LCNS, Vol 2751, pp. 195–209, 2003

On Exact and Approximation Algorithms for Distinguishing Substring Selection

Jens Gramm*, Jiong Guo**, and Rolf Niedermeier**

Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Sand 13,
D-72076 Tübingen, Fed. Rep. of Germany
{gramm,guo,niedermr}@informatik.uni-tuebingen.de

Abstract. The NP-complete Distinguishing Substring Selection problem (DSSS for short) asks, given a set of "good" strings and a set of "bad" strings, for a solution string which is, with respect to Hamming metric, "away" from the good strings and "close" to the bad strings.
Studying the parameterized complexity of DSSS, we show that DSSS is W[1]-hard with respect to its natural parameters. This, in particular, implies that a recently given polynomial-time approximation scheme (PTAS) by Deng *et al.* cannot be replaced by a so-called *efficient* polynomial-time approximation scheme (EPTAS) unless an unlikely collapse in parameterized complexity theory occurs.
By way of contrast, for a special case of DSSS, we present an exact fixed-parameter algorithm solving the problem efficiently. In this way, we exhibit a sharp border between fixed-parameter tractability and intractability results.
Keywords. Algorithms and complexity, parameterized complexity, approximation algorithms, exact algorithms, computational biology.

1 Introduction

Recently, there has been strong interest in developing polynomial-time approximation schemes (PTAS's) for several string problems motivated by computational molecular biology [6, 15, 16]. More precisely, all these problems adhere to a scenario where we are looking for a string which is "close" to a given set of strings and, in some cases, which shall also be "far" from another given set of strings (see Lanctot *et al.* [14] for an overview on these kinds of problems and their applications in molecular biology). The underlying distance measure is Hamming metric. The list of problems in this context includes Closest (Sub)String [15], Consensus Patterns [16], and Distinguishing (Sub)String Selection [6]. All these problems are NP-complete, hence polynomial-time exact solutions are out of reach and PTAS's might be the best one can hope for. PTAS's, however, often carry huge hidden constant factors that make them useless from a practical point of view. This difficulty also occurs with the problems mentioned above. Hence, two natural questions arise.

1. To what extent can the above approximation schemes be made really practical? [1]
2. Are there, besides pure heuristics, theoretically satisfying approaches to solve these problems exactly, perhaps based on a parameterized point of view [2, 10]?

* Supported by the Deutsche Forschungsgemeinschaft (DFG), project OPAL (optimal solutions for hard problems in computational biology), NI 369/2.
** Partially supported by the Deutsche Forschungsgemeinschaft (DFG), junior research group PIAF (fixed-parameter algorithms), NI 369/4.
[1] As Fellows [10] put it in his recent survey, "it would be interesting to sort out which problems with PTAS's have any hope of practical approximation". Also see the new survey by Downey [7] for a good exposition on this issue.

In this paper, we address both these questions, focusing on the Distinguishing Substring Selection problem (DSSS):

Input: Given an alphabet Σ of constant size, two sets of strings over Σ,
- $S_g = \{s_1, \ldots, s_{k_g}\}$, each string of length at least L (the "good" strings),[2]
- $S_b = \{s'_1, \ldots, s'_{k_b}\}$, each string of length at least L (the "bad" strings),

and two non-negative integers d_g and d_b.
Question: Is there a length-L string s over Σ such that
- in every $s_i \in S_g$, for *every* length-L substring t_i, $d_H(s, t_i) \geq d_g$ and
- every $s'_i \in S_b$ has *at least one* length-L substring t'_i with $d_H(s, t'_i) \leq d_b$?

Here, $d_H(s, t_i)$ denotes the Hamming distance between strings s and s_i. Following Deng *et al.* [6], we distinguish DSSS from Distinguishing String Selection (DSS) in which all good and bad strings have the same length L; note that Lanctot *et al.* [14] did not make this distinction and denoted both problems as DSS.
The above mentioned Closest Substring is the special case of DSSS where the set of good strings is empty. Furthermore, Closest String is the special case of Closest Substring where all given strings and the goal string have the same length. Since Closest String is known to be NP-complete [12, 14], the NP-completeness of Closest Substring and DSSS immediately follows.
All the mentioned problems carry at least two natural input parameters ("distance" and "number of input strings") which often are small in practice when compared to the overall input size. This leads to the important question whether the seemingly inevitable "combinatorial explosion" in exact algorithms for these problems can be restricted to some of the parameters—this is the parameterized complexity approach [2, 7, 8, 10]. In [13], it was shown that for Closest String this can successfully be done for the "distance" parameter as well as the parameter "number of input strings". However, Closest String is the easiest of these problems. As to Closest Substring, fixed-parameter intractability (in the above sense of restricting combinatorial explosion to parameters) was recently shown with respect to the parameter "number of input strings" [11]. More precisely, a proof of W[1]-hardness (see [8] for details on parameterized complexity theory) was given. It was conjectured that Closest Substring is also fixed-parameter intractable with respect to the distance parameter, but it is an open question to prove (or disprove) this statement.[3]
Now, in this work, we show that DSSS is fixed-parameter intractable (i.e., W[1]-hard) with respect to all natural parameters as given in the problem definition and, thus, in particular, with respect to the distance parameters. Besides of the interest in its own concerning the impossibility[4] of efficient exact fixed-parameter algorithms, this result also has important consequences concerning approximation algorithms. More precisely, our result implies that no efficient polynomial-time approximation scheme (EPTAS) in the sense of Cesati and Trevisan [5] is available for DSSS. As a consequence, there is strong theoretical support for the claim that the recent PTAS of Deng *et al.* [6] cannot be made practical. In addition, we indicate an instructive border between fixed-parameter tractability and fixed-parameter intractability for DSSS which lies between alphabets of size two and alphabets of size greater than two. Two proofs in Sect. 4 had to be omitted due to the lack of space.

