A Fixed-Parameter Algorithm for String-to-String Correction*

Faisal N. Abu-Khzam[1], Henning Fernau[2], Michael A. Langston[3], Serena Lee-Cultura[4] and Ulrike Stege[4]

[1] Department of Computer Science and Mathematics, Lebanese American University, Lebanon
Email: faisal.abukhzam@lau.edu.lb
[2] Universität Trier, FB IV—Abteilung Informatik, D-54286 Trier, Germany
Email: fernau@uni-trier.de
[3] Department of Electrical Engineering and Computer Science, University of Tennessee, USA
Email: langston@eecs.utk.edu
[4] Department of Computer Science, University of Victoria, Canada
Email: {sleecult,ustege}@uvic.ca

Abstract

In the String-to-String Correction problem we are given two strings x and y and a positive integer k, and are asked whether it is possible to transform y into x with at most k single character deletions and adjacent character exchanges. We present a first simple fixed-parameter algorithm for String-to-String Correction that runs in $\mathcal{O}^*(2^k)$. Moreover, we present a search tree algorithm that exhibits a novel technique of bookkeeping called *charging*, which leads to an improved algorithm whose running time is in $\mathcal{O}^*(1.62^k)$.

1 Introduction

In computational biology, information retrieval, linguistics, and a variety of other application areas, efficient algorithms for string editing problems are in high demand. One major type of string editing problems is what is called *string sorting* or *string correction*.

String sorting is needed, for example, in computational biology when determining the combinatorial genome rearrangement distance of two given genomes (see [15] for a recent survey on algorithmic approaches). Here, a given string (permutation) of numbers is to be sorted in a minimum number of operations, which often include more complex operations like substring inversions and adjacent block transpositions. The problem also finds application in text mining, mainly because transpositions are common typing errors (see [17] for a survey, and [5, 18] for other relevant applications).

Although a lot of attention was given to string sorting problems in the context of genome rearrangements, by far not all relevant models are studied or sufficiently solved. While string sorting by inversions (or reversals) only is solvable in polynomial time for signed permutations [1, 12, 13], the problem is $\mathcal{NP}$-complete for unsigned permutations [4] and cannot be approximated within 1.0008 unless $\mathcal{P} = \mathcal{NP}$ [3]. Approaches for solving this problem include an exact LP formulation [14] and a 1.375-approximation algorithm [2]. The classical complexity of sorting (unsigned) permutations by (adjacent block) transpositions is open. A 1.375-algorithm is the best known solution [7]. Further, a 2-approximation algorithm is known when sorting unsigned permutations by reversals and transpositions [11].

In this paper we study a variant of the string sorting problem where deletions and single character interchanges (a special case of adjacent block transpositions) are the allowed operations.

Before introducing the problem, we note that string sorting can equivalently be viewed as the string correction problem: given two strings, decide on the number of operations necessary to transform the second string into the first. The string correction problem is solvable in polynomial time when the allowable operations are (1) simple edits (i.e., single character deletions, single character insertions, single character replacements, and adjacent character exchanges [16]), (2) only single character deletions [19], and (3) only adjacent character exchanges [16]. The problem turns out to be $\mathcal{NP}$-complete, however, when operations are restricted to single character deletions and adjacent character exchanges [19]. It is this classical variant that we study here.

String-to-String Correction [10]
Given: an alphabet Σ, two strings $x, y \in \Sigma^*$
Parameter: an integer k
Question: Is there a way to transform y into x by a sequence of at most k *edit operations*, where such an operation can be only a single character deletion or an adjacent character exchange (also known as transposition or *swap*) within y?

We are not aware of any prior work on parameterized or approximation algorithms for this specific problem.

Our Framework: Parameterized Complexity. A *parameterized problem* P is a subset of $\Sigma^* \times \mathbb{N}$, where Σ is a fixed alphabet and $\mathbb{N}$ is the set of all non-negative integers. Therefore, each instance of the parameterized problem P is a pair (I, k), where the second component k is called the *parameter*. The language $L(P)$ is the set of all YES-instances of P. We say that the parameterized problem P is *fixed-parameter tractable* [6] if there is an algorithm that decides whether an input (I, k) is a member of $L(P)$ in time $f(k)|I|^c$, where c is a fixed constant and $f(k)$ is a function independent of the overall input length $|I|$. We can also write $\mathcal{O}^*(f(k))$ for this run-time bound. Notice that parameterized algorithms are particularly useful when the parameter values are expected to be

*Initial work on this research was done when some of the authors met at the Dagstuhl Seminar 08431 'Moderately Exponential Time Algorithms' in October 2008; this work was partially supported by an NSERC operating grant awarded to Ulrike Stege.

small. This can be generally assumed for problems where the parameter can be seen as an error count, as with typing errors in our particular problem.

