Minimum Common String Partition: Exact Algorithms

Marek Cygan
University of Warsaw, Poland

Alexander S. Kulikov
Steklov Mathematical Institute at St. Petersburg, Russian Academy of Sciences, Russia
St. Petersburg State University, Russia

Ivan Mihajlin
Steklov Mathematical Institute at St. Petersburg, Russian Academy of Sciences, Russia

Maksim Nikolaev
Steklov Mathematical Institute at St. Petersburg, Russian Academy of Sciences, Russia

Grigory Reznikov
National Research University Higher School of Economics, St. Petersburg, Russia

Abstract

In the minimum common string partition problem (MCSP), one gets two strings and is asked to find the minimum number of cuts in the first string such that the second string can be obtained by rearranging the resulting pieces. It is a difficult algorithmic problem having applications in computational biology, text processing, and data compression. MCSP has been studied extensively from various algorithmic angles: there are many papers studying approximation, heuristic, and parameterized algorithms. At the same time, almost nothing is known about its exact complexity. In this paper, we present new results in this direction. We improve the known 2^n upper bound (where n is the length of input strings) to ϕ^n where $\phi \approx 1.618...$ is the golden ratio. The algorithm uses Fibonacci numbers to encode subsets as monomials of a certain implicit polynomial and extracts one of its coefficients using the fast Fourier transform. Then, we show that the case of constant size alphabet can be solved in subexponential time $2^{O(n \log\log n/ \log n)}$ by a hybrid strategy: enumerate all long pieces and use dynamic programming over histograms of short pieces. Finally, we prove almost matching lower bounds assuming the Exponential Time Hypothesis.

2012 ACM Subject Classification Theory of computation → Parameterized complexity and exact algorithms; Theory of computation → Algorithm design techniques

Keywords and phrases similarity measure, string distance, exact algorithms, upper bounds, lower bounds

Digital Object Identifier 10.4230/LIPIcs.ESA.2021.35

Funding *Marek Cygan*: The work of Marek Cygan is part of a project TOTAL that has received funding from the European Research Council (ERC) under the European Union Horizon 2020 research and innovation programme (grant agreement No 677651).
Alexander S. Kulikov: The research presented in Section 6 is supported by Russian Science Foundation (18-71-10042).
Maksim Nikolaev: The research presented in Section 6 is supported by Russian Science Foundation (18-71-10042). The research presented in Section 3 is supported by Ministry of Science and Education grant 075-15-2019-1620.

1 Overview

Two strings $s_1, s_2 \in \Sigma^n$ are said to have a common partition of size t if one can cut s_1 into t blocks (by $t-1$ cuts), rearrange them, and get s_2. This is a string similarity measure having applications in computational biology, text processing, and data compression. The minimum common string partition problem (MCSP) is to find the minimum size of a common partition of two input strings. There are two natural special cases of the problem: in d-MCSP, every symbol appears at most d times in each input string; in MCSP^c, the size of the alphabet Σ is at most c.

1.1 Known results

The problem has been studied extensively from various algorithmic angles.

Computational complexity. Already 2-MCSP is APX-hard, hence MCSP is both NP-hard and APX-hard [8].

Paramaterized algorithms. The problem is fixed parameter tractable (FPT) with respect to combinations of various parameters [6, 11, 2, 3]. For example, [2] gives an $O^*(d^{2k})$ algorithm[1] for d-MCSP, whereas [3] gives an FPT algorithm with respect to t only.

Approximation algorithms. The best known approximation ratio for a polynomial-time algorithm is $O(\log n \log^* n)$ [5]. There is also an $O(d)$-approximation algorithm [15]. The best known approximation ratios for 2-MCSP and 3-MCSP are 1.1037 and 4, respectively [8].

Heuristic algorithms. One natural heuristic approach for MCSP is greedy: cut out a longest common substring and repeat. It is well studied both from practical and theoretical points of view. Its approximation factor is: $O(\log n)$ for many families of inputs [17], between $\Omega(n^{0.46})$ and $O(n^{0.69})$ in the worst case [4, 13]. It can be implemented in linear time [9].

1.2 New results

Much less is known about *the exact* complexity of MCSP. The best known upper bound is $O^*(2^n)$ [7]. This aligns well with what is known for many other permutation problems: say, the shortest common superstring problem, the shortest common supersequence problem, the traveling salesman problem. Whereas many approximation, parameterized, and heuristic algorithms are known for these problems, breaking the 2^n barrier for any of them is a long standing open problem. (For the mentioned problems, n denotes the number of input strings, sequences, or nodes.)

In this paper, we present the following new exact algorithms and lower bounds for MCSP. We start by showing two straightforward $O^*(2^n)$ algorithms: the first one enumerates all possible partitions of both input strings and is perhaps the simplest exact algorithm for the problem; the second one is a straightforward application of the dynamic programming method. Both algorithms use exponential space.

[1] The $O^*(\cdot)$ notation is common in the field of algorithms for NP-hard problems: like $O(\cdot)$ hides constant factors, it suppresses factors that grow polynomially in the input length.

1.2.1 Improving time to ϕ^n: dynamic programming

Then, we show how to improve the dynamic programming algorithm to get the running time $O^*(\phi^n)$ where $\phi \approx 1.618...$ is the golden ratio. In short, the improvement is based on the following idea. For two strings to have a common partition, their multisets of symbols should coincide. Then, if one cuts the first string into blocks and maps all blocks of length at least two to the second string (so that the mapped parts do not overlap), all the remaining blocks of length one are mapped automatically.

1.2.2 Improving space to polynomial: FFT and Fibonacci encoding

The improved algorithm still uses exponential space (as it is based on dynamic programming). Again, avoiding exponential space in algorithms for NP-hard problems may be a challenging task. For example, the best known algorithm for the coloring problem has running time $O^*(2^n)$ and uses exponential space [1]; bringing down the space consumption to polynomial is an open problem. Also, for the optimal 2-constraint satisfaction problem (as well as for its special cases: maximum 2-satisfiability and maximum cut), all known better-than-2^n-time algorithms use exponential space [18, 14]; improving 2^n time while keeping polynomial space is an open problem. For the MCSP problem, we show how to get a polynomial space algorithm with running time $O^*(\phi^n)$.

The approach that we use is inspired by generating functions. Let $c_0, c_1, c_2, \dots = \{c_i\}_{i=0}^{\infty}$ be an integer sequence of interest, where term c_i is expressed in a complex way through the previous terms, so that it is difficult to directly find a closed form for it. Occasionally, one can solve this problem in three steps.

1. Pack a sequence into a polynomial (or, rather, formal power series):
$$T(x) = c_0 + c_1x + c_2x^2 + \dots = \sum_{i=0}^{\infty} c_i x^i \,.$$
2. By means of algebraic manipulations, find a closed form expression for $T(x)$.
3. Then, c_i is the coefficient of x_i in Taylor expansion of $T(x)$ that can be found using the value of the i^{th} derivative of T at zero.

