Fixed-Parameter Algorithms for Closest String and Related Problems[*]

Jens Gramm[†] Rolf Niedermeier
Wilhelm-Schickard-Institut für Informatik, Universität Tübingen,
Sand 13, D-72076 Tübingen, Fed. Rep. of Germany
gramm,niedermr@informatik.uni-tuebingen.de

Peter Rossmanith
Institut für Informatik, Technische Universität München,
Arcisstr. 21, D-80290 München, Fed. Rep. of Germany
rossmani@in.tum.de

Abstract

Closest String is one of the core problems in the field of consensus word analysis with particular importance for computational biology. Given k strings of same length and a nonnegative integer d, find a "closest string" s such that none of the given strings has Hamming distance greater than d from s. Closest String is NP-complete. In biological applications, however, d usually is very small. We show how to solve Closest String in linear time for constant d—the exponential growth in d is bounded by $O(d^d)$. We extend this result to the closely related problems d-Mismatch and Distinguishing String Selection. Moreover, we also show that Closest String is solvable in linear time when k is constant and d is arbitrary. In summary, this means that Closest String is fixed-parameter tractable with respect to parameter d and with respect to parameter k. Finally, the practical usefulness of our findings is substantiated by some experimental results.

[*]An extended abstract of this paper was presented at the 12th International Symposium on Algorithms and Computation (ISAAC 2001), Springer-Verlag, LNCS 2223, pages 441–453, held in Christchurch, New Zealand, December 19–21, 2001.

[†]Supported by the Deutsche Forschungsgemeinschaft (DFG), project OPAL (optimal solutions for hard problems in computational biology), NI 369/2-1.

1 Introduction

Finding signals in DNA is a major problem in computational biology. A recently intensively studied facet of this problem is based on consensus word analysis [21, Section 8.6]. A central problem herein is the so-called CLOSEST STRING (or, equivalently, CONSENSUS STRING or CENTER STRING) problem:

> **Input:** k strings $s_1, s_2, \ldots, s_k$ over alphabet Σ of length L each, and a nonnegative integer d.
> **Question:** Is there a string s such that $d_H(s, s_i) \leq d$ for all $i = 1, \ldots, k$?[1]

Here, $d_H(s, s_i)$ denotes the Hamming distance between strings s and s_i. Related problems we also study here are the d-MISMATCH problem (which generalizes CLOSEST STRING in the way that we look for center strings of *aligned* substrings of a given set of strings) [25, 26] and the DISTINGUISHING STRING SELECTION problem [15] (for a brief overview on biological applications concerning signal finding and primer design, e.g., refer to [15]). All these problems are, in general, NP-hard [10, 15][2]. Hence, polynomial time algorithms are out of reach.

Despite their hardness, these problems need to be solved in practice. One line of research is thus to study their approximability. Improving previous work [11, 15], Li *et al.* [17] finally came up with a polynomial time approximation scheme (PTAS) for CLOSEST STRING. The constants and polynomials occurring in the running time, however, make this result of little practical value. Another very promising approach is to study the parameterized complexity [1, 5, 8] of these problems. Consider the two most natural parameters of CLOSEST STRING: the maximum Hamming distance d allowed and the number k of given input strings. Under the natural assumption that either d or k is (very) small (in particular, in biological applications it is appropriate to assume small d, e.g., d smaller than 10 [6, 15]), it is important to ask whether efficient polynomial or even better linear time algorithms are possible when d or k are constants. Put in other (parameterized complexity) terms, this is the question of the *fixed-parameter tractability* of these problems.

We present the following results. CLOSEST STRING can be solved in time $O(kL + kd \cdot d^d)$, yielding a linear time search tree algorithm for constant d. This answers the open question of Evans and Wareham [6] for the parameterized complexity of CLOSEST STRING with parameter d. Furthermore, we can

[1] From a linguistic point of view, a "closest" string would only mean a string with minimum Hamming distance to the given strings. Beyond that, we use the term "closest" string here for a string which has Hamming distance at most d to all given strings.

[2] Frances and Litman [10] show the NP-completeness of CLOSEST STRING, considering it from the viewpoint of coding theory where the problem is known as MINIMUM RADIUS problem.

generalize our result to d-MISMATCH, improving work and positively answering an open question of Stojanovic *et al.* [25] where a linear time algorithm for only $d = 1$ was given. Also, our result is extendible to DISTINGUISHING STRING SELECTION, for which we can derive a linear time algorithm in case of constant distance parameters and constant alphabet size. To some extent, our results are related to recent results for the problem of identifying motifs in orthologous sequences from multiple species [3], where exact solutions were also obtained by developing algorithms whose running time is exponential with respect to a parameter (in that case, substring length L). In particular, our result supports the intuition that CLOSEST STRING is easier than the more general CLOSEST SUBSTRING, in which we search a closest string of length L that has maximum Hamming distance d only to a substring of every input string: On the one hand, both problems are NP-complete and both allow a PTAS [17]. On the other hand, we show that CLOSEST STRING is fixed-parameter tractable with respect to k, i.e., the number of input strings (answering an open question of [6]), while it was shown that CLOSEST SUBSTRING is W[1]-hard with respect to k [9]; informally speaking, W[1]-hardness means that a fixed-parameter algorithm is highly unlikely [5].
Finally, we indicate the practical usefulness and potential of our algorithms by some experimental results.

