Metadata of the chapter that will be visualized in SpringerLink

Book Title	String Processing and Information Retrieval	
Series Title		
Chapter Title	Adaptive Computation of the Swap-Insert Correction Distance	
Copyright Year	2015	
Copyright HolderName	Springer International Publishing Switzerland	
Corresponding Author	Family Name	**Barbay**
	Particle	
	Given Name	**Jérémy**
	Prefix	
	Suffix	
	Division	Departamento de Ciencias de la Computación
	Organization	Universidad de Chile
	Address	Santiago, Chile
	Email	jeremy@barbay.cl
Corresponding Author	Family Name	**Pérez-Lantero**
	Particle	
	Given Name	**Pablo**
	Prefix	
	Suffix	
	Division	Escuela de Ingeniería Civil en Informática
	Organization	Universidad de Valparaíso
	Address	Valparaíso, Chile
	Email	pplantero@yahoo.com

Abstract

The Swap-Insert Correction distance from a string S of length n to another string L of length $m \geq n$ on the alphabet $[1..d]$ is the minimum number of insertions, and swaps of pairs of adjacent symbols, converting S into L. Contrarily to other correction distances, computing it is NP-Hard in the size d of the alphabet. We describe an algorithm computing this distance in time within $O(d^2nmg^{d-1})$, where there are n_α occurrences of α in S, m_α occurrences of α in L, and where $g = \max_{\alpha \in [1..d]} \min\{n_\alpha, m_\alpha - n_\alpha\}$ measures the difficulty of the instance. The difficulty g is bounded by above by various terms, such as the length of the shortest string S, and by the maximum number of occurrences of a single character in S. The latter bound yields a running time within $O(d(n+m) + (d/(d-1)^{d-2}) \cdot n^d(m-n))$ in the worst case over instances of fixed lengths n and m for S and L, which further simplifies to within $O(n^d(m-n)+m)$ when d is fixed, the state of the art for this problem. This illustrates how, in many cases, the correction distance between two strings can be easier to compute than in the worst case scenario.

Keywords (separated by '-') Adaptive - Dynamic programming - Edit distance - Insert - Swap

Author Proof

Adaptive Computation of the Swap-Insert Correction Distance

Jérémy Barbay[1(✉)] and Pablo Pérez-Lantero[2(✉)]

[1] Departamento de Ciencias de la Computación, Universidad de Chile, Santiago, Chile
jeremy@barbay.cl

[2] Escuela de Ingeniería Civil en Informática, Universidad de Valparaíso, Valparaíso, Chile
pplantero@yahoo.com

Abstract. The Swap-Insert Correction distance from a string S of length n to another string L of length $m \geq n$ on the alphabet $[1..d]$ is the minimum number of insertions, and swaps of pairs of adjacent symbols, converting S into L. Contrarily to other correction distances, computing it is NP-Hard in the size d of the alphabet. We describe an algorithm computing this distance in time within $O(d^2nmg^{d-1})$, where there are n_α occurrences of α in S, m_α occurrences of α in L, and where $g = \max_{\alpha\in[1..d]} \min\{n_\alpha, m_\alpha - n_\alpha\}$ measures the difficulty of the instance. The difficulty g is bounded by above by various terms, such as the length of the shortest string S, and by the maximum number of occurrences of a single character in S. The latter bound yields a running time within $O(d(n+m)+(d/(d-1)^{d-2})\cdot n^d(m-n))$ in the worst case over instances of fixed lengths n and m for S and L, which further simplifies to within $O(n^d(m-n)+m)$ when d is fixed, the state of the art for this problem. This illustrates how, in many cases, the correction distance between two strings can be easier to compute than in the worst case scenario.

Keywords: Adaptive · Dynamic programming · Edit distance · Insert · Swap

1 Introduction

Given two strings S and L on the alphabet $\Sigma = [1..d]$ and a list of correction operations on strings, the String-to-String Correction distance is the minimum number of operations required to transform the string S into the string L. Introduced in 1974 by Wagner and Fischer [7], this concept has many applications, from suggesting corrections for typing mistakes, to decomposing the changes between two consecutive versions into a minimum number of correction steps, for example within a control version system.

J. Barbay and P. Pérez-Lantero—Partially supported by Millennium Nucleus Information and Coordination in Networks ICM/FIC RC130003.

© Springer International Publishing Switzerland 2015
C. Iliopoulos et al. (Eds.): SPIRE 2015, LNCS 9309, pp. 1–12, 2015.
DOI: 10.1007/978-3-319-23826-5_3

Author Proof

Each distinct set of correction operators yields a distinct correction distance on strings. For instance, Wagner and Fischer [7] showed that for the three following operations, the `insertion` of a symbol at some arbitrary position, the `deletion` of a symbol at some arbitrary position, and the `substitution` of a symbol at some arbitrary position, there is a dynamic program solving this problem in time within $O(nm)$ when S is of length n and L of length m. Similar complexity results, all polynomial, hold for many other different subsets of the natural correction operators, with one striking exception: Wagner [6] proved the NP-hardness of the Swap-Insert Correction distance, denoted $\delta(S, L)$ through this paper, i.e. the correction distance when restricted to the operators `insertion` and `swap` (or, by symmetry, to the operators `deletion` and `swap`).

