Encoding Hard String Problems with Answer Set Programming

Dominik Köppl
Department of Computer Science, Universität Münster, Germany

Abstract

Despite the simple, one-dimensional nature of strings, several computationally hard problems on strings are known. Tackling hard problems beyond sizes of toy instances with straight-forward solutions is infeasible. To solve these problems on datasets of even small sizes, effort has to be put into the conception of algorithms leveraging profound characteristics of the input. Finding these characteristics can be eased by rapidly creating and evaluating prototypes of new concepts in how to tackle hard problems. Such a rapid-prototyping method for hard problems is answer set programming (ASP). In this light, we study the application of ASP on five NP-hard optimization problems in the field of strings. We provide MAX-SAT and ASP encodings, and empirically reason about the merits and flaws when working with ASP solvers.

2012 ACM Subject Classification Theory of computation; Computing methodologies → Artificial intelligence; Theory of computation → Discrete optimization; Hardware → Theorem proving and SAT solving

Keywords and phrases optimization problems, answer set programming, MAX-SAT encoding, NP-hard string problems

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.17

Supplementary Material *Software (Source Code)*: `https://github.com/koeppl/aspstring`
archived at `swh:1:dir:fccb46f95c8f782e61c7e63a9a154a6e2fd8dad9`

Funding Supported by JSPS KAKENHI Grant Numbers JP21K17701 and JP23H04378.

Acknowledgements We thank Mutsunori Banbara for drawing our attention to the ASP language.

1 Introduction

Despite the fact that most string problems found in literature are solvable in polynomial time or even close to linear time or beyond, there are several problems that are known to be NP-hard. Among those, we focus on five problems that are well-perceived regarding the number of publications studying these problems: Closest String (csp)[1], Closest Substring (css), Longest Common Subsequence (lcs), Minimum Common String Partition (mcsp), and Shortest Common Superstring (scs). These problems have been studied under various viewpoints. With respect to fixed-parameter tractability (FPT), Bulteau et al. [9] gave a comprehensive survey on various NP-hard problems related to strings; this survey comprises the problems studied in this paper. Also, Basavaraju et al. [2] studied the kernelization of a majority of our problems. We address other related work in the individual sections of each problem, but omit references to approximation algorithms due to their amount, and because we put focus on the *exact* solution of the aforementioned problems formulated as optimization problems.

[1] We stick to the commonly used abbreviation csp in literature despite that cs would fit better with the abbreviations of the other problems.

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).
Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 17; pp. 17:1–17:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A major problem in tackling these problems in practice is that naive solutions quickly become impractical with respect to the time complexity. Tailored algorithms[2] are hard to implement, and thus a burden on the algorithm engineering side. Our contribution is to advertise *answer set programming* (ASP) as a rapid-prototype programming tool for solving NP-hard string problems on small instances. ASP is a declarative programming language geared towards solving hard problems [40, 12]. ASP has been successfully applied in robotics [3], or for computing the n-queens and the knight's tour problem [18]. There is also a competition on ASP solvers on various classic problems addressing mainly problems on graphs [28]. See [19, 20] and the references therein for an overview of other use cases.

Although well-devised algorithms can outperform ASP-based approaches, the programming effort for writing in an expressive, declarative programming language such as ASP is considerably small. In this paper, we devise MAX-SAT encodings for the above addressed problems, and subsequently translate these encodings into the ASP language. With respect to tackling hard string problems via MAX-SAT encodings we are aware of the work of Bannai et al. [1] who studied MAX-SAT encodings for repetitiveness measures that are also known to be NP-hard.

2 Preliminaries

Common to all problems treated in this paper is the input of a set of m strings $\mathcal{S} = \{S_1, \ldots, S_m\}$. For simplicity, we assume that all strings have the same length n, and that all characters are drawn from an alphabet Σ of size $\sigma = |\Sigma|$. Hence, $|S_x| = n$ denotes the length of each input string and $S_x[i] \in \Sigma$ for all $i \in [1..n]$ and $x \in [1..m]$. Except for MCSP, the output is a string T that is object to an optimization argument with respect to the input strings (and, additionally for CSS, with respect to an integer parameter specifying the length of T).

Encoding Annotations. Beginning with the next section, we state rules and constraints with numbered equations, and add to each equation, in square brackets, the number of generated clauses and the size of each such clause. For instance, the equation

$$[\mathcal{O}(n), \mathcal{O}(1)] \quad \forall i \in [1..n] : p_i \implies p_{i+1} \tag{1}$$

defines n clauses, each of the form $(\neg p_i \vee p_{i+1})$, so its complexity is $[\mathcal{O}(n), \mathcal{O}(1)]$.

Experiments. We implemented our MAX-SAT-formulations in the ASP language, and used the solver `clingo` [26, 27][3] for evaluation. We compare the results with brute-force approaches written in the python language on randomly generated data. Our filenames are formatted like `s03m04n005i1` to denote that the alphabet size is $\sigma = 3$, the number of strings is $m = 4$, the length of each string is $n = 5$, and this file is the $i = 1$-st sample of a batch of files with the same characteristics (σ, m and n). For MCSP, we have file formats like `2s02n008i2.txt` where the prefix 2 denotes that $m = 2$ is fixed. For the MCSP files, we assume that the two strings given have the same Parikh vector. Our implementations and datasets are available at `https://github.com/koeppl/aspstring`. For the evaluation, all experiments ran single-threaded on a machine with Intel Core i3–9100 CPU and Debian 11.

[2] Meaning that such algorithms usually are based on theoretical results that can be put hardly into practice.
[3] `https://github.com/potassco/clingo`

	1	2	3	4	5	6	7	8	9	10	11	12	13
$S_1 =$	l	n	e	e	p	l	e	s	s	n	e	l	s
$S_2 =$	s	l	e	e	p	s	l	s	s	n	e	s	n
$S_3 =$	n	l	e	l	p	l	e	s	s	n	s	s	s
$S_4 =$	s	n	e	e	p	l	e	l	s	n	s	s	s
$S_5 =$	s	l	l	e	e	l	e	s	s	n	s	s	s

	1	2	3	4	5	6	7	8	9	10	11	12	13
$S_1 =$	$\boxed{\texttt{l}}$	$\boxed{\texttt{n}}$	e	e	p	l	e	s	s	n	e	$\boxed{\texttt{l}}$	s
$S_2 =$	s	l	e	e	p	$\boxed{\texttt{s}}$	$\boxed{\texttt{l}}$	s	s	n	e	s	$\boxed{\texttt{n}}$
$S_3 =$	$\boxed{\texttt{n}}$	l	e	$\boxed{\texttt{l}}$	p	l	e	s	s	n	$\boxed{\texttt{s}}$	s	s
$S_4 =$	s	$\boxed{\texttt{n}}$	e	e	p	l	e	$\boxed{\texttt{l}}$	s	n	$\boxed{\texttt{s}}$	s	s
$S_5 =$	s	l	$\boxed{\texttt{l}}$	e	$\boxed{\texttt{e}}$	l	e	s	s	n	$\boxed{\texttt{s}}$	s	s
$T =$	s	l	e	e	p	l	e	s	s	n	e	s	s

Figure 1 Example for CSP (Sect. 3) with $n = 13$. The input set $\mathcal{S} = \{S_1, \ldots, S_5\}$ is shown on the left figure. The right figure shows that the solution $T = \texttt{sleeplessness}$ has three mismatches with each of the input strings in the Hamming distance. Mismatching characters are highlighted by surrounding boxes.

3 Closest String Problem (CSP)

The Closest String Problem (CSP)[4] asks for a string T such that $\max_{x \in [1..m]} \mathrm{dist}_{\mathrm{ham}}(S_x, T)$ is minimal, where the *Hamming distance* $\mathrm{dist}_{\mathrm{ham}}$ is given by $\mathrm{dist}_{\mathrm{ham}}(S_x, T) := |\{i \in [1..n] : S_x[i] \neq T[i]\}|$. An example is shown in Fig. 1. Here, and in the following examples we stick to the alphabet $\Sigma := \{\texttt{e}, \texttt{l}, \texttt{p}, \texttt{n}, \texttt{s}\}$ with size $\sigma = 5$.

Related Work. Frances and Litman [24] and Lanctôt et al. [39] proved that CSP and its generalization, the Closest Substring Problem (CSS), are NP-hard for any alphabet with $\sigma \geq 2$ in n and m. The parameterized complexities have been surveyed in [48, Section 5.1] and [57], with focus also on CSS. For the decision problem with a Hamming distance of d, Gramm et al. [32] showed that CSP can be solved in $\mathcal{O}(mn + d^d)$ time or $2^{2^{\mathcal{O}(m \log m)}} \mathcal{O}(\log n)$ time. Regarding integer linear programming (ILP), Chimani et al. [13] gave ILP formulations, also for CSS. There is a line of research on further practical ILP formulations [16, 43, 54]. Finally, Knop et al. [38] gave also an ILP formulation and an exact algorithm running in $m^{\mathcal{O}(m^2)} \mathcal{O}(\log n)$ time.

With respect to different kinds of optimization approaches, Kelsey and Kotthoff [37] studied CSP as a constraint satisfaction problem, Huan et al. [35] provided an ant colony optimization algorithm, and Vilca and de Freitas [55] gave a specialized algorithm for fixed $m = 3$.

3.1 MAX-SAT encoding

We use the known fact that we have to select, for the i-th character of the output T, a character appearing at the i-th position of one of the input strings.