2 Preliminaries

In this section, we give definitions and small results that will be useful later on. For a string s of length L, we use $s[p]$, $1 \le p \le L$, to denote the character at position p in s. Then, $d_H(s_i, s_j)$ denotes the Hamming distance between strings s_i and s_j of same length L, i.e., $|\{\, p : s_i[p] \neq s_j[p] \,\}|$. Given a set of strings $S = \{s_1, s_2, \ldots, s_k\}$, each of length L, then a string s is an *optimal closest string* for S iff there is no string s' with $\max_{i=1,\ldots,k} d_H(s', s_i) < \max_{i=1,\ldots,k} d_H(s, s_i)$. By way of contrast, s is an *optimal median string* for S iff there is no string s' with $\sum_{i=1,\ldots,k} d_H(s', s_i) < \sum_{i=1,\ldots,k} d_H(s, s_i)$.
Note that, given a set of strings $S = \{s_1, s_2, \ldots, s_k\}$, an optimal median string can be easily computed by choosing in every column a letter occurring most often. If a letter is chosen in this way, we call it *majority vote*; it is, however, not necessarily unique.
Given a set of k strings of length L, we can think of these strings as a $k \times L$ character matrix. By *columns* of a CLOSEST STRING instance, we refer to the columns of this matrix. In the following, we state that after reordering the columns of the CLOSEST STRING instance, we can easily obtain solutions for the

original instance from solutions for the reordered instance. For reordering the columns, we introduce a permutation on strings as follows. Given a string $s = c_1c_2\ldots c_L$ of length L with $c_1,\ldots,c_L \in \Sigma$ for alphabet Σ and a permutation $\pi : \{1,\ldots,L\} \to \{1,\ldots,L\}$. Then, $\pi(s) = c_{\pi(1)}c_{\pi(2)}\ldots c_{\pi(L)}$. The following lemma is obvious.

Lemma 1. *Given a set of strings $S = \{s_1, s_2, \ldots, s_k\}$, each of length L, and a permutation $\pi : \{1,\ldots,L\} \to \{1,\ldots,L\}$. Then s is an optimal closest string for $\{s_1, s_2, \ldots, s_k\}$ iff $\pi(s)$ is an optimal closest string for $\{\pi(s_1), \pi(s_2), \ldots, \pi(s_k)\}$.* □

Several columns can be identified due to *isomorphism*. The reason for this is the fact that the columns are independent from each other in the sense that the distance from the closest string is measured columnwise. For instance, consider the case of the two columns $(a,a,b)^t$ and $(b,b,a)^t$ when $k = 3$. Clearly, these two columns are isomorphic because they express the same structure, except that the symbols play different roles. For finding the optimal closest string, however, only the structure matters. Isomorphic columns form *column types*.
This can be generalized as follows. W.l.o.g., let a always denote the letter that occurs most often in a column, let b always denote the letter that has the secondly most often occurrences and so on. This property of being *normalized*, as we will refer to it in the following, can be easily achieved by a simple linear time preprocessing of the input instance. In addition, solving the normalized problem optimally, one again can compute the optimal solution of the original problem instance by simply reversing the above mapping done by the preprocessing. Hence:

Lemma 2. *To compute an optimal closest string, it is sufficient to solve a normalized and reordered instance. From this, the solution of the original instance can be derived in linear time.* □

In the following, we call two input instances *isomorphic* if there is a one-to-one correspondence between the columns of both instances such that each thus determined pair of columns is isomorphic. The following lemma shows that it is sufficient to solve an instance with alphabet size $|\Sigma'| \leq k$.

Lemma 3. *A* Closest String *instance with arbitrary alphabet Σ, $|\Sigma| > k$, is isomorphic to a* Closest String *instance with alphabet Σ', $|\Sigma'| = k$.*

Proof. Assume that there is an input instance with $|\Sigma| > k$. Clearly, in each column appear at most k different symbols from Σ. Since columns are independent from each other, to solve the underlying Closest String problem it suffices to represent the "logical structure" of a column by an isomorphic input instance. To do this, at most k symbols per column are enough. □

With the following observation by Evans and Wareham [6], we find that it is sufficient to solve instances containing at most kd columns. This yields a so-called "reduction to problem kernel" [5], which allows an efficient preprocessing of the input instance. We call a column *dirty* iff it contains at least two different symbols from alphabet Σ. Clearly, "all the work" in solving Closest String concentrates on the dirty columns of the input instance.

Lemma 4. *Given a* Closest String *instance with k strings $s_1, \ldots, s_k$ of length L and a positive integer d. If the resulting $k \times L$ matrix has more than kd dirty columns then there is no string s with* $\max_{i=1,\ldots,k} d_H(s, s_i) \leq d$.

Proof. Each of the k input strings differs from a solution in at most d positions. These mismatches give rise to dirty columns. Since there are k strings with at most d mismatches each, an instance can have a solution only when there are at most kd dirty columns. □

3 A Linear Time Solution for Constant d

We show that Closest String, although NP-complete in general, is solvable in linear time for constant d, which means that it is fixed-parameter tractable with respect to distance parameter d. In the first subsection, we give the basic algorithm and then we show that it has a linear running time for constant d. In the subsequent two subsections, we present heuristic improvements of the strategy and enhancements, e.g., solving the more general d-Mismatch and the Distinguishing String Selection problems.

3.1 Bounded Search Tree Algorithm

In Fig. 1, we outline a recursive procedure solving Closest String. It is based on the bounded search tree paradigm that frequently is successfully applied in the development of fixed-parameter algorithms [1, 5, 8]. For the correctness of the algorithm we need the following simple observation.

Lemma 5. *Given a set of strings $S = \{s_1, s_2, \ldots, s_k\}$ and a positive integer d. If there are $i, j \in \{1, \ldots, k\}$ with $d_H(s_i, s_j) > 2d$ then there is no string s with* $\max_{i=1,\ldots,k} d_H(s, s_i) \leq d$.

Proof. The Hamming distance satisfies the triangle inequality. If $d_H(s_i, s_j) > 2d$ and we are given an arbitrary string s then we, therefore, know that $d_H(s, s_i) + d_H(s, s_j) > 2d$. It follows that $d_H(s, s_i) > d$ or $d_H(s, s_j) > d$ (or both). □

Algorithm D, recursive procedure $CSd(s, \Delta d)$
Global variables: Set of strings $S = \{s_1, s_2, \ldots, s_k\}$, integer d.
Input: Candidate string s and integer Δd.
Output: A string $\hat{s}$ with $\max_{i=1,\ldots,k} d_H(\hat{s}, s_i) \leq d$ and $d_H(\hat{s}, s) \leq \Delta d$ if it exists, and "not found," otherwise.

