Swiftly Computing Center Strings

Franziska Hufsky[1,3], Léon Kuchenbecker[2], Katharina Jahn[2], Jens Stoye[2], and Sebastian Böcker[1]

[1] Lehrstuhl für Bioinformatik, Friedrich-Schiller-Universität Jena, Ernst-Abbe-Platz 2, Jena, Germany, {franziska.hufsky,sebastian.boecker}@uni-jena.de
[2] AG Genominformatik, Technische Fakultät, Universität Bielefeld, Germany, kjahn@cebitec.uni-bielefeld.de, {lkuchenb,stoye}@techfak.uni-bielefeld.de
[3] International Max Planck Research School, Jena, Germany

Abstract. The center string (or closest string) problem is a classical computer science problem with important applications in computational biology. Given k input strings and a distance threshold d, we search for a string within Hamming distance d to each input string. This problem is NP-complete. In this paper, we focus on exact methods for the problem that are also fast in application. First, we introduce data reduction techniques that allow us to infer that certain instances have no solution, or that a center string must satisfy certain conditions. Then, we describe a novel search tree strategy that is very efficient in practice. Finally, we present results of an evaluation study for instances from a biological application. We find that data reduction is mandatory for the notoriously difficult case $d = d_{\text{opt}} - 1$.

1 Introduction

The Center String problem (also called Closest String problem) is defined as follows: Given k strings of length L over an alphabet Σ and a distance threshold d, find a string of length L that has Hamming distances at most d to each of the given strings.

The Center String problem has been studied extensively in theoretical computer science and particularly in computational biology [5,9], and has various applications such as degenerate PCR primer design [10] or motif finding [2,5]. We are particularly interested in its application as part of finding approximate gene clusters: The increasing speed of genome sequencing and the resulting number of available data offers the possibility of comparing gene order of whole genomes. During the course of evolution, speciation results in the divergence of genomes that initially have the same gene order and content. Conserved gene order is evidence for some biological signal [11]. Approximate gene cluster models account for reordering inside the gene cluster, as well as additional and missing genes in the compared genomes [1, 8]. The *center gene cluster model* limits the distance between the gene cluster and each of the approximate occurrences. For given approximate occurrences, finding the center gene cluster is equivalent to finding a center string for binary input strings.

Previous Work. The Center String problem is NP-complete even for three strings [3, 5], hence no polynomial time algorithm can exist unless P = NP. Different approaches have been studied for the problem: Ma and Sun [7] presented a polynomial time approximation scheme with time complexity $O(n^{O(\epsilon^{-2})})$ for an approximation ratio of $1+\epsilon$ for any $\epsilon > 0$. Also, heuristics and parallel implementations with good practical running times have been developed [6]. The drawback of these approaches is that they cannot guarantee to find an exact solution.

Parameterized algorithms use a parameter to describe the complexity of a problem instance and restrict the running time using this parameter, while at the same time guarantee to find optimal solutions. Parameters that have been studied in the literature for the Center String problem are the distance threshold d and the number of input strings k. For the latter parameter, Gramm *et al.* [4] showed that the problem is fixed-parameter tractable using an Integer Linear Program. Evaluations indicate that this approach is of theoretical interest only and impractical for $k \geq 5$. Regarding the distance threshold d, in the same paper an algorithm was given with running time $O(kL + kd^{d+1})$. Later, Ma and Sun [7] presented an algorithm with running time $O\big(kL + kd \cdot 16^d \, (|\Sigma| - 1)^d\big)$. Recently, Wang and Zhu [9] improved the running time to $O\big(kL + kd \cdot 9.53^d \, (|\Sigma| - 1)^d\big)$. All of these algorithms are based on the search tree paradigm. Note that for binary strings the term $(|\Sigma| - 1)^d$ vanishes.

Our Contribution. In this paper, we focus on exact methods that are also swift in application. We have developed an advanced preprocessing to quickly filter out unsolvable instances. Additionally, we compute rules that can be used within search tree algorithms to bound the search space, excluding unsolvable instances. We show how to integrate this information into the algorithms from [4, 7]. We then present a new search tree strategy called *MismatchCount* that, despite its bad worst case running time, works extremely well in practice. We implemented all three algorithms to evaluate their performance in combination with our preprocessing. We then present results of our experimental evaluation, showing that preprocessing and the novel algorithm improve running times by several orders of magnitude. We find that particularly the case $d = d_{\text{opt}} - 1$ is notoriously difficult for all approaches, where d_{opt} is the smallest distance value for which a solution exists.

2 Preliminaries

Given a string s over a finite alphabet Σ, let $s[i]$ indicate the ith character of s and $s[i, j]$ the substring of s starting at position i and ending at position j. The length of s is denoted by $|s|$.