[2] Deng *et al.* [6] let all good strings be of same length L; we come back to this restriction in Sect. 4. The terminology "good" and "bad" has its motivation in the application [14] of designing genetic markers to distinguish the sequences of harmful germs (to which the markers should bind) from human sequences (to which the markers should not bind).
[3] In fact, more hardness results for *unbounded* alphabet size are known [11]. Here, we refer to the practically most relevant case of constant alphabet size.
[4] Unless an unlikely collapse in structural parameterized complexity theory occurs [10].

2 Preliminaries and Previous Work

Parameterized Complexity. Given a graph $G = (V, E)$ with vertex set V, edge set E, and a positive integer k, the NP-complete Vertex Cover problem is to determine whether there is a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C. Vertex Cover is *fixed-parameter tractable* with respect to the parameter k. There now are algorithms solving it in less than $O(1.3^k + kn)$ time. The corresponding complexity class is called FPT. By way of contrast, consider the NP-complete Clique problem: Given a graph $G = (V, E)$ and a positive integer k, Clique asks whether there is a subset of vertices $C \subseteq V$ with at least k vertices such that C forms a clique by having all possible edges between the vertices in C. Clique appears to be *fixed-parameter intractable*: It is *not* known whether it can be solved in $f(k) \cdot n^{O(1)}$ time, where f might be an arbitrarily fast growing function only depending on k.
Downey and Fellows developed a completeness program for showing fixed-parameter intractability [8]. We very briefly sketch some integral parts of this theory.
Let $L, L' \subseteq \Sigma^* \times \mathbf{N}$ be two parameterized languages.[5] For example, in the case of Clique, the first component is the input graph and the second component is the positive integer k, that is, the parameter. We say that L *reduces to* L' *by a standard parameterized m-reduction* if there are functions $k \mapsto k'$ and $k \mapsto k''$ from $\mathbf{N}$ to $\mathbf{N}$ and a function $(x, k) \mapsto x'$ from $\Sigma^* \times \mathbf{N}$ to Σ^* such that

1. $(x, k) \mapsto x'$ is computable in time $k''|x|^c$ for some constant c and
2. $(x, k) \in L$ iff $(x', k') \in L'$.

Observe that in the subsequent section we will present a reduction from Clique to DSSS, mapping the Clique parameter k into all *four* parameters of DSSS; i.e., k' in fact is a four-tuple $(k_g, k_b, d_g, d_b) = (1, \binom{k}{2}, k + 3, k - 2)$ (see Sect. 3.1 for details). Notably, most reductions from classical complexity turn out *not* to be parameterized ones. The basic reference degree for fixed-parameter intractability, W[1], can be defined as the class of parameterized languages that are equivalent to the Short Turing Machine Acceptance problem (also known as the k-Step Halting problem). Here, we want to determine, for an input consisting of a nondeterministic Turing machine M and a string x, whether or not M has a computation path accepting x in at most k steps. This can trivially be solved in $O(n^{k+1})$ time and we would be surprised if this can be much improved. Therefore, this is the parameterized analogue of the Turing Machine Acceptance problem that is the basic generic NP-complete problem in classical complexity theory, and the conjecture that FPT $\neq$ W[1] is very much analogous to the conjecture that P $\neq$ NP. Other problems that are W[1]-*hard* (and also W[1]-*complete*) include Clique and Independent Set, where the parameter is the size of the relevant vertex set [8]. W[1]-hardness gives a concrete indication that a parameterized problem with parameter k is unlikely to allow for a solving algorithm with $f(k) \cdot n^{O(1)}$ running time, i.e., restricting the combinatorial explosion to k.

Approximation. In the following, we explain some basic terms of approximation theory, thereby restricting to minimization problems. Given a minimization problem, a solution of the problem is $(1 + \epsilon)$-*approximate* if the cost of the solution is d, the cost of an optimal solution is d_{opt}, and $d/d_{opt} \leq 1 + \epsilon$. A *polynomial-time approximation scheme (PTAS)* is an algorithm that computes, for any given real $\epsilon > 0$, a $(1+\epsilon)$-approximate solution in polynomial time where ϵ is considered to be constant.

[5] Generally, the second component (representing the parameter) can also be drawn from Σ^*; for most cases, assuming the parameter to be a positive integer (or a tuple of positive integers) is sufficient.

For more details on approximation algorithms, refer to [4]. Typically, PTAS's have a running time $n^{O(1/\epsilon)}$, often with large constant factors hidden in the exponent which make them infeasible already for moderate approximation ratio. Therefore, Cesati and Trevisan [5] proposed the concept of an *efficient* polynomial-time approximation scheme (EPTAS) where the PTAS is required to have an $f(\epsilon) \cdot n^{O(1)}$ running time where f is an arbitrary function depending only on ϵ and not on n. Notably, most known PTAS's are *not* EPTAS's [7, 10].

Previous Work. Lanctot *et al.* [14] initiated the research on the algorithmic complexity of distinguishing string selection problems. In particular, besides showing NP-completeness (an independent NP-completeness result was also proven by Frances and Litman [12]), they gave a polynomial-time factor-2-approximation for DSSS. Building on PTAS algorithms for Closest String and Closest Substring [15], Deng *et al.* [6] recently gave a PTAS for DSSS.
There appear to be no nontrivial results on exact or fixed-parameter algorithms for DSSS. Since Closest Substring is a special case of DSSS, however, the fixed-parameter intractability results for Closest Substring [11] also apply to DSSS, implying that DSSS is W[1]-hard with respect to the parameter "number of input strings". Finally, the special case DSS of DSSS (where all given input strings have exactly the same length as the goal string) is solvable in $O((k_g + k_b) \cdot L \cdot (\max\{d_b + 1, (d'_g + 1) \cdot (|\Sigma| - 1)\})^{d_b})$ time with $d'_g = L - d_g$ [13], i.e., for constant alphabet size, it is fixed-parameter tractable with respect to the aggregate parameter (d'_g, d_b). In a sense, DSS relates to DSSS as Closest String relates to Closest Substring and, thus, DSS should be regarded as considerably easier and of less practical importance than DSSS.

