Designing and Implementing Algorithms for the Closest String Problem

Shota Yuasa[1], Zhi-Zhong Chen[1(✉)], Bin Ma[2], and Lusheng Wang[3]

[1] Division of Information System Design, Tokyo Denki University, Hatoyama, Saitama 350-0394, Japan
zzchen@mail.dendai.ac.jp

[2] School of Computer Science, University of Waterloo, 200 University Ave. W, Waterloo, ON N2L3G1, Canada
binma@uwaterloo.ca

[3] Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Kowloon, Hong Kong SAR
cswangl@cityu.edu.hk

Abstract. Given a set of n strings of length L and a radius d, the closest string problem (CSP for short) asks for a string t_{sol} that is within a Hamming distance of d to each of the given strings. It is known that the problem is NP-hard and its optimization version admits a polynomial time approximation scheme (PTAS). A number of parameterized algorithms have been then developed to solve the problem when d is small. Among them, the relatively new ones have not been implemented before and their performance in practice was unknown. In this study, we implement all of them by careful engineering. For those that have been implemented before, our implementation is much faster. For some of those that have not been implemented before, our experimental results show that there exist huge gaps between their theoretical and practical performances. We also design a new parameterized algorithm for the binary case of CSP. The algorithm is deterministic and runs in $O\left(nL + n^2 d \cdot 6.16^d\right)$ time, while the previously best deterministic algorithm runs in $O\left(nL + nd^3 \cdot 6.731^d\right)$ time.

1 Introduction

An instance of the closest string problem (CSP for short) is a pair (S, d), where S is a set of strings of the same length L over an alphabet Σ and d is a nonnegative integer. The objective is to find a string t_{sol} of length L such that $d(t_{sol}, s) \leq d$ for every $s \in S$. The problem is fundamental and has been extensively studied in a variety of applications in bioinformatics, such as finding signals in DNA or protein family, and modif finding. Unfortunately, it is NP-hard [4,7].

Although CSP is NP-hard in general, we can still solve CSP exactly in reasonable amount of time via parameterized algorithms when d is small. Indeed, Gramm, Niedermeier and Rossmanith [5] designed the first parameterized algorithm that runs in $O(nL + nd \cdot d^d)$ time. Ma and Sun [8] present an algorithm whose time complexity is $O(nL + nd \cdot (16|\Sigma|)^d)$. Chen and Wang [2]

M. Xiao and F. Rosamond (Eds.): FAW 2017, LNCS 10336, pp. 79–90, 2017.
DOI: 10.1007/978-3-319-59605-1_8

improve the time complexity to $O(nL + nd \cdot 8^d)$ for binary strings and to $O(nL + nd \cdot (\sqrt{2}|\Sigma| + \sqrt[4]{8}(\sqrt{2}+1)(1+\sqrt{|\Sigma|-1}) - 2\sqrt{2})^d)$ for arbitrary alphabets Σ. Chen, Ma, and Wang [1] further improved the time complexity to $O(nL + nd^3 \cdot 6.731^d)$ for binary strings and to $O(nL + nd \cdot (1.612(|\Sigma| + \beta^2 + \beta - 2))^d)$ for arbitrary alphabets, where $\beta = \alpha^2 + 1 - 2\alpha^{-1} + \alpha^{-2}$ with $\alpha = \sqrt[3]{\sqrt{|\Sigma|-1}+1}$. In theory, this algorithm has the best time complexity among all known deterministic parameterized algorithms for CSP when $|\Sigma|$ is small (such as binary strings and DNA strings). Chen, Ma, and Wang [3] designed a randomized algorithm that runs in $O^*(9.81^d)$ and $O^*(40.1^d)$ expected time for DNA and protein strings, respectively. In particular, for binary strings, their randomized algorithm runs in $O(nL + n\sqrt{d} \cdot 5^d)$ expected time. Hence, in theory, the randomized algorithms in [3] look faster than the deterministic algorithms in [1,2].

All the aforementioned algorithms for CSP have been rigorously analyzed and hence their theoretical performance is known. On the other hand, there is another type of algorithms for CSP which solve the problem exactly but their theoretical performance has never been rigorously analyzed. For convenience, we refer to algorithms of this type as *heuristic* algorithms. Among heuristic algorithms, *TraverString* [11] and *qPMS9* [9] are known to have the best performance in practice. Indeed, they are much faster than *Provable*, which is obtained by implementing the algorithm for CSP in [2].

Ideally, we want to find an algorithm for CSP which not only has a good theoretical time-bound but also can be implemented into a program which runs faster than the other known algorithms for CSP (including the best-known heuristic algorithms, namely, *TraverString* [11] and *qPMS9* [9]). To find such an algorithm, one way is to implement the known algorithms (by careful engineering) which have good theoretical time-bounds. Unfortunately, the recent algorithms in [1,3] for CSP had not been implemented previously and hence their performance in practice was previously unknown. Moreover, although the algorithm for CSP in [2] has the best theoretical time-bound for large alphabets (such as protein strings), its simple implementation done in [2] only yields a program that is much slower than the best-known heuristic algorithms for CSP.