▶ **Lemma 1** ([37, Lemma 2]). *For each $i \in [1..n]$, $T[i] = S_x[i]$ for an $x \in [1..m]$.*

Let us define $\Sigma_i := \{S_1[i], \ldots, S_m[i]\}$ to be the set of characters appearing at text position i of all input strings. Then $\sigma_i := |\Sigma_i| \leq \min(m, \sigma)$, and σ_i can be much less than m or σ if the number of distinct characters is small. We can express the alphabets per position Σ_i by a Boolean matrix $M[1..n][1..\sigma]$ with $M[i][c] = 1$ if $c \in \Sigma_i$.

[4] Alternative names are, among others, Minimum Radius, Center String or Consensus String problem.

Further, we define the variables $T_{i,c} = 1$ to encode that $T[i] = c$, for $i \in [1..n], c \in \{S_1[i], \ldots, S_m[i]\}$. To state that $T[i] = S_x[i]$, we want that, for a fixed position $i \in [1..n]$, only one $T_{i,c}$ is set:

$$[\mathcal{O}(n), \mathcal{O}(\min(m, \sigma))] \quad \forall i \in [1..n] : \sum_{c \in \Sigma_i} T_{i,c} = 1 \tag{CSP1}$$

Next, we define the cost variables $C_{i,x}$ for all $i \in [1..n]$ and $x \in [1..m]$ with $C_{i,x}$ being set if $T[i] \neq S_x[i]$. Thus the Hamming distance between T and S_x is $\mathrm{dist}_{\mathrm{ham}}(T, S_x) = \sum_{i \in [1..n]} C_{i,x}$. Therefore:

$$[\mathcal{O}(nm\sigma), \mathcal{O}(1)] \quad \forall i \in [1..n], c \in \Sigma_i, x \in [1..m] : T_{i,c} \wedge S_x[i] \neq c \implies C_{i,x} \tag{CSP2}$$

A statement for setting $C_{i,x}$ to false is not needed as the optimizer will try to do so if it does not violate (CSP2). This is achieved by the following objective:

$$[\mathcal{O}(1), \mathcal{O}(mn)] \quad \text{minimize} \max_{x \in [1..m]} \sum_{i \in [1..n]} C_{i,x} \tag{CSP3}$$

Complexities. We have $\mathcal{O}(n\sigma)$ selectable variables ($T_{i,c}$), $\mathcal{O}(nm)$ helper variables ($C_{i,x}$), $\mathcal{O}(nm\sigma)$ clauses (CSP2). The largest clause contains $\mathcal{O}(mn)$ variables (CSP3).

Implementation. Our implementation in ASP is given in Listing 1. In all listings, the percent sign `%` introduces a comment until the end of the line, which we use to refer to the MAX-SAT equation that is represented by the respective line of code. Red curly arrows symbolize line breaks. If not otherwise stated, in all code listings onwards, we assume that the input is of the form `s(X,I,C)`, denoting that $S_{\mathtt{X}}[\mathtt{I}] = \mathtt{C} \in \Sigma$. We use the helper variables `mat(X,I)` to denote the existence of $S_{\mathtt{X}}[\mathtt{I}]$. For encoding (CSP3) in ASP, we additionally define the helper variables `cost` and `mcost` encoding $\sum_{i \in [1..n]} C_{i,\mathtt{X}}$ and $\max_{x \in [1..m]} \mathtt{cost}(x)$, respectively. The `#show` directives at the end define the variables the solver has to output. The evaluation for our implementation is deferred until we have introduced the CSS problem, which we conjointly evaluate in Sect. 4.2.

Listing 1 ASP for CSP (Sect. 3).

```
mat(X,I) :- s(X,I,_).
1 {t(I,C) : s(_,I,C)} 1 :- mat(_,I). %(CSP1)
c(X,I) :- t(I,C), s(X,I,A), C != A. %(CSP2)
cost(X,C) :- C = #sum {1,I : c(X,I)}, mat(X,_). %(CSP3)
mcost(M) :- M = #max {C : cost(_,C)}.
#minimize {M : mcost(M)}.
#show t/2. #show mcost/1. #show cost/2.
```

4 Closest Substring (CSS)

For the CSS problem, we additionally require a parameter λ as input to specify the length of the output string T. CSS asks for the string T with $|T| = \lambda$ such that $\max_{x \in [1..m]} \mathrm{dist}_\lambda(S_x, T)$ is minimal, where $\mathrm{dist}_\lambda(S_x, T) := \min_{i \in [1..n-\lambda+1]} \mathrm{dist}_{\mathrm{ham}}(S_x[i..i + \lambda - 1], T)$ is the number of mismatches we need to be able to detect T via approximate pattern matching in S_x with $\mathrm{dist}_\lambda(S_x, T)$ mismatches. An example is shown in Fig. 2.

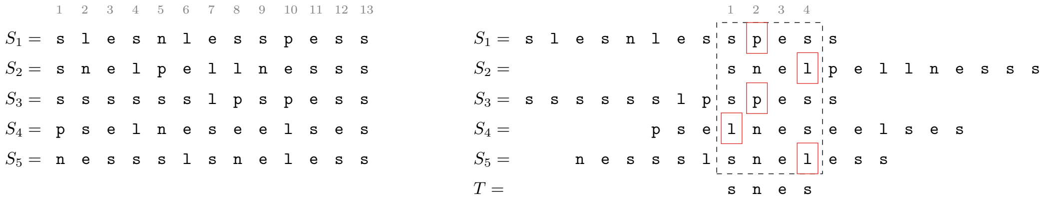

Figure 2 Example for CSS (Sect. 4) with $n = 13$ and query length $\lambda = 4$. The input is shown on the left figure. We can observe in the right figure that $T =$ **snes** is the CSS having one mismatch with each of the input strings in the Hamming distance by horizontally shifting the input strings.

Related Work. The decision problem for δ mismatches is also called δ-MISMATCH problem. Gramm et al. [32, Theorem 2] solved the decision problem in $\mathcal{O}(m\lambda + (n - \lambda)m\delta^{\delta+1})$ time. Marx [46] showed that CSS can be solved in $\mathcal{O}(\sigma^{\delta(\lg\delta+2}(nm)^{\mathcal{O}(\lg\delta)})$ or $\mathcal{O}((\sigma\delta)^{\mathcal{O}(m\delta)}(nm)^{\mathcal{O}(\log\log m)})$ time. A survey on further results can be found in [31]. With respect to other optimization approaches, we are aware of a genetic algorithm [47].

4.1 MAX-SAT encoding

Following [32, Section 3.3], we reduce CSS to CSP by selecting shifts $d_x \in [0..n - \lambda]$ of each input string S_x such that the CSP of $\{S_1[1 + d_1..\lambda + d_1], \ldots, S_m[1 + d_m..\lambda + d_m]\}$ is a solution of CSS if we take the minimum distance over all shifts d_x.

In what follows, we represent the shifts by a matrix of selectable Boolean variables of size $\mathcal{O}(m(n - \lambda))$. We redefine the alphabet for the i-th character to be $\Sigma_i := \{S_1[i + d_1], \ldots, S_m[i + d_m]\}$. We define the variables $T_{i,c}$ and $C_{i,x}$ as before. We copy (CSP1) as it is since it only states from which string S_x we select the i-th character of T, except that we have $\mathcal{O}(\lambda)$ instead of $\mathcal{O}(n)$ clauses since $|T| = \lambda$. The major difference is that for checking equality, we must add the offsets and obtain the following modification of (CSP2):

$$[\mathcal{O}(\lambda nm\sigma), \mathcal{O}(1)] \quad \forall i \in [1..\lambda], c \in \Sigma_i, x \in [1..m] : T_{i,c} \wedge S_x[i + d_x] \neq c \implies C_{i,x} \quad \text{(CSS2)}$$

The additional n-term in the complexity stems from the fact that the offsets d_x are represented as a two-dimensional binary array. The other equations as well as the objective are kept in the same way.

Complexities. We have $\mathcal{O}(\lambda\sigma + m(n - \lambda))$ selectable variables ($T_{i,c}$ and d_x), $\mathcal{O}(\lambda m)$ helper variables ($C_{i,x}$), $\mathcal{O}(\lambda mn\sigma)$ clauses. The largest clause has size $\mathcal{O}(\lambda m)$. Our implementation in ASP is given in Listing 2, where we expect an additional input of the form `#const lambda=`λ`.` for the requested substring length λ.

Listing 2 ASP for CSS (Sect. 4).

```
mat(X,I) :- s(X,I,_).
1 {d(X,D) : D = 0..n-lambda} 1 :- mat(X,0).
sigma(I,C) :- s(X,J,C), d(X,D), J-D >= 0, I = J-D.
1 {t(I,C) : sigma(I,C)} 1 :- mat(_,I), I < lambda. %(CSP1)
c(X,I) :- t(I,C), s(X,J,A), d(X,D), I+D == J, I < lambda, A != C. %(CSS2)
cost(X,C) :- C = #sum {1,I : c(X,I)}, mat(X,_). %(CSP3)
mcost(M) :- M = #max {C : cost(_,C)}.
#minimize {M : mcost(M)}.
#show t/2. #show mcost/1. #show cost/2.
```

Table 1 Evaluation for the Closest String Problem (csp) for $\lambda = 0$ and Closest Substring Problem (css) for $\lambda > 0$. The column *dist* shows the maximum Hamming distance of the reported string to all input strings. The column *rules* is the number of created SAT rules, *vars* is the number of variables, and *choices* is the number of choices or configurations the solver or brute-force algorithm tries. Reported times are in seconds ([s]).