The *Hamming distance* $d_{\text{H}}(s, t)$ of two strings s and t of the same length L is the number of positions p with $s[p] \neq t[p]$. Let $R = \{p_1, \ldots, p_m\} \subseteq \{1, \ldots, L\}$ be a set of positions such that $p_i < p_{i+1}$ for all $1 \leq i < m$. Then $s|_R := s[p_1] \ldots s[p_m]$ denotes the subsequence of s restricted to the positions in R. We

define the Hamming distance of two strings s and t restricted to R as $d_{\mathrm{H}}^{R}(s,t) := d_{\mathrm{H}}(s|_R, t|_R)$. For two strings s and t, let $D_{s,t} := \{p : s[p] \neq t[p]\} \subseteq \{1,\ldots,L\}$ denote the set of positions where s and t differ, and let $E_{s,t} := \{p : s[p] = t[p]\} = \{1,\ldots,L\} \setminus D_{s,t}$ be the set of positions where s and t are identical. Note that $d_{\mathrm{H}}^{D_{s,t}}(s,t) = d_{\mathrm{H}}(s,t)$. For k input strings $s_1,\ldots,s_k$, we write $D_{i,j} := D_{s_i,s_j}$ and $E_{i,j} := E_{s_i,s_j}$. For strings over the binary alphabet $\Sigma = \{0,1\}$, which is our default, we define $\overline{s[p]} = 1 - s[p]$.

The Center String problem is defined as follows: Given strings $s_1,\ldots,s_k$ of length L over an alphabet Σ, and a distance threshold d, find a string $\hat{s}$ of length L, called *center string*, that has Hamming distances at most d to each of the given strings.

For k strings $s_1,\ldots,s_k$ and distance threshold d, we can construct a *naïve kernel* as follows [4]: A position p is called *clean* if all sequences coincide at this position, i.e. $s_i[p] = s_j[p]$ for all $1 \leq i < j \leq k$, otherwise it is called *dirty*. One can easily see that there can be at most kd dirty positions if an instance allows for a center string of distance d. If a position is not dirty, then all strings share the same character at this position, and the center string will also share this character. So, we can remove all positions but the dirty ones, and get an instance of length $L \leq kd$.

In our algorithms, we assume a distance threshold d to be given. In applications, we might not know the distance threshold d in advance, but instead search for a center string minimizing d. We can do so by calling our algorithms repeatedly, increasing $d = 0, 1, 2, \ldots$ until a solution is found for $d = d_{\mathrm{opt}}$. Both in theory and in our experimental evaluation, we find that the running time of this iteration is governed by the last subroutine calls with $d = d_{\mathrm{opt}} - 1$ and $d = d_{\mathrm{opt}}$. That is why in our evaluations we will put special focus on these two cases.

In the following, we present a data reduction that will often allow us to conclude that no solution can exist for a particular distance threshold d. But in case we cannot rule out the existence of a center string by data reduction (what is obviously the case when $d = d_{\mathrm{opt}}$) we still have to decide whether a valid center strings exists. All algorithms for doing so, such as [4,7,9] and the *MismatchCount* algorithm presented below, are based on the search tree paradigm: In principle, we scan through all 2^L possible binary strings and test whether any such string is a center string of the input. The algorithms differ in the order in which they process the 2^L strings and, in particular, how they constrain the search space to speed up computations.

3 Data Reduction

Our data reduction is based on the pairwise comparison of the input strings. Given an instance $s_1,\ldots,s_k$ and d of the Center String problem, we can divide all pairs of strings $\{s_i, s_j\}$ into three groups: pairs with distance less than $2d-1$, greater than $2d$, or equal to $2d$ or $2d-1$. If there exist two strings s_i, s_j with Hamming distance $d_{\mathrm{H}}(s_i, s_j) > 2d$, then the instance has no solution.

This follows from the fact that a center string $\hat{s}$ can have at most distance d to each of s_i and s_j and, hence, $d_{\mathrm{H}}(s_i, s_j) \leq d_{\mathrm{H}}(s_i, \hat{s}) + d_{\mathrm{H}}(\hat{s}, s_j) \leq 2d$. So, $d \geq \frac{1}{2} \max_{i,j} d_{\mathrm{H}}(s_i, s_j)$ must hold for the instance to have a solution.