So, in this paper, we re-implement the algorithm for CSP in [2] by careful engineering. For convenience, we refer to this algorithm as the *2-string* algorithm as in [1]. We also carefully implement the *3-string* algorithm in [1] and the two best randomized algorithms (namely, *NonRedundantGuess* and *LargeAlphabet*) in [3], because they have good theoretical time-bounds. Our experimental results show that *NonRedundantGuess* and *LargeAlphabet* are actually much slower than the deterministic algorithms, although their theoretical time-bounds look better. Of special interest is that for large alphabets (such as protein strings), our careful implementation of the *2-string* algorithm is much faster than all the other algorithms including *TraverString* and *qPMS9*. Hence, for large alphabets, the *2-string* algorithm is an ideal algorithm because it not only has the best-known theoretical time-bound but also can be implemented into a program that outperforms all the other known programs for CSP in practice.

We also design and implement a new algorithm for the binary case of CSP. The new algorithm runs in $O(nL + n^2 d \cdot 6.16^d)$ time and is hence faster than the previously best deterministic algorithm (namely, the *3-string* algorithm). It is worth pointing out that although the best-known randomized algorithm (namely, *NonRedundantGuess*) for the binary case runs in $O(nL + n\sqrt{d} \cdot 5^d)$ expected time, its running time is random and it may fail to find a solution even if one exists. Hence, one cannot say that *NonRedundantGuess* is better than our new deterministic algorithm. Indeed, as aforementioned, it turns out that *NonRedundantGuess* is actually very slow in practice. Another drawback of randomized algorithms is that they cannot be used to enumerate all solutions. In real applications of CSP, we actually need to enumerate all solutions rather than finding a single one. We can claim that all the deterministic algorithms can be used for this purpose with their time bounds remaining intact.

Nishimura and Simjour [10] designed an algorithm for (implicitly) enumerating all solutions of a given instance (S, d). For binary strings, their algorithm runs in time $O\left(nL + nd \cdot ((n+1)(d+1))^{\lceil \log_{(1-\delta/2)} \epsilon \rceil} 5^{d(1+\epsilon+\delta)}\right)$ for any $0 < \delta \leq 0.75$ and $0 \leq \epsilon \leq 1$, and they claim that their time-bound is asymptotically better than $O(nL + nd^3 \cdot 6.731^d)$ which is achieved by the 3-string algorithm. In order for their claim to hold, $\epsilon + \delta < \log_5 6.731 - 1$ and $5^d \geq (n+1)^{\gamma}(d+1)^{\rho}$, where γ and ρ are the minimum values of the functions $\frac{\log_{(1-\delta/2)} \epsilon}{\log_5 6.731 - 1 - \epsilon - \delta}$ and $\frac{(\log_{(1-\delta/2)} \epsilon) - 2}{\log_5 6.731 - 1 - \epsilon - \delta}$ under the condition $\epsilon + \delta < \log_5 6.731 - 1$, respectively. Since $\gamma \geq 1795$ and $\rho \geq 1765$, their claim holds only when $5^d \geq (n+1)^{1795}(d+1)^{1765}$ (i.e., the parameter d is very large). In other words, their claim is false when $5^d < (n+1)^{1795}(d+1)^{1765}$ (i.e., the parameter d is not very large, which is often the case for a fixed-parameter algorithm to be meaningful).

2 Notations

Throughout this paper, Σ denotes a fixed alphabet and a string always means one over Σ. For a string s, $|s|$ denotes the length of s. For each $i \in \{1, 2, \ldots, |s|\}$, $s[i]$ denotes the letter of s at its i-th position. A *position set* of a string s is a subset of $\{1, 2, \ldots, |s|\}$. For two strings s and t of the same length, $d(s, t)$ denotes their Hamming distance.

Two strings s and t of the same length L *agree* (respectively, *differ*) *at a position* $i \in \{1, 2, \ldots, L\}$ if $s[i] = t[i]$ (respectively, $s[i] \neq t[i]$). The *position set where* s *and* t *agree* (respectively, *differ*) is the set of all positions $i \in \{1, 2, \ldots, L\}$ where s and t agree (respectively, differ). The proofs in the paper frequently use the position sets where a few strings are the same as or different from each other. The following special notations will be very useful. For two or more strings $s_1, \ldots, s_h$ of the same length, $\{s_1 \equiv s_2 \equiv \cdots \equiv s_h\}$ denotes the position set where s_i and s_j agree for all pairs (i, j) with $1 \leq i < j \leq h$, while $\{s_1 \not\equiv s_2 \not\equiv \cdots \not\equiv s_h\}$ denotes the position set where s_i and s_j differ for *all* pairs (i, j) with $1 \leq i < j \leq h$. Moreover, for a sequence $s_1, \ldots, s_h, t_1, \ldots, t_k$ of strings of the same length with $h \geq 2$ and $k \geq 1$, $\{s_1 \equiv s_2 \equiv \cdots \equiv s_h \not\equiv t_1 \not\equiv t_2 \not\equiv \cdots \not\equiv t_k\}$ denotes $\{s_1 \equiv s_2 \equiv \cdots \equiv s_h\} \cap \{s_h \not\equiv t_1 \not\equiv t_2 \not\equiv \cdots \not\equiv t_k\}$.

