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:fccb46f95c8f782e61c7e63a9a154a6e2f8dad9

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), 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.
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 \(S = \{S_1, \ldots, S_m\}\). For simplicity, we assume that all strings have the same length \(n\), and that all characters are drawn from an alphabet \(\Sigma\) of size \(\sigma = |\Sigma|\). Hence, \(|S_x[n]| = n\) 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

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

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

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 \((\sigma, m \text{ and } n)\). 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
3 Closest String Problem (CSP)

The Closest String Problem (CSP)\footnote{Alternative names are, among others, Minimum Radius, Center String or Consensus String problem.} asks for a string $T$ such that $\max_{x \in [1..m]} \text{dist}_{\text{ham}}(S_x, T)$ is minimal, where the Hamming distance $\text{dist}_{\text{ham}}$ is given by $\text{dist}_{\text{ham}}(S_x, T) := |\{i \in [1..n] : S_x[i] \neq T[i]\}|$. An example is shown in Fig. 1. Here, and in the following examples we stick to the alphabet $\Sigma := \{e, l, p, n, s\}$ 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 (csp), 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 csp. For the decision problem with a Hamming distance of $d$, Gramm et al. [32] showed that csp can be solved in $O(mn + d^4)$ time or $2^{O((m \log m))} O(\log n)$ time. Regarding integer linear programming (ILP), Chimani et al. [13] gave ILP formulations, also for csp. 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 $m^O(m^2) O(\log n)$ 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 $\sigma_i$ can be much less than $m$ or $\sigma$ if the number of distinct characters is small. We can express the alphabets per position $\Sigma_i$ by a Boolean matrix $M[1..n][1..\sigma]$ with $M[i][c] = 1$ if $c \in \Sigma_i$.
Further, we define the variables \( T_{i,c} = 1 \) to encode that \( T[i] = c \), for \( i \in [1..n], c \in \{S_1[i],...,S_m[i]\} \). 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:

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

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 \( \text{dist}_{\text{ham}}(T,S_x) = \sum_{i \in [1..n]} C_{i,x} \). Therefore:

\[
[O(nm\sigma),O(1)] \forall i \in [1..n], c \in \Sigma, x \in [1..m] : T_{i,c} \land S_x[i] = c \implies C_{i,x} \tag{CSP2}
\]

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:

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

**Complexities.** We have \( O(n \sigma) \) selectable variables (\( T_{i,c} \)), \( O(nm) \) helper variables (\( C_{i,x} \)), \( O(nm \sigma) \) clauses (CSP2). The largest clause contains \( O(mn) \) 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 \( S_x[I] = C \in \Sigma \). We use the helper variables \( \text{mat}(X,I) \) to denote the existence of \( S_x[I] \). For encoding (CSP3) in ASP, we additionally define the helper variables \( \text{cost} \) and \( \text{mcost} \) encoding \( \sum_{i \in [1..n]} C_{i,X} \) and \( \max_{x \in [1..m]} \text{cost}(x) \), 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).

\[
\begin{align*}
\text{mat}(X,I) &\right s(X,I,\_). \\
1 \{ t(I,C) : s(\_,I,C) \} &\right \text{mat}(\_,I). \%(CSP1) \\
\text{c}(X,I) &\right t(I,C), s(X,I,A), C \neq A. \%(CSP2) \\
\text{cost}(X,C) &\right C = \#\text{sum} \{1,I : \text{c}(X,I)\}, \text{mat}(X,\_). \%(CSP3) \\
\text{mcost}(M) &\right M = \#\text{max} \{C : \text{cost}(\_,C)\}. \\
\text{\#minimize} \{M : \text{mcost}(M)\}. \\
\text{\#show t/2. \#show mcost/1. \#show cost/2.}
\end{align*}
\]