			ASP				brute-force	
file	λ	dist	rules	vars	choices	time [s]	choices	time [s]
s05m09n009i0	0	6	1288	321	725	0.01	640 000	5.47
s05m09n009i0	7	4	1932	1122	1663	0.02	78 125	1.96
s05m09n009i0	8	5	1764	969	3666	0.05	390 625	7.29
s05m09n009i0	9	6	1427	330	676	0.01	1 953 125	21.59
s06m07n009i1	0	7	1078	268	1767	0.02	768 000	5.12
s06m07n009i1	7	4	1765	1069	3235	0.04	279 936	5.49
s06m07n009i1	8	5	1550	868	1314	0.02	1 679 616	24.45
s06m07n009i1	9	7	1194	275	2058	0.02	10 077 696	87.80
s06m08n009i0	0	6	1191	295	1074	0.01	750 000	5.67
s06m08n009i0	7	5	1907	1147	4266	0.05	279 936	6.23
s06m08n009i0	8	6	1698	954	4021	0.05	1 679 616	27.90
s06m08n009i0	9	6	1319	303	1273	0.01	10 077 696	100.23
s06m08n009i1	0	7	1248	299	2378	0.02	1 800 000	13.63
s06m08n009i1	7	5	1971	1203	4834	0.07	279 936	6.27
s06m08n009i1	8	6	1770	1012	5093	0.08	1 679 616	27.77
s06m08n009i1	9	7	1380	307	2163	0.02	10 077 696	99.98
s06m08n009i2	0	7	1248	299	2128	0.02	1 800 000	13.61
s06m08n009i2	7	5	1907	1147	5556	0.07	279 936	6.28
s06m08n009i2	8	6	1698	955	5552	0.08	1 679 616	27.91
s06m08n009i2	9	7	1380	307	2210	0.02	10 077 696	99.84
s06m09n009i0	0	7	1303	322	1837	0.02	800 000	6.81
s06m09n009i0	7	4	2142	1301	4331	0.05	279 936	7.02
s06m09n009i0	8	5	1920	1093	5334	0.08	1 679 616	31.38
s06m09n009i0	9	7	1443	331	1962	0.02	10 077 696	111.16
s06m09n009i1	0	7	1396	328	1849	0.02	2 700 000	22.97
s06m09n009i1	7	5	2177	1334	5341	0.07	279 936	7.04
s06m09n009i1	8	6	1920	1100	5693	0.10	1 679 616	31.20
s06m09n009i1	9	7	1542	337	1746	0.02	10 077 696	110.05
s06m09n009i2	0	6	1336	324	1874	0.02	1 080 000	9.07
s06m09n009i2	7	4	2177	1333	3706	0.05	279 936	6.92
s06m09n009i2	8	5	1946	1114	4565	0.06	1 679 616	30.52
s06m09n009i2	9	6	1478	333	1920	0.02	10 077 696	107.62

4.2 Evaluation of csp and css

Although there are efficient heuristics like choosing a majority string [8], we compared our ASP encoding for csp to a basic brute-force algorithm that enumerates all possible assignments for the characters of the closest substring. The number of possible configurations for T is $c_{\mathcal{S}} := \prod_{i=1}^{n} \sigma_i \in \mathcal{O}(\min(\sigma^n), m^n)$ dependent on the shape of the strings in $\mathcal{S}$. A brute-force algorithm trying each configuration spends $\mathcal{O}(c_{\mathcal{S}} nm)$ time on computing the Hamming distances of the resulting string T with all strings of $\mathcal{S}$.

This algorithm can be easily adopted for css. For that, we consider all possible offsets of the input strings like in the ASP encoding. Hence, the number of configurations is the number of configurations for the csp instance, multiplied by $(n - \lambda)^m$ for each possible offset value. If λ is small, then it suffices to compute all configurations of T, which are σ^λ many, and compute the Hamming distances in $\mathcal{O}(\lambda m)$ time for each such configuration. We implemented the former brute-force approach, whose time complexity grows exponentially with all parameters σ, n, and m, for randomly generated strings. We can observe this case in Table 1, where the ASP implementation outperforms the brute-force approach.

Table 2 Evaluation of the Closest String problem (SCP) on datasets provided by Torres and Hoshino [54]. The column *distance* is the maximal Hamming distance of the output to any of the input strings.

file	distance	rules	vars	choices	time [s]
rand-4-150-150-5-2	2	31 329	12 942	19	0.06
rand-4-50-50-5-2	2	10 529	4342	21	0.015
rand-4-100-100-5-2	2	20 929	8642	24	0.031
rand0-2-10-10-20-5	4	4286	842	43	0.011
rand0-2-10-10-20-4	4	4286	842	60	0.011
rand0-4-10-10-20-5	4	4887	1179	65	0.012
rand0-2-10-10-20-3	5	4474	848	72	0.012
rand-20-50-50-5-2	2	17 323	4549	78	0.018
rand-20-150-150-5-2	2	55 819	13 197	100	0.082
rand0-20-10-10-20-5	4	5573	1359	121	0.013
rand-20-100-100-5-2	2	37 213	8894	129	0.041
rand-4-150-150-5-1	5	31 329	12 942	189	0.117
rand-4-50-50-5-1	5	10 529	4342	199	0.021
rand0-2-10-10-20-2	6	4474	922	202	0.014
rand-4-100-100-5-1	5	20 929	8642	248	0.056
rand0-2-10-10-20-1	7	4474	922	265	0.015
rand-4-50-50-10-2	5	12 279	3082	494	0.035
rand-20-100-100-5-1	5	37 213	8894	501	0.068
rand-20-150-150-5-1	5	55 819	13 197	525	0.131
rand-20-50-50-5-1	5	18 595	4585	548	0.029
rand0-4-10-10-20-4	5	5008	1264	555	0.018
rand0-4-10-10-20-3	5	4869	1241	627	0.019
rand0-20-10-10-20-4	5	5800	1384	998	0.027
rand-20-50-50-10-2	5	26 397	3511	1057	0.053
rand0-20-10-10-20-3	6	6520	1477	2369	0.058
rand-20-50-50-15-2	7	40 426	5288	3512	0.235
rand-4-50-50-15-2	7	18 454	4622	4192	0.251
rand-4-50-50-10-1	8	12 279	3082	7320	0.343
rand0-4-10-10-20-2	8	5255	1334	18 095	0.373
rand-20-50-50-10-1	9	28 623	3574	23 622	1.255
rand0-20-10-10-20-2	9	6964	1540	48 538	1.265
rand-4-50-50-20-2	10	24 654	6162	98 610	12.379
rand-4-50-50-15-1	11	18 454	4622	119 367	8.76
rand-20-50-50-20-2	10	53 844	7047	168 793	28.348
rand0-4-10-10-20-1	11	5404	1360	770 565	19.168
rand-20-50-50-15-1	12	42 864	5357	2 716 507	358.345
rand-4-50-50-20-1	15	24 654	6162	39 265 111	7009.909

In Table 2, we depict the results of a larger evaluation on the datasets provided in [54][5], which are also used in [16, 43]. We kept their file naming, which is the format `rand`-σ-$\frac{m}{2}$-$\frac{m}{2}$-n-i, where i is an iteration counter to have multiple files with the same characteristics (m, n, and σ). The prefix `rand` can be followed by a zero. We observe that larger distances correlate with the number of choices, affecting the overall running time. Even for large inputs with short distances like the dataset `rand`-4-150-150-5-1, the running time is short.

[5] `https://github.com/jeanpttorres/dssp`

	1	2	3	4	5	6	7	8	9	10	11	12	13
$S_1 =$	n	s	l	l	e	e	p	e	p	l	e	s	s
$S_2 =$	s	l	s	e	l	e	e	p	p	l	e	s	s
$S_3 =$	l	s	l	e	p	e	p	n	l	e	s	n	s
$S_4 =$	e	s	l	e	p	e	p	l	n	e	s	l	s
$S_5 =$	n	s	l	e	e	p	p	l	l	e	e	s	s

	1	2	3	4	5	6	7	8	9	10	11	12	13
$S_1 =$	n	$\boxed{\texttt{s}}$	$\boxed{\texttt{l}}$	l	$\boxed{\texttt{e}}$	$\boxed{\texttt{e}}$	$\boxed{\texttt{p}}$	e	p	$\boxed{\texttt{l}}$	$\boxed{\texttt{e}}$	$\boxed{\texttt{s}}$	$\boxed{\texttt{s}}$
$S_2 =$	$\boxed{\texttt{s}}$	l	s	e	$\boxed{\texttt{l}}$	$\boxed{\texttt{e}}$	$\boxed{\texttt{e}}$	$\boxed{\texttt{p}}$	p	$\boxed{\texttt{l}}$	$\boxed{\texttt{e}}$	$\boxed{\texttt{s}}$	$\boxed{\texttt{s}}$
$S_3 =$	l	$\boxed{\texttt{s}}$	$\boxed{\texttt{l}}$	$\boxed{\texttt{e}}$	p	$\boxed{\texttt{e}}$	$\boxed{\texttt{p}}$	n	$\boxed{\texttt{l}}$	$\boxed{\texttt{e}}$	$\boxed{\texttt{s}}$	n	$\boxed{\texttt{s}}$
$S_4 =$	e	$\boxed{\texttt{s}}$	$\boxed{\texttt{l}}$	$\boxed{\texttt{e}}$	p	$\boxed{\texttt{e}}$	$\boxed{\texttt{p}}$	$\boxed{\texttt{l}}$	n	$\boxed{\texttt{e}}$	$\boxed{\texttt{s}}$	l	$\boxed{\texttt{s}}$
$S_5 =$	n	$\boxed{\texttt{s}}$	$\boxed{\texttt{l}}$	$\boxed{\texttt{e}}$	$\boxed{\texttt{e}}$	$\boxed{\texttt{p}}$	p	$\boxed{\texttt{l}}$	l	$\boxed{\texttt{e}}$	$\boxed{\texttt{e}}$	s	$\boxed{\texttt{s}}$
$T =$	s	l	e	e	p	l	e	s	s				

Figure 3 Example for LCS (Sect. 5) with $n = 13$. The input is shown on the left figure. In the right figure, we highlighted the subsequences matching $T =$ `sleepless` by surrounding the respective characters with boxes in each input string. Here, $T =$ `sleepless` is the LCS of all input strings.