Another useful concept is that of a *partial string*, which is a string whose letters are only known at its certain positions. If s is a string of length L and P is a position set of s, then $s|_P$ denotes the partial string of length L such that $s|_P[i] = s[i]$ for each position $i \in P$ but $s|_P[j]$ is unknown for each position $j \in \{1, 2, \ldots, L\} \setminus P$. Let t be another string of length L. For a subset P of $\{1, 2, \ldots, L\}$, the *distance* between $s|_P$ and $t|_P$ is $|\{i \in P \mid s[i] \neq t[i]\}|$ and is denoted by $d(s|_P, t|_P)$. For two disjoint position sets P and Q of s, $s|_P + t|_Q$ denotes the partial string $r|_{P\cup Q}$ such that $r|_{P\cup Q}[i] = \begin{cases} s[i], \text{ if } i \in P; \\ t[i], \text{ if } i \in Q. \end{cases}$.

At last, when an algorithm exhaustively tries all possibilities to find the right choice, we say that the algorithm *guesses* the right choice.

3 A New Algorithm for the Binary Case

A sequence $(x_1, \ldots, x_k)$ of nonnegative integers is *superdecreasing* if for all $1 \leq i \leq k-1$, $x_i \geq \sum_{j=i+1}^{k} x_j$. For a nonnegative integer x and a positive integer k, let $\mathcal{S}_k(x)$ denote the set of all superdecreasing sequences $(x_1, \ldots, x_k)$ of k nonnegative integers with $\sum_{i=1}^{k} x_i = x$.

Lemma 1. *Let k be a positive integer, and x and X be two nonnegative integers. Consider the function f_X that maps each $(x_1, \ldots, x_k) \in \mathcal{S}_k(x)$ to $\prod_{i=1}^{k} \binom{X+x_i}{x_i}$. Then, f_X reaches its maximum value when $x_1 = \lceil \frac{x}{2} \rceil$.*

Lemma 2. *Let k, x, X, and f_X be as in Lemma 1. Then,*

$$\sum_{(x_1,\ldots,x_k)\in\mathcal{S}_k(x)} f_X(x_1, \ldots, x_k) \leq \binom{X + \lceil \frac{x}{2} \rceil}{\lceil \frac{x}{2} \rceil}\binom{X + \lfloor \frac{x}{2} \rfloor}{\lfloor \frac{x}{2} \rfloor} \cdot 2^x \cdot \left(\frac{4}{3}\right)^k .$$

Theorem 1. *The algorithm in Fig. 1 is correct and runs in $O^*\left(6.16^d\right)$ time.*

Proof. The algorithm is a simple modification of the *2-string* algorithm in [2] (see Fig. 2). The only difference between the two is the way of guessing $t_{sol}|_{A_1}$. In more details, the *2-string* algorithm guesses $t_{sol}|_{A_1}$ by guessing $\{t_{sol} \not\equiv s_0\} \cap A_1$ and further obtaining $t_{sol}[p]$ by flipping $s_0[p]$ for each $p \in \{t_{sol} \not\equiv s_0\} \cap A_1$. So, the algorithm is clearly correct.

We next analyze the time complexity. Fix an i with $2 \leq i \leq \log d$. For each $1 \leq j \leq i-1$, let $\lambda_j = d(s_i|_{A_i}, t_{sol}|_{A_i})$. Since $d(s_i, t_{sol}) \leq d$ and $\lambda \leq \lambda_1$, we have

$$\lambda + \sum_{j=2}^{i-1} \lambda_j + |A_i| - \delta_i + \sum_{j=i+1}^{\log d} \delta_j \leq d. \tag{1}$$

Moreover, since $d(s_0, s_1)$ is maximized over all pairs of strings in S, $d(s_i, s_0) + d(s_i, s_1) \leq 2d(s_0, s_1) = 2|A_1|$. Summing up the contribution of the positions in

```
Input: An instance $\langle S, d\rangle$ of the binary case of CSP.
Output: A solution to $\langle S, d\rangle$ if one exists, or NULL otherwise.
1. Select a pair of strings from $S$ whose Hamming distance is maximized over
   all pairs of strings in $S$. Without loss of generality, assume that $(s_0, s_1)$ is
   such a pair.
2. Let $P = A_1 = \{s_0 \not\equiv s_1\}$.
3. Guess $s_h \in S$ (by trying every string in $S$) such that $d(s_h|_P, t_{sol}|_P)$ is
   minimized over all strings in $S$, where $t_{sol}$ is a fixed (unknown) solution
   to $\langle S, d\rangle$. Let $\lambda = d(s_h|_P, t_{sol}|_P)$.
4. Guess $t_{sol}|_P$ (by trying all choices of $\lambda$ positions in $P$ and flipping the
   letters of $s_h$ in the $\lambda$ chosen positions). Let $t = t_{sol}|_P + s_0|_{\{1,2,\ldots,|s_0|\}\setminus P}$
   and $b = d - d(s_0|_P, t|_P)$.
5. For $i = 2, 3, \ldots, \log d$ (in this order), perform the following steps:
    5.1. If every string in $S$ is within a Hamming distance at most $d$ from $t$,
         then output $t$ and halt.
    5.2. Find a string in $S$ whose Hamming distance from $t$ is at least $d + 1$.
         Without loss of generality, assume that $s_i$ is such a string. Let $A_i =$
         $\{s_0 \equiv s_1 \equiv \cdots \equiv s_{i-1} \not\equiv s_i\}$.
    5.3. Let $\ell = d(t, s_i) - d$ and $R = \{s_i \not\equiv t\} \setminus P$.
    5.4. If $\ell > \min\{b, |R|\}$, then return NULL.
    5.5. Guess $\delta_i = d(t|_{A_i}, t_{sol}|_{A_i})$ (by trying all integers in $\{\ell, \ell+1, \ldots, b\}$).
    5.6. Guess $t_{sol}|_{A_i}$ and modify $t$ accordingly (by trying all choices of $\delta_i$
         positions in $R$ and flipping the letters of $t$ in the chosen $\delta_i$ positions).
    5.7. Update $b = \min\{b - \delta_i, \delta_i - \ell\}$ and $P = P \cup R$.
6. Return NULL.
```