(D0) If $(\Delta d < 0)$, then return "not found";

(D1) If $(d_H(s, s_i) > d + \Delta d)$ for some $i \in \{1, \ldots, k\}$ then return "not found";

(D2) If $(d_H(s, s_i) \leq d)$ for all $i = 1, \ldots, k$ then return s;

(D3) Choose some $i \in \{1, \ldots, k\}$ such that $d_H(s, s_i) > d$:
 $P := \{ p \mid s[p] \neq s_i[p] \}$;
 Choose any $P' \subseteq P$ with $|P'| = d + 1$;
 For all $p \in P'$ do
  $s' := s$;
  $s'[p] := s_i[p]$;
  $s_{ret} := CSd(s', \Delta d - 1)$;
  If $s_{ret} \neq$"not found" then return s_{ret};

(D4) Return "not found"

Figure 1: **Algorithm D**. Inputs are a Closest String instance consisting of a set of strings $S = \{s_1, s_2, \ldots, s_k\}$, and an integer d. First, we apply the reduction to a problem kernel as shown in Lemma 4—we select the dirty columns. If there are more than kd many, then we reject the instance. If there are at most kd many, then we invoke the recursion with $CSd(s_1, d)$.

The idea of our strategy is to start with one of the given strings, e.g., s_1, as a "candidate string." If there is a string s_i, $i = 2, \ldots, k$, that differs from the candidate string in more than d positions, we recursively try several ways to move the candidate string "towards" s_i; moving closer here means that we select a position in which the candidate string and s_i differ and set this position in the candidate string to the character of s_i at this position. We stop either if we moved the candidate "too far away" from s_1 or if we found a solution. By a careful selection of subcases of this recursion we can limit the size of this search tree to $O(d^d)$, as will be shown in the following theorem.

Theorem 1. *Given a set of strings $S = \{s_1, s_2, \ldots, s_k\}$, and an integer d, Algorithm D (Fig. 1) determines in time $O(kL + kd \cdot d^d)$ whether there is a string s with $\max_{i=1,\ldots,k} d_H(s, s_i) \leq d$ and computes such an s if one exists.*

Proof. **Running time.** Prior to the recursion, we perform the reduction to problem kernel as described in Lemma 4. This preprocessing, reducing the size of the input instance to $O(kd)$, can be done in time $O(kL)$.

Now, we consider the recursive part of the algorithm. Parameter Δd is initialized to d. Every recursive call decreases Δd by one. The algorithm stops when $\Delta d < 0$. Therefore, the algorithm builds a search tree of height at most d. In one step of the recursion, the algorithm chooses, given the current candidate string s, a string s_i such that $d_H(s, s_i) > d$. It creates a subcase for $d+1$ of the positions in which s and s_i disagree (there are more than d but at most $2d$ such positions). This yields an upper of $(d+1)^d$ bound on the search tree size.[3]

Each step of the recursion needs only linear time $O(kd)$: Before starting the recursion, we build a table containing the distances of the candidate s_1 to all other given strings. Using this table, instructions (D1) and (D2) can be done in time $O(k)$. In instruction (D3), we need time $O(k)$ to select the s_i for branching and time $O(kd)$ (observe Lemma 4) to find the positions in which s and s_i differ. For $d+1$ of those differing positions (there are at most $2d$ many) we modify the candidate string, update the table of distances, and call the procedure recursively. Since we changed only one position, we can update the table of distances in time $O(k)$.

Correctness. We have to show that Algorithm D will find a string s with $\max_{i=1,\ldots,k} d_H(s, s_i) \leq d$, if such a s exists. The preprocessing is correct by Lemma 4. Regarding the search tree, we explicitly show only the correctness of the first recursive step; the correctness of the algorithm then follows with an inductive application of the argument.

In the situation that s_1 satisfies $\max_{i=1,\ldots,k} d_H(s_1, s_i) \leq d$, we immediately find a solution, namely s_1. If s_1 is not a solution but there exists a closest string s for this instance with distance value d, then there is a string s_i, $i = 2, \ldots, k$, such that $d_H(s_1, s_i) > d$. For branching, we consider the positions where s_1 and s_i differ, i.e., $P := \{\, p : s_1[p] \neq s_i[p] \,\}$. Algorithm D successively creates subcases for $d+1$ positions p from P in order to create a new candidate by altering the respective position p from $s_1[p]$ to $s_i[p]$. Such a "move" is correct if we choose a position p from $P_1 := \{\, p : s_1[p] \neq s[p] = s_i[p] \,\}$. Now, we show that (at least) one of our $d+1$ moves is a correct one. We observe that $P = P_1 \cup P_2$ for $P_2 := \{\, p : s[p] \neq s_i[p] \,\}$. Since $d_H(s, s_i) \leq d$ we know that $|P_2| \leq d$. Therefore, at least one of our $d+1$ subcases will try a position from P_1.

Regarding instruction (D1), we can analogously to Lemma 5 observe that it is correct to omit those branches where the candidate string s satisfies $d_H(s, s_i) >$

[3] If there are two strings s_i, s_j with $d_H(s_i, s_j) = 2d$, we can use a better strategy: We know that a solution has to differ from both s_i and s_j in d positions. We can search a solution by trying all ways to partition the set of positions p with $s_i[p] \neq s_j[p]$ into two sets of size d. In the candidate, we give to one set of positions the characters of s_i, to the second set the characters of s_j. If we always had such a situation then the resulting running time would be $O(kL + kd \cdot 2^{2d})$.

$d + \Delta d$ for some string s_i of the given strings $s_1, \ldots, s_k$. $\square$

With Algorithm D, we find a solution if one exists. We find *all* solutions if the given distance parameter d is optimal. We do not necessarily find all solutions to a given instance when d is not optimal. Using binary search, however, we clearly can find the optimal distance value upperbounded by d at the cost of a time factor $O(\log d)$.

Note that the interleaving technique described in [19] to build a new problem kernel on every level of the search tree seems not to be of use here, because changing levels of recursion do not influence the size of the problem kernel.