What we use can be viewed as a discrete and algorithmic version of this approach. Given strings $s_1, s_2 \in \Sigma^n$, we define a finite sequence $\{c_i\}_{i=0}^{N}$ with the following property: s_1, s_2 have a common partition of size t iff a particular term c is positive. This term may be expressed through the other terms, but it will take a lot of space, so instead we find it in three steps.

1. Pack a sequence into a polynomial P of finite degree such that the term c is a coefficient of a particular monomial m in P.
2. Describe an efficient way of evaluating P at a given point.
3. Using fast Fourier transform, extract the coefficient of m using values of P at specific points.

The algorithm uses Fibonacci encoding in order to keep the degree of the polynomial low enough.

1.2.3 Subexponential running time for small alphabets: hybrid strategy

Then, we show that for the constant size alphabets, one can even get a subexponential running time: $2^{O(\frac{n \log\log n}{\log n})}$. The main idea is to track short and long blocks differently: we just guess the cut points of all long blocks (this is feasible as there are not too many of them) and we exploit dynamic programming to handle the short ones. By balancing these two cases carefully, one gets a speed up to a subexponential time.

1.2.4 Lower bounds

Finally, we show that substantial improvement of the presented upper bounds is difficult. Namely, we prove that under the Exponential Time Hypothesis (stating that there are no subexponential time algorithms for the satisfiability problem), MCSP cannot be solved in subexponential time (of the form $2^{o(n)}$). This is a simple application of the NP-hardness proof of MCSP. Then, we present a reduction from the general case to the special case of constant size alphabet that gives a lower bound $2^{\Omega(n/\log n)}$ (under ETH) for that special case.

Notation

Throughout the paper, we use the following notation. Let s be a string of length n. We use 1-based indexing and slice notation: $s = s[1]s[2]\cdots s[n]$; for $1 \le l \le r \le n$, $s[l:r] = s[l]s[l+1]\cdots s[r]$. By $[n]$ we denote the set $\{1,\dots,n\}$.

2 2^n time and space: enumerating all partitions

In this section, we present perhaps the most straightforward algorithm. As simple as it is, it has an important feature that many other algorithms lack: it works for any number of input strings. See Algorithm 1. Its running time is $O^*(2^n)$ as the number of different partitions of a string of length n is 2^{n-1}: there are $n-1$ places to make a cut.

Algorithm 1 Enumerating all partitions.

```
    Input: strings $s_1,\dots,s_j$ of length $n$.
    Output: the minimum size of a common partition of $s_1,\dots,s_j$.
 1: $T \leftarrow$ associative array
 2: for every partition of $s_1$ into substrings $p_1,\dots,p_k$ do
 3:     $T[\text{multiset}\{p_1,\dots,p_k\}] \leftarrow j$-tuple $(1,0,\dots,0)$
 4: for $i$ from 2 to $j$ do
 5:     for every partition of $s_i$ into substrings $p_1,\dots,p_k$ do
 6:         if $T$ has key multiset$\{p_1,\dots,p_k\}$ then
 7:             $T[\text{multiset}\{p_1,\dots,p_k\}][i] \leftarrow 1$
 8: $t \leftarrow +\infty$
 9: for every key $P$ of $T$ do:
10:     if $T[P] = (1,\dots,1)$ then
11:         $t \leftarrow \min(t, |P|)$
12: return $t$
```

Hereinafter, we return the size of the optimal partition instead of the partition itself. This is equivalent to the original problem, since there is a polynomial Algorithm 2 for finding the optimal partition given an oracle for minimum size of a common partition. In this algorithm, we first look for the longest prefix p of s_1 that can be matched somewhere in s_2, such that after replacing it and its match with a new symbol $\#_1$ the size of the optimal partition stay the same. To make sure that the size of the optimal partition does not change, we use the oracle. It is clear that in the optimal partition $\#_1$ is not a prefix of some longer block. After we find p, we replace it and its match with $\#_1$ and start looking for the next longest block in the optimal partition that we replace with a new symbol $\#_2$ and so on. In the end we obtain a partition $\mathcal{P} = \{p_1, p_2, \dots, p_t\}$ such that $s_1 = p_1 \dots p_t$ and p_i is the longest possible block in a common partition of the minimum size t given previous $i-1$ blocks.

Algorithm 2 Constructing the optimal partition.

```
    Input: strings $s_1, s_2$ of length $n$, oracle $\mathcal{O}$ for the minimum size of a common partition.
    Output: a common partition of $s_1, s_2$ of the minimum size.
 1: $\mathcal{P} \leftarrow \emptyset$
 2: $t \leftarrow \mathcal{O}(s_1, s_2)$
 3: for $i$ from $1$ to $t$ do
 4:     $\#_i \leftarrow$ a symbol that does not occur in $s_1$ and $s_2$
 5:     for all $j$ from $n$ down to $i$, all $1 \le l \le r \le n$ do
 6:         $s_1' \leftarrow s_1[1 : i-1] \#_i s_1[j+1 : n]$
 7:         $s_2' \leftarrow s_2[1 : l-1] \#_i s_2[r+1 : n]$
 8:         if $s_1[i : j] = s_2[l : r]$ and $\mathcal{O}(s_1', s_2') = t$ then
 9:             $\mathcal{P} \leftarrow \mathcal{P} \cup s_1[i : j]$
10:             $s_1 \leftarrow s_1'$
11:             $s_2 \leftarrow s_2'$
12:             break
13: return $\mathcal{P}$
```

3 Improving time to ϕ^n: dynamic programming

Here, we present a dynamic programming solution with roughly the same running time and space as the previous solution. Later, we will be able to improve both time and space of this algorithm. The algorithm works by solving the following subproblem: for $0 \le k \le n$, let $C(k)$ be the set of all pairs (S, t), where $S \subseteq [n]$, $|S| = k$, and $1 \le t \le n$, such that one can cut $s_1[1 : k]$ into t blocks and match them to the subsequence of s_2 specified by S. The minimum size of a common partition is then simply the minimum t such that (S, t) is contained in $C(n)$. To compute $C(k)$, we first fix the length i of the last block of $s_1[1 : k]$. Then, this block should be matched somewhere in s_2, whereas $C(k - i)$ can be used to find the positions where the remaining $t - 1$ blocks should be matched. The formal pseudocode is given in Algorithm 3. The running time and space is $O^*(2^n)$ as the number of different subsets is 2^n whereas all other steps clearly take polynomial time.