Our Contributions.
Foremost, we present a fixed-parameter algorithm running in time $\mathcal{O}^*(1.62^k)$ for String-to-String Correction. We also present a related $\mathcal{FPT}$ result (namely, an $\mathcal{O}^*(2.4143^k)$ algorithm) for the similar string edit problem, namely for the problem variant where insertions, swaps and deletions are allowed when transforming one string into another. In doing so, we introduce a novel paradigm called *charging* that allows to improve on the estimates of the upper-bounds of search tree sizes. Within our problem, we know in certain situations that a symbol a must move at least ℓ steps to the right by swapping operations. We cannot perform these swaps right now, because it could be better to first move some symbols further to the right that are currently to the right of a. Therefore, we annotate a by replacing it with (a, ℓ). We call this specific replacement operation *charging*. Since we know that we have to perform these swaps later on, we can reduce the parameter budget k already at this stage by ℓ. We have not seen this specific technique elsewhere and believe it can be useful for other string problems, as well. We note that similar but much simpler idea is the idea of *bonus points* introduced in [8]. However, the parameter budget reductions in [8] were due to later polynomial-time phases of the algorithm. Here, the charge is used up by reduction rules that can trigger within each branching step.

The Presentation of the Paper.
The paper is structured as follows. Section 2 discusses some basic properties of String-to-String Correction. In Section 3 we prove that the problem is fixed-parameter tractable showing an algorithm with a running time $\mathcal{O}^*(2^k)$. In Section 4 we describe a refined fixed-parameter algorithm running in time $\mathcal{O}^*(1.62^k)$. We conclude the paper with several discussions and comments.

2 Basic Properties and Terminology

We observe that any instance of String-to-String Correction with $|x| > |y|$ is a NO-instance. Hence, a feasible solution to a given instance gives an injective mapping $M : \{1, \ldots, |x|\} \to \{1, \ldots, |y|\}$ explaining which positions of x can be found where in string y: Assuming $x = x_1 \cdots x_n$ and $y = y_1 \ldots y_m$, where $n \leq m$ and $x_i, y_j \in \Sigma$ for all $1 \leq i \leq n$, $1 \leq j \leq m$, M satisfies $\forall 1 \leq i \leq n : y_{M(i)} = x_i$ with $M(i) \neq M(j)$ for $i \neq j$.

We remark that it is sufficient to compute such a mapping M corresponding to a feasible solution as, once given the mapping, we can delete every character in y which was not mapped to by M from x. Then, using the result in [16], we can solve String-to-String Correction for x and the shortened y for swaps only in polynomial time.

Let $\phi(y, a)$ give the index of the **f**irst occurrence of character $a \in \Sigma$ in y. For $|y| > 0$, let $\tau(y)$, the ***t**ail*, denote the string obtained from y by deleting its first character. Let $\sigma(y, i)$ denote the string obtained from y by **s**wapping the two characters y_i and y_{i+1}. Let $\delta(y, i)$ denote the string resulting from y by **d**eleting character y_i. Observe that $|\sigma(y, i)| = |y|$ and $|\delta(y, i)| = |y| - 1$. A sequence $\omega = \omega_1 \cdots \omega_k$ of edit operations yields the transformed string $\omega(y) = \omega_k(\ldots(\omega_1(y))\ldots)$. Then clearly $|y| \geq |\omega(y)| \geq |y| - k$.

3 A simple algorithm

The following preliminary lemmas allow us to process the string y of an input instance (x, y, k) in a left-to-right manner, and to assume that a solution exists only if there is a solution M such that $M(1) = \phi(y, x_1)$ (i.e., the first symbol in x can be matched with its first occurrence in y).

Lemma 1. *Let $x, y \in \Sigma^*$ be an input of a* String-to-String Correction *instance. Let $J = \phi(y, x_1)$. Then, in any optimum solution, the symbols $y_1 \ldots y_{J-1}$ are either to be deleted or swapped to the right, finally beyond the J^{th} letter y_J.*

Proof. Every symbol from the prefix $y_1 \ldots y_{J-1}$ must disappear, since x_1 must be the first symbol in the transformed string resulting from y. There are two ways to realize this: for each character $y_1 \ldots y_{J-1}$, we either delete it from y or we swap it with the adjacent character to its right. Swapping them to the left, or keeping some of them without any swap operation, does not lead to a solution since they all disagree with x_1. So they must be moved to the right of y_J (or further), and swapping them first to the left and then to the right can be avoided in an optimum solution. □

Lemma 2. *Let $x, y \in \Sigma^*$ be an input of* String-to-String Correction *and let $J = \phi(y, x_1)$. Then we can assume that all deletion-operations on the symbols $y_1 \ldots y_{J-1}$ can be performed before any operations involving the characters in substring $y_J \ldots y_m$.*

Proof. Assume otherwise that a solution is obtained by performing swap operations to the right of y_J and moving new symbols to the left of y_J. Then by Lemma 1, these symbols will have to be swapped back, incurring an additional (unnecessary) cost. □

Lemma 3. *Let $x, y \in \Sigma^*$ be an input of* String-to-String Correction *and let ω be an optimum sequence of operations to transform y into $\omega(y) = x$. Then, there is no $\omega_i = \sigma(y, j)$ with $y_j = y_{j+1}$. (In other words, consecutive identical symbols are never swapped in an optimum solution.)*