3.2 Heuristic Improvements

Avoiding the multiple traversal of subtrees. Since the search tree size is the critical factor in the algorithm's running time, the goal is to keep it as small as possible. In particular, there is no use in visiting the same "configurations" twice or more times. In the following, we describe two ways to avoid visiting such useless branches of the search tree.

Keeping in mind the initial candidate string, there is no use in changing a position that has already been changed before. We can avoid this by testing for each position considered for a change whether it still equals the position in the initial candidate. We branch only if it does.

Still, we will run into situations where the same candidate string is considered multiple number of times. For instance, when the solution differs from the candidate in positions p_1 and p_2, we find it by changing first p_1 and then p_2, and we can also find it by changing first p_2 and then p_1. We can avoid this by storing all characters that have been tried at a position on a higher level of recursion (there are at most k many for every position). For branching, we consider only changing a position in the candidate if we didn't already try this character at this position at a higher level of recursion. This test can be done in the time bounds of Algorithm D.

Selection of the candidate string to start with. In Algorithm D, we initialized the candidate string s to s_1. We are, however, free to choose the initial candidate string from all s_i, $i = 1, \ldots, k$. Since we search the solution in the neighborhood of the initial candidate string s, a good choice is an s which is presumably close to the (unknown) solutions. A possible strategy is to select the string with a minimum median distance to all other strings.

3.3 Enhancements and Related Problems

Finding all solutions. So far, we assumed that Algorithm D is invoked with a minimal distance parameter d or the minimal d is found, e.g. using binary search. In some situations, it may, however, be desirable for the user to know all solutions that can be found with a non-minimal d. We already mentioned in Subsection 3.1 that Algorithm D, invoked with non-optimal d, does not necessarily compute *all* solutions for a given instance. This is not possible in the claimed time bound since there may be more than $O(kd \cdot d^d)$ solutions, e.g., for an instance containing only one input string of length L and allowing $d = 1$ mismatches. We can, however, extend the given algorithm such that, even for non-minimal d, it expands the set of found solutions to the set of all solutions. If we find a solution after changing d' positions in the initial candidate string, we are allowed to change $\Delta d = d - d'$ further positions. We can try all ways to change Δd of those positions in which the solution still equals the initial candidate string. A recursive strategy similar as the one in the main algorithm allows to further prune this search space, e.g., we only have to consider changes that do not increase the distance to one of the input strings to a value larger than d. The upper bound for the number of branching subcases is L, yielding a worst-case bound of $O(L^{\Delta d}|\Sigma^{\Delta d}|)$ for the expansion of a solution. In practice, we will often find the solutions with a small value of Δd. Therefore, for finding *all* solutions, our strategy of "expanding" the solutions of Algorithm D will still be superior to a brute-force check of all strings of length L.

Solving the d-Mismatch problem. Let $s_{i,p,L}$ denote the length-L substring of a given string s_i starting at position p, i.e., $s_{i,p,L} = s_i[p]s_i[p+1] \ldots s_i[p+L-1]$. Then, given strings $s_1, s_2, \ldots, s_k$ of length n and nonnegative integers k and L, the d-Mismatch problem is the question of whether there is a string s of length L and a position p with $1 \leq p \leq n - L + 1$, such that $d_H(s, s_{i,p,L}) \leq d$ for all $i = 1, \ldots, k$. For $n = L$, this problem is equivalent to Closest String. Stojanovic *et al.* [25, 26] give a linear-time algorithm for the 1-mismatch problem and report about its use in practice. For constant $d > 1$, they explicitly ask for a polynomial time algorithm.

We can solve d-Mismatch using Algorithm D in linear time for constant d. A trivial solution is to invoke the algorithm for each of the $n - L + 1$ possible positions in the $n \times k$ matrix which is induced by the input. This yields a worst-case running time of $O(n \cdot (kL + kd \cdot d^d)$, i.e., a quadratic[4] running time for constant d. Improving this, we achieve a linear running time for constant d

[4]Note that the input size is nk and that L is in $O(n)$.

as follows. We use the problem kernel of size kd for Closest String as given in Lemma 4. Considering only the first L columns of the $n \times k$ matrix, we can, in time $O(kL)$, build a FIFO queue of dirty columns. We update this queue in time $O(k)$ when shifting the window of L consecutive columns under consideration from position p (containing columns p to $p+L-1$) to position $p+1$ (containing columns $p+1$ to $p+L$), $p = 1, \ldots, n-L$. (1) If column p is dirty, we delete it from the front end of the queue. (2) If the "new" column $p+L$ is dirty, we append it to the back end of the queue.

Thus, we can maintain the queue of dirty columns, at each position taking only time $O(k)$. After a one-position-shift in the $n \times k$ matrix, Algorithm D is invoked on the columns in the queue only if the queue contains at most kd columns. The described strategy yields the following result.

Theorem 2. *d-Mismatch is solvable in time $O(kL + (n-L)kd \cdot d^d)$ which is $O(n \cdot k)$ for constant d.* □

Solving the Distinguishing String Selection (DSS) **problem.** In this problem, we are given "good" strings $s_1, \ldots, s_{k_1}$, "bad" strings $s'_1, \ldots, s'_{k_2}$, and nonnegative integers d_1, d_2. We are looking for a solution string s that is "close" to the good strings, i.e., $\max_{i=1,\ldots,k_1} d_H(s, s_i) \leq d_1$, and "far away" from the bad strings, i.e.,

$$\min_{j=1,\ldots,k_2} d_H(s, s'_j) \geq L - d_2.$$

Lanctot *et al.* [15] give a polynomial time factor 2 approximation algorithm for this problem. We can adapt Algorithm D to solve DSS.

We start with an observation similar to the one made in Lemma 5.