Solving Trivial Positions. Some positions of the solution string can be trivially solved. This is based on the following observation:

Lemma 1. *Given strings $s_1, \ldots, s_k$ and a center string $\hat{s}$ with distance d. For two strings s_i, s_j such that $d_{\mathrm{H}}(s_i, s_j) = 2d$ or $d_{\mathrm{H}}(s_i, s_j) = 2d - 1$, we have*

$$\hat{s}[p] = s_i[p] = s_j[p] \text{ for all } p \in E_{i,j} \ .$$

Proof. A center string with distance at most d to all strings is located central between the two strings s_i and s_j with distance $2d$ and hence has distance d to both of them. Thus, all positions fixed between s_i and s_j must also be fixed in $\hat{s}$. Our reasoning can be extended to string pairs with distance $2d - 1$: We need to change, in at least one of the strings, d positions and $E_{i,j}$ is the set of equal positions between *both* strings, hence we are still not allowed to change any position $p \in E_{i,j}$. □

As a reduction rule, if we find two strings s_i, s_j with $d_{\mathrm{H}}(s_i, s_j) \geq 2d-1$, then we can set $\hat{s}[p] := s_i[p]$ for all $p \in E_{i,j}$ and mark these positions as "permanent". Let $\mathcal{P}$ denote this set of permanent positions. By doing this for all s_i, s_j with $d_{\mathrm{H}}(s_i, s_j) = 2d$ or $d_{\mathrm{H}}(s_i, s_j) = 2d - 1$, we may run into conflicting situations where we have to permanently set a certain position to '0' and '1' simultaneously. We call such a situation a *conflict* and infer that the instance has no solution for the current choice of d. If we do not have a conflict, then applying this data reduction results in a partially solved solution string $\hat{s}$ with $\hat{s}[p] = c \in \Sigma$ fixed for all $p \in \mathcal{P}$, whereas all positions not in $\mathcal{P}$ still have to be decided.

Computation of Position Subsets. We next focus on pairs of strings s_i, s_j with $d_{\mathrm{H}}(s_i, s_j) = \delta < 2d - 1$. For a given center string $\hat{s}$ we define

$$X_{i,j}(\hat{s}) := \left\{ p \in E_{i,j} \ : \ s_i[p] = s_j[p] \neq \hat{s}[p] \right\}$$

as the set of positions where s_i and s_j agree, but disagree with the center string $\hat{s}$. We extend the reasoning behind Lemma 1 as follows:

Lemma 2. *Given strings $s_1, \ldots, s_k$ and a center string $\hat{s}$ with distance d. For two strings s_i, s_j such that $d_{\mathrm{H}}(s_i, s_j) < 2d - 1$, we have*

$$|X_{i,j}(\hat{s})| \leq d - \tfrac{1}{2} d_{\mathrm{H}}(s_i, s_j) \ .$$

Proof. Let $D := D_{i,j}$. Regarding the distances between $\hat{s}|_D$ and $s_i|_D$ as well as $s_j|_D$, we can state that $\hat{s}|_D$ has to at least one of the strings $s_i|_D$ or $s_j|_D$ a distance at least $\frac{1}{2} d_{\mathrm{H}}(s_i, s_j)$:

$$\max \left\{ d_{\mathrm{H}}(s_i|_D, \hat{s}|_D), d_{\mathrm{H}}(s_j|_D, \hat{s}|_D) \right\} \geq \tfrac{1}{2} d_{\mathrm{H}}(s_i, s_j) \ .$$

This is true since d_{H} is a metric and the triangle inequality holds, $d_{\mathrm{H}}(s_i|_D) \leq d_{\mathrm{H}}(s_i|_D, \hat{s}|_D) + d_{\mathrm{H}}(s_j|_D, \hat{s}|_D)$. Since we need a distance of at least $\frac{1}{2}d_{\mathrm{H}}(s_i, s_j)$ to solve the positions from D, a distance of at most $d - \frac{1}{2}d_{\mathrm{H}}(s_i, s_j)$ remains to solve the positions from E. □

Lemma 2 implies that the maximum number of positions $p \in E_{i,j}$ we are allowed to choose in the center string with $\hat{s}[p] \neq s_i[p]$ is bounded by $d - \frac{1}{2}d_{\mathrm{H}}(s_i, s_j)$. We can transform this observation into a reduction rule as follows: When, during search tree traversal or by other reduction rules, we have a partially solved solution string $\hat{s}$ such that

$$|X_{i,j}(\hat{s})| > d - \tfrac{1}{2}d_{\mathrm{H}}(s_i, s_j)$$

for any pair s_i, s_j, then we can infer that $\hat{s}$ cannot be extended to a solution for the current choice of d. For each pair s_i, s_j, we therefore set $x_{i,j} := d - \frac{1}{2}d_{\mathrm{H}}(s_i, s_j)$ and store all tuples $(E_{i,j}, x_{i,j})$ in an array $\mathcal{T}$.

