Existential Length Universality

Paweł Gawrychowski
Institute of Computer Science, University of Wrocław, Wrocław, Poland
gawry@cs.uni.wroc.pl

Martin Lange
School of Electr. Eng. and Comp. Sc., University of Kassel, Kassel, Germany
martin.lange@uni-kassel.de

Narad Rampersad
Department of Math/Stats, University of Winnipeg, Winnipeg, Canada
narad.rampersad@gmail.com

Jeffrey Shallit
School of Computer Science, University of Waterloo, Waterloo, Canada
shallit@cs.uwaterloo.ca

Marek Szykuła
Institute of Computer Science, University of Wrocław, Wrocław, Poland
msz@cs.uni.wroc.pl

Abstract

We study the following natural variation on the classical universality problem: given a language $L(M)$ represented by M (e.g., a DFA/RE/NFA/PDA), does there exist an integer $\ell \geq 0$ such that $\Sigma^\ell \subseteq L(M)$? In the case of an NFA, we show that this problem is NEXPTIME-complete, and the smallest such ℓ can be doubly exponential in the number of states. This particular case was formulated as an open problem in 2009, and our solution uses a novel and involved construction. In the case of a PDA, we show that it is recursively unsolvable, while the smallest such ℓ is not bounded by any computable function of the number of states. In the case of a DFA, we show that the problem is NP-complete, and $e^{\sqrt{n \log n}(1+o(1))}$ is an asymptotically tight upper bound for the smallest such ℓ, where n is the number of states. Finally, we prove that in all these cases, the problem becomes computationally easier when the length ℓ is also given in binary in the input: it is polynomially solvable for a DFA, PSPACE-complete for an NFA, and co-NEXPTIME-complete for a PDA.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory; Theory of computation → Problems, reductions and completeness

Keywords and phrases decision problem, deterministic automaton, nondeterministic automaton, pushdown automaton, regular expression, regular language, universality

Digital Object Identifier 10.4230/LIPIcs.STACS.2020.16

Related Version A full version of the paper is available at `https://arxiv.org/abs/1702.03961`.

Funding *Narad Rampersad*: Supported in part by a grant from NSERC.
Jeffrey Shallit: Supported in part by a grant from NSERC.
Marek Szykuła: Supported in part by the National Science Centre, Poland under project number 2017/25/B/ST6/01920.

1 Introduction

The classical universality problem is the question, for a given language L over an alphabet Σ, whether $L = \Sigma^*$. Depending on how L is specified, the complexity of this problem varies. For example, when L is given as the language accepted by a DFA M, the problem is solvable

in linear time (reachability of a non-final state) and further is NL-complete [10]. When L is given by an NFA or a regular expression, it is PSPACE-complete [1]. When L is specified by a PDA (push-down automaton) or a context-free grammar, the problem is undecidable [9].

Studies on universality problems have a long tradition in computer science and still attract much interest. For instance, the universality has been studied for visibly push-down automata [2], where the question was shown to be decidable in this model (in contrast to undecidability in the ordinary model); timed automata [3]; the language of all prefixes (resp., suffixes, factors, subwords) of the given language [14]; and recently, for partially (and restricted partially) ordered NFAs [11].

In this paper, we study a basic variation of the universality problem, where instead of testing the full language, we ask whether there is a single length that is universal for the language. We focus on the following two problems:

▶ **Problem 1** (Existential length universality)**.** *Given a language L represented by a machine M of some type (DFA/RE/NFA/PDA) over an alphabet Σ of a fixed size, does there exist an integer $\ell \geq 0$ such that $\Sigma^\ell \subseteq L$?*

▶ **Problem 2** (Specified-length universality)**.** *Given a language L by a machine M of some type (DFA/RE/NFA/PDA) over an alphabet Σ of a fixed size and an integer ℓ (given in binary), is $\Sigma^\ell \subseteq L$?*

Furthermore, if such an ℓ exists, we are interested in how large the smallest ℓ can be.

▶ **Definition 3.** *The* minimum universality length *of a language L over an alphabet Σ is the smallest integer $\ell \geq 0$ such that $\Sigma^\ell \subseteq L$.*

1.1 Motivation

From the mathematical point of view, Problems 1 and 2 are natural variations of universality that surprisingly, to the best of our knowledge, have not been thoroughly investigated in the literature. Both problems can be seen as an interesting generalization of the famous Chinese remainder theorem to languages, in the sense that given periodicities with multiple periods, we ask where all these periodicities coincide. Hence, languages stand as succinct representations of integers. Moreover, both problems are motivated by potential applications in verification listed below. Finally, the techniques developed to study them are interesting on their own and are likely to find applications elsewhere. In particular, to solve the case of an NFA, we develop a novel formalism that helps to build NFAs with particular properties. Indeed, similar constructions to some of the first ingredients in our proof (variables and the Incrementation Gadget), were recently independently discovered to solve the problem of a maximal chain length of the Green relation components of the transformation semigroup of a given DFA [5].

Games with imperfect information

We consider games with imperfect information on a labeled graph [4], which are used to model, e.g., reactive systems. In such a game there are two players, P modeling the program and E modeling the environment. The game starts at the initial vertex. In one round, P chooses a label (action) and E chooses an edge (effect) from the current vertex with this label. If P is deterministic and cannot see the choices of E, then a strategy for P is just a word over the alphabet of labels. The game can end under various criteria, and the sequence of the resulting labels determines which player wins.

Under one of the simplest ending criteria, the game lasts for a given number of steps known to P. This models the situation when we are interested in the status of the system after some known amount of time. For example, this occurs if a system must work effectively for a specified duration, but in the end, we must be able to shut it down, which requires that there are no incomplete processes still running inside. Another example could be a distributed system, where processes have limited possibilities to communicate with the others and are affected by the environment; in general, processes do not know if the system has reached its global goal; hence, for a given strategy, we may need a guarantee of reaching the goal after a known number of rounds, instead of monitoring termination externally.

