied, to paste the same path at a different location, *regardless of the numbering of the helices and of their relative positions in the 2D view*. Arbitrarily complex strands can thus be copied and pasted across the design, as shown on Fig. 8. For instance, a strand binding two consecutive helices in a nanotube can be copied all around the nanotube: *ENSnano* will automatically loop the strand around the nanotube from its last to its first helix. In addition to copying strands on empty positions, it is also possible to copy crossovers on existing strands, see Fig. 8. The basic copy and paste functionality is performed by pressing **Ctrl/⌘+C** after selecting the source strands/crossovers and then **Ctrl/⌘+V** and click to place a copy.

Paste & repeat. In addition to the classic copy and paste feature. *ENSnano* features a geometry-aware *paste and repeat* which remembers as well the translation (in the grid *and* along the helices) between the original and the first pasted pattern, and keeps pasting the pattern with the same translation over and over. This is particularly useful for large repetitive designs such as rectangular part of an origami or SST nanotubes. After copying




■ **Figure 9 Paste & repeat:** Starting from the green strand made of two domains of 56 nucleotides each on helices 0 and 1 of a 8-helices nanotube, this strand is copied with `Ctrl/⌘+C` and pasted with `Ctrl/⌘+J`, one helix below and 7 nucleotides forward. Repeating `Ctrl/⌘+J` compulsively 7 more times creates automatically the other strands, applying repetitively the same translation and thus filling the nanotube with strands in no time. Note that ENSnano is aware of the nanotube-grid geometry and places the 7th pasted olive strand appropriately, binding helices 7 and 0.


the strands or crossover pattern with `Ctrl/⌘+C`, the first duplication is made by pressing `Ctrl/⌘+J`. Once the first copy is positioned, the path from the original to the copy is memorized. Pressing `Ctrl/⌘+J` repetitively, will copy over and over the pattern with the same offset as long as there are helices to support them, as illustrated in Fig. 9.

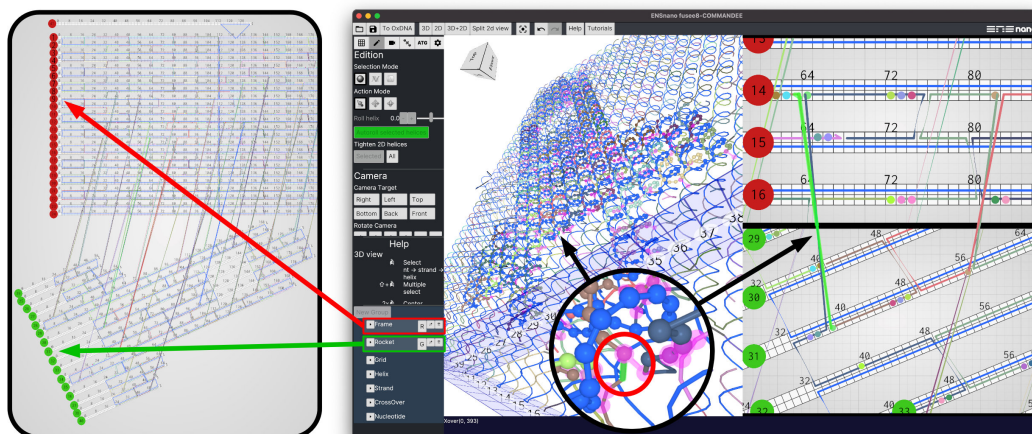
3.5 Crossover suggestions

Crossover suggestions are currently done by assigning helices to either the green or the red family. Color family are assigned in the organizer, by clicking on the family button which displays either: \emptyset for none, R for red, and G for green. The easiest is to create a group for each family of helix and to set the color family for each group at once by clicking on the family button of the group, see Fig. 10. Crossover suggestion is enabled as soon as there are helices of the red and green families. Setting all the families back to \emptyset disable the crossover suggestion.


3.6 The basic 3D rigid body physics engine

The tab  presents the three possible modes of our rigid body physics engine:

- the “Roll” button executes a simple roll of all the helices (see Sec. 2). This is a good way to check the quality of the crossovers. Note that the roll of each helix can be set individually in the tab .
- the “Rigid grid” button runs the physics engine considering that all the helices belonging to the same grid act as one rigid body. This allows to evaluate the interactions between the various components of a design, in particular, the balance of the crossovers between them.
- the “Rigid helices” button runs the physics engine as explained in Sec. 2. In this mode, several parameters can be tuned live: the stiffness of the crossovers, the ambient friction (we recommend to start the simulation with a higher friction and to decrease it with time), and the mass of the nucleotides (beware that tuning this parameter live may explode the design as it will apply some rocket effect). We strongly recommend not to use the volume exclusion option which is inefficient and slow right now. Note that opting for “jiggling unmatched nucleotides”, simulates Brownian motion by adding constantly a random noise to their positions (the rate and amplitude of the noise can be tuned as well).



■ **Figure 10 Crossover suggestions between red and green color families.** The helices are partitioned into two groups in the organizer: “Frame” and “Rocket” assigned resp. to the red and green families. One can see the pair of matching dots in the split view marking recommended positions for crossovers.

The two last modes are still at their infancy. The “Rigid grid” mode is useful when combined with the “Guess grid” feature in the tab , when importing a design from (s)cadnano which has no grid and no 3D embedding. Because the volume exclusion implementation is right now inefficient, the “rigid helices” mode should only be considered as a stability indicator, as it may converge to unrealistic configurations. Note that running the rigid body engine can be undone with `Ctrl/⌘+Z` or the Undo button.

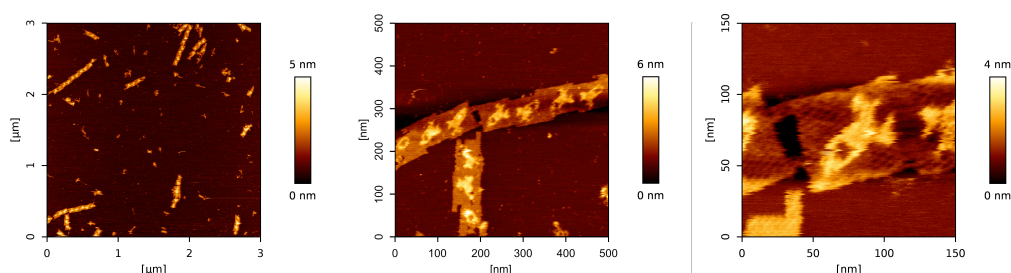
3.7 Sequences export and scaffold sequence optimization

The tab `ATG` gathers the sequence-related tools. As far as sequences are concerned, ENSnano is, *for now*, “DNA origami-oriented” (this will change in the future). This sequence tab allows to choose the scaffold strand and to assign its sequence. One can load any text file containing a long enough sequence for that purpose. The default scaffold sequence is the M13mp18. In ENSnano, the user can either set the starting position of the cyclic scaffold sequence, or let ENSnano *optimize it*. When asked to optimize, ENSnano tries to minimize the risk of error in the strand synthesis, by finding the position that minimizes the number of problematic patterns such as $C^n \geq 4$, $G^n \geq 4$ or $(A|T)^n \geq 7$. The higher the n , the heavier the weight of the problem. Once optimized, ENSnano reports on the number of remaining problematic patterns as shown in Fig. A.4. This position is retained and saved for later reuse. This optimization phase allows usually to avoid any C^{5+} or G^{5+} pattern in no time.

Note that nucleotides that are not matched with the scaffold are given a ? symbol in the exported excel file.

4 Experimental validation

We have already annealed successfully several DNA origami designed with ENSnano. We present here one of them designed together with the students of the class CR11 on Molecular computing at the ÉNS de Lyon in December 2020: a rocket DNA origami consisting of two layers made of parallel helices making an odd angle. Please refer to ENSnano website [19] for the full design and staple sequences. Designing this origami was particularly easy with



■ **Figure 11** AFM images of our rocket design. These were obtained on a JPK Fastscan Nanoworld 4 equipped with a Nanoworld USC-F0.3-k0.3 tip in tapping mode – 20 μ L sample of: m13mp18 scaffold at 1nM with staples at 10nM in 1 \times TAE buffer with 12.5mM magnesium.

ENSnano thanks to the crossover recommendation and grid systems. We were able to place the crossovers between the two layers, where the strands of the top and bottom layers were the closest, painlessly, and without any calculation. Splitting the 2D view was of great help as well. The possibility to arrange freely the helices in the 2D view allowed to check readily the design in the 2D view as well as in the 3D view.

The computed strands were annealed at 10nM together with the m13mp18 scaffold at 1nM in 1 \times TAE buffer with 12.5mM magnesium: starting from 95 $^{\circ}$ C and decreasing to 55 $^{\circ}$ C at -1° C/min and then from 55 $^{\circ}$ C to 45 $^{\circ}$ C at -1° C/15min and then hold at 25 $^{\circ}$ C. AFM images of the resulting DNA origami are presented in Fig. 11.

5 Conclusion and upcoming features

With this first version of ENSnano, its 3D and 2D fast, precise, and versatile editing capacities, and its stability, we have set the basis for an upcoming complete, GUI based, cross-platform, DNA nanostructure design suite. This software will evolve at a fast pace in the upcoming months. Here is a short list of features, we plan to develop next:

- *Versatility*: presently the design is limited to straight DNA helices. We plan to add soon curved double strands and free single strands to the toolbox. We also plan to add soon “decorations” (fluorophores, biotin,...). We also plan to extend our concept of grid to more complex structures (2D as well as 3D).
- *Sequences*: ENSnano is presently limited to DNA origami in terms of sequence export. We are currently working on an interface to set the sequences for the parts of the design which are not matched with a scaffold.
- *Programmability*: ENSnano file format is a standard json dictionary, very similar to scadnano file format. Its complete structure is described in appendix B. Even if it is fairly easy to develop a python code to produce ENSnano file, we plan to propose as soon as possible, a python framework to produce ENSnano design file programmatically.
- *Automation*: Designing the staples is a tedious task. Adding auto-stapling algorithms will be of great help to the designer. Automatic scaffolding will however require a new kind of abstraction of a DNA design.
- *Simulation*: improving the rigid body physics engine will probably require a lot of work. Adding a proper volume exclusion is certainly the most interesting direction to follow.

References

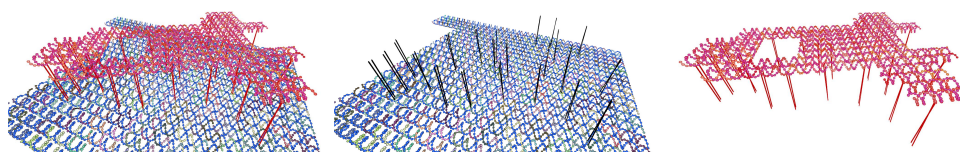
- 1 Erik Benson, Abdulmelik Mohammed, Johan Gardell, Sergej Masich, Eugen Czeizler, Pekka Orponen, and Björn Högberg. DNA rendering of polyhedral meshes at the nanoscale. *Nature*, 523(7561):441–444, 2015. doi:10.1038/nature14586.
- 2 Junghuei Chen and Nadrian C. Seeman. Synthesis from DNA of a molecule with the connectivity of a cube. *Nature*, 350(6319):631–633, 1991. doi:10.1038/350631a0.
- 3 Elisa de Llano, Haichao Miao, Yasaman Ahmadi, Amanda J Wilson, Morgan Beeby, Ivan Viola, and Ivan Barisic. Adenita: interactive 3D modelling and visualization of DNA nanostructures. *Nucleic Acids Research*, 48(15):8269–8275, July 2020. doi:10.1093/nar/gkaa593.
- 4 David Doty, Benjamin L Lee, and Tristan Stérin. scadnano: A Browser-Based, Scriptable Tool for Designing DNA Nanostructures. In Cody Geary and Matthew J. Patitz, editors, *26th International Conference on DNA Computing and Molecular Programming (DNA 26)*, volume 174 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DNA.2020.9.
- 5 Shawn M. Douglas, Adam H. Marblestone, Surat Teerapittayanon, Alejandro Vazquez, George M. Church, and William M. Shih. Rapid prototyping of 3D DNA-origami shapes with caDNano. *Nucleic Acids Research*, 37(15):5001–5006, June 2009. doi:10.1093/nar/gkp436.
- 6 Jonathan P. K. Doye, Thomas E. Ouldridge, Ard A. Louis, Flavio Romano, Petr Šulc, Christian Matek, Benedict E. K. Snodin, Lorenzo Rovigatti, John S. Schreck, Ryan M. Harrison, and William P. J. Smith. Coarse-graining DNA for simulations of DNA nanotechnology. *Physical Chemistry Chemical Physics*, 15(47):20395–20414, 2013. doi:10.1039/C3CP53545B.
- 7 Pierre Étienne Meunier, Nicolas Levy, and Damien Woods. Codenano: a code-based tool for designing DNA nanostructures. <https://dna.hamilton.ie/2019-07-18-codenano.html>, 2020.
- 8 Thomas Gerling, Klaus F. Wagenbauer, Andrea M. Neuner, and Hendrik Dietz. Dynamic DNA devices and assemblies formed by shape-complementary, non-base pairing 3D components. *Science*, 347(6229):1446–1452, 2015. doi:10.1126/science.aaa5372.
- 9 Martin Glaser, Sourav Deb, Florian Seier, Amay Agrawal, Tim Liedl, Shawn Douglas, Manish K. Gupta, and David M. Smith. The art of designing DNA nanostructures with CAD software. *Molecules*, 26(8), 2021. doi:10.3390/molecules26082287.
- 10 Arnav Goyal, Dixita Limbachiya, Shikhar Kumar Gupta, Foram Joshi, Sushant Pritmani, Akshita Sahai, and Manish K Gupta. Dna pen: A tool for drawing on a molecular canvas, 2013. arXiv:1306.0369.
- 11 Shikhar Kumar Gupta, Foram Joshi, Dixita Limbachiya, and Manish K. Gupta. 3DNA: A tool for DNA sculpting. *CoRR*, abs/1405.4118, 2014. arXiv:1405.4118.
- 12 Chao-Min Huang, Anjelica Kucinic, Joshua A. Johnson, Hai-Jun Su, and Carlos E. Castro. Integrated computer-aided engineering and design for DNA assemblies. *Nature Materials*, 2021. doi:10.1038/s41563-021-00978-5.
- 13 iced repository. <https://github.com/hecrj/iced>.
- 14 Hyungmin Jun, Tyson R. Shepherd, Kaiming Zhang, William P. Bricker, Shanshan Li, Wah Chiu, and Mark Bathe. Automated sequence design of 3d polyhedral wireframe DNA origami with honeycomb edges. *ACS Nano*, 13(2):2083–2093, February 2019. doi:10.1021/acsnano.8b08671.
- 15 Hyungmin Jun, Xiao Wang, William P. Bricker, Steve Jackson, and Mark Bathe. Rapid prototyping of wireframe scaffolded dna origami using athena. *bioRxiv*, 2020. doi:10.1101/2020.02.09.940320.
- 16 Hyungmin Jun, Fei Zhang, Tyson Shepherd, Sakul Ratanalert, Xiaodong Qi, Hao Yan, and Mark Bathe. Autonomously designed free-form 2D DNA origami. *Science Advances*, 5(1), 2019. doi:10.1126/sciadv.aav0655.
- 17 Yonggang Ke, Luvena L. Ong, William M. Shih, and Peng Yin. Three-dimensional structures self-assembled from dna bricks. *Science*, 338(6111):1177–1183, 2012. doi:10.1126/science.1227268.

- 18 Jae Young Lee, Jae Gyung Lee, Giseok Yun, Chanseok Lee, Young-Joo Kim, Kyung Soo Kim, Tae Hwi Kim, and Do-Nyun Kim. Rapid computational analysis of dna origami assemblies at near-atomic resolution. *ACS Nano*, 15(1):1002–1015, January 2021. doi:10.1021/acsnano.0c07717.
- 19 Nicolas Levy and Nicolas Schabanel. Ensnano: A software for designing 3d DNA/rna nanostructures, May 2021. URL: <http://www.ens-lyon.fr/ensnano/>.
- 20 Ronny Lorenz, Stephan H. Bernhart, Christian Höner zu Siederdisen, Hakim Tafer, Christoph Flamm, Peter F. Stadler, and Ivo L. Hofacker. ViennaRNA package 2.0. *Algorithms Mol. Biol.*, 6:26, 2011.
- 21 Christopher Maffeo and Aleksei Aksimentiev. MrDNA: a multi-resolution model for predicting the structure and dynamics of DNA systems. *Nucleic Acids Research*, 48(9):5135–5146, March 2020. doi:10.1093/nar/gkaa200.
- 22 Nicholas D. Matsakis and Felix S. Klock. The rust language. *Ada Lett.*, 34(3):103–104, 2014. doi:10.1145/2692956.2663188.
- 23 Michael Matthies, Nayan P. Agarwal, Erik Poppleton, Foram M. Joshi, Petr Šulc, and Thorsten L. Schmidt. Triangulated wireframe structures assembled using single-stranded dna tiles. *ACS Nano*, 13(2):1839–1848, February 2019. doi:10.1021/acsnano.8b08009.
- 24 Luvena L. Ong, Nikita Hanikel, Omar K. Yaghi, Casey Grun, Maximilian T. Strauss, Patrick Bron, Josephine Lai-Kee-Him, Florian Schueder, Bei Wang, Pengfei Wang, Jocelyn Y. Kishi, Cameron Myhrvold, Allen Zhu, Ralf Jungmann, Gaetan Bellot, Yonggang Ke, and Peng Yin. Programmable self-assembly of three-dimensional nanostructures from 10,000 unique components. *Nature*, 552(7683):72–77, 2017. doi:10.1038/nature24648.
- 25 Erik Poppleton, Joakim Bohlin, Michael Matthies, Shuchi Sharma, Fei Zhang, and Petr Šulc. Design, optimization and analysis of large DNA and RNA nanostructures through interactive visualization, editing and molecular simulation. *Nucleic Acids Research*, 48(12):e72–e72, May 2020. doi:10.1093/nar/gkaa417.
- 26 Paul W. K. Rothmund. Folding DNA to create nanoscale shapes and patterns. *Nature*, 440(7082):297–302, 2006. doi:10.1038/nature04586.
- 27 Paul W. K. Rothmund, Nick Papadakis, and Erik Winfree. Algorithmic self-assembly of DNA Sierpinski triangles. *PLoS Biology*, 2:2041–2053, 2004.
- 28 Anupama J. Thubagere, Wei Li, Robert F. Johnson and Zibo Chen, Shayan Doroudi, Yae Lim Lee, Gregory Izatt, Sarah Wittman, Niranjana Srinivas, Damien Woods, Erik Winfree, and Lulu Qian. A cargo-sorting DNA robot. *Science*, 357(6356), 2017. doi:10.1126/science.aan6558.
- 29 Anupama J Thubagere, Wei Li, Robert F Johnson, Zibo Chen, Shayan Doroudi, Yae Lim Lee, Gregory Izatt, Sarah Wittman, Niranjana Srinivas, Damien Woods, et al. A cargo-sorting DNA robot. *Science*, 357(6356), 2017.
- 30 Rémi Veneziano, Sakul Ratanalert, Kaiming Zhang, Fei Zhang, Hao Yan, Wah Chiu, and Mark Bathe. Designer nanoscale DNA assemblies programmed from the top down. *Science*, 352(6293):1534, June 2016. doi:10.1126/science.aaf4388.
- 31 wgpu repository. <https://github.com/gfx-rs/wgpu-rs>.
- 32 Sean Williams, Kyle Lund, Chenxiang Lin, Peter Wonka, Stuart Lindsay, and Hao Yan. Tiamat: A three-dimensional editing tool for complex DNA structures. In Ashish Goel, Friedrich C. Simmel, and Petr Sosík, editors, *DNA Computing*, pages 90–101, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- 33 winit repository. <https://github.com/rust-windowing/winit>.
- 34 Sungwook Woo and Paul WK Rothmund. Programmable molecular recognition based on the geometry of DNA nanostructures. *Nature chemistry*, 3(8):620, 2011.
- 35 Damien Woods, David Doty, Cameron Myhrvold, Joy Hui, Felix Zhou, Peng Yin, and Erik Winfree. Diverse and robust molecular algorithms using reprogrammable DNA self-assembly. *Nature*, 567(7748):366–372, 2019.
- 36 Peng Yin, Rizal F. Hariadi, Sudheer Sahu, Harry M. T. Choi, Sung Ha Park, Thomas H. LaBean, and John H. Reif. Programming dna tube circumferences. *Science*, 321(5890):824–826, 2008. doi:10.1126/science.1157312.

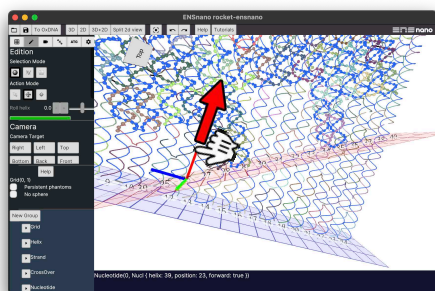
- 37 Joseph H. Zadeh, Conrad D. Steenberg, Justin S. Bois, Brian R. Wolfe, Marshall B. Pierce, Asif R. Khan, Robert M. Dirks, and Niles A. Pierce. NUPACK: Analysis and design of nucleic acid systems. *Journal of Computational Chemistry*, 32:170–173, 2011.
- 38 Fei Zhang, Shuoxing Jiang, Siyu Wu, Yulin Li, Chengde Mao, Yan Liu, and Hao Yan. Complex wireframe DNA origami nanostructures with multi-arm junction vertices. *Nature Nanotechnology*, 10(9):779–784, 2015. doi:10.1038/nnano.2015.162.

A Omitted figures

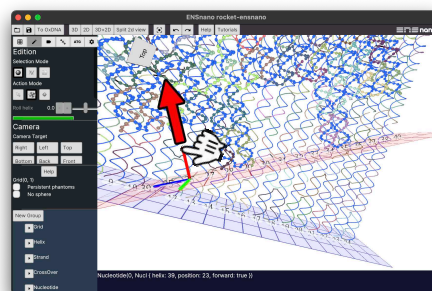
This section gathers the figures cast away in the appendix due to space constraints.



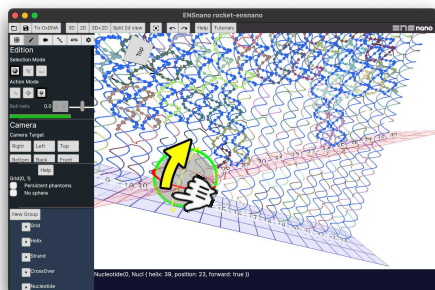
■ **Figure A.1 Using ENSnano's groups to hide elements of the design. (left)** The two layers of the rocket design which are spread apart to ease the reading of the crossovers in the figures. Two overlapping heterogeneous groups, gathering helices and crossovers, have been created in the organizer, and can be shown or hidden on demand in the 3D view: **(middle)** A first group gathering the helices in the rectangular base, and the crossovers between the two layers; **(right)** A second group gathering the helices of the rocket and the crossovers between the two layers.



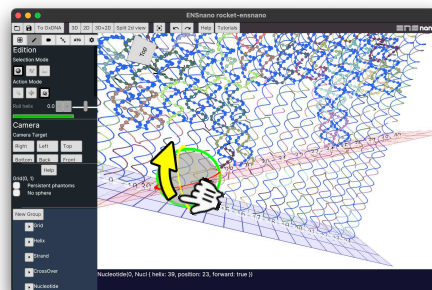
(A) Translation handles aligned with the canonical axes



(B) Translation handles aligned with the object axes

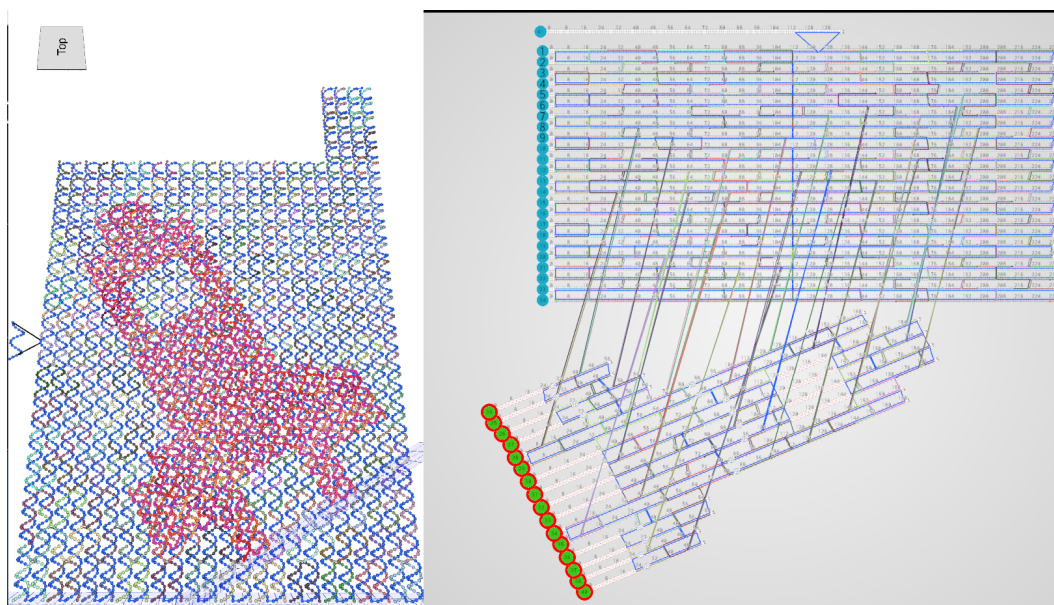


(C) Rotation handles aligned with the canonical axes



(D) Rotation handles aligned with the object axes

■ **Figure A.2** The move and rotate handles to rearrange grids and helices in 3D.



■ **Figure A.3 Comparison between the 2D and 3D interfaces.** The 3D interface shows the shape, that we wish that our design will adopt. Since this design is made of two layers making an odd angle. It cannot be faithfully represented in 2D. By separating the helices of the two layers, and adjusting the orientation of the helices composing the rocket layer, one can build a 2D representation of the design that is as close as possible to its intended 3D structure.



■ **Figure A.4** Report on scaffold sequence position optimization.

B ENSnano File format

B.1 Top-level structure

In ENSnano, designs are loaded from a json file. Here are the top-level element:

- **helices:** a dictionary that maps positive integer to **Helix** object (see the definition of **Helix** object in B.3).
- **strands:** a dictionary that maps positive integer to **Strand** object (see the definition of **Strand** objects in B.4).
- **dna_parameters:** (optional) contains the geometric DNA parameters used for the design. If this field is omitted, default values will be used. The members of this field, and their default values are given in B.2.

- `grids`: an array containing the `GridDescriptor` objects, that represent the grids of the design.
- `scaffold_id`: (optional) the id of the scaffold strand.
- `scaffold_sequence`: (optional) the sequence of the scaffold.
- `scaffold_shift`: (optional) the starting position of the scaffold sequence.
- `groups`: (optional) a dictionary mapping integers (identifier of helices) to a boolean indicating in which group they belong for the cross-over suggestion.
- `small_spheres`: (optional) a set containing identifiers of the grids whose helices do not display their nucleotides as spheres.
- `no_phantoms`: (optional) a set containing identifiers of the grids that do not display phantom helices.

B.2 DNA parameters

The members of the field `dna_parameters` are:

- `z_step`: the distance in nanometers between two consecutive bases along the axis of an helix. The default value is 0.332nm.
- `helix_radius`: the radius of an helix in nanometers. The default value is 1nm.
- `groove_angle`: the small angle between paired nucleotides. The default value is $2\pi \cdot \frac{12}{12+22}$. This values comes from the fact that the width of the major groove is 22 Å and the width of the minor groove is 12 Å.
- `inter_helix_gap`: the distance between two neighbouring helices on a grid. The default value is 0.65nm as suggested in [34, 29].
- `bases_per_turn`: the number of base pairs per full turn of an helix. The default value is 10.44 bases per full turn as suggested in [34].

These parameters are illustrated in Figure B.5. The coordinates of a nucleotide at index i on an helix, in the helix referential are:

$$x = i \cdot z_step, \quad y = \sin(\theta_i), \quad \text{and} \quad z = \cos(\theta_i)$$

and

$$\theta_i = \rho - i \cdot \frac{2\pi}{\text{bases_per_turn}} + \frac{\pi}{2} + \begin{cases} \text{groove_angle} & \text{if the nucleotide is on the forward strand} \\ 0 & \text{otherwise} \end{cases}$$

where ρ is the roll of the helix. A shift of $\frac{\pi}{2}$ is added so that the nucleotide at index 0 on the backward strand of an helix is at the top position, following the `cadnano` convention.

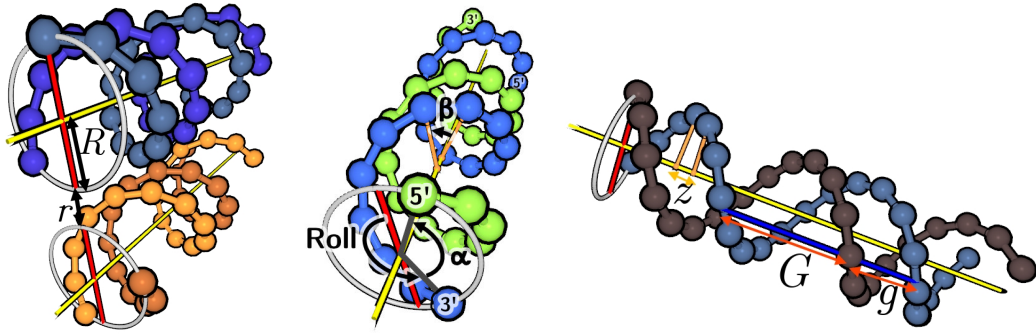
Note that θ_i decreases as index i increases. This is because strands turn clockwise when going from 5' to 3'.

B.3 Helix Object

The field of an helix object are

- `position`: a 3D point giving the origin of the helix.
- `orientation`: a `Rotor`² giving the orientation of the helix. See Subsection B.5 to see how to obtain a rotor from a direction vector.
- `roll`: an additional roll performed on the helix.

² <https://docs.rs/ultraviolet/0.8.0/ultraviolet/rotor/index.html>



■ **Figure B.5 The configurable DNA geometric parameters:** R : the radius of an helix, labelled `helix_radius`; r : the distance between two neighbour helices on a grid, labelled `inter_helix_gap`; α : the minor groove angle, labelled `groove_angle`; β : the angle between two consecutive base-pairs. This angle can be obtained by the formula $\beta = \frac{2\pi}{\text{bases_per_turn}}$; z : the distance in nanometers between two consecutive bases along the axis, labelled `z_step`; G : the length of the major groove; g : the length of the minor groove; G , g and α are related to each other by the formula $\alpha = 2\pi \frac{g}{G+g}$.

- `grid_position`: (optional) the position of the helix on the grid it is attached to. This field override the `position` and `orientation` fields.
- `isometry2d`: (optional) an isometry³ that determines the position and orientation of the helix representation in the 2D view.

B.4 Strand Object

The field of a strand object are:

- `domains`: the (ordered) vector of domains, where each domain is either an insertion or a directed interval of a helix.
- `color`: an integer encoding the color of the strand with the formula $c = 65536 \times R + 256 \times G + B$ where $R, G, B \in \{0, \dots, 255\}$ are its red, green and blue components.
- `junctions`: (optional) an array of objects representing the junctions between the strand's domains.
- `cyclic`: a boolean indicating whether the strand is cyclic or not.

B.5 A word about rotors

In ENSnano's file format, rotations are represented by *rotors*.

A 3-dimensional rotor has a scalar part $s \in \mathbb{R}$ and a bivector part $bv \in \mathbb{R}^3$. A rotor can be obtained by taking the geometric product of two vectors. The geometric product of a and b is defined by $ab = (a \cdot b + a \wedge b)$, i.e. it is a rotor with scalar $s = a \cdot b$ and bivector $bv = a \wedge b$.

By properties of the scalar and exterior product of vectors, one can write

$$ab = \cos(\theta) + \Omega \sin(\theta)$$

where $\Omega = \frac{a \wedge b}{\|a \wedge b\|}$ is the axis of the rotation that transforms a in b and θ is the angle between a and b .

³ <https://docs.rs/ultraviolet/0.8.0/ultraviolet/transform/struct.Isometry2.html>

Applying the rotor ab to a vector u will rotate the vector u in the oriented plane defined by a and b by twice the angle between a and b .

Obtaining a rotor from a direction vector. For the reader who would like to write a program to output a design in ENSnano file format, we provide the following procedure that produces a rotor from a direction vector.

```
def to_rotor(v):
    """ Transform a vector v = (x, y, z) to a rotor
    describing the rotation from the x-axis
    to the vector
    """
    x, y, z = v
    norm = (x*x + y * y + z*z)**0.5
    x = x/norm
    y = y/norm
    z = z/norm
    s = 1. + x
    norm_bv = (z * z + y * y)**0.5
    if norm_bv < 1e-5:
        if x > 0:
            return (1., (0., 0., 0.))
        else:
            return(0., (1., 0., 0.))
    norm = (z * z + y * y + s*s)**0.5
    bv = (-y/norm, -z / norm, 0)
    s = s / norm
    return(s, bv)
```


Directed Non-Cooperative Tile Assembly Is Decidable

Pierre-Étienne Meunier ✉

Albédo Énergie, Le Bourget-du-Lac, France

Damien Regnault ✉

IBISC, Université Évry, Université Paris-Saclay, 91025, Evry, France

Abstract

We provide a complete characterisation of all final states of a model called *directed non-cooperative tile self-assembly*, also called *directed temperature 1 tile assembly*, which proves that this model cannot possibly perform Turing computation. This model is a deterministic version of the more general *undirected* model, whose computational power is still open. Our result uses recent results in the domain, and solves a conjecture formalised in 2011. We believe that this is a major step towards understanding the full model.

Temperature 1 tile assembly can be seen as a two-dimensional extension of finite automata, where geometry provides a form of memory and synchronisation, yet the full power of these “geometric blockings” was still largely unknown until recently (note that nontrivial algorithms which are able to build larger structures than the naive constructions have been found).

2012 ACM Subject Classification Mathematics of computing; Theory of computation → Models of computation

Keywords and phrases Self-assembly, Molecular Computing, Models of Computation, Computational Geometry

Digital Object Identifier 10.4230/LIPIcs.DNA.27.6

Related Version *Extended Version*: <https://arxiv.org/abs/2011.09675>

Funding *Pierre-Étienne Meunier*: Supported by H2020-LC-SC3 award number 957823, H2020-LC-SC3 award number 953020, H2020-LC-SC3 award number 101033700, H2020-ERC award number 772766 and SFI grant 18/ERCS/5746 (this manuscript reflects only the authors’ view and the European funding institutions are not responsible for any use that may be made of the information it contains).

Acknowledgements We thank Damien Woods for his support and advice.

1 Introduction

Self-assembly is the process by which independent, unsynchronised components coalesce into complex forms and patterns, using geometry and local constraints to exchange information, and perform different sorts of computations. In particular, self-assembly is the process by which molecules, and in particular biomolecules, acquire their shape (and therefore their function).

A computational theory of self-assembly has a wealth of applications in a large range of fields and scales. At the molecular level, programming molecules would enable us to interact with living organisms, potentially defeating the geometric strategies used by nasty viruses to penetrate cells. Smart materials with new properties such as self-reproduction and self-repairing are another example. At a much larger scale, industrial processes could also benefit from a better understanding of self-assembly, as it could streamline processes and make industrial robots simpler.



© Pierre-Étienne Meunier and Damien Regnault;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on DNA Computing and Molecular Programming (DNA 27).

Editors: Matthew R. Lakin and Petr Šulc; Article No. 6; pp. 6:1–6:21

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This theory has already yielded experimental realisations such as DNA Origami [24], allowing anyone to make their own molecules of any prescribed shape of a diameter between 10nm and 500nm. DNA Self-Assembly has also been used to build fractal shapes [26], information retrieval circuits [23], cyclic machines using DNA as machine material *and* as fuel [30]. Another recent application has been the amplification of minuscule concentrations of a molecular compound in solution, by using it as a “seed” for self-assembling large structures [21]. DNA storage [2] has also been proposed and implemented as a technique to store a tremendous amount of information in a tiny space, with millions of years of potential durability.

These developments have happened in parallel to, and with interactions with work on the computer science theory of tile assembly. The most studied model in that direction is the abstract Tile Assembly Model (aTAM), created by Winfree [29, 25] with inspiration from Wang tilings [28]. This model studies assemblies made of *square tiles* with colours on their borders. Using a finite set of tile *types*, and an assumed infinite supply of each type, the assembly process starts with an initial “seed” assembly, and proceeds nondeterministically and asynchronously, one tile at a time. Unlike Wang tilings, which is mostly concerned with (potentially undecidable) full covers of the plane, the abstract Tile Assembly Model studies the *assembly sequence* of an assembly, which is the sequence of binding events necessary to build a shape.

In the fully general abstract Tile Assembly Model, tile borders have a *glue strength* on their border, and the model has a global assembly threshold called the “*temperature*”: in order to remain stably attached, the sum of glue strengths on the attached borders of a tile must be at least equal to the temperature. One of the key complexity measures of this model is *program-size complexity*, meaning the number of tile types in the tileset. The fact that this model can simulate Turing machines has been used to encode complex shapes with a number of tile types logarithmic in the Kolmogorov complexity of the shapes [27]. Moreover, the aTAM model is also *intrinsically universal*, meaning that there is a single finite “universal” tileset capable of simulating any other tileset up to a constant scaling factor [6]. Over the years, a number of consequences and extensions of that result have also been studied [7, 4, 5], and intrinsic universality has also been used to classify models according to their simulation power [16].

1.1 Noncooperative self-assembly

Noncooperative self-assembly is a restriction of the aTAM to a temperature of 1, meaning that tiles always attach to an existing assembly as soon as at least one side has its colour matching the colour of the current assembly. In other words, the assembly cannot “wait” for two different “branches” to meet at a point in the plane before growing further. The restriction of this model to one-dimension is exactly equivalent to finite automata, where tiles map to the edges of the automaton, and border colours to states.

The only form of synchronisation in this model is by geometric “blocking”, where two branches compete for a position in the plane, and the first one to get there can continue to grow. The fundamental question of noncooperative self-assembly is whether this rather weak form of communication is sufficient to achieve synchronisation. This has been an open problem since the early days of the field, and research in variants of the model has shown surprising results, in that every variation of the noncooperative model, however minor, seems to endow it with arbitrary computational capabilities. In the three-dimensional extension, for example, one can arrange little “bridges” and “tunnels” to block one branch of a test while allowing the other one to continue, which allows one to read and write bits [3]. In

two dimensions, using random assembly sequences rather than asynchronous ones yields the same result [3], and so do negative glues [22], polyomino tiles (with at least three unit squares) [9], polygonal tiles, provided they have at least seven sides [12]. Separating the assembly process into stages with different sets of tiles available at each stage also makes the model Turing-universal [1], which is also the case for a model with detachable tiles [13].

On the negative side, no tileset is intrinsically universal at temperature 1 [20], meaning that no tileset can simulate all temperature 1 tile assembly systems, even when rescaled. Moreover, it has recently been shown that long enough *paths* built by a temperature-1 tile assembly system are *pumpable*, meaning that their growth can only be controlled within a finite radius, after which they degenerate into simple periodic paths [19]. Moreover, disallowing mismatches means that all assemblies are periodic [14] (note that the “no mismatches” condition is not known to be decidable).

One particularly puzzling fact about 2D noncooperative self-assembly is that even though it seems computationally weak, a handful of nontrivial algorithms have been designed, including assemblies of diameter $\Omega(n \log n)$, produced by a tileset of size n [15, 17]. In three-dimensions, recent results have also shown how to build thin rectangles [11, 10] with almost matching upper and lower bounds.

1.2 Main results

Our main result is that the assemblies producible by directed non-cooperative tile assembly are a finite union of ultimately periodic assemblies. We state this semi-formally here, even though not all terms have been defined yet:

► **Definition 1.** *The complexity of a finite assembly is 0. For $i \geq 1$, the complexity of an assembly α is i if α is either defined as $\alpha = \bigcup_{\ell \in \mathbb{N}} (\beta + \ell \vec{v})$ where β is a assembly of complexity $i - 1$, or if α is a finite union of assemblies of complexity at most i .*

► **Theorem 2.** *Let $\mathcal{T} = (T, \sigma, 1)$ be a noncooperative tile assembly system. If \mathcal{T} has a unique terminal assembly α (or otherwise said, if \mathcal{T} is directed), then the complexity of α is less than 2.*

This result implies that a terminal assembly can be described by a finite number of finite assemblies and vectors. Along the proofs, we distinguish four different cases of terminal assemblies and for each class, we bound the size of the assemblies and the number of vectors used to describe a terminal assembly. Moreover, an algorithm which computes this description can be deduced from our work. Thus, directed tile assembly systems cannot perform Turing computation.

Our proof uses techniques similar to a theorem from 2011 [8] which achieves a complete characterisation of producible assemblies, but conditioned on an unproven conjecture (called the *pumpability conjecture*). In contrast to that proof, our work does not rely on any unproven hypothesis, and improve the previous characterisation of producible assemblies which implies the pumpability conjecture (this point is discussed for each case of the classification later). In order to do so, we use a result published in [19], showing that long enough paths are pumpable. That result is itself weaker than the original pumpability conjecture [8], in that the bound includes the size of the seed, whereas the pumpability conjecture is that any long enough subpath, even arbitrarily far from the seed, is pumpable. This subtle difference is important, since without our Theorem 2, seeds could be assumed to encode computation, for example by using complicated shapes. Theorem 2 shows that this is not the case.

2 Definitions

Some of these definitions come from [19]. As usual, let \mathbb{R} be the set of real numbers, let \mathbb{Z} be the set of all integers, let \mathbb{N} be the set of all natural numbers including 0, and let \mathbb{N}^* be the set of all natural numbers excluding 0. The domain of a function f is denoted $\text{dom}(f)$, and its range (or image) is denoted $f(\text{dom}(f))$.

2.1 Abstract tile assembly model

A *tile type* is a unit square with four sides, each consisting of a glue *type* and a nonnegative integer *strength*. Let T be a finite set of tile types. The sides of a tile type are respectively called north, east, south, and west.

An *assembly* is a partial function $\alpha : \mathbb{Z}^2 \dashrightarrow T$ where T is a set of tile types and the domain of α (denoted $\text{dom}(\alpha)$) is connected.¹ The translation of an assembly α by a vector \vec{v} , written $\alpha + \vec{v}$, is the assembly β defined for all $(x, y) \in (\text{dom}(\alpha) + \vec{v})$ as $\beta(x, y) = \alpha((x, y) - \vec{v})$. We let \mathcal{A}^T denote the set of all assemblies over the set of tile types T . In this paper, two tile types in an assembly are said to *bind* (or *interact*, or are *stably attached*), if the glue types on their abutting sides are equal, and have strength ≥ 1 . An assembly α induces an undirected weighted *binding graph* $G_\alpha = (V, E)$, where $V = \text{dom}(\alpha)$, and there is an edge $\{a, b\} \in E$ if and only if the tiles at positions a and b interact, and this edge is weighted by the glue strength of that interaction. The assembly is said to be τ -stable if every cut of G_α has weight at least τ .

A *tile assembly system* is a triple $\mathcal{T} = (T, \sigma, \tau)$, where T is a finite set of tile types, σ is a τ -stable (hence connected) assembly called the *seed*, and $\tau \in \mathbb{N}$ is the *temperature*. Throughout this paper, $\tau = 1$. Note also that the seed may be large, and placed at an arbitrary position in the plane. And indeed, in this paper, we will sometimes define multiple “intuitively equivalent” tile assembly systems where only the position of the seed differs.

Given two τ -stable assemblies α and β , we say that α is a *subassembly* of β , and write $\alpha \sqsubseteq \beta$, if $\text{dom}(\alpha) \subseteq \text{dom}(\beta)$ and for all $p \in \text{dom}(\alpha)$, $\alpha(p) = \beta(p)$. We also write $\alpha \rightarrow_1^T \beta$ if we can obtain β from α by the binding of a single tile type, that is: $\alpha \sqsubseteq \beta$, $|\text{dom}(\beta) \setminus \text{dom}(\alpha)| = 1$ and the tile type at the position $\text{dom}(\beta) \setminus \text{dom}(\alpha)$ stably binds to α at that position. We say that γ is *producible* from α , and write $\alpha \rightarrow^T \gamma$ if there is a (possibly empty) sequence $\alpha_1, \alpha_2, \dots, \alpha_n$ where $n \in \mathbb{N} \cup \{\infty\}$, $\alpha = \alpha_1$ and $\alpha_n = \gamma$, such that $\alpha_1 \rightarrow_1^T \alpha_2 \rightarrow_1^T \dots \rightarrow_1^T \alpha_n$. A sequence of $n \in \mathbb{Z}^+ \cup \{\infty\}$ assemblies $\alpha_0, \alpha_1, \dots$ over \mathcal{A}^T is a \mathcal{T} -*assembly sequence* if, for all $1 \leq i < n$, $\alpha_{i-1} \rightarrow_1^T \alpha_i$.

Given two τ -stable assemblies α and β , the union of α and β , write $\alpha \cup \beta$, is an assembly defined if and only if and for all $p \in \text{dom}(\alpha) \cap \text{dom}(\beta)$, $\alpha(p) = \beta(p)$ and either at least one tile of α binds with a tile of β or $\text{dom}(\alpha) \cap \text{dom}(\beta) \neq \emptyset$. Then, for all $p \in \text{dom}(\alpha)$, we have $(\alpha \cup \beta)(p) = \alpha(p)$ and for all $p \in \text{dom}(\beta)$, we have $(\alpha \cup \beta)(p) = \beta(p)$.

The set of *productions*, or *producible assemblies*, of a tile assembly system $\mathcal{T} = (T, \sigma, \tau)$ is the set of all assemblies producible from the seed assembly σ and is written $\mathcal{A}[\mathcal{T}]$. An assembly α is called *terminal* if there is no β such that $\alpha \rightarrow_1^T \beta$. The set of all terminal assemblies of \mathcal{T} is denoted $\mathcal{A}_\square[\mathcal{T}]$. If there is a unique terminal assembly, *i.e.* $|\mathcal{A}_\square[\mathcal{T}]| = 1$, then \mathcal{T} is *directed*. In this paper, this unique terminal assembly is denoted α .

¹ Intuitively, an assembly is a positioning of unit-sized tiles, each from some set of tile types T , so that their centers are placed on (some of) the elements of the discrete plane \mathbb{Z}^2 and such that those elements of \mathbb{Z}^2 form a connected set of points.

2.2 Paths

Let T be a set of tile types. A *tile* is a pair $((x, y), t)$ where $(x, y) \in \mathbb{Z}^2$ is a position and $t \in T$ is a tile type. Intuitively, a path is a finite or one-way-infinite simple (non-self-intersecting) sequence of tiles placed on points of \mathbb{Z}^2 so that each tile in the sequence interacts with the previous one, or more precisely:

- **Definition 3 (Path).** A path is a (finite or infinite) sequence $P = P_0P_1P_2\dots$ of tiles $P_i = ((x_i, y_i), t_i) \in \mathbb{Z}^2 \times T$, such that:
- for all P_j and P_{j+1} defined on P it is the case that t_j and t_{j+1} interact, and
 - for all P_j, P_k such that $j \neq k$ it is the case that $(x_j, y_j) \neq (x_k, y_k)$.

By definition, paths are simple (or self-avoiding), and this fact will be repeatedly used throughout the paper. For a tile P_i on some path P , its x-coordinate is denoted x_{P_i} and its y-coordinate is denoted y_{P_i} . The *concatenation* of two paths P and Q is the concatenation PQ of these two paths as sequences, and is a path if and only if (1) the last tile of P interacts with the first tile of Q and (2) P and Q do not intersect each other.

For a path $P = P_0\dots P_iP_{i+1}\dots P_j\dots$, we define the notation $P_{i,i+1,\dots,j} = P_iP_{i+1}\dots P_j$, i.e. “the subpath of P between indices i and j , inclusive”. Whenever P is finite, i.e. $P = P_0P_1P_2\dots P_{n-1}$ for some $n \in \mathbb{N}$, n is termed the *length* of P and denoted by $|P|$. In the special case of a subpath where $i = 0$, we say that $P_{0,1,\dots,j}$ is a prefix of P and when $j = |P| - 1$, we say that $P_{i,\dots,|P|-1}$ is a suffix of P . For any path $P = P_0P_1P_2\dots$ and integer $i \geq 0$, we write $\text{pos}(P_i) \in \mathbb{Z}^2$, or $(x_{P_i}, y_{P_i}) \in \mathbb{Z}^2$, for the position of P_i and $\text{type}(P_i)$ for the tile type of P_i . Hence if $P_i = ((x_i, y_i), t_i)$ then $\text{pos}(P_i) = (x_{P_i}, y_{P_i}) = (x_i, y_i)$ and $\text{type}(P_i) = t_i$. A “*position of P* ” is an element of \mathbb{Z}^2 that appears in P (and therefore appears exactly once), and an *index i* of a path P of length $n \in \mathbb{N}$ is a natural number $i \in \{0, 1, \dots, n - 1\}$. For a path $P = P_0P_1P_2\dots$ we write $\text{pos}(P)$ to mean “the sequence of positions of tiles along P ”, which is $\text{pos}(P) = \text{pos}(P_0)\text{pos}(P_1)\text{pos}(P_2)\dots$. For a finite path $P = P_0P_1P_2\dots P_{|P|-1}$, we define P^{\leftarrow} as the path $P_{|P|-1}P_{|P|-2}\dots P_0$. The vertical height of a path P is defined as $\max\{|y_{P_i} - y_{P_j}| : 0 \leq i \leq j \leq |P| - 1\}$ and its horizontal width is $\max\{|x_{P_i} - x_{P_j}| : 0 \leq i \leq j \leq |P| - 1\}$.

Although a path is not an assembly, we know that each adjacent pair of tiles in the path sequence interact implying that the set of path positions forms a connected set in \mathbb{Z}^2 and hence every path uniquely represents an assembly containing exactly the tiles of the path, more formally: for a path $P = P_0P_1P_2\dots$ we define the set of tiles $\text{asm}(P) = \{P_0, P_1, P_2, \dots\}$ which we observe is an assembly² and we call $\text{asm}(P)$ a *path assembly*.

Given a tile assembly system $\mathcal{T} = (T, \sigma, 1)$ the path P is a *producible path of \mathcal{T}* if $\text{asm}(P)$ does not intersect³ the seed σ and the assembly $(\text{asm}(P) \cup \sigma)$ is producible by \mathcal{T} , i.e. $(\text{asm}(P) \cup \sigma) \in \mathcal{A}[\mathcal{T}]$, and P_0 interacts with a tile of σ . Consider an assembly α (resp. a path Q), as a convenient abuse of notation we sometimes write $\sigma \cup P$ (resp. $P \cup Q$) as a shorthand for $\sigma \cup \text{asm}(P)$ (resp. $\text{asm}(P) \cup \text{asm}(Q)$).

Note that producible paths may not necessarily result in producible assemblies: indeed, in this paper, we will need to reason on multiple translations of a single path, and only later prove that these translations are actually connected to the seed. Therefore, we must be able to talk about these “temporarily disconnected” paths, while proving that they actually result in producible assemblies.

² I.e. $\text{asm}(P)$ is a partial function from \mathbb{Z}^2 to tile types, and is defined on a connected set.

³ Formally, the non-intersection of a path $P = P_0P_1\dots$ and a seed assembly σ is defined as: $\forall t$ such that $t \in \sigma$, $\nexists i$ such that $\text{pos}(P_i) = \text{pos}(t)$.

6:6 Directed Non-Cooperative Tile Assembly Is Decidable

Given a directed tile assembly system $\mathcal{T} = (T, \sigma, 1)$ and its unique terminal assembly α , the path P is a *path of α* if $\text{asm}(P)$ is a subassembly of α . We define the set of paths of α to be:

$$\mathbf{P}[\alpha] = \{P \mid P \text{ is a path and } \text{asm}(P) \text{ is a subassembly of } \alpha\}$$

Note that, for any tiles $((x, y), t) \in \alpha$ and $((x', y'), t') \in \alpha$ there is a path $P \in \mathbf{P}[\alpha]$ such that for some $P_0 = ((x, y), t)$ and $P_{|P|-1} = ((x', y'), t')$.

For $A, B \in \mathbb{Z}^2$, we define $\overrightarrow{AB} = B - A$ to be the vector from A to B , and for two tiles $P_i = ((x_i, y_i), t_i)$ and $P_j = ((x_j, y_j), t_j)$ we define $\overrightarrow{P_i P_j} = \text{pos}(P_j) - \text{pos}(P_i)$ to mean the vector from $\text{pos}(P_i) = (x_i, y_i)$ to $\text{pos}(P_j) = (x_j, y_j)$. The translation of a path P by a vector $\vec{v} \in \mathbb{Z}^2$, written $P + \vec{v}$, is the path Q such that $|P| = |Q|$ and for all indices $i \in \{0, 1, \dots, |P| - 1\}$, $\text{pos}(Q_i) = \text{pos}(P_i) + \vec{v}$ and $\text{type}(Q_i) = \text{type}(P_i)$.

2.3 Intersections

If two paths, or two assemblies, or a path and an assembly, share a common position we say that they *intersect* at that position. Furthermore, we say that two paths, or two assemblies, or a path and an assembly, *agree* on a position if they both place the same tile type at that position and *conflict* if they place a different tile type at that position. We say that a path P is *fragile* to mean that there is a producible assembly α that conflicts with P . Intuitively, if we grow α first, then there is at least one tile that P cannot place. In directed tile assembly systems, which are the subject of our main result, since the terminal assembly is unique there are no fragile paths in $\mathbf{P}[\alpha]$.

Let P and Q be two paths. We say that Q *grows from P* at index i , if the only intersection between Q and P occurs at $\text{pos}(Q_0) = \text{pos}(P_i)$ and is an agreement. Note that if α is the terminal assembly of some tile assembly system \mathcal{T} , and $P \in \mathbf{P}[\alpha]$, the assertions “ Q grows from P ” or “ Q is an arc of P ” do not imply that $Q \in \mathbf{P}[\alpha]$, since Q might conflict with the seed. We say that Q is an *arc of P* between indices $i < j$ if and only if the only two intersections between Q and P , which occur at $\text{pos}(Q_0) = \text{pos}(P_i)$ and $\text{pos}(Q_{|Q|-1}) = \text{pos}(P_j)$ are both agreements and neither Q nor Q^\leftarrow are subpaths of P ⁴. The *width of an arc Q of P* is defined by $|j - i|$.

2.4 Pumping a path, possibly in both directions

Next, for a path P , we define sequences of points and tile types (not necessarily a path, since these sequences might not be simple) called the *pumping of P* or the *bi-pumping of P* :

► **Definition 4** (Infinite and bi-infinite pumping of P). *Let $\mathcal{T} = (T, \sigma, 1)$ be a tile assembly system and a path P of length at least 2, such that $\text{type}(P_0) = \text{type}(P_{|P|-1})$. We say that the “infinite pumping of P ”, denoted by $(P)^*$, is the infinite sequence \bar{q} of elements from $\mathbb{Z}^2 \times T$ defined by:*

$$\bar{q}_k = P_{k \bmod (|P|-1)} + \left\lfloor \frac{k}{|P|-1} \right\rfloor \overrightarrow{P_0 P_{|P|-1}} \text{ for } k \in \mathbb{N}$$

We say that the “bi-infinite pumping of P ”, denoted by $^(P)^*$, is the bi-infinite sequence \bar{q} of elements from $\mathbb{Z}^2 \times T$ defined by:*

$$\bar{q}_k = P_{k \bmod (|P|-1)} + \left\lfloor \frac{k}{|P|-1} \right\rfloor \overrightarrow{P_0 P_{|P|-1}} \text{ for } k \in \mathbb{Z}$$

⁴ The condition that neither Q nor Q^\leftarrow are subpaths of P is only required when $|Q| = 2$, to avoid the cases where $j = i + 1$ or $j = i - 1$.

In this article, we will only consider cases where \bar{q} is simple, *i.e.* where for any $s < t$, if $P + s\overrightarrow{P_0P_{|P|-1}}$ intersects with $P + t\overrightarrow{P_iP_j}$, then $t = s + 1$ and the only intersection is an agreement between $P_0 + t\overrightarrow{P_0P_{|P|-1}}$ and $P_{|P|-1} + s\overrightarrow{P_0P_{|P|-1}}$. A sufficient condition for this is that the only intersection between P and $P + \overrightarrow{P_0P_{|P|-1}}$ is an agreement between $P_0 + \overrightarrow{P_0P_{|P|-1}}$ and $P_{|P|-1}$ (folklore, see [19] for example). If this condition is satisfied then P is called a *good candidate* and $(P)^*$ and $*(P)^*$ are both paths. Note that, for all $k \in \mathbb{N}$ (resp. $k \in \mathbb{Z}$), we have $(P)^*_{k+|P|-1} = (P)^*_k + \overrightarrow{P_0P_{|P|-1}}$ (resp. $*(P)^*_{k+|P|-1} = *(P)^*_k + \overrightarrow{P_0P_{|P|-1}}$).