Proof. Assume there exists such an optimum sequence of operations $\omega' = \omega'_1\omega'_2 \ldots \omega'_k$ with $\omega'(y) = x$ and there exists an operation ω'_i in ω' such that $\omega'_i = \sigma(y, j)$ with $y_j = y_{j+1}$. For $\omega'' := \omega'_1\omega'_2 \ldots \omega'_{i-1}\omega'_{i+1} \ldots \omega'_k$ we have that $\omega''(y) = x$ with $|\omega''| < k$, contradicting optimality of ω'. □

Algorithm 1 A simple search tree algorithm for String-to-String Correction: sS2S

Input: two strings $x, y \in \Sigma^*$, an integer k
Output: TRUE if y can be transformed into x with at most k edit operations; FALSE otherwise.

```
if |x| = 0 then
   return (|y| ≤ k)
if φ(y, x_1) = 1 then
   return sS2S(τ(x), τ(y), k)
{Branch on φ(y, x_1) − 1}
if sS2S(x, δ(y, φ(y, x_1) − 1), k − 1) then
   return TRUE
return sS2S(x, σ(y, φ(y, x_1) − 1), k − 1)
```

Proposition 1. *Let $x, y \in \Sigma^*$ be an input of* String-to-String Correction. *If y can be transformed into x, then there is an optimum solution yielding a mapping M with $M(1) = \phi(y, x_1)$.*

Proof. Let $J = \phi(y, x_1)$. If $M'(1) \neq J$ in some optimum solution M', then there is an $i > 1$ such that $M'(1) = i$ and $i > J$. It follows that all symbol to the left of y_i are either deleted or moved to the right of y_i (Lemma 1). If y_J is to be deleted, then deleting y_i and keeping y_J would give an alternative solution whose cost is no larger than the cost corresponding to M'. On the other hand, if y_J is moved to the right of y_i, then we would reach a point where two consecutive occurrences of the same symbol are swapped (e.g., y_J and y_i), yielding a contradiction to M' being a mapping for an optimal solution (Lemma 3). □

Corollary 1. *If the first symbols of x and y agree, that is $\phi(y, x_1) = 1$, then the two instances (x, y, k) and $(\tau(x), \tau(y), k)$, of* String-to-String Correction, *are equivalent, that is (x, y, k) is a* YES*-instance for* String-to-String Correction *if and only if $(\tau(x), \tau(y), k)$ is a* YES*-instance for* String-to-String Correction. □

The last proposition suggests the simple search-tree algorithm sS2S (Algorithm 1). The correctness of this algorithm follows by a simple induction using the previous lemmas and observations. This allows us to conclude:

Proposition 2. String-to-String Correction *is fixed-parametertractable; more precisely, it can be solved in time $\mathcal{O}^*(2^k)$.* □

Finally, observe the following case where we can prune the search tree: If $\phi(y, x_1) > k+1$, we can answer NO, since it takes at least k operations to take care of the prefix $y_1 \dots y_{\phi(y,x_1)}$ (see Lemma 1).

4 Improving on search trees

We now describe an algorithm that is more complex than Algorithm 1, but that yields a noticeable, improved running time. The improvement is mostly due to a novel annotation scheme that is potentially useful also for the analysis of other algorithms: by so-called *charging* and *uncharging* of operations, we can reduce the budget (the parameter k in this case) in certain situations by charging characters with swap operations that can be later performed "for free," as they are prepaid.

Moreover, we will see that whenever we decide to delete a (currently processed) character a at position $I < \phi(y, x_1)$, we can also delete any occurrence of symbol a at the positions before I. Recall that the first character of x is matched to $J = \phi(y, x_1)$ (cf. Proposition 1) and all characters to the left of y_J must either be deleted or swapped to the right (Lemma 1).

We annotate characters from y with elements in $\{0, 1, 2, -\}$ where a symbol (a, i), $i \geq 0$, denotes an occurrence of character $a \in \Sigma$ that has been determined not to be deleted; instead this a is to be swapped at least i times to the right. If $i = -$, then (a, i) is not to be deleted and cannot be swapped to the right. Note that the negative charge $(i = -)$ is used to model the mapping of the first occurrence of x_1 within y.

We introduce the *charging operation* $\mu(y, j, i)$ that replaces character y_j at position j with the annotated form (y_j, i). Note that μ may also be used to change the charge; in that case, (y_j, ℓ) is replaced by (y_j, i) upon applying $\mu(y, j, i)$.

Note that from now on sequence y may be annotated, that is some characters can be charged. Further, also an annotated occurrence (a, i) of a in y will be called an *occurrence of a in y*. We also say that y *contains* a. When we only look for symbols a within y, we speak of *uncharged occurrences*. If $a \in \Sigma$ occurs twice in a string $x \in \Sigma^*$, we also say that the two occurrences of a define a *twin occurrence* of a in x.

4.1 Reduction rules to handle annotations

We next introduce a number of reduction rules where reduction rule i is applied if none of the preceding reduction rules can be applied to the instance. These reduction rules serve as preprocessing step of our recursive algorithm (Algorithm 2). We describe each reduction rule assuming an original instance x, y, k and a new instance x', y', k'.