Fig. 1. The new algorithm for the binary case

```
Input: An instance $\langle \mathcal{S}, d, t, P, b\rangle$ of ECSP.
Output: A solution to $\langle \mathcal{S}, d, t, P, b\rangle$ if one exists, or NULL otherwise.
1. If there is no $s \in \mathcal{S}$ with $d(t, s) > d$, then output $t$ and halt.
2. If $d = b$, then find a string $s \in \mathcal{S}$ such that $d(t, s)$ is maximized over all
   strings in $\mathcal{S}$; otherwise, find an arbitrary string $s \in \mathcal{S}$ such that $d(t, s) > d$.
3. Let $\ell = d(t, s) - d$ and $R = \{s \not\equiv t\} \setminus P$.
4. If $\ell > \min\{b, |R|\}$, then return NULL.
5. Guess $t_{sol}|_R$ by performing the following steps, where $t_{sol}$ is a fixed (un-
   known) solution to $\langle \mathcal{S}, d, t, P, b\rangle$.
    5.1 Guess two sets $X$ and $Y$ such that $Y \subseteq X \subseteq R$, $\ell \le |X| \le b$, and
        $|Y| \le |X| - \ell$.
    5.2 For each $i \in Y$, guess a letter $z_i$ different from both $s[i]$ and $t[i]$. Let
        the partial string $\hat{s}|_Y$ be such that $\hat{s}|_Y[i] = z_i$ for all $i \in Y$.
    5.3 Let $t_{sol}|_R = \hat{s}|_Y + s|_{X \setminus Y} + t|_{R \setminus X}$.
6. Let $t' = t_{sol}|_R + t|_{\{1,2,\ldots,|t|\}\setminus R}$ and $b' = \min\{b - |X|, |X| - \ell - |Y|\}$. (Com-
   ment: $d(t, t') = |X|$.)
7. Solve $\langle \mathcal{S} \setminus \{s\}, d, t', P \cup R, b'\rangle$ recursively.
8. Return NULL.
```

Fig. 2. The 2-string algorithm given in [2]

each A_j $(1 \leq j \leq i)$ towards $d(s_i, s_0) + d(s_i, s_1)$, we have $|A_1| + \sum_{j=2}^{i-1} 2|\delta_j - \lambda_j| + 2|A_i| \leq d(s_i, s_0) + d(s_i, s_1) \leq 2|A_1|$. Thus,

$$\sum_{j=2}^{i-1}(\delta_j - \lambda_j) + |A_i| \leq \sum_{j=2}^{i-1} |\delta_j - \lambda_j| + |A_i| \leq \frac{|A_1|}{2}. \tag{2}$$

Adding up Eqs. 1 and 2, we have $\lambda + \sum_{j=2}^{i-1} \delta_j + 2|A_i| - \delta_i + \sum_{j=i+1}^{\log d} \delta_j \leq \frac{|A_1|}{2} + d$. Let $d' = \sum_{j=2}^{\log d} \delta_j$. Then, $\lambda + 2|A_i| - 2\delta_i + d' \leq \frac{|A_1|}{2} + d$. So, $|A_i| \leq \frac{|A_1|}{4} + \frac{d}{2} - \frac{d'}{2} - \frac{\lambda}{2} + \delta_i$. Since $|A_1| + 2d' = d(s_0, t_{sol}) + d(s_1, t_{sol}) \leq 2d$, we now have $|A_1| \leq 2(d - d')$ and hence $|A_i| \leq d - d' - \frac{\lambda}{2} + \delta_i$.

Let $X = d - d' - \frac{\lambda}{2}$ and $k = \log d$. By Lemma 2, the exponential factor in the time complexity of the algorithm is bounded from above by

$$\begin{aligned}
&\binom{|A_1|}{\lambda} \sum_{(\delta_2,\ldots,\delta_k)\in\mathcal{S}_{k-1}(d')} \prod_{i=2}^{k} \binom{X+\delta_i}{\delta_i} \\
&\leq \binom{|A_1|}{\lambda}\binom{X+\lceil\frac{d'}{2}\rceil}{\lceil\frac{d'}{2}\rceil}\binom{X+\lfloor\frac{d'}{2}\rfloor}{\lfloor\frac{d'}{2}\rfloor} \cdot 2^{d'} \cdot \left(\frac{4}{3}\right)^{k-1} \\
&\leq d \cdot \binom{2(d-d')}{\lambda}\binom{X+\lceil\frac{d'}{2}\rceil}{\lceil\frac{d'}{2}\rceil}\binom{X+\lfloor\frac{d'}{2}\rfloor}{\lfloor\frac{d'}{2}\rfloor} \cdot 2^{d'} \\
&\leq d \cdot \binom{2(d-d')}{\lambda}\binom{d-\frac{\lambda}{2}-\lfloor\frac{d'}{2}\rfloor}{\lceil\frac{d'}{2}\rceil}\binom{d-\frac{\lambda}{2}-\lceil\frac{d'}{2}\rceil}{\lfloor\frac{d'}{2}\rfloor} \cdot 2^{d'}.
\end{aligned}$$