Removing redundant information from $\mathcal{T}$ may lead to further trivially solved positions. This is done by removing, for all $1 \leq i < j \leq k$, all positions $p \in \mathcal{P} \cap E_{i,j}$ from $E_{i,j}$. Moreover, if $\hat{s}[p] \neq s_i[p]$ then we decrease $x_{i,j}$ by one.

For $x_{i,j} = 0$ we set all positions p from $E_{i,j}$ to "permanent" and include them in $\mathcal{P}$. Since $\mathcal{P}$ has changed, we continue our data reduction again until there is no tuple $(E_{i,j}, x_{i,j})$ with $x_{i,j} = 0$ in $\mathcal{T}$. For $x_{i,j} < 0$ we can easily infer that there must exist a conflict and, hence, the instance has no valid solution for this distance threshold d.

Cascading. To further enlarge the number of solved positions we consider all pairs of strings s_i, s_j with $x_{i,j} = 1$ and use *cascading*. A valid center string $\hat{s}$ has to agree with s_i in at least $|E_{i,j}| - 1$ positions from $E_{i,j}$, hence for binary strings at most one position $p \in E_{i,j}$ can be set to $\hat{s}[p] = \overline{s_i[p]}$.

To this end, we test for all positions $p \in E_{i,j}$ what we can infer from setting $\hat{s}[p] = \overline{s_i[p]}$. This implies $x_{i,j} = 0$, hence the remaining positions $q \in E_{i,j}$, $q \neq p$, are added to $\mathcal{P}$ and the tuple set $\mathcal{T}$ is reduced. If we run into a conflict during this reduction, we know that setting $\hat{s}[p] = \overline{s_i[p]}$ cannot result in a valid solution. In this case, we infer $\hat{s}[p] = s_i[p]$ and permanently set position p.

Unfortunately, if not running into a conflict, setting $\hat{s}[p] = s_i[p]$ is not mandatory. However, we get a partially solved solution string $\hat{s}_{p,v}$ and a set of "potentially permanent" positions $\mathcal{P}_{p,v}$ depending on the position p and the value $v = \overline{s_i[p]}$. We store this information in a set of rules $\mathcal{R}$.

We can use the set of rules $\mathcal{R}$ when solving the remaining instance by, say, a search tree algorithm. If, during the search tree traversal, we decide to set $\hat{s}[p] = v$ for the solution string $\hat{s}$, then we can immediately start the above data reduction: For all positions $q \in \mathcal{P}_{p,v} \setminus \mathcal{P}$, we set the solution string $\hat{s}[q] = \hat{s}_{p,v}[q]$. For the remaining positions $q \in \mathcal{P}_{p,v} \cap \mathcal{P}$, the condition $\hat{s}[q] = \hat{s}_{p,v}[q]$ must be met, otherwise we run into a conflict and, thus, this branch of the search tree does not lead to a valid solution.

4 Integration into Search Tree Algorithms

We can use the information derived during preprocessing, stored in the sets $\mathcal{P}, \mathcal{T}, \mathcal{R}$, to speed up the algorithms of Ma and Sun [7] and Gramm *et al.* [4]. Integrating the set of solved positions $\mathcal{P}$ into the algorithm of Ma and Sun is straightforward, as this algorithm tackles the more general Neighbor String problem. In all other cases, it is necessary to interweave the use of $\mathcal{P}, \mathcal{T}, \mathcal{R}$ with the actual search tree algorithms. Here, we use the information from $\mathcal{P}, \mathcal{T}, \mathcal{R}$ to shrink the search tree, by excluding search tree branches which cannot lead to a valid solution. To do so, we simply test if the (partial) string candidate of the search tree is already conflicting with this information. The integration of $\mathcal{P}, \mathcal{T}, \mathcal{R}$ is somewhat different for the algorithms of Ma and Sun and Gramm *et al.*, we defer the technical details to the full version of this paper. Unfortunately, the use of $\mathcal{P}, \mathcal{T}, \mathcal{R}$ does not change the worst-case running times of both algorithms. But our preprocessing, as an algorithm engineering technique, allows us to speed up the algorithms in practice, as demonstrated in Sect. 6.

5 Algorithm MismatchCount

After we have applied our data reduction rules, we have to solve the remaining instance using a search tree algorithm, like those from [4, 7]. In this section we present another such procedure, *MismatchCount*, that is very efficient in practice, as we will show below. Given binary strings $s_1, \ldots, s_k$ of length L and a distance threshold d, the MismatchCount algorithm solves the Closest String problem as follows: We iterate through all strings s with distance at most d to a chosen string s_i — without loss of generality, we may choose that string to be s_1. This leaves us with a search space of size $\sum_{d'=0}^{d} \binom{L}{d'}$. We present an enumeration scheme for those s that allows efficient testing for the center condition on each candidate, and that makes it possible to skip large areas of the search space based on information gained while checking those candidates.