Lemma 6. *Given two sets of strings $S_1 = \{s_1, \ldots, s_{k_1}\}$ and $S_2 = \{s'_1, \ldots, s'_{k_2}\}$ and positive integers d_1 and d_2. If there are $i \in \{1, \ldots, k_1\}$ and $j \in \{1, \ldots, k_2\}$ with $d_H(s_i, s'_j) < L - (d_1 + d_2)$, then there is no string s satisfying both $\max_{i=1,\ldots,k_1} d_H(s, s_i) \leq d_1$ and $\min_{j=1,\ldots,k_2} d_H(s, s'_j) \geq L - d_2$.*

Proof. Assume that there are strings s, s_i, s'_j such that (1) s is "close" to s_i, i.e., $d_H(s, s_i) \leq d_1$, (2) s is "far away" from s'_j, i.e., $d_H(s, s'_j) \geq L - d_2$, and (3) s_i and s'_j satisfy the lemma's premise, i.e., $d_H(s_i, s'_j) < L - d_1 - d_2$. Since the Hamming distance satisfies the triangle inequality, we know that $d_H(s, s'_j) \leq d_H(s, s_i) + d_H(s_i, s'_j)$. Together with assumptions (1) and (3) this yields $d_H(s_i, s'_j) < L - d_2$. This, however, contradicts assumption (2) and shows that there are no s, s_i, s'_j that meet all three assumptions at the same time. □

In what follows, we describe how to modify Algorithm D in order to solve DSS. Using Lemma 6, we can detect instances that cannot have a solution,

i.e., instances where a good and a bad string have Hamming distance less than $L-(d_1+d_2)$. For this reason, we can extend instruction (D1) in Algorithm D by returning not only when $d_H(s,s_i) > d_1+\Delta d_1$ for the candidate s and a good string s_i but also when $d_H(s,s'_j) < L-(d_2+\Delta d_1)$ for a bad string s'_j.
Of course, a solution in instruction (D2) is now found, when the new goal is met, i.e., $\max_{i=1,\ldots,k_1} d_H(s,s_i) \leq d_1$ *and* $\min_{j=1,\ldots,k_2} d_H(s,s'_j) \geq L-d_2$.
Also instruction (D3) has to be extended. As long as the branching shown in (D3) applies, we still use it: If there is a good string s_i which our candidate s is too far away from, i.e., $d_H(s,s_i) > d_1$, we branch on d_1+1 many positions in which s and s_i differ.
When the candidate s satisfies $d_H(s,s_i) \leq d_1$ for all $i=1,\ldots,k_1$ but is too close to one of the bad strings s'_j, i.e., $d_H(s,s'_j) < L-d_2$, we introduce a new branching. We have to increase $d_H(s,s'_j)$ by changing in s a position p with $s[p] = s'_j[p]$. Since a solution $s^\star$ can have at most d_2 many positions p with $s^\star[p] = s_j[p]$ it is sufficient to branch on d_2+1 positions with $s[p] = s'_j[p]$. We do, however, not know to which character $s[p]$ should be set. Trying all characters in this situation gives us an upper bound of $(d_2+1)\cdot|\Sigma-1|$ for the subcases to branch into. This yields the following result.

Theorem 3. DSS *is solvable in time* $O\big((k_1+k_2)L\cdot(\max(d_1+1,(d_2+1)|\Sigma|))^{d_1}\big)$. □

4 A Linear Time Solution for Constant k

We make use of *integer linear programming* [18, 24]. There is a famous result of Lenstra [16][5] that applies to fixed-parameter algorithms (also see [13, 14] for more details). Lenstra's result basically says that integer linear programs (ILPs for short) with a constant number of variables can be solved in linear time. More precisely, with Kannan's [13] improvements we have the following.

Theorem 4 (Lenstra). *The integer programming feasibility problem can be solved with $O(p^{9p/2}L)$ arithmetic operations in integers of $O(p^{2p}L)$ bits in size, where p is the number of ILP variables and L is the number of bits in the input.* □

Note that the fixed-parameter result also needs space exponential in the parameter p.

[5] It won the Fulkerson Prize 1985 as an outstanding paper in the area of discrete mathematics.

The goal is to give an ILP formulation for Closest String such that the number of variables solely depends on the parameter value k, the number of input strings. The key to this lies in the notion of column types.

Example 1. *For $k = 3$, the set of all possible column types for a* Closest String *instance consists of*

$$(a, a, a)^t, (a, a, b)^t, (a, b, a)^t, (b, a, a)^t, (a, b, c)^t.$$

$\square$

Generally, the number of column types for k strings depends only on k (namely, it is given by the Bell number $B(k) \leq k!$, cf. [20]). Using the column types, Closest String can be formulated as an ILP having only $B(k) \cdot k$ variables. Let the underlying alphabet be Σ. The ILP can be formulated as follows. Since we may assume that the instance is normalized (Lemma 2), every column contains at least one occurrence of symbol a. The ILP uses $B(k) \cdot k$ variables $x_{t,\varphi}$, where t denotes a column type and $\varphi \in \Sigma$. The value of $x_{t,\varphi}$ denotes the number of columns of column type t whose corresponding character in the desired solution string of Closest String is set to φ. Thus, the ILP seeks to minimize

$$\max_{1 \leq i \leq k} \sum_{t} \sum_{\varphi \in (\Sigma - \{\varphi_{t,i}\})} x_{t,\varphi},$$

where $\varphi_{t,i}$ denotes the alphabet symbol at the ith entry of column type t. The following two constraints have to be fulfilled when minimizing the above function.

1. All variables $x_{t,\varphi}$ have to be nonnegative integers.
2. Let $\#_t$ denote the number of columns of type t in the input instance (taking into account isomorphism as described before). Then,

   $$\sum_{\varphi \in \Sigma} x_{t,\varphi} = \#_t$$

   for every column type t.

Actually, Theorem 4 refers to the integer linear programming feasibility problem and, moreover, a Closest String instance also gives the maximum distance d allowed. Thus, we may obtain the following "feasibility formulation" where the above two constraints remain unchanged but the goal function that had to be minimized now translates into a third set of constraints, namely:

$$\sum_{t} \sum_{\varphi \in (\Sigma - \{\varphi_{t,i}\})} x_{t,\varphi} \leq d$$

for every string i, $1 \leq i \leq k$. Altogether, this yields fixed-parameter tractability for Closest String with respect to parameter k. Note, however, that the combinatorial explosion in k is huge and this approach appears to be impractical for $k > 4$ as some experimental investigations indicated. In this case, however, ILP heuristics such as branch-and-bound strategies may extend the range of practical applicability.