3 Fixed-Parameter Intractability of DSSS

We show that DSSS is, even for binary alphabet, W[1]-hard with respect to the aggregate parameter (d_g, d_b, k_g, k_b). This also means hardness for every single of these parameters. With [5], this implies that DSSS does not have an EPTAS.
To simplify presentation, in the rest of this section we use the following technical terms. Regarding the good strings, we say that a length-L string s *matches* an $s_i \in S_g$ or, equivalently, s is a *match* for s_i, if $d_H(s, t_i) \geq d_g$ for every length-L substring t_i of s_i. Regarding the bad strings, we say that a length-L string s *matches* an $s'_i \in S_b$ or, equivalently, s is a *match* for s'_i, if there is a length-L substring t'_i of s'_i with $d_H(s, t'_i) \leq d_b$. Both these notions of matching for good as well as for bad strings generalize to sets of strings in the natural way.
Our hardness proof follows a similar structure as the W[1]-hardness proof for Closest Substring [11]. We give a parameterized reduction from Clique to DSSS. Here, however, the reduction has novel features in two ways. Firstly, from the technical point of view, the reduction becomes much more compact and, thus, more elegant. Secondly, for Closest Substring with binary alphabet, we could only show W[1]-hardness with respect to the number of input strings. Here, however, we can show W[1]-hardness with respect to, among others, parameters d_g and d_b. This has strong implications: Here, we can conclude that DSSS has no EPTAS, which is an open question for Closest Substring [11].

3.1 Reduction from Clique to DSSS

A Clique instance is given by an undirected graph $G = (V, E)$, with a set $V = \{v_1, v_2, \ldots, v_n\}$ of n vertices, a set E of m edges, and a positive integer k denoting the desired clique size. We describe how to generate two sets of strings over alphabet

$\{0,1\}$, S_g (containing one string s_g of length $L := nk+5$) and S_b (containing $\binom{k}{2}$ strings, each of length $m \cdot (2nk+5)+(m-1)$), such that G has a clique of size k iff there is a length-L string s which is a match for S_g and also for S_b; this means that $d_H(s, s_g) \geq d_g$ with $S_g := \{s_g\}$ and $d_g := k+3$, and every $s'_b \in S_b$ has a length-L substring t'_b with $d_H(s, t'_b) \leq d_b$ and $d_b := k-2$. In the following we use "$\circ$" to denote the concatenation of strings.

Good string. $S_g := \{s_g\}$ where $s_g = 0^L$, the all-zero string of length L.

Bad strings. $S_b := \{s'_{1,2}, \ldots, s'_{1,k}, s'_{2,3}, s'_{2,4}, \ldots, s'_{k-1,k}\}$, where *every* $s'_{i,j}$ has length $m \cdot (2nk+5)+(m-1)$ and encodes the whole graph; in the following, we describe how we generate a string $s'_{i,j}$.

We encode a vertex $v_r \in V$, $1 \leq r \leq n$, in a length-n string by setting the rth position of this string to "1" and all other positions to "0", i.e.,

$$\langle \text{vertex}(v_r) \rangle := 0^{r-1}10^{n-r}.$$

In $s'_{i,j}$, we encode an edge $\{v_r, v_s\} \in E$, $1 \leq r < s \leq n$, by a length-(nk) string

$$\langle \text{edge}(i,j,\{v_r,v_s\}) \rangle := \underbrace{0^n \ldots 0^n}_{(i-1)} \circ \langle \text{vertex}(v_r) \rangle \circ \underbrace{0^n \ldots 0^n}_{(j-i-1)} \circ \langle \text{vertex}(v_s) \rangle \circ \underbrace{0^n \ldots 0^n}_{(k-j)}.$$

Furthermore, we define

$$\langle \text{edge\_block}(i,j,\{v_r,v_s\}) \rangle := \langle \text{edge}(i,j,\{v_r,v_s\}) \rangle \circ 01110 \circ \langle \text{edge}(i,j,\{v_r,v_s\}) \rangle.$$

We choose this way of constructing the $\langle \text{edge\_block}(\cdot,\cdot,\cdot) \rangle$ strings for the following reason: Let $\langle \text{edge}(i,j,\{v_r,v_s\}) \rangle[h_1,h_2]$ denote the substring of $\langle \text{edge}(i,j,\{v_r,v_s\}) \rangle$ ranging from position h_1 to position h_2. Then, every length $L = nk+5$ substring of $\langle \text{edge\_block}(\cdot,\cdot,\cdot) \rangle$ which contains the "01110" substring will have the form

$$\langle \text{edge}(i,j,\{v_r,v_s\}) \rangle[h, nk] \circ 01110 \circ \langle \text{edge}(i,j,\{v_r,v_s\}) \rangle[1, h-1]$$

for $1 \leq h \leq nk+1$. This will be important because our goal is that a match for a solution in a bad string contains all information of $\langle \text{edge}(i,j,\{v_r,v_s\}) \rangle$. It is difficult to enforce that a match starts at a particular position but we will show that we are able to enforce that it contains a "111" substring which, by our construction, implies that the match contains all information of $\langle \text{edge}(i,j,\{v_r,v_s\}) \rangle$.

Then, given $E = \{e_1, \ldots, e_m\}$, we set

$$s'_{i,j} := \langle \text{edge\_block}(i,j,e_1) \rangle \circ 0 \circ \langle \text{edge\_block}(i,j,e_2) \rangle \circ \cdots \circ \langle \text{edge\_block}(i,j,e_m) \rangle.$$

Parameter values. We set $L := nk+5$ and generate $k_g := 1$ good string, $k_b := \binom{k}{2}$ bad strings, and we set distance parameters $d_g := k+3$ and $d_b := k-2$.