Algorithm 3 Dynamic programming.

```
    Input: strings $s_1, s_2$ of length $n$.
    Output: the minimum size of a common partition of $s_1, s_2$.
1: $C(0) \leftarrow \{(\emptyset, 0)\}$
2: for $k$ from $1$ to $n$ do
3:     $C(k) \leftarrow \emptyset$
4:     for all $1 \le i \le k$, all $1 \le l \le n - i + 1$, all $(S, t) \in C(k - i)$ do
5:         if $s_1[k - i + 1 : k] = s_2[l : l + i - 1]$ and $S \cap \{l, l+1, \ldots, l+i-1\} = \emptyset$ then
6:             $C(k) \leftarrow C(k) \cup \{(S \cup \{l, l+1, \ldots, l+i-1\}, t+1)\}$
7: return $\min\{t \colon (S, t) \in C(n)\}$
```

In the previous algorithm, we considered subsets of s_2 that have a common partition of particular size with the prefix of s_1. Here, we are going to ignore pieces of size one in every such partition: indeed, if one successfully matched longer pieces, then the pieces of size one will match as well.

With this in mind, one may consider only subsets of s_2 that can be cut into pieces of length at least two and matched to some subset of the prefix of s_1: the remaining part of the prefix is assumed to be filled with pieces of length one.

In addition to ignoring pieces of length one, we also plug the "holes" of size one in our subsets, since we can plug them only with pieces of size one and may do it immediately. To formalize that, we introduce the function Plug whose pseudocode is given in Algorithm 4. Say that there is a hole at position $i \in [n]$ in a set $S \subseteq [n]$ if $i \notin S$ whereas each of $i-1$ and $i+1$ either belong to S or does not belong to $[n]$.

Algorithm 4 The function Plug.

```
Input: subset $S \subseteq [n]$.
Output: hole-free superset of $S$ of minimum size.
1: for $i$ in $[n]$ do
2:     if $S$ has a hole at position $i$ then
3:         $S \leftarrow S \cup \{i\}$
4: return $S$
```

Now we are ready to describe the subproblem of our algorithm. For $0 \le k \le n$, let $C(k)$ be the set of all pairs (S,t), for which there exist a pair (T,β) such that $S = \text{Plug}(T)$, $t = \beta + |S| - |T|$, and one can cut the subsequence of s_2 specified by T into β blocks of size at least two and match them to some subsequence of $s_1[1:k]$. Thus, t is a number of blocks into which S can be cut so that all the blocks of size at least two match to $s_1[1:k]$ and blocks of size one (that is, plugged holes in T) match somewhere in s_1 (not necessarily in $s_1[1:k]$). The minimum size of a common partition is then the minimum value of $t+n-|S|$ such that (S,t) is contained in $C(n)$. Here, $n-|S|$ is the number of ignored but not plugged blocks of size one. Clearly, $C(k) \subseteq C(k')$ for $k < k'$. To compute $C(k)$, we first fix the length $i \neq 1$ of the long block $s_1[k-i+1:k]$ and $j \ge 0$ blocks of size one before it that we wish to ignore. Then, this long block should be matched somewhere in s_2, whereas $C(k-i-j)$ can be used to find the positions where the remaining long blocks should be matched. After we correctly match the long block somewhere in s_2, we append it to the corresponding S from $C(k-i-j)$, plug the holes and recalculate the number of blocks as the number of blocks in S before appending plus one plus the number of holes plugged after appending. If $i=0$, then we may skip the choice of j and simply get $C(k-1)$. The formal pseudocode is given in Algorithm 5.

Algorithm 5 Improved dynamic programming algorithm.

```
Input: strings $s_1, s_2$ of length $n$.
Output: the minimum size of a common partition of $s_1, s_2$.
1: $C(0) \leftarrow \{(\emptyset, 0)\}$
2: for $k$ from 1 to $n$ do
3:     $C(k) \leftarrow C(k-1)$     (case $i = 0$)
4:     for all $2 \le i \le k$, all $0 \le j \le n-i$, all $1 \le l \le n-i+1$, all $(S,t) \in C(k-i-j)$ do
5:         if $s_1[k-i+1:k] = s_2[l:l+i-1]$ and $S \cap \{l, l+1, \ldots, l+i-1\} = \emptyset$ then
6:             $S' \leftarrow \text{Plug}(S \cup \{l, l+1, \ldots, l+i-1\})$
7:             $C(k) \leftarrow C(k) \cup \{(S', t+1+|S'|-|S|)\}$
8: return $\min\{t+n-|S| : (S,t) \in C(n)\}$
```

▶ **Theorem 1.** *Algorithm 5 solves* MCSP *in time and space $O^*(\phi^n)$.*

Proof. Correctness. For every $(S, t) \in C(n)$, the total number of blocks in the corresponding partition is the number of long blocks plus the number of plugged holes $(= t)$ plus $|[n] \setminus S|$ blocks of size one we ignore. If the optimal partition contains no pieces longer than one, then we never fulfill the if condition, but nevertheless $C(n)$ will contain $(\emptyset, 0)$ dragged through the whole cycle (thanks to line 3), that corresponds to the partition of size n as required.

Running time. It is sufficient to count all the subsets S we considered. For that, it is convenient to treat every such subset as a sequence $v \in \{0,1\}^n$ without lonely zeros and ones. Let $f_0(n)$ be the number of such sequences that end up with zero and $f_1(n)$ be the number of such sequences that end up with one. Since every such sequence ending with one ends with at least two ones, we can write the recurrence relation $f_1(n) = \sum_{i=2}^{n-2} f_0(i)$. Similarly, $f_1(n) = f_0(n)$, hence $f_1(n) = \sum_{i=2}^{n-2} f_1(i)$ that, together with the initial conditions $f_1(2) = f_1(3) = 1$, describes the shifted sequence of Fibonacci numbers, thus $f_1(n) = O(\phi^n)$. ◀

4 Improving space to polynomial: FFT and Fibonacci encoding

In this section, we improve the space of the previously considered algorithm to polynomial. The following theorem provides a basic toolkit for this.

▶ **Theorem 2.** *Assume that for two strings $s_1, s_2 \in \Sigma^n$ and a parameter t one can construct a multivariate polynomial (of finite degree) $P(X)$ over a (finite) variable set X and a monomial $m = \prod_{x_i \in X} x_i^{d_i}$, where $d_i \in \mathbb{Z}_{\geq 0}$, with the following three properties:*

1. *Its coefficients are non negative and less than $W = 2^{\mathrm{poly}(n)}$.*
2. *For fixed values of variables X, one can compute $P(X)$ in time polynomial in the length of the binary representation of the values.*
3. *There is a common partition of s_1, s_2 of size t if and only if the coefficient of m in P is non-zero.*

Then, MCSP can be solved in time

$$O^*\left(\mathrm{polylog}\left(\prod_{x_i \in X} \deg_{x_i}(P)\right) \prod_{x_i \in X} \deg_{x_i}(P)\right)$$

and space

$$O^*\left(\mathrm{polylog}\left(\prod_{x_i \in X} \deg_{x_i}(P)\right)\right),$$

where $\deg_{x_i}(P)$ is the degree of variable x_i in P, that is, the maximum degree of x_i across the monomials of P with non-zero coefficients.