5 Longest Common Subsequence (LCS)

The LCS problem asks for the longest string T such that T is a subsequence of S_x for every $x \in [1..m]$. See Fig. 3 for an example.

Existence. A solution exists if all strings share at least one common character in the alphabet.

Related Work. Maier [45] showed that LCS is NP-hard for $\sigma \geq 2$, and the same holds for SCS with $\sigma \geq 5$. Later, Blin et al. [5] gave a proof that LCS stays NP-hard even if the input strings are well-compressible with the run-length encoding. For exact algorithms, we can extend the classic dynamic programming (DP) algorithm of Wagner and Fischer [56] to m strings, which then takes $\mathcal{O}(n^m)$ time. Irving and Fraser [36] gave two algorithms running in $\mathcal{O}(mn(n-\ell)^{m-1})$ or $\mathcal{O}(m\ell(n-\ell)^{m-1} + m\sigma n)$ time, where ℓ is the length of the output. This result implies that LCS is FPT in m and $n - \ell$. Bulteau et al. [10] improved the result of [36] with an algorithm running in $\mathcal{O}((n - \ell + 1)^{n-\ell+1} mn)$ time, which is an FPT in the number of deletions $n - \ell$. Finally, there is a genetic algorithm [34] and an ant colony optimization algorithm [50].

5.1 MAX-SAT encoding

Our idea is to select a subsequence T_x for each input string S_x and maximize the length of T_x under the constraint that all T_x's have to be equal. The subsequence T_x of S_x is given by a sequence of indices $i_1 < \ldots < i_{|T_x|}$ such that $S_x[i_1] \cdots S_x[i_{|T_x|}] = T_x$. We can encode the subsequences T_x by the selectable variables $C_{x,\ell,i}$ encoding whether $T_x[\ell] = S_x[i]$, for each $x \in [1..m], \ell \in [1..n]$. We make use of $C_{x,\ell,i}$ as follows. First, for each $T_x[\ell]$, we define the range for the selectable variables $C_{x,\ell,i}$.[6]

$$[\mathcal{O}(nm), \mathcal{O}(n)] \quad \forall x \in [1..m], \ell \in [1..n] : \sum_{i \in [\ell..n]} C_{x,\ell,i} \geq 0 \qquad \text{(LCS1)}$$

[6] Logically, we would expect in (LCS1) a "≤ 1" instead of a "≥ 0". However, the former suffices together with the following constraints and is cheaper than "≤ 1".

If we have selected $T_x[\ell]$ to be $S_x[i]$, then $T_x[\ell-1]$ must be a character chosen in $S_x[1..i-1]$:

$$[\mathcal{O}(n^2 m), \mathcal{O}(n)] \quad \forall x \in [1..m], \ell \in [2..n], i \in [\ell..n] :$$
$$C_{x,\ell,i} \implies \sum_{j \in [1..i-1]} C_{x,\ell-1,j} = 1 \tag{LCS2}$$

Next, we define the helper variables $V_{x,\ell}$ encoding whether T_x has a length of at least ℓ, for each $x \in [1..m], \ell \in [1..n]$. If we have selected a character for $T_x[\ell]$ via $C_{x,\ell,i}$, then we set $V_{x,\ell}$ to true to specify that T_x has a length of at least ℓ.

$$[\mathcal{O}(nm), \mathcal{O}(n)] \quad \forall x \in [1..m], \ell \in [1..n] : \bigvee_{i \in [1..n]} C_{x,\ell,i} \implies V_{x,\ell} \tag{LCS3}$$

We now restrict all T_x's to be of equal length, which we do in a Round-Robin encoding:

$$[\mathcal{O}(nm), \mathcal{O}(1)] \quad \forall x \in [1..m], \ell \in [1..n] : V_{x,\ell} \implies V_{(x+1) \bmod n, \ell} \tag{LCS4}$$

Here, $\bmod n : \{1, 2, \ldots\} \to [1..n]$ is the modulo operation with $n \bmod n = n$ and $(n + 1) \bmod n = 1$. To achieve that all T_x store the same characters, we use the following constraint.

$$[\mathcal{O}(n^3 m), \mathcal{O}(1)] \quad \forall x \in [1..m], \ell \in [1..n], i, j \in [1..n] :$$
$$C_{x,\ell,i} \wedge C_{(x+1) \bmod m, \ell, j} \implies S_x[i] = S_{(x+1) \bmod m}[j] \tag{LCS5}$$

Finally, we enforce that we need to select a position for $T_x[\ell]$ if $V_{x,\ell}$ is set:

$$[\mathcal{O}(nm), \mathcal{O}(n)] \quad \forall x \in [1..m], \ell \in [1..n] : V_{x,\ell} \implies \bigvee_{i \in [\ell..n]} C_{x,\ell,i} \tag{LCS6}$$

Alternatively to (LCS5) and (LCS6), we can state that the next subsequence must select one of the text positions j for $T_{x+1}[\ell]$ with $S_{x+1}[j] = S_x[i]$.

$$[\mathcal{O}(n^2 m), \mathcal{O}(n)] \quad \forall x \in [1..m], \ell \in [1..n], i \in [1..n] :$$
$$C_{x,\ell,i} \implies \sum_{j : S_x[i] = S_{(x+1) \bmod n}[j]} C_{(x+1) \bmod m, \ell, j} = 1 \tag{LCS5'}$$

Finally, we formulate our optimization problem as

$$[\mathcal{O}(1), \mathcal{O}(n)] \quad \text{maximize} \sum_{\ell \in [1..n]} V_{1,\ell} \tag{LCS7}$$

Complexities. Our implementation in ASP is given in Listing 3. We have $\mathcal{O}(mn^2)$ selectable variables ($C_{x,\ell,i}$), $\mathcal{O}(mn)$ helper variables ($V_{x,\ell}$), and $\mathcal{O}(n^2 m)$ clauses (LCS5'). The largest clause has $\mathcal{O}(n)$ variables. An improvement for short LCS solutions could be to encode the existence problem for a fixed length λ in ASP such that we have $\mathcal{O}(m\lambda)$ selectable variables for encoding T_x, and call this encoding while varying λ to find the largest value for λ admitting a solution.

Table 3 Evaluation of the LONGEST COMMON SUBSEQUENCE problem (LCS).

		ASP				brute-force	
file	length	rules	vars	choices	time [s]	choices	time [s]
s02m11n023i1	10	166 538	21 494	23 617	1.00	8 388 608	47.29
s02m10n023i2	10	151 627	19 540	34 146	1.02	8 388 608	43.35
s02m09n023i1	11	137 112	17 586	10 964	0.61	8 388 608	39.73
s03m08n023i1	8	138 002	15 632	4831	0.40	8 388 608	39.20
s04m09n023i1	6	162 617	17 586	3927	0.39	8 388 608	39.07
s03m11n023i2	8	188 366	21 494	20 672	1.18	8 388 608	38.99
s03m08n023i2	7	136 795	15 632	11 046	0.63	8 388 608	38.54
s03m07n023i2	9	119 551	13 678	5945	0.40	8 388 608	37.59
s04m11n023i1	6	197 886	21 494	5767	0.58	8 388 608	37.06
s03m08n023i0	8	136 968	15 632	6301	0.45	8 388 608	37.05
s03m08n022i0	8	120 880	14 256	5467	0.37	4 194 304	17.87
s03m08n022i1	7	120 416	14 256	3970	0.32	4 194 304	17.69
s02m11n022i1	11	146 880	19 602	11 779	0.53	4 194 304	17.61
s03m07n022i2	9	105 785	12 474	2763	0.24	4 194 304	17.34
s03m11n022i2	7	165 908	19 602	7974	0.63	4 194 304	17.31
s04m11n022i1	6	175 570	19 602	8045	0.58	4 194 304	17.02
s02m09n022i1	12	121 186	16 038	6522	0.27	4 194 304	16.85
s03m08n022i2	8	120 313	14 256	4442	0.34	4 194 304	16.80
s04m09n022i1	6	143 324	16 038	5791	0.45	4 194 304	16.72
s02m10n022i2	10	135 128	17 820	9640	0.47	4 194 304	15.94

Listing 3 ASP for LCS (Sect. 5).

```
mat(X,I) :- s(X,I,_).
0 {c(X,L,I) : mat(X,I), I >= L} :- mat(X,L). %(LCS1)
1 {c(X,L,J) : J < I, mat(X,J)} 1 :- c(X,L+1,I), mat(X,L), mat(X,L+1). %(
    ↪ LCS2)
v(X,L) :- c(X,L,I), mat(X,I), mat(X,L). %(LCS3)
v(X+1,L) :- v(X,L), mat(X,L), mat(X+1,L). %(LCS4)
v(0,L) :- v(m-1,L).
:- c(X+1,L,J), c(X,L,I), s(X,I,D), not s(X+1,J,D). %(LCS5)
:- c(0,L,J), c(m-1,L,I), s(m-1,I,D), not s(0,J,D).
1 {c(X,L,I) : mat(X,I), I >= L} :- v(X,L). %(LCS6)
#maximize {1,L : v(0,L)}. %(LCS7)
#show c/3.
```