Example. Let $G = (V,E)$ with $V := \{v_1, v_2, v_3, v_4\}$ and $E := \{\{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_3, v_4\}\}$ as shown in Fig. 1(a) and let $k = 3$. Fig. 1(b) displays the good string s_g and the $\binom{k}{2} = 3$ bad strings $s'_{1,2}$, $s'_{1,3}$, and $s'_{2,3}$. Additionally, we show the length-$(nk+5)$, i.e., length-17, string s which is a match for $S_g = \{s_g\}$ and a match for $S_b = \{s'_{1,2}, s'_{1,3}, s'_{2,3}\}$ and, thus, corresponds to the k-clique in G.

3.2 Correctness of the Reduction

We show the two directions of the correctness proof for the above construction by two lemmas.

Lemma 1 *For a graph with a k-clique, the construction in Sect. 3.1 produces an instance of* DSSS *that has a solution, i.e., there is a length-L string s such that $d_H(s, s_g) \geq d_g$ and every $s'_{i,j} \in S_b$ has a length-L substring $t'_{i,j}$ with $d_H(s, t'_{i,j}) \leq d_b$.*

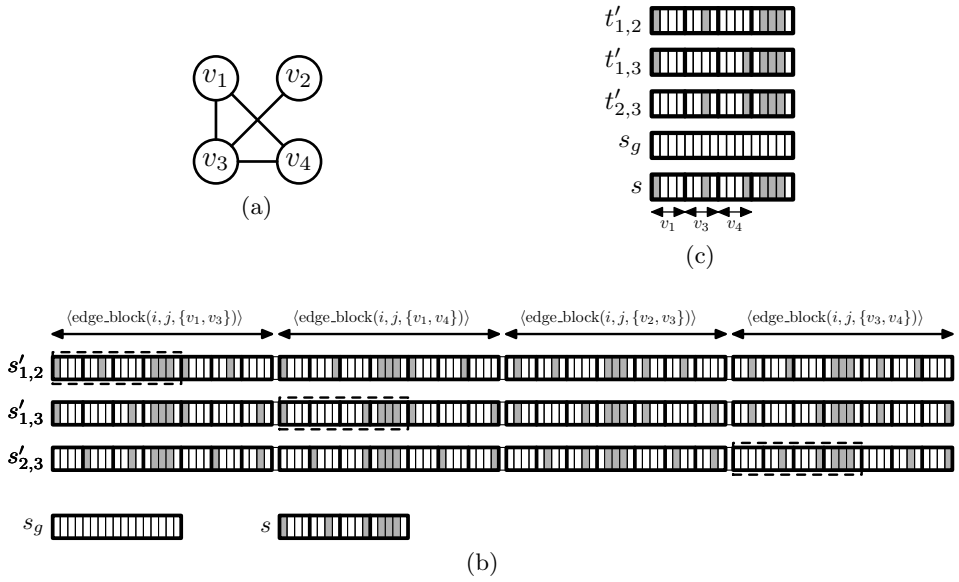

Fig. 1. Example for the reduction from a Clique instance to a DSSS instance with binary alphabet. (a) A Clique instance $G = (V, E)$ with $k = 3$. (b) The produced DSSS instance. We indicate the "1"s of the construction by grey boxes, the "0"s by white boxes. We display the solution s that is found since G has a clique of size $k = 3$; matches of s in $s'_{1,2}$, $s'_{1,3}$, and $s'_{2,3}$ are indicated by dashed boxes. By bold lines we indicate the substrings by which we constructed the bad strings: each $\langle \text{edge\_block}(\cdot,\cdot,e)\rangle$ substring is built from $\langle \text{edge}(\cdot,\cdot,e)\rangle$ for some $e \in E$, consisting of k length-n substrings, followed by "01110", followed again by $\langle \text{edge}(\cdot,\cdot,e)\rangle$. (c) Alignment of the matches $t'_{1,2}$, $t'_{1,3}$, and $t'_{2,3}$ (marked by dashed boxes in (b)) with s_g and s.

Proof. Let $h_1, h_2, \ldots, h_k$ denote the indices of the clique's vertices, $1 \le h_1 < h_2 < \cdots < h_k \le n$. Then, we can find a solution string

$$s := \langle \text{vertex}(v_{h_1})\rangle \circ \langle \text{vertex}(v_{h_2})\rangle \circ \cdots \circ \langle \text{vertex}(v_{h_k})\rangle \circ 01110.$$

For every $s'_{i,j}$, $1 \le i < j \le k$, the bad string $s'_{i,j}$ contains a substring $t'_{i,j}$ with $d_H(s, t'_{i,j}) \le d_b = k - 2$, namely

$$t'_{i,j} := \langle \text{edge}(i, j, \{v_{h_i}, v_{h_j}\})\rangle \circ 01110.$$

Moreover, we have $d_H(s, s_g) \ge d_g = k + 3$. □

Lemma 2 *A solution for the* DSSS *instance produced from a graph G by the construction in Sect. 3.1 corresponds to a k-clique in G.*

Proof. We prove this statement in several steps:
(1) We observe that a solution for the DSSS instance has at least $k + 3$ "1"s since $d_H(s, s_g) \ge d_g = k + 3$ and s_g consists only of "0"s.
(2) We observe that a solution for the DSSS instance has at most $k+3$ many "1"s: Following the construction, every length-L substring $t'_{i,j}$ of every bad string $s'_{i,j}$, $1 \le i < j \le k$, contains at most five "1"s and $d_H(s, t'_{i,j}) \le k - 2$.
(3) A match $t'_{i,j}$ for s in the bad string $s'_{i,j}$ contains exactly five "1"s: This follows from the observation that *any* length-L substring in a bad string contains *at most* five "1"s together with (1) and (2): Only if $t'_{i,j}$ contains five "1"s and all of them coincide with "1"s in s, we have $d_H(s, t'_{i,j}) \le (k + 3) - 5 = k - 2$.