1. If a symbol a occurs more times in x than in y, then answer NO.
2. If a symbol a occurs in y but not in x then $x' = x$, y' is obtained by deleting all ℓ occurrences of a from y and $k' = k - \ell$.
3. Let $x_1 = a$ and let J be the position of the (first) occurrence of $(a, -)$ in y. If a symbol b occurs only once in y, namely at position $j < J$ and if it is still uncharged, then $x' = x$, $y' = \mu(y, j, 0)$, and $k' = k$.
4. If y contains the successive neighbours $(a, i)(b, -)$, where $i \in \{1, 2\}$ and $a \neq b$, at positions j and $j+1$, then $x' = x$, y' is obtained by swapping the two characters and decreasing a's charge, that is by replacing y_j with $(b, -)$ and y_{j+1} by $(a, i-1)$ and $k' = k$.
5. Let $x_1 = a$ and let J be the position of the (first) occurrence of $(a, -)$ in y. If y contains the successive neighbours $(b, i)b$, $i \neq -$, at positions j and $j+1$, $j < J$, then $x' = x$, y' is obtained from y by charging b's occurrence at position $j+1$, i.e., $y' = \mu(y, j+1, 0)$ and $k' = k$.
6. If x starts with $a\alpha b$, $a \neq b$, where α is some string not containing b, and y starts with $\beta(b, 0)(a, i)$ with $i \in \{0, -\}$, where β contains no a, then $x' = x$, $y' = \mu(y, |\beta| + 1, 1)$ and $k' = k - 1$.
7. If $x_1 = a$ and if j is the position of the first occurrence of character a in y with $y_j \in \{(a, 1), (a, 2)\}$, then answer NO.
8. If $x_1 = y_1$ or $y_1 = (x_1, 0)$ or $y_1 = (x_1, -)$, then $x' = \tau(x)$, $y' = \tau(y)$ and $k' = k$.
9. If $x_1 = a$, $x_2 = b$, $x_3 = b$, y_1 equals (possibly charged) character b, $y_2 = a$ and ($y_3 = b$ or $y_3 = (b, 0)$), then $x' = \delta(x, 3)$, $y' = \delta(y, 3)$ and $k' = k$.
10. Let $x_1 = a$ and let J be the position of the first occurrence of $(a, -)$ in y. If $y_j = b$ and if $y_i = (b, s)$ with $j < i < J$, then charge y_j with 1, i.e., $x' = x$, $y' = \mu(y, j, 1)$ and $k' = k - 1$.
11. If $x_1 = a$ and y contains $(b, -)$ (before any kind of a-occurrence) followed by character $(a, -)$ as a (not necessarily contiguous) subsequence where $a \neq b$, then answer NO.

Lemma 4. *The reduction rules are sound.*

Proof. 1. By problem definition.

2. Similarly.

3. Since b is unique in y, it is not to be deleted. Further, at its current position, it should move right.

4. Neither the annotated a nor b can be deleted from y. (a, i) with $i > 0$ must swap right, $(b, -)$ should swap left or stay. Moreover, after the swap the charge of a has to be decreased by 1.

5. This is a consequence of Lemma 3.

6. The (annotated) character $y_1 = (b, 0)$ is not to be deleted, because it is charged with zero. Since it cannot be matched with x_1, $(b, 0)$ must swap with the right, which is expressed by increasing b's charge.

7. Since character (a, i) is a's first occurrence in y it has to be matched with x_1. However, the charge disallows this since the character is forced to swap at least ones with a right neighbor character.

8. Since $M(x_1) = 1$ and no charge of y_1 forces y_1 to be swapped.

9. According to Proposition 1, $M(1) = 2$. Character y_1 either must be deleted or swapped with character y_2. In the first case, Reduction Rule 8 would be applied afterwards twice, resulting into an instance consisting of $x'' = \tau(\tau(x))$, $y'' = \tau(\tau(\delta(y, 1)))$ and $k'' = k - 1$. Note that deleting y'_1 in instance (x', y', k') results into the same instance after reducing, namely $\tau(x') = \tau(\delta(x, 3)) = x''$ and $\tau(\delta(y', 1)) = \tau(\delta(\delta(y, 3), 1)) = y''$. In the case of swapping y_1 and y_2 we again can reduce to $x'' = \tau(\tau(\tau(x)))$, $y'' = \tau(\tau(\tau(\sigma(y, 1))))$ and $k'' = k$. Also here, deleting the third character in x and y first before swapping does not make a difference in the resulting sequence.

10. Recall that every character in the prefix up to the character before $(a, -)$ in y has to be either deleted or swapped passed $(a, -)$. Assume it would be better to delete character b at position j in y. But then it would be cheaper to delete b at position i instead, since there is a smaller travel-distance to pass. Since b at position i is not to be deleted, deleting y_j yields a contradiction.