In the proof, we use the following theorem proved in [10] (Theorem 3.2).

▶ **Theorem 3.** *Let $P(x) = \sum_{i=0}^{d} p_i x^i$ be a polynomial of degree at most d with non-negative integer coefficients less than W. Given an arithmetic circuit $C(x, p)$ of size $\mathrm{polylog}(d, W)$ which evaluates P modulo a prime $p = d\,\mathrm{polylog}(d, W)$ at an integer point x, any coefficient of $P(x)$ can be found in time $d\,\mathrm{polylog}(d, W)$ and space $\mathrm{polylog}(d, W)$.*

Here we calculate modulo p just to prevent the numbers we operate from growing exponentially fast.

Proof of Theorem 2. Let $X = \{x_1, \ldots, x_q\}$. In order to apply Theorem 3, we need to build a univariate polynomial $Q(x)$ out of $P(x_1, \ldots, x_q)$. To do this, we use Kronecker substitution [16]:

$$Q(x) = P\left(x, x^{\deg_{x_1}(P)+1}, x^{(\deg_{x_1}(P)+1)(\deg_{x_2}(P)+1)}, \ldots, x^{\prod_{i=1}^{q-1}(\deg_{x_i}(P)+1)}\right).$$

That is, we replace x_i by $x^{\prod_{j=1}^{i-1}(\deg_{x_j}(P)+1)}$. Then, P contains a monomial $m = \prod_{i\in[q]} x_i^{a_i}$ if and only if Q contains a monomial $x^{\sum_{i\in[q]} a_i \cdot \prod_{j=1}^{i-1}(\deg_{x_j}(P)+1)}$.

Since $W = 2^{\mathrm{poly}(n)}$, $\mathrm{polylog}(d, W) = O^*(\mathrm{polylog}(d))$. ◀

As a warm up, we show how to reduce the space complexity of the 2^n dynamic programming Algorithm 3 to polynomial.

▶ **Theorem 4.** MCSP *can be solved in time $O^*(2^n)$ and polynomial space.*

Proof. We construct a series of polynomials P_k for $0 \le k \le n$ associated with the steps of the dynamic programming algorithm. Each P_k corresponds to $C(k)$ in the mentioned algorithm in the following way: P_k has a monomial $\alpha z^t y^{|S|} \prod_{i\in S} x^{2^{i-1}}$ for some $\alpha > 0$ depending on the monomial if and only if $(S, t) \in C(k)$. The pseudocode for computing P_n is given in Algorithm 6. It is not difficult to see that this is a polynomial time algorithm.

Algorithm 6 Computing P_n.

```
    Input: strings $s_1, s_2$ of length $n$, values $x, y, z$.
    Output: value of $P_n$ at the point $(x, y, z)$.
1: $P_0(x, y, z) \leftarrow 1$
2: $P_k(x, y, z) \leftarrow 0$ for all $1 \le k \le n$
3: for $k$ from 1 to $n$ do
4:     for all $1 \le i \le k$, all $1 \le l \le n - i + 1$ do
5:         if $s_1[k - i + 1 : k] = s_2[l : l + i - 1]$ then
6:             $P_k(x, y, z) \leftarrow P_k(x, y, z) + zy^i x^{2^{l+i-1} - 2^{l-1}} P_{k-i}(x, y, z)$
7: return $P_n(x, y, z)$
```

We claim that s_1 and s_2 have a common partition of size t if and only if P_n contains a monomial $z^t y^n x^{2^n-1}$. One direction is straightforward. Assume that there is a partition of s_1 into pieces $(l_1, r_1), \ldots, (l_t, r_t)$, where $l_1 = 1, r_t = n$ and $r_i = l_{i+1} - 1$ for all $i \in [t-1]$ and it corresponds to a partition of s_2 into pieces $(l'_1, r'_1), \ldots, (l'_t, r'_t)$ such that $s_1[l_i : r_i] = s_2[l'_i : r'_i]$ for all $i \in [t]$. Then P_{r_i} has a monomial

$$zy^{r_i - l_i + 1} x^{2^{r'_i} - 2^{l'_i - 1}} P_{r_{i-1}}.$$

Then, by induction, P_{r_i} has a monomial

$$z^i y^{\sum_{j=1}^{i}(r_j - l_j + 1)} x^{\sum_{j=1}^{i} 2^{r'_j} - 2^{l'_j - 1}}.$$

Hence, $P_n = P_{r_t}$ has a monomial

$$z^t y^{\sum_{j=1}^{t}(r_j - l_j + 1)} x^{\sum_{j=1}^{t} 2^{r'_j} - 2^{l'_j - 1}} = z^t y^n x^{2^n - 1}.$$

For the reverse direction, suppose P_n has a monomial $z^t y^n x^{2^n-1}$. It could only be obtained as a product $\prod_{i=1}^{t} zy^{r_i-l_i+1}x^{2^{r_i}-2^{l_i-1}}$ for some $(l_1, r_1), \ldots, (l_t, r_t)$ during the execution of the algorithm. An important invariant of each such terms is that the degree of y is equal to the Hamming weight of the degree of x. (Here, by the Hamming weight of an integer we mean the sum of the bits of its binary representation.)

As $\sum_{i=1}^{t}(r_i - l_i + 1)$ is equal to the degree of y, it is equal to n. If they are all disjoint, we have a valid partition. Suppose there exist i and j such that (l_i, r_i) and (l_j, r_j) intersect. Consider the product

$$(zy^{r_i-l_i+1}x^{2^{r_i}-2^{l_i-1}})(zy^{r_j-l_j+1}x^{2^{r_j}-2^{l_j-1}}) = z^2 y^{r_i+r_j-l_i-l_j+2} x^{\left(\sum_{t=l_i}^{r_i} 2^{t-1}\right)+\left(\sum_{t=l_j}^{r_j} 2^{t-1}\right)}.$$

Now, let us look at the Hamming weight of the degree of x. As it is the sum of $d = r_i + r_j - l_i - l_j + 2$ powers of 2, it is at most d. But as (l_i, r_i) and (l_j, r_j) intersect, there is at least one carry, so it is actually less than d. Then, when we multiply all the terms, as the Hamming weight of the sum is not more than the sum of Hamming weights, we have that Hamming weight is strictly less than the degree of y, so it is less than n. But the Hamming weight of $2^n - 1$ is equal to n, a contradiction.