Now, using Stirling's formula, the exponential factor in the time complexity of the algorithm is bounded from above by

$$\frac{(2(d-d'))^{2(d-d')}}{\lambda^\lambda \cdot (2(d-d')-\lambda)^{2(d-d')-\lambda}} \cdot \left(\frac{\left(d-\frac{\lambda}{2}-\frac{d'}{2}\right)^{d-\frac{\lambda}{2}-\frac{d'}{2}}}{\left(\frac{d'}{2}\right)^{\frac{d'}{2}} \cdot \left(d-\frac{\lambda}{2}-d'\right)^{d-\frac{\lambda}{2}-d'}}\right)^2 \cdot 2^{d'}.$$

Let $a = \frac{\lambda}{d}$ and $b = \frac{d'}{d}$. Then, the above bound becomes c^d, where

$$\begin{aligned}
c &= \frac{(2(1-b))^{2(1-b)}}{a^a \cdot (2(1-b)-a)^{2(1-b)-a}} \cdot \left(\frac{\left(1-\frac{a}{2}-\frac{b}{2}\right)^{1-\frac{a}{2}-\frac{b}{2}}}{\left(\frac{b}{2}\right)^{\frac{b}{2}} \cdot \left(1-\frac{a}{2}-b\right)^{1-\frac{a}{2}-b}}\right)^2 \cdot 2^b \\
&= \frac{2^{2-b}(1-b)^{2(1-b)}(2-a-b)^{2-a-b}}{a^a b^b (2-a-2b)^{2(2-a-2b)}}
\end{aligned}$$

Note that $a + b \leq 1$ for $\lambda + d' \leq d(s_0, t_{sol}) \leq d$. So, by numerical calculation, one can verify that $c \leq 6.16$ no matter what a and b are (as long as $a + b \leq 1$). Thus, the time complexity of the algorithm is $O^*\left(6.16^d\right)$.

4 Previous Algorithms

A number of parameterized algorithms whose time complexity has been rigorously analyzed have not been implemented. One objective of this paper is to implement the recent algorithms and see their performance in practice. We only sketch the *2-string* algorithm for CSP below.

The 2-String Algorithm has actually been implemented in [2]. However, as demonstrated in [11], the implementation in [2] yields a program (called *Provable*) which runs much slower than *TraverString*. In this paper, we give a different implementation of the *2-string* algorithm.

The algorithm is actually designed for a more general problem, called the *extended closest string problem* (ECSP for short). An instance of ECSP is a quintuple $(\mathcal{S}, d, t, P, b)$, where $\mathcal{S}$ is a set of strings of the same length L, t is a string of length L, d is a positive integer, P is a subset of $\{1, 2, \ldots, L\}$, and b is a nonnegative integer. The objective is to find a string t_{sol} of length L such that $t_{sol}|_P = t|_P$, $d(t_{sol}, t) \le b$, and $\forall s \in \mathcal{S},\ d(t_{sol}, s) \le d$. Intuitively speaking, we want to transform t into a solution t_{sol} by modifying $\le b$ positions of t outside P.

To solve a given instance (S, d) of CSP, it suffices to solve the instance $(S \setminus \{t\}, d, t, \emptyset, d)$ of ECSP, where t is an arbitrary string in S. The algorithm for ECSP is detailed in Fig. 2.

5 Implementing the Algorithms

Because each step of *NonRedundantGuess* and *LargeAlphabet* is very simple, their implementation is rather straightforward. So, we only describe the main ideas used in our implementation of the deterministic algorithms below.

Basically, each of the deterministic algorithms maintains a string t, a set P of fixed positions of t, and a bound b, and tries to transform t into a solution by selecting and modifying at most b *unfixed* positions (i.e., positions outside P) of t. It is possible that there is no way to transform t into a solution by modifying at most b unfixed positions of t. We want to efficiently decide if this is really the case. The next lemma can be used for this purpose.

Lemma 3 [11]**.** *Let u, v, and w be three strings of the same length K. Then, there is a string t_{sol} of length K such that $d(t_{sol}, u) \le d_u$, $d(t_{sol}, v) \le d_v$, and $d(t_{sol}, w) \le d_w$ if and only if the following conditions hold:*

1. $d_u \ge 0$, $d_v \ge 0$, *and* $d_w \ge 0$.
2. $d(u, v) \le d_u + d_v$, $d(u, w) \le d_u + d_w$, *and* $d(v, w) \le d_v + d_w$.
3. $d_u + d_v + d_w \ge |\{u \equiv v \not\equiv w\}| + |\{u \equiv w \not\equiv v\}| + |\{v \equiv w \not\equiv u\}| + 2|\{u \not\equiv v \not\equiv w\}|$.