11. The $(b, -)$ preceding $(a, -)$ should not travel right but at the same time is not to be deleted. Thus $(a, -)$ can never be transported to the y_1. □

4.2 The search tree algorithm

The basic branching strategy of the more sophisticated branching algorithm S2S is inherited from Alg. 1. Again, symbols that we branch on are either deleted or swapped to the right. The algorithm is enriched with the concept of charging and extensively uses reduction rules as described before. Note that whenever we try to branch on a symbol that is already charged, we can safely ignore the branch of the case when that symbol is to be deleted, since we already decided earlier not to delete that symbol. Details of the algorithm can be found in Alg. 2. The correctness and run time are treated in the next subsection.

4.3 The correctness and run time of Algorithm S2S

To verify the correctness of Algorithm S2S (Alg. 2) we first observe that $J > 1$ whenever we branch. The correctness of the reductions (lines 1–12) follows from the soundness of the reduction rules, lemmas and propositions above. Note that, if no reduction applies, the algorithm branches, distinguishing the cases $J = 2$ and $J > 2$. We next verify the correctness of the different branching cases in the algorithm.

Lemma 5. *Let $x_1 = a$ occur at position J in y. If branching takes place at position $J - 1$, then y_{J-1} is uncharged (before branching).*

Proof. Let $x_1 = a$ occur at position J in y. Due to charging, we first turn it into $(a, -)$, our general situation that we encounter when we branch. This is justified by Proposition 1. Consider branching at $J - 1$. Assume that $y_{J-1} = (d, i)$ for some $i > 0$. Reduction Rule 4 then lets all positively charged symbols that are immediate left neighbours to a swap places with a, this way decreasing J. Finally, either $J = 1$ is reached, a case again handled by reduction, or $y_{J-1} \in \{(d, 0), d\}$. If $y_{J-1} = (d, 0)$, no branching takes place (since reduction rule 6 and then reduction rule 4 would trigger). In the second case, we actually do branch. □

Algorithm 2 A search tree algorithm for String-to-String Correction: S2S

Input: Two strings $x, y \in \Sigma^*$, an integer k.
Output: TRUE if y can be transformed into x with at most k edit operations; FALSE otherwise.

```
if |x| = 0 then
  return (|y| ≤ k)
Exhaustively perform the reduction rules.
J ← φ(y, x_1) {determine the first occurrence of the character x_1 in y.}
if J > k + 1 then
  return FALSE
if y_J = x_1 then
  {y_J is uncharged}
  return S2S(x, μ(y, J, −), k) {minus-charge first occurrence of x_1 in y and recurse}
{y_J ∈ {(x_1, −), (x_1, 0)}}
if J = 2 then
  if (y_1 is not an occurrence of character x_1) or (positions 1 and 3 of y are a twin occurrence) then
    {Branch on y_1}
    if S2S(x, δ(y, 1), k − 1) then
      return TRUE
    else
      return S2S(x, μ(y, 1, 2), k − 2)
  else
    {Branch on y_3}
    if S2S(x, δ(y, 3), k − 1) then
      return TRUE
    else
      y ← μ(y, 1, 1); k ← k − 1
      return S2S(x, μ(y, 3, 1), k − 1)
else
  {J > 2; elimination phase of twin occurrences of Σ in y_1 ... y_{J−1}.
  Please find the pseudocode for this case in Alg. 3.}
```

For the ease of notation, we will use in the following the uncharging morphism h that maps d and (d,i) for $i \in \{-,0,1,2\}$ onto d.

The case $J = 2$:

Let us first analyze the case $J = 2$, with the subcase $x_1 = a$, $x_2 = b$, $h(y_1) = d$ and $h(y_2) = a$. We can depict this case by the array:

$$\begin{bmatrix} x & = & a & b & \dots \\ h(y) & = & d & a & \dots \end{bmatrix}.$$

Note that $y_2 = (a,-)$ or $y_2 = (a,0)$ due to Rule 7. By reduction, $y_1 = d$: Namely, if $y_1 = (d,0)$, Reduction Rule 6 would turn y_1 into $(d,1)$, since $a \neq b$. If $y_1 = (d,i)$ for $i = 1,2$, then Reduction Rule 4 would swap y_1 and y_2. If $y_1 = (d,-)$, Reduction Rule 11 would answer NO.

Consider the subcase that $b \neq d$. We branch on $J-1 = 1$ of type 2. Namely, when keeping d, d must be swapped both over a and over the first occurrence of b in y; this justifies the charge. This is therefore a $(1,2)$-branch.