5.2 Evaluation

A DP approach would need $\mathcal{O}(n^m)$ time (cf. [15, Chapter IV, Section 15.4] for a textbook reference). Here, we stick to a trivial approach that tries all distinct subsequences of the first string S_1, and for each such subsequence we check whether it is a subsequence of all other input strings. The number of these subsequences is at most $2^n - 1$. If we select these subsequences with respect to their lengths, starting with the longest possible one, we can terminate whenever the selected subsequence is found in all other strings. In the worst

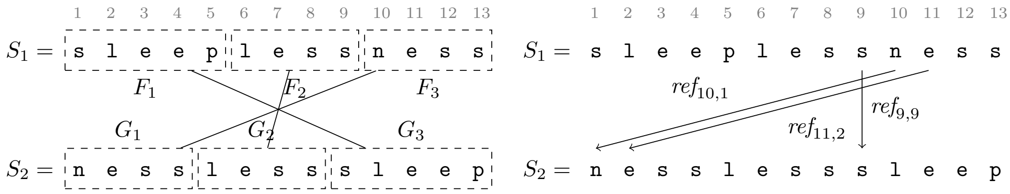

Figure 4 Example for MCSP (Sect. 6) with $n = 13$. We can factorize $S_1 = F_1F_2F_3$ into three factors, with $F_1 = G_3$, $F_2 = G_2$ and $F_3 = G_1$ such that $S_2 = G_1G_2G_3$. Hence, the solution for this example is a partition of length three. On the right is a partial assignment of the variable *ref* based on this partition, where *ref* induces a factor starting at position 10 in S_1.

case, the time complexity of this approach grows exponentially in n, but only linearly in m, independent of the alphabet size. We therefore restrict our evaluation in Table 3 to scaling n while keeping the other parameters unchanged. Like in Sect. 4.2, the ASP implementation outperforms the brute-force approach. However, a DP implementation might outperform the ASP implementation by re-using memoized results.

6 Minimum Common String Partition (MCSP)

For the special case of $m = 2$ input strings S_1 and S_2, the MCSP problem, introduced by Goldstein et al. [29] and Swenson et al. [52], asks, for a given $z \in [1..n]$, a factorization of S_1 into $S_x = F_1 \cdots F_z$ and a permutation π of $[1..z]$ such that $F_{\pi(1)} \cdots F_{\pi(z)} = S_2$. The optimization problem is to find the smallest z for which a solution exists. We give an example in Fig. 4.

Existence. A sufficient condition for whether a solution for any $z \in [1..n]$ exists is to check that the Parikh vectors of S_1 and S_2 are the same, such that at least a permutation on $[1..n]$ exist to rearrange the characters of S_1 to form S_2.

Related Work. While introducing MCSP, Goldstein et al. [29] also showed that it is NP-hard. Bulteau and Komusiewicz [11] showed that MCSP is FPT in z. For constant alphabets ($\sigma = \mathcal{O}(1)$), Cygan et al. [17] presented an exact algorithm running in $2^{\mathcal{O}(n \lg\lg n / \lg n)}$ time. Recently, Chromý and Sinnl [14] studied a DP algorithm. It is known that MCSP can be tackled by probabilistic tree searches [7], ILP formulations [6, 23], and an ant colony optimization algorithm [22].

6.1 MAX-SAT encoding

We adapt the MAX-SAT encoding of Bannai et al. [1] for the shortest bidirectional macro scheme problem [51]. To this end, we define the sets $\mathcal{M}_i := \{j \in [1..n] \mid S_1[i] = S_2[j]\} \subset [1..n]$ specifying the positions in S_2 that match with $S_1[i]$. In what follows, we make use of the following selectable Boolean variables:

- p_i or q_i encode if $S_1[i]$ or $S_2[i]$ is the start of a factor, respectively, for $i \in [1..n]$.
- $ref_{i \to j}$ encodes whether position i of S_1 references position j of S_2, and vice versa, for $i \in [1..n]$ and $j \in \mathcal{M}_i$.

We have $\mathcal{O}(n^2)$ Boolean variables, which we use as follows. On the one hand, each position in S_1 has exactly one reference:

$$[\mathcal{O}(n), \mathcal{O}(n)] \quad \forall i \in [1..n] : \sum_{j \in \mathcal{M}_i} \mathit{ref}_{i \to j} = 1 \tag{MCSP1}$$

On the other hand, each position in S_2 has exactly one reference:

$$[\mathcal{O}(n), \mathcal{O}(n)] \quad \forall j \in [1..n] : \sum_{i \in [1..n]} \mathit{ref}_{i \to j} = 1 \tag{MCSP2}$$

In what follows, we add implications for the factor starting positions that are due to how we set the references. First, a factor starts always at the first text position, so p_1 and q_1 are always true. If $S_1[i]$ references $S_2[i]$ and i is a factor starting position of S_1, so is j for S_2.

$$[\mathcal{O}(n^2), \mathcal{O}(1)] \quad \forall i \in [1..n], j \in \mathcal{M}_i : p_i \wedge \mathit{ref}_{i \to j} \implies q_j \tag{MCSP3}$$

Next, if $S_1[i]$ references $S_2[i]$ and j is a factor starting position of S_2, so is i for S_1. We only have to check that condition for q_1 since all other constraints set p_i and constraint (MCSP3) then implies that q_j has to be set.

$$[\mathcal{O}(n), \mathcal{O}(1)] \quad \forall i \in [1..n] : q_1 \wedge \mathit{ref}_{i \to 1} \implies p_i \tag{MCSP4}$$

Another condition is that if the previous text positions have mismatching characters, we cannot extend the factor to the left.

$$[\mathcal{O}(n^2), \mathcal{O}(1)] \quad \forall i \in [1..n], j \in \mathcal{M}_i \text{ with } S_1[i-1] \neq S_2[j-1] : \mathit{ref}_{i \to j} \implies p_i \tag{MCSP5}$$

Even if the previous characters match, when the reference of the previous text positions is different, we need to make a factor starting position:

$$[\mathcal{O}(n^2), \mathcal{O}(1)] \quad \forall i \in [2..n], \forall j \in \mathcal{M}_i \text{ such that } j > 1 \text{ and } S_2[i-1] = S_2[j-1] :$$
$$\neg \mathit{ref}_{i-1 \to j-1} \wedge \mathit{ref}_{i \to j} \implies p_i \tag{MCSP6}$$

$$[\mathcal{O}(1), \mathcal{O}(n)] \quad \text{Finally, we minimize } \sum_{i \in [1..n]} p_i \tag{MCSP7}$$

Complexities. We have $\mathcal{O}(n^2)$ selectable variables, and $\mathcal{O}(n^2)$ clauses (MCSP3). The largest clause has $\mathcal{O}(n)$ variables (MCSP2). Our implementation in ASP is given in Listing 4. Note that we start counting at zero, so `p(0).` is equivalent to setting p_1 to true. Instead of `mat` we use the helper variables `spos` and `tpos` denoting the existence of $S_1[i]$ and $S_2[i]$, respectively.

Listing 4 ASP for MCSP (Sect. 6).

```
spos(I) :- s(0,I,_).
tpos(J) :- s(1,J,_).
p(0). q(0).
arc(I,J) :- s(0,I,C), s(1,J,C).
1 {ref(I,J) : arc(I,J)} 1 :- spos(I). %(MCSP1)
1 {ref(I,J) : arc(I,J)} 1 :- tpos(J). %(MCSP2)
q(J) :- p(I), ref(I,J). %(MCSP3)
p(I) :- q(1), ref(I,1). %(MCSP4)
p(I) :- ref(I,J), s(0,I-1,C), s(1,J-1,D), C != D. %(MCSP5)
p(I) :- not ref(I-1,J-1), ref(I,J). %(MCSP6)
#minimize {1,X : p(X)}. %(MCSP7)
#show ref/2. #show p/1. #show q/1.
```

Table 4 Evaluation of the Minimum Common String Partition problem (MCSP). Note that the time for the ASP solution is in milliseconds. The column z denotes the number of factors of the returned partition.