(4) All $t'_{i,j}$, $1 \leq i < j \leq k$, and s must contain a "111" substring, located at the same position: To show this, let $t'_{i,j}$ be a match of s in a bad string $s'_{i,j}$ for some $1 \leq i < j \leq k$. From (3), we know that the match $t'_{i,j}$ must contain exactly five "1"s. Thus, since a substring of a bad string contains five "1"s only if it contains a "111" substring, $t'_{i,j}$ must also contain a "111" substring (which separates in $s'_{i,j}$ two substrings $\langle \text{edge}(i,j,e) \rangle$ for some $e \in E$). All "1"s in $t'_{i,j}$ have to coincide with "1"s chosen from the $k-3$ "1"s in s. In particular, the position of the "111" substring must be the same in the solution and in $t'_{i,j}$ for all $1 \leq i < j \leq k$. This ensures a "synchronization" of the matches.
(5) W.l.o.g., all $t'_{i,j}$, $1 \leq i < j \leq k$, and s all end with the "01110" substring: From (4), we know that all $t'_{i,j}$ contain a "111" substring at the same position. If they do not all end with "01110", we can shift them such that the contained "111" substring is shifted to the appropriate position, as we describe more precisely in the following. Recall that every length-L substring which contains the "111" substring of $\langle \text{edge\_block}(i,j,e) \rangle$ has the form $\langle \text{edge}(i,j,e) \rangle[h, nk] \circ 01110 \circ \langle \text{edge}(i,j,e) \rangle[1, h-1]$ for $1 \leq h \leq nk$ and $e \in E$. Since all $t'_{i,j}$, $1 \leq i < j \leq k$, contain the "111" substring at the same position, they all have this form for the same h. Then, we can, instead, consider $\langle \text{edge}(i,j,e) \rangle[1, nk] \circ 01110$ and, by a circular shift, move the "111" substring in the solution to the appropriate position. Considering the solution s and the matches $t'_{i,j}$ for all $1 \leq i < j \leq k$ as a character matrix, this is a reordering of columns and, thus, the pairwise Hamming distances do not change.
(6) We divide the first nk positions of the matches and the solution into k "sections", each of length n. In s, each of these sections has the form $\langle \text{vertex}(v) \rangle$ for a vertex $v \in V$ by the following argument: By (5), all matches in bad strings end with "01110" and, by the way we constructed the bad strings, each of their sections either consists only of "0"s or has the form $\langle \text{vertex}(v) \rangle$ for a vertex $v \in V$. If the section encodes a vertex, it contains one "1" which has to coincide with a "1" in s. For the ith section, $1 \leq i \leq k$, the matches in strings $s'_{i,j}$ for $i < j \leq k$ and in strings $s'_{j,i}$ for $1 \leq j < i$, encode a vertex in their ith section. Therefore, every of the k sections in s contains a "1" and, since s (by (1) and (2)) contains $k+3$ many "1"s and (by (4)) ends with "01110", each of its sections contains exactly one "1". Therefore, every section of s can be read as the encoding $\langle \text{vertex}(v) \rangle$ for a $v \in V$.
Conclusion. Following (6), let v_{h_i}, $1 \leq i \leq k$, be the vertex encoded in the ith length-n section of s. Now, consider some $1 \leq i < j \leq k$. Solution s has a match in $s'_{i,j}$ iff there is an $\langle \text{edge}(i,j,\{v_{h_i}, v_{h_j}\}) \rangle \circ 01110$ substring in $s'_{i,j}$ and this holds iff $\{v_{h_i}, v_{h_j}\} \in E$. Since this is true for all $1 \leq i < j \leq k$, all $v_{h_1}, v_{h_2}, \ldots, v_{h_k}$ are pairwisely connected by edges in G and, thus, form a k-clique. □

Lemmas 1 and 2 yield the following theorem.

Theorem 1 DSSS *with binary alphabet is* W[1]*-hard for every combination of the parameters* k_g, k_b, d_g, *and* d_b.[6] □

Theorem 1 means, in particular, that DSSS with binary alphabet is W[1]-hard with respect to every single parameter k_g, k_b, d_g, and d_b. Moreover, it allows us to exploit an important connection between parameterized complexity and the theory of approximation algorithms as follows.

Corollary 1 *There is no EPTAS for* DSSS *unless* W[1] = FPT.

Proof. Cesati and Trevisan [5] have shown that a problem with an EPTAS is fixed-parameter tractable with respect to the parameters that correspond to the objective

[6] Note that this is the strongest statement possible for these parameters because it means that the combinatorial explosion cannot be restricted to a function $f(k_g, k_b, d_g, d_b)$.

functions of the EPTAS. In Theorem 1, we have shown W[1]-hardness for DSSS with respect to d_g and d_b. Therefore, we conclude that DSSS cannot have an EPTAS for the objective functions d_g and d_b unless W[1] = FPT. □

4 Fixed-Parameter Tractability for a Special Case

In this section, we give a fixed-parameter algorithm for a *modified version* of DSSS. First of all, we restrict the problem to a binary alphabet $\Sigma = \{0, 1\}$. Then, the problem input consists, similar as in DSSS, of two sets S_g and S_b of binary strings, here with all strings in S_g being of length L. Increasing the number of good strings, we can easily transform an instance of DSSS into one in which all good strings have the same length L by replacing each string $s_i \in S_g$ by a set containing all length-L substrings of s_i. Therefore, in the same way as Deng *et al.* [6] we assume in the following that all strings in S_g have length L. We now consider, instead of the parameter d_g from the DSSS definition, the "dual parameter" $d'_g := L - d_g$ such that we require a solution string s with $d_H(s, s_i) \geq L - d'_g$ for all $s_i \in S_g$. The idea behind is that in some practical cases it might occur that, while d_g is rather large, d'_g is fairly small. Hence, restricting the combinatorial explosion to d'_g might sometimes be more natural than restricting it to d_g. Parameter d'_g is said to be *optimal* if there is an s with $d_H(s, s_i) \geq L - d'_g$ for all $s_i \in S_g$ and if there is no s' with $d_H(s', s_i) \geq L - d'_g + 1$ for all $s_i \in S_g$. The question addressed in this section is to find the minimum integer d_b such that, for the optimal parameter value d'_g, there is a length-L string s with $d_H(s, s_i) \geq L - d'_g$ for every $s_i \in S_g$ and such that every $s'_i \in S_b$ has a length-L substring t'_i with $d_H(s, t'_i) \leq d_b$. Naturally, we also want to compute the length-L solution string s corresponding to the found minimum d_b. We refer to this modified version of DSSS as MDSSS. We can read the set S_g of k_g length-L strings as a $k_g \times L$ character matrix. We call a column in this matrix *dirty* if it contains "0"s as well as "1"s.