► **Definition 5** (Pumpable path). *Let $\mathcal{T} = (T, \sigma, 1)$ be a directed tile assembly system and let α be its unique terminal assembly. We say that a good candidate P is pumpable if $(P)^* \in \mathbf{P}[\alpha]$ and bi-pumpable if $*(P)^* \in \mathbf{P}[\alpha]$. A good candidate that is pumpable but not bi-pumpable is called simply pumpable.*

An ultimately periodic path P can be written as $Q(R)^*$ where Q is a finite path and R is a good candidate. Q is called the *transient* part of P and $(R)^*$ is called the *periodic* part of P .

In our context, we will use the following version of the pumping lemma of [19] where there are no fragile path and the bound is replaced by a generic function $f(x, y)$ where x is the number of tile types and y is the size of the seed (the bound computed in [19] might not be optimal, and could be improved independently of the results presented here).

► **Theorem 6.** *There exists a function $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ such that for any directed tile assembly system $\mathcal{T} = (T, \sigma, 1)$ and any of its producible path P , if P has vertical height or horizontal width at least $f(|T|, |\sigma|)$, then there exist $0 \leq i < j \leq |P| - 1$ such that $P_{i,\dots,j}$ is pumpable.*

Here is the pumpability conjecture which will be a corollary of our result and which was stated in the study of [8], note that we do not consider here that the size of the seed could be reduced to 1 and we have to take it into account.

► **Theorem 7.** *There exists a function $f' : \mathbb{N}^2 \rightarrow \mathbb{N}$ such that for any directed tile assembly system $\mathcal{T} = (T, \sigma, 1)$ and a path P of α , if $|P| \geq f'(|T|, |\sigma|)$, then there exist $0 \leq i < j \leq |P| - 1$ such that for all $\ell \in \mathbb{N}$ either $P_{i,\dots,j} + \ell\overrightarrow{P_iP_j}$ or $P_{i,\dots,j} - \ell\overrightarrow{P_iP_j}$ is in α .*

3 Proof of our main theorem

3.1 Roadmap

An assembly α is \vec{v} -periodic if it is invariant by the translation of vector \vec{v} , *i.e.* $\alpha + \vec{v} = \alpha$. We say that an assembly α is bi-periodic if there exist two non-colinear vectors \vec{u} and \vec{v} such that α is \vec{u} -periodic and \vec{v} -periodic. An assembly is simply periodic if it is not bi-periodic and if there exists a vector \vec{v} such that α is \vec{v} -periodic. Assemblies that are neither bi-periodic nor simply periodic are called *nonperiodic*. Then, like in the original paper [8], we decompose terminal assemblies into four classes: finite, infinite with/without comb, periodic with/without comb and bi-periodic.

The complexity of a finite terminal assembly α of a tile assembly system $\mathcal{T} = (T, \sigma, 1)$ is 0. Moreover, the pumping lemma (Theorem 6) implies that α fits in a square of width $2f(|T|, |\sigma|) + |\sigma|$ and thus its size is bounded by $4(f(|T|, |\sigma|) + |\sigma|)^2$. In this case, the pumpability conjecture holds since we can claim that any path of length at least $4f(|T|, |\sigma|)^2 + |\sigma| + 1$ is pumpable.

To deal with the three remaining cases, we first show in Section 3.3 that “ α is \vec{v} -periodic” is equivalent of “there exists a bi-pumpable path P in $\mathbf{P}[\alpha]$ such that $\overrightarrow{P_0 P_{|P|-1}} = \vec{v}$ ”. Then, we proceed to characterise bi-periodic terminal assemblies in Section 3.4, the infinite nonperiodic ones in Section 3.5 and finally the simply periodic ones in Section 3.6 (a hybrid case of the two previous ones). Note that due to space constraint, we omit some details, in particular details of the original study [8] and focus on improving/patching it. A more self-contained version of this article is available on arXiv [18].

3.2 Relationship with the pumpability conjecture

The relationship between this result and the *pumpability conjecture* [8] is a bit subtle and deserves to be discussed. Indeed, the original statement of the conjecture is that in a directed tile assembly system, any *part of a path* long enough to have a repeated tile type is pumpable, meaning that this part can be repeated infinitely.

In contrast to that statement, [19] proved a weaker statement, namely that only the *initial* segment (starting from the seed) can be pumped, if that initial segment is long enough⁵.

In this paper, we prove that the weaker statement actually implies the stronger one: indeed, we prove that the only terminal assembly that can be built by a directed system is made of pumped paths. Therefore, we prove that if a system is directed, any path P appearing in the terminal assembly is the concatenation of one, two or three (possibly infinite) fragments of periodic paths, which implies that any long enough segment of P contains at least one full period of one of these periodic paths, which is exactly the original pumpability conjecture.

3.3 Link between periodic assembly and bi-pumpable paths

In this subsection, Corollary 9 and Lemma 10 (see Appendix A for the proof of Lemma 10) show the equivalence between the statement “ α is \vec{v} -periodic” and “there exists a bi-pumpable path P where $\vec{v} = \overrightarrow{P_0 P_{|P|-1}}$ ”. Lemma 11 gives a sufficient condition for a pumpable path to be bi-pumpable. These results and the proofs of this section come from the original paper [8]. We have just reorganize the arguments to show the new stronger Lemma 8 which implies Corollary 9 (the first direction of the equivalence) and is later useful to improve the precision of the characterisations of the different classes. It stipulates that we can grow the same terminal assembly α starting from any tile of $*(P)^*$ as the seed, and that the resulting tile assembly system is also directed. Later, this lemma will allow us to grow and pump paths easily. The proof is by contradiction: assuming the assembly weren’t the same, had we started from a different seed, we show that we can get conflicts, contradicting the assumption that \mathcal{T} is directed.

► **Lemma 8.** *Let $\mathcal{T} = (T, \sigma, 1)$ be a directed tile assembly system and α its unique terminal assembly. If a path $P \in \mathbf{P}[\alpha]$ is bi-pumpable then for any $i \in \mathbb{Z}$, the tile assembly system $(T, *(P)_i^*, 1)$ (i.e. \mathcal{T} , with the seed σ replaced by the assembly made of a single tile defined as $*(P)_i^*$), is directed and its terminal assembly is α .*

Proof. Since $*(P)^*$ is in $\mathbf{P}[\alpha]$, let β be any finite assembly producible by $(T, \sigma, 1)$, such that $*(P)_i^*$ is a tile of β . Since \mathcal{T} is directed, $*(P)^*$ and β cannot possibly conflict, hence $*(P)^* \cup \beta$ is producible by $(T, \sigma, 1)$. Let therefore R be any path producible by $(T, *(P)_i^*, 1)$. If R does

⁵ That result also applies to nondirected tile assembly systems, in which case long paths can be either pumped or blocked, meaning that another assembly can be built first and prevent the path from growing.

not conflict with β nor with $*(P)^*$, then $(R \cup *(P)^* \cup \beta)$ is producible by $(T, \sigma, 1)$, and hence R is in $\mathbf{P}[\alpha]$. For the sake of contradiction suppose that such a conflict exists. We assume without loss of generality that the first such conflict along R happens between $R_{|R|-1}$ (i.e. at the last tile of R , which can always be achieved by considering a prefix of R) and either β or $*(P)^*$. There are two cases:

- If this conflict is with $*(P)_j^*$ for $j \neq i$, then since β and R are finite and $\overrightarrow{P_0 P_{|P|-1}}$ is non-null, there exists $\ell \in \mathbb{Z}$ such that $R + \ell \overrightarrow{P_0 P_{|P|-1}}$ does not intersect with β . Note that $R_0 + \ell \overrightarrow{P_0 P_{|P|-1}} = *(P)_{i+\ell(|P|-1)}^*$ and that $R_{|R|-1} + \ell \overrightarrow{P_0 P_{|P|-1}}$ conflicts with $*(P)_{j+\ell(|P|-1)}^*$. By definition of β and P , the assembly $\gamma = \beta \cup *(P)_{i,\dots,i+\ell(|P|-1)}^*$ (or $\gamma = \beta \cup *(P)_{i+\ell(|P|-1),\dots,i}^*$ if $\ell < 0$) is producible by $(T, \sigma, 1)$. By definition of ℓ , the tile $R_{|R|-1} + \ell \overrightarrow{P_0 P_{|P|-1}}$ is not a tile of β and since $j \neq i$, we have $i + \ell(|P| - 1) \neq j + \ell(|P| - 1)$ thus $R_{|R|-1} + \ell \overrightarrow{P_0 P_{|P|-1}}$ is not a tile of γ . Therefore, the assembly $\gamma \cup (R + \ell \overrightarrow{P_0 P_{|P|-1}})$ is producible by $(T, \sigma, 1)$ and is in conflict with $*(P)^* \in \mathbf{P}[\alpha]$, which is a contradiction.
- Otherwise, this conflict occurs with β . Since β and R are finite and $\overrightarrow{P_0 P_{|P|-1}}$ is not null, there exists $\ell \in \mathbb{N}$ such that neither $\beta + \ell \overrightarrow{P_0 P_{|P|-1}}$ nor $R + \ell \overrightarrow{P_0 P_{|P|-1}}$ intersect with β . Since $*(P)^*$ is $\overrightarrow{P_0 P_{|P|-1}}$ -periodic then $*(P)^*$ does not conflict with $\beta + \ell \overrightarrow{P_0 P_{|P|-1}}$ nor with $R + \ell \overrightarrow{P_0 P_{|P|-1}}$. Then the two assemblies $(\beta \cup *(P)^* \cup (\beta + \ell \overrightarrow{P_0 P_{|P|-1}}))$ and $(\beta \cup *(P)^* \cup (R + \ell \overrightarrow{P_0 P_{|P|-1}}))$ are both producible by $(T, \sigma, 1)$, but these two assemblies conflict, which contradicts the hypothesis that $(T, \sigma, 1)$ is directed.

Thus, any path R producible by $(T, *(P)_i^*, 1)$ is producible by $(T, \sigma, 1)$. Therefore, if two assemblies producible by $(T, *(P)_i^*, 1)$ conflicted, then the same conflict can be achieved in $(T, \sigma, 1)$, contradicting the hypothesis that $(T, \sigma, 1)$ is directed. Thus $(T, *(P)_i^*, 1)$ is directed. The terminal assembly α contains $*(P)_i^*$ therefore, α is the unique terminal assembly of $(T, *(P)_i^*, 1)$. ◀

As a corollary of this result, any path Q that grows on $*(P)^*$ is in $\mathbf{P}[\alpha]$. Moreover, since for any $\ell \in \mathbb{Z}$, $*(P)^* + \ell \overrightarrow{P_0 P_{|P|-1}} = *(P)^*$ then $Q + \ell \overrightarrow{P_0 P_{|P|-1}}$ also grows on $*(P)^*$ and is in $\mathbf{P}[\alpha]$ which leads to the following corollary:

► **Corollary 9.** *Let $\mathcal{T} = (T, \sigma, 1)$ be a directed tile assembly system and let α be the unique terminal assembly of \mathcal{T} . If $P \in \mathbf{P}[\alpha]$ is bi-pumpable then α is $\overrightarrow{P_0 P_{|P|-1}}$ -periodic.*

► **Lemma 10.** *Let $\mathcal{T} = (T, \sigma, 1)$ be a directed tile assembly system and let α be the unique terminal assembly of \mathcal{T} . If α is periodic then there exists a path $P \in \mathbf{P}[\alpha]$ that is bi-pumpable.*

► **Lemma 11.** *Let $\mathcal{T} = (T, \sigma, 1)$ be a directed tile assembly system and let α be the unique terminal assembly of \mathcal{T} . Consider a pumpable path P of $\mathbf{P}[\alpha]$ and a path Q growing on $(P)^*$ at index $i \geq |P| - 1$ such that Q and $Q + \overrightarrow{P_0 P_{|P|+1}}$ intersect then P is bi-pumpable.*

Proof. Since P is in $\mathbf{P}[\alpha]$, there is a finite producible subassembly β of α such that P_0 is a tile of β . For the sake of contradiction suppose that there is a conflict between $*(P)^*$ and β otherwise P would be bi-pumpable.

Let R be the largest prefix of Q which does not intersect with $*(P)^*$ then for all $\ell \in \mathbb{N}$, $R + \ell \overrightarrow{P_0 P_{|P|-1}}$ grows on $(P)^*$. Moreover, if $R \neq Q$ then R still intersects with $R + \overrightarrow{P_0 P_{|P|-1}}$. Indeed, without loss of generality, suppose that Q and $*(P)^*$ agree (the following reasoning does not rely on the tile type) then there exists $j < 0$ such that RP_j is an arc of $*(P)^*$ of width greater than $|P|$. Then RP_j and $P_{j,\dots,i}$ form a cycle which delimits a finite area of the 2D plane. The arc $(RP_j) + \overrightarrow{P_0 P_{|P|-1}}$ starts in $P_{i+|P|-1}$, a tile which is not in the finite area since $i + |P| - 1 > i$, and ends in $P_{j+|P|-1}$, a tile which is in the finite area since $j < j + |P| - 1 < i$. Then $R + \overrightarrow{P_0 P_{|P|-1}}$ must cross R to reach P_j .

Note that $R + \overrightarrow{P_0 P_{|P|-1}}$ intersects with both R and $R + 2\overrightarrow{P_0 P_{|P|-1}}$. Moreover all these intersections are agreements (because for ℓ large enough the translations of these three paths by $\ell\overrightarrow{P_0 P_{|P|-1}}$ do not intersect with β and thus are in $\mathbf{P}[\alpha]$). Then, the assembly $\gamma = R \cup (R_{1,2,\dots,|R|-1} + \overrightarrow{R_0 R_{|R|-1}}) \cup (R + 2\overrightarrow{R_0 R_{|R|-1}})$ is well-defined. By definition of growing, the only intersection between $(P)^*$ and γ is R_0 and $R_0 + 2\overrightarrow{R_0 R_{|R|-1}}$. Thus there exists an arc A growing on $(P)^*$ of width $2(|P| - 1) > |P|$ and such that $\text{asm}(A)$ is a subassembly of γ . By definition of R , for all $\ell \in \mathbb{N}$, the arc $A + \ell\overrightarrow{P_0 P_{|P|-1}}$ also grows on $(P)^*$.

Since β is finite and $\overrightarrow{P_0 P_{|P|-1}}$ is not null, there is an integer $L \in \mathbb{N}$ such that for all $\ell \geq L$, neither $A + \ell\overrightarrow{P_0 P_{|P|-1}}$ nor $\beta + \ell\overrightarrow{P_0 P_{|P|-1}}$ intersect β . Since the width of A is strictly greater than $|P|$, we can find $\ell, \ell' > L$ and $0 < a < b < c$ such that $A + \ell$ is an arc of $(P)^*$ between $(P)_c^*$ and $(P)_a^*$ and there is conflict between $\beta + \ell'\overrightarrow{P_0 P_{|P|-1}}$ and $(P)_b^*$. Then there exists a path S such that $\text{asm}(S)$ is a subassembly of $\beta + \ell'\overrightarrow{P_0 P_{|P|-1}}$, $S_0 = P_0 + \ell'\overrightarrow{P_0 P_{|P|-1}}$ and the only conflict between $(P)^*$ and S occurs between $S_{|S|+1}$ and P_b . By definition of ℓ and ℓ' , the paths $A + \ell\overrightarrow{P_0 P_{|P|-1}}$ and $S_{0,\dots,|S|-2}$ are both in $\mathbf{P}[\alpha]$ and thus cannot conflict. Consider the following assembly $\delta = \beta \cup (P)^* \cup (A + \ell\overrightarrow{P_0 P_{|P|-1}}) \cup S_{0,\dots,|S|-2}$. Removing the tile P_b disconnects $(P)^*$ in two parts, but adding $A + \ell\overrightarrow{P_0 P_{|P|-1}}$ reconnects them ($a < b < c$), and it is therefore possible to remove the tile P_b in δ and to replace it by the tile $S_{|S|-1}$ which can bind with $S_{|S|-2}$ contradicting the hypothesis that \mathcal{T} is directed. \blacktriangleleft

3.4 Characterisation of the bi-periodic terminal assemblies

The characterisation of the bi-periodic terminal assemblies does not rely on the pumping lemma. Thus the result of the original paper [8] still holds for this case. Here is a summary, if the terminal assembly α of a tile assembly system $\mathcal{T} = (T, \sigma, 1)$ is bi-periodic, by Lemma 10 there exists two paths P of Q of $\mathbf{P}[\alpha]$ which are bi-pumpable and such that $\overrightarrow{P_0 P_{|P|-1}}$ is not colinear with $\overrightarrow{Q_0 Q_{|Q|-1}}$. Moreover for all $\ell \in \mathbb{Z}$, $*(P)^* + \ell\overrightarrow{Q_0 Q_{|Q|-1}}$ and $*(Q)^* + \ell\overrightarrow{P_0 P_{|P|-1}}$ are in $\mathbf{P}[\alpha]$. All these paths can be used to “tile the plane” with a periodic grid-like structure (see Figure B.1). By considering the assembly β which is the restriction of α to a “cell” of this grid, we obtain that $\alpha = \bigcup_{\ell, \ell' \in \mathbb{Z}} (\beta + \ell\vec{u} + \ell'\vec{v})$. For the main Theorem 2, α is an assembly of complexity 2 defined by the union of the four assemblies of complexity 2: $\bigcup_{\ell, \ell' \in \mathbb{N}} (\beta + \ell\vec{u} + \ell'\vec{v})$, $\bigcup_{\ell, \ell' \in \mathbb{N}} (\beta + \ell\vec{u} - \ell'\vec{v})$, $\bigcup_{\ell, \ell' \in \mathbb{N}} (\beta - \ell\vec{u} + \ell'\vec{v})$ and $\bigcup_{\ell, \ell' \in \mathbb{N}} (\beta - \ell\vec{u} - \ell'\vec{v})$.

Here we improve this result by introducing paths *without redundancy* where there are no repetition of a tile type along the path (except at its extremities), see Definition 12. Of course, the length of such a path is bounded by $|T| + 1$. Lemma 8 allow us to extract bi-pumpable paths without redundancy from bi-pumpable paths and then the size of the cell (and thus of β) of the periodic grid-like structure becomes bounded by $O(|T|^2)$.

► **Definition 12.** A path P is *without redundancy* if for all $0 \leq i < j \leq |P| - 1$, $\text{type}(P_i) = \text{type}(P_j)$ implies that $i = 0$ and $j = |P| - 1$.

► **Theorem 13.** Let $\mathcal{T} = (T, \sigma, 1)$ be a directed tile assembly system, and let α be its unique terminal assembly. If α is bi-periodic, then there exists an assembly β and two vectors \vec{u} and \vec{v} such that $|\beta| \leq 4|T|^2$ and $\alpha = \bigcup_{\ell, \ell' \in \mathbb{Z}} (\beta + \ell\vec{u} + \ell'\vec{v})$.

Proof. If α is bi-periodic then by Lemma 10, there exists a path P of $\mathbf{P}[\alpha]$ which is bi-pumpable. By definition of a bi-pumpable path, P_0 and $P_{|P|-1}$ have the same tile type, then there exists $0 \leq i < j \leq |P| - 1$ such that the path $R = \overrightarrow{P_{i,\dots,j}}$ is without redundancy. Consider $0 \leq i', j' \leq |P| - 1$ such that $\text{pos}(R_{i'}) = \text{pos}(R_{j'}) + \overrightarrow{R_0 R_{|R|-1}}$, since $\overrightarrow{R_0 R_{|R|-1}}$ is

not null then $i' \neq j'$. By definition, R is in $\mathbf{P}[\alpha]$ and by Lemma 8, we can consider that $R_{|R|-1}$ is the seed, then $R + \overrightarrow{R_0 R_{|R|+1}}$ is in $\mathbf{P}[\alpha]$ and $R_{i'}$ and $R_{j'}$ have the same tile type which implies $i' = 0$ and $j' = |R| - 1$, *i.e.* R is a good candidate. Since we can consider that $R_{|R|-1}$ is the seed then ${}^*(R)^*$ is a bi-infinite path of $\mathbf{P}[\alpha]$ and R is bi-pumpable.

Since α is bi-periodic there exists a non null vector \vec{v} which is not colinear with $\overrightarrow{R_0 R_{|R|-1}}$ and such that α is \vec{v} -periodic. Then ${}^*(R)^* + \vec{v}$ is a path of $\mathbf{P}[\alpha]$ and since R is without redundancy, by a reasoning similar to the one of the previous paragraph, ${}^*(R)^*$ and ${}^*(R)^* + \vec{v}$ cannot intersect. Then we consider the shortest path Q such that $Q_0 = R_0$ and $Q_{|Q|-1} = R_0 + \vec{v} + \ell \overrightarrow{R_0 R_{|R|-1}}$ for some $\ell \in \mathbb{Z}$, *i.e.* $Q_{|Q|-1}$ has the same tile type than R_0 and belongs to ${}^*(R)^* + \vec{v}$. Again, it is possible to find $0 \leq i' \leq j' \leq |Q| - 1$ such that $S = Q_{i', \dots, j'}$ is a good candidate without redundancy but proving that S is bi-pumpable is more tricky. By Lemma 8, we consider that the seed is $Q_0 = R_0$. From this seed, we grow $Q_{1, \dots, j'}$ and then we pump S is both direction until either assembling ${}^*(S)^*$ or to obtain a path S' whose growth was blocked by $Q_{1, \dots, j'}$. In the second case, the path S' is in $\mathbf{P}[\alpha]$ and thus cannot conflict with $Q_{i', \dots, |Q|-1}$ and we obtain a contradiction by using Lemma 8 and considering that the seed is $Q_{|Q|-1}$ (a tile of ${}^*(R)^* + \vec{v}$) this time: from this seed, we grow $Q_{i', \dots, |Q|-1}$ and S' . In this case, $Q_{0, \dots, i'-1}$ is not here to block the growth of S' and at least one more tile can be added, creating a conflict with $Q_{0, \dots, i'-1}$ which is a contradiction. Then S is bi-pumpable and for the sake of contradiction, if $\overrightarrow{S_0 S_{|S|-1}}$ and $\overrightarrow{R_0 R_{|R|-1}}$ are colinear then α is $\overrightarrow{S_0 S_{|S|-1}}$ -periodic and ${}^*(R)^*$ intersect with ${}^*(R)^* + \overrightarrow{S_0 S_{|S|-1}}$, since R is without redundancy then $\overrightarrow{S_0 S_{|S|-1}} = \ell \overrightarrow{R_0 R_{|R|-1}}$ for some $\ell \in \mathbb{N}$. Using the assembly $\text{asm}(Q_{1, \dots, i'}) \cup \text{asm}(Q_{j', \dots, |Q|-1} - \overrightarrow{S_0 S_{|S|-1}})$ we can find a path Q' such that $|Q'| < |Q|$, $Q'_0 = Q_0$ and $Q'_{|Q'|-1} = Q_{|Q|-1} - \ell \overrightarrow{R_0 R_{|R|-1}}$, contradicting the definition of Q .

As explained in the beginning of this section, these two bi-pumpable paths R and S create a periodic grid-like structure and α can be characterised by its restriction to a “cell” of this grid. In our case, the cell is delimited by four paths of length bounded by $|T|$ and thus the size of the assembly is bounded by $4|T|^2$. ◀

Any path P of length $O(|T|^3)$ would have to pass by at least $O(|T|)$ cells of the periodic grid-like structure and thus P must intersect the translations of one bi-pumpable path R at least $|R|$ times, among these intersections two have the same tile type. Using Lemma 10, the subpath of P between these two tiles can be pumped and the pumpability conjecture holds in this case.

3.5 Characterisation of the infinite nonperiodic terminal assemblies

We present here a summary of the analysis relying on the pumpability conjecture of the infinite nonperiodic terminal assemblies done in [8] before explaining how to patch this result when replacing the pumpability conjecture (Theorem 7) by the pumping lemma (Theorem 6).

Note that if α is nonperiodic then Lemma 10 implies that all pumpable paths of $\mathbf{P}[\alpha]$ are simply pumpable. Any nonperiodic assembly can be decomposed in three parts (see Figure B.2 for an example): the first part is a finite assembly which contains the seed, the second part is made of some simply pumpable paths growing from this finite assembly called *combs* used to generate periodic paths called the *backbone* of the combs, and the third part is made of paths growing on the backbone of a comb called the *teeth* of the comb. More formally a comb C is a pumpable path of α which is linked to the seed by a producible path containing no pumpable subpath, the backbone of C is $(C)_{|C|, \dots, +\infty}^*$ and a tooth t is a path growing on the backbone of a comb C . It was shown in [8] that if an infinite ultimately

periodic tooth t is in $\mathbf{P}[\alpha]$ then for any $\ell \in \mathbb{N}$, $t + \ell \overrightarrow{C_0 C_{|C|-1}}$ also grow on the backbone of the comb and also belongs to $\mathbf{P}[\alpha]$. Also, only finite path can grow on the periodic part of a comb. For the simple example of Figure B.2, the path P of Figure B.3 allow us to describe the terminal assembly with a finite amount of information.

Lemma 11 is the key to obtain this result: a tooth cannot intersect with its translation by $\overrightarrow{C_0 C_{|C|-1}}$ otherwise the comb would be bi-pumpable. Moreover, if an infinite ultimately periodic path P grows on the periodic part of an ultimately periodic tooth t then P would either intersect $t + \overrightarrow{C_0 C_{|C|-1}}$ (or $t - \overrightarrow{C_0 C_{|C|-1}}$) and C would be bi-pumpable or P would intersect with one of its copy growing on the periodic part of the tooth t and then the periodic part of the tooth would be bi-pumpable in this case. Thus only finite path can grow on the periodic part of a tooth and the pumpability conjecture (Theorem 7) allows us to bound their size. As stated in [8], the pumpability conjecture is needed only three times: the first time to locate the combs, the second time to create an ultimately periodic tooth and a last time to bound the length of the paths growing on the periodic part of a tooth⁶.

To obtain a similar result we have to explain how to use the pumping lemma instead of the pumpability conjecture. Consider again the finite producible path P of Figure B.3: there are five indices $0 \leq i_1 < j_1 < t < i_2 < j_2 \leq |P| - 1$ such that P_{i_1, \dots, j_1} is a comb, P_k is the first tile of a tooth, P_{i_2, \dots, j_2} is pumpable and belongs to the tooth and $P_{j_2+1, \dots, |P|-1}$ is a path growing on the periodic part of the tooth. The pumping lemma of [19] is able to find one pumpable subpath P_{i_1, \dots, j_1} of P but it may seem too weak to find another, different pumpable subpath P_{i_2, \dots, j_2} and too weak to bound the size of $P_{j_2+1, \dots, |P|-1}$. However, the pumping Lemma of [19] can be applied to $P_{k+1, \dots, |P|-1}$ (where P_t is the first tile of the tooth) considered as a path producible by the directed tile assembly system $(T, \sigma \cup P_{0,1, \dots, k}, 1)$, whose terminal assembly is also α (see Figure B.4). This remark shows that the following result is a direct corollary of the pumping lemma of [19].

► **Corollary 14.** *There exists a function $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ such that for any directed tile assembly system $\mathcal{T} = (T, \sigma, 1)$ and for any producible path P and $0 \leq i \leq |P| - 1$, if $P_{i, \dots, |P|-1}$ has vertical height or horizontal width at least $f(|T|, |\sigma| + i)$, then there exist $i \leq j < k \leq |P| - 1$ such that $P_{j, \dots, k}$ is pumpable.*

The result of [8] still holds with this corollary and this is why the authors of [19] claimed that their result allows to solve the conjecture with the proof sketch of [8]. Nevertheless, to provide a bound on the size of the assemblies needed to characterise α , we need to bound for a comb where the first ultimately periodic tooth in $\mathbf{P}[\alpha]$ appears on the backbone of a comb. Indeed a tooth growing at the beginning of the backbone may be blocked by the seed or a previous assembly and will not belong to $\mathbf{P}[\alpha]$. To locate this tooth, we show that there is an index for any periodic path such that any path growing after this index is in $\mathbf{P}[\alpha]$.

► **Lemma 15.** *Let $\mathcal{T} = (T, \sigma, 1)$ be a directed tile assembly system whose terminal assembly is α . If there is a simply pumpable path P in $\mathbf{P}[\alpha]$ and a producible finite assembly β such that P_0 is a tile of β , then there is an index i such that any path growing on $(P)^*$ at index $j \geq i$ is in $\mathbf{P}[\alpha]$. Moreover, i only depends on $|\beta|$, $|P|$ and $|T|$.*

Proof. Without loss of generality we assume that P_0 is the only intersection between β and $(P)^*$. Let $j = (4|\beta| + 2)(|P| - 1) + 1$ and let γ be the assembly defined as $\gamma = \beta \cup (P)_{0,1, \dots, j}^*$, note that γ is a subassembly of α . Since β is finite and $\overrightarrow{P_0 P_{|P|-1}}$ is not null, there is an

⁶ In the extended version of this article [18], we improve this result and show a more efficient way which avoid using the conjecture for the last case. This result is omitted due to space constraint.

integer $i > j + (|P| - 1)$ such that the distance between any tile of $(P)_{i,i+1,\dots,+\infty}^*$ and any tile of β is at least $f(|T|, |\gamma|) + 1$ (see Theorem 6 for the definition of f). Note that i only depends on $|\beta|$, $|P|$ and $|T|$. See Figure B.5 for an illustration of the following reasoning.

Let Q be a path growing on $(P)^*$ at position P_k with $k \geq i$ (Q is in red in Subfigure a of Figure B.5). For the sake of contradiction, assume that Q is not in $\mathbf{P}[\alpha]$, which implies that Q conflicts with β and by the definition of i the vertical height or the horizontal width of Q is at least $f(|T|, |\gamma|) + 1$. Let $m = \max\{n : Q_{0,\dots,n} \text{ does not intersect with } \beta\}$ and by definition of m , $Q_{0,1,\dots,m}$ is in $\mathbf{P}[\alpha]$.

Consider the finite area A of $2D$ plane (in light red in Subfigure b of Figure B.5), whose border is delimited by $(P)_{0,1,\dots,k}^*$, $Q_{0,1,\dots,m}$ and β . Let R be the translation of Q by $\overrightarrow{\ell P_0 P_{|P|-1}}$ for some $\ell \in \mathbb{N}$ such that R_0 is a tile of $P + \overrightarrow{P_0 P_{|P|-1}}$ (R grows on $(P)^*$ at an index between $|P|$ and $2|P| - 1$). Note that for all $0 \leq \ell \leq 4|\beta|$, the path $R + \overrightarrow{\ell P_0 P_{|P|-1}}$ starts to grow in the finite area A . Since there are at most $4|\beta|$ positions that are neighbors of a tile of β , this implies that if for all $0 \leq \ell \leq 4|\beta|$, the paths $R + \overrightarrow{\ell P_0 P_{|P|-1}}$ intersect with β then there exists $\ell \in \mathbb{N}$ such that $R + \overrightarrow{\ell P_0 P_{|P|-1}}$ and $R + \overrightarrow{(\ell + 1) P_0 P_{|P|-1}}$ intersect each other before intersecting β , which by Lemma 11 would imply that P is bi-periodic. Therefore, there is at least one $0 \leq \ell \leq 4|\beta|$ such that $R + \overrightarrow{\ell P_0 P_{|P|-1}}$ does not intersect β and thus is in $\mathbf{P}[\alpha]$. Moreover $R + \overrightarrow{\ell P_0 P_{|P|-1}}$ grows on γ , thus we can consider that $R + \overrightarrow{\ell P_0 P_{|P|-1}}$ is producible by $(T, \gamma, 1)$ and since the vertical height or the horizontal width of R is at least $f(|T|, |\gamma|)$, by the Pumping Lemma (Theorem 6), we can find an ultimately periodic path S of $\mathbf{P}[\alpha]$ growing on $(P)_{0,1,\dots,(4|\beta|+1)(|P|-1)}^*$ and which does not intersect with γ .

Since S is infinite, it cannot fit into the finite area A and thus S must either intersect $(P)_{j+1,j+2,\dots,k}^*$ or $Q_{0,1,\dots,m}$ (see Subfigure c of Figure B.5). In the first case, a subpath of S is an arc of $(P)^*$ of width at least $j - (4|\beta| + 1)(|P| - 1) > |P|$ and in the second case an arc of $(P)^*$ of width $k - (4|\beta| + 1)(|P| - 1) > |P|$ is a subassembly of $\text{asm}(S) \cup \text{asm}(Q_{0,\dots,m})$. As explained in the proof of Lemma 11, an arc of width at least $|P|$ must intersect with its translation by $\overrightarrow{P_0 P_{|P|-1}}$ and then by Lemma 11, P should be bi-pumpable which is a contradiction. \blacktriangleleft

► **Theorem 16.** *Let $\mathcal{T} = (T, \sigma, 1)$ be a directed tile assembly system, and let α be its unique terminal assembly, if α is nonperiodic then the complexity of α is bounded by 2 and the size of characterisation depend only of $|\sigma|$ and $|T|$.*

Proof. The pumping Lemma (Theorem 6) shows that the restrictions of α to a square of width $2f(|T|, |\sigma|) + |\sigma|$ can contain the seed, all the combs and all the paths linking the seed to the comb, this finite assembly is of complexity 0.

Let C be a comb. Then its backbone is an assembly of complexity 1, which can be characterised by C and $\overrightarrow{C_0 C_{|C|-1}}$. Moreover, let j be the index associated to $(C)^*$ by Lemma B.5 (j depends only of $|T|$ and $|\sigma|$ in this case).

Now, consider a finite tooth t of $\mathbf{P}[\alpha]$ growing on $(C)^*$ at index $i > |C| - 1$. If $i \leq j + |C| - 1$, we add t (an assembly of complexity 0) to α . Else, $i > j + |C| - 1$, and there exists a tooth t' in $\mathbf{P}[\alpha]$ such that $T = T' + \overrightarrow{\ell C_0 C_{|C|-1}}$ for some $\ell \in \mathbb{N}$ and T' grow on $(C)_k^*$ with $j \leq k \leq j + |P| - 1$. Therefore, by adding $\bigcup_{\ell \in \mathbb{N}} (t' + \overrightarrow{\ell C_0 C_{|C|-1}})$ to the resulting assembly, we also add the tooth t , and this union's characterisation has complexity 1.

Now if the tooth t is ultimately periodic, we can apply the same reasoning. In this case, we only need to prove that t and all the paths growing on the periodic part of t form an assembly of complexity 1 (the same reasoning will produce an assembly of complexity 2). Corollary 14 and Lemma 15 allow us to bound the length of the transient part of t , which

depends only on $|\sigma|$ and $|T|$. Any path growing on the periodic part of the tooth is finite and by using the same reasoning with the comb and its finite tooth (which requires using Corollary 14 and Lemma 15 again), we obtain that the tooth t and all the paths growing on its periodic part form an assembly of complexity 1 and the length of the finite path growing on the periodic part of the tooth is bounded by a function depending only of $|\sigma|$ and $|T|$. We do not give the exact characterisation size, since this technique is unlikely to yield a tight bound. ◀

3.6 The simply periodic terminal assembly

In the original paper [8], the simply periodic terminal assembly were not studied in details. By Lemma 10, there exists a path P which is bi-pumpable and then $*(P)^*$ cuts the $2D$ plane into two parts: its left and right side. Some ultimately periodic paths may grow on $*(P)^*$, stay in one of the two sides and behave as the teeth of the previous section. This class of terminal assembly is a mix of the two previous ones.

We go in further details here, as in Section 3.4, we consider bi-pumpable paths without redundancy (see Definition 12). If P and Q are two bi-pumpable paths without redundancy, then the following Lemma (see Appendix A for the proof) shows that, by potentially reversing one of the two paths, we can consider that $\overrightarrow{P_0 P_{|P|-1}} = \overrightarrow{Q_0 Q_{|Q|-1}}$. This remark allow us to introduce an order on the bi-pumpable paths without redundancy of $\mathbf{P}[\alpha]$ in Definition 18.

► **Lemma 17.** *Let $\mathcal{T} = (T, \sigma, 1)$ be a directed tile assembly system and let α be its unique simply periodic terminal assembly, if P and Q are two bi-pumpable paths of $\mathbf{P}[\alpha]$ without redundancy then either $\overrightarrow{P_0 P_{|P|-1}} = \overrightarrow{Q_0 Q_{|Q|-1}}$ or $\overrightarrow{P_0 P_{|P|-1}} = -\overrightarrow{Q_0 Q_{|Q|-1}}$.*

► **Definition 18.** *If P and Q are two bi-pumpable paths without redundancy of $\mathbf{P}[\alpha]$ such that $\overrightarrow{P_0 P_{|P|-1}} = \overrightarrow{Q_0 Q_{|Q|-1}}$, we say P is greater or equal to Q , denoted by $P \geq Q$, if and only if $*(P)^*$ is inside the left-hand side of the (directed) curve defined by $*(Q)^*$ (as in [19], the “left-hand side” is considered as if we were walking on the curve). Moreover, if $*(P)^* \neq *(Q)^*$, we say that P is strictly greater than Q , denoted by $P > Q$.*

Lemma 19 and Lemma 20 aim to show that there is maximum path P^+ and a minimum path P^- for the order defined in 18 which means that there is no infinite sequence $P^{(0)} < P^{(1)} < \dots$. To achieve this goal we show in Lemma 19 that the tile type which appear in a bi-pumpable path P without redundancy can only appear in $*(P)^*$ otherwise α would be bi-periodic. We conclude in Lemma 20 by showing that if there would be an infinite sequence of increasing paths, one of them would have to use again a tile type of a previous path leading to a contradiction.

Lemma 21 implies that the paths growing in the left side of P^+ or in the right side of P^- behave as the teeth of Section 3.5 and we conclude in Lemma 22 by showing that a simply periodic terminal assembly can be described as follow: $*(P^+)^*$ and $*(P^-)^*$ cut the $2D$ plane in three parts, one them is a *stripe* which can be characterised by an assembly of size $O(|T^2|)$, as in the analysis of bi-periodic terminal assembly, while the paths growing in the left side of $*(P^+)^*$ or the right side of $*(P^-)^*$ behave as teeth (Lemma 21), as in the analysis of nonperiodic terminal assembly.

► **Lemma 19.** *Let $\mathcal{T} = (T, \sigma, 1)$ be a directed tile assembly system and let α be its unique terminal assembly. If there is a tile A of α and a bi-pumpable path P without redundancy of $\mathbf{P}[\alpha]$ such that A is not a tile of $*(P)^*$ and such that A and P_0 have the same tile type then α is bi-periodic.*

Proof. If $\overrightarrow{P_0A}$ and $\overrightarrow{P_0P_{|P|-1}}$ are colinear, then we have $\overrightarrow{P_0A} = \ell \overrightarrow{P_0P_{|P|-1}}$ for some $\ell \in \mathbb{Z}$ (see the proof of Lemma 17) and then A is tile of $*(P)^*$ which is a contradiction. If $\overrightarrow{P_0A}$ and $\overrightarrow{P_0P_{|P|-1}}$ are not colinear then consider a path Q of $\mathbf{P}[\alpha]$ such that $Q_0 = P_0$ and $Q_{|Q|-1} = A$. By Lemma 8, we consider that the seed is P_0 and then $Q - \overrightarrow{P_0A}$ is a path of $\mathbf{P}[\alpha]$ which implies that there is no conflict from $Q - \overrightarrow{P_0A}$ and $*(P)^*$. Then from P_0 , the paths Q followed by $*(P)^* + \overrightarrow{P_0A}$ can grow and these two paths are in $\mathbf{P}[\alpha]$. By iterating this reasoning, for all $\ell \in \mathbb{Z}$, the path $Q + \ell \overrightarrow{P_0A}$ is in $\mathbf{P}[\alpha]$ which implies that α is $\overrightarrow{P_0A}$ -periodic by a reasoning similar to the one of Corollary 9. The assembly α would be bi-periodic which is a contradiction. \blacktriangleleft

► **Lemma 20.** *Let $\mathcal{T} = (T, \sigma, 1)$ be a directed tile assembly system whose terminal assembly α is simply periodic. Then there are two paths P^+ and P^- without redundancy that are bi-pumpable, such that P^+ is maximum and P^- is minimum for the order defined in Definition 18.*

Proof. Since α is simply periodic then by Lemma 10, there is a bi-pumpable path in $\mathbf{P}[\alpha]$ and one of its subpath is bi-pumpable without redundancy. Then there is at least one bi-pumpable path $P^{(0)}$ of $\mathbf{P}[\alpha]$ without redundancy.

For the sake of contradiction, suppose that there are more than $|T|^2$ bi-pumpable paths without redundancy of $\mathbf{P}[\alpha]$ such that $P^{(0)} < P^{(1)} < P^{(2)} < \dots < P^{(|T|^2)}$. Let $A^{(0)}$ be any tile of $P^{(0)}$. Since the length of a path without redundancy is at least 2 and less than $|T| + 1$ then the case where for some $0 \leq i \leq |T|^2$, $|P^{(i+1)}| < |P^{(i)}|$ occurs at most $|T|$ times consecutively. Since we have $|T|^2$ paths, the case where $|P^{(i+1)}| \geq |P^{(i)}|$ occurs at least $|T|$ times. In such a case, since $*(P^{(i)})^* \neq *(P^{(i+1)})^*$ there exists a tile of $P^{(i+1)}$ which is not a tile of $*(P^{(i)})^*$ and which is in the left side of $*(P^{(i)})^*$ and thus this tile is not a tile of $*(P^{(0)})^*, *(P^{(1)})^*, \dots, *(P^{(i)})^*$. Then we can create a sequence of tiles $A^{(0)}, A^{(1)}, \dots, A^{(|T|+1)}$ such that each tile of the sequence belongs to a bi-pumpable path without redundancy whose pumping does not pass by the other tiles. Two tiles of this sequence share a common tile type which by Lemma 19 means that α is bi-periodic. Then the sequence $P^{(0)} < P^{(1)} < P^{(2)} < \dots$ is finite and let P^+ be the last bi-pumpable path without redundancy of this sequence and P^+ is maximum. A similar reasoning shows that there is a minimum path P^- . \blacktriangleleft

► **Lemma 21.** *Let $\mathcal{T} = (T, \sigma, 1)$ be a directed tile assembly system whose terminal assembly α is simply periodic. If there is a bi-pumpable path $P \in \mathbf{P}[\alpha]$ without redundancy, and an arc Q that grows in the left (resp. right) side of P , then there exists a bi-pumpable path $R \in \mathbf{P}[\alpha]$ without redundancy such that $R > P$ ($R < P$).*

Proof. If Q and $Q + \overrightarrow{P_0P_{|P|-1}}$ intersect, both paths are in $\mathbf{P}[\alpha]$ (since α is $\overrightarrow{P_0P_{|P|-1}}$ -periodic by Corollary 9) and then the tiles at their intersection have the same tile type. Since $\overrightarrow{P_0P_{|P|-1}}$ is not null, there are two indices $0 \leq i < j \leq |Q| - 1$ such that $R = Q_{i,i+1,\dots,j}$ is a path without redundancy. By Lemma 8 and since Q_0 and $Q_{|Q|-1}$ are tiles of $*(P)^*$, we can consider that either $Q_{0,\dots,i}$ or $Q_{j,\dots,|Q|-1}$ is the seed and if R would not be bi-pumpable, we can obtain a conflict with one of these two paths as done in the proof of Theorem 13. By Lemma 17, we can consider that $\overrightarrow{P_0P_{|P|-1}} = \overrightarrow{R_0R_{|R|-1}}$ and hence, since Q grows in the left side of $*(P)^*$, R and $*(R)^*$ are in the left side of $*(P)^*$. Therefore, $R \geq P$ but $*(P)^* = *(R)^*$ would contradict the definition of an arc. Hence, $R > P$.

In the second case, Q and $Q + \overrightarrow{P_0P_{|P|-1}}$ do not intersect which implies that the width of Q is less than $|P| - 1$ (an arc of width greater than $|P|$ intersect with its translation by $\overrightarrow{P_0P_{|P|-1}}$). Without loss of generality we can assume that there are two indices $0 \leq i < j \leq |P| - 1$ such that Q starts at P_i and ends at P_j . Let R be the path defined by

$R = P_{0,1,\dots,i}Q_{1,2,\dots,|Q|-2}P_{j,j+1,\dots,|P|-1}$. By definition of an arc, $R \neq P$. If R has a redundancy, there are two integers $0 \leq k \leq |P| - 1$ and $1 \leq k' \leq |Q| - 2$ such that $\text{type}(P_k) = \text{type}(Q_{k'})$. By definition of an arc, $Q_{k'}$ is not a tile of $*(P)^*$, and hence, by Lemma 19, α is bi-periodic, which contradicts the hypotheses of this lemma. Therefore, R is without redundancy, moreover since Q does not intersect with $Q + \overrightarrow{P_0 P_{|P|-1}}$ then R is a good candidate. Since $\overrightarrow{R_0 R_{|R|-1}} = \overrightarrow{P_0 P_{|P|-1}}$ and α is $\overrightarrow{P_0 P_{|P|-1}}$ -periodic then R is bi-pumpable. By definition of R and Q , R and $*(R)^*$ is in the left side of $*(P)^*$ and by definition of an arc $R \neq P$, thus $R > Q$. ◀

► **Theorem 22.** *Let $\mathcal{T} = (T, \sigma, 1)$ be a directed tile assembly system, and let α be its unique terminal assembly, if α is simply periodic then its complexity is bounded by 2 and the size of characterisation depend only of $|\sigma|$ and $|T|$.*

Proof. By Lemma 20, there exists two bi-pumpable paths without redundancy P^+ and P^- of $\mathbf{P}[\alpha]$ which are maximum and minimum for the order defined in 18. By Lemma 21, the paths growing in left side of $*(P^+)^*$ cannot intersect with their translation by $\overrightarrow{P_0^+ P_{|P^+|-1}^+}$, then they behave as the teeth of Section 3.5. Let C be the union of a ultimately periodic tooth t and the paths growing on its periodic part, then C admits a characterisation of complexity 1 (see the proof of Theorem 16). By adding all the translations by $\overrightarrow{\ell P_0^+ P_{|P^+|-1}^+}$ for $\ell \in \mathbb{N}$ to a copy of C and adding all the translations by $\overrightarrow{-\ell P_0^+ P_{|P^+|-1}^+}$ for $\ell \in \mathbb{N}$ to an other copy, all the translations of t and the paths growing on its periodic part are an assembly of complexity 2. Thus, all the paths growing on left side of $*(P^+)^*$ is an assembly of complexity 2. A similar reasoning holds for the right side of $*(P^-)^*$.

Consider the shortest path between a tile of $*(P^+)^*$ and a tile of $*(P^-)^*$ (Q is empty if the two paths intersect). If there exist $0 \leq i < j \leq |Q| - 1$ such that $R = Q_{i,\dots,j}$ is without redundancy then R is bi-pumpable (with a reasoning similar to the proof of 13, since both Q_0 and $Q_{|Q|-1}$ belong to bi-pumpable path). By Lemma 17, we can consider that $\overrightarrow{P_0^+ P_{|P^+|-1}^+} = \overrightarrow{R_0 R_{|R|-1}} = \overrightarrow{P_0^- P_{|P^-|-1}^-}$ and we can find a path strictly shorter than Q which is a subassembly of $\text{asm}(Q_{0,\dots,i}) \cup \text{asm}(Q_{j,\dots,|Q|-1} - \overrightarrow{R_0 R_{|R|-1}})$ with the same property than Q which is a contradiction. The paths P^+, Q, P^- and $Q + \overrightarrow{P_0^+ P_{|P^+|-1}^+}$ define a cycle and let the finite assembly β be the restriction of α to the finite area of this cycle. The restriction of α to the “stripe” defined by the intersection of the right side of P^+ and the left side of P^- is $\bigcup_{\ell \in \mathbb{Z}} \beta + \overrightarrow{\ell P_0^+ P_{|P^+|-1}^+}$ which is an assembly of complexity 1. Note that if Q is not the empty path then by Lemma 19, P^+, P^- and Q cannot share a common tile type except at the extremities of Q , then the periodic assembly $*(P^+)^* \cup *(P^-)^* \cup (\bigcup_{\ell \in \mathbb{Z}} Q + \overrightarrow{\ell P_0^+ P_{|P^+|-1}^+})$ that delimits the stripe can be described by a path of length less than $|T|$.⁷ ◀

References

- 1 B. Behsaz, J. Mañuch, and L. Stacho. Turing universality of step-wise and stage assembly at Temperature 1. In *DNA18: Proc. of International Meeting on DNA Computing and Molecular Programming*, volume 7433 of *LNCS*, pages 1–11. Springer, 2012.
- 2 L. Ceze, J. Nivala, and K. Strauss. Molecular digital data storage using DNA. *Nat Rev Genet*, 20(8):456–466, August 2019.

⁷ The same reasoning could be done for the bi-periodic terminal assembly and the 2D periodic grid can be described by a path of length less than $|T|$: there is no other way than to hardcode the 2D periodic grid.

- 3 Matthew Cook, Yunhui Fu, and Robert T. Schweller. Temperature 1 self-assembly: deterministic assembly in 3D and probabilistic assembly in 2D. In *SODA: Proceedings of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 570–589, 2011. Arxiv preprint: [arXiv:0912.0027](#).
- 4 Erik D. Demaine, Martin L. Demaine, Sándor P. Fekete, Matthew J. Patitz, Robert T. Schweller, Andrew Winslow, and Damien Woods. One tile to rule them all: Simulating any tile assembly system with a single universal tile. In *ICALP: Proceedings of the 41st International Colloquium on Automata, Languages, and Programming*, volume 8572 of *LNCS*, pages 368–379. Springer, 2014. Arxiv preprint: [arXiv:1212.4756](#).
- 5 Erik D. Demaine, Matthew J. Patitz, Trent A. Rogers, Robert T. Schweller, Scott M. Summers, and Damien Woods. The two-handed tile assembly model is not intrinsically universal. In *ICALP: Proceedings of the 40th International Colloquium on Automata, Languages, and Programming*, volume 7965 of *LNCS*, pages 400–412. Springer, 2013. Arxiv preprint: [arXiv:1306.6710](#).
- 6 David Doty, Jack H. Lutz, Matthew J. Patitz, Robert T. Schweller, Scott M. Summers, and Damien Woods. The tile assembly model is intrinsically universal. In *FOCS: Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science*, pages 439–446. IEEE, 2012. Arxiv preprint: [arXiv:1111.3097](#).
- 7 David Doty, Jack H. Lutz, Matthew J. Patitz, Scott M. Summers, and Damien Woods. Intrinsic universality in self-assembly. In *STACS: Proceedings of the 27th International Symposium on Theoretical Aspects of Computer Science*, pages 275–286, 2009. Arxiv preprint: [arXiv:1001.0208](#).
- 8 David Doty, Matthew J. Patitz, and Scott M. Summers. Limitations of self-assembly at temperature 1. *Theoretical Computer Science*, 412(1–2):145–158, 2011. Arxiv preprint: [arXiv:0903.1857v1](#).
- 9 Sándor P. Fekete, Jacob Hendricks, Matthew J. Patitz, Trent A. Rogers, and Robert T. Schweller. Universal computation with arbitrary polyomino tiles in non-cooperative self-assembly. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, pages 148–167. SIAM, 2015. [arXiv:1408.3351](#).
- 10 David Furcy and Scott M Summers. Optimal self-assembly of finite shapes at temperature 1 in 3D. *Algorithmica*, 80(6):1909–1963, 2018.
- 11 David Furcy, Scott M Summers, and Christian Wendlandt. New bounds on the tile complexity of thin rectangles at temperature-1. In *DNA25: International Conference on DNA Computing and Molecular Programming*, pages 100–119. Springer, 2019.
- 12 Oscar Gilbert, Jacob Hendricks, Matthew J Patitz, and Trent A Rogers. Computing in continuous space with self-assembling polygonal tiles. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, pages 937–956. SIAM, 2016. Arxiv preprint: [arXiv:1503.00327](#).
- 13 Natasa Jonoska and Daria Karpenko. Active tile self-assembly, part 1: Universality at temperature 1. *Int. J. Found. Comput. Sci.*, 25(2):141–164, 2014. [doi:10.1142/S0129054114500087](#).
- 14 Ján Maňuch, Ladislav Stacho, and Christine Stoll. Two lower bounds for self-assemblies at Temperature 1. *Journal of Computational Biology*, 17(6):841–852, 2010.
- 15 Pierre-Étienne Meunier. Non-cooperative algorithms in self-assembly. In *UCNC: Unconventional Computation and Natural Computation*, volume 9252 of *LNCS*, pages 263–276. Springer, 2015.
- 16 Pierre-Étienne Meunier, Matthew J. Patitz, Scott M. Summers, Guillaume Theyssier, Andrew Winslow, and Damien Woods. Intrinsic universality in tile self-assembly requires cooperation. In *SODA: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 752–771, 2014. Arxiv preprint: [arXiv:1304.1679](#).
- 17 Pierre-Étienne Meunier and Damien Regnault. Non-cooperatively assembling large structures. In *DNA Computing and Molecular Programming - 25th International Conference, DNA 25, Seattle, WA, USA, August 5-9, 2019, Proceedings*, 2019.

- 18 Pierre-Étienne Meunier and Damien Regnault. On the directed tile assembly systems at temperature 1. *arXiv preprint*, 2020. In submission with title “Directed non-cooperative tile assembly is decidable”. [arXiv:2011.09675](#).
- 19 Pierre-Étienne Meunier, Damien Regnault, and Damien Woods. The program-size complexity of self-assembled paths. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 727–737. ACM, 2020. Arxiv preprint: [arXiv:2002.04012 \[cs.CC\]](#). doi:10.1145/3357713.3384263.
- 20 Pierre-Étienne Meunier and Damien Woods. The non-cooperative tile assembly model is not intrinsically universal or capable of bounded Turing machine simulation. In *STOC: Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 328–341, Montreal, Canada, 2017. ACM. Arxiv preprint with full proofs: [arXiv:1702.00353v2 \[cs.CC\]](#).
- 21 Dionis Mineev, Christopher M. Wintersinger, Anastasia Ershova, and William M. Shih. Robust nucleation control via crisscross polymerization of dna slats. *bioRxiv*, 2019. doi:10.1101/2019.12.11.873349.
- 22 Matthew J. Patitz, Robert T. Schweller, and Scott M. Summers. Exact shapes and Turing universality at Temperature 1 with a single negative glue. In *DNA 17: Proceedings of the Seventeenth International Conference on DNA Computing and Molecular Programming*, LNCS, pages 175–189. Springer, 2011. Arxiv preprint: [arXiv:1105.1215](#).
- 23 Lulu Qian, Erik Winfree, and Jehoshua Bruck. Neural network computation with DNA strand displacement cascades. *Nature*, 475(7356):368–372, 2011.
- 24 Paul W. K. Rothemund. Folding DNA to create nanoscale shapes and patterns. *Nature*, 440(7082):297–302, March 2006. doi:10.1038/nature04586.
- 25 Paul W. K. Rothemund and Erik Winfree. The program-size complexity of self-assembled squares (extended abstract). In *STOC: Proceedings of the thirty-second annual ACM Symposium on Theory of Computing*, pages 459–468, Portland, Oregon, 2000. ACM. doi:10.1145/335305.335358.
- 26 Paul W.K. Rothemund, Nick Papadakis, and Erik Winfree. Algorithmic self-assembly of DNA Sierpinski triangles. *PLoS Biology*, 2(12):2041–2053, 2004.
- 27 David Soloveichik and Erik Winfree. Complexity of self-assembled shapes. *SIAM Journal on Computing*, 36(6):1544–1569, 2007. doi:10.1137/S0097539704446712.
- 28 Hao Wang. Proving theorems by pattern recognition – II. *The Bell System Technical Journal*, XL(1):1–41, 1961.
- 29 Erik Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, California Institute of Technology, June 1998.
- 30 Bernard Yurke, Andrew J Turberfield, Allen P Mills, Friedrich C Simmel, and Jennifer L Neumann. A DNA-fuelled molecular machine made of DNA. *Nature*, 406(6796):605–608, 2000.

A Omitted Proof

Here is the proof of Lemma 10.

Proof. By hypothesis, there exists a non-null vector \vec{v} such that α is \vec{v} -periodic. Let A be any tile of α , then $A + \vec{v}$ is also a tile of α (because α is \vec{v} -periodic) and there exists a finite path $P \in \mathbf{P}[\alpha]$ such that $P_0 = A$ and $P_{|P|-1} = A + \vec{v}$.

Let Q be the shortest such path of $\mathbf{P}[\alpha]$, i.e. the shortest path such that $Q_0 + s\vec{v} = Q_{|Q|-1}$ for $s = 1$ or $s = -1$. The path Q exists since P itself satisfies the criterion $P + s\vec{v} = P$ for $s = 1$. Since α is \vec{v} -periodic, then for $\ell \in \mathbb{Z}$, $Q + \ell\vec{v}$ is in $\mathbf{P}[\alpha]$. There are two cases:

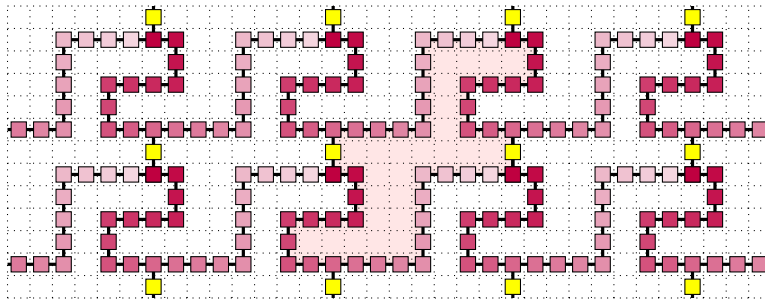
- If Q and $Q + \vec{v}$ intersect only in one point at their ends (either at $Q_{|Q|-1}$ and $Q_0 + \vec{v}$ or at Q_0 and $Q_{|Q|-1} + \vec{v}$, depending on whether $s = 1$ or $s = -1$), then Q is a good candidate and $*(Q)^*$ is a simple bi-infinite path, and is in $\mathbf{P}[\alpha]$, meaning that Q is bi-pumpable, which is our conclusion (with $P = Q$).