		ASP				brute-force	
file	z	rules	vars	choices	time [ms]	choices	time [s]
2s03n009i2	4	443	124	25	1.0	986 409	6.24
2s02n009i0	4	586	165	61	2.0	986 409	6.31
2s02n009i1	4	586	165	59	2.0	986 409	6.43
2s03n009i0	6	426	124	52	1.0	986 409	6.44
2s03n009i1	2	367	116	30	1.0	986 409	6.49
2s02n009i2	6	521	149	39	1.0	986 409	6.95
2s03n010i1	4	604	162	67	2.0	9 864 100	68.81
2s02n010i0	4	510	213	108	2.0	9 864 100	70.92
2s03n010i0	6	484	147	37	1.0	9 864 100	71.04
2s03n010i2	4	584	164	47	2.0	9 864 100	71.38
2s02n010i2	4	637	189	77	2.0	9 864 100	73.78
2s02n010i1	3	639	187	103	2.0	9 864 100	74.28

Table 5 Evaluation of the Minimum Common String Partition problem (MCSP) on prefixes of the SARS-CoV-2 dataset.

length	z	rules	vars	choices	time [s]
10	4	447	146	34	0.001
20	12	1273	445	269	0.003
30	14	2282	911	1951	0.017
40	16	3720	1685	4683	0.047
50	21	5468	2442	2 050 092	18.609
60	24	7451	3422	6 866 999	80.256

6.2 Evaluation

Without leveraging the actual contents of the characters like in our SAT formulation, a naive way is to factorize both strings S_1 and S_2 with factors of the same lengths, and check whether there exists a permutation such that we can match factors of S_1 with factors of S_2. To this end, we iterate over the size z of the partition from 1 to n. For each $z \in [1..n]$, we partition S_1 into z factors $S_1 = F_1 \cdots F_z$. There are $\binom{n}{z}$ such ways to partition S_1. For each permutation π_z on $[1..z]$, we define the factorization $G_1 \cdots G_z = S_2$ with $|G_x| = |F_{\pi(x)}|$ for all $x \in [1..z]$. If $G_x = F_{\pi(x)}$ for all $x \in [1..z]$, then we have found a solution, and terminate. The number of configurations is $\sum_{z=1}^{n} \binom{n}{z} z!$, and each check takes $\mathcal{O}(n)$ time. Like the brute-force approach for LCS (Sect. 5.2), this approach has an exponential dependency on the text length n. In Table 4, we observe that specifying the choices for the references for each position individually (as we do in our ASP encoding) reduces the number of choices significantly when compared to the choices the brute-force algorithm processes.

Since our ASP encoding for MCSP seems quite efficient, we subsequently performed a benchmark on real data. In detail, we conducted an experiment by scaling the prefix length of a given input sequence, and report results in Table 5. For that, we used the SARS-CoV-2

	1	2	3	4	5
$S_1 =$	e	e	p	l	e
$S_2 =$	l	e	s	s	n
$S_3 =$	p	l	e	s	s
$S_4 =$	s	l	e	e	p
$S_5 =$	s	n	e	s	s

	1	2	3	4	5	6	7	8	9	10	11	12	13
$S_1 =$			e	e	p	l	e						
$S_2 =$						l	e	s	s	n			
$S_3 =$					p	l	e	s	s				
$S_4 =$	s	l	e	e	p								
$S_5 =$									s	n	e	s	s
$T =$	s	l	e	e	p	l	e	s	s	n	e	s	s

Figure 5 Example for SCS (Sect. 7) with $n = 5$. The input is shown on the left figure. By the right figure, the SCS is $T =$ `sleeplessness`, where we shifted the input strings to match their occurrences in T.

reference in FASTA format introduced in the analysis of Farkas et al. [21] [7], after removing the header line and the newline characters. For each extracted prefix of this FASTA file, we created an instance for MCSP, where the second string is a random permutation of the original prefix. We can observe in Table 5 that the output size z exponentially correlates with the number of choices and the running time.

7 Shortest Common Superstring (SCS)

The SCS problem asks for the shortest string T such that S_x is a substring of T, for all $x \in [1..m]$. Figure 5 shows an example.

Existence. A trivial common superstring is the concatenation $S_1 \cdots S_m$. Permuting the strings and removing overlapping parts lead to the solution [25].

Related Work. Gallant et al. [25] showed that SCS is NP-hard for $n \geq 3$ with respect to the number of strings m and unbounded alphabet size, but can be solved in linear time if $n \leq 2$. For binary alphabet $\sigma = 2$, they showed that the problem is still NP-hard for $n = \Omega(\log(nm))$. It is known that SCS can be solved with neural networks [44] and genetic algorithms [30]. Most research on SCS is devoted to the analysis and improvement of the approximation algorithm presented by Tarhio and Ukkonen [53, Theorem 2.3]. This algorithm builds the so-called *overlap graph* of $\mathcal{S}$. The authors observed that a Hamiltonian path on the overlap graph [49] maximizing the weights of the selected edges solves SCS.

7.1 Reduction to Hamiltonian Path

We follow the idea of Tarhio and Ukkonen [53] by reducing SCS to the search of the Hamiltonian path maximizing the weights of the selected edges. The ASP encoding of finding a Hamiltonian cycle in an unweighted graph has already been studied in [42, 41]. We build on one of their approaches and extend it by maximizing the weights while omitting the weight of one edge to turn the cycle into a Hamiltonian path[8]. An *overlap graph* $(\mathcal{S}, A, w)$ is a weighted

[7] `https://github.com/cfarkas/SARS-CoV-2-freebayes`

[8] We make a distinction between Hamiltonian path and Hamiltonian cycle in the sense that the cycle visits exactly one node twice.

directed graph, having the input strings $\mathcal{S}$ as nodes and the arcs $A := \{(S_x, S_y) : x \neq y\}$. The weights are defined by a weight function $w : A \to [0..n]$ with $w(S_x, S_y) := \max\{|U| : U$ is suffix of S_x and prefix of $S_y\}$. Hence, $w(S_x, S_y)$ is the number of overlapping characters, which we can omit if we want to build the superstring of S_x and S_y that starts with S_x. With respect to the overlap graph, a *path* is a sequence of strings, and a *Hamiltonian path* in the overlap graph is a path that visits each node exactly once, i.e., a permutation π of the list $[S_1, \ldots, S_m]$. Our goal is to find a permutation that maximizes $\sum_{x=1}^{m-1} w(S_{\pi(x)}, S_{\pi(x+1)})$, i.e., to find the Hamiltonian path whose arcs have maximal weights in sum.

7.2 MAX-SAT encoding

We define the following $\mathcal{O}(m^2)$ Boolean variables:

- $\mathit{cycle}_{x,y}$ encoding whether we have the arc (S_x, S_y) in our Hamiltonian cycle, for $x, y \in [1..m]$;
- $\mathit{reach}_{x,y}$ encoding whether we can reach S_y from S_x by following the transitive closure of *cycle*, for $x, y \in [1..m]$;
- start_x encoding whether our superstring starts with S_x, for $x \in [1..m]$.

First, we select arcs from the overlap graph for $\mathit{cycle}_{x,y}$. To this end, for each string S_x, we select exactly one out-going arc and one in-coming arc:

$$[\mathcal{O}(m), \mathcal{O}(m)] \quad \forall x \in [1..m] : \sum_{y=1}^{m} \mathit{cycle}_{x,y} = 1 \text{ and } \forall y \in [1..m] : \sum_{x=1}^{m} \mathit{cycle}_{x,y} = 1 \tag{SCS1}$$

The transitive closure of *cycle* can be encoded as follows. First we initialize *reach* by the direct connections due to *cycle*.

$$[\mathcal{O}(m^2), \mathcal{O}(1)] \quad \forall x, y \in [1..m], x \neq y : \mathit{cycle}_{x,y} \implies \mathit{reach}_{x,y} \tag{SCS2}$$

Next, if we can reach y from x, and there is an arc (y, z), then we can reach z from x:

$$[\mathcal{O}(m^3), \mathcal{O}(1)] \quad \forall x, y, z \in [1..m], x \neq y \neq z : \mathit{reach}_{x,y} \wedge \mathit{cycle}_{y,z} \implies \mathit{reach}_{x,z} \tag{SCS3}$$

To make the path selected by $\mathit{cycle}_{x,y}$ an Hamiltonian path, we want that all strings are connected via *reach*:

$$[\mathcal{O}(m^2), \mathcal{O}(1)] \quad \forall x, y \in [1..m], x \neq y : \mathit{reach}_{x,y} = 1 \tag{SCS4}$$

For the Hamiltonian path it is left to select a designated start string[9].

$$[\mathcal{O}(1), \mathcal{O}(m)] \quad \sum_{y=1}^{m} \mathit{start}_y = 1 \tag{SCS5}$$

Finally, our objective is to maximize the weights on the path starting from start_x of length m:

$$[\mathcal{O}(1), \mathcal{O}(m^2)] \quad \text{maximize} \sum_{x,y \in [1..m]:\ \mathit{cycle}_{x,y} \wedge \neg \mathit{start}_y} w(x, y) \tag{SCS6}$$

[9] It actually suffices to check in (SCS4) that all strings can be reached from this start string, but doing so had a negative impact on the overall running time in the experiments.

Table 6 Evaluation of the SHORTEST COMMON SUPERSTRING problem (SCS). $|T|$ is the length of the SCS.

		ASP				brute-force	
file	$\|T\|$	rules	vars	choices	time [s]	choices	time [s]
s02m10n008i0	42	2090	1416	198 756	3.58	10 240	0.02
s02m10n008i1	33	2206	1465	1 854 941	40.73	10 240	0.02
s02m10n008i2	39	2200	1464	1 401 686	29.49	10 240	0.02
s02m11n008i0	49	2639	1825	2 150 681	44.96	22 528	0.03
s02m11n008i1	35	2699	1861	6 652 411	154.48	22 528	0.03
s02m11n008i2	50	2611	1817	6 980 725	136.00	22 528	0.02

Complexities. We have $\mathcal{O}(m^2)$ selectable variables and $\mathcal{O}(m^3)$ clauses (SCS3). The largest clause has $\mathcal{O}(m^2)$ variables (SCS6). Our implementation in ASP is given in Listing 5. We expect an input of the form `w(X,Y,C)` encoding the weight $w(\mathtt{X},\mathtt{Y}) = \mathtt{C}$. The helper variables `node` and `gain` define the nodes of the overlap graph and the value of the optimization argument in (SCS6), respectively.