The main question for such a game is: does P have a winning strategy? This is equivalent to asking whether all strategies of P are losing. In other words, if L is the language of all losing strategies of P, then we ask whether all words of the given length are in L. For instance, if the winning criterion for P is just being in one of the specified vertices, then L is directly defined by the NFA obtained from the graph, where we mark all the non-specified vertices to be final. We can also ask whether the system is unsafe, i.e., we cannot find a strategy for some number of steps, which corresponds to existential length universality.

One can mention the relationship with the "firing squad synchronization problem" [12, 13]. In this classical problem, we are given a cellular automaton of n cells, with one active cell, and the goal is to reach a state in which all cells are simultaneously active. Thinking of an evolving device where the state at time i is represented by the strings of length i in L, our problem concerns how many time steps are needed until "all cells are active", that is, until all strings of length i are accepted.

Formal specifications

Our problems are strongly related to a few other questions that can be applied in formal verification, where program correctness is often expressed through inclusion problems [17]. The universality problem is closely related to the inclusion problem: clearly, universality is a special case of inclusion if the underlying language model can express Σ^*; and inclusion can be reduced to universality when this model is closed under unions and includes some (simple) regular languages (possibly folklore, cf. [6]). These reductions carry through to the *given* and the *existential length inclusion* problems. We can consider the constrained inclusion problems corresponding to the universality problems mentioned above; for instance, *existential length inclusion* asks for two languages K and L whether $K \cap \Sigma^\ell \subseteq L$ for some ℓ. Specified-length inclusion contains the essence of a specialized program verification problem: do all program runs *of a particular duration* adhere to a given specification? Likewise, existential length universality can be used to check whether there is some number ℓ such that running the given program for exactly ℓ steps ensures that the specification is met.

Another question of potential interest in program verification is the *bounded-length universality* problem, i.e., whether $\Sigma^{\leq \ell} \subseteq L$ for a given ℓ, resp. its inclusion variant. This could then be used to check whether an implementation meets its specification up to some point, in order to know, for example, whether the safety of a program can be guaranteed for as long as it is terminated externally at some point. The complexity of bounded-length universality is the same as that of specified-length universality, which is easy to show by modifying our proofs.

Table 1 Computational complexity of universality problems.

	DFA	RE	NFA	PDA
Universality	NL-c	PSPACE-c	PSPACE-c	Undecidable
Existential length universality (Problem 1)	NP-c	PSPACE-hard, in NEXPTIME	NEXPTIME-c	Undecidable
Specified-length universality (Problem 2)	PTIME	PSPACE-c	PSPACE-c	co-NEXPTIME-c
Minimal universality length	subexponential	*open*	doubly exponential	uncountable

1.2 Contribution

We have studied the problems in four cases, when L is represented by a DFA, NFA, regular expression, or PDA.

In the case where M is a DFA, existential length universality is NP-complete, and there exist n-state DFAs for which the minimal universality length is of the form $e^{\sqrt{n \log n}(1+o(1))}$, which is the best possible even when the input alphabet is binary. Specified-length universality is solvable in polynomial time.

The case of existential length universality where M is an NFA was formulated as an open question in May 2009, as mentioned in [15], and our solution requires the most involved construction of all problems studied in this paper. We show that this problem is NEXPTIME-complete.

In the case when M is a regular expression, existential length universality is PSPACE-hard and in NEXPTIME, and there are examples where the minimal universality length is exponential. The question about the exact complexity class remains open in this case. Specified-length universality for REs and also for NFAs is PSPACE-complete, which follows from modifying the PSPACE-hardness proof of the usual universality [1, Section 10.6].

Finally, in the case where M is a PDA, existential length universality is recursively unsolvable, while the minimal universality length grows faster than any computable function, which follows from the undecidability of the universality of a PDA. On the other hand, specified-length universality is co-NEXPTIME-complete, which we show by another original construction[1], though less involved than that for NFAs, reducing from an exponential variant of the tiling problem [16].

While for proving hardness we use larger alphabets, a standard binarization applies to our problems, so all the complexity results remain valid when the alphabet is binary.

Our results are summarized in Table 1. In the conference version, we present only the most involved case of the NEXPTIME-completeness of the existential length universality of an NFA. Due to the length of the proof, the main ideas are exposed, with some technical details omitted. These, and all the other proofs, are available in the full version of the paper.

2 The Case Where M is an NFA

The classical universality problem for regular expressions and so for NFAs is known to be PSPACE-complete [1, Section 10.6]. Also, if the NFA does not accept Σ^*, then the length of the shortest non-accepted words is at most exponential. Specified-length universality for NFAs is also PSPACE-complete, which can be shown by a modification of the usual proof for universality. However, we show that existential length universality is harder: it

[1] We thank an anonymous referee for pointing out that coNEXPTIME-hardness of given-length universality for PDA could also be obtained through a modification of the proof of [18, Theorem 8.1].

is NEXPTIME-complete, and there are examples where the minimal universality length is approximately doubly exponential in the number of states of the NFA.

We begin with upper bounds, which follow by determinization and the results for DFAs.

▶ **Proposition 4.** *Let M be an NFA with n states. If there exists an ℓ such that M accepts all strings of length ℓ, then the smallest such ℓ is $\leq e^{2^{n/2}\sqrt{n \log 2}(1+o(1))}$.*

▶ **Proposition 5.** *Existential length universality (Problem 1) for NFAs is in NEXPTIME.*

The difficult part is to show that existential length universality for NFAs is NEXPTIME-hard. Note that the usual method which is applied to show PSPACE-hardness of the classical universality problem does not seem enough in this case and, after a suitable modification, results only in a proof of PSPACE-hardness. The reason for this difficulty is that, to show NEXPTIME-hardness, we need to be able to construct NFAs whose minimum universality length is larger than exponential, which is a non-trivial task in itself. The NFA constructed by the reduction must have a polynomial size, whereas to solve an NEXPTIME-complete problem we may need exponential memory. If the length of a word is superexponential, then some subsets of the states of the NFA must be repeated multiple times.