The mismatch positions for d mismatches in s_1 (and therefore the center string candidates s) are enumerated, equivalently to generating all binary numbers of length m with d bits set to 1, in reverse order.

An example for the placement of at most three mismatches is shown in Fig. 1. For every s, its Hamming distance to the remaining strings $s_2, s_3, \ldots, s_k$ has to be checked. Rather than recomputing these distances entirely new for each candidate, the Hamming distances from the previous candidate s' are updated by increasing (resp. decreasing) the distances according to the changed positions. The running time for verifying a center candidate s is therefore bounded by $O(g \cdot k)$, where g is the number of positions changed from s' to s.

The overall number of changes performed during the enumeration of all center candidates can be determined like this: using the presented enumeration scheme, each position p in s is changed once to '1' and once to '0' for every configuration of $s[1, p-1]$ with at most d mismatches to $s_1[1, p-1]$. There are $\binom{p-1}{d'}$ such configurations for each $d' = 1, 2, \ldots, d$. Summing over all possible combinations

$d_{\mathrm{H}}(s, s_1) = 0$	$d_{\mathrm{H}}(s, s_1) = 1$	$d_{\mathrm{H}}(s, s_1) = 2$	$d_{\mathrm{H}}(s, s_1) = 3$
0 0 0 0 0	1 0 0 0 0	1 1 0 0 0	1 1 1 0 0
	0 1 0 0 0	1 0 1 0 0	1 1 0 1 0
	0 0 1 0 0	1 0 0 1 0	1 1 0 0 1
	0 0 0 1 0	1 0 0 0 1	1 0 1 1 0
	0 0 0 0 1	0 1 1 0 0	1 0 1 0 1
		0 1 0 1 0	1 0 0 1 1
		0 1 0 0 1	0 1 1 1 0
		0 0 1 1 0	0 1 1 0 1
		0 0 1 0 1	0 1 0 1 1
		0 0 0 1 1	0 0 1 1 1

Fig. 1. Enumeration scheme for all strings s with Hamming distance at most 3 to a bit string s_1 of length 5. The '0's denote matches between s and s_1 at the respective positions, while '1's denote mismatches.

of p and d', the number of bit changes performed can be bounded by $O(2^L)$. Since for each character change in s, k Hamming distances need to be updated, the overall worst-case running time of the algorithm is bounded by $O(k \cdot 2^L)$.

However, this worst-case analysis refers to the exploration of all legal mismatch configurations of s. As already mentioned above, the enumeration scheme enables us to skip large areas of that search space. Using the maximum Hamming distance $d_{\max} = \max_{i=2,\ldots,k}(d_{\mathrm{H}}(s, s_i))$ computed in each iteration, we can derive a lower bound for the number of positions that have to be changed in s in order to fulfill the center condition. Therefore, for each candidate s taken into consideration, we compute $c_{\min} = \lceil \frac{d_{\max} - d}{2} \rceil$, where $2 \cdot c_{\min}$ is the minimum number of positions in s that have to be changed when its successor is generated. This bound can be used in two ways: We cannot change $2 \cdot c_{\min}$ positions in s by changing the positions of less than $c_{\min}$ mismatches. Therefore, if currently all candidates s with $d_{\mathrm{H}}(s_1, s) = d$ are enumerated and we encounter a candidate that reveals a $c_{\min} > d$, we can proceed to the generation of candidates with $d_{\mathrm{H}}(s_1, s) = c_{\min}$, omitting the enumeration for all s with $d_{\mathrm{H}}(s_1, s) \in \{d, d+1, \ldots, c-1\}$.

Furthermore, even if $c_{\min}$ does not exceed d for a currently observed candidate, we can use that bound to skip the enumeration of certain candidates. Since we know that we have to change at least $2 \cdot c_{\min}$ positions in s, we can omit all enumeration steps that involve less than $c_{\min}$ mismatch positions.

Applying the data reduction to this algorithm is straightforward. Let $\mathcal{Q} := \{1, \ldots, L\} \setminus \mathcal{P}$ be the set of positions that are not permanent. Then, the reduced instance is $s_1|_{\mathcal{Q}}, \ldots, s_k|_{\mathcal{Q}}$. When estimating for every candidate s its Hamming distance to each remaining string s_i, the additional amount $d_{\mathrm{H}}(\hat{s}|_{\mathcal{P}}, s_i|_{\mathcal{P}})$ has to be added to the distances of the reduced strings. This is done only once at the beginning, since the Hamming distances are updated during the algorithm.