- Otherwise, there exists another intersection between Q and $Q + \vec{v}$, i.e. there exists $0 \leq i, j \leq |Q| - 1$ such that $Q_i = Q_j + \vec{v}$ and if $j > i$ (resp. $i > j$) then $(i, j) \neq (0, |Q| - 1)$ (resp. $(j, i) \neq (0, |Q| - 1)$). Thus, $Q_{i, \dots, j}$ (resp. $Q_{j, \dots, i}$) contradicts our assumption that Q is the shortest path intersecting $Q + s\vec{v}$. ◀

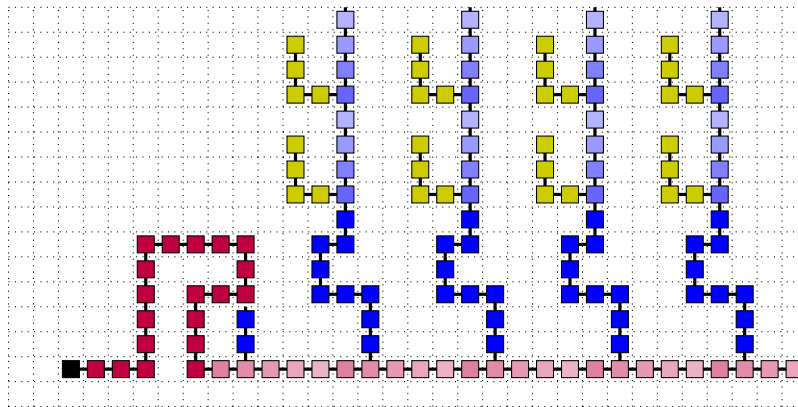
Here is the proof of Lemma 17.

Proof. By Corollary 9, α is $\overrightarrow{P_0 P_{|P|-1}}$ -periodic and $\overrightarrow{Q_0 Q_{|Q|-1}}$ -periodic then both vectors are colinear since α is simply periodic. Then $*(P)^* + \overrightarrow{Q_0 Q_{|Q|-1}}$ intersects with $*(P)^*$ and since P is without redundancy, we have $\overrightarrow{Q_0 Q_{|Q|-1}} = \ell \overrightarrow{P_0 P_{|P|-1}}$ for some $\ell \in \mathbb{Z}$. Similarly $\overrightarrow{P_0 P_{|P|-1}} = \ell' \overrightarrow{Q_0 Q_{|Q|-1}}$ for some $\ell' \in \mathbb{Z}$ and then either $\overrightarrow{P_0 P_{|P|-1}} = \overrightarrow{Q_0 Q_{|Q|-1}}$ or $\overrightarrow{P_0 P_{|P|-1}} = -\overrightarrow{Q_0 Q_{|Q|-1}}$. ◀

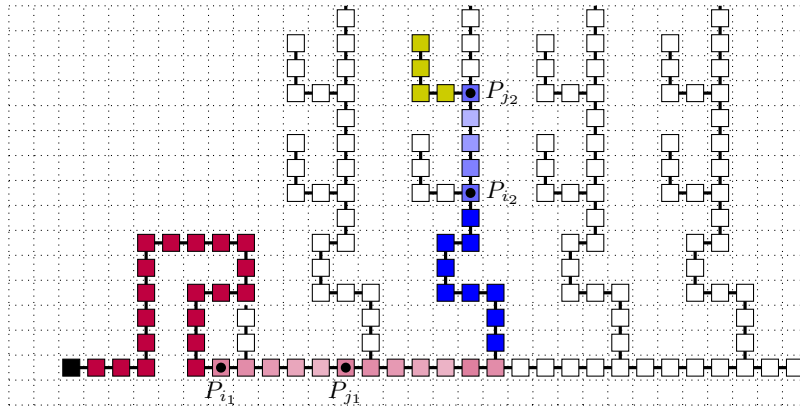
B Omitted Figures



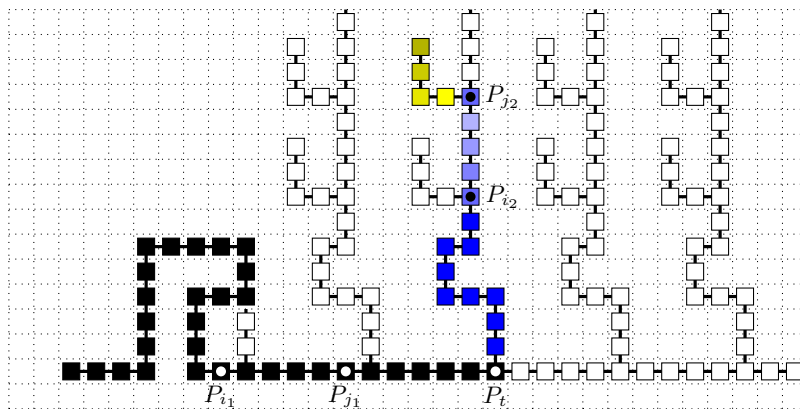
■ **Figure B.1** Illustration of a bi-pumpable terminal assembly. We consider two bi-pumpable paths P and Q such that $\overrightarrow{P_0 P_{|P|-1}}$ is not colinear with $\overrightarrow{Q_0 Q_{|Q|-1}}$. The 2D plane is filled by these paths and α can be characterised by its restriction to the red area and the vectors $\overrightarrow{P_0 P_{|P|-1}}$ and $\overrightarrow{Q_0 Q_{|Q|-1}}$.



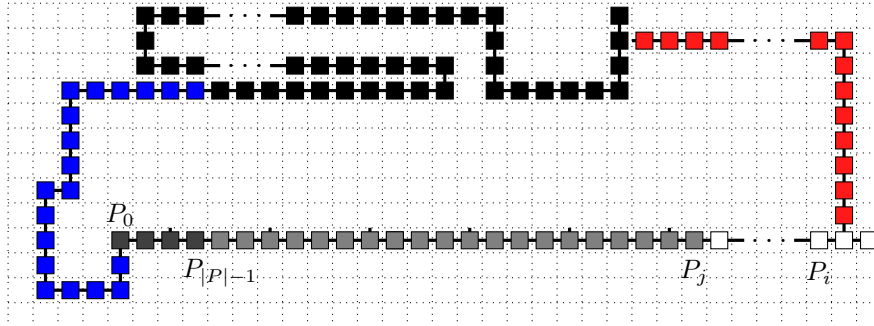
■ **Figure B.2** An aperiodic terminal assembly α : the seed is in black, a comb and its backbone are in light red, a finite path linking the seed to the comb is in dark red, a comb and its translation are in blue (dark blue for the transient part and light blue for the periodic part) and a finite path growing on the periodic part of the tooth is in yellow. Note that the first comb cannot fully grow.



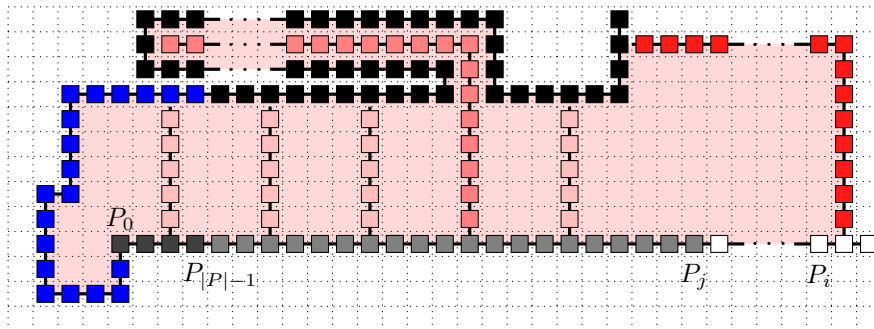
■ **Figure B.3** The producible path P (the tiles which does not belong to P are in white).



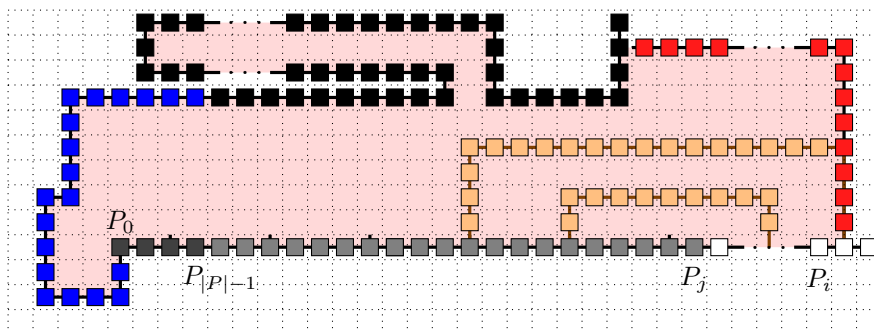
■ **Figure B.4** The path $P_{[t+1, \dots, |P|-1]}$ is producible by $(T, \sigma \cup P_{[0, \dots, t]})$. The terminal assembly is still α , the new seed is in black.



a) The seed σ is in black, P (in dark grey) is simply pumpable and the pumping of P is in light grey up to index j , and in white afterwards. Assembly β is the union of the seed and the blue path. γ is the union of β and the gray paths. Path $Q_{0,\dots,m}$ is in red and intersects the seed.



b) All the translations of Q growing on the light gray part of $(P)^*$ start in the red area of the grid. If β blocks them all, some of the translations of Q must intersect with each other before intersecting β . Thus, one of the translations must fully grow.



c) The path S is ultimately periodic and grows on $(P)^*_{|P|,\dots,j-|P|-1}$. There is two examples of S in orange, one leaves the red area by intersecting $Q_{0,\dots,m}$ and the other one by intersecting $P_{j+1,\dots,k}$. In both cases, we can find an arc of $(P)^*$ of width at least $|P|$.

■ **Figure B.5** Illustration of Proof 15.

Molecular Machines from Topological Linkages

Keenan Breik

The University of Texas at Austin, TX, USA

Austin Luchsinger

The University of Texas at Austin, TX, USA

David Soloveichik

The University of Texas at Austin, TX, USA

Abstract

Life is built upon amazingly sophisticated molecular machines whose behavior combines mechanical and chemical action. Engineering of similarly complex nanoscale devices from first principles remains an as yet unrealized goal of bioengineering. In this paper we formalize a simple model of mechanical motion (mechanical linkages) combined with chemical bonding. The model has a natural implementation using DNA with double-stranded rigid links, and single-stranded flexible joints and binding sites. Surprisingly, we show that much of the complex behavior is preserved in an idealized topological model which considers solely the graph connectivity of the linkages. We show a number of artifacts including Boolean logic, catalysts, a fueled motor, and chemo-mechanical coupling, all of which can be understood and reasoned about in the topological model. The variety of achieved behaviors supports the use of topological chemical linkages in understanding and engineering complex molecular behaviors.

2012 ACM Subject Classification Theory of computation → Models of computation; Theory of computation → Computational geometry

Keywords and phrases chemical computation, mechanical computation, bioengineering, models of biochemistry, molecular machines, mechanical linkages, generic rigidity

Digital Object Identifier 10.4230/LIPIcs.DNA.27.7

Funding This work was supported by NSF grant CCF-1901025.

Acknowledgements We thank Tosan Omabegho for introducing us to chemical linkages and for extensive discussions.

1 Introduction

Living cells, by far, show the most complex chemical behavior known. Their primary functional parts are molecular machines that derive their behavior from internal mechanical motion [2]. There are large gaps in our understanding of how function originates from mechanical reconfiguration, and engineering such machines from scratch stands as the grand challenge of bioengineering.

Even simple chemistry involves intricate physics, which makes it a challenge to model molecular machines. But we may not need that intricacy to capture their rich behavior. It would be interesting if the essence of their behavior could be reproduced by a simple mechanical model. But how simple can the model be, what features should it include, and how should its parts be assembled?

We look to linkages, a simple tool from mechanical engineering. A linkage is a set of rigid rods (called links) connected at rotary joints. This well-studied tool has been shown to be capable of very complex motion, such as tracing any arbitrary curve [9, 8]. Reference [4] provides an excellent overview of the capabilities of linkages. Linkages have also found use in studying biological mechanisms. In reference [14], linkages are used to model the mechanical behavior of proteins. Most closely related to this paper, however, is Omabegho's work



© Keenan Breik, Austin Luchsinger, and David Soloveichik;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on DNA Computing and Molecular Programming (DNA 27).

Editors: Matthew R. Lakin and Petr Petr Šulc; Article No. 7; pp. 7:1–7:20

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

introducing chemical linkages where joints can form chemical bonds [12]. There, chemical linkages are used to model the role of allostery in enzyme behavior. Motivated by the above work, we reimagine chemical linkages as a basic model for rich chemical behavior, and explore the variety of behaviors that can be captured by the model.

Linkages seem like a minimal mechanical model of biochemistry, but we show how to simplify them further and still produce complex behavior. The link lengths in a linkage determine how the linkage can move. This in turn controls which joints can chemically bind. Surprisingly, in many cases which joints can bind could be fully determined by just the topology of the underlying graph. Focusing on simple graphs makes systems easier to design and analyze.

On top of having interesting behavior theoretically, chemical linkages could also lead to real molecular machines, like artificial enzymes. Their rich behavior comes from simple parts: links, joints, binding sites. These may be possible to build directly. Double-stranded DNA may be rigid enough to act as a link. Single-stranded DNA might implement a joint as a so-called compliant mechanism. Orthogonal DNA sequences could act as binding sites. However, this work does not further explore practical implementation.

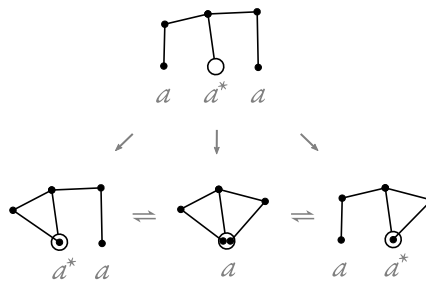
The contributions of this work are as follows. We formalize chemical linkages, previously described only informally in [12]. We also introduce a new topological model, which bases bond formation on simple graph topology. To articulate our new model, we expand on traditional characterizations of graph rigidity given by Laman [10] and Henneberg [5]. We design artifacts (which work in both models) showing that surprisingly complex behavior can be developed from first principles. The constructions include Boolean logic and signal propagation, catalytic splitting reminiscent of ATP hydrolysis, fueled directed cycles, and chemo-mechanical coupling. The latter constructions are motivated by the coupling of fuel consumption with other processes prevalent in biological molecular machines such as kinesin, myosin, and dynein [1, 3].

2 Examples

Mechanical linkages are well studied and common in mechanical engineering. Even one of the most basic linkages, the lever, is found as a component in countless machines and tools. Linkages have also been shown to be capable of very complex motion [11]. Adding binding sites makes linkages interesting as a model of chemical machines. This section shows how chemical linkages work by example, while Section 3 defines them formally.

In this work we consider a single-copy regime where, unless otherwise stated, a single copy of each linkage is present in a given system (see Conclusion for additional discussion).

► **Example 1 (Binding sites).** As the following example shows, joints can have binding sites. The star $*$ means a (solid dot) and a^* (hollow circle) are complements and so can bind. For their sites to bind, joints have to overlap. When an a and a^* overlap, we may label them together with just a . Unless noted otherwise, we consider strong bonds which may not break. (Appendix A discusses the approximation of strong bonds using bonds that may break.) Although bonds are strong, joints with the same binding sites may displace one another. So to get from the left bound state to the right bound state, the right a^* joint must colocalize and displace the left a^* joint. The four possible states of the system are the following.



We consider a state to change only when the set of bonds changes. Physically, the shape of a linkage can move among an infinite continuum of conformations. But it would move rapidly and randomly among the conformations allowed by its bonds as Brownian motion and low Reynolds number dominate molecular dynamics. This is why we say this example has four discrete states and not an infinite continuum of states.

► **Example 2 (Allostery).** The following example shows that geometry can prevent complements from binding. There are two bonds that can form, the a bond and the b bond. But after one forms, the other can no longer reach to overlap.

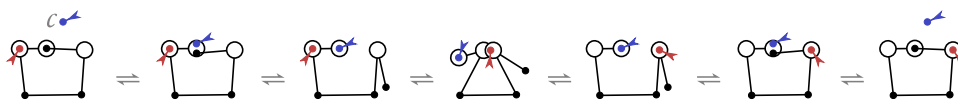


Note that links are allowed to cross over each other. Allowing link crossing simplifies the mathematical model and is standard in the analysis of mechanical linkages. In a physical realization, the links might be offset at different heights to allow such crossing.

► **Example 3 (Simple catalysis).** The following example is a system where there are two states, left and right, that cannot reach each other. For each state, two of its (infinitely many) conformations are shown with \approx between them. The link lengths keep the small red linkage from getting close enough to the opposite joint to displace onto it. (A small flag represents irrelevant omitted parts, so the red linkage shown could be a portion of a larger linkage that includes the joint with the flag.) So the red linkage stays bound to whichever joint it starts bound to.



But we can add a linkage that allows the states to reach each other. In the following example, consider adding the blue linkage at the top to mediate the state change. The blue linkage with binding site c^* can enter and displace the center bond. This allows the red linkage to reach the opposite joint. Afterward, the blue linkage is displaced by the center bond and leaves. Thus the blue linkage is unchanged, but the black linkage has changed state. The blue linkage acts as a catalyst.

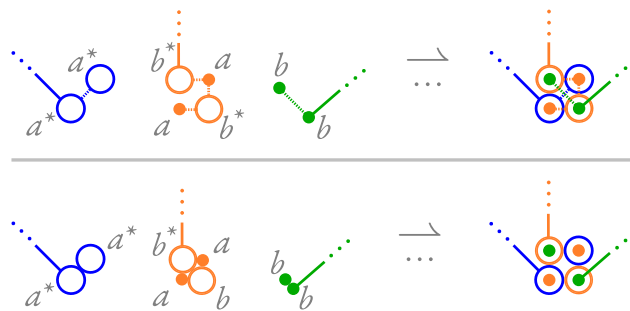


7:4 Molecular Machines from Topological Linkages

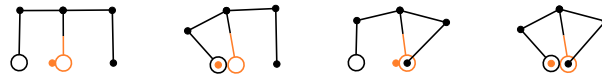
► **Example 4** (Compound catalysis). The following is a catalytic system that involves three catalysts, or one compound catalyst, depending on the reader's perspective. The linkage can again be in one of two states unable to reach each other. But when all three of the catalytic linkages are present, the state can change. The intermediate displacement states are left out.



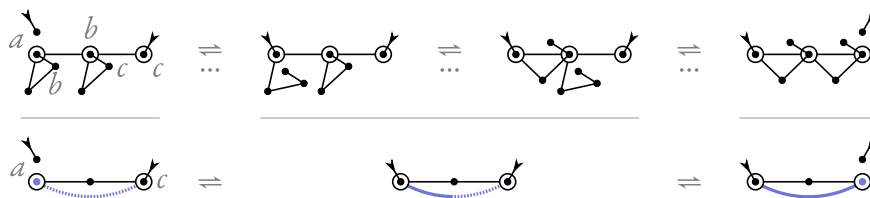
► **Example 5** (Many binding sites). While each joint may only have one binding site, the effect of multiple binding sites per joint may be achieved via zero-length edges. Below, zero-length edges between joints are indicated by dashed lines. We omit these edges for visual clarity but instead rely on color to disambiguate which binding sites are connected together with zero-length edges. (Although some information is lost going from the top to bottom figure, the bottom notation will be sufficient for our purposes.)



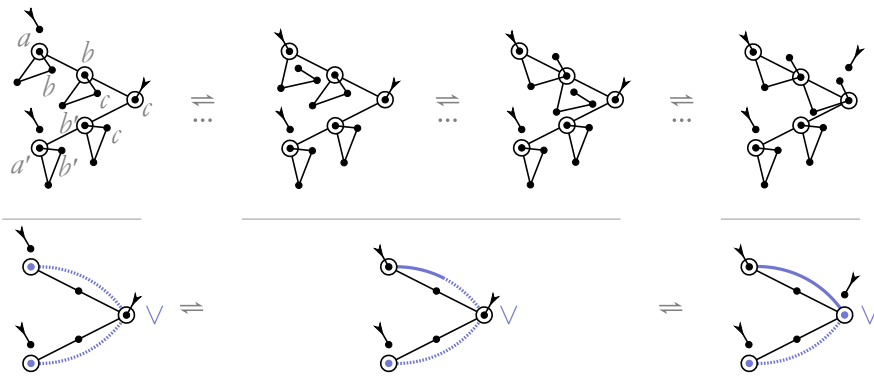
This effectively allows a joint to form multiple bonds at the same time.



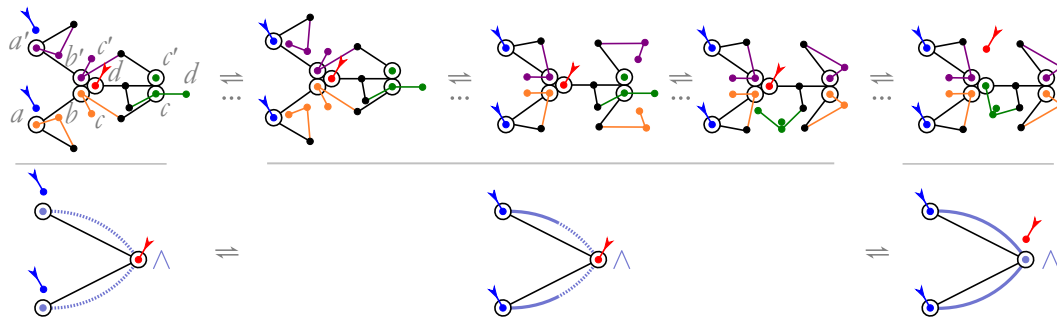
► **Example 6** (Signal cascade). The following is a system that transmits the signal of whether a bond is formed. The signal travels along a sequence of links. The effect is that the flag with binding site a and the flag with binding site c can never be free at the same time. By repeating this pattern, we can get this effect across any number of links. Since signal cascading is useful as a modular gadget, we abstract it with visual notation using blue arcs as shown in the bottom half of the figure.



► **Example 7** (Advanced cascades). The following systems show how cascades can be combined for various effects. By combining two signal cascades with a common joint, signal cascades may behave like a logical OR. The rightmost flag may be freed after either of the leftmost flags have bonded.



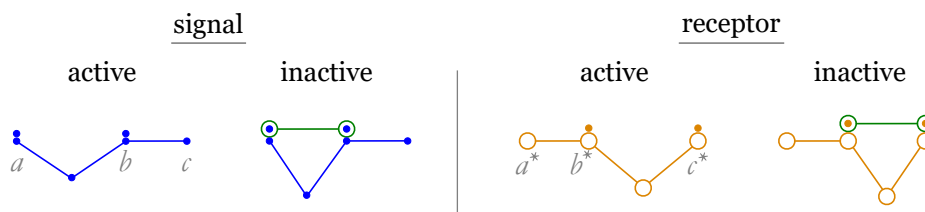
A logical AND may be achieved with signal cascades as well. Both of the leftmost flags (a and a') must bond to the linkage before the d flag can be freed. Using the notation introduced in Example 5, directly adjacent binding sites indicate a single joint with multiple binding sites.



By reversing the AND mechanism shown above, we can effectively implement a fanout.

► **Example 8** (Active/inactive signal receptors and sequential AND gate). We now show a construction for intermolecular signals and their corresponding receptors, with both capable of activation and inactivation by other signals. In particular, we show an example which mimics the sequential AND gates of reference [15] operating via DNA strand displacement.

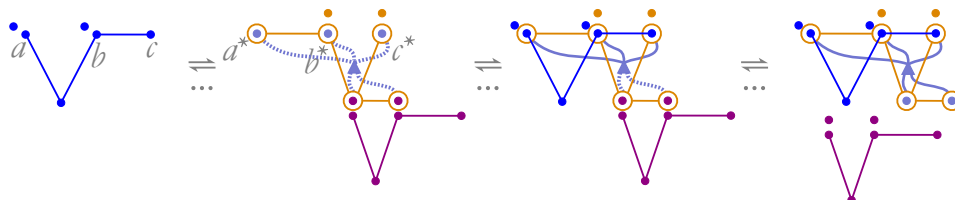
In our construction, whether signals and their receptors are active depends on whether their three binding sites can be simultaneously bound or not (shown below). We later refer to such complete binding as *docking*. Docking is prevented by geometrical constraints if either the signal (blue) or receptor (orange) is inactive. Intuitively, the distance between the joints within signals and receptors needs to match exactly for them to dock. But this cannot happen when either the signal or receptor has a joint distance that is fixed by another linkage. In the figure, the green linkage fixes the distance between the a and b joints (or b^* and c^* for the receptor) to a length that is too short. So only active signals and receptors may dock.



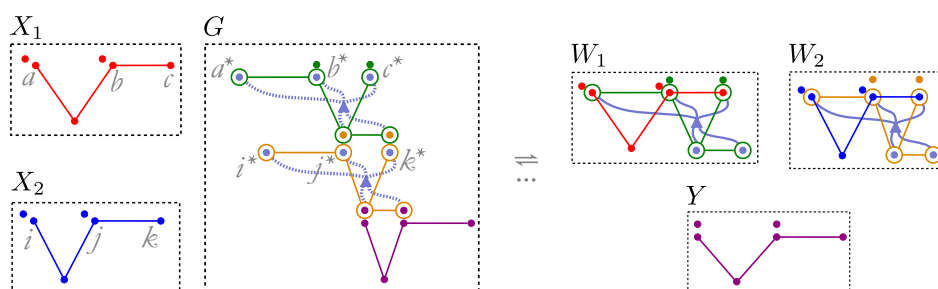
The following figure shows a modification of the orange receptor which holds the output (purple) signal linkage inactive until the input (blue) docks with it. Here, the blue triangular symbol represents a cascade combination of an AND and a fanout. All three signal cascades

7:6 Molecular Machines from Topological Linkages

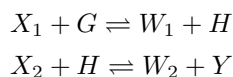
entering the top of the triangle must fire before the two signals on bottom can fan out. This gadget ensures that all three bonds a , b , and c must form between the signal and receptor before the purple linkage is released. Notice that the output signal linkage can be an input to another receptor downstream, and so such systems are composable.



Modifying this scheme further, the following example shows a composable sequential AND gate.



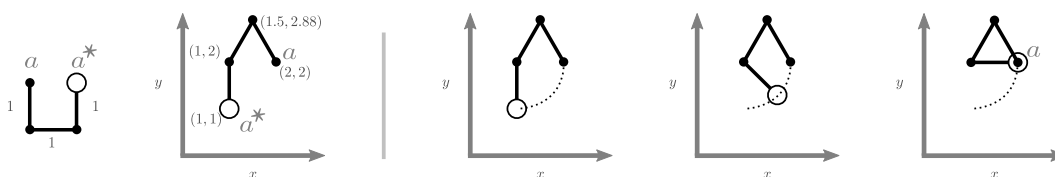
Written in terms of abstract chemical reactions, the above system implements the following behavior:



where X_1 and X_2 are the input signal linkages and Y is the output signal linkage. Observe that the purple output signal linkage and the orange receptor are initially both in their inactive states. The red input linkage must dock with the green receptor first in order to activate the orange receptor (splitting complex G in the process). Then the blue input signal linkage docks with the now active orange receptor, displacing and activating the purple output signal linkage. Appendix B shows the state-change details for this construction.

Note that DNA strand displacement cascades are based upon the toehold sequestering mechanism which allows activating and inactivating both signals and their displacement targets. Targets (receptors) are activated by opening their toehold domains, while signals are activated by opening their toehold binding domains. The above construction shows one way in which signal and receptor activation can be recapitulated by chemical linkages.¹

¹ In the toehold sequestering mechanism, activation is a kinetic effect: toehold binding increases the effective local concentration of the signal strand near the target, which promotes displacement. Displacement can still occur without preceding toehold binding, but is much slower. Our chemical linkages implementation does not attempt to capture such physical details of toehold sequestering but rather the higher-level activation/inactivation behavior. The physical correlate for activation in our model is geometric compatibility rather than an increase in local concentration. (Although beyond the scope of this paper, a kinetic model of chemical linkages, for example operating via Gillespie kinetics, with weak bonds representing toeholds (see Appendix A) is needed to capture the kinetic mechanism of toehold sequestering.)



■ **Figure 1** (Left) A chemical linkage and a conformation p of that linkage. (Right) A motion q of conformation p . Left to right: conformation $q(0)$ of q , conformation $q(t)$ where $0 < t < 1$, and conformation $q(1)$. Notice q is a binding motion that forms a new bond.

3 Formal model

The previous section relied on intuitive explanations of chemical linkages, as does prior work [12]. It would be useful to have a precise, general definition. This would support engineering, guide simulations, and enable proofs. This section formally defines chemical linkage systems and their state space.

3.1 Chemical linkages

A *mechanical linkage* is a pair (G, ℓ) . G is a connected graph with vertices V and edges E . We also call vertices *joints* and edges *links*. $\ell : E \rightarrow \mathbb{R}_{\geq 0}$ is a map that gives each link a length. Link lengths alone are not enough to define how the graph sits in space. So to uniquely determine the shape of a linkage, we use conformations.

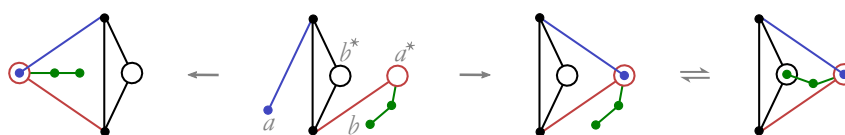
A *conformation* of a linkage (G, ℓ) is a map $p : V \rightarrow \mathbb{R}^2$ where $|p_u - p_v| = \ell(u, v)$ for each pair u and v of linked joints.² Intuitively, a conformation is a drawing of a linkage in the plane with the right link lengths. In some drawings, joints may overlap. The *overlap* of a conformation is the partition of its joints where each element is a set of joints that overlap. For example, the partition $\{\{u, v\}, \{w\}, \{x\}\}$ of $\{u, v, w, x\}$ would mean that joints u and v overlap while no joint overlaps w and no joint overlaps x .

A *chemical linkage* is a mechanical linkage with a function $d : V \rightarrow \Sigma$ that puts a binding domain on each joint. Σ is an alphabet of starred and unstarred symbols called *domains*. Domains x and x^* are said to be complementary and thus capable of binding. Intuitively, this evokes a chemistry where opposites bind like complementary DNA domains.³ Note that we use the phrase “multiple binding sites” to refer to zero-length edges effectively allowing a joint to have more than one binding domain (see Example 5). In this paper we use the word linkage to refer to mechanical or chemical linkage if clear from context.

A *matching* of a conformation is a set of unordered pairs of its joints such that (1) each pair consists of overlapping joints which have complementary domains and (2) no two pairs share a common joint. A matching of a conformation is a *binding* if it is not a subset of any other matching of that conformation (i.e. it is a maximal matching). Intuitively, elements of the binding represent a bond between two joints. Since a binding is a *maximal* matching, we consider bonds to form as soon as joints become overlapping. In the case where three or more joints overlap, a conformation could have multiple bindings (for example consider the lower middle state in Figure 3).

² For simplicity, we focus on two dimensions in this work. It is an open question how some of this work generalizes to three dimensions (see also Conclusion).

³ Other choices of domain chemistry are of course possible, where binding might be like-like. Such binding rules are not explored in this work.



■ **Figure 2** This figure illustrates why bindings alone do not sufficiently capture the behavior of chemical linkages. Middle left: a state where bonds a and b can form. If a bonds first, it can lead to two different states. In one state (left), the binding of a prevents the binding of b due to link lengths. In the other state (middle right), the green link can still reach a state where b is bonded (right). Notice the left and middle right conformations have the same binding, but the conformation geometry dictates which states can be reached.

The natural notion of motion captures how transformations may be applied to conformations. Let $[0, 1]$ be the interval of real numbers from 0 to 1. A *reconfiguration* of a conformation p is a map $q : V \rightarrow ([0, 1] \rightarrow \mathbb{R}^2)$ where each $q(t)$ is a conformation and $q(0) = p$. Note that for convenience, $q(t) = u \mapsto q_u(t)$ is the conformation at time t , and $q_u = t \mapsto q_u(t)$ is the trajectory of joint u . A *motion* of a conformation p is a reconfiguration q where each q_u is continuous. Intuitively, this means a motion preserves link lengths and never flips parts of a linkage: for example in Figure 2, transitioning from the left state to the middle right state, without breaking bonds, is not a motion.

A motion q is a *step motion* if there exists a same binding of conformation $q(t)$ for all $t \in [0, 1)$. A step motion is a *binding motion* if conformation $q(0)$ has some binding that is a subset of a binding of conformation $q(1)$. Intuitively, a step motion maintains the overlap of bound joints and a binding motion only ever (potentially) creates more bonds. For conformations x and y , we write $x \rightarrow y$ if there exists a binding motion from x to y . We write \rightarrow^* to mean the reflexive, transitive closure of \rightarrow . We write $x \leftrightarrow y$ if $x \rightarrow y$ and $y \rightarrow x$. Similarly, we write $x \leftrightarrow^* y$ if $x \rightarrow^* y$ and $y \rightarrow^* x$.

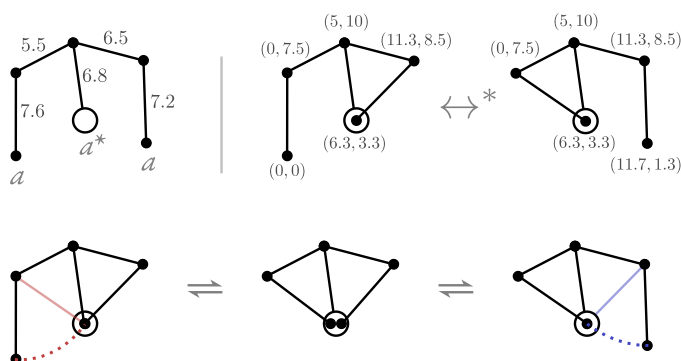
Although this work only uses binding motions, we can also define motions which are allowed to reduce the number of bonds – for completeness. A step motion is a *breaking motion* if conformation $q(0)$ has a binding that is a superset of a binding of conformation $q(1)$.

3.2 Chemical linkage states and systems

Binding motions fully capture the behavior of chemical linkages. But most binding motions do not lead to interesting changes in a given conformation. Such changes arise when a conformation's binding is altered by the binding motion (e.g., see Figure 2). In a sense, some conformations are equivalent, while others are not. We use the notion of states to capture conformation equivalence and identify the significant binding motions.

A state is an equivalence class of conformations defined as follows. Two conformations are in the same state if they can both reach all of the same conformations through a binding motion. Formally, two conformations x and y are in the same state if for every other conformation z , $x \rightarrow z$ if and only if $y \rightarrow z$. State b is *directly reachable* from a , written $a \rightarrow b$, if there exist conformations x in state a and y in state b such that $x \rightarrow y$. If $a \rightarrow b$ and $b \rightarrow a$, we write $a \rightleftarrows b$. We say state b is *reachable* from a , written $a \rightsquigarrow b$, if $x \rightarrow^* y$. If $a \rightsquigarrow b$ and $b \rightsquigarrow a$, we write $a \rightleftarrows^* b$. Figure 3 (bottom) shows an example of state reachability.

We can treat a set of linkages exactly like a single linkage by forming the disjoint union of its linkages. Intuitively, we pretend all its linkages are one big linkage. This way we can use all the vocabulary of linkages for a set of linkages. We refer to sets of chemical linkages as a system. Formally, a *chemical linkage system* is a pair (C, s) where C is a set of



■ **Figure 3** (Top) A chemical linkage c and two conformations of c which may be transformed into one another via a sequence of binding motions. (Bottom) Three distinct states of c . In the middle state, both of the two a binding sites and the a^* binding site overlap. The left and middle states are directly reachable from one another via the binding motion shown in red. Likewise, the middle and right states are directly reachable from one another via the blue binding motion. The left and right states are reachable from one another, but not directly reachable (since the binding changes in the middle state).

chemical linkages and s is an initial state. As Figure 2 shows, the initial state is important for determining state reachability for a given chemical linkage system. In a conformation of a linkage system, we often call a set of linkages a *complex* if there exists a binding which makes that set of linkages connected.

3.3 Complexity of simulation

With the formal model established, we now have a notion of how to describe the behavior of chemical linkages. To understand and predict this behavior, we need to know a given linkage's state space. Computing the entire state space for a system is certainly a hard problem. But being able to easily check if one state is directly (one step) reachable from another would make it easier to design and analyze chemical linkage systems.

Unfortunately, the problem of deciding direct reachability between two states is PSPACE-hard because the subproblem of finding a motion between conformations is PSPACE-hard [6]⁴. This does not bode well for the future development of simulation tools for this model. For such simulation tools, we would want to have a fast algorithm to check direct reachability. This is part of our motivation for introducing the topological linkages model described in the next section.

4 Topological linkages

Linkages provide a model of physical constraints that seems minimal. They involve only two simple physical parts, links and joints, neither of which can be removed. Despite that intuition, this section describes a surprising new simpler model that still has interesting complex behavior. In fact, our simpler model captures the behavior of all of the examples in Section 2.

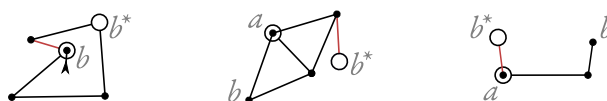
⁴ While the general mover's problem (reachability) was shown to be PSPACE-hard in [13], we can more easily adopt the formulation of [6] by fixing some joints' relative positions to each other via a rigid "frame." A study of motion planning can be found in [11].

To distinguish the two models, we call the original metric and the simpler topological.

4.1 Topological motivation

In this section we build the case for a version of the chemical linkages model which ignores link lengths. The utility of such a model may seem surprising given that the constructions developed in previous sections relied on some joint being able or not being able to “reach” to another joint in metric space. Rather than fixing link lengths beforehand and asking whether a joint can reach another, we instead ask how *constrained* must the link lengths be for such reach to be possible.

Recall Examples 2 and 3 from Section 2 and how certain states were not reachable due to the choices of link lengths. We present these two examples again below, along with a third example.



In each example, we consider the potential bonding of the b and b^* joints. Without link lengths being previously fixed, observe the link length constraints implied for each edge. Once the lengths of the black links are fixed, we ask how constrained the red link lengths are.

In the right example, a range of lengths for the red link will allow bond b to form (as long as the link has sufficient length). On the other hand, in the two examples on the left, bond b may be formed only if the red links happen to be the *exact* right length. Assuming link lengths are somehow “generically” chosen, bond b will not form in the left two examples because the red link lengths will not reach. Note that from a practical perspective, avoiding such exact-length coincidences is easy. Contrarily, setting lengths exactly in a molecular implementation may be onerous.

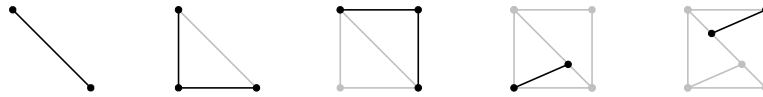
Our key observation is that this exact-length constraint arises in these examples because the forming of bond b would cause the red link to connect two joints which are already rigidly connected (we refer to this later as overbracing). Note that we say two joints are rigidly connected if their distance is fixed for every motion. In the left example it is clear that the top-most link already fixes the distance of its two joints. In the middle example the rigidity of the linked rhombus fixes the distance of the bottom-left and top-right joints.

Each of the constructions presented in Section 2 exhibit the behavior of these left two examples. We will define the topological linkage model with regard to overbracedness, rather than metric lengths. To formally define overbracing edges we must first discuss the notion of rigidity.

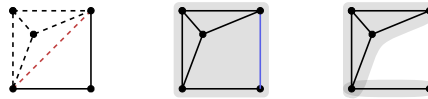
4.2 Rigidity

Intuitively, a graph is rigid if all of its joints have to move together. A rigid graph is minimally rigid if removing any edge results in a non-rigid graph. These properties naturally generalize to subgraphs as well.

We provide a definition of graph rigidity based on the characterization captured by Henneberg operations [5, 4]. Henneberg characterized minimally rigid graphs as the graphs which can be constructed, starting from a single edge, by executing some sequence of two types of operations. A V operation adds a new node u and two new edges that connect u to two existing nodes in the graph. A T operation adds a new node u that splits an existing edge and adds a new edge from u to an existing node. Figure 4 shows an example.



■ **Figure 4** A sequence of Henneberg operations. An edge to start, then two V operations, then two T operations. The graph produced at each step is minimally rigid.



■ **Figure 5** An overbraced graph (left) and its overbracing edges (dashed). If we remove an overbracing edge (red), this graph happens to become minimally rigid (center). We can verify this with Henneberg operations. If we remove a non-overbracing edge (blue), the graph is no longer minimally rigid (right). We can verify this with Laman’s theorem.

So we say a graph G is *minimally rigid* if some sequence of Henneberg operations turns a single edge into G . We build upon this definition to formalize overbracing discussed previously. A graph G is *overbraced* if it has a subgraph H that is minimally rigid and G contains an additional edge between two nodes of H . This edge is called an *overbracing edge*. Figure 5 shows an example.

Naturally, these Henneberg operations are useful in verifying if a given graph is minimally rigid. What they do not do, however, is provide an easy method for showing a graph is not minimally rigid.

Luckily, when a graph is not rigid, Laman’s theorem [10] guarantees there is a simple proof. To state this powerful theorem, say the *excess* of a graph with v vertices and e edges is $e - (2v - 3)$. Recall that for a graph G , an induced subgraph I is a graph formed from a subset of vertices from G taking all edges connecting pairs of vertices in that subset.

► **Theorem 9** (Laman’s theorem). *A graph G is minimally rigid iff (1) the excess of G is 0, and (2) the excess of any induced subgraph of G is at most 0.*

Laman’s theorem lends itself to proving that a graph is not minimally rigid. We just show that the graph has non-zero excess, or we give an induced subgraph with positive excess.

We now want to transition from checking minimal rigidity to checking overbracedness, which is the real object of our attention since we use overbracedness as a substitute for “being unable to reach”. First, the following corollary formally confirms our intuition that a minimally rigid graph cannot be overbraced. Via this corollary, we can use Henneberg operations or Laman’s theorem to show that a graph is not overbraced.

► **Corollary 10.** *If a graph G is overbraced, then G is not minimally rigid.*

Proof. Let G be an overbraced graph. By definition of overbraced, G has some minimally rigid subgraph $H = (V, E)$ and some additional edge e between two nodes of H . By Theorem 9, the excess of H is 0. Let $I = (V, E')$ be the induced subgraph of G on nodes V . Since $e \in E'$ and $e \notin E$, the excess of I must be greater than 0. So, G is not minimally rigid. ◀

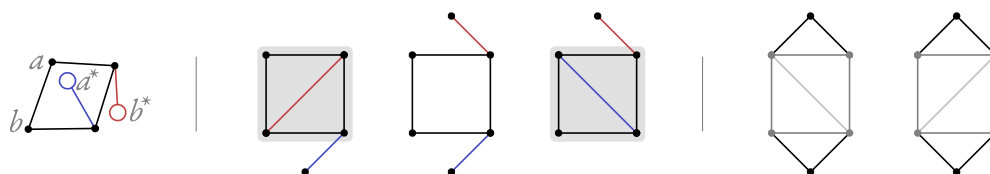
Our constructions may not be minimally rigid as a whole (may have non-rigid parts) as in Figure 5 (right). In this case, to argue that a graph is not overbraced we rely on the following Lemma.

7:12 Molecular Machines from Topological Linkages

► **Lemma 11.** *If a graph G is a subgraph of some minimally rigid graph M , then G is not overbraced.*

Proof. We prove this by contrapositive. Let G be an overbraced graph. By definition of overbraced, if we add edges and/or vertices to an overbraced graph it will still be overbraced. This means that any graph M which has G as a subgraph must also be overbraced. By Corollary 10, M is not minimally rigid. ◀

For the examples in Section 4.1, we relied on mechanical intuition to see that the endpoints of the red edge are rigidly connected. We are now ready to show how formal rigidity arguments can be used to show this.



The figure above shows the linkage from Example 2 (left) and its three states (middle). We can prove that none of the three states are overbraced using Henneberg operations and Lemma 11. Two minimally rigid graphs are shown (right), constructed via Henneberg operations. The edge shading shows the order of the operations. This shows that each of the three states is a subgraph of a minimally rigid graph. Thus, by Lemma 11, they are not overbraced. However the following would-be state, not shown in the example, is prevented not just by reach, but by mere rigidity.



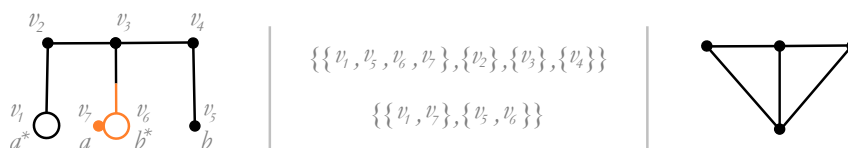
This figure shows that the state (left) has a minimally rigid subgraph H and an edge between two vertices in H (right). By definition, this state is overbraced.

The figure below shows similar analysis of the prevented state from Example 3.



Example 3 (left) and its prevented state (right). The state on the right is shown to be overbraced, as it has a minimally rigid subgraph (solid black edge) and an edge connecting two vertices from that subgraph (dashed edge). In fact, any graph with multiple edges between vertices is overbraced by our definition. However, in the presence of the catalyst no state in the transition sequence is overbraced. The reader can verify this again using Henneberg operations. In fact, this kind of analysis may be applied to *all* the examples in Section 2, verifying that the topological behavior of each follows the metric behavior.

Finally, we touch upon the computational complexity of checking overbracedness. Recall that one of the motivations for the topological linkages model is that checking direct (one-step) reachability in our original metric model was PSPACE-hard. Can we improve upon this with the simplification of the topological model? Laman's Theorem does not suggest a fast algorithm since there might be exponentially many induced subgraphs. However, a well-known algorithm called the pebble game does just that [7]. The algorithm runs in quadratic time and reports whether the given graph is rigid, what its rigid components are if it is not, and where it is overbraced.



■ **Figure 6** A topological linkage (left). Note that zero-length edges are denoted by adjacent same-colored joints. An overlap and a binding for the given linkage (middle). The collapse of the given overlap (right). Notice that the collapse is not overbraced, so it is a state.

4.3 Formal model

We define a kind of chemical linkage that completely ignores lengths. Figure 6 illustrates these definitions. A *topological linkage* is a triple (G, d, ℓ) . G is a connected graph with vertices V and edges E . $d : V \rightarrow \Sigma$ is a map that puts a binding domain on each joint and $\ell : E \rightarrow \{0, +\}$ is a map that labels each edge as a zero-length edge or positive-length edge. Recall that the use of zero-length edges in the metric model effectively describes multiple binding sites on one joint. We distinguish between zero-length and positive-length edges for the same effect here. A topological linkage has no other specified lengths or geometry. Its only structure comes from the topology of its graph.

We define the state space of a topological linkage with no appeal to motion or conformations. Instead, we use a partition of the topological linkage's joints that represents which joints are meant to overlap. An *overlap* of a topological linkage is a partition of its joints such that (1) no two joints are in the same partition part if they are connected by a positive-length edge and (2) any two joints connected by a zero-length edge are in the same partition part. Intuitively, joints connected by a zero-length edge already overlap, while joints connected by a positive-length edge cannot overlap. A *matching* of an overlap is a set of unordered pairs of its joints such that (1) each pair consists of joints from the same part of the overlap and which have complementary domains and (2) no two pairs share a common joint. A matching of an overlap is a *binding* if it is not a subset of any other matching of that overlap. Note that an overlap may have multiple bindings. The *collapse* of a topological linkage relative to an overlap is the graph that results from the following operations: (1) remove all zero-length edges, and (2) perform vertex contraction on the vertices in each part of the overlap.⁵ A *state* is an overlap whose collapse is not overbraced.⁶

Similar to metric linkages, we define a notion of reachability for topological linkage states. State b is *directly reachable* from a , written $a \rightarrow b$, if a has a binding that is a subset of a binding of b . If $a \rightarrow b$ and $b \rightarrow a$, we write $a \rightleftarrows b$. We define \rightsquigarrow to be the reflexive, transitive closure of \rightarrow and say state b is *reachable* from a if $a \rightsquigarrow b$. If $a \rightsquigarrow b$ and $b \rightsquigarrow a$, we write $a \rightleftarrows b$. Also similar to metric linkages, we define a *topological linkage system* as a pair (T, s) , where T is a set of topological linkages, and s is an initial state for that set of linkages.

⁵ We consider vertex contraction that may lead to a multigraph in cases where two vertices to be contracted, v_i and v_j , are both adjacent to some other vertex w .

⁶ Another reasonable approach may be to allow a system's initial state to be overbraced but to disallow forming bonds that lead to additional overbracing. However, without loss of generality overbracing edges could be removed from the initial state without affecting the behavior.

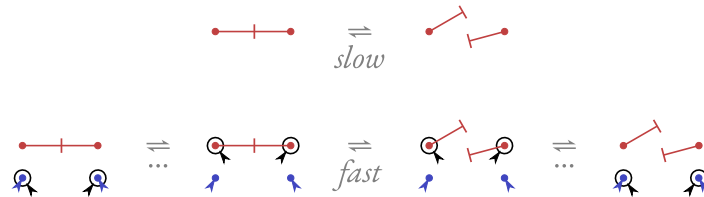
5 Fueled machines

One of the goals of this paper is to recreate some of the rich behavior of molecular machines. Thus far, we have presented constructions in the metric model (Section 2) and have shown that these constructions also work in the topological model (Section 4.2). This section develops additional complex behavior abstracting the ability of biological machines to consume fuel and couple this consumption with driving other processes. These constructions, although explained in the topological model, can also be understood in the metric model.

5.1 Hydrolysis

The molecular machines in living cells are fueled largely by ATP hydrolysis. We can imagine the molecule ATP as composed of two parts, ADP and P_i . For our purposes, we write this as $\text{ATP} \rightleftharpoons \text{ADP} + P_i$. The forward reaction is hydrolysis, which splits ATP. Normally, hydrolysis and its reverse are slow, which makes ATP stable in isolation. But if ATP docks with certain catalysts, both directions become fast. To make sure that hydrolysis happens more than its reverse, cells keep the concentration of the wastes ADP and P_i low.

It is not clear how to engineer systems to turn hydrolysis into work. But we can start by figuring out how to do so with linkages. The following linkage system abstracts a hydrolysis-like splitting event. The top red bar plays the role ATP. The two small linkages below it represent the catalyst that docks with it. Once the catalyst docks, the red bar can split.

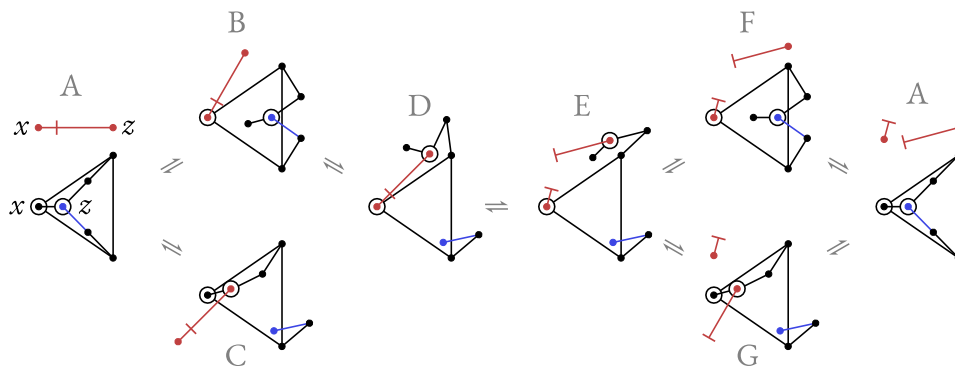


For now, we will take these splitting pseudo-linkages as a primitive, as does prior work [12], and we will focus on a construction that uses it.

5.2 Motor

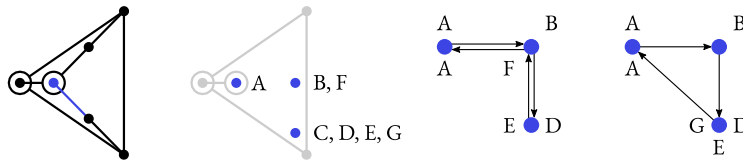
Mechanical work can be coupled to the motion of the catalyst if the catalyst undergoes an overall cyclic motion. A catalyst for binding and splitting ATP is shown below.

While we discuss the example in terms of topological states, we continue to use a visual notation which contains implicit link lengths. In this way our visual representation shows a particular metric implementation which remains compatible with the metric model.



The catalyst is asymmetric in a way that yields the following behavior. If the catalyst first binds ATP on the left (state B above), then it can subsequently bind on the right (state D). However, if the catalyst first binds the ATP on the right (state C), it is prevented from subsequently binding it on the left because that displacement passes through an overbraced state (not shown). After ATP splits into ADP and P_i (state E), the catalyst can unbind in any order (since the two binding sites are now split, no overbracing occurs).

Observe that the catalyst itself is always in one of three distinct states. The motion of the catalyst is determined by the order of detachment for ADP and P_i . Shown below is a depiction of how mechanical work is coupled to the motion of the catalyst from state to state.



If the catalyst unbinds on the right and then on the left, which is in the opposite order of binding, then it undoes any mechanical work done in the process of binding. However, half the time, the catalyst unbinds on the left and then on the right. This results in an overall biased work cycle, capable of driving mechanical work.

5.3 ATP from linkages

The following construction shows that we do not need to assume ATP hydrolysis as a primitive. Instead, here we present a pure linkage system that behaves from the outside just like the primitive. So we can actually treat the primitive not as an assumption, but as an abstraction.

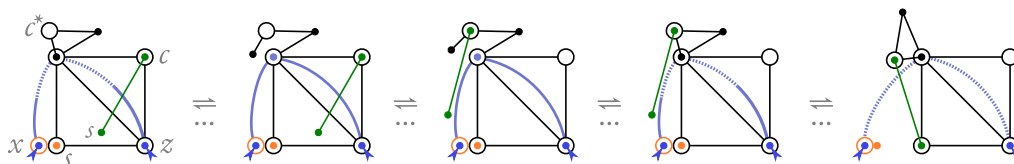


The two states, whole (left) and split (right), cannot reach each other. Recall that a dashed arc represents the gadget from Example 6. At least one of the binding sites at its endpoints must be bound. The gadget does not physically attach the halves, so the split state is two separate parts, despite a dashed arc appearing to connect them.

To go from whole to split, the long green link bound at the joint marked c would have to relocate to the unbound joint marked c^* . That would have to break a bond, which is not possible. But by adding the following catalyst, the long link can relocate.



When bound to the whole, the catalyst enables the following path between whole and split.

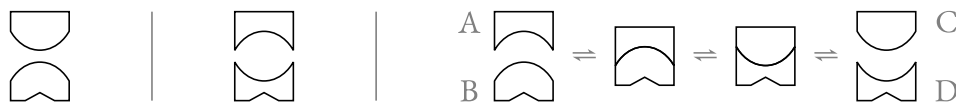


It might seem that the above ATP construction and the motor construction of Section 5.2 require different parities for the ATP-catalyst interaction. In the motor, the ATP displaces joints in the catalyst. In the ATP in this section, the catalyst displaces joints in the ATP. Luckily, the two displacements can be combined as shown in Appendix C.

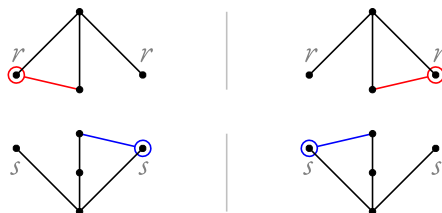
5.4 Chemo-mechanical coupling

Chemical coupling is a powerful tool. By chemical coupling we mean a reaction like $A + B \rightleftharpoons C + D$ with no side reactions like $A \rightleftharpoons C$ or $B \rightleftharpoons D$. Such a coupling allows a high concentration of A to behave as fuel to drive B into D , even when B turning into D is thermodynamically unfavorable.

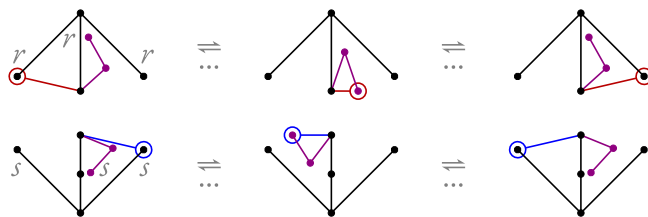
Here we will develop a construction that achieves chemical coupling. The following abstract diagram illustrates the target behavior. When the two linkages meet, they can only dock if their states are complementary. While docked they can switch states as long as they stay complementary. Otherwise, there is no docking and no state change.