To overcome this major technical hurdle we construct an NFA with minimum universality length being roughly doubly exponential by using an indirect approach. We design the automaton such that there are many exponentially long cycles on subsets of the states and it accepts all words only for the lengths that are solutions given by the Chinese Remainder Theorem for these cycle lengths. By exhibiting a family of NFAs with large minimal universality lengths we show that our construction is essentially tight. The techniques are rather involved, and hence we first develop an intermediate formalism that will be used for both tasks. Having developed our formalism, we are able to solve the first task. To solve the second task, we proceed in two steps. First, we reduce an auxiliary logic decision problem concerning the divisibility of integers to existential length universality through our formalism. Second, we reduce a canonical NEXPTIME-complete problem to this divisibility problem.

2.1 A Programming Language

We define a simple programming language that will be used to construct NFAs with particular properties in a convenient way. Our language is nondeterministic, i.e., the programs can admit many possible computations of the same length. In contrast to the usual programming languages, we are interested only in this set of admitted computations by the program.

A program will be translated in polynomial time directly to an NFA, or, more precisely, to an extended structure called a *gadget*, which is defined below. A computation of the program will correspond to a word for the constructed NFA. If the computation is not admitted, the word will be always accepted. Otherwise, usually, the word will not be accepted, with some exceptions when we additionally make some states final in the NFA.

2.1.1 Gadget Definition

Let $m \geq 1$ be a fixed integer. A *variable* V is a set of states $\{v_1, \ldots, v_m, \bar{v}_1, \ldots, \bar{v}_m\}$. These states are called *variable states*, and m is the *width* of the variable. Besides variable states, in our NFAs there will be also *control flow states*, and the unique special final state q_{acc}, which will be fixed by all transitions.

A *gadget* G is a 7-tuple $(P^G, \mathcal{V}^G, \Sigma^G, \delta^G, s^G, t^G, F^G)$. When specifying the elements of such a tuple, we usually omit the superscript if it is clear from the context. P is a set of control flow states, $\mathcal{V}$ is a set of (disjoint) variables on which the gadget *operates*, $s, t \in P$

are distinguished *start* and *target* control flow states, respectively, and $F \subseteq P$ is a set of final states. The set of states of G is $Q = \{q_{\text{acc}}\} \cup P \cup \bigcup_{V \in \mathcal{V}} V$. Then $\delta \colon Q \times \Sigma \to 2^Q$ is the transition function, which is extended to a function $2^Q \times \Sigma^* \to 2^Q$ as usual. We always have $\delta(q_{\text{acc}}, a) = \{q_{\text{acc}}\}$ for every $a \in \Sigma$.

The NFA of G is $(Q, \Sigma, \delta, s, F)$. A *configuration* is a subset $C \subseteq Q$. Given a configuration C, we say a state is *active* if it belongs to C. We say a configuration C is *proper* if it does not contain q_{acc}. Given a proper configuration C and a word w, we say that w is a *proper computation from* C if the obtained configuration after reading w is also proper, i.e., $\delta(C, w)$ is proper. Therefore, from a non-proper configuration we cannot obtain a proper one after reading any word since q_{acc} is always fixed, and so every non-proper computation from $\{s\}$ is an accepted word by the NFA.

We say that a variable V is *valid* in a configuration $C \subseteq Q$ if for all $1 \leq i \leq m$, $v_i \in C$ if and only if $\bar{v}_i \notin C$. In other words, the states $\bar{v}_1, \ldots, \bar{v}_m$ are complementary to the states $v_1, \ldots, v_m$. A valid variable stores an integer from $\{0, \ldots, 2^m - 1\}$ encoded in binary; the states v_1 and v_m represent the least and the most significant bit, respectively. Formally, if V is valid in a configuration C, then its *value* $V(C)$ is defined as $V(C) = \sum_{\substack{1 \leq i \leq m \\ v_i \in V \cap C}} 2^{i-1}$.

We say that a configuration C is *initial* for a gadget G if it is proper, contains the start state s but no other control flow states, and the gadget's variables are valid in C (if not otherwise stated, which is the case for some gadgets). A *final* configuration is a proper configuration that contains the target state t and no other control flow states. A *complete computation* is a proper computation from an initial configuration to a final configuration. Every gadget will possess some properties about its variables and the length of complete computations according to its semantics. These properties are of the form that, depending on an initial configuration C, there exists or not a complete computation of some length from C to a final configuration C', where C' also satisfies some properties. Usually, proper computations from an initial configuration will have bounded length (but not always, as we will also create cycles). Also usually, proper configurations will have exactly one active control flow state (with the exception of the Parallel Gadget, introduced later). If a variable is not required to be valid in C, then these properties will not depend on its active states in C.

We start from defining *basic* gadgets, which are elementary building blocks, and then we will define *compound* gadgets, which are defined using the other gadgets inside.

2.1.2 Basic Gadgets

Selection Gadget. This gadget is denoted by $\textsc{Select}(V)$, where V is a variable. It allows a nondeterministic selection of an arbitrary value for V. An initial configuration for this gadget does not require that V is valid. For every integer $c \in \{0, \ldots, 2^m - 1\}$ and for every initial configuration C, there exists a complete computation from C to a final configuration C' such that $V(C') = c$.