**4 Closest Substring (CSS)**

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

The decision problem for $\delta$ mismatches is also called $\delta$-MISMATCH problem. Gramm et al. [32, Theorem 2] solved the decision problem in $O(m\lambda + (n - \lambda)m\delta^{\delta+1})$ time. Marx [46] showed that css can be solved in $O(\sigma^{\delta}(lg^{\delta} + 2(nm))O(\log \delta))$ or $O((\sigma\delta)^{O(m\delta)(nm)}O(\log \log m))$ 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 $\{S_1[1 + d_1..\lambda + d_1],...,S_m[1 + d_m..\lambda + d_m]\}$ 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 $O(m(n - \lambda))$. We redefine the alphabet for the $i$-th character to be $S_i := \{S_1[i + d_1],...,S_m[i + d_m]\}$. 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 $O(\lambda)$ instead of $O(n)$ clauses since $|T| = \lambda$. The major difference is that for checking equality, we must add the offsets and obtain the following modification of (CSP2):

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

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

### 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).  
theta(I,C) :- s(X,J,C), d(X,D), J-D >= 0, I = J-D.  
1 {t(I,C) : theta(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 {I, 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.
17:6 Encoding Hard String Problems with Answer Set Programming

Although there are efficient heuristics like choosing a majority string [8], we compared in Table 1, where the ASP implementation outperforms the brute-force approach.

The Hamming distances of the resulting string brute-force algorithm trying each configuration spends $O(n^2)$ for $n$.

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 $\lambda$ is small, then it suffices to compute all configurations of $T$, which are $\sigma^\lambda$ many, and compute the Hamming distances in $O(\lambda m)$ time for each such configuration. We implemented the former brute-force approach, whose time complexity grows exponentially with all parameters $\sigma$, $n$, and $m$, for randomly generated strings. We can observe this case in Table 1, where the ASP implementation outperforms the brute-force approach.

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 $c_S := \prod_{i=1}^{\lambda} \sigma_i \in O(\min(\sigma^n), m^n)$ dependent on the shape of the strings in $S$. A brute-force algorithm trying each configuration spends $O(c_S n m)$ time on computing the Hamming distances of the resulting string $T$ with all strings of $S$.

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 $\lambda$ is small, then it suffices to compute all configurations of $T$, which are $\sigma^\lambda$ many, and compute the Hamming distances in $O(\lambda m)$ time for each such configuration. We implemented the former brute-force approach, whose time complexity grows exponentially with all parameters $\sigma$, $n$, and $m$, for randomly generated strings. We can observe this case in Table 1, where the ASP implementation outperforms the brute-force approach.
In Table 2, we depict the results of a larger evaluation on the datasets provided in [54], which are also used in [16, 43]. We kept their file naming, which is in the format `rand-σ-\(\frac{m}{2}\)-\(\frac{n}{2}\)-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
### 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 \( O(n^m) \) time. Irving and Fraser [36] gave two algorithms running in \( O(mn(n - \ell))^{m-1} \) or \( O(m\ell(n - \ell)^{m-1} + m\sigma n) \) time, where \( \ell \) 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 \( O((n - \ell + 1)^{n-\ell+1}mn) \) 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 \( C_{x,\ell,i} \).

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

\[6 \] Logically, we would expect in (LCS1) a “\( \leq 1 \)” instead of a “\( \geq 0 \)”. However, the former suffices together with the following constraints and is cheaper than “\( \leq 1 \)”.

---

\[ \text{Figure 3} \] 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 \( T = \text{sleepless} \) by surrounding the respective characters with boxes in each input string. Here, \( T = \text{sleepless} \) is the LCS of all input strings.
If we have selected $T_x[\ell]$ to be $S_x[i]$, then $T_x[\ell - 1]$ must be a character chosen in $S_x[1..i-1]$:

\[ \forall x \in [1..m], \ell \in [2..n], i \in [\ell..n] : \]

\[ C_{x,\ell,i} \implies \sum_{j \in [1..i-1]} C_{x,\ell-1,j} = 1 \]  \hspace{1cm} (LCS2)