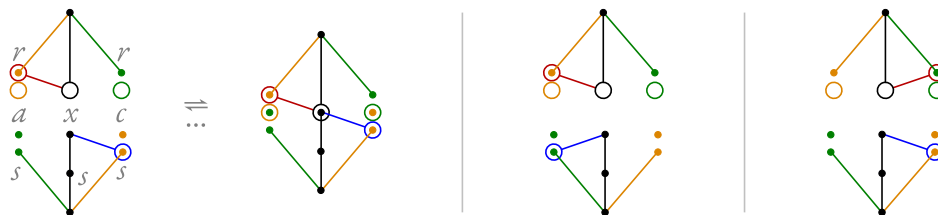
The following are two simple linkages each with two states, left and right. For each linkage, there is a barrier between its two states. The left state cannot reach the right state.



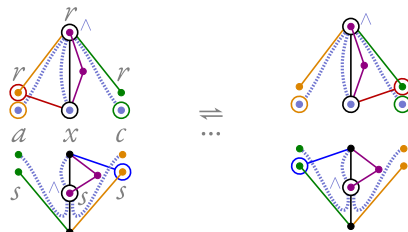
We can remove the barrier if we add an internal catalytic part. The following construction shows an example of this. It adds an arm with two links. The new arm can displace the matching domain of the original arm and carry it to the opposite side.



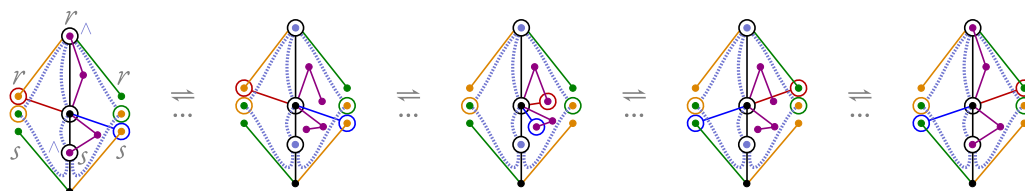
We can also allow the two linkages to interact by docking. The following construction shows an example of this. It adds matching domains, a, x, c , to three joints on each linkage. The dashed circle indicates a joint with two domains. The two linkages dock when the three joints all bind their partner. Notice that the two linkages can dock only when their states are complements since the result would be overbraced otherwise.



The following construction shows how we can prevent catalysis unless docked. Recall that the dashed lines represent the AND gadget from Example 7. So the internal catalytic arm can come free only when all of the docking sites are bound. This way the following two system states can reach each other, but only because their linkage states are complements.



The following figure shows the sequence of system states that flips the linkage states atomically, as a unit. This implements the chemo-mechanical coupling $A + B \rightleftharpoons C + D$ that we had as our goal.



6 Conclusion

Along with defining the metric and topological chemical linkage models, we have provided several examples of the complex behavior captured by them.

Throughout this work we have assumed a single-molecule regime where exactly one copy of the linkages shown is present in the system. Indeed, having multiple copies introduces potential problems. For example, in Example 3, two copies of the system can catalyze each other's state change even in the absence of the blue catalyst. Nonetheless, we imagine that an implementation of a chemical linkage would utilize other kinds of geometry to prevent such issues (e.g., through volume exclusion not captured by our linkage model). Indeed, linkages have a history of being used as a small part of a whole system (e.g., the steam engine is not entirely a linkage system, but the linkage model provided valuable insight into its function).

Some important theoretical and practical questions remain. One of the most immediate questions is whether or not the topological model captures the full power of the metric model. Are some behaviors easier to achieve when using explicit edge lengths? Also, this work considered two-dimensional linkages. Can this be generalized to three dimensions? Note that minimal rigidity can be generalized to 3D via Henneberg-like operations [16].

While the topological model simplifies the design and analysis of chemical linkages, what about their actual construction? The lengths that were removed for topological analysis will have to be added back in the real world. For this, we give the following conjecture:

► **Conjecture 12.** *Given any topological chemical linkage system, there exists a metric chemical linkage system which has the same reachable state space.*

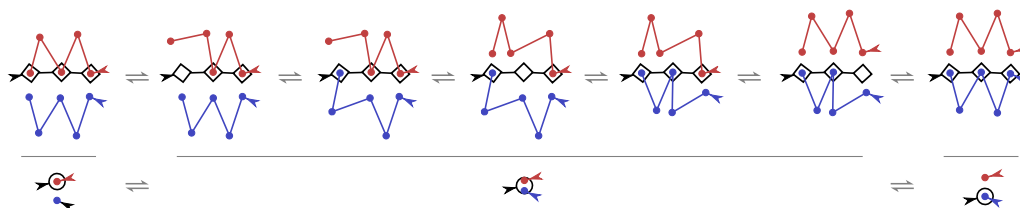
Ultimately, we believe that chemical linkages and other simple chemo-mechanical models hold promise. Maybe they can help us understand the behaviors we see in living cells. And maybe they can help us mimic them.

References

- 1 R. Dean Astumian, Shayantani Mukherjee, and Arieh Warshel. The physics and physical chemistry of molecular machines. *ChemPhysChem*, 17(12):1719–1741, 2016.
- 2 E. Branscomb, T. Biancalani, N. Goldenfeld, and M. Russell. Escapement mechanisms and the conversion of disequilibria; the engines of creation. *Physics Reports*, 677:1–60, 2017.
- 3 Aidan I. Brown and David A. Sivak. Theory of nonequilibrium free energy transduction by molecular machines. *Chemical Reviews*, 120(1):434–459, January 2020.
- 4 Erik D. Demaine and Joseph O’Rourke. *Geometric folding algorithms: linkages, origami, polyhedra*. Cambridge university press, 2007.
- 5 Lebrecht Henneberg. *Die graphische Statik der starren Systeme*, volume 31. BG Teubner, 1911.
- 6 John Hopcroft, Deborah Joseph, and Sue Whitesides. Movement problems for 2-dimensional linkages. *SIAM journal on computing*, 13(3):610–629, 1984.
- 7 Donald J. Jacobs and Michael F. Thorpe. Generic rigidity percolation: the pebble game. *Physical review letters*, 75(22):4051, 1995.
- 8 Michael Kapovich and John J. Millson. Universality theorems for configuration spaces of planar linkages. *Topology*, 41(6):1051–1107, 2002.
- 9 Alfred B. Kempe. On a general method of describing plane curves of the nth degree by linkwork. *Proceedings of the London Mathematical Society*, pages 213–216, 1875.
- 10 Gerard Laman. On graphs and rigidity of plane skeletal structures. *Journal of Engineering mathematics*, 4(4):331–340, 1970.
- 11 Steven M. LaValle. *Planning algorithms*. Cambridge university press, 2006.
- 12 Tosan Omabegho. Allosteric linkages that emulate a molecular motor enzyme. *bioRxiv*, 2021. URL: <https://www.biorxiv.org/content/10.1101/2021.04.20.440673v1>.
- 13 John H. Reif. Complexity of the mover’s problem and generalizations. In *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, pages 421–427, 1979.
- 14 Jason W. Rocks, Nidhi Pashine, Irmgard Bischofberger, Carl P. Goodrich, Andrea J. Liu, and Sidney R. Nagel. Designing allostery-inspired response in mechanical networks. *Proceedings of the National Academy of Sciences*, 114(10):2520–2525, 2017.
- 15 Georg Seelig, David Soloveichik, David Yu Zhang, and Erik Winfree. Enzyme-free nucleic acid logic circuits. *Science*, 314(5805):1585–1588, 2006.
- 16 Tiong-Seng Tay and Walter Whiteley. Generating isostatic frameworks. *Structural Topology 1985 Núm 11*, 1985.

A Weak bonds

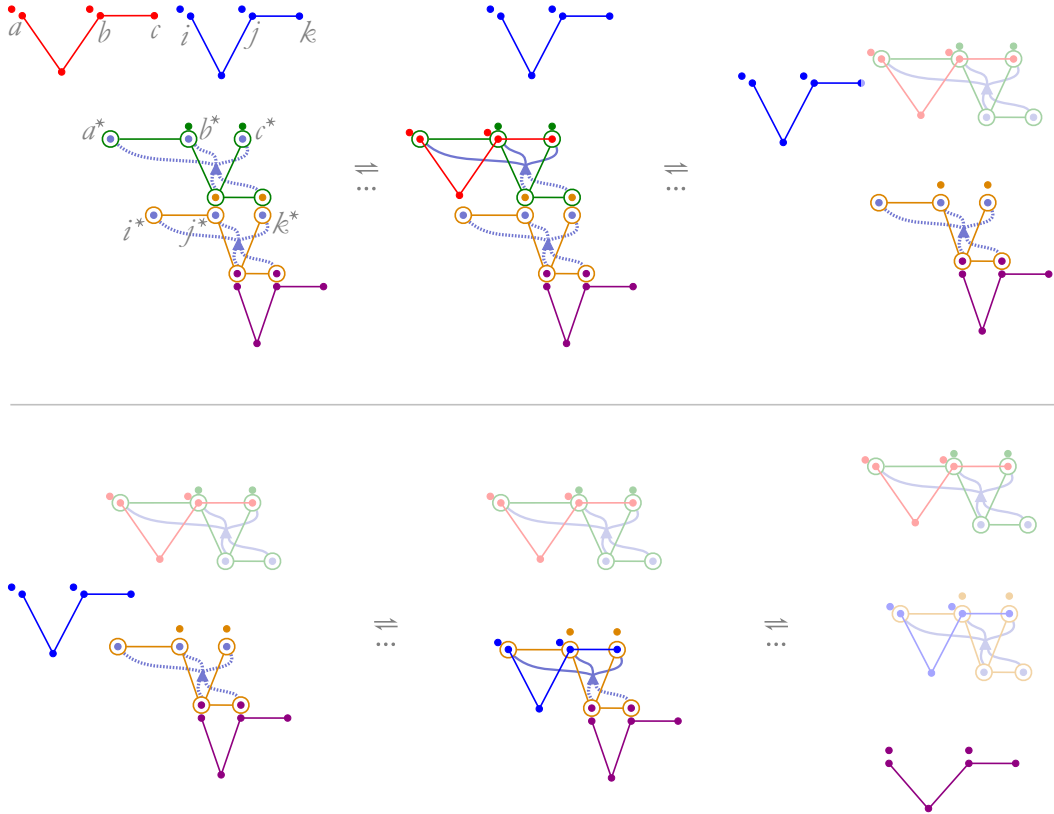
In the work above, we rely on strong unbreakable bonds. We can extend consideration to weak bonds which may break and reform. Indeed, we could consider weak bonds as a starting point of our model and construct strong bonds from weak bonds. The following example shows how a group of weak bonds (indicated as a diamond rather than a circle) can mimic strong bond displacement.



This example combines three weak bonds. Each can break individually, but they are unlikely to all break at the same time. So the only way the bottom blue linkage is likely to replace the top red linkage is by gradually displacing it. Note that this is very similar to DNA strand displacement. The bottom of the figure shows our standard representation of strong bond displacement.

B Sequential AND details

Here we show the state change sequence for the sequential AND construction.



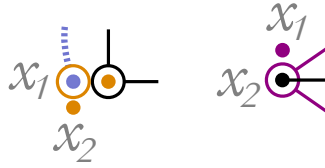
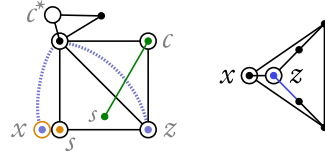
In the initial system state (top left), the red linkage and blue linkage are separate from the green, orange, and purple linkages (which are all bonded to one another). Since the red signal and the green receptor are both active, they may dock with one another (top center). This bonding triggers the AND/FANOUT gadget, displacing the linkage pair and activating the orange receptor (top right). With the blue signal and the orange receptor both active (bottom left), they may dock to trigger the displacement mechanism (bottom center). This results in the purple signal linkage becoming active (bottom right) only when the red and blue signal links have bonded with their corresponding receptors (in the correct order).

C Modified ATP and catalyst

In order to apply the ATP from Section 5.3 to the motor from Section 5.2, we need to ensure that binding of the ATP and the catalyst results in displacement of joints both in the ATP and the catalyst. We can achieve this by splitting the original binding sites x and z into two different binding sites each of opposite complementarity allowing for a separation of

7:20 Molecular Machines from Topological Linkages

responsibility. As shown below with respect to the x binding site, one site (x_1) would be responsible for triggering the signal cascades within the ATP linkage. The other (x_2) would similarly displace the black joint in the motor.



Small Tile Sets That Compute While Solving Mazes

Matthew Cook 

Institute of Neuroinformatics, University of Zürich and ETH Zürich, Switzerland

Tristan Stérin   

Hamilton Institute, Department of Computer Science, Maynooth University, Ireland

Damien Woods   

Hamilton Institute, Department of Computer Science, Maynooth University, Ireland

Abstract

We ask the question of how small a self-assembling set of tiles can be yet have interesting computational behaviour. We study this question in a model where supporting walls are provided as an input structure for tiles to grow along: we call it the Maze-Walking Tile Assembly Model. The model has a number of implementation prospects, one being DNA strands that attach to a DNA origami substrate. Intuitively, the model suggests a separation of signal routing and computation: the input structure (maze) supplies a routing diagram, and the programmer's tile set provides the computational ability. We ask how simple the computational part can be.

We give two tiny tile sets that are computationally universal in the Maze-Walking Tile Assembly Model. The first has four tiles and simulates Boolean circuits by directly implementing NAND, NXOR and NOT gates. Our second tile set has 6 tiles and is called the Collatz tile set as it produces patterns found in binary/ternary representations of iterations of the Collatz function. Using computer search we find that the Collatz tile set is expressive enough to encode Boolean circuits using blocks of these patterns. These two tile sets give two different methods to find simple universal tile sets, and provide motivation for using pre-assembled maze structures as circuit wiring diagrams in molecular self-assembly based computing.

2012 ACM Subject Classification Theory of computation → Models of computation; Theory of computation → Computational geometry

Keywords and phrases model of computation, self-assembly, small universal tile set, Boolean circuits, maze-solving

Digital Object Identifier 10.4230/LIPIcs.DNA.27.8

Related Version *Full Version*: <https://arxiv.org/abs/2106.12341>

Supplementary Material *Software (Source Code)*: <https://github.com/tcosmo/mawatam>
archived at `swh:1:dir:b7eca563fe310db9000aae3a6e2ede44fc1df99c`

Funding Research of Woods and Stérin supported by European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 772766, Active-DNA project), and Science Foundation Ireland (SFI) under Grant number 18/ERCS/5746, both to D Woods.

Acknowledgements We thank Trent Rogers, Niall Murphy, Pierre Marcus and Nicolas Schabanel for useful discussions on the Maze-Walking Tile Assembly Model.

1 Introduction

We can think of solving a maze as performing computation: the input is a maze, some starting location(s) and an ending location, and the answer to the computation is a yes/no answer signifying whether the exit is reachable from the start, or even an explicit path from start to exit. Figure 1(a,b) shows how a maze encodes a circuit of OR gates: solving the maze



© Matthew Cook, Tristan Stérin, and Damien Woods;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on DNA Computing and Molecular Programming (DNA 27).

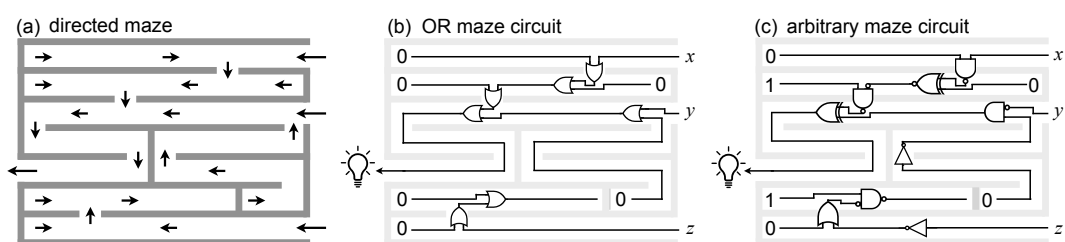
Editors: Matthew R. Lakin and Petr Šulc; Article No. 8; pp. 8:1–8:20

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

8:2 Small Tile Sets That Compute While Solving Mazes



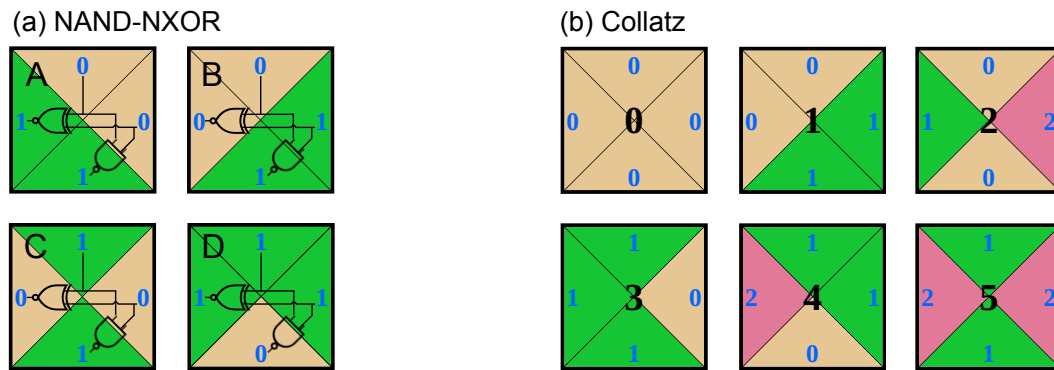
■ **Figure 1** Mazes, computation and Boolean circuits. Solving (a) a *directed* maze, where paths have directions, is formally equivalent to executing (b) an OR circuit if we ask: are any of the input bits that are set to 1 connected to the output gate? The example in (b) accepts any 3-bit input x, y, z that sets x or y to 1, irrespective of z ; equivalently the maze is solvable from the top two inputs only. Even such a simple setup, allowing for arbitrary mazes, can compute by solving any suitably-encoded problem in the class nondeterministic logspace (NL) [1, 29]. (c) We generalise this notion of “computation via maze-solving” in a natural way by having the maze specify arbitrary Boolean gates along the route that need to be evaluated. In the Maze-Walking Tile Assembly Model defined in Section 3.1, tiles flow through the maze, building paths from the entrances to the exit, evaluating the circuit as they go.

is equivalent to executing the OR circuit with all inputs set to bit 1; and asking about paths in the maze is equivalent to setting some inputs to 1 and seeing which paths have 1 flowing all the way through them. It then becomes meaningful to ask about the computational power of systems capable of solving mazes [1, 29], for example molecular walker-based systems.

The difficulty of maze-solving varies with the complexity of the maze, such as number of dimensions, grid layout versus more general graph, degree of nodes, or whether graph edges are directed or undirected. In computational complexity theory terminology, solving mazes and more general graph reachability problems lie within the class NL [1, 28, 29], i.e. problems solvable on a nondeterministic Turing machine that uses temporary workspace only logarithmic in input length. At the simplest level, and perhaps counter-intuitively, a system that solves a directed maze consisting of (a number of possibly disconnected) straight line segments has enough computational power to solve any problem in L, the deterministic version of NL [22].¹ Thus maze-solving lies between L and NL, depending on the complexity of the setup.

Here, we suggest two modifications to the maze-solving problem, which are expressive enough to endow maze solvers with significant computational power (their prediction problem becomes P-complete), yet, we contend, simple enough to be experimentally feasible using DNA engineering and computing principles. The first, and most important, is that we generalise mazes to have paths patterned with logic gates that must be solved in order to pass by them (Figure 1(c)). For a maze-walker this would mean it should be able to input one or two bits of information from the site it stands upon, compute, and then output one or two bits to adjacent sites. The second, mainly to keep things simple, is that we assume mazes are directed (meaning a pair of adjacent positions have one directed edge between

¹ The PATHREACHABILITY problem is L-complete: given a directed graph whose edges form a set of disconnected line segments (in- and out-degree ≤ 1), and two nodes s and t , is t reachable from s ? A deterministic Turing machine can start at s and walk along the graph use only logarithmic workspace (in input length) to keep track of the current node, answering “yes” if it reaches t and “no”, if it instead reaches a dead-end. Hence the problem is in L. Conversely, the set of configurations of a deterministic logspace Turing Machine can be encoded as a polynomial-sized instance of PATHREACHABILITY making that problem L-complete [22].



■ **Figure 2** Two small tiles sets. (a) NAND-NXOR tile set with 4 tile types. The south side computes the NAND of north and east, and west computes the NXOR of north and east. (b) Collatz tile set with 6 tiles, named for its relationship to the Collatz problem.

them that dictates the direction of information flow) and have no cycles. Since we allow for fanout of 0, 1 or 2 per site, one needs to generalise the typical notion of maze-solving somewhat: Are walkers replicating themselves to handle fanout of 2? Or are they leaving little bit-encoding messages for other walkers/themselves to pick up later? How do they handle fanin of 2? These considerations lend themselves to various models, however here we focus on having information-manipulating *tiles* flow through the maze, much like lava flowing down a complex volcanic hillside, but clever lava that computes as it moves. Our model is called the Maze-Walking Tile Assembly Model, or Maze-Walking TAM.

The programmer specifies a set of square tiles, with glues on the sides. A problem instance, or maze, is a set of polyominoes, painted with information-encoding glues. Starting at special input locations, tiles attach one at a time, asynchronously and in parallel, wherever they match glues on two sides.² A typical maze can be thought of as sending a unary (“route finding”) signal, whereas our mazes send bits and allow them to meet, interact and be changed.

In this setting, if we allow arbitrary numbers of tiles (or a clever enough walker, or a complex enough asynchronous cellular automaton rule) it is not difficult to see how to simulate arbitrary Boolean circuits. Take a circuit, make it planar by replacing each wire crossing with a crossover gate, then lay the circuit out on a maze-like grid with input gates on the east, and the output gate on the west. Then simply build a maze with walls tracing out the circuit wiring diagram and painted with arrows (wire directions) and logic gates, and require the output bit(s) to satisfy the circuit logic. The question we ask is: How clever does the maze-solver need to be in this computational setting? More precisely, we ask how many tile types are needed to execute any Boolean circuit in the Maze-Walking TAM?

1.1 Main results

Our first main result is for the NAND-NXOR tile set shown in Figure 2(a). In the theorem statement, by *simulated* we mean that the function computed by the circuit c is also computed by an instance of the Maze-Walking TAM (see Section 3.1).

² The model is equivalent to the abstract Tile Assembly Model [39, 52, 35, 16], with multiple disconnected seed assemblies, and where we have all tile bindings are by attachment to an assembly by two matching glues.

► **Theorem 1.** Any Boolean circuit c is simulated by the 4-tile NAND-NXOR tile set in the Maze-Walking TAM using assemblies containing ≤ 6 tiles per gate and 34 tiles per crossover gate in a planarisation of c .

Our second main result is for the Collatz tile set which has 6 tiles (Figure 2(b)) and is so-named because of its ability to embed iterations of the Collatz function (see Appendix A).

► **Theorem 2.** Any Boolean circuit c is simulated by the 6-tile Collatz tile set in the Maze-Walking TAM using assemblies containing ≤ 14 tiles per gate and 33 tiles per crossover gate in a planarisation of c .

We finish this section with a discussion of our two tile sets and some future directions. Section 2 sets these results in the context of other theoretical results and experimental directions. Section 3 defines the Maze-Walking TAM. We prove our two main theorems in Sections 4 and 5. Appendix A gives some background on the Collatz tile set.

1.2 Discussion: the NAND-NXOR and Collatz tile sets

Theorems 1 and 2 place focus on the size of assemblies that simulate gates. They omit estimates of the additional tiles (assemblies) required for the circuit wiring diagram, which warrants comment. Our work is partially motivated by a desire to build instances of the Maze-Walking TAM, and in doing so we would highly optimise any implemented circuit wiring diagram. Example circuit implementations, that recognise 3-bit prime numbers, are shown in Figures 3(j3) and 5(j1), both of which are optimised for short wire length. If we want to have a general wiring procedure for all circuits, and thus not optimised for particular classes of circuits, the overhead incurred will be rather large, typically $O(s^2)$ space for a circuit with s gates [9]. In practice we would not use such overly-bloated constructions.

The NAND-NXOR tile set was found by explicitly trying to find a small tile set: hence its use of a universal gate (NAND) on the south side (output). The NXOR gate (west side) helps with wire routing allows for even smaller gates than going via NAND-only-based circuit simulation. The Collatz tile set came out of thinking about iterations of the Collatz function in a local digit-by-digit, or tile-by-tile, way. In [46] a cellular automaton-like model is shown to simulate instances of the Collatz function – assemblies of our Collatz tile set show up in iterations (configurations) of that model. The Collatz tile set, along with the non-local rule in [46] (which can be simulated by the addition of two additional tile types, see Appendix A), is expressive enough to run Collatz. Here we applied computer search to the Collatz tile set to search for seed structures and assemblies that could be used to compute more generally. We leave as an open question as to what extent such structures, or other computational structures, naturally appear during iterations of the Collatz function – something the Collatz tile set might help us see.

For running Boolean circuits, if the only metric we cared about was tile set size, the NAND-NXOR tile set wins. However, looking beyond circuits, the Collatz tile set is capable of directly implementing certain arithmetical operations, such as computing powers of 2, powers of 3, and converting from base 3 to base 2 [46] (see Appendix A). These constructions use much simpler connected seeds than those given in the proof of Theorem 2, and lead to more efficient (smaller) assemblies than computing via tiles-simulating-circuits, for these kinds of arithmetical problems. In this paper, we used computer search to find that tiles capable of such arithmetical operations are also capable of running circuits, we leave it as future work to discover what other operations they are efficiently capable of.

Theorems 1 and 2 prove that the problem of predicting a tile at distance n from a size n connected seed, is P-hard (and in fact it is also P-complete if we assume directed/deterministic growth [43] since a deterministic Turing Machine simulates the entire assembly process in time polynomial in n). It is natural to ask if having maze-like (i.e. disconnected) seeds is necessary for such computational efficiency: we conjecture “yes”. That is, for both tile sets, we conjecture that prediction of the tile type that goes at a given position, at distance n from a size n connected seed and assuming directed growth, is in the complexity class NL. In particular this would mean that simulation of arbitrary Boolean circuits in the direct manner shown here is impossible, assuming the widely-believed conjecture $NL \neq P$. For the Collatz tile set, and for connected seeds of a certain form, we know that prediction is in NL (Appendix A). If one could show that prediction is P-hard, for seeds/inputs that represent natural numbers that occur during iterations of the Collatz function, one could in fact show that the Collatz process embeds rather powerful computational capabilities. Certainly a result of that form would change the perspective on the Collatz conjecture itself.

Our results were developed with assistance of a simulator: <https://github.com/tcosmo/mawatam>. The reader is invited to experience the results of this paper through the simulator.

1.3 Future work

Experimentally, future work involves implementing instances of the Maze-Walking TAM in the wet-lab, for instance, using a DNA origami as the underlying structure to encode maze seeds [7], building on the systems discussed in Section 2.2. One experimentally-relevant criticism of this work could be to ask why we focus on such small tile sets when we know that with DNA it is possible to build systems with hundreds of algorithmic DNA tiles [57]. First, we would say that no algorithmic system of such a high tile complexity, and that runs on the back of a DNA origami, has been engineered to date. Secondly, and of more relevance to this work, is that we are exploring the fundamental boundary and complexity trade-offs between computational power and systems size.

Theoretically, our work leaves open the following questions:

- Can Boolean circuit simulation, or any kind of universal computation, be achieved in the Maze-Walking TAM using tile sets with less than 4 tiles?
- Can interesting behaviour occur in the Maze-Walking TAM with just 1 tile? (At first sight, this question may look odd, however one could imagine encoding a bit by the absence or presence of a tile at a given position in the final assembly, leaving room for expressiveness in the Maze-Walking TAM with 1 tile.)
- Is the Maze-Walking TAM, with ≤ 4 tiles, intrinsically universal [17, 56] for the aTAM?

2 Related work: theoretical and experimental

2.1 Other routes to finding small universal tile sets

Existing small/simple universal models of computation [58] include the efficiently universal [11, 30] 2-state one-dimensional cellular automaton Rule 110, as well as universal Turing machines with just 22 instructions (5 states & 5 symbols, or 4 states & 6 symbols) [31, 38] or even just with 8 instructions (3 states, 3 symbols, but with the tape input embedded in an infinitely repeated pattern) [32].

In the context of the theory of molecular computing, and algorithmic self-assembly in particular, the smallest computationally universal self-assembling tile set to date seems to be a 7-tile system that can be derived from [57].³ However, that construction leads to large spatial blowup via Rule 110 simulation of $O(s^4 \log^2 s)$ for circuits of size s (Corollary S1.3, SI-A [57]). Another construction uses $O(w^2 d)$ tile types (for a depth d , width w circuit), essentially by hardcoding the routing of the circuit diagram in tile types (Theorem S1.5, SI-A [57]). Even direct implementation of a small universal Turing machine as a self-assembling tile set, using known methods, although presumably achievable with a few dozen tile types, would require large input encodings [58]. Other methods to obtain a single universal, or intrinsically universal, tile set, or even a single tile, also use indirect and large, albeit constant-factor in some cases, encoding methods [17, 15, 14, 43].

By allowing for more tile types than our constructions, one could have a maze with glues that explicitly encode gate type (one of sixteen), as well as glues encoding two bits at a time: that way a single tile attachment event could read two bits and a gate type simultaneously. This idea yields a constant-size tile set with perhaps a few dozen tile types. Although larger than ours, such an approach would have experimental merit. Cantu, Luchsinger, Schweller, and Wylie simulate Boolean circuits with tiles in a covert manner [5].

2.2 DNA-based implementations and related models

As future work we plan to give DNA-based designs and implementation for the Maze-Walking TAM. We imagine a 2D information-encoding structure that provides the maze pattern, for example a single flat DNA origami [40], or several DNA origamis tiled together [55, 27, 48, 49], or perhaps even a suitable DNA DX-tile, or single-stranded tile, structure [50, 53, 59, 57]. DNA-based systems for maze-solving have been implemented experimentally: using DNA origami (for the maze) along with hairpin activation [7] or controlled opening of track locations [51] for movement. The phenomenon of DNA condensation was also used for maze exploration [34]. Computation via tile-attachment in the Maze-Walking TAM could be implemented using design principles from algorithmic DNA self-assembly [57, 19], DNA-based molecular walkers that walk on 1D tracks and 2D DNA origami surfaces [60, 42, 41, 33, 20, 47], and other DNA systems that compute on surfaces [3, 4, 44, 6, 8]. Finally, there has been some theoretical and simulation-based analyses of molecular walkers [13, 37, 26, 12] including maze-solving walkers [45], as well as papers that study computation on surfaces [36, 10, 2] using a similar setup to ours but without molecular orientation and using different rule formats. All of these models (and ours) describe sub-classes of asynchronous cellular automata.

3 Definitions

3.1 Maze-Walking TAM definition

A *maze* is collection of non-intersecting polyominoes positioned on \mathbb{Z}^2 where each exterior unit-length square-side polyomino edge is labeled with a glue $g = (g', p)$ where $g' \in G$ is from a finite set of *glue types* G , that includes the null glue, and $p \in \{z + 0.5 \mid z \in \mathbb{Z}\}^2$ is a glue position. An *instance* of the Maze-Walking TAM $\mathcal{T} = (T, M)$ has a set of tile types T , where each $t \in T$ is a unit-sized square whose four sides labelled with four glue types from

³ In Figure S4(b), SI A, [57], gates g and f can be used to simulate Rule 110, and that in turn can be simulated by 4 tiles each. These 8 tiles can be further optimised to 7 tiles by sharing one glue type between both half-layers.

G , and a maze M . The process of *self-assembly* proceeds by tiles (instances of tile types) attaching asynchronously, and one at a time, wherever they match non-null glues on two sides (i.e. two-sided cooperative binding in the abstract Tile Assembly Model [52, 39, 35, 16]). An *assembly* is a maze with tiles attached (thus, assemblies may be connected or disconnected in 2D), and a *terminal assembly* is an assembly such that no tile can be attached.

The tile set T is said to *compute the function* $f : \{0, 1\}^n \rightarrow \{0, 1\}$ in the Maze-Walking TAM if there is a maze M' with n empty (no tile) tile positions $p_0, p_1, \dots, p_{n-1} \in \mathbb{Z}^2$ and an empty (no glue) output glue position $o \in \{z + 0.5 \mid z \in \mathbb{Z}\}^2$, such that adding n *input tiles* at p_0, p_1, \dots, p_{n-1} to M' is the new maze called M_x where the process of self-assembly on M_x yields a set of terminal assemblies that each have the bit $f(x)$ encoded by the glue at position o . (Here, we imagine a many-one encoding function from glue types to bits.)

Maze-Walking TAM systems may be *directed* (one terminal assembly), or *undirected* (several terminal assemblies). In this paper the systems we study are directed, which is equivalent to saying that, for all sequences of tile additions, at each position $p \in \mathbb{Z}^2$, there is at most one choice for what tile appears at p . Thus, in this paper, for a function f , for each $x \in \{0, 1\}^n$ there is an associated maze M_x such that $\mathcal{T}_x = (T, M_x)$ has a single terminal assembly that is said to compute $f(x)$. Finally, a Boolean circuit c (defined below) is said to be simulated by a tile set if the tile set computes the same function as c .

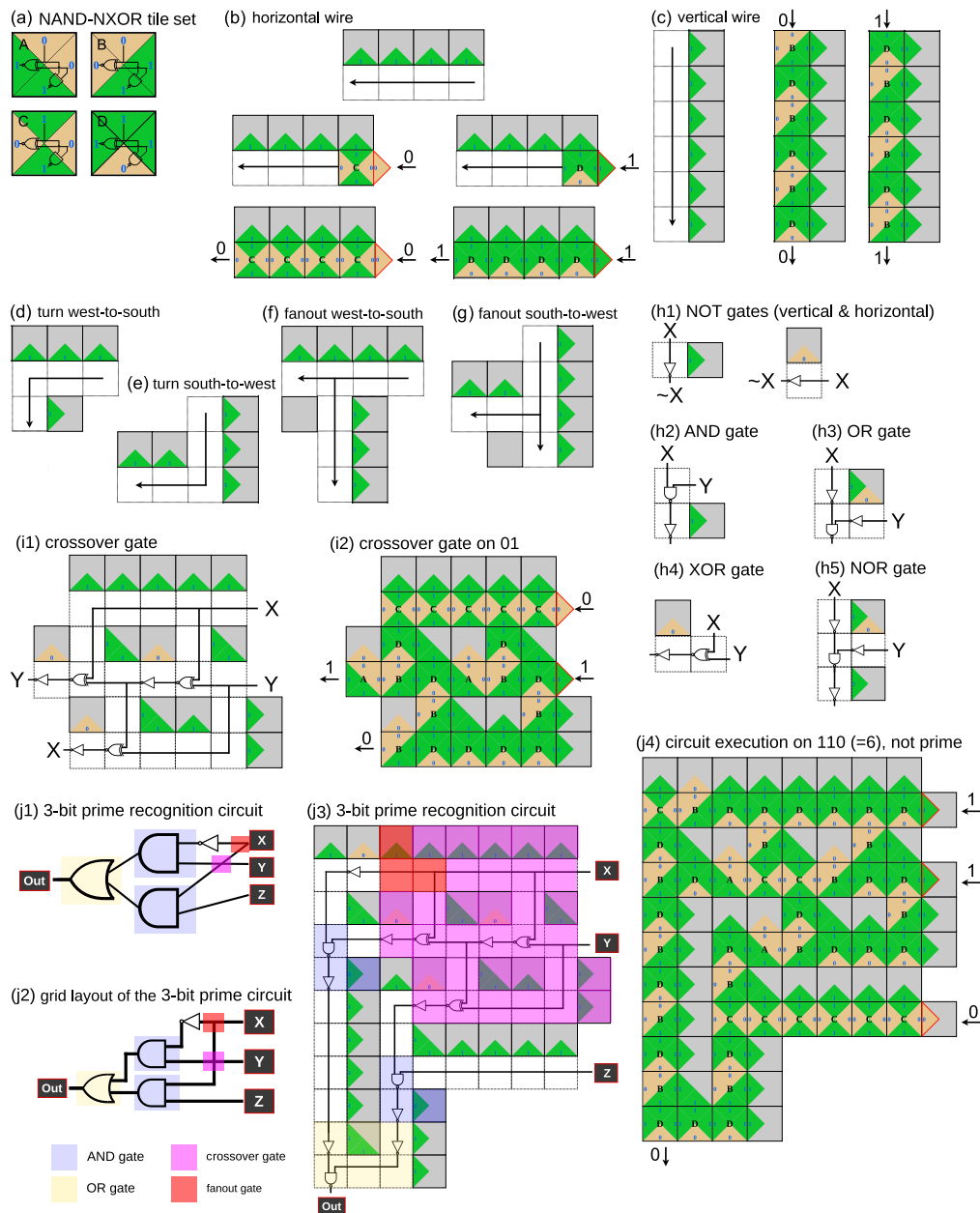
3.2 Boolean circuit definition

A Boolean circuit is a directed acyclic graph, where edges are called *wires*, and nodes are called *gates* and are labelled. In this paper, gates have out-degree 1 or 2, except for output gates that have out-degree 0. Also, a node's label is one of: input (with in-degree 0), output (with in-degree 1, out-degree 0), constant 0 or constant 1 (in-degree 0), fanout gates (in-degree 1, out-degree 2; makes two copies of its input), or is one of the *compute* gates (\neg , NOT of in- and out-degree 1, or any of the in-degree 2 out-degree 1 gates that compute functions on bits, e.g. OR, AND, NAND, NXOR,⁴ etc.). Also, we define an additional gate called a *crossover gate* (in- and out-degree of 2) which swaps its inputs, used to planarise a non-planar circuit (see below). Circuits compute, from the input gates and constant gates to the output gate, by modifying bits according to the functions specified by gate labels.

The *size* of a circuit is its number of gates, and its *depth* is the length of the longest path from any input gate to the output gate. A circuit c computes a Boolean (no/yes) function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ on $n \in \mathbb{N}$ Boolean variables, by its gates computing the bit value at the output in the usual way from the n input bits. A circuit is said to be planar if its graph is planar (can be laid out in the plane without wire crossings).

A *planarisation* of a Boolean circuit c is another Boolean circuit \hat{c} where \hat{c} computes the same function as c , has a planar embedding in \mathbb{R}^2 , and \hat{c} has exactly the gates of c plus zero or more 2-in 2-out crossover gates (that allow crossing of signals between a pair of wires that would otherwise intersect in the plane). In other words, c is converted to \hat{c} by adding crossover gates so that \hat{c} has a planar embedding. An example is shown in Figure 3(j2). A *planar Boolean circuit* c is a Boolean circuit where $\hat{c} = c$, i.e. \hat{c} has zero crossover gates.

8:8 Small Tile Sets That Compute While Solving Mazes



■ **Figure 3** Circuit-simulating gadgets for the NAND-NXOR tile set. In all parts of the construction growth proceeds to the west and south (and never north nor east). (a) NAND-NXOR tile set. Seed structures to implement (b) horizontal west-growing and (c) vertical south-growing wires. Examples of communicating of 0 and 1 are shown for each. Vertical wires are of even length; in cases where odd length is required we use a horizontal NOT gates during a turn from south-to-west (see proof of Theorem 1). (d) Turn west-to-south, (e) turn south-to-west, (f) fanout west-to-south, and (g) fanout south-to-west. The two isolated unit-size squares in (f,g) are there only to prevent unintended cooperative growth after a fanout. (h1–5) Various logic gates (full set in Figure 4). (i1) Crossover gate with an example in (i2) with design based on the 3 XOR gates construction given in [5]. (j1) An example Boolean circuit that decides whether a 3-bit number is prime. (j2) Circuit converted to a grid layout and (j3) implemented using NAND-NXOR tile gadgets. The implementation in (j3) is somewhat optimised for space efficiency. (j4) The terminal assembly (execution) for the circuit example on non-prime input $6_{10} = 110_2$.

4 Four tiles: the NAND-NXOR tile set

The NAND-NXOR tile set is depicted in Figure 3(a). One of the ideas underlying all of the constructions in this paper can be understood by the way horizontal wires are implemented with the NAND-NXOR tile set, Figure 3(b). A specific $n \times 1$ polyomino seed advertises “1” glues along its south side, which facilitates propagation to the west of any bit presented as a glue coming from the east, following the assembly rules prescribed by the tile set. As described in the proof of Theorem 1, the implementation of Boolean circuits using the NAND-NXOR tile set is based on canonical constructions of logic gates exploiting NAND, NOT and NXOR functions as primitive building blocks, Figure 3(h1–h5).

► **Theorem 1.** Any Boolean circuit c is simulated by the 4-tile NAND-NXOR tile set in the Maze-Walking TAM using assemblies containing ≤ 6 tiles per gate and 34 tiles per crossover gate in a planarisation of c .

Proof. A circuit is simulated by appropriately placing gadgets together to form a maze.

Tiles simulating wires and gates. We will show that the gadgets in Figure 3 are building blocks (for a maze) that advertise glues designed to force directed growth when given appropriate bit-encoding glue input(s).

Figure 3(b,c) details how the NAND-NXOR tile set simulates horizontal and vertical wires. Vertical tile-wires have a parity constraint: in a vertical wire carrying the bit $x \in \{0, 1\}$, every second tile correctly advertises x to the south, and every other tile advertises its negation $\sim x$. If the circuit’s layout requires a turn from south-to-west, from an *odd length* vertical wire (advertises $\sim x$) then a single horizontal negation gadget (Figure 3(h1, right)) is placed at the bottom of the wire to change the signal to x (correct the “error”). With that correction, vertical and horizontal wire segments can be used to send a signal from the origin to any location in the south-west quadrant of \mathbb{Z}^2 .

Figure 3(d–g,h1–h5,i1) shows two turns (south-to-west and west-to-south) and two kinds of fanout-2 gates, as well as a number of compute gates and a crossover gate. In addition NAND, and NXOR, gates are shown in Figure 3(a): present inputs x, y at North and East, and read NXOR(x, y) on West and/or NAND(x, y) on South. (For completeness, Figure 4 gives direct simulations of all 16 possible gates with 1 or 2 inputs and one output.) No gate is larger than NOR (see Figure 3(h5) and Figure 4), which uses 6 tiles. The crossover gate is simulated using 34 tiles (intuitively, it uses a well-known idea of implementing crossover with three XOR gates and three fanout gates). This gives the size bounds on tiles per gate and crossovers in the theorem statement.

We claim that each gadget in Figures 3(b–g,h1–h5,i1) and Figure 4 is directed, meaning that after input glue(s) are given to the gadget, then for each unit-sized outlined/dotted empty square region in the gadget there is exactly one tile type that can be placed. This can be seen by noting that (i) for all gadgets, and all inputs to a gadget, tiles attach using their North and East sides only, and by (ii) the fact that the NAND-NXOR tile set is deterministic on North and East sides.

Laying the circuit out on a grid. For the Boolean circuit c , let \hat{c} be its planarisation as defined in Section 3.2; a planarisation always exists – just draw the circuit on the plane replacing each of the $s' \in \mathbb{N}$ wire crossings with a crossover gate (various planarisations may be used to optimise s' , or other circuit parameters).

⁴ In this paper we use the notation $\text{NXOR}(x, y) = \text{NOT}(\text{XOR}(x, y))$ (and read “NOT exclusive OR”) to denote what is more commonly, but confusingly, written XNOR (read “exclusive NOR”).

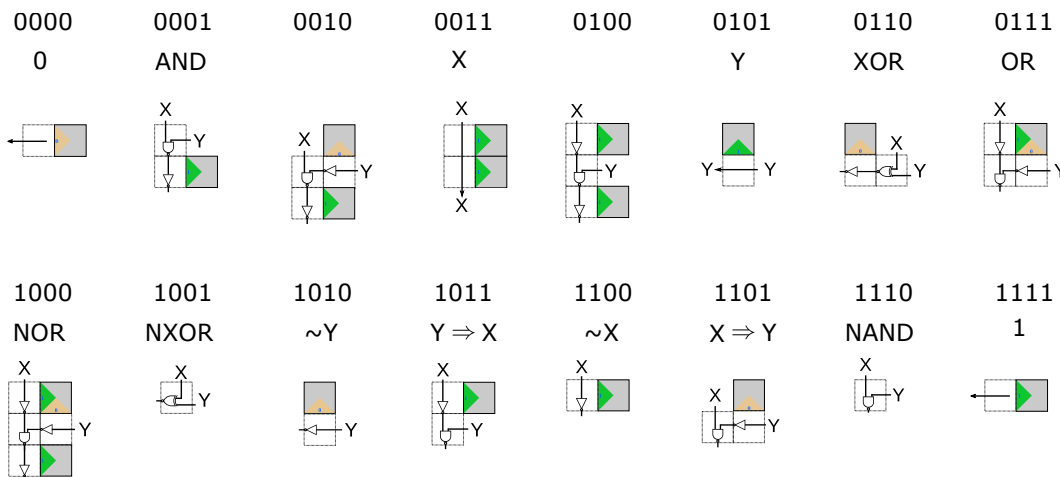
Second, we layer c : meaning that we organise gates (including crossover gates) of c into consecutive layers with layer 0 containing all input and constant gates, and so that layer i contains gates that take their inputs from the outputs of gates in layers $< i$. The number of layers is equal to the depth d of c , with the output gate being the sole gate in layer $d - 1$. More precisely, layer i is located at x-coordinate $-i$ (our convention is to draw circuits from right to left).

Third, we increase the height between gates, and width between layers, so that there is enough room to draw all wires so that they are composed of horizontal and vertical segments only (where information flows to the west and to the south, respectively), that meet at right angles (thus wires have south-to-west and west-to-south turns, only). We call the resulting circuit a grid-layout circuit, and an example given in Figure 3(j2). Using the gadgets described above, the maze/seed structure traces out the wires and gate locations according to the south-west grid-layout circuit, leaving enough room so that gates and wires do not intersect.

Computation. For any circuit c we have described (at a high level) how to lay out a maze M' , in the notation of Section 3.1. We next need to encode circuit inputs, as follows. Since input gates are instances of gates, we assume that in M' there are n tile positions that are empty and positioned adjacent to wires (so that their bit values will feed into a layer of gates via horizontal wire gadgets). Let n be the number of inputs to c and let $x = x_0x_1 \cdots x_{n-1} \in \{0, 1\}^n$ denote an input to c . To the maze M' we add n more tiles so that the n input glue positions of the maze are of respective types $x_0x_1 \cdots x_{n-1}$, to give an maze M_x that encodes x (the example in Figure 3(j4) has 3 encoded input bits).

Assembly proceeds, starting at each of the n input glues in parallel (and at any positions that encode 0/1 constant bits), according to the Maze-Walking TAM definition (Section 3.1). Throughout the entire self-assembly process, at each position there is exactly one tile type that can be placed (this is because it is true for individual gadgets as already argued). Also, the self-assembly process terminates, for the simple reason that no tile can attach outside of the bounding box of the maze M_x . Thus one terminal assembly is eventually produced, that by its definition, encodes an execution of the circuit c with the output bit presented at the glue position that represents the simulated circuit output gate (labeled “out” in the example in Figure 3(j3)). ◀

► **Example 3.** Figure 3(j1-j4) illustrates the general construction described in Theorem 1 in the context of a circuit that recognises prime numbers on 3 bits, i.e. the circuit will output 1 if and only if $xyz \in \{010, 011, 101, 111\}$ which are the binary encodings of numbers $\{2, 3, 5, 7\}$. The circuit implements the formula: $((\text{NOT } x) \text{ AND } y) \text{ OR } (x \text{ AND } z)$ and uses one crossover as well as one fanout gate, Figure 3(j1). To facilitate the final Maze-Walking TAM implementation, the circuit is laid out on a grid using only south-to-west and west-to-south turns, Figure 3(j2). Then, the circuit is implemented with tiles, Figure 3(j2), using the gadgets of Figure 3 and finally, the circuit executes on input $110_2 = 6$ and outputs 0 as 6 is not prime, Figure 3(j4). Note two details: (1) The implementation of the crossover gate, Figure 3(i1), contains three embedded XOR gadgets and three embedded fanout gadgets – using tiles to implement a known construction to simulate crossover with XORs. (2) The way the OR gate is implemented in Figure 3(j3) (yellow overlay) is slightly different than Figure 3(h3) as the negation of the east-coming input is performed vertically instead of horizontally; this is an optimisation that exploits the difference in length parity of the two vertical wires coming in to the gate.



■ **Figure 4** Implementation of all 2-input 1-output Boolean gates using gadgets over the NAND-NXOR tile set in the Maze-Walking TAM. The gadgets are ordered with respect to their truth table, which refers to the 4-bit output of the 4 respective inputs 00, 01, 10, 11; i.e. the canonical truth-table definition of a 2-in 1-out gate (we use the same notation for gates with one (NOT, identity) or zero inputs (constants)). For instance, the truth table 1101 encodes gate g such that $g(00) = 1$, $g(01) = 1$, $g(10) = 0$ and $g(11) = 1$. The common English name of the gate is also given when there is one. The constant gadgets (0000 and 1111) are used to simulate constant gates (0/1) and circuit input gates $x_i \in \{0, 1\}$, and require the presence of an additional glue (not shown) to trigger growth, e.g. by being placed next to a wire gadget as shown in Figure 3(j4).

5 Six tiles: the Collatz tileset

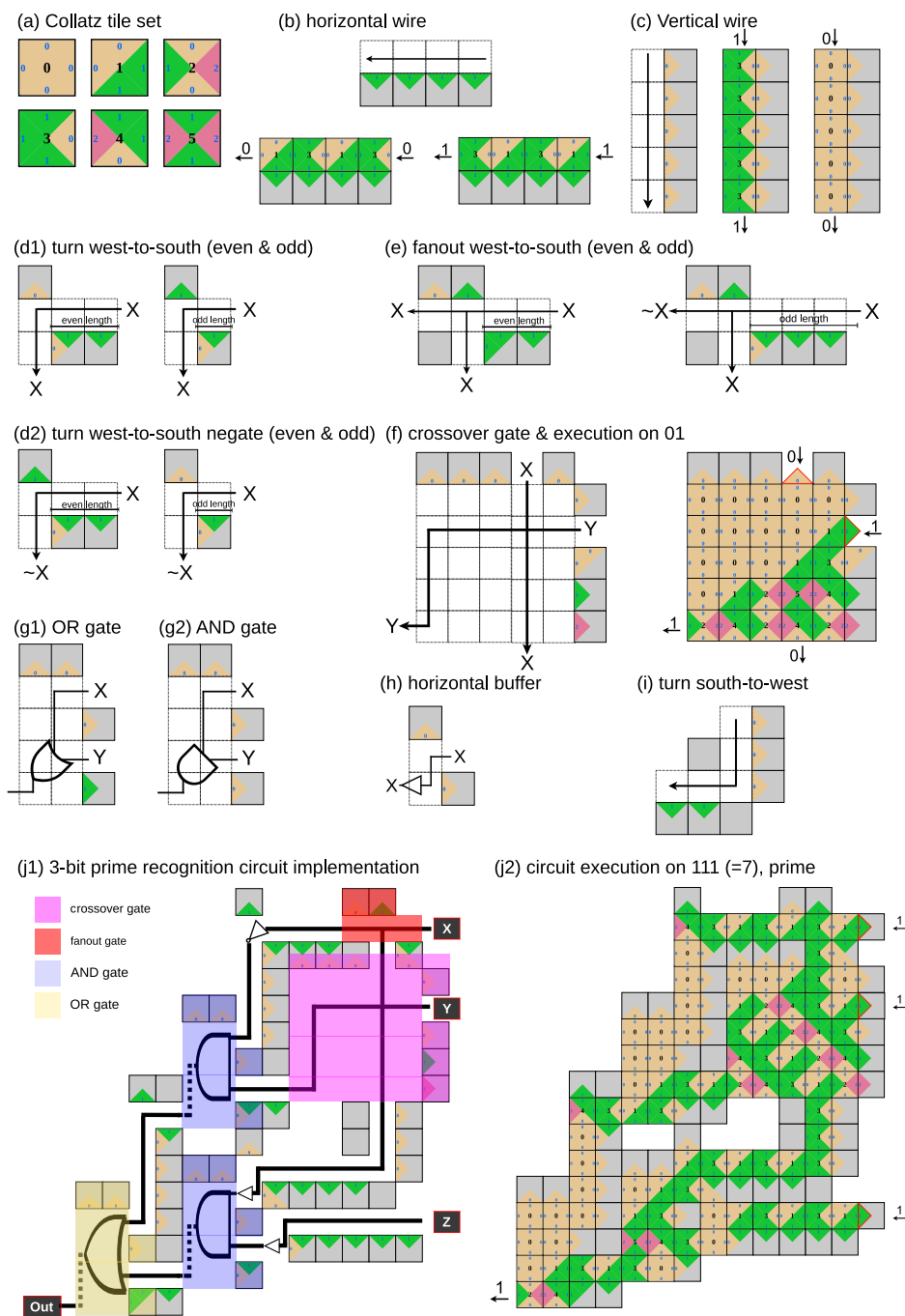
In this section, we illustrate efficient Boolean circuit simulation in the Maze-Walking TAM with the Collatz tile set which consists of of 6 tile types and 3 glues and is shown in Figure 2(b).

On the one hand, the NAND-NXOR tile set was explicitly designed to compute, via the placement of a single tile, the universal NAND function. From there it was augmented (with bits on the west sides) that facilitate simulation of circuit wiring, and efficient simulation (few tiles) of non-NAND gates. On the other hand, the Collatz tile set came about from studies on the Collatz problem. Specifically, glue patterns in some tiled regions (e.g. rectangles) relate to notoriously hard mathematical problems such as the Collatz conjecture [46] or an open problem of Erdős’ [18, 25]: Is it the case that for all $n > 8$ there is at least one 2 in the ternary representation of 2^n ? For more details see Appendix A. We noticed that this pattern complexity could be leveraged, with the aid of computer search⁵, to build gadgets for computation in the Maze-Walking TAM (Figure 5).

► **Theorem 2.** Any Boolean circuit c is simulated by the 6-tile Collatz tile set in the Maze-Walking TAM using assemblies containing ≤ 14 tiles per gate and 33 tiles per crossover gate in a planarisation of c .

⁵ Computer search was performed through the Maze-Walking TAM simulator: <https://github.com/tcosmo/mawatam>

8:12 Small Tile Sets That Compute While Solving Mazes



■ **Figure 5** Circuit-simulating gadgets for the Collatz tile set. Growth proceeds to the west and south exclusively. (a) the Collatz tile set. Seed structures to implement (b) horizontal west-growing and (c) vertical south-growing wires. Horizontal wires are of even length. When turning to the south the appropriate turn can be used to transmit the signal (d1) or its negation (d2). (e) Fanout gadgets depending on the parity of the incoming horizontal wire, if the length is odd, the gadget also negates the west-going signal. (f) The smallest crossover gate found by computer search. (g) Common Boolean gates, also found by computer search. (h) The buffer gadget is used to change the parity of an horizontal wire. (i) Turn south-to-west. (j1) Collatz-tileset implementation of the 3-bit prime recognition circuit and (j2) execution of the circuit on $7_{10} = 111_2$ which is prime.

Proof.

Tiles simulating wires and gates. We will show that the gadgets in Figure 5 can be used to build mazes that simulate arbitrary Boolean circuits and that the growth triggered by the placement of input tiles is directed, which in turn implies that the correct bit is output by the simulation of c on some binary input word x .

Figure 5(b,c) details how the Collatz tile set simulates horizontal and vertical wires. Horizontal tile-wires have a parity constraint: in a horizontal wire carrying the bit $x \in \{0, 1\}$, every second tile correctly advertises x to the west, and every other tile advertises its negation $\sim x$. To handle this, there are two west-to-south turns, one for turning from even length, and one for turning from odd length, horizontal wires Figure 5(d1). Only west-to-south fanout is used in the constructions with this tileset, Figure 5(e). This fanout gate comes in two variants whether it is applied at an even or an odd horizontal wire position. If the gadget is applied at an odd wire position, it has the particularity of negating the output west-going signal.

Negating a signal (either to correct a horizontal parity effect, or to simulate a NOT gate) can be achieved in several ways. If the signal ever turns south, this can easily be done thanks to Figure 5(d2) which implements both a turn and a negation at the same time. If the signal never turns south, the programmer can use an odd-length horizontal wire which implements a negation. If using an odd-length horizontal wire is not possible given the constraints on circuit layout, the programmer can use the horizontal buffer gadget Figure 5(h) which has the effect of copying the incoming signal to the next immediate column to the west which inverts the parity constraint of the horizontal wire and allows it to reproduce the behavior of an odd-length horizontal wire. This method is used in Figure 5(j1), for instance on the horizontal wire which connects the input Z to its target AND gate.

Glue labelled polyominoes, or seed structures, for south-to-west turns is shown in Figure 5(i). Notably, a growth stopper (1×1 polyomino, with four null glues) is used to prevent spurious growth that would happen in the north-west direction otherwise.

A crossover gadget seed structure is given in Figure 5(f), it was the smallest found by computer search and it costs 33 tiles. The gate preserves the horizontal alignment of the incoming northern bit: it exits at the south of the gate at the same x -position that it entered. However, the incoming eastern bit is deviated three units to the south.

Seed (polyomino) structures that simulate Boolean (compute) gates are rectangular and were found by computer search using the input convention that signals come from the east and, if there are two of them the inputs should be one vertical block apart⁶. Figure 5(g1,g2) gives the seed structure of an OR gate and an AND gate. For completeness, Figure 6 gives the implementation of all Boolean gates, the biggest of them is NOR with a cost of 14 tiles. This gives the tiles bounds per gate and crossover in the theorem statement. Remarkably, seed structures for AND, OR, NAND, NOR are very similar in the sense that they differ by at most 2 glues.