Finally, we can solve MCSP by finding the smallest t such that $z^t y^n x^{2^n-1}$ is present in P_n. As we already have a polynomial time algorithm for P_n, its coefficients are not greater than $(n^2)^n = 2^{O(n \log n)}$ and total degree is $O^*(2^n)$, by Theorem 2 there exists an $O^*(2^n)$ time and polynomial space algorithm for MCSP. ◀

We are now going to infuse the previous algorithm with ideas from the improved dynamic programming and introduce a better encoding to get a speed up. In the previous section, we were encoding each segment (l, r) as $2^r - 2^{l-1}$. This is a natural way to represent a subset: this number has ones in its binary representation exactly at positions from l to r. Then we were using a property of the binary representation that the sum of encodings of two intersecting segments has a Hamming weight strictly less than sum of the Hamming weights of the terms. Another way to look at it: for any two intersecting segments (l_1, r_1),(l_2, r_2) there is a collection of nonintersecting segments $(l'_1, r'_1), \ldots, (l'_p, r'_p)$ such that:

$$\sum_{i=1}^{2} 2^{r_i} - 2^{l_i-1} = \sum_{i=1}^{p} 2^{r'_i} - 2^{l'_i-1} \quad \text{, but}$$

$$\sum_{i=1}^{2} r_i - l_i + 1 > \sum_{i=1}^{p} r'_i - l'_i + 1$$

Here we can take as (l'_i, r'_i) all the substrings of consecutive 1s in the binary representation of $\sum_{i=1}^{2} 2^{r_i} - 2^{l_i-1}$.

We are going to do something similar but sacrifice some of this clarity for efficiency. We will encode each segment (l, r) as $F'(r) - F'(l-1)$, where $F'(i)$ is the $(i+1)$-th Fibonacci number (so $F'(-1) = 0$ and $F'(0) = 1$). Then we will show that this encoding has a similar property as a binary one but only for segments of size greater than one.

▶ **Theorem 5.** MCSP *can be solved in time* $O^*(\phi^n)$ *and polynomial space.*

Proof. Consider a polynomial defined by Algorithm 7.

We call a piece of the partition *long* if it has length greater than one. The meaning of the indices is the following:

- k is the length of the prefix of s_1 that we are currently processing,

Algorithm 7 Computing P_n

```
    Input: strings $s_1, s_2$ of length $n$, values $x, y, z, w$.
    Output: value of $P_n$ at the point $(x, y, z, w)$.
1: $P_k \leftarrow 1$ for all $0 \le k \le n$
2: for $k$ from $1$ to $n$ do
3:     for all $1 < i \le j \le k$, all $1 \le l \le n - i + 1$, all $n - i + 1 \le q \le n$ do
4:         if $s_1[j - i + 1 : j] = s_2[l : l + i - 1]$ then
5:             $r \leftarrow l + i - 1$
6:             $P_k \leftarrow P_k + w^{q-r+1} y^{q-l+1} z^{q^2-(l-1)^2} x^{F'(q)-F'(l-1)} P_{j-i}$
7:             $P_k \leftarrow P_k + w^{q-(r-l)} y^q z^{q^2} x^{F'(q)-1} P_{j-i}$
8: return $P_n$
```

- j is the position of the rightmost symbol of the last long piece in $s_1[1 : k]$,
- i is the length of this long piece $s_1[j - i + 1 : j]$,
- l and r are the endpoints of a potential match of this long piece in s_2,
- q is the right endpoint of the block $s_2[r + 1 : q]$ that is cut into pieces of length one.

In short, we consider only long pieces in s_1 and every long piece $s_1[j - i + 1 : j]$ we correspond with a) block $s_2[l : q]$ that consists of $q - r + 1$ pieces: its match $s_2[l : r]$ and the group $s_2[r + 1 : q]$ of pieces of length one after it; b) block $s_2[1 : q]$ that consists of its match and two groups $s_2[1 : l - 1]$ and $s_2[r + 1 : q]$ of pieces of length one.

Now we need to show that s_1 and s_2 have a common partition of size t if and only if P_n contains a monomial $w^t x^{F'(n)-1} y^n z^{n^2}$. The if part is provided by the following lemma:

▶ **Lemma 6.** *If there is a common partition of s_1 and s_2 of size t, then there is a monomial $w^t x^{F'(n)-1} y^n z^{n^2}$ in P_n.*

Proof. Let $(l_1, r_1), \dots, (l_d, r_d)$ be a set of long pieces in the common partition. Without loss of generality we may assume that all (l_i, r_i) are sorted, that is $\forall i : r_i \le l_{i+1}$. Each piece (l_i, r_i) for $i > 1$ may contribute a monomial

$$w^{q_i - r_i + 1} x^{F'(q_i)-F'(l_i-1)} y^{q_i-(l_i-1)} z^{q_i^2-(l_i-1)^2},$$

where $q_i = l_{i+1} - 1$, $q_d = n$, and (l_1, r_1) may contribute a monomial $w^{q_1-r_1+l_1} x^{F'(q_1)-1} y^{q_1} z^{q_1^2}$, where $q_1 = l_2 - 1$. Here, for $i = 1$ we choose the monomial that takes into account pieces of size one before and after the long block (hence, $q_1 - r_1 + l_1$ in the exponent of w) and for $i > 1$ we choose the monomial that takes into account pieces of size one only after the long block (hence, $q_i - r_i + 1$ in the exponent of w). Thus, the sum of all the exponents of w is equal to the number of pieces in common partition, that is, t.

Multiplying all monomials together, we obtain the following monomial present in P_n:

$$w^{q_1-r_1+l_1} x^{F'(q_1)-1} y^{q_1} z^{q_1^2} \prod_{i=2}^{d} w^{q_i-r_i+1} x^{F'(q_i)-F'(l_i-1)} y^{q_i-(l_i-1)} z^{q_i^2-(l_i-1)^2} =$$
$$= w^t x^{F'(n)-1} y^n z^{n^2}.$$ ◀

Now we are getting to the tricky part. We need to prove that if P_n contains a monomial $w^t x^{F'(n)-1} y^n z^{n^2}$ then s_1 and s_2 have a common partition with k pieces.

For every set $S = \{(l_1, r_1), \dots, (l_d, r_d) \mid (l_i, r_i) \subset [n]\}$ of d intervals let $m(S)$ denote a monomial

$$m(S) := \prod_{i=1}^{d} x^{F'(r)-F'(l-1)} y^{r_i-l_i+1} z^{r_i^2-(l_i-1)^2} =$$
$$= x^{\sum_i F'(r_i)-F'(l_i-1)} y^{\sum_i r_i-l_i+1} z^{\sum_i r_i^2-(l_i-1)^2}.$$

The point of introducing such notation is that polynomial P_n is simply a sum of monomials of the form $w^\alpha m(S)$ for some $S = \{(l_1, r_1), \dots, (l_d, r_d)\}$ where all the intervals are long, so the $m(S)$ part relates to *what* is covered in s_2 and w^α part relates to *how* it is covered (that is, how many blocks).