6 Computational Results

We performed our tests on a data set obtained by applying the approximate gene cluster algorithm described in [1] to five γ-proteobacteria genomes from the NCBI Genome database[4], see Tab. 1. The gene classification is based on COG[5] functional categories.

Table 1. Five γ-proteobacteria from the NCBI Genome database, used for detection of approximate gene clusters to generate biological instances of the center string problem. 'Refseq' denotes reference sequence from NCBI Genome database, 'PC' number of protein-coding genes.

Species name	Refseq	Genes	PC
Buchnera aphidicola str. APS	NC_002528	607	564
Escherichia coli str. K-12 substr. MG1655	NC_000913	4493	4149
Haemophilus influenzae Rd KW20	NC_000907	1789	1657
Pasteurella multocida subsp. multocida str. Pm70	NC_002663	2092	2015
Xylella fastidiosa 9a5c	NC_002488	2838	2766

The generation of center string instances from gene cluster predictions works as follows: Each gene cluster consists of five approximate occurrences, one on each genome, that are transformed into binary strings based on their gene composition. Since the instances generated from a single cluster are too short to evaluate the performance of our algorithms, larger instances are created by concatenation until the length L is reached. Additional strings are constructed in the same fashion, incorporating further cluster occurrences.

We created 50 instances for each combination of k and L with $k = 20, 30, 40, 50$ and $L = 250, 300, \ldots, 500$. The origin of our data, based on finding approximate gene clusters, results in many clean columns that are trivially solved. We keep only the dirty columns, representing the "hard part" of the instances. In our dataset, there were between 36.2% and 57.7% dirty columns. We stress that results in the following sections are reported for this naïve kernel. In the further evaluation we examine only the 567 instances with $d_{\text{opt}} \leq 40$ and we concentrate on the computation of center strings for $d = d_{\text{opt}}$ and $d = d_{\text{opt}} - 1$, since these are the computationally hard instances, see Fig. 3 below. For the search tree algorithms evaluated below, the search tree size grows (super-) exponentially with increasing d, hence the algorithms' running times are usually dominated by these cases.

Excluding Unsolvable Instances by Preprocessing. Our preprocessing allows us to exclude unsolvable instances more efficiently than the naïve kernel, when d is too small for a center string to exist. This is of particular interest as here search tree algorithms have to scan the complete search tree to ensure that no

[4] `http://www.ncbi.nlm.nih.gov/sites/entrez?db=genome`
[5] `http://www.ncbi.nlm.nih.gov/COG/`

solution exists. Recall that the naïve kernel tests if there exist more than kd dirty columns, in which case the instance cannot have a solution for this choice of d. Table 2 shows the number of excluded instances via preprocessing, for $d = d_{\text{opt}} - 1$. Our improved preprocessing always filters out more instances than the naïve kernel does. For different k, we can exclude between 15.9 % and 44.4 % of instances that have not been filtered by the naïve kernel. We note that for $d = d_{\text{opt}} - 2$, more than 99 % of the instances are rejected by the naïve kernel or since $d < \frac{1}{2} \max_{i,j} d_{\text{H}}(s_i, s_j)$. Clearly, no instances are rejected for $d = d_{\text{opt}}$.

Table 2. Percentage of instances excluded by preprocessing, for $d = d_{\text{opt}} - 1$.

number of sequences k	20	30	40	50
naïve kernel (%)	56.5	59.1	67.4	68.4
our preprocessing, from remaining (%)	24.3	15.9	36.4	44.4
total excluded instances (%)	67.1	65.6	79.3	82.4

Solving Trivial Positions by Preprocessing. The second advantage of our method is the computation of positions that can be trivially solved during preprocessing, see Fig. 2. The percentage of fixed positions is especially high for the important case $d = d_{\text{opt}}$. In fact, an average of 41.0 % of the positions was fixed for these instances during preprocessing. We also observe that there is no "twilight zone" of fixed positions: In 57.8 % of the instances, more than 40 % of positions were fixed; in 38.5 % the data reduction did not fix any positions; and in less than 3.7 % of the instances we observed a fixation of 0–40 % of positions.