Next, we define the helper variables $V_{x,\ell}$ encoding whether $T_x$ has a length of at least $\ell$, 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 $\ell$.

\[ \forall x \in [1..m], \ell \in [1..n] : \bigvee_{i \in [1..n]} C_{x,\ell,i} \implies V_{x,\ell} \]  \hspace{1cm} (LCS3)

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

\[ \forall x \in [1..m], \ell \in [1..n] : V_{x,\ell} \implies V_{x,(\ell+1) \mod n,\ell} \]  \hspace{1cm} (LCS4)

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

\[ \forall x \in [1..m], \ell \in [1..n], i,j \in [1..n] : \]

\[ C_{x,\ell,i} \land C_{(x+1) \mod m,\ell,j} \implies S_x[i] = S_{(x+1) \mod m}[j] \]  \hspace{1cm} (LCS5)

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

\[ \forall x \in [1..m], \ell \in [1..n] : V_{x,\ell} \implies \bigvee_{i \in [\ell..n]} C_{x,\ell,i} \]  \hspace{1cm} (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 $S_{x+1}[j] = S_x[i]$.

\[ \forall x \in [1..m], \ell \in [1..n], i \in [1..n] : \]

\[ C_{x,\ell,i} \implies \sum_{j:S_x[i]=S_{(x+1) \mod n}[j]} C_{(x+1) \mod m,\ell,j} = 1 \]  \hspace{1cm} (LCS5’)

Finally, we formulate our optimization problem as

\[ \text{maximize } \sum_{\ell \in [1..n]} V_{1,\ell} \]  \hspace{1cm} (LCS7)