It is easy to check that if there are no intersecting intervals in S and they cover the whole $[n]$ then $m(S) = x^{F'(n)-1} y^n z^{n^2}$ and we have a common partition of size α. What about intersecting intervals? The following lemma deals with this case:

▶ **Lemma 7.** *If S contains intersecting intervals and $m(S) = x^{F'(n)-1} y^n z^{n^2}$ then there is a set S' of non-intersecting intervals such that $m(S') = x^{F'(n)-1} y^\alpha z^\beta$ and $(\alpha < n) \vee (\beta < n^2)$.*

Proof. Suppose there are two long intersecting intervals (l_1, r_1) and (l_2, r_2) in S. We can show that there is a set of long non intersecting intervals $(l'_1, r'_1), \dots, (l'_p, r'_p)$ such that:

$$\sum_{j=1}^{p} (F'(r'_j) - F'(l'_j - 1)) = (F'(r_1) - F'(l_1 - 1)) + (F'(r_2) - F'(l_2 - 1)),$$

and one of the following two statements is true:

- $\sum_{j=1}^{p} (r'_j - l'_j + 1) < (r_1 - l_1 + 1) + (r_2 - l_2 + 1)$,
- $\sum_{j=1}^{p} (r'_j - l'_j + 1) = (r_1 - l_1 + 1) + (r_2 - l_2 + 1)$, but
  $\sum_{j=1}^{p} (r'^2_j - (l'_j - 1)^2) < (r_1^2 - (l_1 - 1)^2) + (r_2^2 - (l_2 - 1)^2)$.

If we replace (l_1, r_1) and (l_2, r_2) with this intervals we obtain a set S_1 such that $m(S_1) = y^\alpha z^\beta x^{F'(n)}$ and $(\alpha < n) \vee (\beta < n^2)$. We can keep replacing intersecting pairs with non-intersecting intervals preserving the value of $\deg_x$. It is clear that this process will eventually stop since $\deg_y \geq 0$ and $\deg_z \geq 0$, and thus the resulting set S' is well-defined and consists of non-intersecting intervals.

It remains to present the set $\{(l'_1, r'_1), \dots, (l'_p, r'_p)\}$ for every two intersecting intervals $\{(l_1, r_1), (l_2, r_2)\}$.

Let $T = \{(a_i, b_i), \dots (a_p, b_p)\}$, such that $\forall i \in [p] : 0 \leq a_i < b_i - 1 \leq n - 1$ or $a_i = b_i$ (the later represents an interval of size 0 and is used only to reduce the number of cases in the analysis). We will use the following notation:

- $f(T) = \sum_{i=1}^{|T|} F'(b_i) - F'(a_i)$,
- $g(T) = \sum_{i=1}^{|T|} b_i - a_i$,
- $h(T) = \sum_{i=1}^{|T|} b_i^2 - a_i^2$.

Let A be a set $\{(a, b), (c, d)\}$ such that $(b \geq d > a \geq c) \vee (b \geq d > c \geq a)$. As there is no difference in analysis (we may replace $(a, b), (c, d)$ with $(a, d), (b, c)$), we consider only the first case. We are going to go through rigorous case analysis to show that we can always construct a set B such that $f(A) = f(B)$ and either $g(A) > g(B)$ or $g(A) = g(B)$ but $h(A) > h(B)$:

1. If $d > a + 2 \vee d = a + 1$ and $a > c + 2 \vee a = c + 1$ then

$$B = \{(c, a - 1), (a + 1, d), (b - 1, b + 1)\},$$

$$g(B) = d - c < b - a + d - c = g(A);$$

2. If $d > a + 2 \vee d = a + 1$ and $a = c + 2$ then
$$B = \{(c-2, c), (a+1, d), (b-1, b+1)\},$$
$$g(B) = d - a + 3 < b - a + d - c = g(A);$$
3. If $d = a + 2$ then
$$B = \{(c, d-1), (b-1, b+1)\},$$
$$g(B) = d - a + 1 < b - a + d - c = g(A);$$
4. If $a = c$ and $d > a + 3 \vee d = a + 2$ and $b - a > 2$ then
$$B = \{(a-2, a), (a+2, d), (b-1, b+1)\},$$
$$g(B) = d - a + 2 < b - a + d - c = g(A);$$
5. If $a = c$ and $b - a = 2$ then
$$B = \{(a, b+1)\},$$
$$g(B) = b - a + 1 < b - a + d - c = g(A);$$
6. If $a = c$ and $d = a + 3$ and $b > a + 3$ then
$$B = \{(a-2, d-1), (b-1, b+1)\},$$
$$g(B) = d - a + 3 < b - a + d - c = g(A);$$
7. If $a = c$ and $d = b$ and $b = a + 3$
$$B = \{(a-2, b+1)\},$$
$$g(B) = d - a + 3 = b - a + d - c = g(A),$$
$$h(B) = (a+4)^2 - (a-2)^2 = 12a + 12 < 12a + 18 = 2((a+3)^2 - a^2) = h(A).$$

In some of the cases, it may turn out that for some interval from B its left endpoint is negative or its right endpoint is greater than n. In fact, the latter case is impossible: if long interval (a, b) covers at least $n + 1$ then $F'(b) - F'(a) > F'(n) - 1$. In the former case we will use the additional manipulations with B:

2. If $c = 0$ then we drop $(-2, 0)$. If $c = 1$ then we replace $(-1, 1)$ with $(0, 2)$;
4. If $a = 0$ then we drop $(-2, 0)$. If $a = 1$ then we replace $(-1, 1)$ with $(0, 2)$;
6. If $a = 0$ then we replace $(-2, 2)$ with $(0, 2)$. If $a = 1$ then we replace $(-1, 3)$ with $(2, 4)$;
7. If $a = 0$ then we replace $(-2, 4)$ with $(0, 4)$. If $a = 1$ then we replace $(-1, 5)$ with $(4, 6)$.

Now to get the desired result we need to apply this case analysis to $\{(l_1-1, r_1), (l_2-1, r_2)\}$, get a set $B = \{(a'_1, b'_1) \dots (a'_p, b'_p)\}$ and get a collection $(a'_1 + 1, b'_1), \dots, (a'_p + 1, b'_p)$ which would satisfy all the intended properties. ◀

Suppose that there exists a set of intervals S with intersections and $m(S) = x^{F'(n)-1}y^n z^{n^2}$. Let S' be the corresponding set from the statement of the lemma. Since S' contains only non-intersecting intervals, $\deg_x(m(S')) \leq F'(n) - 1$ and equality can be achieved only if this intervals cover the whole $[n]$. If so, then $\deg_y(m(S')) = n = \deg_y(m(S))$ and $\deg_z(m(S')) = n^2 = \deg_z(m(S))$. But at least one of $\deg_y$ and $\deg_z$ decreases while we go from S to S', which is a contradiction. Thus, the monomial $w^t x^{F'(n)-1}y^n z^{n^2}$ may only correspond to some set without intersections that covers the whole $[n]$ and in turn corresponds to a common partition of size t.