The gadget is illustrated in Fig. 1. It consists of control flow states $P = \{s = p_0, p_1, \ldots, p_{m-1}, p_m = t\}$, one variable V, and letters $\Sigma = \{\alpha_0, \alpha_1\}$. The letters α_0 and α_1 allow moving the active control flow state over the states $p_0, p_1, \ldots, p_m$ and, at each transition, choosing either v_1 or $\bar{v}_1$ to be active. Also, each v_i and $\bar{v}_i$ are shifted to v_{i+1} and $\bar{v}_{i+1}$, respectively, and both v_m and $\bar{v}_m$ are mapped to no state ($\emptyset$), which ensures that the initial content of V is neglected. Note that a word $w = \alpha_{b_1} \ldots \alpha_{b_m}$, for $b_i \in \{0, 1\}$, sets the value of the variable to $\sum_{1 \leq i \leq m} 2^{i-1} b_i$.

The semantic properties are summarized in the following

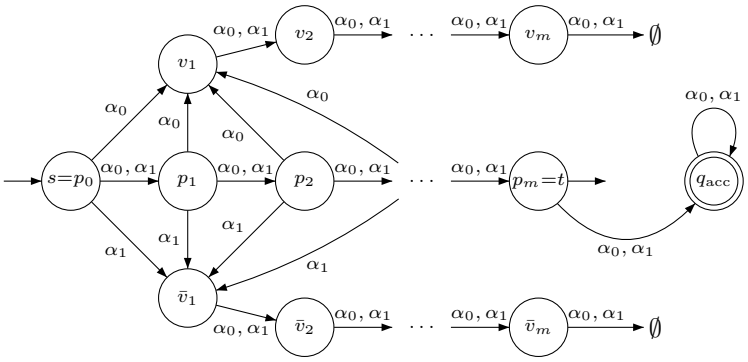

Figure 1 Selection Gadget.

▶ **Lemma 6.** *Let C be an initial configuration for the Selection Gadget* Select(V)*. For every value $c \in \{0, \ldots, 2^m - 1\}$, there exists a complete computation in Σ^m from C to a proper configuration C' such that $V(C') = c$. Every complete computation has length m, and every longer computation is not proper.*

Equality Gadget. This gadget is denoted by $U = V$, where U and V are two distinct variables. It checks if the values of valid variables U and V are equal in the initial configuration. If so, the gadget admits a complete computation, which is of length m; otherwise, every word of length at least m is a non-proper computation.

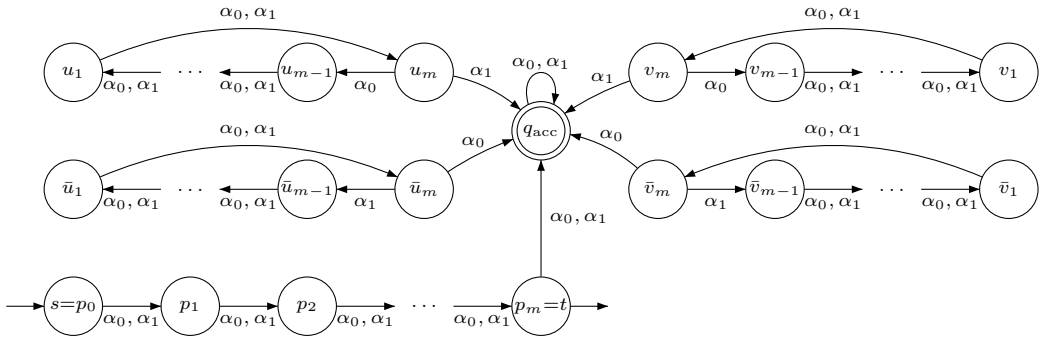

Figure 2 Equality Gadget.

The gadget is illustrated in Fig. 2. It consists of control flow states $P = \{s = p_0, p_1, \ldots, p_m = t\}$, two variables U and V, and letters $\Sigma = \{\alpha_0, \alpha_1\}$. The letters α_0 and α_1 allow moving the active control flow state over the states $s = p_0, p_1, \ldots, p_m = t$ and, at each transition, checking if the corresponding positions of U and V agree.

Inequality Gadget. This gadget is denoted by $U \neq V$, where U and V are two distinct variables. It checks if the values of the valid variables U and V are different. The construction is very similar to the Equality Gadget and its complete computations have length $m + 1$.

Incrementation Gadget. This gadget is denoted by $V{+}{+}$, where V is a variable. It increases the value of the valid variable V by 1. If the value of V is the largest possible $(2^m - 1)$, then the gadget does not allow to obtain a proper configuration by any word of length $m + 1$. Variable V must be valid in an initial configuration. The construction is similar to Equality and Inequality Gadgets and enforces a written addition of one to the value of V interpreted in binary. All complete computations have length $m + 1$.

Assignment Gadget. This gadget is denoted either by $U \leftarrow c$ or by $U \leftarrow V$, where $c \in \{0, \ldots, 2^m - 1\}$ and U and V are two distinct variables. It assigns to U either the fixed constant c or the value of the other variable V. Variable V must be valid in an initial configuration, but U does not have to be. It consists of control flow states $P = \{s, t\}$, a variable U or two variables U, V, and the unary alphabet $\Sigma = \{\alpha\}$. The transitions of α map s to t, and additionally map either s to the states of U encoding value c or the states of V to the corresponding states of U.

In fact, the case $U \leftarrow V$ could be alternatively implemented by a Selection Gadget followed by an Equality Gadget, although it will add more states and letters.

Waiting Gadget. This gadget is denoted by $\textsc{Wait}_D$, where D is a fixed positive integer. This is a very simple gadget which just does nothing for D number of letters. There is exactly one complete computation, which has length D.

2.1.3 Joining Gadgets Together

Compound gadgets are defined by other gadgets, which are joined together in the way specified by a program. The method of definition is the only difference, as compound gadgets are objects of the same type as basic gadgets. They will be also used incrementally to define further compound gadgets.