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

<table>
<thead>
<tr>
<th>file</th>
<th>length</th>
<th>rules</th>
<th>vars</th>
<th>choices</th>
<th>time [s]</th>
<th>choices</th>
<th>time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>s02m11n023i1</td>
<td></td>
<td>10</td>
<td>166</td>
<td>21</td>
<td>494</td>
<td>23</td>
<td>617</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.00</td>
<td>8</td>
<td>388</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>608</td>
<td>47</td>
</tr>
<tr>
<td>s02m10n023i2</td>
<td></td>
<td>10</td>
<td>151</td>
<td>19</td>
<td>540</td>
<td>34</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.02</td>
<td>8</td>
<td>388</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>608</td>
<td>43</td>
</tr>
<tr>
<td>s03m09n023i1</td>
<td></td>
<td>11</td>
<td>137</td>
<td>17</td>
<td>586</td>
<td>10</td>
<td>964</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.61</td>
<td>8</td>
<td>388</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>608</td>
<td>39</td>
</tr>
<tr>
<td>s03m08n023i1</td>
<td></td>
<td>8</td>
<td>138</td>
<td>15</td>
<td>632</td>
<td>48</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.40</td>
<td>8</td>
<td>388</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>608</td>
<td>39</td>
</tr>
<tr>
<td>s04m09n023i1</td>
<td></td>
<td>6</td>
<td>162</td>
<td>17</td>
<td>586</td>
<td>39</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.39</td>
<td>8</td>
<td>388</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>608</td>
<td>39</td>
</tr>
<tr>
<td>s03m11n023i2</td>
<td></td>
<td>8</td>
<td>188</td>
<td>21</td>
<td>494</td>
<td>20</td>
<td>672</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.18</td>
<td>8</td>
<td>388</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>608</td>
<td>39</td>
</tr>
<tr>
<td>s03m08n023i2</td>
<td></td>
<td>7</td>
<td>136</td>
<td>15</td>
<td>632</td>
<td>11</td>
<td>104</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.63</td>
<td>8</td>
<td>388</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>608</td>
<td>38</td>
</tr>
<tr>
<td>s03m07n023i2</td>
<td></td>
<td>9</td>
<td>119</td>
<td>13</td>
<td>678</td>
<td>59</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.40</td>
<td>8</td>
<td>388</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>608</td>
<td>37</td>
</tr>
<tr>
<td>s04m11n023i2</td>
<td></td>
<td>6</td>
<td>197</td>
<td>21</td>
<td>494</td>
<td>57</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.58</td>
<td>8</td>
<td>388</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>608</td>
<td>37</td>
</tr>
<tr>
<td>s03m08n023i0</td>
<td></td>
<td>8</td>
<td>136</td>
<td>15</td>
<td>632</td>
<td>6</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.45</td>
<td>8</td>
<td>388</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>608</td>
<td>37</td>
</tr>
<tr>
<td>s03m08n022i0</td>
<td></td>
<td>8</td>
<td>120</td>
<td>14</td>
<td>526</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.37</td>
<td>4</td>
<td>194</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>304</td>
<td>17</td>
</tr>
<tr>
<td>s03m08n022i1</td>
<td></td>
<td>7</td>
<td>120</td>
<td>14</td>
<td>526</td>
<td>39</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.32</td>
<td>4</td>
<td>194</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>304</td>
<td>17</td>
</tr>
<tr>
<td>s02m11n022i1</td>
<td></td>
<td>11</td>
<td>146</td>
<td>19</td>
<td>602</td>
<td>11</td>
<td>779</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.53</td>
<td>4</td>
<td>194</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>304</td>
<td>17</td>
</tr>
<tr>
<td>s03m07n022i2</td>
<td></td>
<td>9</td>
<td>105</td>
<td>12</td>
<td>474</td>
<td>27</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.24</td>
<td>4</td>
<td>194</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>304</td>
<td>17</td>
</tr>
<tr>
<td>s03m11n022i2</td>
<td></td>
<td>7</td>
<td>165</td>
<td>19</td>
<td>602</td>
<td>79</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.63</td>
<td>4</td>
<td>194</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>304</td>
<td>17</td>
</tr>
<tr>
<td>s04m11n022i1</td>
<td></td>
<td>6</td>
<td>175</td>
<td>19</td>
<td>602</td>
<td>80</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.58</td>
<td>4</td>
<td>194</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>304</td>
<td>17</td>
</tr>
<tr>
<td>s02m09n022i1</td>
<td></td>
<td>12</td>
<td>121</td>
<td>16</td>
<td>638</td>
<td>62</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.27</td>
<td>4</td>
<td>194</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>304</td>
<td>16</td>
</tr>
<tr>
<td>s03m08n022i2</td>
<td></td>
<td>8</td>
<td>120</td>
<td>14</td>
<td>526</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.34</td>
<td>4</td>
<td>194</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>304</td>
<td>16</td>
</tr>
<tr>
<td>s04m09n022i1</td>
<td></td>
<td>6</td>
<td>143</td>
<td>16</td>
<td>638</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.45</td>
<td>4</td>
<td>194</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>304</td>
<td>16</td>
</tr>
<tr>
<td>s02m10n022i2</td>
<td></td>
<td>10</td>
<td>135</td>
<td>17</td>
<td>820</td>
<td>95</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.47</td>
<td>4</td>
<td>194</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>304</td>
<td>16</td>
</tr>
</tbody>
</table>

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 $O(n^m)$ 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
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 $i \in [1..n]$.
- $ref_{i \rightarrow j}$ encodes whether position $i$ of $S_1$ references position $j$ of $S_2$, and vice versa, for $i \in [1..n]$ and $j \in \mathcal{M}_i$.