Finally, we can solve MCSP problem by finding the smallest t such that $w^t x^{F'(n)-1}y^n z^{n^2}$ is present in P_n. As we already have a polynomial time algorithm for P_n, its coefficients are not greater than $(2n^3)^n = 2^{O(n \log n)}$ and total degree is $O^*(\phi^n)$, by Theorem 2 there is $O^*(\phi^n)$ time and polynomial space algorithm for MCSP. ◀

5 Subexponential running time for small alphabets: hybrid strategy

For the case of a constant size alphabet, we provide a subexponential time algorithm, that collects information about partitions of s_1 and s_2 separately and then just checks if they have any partition in common. The main idea is to divide all the pieces in a partition into two groups: long ones and short ones. The good thing about long ones is that there are not too many of those due to their length. The good thing about short ones is that there are not too many possible short strings over our alphabet due to its small size. By balancing these two cases, we get the desired running time.

▶ **Theorem 8.** MCSP *can be solved in time and space* $2^{O\left(\frac{n\log|\Sigma|\log\log n}{\log n}\right)}$ *when* $|\Sigma| = n^{o(1)}$.

Proof. Let μ be a parameter whose value we will choose later. For every multiset S of strings no longer than μ, we define *a histogram* $\operatorname{hist}_\mu(S) = \{(s, \operatorname{count}(S,s)) \mid s \in \Sigma^*, |s| \le \mu\}$, where $\operatorname{count}(S,s)$ stands for a number of occurrences of s in S. As before, for both input strings we construct sets of their partitions, but now we treat each partition $\mathcal{P}$ as a pair $(L, \operatorname{hist}_\mu(S))$ of a multiset of long pieces $L = \{s \in \mathcal{P}, |s| > \mu\}$ and a histogram of a multiset of short ones $S = \{s \in \mathcal{P}, |s| \le \mu\}$.

The construction of all possible partitions goes as follows: we iterate over all possible multisets of long pieces and for each of them we compute all possible histograms of short ones. As the number of long pieces is not greater than n/μ, there are at most $\binom{2n}{2n/\mu}$ ways to choose their ends.

Once we have fixed the long pieces, we want to enumerate all possible ways to cut the rest into small pieces. In order to do this, we use dynamic programming. Let $s_1, s_2, \dots, s_k$ be the multiset of strings after removing all the long pieces. By $R(\{s_1, s_2, \dots, s_k\})$ we define the set of all possible ways to cut these strings into pieces no longer than μ. Then,

$$R(\{s_1, s_2, \dots, s_k\}) = \\ = \bigcup_{i=1}^{\min\{\mu,|s_k|\}} \{C \cup s_k[|s_k| - i + 1 : |s_k|] \mid C \in R(\{s_1, s_2, \dots, s_k[1 : |s_k| - i]\})\},$$

and $R(\{s_1, s_2, \dots, s_k\})$ may be computed in $O^*(n^{|\Sigma|^\mu})$, as $n^{|\Sigma|^\mu}$ is the upper bound on the number of all possible histograms: $\operatorname{count}(S,s) \le n$ for every $s \in S$ and there are at most $|\Sigma|^{\mu+1}$ distinct s, and it takes polynomial in n and linear in size time to obtain R if all smaller ones are already computed.

Thus, the total time is $O^*\left(\binom{2n}{2n/\mu} \cdot n^{|\Sigma|^\mu}\right)$ and we wish to choose μ so that this time is small as possible. Since $\binom{2n}{2n/\mu} = 2^{O(\frac{n}{\mu}\log\mu)}$ and $n^{|\Sigma|^\mu} = 2^{O(|\Sigma|^\mu \log n)}$, we may choose $\mu = c\log_2 n/\log_2|\Sigma|$ with $c < 1$, and then the total time is $2^{O\left(\frac{n\log|\Sigma|\log\log n}{\log n}\right)}$. ◀

6 Lower bounds: reductions to MIS and to binary alphabet

In this section, we prove the ETH based lower bounds: assuming Exponential Time Hypothesis, the best upper bounds for MCSP and MCSP^c are $2^{\Omega(n)}$ and $2^{\Omega(n/\log n)}$, respectively.

▶ **Lemma 9.** *Assuming* ETH, *there is no* $2^{o(n)}$ *time algorithm for* MCSP.

Proof. [8] proves NP-hardness of MCSP by reducing the maximum independent set on degree-3 graphs (3-MIS) to MCSP. The reduction is linear: given a graph with n nodes, it produces an instance of MCSP with strings of length $O(n)$. In turn, [12] shows that there is no $2^{o(n)}$ time algorithm for 3-MIS under ETH. This implies that the same is true for MCSP. ◀

▶ **Theorem 10.** *Assuming* ETH, *there is no* $2^{o(n/\log n)}$ *time algorithm for* MCSP *on constant size alphabets.*

Proof. We show how to reduce an instance of the general MCSP of length n to an instance of MCSP over binary alphabet that is $4\log n$ times longer. Since the general MCSP cannot be solved in time $2^{o(n)}$ under ETH, we obtain a lower bound $2^{\Omega(n/\log n)}$ for its constant-size alphabet version.

Consider an instance of MCSP that consists of strings s_1 and s_2 of length n. Since there are at most n distinct symbols in both strings, we can encode each symbol c via binary string $\mathrm{b}(c)$ of length $\log_2 n$. For every symbol c we define *a gadget* $\psi(c)$ as a string

$$\psi(c) = 0\ \mathrm{b}(c)[1]\ 0\ \mathrm{b}(c)[2]\ 0 \dots 0\ \mathrm{b}(c)[\log_2 n]\ 0\ 1\ 1\ 1\ 0\ \mathrm{b}(c)[1]\ 0\ \mathrm{b}(c)[2]\ 0 \dots 0\ \mathrm{b}(c)[\log_2 n]\ 0.$$

We call the central 1 of a gadget its *pivot.*

We transform $s_1 = s_1[1]s_1[2]\dots s_1[n]$ into a string $s_1' = \psi(s_1[1])\psi(s_1[2])\dots\psi(s_1[n])$ and $s_2 = s_2[1]s_2[2]\dots s_2[n]$ into a string $s_2' = \psi(s_2[1])\psi(s_2[2])\dots\psi(s_2[n])$. It is clear that (s_1', s_2') is an instance of MCSP over binary alphabet of size $n \cdot \lceil 4\log_2 n + 5\rceil$. We show that there is a common partition of s_1 and s_2 of size at most t if and only if there is a common partition of s_1' and s_2' of size at most t.