The general scheme for creating a compound gadget by joining gadgets $G_1, \ldots, G_k$ operating on variables from the sets $\mathcal{V}_1, \ldots, \mathcal{V}_k$ (all of width m), respectively, is as follows:

1. There are fresh (unique) control flow states of the gadgets, and there are the variables from $\mathcal{V}_1 \cup \cdots \cup \mathcal{V}_k$. Thus when gadgets operate on the same variable, its states are shared.
2. The alphabet contains fresh (unique) copies of the letters of the gadgets.
3. Final states in the gadgets are also final in the compound gadget.
4. The transitions are defined as in the gadgets, whereas the transitions of a letter from a gadget G_i map every control flow state that does not belong to G_i to $\{q_{\text{acc}}\}$ and fix the states of the variables on which G_i does not operate.
5. Particular definitions of compound gadgets may additionally identify some of the start and target states of the gadgets and may add more (fresh) control flow states and letters.

In our constructions, the control flow states with their transitions will form a directed graph, where the out-degree at every state of every letter is one (except the Parallel Gadget, defined later, which is an exception from the above scheme) – it either maps a control flow state to another one or to q_{acc}. States of variables will never be mapped to control flow states. This will ensure that during every proper computation from an initial configuration, exactly one control flow state is active. The active control flow state will determine which letters can be used by a proper computation, i.e., the letters from the gadget owning this state (but in which it is not the target state).

Moreover, we will ensure that whenever a proper computation activates the start state of an internal gadget G_i, the current configuration restricted to the states of G_i is an initial configuration for G_i – this boils down to assuring that the variables required to be valid have been already initialized (e.g., by a Selection Gadget or an Assignment Gadget). Hence, complete computations for the compound gadget will contain complete computations for the internal gadgets, and the semantic properties of the compound gadget are defined in a natural way from the properties of the internal gadgets.

Now we define basic ways to join gadgets together. Let $G_1, \ldots, G_k$ be some gadgets with start states $s^{G_1}, \ldots, s^{G_k}$ and target states $t^{G_1}, \ldots, t^{G_k}$, respectively.

Sequence Gadget. For each $i = 1, \ldots, k-1$, we identify the target state t^{G_i} with the start state $s^{G_{i+1}}$. Then s^{G_1} and t^{G_k} are respectively the start and target states of the Sequence Gadget. We represent this construction by writing $I_1 \ \ldots \ I_k$. Complete computations for this gadget are concatenations of complete computations for the internal gadgets.

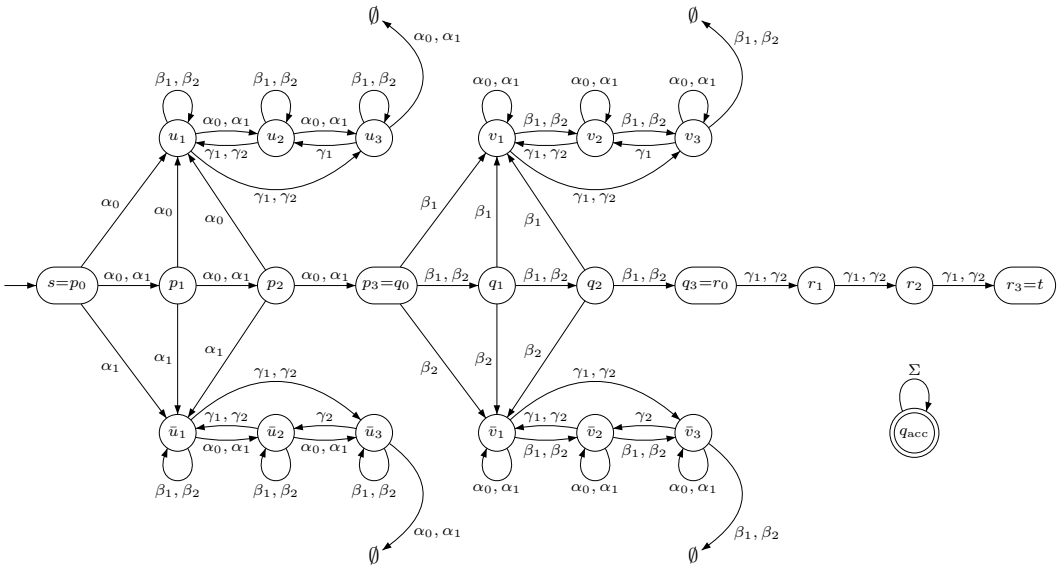

Figure 3 The complete NFA of the Sequence Gadget SELECT(U) SELECT(V) $U = V$. All omitted transitions go to q_{acc}. The states p_i, q_i, r_i and letters $\alpha_i, \beta_i, \gamma_i$ belong to the three gadgets, respectively, that is: $p_i = p_i^{\mathrm{SELECT}(U)}$, $\alpha_i = \alpha_i^{\mathrm{SELECT}(U)}$, $q_i = p_i^{\mathrm{SELECT}(V)}$, $\beta_i = \alpha_i^{\mathrm{SELECT}(V)}$, $r_i = p_i^{U=V}$, $\gamma_i = \alpha_i^{U=V}$.

For example, for $k = 3$ and $m = 3$, the Sequence Gadget SELECT(U) SELECT(V) $U = V$ is shown in Fig. 3. It has the property that every complete computation has length $3m = 9$ and is a concatenation of complete computations for the three gadgets; a final configuration C' is such that $U(C') = V(C')$. There exists a complete computation for every possible value of both variables, and longer computations are not proper.