Algorithm 3 The pseudocode for the case $J > 2$.

```
{J > 2; elimination phase of twin occurrences of Σ in y_1 ... y_{J-1}}
if y_1 ... y_{J-1} contains a twin occurrence of some (uncharged) e ∈ Σ then
    Let R be the rightmost position in y_1 ... y_J with y_R = e.
    {By RR 10, we know any other e in y_1 ... y_{R-1} is not charged. We branch on R:}
    if S2S(x, μ(y, R, 1), k − 1) then
        return TRUE
    else
        y ← delete all occurrences of e in y_1 ... y_R from y
        {Assume t occurrences of e were deleted from y in the line above}
        return S2S(x, y, k − t)
else
    {At this stage, if e ∈ Σ occurs at two positions in y_1 ... y_J then at most e's leftmost position is not charged}
    d ← y_{J−1}{Then, by RR10, RR6 and RR4, d is not charged.}
    if there is another uncharged symbol c in y_1 ... y_{J−2} then
        let I be c's position in y
    else if there exists character (c, 0) before position J in y then
        Let I be the rightmost position with y_I = (c, 0)
    else
        I ← 1
    if the leftmost position of c in x precedes the leftmost position of d in x or I = 1 then
        if S2S(x, δ(y, J − 1), k − 1) then
            return TRUE
        else
            if y_I = (c, 0) or y_I = c then
                y ← μ(y, J − 2, 1); k ← k − 1
            return S2S(x, μ(y, J − 1, 1), k − 1))
    else
        if S2S(x, δ(y, I), k − 1) then
            return TRUE
        else
            return S2S(x, μ(y, I, 2), k − 2)
```

Consider the subcase $x_1 = a$, $x_2 = b$, $x_3 = c$, $h(y_1) = b$, $y_2 = (a,-)$, $h(y_3) = b$. (The case $(a,0)$ is again treated similarly.) Pictorially, we consider:

$$\begin{bmatrix} x & = & a & b & c & \dots \\ h(y) & = & b & a & b & \dots \end{bmatrix}.$$

Again, $y_1 = b$. If $c = b$, Reduction Rule 9 would apply. Hence, we assume $c \neq b$ in what follows. In the branch that keeps the first occurrence of b, i.e., turning it first into $(b,0)$, this is turned into $(b,1)$ by Rule 6 and then swapped with $(a,-)$ by Reduction Rule 4, finally reducing it to $(b,0)$ by cutting off $(a,-)$ using Rule 8. Again by reduction (Rule 10), the second occurrence of b in $h(y)$ is then turned into $(b,0)$. Reduction Rule 8 will then delete the prefix ab from x and the prefix $(a,0)(b,0)$ from y. Finally, the first symbol $(b,0)$ in y will be changed into $(b,1)$ by reduction (see Rule 5), since $c \neq b$, justifying the $(1,2)$-branch.

Consider the subcase $x_1 = a$, $x_2 = b$, $x_3 = c$, $h(y_1) = b$, $y_2 = (a,-)$, $h(y_3) = d$. Assume $d \neq b$ due to the previous case. Pictorially, we consider:

$$\begin{bmatrix} x & = & a & b & c & \dots \\ h(y) & = & b & a & d & \dots \end{bmatrix}.$$

Again, $y_1 = b$. Possibly, $d = c$ in the following discussion.

There must be two occurrences of b in $h(y)$, since otherwise a reduction rule would apply (see Rule 3). Consider the case that y_3 is not deleted. (This also applies to the case when y_3 is already charged and no branching takes place.) We argue that we have to swap the first occurrence of b in y, i.e., y_1, at least once to the right, justifying the according charge. If we would delete y_1, we would have to swap the second b in $h(y)$ finally with the current y_3 (possibly, more swaps are necessary), leading to ≥ 2 edits. If we swap y_1 instead, we could arrive at the same situation by deleting the second occurrence of b in y, but with no more cost. Notice that there must be a yet uncharged occurrence of b by Reduction Rule 3. This justifies the additional swap, reflected by charging y_1.

Eliminating twin occurrences:

Consider now the elimination phase of twin occurrences in $y_1 \dots y_{J-1}$. Since we have decided that y_J is converted to some $(a,-)$, all symbols to the left of $(a,-)$ must be finally deleted or swapped to the right. Assume there are two (or more) occurrences of e in $y_1 \dots y_{J-1}$ before the elimination phase, say, $y_\ell = y_r = e$ with $\ell < r$. Consider the case when exactly one of the e's is deleted (and the other is swapped beyond $(a,-) = y_J$). To minimize the "distance" to $(a,-)$, it is always no worse to delete y_ℓ and swap y_r than the other way round. Therefore, if one occurrence of e is deleted, also all occurrences of e to its left can be deleted. This is the reason why the branching is correct. Since we only branch when there is a twin occurrence of e, clearly, a $(1,2)$-branch is obtained, since in the case when the rightmost e-occurrence is not deleted, it can be charged with at least one as it has to be swapped beyond $(a,-)$. Moreover, by contraposition, if y_ℓ is swapped, y_r should be swapped, as well. This explains why the deterministic decision for charging e with zero is correct, once we know that an occurrence of e to its left is charged.

Hence, after the elimination phase of twin occurrences, if $e \in \Sigma$ occurs at two positions in $h(y_1 \dots y_J)$, then at most its leftmost position j might satisfy $y_j = h(y_j)$.

The case $J > 2$:

Let c, d be the rightmost two uncharged symbols of $y_1 \dots y_{J-1}$. Due to the twin elimination phase, they are different (and also different from $a = h(y_J)$).

Hence, both c and d must occur in x (otherwise, Reduction Rule 2 would apply). As argued before, $y_{J-1} = d$. There are two subcases to be taken care of: Either (1) $x = a\alpha c\beta d$ with no occurrence of c, d in α and no occurrence of d in β, or (2) $x = a\alpha d\beta c$ with no occurrence of c, d in α and no occurrence of c in β. So, the case distinction is according to the question whether c precedes d in x or vice versa.

