LCNS, Vol 2223, pp. 441–452, Springer 2001

Exact Solutions for Closest String and Related Problems

Jens Gramm[1]*, Rolf Niedermeier[1], and Peter Rossmanith[2]

[1] Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Sand 13, D-72076 Tübingen, Fed. Rep. of Germany
`{gramm, niedermr}@informatik.uni-tuebingen.de`
[2] 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 positive 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. We show how to solve Closest String in linear time for constant d (the exponential growth is $O(d^d)$). We extend this result to the closely related problems d-Mismatch and Distinguishing String Selection. Moreover, we discuss fixed parameter tractability for parameter k and give an efficient linear time algorithm for Closest String when $k = 3$. Finally, the practical usefulness of our findings is substantiated by some experimental results.

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 [10, Section 8.6]. A central problem herein is the so-called Closest String (or, equivalently, Consensus String or Center String) problem: Given k strings $s_1, s_2, \ldots, s_k$ over alphabet Σ of length L each, and a positive integer d, is there a string s such that $d_H(s, s_i) \leq d$ for all $i = 1, \ldots, k$? 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) [11, 12] and the so-called Distinguishing String Selection problem [6] (for a brief overview on biological applications concerning signal finding and primer design refer to, e.g., [6]). All these problems are, in general, NP-hard [4, 6][1].

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

[1] Frances and Litman [4] show the NP-completeness of Closest String, considering it from the viewpoint of coding theory (so-called Minimum Radius problem).

Despite their hardness, these problems need to be solved in practice. Li *et al.* [9] gave 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, 2] 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., $d < 10$ [3]), it is important to ask whether efficient polynomial or even better linear time algorithms are possible when d or k are constants. Put in slightly more general terms, this is the question for 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 [3] for the parameterized complexity of Closest String with parameter d. Furthermore, we can generalize our result to d-Mismatch, improving work and positively answering an open question of Stojanovic *et al.* [11], 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. Our second, technically more involved main result is that Closest String can be solved efficiently in linear time for $k = 3$. Using an integer linear program formulation, we can observe that the problem is fixed parameter tractable with respect to k—the exponential term in k is huge, however. Finally, we indicate the practical usefulness and potential of our algorithms by some experimental results based on implementations of our linear time algorithm for constant d. Due to the lack of space, we omit some proofs and details.

2 Preliminaries

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., $|\{\, p \mid s_i[p] \neq s_j[p] \,\}|$. Given a set of strings $S = \{s_1, s_2, \ldots, s_k\}$, each string 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 $\sum_{i=1,\ldots,k} d_H(s', s_i) < \sum_{i=1,\ldots,k} d_H(s, s_i)$. An optimal median string for $S = \{s_1, s_2, \ldots, s_k\}$ can be computed by choosing in every column the letter occurring most often. We call this a *majority vote*; it, however, is 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. The *columns* of a Closest String instance are the columns of this matrix. For reordering the columns, we use a permutation on strings as follows. Given a string $s = c_1 c_2 \dots c_L$ with $c_1, \dots, c_L \in \Sigma$ for alphabet Σ and a permutation $\pi : \{1, \dots, L\} \to \{1, \dots, L\}$. Then, $\pi(s) = c_{\pi(1)} c_{\pi(2)} \dots c_{\pi(L)}$.

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

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. Isomorphic columns form *column types*.
This can be generalized as follows. W.l.o.g., let a always denote the letter that occurs in a column most often, let b always denote the letter that has the second most often occurrences and so on. This property of being *normalized* 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.

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

With the following observation by Evans and Wareham [3], we find that it is sufficient to solve instances containing less than kd columns. This yields a so-called "reduction to problem kernel" [1, 2]. 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 of length L and integer d. If the resulting $k \times L$ matrix has more than kd dirty columns, then there is no solution to this instance.*

3 A linear time solution for constant *d* and applications

We show that Closest String, although NP-complete in general, is solvable in linear time for constant d, discuss heuristic improvements, and apply this result to the related problems of d-Mismatch and Distinguishing String Selection.

```
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 $\Delta d$.
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 $i \in \{1, \ldots, k\}$ 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"
```

Fig. 1. Algorithm D. Inputs are a CLOSEST STRING instance consisting of a set of strings $S = \{s_1, s_2, \ldots, s_k\}$ of length L, and an integer d. First, we perform a preprocessing performing the reduction to 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 $CSd(s_1, d)$.