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 $S_2 = F_1 \cdots F_z$ and a permutation $\pi$ 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 $2^{\mathcal{O}(n^{1 + \lg n}/\lg n)}$ time. Recently, Chromý and Simnl [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

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

$$[O(n), O(n)] \quad \forall i \in [1..n] : \sum_{j \in M_i} ref_{i \rightarrow j} = 1 \quad \text{(MCSP1)}$$

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

$$[O(n), O(n)] \quad \forall j \in [1..n] : \sum_{i \in [1..n]} ref_{i \rightarrow j} = 1 \quad \text{(MCSP2)}$$

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 $S_2$.

$$[O(n^2), O(1)] \quad \forall i \in [1..n], j \in M_i : p_i \land ref_{i \rightarrow j} \implies q_j \quad \text{(MCSP3)}$$

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.

$$[O(n), O(1)] \quad \forall i \in [1..n] : q_1 \land ref_{i \rightarrow 1} \implies p_i \quad \text{(MCSP4)}$$

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

$$[O(n^2), O(1)] \quad \forall i \in [1..n], j \in M_i \text{ with } S_1[i-1] \neq S_2[j-1] : ref_{i \rightarrow j} \implies p_i \quad \text{(MCSP5)}$$

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

$$[O(n^2), O(1)] \quad \forall i \in [2..n], j \in M_i \text{ such that } j > 1 \text{ and } S_2[i-1] = S_2[j-1] : \neg ref_{i-1 \rightarrow j-1} \land ref_{i \rightarrow j} \implies p_i \quad \text{(MCSP6)}$$