Choose Gadget. This gadget allows selecting one of the given gadgets nondeterministically. We add a fresh start state s and k unique letters $\alpha_1, \ldots, \alpha_k$. The action of a letter α_i maps s to $\{s^{G_i}\}$, maps control flow states from the gadgets to $\{q_{\mathrm{acc}}\}$, and fixes variables states. All target states t^{G_i} are identified into the target state t of the Choose Gadget. We represent this construction by: **choose:** I_1 **or:** $\ldots$ **or:** I_k **end choose**.

Note that for this gadget there may exist complete computations of different lengths, even for the same initial configuration. Nevertheless, there exists an upper bound on the length such that every computation longer than this bound is not proper (which is 1 plus the maximum from the bounds for the internal gadgets).

Using the above constructions, we can easily develop **If-Else Gadget** and **While Gadget** (which use Equality or Inequality Gadgets for checking conditions). A special version of the latter is **While-True Gadget**, which creates an unconditional loop by identifying the start and target states of the internal gadget.

Then we can define **Addition Gadget** and **Multiplication Gadget** for performing the corresponding arithmetic operations. We also need **Primality Gadget** testing if the value of a variable is prime (there is also a negated version), and a **Prime Number Gadget**

computing the i'th prime number. For each of the defined gadgets so far, except for While Gadget in general, there always exists an upper bound on the length of every complete computation, and every longer computation is not proper. This bound is at most exponential in the size of the gadget, because proper computations cannot repeat the same configuration.

2.2 An NFA With a Large Minimal Universality Length

Our first application is to show a lower bound on the maximum minimal universality length. Alg. 1 gives the program encoding our NFA. The numbers in the brackets [ ] at the right denote the length of complete computations for the gadget (or for a part of it) in the current line. In line 7, the annotation indicates that the start control flow state of this Waiting Gadget is final, so the NFA of the program has two final states in total.

Algorithm 1 Large minimal universality length.

Variables: X, Y

```
 1: SELECT(Y)                                   ▷ [m]
 2: X ← 0                                       ▷ [1]
 3: while true do
 4:     choose:                                 ▷ [1]
 5:         X = Y                               ▷ [m]
 6:         X ← 0                               ▷ [1]
 7:         [start final state] WAIT_{m+1}      ▷ [m+1]
 8:     or:
 9:         X ≠ Y                               ▷ [m+1]
10:         X++                                 ▷ [m+1]
11:     end choose
12: end while
```

The idea of the program is as follows: In the beginning, we select an arbitrary value for Y, and then in an infinite loop, we increment X modulo $Y + 1$. The Choose Gadget in lines 4–11 is, in fact, an If-Else Gadget with condition $X = Y$, unrolled for easier calculation of computation lengths. Every iteration (complete computation of the Choose Gadget) of the loop takes the same number of letters $(2m + 3)$, hence given a computation length we know that we must perform $d - 1$ complete iterations and end in the d'th iteration, for some d. A proper computation of this length can avoid the final state in line 7 only in the iterations where the value of X does not equal the value of Y. We can ensure this for every length smaller than $\mathrm{lcm}(1, 2, \ldots, 2^m) \cdot (2m + 3)$, as we can always select Y such that $Y + 1$ does not divide $d + 1$, but it is not possible for length $\mathrm{lcm}(1, 2, \ldots, 2^m) \cdot (2m + 3)$. After a detailed technical analysis and a calculation we get:

▶ **Theorem 7.** *For a* 15*-letter alphabet, the minimal universality length can be as large as* $e^{2^{n/11}(1+o(1))}$.

2.3 Controlling the Computation Length

As we noted, because of Choose Gadgets, complete computations may have different lengths. For example, the Addition Gadget for an initial configuration C performs $V(C)$ iterations of its internal while loop. Moreover, two or more branches of a Choose Gadget may admit complete computations of different lengths even for the same current configuration. This is an obstacle that makes it difficult or impossible to further rely on the exact length $|w|$ of a proper computation, based on which we would like to decide if w must be accepted. Therefore, if we want to still use our constructions, we need a possibility to ensure that all complete computations have a fixed known length, and furthermore, that there are no proper computations longer than that length.

Delaying Gadget. The first new ingredient is the Delaying Gadget $\textsc{Delay}_D$, where D is a fixed integer ≥ 0. It is a stronger version of Waiting Gadget. Using a While Gadget with an Inequality Gadget and an Incrementation Gadget, it enforces proper computations that incrementally count from 0 to D. Complete computations have length exactly $T(D)$, which is a function polynomial in D and exponential in the size of the gadget.

Parallel Gadget. The idea to control the computation length is to implement computation in parallel. A given gadget for which there may exist complete computations of different lengths is computed in parallel with a Delaying Gadget. When the computation is completed for the given gadget, we still must wait in its target state until the computation for the Delaying Gadget is also finished. In this way, as long as complete computations for the Delaying Gadget are always longer than those for the given gadget (we can ensure this by choosing D), complete computations for the joint construction will have fixed length $T(D) + 1$. The joint computation is realized by replacing the alphabet with new letters for every combined pair of actions in both gadgets.

2.4 Length Divisibility

In Subsection 2.2 we have constructed an NFA for which every word encoding a proper computation must be accepted or can be not accepted depending on its length, namely, it is always accepted if the length is divisible by $\operatorname{lcm}(1, 2, \ldots, 2^m) \cdot \mathcal{O}(m)$. We generalize this idea so that we will be able to express more complex properties about the length of which all words must be accepted.

We are going to test whether $|w|$ satisfies some properties, in particular, whether a function of $|w|$ is divisible by some integers. This extends the idea from Alg. 1, which just verifies whether $|w|/r$, for some constant r, is not divisible by some integer from 2 to 2^m. We define the *divisibility program* shown in Alg. 2. It is constructed for given numbers k and m, and a *verifying procedure*, which is any gadget satisfying some technical properties: Complete computations must be exponentially bounded by some L, longer computations must not be proper, and all outgoing transitions from the target state must go to q_{acc}; these conditions will allow synchronizing the length of all complete computations to one length $T(D) + 1 > L$ with a Parallel Gadget. Furthermore, the values of variables X_i and X'_i may not be modified and the existence of complete computations cannot depend on the setting of any internal variables in the initial configuration; these conditions ensure that the gadget can be activated repetitively in the same proper computation of the whole program. Finally, there may not be final states.