3.1 Bounded search tree algorithm

In Fig. 1, we outline a recursive algorithm solving CLOSEST STRING. It is based on the well-known bounded search tree paradigm oftenly successfully applied in parameterized complexity [1, 2]. 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 $\max_{i=1,\ldots,k} d_H(s, s_i) \leq d$.*

Theorem 1 *Algorithm D solves* CLOSEST STRING *in time $O(kL + kd \cdot d^d)$.*

Proof. (Sketch) **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 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 bound of $(d+1)^d$ on the search tree size. Every step of the recursion only needs linear time $O(kd)$. Before starting the recursion, we build a table containing the distances of the candidate s to all other given strings in time $O(kd)$. 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 $O(kd)$ to find the positions in which s and s_i differ. For $d+1$ of the differing positions we modify the candidate, 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 one exists. Here, 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., $P := \{\, p \mid s_1[p] \neq s_i[p] \,\}$. 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 $P_1 := \{\, p \mid s_1[p] \neq s[p] = s_i[p] \,\}$. 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 $P_2 := \{\, p \mid s[p] \neq s_i[p] \,\}$. 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 $d_H(s, s_i) > d + \Delta d$ for some string s_i of the given strings $s_1, \ldots, s_k$. □

With Algorithm D, we can 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 can find the optimal distance value $\leq d$ at the cost of a constant time factor.
Finally, we note that we can further improve the exponential term d^d significantly by asymptotic considerations. We defer this to the long version of the paper.

3.2 Heuristic improvements

Since the search tree size is the critical component in the algorithm's running time, the goal is to keep it as small as possible. Keeping in mind the initial candidate string, there is no use in changing a position that has already been changed before. In addition, we store all characters that have been tried on the same or a higher level of recursion (and restored again after the corresponding branch of the search tree has been visited); there are at most k many for every position. For branching, we consider only setting a character at a position if we didn't already try this character on the same or a higher level of recursion.

Concerning branching, a good strategy seems to be to select, of the strings s_i, $i = 1, \ldots, k$ with $d_H(s, s_i) > \Delta d$, the string with maximal $d_H(s, s_i)$. Moreover, since we search the solution in the neighborhood of the initial 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

As mentioned before, our basic algorithm does not find *all* solutions. We can, however, modify it in order to deliver all solutions in time generally better than a trivial brute force approach. We omit the details.

Solving d-Mismatch. Let $s_{i,p,L}$ denote the length-L substring of a given string s_i starting at position p. Then, given strings $s_1, s_2, \ldots, s_k$ of length n and integers L and d, 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 $i = 1, \ldots, k$. We achieve a linear running time for constant d 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 when shifting the window of L consecutive columns under consideration from position p (containing columns p to $p + L - 1$) to position $p + 1$ in time $O(k)$: (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 less than kd columns:

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

Solving Distinguishing String Selection (DSS). In this problem, we are given "good" strings $s_1, \ldots, s_{k_1}$, "bad" strings $s'_1, \ldots, s'_{k_2}$, and positive integers d_1, d_2. We ask for an s "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 ones, i.e., $\min_{j=1,\ldots,k_2} d_H(s, s'_j) \geq L - d_2$.

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

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., $\max_{i=1,\dots,k_1} d_H(s,s_i) \le d_1$ *and* $\min_{j=1,\dots,k_2} d_H(s,s'_j) \ge L - d_2$.
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 $d_H(s,s_i) \le d_1$ for all $i = 1,\dots,k_1$, 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 $(d_2 + 1) \cdot |\Sigma|$ for the subcases to branch into.