Finally, we minimize

$$\sum_{i \in [1..n]} p_i \quad \text{(MCSP7)}$$

**Complexities.** We have $O(n^2)$ selectable variables, and $O(n^2)$ clauses (MCSP3). The largest clause has $O(n)$ 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).

```prolog
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(I), ref(I,J). % (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.

<table>
<thead>
<tr>
<th>file</th>
<th>$z$</th>
<th>rules</th>
<th>vars</th>
<th>choices</th>
<th>time [ms]</th>
<th>choices</th>
<th>time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>2s03n009i2</td>
<td>4</td>
<td>443</td>
<td>124</td>
<td>25</td>
<td>1.0</td>
<td>986409</td>
<td>6.24</td>
</tr>
<tr>
<td>2s02n009i6</td>
<td>4</td>
<td>586</td>
<td>165</td>
<td>61</td>
<td>2.0</td>
<td>986409</td>
<td>6.31</td>
</tr>
<tr>
<td>2s02n009i1</td>
<td>4</td>
<td>586</td>
<td>165</td>
<td>59</td>
<td>2.0</td>
<td>986409</td>
<td>6.43</td>
</tr>
<tr>
<td>2s03n009i9</td>
<td>6</td>
<td>426</td>
<td>124</td>
<td>52</td>
<td>1.0</td>
<td>986409</td>
<td>6.44</td>
</tr>
<tr>
<td>2s03n009i1</td>
<td>2</td>
<td>367</td>
<td>116</td>
<td>30</td>
<td>1.0</td>
<td>986409</td>
<td>6.49</td>
</tr>
<tr>
<td>2s02n009i2</td>
<td>6</td>
<td>521</td>
<td>149</td>
<td>61</td>
<td>2.0</td>
<td>986409</td>
<td>6.95</td>
</tr>
<tr>
<td>2s03n010i1</td>
<td>4</td>
<td>604</td>
<td>162</td>
<td>67</td>
<td>2.0</td>
<td>9864100</td>
<td>68.81</td>
</tr>
<tr>
<td>2s02n010i0</td>
<td>4</td>
<td>510</td>
<td>213</td>
<td>108</td>
<td>2.0</td>
<td>9864100</td>
<td>70.92</td>
</tr>
<tr>
<td>2s03n010i2</td>
<td>6</td>
<td>484</td>
<td>147</td>
<td>37</td>
<td>1.0</td>
<td>9864100</td>
<td>71.04</td>
</tr>
<tr>
<td>2s03n010i2</td>
<td>4</td>
<td>584</td>
<td>164</td>
<td>47</td>
<td>2.0</td>
<td>9864100</td>
<td>71.38</td>
</tr>
<tr>
<td>2s02n010i2</td>
<td>4</td>
<td>637</td>
<td>189</td>
<td>77</td>
<td>2.0</td>
<td>9864100</td>
<td>73.78</td>
</tr>
<tr>
<td>2s02n010i1</td>
<td>3</td>
<td>639</td>
<td>187</td>
<td>103</td>
<td>2.0</td>
<td>9864100</td>
<td>74.28</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>length</th>
<th>$z$</th>
<th>rules</th>
<th>vars</th>
<th>choices</th>
<th>time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4</td>
<td>447</td>
<td>146</td>
<td>34</td>
<td>0.001</td>
</tr>
<tr>
<td>20</td>
<td>12</td>
<td>1273</td>
<td>445</td>
<td>269</td>
<td>0.003</td>
</tr>
<tr>
<td>30</td>
<td>14</td>
<td>2282</td>
<td>911</td>
<td>1951</td>
<td>0.017</td>
</tr>
<tr>
<td>40</td>
<td>16</td>
<td>3720</td>
<td>1685</td>
<td>4683</td>
<td>0.047</td>
</tr>
<tr>
<td>50</td>
<td>21</td>
<td>5468</td>
<td>2442</td>
<td>2050092</td>
<td>18.609</td>
</tr>
<tr>
<td>60</td>
<td>24</td>
<td>7451</td>
<td>3422</td>
<td>8669999</td>
<td>80.256</td>
</tr>
</tbody>
</table>

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 $\pi_x$ 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 $O(n)$ 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
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 $n = \Omega(\log(\text{num}))$. 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 $S$. 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. An overlap graph $(S, A, w)$ 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 $S$ 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 $w(S_x, S_y) := \max\{|U| : U \text{ is suffix of } S_x \text{ and prefix of } S_y\}$. 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 $\pi$ 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 $O(m^2)$ Boolean variables:

- $cycle_{x,y}$ encoding whether we have the arc $(S_x, S_y)$ in our Hamiltonian cycle, for $x, y \in [1..m]$;
- $reach_{x,y}$ 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 $cycle_{x,y}$. To this end, for each string $S_x$, we select exactly one out-going arc and one in-coming arc:

$$\forall x \in [1..m] : \sum_{y=1}^{m} cycle_{x,y} = 1 \quad \text{and} \quad \forall y \in [1..m] : \sum_{x=1}^{m} cycle_{x,y} = 1 \quad (SCS1)$$

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

$$\forall x, y \in [1..m], x \neq y : cycle_{x,y} \Rightarrow reach_{x,y} \quad (SCS2)$$

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

$$\forall x, y, z \in [1..m], x \neq y \neq z : reach_{x,y} \wedge cycle_{y,z} \Rightarrow reach_{x,z} \quad (SCS3)$$

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

$$\forall x, y \in [1..m], x \neq y : reach_{x,y} = 1 \quad (SCS4)$$

For the Hamiltonian path it is left to select a designated start string$^9$.

$$\sum_{y=1}^{m} start_y = 1 \quad (SCS5)$$

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

$$\maximize_{x,y \in [1..m]} \sum_{cycle_{x,y} \wedge \neg start_y} w(x, y) \quad (SCS6)$$

$^9$ 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.

| file          | $|T|$ | rules | vars | choices | time [s] | choices | time [s] |
|---------------|------|-------|------|---------|---------|---------|---------|
| s02m10n008i0  | 42   | 2090  | 1416 | 198756  | 3.58    | 10240   | 0.02    |
| s02m10n008i1  | 33   | 2206  | 1465 | 1854941 | 40.73   | 10240   | 0.02    |
| s02m10n008i2  | 39   | 2200  | 1464 | 1401686 | 29.49   | 10240   | 0.02    |
| s02m11n008i0  | 49   | 2639  | 1825 | 2150681 | 44.96   | 22528   | 0.03    |
| s02m11n008i1  | 35   | 2699  | 1861 | 6652411 | 154.48  | 22528   | 0.03    |
| s02m11n008i2  | 50   | 2611  | 1817 | 6980725 | 136.00  | 22528   | 0.02    |

Complexities. We have $O(m^2)$ selectable variables and $O(m^3)$ clauses (SCS3). The largest clause has $O(m^2)$ 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 $w(X,Y) = C$. 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).