The above ILP approach, however, at least serves as a tool to help deciding whether a problem is fixed-parameter tractable and maybe after that it is possible to come up with a more efficient, direct approach to solve the given problem. As to Closest String, the ILP approach is the only one known to us that yields fixed-parameter tractability with respect to parameter k. In the conference version of this work, a direct combinatorial approach (avoiding ILPs) was given for $k = 3$, but already $k = 4$ remained open due to the enormous combinatorial complexity. Finally, note that there is an alternative ILP formulation for Closest String given by Ben-Dor *et al.* [2], where the variables have only binary values but the number of variables is $|\Sigma| \cdot L$ (for alphabet Σ and string length L). Hence, this ILP formulation does *not* imply the fixed-parameter tractability of Closest String with respect to parameter k. In conclusion, it remains open to give further examples besides Closest String where the described ILP approach turns out to be fruitful. More generally, it would be interesting to see more connections between fixed-parameter algorithms and integer linear programming.

We developed a C++ program using the GNU GLPK library to test the performance of a heuristic ILP solver using a branch-and-bound strategy on ILPs generated from Closest String instances. For our tests, we randomly generated Closest String instances (see Section 5 for an explanation of how these instances were generated) and translated them both to the ILP formulation given in [2] and to the ILP formulation presented in this section. In Figure 2(a), we display the running times of the GNU GLPK solver on instances with $L = 256$ and $d = 32$, measured on a SUN Blade workstation with Ultrasparc IIe processor (500 MHz) and 512 MB main memory. It is no surprise that the ILP formulation presented here turns out to be preferable in case of a small number of long input strings.

5 Empirical Results

We implemented Algorithm D using the programming language C, including the heuristic improvements discussed in Subsection 3.2 and also with the optional extension to solve d-Mismatch and DSS as shown in Subsection 3.3. The

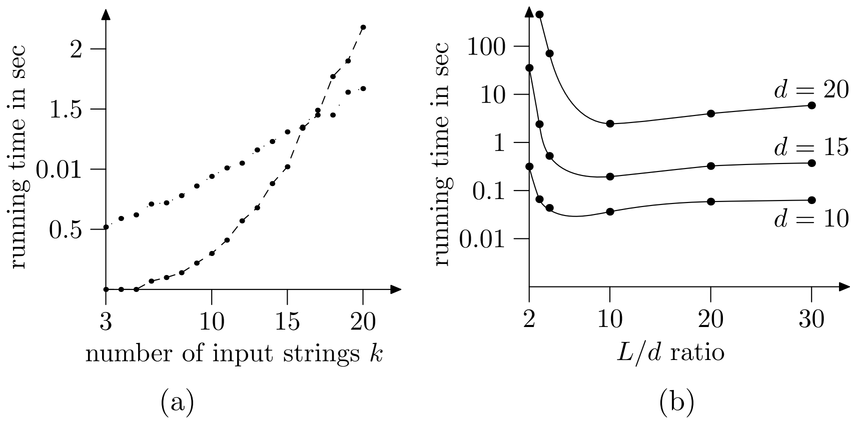

Figure 2: (a) Comparing the running times of a heuristic ILP solver using a branch-and-bound strategy for the ILP formulation given in [2] (dotted) and the one given in this paper (solid line); we used random instances with $L = 256$ and $d = 32$. (b) Comparing, on a logarithmic scale, the running time of Algorithm *csalg* on Closest String instances for differing length/mismatch ratio L/d.

following tests were performed on a Linux PC with 750 MHz processor and 192 MB main memory.

We report about tests on random instances which are generated as follows. Given as parameters the string length L, the number of strings k, the distance parameter d and the alphabet size $|\Sigma|$, we create a random instance by first computing a random string s of length L. Then, we compute k strings differing from s in d positions, each time switching d randomly chosen positions in s. The displayed results are average results taken from a range of 25 such random instances. We use alphabet size $|\Sigma| = 4$ if not indicated otherwise.

For the tests, we used the algorithm that scans the whole search tree for possible solutions, i.e., it does not stop when the first solution is found. Reasons are that one could want to find as many solutions as possible, and that, in this way, the running time is not affected by the location of a solution in the search tree.

Length/mismatch ratio. Our experiments with randomly generated data show that not only the number of mismatches but, moreover, the ratio of string length to the number of mismatches has a major impact on the difficulty of solving the problem. The results from Fig. 2(b) show that an increasing length L and a thereby increasing L/d ratio for a fixed value of d, can, in a limited range, significantly decrease the running time of the problem. This can be explained as follows. Due to the random generation of the input, with a larger L the

mismatches in the strings are more distributed over the columns than for a smaller L. Therefore, with large L/d ratio, the number of search tree branches for which solutions seem possible will be smaller. In contrary, we encounter large search tree sizes for a small L/d ratio (e.g., we found an average search tree size of 40205 for $L = 20, d = 10, k = 25$ compared to 2733 for $L = 30, d = 10, k = 25$). At some point, of course, this decrease in running time is outweighed by the linear time factor needed for the reduction to the problem kernel.
Thus, when considering the values of mismatches d for which we can process Closest String instances in practice, we also have to take the L/d ratio into account. For instance, for a "hard" ratio of 2, i.e., the string length is twice the number of mismatches, we solve instances with $d = 15$ ($L = 30$, $k = 50$) in 200 seconds, and for an "easier" ratio of 3 we solve instances with $d = 20$ ($L = 60$, $k = 50$) in 98 seconds.

Number of input strings. When considering the running times for an increasing number of input strings (for fixed values of L, d), we encounter two competing factors. On the one hand, an increase in the number of strings means an increase in the linear time factor which has to be spent in every node of the search tree. On the other hand, a growing number of strings means a growing number of constraints on the solutions and, therefore, a decreasing size of the search tree. Our experience with random data sets shows a high running time for small numbers of strings, decreasing with growing number of strings up to some turning point. From then on running time increases again since the linear factor spent in each search tree node becomes crucial. An example: for $L = 24$, $d = 12$, we need 6.2 sec for $k = 10$ (search tree size 934892), 4.3 seconds for $k = 100$ (search tree size 145390), and 8.5 seconds for $k = 400$ (search tree size 91879).