There is an infinite while loop, which consists of two parts. In the first part, a nondeterministic choice is made (line 4): either to run the verifying procedure or to wait. For the verifying procedure (line 5) we use the Parallel Gadget; this ensures that this part finishes after exactly $T(D) + 1 > L$ letters. In the waiting case (line 7), we use the Delaying Gadget with the same value of D as that in the Parallel Gadget. Then there is a single final state (line 8). In the second part (lines 10–15), every variable X'_i counts the number of iterations of the while loop modulo X_i. The *for* loop denotes that the body is instantiated for every i (it is a Sequence Gadget). Every complete computation of the second part (lines 10–15) has exactly $(2m + 3)k$ letters.

The idea is that, for certain lengths, every proper computation must end with a configuration with the non-final control flow states in line 5 (these are precisely the two target states, of the verifying procedure and of the Delaying Gadget) or with the final state in line 8. However, for the first option, it must succeed in the last iteration with the verifying procedure

Algorithm 2 Divisibility program.

Variables: $X_1, \ldots, X_k, X'_1, \ldots, X'_k$

1: $\textsc{Select}(X_1), \ldots, \textsc{Select}(X_k)$. ▷ $[km]$
2: $X'_1 \leftarrow 0, \ldots, X'_k \leftarrow 0$ ▷ $[k]$
3: **while true do**
4: **choose:** ▷ $[1]$
5: execute in parallel { $\textsc{Delay}_D$; Verifying procedure } ▷ $[T(D)+1]$
6: **or:**
7: $\textsc{Delay}_D$ ▷ $[T(D)]$
8: [final state] $\textsc{Wait}_1$ ▷ $[1]$
9: **end choose**
10: **for** $i = 1, \ldots, k$ **do**
11: X'_i++ ▷ $[m+1]$
12: **if** $X'_i = X_i$ **then** ▷ $[m+1$ if $X'_i = X_i$, and $m+2$ otherwise$]$
13: $X'_i \leftarrow 0$ ▷ $[1]$
14: **end if**
15: **end for**
16: **end while**

when $X'_i = \ell' \bmod X_i$. In other words, for some selection of the values for $X_1, \ldots, X_k$, there must exist a complete computation of the verifying procedure from an initial configuration with these values for X_i and $X'_i = \ell' \bmod X_i$. Due to these auxiliary variables X'_i, the verifying procedure can check the divisibility of ℓ by X_i.

▶ **Lemma 8.** *Consider Alg. 2 for some k, m, and a verifying procedure. There exist integers $r_1 \geq 1$ and $r_2 \geq 1$ such that the NFA of Alg. 2 accepts all words of a length ℓ if and only if there exist an integer $\ell' \geq 0$ such that:*

- *$\ell = r_1 \cdot \ell' + r_2$, and*
- *for every initial configuration C of the verifying procedure where variables $X_1, \ldots, X_k$ are valid and $X'_i(C) = \ell' \bmod X_i(C)$ for all $1 \leq i \leq k$, there does not exist a complete computation for the verifying procedure.*

2.4.1 Existential Divisibility Formulas

We develop a method for verifying the properties of the computation length in a flexible way. We use a subset of first-order logic, where formulas are in a special form. For a given integer m, we say that a formula φ is in *existential divisibility* form if its only free variable is ℓ' (not necessarily occurring in φ) and it has the following form:

$$\exists_{X_1,\ldots,X_k \in \{0,\ldots,2^m-1\}} \psi(X_1, \ldots, X_k, \ell').$$

Formula ψ is any propositional logic formula that uses operators $\wedge$, $\vee$, and whose simple propositions are of the following possible forms:

1. $(X_i = c)$, where $c \in \{0, \ldots, 2^m - 1\}$,
2. $(X_h = X_i + X_j)$,
3. $(X_h = X_i \cdot X_j)$,
4. X_i is prime,
5. X_i is the X_j'th prime number,
6. $(X_i \mid \ell')$ or $(X_i \nmid \ell')$,

where X_i, X_j, X_h are some variables from $\{X_1, \ldots, X_k\}$.

Given a φ, we can ask for what integer values of ℓ' the formula is satisfied, and in particular whether it is not a tautology over positive integers.

▶ **Problem 9** (Non-satisfiability of existential divisibility formulas). *Given an existential divisibility formula φ, is there a positive integer ℓ' such that $\varphi(\ell')$ is not satisfied?*

Verifying Gadget. For the formula ψ occurring in an existential divisibility formula φ, we construct the gadget VERIFY(ψ) for verifying ψ. The gadget uses the external variables $X_1, \ldots, X_k$, which are assumed to correspond with those in ψ, and the external auxiliary variables $X'_1, \ldots, X'_k$. There are also some fresh internal variables. The value of ℓ' is not given, but instead, we assume that the value of every X'_i is equal to ℓ' mod X_i (and 0 when $X_i = 0$), hence we will be able to check the divisibility of ℓ'.

The construction uses our components designed so far. It is built using Sequence Gadgets for conjunctions, Choose Gadgets for disjunctions, and other appropriate gadgets for (1)–(6).

The construction is such that, for an initial configuration C for VERIFY(ψ) with valid variables $X_1, \ldots, X_k$ and where $X'_i(C) = \ell'$ mod $X_i(C)$, there exists a complete computation if and only if $\psi(X_1, \ldots, X_k, \ell')$ is satisfied. Note that the gadget meets the conditions of the verifying procedure in Alg. 2.