```asp
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 $O(nm + m^2)$ 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 $O(m^{22^m})$ 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.

<table>
<thead>
<tr>
<th>prob.</th>
<th>#sel.vars</th>
<th>#h.vars</th>
<th>#clauses</th>
<th>max cl.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSP</td>
<td>$O(n\sigma)$</td>
<td>$O(nm)$</td>
<td>$O(nm\sigma)$</td>
<td>$O(nm)$</td>
</tr>
<tr>
<td>CSS</td>
<td>$O(\lambda\sigma + (n-\lambda)m)$</td>
<td>$O(\lambda m)$</td>
<td>$O(nm\sigma\lambda)$</td>
<td>$O(\lambda m)$</td>
</tr>
<tr>
<td>LCS</td>
<td>$O(n^2 m)$</td>
<td>$O(mn)$</td>
<td>$O(n^2 m)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>MCSP</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>SCS</td>
<td>$O(m^2)$</td>
<td>$O(1)$</td>
<td>$O(m^3)$</td>
<td>$O(m^2)$</td>
</tr>
</tbody>
</table>

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


For small values of $\lambda$, 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 $T[i] = c$:

$$\forall i \in [1..\lambda] : \sum_{c \in \Sigma} T_{i,c} = 1$$  \hspace{1cm} (CSS1')

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

$$\forall i \in [1..\lambda], c \in \Sigma, x \in [1..m], o \in [1..n-\lambda] :$$

$$T_{i,c} \land S_x[i+o] \neq c \implies C_{i,x,o}$$  \hspace{1cm} (CSS2')

\section*{A Alternative CSS Encoding}

For small values of $\lambda$, 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 $T[i] = c$:

$$\forall i \in [1..\lambda] : \sum_{c \in \Sigma} T_{i,c} = 1$$  \hspace{1cm} (CSS1')

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

$$\forall i \in [1..\lambda], c \in \Sigma, x \in [1..m], o \in [1..n-\lambda] :$$

$$T_{i,c} \land S_x[i+o] \neq c \implies C_{i,x,o}$$  \hspace{1cm} (CSS2')
Table 8 Used Entities.

<table>
<thead>
<tr>
<th>entity</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma$</td>
<td>alphabet</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>alphabet size, $\sigma =</td>
</tr>
<tr>
<td>$S$</td>
<td>set of input strings ${S_1, \ldots, S_m}$</td>
</tr>
<tr>
<td>$m$</td>
<td>size of $S$, i.e., $m =</td>
</tr>
<tr>
<td>$n$</td>
<td>length of an input string</td>
</tr>
<tr>
<td>$S_x$</td>
<td>input string</td>
</tr>
<tr>
<td>$T$</td>
<td>string to output</td>
</tr>
<tr>
<td>$\ell$</td>
<td>length for a subsequence</td>
</tr>
<tr>
<td>$\delta$</td>
<td>distance of the output to all $S_x$</td>
</tr>
<tr>
<td>$i, j$</td>
<td>indices for text positions in an input string</td>
</tr>
<tr>
<td>$x, y$</td>
<td>indices for an input string</td>
</tr>
<tr>
<td>$c$</td>
<td>character in $\Sigma$</td>
</tr>
</tbody>
</table>

The objective function becomes

\[
\begin{align*}
[O(1), O(mn^2)] & \quad \text{minimize} \quad \max_{x \in [1..m]} \min_{o \in [1..n-\lambda]} \sum_{i \in [1..n]} C_{i,x,o} \\
& \quad \text{(CSS3')}
\end{align*}
\]