In the following, we present an algorithm solving MDSSS. We conclude this section by pointing out the difficulties arising when giving up some of the restrictions concerning MDSSS.

4.1 Fixed-Parameter Algorithm

We present an algorithm that shows the fixed-parameter tractability of MDSSS with respect to the parameter d'_g. There are instances of MDSSS where d'_g is in fact smaller than the parameter d_g. In these cases, solving MDSSS could be a way to circumvent the combinatorial difficulty of computing exact solutions for DSSS; notably, DSSS is not fixed-parameter tractable with respect to d_g (Sect. 3) and we conjecture that it is not fixed-parameter tractable with respect to d'_g. The structure of the algorithm is as follows.

Preprocessing: Process all non-dirty columns of the input set S_g. If there are more than $d'_g \cdot k_g$ dirty columns then reject the input instance. Otherwise, proceed on the thereby reduced set S_g consisting only of dirty columns.

Phase 1: Determine all solutions s such that $d_H(s, s_i) \geq L - d'_g$ for every $s_i \in S_g$ for the optimal d'_g.

Phase 2: For every s found in Phase 1, determine the minimal value of d_b such that every $s'_i \in S_b$ has a length-L substring t'_i with $d_H(s, t'_i) \leq d_b$. Finally, find the minimum value of d_b over all examined choices of s.

Note that, in fact, Phase 1 and Phase 2 are interleaved. Phase 1 of our algorithm extends the ideas behind a bounded search tree algorithm for Closest String in [13]. There, however, the focus was on finding *one* solution whereas, here, we

require to find *all* solutions for the optimal parameter value. This extension was only mentioned in [13] and it will be described here.

Preprocessing. Reading the set S_g as a $k_g \times L$ character matrix, we set, for an all-"0" (all-"1") column in this matrix, the corresponding character in the solution to "1" ("0"); otherwise, we would not find a solution for an *optimal* d'_g. If the number of remaining dirty columns is larger than $d'_g \cdot k_g$ then we reject the input instance since no solution is possible.

Phase 1. The precondition of this phase is an optimal parameter d'_g. Since, in general, the optimal d'_g is not known in advance, it can be found by looping through $d'_g = 0, 1, 2, \ldots$, each time invoking the procedure described in the following until we meet the optimal d'_g. Notably, for each such d'_g value, we do not have to redo the preprocessing, but only compare the number of dirty columns against $d'_g \cdot k_g$.

Phase 1 is realized as a recursive procedure: We maintain a length-L candidate string s_c which is initialized as $s_c := \mathrm{inv}(s_1)$ for $s_1 \in S_g$, where $\mathrm{inv}(s_1)$ denotes the bitwise complement of s_1. We call a recursive procedure `Solve_MDSSS`, given in Fig. 2, working as follows.

If s_c is far away from all strings in S_g (i.e., $d_H(s_c, s_i) \geq L - d'_g$ for all $s_i \in S_g$) then s_c already is a solution for Phase 1. We invoke the second phase of the algorithm with the argument s_c. Since it is possible that s_c can be further transformed into another solution, we continue the traversal of the search tree: we select a string $s_i \in S_g$ such that s_c is not allowed to be closer to s_i (i.e., $d_H(s_c, s_i) = L - d'_g$); such an s_i must exist since parameter d'_g is optimal. We try all possible ways to move s_c away from s_i (such that $d_H(s_c, s_i) = L - (d'_g - 1)$), calling the recursive procedure `Solve_MDSSS` for each of the produced instances.

Otherwise, if s_c is not a solution for Phase 1, we select a string $s_i \in S_g$ such that s_c is too close to s_i (i.e., $d_H(s_c, s_i) < L - d'_g$) and try all possible ways to move s_c away from s_i, calling the recursive procedure for each of the produced instances.

The invocations of the recursive procedure can, thus, be described by a search tree. In the above recursive calls, we omit those calls trying to change a position in s_c which has already been changed before. Therefore, we also omit further invocations of the recursive procedure if the current node of the search tree is already at depth d'_g of the tree; otherwise, s_c would move too close to s_1 (i.e., $d_H(s_c, s_1) < L - d'_g$).

Phase 1 is given more precisely in Fig. 2. It is invoked by `Solve_MDSSS`$(\mathrm{inv}(s_1), d'_g)$.

Phase 2. The second phase deals with determining the minimal value of d_b such that there is a string s in the set of the solution strings found in the first phase with $d_H(s, t'_i) \leq d_b$ for $1 \leq i \leq k_b$, where t'_i is a length-L substring of s'_i.

For a given solution string s from the first phase and a string $s'_i \in S_b$, we use Abrahamson's algorithm [1] to find the minimum of the number of mismatches between s and every length-L substring of s'_i in $O(|s_i|\sqrt{L \log L})$ time. This minimum is equal to $\min_{t'_i} d_H(s, t'_i)$, where t'_i is length-L substring of s'_i. Applying this algorithm to all strings in S_b, we get the value of d_b for s, $\max_{i=1,\ldots,k_b} \min_{t'_i} d_H(s, t'_i)$. The minimum value of d_b is then the minimum distance of a solution string from Phase 1 to all bad strings, and s which achieves this minimum distance is the corresponding solution string.