We claim that each gadget in Figure 5 and Figure 6 is directed, meaning that after input glues are supplied to the gadget then for each dotted region in the gadget there is exactly one tile type that can be placed. This can be seen by noting that (i) all gadgets use either North and East sides to attach or South and East sides to attach (South and East attachments are only used for horizontal wires and turn south-to-west gadgets, Figure 5(b,i)), (ii) North

⁶ Using computer search, we were able to find rectangular seed structures of Boolean gates corresponding to all the input conventions that we experimented with. This leads us to believe that the ability of the Collatz tileset to simulate Boolean gates is not tied to a particular input convention.

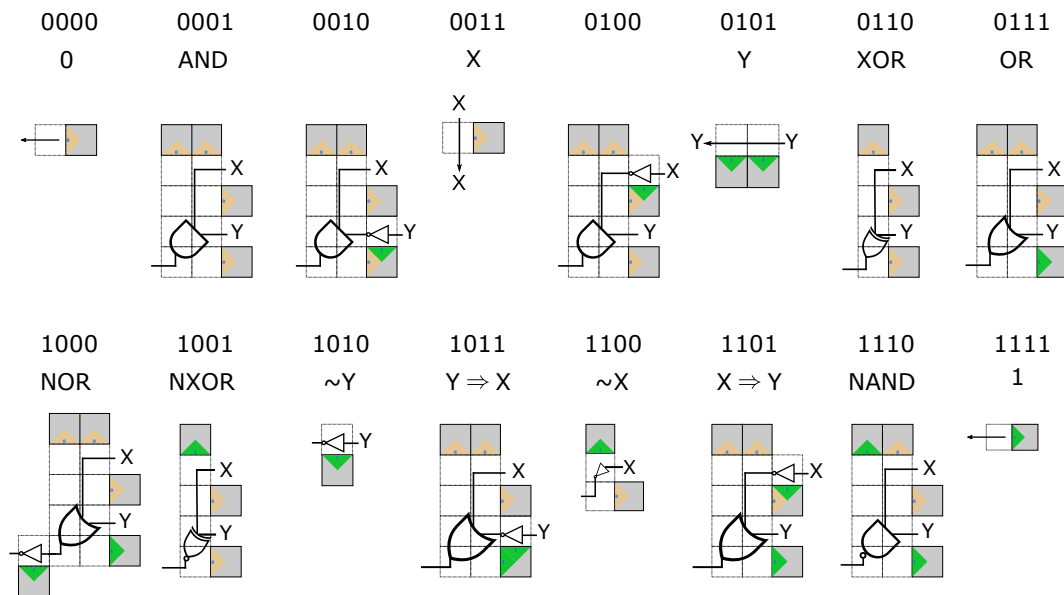
8:14 Small Tile Sets That Compute While Solving Mazes

and East attachments cannot compete with South and East attachments because all signals travel in the south-west direction and South and East constraints are never given directly by the seed but occur after tiles attach, and (iii) the Collatz tile set is deterministic on North and East sides and South and East sides.

Laying the circuit out on a grid. We use the same circuit layout technique given in the the proof of Theorem 1.

Computation. Similarly to the proof of Theorem 1, throughout the entire assembly process, because of the directedness of all the gadgets that we use, at each position there is exactly one tile type that can be placed. Thus one final assembly is produced, that encodes an execution of the circuit, and in particular outputs the same bit as the n -bit circuit c on any input word $x \in \{0, 1\}^n$. ◀

► **Example 4.** The 3-bit prime recognition circuit in Figure 3(j1,j2) is implemented using the Collatz tile set in Figure 5(j1,j2).



■ **Figure 6** Implementation of all 2-input 1-output Boolean gates using gadgets over the Collatz tile set in the Maze-Walking TAM. The gadgets are ordered with respect to their truth table which refers to the 4-bit output of the 4 respective inputs 00, 01, 10, 11; i.e. the canonical truth-table definition of a 2-in 1-out gate (we use the same notation for gates with one (NOT, identity) or zero inputs (constants)). For instance, the truth table 1101 encodes gate g such that $g(00) = 1$, $g(01) = 1$, $g(10) = 0$ and $g(11) = 1$. The common English name of the gate is also given when there is one. The constant gadgets (0000 and 1111) are used to simulate constant gates (0/1) and circuit input gates $x_i \in \{0, 1\}$, and require the presence of an additional glue (not shown) to trigger growth, e.g. by being placed next to a wire gadget as shown in Figure 5(j2).

References

- 1 Eric Allender, David A Mix Barrington, Tanmoy Chakraborty, Samir Datta, and Sambuddha Roy. Planar and grid graph reachability problems. *Theory of Computing Systems*, 45(4):675–723, 2009.
- 2 Tatiana Brailovskaya, Gokul Gowri, Sean Yu, and Erik Winfree. Reversible computation using swap reactions on a surface. In Chris Thachuk and Yan Liu, editors, *DNA Computing and Molecular Programming*, pages 174–196, Cham, 2019. Springer International Publishing.
- 3 Hieu Bui, Vincent Miao, Sudhanshu Garg, Reem Mokhtar, Tianqi Song, and John Reif. Design and analysis of localized DNA hybridization chain reactions. *Small*, 13(12):1602983, 2017.
- 4 Hieu Bui, Shalin Shah, Reem Mokhtar, Tianqi Song, Sudhanshu Garg, and John Reif. Localized DNA hybridization chain reactions on DNA origami. *ACS Nano*, 12(2):1146–1155, February 2018.
- 5 Angel A Cantu, Austin Luchsinger, Robert Schweller, and Tim Wylie. Covert computation in self-assembled circuits. *Algorithmica*, 83(2):531–552, 2021. arXiv preprint: [arXiv:1908.06068](https://arxiv.org/abs/1908.06068).
- 6 A. R. Chandrasekaran, O. Levchenko, D. S. Patel, M. MacIsaac, and K. Halvorsen. Addressable configurations of DNA nanostructures for rewritable memory. *Nucleic Acids Res*, 45(19):11459–11465, November 2017.
- 7 Jie Chao, Jianbang Wang, Fei Wang, Xiangyuan Ouyang, Enzo Kopperger, Huajie Liu, Qian Li, Jiye Shi, Lihua Wang, Jun Hu, Lianhui Wang, Wei Huang, Friedrich C. Simmel, and Chunhai Fan. Solving mazes with single-molecule DNA navigators. *Nature Materials*, 18(3):273–279, March 2019.
- 8 Gourab Chatterjee, Neil Dalchau, Richard A. Muscat, Andrew Phillips, and Georg Seelig. A spatially localized architecture for fast and modular DNA computing. *Nature Nanotechnology*, 12(9):920–927, September 2017.
- 9 Marek Chrobak and Thomas H Payne. A linear-time algorithm for drawing a planar graph on a grid. *Information Processing Letters*, 54(4):241–246, 1995.
- 10 Samuel Clamons, Lulu Qian, and Erik Winfree. Programming and simulating chemical reaction networks on a surface. *Journal of The Royal Society Interface*, 17(166):20190790, 2020.
- 11 Matthew Cook. Universality in elementary cellular automata. *Complex Systems*, 15, 2004.
- 12 Neil Dalchau, Harish Chandran, Nikhil Gopalkrishnan, Andrew Phillips, and John Reif. Probabilistic analysis of localized DNA hybridization circuits. *ACS synthetic biology*, 4(8):898–913, 2015.
- 13 Frits Dannenberg, Marta Kwiatkowska, Chris Thachuk, and Andrew J. Turberfield. DNA walker circuits: Computational potential, design, and verification. In David Soloveichik and Bernard Yurke, editors, *DNA Computing and Molecular Programming*, pages 31–45, Cham, 2013. Springer International Publishing.
- 14 Erik D. Demaine, Martin L. Demaine, Sándor P. Fekete, Matthew J. Patitz, Robert T. Schweller, Andrew Winslow, and Damien Woods. One tile to rule them all: Simulating any tile assembly system with a single universal tile. In *ICALP: Proceedings of the 41st International Colloquium on Automata, Languages, and Programming*, volume 8572 of *LNCS*, pages 368–379. Springer, 2014. Arxiv preprint: [arXiv:1212.4756](https://arxiv.org/abs/1212.4756).
- 15 Erik D. Demaine, Matthew J. Patitz, Trent A. Rogers, Robert T. Schweller, Scott M. Summers, and Damien Woods. The two-handed tile assembly model is not intrinsically universal. In *ICALP: Proceedings of the 40th International Colloquium on Automata, Languages, and Programming*, volume 7965 of *LNCS*, pages 400–412. Springer, 2013. Arxiv preprint: [arXiv:1306.6710](https://arxiv.org/abs/1306.6710).
- 16 David Doty. Theory of algorithmic self-assembly. *Communications of the ACM*, 55(12):78–88, 2012.
- 17 David Doty, Jack H. Lutz, Matthew J. Patitz, Robert T. Schweller, Scott M. Summers, and Damien Woods. The tile assembly model is intrinsically universal. In *FOCS: Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science*, pages 439–446. IEEE, 2012. Arxiv preprint: [arXiv:1111.3097](https://arxiv.org/abs/1111.3097).

- 18 Paul Erdős. Some unconventional problems in number theory. *Mathematics Magazine*, 52(2):67–70, 1979. doi:10.1080/0025570X.1979.11976756.
- 19 Constantine Evans. *Crystals that count! Physical principles and experimental investigations of DNA tile self-assembly*. PhD thesis, Caltech, 2014.
- 20 Hongzhou Gu, Jie Chao, Shou-Jun Xiao, and Nadrian C Seeman. A proximity-based programmable DNA nanoscale assembly line. *Nature*, 465(7295):202–205, 2010.
- 21 William Hesse, Eric Allender, and David A Mix Barrington. Uniform constant-depth threshold circuits for division and iterated multiplication. *Journal of Computer and System Sciences*, 65(4):695–716, 2002.
- 22 Neil Immerman. *Descriptive Complexity*. Springer, 1999.
- 23 Jeffrey C. Lagarias. The $3x + 1$ problem and its generalizations. *The American Mathematical Monthly*, 92(1):3–23, 1985. URL: <http://www.jstor.org/stable/2322189>.
- 24 Jeffrey C. Lagarias. The $3x + 1$ problem: An annotated bibliography (1963–1999) (sorted by author), 2003. arXiv:math/0309224.
- 25 Jeffrey C. Lagarias. Ternary expansions of powers of 2. *Journal of the London Mathematical Society*, 79(3):562–588, 2009. doi:10.1112/jlms/jdn080.
- 26 Matthew R Lakin, Rasmus Petersen, Kathryn E Gray, and Andrew Phillips. Abstract modelling of tethered DNA circuits. In *International Workshop on DNA-Based Computers*, pages 132–147. Springer, 2014.
- 27 Wenyan Liu, Hong Zhong, Risheng Wang, and Nadrian C Seeman. Crystalline two-dimensional DNA-origami arrays. *Angewandte Chemie International Edition*, 50(1):264–267, 2011.
- 28 Christopher Moore and Stephan Mertens. *The nature of computation*. Oxford University Press, 2011.
- 29 Niall Murphy and Damien Woods. AND and/or OR: Uniform polynomial-size circuits. In *MCU 2013: Machines, Computations and Universality. Electronic Proceedings in Theoretical Computer Science (EPTCS)*, volume 128, pages 150–166, 2012. arXiv:1212.3282v2.
- 30 Turlough Neary and Damien Woods. P-completeness of cellular automaton Rule 110. In *ICALP: International Colloquium on Automata, Languages, and Programming*, volume 4051, part 1 of LNCS, pages 132–143. Springer, 2006.
- 31 Turlough Neary and Damien Woods. Four small universal Turing machines. *Fundamenta Informaticae*, 91(1):123–144, 2009.
- 32 Turlough Neary and Damien Woods. Small weakly universal Turing machines. In *International Symposium on Fundamentals of Computation Theory*, pages 262–273. Springer, 2009.
- 33 Tosan Omabegho, Ruojie Sha, and Nadrian C Seeman. A bipedal DNA brownian motor with coordinated legs. *Science*, 324(5923):67–71, 2009.
- 34 Günther Pardatscher, Dan Bracha, Shirley S Daube, Ohad Vonshak, Friedrich C Simmel, and Roy H Bar-Ziv. DNA condensation in one dimension. *Nature nanotechnology*, 11(12):1076–1081, 2016.
- 35 Matthew J. Patitz. An introduction to tile-based self-assembly and a survey of recent results. *Natural Computing*, 13(2):195–224, 2014.
- 36 Lulu Qian and Erik Winfree. Parallel and scalable computation and spatial dynamics with DNA-based chemical reaction networks on a surface. In Satoshi Murata and Satoshi Kobayashi, editors, *DNA Computing and Molecular Programming*, pages 114–131. Cham, 2014. Springer International Publishing.
- 37 John H. Reif and Sudheer Sahu. Autonomous programmable DNA nanorobotic devices using dnazymes. *Theoretical Computer Science*, 410(15):1428–1439, 2009. Aspects of Molecular Self-Assembly.
- 38 Yuri Rogozhin. Small universal turing machines. *Theoretical Computer Science*, 168(2):215–240, 1996.
- 39 Paul W K Rothemund and Erik Winfree. The program-size complexity of self-assembled squares. In *STOC: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 459–468. ACM, 2000.
- 40 Paul WK Rothemund. Folding DNA to create nanoscale shapes and patterns. *Nature*, 440(7082):297–302, 2006.

- 41 Sudheer Sahu, Thomas H LaBean, and John H Reif. A DNA nanotransport device powered by polymerase phi29. *Nano letters*, 8(11):3870–3878, November 2008.
- 42 William B Sherman and Nadrian C Seeman. A precisely controlled DNA biped walking device. *Nano letters*, 4(7):1203–1207, 2004.
- 43 David Soloveichik and Erik Winfree. Complexity of self-assembled shapes. *SIAM Journal on Computing*, 36(6):1544–1569, 2007. doi:10.1137/S0097539704446712.
- 44 Tianqi Song, Shalin Shah, Hieu Bui, Sudhanshu Garg, Abeer Eshra, Daniel Fu, Ming Yang, Reem Mokhtar, and John Reif. Programming DNA-based biomolecular reaction networks on cancer cell membranes. *Journal of the American Chemical Society*, 141(42):16539–16543, October 2019.
- 45 Darko Stefanovic. Maze exploration with molecular-scale walkers. In Adrian-Horia Dediu, Carlos Martín-Vide, and Bianca Truthe, editors, *Theory and Practice of Natural Computing*, pages 216–226, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- 46 Tristan Stérin and Damien Woods. The Collatz process embeds a base conversion algorithm. In Sylvain Schmitz and Igor Potapov, editors, *RP2020: 14th International Conference on Reachability Problems*, volume 12448 of *LNCS*, pages 131–147. Springer, 2020. arXiv:2007.06979 [cs.DM].
- 47 Anupama J. Thubagere, Wei Li, Robert F. Johnson, Zibo Chen, Shayan Doroudi, Yae Lim Lee, Gregory Izatt, Sarah Wittman, Niranjana Srinivas, Damien Woods, Erik Winfree, and Lulu Qian. A cargo-sorting DNA robot. *Science*, 357(6356), 2017.
- 48 Grigory Tikhomirov, Philip Petersen, and Lulu Qian. Fractal assembly of micrometre-scale DNA origami arrays with arbitrary patterns. *Nature*, 552(7683):67–71, 2017.
- 49 Grigory Tikhomirov, Philip Petersen, and Lulu Qian. Programmable disorder in random DNA tilings. *Nature nanotechnology*, 12(3):251, 2017.
- 50 Bryan Wei, Mingjie Dai, and Peng Yin. Complex shapes self-assembled from single-stranded DNA tiles. *Nature*, 485(7400):623–626, 2012.
- 51 Shelley F. J. Wickham, Jonathan Bath, Yousuke Katsuda, Masayuki Endo, Kumi Hidaka, Hiroshi Sugiyama, and Andrew J. Turberfield. A DNA-based molecular motor that can navigate a network of tracks. *Nature Nanotechnology*, 7(3):169–173, March 2012. doi:10.1038/nnano.2011.253.
- 52 Erik Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, California Institute of Technology, 1998.
- 53 Erik Winfree, Furong Liu, Lisa A Wenzler, and Nadrian C Seeman. Design and self-assembly of two-dimensional DNA crystals. *Nature*, 394(6693):539–544, 1998.
- 54 Günther J. Wirsching. *The dynamical system generated by the $3n + 1$ function*. Springer, Berlin New York, 1998.
- 55 Sungwook Woo and Paul WK Rothemund. Programmable molecular recognition based on the geometry of DNA nanostructures. *Nature chemistry*, 3(8):620, 2011.
- 56 Damien Woods. Intrinsic universality and the computational power of self-assembly. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 373(2046):20140214, 2015.
- 57 Damien Woods, David Doty, Cameron Myhrvold, Joy Hui, Felix Zhou, Peng Yin, and Erik Winfree. Diverse and robust molecular algorithms using reprogrammable DNA self-assembly. *Nature*, 567(7748):366–372, 2019.
- 58 Damien Woods and Turlough Neary. The complexity of small universal Turing machines: A survey. *Theoretical Computer Science*, 410(4-5):443–450, 2009.
- 59 Hao Yan, Thomas H. LaBean, Liping Feng, and John H. Reif. Directed nucleation assembly of DNA tile complexes for barcode-patterned lattices. *Proceedings of the National Academy of Sciences*, 100(14):8103–8108, 2003.
- 60 P. Yin, H. Yan, X. G. Daniell, A. J. Turberfield, and J. H. Reif. A unidirectional DNA walker that moves autonomously along a track. *Angewandte Chemie*, 43(37):4906–4911, September 2004.

A Origins of the Collatz tile set

The Collatz problem is a notoriously hard open problem which lies at the intersection of mathematics and computer science [23, 54, 24]. Formulated in the 30s, the Collatz problem is dauntingly simple to express: consider the Collatz map $T : \mathbb{N} \rightarrow \mathbb{N}$ defined by $T(x) = x/2$ if x is even and $T(x) = (3x + 1)/2$ if x is odd. The Collatz conjecture states that iterating T , starting from any $n \in \{1, 2, 3, \dots\}$, eventually yields 1.

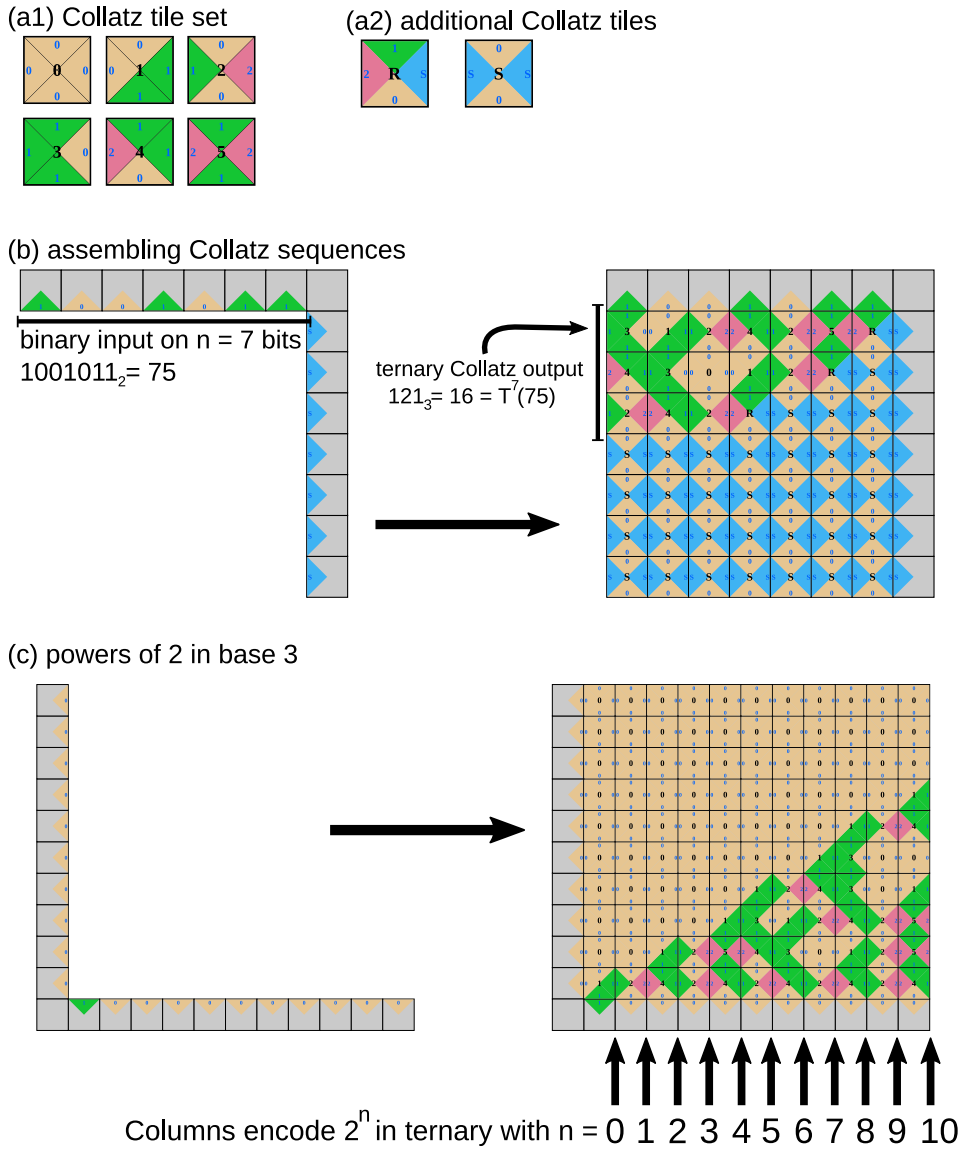
The Collatz tile set consists of six tiles, named “0” to “5”, which are depicted in Figure 7(a1). Vertical glues (north and south) are binary digits (0 and 1) while horizontal glues (east and west) are ternary digits (0, 1 and 2). Each tile is uniquely identified by its north-east corner (pair of glues) or its south-east corner or its south-west corner. Tile names are linked to the tile’s glues by the following arithmetical relation: for the tile named x (with $0 \leq x < 6$) we have:

$$x = 3N + E = 2W + S \tag{1}$$

where N, E, W and S respectively denote the values of the North, East, West and South glues. This tile set, among all tile sets which use binary (0, 1) vertical glues and ternary (0, 1, 2) horizontal glues, is the largest tile set for which (1) holds, by the following argument. Indeed, (1) corresponds to the Euclidean division of x by 3 and by 2, meaning that, for a given pair $(N, E) \in \{0, 1\} \times \{0, 1, 2\}$ there is a unique corresponding pair $(S, W) \in \{0, 1\} \times \{0, 1, 2\}$. Since there are 6 different (N, E) pairs we deduce that there are exactly 6 different tiles with binary vertical glues and ternary horizontal glues that satisfy (1). Moreover, analogous tile sets can be generated for any relatively prime p, q (not only $p = 2, q = 3$), further suggesting its naturalness as an object of study.

Computing Collatz trajectories with the Collatz tile set plus two more tiles. Together with the two extra tiles depicted in Figure 7(a2), the Collatz tile set is able to assemble Collatz trajectories starting from a straightforward north-east L-shaped seed as depicted in Figure 7(b). Input to the Collatz iterations are given in binary on the north-most glues (with LSB to the east). If the binary input x is of size n , we place n “S” on the vertical portion of the seed (to the east). The assembly process is directed (i.e. deterministic in these sense of which tile type is placed where) and, after it is finished, the n^{th} Collatz iterate of the binary input x , that is $T^n(x)$, will be written in ternary along the west-most glues of the assembly (ignoring “S” glues). In the example of Figure 7(b) we read $T^7(1001011_2) = 121_3$ meaning that, in base 10, $T^7(75) = 16$. This phenomenon can be proven using the results of [46], more precisely by identifying the Collatz tile set to the local rule of the CA-like system introduced in [46] and the two additional tiles to the non-local rule in Figure (1a)[right] of [46]). In practice, the two additional tiles are merely responsible for deleting trailing 0s in binary (which corresponds to the $/2$ part of the Collatz map) while the Collatz tile set does the heavier work of computing $3x + 1$ in binary while maintaining a correspondence between base 2 and base 3 encodings.

Predicting patterns produced by the Collatz tile set. The computational complexity of predicting what tile will be placed at a given position of the square area defined by the north-east L-shaped seed in Figure 7(b) is an open question [46]. However, if we restrict ourselves to using the Collatz tile set alone, without the two additional tiles, the prediction problem is in NL for each of the three L-shaped seeds: north-east, south-east and south-west (for any length $n \in \mathbb{N}$). That is because the relationship between pairs of tile sides, expressed



■ **Figure 7** The Collatz tile set and its relationship with the Collatz problem and Erdős’ conjecture. (a1) The Collatz tile set. (a2) Two additional tiles which allow to assemble Collatz trajectories from simple north-east L-shaped seeds. (b) Assembling the first 7 steps of the Collatz trajectory of $75 = 1001011_2$. The output, $T^7(75)$ can be read in base 3 on the west-most glues of the final assembly (ignoring “S” glues). Here, $T^7(1001011_2) = 121_3$ meaning, in base 10, $T^7(75) = 16$. (c) Constructing successive powers of 2 in base 3: the column marked with arrow number n encodes 2^n in base 3. Erdős’ conjecture states that, for $n > 8$, 2^n contains at least one 2 in base 3 [18].

8:20 Small Tile Sets That Compute While Solving Mazes

in (1), can be generalised to any rectangular assembly to give a simple arithmetical formula computable in nondeterministic logspace [46, 21] ($3^h N + E = 2^w W + S$, where now N, E, W, S denote binary/ternary numbers written in glue sequences along the respective North, East, West and South sides of a $w \times h$ rectangle). This fact means that, assuming a widely-believed conjecture in complexity theory (namely, $NL \neq P$), it is not possible to simulate arbitrary, polynomial size, Boolean circuits using the 6-tile Collatz tile set with those simple L-shaped connected seeds (within area polynomial in circuit size).

Although rectangular assemblies made with the Collatz tile set are simple to predict, they also relate to hard open questions in number theory. Notably to the following conjecture by Erdős [18]: For all $n > 8$, there is at least one digit 2 in the ternary representation of 2^n . Indeed, starting from the straightforward south-west L-shaped seed of Figure 7(c), consisting of m vertical 0s and a horizontal 1 followed by $m - 1$ horizontal 0s, an induction proves that consecutive columns of the assembly will encode successive powers of two in ternary. For instance, on the first 4 columns pointed by an arrow in Figure 7(c) we can successively read: “1”, “2”, “11”, “22” which are the ternary encodings of 1, 2, 4 and 8, the four first powers of 2. Erdős conjecture then becomes: any column to the east of the 10th column of the assembly (counting from the easternmost input column of vertical 0s), will contain a glue “2” (in red). This problem can be seen as a potentially simpler conjecture than the Collatz conjecture [25].

Predicting Minimum Free Energy Structures of Multi-Stranded Nucleic Acid Complexes Is APX-Hard

Anne Condon ✉ 

The University of British Columbia, Vancouver, Canada

Monir Hajiaghayi ✉

The University of British Columbia, Vancouver, Canada

Chris Thachuk ✉ 

The University of Washington, Seattle, WA, USA

Abstract

Given multiple nucleic acid strands, what is the minimum free energy (MFE) secondary structure that they can form? As interacting nucleic acid strands are the basis for DNA computing and molecular programming, e.g., in DNA self-assembly and DNA strand displacement systems, determining the MFE structure is an important step in the design and verification of these systems. Efficient dynamic programming algorithms are well known for predicting the MFE pseudoknot-free secondary structure of a single nucleic acid strand. In contrast, we prove that for a simple energy model, the problem of predicting the MFE pseudoknot-free secondary structure formed from multiple interacting nucleic acid strands is NP-hard and also APX-hard. The latter result implies that there does not exist a polynomial time approximation scheme for this problem, unless $P = NP$, and it suggests that heuristic methods should be investigated.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness; Applied computing → Chemistry

Keywords and phrases Nucleic Acid Secondary Structure Prediction, APX-Hardness, NP-Hardness

Digital Object Identifier 10.4230/LIPIcs.DNA.27.9

Funding *Anne Condon*: Supported by an NSERC Discovery Grant.

Monir Hajiaghayi: Supported by an NSERC Discovery Grant.

Chris Thachuk: Supported by a Banting Fellowship, ERC AdG VERIWARE, NSF-CCF-1213127, and NSF-CCF-2106695.

Acknowledgements We thank Erik Winfree for helpful discussions and proposing the problem and we also thank DNA 27 reviewers for their feedback.

1 Introduction

Computational methods are widely used to help understand the structure and function of DNA and RNA molecules. A central challenge has been reliable prediction of nucleic acid secondary structure. In both biological and molecular computing contexts, thermodynamic analyses are widely used for this purpose. Much work has focused on prediction of pseudoknot-free secondary structures, since such structures are common in both biological and designed systems and since pseudoknot-free structures are easier to handle algorithmically [12, 9, 15]. In this paper, we show that, while efficient thermodynamics-based approaches are well known for prediction of pseudoknot-free secondary structures of single strands, the problem of predicting pseudoknot-free secondary structures of multiple interacting strands is computationally intractable unless $P = NP$. Here and throughout, we consider a method to be efficient if its running time is bounded by a fixed polynomial in the total length of the strands.



© Anne Condon, Monir Hajiaghayi, and Chris Thachuk;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on DNA Computing and Molecular Programming (DNA 27).

Editors: Matthew R. Lakin and Petr Šulc; Article No. 9; pp. 9:1–9:21

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In what follows, we briefly summarize significant contributions on development of algorithms for predicting the pseudoknot free secondary structure of a single nucleic acid strand, or of multiple interacting strands. Table 1 also presents a summary of the time complexity of pseudoknot-free secondary structure and partition function prediction. When the input has multiple strands, we separate the cases where the number of strands is bounded by a fixed constant c , and when the number of strands is unbounded, i.e., can grow with the input size. Throughout this work, we focus on the latter case.

■ **Table 1** Computational complexity of predicting nucleic acid MFE pseudoknot-free secondary structures and partition functions, when the input is a single strand, multiple strands with a constant bound c on the number of strands, and multiple strands where the number of strands can grow with the input length n . In each case, n is the total number of bases in the input strand(s). We note that, for a single strand, a work by Bringmann et al. [3] presents an exact sub-cubic algorithm using a simple base pair model. The bold term shows our contribution and the question marks show that the complexity of the corresponding problems is as yet unresolved.

Input Type	MFE	Partition Function
Single Strand	$P[O(n^3)]$ [17, 18, 24]	$P[O(n^3)]$ [16]
Multiple Strands, Bounded ($\leq c$)	?	$P[O(n^3(c-1)!)]$ [4]
Multiple Strands, Unbounded	APX-hard [this work]	?

For single strands with length n , dynamic programming algorithms with $O(n^3)$ run time have long been used to efficiently predict minimum free energy (MFE) pseudoknot-free secondary structures, first for a simple “base pair” thermodynamic [17, 18, 24] model in which the free energy of a secondary structure is only dependent on the number of its base pairs, and later for more sophisticated energy models that account for entropic loop penalties, stacked pairs and other structural features. However, very recently, Bringmann et al., [3] proposed a truly sub-cubic algorithm to predict MFE secondary structures for a simple base pair energy model. Dynamic programming methods can also be used to efficiently calculate the partition function for a given strand, making it possible to compute the probability of base pair formation in equilibrium [16].

In addition to prediction of secondary structure of single strands, there has also been much interest in prediction of complexes that result when base pairs form between two or more strands. Such predictions can be used to understand the affinity of binding between a nucleic acid oligonucleotide and its potential target in biological processes such as RNA interference. Prediction of multi-stranded secondary structures is also important because methods for biomolecular programming and construction of nano-devices, such as self-assembly of complex DNA shapes and DNA strand displacement systems, are based on the formation of such complexes [20, 6]; prediction methods such as that provided by NUPACK [22, 5] can guide the design of such programs and devices.

An energy model for single-stranded secondary structure formation can be extended to obtain a model for multi-stranded complex formation by (i) charging an additional strand association penalty, typically a constant times the number of strands involved in the complex, and (ii) accounting for rotational symmetries [4]. Predicting MFE pseudoknot-free secondary structures formed from two (or any constant number) of strands with respect to a model that only accounts for strand association penalties is a straightforward extension of dynamic programming algorithms for single strands [23, 21, 2]. However, it is not clear how such algorithms can efficiently account for rotational symmetries that can arise when two or more indistinguishable strands interact [4]. Nevertheless, Dirks et al. [4] showed how to

efficiently calculate the partition function for a constant number of interacting molecules that form pseudoknot-free structures, by showing how rotational symmetry could be accounted for, while simultaneously addressing algorithmic overcounting issues that arise in partition function calculation. However, the partition function calculation method of Dirks et al. [4] requires a separate dynamic programming computation on all possible orderings of strands that interact to form a single complex. As a result, the method does not run in polynomial time when the number of participating strands grows with the overall input size (total length of strands). This situation can arise, for example, in DNA strand displacement systems. Also, surprisingly, while the partition function for a constant number of interacting strands can be calculated efficiently, it is not known how to efficiently calculate the MFE pseudoknot-free secondary structure of a constant number of interacting strands.

Thus, a basic open question is: can we efficiently compute the MFE pseudoknot-free secondary structure for a multi-set of DNA or RNA strands?

In this paper, we provide a negative answer to this question. Given a set of nucleic acid strands and a positive integer k , let MULTI-PKF-SSP be the problem of determining whether the strands can form a pseudoknot-free secondary structure with at least k base pairs. We show that MULTI-PKF-SSP is NP-hard, meaning that the existence of an efficient method for MFE pseudoknot-free secondary structure prediction of a multi-set of strands would imply all problems in the complexity class NP, which includes problems that are widely believed to be intractable, would have polynomial time algorithms. The hardness result holds whether or not rotational symmetries are accounted for in the energy model. Our proof uses a reduction from a variant of 3-dimensional matching (3DM), already known to be NP-hard, and employs code word designs with high pairwise edit distance [19].

In light of this NP-hardness result, another natural question is whether there is an efficient method to find a pseudoknot-free secondary structure whose energy is a close estimate of the energy of the MFE structure. We also provide a negative answer to this approximation question, by showing a limit to the accuracy of any such method, assuming that $\text{NP} \neq \text{P}$. Specifically, if there is a *polynomial time approximation scheme (PTAS)* that could find a pseudoknot-free secondary structure whose free energy closely approximates that of the MFE for any given multi-set of strands, then again $\text{NP} = \text{P}$. A PTAS is a polynomial time algorithm that receives as input an instance of an optimization problem and an arbitrary parameter $\epsilon > 0$, and returns an output whose value (in our case, the number of base pairs in the MFE structure) is within a factor $1 - \epsilon$ of the value of the optimal solution. The running time of a PTAS could be dependent on ϵ , but it must be polynomial in the input size for every fixed ϵ . Formally, we show that the optimization problem of finding the MFE structure for a multi-set of nucleic acid strands is hard for the complexity class APX, the class of NP optimization problems that have constant factor approximation algorithms. We show this result by establishing that our reduction from 3-dimensional matching to MFE structure prediction is an approximation-preserving reduction.

We note that hardness results have already been proved for variants of pseudoknotted secondary structure prediction. While dynamic programming can be used to predict MFE structures and partition functions for certain restricted classes of pseudoknotted structures, the general problem of predicting MFE pseudoknotted structures is NP-hard, even for a single strand [1, 14, 13]. The first two NP-hardness results, [1, 14] also use a simple energy model called *stacking* where only consecutive base pairs forming a stack contribute to the free energy of a strand. Hardness results can be valuable even with simple energy models; it would seem unlikely that the prediction problem becomes easier if the energy model is more sophisticated.

The rest of the paper is organized as follows. We provide preliminary definitions, problem statements and an overview of some useful theorems in Section 2. We outline the string properties and designs required for our reduction, in Section 3. We provide a polynomial-time reduction from a variant of 3DM to MULTI-PKF-SSP in Section 4, and prove its correctness in Section 5. In Section 6, we also infer that an optimization version of the problem is hard for the complexity class APX. This implies that there is no PTAS for approximating the optimal secondary structure of multi-stranded systems, unless $\text{NP} = \text{P}$. The proofs of some lemmas have been moved to Appendix A.

2 Preliminaries

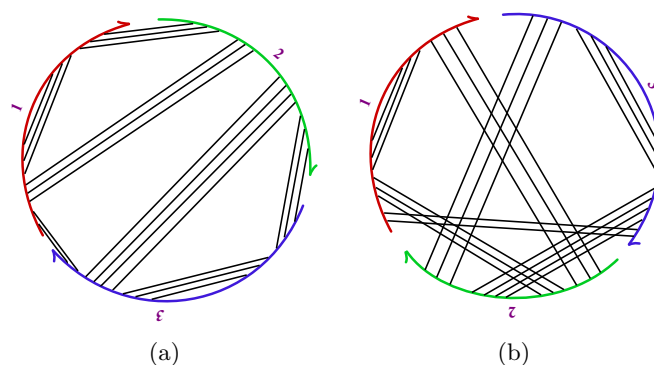
We review some basic terminology and prior work in order to precisely formulate the problem description and proof techniques.

A single DNA or RNA strand is a sequence of nucleotide bases, which we represent using the character set $\{\text{A}, \text{C}, \text{G}, \text{T}\}$ or $\{\text{A}, \text{C}, \text{G}, \text{U}\}$ respectively, with the left end of the sequence corresponding to the 5' end of the strand and the right end corresponding to the 3' end. Bonds can form between Watson-Crick base pairs, namely C-G and A-U for RNA and C-G and A-T for DNA [4].

We assume that consecutive bases within a sequence cannot pair with each other. This is consistent with actual structures, where there are typically at least three bases separating any two bases that are paired with each other. If sequences are numbered consecutively starting from 1, we can represent a base pair as a tuple (i, j) , such that $i < j - 1$, which specifies that the base at position i in the sequence is paired with the base at position j and j is not consecutive with i . A secondary structure is a set of base pairs such that no base is in two pairs. That is, if (i, j) and (i', j') are in the structure then i, j, i' and j' are all distinct.

Base pairing between two strands occurs in an antiparallel format. That is, the Watson-Crick complement of strand $x = 5'-x_1 \cdots x_n-3'$ is the strand $3'-\bar{x}_1 \cdots \bar{x}_n-5' \equiv 5'-\bar{x}_n \cdots \bar{x}_1-3'$, where (x_i, \bar{x}_i) is a Watson-Crick base pair. For example, the Watson-Crick complement of 5'-ACTCG-3' is 5'-CGAGT-3'. Throughout we will use the term *complement* to mean Watson-Crick complement and denote the complement of x by \bar{x} .

Similar to the single-stranded model, the secondary structure formed by m interacting strands is a set of Watson-Crick base pairs. To specify the secondary structure, we assign identifiers from 1 to m to the strands, and each base is named by a strand identifier and a position on the corresponding strand. For instance if base i in strand s pairs with base j in strand t , where $s \leq t$ and $i < j - 1$ if $s = t$, the base pair is denoted as (i_s, j_t) . A multi-stranded secondary structure can be represented as a polymer graph by ordering and depicting the directional (5' to 3') strands around the circumference of a circle, with edges along the circumference representing adjacent bases, and straight line edges connecting paired bases. Each such ordering of m strands is a circular permutation of the strands, and there are $(m - 1)!$ possible orderings. A secondary structure consists of one or more *complexes* that correspond to the connected components in the polymer graph representation. If the polymer graph of any one of these possible orderings has no crossing lines, then the secondary structure is called *pseudoknot-free* [4]. For example, Figure 1 shows the two possible circular permutations for three strands 1, 2, and 3, and the connected polymer graphs for the same secondary structure. Since Figure 1(a) has no crossing lines, the structure is pseudoknot-free.



■ **Figure 1** a) A polymer graph representation of a pseudoknot-free secondary structure for the strand set $\{1, 2, 3\}$ with ordering 123. b) A second polymer graph for the same structure, with strand ordering 132.

2.1 The simple energy model

Here, we employ a very simple extension of the “base pair free energy” model for secondary structures [18]. In that model, the score of each base pair is -1 and the overall score (free energy) of a single-stranded secondary structure is its total number of base pairs. So, the more base pairs in a secondary structure of a single strand, the lower its score.

Where there are multiple interacting strands, there is an entropic penalty for strands to associate via base pairing, i.e., a penalty for reducing the number of complexes [4]. In our simplified model, we define the strand association penalty to be $K_{\text{assoc}} \geq 0$. Thus, for a pseudoknot-free secondary structure S consisting of m strands, l ($\leq m$) complexes, and p base pairs, the overall score, or free energy, of S is

$$\mathbf{E}(S) = p(-1) + (m - l)K_{\text{assoc}}.$$

For example, the secondary structure in Figure 1(a) has score $21(-1) + (3 - 1)K_{\text{assoc}} = -21 + 2K_{\text{assoc}}$. For strands s_1, \dots, s_m , an *optimal* pseudoknot-free secondary structure S_{opt} satisfies $\mathbf{E}(S_{\text{opt}}) \leq \mathbf{E}(S)$ for any pseudoknot-free secondary structure S of s_1, \dots, s_m .

Since there can be a tradeoff between the number of base pairs and the number of complexes, then it is possible under this model for an optimal pseudoknot-free secondary structure to have less than the maximum number of possible base pairs. However, our proofs have been constructed so that pseudoknot-free MFE secondary structures will have a maximum number of base pairs for any reasonable value of the constant K_{assoc} . We will proceed with our problem definitions under the assumption that $K_{\text{assoc}} = 0$ and formally argue later that the results hold for all constants $K_{\text{assoc}} \geq 0$.

2.2 Problem definitions

We now formally define the main problem of interest in this paper as a decision problem.

► **Problem 1** (MULTI-PKF-SSP).

Instance: m nucleic acid strands and a positive integer k .

Question: Do the m strands form a pseudoknot-free secondary structure containing at least k base pairs?



■ **Figure 2** An instance of the restricted 3-dimensional matching problem, $3DM(3)$, where $X = \{x_1, x_2, x_3\}$, $Y = \{y_1, y_2, y_3\}$, $Z = \{z_1, z_2, z_3\}$. (a) The set of permitted triples, $\mathcal{T} = \{(x_1, y_2, z_2), (x_2, y_1, z_1), (x_2, y_3, z_2), (x_3, y_3, z_3)\}$. (b) A valid matching $\mathcal{M} \subseteq \mathcal{T}$.

We will describe a polynomial-time reduction from a restriction of the NP-hard 3-dimensional matching problem to MULTI-PKF-SSP. A 3-dimensional matching is defined as follows. Let X , Y , and Z be finite, disjoint sets, and let \mathcal{T} be a subset of $X \times Y \times Z$. That is, \mathcal{T} consists of triples (x, y, z) such that $x \in X$, $y \in Y$, and $z \in Z$. Now $\mathcal{M} \subseteq \mathcal{T}$ is a *3-dimensional matching* if the following holds: for any two distinct triples $(x_i, y_j, z_k) \in \mathcal{M}$ and $(x_a, y_b, z_c) \in \mathcal{M}$, we have $x_i \neq x_a$, $y_j \neq y_b$, and $z_k \neq z_c$.

For convenience in our construction, we use a restriction of the 3-dimensional matching problem, called $3DM(3)$, that requires each element to appear in at most three triples of \mathcal{T} .

► **Problem 2** ($3DM(3)$).

Instance: A set $\mathcal{T} \subseteq X \times Y \times Z$, where $|X| = |Y| = |Z| = n$ and each element of X , Y and Z appears in at most 3 triples of \mathcal{T} .

Question: Does there exist a matching $M \subseteq \mathcal{T}$, with $|M| = n$?

► **Theorem 1** (Garey & Johnson (1979) [7]). *$3DM(3)$ is NP-complete.*

Next we define an optimization version of the MULTI-PKF-SSP decision problem:

► **Problem 3** (MAX-MULTI-PKF-SSP).

Instance: A set of m nucleic acid strands.

Optimization Problem: Determine a pseudoknot-free secondary structure of the m strands with maximum number of base pairs.

An optimization problem is in APX if it has a constant factor approximation algorithm, i.e., an efficient method that can determine a solution whose score is within some fixed multiplicative factor of that of an optimal solution. A problem is APX-hard if for some constant c , a c -approximation algorithm for the problem would imply that $NP = P$. One way to prove a problem is APX-hard is to show an approximation-preserving reduction from a known APX-hard problem. A problem is APX-complete if it is APX-hard and is in APX. We derive our hardness result for the MAX-MULTI-PKF-SSP problem by a reduction from the MAX- $3DM(3)$ problem, an optimization version of the $3DM(3)$ problem:

► **Problem 4** ($\text{MAX-3DM}(3)$).

Instance: A set $\mathcal{T} \subseteq X \times Y \times Z$, where $|X| = |Y| = |Z| = n$ and each element of X , Y and Z appears in at most 3 triples of \mathcal{T} .

Optimization Problem: Find a maximum size 3-dimensional matching $M \subseteq \mathcal{T}$.

Kann [11] showed that $\text{MAX-3DM}(3)$ is MaxSNP-complete and thus APX-complete. Hardness of approximation was established by demonstrating that it is NP-hard to decide whether an arbitrary instance of the problem has a matching of size n or a matching of size at most $(1 - \epsilon_0)n$, for some $\epsilon_0 > 0$.

► **Theorem 2** (Kann (1994) [11]). *MAX-3DM(3) is APX-complete.*

3 String designs and their properties

In this section we show how to design strings with properties that are useful in our reduction. We follow standard string notation: for a string $a = a_1 \dots a_n$ we denote its i^{th} character (or symbol) by a_i and its length by $|a| = n$; for any symbol B , we let B^l denote a string of length l consisting of only B 's. The following related string properties are of particular interest to us.

1. A *pairwise sequence alignment*, or simply *alignment*, of strings a and b is a pair of strings (a', b') with $|a'| = |b'|$, where a' and b' are obtained from a and b respectively by the insertion of zero or more copies of a special *gap* symbol. Moreover, for any i , a'_i and b'_i are not both gap symbols and if neither a'_i nor b'_i is the gap symbol then $a'_i = b'_i$. The alignment can alternatively be considered as a sequence of aligned pairs $(a'_i, b'_i), 1 \leq i \leq |a'|$. A pair is a *gap pair* if either a'_i or b'_i is a gap symbol. We also define an *optimal alignment* of a and b as a pairwise alignment of a and b with a minimum number of gap pairs, amongst all possible alignments.
2. A *longest common subsequence* between strings a and b is a longest subsequence common to the two strings. Note that a subsequence of a string results from the deletion of zero or more of its characters. We denote the length of such a subsequence by $\text{LCS}(a, b)$. A longest common subsequence corresponds to an optimal alignment of a and b and $\text{LCS}(a, b)$ is equal to the total number of gap-free pairs of symbols in the alignment.
3. The *insertion-deletion distance* $\mathbf{d}_{\text{LCS}}(a, b)$ between strings a and b is the minimum number of insertions and deletions of symbols needed to convert a into b (or equivalently to convert b to a). Equivalently, the insertion-deletion distance between a and b is equal to the number of gap pairs in an optimal alignment of a and b .

The insertion-deletion distance and length of the longest common subsequence of two strings are related by the following known result.

► **Theorem 3** ([8]). *Given two strings a and b , where $|a| = n$ and $|b| = n'$, then $\mathbf{d}_{\text{LCS}}(a, b) = k$ if and only if $\text{LCS}(a, b) = \frac{(n+n'-k)}{2}$.*

Note that if a and b are equi-length strings, then k is an even number.

In the next theorem, we show how to efficiently construct a “large” set of relatively short, equi-length strings that have high pairwise insertion-deletion distance. The construction employs a greedy codeword design used also in Justesen [10] and Schulman and Zuckerman [19].

► **Theorem 4.** *Let $w > 0$ and $\delta > 0$. For any n , a set of at least wn equi-length strands over the alphabet $\{A, T\}$, each of length $k \log n$ for some constant k (that depends on w and δ), can be designed in $2^{O(\log n)}$ time, such that the insertion-deletion distance between any pair in the set is at least $\delta \log_2 n$. Moreover, all strands in the set have at least $\lceil \delta \log_2 n/2 \rceil$ A's and at least $\lceil \delta \log_2 n/2 \rceil$ T's.*

Proof. We construct the desired set using a greedy algorithm that is specified in terms of a quantity $t = \Theta(\log_2 n)$ that we determine in the penultimate paragraph of this proof. From $\{A, T\}^t$, first put the two strings A^t and T^t in the set. Once $i \geq 2$ strings are in the set, choose any string from $\{A, T\}^t$ whose insertion-deletion distance from all i strings already in the set is at least $\delta \log_2 n$, and add it to the set. Continue until no more strings can be chosen with the desired insertion-deletion distance. Finally, remove the strings A^t and T^t . This algorithm runs in time $2^{O(\log n)}$. The number of strings in $\{A, T\}^t$ that have insertion-deletion distance at most $2d$ from a given string s in $\{A, T\}^t$ is at most $\binom{t}{d} 2^d$ (see proof of Lemma 2 of Schulman and Zukerman [19]). If $d = \lceil \delta \log_2 n/2 \rceil$, then our set has the desired property that the insertion-deletion distance between any pair in the set is at least $\delta \log_2 n$. Furthermore all strings in the set, once A^t and T^t are removed, must have at least $\lceil \delta \log_2 n/2 \rceil$ A's and at least $\lceil \delta \log_2 n/2 \rceil$ T's; otherwise, their insertion-deletion distance from A^t and T^t , would be less than $\delta \log_2 n$.

The number of strings in the set before removal of A^t and T^t is at least $wn + 2$ if we choose t so that

$$2^t / \binom{t}{d} 2^d \geq 2^{t/2} \geq wn + 2.$$

These inequalities hold if t is a sufficiently large constant times $\log_2 n$. For the first inequality, from Stirling's formula we have that $\binom{t}{d} < (te/d)^d$, and so the inequality holds if $2d \log_2(te/d) + d \leq t/2$. This in turn holds if $t = \eta d$ ($= \eta \lceil \delta \log_2 n/2 \rceil$) where we choose constant η so that $\eta e \leq 2^{n/4-1/2}$. For the second inequality, we simply need that $t \geq 2 + 2 \log_2 w + 2 \log_2 n$.

Finally, since the strings A^t and T^t are removed and all other strings have insertion-deletion distance at least $\delta \log_2 n$ from strings A^t and T^t , all strands in the set have at least $\delta \log_2 n$ A's and at least $\delta \log_2 n$ T's. ◀

Our design also makes use of a *padding* function. Let ρ^i denote the padding function that, applied to a string, inserts i A's (called padded A's) at the start of, and between every pair of symbols in, the string.

► **Definition 5** (padding function ρ^i). *Let $a = a_1 a_2 \dots a_n$ be a string. Then $\rho^i(a) = A^i a_1 A^i a_2 \dots A^i a_n$.*

If $\mathbf{d}_{\text{LCS}}(a, b) = k$ then $\mathbf{d}_{\text{LCS}}(\rho^i(a), \rho^i(b))$ may be less than k . To illustrate why, first consider the function ρ^1 , defined as $\rho^1(a_1 a_2 \dots a_n) = A^1 a_1 A^1 a_2 \dots A^1 a_n$. If we choose $a = \text{AATATT}$, and $b = \text{TTATAA}$, then $\mathbf{d}_{\text{LCS}}(a, b) = 6$ whereas $\mathbf{d}_{\text{LCS}}(\rho^1(a), \rho^1(b)) = 4$, as shown in Figure 3. This appears to contradict the assertion in Lemma 2 of Schulman and Zukerman [19] that $\mathbf{d}_{\text{LCS}}(\rho^1(a), \rho^1(b)) \geq \mathbf{d}_{\text{LCS}}(a, b)$. Adapting this example, if

$$a' = A^5 A^5 T^5 A^5 T^5 T^5 \text{ and } b' = T^5 T^5 A^5 T^5 A^5 A^5,$$

then $\mathbf{d}_{\text{LCS}}(a', b') = 30$, while $\mathbf{d}_{\text{LCS}}(\rho^5(a'), \rho^5(b')) = 20$.

We next show a lower bound on $\mathbf{d}_{\text{LCS}}(\rho^i(a), \rho^i(b))$ in terms of $\mathbf{d}_{\text{LCS}}(a, b)$.

<pre style="margin: 0;"> A ATATT TTATA A </pre>	<pre style="margin: 0;"> A A AAATAAATAT ATATAAATAAA A </pre>
(a)	(b)

■ **Figure 3** Padding can reduce insertion-deletion distance. (a) The ATA substrings of the two strings of length 6 forms a LCS, leaving a total of six symbols unmatched. (b) When the strings are 1-padded, the leftmost A of the first string and the rightmost A of the second string, plus the padded A's, become part of the LCS.

► **Lemma 6.** *Let a and b be equi-length strings over $\{A, T\}$. Then*

$$\mathbf{d}_{\text{LCS}}(\rho^i(a), \rho^i(b)) \geq \mathbf{d}_{\text{LCS}}(a, b)/2.$$

Let a and b be strands and let $S(a)$ and $S(a, b)$ be secondary structures for strand a and pair (a, b) respectively. The base pairs of (a, b) may be inter-molecular and/or intra-molecular. We define the *unpairedness* of $S(a)$ or $S(a, b)$ to be the number of bases that are not paired in $S(a)$ or $S(a, b)$, respectively. The next lemma provides lower bounds on the unpairedness of structures formed from padded strings.

► **Lemma 7.** *Let a' and b' be any strands over the alphabet $\{A, T\}$, let $a = \rho^5(a')$, let $b = \rho^5(b')$, and let s be any substrand of a or \bar{a} . Let $S(s)$, $S(a, b)$, $S(\bar{a}, \bar{b})$ and $S(a, \bar{b})$ be any pseudoknot-free secondary structures for s , (a, b) , (\bar{a}, \bar{b}) and (a, \bar{b}) , respectively. Then*

1. *The unpairedness of $S(s)$ is at least $\frac{1}{3}|s|$.*
2. *The unpairedness of $S(a, b)$ is at least $\frac{2}{3}(|a| + |b|)$.*
3. *The unpairedness of $S(\bar{a}, \bar{b})$ is at least $\frac{2}{3}(|\bar{a}| + |\bar{b}|)$.*
4. *The unpairedness of $S(a, \bar{b})$ is at least $\frac{1}{3}\mathbf{d}_{\text{LCS}}(a, b)$.*

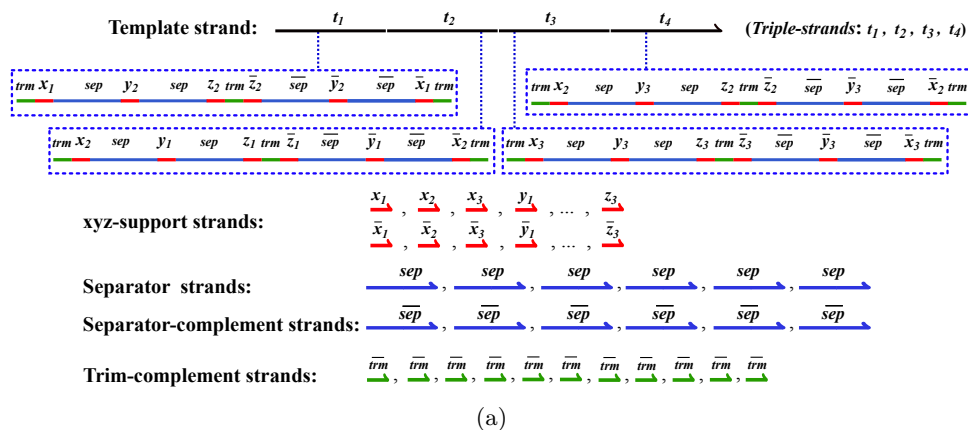
► **Definition 8.** *A set \mathcal{S} of strands is k -robust if the following properties hold:*

1. *All strands of \mathcal{S} have the same length.*
2. *All strands of \mathcal{S} have at least k A's and at least k T's.*
3. *For any a and b in the set, the unpairedness of optimal structures for a , \bar{a} , (a, b) , (\bar{a}, \bar{b}) , and (a, \bar{b}) is at least k .*

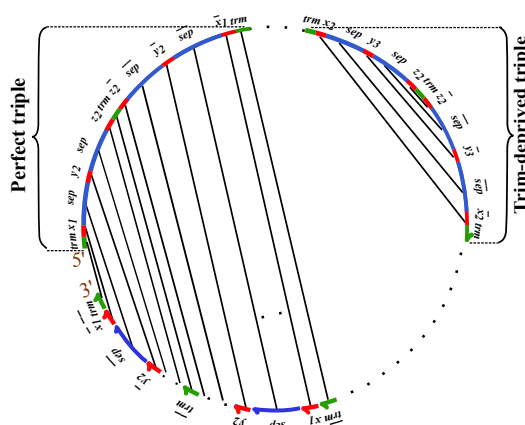
► **Theorem 9.** *Let $w > 0$. For any n , a $\log_2 n$ -robust set of at least wn strands, each of length $p \log_2 n$ for some constant p , can be designed in $2^{O(\log n)}$ time.*

Proof. Using Theorem 4, for any $w > 0$ and $\delta = 6$ we can obtain, in time $2^{O(\log n)}$, a set \mathcal{S}' of at least wn strands, each of length $k \log_2 n$ for some constant k , such that the insertion-deletion distance between any pair of strands in \mathcal{S}' is at least $6 \log_2 n$. Moreover, all strands in \mathcal{S}' have at least $3 \log_2 n$ A's and at least $3 \log_2 n$ T's. This latter property implies that the strands in \mathcal{S}' have length at least $6 \log_2 n$.