In case (1), we branch at $y_{J-1} = d$. We claim that when d is not deleted, the (not necessarily immediately) preceding symbol $c \in \Sigma$ at position I is not deleted, either. Namely, if c was deleted, then another occurrence of c must be swapped with d to produce the sequence given in x, which does not have a better correction cost than swapping c over $(a, -)$ (after swapping d) and possibly deleting another c-occurrence. Due to the swapping, we can charge d and c by one. The same discussion applies if $y_I = (c, 0)$. In particular, the same charging is possible. We discuss the last possibility, when $I = 1$ is chosen. How can y_I get charged? If it is charged because of an earlier branch of this type, then clearly all symbols left to some $(f, -)$-symbol chosen in that earlier branch have been charged and will hence be swapped finally with $(f, -)$ with the help of the reduction rules. Therefore, its charge will have been dropped to zero, in contrast to our assumption that $y_I = y_1 = (c, 1)$ was chosen. So, this sort of fallback does never apply.

In case (2), we branch at y_I. Assume first that $y_I = c$ or $y_I = (c, 0)$. Due to the structure of x, $c = h(y_I)$ must be swapped at least twice to the right when we decide to swap it. Hence, it is correct to charge it with two if $y_I = c$ or $y_I = (c, 0)$. In the case $y_I = (c, i)$ with $i = 1, 2$ we have $i = 1$, so that the branching of case (1) was applied.

Via a straightforward analysis of the used branching vectors in Algorithm S2S (Alg. 2) we can conclude:

Theorem 1. *Algorithm S2S (Alg. 2) solves* String-to-String Correction *in time* $\mathcal{O}^*(1.6182^k)$. □

5 Further Observations

Observe that the problem of turning a given string y into another given string x by at most k swaps or insertions of symbols can be seen as the problem of turning x into y by using at most k swaps or deletions. Therefore, the *swap-insertion string-to-string edit problem* is also $\mathcal{NP}$-hard and allows an $\mathcal{O}^*(1.6182^k)$ algorithm. By considering the possibility of matching the first symbol of y with a newly inserted symbol (at the left end) of x as a first case, we receive also $\mathcal{FPT}$-results when allowing all three operations.

Corollary 2. *The string-to-string edit problem can be solved in time* $\mathcal{O}^*(1.6182^k)$ *when allowing swaps and insertions, and it can be solved in time* $\mathcal{O}^*(2.4143^k)$ *when allowing swaps, insertions, and deletions.*

Proof. Let us first consider the case when swaps and insertions are the allowed operations. Clearly, $|y| \leq |x|$. So, instead of solving the task of turning y into x with swap and insertion operations, we try to turn x into y with the help of at most k swaps and deletes. As mentioned before, this gives us a matching $M : \{1, \ldots, |y|\} \to \{1, \ldots, |x|\}$. We can interpret this mapping M as follows: (a) Compute the cheapest sequence of transpositions (swaps) that realizes the permutation of the string y indicated by M; this can be done in polynomial time by dynamic programming. Let y' be the resulting string. Clearly, y' is a (non-contiguous) subsequence of x. (b) Insert the missing $|x|-|y|$ letters into y' to obtain x. Since a similar procedure can be used to obtain an optimum solution (with respect to the given M) for the swap-delete edit problem (turning x into y) with needs the same number of operations as the swap-insert edit problem (turning y into x) described above, it is clear that the number of operations that we need for the original swap-insert edit problem is bounded by k, since M was obtained with this bound.

Now consider the case when all three operations are allowed. Clearly, the first symbol of x must be found somehow in the variant of y we are going to construct.

Claim. If the first symbol x_1 of x is matched against a symbol that is inserted into y, then it is, w.l.o.g., inserted at the left end of y.

Namely, assume x_1 is inserted somewhere else in y, say immediately to the left of symbol J. This means that all symbols to the left of J should be deleted in y, since it would be surely cheaper to first swap those symbols to the right and then insert a symbol compared to the other sequence of operations. But, if all symbols to the left of J are deleted, then this gives the same result as inserting x_1 at the left end of y and then deleting the following $J - 1$ symbols in subsequent branches. Note that this possibility would be automatically considered later in the recursion.

With the help of this claim, it is clear that we could consider as a first possibility to insert x_1 to the left end of y. When we do not insert x_1 at that point, we would not insert x_1 anywhere into y in order to match it against the first symbol of x, so that we should match x_1 against the first occurrence of that symbol within y, say with y_J. The arguments given for justifying the main algorithm of this paper do now apply, showing that we reduce the parameter budget either by one (insertion case), or by one (deletion case), or by two (swap case). The according recursion that estimates the search tree size is therefore

$$T(k) \leq 2T(k-1)+T(k-2), \quad T(0) = c_0, \quad T(1) = c_1$$

for some positive constants $c_0 \leq c_1$. Since $(1+\sqrt{2})^2 = 2(1+\sqrt{2})+1$, we can set $T(k) = (1+\sqrt{2})^k$, $c_0 = 1 \leq c_1 = 1+\sqrt{2}$, and the claim regarding the running time follows. □

Since we never used the fact that swaps and deletion operations cost the same, we can also use our algorithm when costs are associated to the edit operations, yielding a running time that is no worse.