Listing 5 ASP for SCS (Sect. 7).

```
node(X) :- w(X,_,_).
1 {cycle(X,Y) : w(X,Y,_)} 1 :- node(X). %(SCS1)
1 {cycle(X,Y) : w(X,Y,_)} 1 :- node(Y).
reach(X,Y) :- cycle(X,Y). %(SCS2)
reach(X,Z) :- reach(X,Y), cycle(Y,Z). %(SCS3)
:- not reach(X,Y), node(X), node(Y). %(SCS4)
1 {start(X) : node(X)} 1. %(SCS5)
gain(D) :- D = #sum {C,X : cycle(X,Y), w(X,Y,C), not start(Y)}. %(SCS6)
#maximize {D : gain(D)}.
#show cycle/2. #show start/1.
```

7.3 Evaluation

The overlap graph can be computed in $\mathcal{O}(nm + m^2)$ time [33]. Given the overlap graph, the easiest approach is to enumerate all $m!$ permutations, and compute the sum of the selected weights in $\Theta(m)$ time. The time bound can be improved by using a DP approach taking $\mathcal{O}(m^2 2^m)$ time[10]. In the experiments of Table 6, we use this DP approach as our brute-force solution. We observe that it outperforms our ASP implementation on all instances. That is due to the fact that (a) our ASP encoding does not make use of more information than the DP approach, and that (b) the number of choices in our encoding for the Hamiltonian path is prohibitively large. As a matter of fact, efficient SAT and ASP encodings for Hamiltonian cycles are actively studied, cf. [58] for SAT and [4] for ASP.

[10] https://leetcode.com/problems/find-the-shortest-superstring/solutions/194891/official-solution/

Table 7 Encoding complexities of the studied problems. Columns *prob.*, *#sel. vars*, *#h.vars*, *#clauses* and *max. cl.* denote the problem name, the number of defined selectable variables, the number of helper variables, the number of clauses, and the maximum size a clause can have.

prob.	#sel.vars	#h.vars	#clauses	max cl.
CSP	$\mathcal{O}(n\sigma)$	$\mathcal{O}(nm)$	$\mathcal{O}(nm\sigma)$	$\mathcal{O}(mn)$
CSS	$\mathcal{O}(\lambda\sigma+(n-\lambda)m)$	$\mathcal{O}(\lambda m)$	$\mathcal{O}(nm\sigma\lambda)$	$\mathcal{O}(\lambda m)$
LCS	$\mathcal{O}(n^2 m)$	$\mathcal{O}(mn)$	$\mathcal{O}(n^2 m)$	$\mathcal{O}(n)$
MCSP	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
SCS	$\mathcal{O}(m^2)$	$\mathcal{O}(1)$	$\mathcal{O}(m^3)$	$\mathcal{O}(m^2)$

8 Conclusion

We provided encodings in ASP for five prominent examples of NP-hard problems in the field of stringology. We summarized the complexities of the encodings in Table 7. We observed that, on the one hand, by leveraging characteristics of the input data such as for MCSP, our solution is far superior than simple brute-force approaches that omit those characteristics. On the other hand, for SCS, we observed that if the problem can be easily reduced to instances of problems like finding a Hamiltonian path, DP approaches are already efficient enough to find the answer faster than an ASP solver. It therefore depends on the nature of the problem we study for whether an application of an ASP solver makes sense. Nevertheless, the programming in ASP is highly expressive as can be seen by the short program codes in Listings 1–5, and therefore can be understood as a tool for rapid prototyping. Other advantages of ASP solvers like clingo are that they can work in parallel, report approximate solutions when reaching a given timeout, and enumerate all solutions, provided that the specified constraints do not exclude one of them. An evaluation of those features is left as future work since it would go beyond the scope of this paper.

References

1. Hideo Bannai, Keisuke Goto, Masakazu Ishihata, Shunsuke Kanda, Dominik Köppl, and Takaaki Nishimoto. Computing NP-hard repetitiveness measures via MAX-SAT. In *Proc. ESA*, volume 244 of *LIPIcs*, pages 12:1–12:16, 2022. `doi:10.4230/LIPIcs.ESA.2022.12`.
2. Manu Basavaraju, Fahad Panolan, Ashutosh Rai, M. S. Ramanujan, and Saket Saurabh. On the kernelization complexity of string problems. *Theor. Comput. Sci.*, 730:21–31, 2018. `doi:10.1016/j.tcs.2018.03.024`.
3. Riccardo Bertolucci, Alessio Capitanelli, Carmine Dodaro, Nicola Leone, Marco Maratea, Fulvio Mastrogiovanni, and Mauro Vallati. An ASP-based framework for the manipulation of articulated objects using dual-arm robots. In *Proc. LPNMR*, volume 11481 of *LNCS*, pages 32–44, 2019. `doi:10.1007/978-3-030-20528-7_3`.
4. Manuel Bichler, Bernhard Bliem, Marius Moldovan, Michael Morak, and Stefan Woltran. Treewidth-preserving modeling in ASP. Technical Report DBAI-TR-2016-97, Technische Universität Wien, 2016.
5. Guillaume Blin, Laurent Bulteau, Minghui Jiang, Pedro J. Tejada, and Stéphane Vialette. Hardness of longest common subsequence for sequences with bounded run-lengths. In *Proc. CPM*, volume 7354 of *LNCS*, pages 138–148, 2012. `doi:10.1007/978-3-642-31265-6_11`.
6. Christian Blum. ILP-based reduced variable neighborhood search for large-scale minimum common string partition. *Electron. Notes Discret. Math.*, 66:15–22, 2018. `doi:10.1016/j.endm.2018.03.003`.

7 Christian Blum, José Antonio Lozano, and Pedro Pinacho Davidson. Iterative probabilistic tree search for the minimum common string partition problem. In *Proc. HM*, volume 8457 of *LNCS*, pages 145–154, 2014. doi:10.1007/978-3-319-07644-7_11.

8 Christina Boucher and Kathleen P. Wilkie. Why large closest string instances are easy to solve in practice. In *Proc. SPIRE*, volume 6393 of *LNCS*, pages 106–117, 2010. doi:10.1007/978-3-642-16321-0_10.

9 Laurent Bulteau, Falk Hüffner, Christian Komusiewicz, and Rolf Niedermeier. Multivariate algorithmics for np-hard string problems. Technical report, European Association for Theoretical Computer Science, 2014.

10 Laurent Bulteau, Mark Jones, Rolf Niedermeier, and Till Tantau. An FPT-algorithm for longest common subsequence parameterized by the maximum number of deletions. In *Proc. CPM*, volume 223 of *LIPIcs*, pages 6:1–6:11, 2022. doi:10.4230/LIPIcs.CPM.2022.6.

11 Laurent Bulteau and Christian Komusiewicz. Minimum common string partition parameterized by partition size is fixed-parameter tractable. In *Proc. SODA*, pages 102–121, 2014. doi:10.1137/1.9781611973402.8.

12 Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. ASP-core-2 input language format. *Theory Pract. Log. Program.*, 20(2):294–309, 2020. doi:10.1017/S1471068419000450.

13 Markus Chimani, Matthias Woste, and Sebastian Böcker. A closer look at the closest string and closest substring problem. In *Proc. ALENEX*, pages 13–24, 2011. doi:10.1137/1.9781611972917.2.

14 Milos Chromý and Markus Sinnl. On solving the minimum common string partition problem by decision diagrams. In *Proc. ICORES*, pages 177–184, 2022. doi:10.5220/0010830200003117.

15 Thomas H Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction To Algorithms*. MIT Press, 2009.

16 Federico Della Croce and Fabio Salassa. Improved lp-based algorithms for the closest string problem. *Comput. Oper. Res.*, 39(3):746–749, 2012. doi:10.1016/j.cor.2011.06.010.

17 Marek Cygan, Alexander S. Kulikov, Ivan Mihajlin, Maksim Nikolaev, and Grigory Reznikov. Minimum common string partition: Exact algorithms. In *Proc. ESA*, volume 204 of *LIPIcs*, pages 35:1–35:16, 2021. doi:10.4230/LIPIcs.ESA.2021.35.

18 Warley Gramacho da Silva, Tiago da Silva Almeida, Rafael Lima de Carvallho, Edeilson Milhomem da Silva, Ary Henrique de Oliveira, Glenda Michele Botelho, and Glêndara Aparecida de Souza Martins. Two classic chess problems solved by answer set programming. *International Journal of Advanced Engineering Research and Science*, 6(4):1–5, 2019. doi:10.22161/ijaers.6.4.43.

19 Esra Erdem, Michael Gelfond, and Nicola Leone. Applications of answer set programming. *AI Magazine*, 37(3):53–68, 2016. doi:10.1609/aimag.v37i3.2678.

20 Andreas A. Falkner, Gerhard Friedrich, Konstantin Schekotihin, Richard Taupe, and Erich Christian Teppan. Industrial applications of answer set programming. *Künstliche Intell.*, 32(2-3):165–176, 2018. doi:10.1007/s13218-018-0548-6.

21 Carlos Farkas, Andy Mella, Maxime Turgeon, and Jody J Haigh. A novel sars-cov-2 viral sequence bioinformatic pipeline has found genetic evidence that the viral 3' untranslated region (utr) is evolving and generating increased viral diversity. *Frontiers in microbiology*, 12(665041):1–14, 2021. doi:10.3389/fmicb.2021.665041.

22 S. M. Ferdous and M. Sohel Rahman. Solving the minimum common string partition problem with the help of ants. *Math. Comput. Sci.*, 11(2):233–249, 2017. doi:10.1007/s11786-017-0293-5.