Search tree size. In Fig. 3(a), we compare the size of the search tree for given instances with the theoretical upper bound of $(d + 1)^d$. We note that the actual search trees are by far smaller than the bound predicts (heuristic improvements!).

Alphabet size. Fig. 3(b) shows examples of the influence of the alphabet size onto the running time. A very small alphabet size turns out to be harder than large alphabet size. The reason is the way how our input instances are generated. With larger alphabet size, the random mismatches that are implanted into the original string will be chosen from a larger set of alphabet symbols and will, therefore, be more "obvious" for the algorithm.

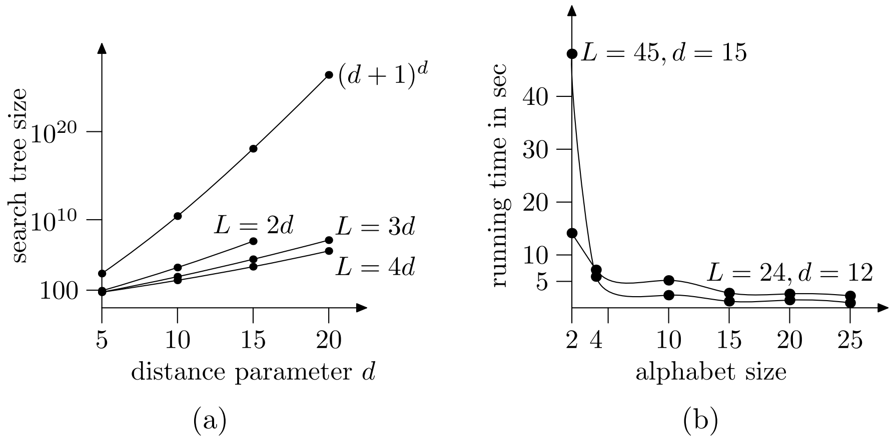

Figure 3: (a) Comparing, on a logarithmic scale, search tree sizes with the theoretical upper bound of $(d+1)^d$. Each line displays results for one fixed L/d ratio. (b) Comparing the running times for increasing alphabet size ($k = 50$).

Applications

Primer design. Primers are short sequences of nucleotides which are designed such that the primer hybridizes to a given DNA sequence (or, in our case, to all of a given set of DNA sequences) in order to provide a start point for DNA strand synthesis by PCR (polymerase chain reaction). The hybridization of primers depends on complex thermodynamic rules, but is largely determined by the number of "mismatching" positions which should be as small as possible. Designing candidates for primers is a task often done by biological experts using the output of multiple alignment programs which is evaluated by hand. We suggest to automize this task as follows. If we search primers of length L which are required to bind to each of a set of homologous sequences with at most d mismatches, we use the following strategy depicted in Fig. 4.

1. Compute an alignment of these sequences.
2. "Slide" a length L window over all the aligned strings, solving a Closest String instance for every window position.

Using our algorithm for d-Mismatch, (2) can be done in time $O(kL + (n - L)kd \cdot d^d)$ (i.e., linear time for constant d) where n is the length of the alignment. Using our algorithm, we can also incorporate additional properties that are desirable for primers. We can adjust the search to find primers without mismatches in initial positions or by preferring solutions with long common substrings as proposed in [23]. As additional filtering, we can select solutions that are favorable in terms of melting temperature.

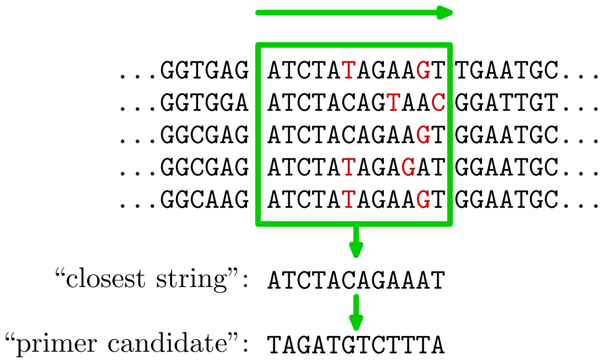

Figure 4: Strategy for designing primer candidates.

In an example experiment, we were faced with a slightly more general setting than described above. Given homologous sequences of parasite DNA as well as host DNA, the goal was to design primers that exclusively bind to the parasite sequences. The given data in this example was an alignment of length 715 with five sequences of parasite DNA (mushrooms) and four sequences of host DNA. We approached the problem by solving a combination of d-Mismatch and DSS on the parasite sequences as set of good strings and host sequences as set of bad strings. The desired length L of primers is between 15 to 20. Since the primers should have as few mismatches as possible, we consider here only $d_1 \leq 3$. E.g., with $L = 15$, $d_1 = 2$, the minimum value for which we find a primer candidate is $d_2 = 7$. For $L = 25$, we find a candidate with $d_1 = 2$ and $d_2 = 18$, or with $d_1 = 3$ and $d_2 = 15$. The advantage of the algorithm in this application is that it quickly (all runs are done in less than a second) finds all positions where primers are possible (and also finds if certain values of L, d_1 and d_2 do no allow a solution), whereas the human expert needs a lot of time and may find only obvious candidates.

Motif search. Another application of Closest String algorithms can be found in motif search. A motif is a common substring that occurs in several DNA sequences and that is only approximately preserved. Motifs are candidates for substrings of non-coding parts of the DNA sequence that have functions related, e.g., to gene expression. A formal definition of the motif search problem leads to the Closest Substring problem. This problem has recently been addressed by a number of heuristic approaches, e.g. [4, 22]. Algorithms like [4] or [22] compute a candidate set of substrings, every of these substrings in a different given string, without actually checking whether this candidate set is (or can be extended to) a set of substrings, one in each given string, that satisfies the Closest String property. Here, the Closest String algorithms can be used to perform this check. The computation of the candidate set is, as a rule, much more time consuming than the CLOSEST STRING check and, therefore, the additional time needed by the CLOSEST STRING algorithm for practically relevant values like $L < 50$ and small values of d can easily be tolerated [12].