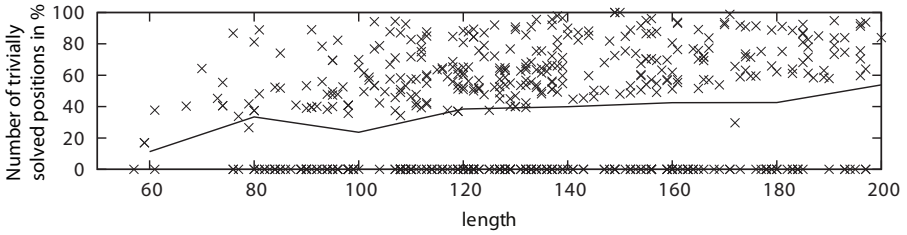

Fig. 2. Percentage of trivially solved positions in $\mathcal{P}$ for $d = d_{\text{opt}}$, plotted against the length L of the instance. Crosses represent individual instances, solid line is average percentage for intervals of width 20.

Running Times. We have implemented the algorithms of Gramm *et al.* [4], Ma and Sun [7], and the *MismatchCount* algorithm from Sect. 5, referred to as "*Gramm*", "*MaSun*" and "*MismatchCount*", respectively. These algorithms

do not include any preprocessing beyond the naïve kernel. Name suffix "*RT*" indicates that preprocessing, algorithm engineering, and the use of $\mathcal{R}$ and $\mathcal{T}$ are enabled. For the *MismatchCount* algorithm, only the information from $\mathcal{P}$ is used, denoted as name suffix "*P*".

All algorithms have been implemented in Java and compiled with the Sun Java Standard Edition compiler version 1.6. All computations were done on a quad-core 2.2 GHz AMD Opteron processor with 5 GB of main memory under the Solaris 10 operating system. The presented running times are the core running times of the algorithms and do not include I/O or removal of clean columns. We set a time limit of ten minutes per instance.

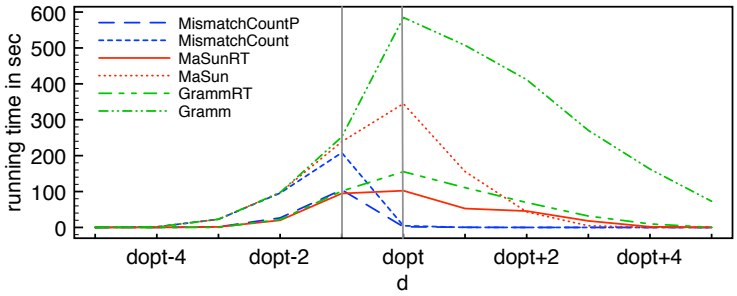

Fig. 3. Average running times for the 395 instances with $d_{\text{opt}} \leq 35$. Running times are depicted in dependency on varying d around d_{opt}.

We first show that running times of all algorithms are truly dominated by the cases $d = d_{\text{opt}} - 1$ and $d = d_{\text{opt}}$. To this end, we consider the 395 instances with $d_{\text{opt}} \leq 35$ of length $57 \leq L \leq 243$ after removing clean columns. Results are shown in Fig. 3. It is clearly visible that it is sufficient to concentrate on the two cases $d = d_{\text{opt}} - 1$ and $d = d_{\text{opt}}$. Algorithms *MaSun* and *Gramm* show large running times for both of these cases, whereas *MismatchCount* reaches its maximum running times for $d = d_{\text{opt}} - 1$ while it is faster for $d = d_{\text{opt}}$. Note that we cannot circumvent calling the algorithm with $d = d_{\text{opt}} - 1$ to ensure that d_{opt} is truly optimal.

Table 3. Percentage of instances rejected within different time limits, for $d = d_{\text{opt}} - 1$. '*MC*' denotes *MismatchCount* algorithm.

	MCP	*MC*	*MaSunRT*	*MaSun*	*GrammRT*	*Gramm*
time limit 600 s (%)	49.4	14.9	51.9	5.4	48.1	0
time limit 1 s (%)	45.6	12.4	44.0	4.1	43.6	0

We now show the dependency of running times on the parameter d_{opt}. Therefore, we pooled the instances with respect to the optimum center distance d_{opt}.

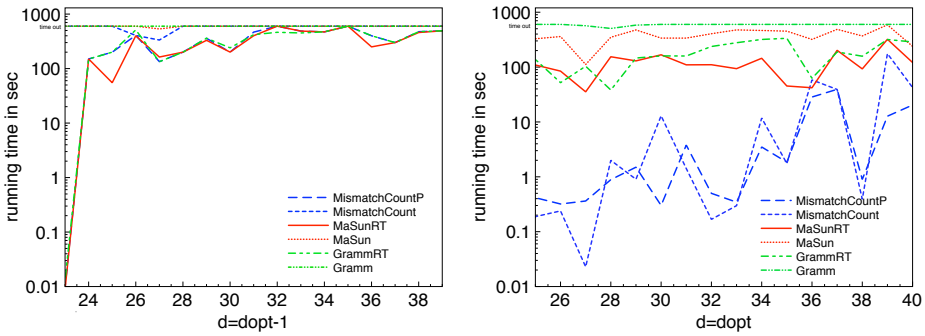

Fig. 4. Average running times for varying d_{opt}. Running times for $d = d_{\text{opt}} - 1$ (left) and $d = d_{\text{opt}}$ (right). Logarithmic scale for running times.