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 Efficient linear time solution for $k = 3$

For a constant number k of strings, Closest String is solvable in linear time with the following argument. The number of column types for k strings depends only on k (namely, it is given by the Bell number $B(k) \le k!$). Using the column types, Closest String can be formulated as an integer linear program (ILP) having only $B(k) \cdot (k-1)$ variables. Since ILPs with a constant number of variables can be solved in linear time [5, 7, 8], this is also true for Closest String with constant k. The algorithms, however, lead to huge running times, even for moderate number of variables. For this reason, we present a direct (not using linear programming) and efficient linear time algorithm that solves Closest String for $k = 3$. We start with transforming the instance into a normalized one and splitting it into "blocks." We obtain a block by reordering (cf. Lemma 1) the columns of the $k \times L$ matrix and considering consecutive columns in the reordered instance as a block. By sorting, the columns are already ordered in the sequence in which we will process them:

(0) *"Identity Case."* All columns of type $(a,a,a)^t$.
(1) *"Diagonal Case."* All blocks of type $(baa, aba, aab)^t$.
(2) *"3/2 Letters Case."* All blocks of type $(aa, ba, cb)^t$, $(aa, bb, ca)^t$, or $(ab, ba, ca)^t$ (the order of these three types among each other does not matter).
(3) *"2/2 Letters Case."* All blocks of type $(aa, ab, ba)^t$, $(ab, aa, ba)^t$, or $(ab, ba, aa)^t$ (it will be shown in Lemma 9 that we can find only one of these three possibilities, since, otherwise, we would have been able to build an additional block of type (1)).
(4) *"Remaining 2 Letters Case."* All blocks of type $(a,a,b)^t$, $(a,b,a)^t$, or $(b,a,a)^t$ (as in case (3), we can find only one of these possibilities, since, otherwise, we would have been able to build an additional block in (3)).
(3′) *"3×3 Letters Case."* All blocks of type $(aaa, bbb, ccc)^t$.

Algorithm 3-Strings
Input: Strings s_1, s_2, s_3.
Output: $CS3(s_1, s_2, s_3)$, which is an optimal closest string for s_1, s_2, s_3.

(K0) Given $\begin{bmatrix} a \cdot s'_1 \\ a \cdot s'_2 \\ a \cdot s'_3 \end{bmatrix}$, then return $a \cdot CS3(s'_1, s'_2, s'_3)$. "Identity Case"

(K1) Given $\begin{bmatrix} baa \cdot s'_1 \\ aba \cdot s'_2 \\ aab \cdot s'_3 \end{bmatrix}$, then return $aaa \cdot CS3(s'_1, s'_2, s'_3)$. "Diagonal Case"

(K2) Given $\begin{bmatrix} aa \cdot s'_1 \\ ba \cdot s'_2 \\ cb \cdot s'_3 \end{bmatrix}$ ($\begin{bmatrix} aa \cdot s'_1 \\ bb \cdot s'_2 \\ ca \cdot s'_3 \end{bmatrix}$ or $\begin{bmatrix} ab \cdot s'_1 \\ ba \cdot s'_2 \\ ca \cdot s'_3 \end{bmatrix}$, resp.), "3/2 Letters Case"
then return $ca \cdot CS3(s'_1, s'_2, s'_3)$ ($ba \cdot CS3(s'_1, s'_2, s'_3)$ or $aa \cdot CS3(s'_1, s'_2, s'_3)$, resp.).

(K3) Given $\begin{bmatrix} aa \cdot s'_1 \\ ab \cdot s'_2 \\ ba \cdot s'_3 \end{bmatrix}$, $\begin{bmatrix} ab \cdot s'_1 \\ aa \cdot s'_2 \\ ba \cdot s'_3 \end{bmatrix}$ or $\begin{bmatrix} ab \cdot s'_1 \\ ba \cdot s'_2 \\ aa \cdot s'_3 \end{bmatrix}$, "2/2 Letters Case"
then return $aa \cdot CS3(s'_1, s'_2, s'_3)$.

(K4) Given $\begin{bmatrix} a^l \\ a^l \\ b^l \end{bmatrix}$, $\begin{bmatrix} a^l \\ b^l \\ a^l \end{bmatrix}$, or $\begin{bmatrix} b^l \\ a^l \\ a^l \end{bmatrix}$ for some integer l, "Remaining 2 Letters Case"
then return $a^{\lceil l/2 \rceil} b^{\lfloor l/2 \rfloor}$.