6 Conclusions

We presented the first fixed-parameter algorithm solving String-to-String Correction parameterized by the number of permitted edit operations (namely, single character deletions or swaps). A main contribution of this paper can be seen as exhibiting the technique of charging parts of a problem instance, so that it is possible to already deduce the budget and hence prove a better run time than a simpler $\mathcal{O}^*(2^k)$-algorithm. These kind of annotations should be useful in other problems, in particular string problems.

Some parts of Algorithm S2S (Alg. 2) and its analysis are dependent on the alphabet size, which is unlimited in the problem formulation. However, it is quite reasonable to assume a fixed upper bound on

its size. For example, what about the restriction to binary alphabets? Are faster algorithms available?

There are some obvious properties that we did not make use of in our algorithm. For example, the total number of deletions is $|y| - |x|$, which could be smaller than k. So if this number of deletions is already performed during the search, we can proceed by performing swappings only. In this latter case, the problem is solvable in polynomial time via dynamic programming. If, on the other hand, the "swappings budget" (i.e., $k - (|y| - |x|)$ is consumed, then we also proceed by a polynomial-time detection of symbols that are to be deleted. We believe that such an either-or approach would lead to a faster algorithm in practice. However, we know of no methodology that would allow to include this in a mathematical analysis of the search tree size; refer to [9] for a more general discussion on the estimate of search tree sizes. Moreover, by the fact that a swap can be replaced by a pair of delete and insert operations, and since the question whether x can be turned into y by using at most $2k$ delete or insert operations can be solved in polynomial time, a simple early-abort criterion can be added to our search tree algorithm.

Let us conclude with a quotation from the survey by Navarro [17]: "Although transpositions are of interest (especially in case of typing errors), there are few algorithms to deal with them." We hope the present paper inspires renewed interest in this area.

References

[1] P. Berman and S. Hannenhalli. Fast Sorting by Reversals. CPM '96: Proc. of the 7^{th} Annual Symposium on Combinatorial Pattern Matching. 168–185, 1996.

[2] P. Berman, S. Hannenhalli and M. Karpinski. 1.375-Approximation Algorithm for Sorting by Reversals. Electronic Colloquium on Computational Complexity (ECCC). 8(47), http://eccc.hpi-web.de/eccc-reports/2001/TR01-047/index.html, 2001

[3] P. Berman and M. Karpinski. On Some Tighter Inapproximability Results. ICALP '99: Proc. of the 26^{th} International Colloquium on Automata, Languages and Programming. 200–209, 1999.

[4] A. Caprara. Sorting by reversals is difficult. RECOMB '97: Proc. of the 1^{st} annual international conference on Computational molecular biology. 75–83, ACM, 1997.

[5] F. J. Damerau. A technique for computer detection and correction of spelling errors. *Comm. ACM*, 7(3), 1964.

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

[7] I. Elias and T. Hartman. A 1.375-Approximation Algorithm for Sorting by Transpositions. Proc. of the 5^{th} International Workshop on Algorithms in Bioinformatics (WABI'05). 204–214, 2005.

[8] H. Fernau and R. Niedermeier. An efficient exact algorithm for constraint bipartite vertex cover. *J. Alg.*, 38(2):374–410, 2001.

[9] H. Fernau and D. Raible. Searching trees: an essay. In J. Chen and S. B. Cooper, editors, *TAMC*, volume 5532 of *LNCS*, pp. 59–70. Springer, 2009.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[11] Q.-P. Gu, S. Peng and H. Sudborough, A 2-approximation algorithm for genome rearrangements by reversals and transpositions, *Theoretical Computer Science* 210(2), 327–339, 1999.

[12] S. Hannenhalli and P.A. Pevzner. Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. Journal of the ACM. 46(1), 1–27, 1999.

[13] H. Kaplan, R. Shamir and R.E. Tarjan, A Faster and Simpler Algorithm for Sorting Signed Permutations by Reversals, SIAM J. Comput. 29(3), 880–892, 2000.

[14] J. D. Kececioglu and D. Sankoff. Exact and Approximation Algorithms for Sorting by Reversals, with Application to Genome Rearrangement. Algorithmica 13(1/2), 180–210, 1995.

[15] Z. Li, L. Wang, and K. Zhang. Algorithmic approaches for genome rearrangement: a review. *IEEE Trans. Systems, Man, and Cybernetics, Part C*, 2006.

[16] R. Lowrance and R. A. Wagner. An extension of the string-to-string correction problem. *J. ACM*, 22:177–183, 1975.

[17] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33, 2001.

[18] K. Toutanova and R. C. Moore. Pronunciation modeling for improved spelling correction. In *ACL '02:* , pp. 144–151, 2001.

[19] R. A. Wagner. On the complexity of the extended string-to-string correction problem. In *STOC '75*, pp. 218–223, 1975.