Apply the padding function ρ^5 to strands in \mathcal{S}' to obtain a new set \mathcal{S} . The strands in \mathcal{S} have length $6k \log_2 n$, which must be at least $36 \log_2 n$. Lemma 6 shows that the insertion-deletion distance between any pair of strands in \mathcal{S} is at least $\delta \log_2 n / 2 = 3 \log_2 n$. Lemma 7 then shows that if a and b are any two strands in the set \mathcal{S} , the unpairedness of the optimal structure of a , or its complement, or of (a, b) , (a, \bar{b}) or (\bar{a}, \bar{b}) , is at least $\min\{\frac{1}{3}|a|, \frac{2}{3}(|a| + |b|), \frac{1}{3}\mathbf{d}_{\text{LCS}}(a, b)\}$. Given that $|a|$ and $|b|$ are at least $36 \log_2 n$ and that $\mathbf{d}_{\text{LCS}}(a, b) = 3 \log_2 n$, this lower bound is at least $\log_2 n$. Therefore, the unpairedness of the set \mathcal{S} is at least $\log_2 n$, as desired. ◀



(a)



(b)

■ **Figure 4** Example of the reduction from the 3DM(3) instance of Figure 2. (a) Strands of the resulting MULTI-PKF-SSP instance, specified at the domain level. (b) Partial MFE structure of the strands. Here, the structure involving triple-strand t_1 , labeled as *perfect triple*, indicates that the triple (x_1, y_2, z_2) is in the solution of the 3DM(3) instance. Triple-strand t_4 is a *trim-deprived triple* since there are no bonds to bases in the middle trim domain. This structure indicates that triple (x_2, y_3, z_2) isn't selected in the solution.

4 The reduction

We show a polynomial time reduction from $3DM(3)$ to MULTI-PKF-SSP. Given an instance $I = (X, Y, Z, \mathcal{T})$ of $3DM(3)$, where $m = |\mathcal{T}|$ and $n = |X| = |Y| = |Z|$, we construct an instance I' of MULTI-PKF-SSP as follows.

Domains used in strands of I'

The strands of I' contain the following domains.

- One domain for each $x \in X$, $y \in Y$, and $z \in Z$ and one domain for each complement. Where no confusion arises, we use $x, \bar{x}, y, \bar{y}, z,$ and \bar{z} to refer to these domains.
- A *separator* and a *separator-complement* domain, denoted by Sep and $\overline{\text{Sep}}$.
- A *trim* domain and a *trim-complement* domain, denoted by Trm and $\overline{\text{Trm}}$ respectively.

Strands of I'

Instance I' consists of the following strands, where each strand is a sequence of domains.

- *Template strand*: One strand that is the concatenation of *triples*. There is one triple for each $(x, y, z) \in \mathcal{T}$, which is the following concatenation of domains:

$$\text{Trm } x \text{ Sep } y \text{ Sep } z \text{ Trm } \bar{z} \overline{\text{Sep}} \bar{y} \overline{\text{Sep}} \bar{x} \text{ Trm}$$

We call the substrands $x \text{ Sep } y \text{ Sep } z$ and $\bar{z} \overline{\text{Sep}} \bar{y} \overline{\text{Sep}} \bar{x}$ of a triple the 5' and 3' *flanks*, respectively. We call the Trm domains at the ends of the triple the end-trims and the middle Trm domain the center-trim.

- *Separator (-complement) support strands*: $2n \text{ Sep}$ strands and another $2n \overline{\text{Sep}}$ strands.
- *xyz-support strands*: For each x, y and z domain, one strand consisting of just that domain and one for its complement, for a total of $6n$ strands.
- *Trim-complement strands*: $2m + n$ copies of $\overline{\text{Trm}}$, which is the complement of the trim domain Trm .

We refer to the xyz-support strands and the separator and separator-complement support strands collectively as the support strands.

► **Lemma 10.** *The total number of support strands is $10n$.*

Proof. This follows immediately from the fact that there are $6n$ xyz-supports and $4n$ separator and separator-complement strands in total. ◀

This completes the description of the reduction at the domain level of detail. Figure 4 (a) shows the resulting MULTI-PKF-SSP instance, specified at the domain level, after a reduction from the $3DM(3)$ instance of Figure 2.

The MFE structure of the resulting set of strands is partially depicted in Figure 4 (b). All domains of the substrand labeled as “perfect triple” are bound to their complements, indicating that the triple (x_1, y_2, z_2) is selected in the solution of the $3DM(3)$ instance, consistent with the solution shown in Figure 2 (b). The other triple that is depicted is a “trim-deprived triple”. This is a triple in which at least one trim domain is unbound. The corresponding triple (x_2, y_3, z_2) does not appear in the solution from Figure 2 (b). Intuitively, there is a trim-complement strand available to bind with each of the $2m$ end-trim domains at the ends of all triples, and in addition the number of xyz-support, separator supports and additional trim-complement strands is necessary and sufficient to have n “perfect triples” in an optimal secondary structure when the $3DM(3)$ instance has a perfect matching of size n .

Sequence design for I'

To complete the reduction, we specify a sequence design for each domain of I' . For the x , y , and z domains, we use the set of sequences of Theorem 9 with $w = 3$, since we need $3n$ domains (plus their complements) in total. Let $E (= \Theta(\log_2 n))$ be the length of these domains. The trim domain $\text{Trm} = \mathbf{G}^E$, i.e., consisting of E copies of \mathbf{G} , and $\overline{\text{Trm}} = \mathbf{C}^E$. The Sep domain is \mathbf{A}^{6E} , and the $\overline{\text{Sep}}$ domain is the complement of the Sep domain, namely \mathbf{T}^{6E} .

The sequence design has the property that there are an equal number of \mathbf{A} and \mathbf{T} bases overall, since for every x , y , z or separator domain there is a corresponding complementary domain. The total number of \mathbf{C} 's in trim-complement strands is $(2m + n)E$. The total number of \mathbf{G} 's in end-trims and center-trims is $3mE$. Since $m \geq n$, the total number of \mathbf{G} 's is at least as great as the total number of \mathbf{C} 's. Therefore, under the assumption that only Watson-Crick base pairs can form, the maximum number of base pairs is limited to the total number of \mathbf{A} (or \mathbf{T}) bases plus the total number of \mathbf{C} bases. Let P denote this quantity.

The instance I' is comprised of the strands of I' and the positive integer P .

► **Lemma 11.** *Instance I' can be constructed in time polynomial in n .*

5 Reduction correctness

We show that if the given instance I of $3\text{DM}(3)$ has a perfect matching then the optimal secondary structure formed from strands in I' is a single complex that has the maximum possible number P of base pairs. We also show that if the optimal matching of I has size $n - i$ then the optimal structure has only $P - \Omega(iE)$ base pairs.

► **Lemma 12.** *If I has a perfect matching, then the strands of I' can form a pseudoknot-free secondary structure, consisting of a single complex and P base pairs, with n perfect triples.*

Proof. Here, in the reduced instance I' , bases in the n triples corresponding to the perfect matching can be bound to the corresponding support strands, to form n perfect triples. The end-trims of the remaining triples can also be bound to two trim-complement strands, while their complementary $5'$ and $3'$ flanks are paired together to make trim-deprived triples. Therefore, as all \mathbf{A} 's and \mathbf{C} 's are paired in this single (connected) complex, the number of base pairs is P . ◀

We next consider the case that the optimal matching of I has size at most $n - i$. Let $\text{Opt}(I')$ denote an optimal pseudoknot-free structure of the reduced instance I' . We establish properties that must hold true of $\text{Opt}(I')$ and conclude that when the optimal matching of I has size at most $n - i$, then $\text{Opt}(I')$ has $P - \Omega(iE)$ base pairs.

With respect to a given structure, we say that a domain is *bound* if at least one of its bases forms a base pair. A domain d in a triple (as part of the template strand) is *connected* to a non-template strand s if there is a sequence of non-template strands s_1, s_2, \dots, s_j where $s_j = s$, such that d forms a base pair with s_1 , s_1 forms a base pair with s_2 , and so on up to s_{j-1} forming a base pair with $s_j = s$.

We partition the triples into four types, depending on the structure they form in $\text{Opt}(I')$.

- *Perfect triples:* The triple binds to the set of non-template strands that are complementary to the triple domains. (This set of non-template strands contains two Sep 's, two $\overline{\text{Sep}}$'s, three $\overline{\text{Trm}}$'s and six xyz-support strands in total.) The set of perfect triples corresponds to a matching of instance I .
- *Trim-deprived triples:* At least one trim of a triple is unbound.

- *Hogger triples*: These are triples which are not trim-deprived, and moreover, the ten domains in the flanks of a hogger triple are bound to, or connected to, at least eleven support strands in total.
- *Flawed triples*: None of the above. In particular, flawed triples are not trim-deprived.

Since neither hogger nor flawed triples are trim-deprived, the support domains that are bound to or connected to either their 5' or 3' flanks cannot bind to other domains on the template strand, or a pseudoknot would form.

► **Lemma 13.** *The total number of trim-deprived and flawed triples in $\text{Opt}(I')$ is at least $(m - n) + i/11$.*

► **Lemma 14.** *Either $\text{Opt}(I')$ has at least $m - n + i/22$ trim-deprived triples, or at least $i/22$ flawed triples.*

Proof. By Lemma 13, the total number of trim-deprived and flawed triples is at least $(m - n) + i/11$. So if the number of trim-deprived triples is less than $m - n + i/22$, then the number of flawed triples must be at least $i/22$. ◀

We now adapt our notion of unpairedness from Section 3 to ACT-unpairedness. Let a and b be strands and let $S(a)$ and $S(a, b)$ be secondary structures for strand a and pair (a, b) respectively. The ACT-unpairedness of $S(a)$ or $S(a, b)$ is the number of A, C and T bases that are not paired in $S(a)$ or $S(a, b)$, respectively.

► **Lemma 15.** *If the number of trim-deprived triples in $\text{Opt}(I')$ is at least $m - n + i/22$, then at least $iE/22$ C's are unpaired in $\text{Opt}(I')$, and so $\text{Opt}(I')$ has ACT-unpairedness $\Omega(iE)$.*

In order to show that many flawed triples cause $\text{Opt}(I')$ to have high ACT-unpairedness, we first derive some useful properties about flawed triples. In what follows, we let $F_{5'} = x \text{Sep}_{xy} y \text{Sep}_{yz} z$ and $F_{3'} = \bar{z} \overline{\text{Sep}}_{yz} \bar{y} \overline{\text{Sep}}_{xy} \bar{x}$ denote the sequences on the 5' and 3' flanks of a given flawed triple. Let $\mathcal{S}_{5'}$ and $\mathcal{S}_{3'}$ be the sets of support strands that are bound to, or connected to, domains of $F_{5'}$ and $F_{3'}$ respectively, in the structure $\text{Opt}(I')$. The sets $F_{5'}$ and $F_{3'}$ are disjoint, since something is bound to the middle trim in $\text{Opt}(I')$, and the structure has no pseudoknots. Since a flawed triple has at most ten support strands bound to it, either $|\mathcal{S}_{5'}| \leq 5$ or $|\mathcal{S}_{3'}| \leq 5$. In the following lemmas, for concreteness, we suppose that $|\mathcal{S}_{5'}| \leq 5$; the argument when $|\mathcal{S}_{3'}| \leq 5$ is obtained by replacing domains and strands with their complements and bases A and T with each other. Let $\text{Opt}(F_{5'})$ be the substructure of $\text{Opt}(I')$ formed by the bases in $F_{5'}$ and the strands in $\mathcal{S}_{5'}$.

► **Lemma 16.** *Suppose that there are $l \geq 2$ bonds between one of the x , y or z domains of $F_{5'}$ and either Sep_{xy} or Sep_{yz} . Then $\text{Opt}(F_{5'})$ has ACT-unpairedness at least $5(l - 1)$.*

► **Lemma 17.** *Suppose that in $\text{Opt}(F_{5'})$, $|\mathcal{S}_{5'}| \leq 5$ and the ACT-unpairedness of $F_{5'}$ is less than $(\log_2 n)/3$. Then the following must hold.*

1. Each Sep domain of $F_{5'}$ is bound to a $\overline{\text{Sep}}$ -support domain.
2. Each x , y and z domain of $F_{5'}$ is bound to an xyz -support domain.

As a consequence, each x , y , and z domain of $F_{5'}$ is bound to a distinct xyz -support of $\mathcal{S}_{5'}$, each Sep domain of $F_{5'}$ is bound to a distinct $\overline{\text{Sep}}$ support of $\mathcal{S}_{5'}$, and $\mathcal{S}_{5'}$ contains exactly three xyz -supports and two $\overline{\text{Sep}}$ supports.

► **Lemma 18.** *Let $F_{5'} = x \text{Sep}_{xy} y \text{Sep}_{yz} z$ be the left flank of a flawed triple. Suppose that in $\text{Opt}(F_{5'})$, $|\mathcal{S}_{5'}| \leq 5$. Then for any constant $\alpha < 1/7$, the ACT-unpairedness of $\text{Opt}(F_{5'})$ is at least $\alpha \log_2 n$.*

► **Lemma 19.** *If the optimal matching of I has size at most $n - i$, then $\text{Opt}(I')$ has $P - \Omega(iE)$ base pairs.*

Proof. By Lemma 14, $\text{Opt}(I')$ either has at least $m - n + i/22$ trim-deprived triples, or at least $i/22$ flawed triples.

First suppose that $\text{Opt}(I')$ has at least $m - n + i/22$ trim-deprived triples. Then by Lemma 15, $\text{Opt}(I')$ has ACT-unpairedness $\Omega(iE)$. Similarly, if $\text{Opt}(I')$ has at least $i/22$ flawed triples, then by Lemma 18, each flawed triple has ACT-unpairedness $\Omega(\log n) = \Omega(E)$, since $E = \Theta(\log n)$. Again, the total ACT-unpairedness is $\Omega(iE)$.

Recall that all A's, C's and T's must be paired in order for the total number of base pairs to be P . Since the total ACT-unpairedness is $\Omega(iE)$, it must be that the number of base pairs in $\text{Opt}(I')$ is at most $P - \Omega(iE)$. ◀

► **Theorem 20.** *MULTI-PKF-SSP is NP-complete.*

Proof. Let I be any instance of MULTI-PKF-SSP, i.e. m nucleic acid strands and a positive integer k . Given a certificate for I , which includes a secondary structure S and an ordering of the m strands, we can check in time polynomial in the total length of the strands whether S is a valid, pseudoknot-free secondary structure and whether it has k base pairs. Therefore, MULTI-PKF-SSP is in NP.

Moreover, in the last section we provided a polynomial time reduction from any instance I of 3DM(3) to an instance I' of MULTI-PKF-SSP. The optimal structure $\text{Opt}(I')$ has P base pairs if I has a perfect matching, by Lemma 12, and $\text{Opt}(I')$ has less than P base pairs if I does not have a perfect matching (by Lemma 19), where P is the total number of A, T and C bases of the strands of instance I' .

Putting these together, we conclude that MULTI-PKF-SSP is NP-complete. ◀

Until now, we have only considered the number of base pairs in the MFE structure under the assumption that there is no penalty for strand association, i.e., $K_{\text{assoc}} = 0$. Our construction has the property that structure $\text{Opt}(I')$ is a single complex when I has a perfect matching. When $K_{\text{assoc}} > 0$ the penalty to bring the $2m + 11n + 1$ strands into a single complex is $(2m + 11n)K_{\text{assoc}}$. However, the number of base pairs formed between complementary domains of distinct strands is at least E , where $E = \Theta(\log n)$. Thus, for any positive constant K_{assoc} the value of E can be scaled by a constant to ensure that a single domain binding is always favourable, even when decreasing the number of complexes by one.

6 Approximability

We proved that the MULTI-PKF-SSP problem is NP-complete in Theorem 20. Given this result, it is natural to investigate whether there is a polynomial time algorithm to approximate the optimal secondary structure of multi-stranded systems. We show in Theorem 22 that the MAX-MULTI-PKF-SSP problem is in fact APX-hard. This result asserts that there is no polynomial time approximation scheme (PTAS) for this problem, unless $P = NP$.

We first note in Lemma 21 that our reduction of Section 4 is approximation-preserving, transforming one optimization problem into another one. We then prove that this construction also maps a solution of MAX-MULTI-PKF-SSP to a solution of MAX-3DM(3).

► **Lemma 21.** *Our reduction from an instance I of MAX-3DM(3) to an instance I' of MAX-MULTI-PKF-SSP has the following properties:*

- *If I has a matching of size n then $|\text{Opt}(I')| = P$.*
- *If I has a matching of size at most $(1 - \epsilon_0)n$ then $|\text{Opt}(I')| \leq P - \alpha \epsilon_0 n E$, for some constant $\alpha > 0$.*

Proof. This lemma directly follows from Lemmas 12 and 19. ◀

► **Theorem 22.** *MAX-MULTI-PKF-SSP is APX-hard.*

Proof. Let I' be an instance of MAX-MULTI-PKF-SSP obtained from an instance I of MAX-3DM(3) where the size of the three sets is n and there are m total triples. First, we review the quantities E and P of the reduction of Section 4. Recall that $E = \Theta(\log n)$ specifies the lengths of xyz-support domains in instance I' obtained from instance I . Our sequence design and Lemma 11 ensure that instance I' has $\Theta(n) + \Theta(m)$ domains of length $\Theta(E)$. Recall that P is the total number of base pairs in an optimal structure for I' if I has a perfect matching. A perfect matching for I would have n triples. It follows that $P = \Theta(n \log_2 n)$.

We now apply Lemma 21 to show APX-hardness of MAX-MULTI-PKF-SSP. Suppose to the contrary that for some $\epsilon > 0$, there is a $(1 - \epsilon)$ -approximation algorithm for this problem. Then, the following hold:

- If I of MAX-3DM(3) has a matching of size n , then on instance I' the algorithm returns a solution with value at least $(1 - \epsilon)|\text{Opt}(I')| = (1 - \epsilon)P$.
- If instance I has a matching of size at most $(1 - \epsilon_0)n$, then on instance I' the algorithm returns a solution with value at most $|\text{Opt}(I')| \leq P - \alpha\epsilon_0 nE$.

Therefore, if $P - \alpha\epsilon_0 nE < (1 - \epsilon)P$ the algorithm can distinguish between the cases where I has a matching of size n or of size at most $(1 - \epsilon_0)n$. By our current assumptions about P and E , the above inequality holds if $\epsilon < \alpha\epsilon_0 nE/P$. This contradicts Theorem 2, on the APX-hardness of MAX-3DM(3). ◀

7 Conclusions

This work resolves an open question on algorithms for pseudoknot-free secondary structure prediction of nucleic acids: Can we efficiently compute the minimum free energy (MFE) pseudoknot-free secondary structure for a multi-set of DNA or RNA strands? We have shown that this problem is NP-hard, and is therefore computationally intractable, unless $P = NP$. A natural question then is whether solutions to the problem can be efficiently approximated, if $P \neq NP$. Unfortunately, there is a limit to the accuracy of any such method. We have shown that the optimization problem of finding the MFE structure for a multi-set of nucleic acid strands is hard for the complexity class APX, the class of NP optimization problems that have constant factor approximation algorithms. The result implies that there does not exist a polynomial time approximation scheme for this problem, unless $P = NP$. Given these results, it suggests that heuristic methods, such as stochastic local search, and randomized algorithms should be investigated for structure prediction of multiple interacting strands.

References

- 1 Tatsuya Akutsu. Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots,. *Discrete Applied Mathematics*, 104(1-3):45–62, August 2000.
- 2 Mirela Andronescu, Zhi Chuan Zhang, and Anne Condon. Secondary structure prediction of interacting RNA molecules. *Journal of Molecular Biology*, 345(5):987–1001, February 2005.
- 3 Karl Bringmann, Fabrizio Grandoni, Barna Saha, and Virginia Vassilevska Williams. Truly Subcubic Algorithms for Language Edit Distance and RNA-Folding via Fast Bounded-Difference Min-Plus Product. *SIAM Journal on Computing*, 48(2):481–512, 2019.
- 4 Robert M. Dirks, Justin S. Bois, Joseph M. Schaeffer, Erik Winfree, and Niles A. Pierce. Thermodynamic analysis of interacting nucleic acid strands. *SIAM Rev.*, 49(1):65–88, 2007. doi:10.1137/060651100.

- 5 Mark E Fornace, Nicholas J Porubsky, and Niles A Pierce. A unified dynamic programming framework for the analysis of interacting nucleic acid strands: Enhanced models, scalability, and speed. *ACS Synthetic Biology*, 9(10):2665–2678, 2020.
- 6 Kenichi Fujibayashi, Rizal Hariadi, Sung Ha Park, Erik Winfree, and Satoshi Murata. Toward reliable algorithmic self-assembly of DNA tiles: A fixed-width cellular automaton pattern. *Nano Letters*, 8(7):1791–1797, July 2008.
- 7 Michael R Garey and David S Johnson. *Computers and Intractability: A guide to NP-completeness*, 1979.
- 8 Dan S Hirschberg. *Pattern matching algorithms*, chapter Serial computations of Levenshtein distances, pages 123–142. Oxford university press, 1997.
- 9 J. A. Jaeger, D. H. Turner, and M. Zuker. Predicting optimal and suboptimal secondary structure for RNA. *Methods in Enzymology*, 183:281–306, 1990.
- 10 J. Justesen. A class of constructive asymptotically good algebraic codes. *Information Theory, IEEE Transactions on*, 18(5):652–656, 1972.
- 11 Viggo Kann. Maximum bounded 3-dimensional matching is MAX SNP-complete. *Information Processing Letters*, 37(1):27–35, 1991.
- 12 Ronny Lorenz, Stephan H Bernhart, Christian Höner zu Siederdissen, Hakim Tafer, Christoph Flamm, Peter F Stadler, and Ivo L Hofacker. ViennaRNA package 2.0. *Algorithms for Molecular Biology : AMB*, 6:26, November 2011.
- 13 R. B. Lyngsø and C. N. Pedersen. RNA pseudoknot prediction in energy-based models. *Journal of Computational Biology*, 7(3-4):409–427, 2000.
- 14 Rune B. Lyngsø. Complexity of pseudoknot prediction in simple models. In Josep Diaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannellä, editors, *Proceedings, Automata, Languages and Programming 31st International Colloquium, ICALP*, volume 3142 of *Lecture Notes in Computer Science*, pages 919–931. Springer Berlin/Heidelberg, January 2004. doi:10.1007/b99859.
- 15 David H. Mathews and Douglas H. Turner. Prediction of RNA secondary structure by free energy minimization. *Current Opinion in Structural Biology*, 16(3):270–278, 2006. doi:10.1016/j.sbi.2006.05.010.
- 16 J. S. McCaskill. The equilibrium partition function and base pair binding probabilities for RNA secondary structure. *Biopolymers*, 29(6-7):1105–1119, June 1990.
- 17 R. Nussinov and A. B. Jacobson. Fast algorithm for predicting the secondary structure of single-stranded RNA. *Proceedings of the National Academy of Sciences of the United States of America*, 77(11):6309–6313, November 1980.
- 18 R. Nussinov, G. Pieczenik, J. Griggs, and D. Kleitman. Algorithms for loop matchings. *SIAM Journal on Applied Mathematics*, 35(1):68–82, 1978.
- 19 L.J. Schulman and D. Zuckerman. Asymptotically good codes correcting insertions, deletions, and transpositions. *IEEE Transactions on Information Theory*, 45(7):2552–2557, 1999. doi:10.1109/18.796406.
- 20 Bryan Wei, Mingjie Dai, and Peng Yin. Complex shapes self-assembled from single-stranded DNA tiles. *Nature*, 485(7400):623–626, 2012.
- 21 S. Wuchty, W. Fontana, I. L. Hofacker, and P. Schuster. Complete suboptimal folding of RNA and the stability of secondary structures. *Biopolymers*, 49(2):145–165, February 1999.
- 22 Joseph N. Zadeh, Brian R. Wolfe, and Niles A. Pierce. Nucleic acid sequence design via efficient ensemble defect optimization. *Journal of Computational Chemistry*, 32(3):439–452, 2011. doi:10.1002/jcc.21633.
- 23 M. Zuker and P. Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic Acids Research*, 9(1):133–148, January 1981.
- 24 Michael Zuker and David Sankoff. RNA secondary structures and their prediction. *Bulletin of Mathematical Biology*, 46(4):591–621, July 1984. doi:10.1007/BF02459506.

A Technical Appendix

A.1 Proof of Lemma 6

Proof. Let $\mathbf{d}_{\text{LCS}}(a, b) = k$. We suppose that $\mathbf{d}_{\text{LCS}}(\rho^i(a), \rho^i(b)) < k/2$ and obtain a contradiction. Let \mathcal{A} be an optimal alignment of $\rho^i(a)$ and $\rho^i(b)$. Throughout, when referring to characters in $\rho^i(a)$ and $\rho^i(b)$, we denote the characters of the original strings a and b by A_o and T_o and the padded A's by A_p . Each pair of characters in alignment \mathcal{A} has one of four types: *original*, with two original characters; *padded*, with two padded characters; *mixed*, with one A_o and one A_p , or *gap*, with one gap symbol. Let n be the length of a and b , and let $\#_{orig}$, $\#_{pad}$ and $\#_{mix}$ denote, in order, the counts of original, padded and mixed pairs, respectively. To prove the lemma, we establish various bounds on these counts, as a function of n and k .

First, note that any alignment of $\rho^i(a)$ and $\rho^i(b)$ has at most $n - \frac{k}{2}$ original pairs: Otherwise, we could use the alignment to obtain an alignment of a and b with less than k gap pairs, which is not possible since $\mathbf{d}_{\text{LCS}}(a, b) = k$. Therefore,

$$\#_{orig} \leq n - \frac{k}{2}. \quad (1)$$

Second, using Theorem 3 and our assumption that $\mathbf{d}_{\text{LCS}}(\rho^i(a), \rho^i(b)) < \frac{k}{2}$, we have that $\text{LCS}(\rho^i(a), \rho^i(b)) \geq (i+1)n - \lfloor \frac{k}{4} \rfloor$, and so

$$\#_{orig} + \#_{pad} + \#_{mix} = \text{LCS}(\rho^i(a), \rho^i(b)) \geq (i+1)n - \lfloor \frac{k}{4} \rfloor. \quad (2)$$

Third, we'll obtain a lower bound on $\#_{orig}$. Note that $2\#_{pad} + \#_{mix}$ is upper bounded by the total number of A_p characters, and so is at most $2in$. Therefore $\#_{pad} + \lceil \frac{\#_{mix}}{2} \rceil \leq in$. Substituting this inequality into Equation 2, we have that

$$\#_{orig} \geq n - \lfloor \frac{k}{4} \rfloor - \lfloor \frac{\#_{mix}}{2} \rfloor. \quad (3)$$

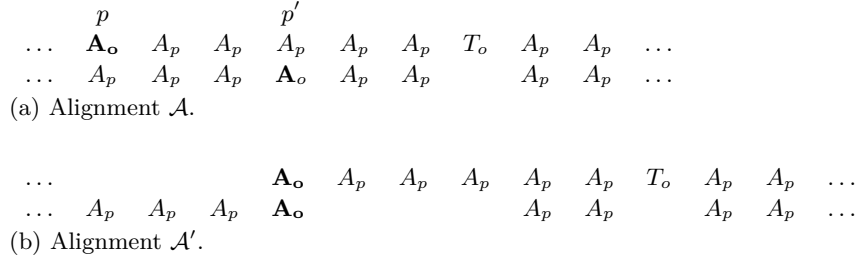
Finally, from inequalities 1 and 3 we have $\#_{mix} \geq k/2$.

We now partition the mixed pairs into two types: *sloppy* and *tight*.

- A mixed pair p of alignment \mathcal{A} is *sloppy* if, among the first i pairs to the right of p , there is at least one gap pair containing a T_o or A_p character. Mixed pairs must be separated by at least i pairs since there are at least i A_p 's between any two A_o 's, and so the gap pairs corresponding to each of the sloppy mixed pairs are distinct. From property (3) above, $\mathbf{d}_{\text{LCS}}(\rho^i(a), \rho^i(b))$ is equal to the number of gap pairs in \mathcal{A} . Since we are assuming that $\mathbf{d}_{\text{LCS}}(\rho^i(a), \rho^i(b)) < \frac{k}{2}$, the alignment \mathcal{A} has less than $\frac{k}{2}$ gapped pairs, and thus has less than $\frac{k}{2}$ sloppy mixed pairs.
- If p is not sloppy, we call it *tight*. Since less than $\frac{k}{2}$ of the mixed pairs are sloppy, at least $\#_{mix} - \frac{k}{2} + 1$ of the mixed pairs are tight.

If p is tight, let p' be the first pair to the right of p that is not a padded pair. Such a pair p' must exist, since our padding function is such that any A_p character is eventually followed by an original character. Pair p' is either a gap pair containing A_o or is a mixed pair, in which case it also contains A_o . In either case, because exactly i A_p 's separate any two original characters, if the A_o character of pair p is in string a then the A_o character of pair p' is in string b and vice versa. In what follows, we refer to p' as p 's *partner*. Note that p' may itself be a tight pair.

Using these $\#_{mix} - \frac{k}{2} + 1$ tight pairs, we now convert alignment \mathcal{A} to another alignment \mathcal{A}' with at least $n - \frac{k}{2} + 1$ original pairs, obtained as follows; see example in Figure 5. Starting from the leftmost pair of alignment \mathcal{A} and working towards the right, find the first tight



■ **Figure 5** Illustration of the construction of Lemma 6. In alignment \mathcal{A} , the pair at position p is a tight pair; its partner is at position p' and is a sloppy pair. Alignment \mathcal{A}' has one more original pair, indicated in bold, than does alignment \mathcal{A} .

mixed pair p of \mathcal{A} and its partner p' . Remove p, p' and all of the intervening (padded) pairs between them from the alignment, and instead pair each padded character from the removed pairs with a gap, and pair the A_o character of p with the A_o character of p' (recall that one of these A_o characters is in string a and the other is in string b). Repeat, starting from the pair just to the right of p' , until the rightmost end of \mathcal{A} is reached.

The number of new original pairs obtained in this manner is at least $\lfloor \frac{\#_{mix}}{2} \rfloor - \lceil \frac{k}{4} \rceil + 1$. To see why:

- If $\#_{mix} - \frac{k}{2} + 1$ is odd, then the number of new original pairs is at least

$$(\#_{mix} - \frac{k}{2})/2 + 1 \geq \lfloor \frac{\#_{mix}}{2} \rfloor - \lceil \frac{k}{4} \rceil + 1.$$

This lower bound is achieved when all but one of the tight mixed pairs are partners.

- If $\#_{mix} - \frac{k}{2} + 1$ is even, then the number of new original pairs is at least

$$(\#_{mix} - \frac{k}{2} + 1)/2 = \lfloor (\#_{mix} - \frac{k}{2})/2 \rfloor + 1/2 + 1/2 = \lfloor \frac{\#_{mix}}{2} \rfloor - \lfloor \frac{k}{2} \rfloor + 1.$$

This lower bound is achieved when all partners are themselves tight mixed pairs.

Therefore, the number of original pairs in alignment \mathcal{A}' is

$$\begin{aligned} \#_{orig} + \lfloor \frac{\#_{mix}}{2} \rfloor - \lceil \frac{k}{4} \rceil + 1 \\ \geq n - \lfloor \frac{k}{4} \rfloor - \lfloor \frac{\#_{mix}}{2} \rfloor + \lfloor \frac{\#_{mix}}{2} \rfloor - \lceil \frac{k}{4} \rceil + 1 \quad (\text{using inequality 3}) \\ = n - \frac{k}{2} + 1. \end{aligned}$$

But as noted earlier, any alignment of $\rho^i(a)$ and $\rho^i(b)$ has at most $n - \frac{k}{2}$ original pairs, and so we have our contradiction. ◀

A.2 Proof of Lemma 7

Proof. To show part 1, first suppose that s is a substrand of $a = \rho^5(a')$. If $|s| \leq 2$, then no bases of s are paired in $S(s)$, given our assumption that consecutive bases in a strand cannot form a base pair, and so part 1 holds. If $|s| \geq 3$, the number of (intra-molecular) base pairs of $S(s)$ is at most the number of T's in s . If $3 \leq |s| \leq 6$ then s can have at most one T, and thus at most one base pair, so s has at least $|s| - 2$ unpaired bases and again part 1 holds. Suppose that $|s| \geq 7$. Because s is a substrand of a padded strand, the number of T's in s is at most $\lceil 2|s|/7 \rceil$: this maximum is achieved if $|s| = 7$ and s both starts and ends with a T. Even if all of the T's of s are paired to A's, the number of unpaired A's is still at least $\lfloor 3|s|/7 \rfloor \geq |s|/3$ since $|s| \geq 7$. The argument when s is a substrand of \bar{a} is obtained by replacing A's with T's in the argument for a substrand of a .

Similarly, the total number of T's in $S(a, b)$ is at most $(|a| + |b|)/6$ and so the unpairedness is at least $4(|a| + |b|)/6$. The argument for the unpairedness of $S(\bar{a}, \bar{b})$ is obtained by replacing A's with T's in the argument for $\{a, b\}$.

Finally, the inter-molecular base pairs of $S(a, \bar{b})$ correspond to a common subsequence of strands a and b , and thus the number of such base pairs is at most $\mathbf{LCS}(a, b) = n - \frac{\mathbf{d}_{\mathbf{LCS}}(a, b)}{2}$ by Theorem 3. Therefore the total number of bases in both a and \bar{b} that do not form inter-molecular base pairs of $S(a, \bar{b})$ is at least $\mathbf{d}_{\mathbf{LCS}}(a, b)$. Now consider any substructure of $S(a, \bar{b})$ within some maximal substrand s of either a or \bar{b} that has no inter-molecular base pairs. The unpairedness of this substructure is at least $\frac{1}{3}|s|$, by part 1 of this Lemma. Thus, over all substrands that do not contain inter-molecular base pairs, at least a fraction $\frac{1}{3}$ of bases are unpaired (not involved in intra-molecular base pairs). Since the total length of such substrands is at least $\mathbf{d}_{\mathbf{LCS}}(a, b)$, the unpairedness of $S(a, \bar{b})$ is at least $\frac{1}{3}\mathbf{d}_{\mathbf{LCS}}(a, b)$. ◀

A.3 Proof of Lemma 11

Proof. Instance I' has one template strand, $2n$ separator supports, $2n$ separator-complement supports $6n$ xyz-support strands, and $2m + n$ trim-complement strands, for a total of $2m + 11n + 1$ strands. The template strand has $13m$ domains and the other strands have one domain each, for a total of $15m + 11n$ domains.

Since every domain in the construction has length $\Theta(\log_2 n)$, instance I' is of size polynomial in n overall. The sequences can also be designed in polynomial time: The sequence design of separator and trim domains is trivial, and the sequences for the x, y, z domains can be designed in time polynomial in n by Theorem 9. ◀

A.4 Proof of Lemma 13

Proof. The number of trim-deprived and flawed triples is $m - p - h$, where m, p , and h are the number of triples, perfect triples, and hogger triples, respectively.

Perfect triples and hogger triples are not trim-deprived. Therefore, any support strand connected to a perfect triple or a hogger triple cannot also be connected to another triple without creating a pseudoknot. Each perfect triple has 10 support strands bound to it, and each hogger triple has at least 11 connected support strands. From Lemma 10, there are $10n$ support strands in total, so $10p + 11h \leq 10n$ and

$$h \leq 10(n - p)/11.$$

Since the optimal matching of I has size at most $n - i$, the number of perfect triples p must be at most $n - i$ and so $n - p \geq i$. Therefore, the total number of trim-deprived and flawed triples is

$$m - p - h \geq m - p - 10(n - p)/11 = m - n + (n - p)/11 \geq (m - n) + i/11. \quad \blacktriangleleft$$

A.5 Proof of Lemma 15

Proof. Each trim-deprived triple forms at most $2E$ CG base pairs, with the G's being in the trims (center-trim and end-trims) of the triple and the C's being in trim-complement strands. Triples that are not trim-deprived form at most $3E$ CG base pairs. There are no other CG base pairs. So, the total number of CG base pairs is at most

$$(m - n + i/22)2E + (m - (m - n + i/22))3E = (2m + n - i/22)E.$$

The total number of trim-complement strands is $2m + n$, each containing E C's. So, the number of unpaired C bases in trim-complements is at least $iE/22$. ◀

A.6 Proof of Lemma 16

Proof. Suppose that there are l bonds between x and Sep_{xy} ; the other cases are similar. Since Sep_{xy} contains only A's, only T's of x can bind with Sep_{xy} . Our sequence design ensures that there are at least five padded A's between any two successive T's of x . Therefore, in order to avoid pseudoknots, if there are l bonds between x and Sep_{xy} , at least $5(l - 1)$ padded A's remain unpaired. ◀

A.7 Proof of Lemma 17

Proof. Suppose to the contrary that the first condition does not hold, i.e., one of $F_{5'}$'s Sep domains is not bound to a $\overline{\text{Sep}}$ support. The total number of T's that can bind to the Sep domain is at most $5.5E$, accounted for as follows. There are at most $3E/6$ T's in the x , y , and z domains of $F_{5'}$ plus at most $5E$ in the remaining support strands, if there are five xyz -support strands. Thus at least $E/2$ of the $6E$ A's in the Sep domain are unpaired. Since $E \geq \log_2 n$, we get a contradiction to the hypothesis of the lemma. Thus the first condition must hold.

Next suppose that the first condition holds but that the second does not; specifically that the x domain of $F_{5'}$ is not bound to an xyz -support domain (the argument is similar for the y or z domains). Recall that domain x contains at least $\log_2 n$ T's, since by design the domains comprise a $\log_2 n$ -robust set. At least $2(\log_2 n)/3$ of the T's must be paired, or the hypothesis of the lemma that the ACT-unpairedness of $F_{5'}$ is less than $(\log_2 n)/3$ would not be true. Since the first condition of the lemma holds, the Sep domain adjacent to x on the $5'$ flank is bound to a $\overline{\text{Sep}}$ strand. Therefore domain x cannot have bonds to domain y or z , or to the Sep domain between y and z , or a pseudoknot would form. Also, the T's in domain x cannot bind to $\overline{\text{Sep}}$ strands, since $\overline{\text{Sep}}$'s are composed only of T's. If there were at least $(\log_2 n)/3$ bonds between x and Sep_{xy} , Lemma 16 would imply that x has ACT-unpairedness at least $5((\log_2 n)/3 - 1) \geq \log_2 n$, again contradicting the hypothesis of the lemma.

Therefore, at least $(\log_2 n)/3$ T's of x must form intramolecular bonds with A's that are also in the x domain. The total length of substrands of x that have either unpaired bases or intramolecular base pairs must be at least $3(\log_2 n)/3$: this lower bound is met if each T, say at position i of x is bound to an A that is either at position $i - 2$ or $i + 2$ (since we assume that no base pair can form between consecutive bases). Part 1 of Lemma 7 therefore implies that x has ACT-unpairedness at least $(\log_2 n)/3$, once again contradicting the hypothesis of the lemma. We conclude that the second condition of the lemma must hold.

Since both conditions hold, it cannot be that two of the x , y , and z domains of $F_{5'}$ are bound to the same xyz -support of $\mathcal{S}_{5'}$, or a pseudoknot would form with bonds between a Sep of $F_{5'}$ and a $\overline{\text{Sep}}$ support. Similarly, it cannot be that both Sep 's have bonds to the same $\overline{\text{Sep}}$. Hence, each Sep domain of $F_{5'}$ is bound to a distinct $\overline{\text{Sep}}$ support of $\mathcal{S}_{5'}$, and $\mathcal{S}_{5'}$ contains exactly three xyz -supports and two $\overline{\text{Sep}}$ supports, completing the proof of the Lemma. ◀

A.8 Proof of Lemma 18

Proof. Let $\alpha < 1/7$. Suppose to the contrary that the ACT-unpairedness of $\text{Opt}(F_{5'})$ is less than $\alpha \log_2 n$. By Lemma 17, $\mathcal{S}_{5'}$ must contain three xyz -supports, say a , b , and c , with a bound to x , b bound to y , and c bound to x .

We first show that in $\text{Opt}(F_{5'})$, there can be at most $\alpha \log_2 n/5$ bases between a Sep domain of $F_{5'}$ and one of the domains x , y , or z adjacent to the Sep domain. Otherwise, by Lemma 16, at least $\alpha \log_2 n$ bases of a would be unpaired, and we get a contradiction. Similarly, there can be at most $\alpha \log_2 n/5$ bases between a $\overline{\text{Sep}}$ domain of $F_{5'}$ and one of the domains a , b , or c adjacent to the $\overline{\text{Sep}}$ domain.

Since $F_{5'}$ is the flank of a flawed triple, either $a \neq \bar{x}$, $b \neq \bar{y}$, or $c \neq \bar{z}$. First suppose that $a \neq \bar{x}$. Since the set of domains is $\log_2 n$ -robust, there can be at most $E - \log_2 n$ base pairs between a and x . By the argument in the previous paragraph, x has at most $\alpha(\log_2 n)/5$ bases to Sep_{xy} . Similarly, if $\overline{\text{Sep}}_{ab}$ is the separator complement between a and b , then a has at most $\alpha(\log_2 n)/5$ bases to $\overline{\text{Sep}}_{ab}$. If a has base pairs with Sep_{xy} , then x cannot have base pairs with $\overline{\text{Sep}}_{ab}$ and vice versa, in order to avoid pseudoknots. Therefore, either a or x has at least $\log_2 n - \alpha(\log_2 n)/5 \geq 34(\log_2 n)/35$ bases that are either unpaired or form intramolecular bonds. By Lemma 7, either a or x has unpairedness at least $11(\log_2 n)/35 \geq (\log_2 n)/4$, proving the lemma. The argument when $c \neq \bar{z}$ is similar to that when $a \neq \bar{x}$.

Finally, suppose that $a = \bar{x}$ and $c = \bar{z}$ but $b \neq \bar{y}$. As noted earlier, b has at most $\alpha(\log_2 n)/5$ bonds with each $\overline{\text{Sep}}$ adjacent to it. Also, at least $\log_2 n$ bases of b are not paired with y , since the set of domains is $\log_2 n$ -robust. Of these, at most $\alpha \log_2 n$ can be unpaired, or again we get a contradiction. Therefore, b has at least $\log_2 n - 2\alpha(\log_2 n)/5 - \alpha \log_2 n = \log_2 n - 7\alpha(\log_2 n)/5$ bonds to the Sep 's adjacent to y , and so b has at least $\frac{1}{2}(\log_2 n - 7\alpha(\log_2 n)/5)$ bonds to Sep_{xy} .

Moreover, $\overline{\text{Sep}}_{ab}$ must have at least $6E - 12\alpha(\log_2 n)/5$ base pairs with Sep_{xy} . This is because $\overline{\text{Sep}}_{ab}$ has at most $\alpha(\log_2 n)/5$ bases with each of a and b , and $\overline{\text{Sep}}_{ab}$ has at most $3\alpha \log_2 n$ bases paired with x . To see why the latter assertion holds, note that otherwise at least $3\alpha \log_2 n$ bases of a are not paired with any strand other than a and thus by Lemma 7, at least $\alpha \log_2 n$ bases of a are unpaired, which again is a contradiction. Therefore, $\overline{\text{Sep}}_{ab}$ has at most $(2/5 + 3)\alpha \log_2 n$ pairs in total with a , x , and b , and since at most $\alpha \log_2 n$ bases of $\overline{\text{Sep}}_{ab}$ can be unpaired, $\overline{\text{Sep}}_{ab}$ has at least $6E - (2/5 + 3 - 1)\alpha \log_2 n = 6E - (12/5)\alpha \log_2 n$ base pairs with Sep_{xy} .

Therefore the total number of bases that are paired with bases of Sep_{xy} is at least $\frac{1}{2}(\log_2 n - 7\alpha(\log_2 n)/5)$ (with b) plus $6E - (12/5)\alpha \log_2 n$ (with $\overline{\text{Sep}}_{ab}$). The total is

$$6E + (1/2 - 7\alpha/10 - \alpha(12/5)) \log_2 n \geq 6E + (1/2 - \alpha(31/10)) \log_2 n.$$

Since $\alpha \leq 1/7$, this quantity is greater than $6E$, again a contradiction since the length of Sep_{xy} is $6E$. \blacktriangleleft

Reactamole: Functional Reactive Molecular Programming

Titus H. Klinge ✉

Drake University, Des Moines, IA, USA

James I. Lathrop ✉

Iowa State University, Ames, IA, USA

Peter-Michael Osera ✉

Grinnell College, Grinnell, IA, USA

Allison Rogers ✉

Grinnell College, Grinnell, IA, USA

Abstract

Chemical reaction networks (CRNs) are an important tool for molecular programming, a field that is rapidly expanding our ability to deploy computer programs into biological systems for a variety of applications. However, CRNs are also difficult to work with due to their massively parallel nature, leading to the need for higher-level languages that allow for easier computation with CRNs. Recently, research has been conducted into a variety of higher-level languages for deterministic CRNs but modeling CRN parallelism, managing error accumulation, and finding natural CRN representations are ongoing challenges.

We introduce REACTAMOLE, a higher-level language for deterministic CRNs that utilizes the functional reactive programming (FRP) paradigm to represent CRNs as a reactive dataflow network. REACTAMOLE equates a CRN with a functional reactive program, implementing the key primitives of the FRP paradigm directly as CRNs. The functional nature of REACTAMOLE makes reasoning about molecular programs easier, and its strong static typing allows us to ensure that a CRN is well-formed by virtue of being well-typed. In this paper, we describe the design of REACTAMOLE and how we use CRNs to represent the common datatypes and operations found in FRP. We also demonstrate the potential of this functional reactive approach to molecular programming by giving an extended example where a CRN is constructed using FRP to modulate and demodulate an amplitude modulated signal.

2012 ACM Subject Classification Software and its engineering → Functional languages; Software and its engineering → Data flow languages

Keywords and phrases Chemical Reaction Network, Functional Reactive Programming, Domain Specific Language

Digital Object Identifier 10.4230/LIPIcs.DNA.27.10

Supplementary Material *Software (Source Code)*: <https://github.com/digMP/haskell-reactamole> archived at `swh:1:dir:9f8a912be45038fedf03c9dd6bf74a6a20f29aff`

Funding *James I. Lathrop*: This research was supported in part by NSF grants 1545028, 1900716, and 1909688.

Peter-Michael Osera: This research was supported in part by NSF grants 1651817 and 2049911.

Acknowledgements We thank the four anonymous reviewers for their feedback on this paper and Noah Susag for his contributions to the REACTAMOLE code.

1 Introduction

Molecular programming harnesses computer science towards designing programmable structures at the nanoscale, unlocking the potential to execute programs in biological systems. This emerging arena holds significant potential for innovations in medicine, nanofabrication, and synthetic biology. One prominent molecular programming language is chemical reaction



© Titus H. Klinge, James I. Lathrop, Peter-Michael Osera, and Allison Rogers; licensed under Creative Commons License CC-BY 4.0

27th International Conference on DNA Computing and Molecular Programming (DNA 27).

Editors: Matthew R. Lakin and Petr Šulc; Article No. 10; pp. 10:1–10:20

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

networks (CRNs), abstractions of chemical reactions [8, 5, 10]. CRNs are Turing-complete and act as an unstructured assembly language for molecular programming, which can then be assembled into DNA to perform computation at the nanoscale [9, 1].

However, the characteristics of CRNs create substantial challenges for general programmability. Due to the nature of chemical reactions, CRNs are massively parallel, with all reactions active at the same time depending on the availability of the reactants. This creates race conditions that can make coding in this framework more difficult and error-prone [22]. CRNs are also unstructured and not easily composed; adding a single reaction can easily have global side effects to its behavior. This is strong motivation for the creation of a high-level programming language to abstract away these barriers.

Consequently, recent research has been conducted into high-level languages for molecular programming such as CRN++ [22] and Kaemika [4]. CRN++ enables programming in a familiar, imperative style that compiles to deterministic CRNs (DCRNs), marking a significant advancement in this realm but also leaving room for continued improvement. Vasic et al. say that CRN++ could be improved by addressing inefficiencies caused by careful avoidance of the inherent parallelism of CRNs, as well as reducing errors accumulated over time [22]. Kaemika supports specifying CRNs in a functional style, including support for high-order functions and recursion [4]. Both CRN++ and Kaemika allow side effects, with Kaemika depending on these side effects for the generation of new species within functions. Additionally, these implementations leave room to improve the synchronicity between language structure and CRN behavior.

We address these downsides by exploring the use of *functional reactive programming (FRP)* for developing CRNs. Functional reactive programming is a paradigm primarily characterized by its reactive nature, responding to stimuli through continuous and discrete time-dependent inputs [2]. In FRP, systems are modeled as graphs where nodes are operations and edges indicate how data flows between these operations, with a particular focus on how change is propagated through this graph. We observe a close correspondence between CRNs and functional reactive programs. The chemical concentrations of a CRN react to changes in their environment similarly and can thus be thought of as *signals*, time-varying data streams, in a functional reactive program. Consequently, CRNs themselves transform these concentrations and can, therefore, be thought of as functions over signals, i.e., *signal functions*. These correspondences make FRP a natural choice to express computation within a CRN.

We use this correspondence to design a *functional reactive molecular programming (FRMP)* language for deterministic CRNs. The heart of this language is the expression of the core constructs of FRP directly in terms of CRNs, gaining the benefits of program composability afforded by the functional reactive paradigm. We explore this approach to molecular programming with REACTAMOLE, an embedded domain specific language (eDSL) for FRMP modeled after Yampa, a prominent functional reactive programming DSL [14]. Furthermore, by combining FRP and CRNs, we open the door for applying recent advancements in programming language theory towards the development of CRNs. For example, in this FRMP paradigm, we can now consider integrating type systems that verify safety properties of functional reactive programs [18] or program synthesizers for functional reactive programs [11].

In Section 2, we review the basic definitions of chemical reaction networks as well as introduce the necessary components of the Haskell programming language and functional reactive programming needed to understand REACTAMOLE. We introduce REACTAMOLE by way of example in Section 3 and describe the design and implementation of REACTAMOLE in Section 4. Finally, we demonstrate the potential of REACTAMOLE by way of a case study – implementing amplitude modulation over real-valued signals – in Section 5.

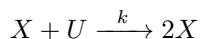
2 Background

In this section, we review the two main topics that we combine in REACTAMOLE: chemical reactions networks (Subsection 2.1) and functional reactive programming (Subsection 2.2). Throughout the remaining sections, we assume familiarity with the basic syntax and semantics of the Haskell programming language. We provide a walkthrough of the parts of Haskell necessary to understand our presentation in Appendix A.

2.1 Chemical Reaction Networks

Scientists and researchers often utilize models to describe the complex interactions of molecules and matter. These models simplify molecular interactions in order to allow algorithms and software to simulate the outcomes of chemical systems. Even though these models make simplifying assumptions, they are still a powerful tool for designing and testing these systems. A chemical reaction network (CRN) is one such model that is often used to model molecule interactions in a well-mixed solution. Even though assumptions are made in each of the many variants of the CRN model, almost all of them retain the power to facilitate general computing.

We adopt much of the notation used by Klinge, Lathrop, and Lutz [20] to formalize the CRN model used here. A CRN is a pair $N = (S, R)$, where S is a set of species (molecules) and R is a set of reactions that operate over those species. Reactions are a triple $\rho = (\mathbf{r}, \mathbf{p}, k)$, where $\mathbf{r} \in \mathbb{N}^S$, $\mathbf{p} \in \mathbb{N}^S$, and $k \in (0, \infty)$. Note that \mathbb{N}^S is the set of functions mapping species to non-negative integers, and we do not allow $\mathbf{r} = \mathbf{p}$. The constant k represents the rate constant for the reaction. In this paper we usually represent CRNs simply as a set of reactions that implicitly define a set of species. For example, the reaction



can be interpreted as the CRN $N = (S, R)$ with $S = \{X, U\}$ and R consisting of only the given reaction.

In this paper, we focus on the deterministic mass-action model of chemical reaction networks, where species are represented by concentrations of molecules. This is in contrast to the stochastic mass-action model, which uses the *number* of molecules for this purpose. The deterministic mass-action model describes interactions with molecules using polynomial autonomous differential equations, and the semantics relate concentrations of species through the reactants, products, and rate constants of all reactions in the system. The single reaction above yields the set of ordinary differential equations (ODEs):

$$\frac{dx}{dt} = x \cdot u \cdot k \qquad \frac{du}{dt} = -x \cdot u \cdot k.$$

Intuitively, since an X and a U react together to produce an additional X in the system, the instantaneous rate of X gained is proportional to the product of the instantaneous amounts of the reactants and the rate constant. Simultaneously, the loss of U occurs at the same rate. Note that we use $x(t)$ or x to denote the concentration of species X .

Similar to Klinge, Lathrop, and Lutz [20], we define an *input/output CRN* (I/O CRN) as a tuple $N = (S, R, I, O)$, where S is a set of species, R is a set of reactions, $I \subseteq S$ is the set of input species, and $O \subseteq S$ is the set of output species. I/O CRNs require that the input species are only used as a catalyst in any reaction, a critical feature that allows us to compose CRNs in REACTAMOLE safely.

2.2 Functional Reactive Programming

Many classes of computation can be expressed as programs that propagate change in response to external stimuli. For example, with:

- Graphical user interfaces (GUIs), interface elements update in response to user input, e.g., mouse movement.
- Spreadsheets, cells that are related via (potential cyclic) references update whenever the user modifies their contents.
- Circuits, input signals propagate through interconnected electrical components.

We could model these phenomena with mutable state. The resulting program would then bear the responsibility of orchestrating how the different pieces of state change in response to the outside world. However, by doing so, we would lose the benefits of composability, a hallmark of the functional style of programming [17].

Functional Reactive Programming (FRP) [6] purifies this stateful situation by modeling values that react with the outside world as *signals*:

```
type Signal a = Time -> a
```

That is, a signal of some arbitrary type `a` is a time-varying value, i.e., a function from time to `a`. For example, an electrical pulse that we might measure using two states, on and off, could be represented as a `Bool`. In an FRP setting, this pulse would be represented as a type, `Signal Bool`, a function describing how the pulse changes over time.

Within FRP, there are a multitude of approaches and variations to address implementation concerns such as space efficiency or design constraints such as modeling discrete versus continuous time and static versus dynamic dependencies between components. In this work, we focus on *arrowized FRP*, which uses the arrow abstraction of Hughes [16], a generalization of composable computation. These arrows take the form of *signal functions* in arrowized FRP, i.e., transformers over signals:

```
type SF a b = Signal a -> Signal b
```

For example, a function `not` that inverts an electrical pulse would have the type `SF Bool Bool`, a signal function that takes a Boolean signal as input and produces a Boolean signal as output.

Some of these signal functions, like `not`, transform our time-varying values directly. Other signal functions are *higher-order* signal functions which take other signal functions as input and produce them as output. These signal function *combinators* allow us to build up more complex signal functions from simpler ones. The most common of these is function composition, traditionally written in the arrow style as the binary operator (`>>>`):

```
(>>>) :: SF a b -> SF b c -> SF a c
```

`f >>> g` is the composition of signal functions `f` and `g` where the output of `f` (of type `b`) is fed into `g` as input. The resulting signal function takes an input for `f` (of type `a`) and produces an output from `g` (of type `c`) as its result. Other common signal function combinators that we will use in the subsequent sections include:

- The *split operator* (`***`) `:: SF a b -> SF c d -> SF (a, c) (b, d)` which takes two input signal functions `f` and `g` and creates a signal function whose inputs and outputs are *pairs* drawn from `f` and `g`.
- The *fanout operator* (`&&&`) `:: SF a b -> SF a c -> SF a (b, c)` which takes two input signal functions `f` and `g` that take a common input type `a` and creates a new signal function that pipes its input independently through both `f` and `g` and produces their outputs as a pair.

3 Introducing Reactamole

In this section, we give a brief summary of REACTAMOLE and its uses. Note that many of the REACTAMOLE primitives discussed in this section are further explained later in the paper.

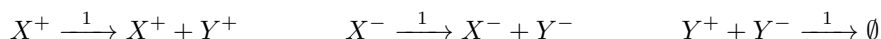
Klinge, Lathrop, and Lutz [20] regarded a chemical reaction network as a device that transforms an input signal into an output signal. The authors defined an *input/output chemical reaction network (I/O CRN)* to be a CRN with some species reserved as “inputs” that can only be used catalytically and some species labeled “outputs.” These I/O CRNs are literally signal functions in the functional reactive programming sense. As a result, I/O CRNs are specified using the arrow combinators that we introduced in Subsection 2.2. Below, we demonstrate the expressive power of these arrowized I/O CRNs.

We begin with I/O CRNs that produce real numbers. Since species concentrations are non-negative, we encode a real number as the difference between two species. (This is a common technique and was used to show that CRNs are equivalent to the general purpose analog computer (GPAC) [13]). Thus, a real-valued signal $x(t)$ is encoded as $x^+(t) - x^-(t)$ where X^+ and X^- are two species.