(K3′) Given $\begin{bmatrix} aaa \cdot s'_1 \\ bbb \cdot s'_2 \\ ccc \cdot s'_3 \end{bmatrix}$, then return $abc \cdot CS3(s'_1, s'_2, s'_3)$. "3 × 3 Letters Case"

(K4′) Given $\begin{bmatrix} aa \\ bb \\ cc \end{bmatrix}$ (or $\begin{bmatrix} a \\ b \\ c \end{bmatrix}$, resp.), "Remaining 3 Letters Case"
then return ab (or a, resp.).

Fig. 2. Algorithm 3-Strings solving Closest String for $k = 3$.

(4′) *"Remaining 3 Letters Case."* All blocks of type $(a, b, c)^t$.

Thus, in a natural way, we obtain various *block types.* We can make sure that after this reordering process no columns are left. An instance in which the columns are ordered as explained, we call *ordered* instance. Transforming an arbitrary instance into an ordered instance can be done in linear time.
Algorithm 3-Strings shown in Fig. 2 considers the single blocks of a normalized and ordered instance, one after the other, and combines their solutions to a solution for the whole problem instance. Later in this section, we will show that the algorithm, in linear time, finds an optimal solution.

Lemma 7 *Let s be an optimal median string for s_1, s_2, and s_3 and let $3 \cdot \max_{i=1,2,3} d_H(s, s_i) - \sum_{i=1,2,3} d_H(s, s_i) \leq 2$ (in particular, this is true when $d_H(s, s_1) = d_H(s, s_2) = d_H(s, s_3)$). Then s is an optimal closest string for s_1, s_2, and s_3.*

Let (Ki)* denote that instruction (Ki), $i \in \{0, 1, 2, 3, 4, 3', 4'\}$, is applied an arbitrary number of times (including zero).

Lemma 8 *Given a normalized and ordered* Closest String *instance, then the only possible successions in the application of instructions in Algorithm 3-Strings are* $(K0)^*(K1)^*(K2)^*(K3)^*(K4)^*$ *and* $(K0)^*(K1)^*(K2)^*(K3')^*(K4')^*$.

Lemma 9 *Given a normalized and ordered* Closest String *instance, we have as type (3) blocks only blocks* $(aa, ab, ba)^t$*, only blocks* $(ab, aa, ba)^t$*, or only blocks* $(ab, ba, aa)^t$.

Lemma 10 *Let* s_1*,* s_2*, and* s_3 *be normalized, and let* s *be an additional string.*
(a) *Let for every column in* $(s_1, s_2, s_3)^t$*, the respective letter in* s *be a majority vote and let* s_1*,* s_2*, and* s_3 *contain no column* $(a, b, c)^t$ *such that the respective letter in* s *is* a*. Further, let* $d_H(s, s_1) \leq d_H(s, s_2) = d_H(s, s_3)$*. Then* s *is an optimal closest string for* s_1*,* s_2*, and* s_3*.*
(b) *Let, for every column in* $(s_1, s_2, s_3)^t$ *that is not* $(a, a, b)^t$*, the respective letter in* s *be a majority vote, and let for every column* $(a, b, c)^t$ *the respective letter in* s *be* c*. Further, let* $d_H(s, s_1) \leq d_H(s, s_2)$ *and either* $d_H(s, s_2) = d_H(s, s_3)$ *or* $d_H(s, s_2) = d_H(s, s_3) - 1$*. Then,* s *is an optimal closest string for* s_1*,* s_2*, and* s_3*.*