As an example, we explain how to use Lemma 3 to prune a search tree for the *2-string algorithm*. Consider a call of the algorithm on input $\langle \mathcal{S}, d, t, P, b\rangle$. For convenience, let $Q = \{1, \ldots, L\} \setminus P$ and $K = L - |P|$. For each $u \in \mathcal{S}$, let $d_u = d - d(t|_P, u|_P)$. Recall that t has been obtained by modifying the fixed positions of some $\tilde{u} \in \mathcal{S}$. Hence, $b = d_{\tilde{u}}$. What the algorithm needs to do is to transform $t|_Q$ into a string t_{sol} of length K such that $d(t_{sol}, u|_Q) \leq d_u$ for all $u \in \mathcal{S}$. To decide if such a transformation exists, we want to check if Conditions 1 through 3 in Lemma 3 hold for every triple $\{u, v, w\}$ of strings in $\mathcal{S}$. However, there are $\Omega(|\mathcal{S}|^3)$ such triples and hence it is time-consuming and wasteful to do the checking for all of them. So, in our implementation, we only check those triples (u, v, w) such that $u = \tilde{u}$, v is the string s selected in Step 2 of the algorithm, and $w \in \mathcal{S} \setminus \{u, v\}$. If the checking fails for at least one such triple, Lemma 3 ensures that t cannot be transformed into a solution by selecting and modifying at most b unfixed positions of t.

5.1 Enumerating Subsets of Unfixed Positions

In Step 5 of the *2-string* algorithm, we need to decide which unfixed positions of t should be selected and further how to modify them. The other deterministic algorithms have the same issue. We only explain how to deal with this issue for the *2-string* algorithm below; the same can be done for the other algorithms.

In Step 5, we need to enumerate all subsets X of R with $\ell \leq |X| \leq b$. Then, for each enumerated X, we need to enumerate all subsets Y of X with $|Y| \leq |X| - \ell$. Furthermore, for each enumerated Y, we need to enumerate all valid ways of modifying the positions of t in Y. Roughly speaking, in the implementation of the *2-string* algorithm done in [2], only after enumerating X and Y, we start to enumerate all valid ways of modifying the positions of t in Y. So, in the implementation in [2], every possible combination of X and Y will be enumerated because only after modifying one or more positions of t in X, we can decide if it is unnecessary to make a certain recursive call on the modified t (i.e., if it is possible to prune a certain branch of the search tree). This seems to be the main reason whey the implementation in [2] yields a slow program for CSP.

To get over the above-mentioned drawback of the implementation in [2], our idea is to enumerate the elements of X and Y one by one and at the same time enumerate all possible ways of modifying each position in Y. In more details, we scan the positions in R one by one (in any order). When scanning a $p \in R$, we need to make two choices depending on whether to include p in X or not. If we decide to exclude p from X, then we proceed to the next position in R. Otherwise, we need to make two choices depending on whether to include p in Y or not. If we decide to exclude p from Y, then we modify t by changing $t[p]$ to $s[p]$ and then use Lemma 3 to check if the modified t can be further transformed into a solution by modifying at most $b - 1$ positions outside $P \cup \{p\}$. If the checking yields a "no" answer, then we can quit scanning the remaining positions in R and hence prune a certain branch of the search tree at an early stage. Similarly, if we decide to include p in Y, then we need to make $|\Sigma| - 2$ choices depending on to which letter we should change $t[p]$. For each of the choices, after modifying position p of t, we use Lemma 3 to check if the modified t can be further transformed into a solution by modifying at most $b - 1$ positions outside $P \cup \{p\}$. If the checking yields a "no" answer, then we can quit scanning the remaining positions in R and hence prune a certain branch of the search tree at an early stage.

```
5. Let $p_1, p_2, \ldots, p_q$ be the positions in $R$. Initialize $X = \emptyset$ and $Y = \emptyset$. For $i = 1, 2, \ldots, q$ (in this order), guess $t_{sol}[p_i]$ by performing Steps 5.1 through 5.3:
  5.1 If $|X| + q - i + 1 = \ell$, then add $p_i$ to $X$, set $t_{sol}[p_i] = s[p_i]$, and proceed to the next $i$ (without performing Steps 5.2 and 5.3).
  5.2 If $|X| + q - i + 1 > \ell$, perform the following steps:
    5.2.1 Guess whether $p_i \in X$ or not.
    5.2.2 If the guess is $p_i \in X$, then add $p_i$ to $X$ and perform Steps 5.2.2.1 through 5.2.2.4:
      5.2.2.1 If $|X| + q - i - |Y| = \ell$, then set $t_{sol}[p_i] = s[p_i]$ and proceed to the next $i$ (without performing Steps 5.2.2.2 through 5.2.2.4 and 5.3).
      5.2.2.2 Guess whether $p_i \in Y$ or not.
      5.2.2.3 If the guess is $p_i \notin Y$, then set $t_{sol}[p_i] = s[p_i]$; otherwise, add $p_i$ to $Y$ and guess $t_{sol}[p_i]$ from $\Sigma \setminus \{t[p_i], s[p_i]\}$.
      5.2.2.4 Proceed to the next $i$ (without performing Step 5.3).
  5.3 Set $t_{sol}[p_i] = t[p_i]$.
```