One of the simplest I/O CRN operations is the *integrator*. An integrator takes a real-valued signal $x(t)$ and produces the real-valued output signal $y(t) = \int_0^t x(s)ds + y_0$. In REACTAMOLE, we provide an integrator as an I/O CRN primitive:

```
integrate :: Double -> CRN Double Double
```

`integrate` is a function that takes a real-valued parameter `y0` and returns a signal function that performs integration. The parameter `y0` corresponds to the constant $y(0)$ in the solution to $y(t)$. We implement the `integrate` function, with parameter `y0`, using an I/O CRN consisting of the three reactions:



The third reaction is an *annihilation* reaction that has no effect on the encoded value of $y(t)$ but ensures that concentrations of Y^+ and Y^- remain close to zero. We can verify the correctness of the above CRN by examining the induced ODEs according to the law of mass action:

$$\frac{dy^+}{dt} = x^+ - y^+y^-, \quad \frac{dy^-}{dt} = x^- - y^+y^-.$$

Since the functions are dual-rail, we know that $x(t) = x^+(t) - x^-(t)$ and $y(t) = y^+(t) - y^-(t)$. Thus, we can rewrite the above ODEs in terms of x and y directly:

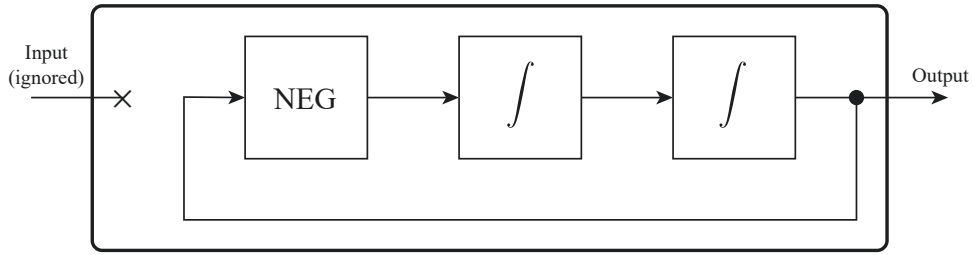
$$\frac{dy}{dt} = \frac{dy^+}{dt} - \frac{dy^-}{dt} = x^+ - x^- = x.$$

By integrating both sides, we obtain the solution $y(t) = \int_0^t x(s)ds + y_0$.

Other primitives we implement in REACTAMOLE include the following.

- `neg :: CRN Double Double` performs numerical negation by reversing the roles of X^+ and X^- . Note that this primitive does not generate additional reactions. Instead, it simply *reinterprets* the output species of the CRN.
- `id :: CRN a a`, the identity signal function, produces its inputs as outputs without modification.
- `proj1 :: CRN (a, b) a` and `proj2 :: CRN (a, b) b` take *pairs* of signals as inputs and project out the individual components of those pairs as output.

10:6 Reactamole



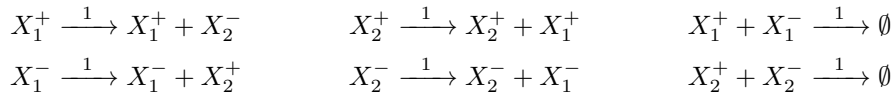
■ **Figure 1** Visual representation of the arrowized `sin` implementation.

- `dup :: CRN a (a, a)` produces a pair where each component is a “copy” of the input. However, in reality, we don’t make a copy of the input species. Multiple consuming CRNs can use the components of the resulting pair because I/O CRNs are catalytic in their inputs.
- `loop :: CRN (a, c) (b, c) -> CRN a b` creates a *feedback* loop where the second component of the output of the CRN is given to itself as input.

Using these primitives, it is already possible to specify complex signals such as sine and cosine. We can define `sin` in the following way which is also illustrated in Figure 1:

```
sin :: CRN a Double
sin = loop (proj2 >>> neg >>> integrate 1 >>> integrate 0 >>> dup)
```

This definition exploits the fact that the sine function satisfies the second-order differential equation $x''(t) = -x(t)$. Since the output of `sin` does not depend on its input, the type signature of `sin` has an abstract input type `a`, meaning that it can receive any input. The `loop` combinator creates a feedback loop that allows the signal $x(t)$ to depend on itself. Since `sin` uses two applications of `integrate`, REACTAMOLE generates the following six reactions:



We can verify the correctness of the `sin` definition by observing that the ODEs associated with the reactions satisfy

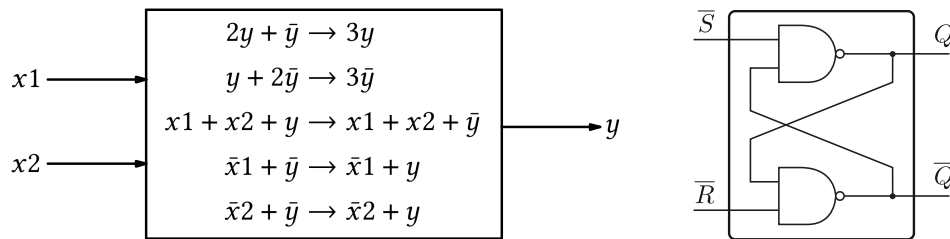
$$\frac{dx_1}{dt} = \frac{dx_1^+}{dt} - \frac{dx_1^-}{dt} = x_2^+ - x_2^- = x_2 \qquad \frac{dx_2}{dt} = \frac{dx_2^+}{dt} - \frac{dx_2^-}{dt} = x_1^- - x_1^+ = -x_1$$

and therefore $x_1(t) = \sin(t)$ and $x_2(t) = \cos(t)$.

We now turn our attention to I/O CRNs that produce Booleans. Similar to real numbers, we encode a Boolean signal using a pair of species (X, \bar{X}) while maintaining the invariant that $x(t) + \bar{x}(t) = 1$. When species X is high, the Boolean is interpreted as *true*; similarly, when \bar{X} is high, the Boolean is interpreted as *false*.

The elementary Boolean signal functions in REACTAMOLE are `not :: CRN Bool Bool` and `nand :: CRN (Bool, Bool) Bool`. Similar to `neg`, the `not` signal function “crosses the wires” of X and \bar{X} without adding any additional species or reactions. The `nand` signal function consists of five reactions, using the implementation provided by Ellis, Klinge, and Lathrop [7] and visualized in Figure 2.

Using these two primitives, we can define the other elementary gates as follows.



■ **Figure 2** REACTAMOLE `nand` and `srLatch` implementations.

```

or, and, nor, xor, xnor :: CRN (Bool, Bool) Bool
or = (not *** not) >>> nand
and = nand >>> not
nor = or >>> not
xor = (nand &&& or) >>> and
xnor = xor >>> not

```

Note that `not` does not introduce additional overhead, so the signal functions `or`, `and` and `nor` are no more complex than `nand`.

We can also use the `loop` combinator to create sequential logic gates such as a *set-reset latch* which is visualized in Figure 2.

```

srLatch :: CRN (Bool, Bool) (Bool, Bool)
srLatch = loop (crossWires >>> (nand *** nand) >>> dup)

```

Here, `crossWires` is a simple signal function that “rearranges the wires” so that the outputs are looped back into the appropriate `nand` gates. Since `srLatch` has two Boolean inputs and is implemented with two `nand` gates, the resulting I/O CRN consists of eight species and nine reactions¹.

Finally, we can create signal functions that employ both real-valued signals and Boolean signals. For example, we include a primitive `isPos :: CRN Double Bool` that tests if a real-valued input is positive. Note that `isPos` is a continuous approximation of a discontinuous function, so its Boolean output is undefined if the input signal is close to zero.

Using `isPos` and the previously defined `sin` signal, we can easily create a `clock` signal that could be employed in clocked sequential circuits.

```

clock :: CRN a Bool
clock = sin >>> isPos

```

¹ REACTAMOLE automatically optimizes the final I/O CRN by combining reactions together when possible. Thus, `srLatch` consists of nine reactions instead of ten because two reactions can be safely combined into one without affecting the underlying ODEs of the CRN.

```

-- Algebraic structures
(>>>) :: CRN a b -> CRN b c -> CRN a c
(***) :: CRN a b -> CRN c d -> CRN (a, c) (b, d)
(&&&) :: CRN a b -> CRN a c -> CRN a (b, c)
first  :: CRN a b -> CRN (a, c) (b, c)
second :: CRN a b -> CRN (c, a) (c, b)
-- Booleans
not    :: CRN Bool Bool
nand   :: CRN (Bool, Bool) Bool
arr1B1 :: (Bool -> Bool) -> CRN Bool Bool
-- Reals
integrate :: Double -> CRN Double Double
neg        :: CRN Double Double
add        :: CRN (Double, Double) Double
mult       :: CRN (Double, Double) Double
isPos      :: CRN Double Bool

-- Switching
(+++) :: CRN a b -> CRN c d
      -> CRN (Either a c)
      (Either b d)
(|||)  :: CRN a c -> CRN b c
      -> CRN (Either a b) c
left  :: CRN a b
      -> CRN (Either a c)
      (Either b c)
right :: CRN a b
      -> CRN (Either c a)
      (Either c b)
entangle :: CRN (Bool, (a,b))
        (Either a b)

```

■ **Figure 3** Functional reactive molecular programming core combinators.

4 Functional Reactive Molecular Programming

The following equivalences form the heart of the functional reactive molecular programming (FRMP) style epitomized by REACTAMOLE:

- Signals, time-varying values, are *interpretations* of collections of species' concentrations as values.
- Signal functions are *typed chemical reaction networks*, a CRN equipped with extra information identifying the species that serve as the inputs and outputs to the function.
- Higher-order signal functions, i.e., signal functions that take other signal functions as input, are *CRN transformers* which produce new CRNs from old ones.

Furthermore, we take inspiration from the arrowized FRP approach of Hughes and thus specify the constructs of FRMP as a collection of *combinators*, higher-order signal functions that users combine to build more complex CRNs from smaller ones.

The heart of FRMP is a compilation pass that transforms these FRP combinators into a CRN that realizes the computation in (abstract) chemistry. We augment the output CRN with *species tags*, additional static information necessary for interpreting the relevant species of the CRN as inputs and outputs to the computation. We call the combination of a CRN with its species tags a *typed chemical reaction network*, formally, a tuple of a CRN N and species tags describing how to interpret its inputs and outputs, written $(N, p^{\text{in}}, p^{\text{out}})$. We represent the compilation process as an interpretation function over signal functions $\llbracket f \rrbracket = (N, p^{\text{in}}, p^{\text{out}})$ which denotes that signal function f compiles to typed CRN $(N, p^{\text{in}}, p^{\text{out}})$.

In the following section, we describe this translation for our core combinators. We organize the various combinators by the types that they operate over, recalling that the type $\text{CRN } a \ b$ represents a CRN (dually, a signal function) that takes a signal of type a as input and produces a signal of type b as output. Figure 3 gives an overview of these combinators by category.²

² We approximate real values using finite-sized Haskell `Double` values.

Species Tags and Signals

We provide one tag for each possible type that we can represent in our implementation. The tags identify the type of signal that the CRN outputs as well as the relevant species that encode that signal.

- The *unit signal tag*, $()$, represents a signal that generates the *unit value*. This is an arbitrary, unique value of that type (written as $()$ in Haskell) that acts effectively as a constant that carries no information. Because of this, a unit signal will not be acted on by any reactions and thus does not need an explicit runtime representation in a CRN.
- A *Boolean signal tag*, (X, \bar{X}) , denotes a Boolean value represented by a pair of species X and \bar{X} in a *dual-rail construction* based on Ellis et al.’s method [7]. These species exhibit an inverse relationship maintained by the constructed CRN as an invariant – when one species has a high concentration, the other has a low concentration. The two species in this dual-rail construction represent **True** and **False**, respectively.
- A *real number tag*, (X^+, X^-) , denotes a real value represented by a pair of molecules X^+ and X^- , where we take the value of the real to be the *difference* between the concentrations of X^+ and X^- .
- A *pair signal tag*, (p_1, p_2) , denotes the “gluing” together of two signals that operate independently of each other. The components of the tag pair are the tags of the individual signals.
- An *either signal tag*, (X, \bar{X}, p_1, p_2) , represents a time-varying discriminated union used in FRMP to achieve dynamic switching. Such a signal is made up of a Boolean signal indicated by species X and \bar{X} as well as two component signals with tags p_1 and p_2 . The Boolean species indicates which of the two signals is currently active.

4.1 Algebraic Structures

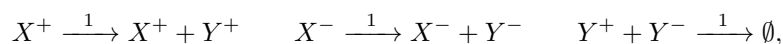
The core of FRP is composing signal functions together. Suppose that we compose together two signal functions, written $f_1 \ggg f_2$. In FRMP, this amounts to composing the two I/O CRNs representing f_1 and f_2 , call them N_1 and N_2 , respectively, by feeding the outputs of N_1 as inputs to N_2 . Because the input species of I/O CRNs are catalytic, i.e., the net rate of the input species to N_1 and N_2 is zero, we achieve composition by simply taking the union of the species and reactions of each CRN (\sqcup) and then substituting all the input species of f_2 with all the output species of f_1 in the resulting CRN, written $[p_1^{\text{out}} \mapsto p_2^{\text{in}}]$.

$$\llbracket f_1 \ggg f_2 \rrbracket = ([p_1^{\text{out}} \mapsto p_2^{\text{in}}](N_1 \sqcup N_2), p_1^{\text{in}}, p_2^{\text{out}}) \quad \text{where} \quad \begin{aligned} \llbracket f_1 \rrbracket &= (N_1, p_1^{\text{in}}, p_1^{\text{out}}) \\ \llbracket f_2 \rrbracket &= (N_2, p_1^{\text{in}}, p_2^{\text{out}}). \end{aligned}$$

This is safe because N_2 has no observable effect on the outputs of N_1 .

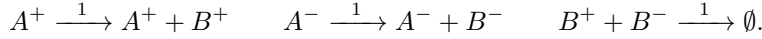
The only caveat we must consider is that the species of the two CRNs are disjoint. Otherwise, unioning their reactions might cause unintended side-effects to their rates. We guarantee this by ensuring that species names are disjoint between CRNs via renaming whenever combining CRNs in this fashion. This is analogous to the notion of α -equivalence in programming languages, where programs are considered equivalent up to renaming of their bound variables.

As a simple example of composition, consider $f_1 = (N_1, (X^+, X^-), (Y^+, Y^-))$ and $f_2 = (N_2, (A^+, A^-), (B^+, B^-))$ where N_1 is the network:

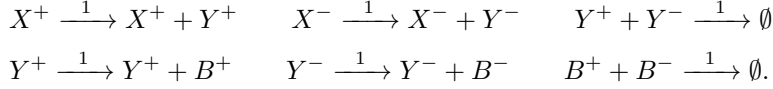


and N_2 is the network:

10:10 Reactamole



N_1 and N_2 each independently perform an integration of their arguments. The composition of these two CRNs $f_1 \ggg f_2 = (N_3, (X^+, X^-), (B^+, B^-))$ where N_3 is the network:



N_3 is precisely the union of the reactions of N_1 and N_2 but with the input species of N_2 , A^+ and A^- replaced by the output species of N_1 , Y^+ and Y^- . Observe that, by virtue of composing two integration functions together, N_3 performs two integrations on its input.

Aggregate data types are represented in REACTAMOLE through *product types*, i.e., tuples. The elements of tuples are, by definition, independent values. Because we maintain disjointness of species names between CRNs, we form tuples by taking the union of these CRNs directly. The split operator between two CRNs, written $f_1 \mathbf{***} f_2$, unions the species and reactions of the two input CRNs together and creates a new output species tag that identifies the output of the CRN as a tuple.

$$\llbracket f_1 \mathbf{***} f_2 \rrbracket = (N_1 \sqcup N_2, (p_1^{\text{in}}, p_2^{\text{in}}), (p_1^{\text{out}}, p_2^{\text{out}})) \quad \text{where } \begin{array}{l} \llbracket f_1 \rrbracket = (N_1, p_1^{\text{in}}, p_1^{\text{out}}) \\ \llbracket f_2 \rrbracket = (N_2, p_2^{\text{in}}, p_2^{\text{out}}). \end{array}$$

For our example, $\llbracket f_1 \mathbf{***} f_2 \rrbracket = (N_4, ((X^+, X^-), (A^+, A^-)), ((Y^+, Y^-), (B^+, B^-)))$ where N_4 is simply the union of the unmodified reactions from N_1 and N_2 .

In contrast, the *fanout* operator between two CRNs, written $f_1 \mathbf{\&\&\&} f_2$, transforms two CRNs that expect the same input type into a single CRN that sends a single input to the two CRNs. The operator has type $(\mathbf{\&\&\&}) :: \text{CRN } a \ b \ \rightarrow \ \text{CRN } a \ c \ \rightarrow \ \text{CRN } a \ (b, c)$. We can implement the fanout operator in terms of composition, split, and a **dup** combinator that produces a signal pair where each component is simply the input species.

$$\llbracket \text{dup } f \rrbracket = (N, p^{\text{in}}, (p^{\text{out}}, p^{\text{out}})) \quad \text{where } \llbracket f \rrbracket = (N, p^{\text{in}}, p^{\text{out}}).$$

We can then define fanout directly as $f \mathbf{\&\&\&} g = \text{dup } \ggg (f \mathbf{***} g)$. Again, this is all safe because CRNs are catalytic in their inputs. Thus, the duplication of the input signal will not lead to interfering behavior between f and g .

Finally, we can also create CRNs that involve *feedback*, i.e., the CRN depends on its own output, with the **loop** combinator. To understand how **loop** is implemented, it is useful to first analyze its type:

```
loop :: CRN (a, c) (b, c) -> CRN a b
```

loop takes in a CRN that expects a pair signal and produces a pair signal. The result is a new CRN where the second component of the input pair is “patched” by the second component of the output pair, both of type c . This leaves behind a CRN that expects an a as input and produces a b as output.

$$\llbracket \text{loop } f \rrbracket = ([p_c^{\text{out}} \mapsto p_c^{\text{in}}]N, p_a^{\text{in}}, p_b^{\text{out}}) \quad \text{where } \llbracket f \rrbracket = (N, (p_a^{\text{in}}, p_c^{\text{in}}), (p_b^{\text{out}}, p_c^{\text{out}}))$$

4.2 Booleans

Booleans in FRMP are represented by a dual-rail encoding, where one species indicates the **True** value and the other **False**. These species exhibit a strictly inverse relationship (i.e., when one has a low concentration the other has a high concentration) and the value of the Boolean is given by the high species. This dual-rail construction is based on Ellis et al.'s method [7] and is necessary for the result to be measurable by reporter molecules in a biological system, since it is difficult to detect the absence of a species.

Boolean Negation

FRMP supports negation through a reinterpretation of this Boolean encoding. Negation is achieved by swapping the Boolean value that each dual-rail chemical species represents. This allows for efficient negation in a manner that does not require creating an additional CRN.

$$\llbracket \text{not } b \rrbracket = (N, p^{\text{in}}, (\bar{X}, X)) \quad \text{where } \llbracket b \rrbracket = (N, p^{\text{in}}, (X, \bar{X}))$$

NAND and Other Logic Gates

The foundation for FRMP's support of Boolean circuits and functions is the robust NAND gate introduced by Ellis et al. [7], as shown in Figure 2. This gate is included as the signal function `nand :: CRN (Bool, Bool) Bool`. All other basic logic gates are implemented using `nand` along with `neg` as described in Section 3.

Lifting

We can also lift (i.e., translate) arbitrary pure Boolean functions into CRNs, which allows us to write Boolean functions in a more natural style, e.g., with variables and conditionals. This is possible because any Boolean function can be represented by a finite number of gates. For example, consider the following Haskell function which determines if a decision between three parties is unanimous:

```
unanimous :: Bool -> Bool -> Bool -> Bool
unanimous x y z = if x then (y && z) else not (y || z)
```

We can lift this function into a CRN using the three-input lifting function:³

```
arr3B1 :: (Bool -> Bool -> Bool -> Bool) -> CRN (Bool, Bool, Bool) Bool
```

`arr3B1 unanimous` produces a CRN that computes `unanimous`. This CRN is constructed using a series of signal function gates that mirror the given Haskell function's sum of products. The sum of products is found by first constructing a matrix of all possible combinations of inputs to the Haskell function and the resulting outputs. All instances with output value **False** are then filtered out, and the signal function is then constructed using the corresponding series of AND, OR, and NOT gates that represent the sum of products given by the matrix. This lifting process is a useful mechanism to quickly create a complex CRN directly from Haskell code.

³ The name `arr` comes from the Haskell `Arrow` typeclass that defines `arr` as its own lifting function.

4.3 Real Numbers

We have already demonstrated our ability to compute over real numbers with CRNs in Section 3. For example, the `integrate` and `sin` signal functions correspond to straightforward CRN implementations. The `neg` combinator is a simple reinterpretation of the input CRN's species tag because $X - Y = -(Y - X)$:

$$\llbracket \text{neg } f \rrbracket = (N, p^{\text{in}}, (X^-, X^+)) \quad \text{where } \llbracket f \rrbracket = (N, p^{\text{in}}, (X^+, X^-))$$

What about other operations, for example, adding together two real signals? It turns out that we cannot craft a CRN that captures the addition of two abstract real signals. But, in our deterministic context, we know that a CRN can be interpreted as an ODE describing the rates of change of each of the species in the system. Under this interpretation, addition of real signals is precisely addition of their respective ODEs.

However, this implementation strategy is fundamentally different than the other CRN operations we have discussed so far. Unlike `integrate`, which treats its input signal as a black box, performing addition and multiplication requires knowing the input signal's ODEs in order to be exact. To prevent unnecessary dependencies, the `add :: CRN (Double, Double) Double` and `mult :: CRN (Double, Double) Double` signal functions in REACTAMOLE are implemented *lazily*. This means that the creation of new species and reactions is delayed until absolutely necessary. In fact, when adding or multiplying real-valued signals, the intermediate sum or product species may be optimized away entirely. For example, in Section 5, we construct a CRN using one application of `add`, two applications of `mult`, and one application of `integrate`, which only generates one pair of species and five reactions. (See Figure 4 for more details.)

Currently, our approach to FRMP does not support lifting real-valued functions to CRNs. However, a large class of functions are known to be produced by analog computers [3], and new techniques are emerging for lifting various real-valued functions to CRNs [12]. We hope to incorporate these ideas into FRMP in the future.

4.4 Switching

An important capability of a functional reactive program is changing the topology of its components at runtime, i.e., *dynamically switching* between different signals. A simple way to encode the behavior is through a conditional, `if t then s1 else s2`, where s_1 and s_2 are arbitrary signals and t is a Boolean signal. As t transitions between truth values, the overall *conditional* signal transitions between s_1 and s_2 . From Subsection 4.1, we know that we can carry t , s_1 , and s_2 as a triple of signals. We give this aggregate signal the type `Either a b`, where `a` and `b` are the types of s_1 and s_2 , respectively. This choice of type comes from the `Either` type in Haskell which encodes a *discriminated union* – a pair of potential values with a *tag* that says which of the two values are present.

However, because the reactions of a CRN are fixed at compilation time, we don't have a built-in mechanism by which a consuming signal can “change” its reactions to go from consuming s_1 to consuming s_2 during runtime. To solve this problem, we then *entangle* the components of the `Either` signal, t , s_1 , and s_2 , to produce a single signal with our desired conditional semantics. The `entangle :: CRN (Bool, (a,b)) (Either a b)` signal function creates these entanglements and the *fanin* operator creates a signal function that merges two entangled values into one. The fanin operator has the following type signature:

```
(|||) :: CRN a c -> CRN b c -> CRN (Either a b) c
```

`f1 ||| f2` takes two signal functions that take arbitrary types `a` and `b` as input and produce a common output type `c`. Fanin will join the signal functions f_1 and f_2 into an **Either** signal and then merge their outputs to produce a unified signal of type `c`. The Boolean component of the **Either** signal then controls whether the output signal is generated from f_1 or f_2 .

This is necessarily a type-directed process; how we combine s_1 and s_2 depends on their encodings and thus their types. For example, consider the case where s_1 and s_2 are both Boolean signals. We can now combine our **Either** signal by observing that:

$$\text{if } t \text{ then } s_1 \text{ else } s_2 \equiv (t \wedge s_1) \vee (\bar{t} \wedge s_2).$$

We can also entangle **Either** signals when s_1 and s_2 are real signals. To do so, we observe a similar equation for reals, recalling that the Boolean signal t is really a pair of species, b and \bar{b} , interpreted in a dual-rail style:

$$\text{if } (b, \bar{b}) \text{ then } s_1 \text{ else } s_2 \approx b \cdot s_1 + \bar{b} \cdot s_2.$$

Note that because b and \bar{b} only approximate 0 and 1, the resulting signal is an approximation of the appropriate output of either s_1 or s_2 . Such an approximation is necessary because a CRN cannot have a discontinuity in its solution. Finally, with primitives defined, we can entangle aggregate signals such as pairs by simply entangling their components and then pairing together the resulting signals.

Below is a definition of a *rectify*, a signal function that passes through only the positive component of a signal.

```
rectify :: CRN Double Double
rectify = isPos &&& dup >>> entangle >>> (id ||| constR1 0)
```

Here, the sub-expression `id ||| constR1 0` defines a signal function that is either the identity function or the constant zero. Thus, `rectify` is a signal function that behaves like the identity function if the input is positive and otherwise behaves like the constant zero.

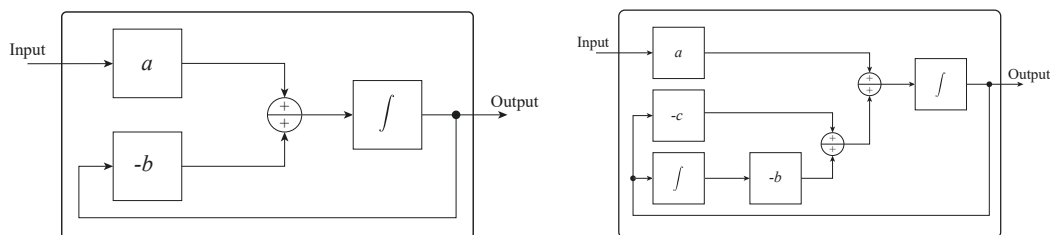
The Reactamole Implementation

REACTAMOLE is an implementation of this approach to functional reactive molecular programming as an *embedded* domain-specific language (eDSL) within Haskell.⁴ By embedding REACTAMOLE in Haskell, we can take direct advantage of Haskell's existing facilities – language constructs, the type system, and its tooling support – in developing CRNs. To maximize the benefits of this embedding, we choose a different, yet equivalent, adaption of FRP into CRNs. We define signal functions to be *genuine* Haskell functions between signal values, i.e., `data SF a b = SF (Signal a -> Signal b)`, where the `Signal` type corresponds to a CRN. This change in encoding allows us to use Haskell's rich higher-order programming facilities to be able to express combinators more concisely. For example, consider the composition of two CRNs with the composition operator `f1 >>> f2`. In REACTAMOLE, because `f1` and `f2` are now Haskell functions, the composition operator between CRNs is simply the composition operator between Haskell functions, `(.)`.

```
(>>>) :: SF a b -> SF b c -> SF a c
(SF f) >>> (SF g) = SF (g . f)    -- N.B., (.) composes right-to-left
```

Other combinators enjoy simpler implementations than what was presented in this section due to the embedded nature of REACTAMOLE.

⁴ REACTAMOLE is available at <https://github.com/digMP/haskell-reactamole>.



■ **Figure 4** Visualization of the low-pass (left) and band-pass (right) filters in REACTAMOLE. The boxes containing constants a , b , and c correspond to constant multiplier signal functions.

5 Case Study: Amplitude Modulation

We now demonstrate REACTAMOLE's expressiveness via a case study: implementing chemical reaction networks to perform amplitude modulation [19]. Amplitude modulation (AM) is a common technique for sending multiple signals through a shared medium. Intuitively, an AM modulator combines a signal $u(t)$ with a sinusoidal *carrier signal* $s(t)$ via multiplication. Many signals $u_1(t), u_2(t), \dots$ can then be simultaneously transmitted through a shared medium $m(t)$ by superimposing the modulated signals. In CRNs, modulated signals need only be added into a single signal, represented by a pair of species (M^+, M^-). This is analogous to radio stations transmitting modulated signals at specified frequencies, which all combine in the atmosphere.

We previously saw how to specify a sine wave in REACTAMOLE. It is not difficult to extend this example in order to generate a carrier signal with a given frequency. We first define a helper signal function called `constMult` that multiplies a signal by a constant.

```
constMult :: Double -> CRN Double Double
constMult d = (constR1 d &&& id) >>> mult
```

Here, `constR1 d` produces a signal function `CRN a Double` that ignores its input and emits a signal with the constant `d`. Using `constMult`, we can now specify a CRN that generates a sine wave with a given frequency.

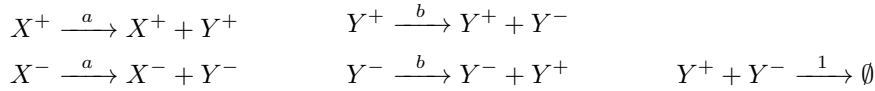
```
carrier :: Double -> CRN a Double
carrier w = loop (proj2 >>> constMult (-w) >>> integrate 1
                >>> constMult w >>> integrate 0
                >>> dup)
```

Note that this implementation is nearly identical to that of `sin`, defined in Section 3. However, by adding the constant multipliers, `carrier 5` will produce a signal $s(t) = \sin(5t)$. The constant `w` is incorporated into the rate constants of the reactions, so `carrier w` also consists of four species and six reactions.

One common component used to implement AM modulation and demodulation is the *low-pass filter*. A simple first-order low-pass filter is realized by integrating the sum of the input and output multiplied by specific parameters a and b , as shown in Figure 4. By choosing the appropriate parameters, we can generate a low-pass filter with a specific cut-off frequency. For example, if we choose a to be 0.0001, then the cut-off frequency will be 0.01 radians per second. The low-pass filter presented in [19] can be specified as follows:

```
lowPass :: Double -> Double -> CRN Double Double
lowPass a b = loop (constMult a *** constMult (-b) >>> add >>> integrate 0 >>> dup)
```

The I/O CRN generated by `lowPass a b` consists of the following five reactions:

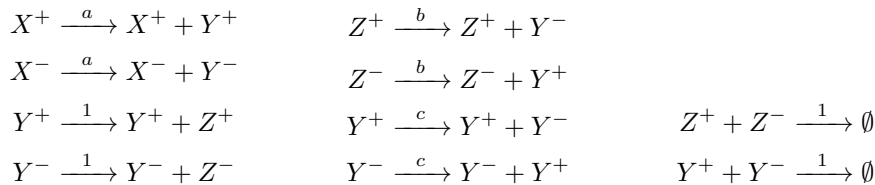


where (X^+, X^-) comprise the input signal. The CRN satisfies the ODE $\frac{dy}{dt} = ax - by$ where $y(t) = y^+(t) - y^-(t)$ and $x(t) = x^+(t) - x^-(t)$.

Another common AM component is the *band-pass filter*, which can be used to select a specific carrier frequency from the medium species and attenuate all other carrier signals present in the medium species. Below is the implementation of the band-pass filter presented in [19], which is also included visually in Figure 4.

```
bandPass :: Double -> Double -> Double -> CRN Double Double
bandPass a b c = loop (first (constMult a)
  >>> second (constMult (-c) &&& (integrate 0 >>> constMult (-b)) >>> add)
  >>> add >>> integrate 0 >>> dup)
```

The I/O CRN generated by `bandPass a b c` consists of the reactions



and satisfies the ODEs $\frac{dy}{dt} = ax - bz - cy$ and $\frac{dz}{dt} = y$. Note that the species (Z^+, Z^-) are internal species to the I/O CRN that are not communicated in its output.

We now show how to modulate and demodulate signals using a carrier frequency. We use the technique in [19] that has the medium species M^+ and M^- and can be specified in REACTAMOLE with:

```
modulate :: Double -> CRN Double Double
modulate w = loop (first (id &&& carrier w >>> mult)
  >>> second neg >>> add >>> integrate 0 >>> dup)
```

Here, `modulate f` produces a pair of species representing the medium $m(t)$ that satisfies $\frac{dm}{dt} = u(t) \cdot \sin(ft) - m(t)$ where $u(t)$ is the input signal. Klinge and Lathrop [19] also proposed a method to superimpose multiple modulated signals through a single medium $m(t)$, which can be easily accomplished in REACTAMOLE with `add :: CRN Double Double`.

Given a signal $m(t)$ that may carry several modulated signals, we now use the same methods in [19] to retrieve a signal. A simple AM demodulator is realized with a band-pass filter to select the desired carrier frequency followed by a function to pass only the positive parts of the signal. A low-pass filter may then be used to remove the carrier frequency to recover an approximation to the original signal.

We can use the `bandPass` and `lowPass` filters, along with `rectify` defined in Subsection 4.4, to extract a signal from a desired carrier frequency.

```
demodulate :: Double -> Double -> CRN Double Double
demodulate w q = bandPass (w/q) (w/q) (w*w) >>> rectify >>> lowPass w w
```

Here, `demodulate w q` generates a CRN that demodulates a signal that has been modulated on a carrier signal at frequency w . The parameter q is used to determine the bandwidth of the band-pass filter, which determines how close two different carriers may be in frequency. Also note that low-pass and band-pass filters may be cascaded to create higher order filters by composing them with the `>>>` combinator.

6 Conclusion

REACTAMOLE unveils new possibilities for the future of molecular programming through exploration of a novel paradigm for the field: functional reactive programming. The language uses typed CRNs – CRNs with extra information about how their chemical species map to Haskell types – to enable complex computation. In particular, REACTAMOLE introduces combinators that allow for computation over basic primitive types as well as the safe manipulation and composition of CRNs.

The representation of CRNs as signal functions in FRP allows for efficient construction of a variety of CRNs using a minimal numbers of species. For example, the NOT gate for Booleans and negation for real values are both achieved through “rewiring” of a signal function (reinterpreting the species’ types) and do not require any additional chemical species. This results in OR and AND gates that use the same number of species as the NAND gate they are built from. Additionally, the use of ODEs to represent CRNs enables addition for some CRNs. We also provide a formal construction of the representations of various Haskell types as CRNs, including Booleans, Reals, Pairs, and Eithers. All of these features make REACTAMOLE a useful tool for safe, robust and automated construction and composition of CRNs for a variety of uses.

Future Work

There are four main areas for further improvement that we hope to pursue for REACTAMOLE. First, there is potential for expanding REACTAMOLE’s lifting support, including optimizing the construction of lifted Boolean functions, enabling lift for multi-output Boolean functions, and adding support for lifting certain real functions using **Either**. Lifting for reals will require some constraints, as it is not possible to lift real functions whose input signals are not known.

We would also like to improve the handling of approximations and delays. Currently, **nand** and **Either** are approximations because there is some unavoidable delay when working with chemical reactions. In the future, it would be helpful to build out support for tracking approximation margins and specifying additional guarantees. Similarly, composing NAND gates propagates delay and this is currently not controllable by the user because the rate constants are hard-coded. Ellis et al. specify a method for converting a τ value to a rate constant [7] which could be leveraged to allow the user to specify a rate constant for the NAND gates in REACTAMOLE.

It would also be useful to expand the back-end options for REACTAMOLE. Currently, the language interacts directly with MATLAB to simulate CRNs, but this could be expanded by allowing options to export to other tools, such as SimBiology, or through building an embedded ODE solver.

Finally, REACTAMOLE may be a good candidate for a standalone language. In particular, we can move beyond the limitations of Haskell and specialize the language features to the molecular programming setting. With this change, we can consider adding a strong, linear temporal logic-based type system to REACTAMOLE to capture fine-grained correctness properties of our CRNs [18]. Such a type system can, in turn, enable the efficient automatic generation of CRNs from specification, i.e., program synthesis [11].

References

- 1 Stefan Badelt, Seung Woo Shin, Robert F. Johnson, Qing Dong, Chris Thachuk, and Erik Winfree. A General-Purpose CRN-to-DSD Compiler with Formal Verification, Optimization, and Simulation Capabilities. In Robert Brijder and Lulu Qian, editors, *DNA Computing and Molecular Programming*, pages 232–248, Cham, 2017. Springer International Publishing.
- 2 Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A Survey on Reactive Programming. *ACM Comput. Surv.*, 45(4), 2013. doi:10.1145/2501654.2501666.
- 3 Olivier Bournez, Daniel Graça, and Amaury Pouly. On the functions generated by the general purpose analog computer. *Information and Computation*, 257:34–57, 2017. doi:10.1016/j.ic.2017.09.015.
- 4 Luca Cardelli. Kaemika App: Integrating Protocols and Chemical Simulation. In Alessandro Abate, Tatjana Petrov, and Verena Wolf, editors, *Computational Methods in Systems Biology*, pages 373–379, Cham, 2020. Springer International Publishing.
- 5 Matthew Cook, David Soloveichik, Erik Winfree, and Jehoshua Bruck. Programmability of Chemical Reaction Networks. In Anne Condon, David Harel, Joost N. Kok, Arto Salomaa, and Erik Winfree, editors, *Algorithmic Bioprocesses*, pages 543–584. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-540-88869-7_27.
- 6 Conal M. Elliott. Push-Pull Functional Reactive Programming. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 25–36, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1596638.1596643.
- 7 Samuel J. Ellis, Titus H. Klinge, and James I. Lathrop. Robust chemical circuits. *Biosystems*, 186:103983, 2019. doi:10.1016/j.biosystems.2019.103983.
- 8 Irving Robert Epstein and John Anthony Pojman. *An Introduction to Nonlinear Chemical Dynamics: Oscillations, Waves, Patterns, and Chaos*. Oxford University Press, 1998.
- 9 François Fages, Guillaume Le Guludec, Olivier Bournez, and Amaury Pouly. Strong turing completeness of continuous chemical reaction networks and compilation of mixed analog-digital programs. In Jérôme Feret and Heinz Koeppl, editors, *Computational Methods in Systems Biology*, pages 108–127, Cham, 2017. Springer International Publishing.
- 10 Martin Feinberg. *Foundations of chemical reaction network theory*. Springer, 2019.
- 11 Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. Synthesizing Functional Reactive Programs. *arXiv:1905.09825 [cs]*, 2019. arXiv:1905.09825.
- 12 Mathieu Hemery, François Fages, and Sylvain Soliman. Compiling Elementary Mathematical Functions into Finite Chemical Reaction Networks via a Polynomialization Algorithm for ODEs. working paper or preprint, 2021. URL: <https://hal.inria.fr/hal-03220725>.
- 13 Xiang Huang, Titus H. Klinge, and James I. Lathrop. Real-time equivalence of chemical reaction networks and analog computers. In Chris Thachuk and Yan Liu, editors, *DNA Computing and Molecular Programming*, pages 37–53, Cham, 2019. Springer International Publishing.
- 14 P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming*, 2002.
- 15 Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196–es, 1996. doi:10.1145/242224.242477.
- 16 John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1):67–111, 2000. doi:10.1016/S0167-6423(99)00023-4.
- 17 John Hughes and John O'Donnell. Expressing and reasoning about non-deterministic functional programs. In Kei Davis and John Hughes, editors, *Functional Programming*, pages 308–328, London, 1990. Springer London.
- 18 Alan Jeffrey. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the sixth workshop on Programming languages meets program verification - PLPV '12*, page 49, Philadelphia, Pennsylvania, USA, 2012. ACM Press. doi:10.1145/2103776.2103783.

- 19 Titus H. Klinge and James I. Lathrop. Modulated signals in chemical reaction networks. *CoRR*, abs/2009.06703, 2020. [arXiv:2009.06703](#).
- 20 Titus H. Klinge, James I. Lathrop, and Jack H. Lutz. Robust Biomolecular Finite Automata. *CoRR*, abs/1505.03931, 2015. [arXiv:1505.03931](#).
- 21 Simon Marlow. Haskell 2010 language report.
- 22 Marko Vasic, David Soloveichik, and Sarfraz Khurshid. CRN++: Molecular Programming Language. *arXiv:1809.07430 [cs]*, 11145, 2018. [doi:10.1007/978-3-030-00030-1](#).

A Haskell

Haskell is a statically-typed, lazy, pure functional programming language [21] used both in industry for its strong correctness guarantees and in academia as a testbed for programming languages research. REACTAMOLE is an embedded domain-specific language [15] within Haskell and as such, writing programs in REACTAMOLE is equivalent to writing Haskell programs. Through this embedding, REACTAMOLE enjoys the benefits of Haskell’s strong static type system to ensure the well-formedness of chemical reaction networks. Here we review the salient features of Haskell necessary to understand our presentation of REACTAMOLE.

A.1 Bindings and Type Signatures

Like other languages, a Haskell program is composed of a collection of top-level bindings, usually function declarations. For example, here is a function that computes a simple numeric result:

```
cToF :: Double -> Double
cToF c = c * 9.0/5.0 + 32
```

A binding typically possesses a *type signature* associated with it that describes the static type of the value bound to that name. In this example, the binding `cToF` has a *function type* `Double -> Double`. The type of value that `cToF` takes as input is given to the left of the arrow (`->`) and the type of its output is given to the right of the arrow. Here, `cToF` is a function that takes a `Double` as input and produces a `Double` as output. If a function takes multiple inputs, we separate each input type by an arrow. For example, the less-than comparison operator has the (simplified) type `(<) :: Int -> Int -> Bool`. In other words, less than is a function that takes two arguments, both `Ints`, and produces a `Bool` as output.

The *definition* of the binding follows the type signature. Above, we declare `cToF` as a function, specifying the parameter to the function `c` after the binding’s name but before the equals sign. The function body is a single expression (`c * 9.0/5.0 + 32`) whose resulting value is returned when the function is evaluated.

A.2 Polymorphism

Haskell’s rich type system allows the programmer to write down functions that take values of *any* type, a feature called *parametric polymorphism*. This is commonly implemented as *generic types* in other languages, such as Java or C#. In Haskell, any name that appears in a type signature that *starts with a lowercase letter* is assumed to be a type variable. For example, this binding:

```
id :: a -> a
id x = x
```

implements the identity function that takes a value of some *arbitrary* type `a` and returns a value of that same type. `id` can be passed any value and the type variable `a` is instantiated to the type of that value, e.g., `id 5` instantiates the `a` to be `Int`.

Type variables also commonly appear as *arguments to other types*. The canonical example of such a parameterized type is the list type, written `[a]`, which corresponds to values that are lists whose elements are all of type `a`. More commonly, you will see parameterized types like the standard data type `Maybe a`, which is, conceptually, a box that *potentially* contains a value of type `a`.

A.3 Functions as Values

Finally, an important concept when working in the abstract setting of functional reactive programming is *functions as first-class values*. Functions can be passed as input to other functions and produced as output. An example of such a *higher-order function* is the composition operator `(.)` which operates analogously to mathematical function composition. We can use composition to “glue” together functions, sending the outputs of one as the inputs of another. For example, here is a simple function that defines `isOdd` by composing an even-testing function and Boolean negation:

```
isEven :: Int -> Bool
isEven x = x `div` 2 == 0

isOdd :: Int -> Bool
isOdd x = (not . isEven) x
```

Note that the output of `(.)` here will be a function that takes an `Int`, feeds it through `isEven`, then takes that resulting `Bool` and feeds it through `not`. However, because this result is a function, it is superfluous to write the argument `x` in the definition of `isOdd` because all we do is pass it to the result of the composition. Instead, we can define `isOdd` *directly* in terms of the composition:

```
isOdd' :: Int -> Bool
isOdd' = not . isEven
```

When a function is defined directly without the need to reference its argument, we call such a function *point-free*. Much of the code we write in REACTAMOLE consists of manipulating polymorphic function-like objects in this higher-order, point-free style.

A.4 Example: Interpreting Functional Reactive Code

As an example of applying these concepts toward understanding REACTAMOLE code, let us revisit the `sin` function from Section 3 and analyze how its type relate to its implementation.

```
sin :: CRN a Double
sin = loop (proj2 >>> neg >>> integrate 1 >>> integrate 0 >>> dup)
```

In doing so, we will effectively walk through the process of *type checking* the code, ensuring that the implementation of `sin` is consistent with its stated type.

Immediately, we can see that `sin` is defined in terms of a call to the `loop` function which has type `loop :: CRN (a, c) (b, c) -> CRN a b`. Without understanding the particulars of `loop`, we can see that `loop` takes a CRN as input and produces a CRN as output. This is good because `sin` is suppose to be a CRN! Furthermore, we can observe that the type of `sin` is `CRN a Double` and the output type of `loop` is `CRN a b`. This means that the type variable `b`

10:20 Reactamole

will be instantiated to `Double`. Consequently, this means that the output of `loop` has type `CRN a Double`, which is consistent with the declared type of `sin`. Now, we must check that the value passed to `loop` has type `CRN (a, c) (Double, c)`, i.e., a CRN that takes a pair of an `a` and a `c` as input and produces a pair of a `Double` and a `c` as output.

Next, let's look at the argument to `loop` and work through its type. The composition operator (`>>>`) has type:

```
(>>>) :: CRN a b -> CRN b c -> CRN a c
```

Intuitively, we can think of (`>>>`) as chaining together the output of one CRN of type `b` with a CRN expecting that same type as input. The result is a CRN that takes as input the intended input of the first CRN and produces as output the intended output of the second CRN.

Furthermore, (`>>>`) is left-associative, so the argument is really parenthesized as:

```
((((proj2 >>> neg) >>> integrate 1) >>> integrate 0) >>> dup)
```

Because of this, we first look at the type of the CRN produced by `proj2 >>> neg`. `proj2` and `neg` have the following types:

- `proj2 :: CRN (a, c) c`
- `neg :: CRN Double Double`

The composition operator feeds the output of `proj2` as the input of `neg`, and so it must be the case that the type variable `c` is instantiated to `Double`. Therefore, the type of the overall subexpression is:

```
proj2 >>> neg :: CRN (a, Double) Double
```

In other words, so far, we have a CRN that takes a pair of an unknown type `a` and a `Double` as input and produces a `Double` as output.

Next, we compose this CRN with the CRN produced by `integrate` of type

```
integrate :: Double -> CRN Double Double
```

In other words, `integrate` is a function that, when given a `Double`, produces a CRN that takes a `Double` as input and produces a `Double` as output. Thus, the expression `integrate 1` has type `CRN Double Double`. If we compose this CRN with `proj2 >>> neg`, we obtain a CRN with type:

```
proj2 >>> neg >>> integrate 1 :: CRN (a, Double) Double
```

Furthermore, chaining the second `integrate` call preserves this type, yielding:

```
proj2 >>> neg >>> integrate 1 >>> integrate 0 :: CRN (a, Double) Double
```

Now we need to compose this whole expression with the last call in the chain, `dup`. The function `dup` has type `dup :: CRN d (d, d)` for some unknown type `d`. When we compose this with our result, we take the output of our built-up expression, `Double`, and feed it to `dup`. This leads to a final type for the sub-expression of `loop`:

```
proj2 >>> neg >>> integrate 1 >>> integrate 0 >>> dup  
:: CRN (a, Double) (Double, Double)
```

Finally, we need to reconcile this type with `loop`. Above, we determined that `loop` expects a value of type `CRN (a, c) (Double, c)`. Through our derivation, we have concluded that the argument has type `CRN (a, Double) (Double, Double)`. Consequently, we know that the argument type matches the expected type of the function as long as we instantiate `c` to `Double`!

Parallel Pairwise Operations on Data Stored in DNA: Sorting, Shifting, and Searching

Tonglin Chen ✉ 🏠

Department of Electrical and Computer Engineering,
University of Minnesota, Minneapolis, MN, USA

Arnav Solanki ✉ 🏠 

Department of Electrical and Computer Engineering,
University of Minnesota, Minneapolis, MN, USA

Marc Riedel¹ ✉ 🏠 

Department of Electrical and Computer Engineering,
University of Minnesota, Minneapolis, MN, USA

Abstract

Prior research has introduced the Single-Instruction-Multiple-Data paradigm for DNA computing (SIMD DNA). It offers the potential for storing information and performing in-memory computations on DNA, with massive parallelism. This paper introduces three new SIMD DNA operations: sorting, shifting, and searching. Each is a fundamental operation in computer science. Our implementations demonstrate the effectiveness of parallel pairwise operations with this new paradigm.

2012 ACM Subject Classification Computing methodologies → Parallel computing methodologies; Applied computing → Computational biology

Keywords and phrases Molecular Computing, DNA Computing, DNA Storage, Parallel Computing, Strand Displacement

Digital Object Identifier 10.4230/LIPIcs.DNA.27.11

Funding The authors are funded by DARPA grant #W911NF-18-2-0032.

Acknowledgements We thank David Soloveichik, Olgica Milenkovic, Boya Wang, and Cameron Chalk.

1 Introduction

Beginning with the seminal work of Adelman a quarter-century ago [1], DNA computing has promised the benefits of massive parallelism in operations. More recently, there has been considerable interest in DNA storage [3, 4]. A particularly promising approach is to encode data by “nicking” DNA with editing enzymes such as PfAgo and CRISPR-Cas9 [9, 12]. A novel paradigm that combines this form of data storage with computation, dubbed “SIMD DNA”, was introduced in 2019 [13]. Data is stored on potentially long DNA strands, divided into “cells”, each storing a single bit. Nicks and denaturing create open toeholds in each cell. Toehold-mediated strand displacement [10, 14] is used to implement computation on the stored values.

This paper first proposes a new encoding system for SIMD DNA computation, suitable for general pairwise operations. Then it presents three novel applications using the new encoding system. The first is a binary bubble sorting algorithm (equivalent to rule 184 with elementary cellular automata [7, 8]). We show that sorting can be performed in only N parallel steps, where N is the number of bits to be sorted. The second application is a left-shifting operation (equivalent to rule 170 with elementary cellular automata), performed in a single parallel step. The third application is a parallel search algorithm that returns an answer as to whether a

¹ corresponding author



query substring is present in a target string. In principle, the algorithm can return an answer in $\log(n)$ steps, but our implementation requires between $\log(n)$ and n steps to complete, depending on the problem size and implementation constraints, where n is the length of the query string. Note that the parallelism is still impressive, assuming that the query string length n is much smaller than the target string length m . All three applications are of immediate practical interest, as many forms of computation on stored data entail some form of sorting, shifting, and searching.

2 Background

2.1 Parallel computation using SIMD

SIMD is a computer engineering acronym for Single Instruction, Multiple Data [6], a form of computation in which multiple processing elements perform the same operation on multiple data points simultaneously. It contrasts with the more general class of parallel computation called MIMD (Multiple Instructions, Multiple Data), where multiple processing elements can perform completely different operations on multiple data points simultaneously. While general MIMD parallelism might be desirable, it is often not practical. Much of the modern progress in electronic computing power has come by scaling up SIMD computation with platforms such as graphical processing units (GPUs).

2.2 SIMD DNA structure

SIMD implemented on DNA is intriguing. It provides a means to transform stored data, perhaps large amounts of it, with a single parallel instruction. We will review the paradigm as we introduce our new encoding scheme and our new applications; of course, we do not claim credit for the original concepts. The reader is referred to [13].

SIMD DNA computation is predicated on the encoding scheme for data. Conceptually, we divide stretches of double-stranded DNA into “domains”, where each domain is a contiguous sequence of nucleotides of some small specified length (typically 5 to 20). A sequence of several (typically 5 to 7) domains maps to a “cell” storing one binary bit. Whether a cell stores a 0 or a 1 depends upon topological variations, specifically the location of nicks, i.e., breaks in the DNA backbone. The nicks always occur on one strand of a double-stranded complex (generally the top strand in our examples); the other remains untouched.

The computation is carried out by a sequence of “instructions”, where each instruction implements DNA strand displacement reactions on cells. Instructions are initiated by single-stranded “instruction strands” added to the solution. After the strand displacement cascades complete, any single-strand fragments in the solution are washed away; the original strand is kept and separated via a magnetic bead. After a sequence of instructions, the data is transformed to its final state. The readout can be performed via fluorescence or with Oxford nanopore devices [2], [9].

The general flow of SIMD DNA computation is summarized as follows and illustrated in Figure 1.

1. Design an encoding structure that best suits the algorithm.
2. Encode the data at specific locations, using enzymes to nick corresponding targets.
3. Gently denature the DNA, allowing segments between adjacent nicks to detach, exposing toeholds.
4. Execute instructions, implemented as strand-displacement operations.
5. Finally, read out data using fluorescence or with nanopores.

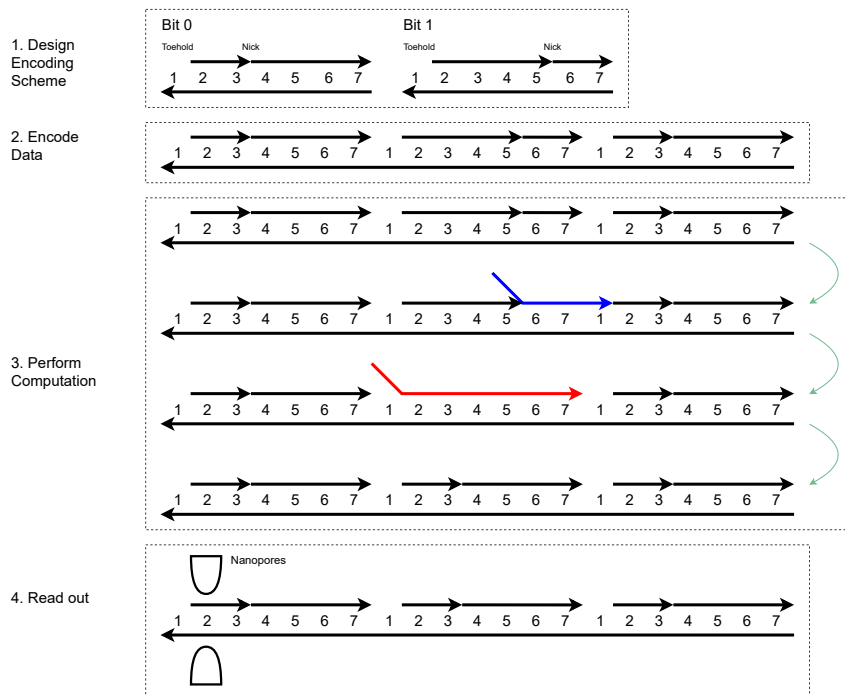


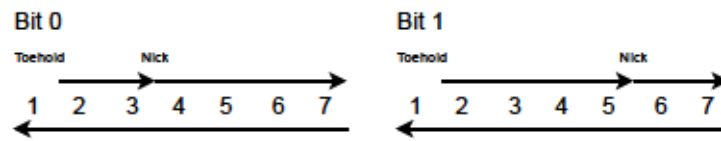
Figure 1 General Outline of SIMD DNA Computations. Arrowheads represent “nicks”: breaks in the DNA backbone, performed with gene editing techniques. Integers represent “domains”: contiguous sequences of nucleotides of some small, specified length. For convenience, we use the numbers 1 through 7 repeatedly; however, each copy of a number represents a distinct domain, consisting of a unique nucleotide sequence. Stage 1 shows the encoding of binary bits 0 and 1, based of different locations of toeholds and nicks. Note that domain 1 is always “exposed”: the DNA backbone of the top strand is nicked, and the DNA is gently denatured until this segment falls off, exposing a toehold at this domain. Stage 2 shows an example of encoding the bits 010. Stage 3 illustrates the step in which computation is performed with strand displacement, in a general sense. Details of this step will be provided for specific algorithms in later sections. Note that, in this generic example, the location of nick in the second cell has changed at the end of stage 3. Stage 4 illustrates how nanopore sequencing could be used to perform readout.

3 Design of Encoding System

Several schemes for encoding binary data were proposed in prior work [13], each chosen to minimize the number of operations for a specific algorithm. Here we propose a new encoding scheme that works well for the broad class of algorithms that consist of parallel, pairwise operations. A requirement for running these algorithms is that the encoding scheme should allow the algorithm to recognize any combination of adjacent bits. This specification comes at the expense of more complexity for some algorithms, i.e., more operations per step than possible with a customized encoding.