Necessity. Given a common partition of s_1 and s_2, we can transform it into a common partition of s_1' and s_2' of the same size simply by replacing each block $b = b[1]b[2]\dots b[|b|]$ with $\psi(b) = \psi(b[1])\psi(b[2])\dots\psi(b[|b|])$.

Sufficiency. Consider a common partition of s_1' and s_2' of size t. Call the block of this partition *long* if it contains at least two pivots of some gadgets. Since the pivot and its neighbors are the only consecutive 1's in gadget, if some long block b_1 of s_1' matched with some long block b_2 of s_2' then all the pivots of b_1 matched with the corresponding pivots of b_2. Note that long blocks, along with each pivot, contain information about the corresponding symbols of its gadgets, as this information is duplicated on both sides of the pivot. This means that we can correspond every long block that contains pivots of gadgets $\psi(c[1])\psi(c[2])\dots\psi(c[|c|])$ with substring $c[1]c[2]\dots c[|c|]$ of s_1 and s_2.

Now we are ready to present a common partition of s_1 and s_2 of size at most t: for every long block in (s_1', s_2') we take the corresponding block in (s_1, s_2) and cut the remains into blocks of size one. Clearly, the result is a common partition of s_1 and s_2: blocks of length at least two do not intersect and are contained in both strings since so do the corresponding long blocks. The remaining blocks of size one match since s_1 and s_2 consists of the same number of each symbol. The size of the constructed partition is equal to the number of blocks in the considered partition of s_1' and s_2' that contain at least one pivot and hence is no more than t. ◀

7 Open problems

There are three natural questions left open by the present studies.

1. Close the gap between a lower bound $2^{\Omega(n/\log n)}$ and an upper bound $2^{O(n\log\log n/\log n)}$ for constant size alphabets.
2. Design a moderately exponential time (of the form c^n for $c < 2$) algorithm for the case of more than two input strings.
3. Roughly, it is the "no-holes" property that allows us to improve the 2^n upper bound for MCSP. What are other problems with the same effect?

References

1 Andreas Björklund, Thore Husfeldt, and Mikko Koivisto. Set partitioning via inclusion-exclusion. *SIAM J. Comput.*, 39(2):546–563, 2009. doi:10.1137/070683933.

2 Laurent Bulteau, Guillaume Fertin, Christian Komusiewicz, and Irena Rusu. A fixed-parameter algorithm for minimum common string partition with few duplications. In Aaron E. Darling and Jens Stoye, editors, *Algorithms in Bioinformatics - 13th International Workshop, WABI 2013, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, volume 8126 of *Lecture Notes in Computer Science*, pages 244–258. Springer, 2013. doi:10.1007/978-3-642-40453-5_19.

3 Laurent Bulteau and Christian Komusiewicz. Minimum common string partition parameterized by partition size is fixed-parameter tractable. In Chandra Chekuri, editor, *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 102–121. SIAM, 2014. doi:10.1137/1.9781611973402.8.

4 Marek Chrobak, Petr Kolman, and Jirí Sgall. The greedy algorithm for the minimum common string partition problem. *ACM Trans. Algorithms*, 1(2):350–366, 2005. doi:10.1145/1103963.1103971.

5 Graham Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Trans. Algorithms*, 3(1):2:1–2:19, 2007. doi:10.1145/1219944.1219947.

6 Peter Damaschke. Minimum common string partition parameterized. In Keith A. Crandall and Jens Lagergren, editors, *Algorithms in Bioinformatics, 8th International Workshop, WABI 2008, Karlsruhe, Germany, September 15-19, 2008. Proceedings*, volume 5251 of *Lecture Notes in Computer Science*, pages 87–98. Springer, 2008. doi:10.1007/978-3-540-87361-7_8.

7 Bin Fu, Haitao Jiang, Boting Yang, and Binhai Zhu. Exponential and polynomial time algorithms for the minimum common string partition problem. In Weifan Wang, Xuding Zhu, and Ding-Zhu Du, editors, *Combinatorial Optimization and Applications - 5th International Conference, COCOA 2011, Zhangjiajie, China, August 4-6, 2011. Proceedings*, volume 6831 of *Lecture Notes in Computer Science*, pages 299–310. Springer, 2011. doi:10.1007/978-3-642-22616-8_24.

8 Avraham Goldstein, Petr Kolman, and Jie Zheng. Minimum common string partition problem: Hardness and approximations. *Electron. J. Comb.*, 12, 2005. URL: http://www.combinatorics.org/Volume_12/Abstracts/v12i1r50.html.

9 Isaac Goldstein and Moshe Lewenstein. Quick greedy computation for minimum common string partition. *Theor. Comput. Sci.*, 542:98–107, 2014. doi:10.1016/j.tcs.2014.05.006.

10 Alexander Golovnev, Alexander S. Kulikov, and Ivan Mihajlin. Families with infants: Speeding up algorithms for NP-hard problems using FFT. *ACM Trans. Algorithms*, 12(3):35:1–35:17, 2016. doi:10.1145/2847419.

11 Haitao Jiang, Binhai Zhu, Daming Zhu, and Hong Zhu. Minimum common string partition revisited. *J. Comb. Optim.*, 23(4):519–527, 2012. doi:10.1007/s10878-010-9370-2.

12 David S. Johnson and Mario Szegedy. What are the least tractable instances of max independent set? In Robert Endre Tarjan and Tandy J. Warnow, editors, *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, 17-19 January 1999, Baltimore, Maryland, USA*, pages 927–928. ACM/SIAM, 1999. URL: http://dl.acm.org/citation.cfm?id=314500.315093.

13 Haim Kaplan and Nira Shafrir. The greedy algorithm for edit distance with moves. *Inf. Process. Lett.*, 97(1):23–27, 2006. doi:10.1016/j.ipl.2005.08.010.

14 Mikko Koivisto. Optimal 2-constraint satisfaction via sum-product algorithms. *Inf. Process. Lett.*, 98(1):24–28, 2006. doi:10.1016/j.ipl.2005.11.013.

15 Petr Kolman and Tomasz Walen. Reversal distance for strings with duplicates: Linear time approximation using hitting set. *Electron. J. Comb.*, 14(1), 2007. URL: http://www.combinatorics.org/Volume_14/Abstracts/v14i1r50.html.

16 Leopold Kronecker. Grundzüge einer arithmetischen theorie der algebraischen grössen. *J. Reine Angew. Math.*, 92:1–122, 1882.

17 Dana Shapira and James A. Storer. Edit distance with move operations. *J. Discrete Algorithms*, 5(2):380–392, 2007. doi:10.1016/j.jda.2005.01.010.

18 Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.*, 348(2-3):357–365, 2005. doi:10.1016/j.tcs.2005.09.023.