6 Conclusion

In this paper, we described new and also practically promising fixed-parameter algorithms for consensus word problems from computational biology. In particular, all our algorithms for these, in general, NP-complete problems work in *linear* time for constant parameter values. This is of particular importance in signal finding and related applications where a small distance parameter d is normal (for instance, in primer design d-values around 5 are not unusual [6, 7]). Our results improve and generalize previous work and answer some open questions [6, 25].

As to future challenges, many questions remain to be answered. For example, we showed that, from the theoretical point of view, CLOSEST STRING is fixed-parameter tractable for fixed k. The shown algorithm, however, suffers from huge constant factors in the running time, even for moderate values of k, that make it impossible to find exact solutions with this algorithm for $k > 4$. Is it possible to give a fixed-parameter algorithm for parameter k that is usable for larger values of k and arbitrary values of L and d?

Another important question is that of the parameterized complexity of the CLOSEST SUBSTRING problem, which further generalizes d-MISMATCH in the way that the starting positions of the substrings of different strings may differ. Li *et al.* gave a polynomial time approximation scheme (PTAS) also for this problem [17], and it was shown that CLOSEST SUBSTRING is W[1]-hard with respect to k [9]. The parameterized complexity with respect to d or both d and k is still open.

Besides answering these questions, more experimental results concerning real biological data and their special properties are also desirable.

Acknowledgement. We thank Michael Weiß from the Biology Department, Universität Tübingen (group for Systematic Biology and Mycology) for his suggestions and for providing us with biological data. We thank Falk Hüffner for his help in the implementation of the algorithms. We are grateful to Gerhard J. Woeginger (Twente) for pointing us to the ILP result of Lenstra.

References

[1] 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.

[2] A. Ben-Dor, G. Lancia, J. Perone, and R. Ravi. Banishing bias from consensus sequences. In *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM)*, number 1264 in LNCS, pages 247–261, 1997. Springer.

[3] M. Blanchette, B. Schwikowski, and M. Tompa. Algorithms for phylogenetic footprinting. *Journal of Computational Biology*, 9(2):211–224, 2002.

[4] J. Buhler and M. Tompa. Finding motifs using random projections. *Journal of Computational Biology*, 9(2):225–242, 2002.

[5] R. G. Downey and M. R. Fellows. *Parameterized Complexity.* Springer, 1999.

[6] 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.

[7] P. A. Evans and H. T. Wareham. Practical algorithms for universal DNA primer design: an exercise in algorithm engineering. In N. El-Mabrouk *et al.* (eds.) *Currents in Computational Molecular Biology 2001*, pages 25–26, Les Publications CRM, 2001.

[8] M. R. Fellows. Parameterized complexity: the main ideas and some research frontiers. In *Proceedings of the 12th International Symposium on Algorithms and Computation (ISAAC)*, number 2223 in LNCS, pages 441–453, 2001. Springer.

[9] M. R. Fellows, J. Gramm, and R. Niedermeier. On the parameterized intractability of Closest Substring and related problems. In *Proceedings of the 19th Symposium on Theoretical Aspects of Computer Science (STACS)*, number 2285 in LNCS, pages 262–273, 2002. Springer.

[10] M. Frances and A. Litman. On covering problems of codes. *Theory of Computing Systems*, 30:113–119, 1997.

[11] L. Gąsieniec, J. Jansson, and A. Lingas. Efficient approximation algorithms for the Hamming center problem. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 905–906, 1999, ACM Press.

[12] J. Gramm, F. Hüffner, and R. Niedermeier. Closest strings, primer design, and motif search. In L. Florea *et al.* (eds), *Currents in Computational Molecular Biology 2002*, poster abstracts of RECOMB 2002, pp. 74–75.

[13] R. Kannan. Minkowski's convex body theorem and integer programming. *Mathematics of Operations Research*, 12:415–440, 1987.

[14] J. C. Lagarias. Point lattices. In R. L. Graham *et al.* (eds.), *Handbook of Combinatorics*, pages 919–966. MIT Press, 1995.

[15] J. K. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang. Distinguishing string selection problems. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 633–642, 1999, ACM Press. To appear in *Information and Computation.*

[16] H. W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8:538–548, 1983.

[17] M. Li, B. Ma, and L. Wang. On the Closest String and Substring Problems. *Journal of the ACM*, 49(2):157–171, 2002.

[18] G. L. Nemhauser and L. A. Wolsey *Integer and Combinatorial Optimization.* John Wiley & Sons, 1988.

[19] R. Niedermeier and P. Rossmanith. A general method to speed up fixed-parameter-tractable algorithms. *Information Processing Letters*, 73: 125–129, 2000.

[20] A. M. Odlyzko. Asymptotic enumeration methods. In R. L. Graham *et al.* (eds.) *Handbook of Combinatorics*, pages 1063–1229. MIT Press, 1995.

[21] P. A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach.* MIT Press, 2000.

[22] P. A. Pevzner and S.-H. Sze. Combinatorial approaches to finding subtle signals in DNA sequences. In *Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 269–278, 2000. AAAI Press.

[23] S. Rahmann. Rapid large-scale oligonucleotide selection for microarrays. To appear in *Proc. of Second Workshop on Algorithms in Bioinformatics*, September 2002, Rome, LNCS, Springer.

[24] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1999.

[25] N. Stojanovic, P. Berman, D. Gumucio, R. Hardison, and W. Miller. A linear-time algorithm for the 1-mismatch problem. In *Proceedings of the 5th Workshop on Algorithms and Data Structures (WADS)*, number 1272 in LNCS, pages 126–135, 1997, Springer.

[26] N. Stojanovic, L. Florea, C. Riemer, D. Gumucio, J. Slightom, M. Goodman, W. Miller, and R. Hardison. Comparison of five methods for finding conserved sequences in multiple alignments of gene regulatory regions. *Nucleic Acids Research*, 27(19):3899–3910, 1999.