For $d = d_{\text{opt}} - 1$ we excluded all instances where $d < L/k$ after removing clean columns, or $d < \frac{1}{2} \max_{i,j} d_{\text{H}}(s_i, s_j)$, as these obviously have no solution, leaving us with 241 instances. In Tab. 3, we show the percentage of instances that were rejected in less than 600 s, all other instances remain undecided by the algorithm. Running times for both $d = d_{\text{opt}} - 1$ and $d = d_{\text{opt}}$ are depicted in Fig. 4. Note that the unmodified algorithms of Gramm *et al.* and Ma and Sun usually run into the time limit at 600 s, true running times are expected to be much higher. We also see that the *MismatchCount* algorithm is much faster for the case $d = d_{\text{opt}}$ than for $d = d_{\text{opt}} - 1$.

7 Conclusion

We have presented an improved preprocessing for the Center String problem. This is based on the observation that for strings with an optimal center at distance d, there usually exist many pairs of strings with distance close or equal to $2d$. Our data reduction allows us to reject more instances that do not have a valid center string, and to draw conclusions about certain positions of a center string. We show how this information can be used in the search tree algorithms of Gramm *et al.* and Ma and Sun. We have also presented the *MismatchCount* algorithm for binary alphabets. In our experimental evaluation, we could show that our data reduction is very efficient and that the *MismatchCount* algorithm outperforms the other two in practice. Our data reduction is particularly helpful for tackling the case $d = d_{\text{opt}} - 1$, where the *MismatchCount* algorithm has maximum running times, as we can exclude more instances.

Currently, the *MismatchCount* algorithm does not use information encoded in $\mathcal{R}$ and $\mathcal{T}$. We are working on a modified version of the algorithm that will allow us to approach even larger instances in reasonable running time, as it will speed up computations for the "neuralgic" case $d = d_{\text{opt}} - 1$.

Acknowledgments

This research was partially funded by DFG grant STO 431/5.

References

1. Böcker, S., Jahn, K., Mixtacki, J., Stoye, J.: Computation of median gene clusters. J. Comput. Biol. 16(8), 1085–1099 (2009)
2. Davila, J., Balla, S., Rajasekaran, S.: Fast and practical algorithms for planted (l, d) motif search. IEEE/ACM Trans. Comput. Biol. Bioinformatics 4(4), 544–552 (2007)
3. Frances, M., Litman, A.: On covering problems of codes. Theory Comput. Systems 30(2), 113–119 (1997)
4. Gramm, J., Niedermeier, R., Rossmanith, P.: Fixed-parameter algorithms for closest string and related problems. Algorithmica 37(1), 25–42 (2003)
5. Lanctot, J. K., Li, M., Ma, B., Wang, S., Zhang, L.: Distinguishing string selection problems. Information and Computation 185(1), 41–55 (2003)
6. Liu, X., He, H., Sykora, O.: Parallel genetic algorithm and parallel simulated annealing algorithm for the closest string problem. In: Li, X., Wang, S., Dong, Z. Y. (eds.) Proc. of Advanced Data Mining and Applications Conference (ADMA 2005). LNCS, vol. 3584, pp. 591–597, Springer (2005)
7. Ma, B., Sun, X.: More efficient algorithms for closest string and substring problems. SIAM J. Comput. 39(4), 1432–1443 (2009)
8. Rahmann, S., Klau, G. W.: Integer linear programming techniques for discovering approximate gene clusters. In: Mandoiu, I., Zelikovsky, A. (eds.) Bioinformatics Algorithms: Techniques and Applications, Wiley Series on Bioinformatics: Computational Techniques and Engineering, chapter 9, pp. 203–222, Wiley (2008)
9. Wang, L., Zhu, B.: Efficient algorithms for the closest string and distinguishing string selection problems. In: Deng, X., Hopcroft, J. E., Xue, J. (eds.) Proc. of Frontiers in Algorithmics Workshop (FAW 2009). LNCS, vol. 5598, pp. 261–270, Springer (2009)
10. Wang, Y., Chen, W., Li, X., Cheng, B.: Degenerated primer design to amplify the heavy chain variable region from immunoglobulin cdna. BMC Bioinformatics 7(Suppl. 4), S9 (2006)
11. Yanai, I., DeLisi, C.: The society of genes: networks of functional links between genes from comparative genomics. Genome Biol. 3(11):research0064 (2002)