23 S. M. Ferdous and Mohammad Sohel Rahman. An integer programming formulation of the minimum common string partition problem. *PLOS ONE*, 10(7):1–16, 2015. doi:10.1371/journal.pone.0130266.

24 Moti Frances and Ami Litman. On covering problems of codes. *Theory Comput. Syst.*, 30(2):113–119, 1997. doi:10.1007/s002240000044.

25 John Gallant, David Maier, and James A. Storer. On finding minimal length superstrings. *J. Comput. Syst. Sci.*, 20(1):50–58, 1980. doi:10.1016/0022-0000(80)90004-5.

26 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In *Proc. ICLP*, volume 52 of *OASIcs*, pages 2:1–2:15, 2016. doi:10.4230/OASIcs.ICLP.2016.2.

27 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.*, 19(1):27–82, 2019. doi:10.1017/S1471068418000054.

28 Martin Gebser, Marco Maratea, and Francesco Ricca. The seventh answer set programming competition: Design and results. *Theory Pract. Log. Program.*, 20(2):176–204, 2020. doi:10.1017/S1471068419000061.

29 Avraham Goldstein, Petr Kolman, and Jie Zheng. Minimum common string partition problem: Hardness and approximations. *Electron. J. Comb.*, 12, 2005. doi:10.37236/1947.

30 Luis C. González, Heidi J. Romero, and Carlos A. Brizuela. A genetic algorithm for the shortest common superstring problem. In *Proc. GECCO*, volume 3103 of *LNCS*, pages 1305–1306, 2004. doi:10.1007/978-3-540-24855-2_139.

31 Jens Gramm. Closest substring. In *Encyclopedia of Algorithms*, pages 324–326. Springer, 2016. doi:10.1007/978-1-4939-2864-4_74.

32 Jens Gramm, Rolf Niedermeier, and Peter Rossmanith. Fixed-parameter algorithms for CLOSEST STRING and related problems. *Algorithmica*, 37(1):25–42, 2003. doi:10.1007/s00453-003-1028-3.

33 Dan Gusfield, Gad M. Landau, and Baruch Schieber. An efficient algorithm for the all pairs suffix-prefix problem. *Inf. Process. Lett.*, 41(4):181–185, 1992. doi:10.1016/0020-0190(92)90176-V.

34 Brenda Hinkemeyer and Bryant A. Julstrom. A genetic algorithm for the longest common subsequence problem. In *Proc. GECCO*, pages 609–610, 2006. doi:10.1145/1143997.1144105.

35 Hoang Xuan Huan, Dong Do Duc, and Nguyen Manh Ha. An efficient two-phase ant colony optimization algorithm for the closest string problem. In *Proc. SEAL*, volume 7673 of *LNCS*, pages 188–197, 2012. doi:10.1007/978-3-642-34859-4_19.

36 Robert W. Irving and Campbell Fraser. Two algorithms for the longest common subsequence of three (or more) strings. In *Proc. CPM*, volume 644 of *LNCS*, pages 214–229, 1992. doi:10.1007/3-540-56024-6_18.

37 Tom Kelsey and Lars Kotthoff. Exact closest string as a constraint satisfaction problem. In *ProcİCCS*, volume 4 of *Procedia Computer Science*, pages 1062–1071, 2011. doi:10.1016/j.procs.2011.04.113.

38 Dusan Knop, Martin Koutecký, and Matthias Mnich. Combinatorial n-fold integer programming and applications. *Math. Program.*, 184(1):1–34, 2020. doi:10.1007/s10107-019-01402-2.

39 J. Kevin Lanctôt, Ming Li, Bin Ma, Shaojiu Wang, and Louxin Zhang. Distinguishing string selection problems. *Inf. Comput.*, 185(1):41–55, 2003. doi:10.1016/S0890-5401(03)00057-9.

40 Vladimir Lifschitz. *Answer Set Programming*. Springer, 2019. doi:10.1007/978-3-030-24658-7.

41 Liu Liu. *The Performance Optimization of ASP Solving Based on Encoding*. PhD thesis, University of Kentucky, 2022.

42 Liu Liu and Miroslaw Truszczynski. Encoding selection for solving hamiltonian cycle problems with ASP. In *Proc. ICLP*, volume 306 of *EPTCS*, pages 302–308, 2019. doi:10.4204/EPTCS.306.35.

43 Xiaolan Liu, Shenghan Liu, Zhifeng Hao, and Holger Mauch. Exact algorithm and heuristic for the closest string problem. *Comput. Oper. Res.*, 38(11):1513–1520, 2011. doi:10.1016/j.cor.2011.01.009.

44 Domingo López-Rodríguez and Enrique Mérida Casermeiro. Shortest common superstring problem with discrete neural networks. In *Proc. ICANNGA*, volume 5495 of *LNCS*, pages 62–71, 2009. doi:10.1007/978-3-642-04921-7_7.

45 David Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, 1978. doi:10.1145/322063.322075.

46 Dániel Marx. The closest substring problem with small distances. In *Proc. FOCS*, pages 63–72, 2005. doi:10.1109/SFCS.2005.70.

47 Holger Mauch. Closest substring problem – results from an evolutionary algorithm. In *Proc. ICONIP*, volume 3316 of *LNCS*, pages 205–211, 2004. doi:10.1007/978-3-540-30499-9_30.

48 Rolf Niedermeier. Ubiquitous parameterization – invitation to fixed-parameter algorithms. In *Proc. MFCS*, volume 3153 of *LNCS*, pages 84–103, 2004. doi:10.1007/978-3-540-28629-5_4.

49 Hannu Peltola, Hans Söderlund, Jorma Tarhio, and Esko Ukkonen. Algorithms for some string matching problems arising in molecular genetics. In *Proc. IFIP*, pages 59–64, 1983.

50 Shyong Jian Shyu and Chun-Yuan Tsai. Finding the longest common subsequence for multiple biological sequences by ant colony optimization. *Comput. Oper. Res.*, 36(1):73–91, 2009. doi:10.1016/j.cor.2007.07.006.

51 James A. Storer and Thomas G. Szymanski. Data compression via textural substitution. *J. ACM*, 29(4):928–951, 1982. doi:10.1145/322344.322346.

52 Krister M. Swenson, Mark Marron, Joel V. Earnest-DeYoung, and Bernard M. E. Moret. Approximating the true evolutionary distance between two genomes. *ACM J. Exp. Algorithmics*, 12:3.5:1–3.5:17, 2008. doi:10.1145/1227161.1402297.

53 Jorma Tarhio and Esko Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. *Theor. Comput. Sci.*, 57:131–145, 1988. doi:10.1016/0304-3975(88)90167-3.

54 Jean P. Tremeschin Torres and Edna Ayako Hoshino. Lp-based heuristics for the distinguishing string and substring selection problems. *Ann. Oper. Res.*, 316(2):1205–1234, 2022. doi:10.1007/s10479-021-04138-5.

55 Omar Vilca and Rosiane de Freitas. An efficient algorithm for the closest string problem. In *Anais do I Encontro de Teoria da Computação*, pages 879–882, Porto Alegre, RS, Brasil, 2016. doi:10.5753/etc.2016.9850.

56 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974. doi:10.1145/321796.321811.

57 Lusheng Wang, Ming Li, and Bin Ma. Closest string and substring problems. In *Encyclopedia of Algorithms*, pages 321–324. Springer, 2016. doi:10.1007/978-1-4939-2864-4_73.

58 Neng-Fa Zhou. In pursuit of an efficient SAT encoding for the Hamiltonian cycle problem. In *Proc. CP*, volume 12333 of *LNCS*, pages 585–602, 2020. doi:10.1007/978-3-030-58475-7_34.

A Alternative CSS Encoding

For small values of λ, the offsets can be quite large. Here, we present an alternative encoding without the offsets. The resulting encoding has fewer variables, but has more variables that are subject to the optimization argument. In what follows, we can encode $T[1..\lambda]$ by the Boolean variables $T_{i,c}$ specifying with $T_{i,c} = 1$ that $T[i] = c$:

$$[\mathcal{O}(\lambda), \mathcal{O}(\sigma)] \quad \forall i \in [1..\lambda] : \sum_{c \in \Sigma} T_{i,c} = 1 \tag{CSS1'}$$

We now let the costs encode the offsets by the variables $C_{i,x,o}$ being set if $S_x[o+i] \neq T[i]$.

$$[\mathcal{O}(\lambda n m \sigma), \mathcal{O}(1)] \quad \forall i \in [1..\lambda], c \in \Sigma_i, x \in [1..m], o \in [1..n-\lambda] :$$
$$T_{i,c} \wedge S_x[i+o] \neq c \implies C_{i,x,o} \tag{CSS2'}$$

Table 8 Used Entities.

entity	meaning
Σ	alphabet
σ	alphabet size, $\sigma = \lvert\Sigma\rvert$
$\mathcal{S}$	set of input strings $\{S_1, \ldots, S_m\}$
m	size of $\mathcal{S}$, i.e., $m = \lvert\mathcal{S}\rvert$
n	length of an input string
S_x	input string
T	string to output
ℓ	length for a subsequence
δ	distance of the output to all S_x
i, j	indices for text positions in an input string
x, y	indices for an input string
c	character in Σ

The objective function becomes

$$[\mathcal{O}(1), \mathcal{O}(mn^2)] \quad \text{minimize} \max_{x \in [1..m]} \min_{o \in [1..n-\lambda]} \sum_{i \in [1..n]} C_{i,x,o} \tag{CSS3'}$$