Fig. 3. Modifying Step 5 of the 2-string algorithm in Figure 2

More formally, we modify Step 5 in the *2-string* algorithm as shown in Fig. 3. A crucial but missing detail in the modified Step 5 is that before we decide to set $t_{sol}[p_i]$ to be a certain letter $a \neq t[p_i]$ in Step 5.1, 5.2.2.1, or 5.2.2.3, we actually use Lemma 3 to check if setting $t_{sol}[p_i] = a$ can lead to a solution as follows. First, we obtain a string u from t by changing $t[p_i]$ to a and changing $t[p_j]$ to $t_{sol}[p_j]$ for all $j \in \{1, 2, \ldots, i-1\}$. We then compute $d_u = b - d(t, u)$, set $v = s$, and compute $d_v = d - d(v|_{P \cup \{p_1,\ldots,p_i\}}, u|_{P \cup \{p_1,\ldots,p_i\}})$. For all $w \in \mathcal{S}$, we further compute $d_w = d - d(w|_{P \cup \{p_1,\ldots,p_i\}}, u|_{P \cup \{p_1,\ldots,p_i\}})$ and now check if u, v, w, d_u, d_v, and d_w altogether satisfy Conditions 1 through 3 in Lemma 3. If this checking fails for at least one w, then we can conclude that setting $t_{sol}[p_i] = a$ cannot lead to a solution, and hence we can quit setting $t_{sol}[p_i] = a$ (i.e., can prune a certain branch of the search tree).

5.2 Sorting the Input Strings

Again, we explain the idea by using the *2-string* algorithm as an example. The idea also applies to the other deterministic algorithms. As mentioned in the above (immediately before Sect. 5.1), when we apply Lemma 3, we only check those triples (u, v, w) such that u is the input string from which the current t has been obtained, v is the string selected in Step 2 of the algorithm, and $w \in \mathcal{S} \setminus \{u, v\}$. So, there are $|\mathcal{S}| - 2$ choices for w and we can try the choices in any order. As one can expect, different orders lead to different speeds. In our implementation, we try the choices for w in descending order of the Hamming distance of w from t. Intuitively speaking, this order seems to enable us to find out that t cannot be transformed into a solution at an earlier stage than other orders.

5.3 On Implementing the Algorithm in Sect. 3

In Step 3 of our new algorithm, we need to guess an input string s_h such that $d(s_h|_P, t_{sol}|_P)$ is minimized over all input strings, where t_{sol} is a fixed solution. To guess s_h, a simple way is to try all input strings (in any order). When trying a particular s_h, we use Lemma 4 to cut unnecessary branches of the search tree.

Lemma 4. *Let P be as in Step 2 of the algorithm. Further let t_{sol} and s_h be as in Step 3 of the algorithm. Then, for every $s_j \in \mathcal{S}$, $d(s_h|_D, t_{sol}|_D) \leq \frac{|D|}{2}$, where $D = \{s_h|_P \not\equiv s_j|_P\}$.*

To use Lemma 4, we first compute $m_j = \frac{d(s_h|_P, s_j|_P)}{2}$ for each $s_j \in \mathcal{S}$ in Step 3. Later in Step 4, we scan the positions in P one by one (in any order). When scanning a position $p \in P$, we need to make two recursive calls – one of them corresponds to flipping the letter of s_h at position p while the other corresponds to keeping the letter of s_h at position p intact. Before making each of the calls, we decrease m_j by 1 for all $s_j \in \mathcal{S}$ such that the letter of s_h at position p has become different from $s_j[p]$ after scanning p; if m_j becomes negative for some $s_j \in \mathcal{S}$, then we know that it is unnecessary to make the recursive call. In this way, we are able to cut certain unnecessary branches of the search tree.

6 Results and Discussion

We have implemented the new algorithm in Sect. 3 and the previously known algorithms reviewed in Sect. 4. As the result, we have obtained a program (written in C) for each of the algorithms. We not only compare these programs against each other but also include *TraverString* and *qPMS9* in the comparison. We do not compare with the algorithm in [6], because its code is not available. The machine on which we ran the programs is an Intel Core i7-975 (3.33GHz, 6MB Cache, 6GB RAM in 64bit mode) Linux PC.

As in previous studies, we generate random instances of CSP as input to the programs. In the generation of instances, we fix $L = 600$ and $n = 20$ but varies d and $|\Sigma|$. As usual, the choices for $|\Sigma|$ we employ in our test are 2 (binary strings), 4 (DNA strings), and 20 (protein strings). Choosing d is less obvious. Clearly, the larger d is, the longer the programs run. To distinguish the programs from each other in terms of speed, we consider the following five ranges of d: (1) $10 \leq d \leq 15$, (2) $28 \leq d \leq 33$, (3) $80 \leq d \leq 85$, (4) $82 \leq d \leq 87$, and (5) $d \in \{89, 92, 95, 98, 101\}$. The choice of these ranges is based on the expectation that some of the programs can be slow even if d is small, while the others can show their significant difference in running time only if d is modest or even large.