Theorem 4 Closest String *for* $k = 3$ *can be solved in linear time.*

Proof. (Sketch) **Running time**. Algorithm 3-Strings makes at most L recursive calls and each call takes only constant time, yielding linear running time.
Correctness. From Lemma 8, we know that the order of instructions is (K0)*(K1)*(K2)*(K3)*(K4)* or (K0)*(K1)*(K2)*(K3$'$)*(K4$'$)*.
Now, the proof is given by considering the instructions separately. We assume a Closest String instance $(s_1' s_1'', s_2' s_2'', s_3' s_3'')$ with $|s_1'| = |s_2'| = |s_3'|$, such that s_1', s_2', and s_3' are those parts of the strings such that (K0), (K1), and (K2) apply to them and produce s'. Then s_1'', s_2'', and s_3'' are processed either by (K3) and (K4), or by (K3$'$) and (K4$'$), resulting in s''. We first show that s' is an optimal closest string for s_1', s_2', and s_3', and then show that $s = s's''$ is an optimal closest string for the whole instance.
Instructions (*K0*)*,* (*K1*)*, and* (*K2*)*:* These instructions are applied to blocks of type (0), (1), and (2). We can easily check that they produce s' with $d_H(s', s_1') = d_H(s', s_2') = d_H(s', s_3')$. Since we choose in every column the letter as majority vote, s' is an optimal median string for s_1', s_2', and s_3'. By Lemma 7 we conclude that s' is an optimal closest string for s_1', s_2', and s_3'.
Instructions (*K3*) *and* (*K4*)*:* Following Lemma 9, we know that (K3) is applied only to one of the three cases mentioned in the instruction. W.l.o.g., we assume that it is only applied on blocks $(aa, ab, ba)^t$. Let l be the total number of the applications of (K3). Thus, (K3) adds a^{2l} to the closest string constructed by (K0) to (K2), resulting in string $\hat{s}$. If $\hat{s_1}$, $\hat{s_2}$, and $\hat{s_3}$ are the strings processed up to this point, we have $d_H(\hat{s}, \hat{s_1}) + l = d_H(\hat{s}, \hat{s_2}) = d_H(\hat{s}, \hat{s_3})$ and all letters in (K3) are chosen as majority vote.

To satisfy the premises of Lemma 10(a), it remains to show that there is no column $(a,b,c)^t$ in s_1, s_2, and s_3 for which we set the respective position in s to a: For column $(a,b,c)^t$, we only set a in instruction (K2) if it occurs together with column $(b,a,a)^t$. In this case, however, we could form a block of type (1), since we have columns $(a,b,a)^t$ and $(a,a,b)^t$, necessary for (K3). This contradicts the assumption that the instance is ordered. It follows that no column $(a,b,c)^t$ is assigned a. Thus, we conclude with Lemma 10(a) that $\hat{s}$ is an optimal closest string for $\hat{s_1}$, $\hat{s_2}$, and $\hat{s_3}$. For instruction (K4) the correctness is shown, by use of Lemma 10(b), in a very similar way as for (K3). We omit the details here.
Instructions (K3′) and (K4′). We are only left with columns of type $(a,b,c)^t$. Instruction (K3′) applies as long as the number of these columns is larger than three. What remains are one or two columns of type $(a,b,c)^t$, which (K4′) takes care of. Given input strings s_1, s_2, and s_3, they are processed after the application of (K4′). We can check that now either $d_H(s,s_1) = d_H(s,s_2) - 1 = d_H(s,s_3) - 1$ or $d_H(s,s_1) = d_H(s,s_2) = d_H(s,s_3) - 1$. Since for all columns we chose the letter in s as majority vote, s is an optimal median string for s_1, s_2, and s_3. By Lemma 7, we conclude that s is an optimal closest string. □

5 Experimental results

We implemented Algorithm D using the programming language C, including the heuristic improvements discussed in Subsections 3.2 and 3.3. We performed tests on a LINUX PC with 750 MHz processor and 192 MB main memory. First, we report about tests on random instances with $|\Sigma| = 4$ where we scan the whole search tree, i.e., we do not stop when the first solution is found. The displayed results are average results taken from a range of ten such random data sets.

Length/mismatch ratio. For instances containing only dirty columns, our experiments with randomly generated data show that not only the number d of mismatches allowed but, moreover, the ratio of string length L to d has a major impact on the difficulty of solving the problem. The results from Fig. 3(a) show that an increasing length L and a thereby increasing L/d ratio for a fixed value of d will significantly decrease the running time of the algorithm up to some point. When considering the values of d for which we can process Closest String instances in practice, we have to take this ratio into account. E.g., 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 about 200 sec, and for an "easier" ratio of 3 we can solve instances with $d = 20$ ($L = 60$, $k = 50$) in 100 sec.