2.4.2 Reduction from Problem 9

We already have all ingredients to reduce Problem 9 to Problem 1 (existential length universality). Given an existential divisibility formula $\varphi(\ell')$, we construct the program from Alg. 2 with the Verifying Gadget VERIFY(ψ) as the verifying procedure. Hence, the formula is translated to an NFA in polynomial time. By Lemma 8, there exist integers $r_1, r_2 \geq 1$ such that the NFA accepts all words of some length ℓ if and only if for some integer $\ell' \geq 0$, $\ell = r_1 \cdot \ell' + r_2$ and $\varphi(\ell')$ is not satisfied. Hence, if φ is not satisfied for some ℓ', then the NFA accepts all words of length ℓ, and if the NFA accepts all words of a length ℓ, then ℓ must be expressible as $r_1 \cdot \ell' + r_2$ and $\varphi(\ell')$ must be not satisfied.

▶ Remark 10. With a few more technical steps, which require, e.g., adding the negation and controlling variable bounds, it is possible to represent the negated Problem 9 as the satisfiability problem of the Presburger arithmetic with the prefix class $\exists\forall^*$ and whose formulas are of a specific form. The Presburger arithmetic with the prefix class $\exists\forall^*$ is NEXPTIME-hard [7], and a little more general one with the prefix class $\exists^*\forall^*$ is Σ_1^{EXP}-complete [8]. However, our problem is a strict subclass of the first case, because the first and the only unbounded variable ℓ' can be checked only for divisibility, all the other variables are exponentially bounded, and the propositions are of particular forms. Hence, we cannot directly infer the hardness from that known result. In the last reduction step, we show that NEXPTIME-hardness still holds for our restricted problem.

2.5 Reduction to the non-satisfiability of existential divisibility formulas

In the final step, we reduce from the canonical NEXPTIME-complete problem: whether a nondeterministic Turing machine N with s states accepts the empty input after at most 2^s steps. The idea is encoding by ℓ' a $2^s \times 2^s$ table representing a proper computation of the machine. Each symbol placed at each cell has assigned a unique prime number, and we define that the symbol is present if and only if ℓ' modulo its prime number is non-zero. Then we construct an existential divisibility formula $\varphi(\ell')$ that is satisfied for an ℓ' if and only if the encoded computation by ℓ' is not correct. Thus, the formula is a disjunction of several cases that express a possible error in the computation.

This reduces (by a polynomial reduction) an NEXPTIME-complete to Problem 9, which was reduced to Problem 1 (existential length universality). Then we can further reduce to the binary case by a standard binarization.

▶ **Theorem 11.** *Existential length universality (Problem 1) for NFAs is NEXPTIME-hard, even if the alphabet is binary.*

References

1. A. V. Aho and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1974.
2. R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proc. of the 36th Annual ACM Symposium on Theory of Computing*, STOC 2004, pages 202–211. ACM, 2004.
3. N. Bertrand, P. Bouyer, T. Brihaye, and A. Stainer. Emptiness and universality problems in timed automata with positive frequency. In *Proceedings of the 38th International Conference on Automata, Languages and Programming - Volume Part II*, ICALP 2011, pages 246–257. Springer, 2011.
4. L. Doyen and J.-F. Raskin. Games with imperfect information: theory and algorithms. In Krzysztof R. Apt and Erich Grädel, editors, *Lectures in Game Theory for Computer Scientists*, pages 185–212. Cambridge University Press, 2011.
5. L. Fleischer and M. Kufleitner. Green's relations in finite transformation semigroups. In Pascal Weil, editor, *CSR*, pages 112–125. Springer, 2017.
6. O. Friedmann and M. Lange. Ramsey-Based Analysis of Parity Automata. In *TACAS*, volume 7214 of *LNCS*, pages 64–78. Springer, 2012.
7. E. Grädel. Dominoes and the Complexity of Subclasses of Logical Theories. *Annals of Pure and Applied Logic*, 43(1):1–30, 1989.
8. C. Haase. Subclasses of Presburger Arithmetic and the Weak EXP Hierarchy. In *CSL–LICS*, CSL–LICS, pages 47:1–47:10. ACM, 2014.
9. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
10. N. D. Jones. Space-bounded reducibility among combinatorial problems. *Journal of Computer and System Sciences*, 11(1):68–85, 1975.
11. M. Krötzsch, T. Masopust, and M. Thomazo. On the complexity of universality for partially ordered NFAs. In *MFCS*, pages 61:1–61:14, 2016.
12. J. Mazoyer. A six-state minimal time solution to the firing squad synchronization problem. *Theoretical Computer Science*, 50(2):183–238, 1987.
13. F. R. Moore and G. G Langdon. A generalized firing squad problem. *Information and Control*, 12(3):212–220, 1968.
14. N. Rampersad, J. Shallit, and Z. Xu. The computational complexity of universality problems for prefixes, suffixes, factors, and subwords of regular languages. *Fundamenta Informaticae*, 116(1–4):223–236, 2012.
15. J. Shallit. Open problems in automata theory: an idiosyncratic view, LMS Keynote Address in Discrete Mathematics, BCTCS 2014, April 10 2014, Loughborough, England. `https://cs.uwaterloo.ca/~shallit/Talks/bc4.pdf`.
16. P. van Emde Boas. The convenience of tilings. In *Complexity, Logic, and Recursion Theory*, pages 331–363. Marcel Dekker Inc., 1997.
17. M. Y. Vardi and P. Wolper. Reasoning About Infinite Computations. *Information and Computation*, 115(1):1–37, 1994.
18. G. Zetzsche. The complexity of downward closure comparisons. `https://arxiv.org/abs/1605.03149`, 2016.