The encoding scheme is shown in Figure 2. Each cell stores a single binary value (a “bit”). Each cell consists of 7 domains. We do not specify the actual nucleotide sequence of the domains here for simplicity. While preparing this cell, the top DNA strand must be nicked before and after domain 1. This strand can then be displaced by denaturing, creating an exposed toehold. Domain 1 is always exposed as a toehold in this representation. Domains 2 through 7 are covered. When storing a bit 0, we will nick the top strand between domains 3 and 4; when storing a bit 1, we will nick between domains 5 and 6. There are four possible

11:4 Sorting, Shifting, and Searching in DNA



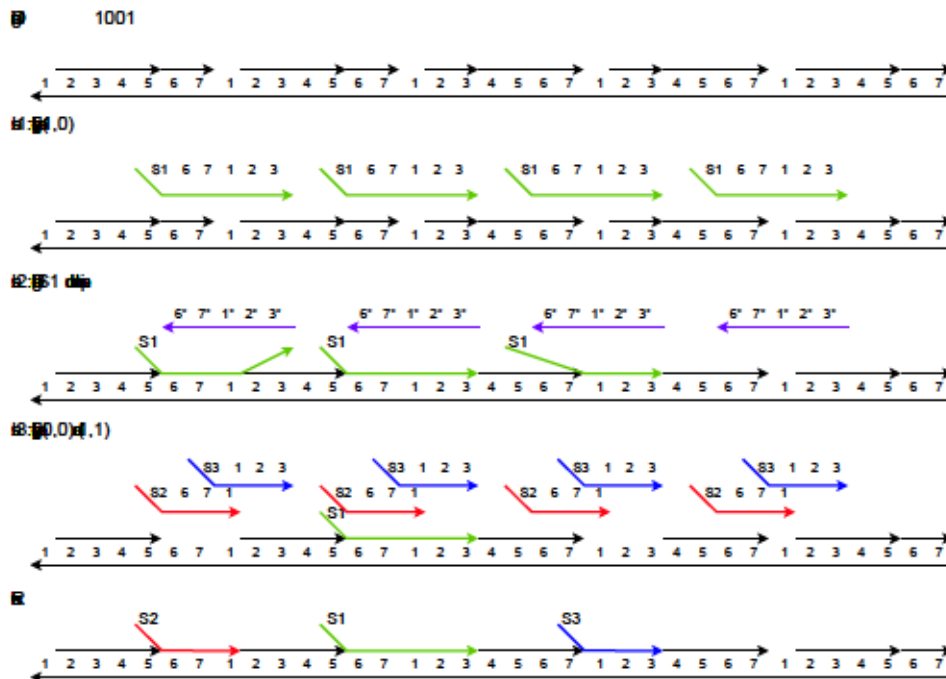
■ **Figure 2** Bit representation in the encoding scheme. Horizontal lines represents DNA strands. Integers represent “domains”: specific sequences of nucleotides. Arrow heads represent nicked positions: places where the phosphodiester bond in the backbone of the DNA strand has been broken, via gene-editing techniques. Cells store binary values. Each cell consist of 7 domains. Domain 1 is always exposed, forming a toehold.

pairings for two adjacent cells. Each will be detected using different domain combinations: for (0, 0), domains 1, 2 and 3; for (0, 1), domain 1 only; for (1, 0), domains 6 through 3 with wrapping at domain 7 and 1; and for (1, 1), domains 6, 7 and 1.

Before describing the implementation of specific algorithms for sorting, shifting, and searching, we will present some general algorithmic steps useful in implementing all of these.

3.1 Identifying Bit Pairs

A common task in our algorithms is “identifying” pairs of adjacent bits, i.e., recognizing the specific pair of cells at a location of interest. We will exploit the fact that domain 1 is always exposed to identify these specific pairs. Figure 3 illustrates our approach on the string 11001, which contains all 4 possible adjacent pairs: 00, 01, 10 and 11.



■ **Figure 3** Example of Identifying Different Pairs of Adjacent Bits.

Identification is performed with three instructions. In instruction 1, the strands (S_1 6 7 1 2 3) are issued to all pairs of bits. Through the toehold at domain 1 between each pair, the strand S_1 binds to domains 6, 7, 1 in the pair (1, 1), leaving domains S_1 , 2, 3 open. In

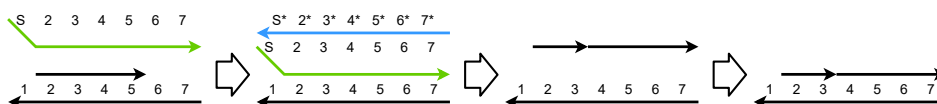
the pair (0, 0), the strand S_1 binds to domains 1, 2, 3, leaving domains S_1 , 6, 7 open. The strand S_1 binds to domains 6, 7, 1, 2, 3, in the pair (1, 0). The strand S_1 does not bind to the pair (0, 1) since the only exposed toehold is domain 1. We can then distinguish the pair (1, 0) from the open domains on strand S_1 .

In instruction 2, using the complementary strands ($6^* 7^* 1^* 2^* 3^*$), the strand S_1 that attaches to the pairs (0, 0) and (1, 1) is pulled out. This is done through the open domains 2, 3 in the pair (0, 0) and the open domains 6, 7 in the pair (1, 1) on strand S_1 . After this instruction, strand S_1 remains only in the pair (1, 0).

In instruction 3, two instruction strands are issued at the same time: (S_2 6 7 1) and (S_3 1 2 3). Here (S_2 6 7 1) will bind to the pair (1, 1) and (S_3 1 2 3) will bind to the pair (0, 0). They will not bind with any other pairs since the only exposed toehold for binding would be domain 1; they will prefer the locations with more exposed domains.

The result is that the adjacent bit pairs (1, 1), (1, 0) and (0, 0) are each *labeled* with strands S_2 , S_1 and S_3 respectively. Pairs (0, 1) are labelled with an exposed toehold at domain 1. This toehold could be replaced by a strand (S_x 4 5 6 7 1) or a strand (S_x 1 2 3 4 5); the choice would be made depending on the use case.

3.2 Rewriting a cell



■ **Figure 4** Example of Rewriting in Three Steps.

By exposing toeholds across domains 2 through 7 in a cell, we can rewrite the content of that cell – so change a 1 to 0 or a 0 to 1 – with three instructions. The idea is that, since there are exposed domains, we can displace the content of the cell with a single strand covering all these domains. Then we can remove the covering strand through the exposed “tag” domain (S in Figure 4) using a complementary strand. The cell is now completely exposed. We can write a new bit to it by hybridizing the strands according to our encoding scheme, leaving domain 1 as a toehold and placing the nick at the desired location.

4 Parallel Binary Bubble Sorting

Sorting is a simple yet fundamental operation in computer science. Here we consider sorting binary values.² Sorting can be used to determine the “weight” of a vector of 0’s and 1’s: the count of the number of 1’s relative to the length of the vector. It can also be used to compute the majority function: whether there are more 1’s than 0’s or not in the input set. Majority is a fundamental operation for many machine-learning algorithms.

Our SIMD DNA implementation performs parallel bubble sorting on binary bits [5]. It can be expressed as a pairwise operation in the form of $f(a, b) = (c, d)$, where (a, b) is the value of the input bit pair, and (c, d) , the outputs, represent the action we take, whether to rewrite or to leave it as it is. The outputs can be 0 or 1, which means that we can arbitrarily change the value of the cell. They can also be X , meaning they remain unchanged. We discuss what kind of pairwise operations can be performed on our encoding in Section 7.1.

² Perhaps counter-intuitively, sorting binary values in hardware is as difficult algorithmically as sorting arbitrary values such as integers or real numbers [5].

11:6 Sorting, Shifting, and Searching in DNA

The sorting operation can be expressed in the following pairwise operation,

$$f(0,0) = (X,0) \quad f(0,1) = (X,X) \quad f(1,0) = (0,1) \quad f(1,1) = (1,X).$$

Algorithmically, the following “bit swapping” is performed:

- If the current bit is 1, it changes it to 0 if and only if its right neighbor is 0.
- If the current bit is 0, it changes it to 1 if and only if its left neighbor is 1.

We argue that repeatedly performing such bit swapping will sort the entire sequence of binary values.

▷ **Claim 1.** Bit swapping will never happen more than once for any consecutive sequence of three bits. Such a sequence consists of two consecutive pairs, sharing the middle bit.

Proof. The only pair of consecutive bits that ever gets rewritten is the pair $(1,0)$ to $(0,1)$. It is impossible to have two consecutive, overlapping pairs $(1,0)$ sharing a common middle bit. ◁

Accordingly, bubble sorting binary values in parallel does not require an odd and even index addressing scheme, as does bubble sorting arbitrary values.

▷ **Claim 2.** Sorting completes in at most $(N - 1)$ parallel steps where N is the total number of bits.

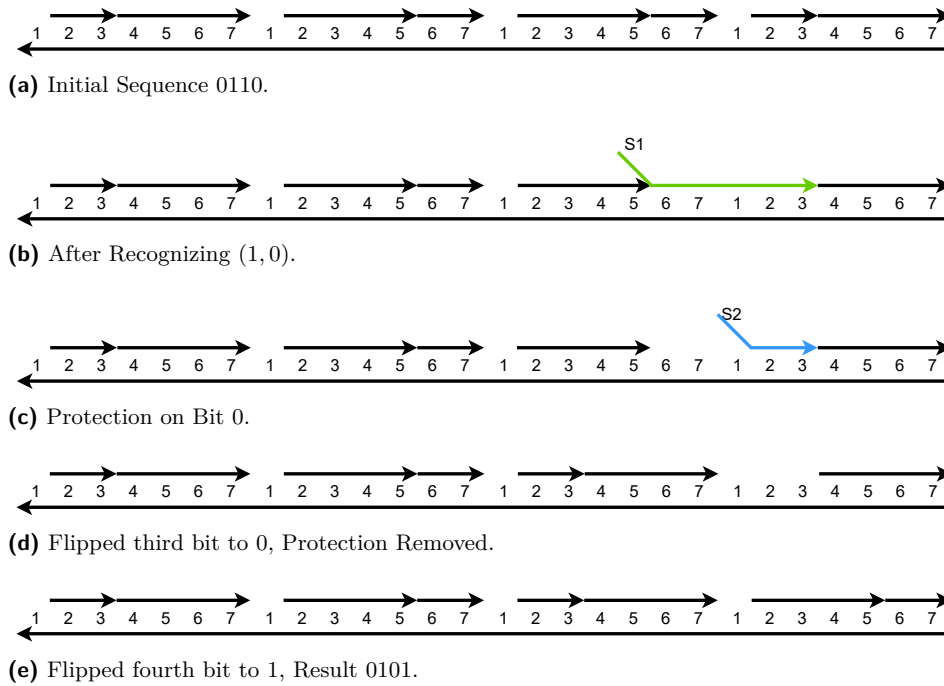
Proof. Suppose we have a sequence of binary bits of length N , in which all bits except the first are 0. When applying the algorithm, the 1 located at the start will be pushed back one position at a time with the $f(1,0) = (0,1)$ bit swap operation. Fully sorting the sequence, i.e., moving the 1 to the last position, requires $N - 1$ total swaps. Now suppose we are sorting an arbitrary bit sequence. We argue that, after $N - 1$ swaps, all the 1's will be at the end of the sequence. To see why, note that an $f(1,0) = (0,1)$ operation moves a 1 forward, while an $f(1,1) = (1,1)$ operation does not affect adjacent 1's. Thus, in $N - 1$ steps, all 1's will have moved to end of the sequence. ◁

4.1 Implementation

Here we give an instruction set for performing parallel binary bubble sort with SIMD DNA, using the encoding in Figure 2. It consists of 12 individual instructions. These are summarized as follows.

1. Label pairs $(1,0)$.
2. Uncover these, leaving domains 6 and 7 for the bits 1 and domains 2 and 3 for the bits 0 open in these pairs.
3. Protect the bits 0 of these pairs by covering the corresponding toehold at domains 2 and 3.
4. Flip the bits 1 to 0 in these pairs.
5. Release the protective covers; flip the bits 0 to 1 in these pairs.

For the initialization, we can use the first two instructions described in Section 3.1, with an additional instruction to fix open domains for bits that do not change. We can use the rewriting method described in Section 3.2 to flip the bits. A full description of the implementation of sorting is provided in Appendix B.



■ **Figure 5** Outline of the SIMD DNA parallel binary sorting algorithm.

5 Parallel Left Shifting

We propose a SIMD DNA implementation of shifting, another fundamental operation in computer science. Shifting left corresponds to multiplying a binary number by 2; shifting right corresponds to dividing it by 2. It is a useful operation in general for aligning data in a variety of algorithms [5]. We present a left shift algorithm, one that shifts all N binary bits one position to the left, with the Least Significant Bit (LSB) remaining unchanged. This operation is, of course, a parallel left shift, moving all bits simultaneously in lockstep. Our implementation requires 11 instructions per shift. Note that unlike usual arithmetic or logical left shift that inserts a bit 0 to the LSB, the left shift operation described here keeps the original LSB, thereby duplicating the LSB. The usual left shift could be implemented by adding instructions rewriting the LSB to 0 after the instructions we provide here.

We describe the shift operation using the following pairwise operation as:

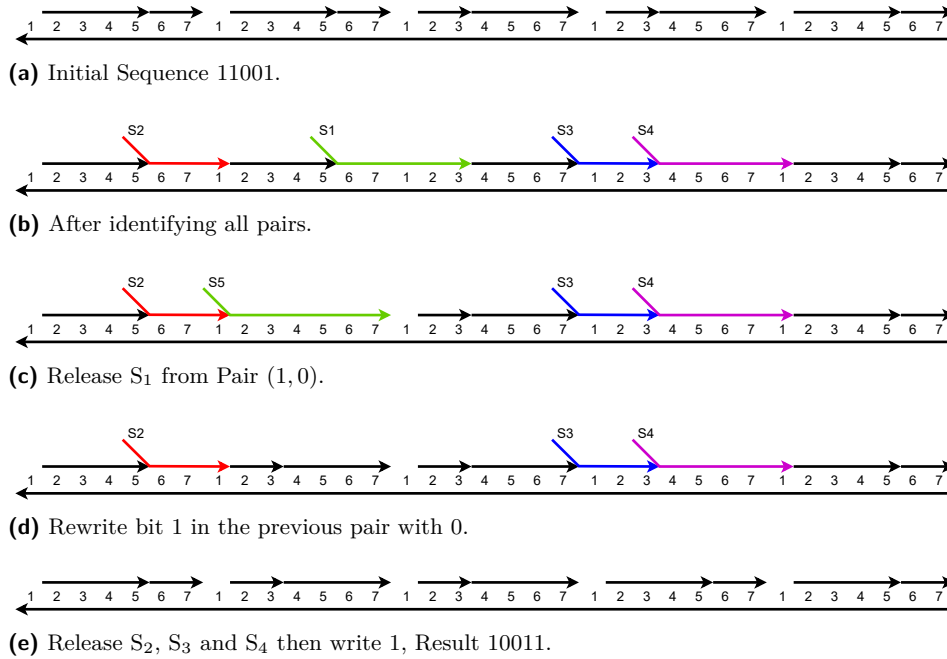
$$f(0, 0) = (0, X) \quad f(0, 1) = (1, X) \quad f(1, 0) = (0, X) \quad f(1, 1) = (1, X)$$

Here X means a value that does not change. For each bit pair, the operation writes the value of the right bit to the left bit. Since only the value of the left bit is changed in each bit pair, the operation is non-overlapping and can be implemented using the encoding scheme we propose. We illustrate with the example of shifting 11001 to 10011, shown in Figure 6.

1. Label all the bit pairs. Cover the toeholds for the pairs (0, 0) and (1, 1).
2. For the pairs (1, 0), flip the bits 1 to 0.
3. For the pairs (0, 1), flip the bits 0 to 1.
4. Finally, uncover all the toeholds for the pairs (0, 0) and (1, 1).

A full description of the implementation of shifting is given in Appendix C.

11:8 Sorting, Shifting, and Searching in DNA



■ **Figure 6** Outline of the SIMD DNA parallel left shift operations. The initial sequence S is 11001 and the result sequence T is 10011. The operation shift each bit to left one position ($T[5:1]=S[4:0]$), while keeping the Least Significant Bit unchanged.

6 Parallel Search Algorithm

Searching is fundamental to all branches of computer science that involve data storage and retrieval. We consider the problem of deciding whether a given substring exists in a stored string of bits. We first discuss a general algorithm that returns an answer to such a question in $\log(n)$ parallel steps, where n is the substring length. We then propose an implementation in SIMD DNA. Due to practical constraints, the time complexity of the implementation is not $O(\log(n))$; it is closer to $O(n)$, depending on the problem size and implementation details. We note that a requirement of our algorithm is that the length of the query string is a power of 2. We discuss the time complexity and constraints in detail in Section 7.3.

6.1 Algorithm

Suppose we have a *query* substring Q of a length n and we would like to search whether it appears in a much longer *target* string A . Pseudo-code for our approach is given as Listing 1. We will elucidate the pseudo-code by stepping through examples.

6.1.1 Parallel search procedure

We illustrate searching for a query string $Q = 1101$ in the following target string A :

$$\begin{aligned}
 A_0 &= 10101010\mathbf{1101}10100011\mathbf{1101}01000100 \\
 A_1 &= a_2a_2a_2a_2a_3a_1a_2a_2a_0a_3a_1a_1a_0a_1a_0 \\
 A_2 &= b_0b_0\mathbf{b_1b_0b_2b_1b_3b_3}
 \end{aligned} \tag{1}$$

■ **Listing 1** Pseudo-code for Parallel Search Algorithm. Note that the operations inside the two **foreach** loops can be performed in parallel since they are independent. The **pair** operation here is to find a corresponding symbol that replaces the two symbols in the lookup table, and the **identity** operation is to look up the symbol that represents the query string.

```

S = Query String
T = Target String
n = length of S
for i in range(0,n-1):
    T_i = T
    truncate first i characters of T_i
    p = 1
    while p <= n:
        j = 0
        while j < (length(T_i)-1):
            a = T_i[j]
            b = T_i[j+1]
            c = pair(a,b) # Pair 2 consecutive cells
            if c.identity(S): # Check if new pair is the query
                return True
            replace a,b in T_i with c
            j += 1
        p = 2*p
    return False

```

The original string is A_0 . In each step, two consecutive symbols are read and replaced with a single symbol. Here $a_0 = 00, a_1 = 01, a_2 = 10, a_3 = 11, b_0 = a_2a_2, b_1 = a_3a_1, b_2 = a_0a_3, b_3 = a_1a_0$. Note that $Q = 1101 = a_3a_1 = b_1$. After three steps, we conclude that the query string exists in the target string, since there are two matches in the string A_2 .

6.1.2 Search procedure with offset

It is possible that the query string does not align with divisions of length n in the target string. Thus we need to repeat the operation with offsets. The following example illustrates the operation with an offset of 2 bits.

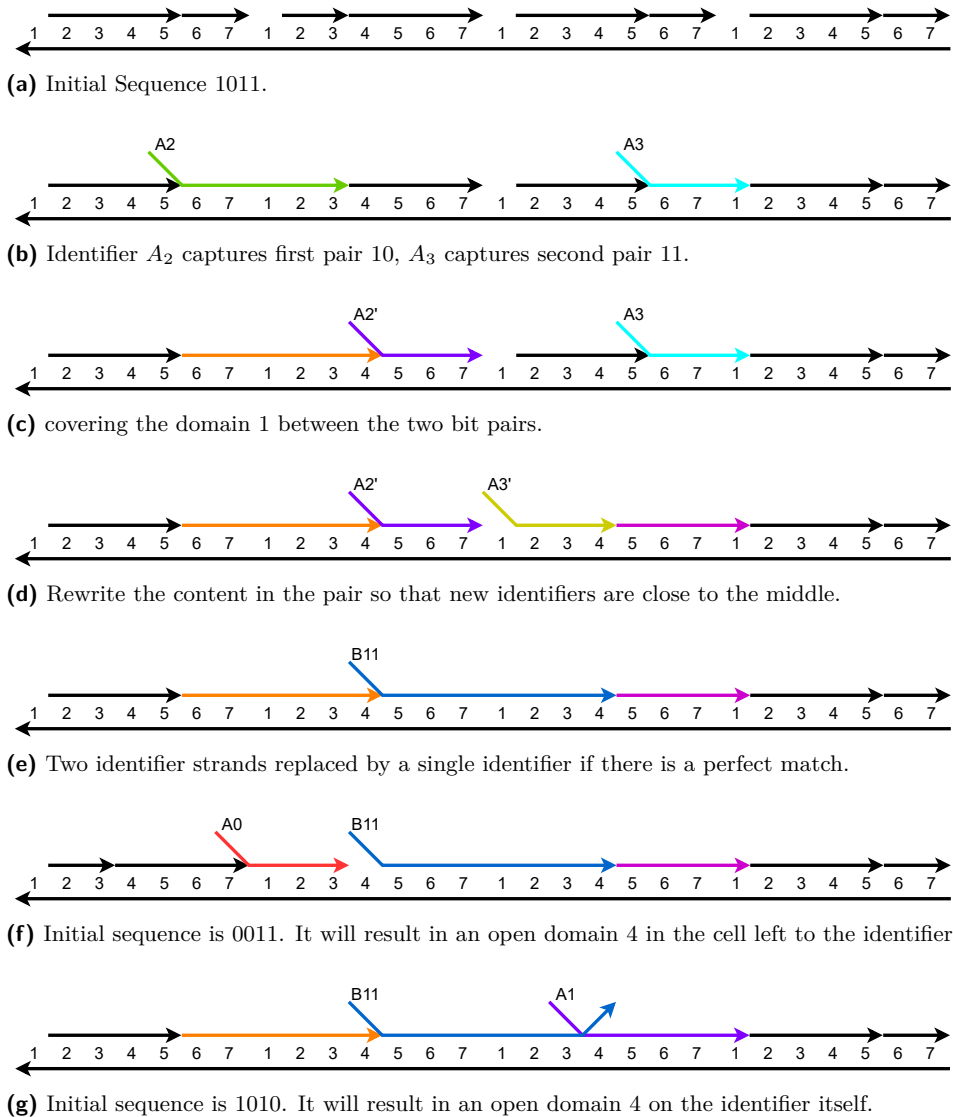
$$\begin{aligned}
 A_0 &= \text{10101011010110000011110001000100} \\
 A_1 &= \text{10a}_2a_2a_3a_1a_1a_2a_0a_0a_3a_3a_0a_1a_0a_1a_0 \\
 A_2 &= \text{10b}_0b_1b_2b_3b_4b_5b_5a_0
 \end{aligned} \tag{2}$$

Here, the replacement is given by the aggregated pairs $a_0 = 00, a_1 = 01, a_2 = 10, a_3 = 11, b_0 = a_2a_2, b_1 = a_3a_1, b_2 = a_1a_2, b_3 = a_0a_0, b_4 = a_3a_3, b_5 = a_0a_1$. Again, an instance of the query string is found in the target string.

Searching for a query string with a given offset requires at most $\log(n)$ steps. In general, for an arbitrary query string of a length n (a power of 2), the search must be performed n times with offsets ranging from 0 to $n - 1$. In principle, all of these searches could be performed in parallel, as none would interfere with any other. Accordingly, our parallel implementation of searching completes in $\log(n)$ steps.

Note that the number of aggregated pair identifiers needed – the a 's and b 's in the example above – grows exponentially with the length of the target string. However, these can be synthesized once and reused for every query. If we consider the restricted problem of searching for a *specific* query string, meaning that we only use pair identifiers for matching pairs, then the number of identifiers needed is $\sum_{i=1}^{\log(n)} 2^i = n - 1$.

6.2 Implementation



■ **Figure 7** Example implementation of search algorithm on target sequence 1011.

To implement the algorithm in SIMD DNA, we do not issue instruction strands to each pair of overlapping bits. Instead, we consider the non-overlapping bit pairs. In the example shown in Figure 7, for the bit sequence 1011, we would consider operations on bit pair 10 and 11, but not on bit pair 01.

Figure 7 shows the critical steps on searching a target sequence 1011. It provides an example of a successful search and also the potential outcome of two failed searches. To implement the search operation with an offset, we can simply skip the number of bits according to the offset. We use the word *symbol* to represent the consecutive cells that we search for on a certain level. For example, in the first level, the symbols are 10 and 11. We can use the bit identifying steps described in Section 3.1 to recognize these symbols. We use identifiers $A_0 = 00$, $A_1 = 01$, $A_2 = 10$, $A_3 = 11$ to represent symbols in this level. We then move on to the next level, searching for consecutive symbols A_2A_3 , which corresponds to the target string 1011.

In the first step of the second level, we first rewrite the topological structure at symbols that appear to be a query result. In this example, A_2 should be found as the left symbol, and A_3 should be found as the second symbol. We pull identifier A_2 out from every *odd* symbol (we only look at the first, third, fifth, etc.) and rewrite the entire symbol with the technique described in Section 3.2. After rewriting, we have the identifier A'_2 that covers domains (5 6 7) in the *right most* cell, as seen in Figure 7c. For the second symbol A_3 , we repeat the step described, except we pull the identifier out from every *even* symbol and the new identifier A'_3 covers domains (2 3 4) in the *left most* cell. Through these steps, we have essentially “moved” the identifier of the matching symbols to the middle. In the final step, we issue the new identifier strand (B_{11} 5 6 7 1 2 3 4) to the location between every two symbols. It will result in a perfect binding only if there is a match at the current symbol level. Figure 7e shows the example of a matching result. Figure 7f and 7g show two potential examples of imperfect binding, indicating a non-matching result. We can pull them out through the open domains either on the identifier itself or a nearby open domain on the base strand. Therefore, the presence of the identifier B_{11} indicates a successful match.

We can repeat the process to recognize multiple symbols at the same level. When we move to the next level $l + 1$, we can use the identifiers from this level l as a starting point for rewriting. To identify a symbol $S_{l+1,c} = S_{l,a}S_{l,b}$ at level $l + 1$, we simply pull out identifiers for $S_{l,a}$ at odd symbols and $S_{l,b}$ at even symbols at level l . Then we “move” the identifier to the middle. Finally, we give identifiers for $S_{l+1,c}$ to the middle of each pair and identify the symbol.

A possible weakness of our implementation is that the strand used for rewriting could potentially be very long. This could cause problems when performing these operations *in vitro* due to branch migration complications. Lastly, this search operation can handle multiple overlapping queries within the reference string, but this requires careful consideration of the base-pair sequence of the cells in designing identifier strands.

7 Discussion

We discuss the features and implementation constraints of the proposed algorithms.

7.1 Ability to compute any non-conflicting pairwise operation

In Section 4 and Section 5, we presented examples of algorithms that perform pairwise operations, namely sorting and shifting, respectively. Given the ability to identify pairs of bits and a universal way to rewrite a cell, we can readily implement any algorithm that performs non-conflicting pairwise operations. Such operations only entail rewriting pairs of adjacent bits. The result of the operation on a specific sequence should always be the same, irrespective of the execution order. To illustrate, consider the following operation:

$$f(0,0) = (X, X) \quad f(0,1) = (X, 1) \quad f(1,0) = (X, X) \quad f(1,1) = (0, X)$$

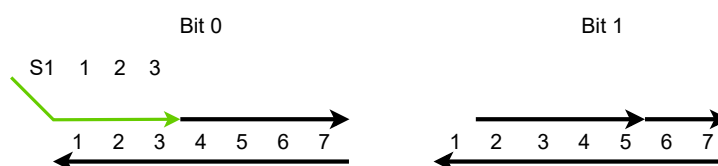
Here X indicates a value that does not change. This operation *is* conflicting. To see why, consider its effect on the sequence 011. The second bit should change to 1 when the operation is applied to the first pair. However, this bit should change to 0 when the operation is applied to the second pair. Depending on the order of execution, the final result will be different. To ensure an operation is non-conflicting, for every three adjacent bits that two operations are performed on, the middle bit should be set to the same value.

11:12 Sorting, Shifting, and Searching in DNA

Non-conflicting operations can be performed in parallel on all bit pairs. In the first step, we identify the four bit pairs described in 3.1. After this step, we supply strands with four labels covering the four bit pairs. Then, we release strands with specific labels one at a time to obtain write access to specific bit pairs. (Write access refers to a domain being exposed.) We rewrite these cells with the operation described in Section 3.2. The full operation requires rewriting all four bit pairs.

We conclude that our encoding scheme and design method are generally applicable to parallel bitwise algorithms, provided that they can be expressed in terms of such non-conflicted pairwise operations.

7.2 Converting to Different Encoding Schemes



■ **Figure 8** One strand could be used to differentiate two bits.

A benefit of the encoding scheme that we are proposing is that it can easily be converted to any other similar scheme since each cell always has an exposed domain 1. In the original SIMD DNA scheme proposed in [13], the authors designed two specific encoding schemes for the two applications proposed (rule 110 and a binary counter). We suggest that our encoding scheme could be used as an intermediate form when converting to other encoding schemes, designed for particular algorithms. Figure 8 illustrates how we can use a single strand (S_1 1 2 3) to differentiate bit values of 0 from bit values of 1. We can use the technique discussed in 3.2 to re-write the data with a different encoding scheme, so long as the scheme also encodes each bit with 7 domains. Complete instructions for performing such encoding changes are given in Appendix A.

7.3 Time Complexity of Parallel Search

While the time complexity of the proposed parallel search is $O(\log(n))$ in principle, where n is the query substring length, the time complexity of our SIMD DNA implementation is somewhat worse. While the abstract search algorithm finds the query in the reference string by pairing individual characters in parallel, and thus completes in $O(\log(n))$ steps, our implementation searches for and identifies distinct symbols sequentially, that is to say, it first searches for a specific symbol across all possible locations at once, then it searches for the next symbol across all locations at once, and so on.

The abstract algorithm assumes all symbols are identified in one pass to allow for further pairing. If we consider all the different symbols in a query string, counting repeated symbols, $\frac{n}{2^i}$ symbols must be searched sequentially at level i in our implementation. Accordingly, the total number of sequential search steps could be as high as $O(n)$. However, at each level, all the occurrences of a specific symbol are identified simultaneously. At level i , each symbol represents a binary string with a length of 2^i , so there are at most 2^{2^i} distinct symbols at level i . For example, in the first level, instead of searching for $\frac{n}{2}$ symbols, we only search for four distinct symbols. In the second level, there are only 16 distinct symbols. Since we only search for distinct symbols, the number of steps in the first few levels will be greatly reduced.

Our parallel search algorithm currently only works on query strings having a length that is a power of two. However, we believe that our implementation could be modified to allow for arbitrary-length query strings. We do not provide details here, as they are cumbersome, but we outline the method as follows.

Note that, in parallel search, the query string is searched reductively: at each level, two symbols are reduced to one symbol. When working with query strings having any arbitrary length, there might be an odd number of symbols in the current level, meaning that the last symbol cannot be reduced for the next level. In this case, we can add a method to identify the trailing odd symbol at the current level and replace it in the next level. The reduction can still be completed in a logarithmic number of levels.

8 Conclusion

We have presented algorithms for basic parallel operations within the SIMD DNA framework. We note that there are, in fact, two layers of parallelism possible:

1. Bit-level Parallelism: instructions applied to all bits in an array at once.
2. Data-level Parallelism: the same instructions applied to *multiple* arrays at once.

While operations on DNA are slow and error-prone, with these levels of parallelism, perhaps DNA computation could scale to a truly impressive regime. Consider the following back-of-an-envelope estimates. Suppose:

- we have 10^{12} independent cells in parallel in a single test tube;
- a single operation takes approximately 10 minutes to complete.
- different cells use the same DNA sequence. Using distinct sequences for different cells, as in our search operation, can result in a solution with multiple competing DNA molecules. At larger scales, this would result in an increase in reagent volume and could diminish reaction rates.

This means that we can perform approximately 10^9 operations per second in a single test tube, already impressive. Now suppose that:

- we have 100 test tubes.

This means we can compute at 100,000 MIPS (million instructions per second). This is comparable to what very respectable existing silicon systems can achieve. The key conceptual difference between the SIMD DNA approach and other forms of DNA computing is that it exploits a substrate on which data is stored. This enables the SIMD parallelism.

Many experimental hurdles remain in demonstrating and deploying this paradigm. DNA synthesis remains prohibitively expensive. A possible alternative is to use gene-editing techniques to encode data on naturally occurring DNA [11].

References

- 1 Leonard M Adleman. Molecular computation of solutions to combinatorial problems. *Science*, pages 1021–1024, 1994.
- 2 Nagendra Athreya, Olgica Milenkovic, and Jean-Pierre Leburton. Detection and mapping of dsDNA breaks using graphene nanopore transistor. *Biophysical Journal*, 116(3):292a, 2019.
- 3 Luis Ceze, Jeff Nivala, and Karin Strauss. Molecular digital data storage using DNA. *Nature Reviews Genetics*, 20(8):456–466, August 2019. doi:10.1038/s41576-019-0125-3.
- 4 George Church, Yuan Gao, and Sriram Kosuri. Next-generation digital information storage in DNA. *Science (New York, N.Y.)*, 337:1628, August 2012. doi:10.1126/science.1226355.

- 5 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- 6 M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.
- 7 Joachim Krug and Herbert Spohn. Universality classes for deterministic surface growth. *Physical Review A*, 38(8):4271, 1988.
- 8 Wentian Li. Power spectra of regular languages and cellular automata. *Complex Systems*, 1(1):107–130, 1987.
- 9 Ke Liu, Chao Pan, Alexandre Kuhn, Adrian Pascal Nievergelt, Georg E Fantner, Olgica Milenkovic, and Aleksandra Radenovic. Detecting topological variations of DNA at single-molecule level. *Nature communications*, 10(1):1–9, 2019.
- 10 David Soloveichik, Georg Seelig, and Erik Winfree. DNA as a universal substrate for chemical kinetics. *Proceedings of the National Academy of Sciences*, 107(12):5393–5398, 2010. doi: 10.1073/pnas.0909380107.
- 11 S. Tabatabaei, Boya Wang, Nagendra Athreya, Behnam Enghiad, Alvaro Hernandez, Christopher Fields, Jean-Pierre Leburton, David Soloveichik, Huimin Zhao, and Olgica Milenkovic. DNA punch cards for storing data on native DNA sequences via enzymatic nicking. *Nature Communications*, 11, December 2020. doi: 10.1038/s41467-020-15588-z.
- 12 S Kasra Tabatabaei, Boya Wang, Nagendra Bala Murali Athreya, Behnam Enghiad, Alvaro Gonzalo Hernandez, Christopher J Fields, Jean-Pierre Leburton, David Soloveichik, Huimin Zhao, and Olgica Milenkovic. DNA punch cards for storing data on native DNA sequences via enzymatic nicking. *Nature communications*, 11(1):1–10, 2020.
- 13 Boya Wang, Cameron Chalk, and David Soloveichik. SIMD||DNA: Single instruction, multiple data computation with DNA strand displacement cascades. In Chris Thachuk and Yan Liu, editors, *DNA Computing and Molecular Programming*, pages 219–235, Cham, 2019. Springer International Publishing.
- 14 Bernard Yurke. A DNA-fuelled molecular machine made of DNA. *Nature*, 406(6796: 605), 2000.

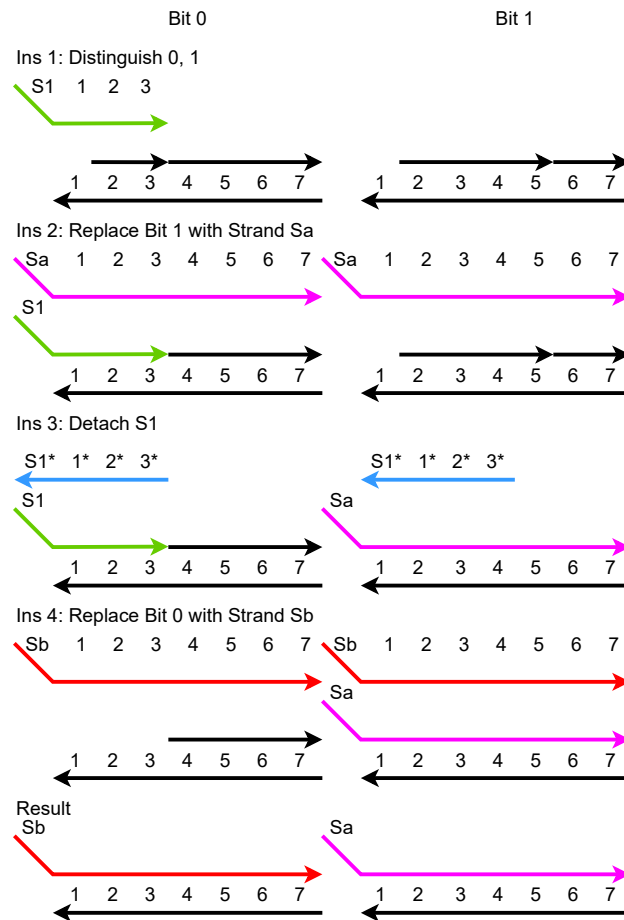
A Instructions for Converting to Another Scheme

Instruction 1 identifies and distinguishes the two different bits. In instruction 1, strand (S_1 1 2 3) is issued. In bit 0, the strand will displace the short strand over domains 2 and 3 but does not edit bit 1 since domain 1 is the only open domain for binding. In instruction 2, all domains in bit 1 are replaced by a single strand covering all domains with identifier S_a . Then in instruction 3, the strand S_1 is detached, so domains 1, 2, and 3 on bit 0 are exposed. In Instruction 4, all domains in bit 0 are replaced by a single strand covering all the domains with the identifier S_b . Then any encoding scheme with 7 domains in 1 cell could be written to the bits by first detaching strand S_a and writing the encoding for bit 1, then detaching strand S_b and writing the encoding for bit 0.

B Detailed Implementation of Each Step for Parallel Sorting

Here we give an instruction set for parallel binary bubble sort with the previously defined encoding scheme. We can implement each step of the sorting algorithm in 12 individual operations. Details of the design are shown in Figure 10.

The 12 instruction falls to 2 stages. The first stage is “identifying.” During instructions 1–4, all the pairs (0, 1) are identified, and in both bit 0 and 1, a toehold is exposed for writing new data. More specifically, Instructions 1 and 2 identify the combination of (1, 0). In instruction 1, (S_1 6 7 1 2 3) is issued to each pair of bits. In pair (0, 0), S_1 and domains 6,



■ **Figure 9** Current coding scheme could be converted to other coding scheme.

7 are exposed. In pair (0, 1), since the only open domain is 1, it will not form a strong enough bind. In pair (1, 0), only S_1 is exposed. In pair (1, 1), S_1 and domains 2, 3 are exposed. In instruction 2, strand (6* 7* 1* 2* 3*) is issued to each pair of bits. Since pair (1, 0) is the only pair that does not have exposure 5 or 2, this strand will detach strand S_1 in each pair except pair (1, 0). After Instruction 2, the toehold between a bit value of 1 and a bit value of 0 in the pair (1, 0) is replaced by a strand with an identifier of S_1 . Instruction 3 seals off the domain exposed in the other pairs during Instruction 1 and 2 so that it will not be edited later. In instruction 4, the strand with identifier S_1 is detached, exposing domains 6 and 7 in the left cell containing bit 1, or domains 2 and 3, in the right cell containing bit 0. After this instruction, toeholds are exposed only in the 1s and 0s in pair (1, 0). Other bits are not affected.

The second stage is flipping the bits in the pair (1, 0). In instruction 5, in the case of a bit value of 0, domains 2 and 3 are temporarily covered by a strand with identifier S_2 so that the writing process will not interfere with the identified 0s at this moment. In instruction 6, a bit value of 1 is replaced by a strand with identifier S_3 via the open toehold at domains 6 and 7. The strand is then detached in instruction 8, exposing all the domains of that bit. Then, the bit value of 0 is written to the location of a bit value of 1 in instruction 8. In instruction 9, the temporary cover for a bit 0 is lifted. Then, in instructions 10 through 12, a bit 1 is written to the location of a bit value of 0 using the same scheme as instructions 6 through 8. Throughout the process, only bits identified in the first stage with toeholds exposed are affected.

C Detailed Implementation of Each Step for Parallel Left Shift cell

The instructions are shown as followed, with an example of shifting 11001 to 10011.

The first three instructions are exactly the same as those for identifying bit pairs in Section 3.1. In instruction 1, the strand (S_1 6 7 1 2 3), which identifies the different patterns of two bits, is issued to each pair of bits. In instruction 2, strand ($6^* 7^* 1^* 2^* 3^*$) is issued, detaching strands with open domains 6 and 7, or 2 and 3. After this instruction, strands with identifier S_1 only remain at pair (1, 0). In instruction 3, we issue two species of strands at the same time: (S_2 6 7 1) and (S_3 1 2 3). (S_2 6 7 1) will bind with pair (1, 1) and (S_3 1 2 3) will bind with pair (0, 0). S_2 will not form a stable binding with pair (0, 0) or (0, 1) because the binding area is only one domain. Same goes with S_3 and pair (1, 1) or (0, 1). After this instruction, only domain 1 between pair (0, 1) is still exposed. In instruction 4, strand (S_4 4 5 6 7 1) is issued. Through the open domain 1 between pair (0, 1), the strand in bit 0 is replaced by S_4 . After this step, the first bit in pair (1, 0) is identified with the strand S_1 , and the first bit in pair (0, 1) is replaced with the strand S_4 .

Instructions 5 to 9 rewrite the first bit in pair (1, 0) to 0. In instruction 5, the strand S_1 is detached, exposing domains 6, 7, 1, 2 and 3. The exposed domains 2 and 3 are sealed off in instruction 6 to not interfere with subsequent instructions. In instruction 7, strand (S_5 2 3 4 5 6 7) is issued through the open toehold on domains 6 and 7 in the bit 1 in pair (1, 0), and displaces the strand in that bit. Since domains 2 and 3 are sealed off, bit 0 will not be modified in this instruction. In instruction 8, strand S_5 is detached, leaving the domains in the bit open. In instruction 9, strands (2 3) and (4 5 6 7), which represent 0, are written to the bit containing open domains.

In the final two instructions, we write 1 to the first bit in pair (0, 1). In instruction 10, 3 strands are issued to each pair of bits: ($S_2^* 6^* 7^* 1^*$), ($S_3^* 1^* 2^* 3^*$) and ($S_4^* 4^* 5^* 6^* 7^* 1^*$). S_2 , S_3 and S_4 are detached through these strands. Since S_4 covers the bit 0 in pair (0, 1), after this step, domain 3 and 4 are exposed in these bits, ready to be written to 1. In the final step, strands (2 3), (2 3 4 5), and (6 7) are issued to each cell. Strand (2 3) and (6 7) will fix the exposed domains from strand S_2 or S_3 , and strand (2 3 4 5) will write bit 1 to the bit with domain 3 and 4 exposed. Details of the design are shown in Figure 11.

For all the pairs of (0, 0) and (1, 1), the first bit in those pairs will not be modified since the toehold 1 will be covered with S_2 or S_3 in the process.

D Detailed Implementation of the Second Level in Parallel Search

Here we discuss the *second* level of the parallel search operation. The first level of search operation uses the instructions that were described in Section 3.1, except we now only issue strands to non-overlapping bit pairs. We use identifiers $A_0 = 00$, $A_1 = 01$, $A_2 = 10$, $A_3 = 11$ to represent symbols in this level. For instance, to search for the target string 1011, we search for the symbol A_2 in odd symbols and A_3 in even symbols. The cases of A_2 in even symbols and A_3 in odd symbols are covered by searching with offset.

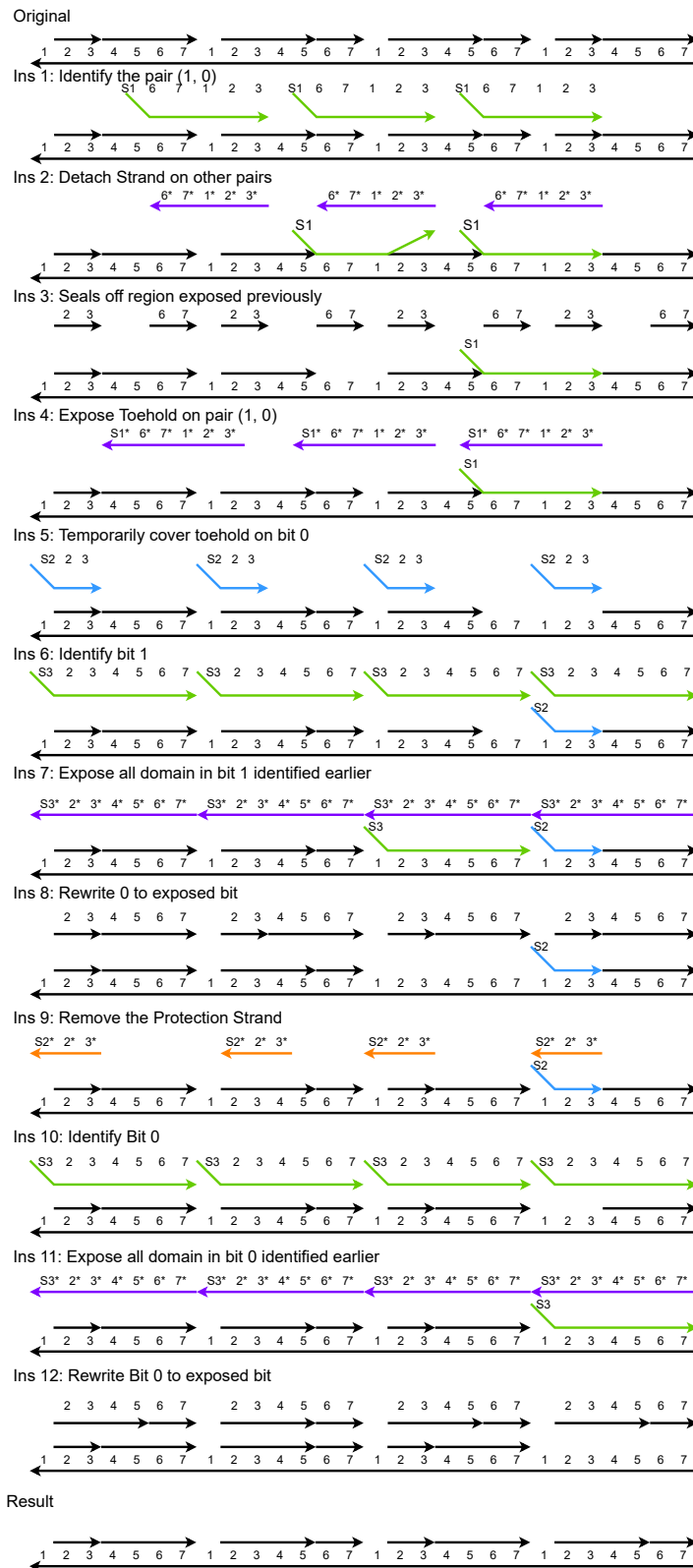
In the first instruction of the second level, we uncover the A_2 in the odd symbols, creating an open region. In instruction 2, we use a long strand to cover the entire right half of the symbol, from the start of identifier A_2 to the rightmost cell. This strand is pulled out in instruction 3. In instruction 4, we use an identifier A_2' to cover domains 5, 6, 7 in the rightmost cell while covering all other domains.

Instructions 5 to 8 are essentially the same as instructions 1 to 4, but with two significant differences. Firstly, since A_3 is the second symbol in the current level of query, we only search for even-numbered symbols (2, 4, 6, etc.). Secondly, instead of rewriting the right half

of the symbol, we write the left half. We make the new identifier A'_3 to cover domains 2, 3, 4 in the left-most cell. In instruction 9, we use identifier $(B_1 1 5 6 7 1 2 3 4)$ to recognize the two consecutive symbols A_2 and A_3 . Since, in the regular encoding, no strand starts from domain 5 or ends at domain 4, it will only form a perfect binding with a matched result.

After the identifier $B_1 1$ binds, we also need to clean up the imperfect bindings in case of a mismatch. Figure 12 shows the instructions for the cleanup process. In instruction 10, we first use the complementary strand $(5^* 6^* 7^* 1^* 2^* 3^* 4^*)$ to pull out the imperfect bond identifier $B_1 1$. Then we issue strands covering the exposed domain. We first issue strands covering fewer domains, then in following instructions, we issue strands covering more domains. As a result, we always obtain a perfect fit; the strands will not be pulled out in potential unrelated rewriting processes.

11:18 Sorting, Shifting, and Searching in DNA



■ **Figure 10** Instructions for Parallel Sorting.

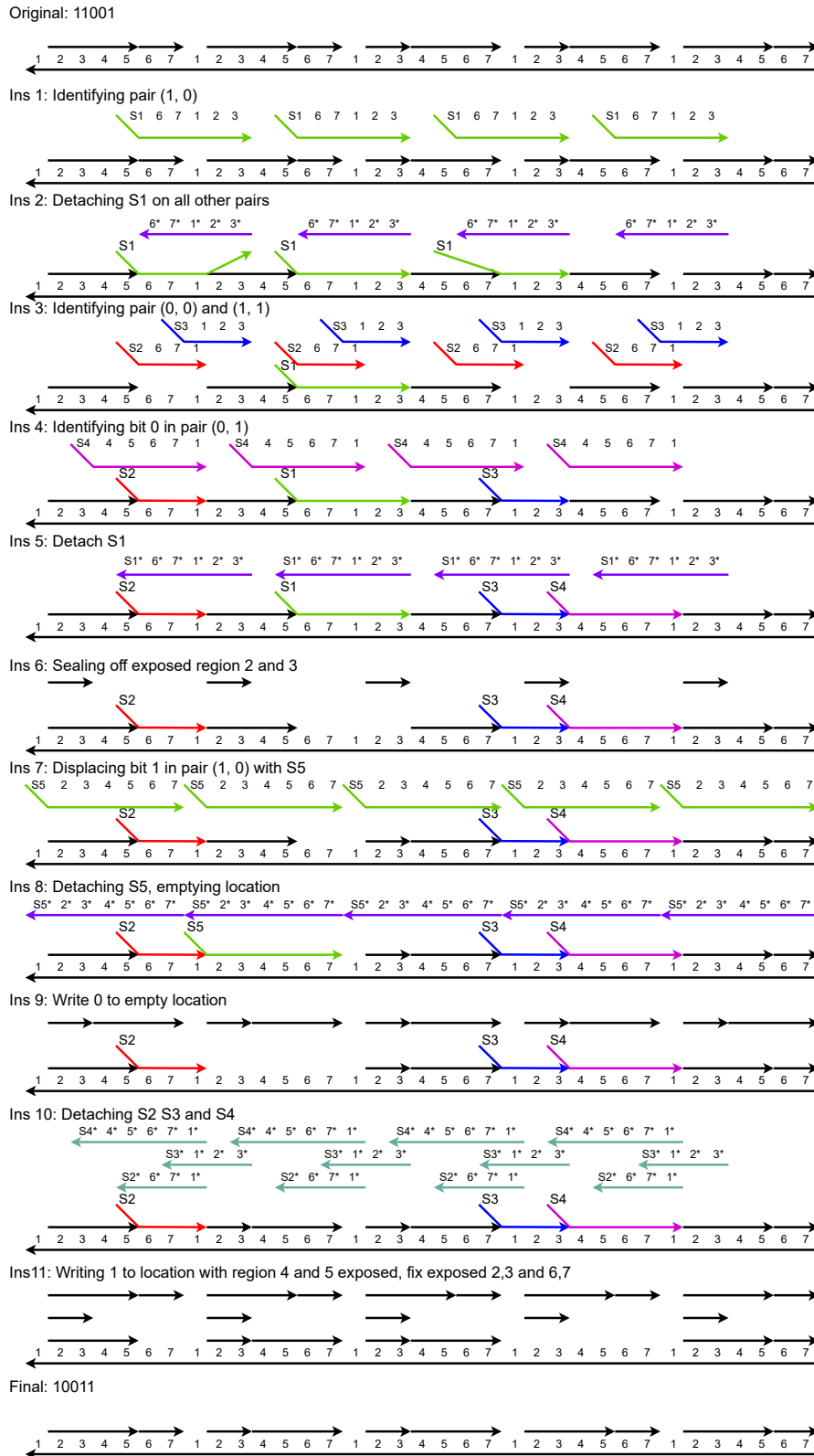
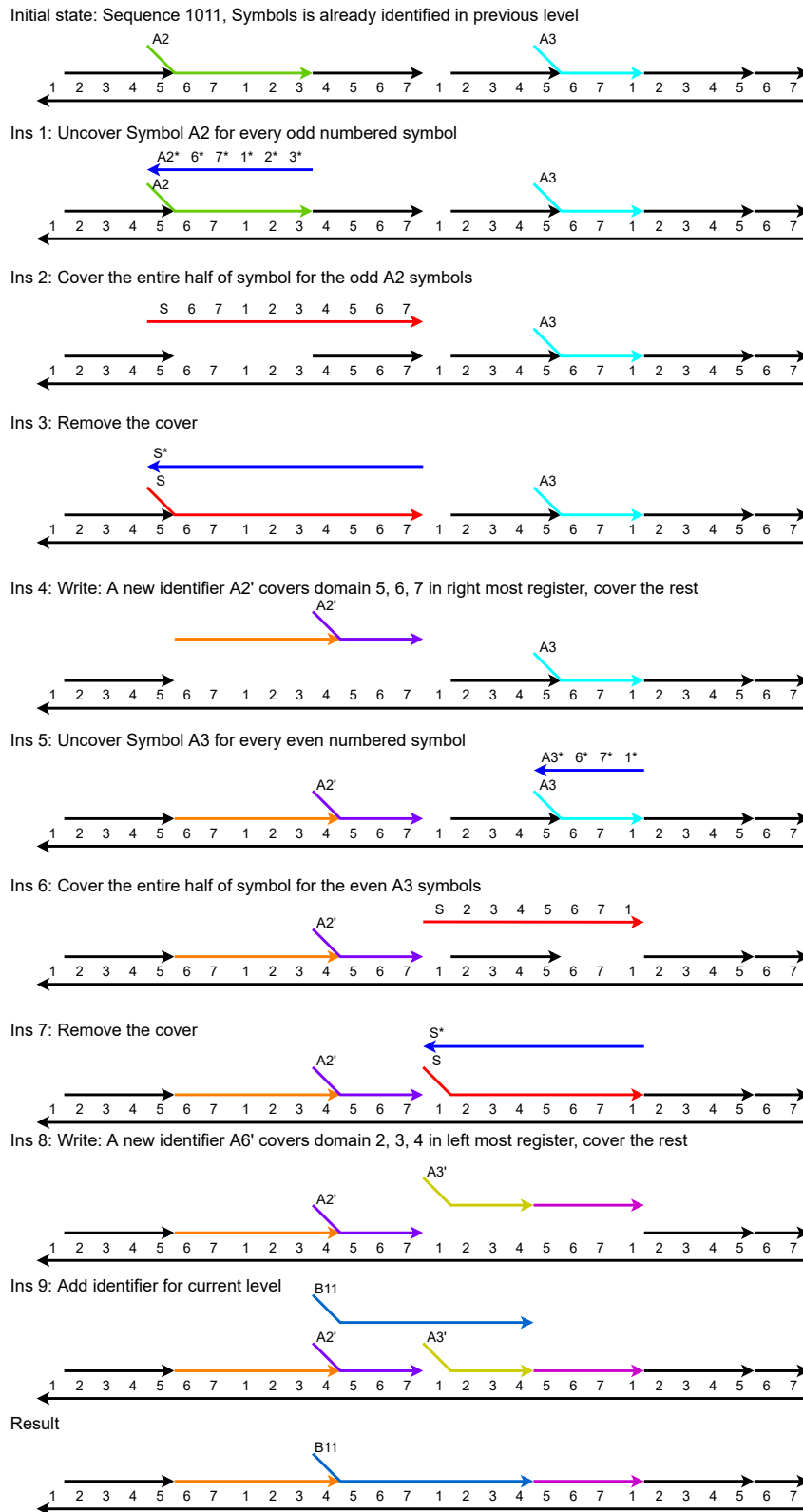


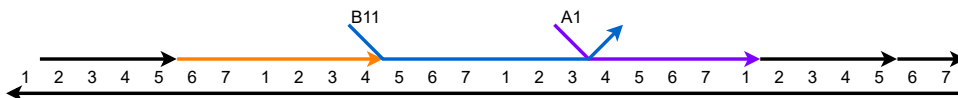
Figure 11 Instructions for the Left Shift cell.

11:20 Sorting, Shifting, and Searching in DNA

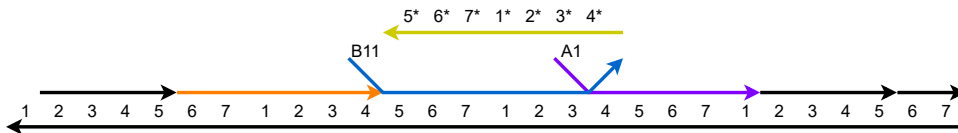


■ **Figure 12** Instructions for a search operation of target sequence 1011.

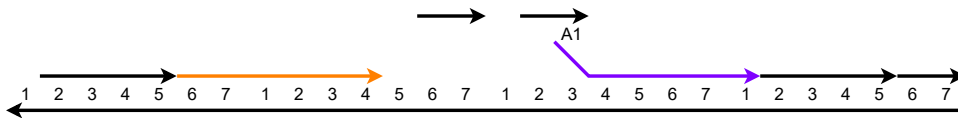
Initial state: Sequence 1010, After the identification step



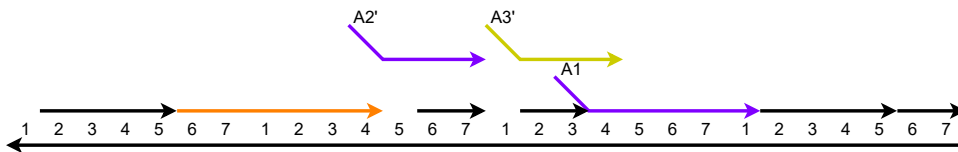
Ins 10: Pull out identifier B11 in an imperfect fit



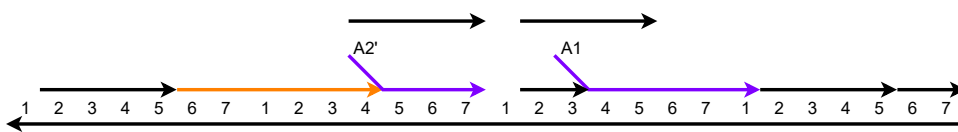
Ins 11: Cover the open domains 6, 7 or 2, 3



Ins 12: Cover the open domains 5, 6, 7 or 2, 3, 4



Ins 13: Cover the open domains 4, 5, 6, 7 or 2, 3, 4, 5



■ **Figure 13** Instructions for the clean up process for a failed searching, these instructions won't affect the result of a successful search.