Number of input strings. When considering the running times for an increasing number k 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 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

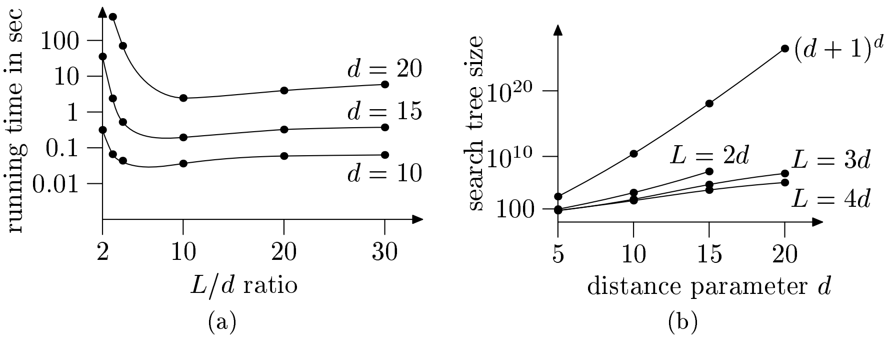

Fig. 3. (a) Comparing, on a logarithmic scale, the running time of Closest String instances for differing length/mismatch ratio L/d ($k = 25$). (b) Comparing, also on a logarithmic scale, search tree sizes to the theoretical upper bound of $(d + 1)^d$. Each line displays results for one fixed L/d ratio ($k = 25$).

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. E.g., for $L = 24$, $d = 12$, we need 6.2 sec for $k = 10$ (search tree size 934892), 4.3 sec for $k = 100$ (search tree size 145390), and 8.5 sec for $k = 400$ (search tree size 91879).

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

*Primer design by solving a combination of d-*Mismatch *and* Distinguishing String Selection. We applied our algorithm to compute candidates for primers, a task the biological expert otherwise does by hand. In our application, we are confronted with probes that may contain parasite DNA (mushrooms) as well as host DNA, and the goal is to design primers that exclusively bind to the parasite sequences. The given data in this example are an alignment of length 715 with five sequences of parasite DNA and four sequences of host DNA. We approach the problem by solving DSS with 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 for d_1 only values ≤ 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 primer candidates meet the specified conditions (and also finds if certain values of L, d_1 and d_2 do not allow a solution), whereas the task is tedious for the human expert who might only find obvious candidates.

6 Conclusion

We described new and also practically promising exact algorithms for consensus word problems motivated by 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, e.g., small distance parameter d is normal (for instance, in primer design d-values around 5 are not unusual [3]). Our results improve and generalize previous work and answer some open questions [3, 11]. It seems hard to extend our results to the more general Closest Substring problem, which is more relevant in biological applications dealing with unaligned sequences.

Acknowledgment. We thank Michael Weiß from the Biology Department (group for Systematic Biology and Mycology, Universität Tübingen) for his suggestions and for providing us with biological data. 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. R. G. Downey and M. R. Fellows. *Parameterized Complexity.* Springer. 1999.
3. P. A. Evans and H. T. Wareham. Practical non-polynomial time algorithms for designing universal DNA oligonucleotides: a systematic approach. Manuscript, April 2001.
4. M. Frances and A. Litman. On covering problems of codes. *Theory of Computing Systems*, 30:113–119, 1997.
5. R. Kannan. Minkowski's convex body theorem and integer programming. *Mathematics of Operations Research*, 12:415–440, 1987.
6. 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, ACM Press. To appear in *Information and Computation.*
7. J. C. Lagarias. Point lattices. In R. L. Graham *et al.* (eds.) *Handbook of Combinatorics*, pages 919–966. MIT Press, 1995.
8. H. W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8:538–548, 1983.
9. M. Li, B. Ma, and L. Wang. Finding similar regions in many strings. In *Proc. of 31st ACM STOC*, pages 473-482, 1999. ACM Press.
10. P. A. Pevzner. Computational Molecular Biology: An Algorithmic Approach. MIT Press, 2000.
11. N. Stojanovic, P. Berman, D. Gumucio, R. Hardison, and W. Miller. A linear-time algorithm for the 1-mismatch problem. In *Proc. of 5th WADS*, number 1272 in LNCS, pages 126–135, 1997, Springer.
12. 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.