Since some of the programs can be very slow for certain d, we set a time limit of 5 hours on each run of each program in our test. For each setting of parameters (e.g., $(L, n, |\Sigma|, d) = (600, 20, 4, 15)$), we generate five instances, pass them to the programs, and require each program to find *all* solutions for each instance. We will summarize our experimental results in several tables. If a program solves all

Table 1. Comparison of the programs for $|\Sigma| = 4$, 20, 2

	Program	$d = 10$	$d = 11$	$d = 12$	$d = 13$	$d = 14$	$d = 15$
	NonRedundantGuess	5.6m	48m	TL	TL	TL	TL
	LargeGuess	< 0.05s	< 0.05s	5.2s	23s	2.1m	8.7m
	2-string	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s
D	*3-string*	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s
	TraverString	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s
N	*qPMS9*	5.9s	5.9s	6.0s	5.9s	5.9s	5.9s
	Program	$d = 80$	$d = 81$	$d = 82$	$d = 83$	$d = 84$	$d = 85$
A	*2-string*	10s	1.6m	37s	8m	4.8m	6.3m
	3-string	11s	1.7m	41s	8.4m	5.4m	7m
	TraverString	5.7m	TL	11m	TL	TL	TL
	qPMS9	5.9s	5.9s	5.9s	5.9s	5.9s	5.9s
	Program	$d = 10$	$d = 11$	$d = 12$	$d = 13$	$d = 14$	$d = 15$
	NonRedundantGuess	TL	TL	TL	TL	TL	TL
	LargeGuess	2.6s	9.6s	13s	54s	6.7m	29m
	2-string	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s
	3-string	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s
p	*TraverString*	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s
r	*qPMS9*	12s	12s	12s	12s	12s	12s
o	Program	$d = 82$	$d = 83$	$d = 84$	$d = 85$	$d = 86$	$d = 87$
t	*2-string*	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s
e	*3-string*	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s
i	*TraverString*	< 0.05s	< 0.05s	< 0.05s	< 0.05s	3.6s	< 0.05s
n	*qPMS9*	10m	53m	21m	TL	1.2h	TL
	Program	$d = 89$	$d = 92$	$d = 95$	$d = 98$	$d = 101$	
	2-string	0.2s	1.6s	2.0s	2.2s	22s	
	3-string	0.2s	2.2s	2.1s	2.1s	25s	
	TraverString	1.6s	1.5m	2.0m	2.2m	TL	
b	Program	$d = 28$	$d = 29$	$d = 30$	$d = 31$	$d = 32$	$d = 33$
i	New	11m	19m	32m	1h	1.9h	TL
n	*2-string*	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s
a	*3-string*	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s
r	*TraverString*	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s	< 0.05s
y	*qPMS9*	3.1s	3.1s	3.1s	3.1s	3.1s	3.1s

of the five instances within the time limit, we calculate the average running time and show it as is in a table; otherwise, we put a TL symbol in the corresponding cell of the table, where TL stands for "time limit".

Table 1 shows the comparison of the programs for DNA, protein, or binary strings. As seen from the table, *NonRedundantGuess* and *LargeGuess* are slower than the other algorithms even though *NonRedundantGuess* and *LargeGuess* have better theoretical time-bounds. In the table, New means our new algorithm in Sect. 3. We exclude the experimental results for *NonRedundantGuess* and *LargeGuess* from the table, because both failed to solve a single instance within the time limit. As seen from the table, our new algorithm in Sect. 3 is much faster than *NonRedundantGuess* and *LargeGuess*, but is much slower than the other deterministic algorithms. The reason why the new algorithm runs slower seems to be that the pruning inequalities in Lemma 3 are much less effective in cutting unnecessary branches of a search tree for our new algorithm.

Based on our above experimental results, we conclude that the *2-string* and the *3-string* algorithms have little difference in running time and they are the stablest algorithms among the tested algorithms. In particular, for large alphabets (such as protein strings), the *2-string* algorithm is an ideal algorithm because it not only has the best-known theoretical time-bound but also can be implemented into a program that outperforms the other known programs for CSP in practice.

References

1. Chen, Z.-Z., Ma, B., Wang, L.: A three-string approach to the closest string problem. J. Comput. Syst. Sci. **78**, 164–178 (2012)
2. Chen, Z.-Z., Wang, L.: Fast exact algorithms for the closest string and substring problems with application to the planted (ℓ, d)-motif model. IEEE/ACM Trans. Comput. Biol. Bioinf. **8**(5), 1400–1410 (2011)
3. Chen, Z.-Z., Ma, B., Wang, L.: Randomized fixed-parameter algorithms for the closest string problem. Algorithmica **74**, 466–484 (2016)
4. Frances, M., Litman, A.: On covering problems of codes. Theoret. Comput. Sci. **30**, 113–119 (1997)
5. Gramm, J., Niedermeier, R., Rossmanith, P.: Fixed-parameter algorithms for closest string and related problems. Algorithmica **37**, 25–42 (2003)
6. Hufsky, F., Kuchenbecker, L., Jahn, K., Stoye, J., Böcker, S.: Swiftly computing center strings. BMC Bioinform. **12**, 106 (2011)
7. Lanctot, K., Li, M., Ma, B., Wang, S., Zhang, L.: Distinguishing string search problems. Inform. Comput. **185**, 41–55 (2003)
8. Ma, B., Sun, X.: More efficient algorithms for closest string and substring problems. SIAM J. Comput. **39**(4), 1432–1443 (2010)
9. Nicolae, M., Rajasekaran, S.: qPMS9: an efficient algorithm for quorum planted motif search. Nat. Sci. Rep. **5** (2015)
10. Nishimura, N., Simjour, N.: Enumerating neighbour and closest strings. In: Thilikos, D.M., Woeginger, G.J. (eds.) IPEC 2012. LNCS, vol. 7535, pp. 252–263. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33293-7_24
11. Tanaka, S.: Improved exact enumerative algorithms for the planted (l, d)-motif search problem (2014). IEEE/ACM Trans. Comput. Biol. Bioinf. **11**, 361–374 (2014)