If we are given a fixed d_b and are asked if there is a string s among the solution strings from the first phase which is a match to all strings in S_b, there is a more efficient algorithm by Amir *et al.* [3] for string matching with d_b-mismatches, which takes only $O(|s'_i|\sqrt{d_b \log d_b})$ time to find all length-L substrings in s'_i whose Hamming distance to s is at most d_b.

Recursive procedure `Solve_MDSSS`$(s_c, \Delta d)$**:**
Global variables: Sets S_g and S_b of strings, all strings in S_g of length L, and integer d'_g.
Input: Candidate string s_c and integer Δd, $0 \leq \Delta d \leq d'_g$.
Output: For optimal d'_g, *each* length-L string $\hat{s}$ with $d_H(\hat{s}, s_i) \geq L - d'_g$ and $d_H(\hat{s}, s_c) \leq \Delta d$.
Remark: The procedure calls, for each computed string $\hat{s}$, Phase 2 of the algorithm.

Method:
(0) if $(\Delta d < 0)$ **then return**;
(1) if $(d_H(s_c, s_i) \leq L - (d'_g + \Delta d))$ for some $i \in \{1, \ldots, k_g\}$ **then return**;
(2) if $(d_H(s_c, s_i) \geq L - d'_g)$ for all $i = 1, \ldots, k_g$ **then**
    /* s_c already is a solution for Phase 1 */
    call Phase_2(s_c, S_b);
    choose $i \in \{1, \ldots, k_g\}$ such that $d_H(s_c, s_i) = L - d'_g$;
    $P := \{\, p \mid s_c[p] = s_i[p] \,\}$;
    for all $p \in P$ **do**
        $s'_c := s_c$;
        $s'_c[p] := \mathrm{inv}(s_c[p])$;
        call `Solve_MDSSS`$(s'_c, \Delta d - 1)$;
    end for
else
    /* s_c is not a solution for Phase 1 */
    choose $i \in \{1, \ldots, k_g\}$ such that $d_H(s_c, s_i) < L - d'_g$;
    $Q := \{\, p \mid s_c[p] = s_i[p] \,\}$;
    choose any $Q' \subseteq Q$ with $|Q'| = d'_g + 1$;
    for all $q \in Q'$ **do**
        $s'_c := s_c$;
        $s'_c[q] := \mathrm{inv}(s_c[q])$;
        call `Solve_MDSSS`$(s'_c, \Delta d - 1)$;
    end for
end if
(3) return;

Fig. 2. Recursive procedure realizing Phase 1 of the algorithm for MDSSS.

4.2 Correctness of the Algorithm

Preprocessing. The correctness of the preprocessing follows in a similar way as the correctness of the "problem kernel" for Closest String observed by Evans *et al.* [9] (proof omitted).

Lemma 3 *Given an* MDSSS *instance with the set S_g of k_g good length-L strings, and a positive integer d'_g. If the resulting $k_g \times L$ matrix has more than $k_g \cdot d'_g$ dirty columns then there is no string s with $d_H(s, s_i) \geq L - d'_g$ for all $s_i \in S_g$.* □

Phase 1. From Step (2) in Fig. 2 it is obvious that every string s, which is output of Phase 1 and for which, then, Phase 2 is invoked, satisfies $d_H(s, s_i) \geq L - d'_g$ for all $s_i \in S_g$. The reverse direction, i.e., to show that Phase 1 finds *every* length-L string s with $d_H(s, s_i) \geq d'_g$ for all $s_i \in S_g$, is more involved; the proof is omitted:

Lemma 4 *Given an* MDSSS *instance, if s is an arbitrary length-L solution string, i.e., $d_H(s, s_i) \geq L - d'_g$ for all $s_i \in S_g$, then s can be found by calling procedure* `Solve_MDSSS`*.* □

Phase 2. The second phase is only an application of known algorithms.

4.3 Running Time of the Algorithm

Preprocessing. The preprocessing can easily be done in $O(L \cdot k_g)$ time. Even if the optimal d'_g is not known in advance, we can simply process the non-dirty columns and count the number L_d of dirty ones; therefore, the preprocessing has to be done only once. Then, while looping through $d'_g = 0, 1, 2, \ldots$ in order to find the optimal d'_g, we only have to check, for every value of d'_g in constant time, whether $L_d \leq d'_g \cdot k_g$.
Phase 1. The dependencies of the recursive calls of procedure `Solve_MDSSS` can be described as a search tree in which an instance of the procedure is the parent node of all its recursive calls. One call of procedure `Solve_MDSSS` invokes at most $d'_g + 1$ new recursive calls. More precisely, if s_c is a solution then it invokes at most d'_g calls and if s_c is not a solution then it invokes at most $d'_g + 1$ calls. Therefore, every node in the search tree has at most $d'_g + 1$ children. Moreover, Δd is initialized to d'_g and every recursive call decreases Δd by 1. As soon as $\Delta d = 0$, no new recursive calls are invoked. Therefore, the height of the search tree is at most d'_g. Hence, the search tree has a size of $O((d'_g + 1)^{d'_g}) = O((d'_g)^{d'_g})$.
Regarding the running time needed for one call of procedure `Solve_MDSSS`, note that, after the preprocessing, the instance consists of at most $d'_g \cdot k_g$ columns. Then, a central task in the procedure is to compute the Hamming distance of two strings. To this end, we initially build, in $O(d'_g \cdot k_g^2) = O(L \cdot k_g)$ time, a table containing the distances of s_c to all strings in S_g. Using this table, to determine whether or not s_c is a match for S_g or to find an s_i having at least d'_g positions coinciding with s_c can both be done in $O(k_g)$ time. To identify the positions in which s_c coincides with an $s_i \in S_g$ can be done in $O(d'_g \cdot k_g)$ time. After we change one position in s_c, we only have to inspect one column of the $k_g \times (d'_g \cdot k_g)$ matrix induced by S_g and, therefore, can update the table in $O(k_g)$ time. Summarizing, one call of procedure `Solve_MDSSS` can be done in $O(d'_g \cdot k_g)$ time.
Together with the $d'_g = 0, 1, 2, \ldots$ loop in order to find the optimal d'_g, Phase 1 can be done in $O((d'_g)^2 \cdot k_g \cdot (d'_g)^{d'_g})$ time.
Phase 2. For every solution string found in Phase 1, the running time of the second phase is $O(N\sqrt{L \log L})$, where N denotes the sum of the length of all strings in S_b [1]. We obtain the following theorem:

Theorem 2 *MDSSS can be solved in* $O(L \cdot k_g + ((d'_g)^2 k_g + N\sqrt{L \log L}) \cdot (d'_g)^{d'_g})$ *time where* $N = \sum_{s'_i \in S_b} |s'_i|$ *is the total size of the bad strings.* □

4.4 Extensions of MDSSS

The special requirements imposed on the input of MDSSS seem inevitable in order to obtain the above fixed-parameter tractability result. We discuss the problems arising when relaxing the constraints on the alphabet size and the value of d'_g.
Non-binary alphabet. Already extending the alphabet size in the formulation of MDSSS from two to three makes our approach, described in Sect. 4.1, combinatorially much more difficult such that it does not yield fixed-parameter tractability any more. A reason lies in the preprocessing. When having an all-equal column in the character matrix induced by S_g, for a three-letter alphabet there are two instead of one possible choices for the corresponding position in the solution string. Therefore, to enumerate all solutions s with $d_H(s, s_i) \geq L - d'_g$ for all $s_i \in S_g$, which is essential for our approach, is not fixed-parameter tractable any more; the number of solutions is too large. Let $L' \leq L$ be the number of non-dirty columns and let the alphabet size be three. Then, aside from the dirty columns, we already have $2^{L'}$ assignments of characters to the positions corresponding to non-dirty columns.
Non-optimal d'_g parameter. Also for non-optimal d'_g parameter, the number of solutions s with $d_H(s, s_i) \geq L - d'_g$ for all $s_i \in S_g$ can become too large and it appears

to be fixed-parameter intractable with respect to d'_g to enumerate them all. Consider the example where $S_g = \{0^L\}$. Then, there are more than $\binom{L}{d'_g}$ strings s with $d_H(s, 0^L) \geq L - d'_g$. (If the value of d'_g is only a fixed number larger than the optimal one, it could, nevertheless, be possible to enumerate all solution strings of Phase 1.)

5 Conclusion

We have shown that Distinguishing Substring Selection, which has a PTAS, cannot have an EPTAS unless FPT = W[1]. It remains open whether this also holds for the tightly related and similarly important computational biology problems Closest Substring and Consensus Patterns, each of which has a PTAS [15, 16] and for each of which it is unknown whether an EPTAS exists. It has been shown that, even for constant size alphabet, Closest Substring and Consensus Patterns are W[1]-hard with respect to the number of input strings [11]; the parameterized complexity with respect to the distance parameter, however, is open for these problems, whereas it has been settled for DSSS in this paper. It would be interesting to further explore the border between fixed-parameter tractability and intractability as initiated in Sect. 4.

References

1. K. Abrahamson. Generalized string matching. *SIAM Journal on Computing*, 16(6):1039–1051, 1987.
2. J. Alber, J. Gramm, and R. Niedermeier. Faster exact solutions for hard problems: a parameterized point of view. *Discrete Mathematics*, 229(1-3):3–27, 2001.
3. A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. In *Proc. of 11th ACM-SIAM SODA*, pages 794–803, 2000.
4. G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation—Combinatorial Optimization Problems and their Approximability Properties.* Springer, 1999.
5. M. Cesati and L. Trevisan. On the efficiency of polynomial time approximation schemes. *Information Processing Letters*, 64(4):165–171, 1997.
6. X. Deng, G. Li, Z. Li, B. Ma, and L. Wang. A PTAS for Distinguishing (Sub)string Selection. In *Proc. of 29th ICALP*, number 2380 in LNCS, pages 740–751, 2002. Springer.
7. R. G. Downey. Parameterized complexity for the skeptic (invited paper). In *Proc. of 18th IEEE Conference on Computational Complexity*, July 2003.
8. R. G. Downey and M. R. Fellows. *Parameterized Complexity.* Springer, 1999.
9. P. A. Evans, A. Smith, and H. T. Wareham. The parameterized complexity of p-center approximate substring problems. Technical report TR01-149, Faculty of Computer Science, University of New Brunswick, Canada. 2001.
10. M. R. Fellows. Parameterized complexity: the main ideas and connections to practical computing. In *Experimental Algorithmics*, number 2547 in LNCS, pages 51–77, 2002. Springer.
11. M. R. Fellows, J. Gramm, and R. Niedermeier. On the parameterized intractability of Closest Substring and related problems. In *Proc. of 19th STACS*, number 2285 in LNCS, pages 262–273, 2002. Springer.
12. M. Frances and A. Litman. On covering problems of codes. *Theory of Computing Systems*, 30:113–119, 1997.
13. J. Gramm, R. Niedermeier, and P. Rossmanith. Exact solutions for Closest String and related problems. In *Proc. of 12th ISAAC*, number 2223 in LNCS, pages 441–453, 2001. Springer. Full version to appear in *Algorithmica*.
14. J. K. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang. Distinguishing string selection problems. In *Proc. of 10th ACM-SIAM SODA*, pages 633–642, 1999.
15. M. Li, B. Ma, and L. Wang. On the Closest String and Substring Problems. *Journal of the ACM*, 49(2):157–171, 2002.
16. M. Li, B. Ma, and L. Wang. Finding similar regions in many sequences, *Journal of Computer and System Sciences*, 65(1):73–96, 2002.