The Swap-Insert Correction distance's difficulty attracted special interest, with two results of importance: Abu-Khzam et al. [1] described an algorithm computing $\delta(S, L)$ in time within $O(1.6181^{\delta(S,L)}m)$, and Meister [4] described an algorithm computing $\delta(S, L)$ in time polynomial in the input size when S and L are strings on a finite alphabet.

The complexity of Meister's result [4], polynomial in m of degree $2d+1$, is a very pessimistic approximation of the computational complexity of the distance. At one extreme, the Swap-Insert Correction distance between two strings which are very similar (e.g. only a finite number of symbols need to be swapped or inserted) can be computed in time linear in n and d. At the other extreme, the Swap-Insert Correction distance of strings which are completely different (e.g. their effective alphabets are disjoint) can also be computed in linear time (it is then close to $n+m$). Even when S and L are quite different, $\delta(S, L)$ can be "easy" to compute: when mostly swaps are involved to transform S into L (i.e. S and L are almost of the same length), and when mostly insertions are involved to transform S into L (i.e. many symbols present in L are absent from S).

Hypothesis: We consider whether the Swap-Insert Correction distance $\delta(S, L)$ can be computed in time polynomial in the length of the input strings for a constant alphabet size, while still taking advantage of cases such as those described above, where the distance $\delta(S, L)$ can be computed much faster.

Our Results: After a short review of previous results and techniques in Section 2, we present such an algorithm in Section 3, in four steps: the intuition behind the algorithm in Section 3.1, the formal description of the dynamic program in Section 3.2, and the formal analysis of its complexity in Section 3.3. In the latter, we define the local imbalance $g_\alpha = \min\{n_\alpha, m_\alpha - n_\alpha\}$ for each symbol $\alpha \in \Sigma$, summarized by the global imbalance measure $g = \max_{\alpha\in\Sigma} g_\alpha$, and prove that our algorithm runs in time within $O(d^2 g^{d-1} nm)$ in the worst case over instances where d, n, m and g are fixed.

We discuss in Section 4 some implied results, and some questions left open. Additional details are deferred to the full version [2].

Author Proof

2 Background

In 1974, motivated by the problem of correcting typing and transmission errors, Wagner and Fischer [7] introduced the String-to-String Correction problem, which is to compute the minimum number of corrections required to change the source string S into the target string L. They considered the following operators: the `insertion` of a symbol at some arbitrary position, the `deletion` of a symbol at some arbitrary position, and the `substitution` of a symbol at some arbitrary position. They described a dynamic program solving this problem in time within $O(nm)$ when S is of length n and L of length m. The worst case among instances of fixed input size $n + m$ is when $n = m/2$, which yields a complexity within $O(n^2)$.

In 1975, Lowrance and Wagner [8] extended the String-to-String Correction distance to the cases where one considers not only the `insertion`, `deletion`, and `substitution` operators, but also the `swap` operator, which exchanges the positions of two contiguous symbols. Not counting the identity, fifteen different variants arise when considering any given subset of those four correction operators. Thirteen of those variants can be computed in polynomial time [6–8]. The two remaining distances, the computation of the Swap-Insert Correction distance and its symmetric the Swap-Delete Correction distance, are equivalent by symmetry, and are NP-hard to compute [6], hence our interest. All our results on the computation of the Swap-Insert Correction distance from S to L directly imply the same results on the computation of the Swap-Delete Correction distance from L to S.

In 2013, Spreen [5] observed that Wagner's NP-hardness proof [6] was based on unbounded alphabet sizes (i.e. the Swap-Insert Correction problem is NP-hard when the size d of the alphabet is part of the input), and suggested that this problem might be tractable for fixed alphabet sizes. He described some polynomial-time algorithms for various special cases when the alphabet is binary, and some more general properties.

In 2014, Meister [4] extended Spreen's work [5] to an algorithm computing the Swap-Insert Correction distance from a string S of length n to another string L of length m on any fixed alphabet size $d \geq 2$, in time polynomial in n and m. The algorithm is explicitly based on finding an injective function $\varphi : [1..n] \to [1..m]$ such that $\varphi(i) = j$ if and only if $S[i] = L[j]$, and the total number of crossings is minimized. Two positions $i < i'$ of S define a *crossing* if and only if $\varphi(i) > \varphi(i')$. Such a number of crossings equals the number of swaps, and the number of insertions is always equal to $m - n$. Meister proved that the time complexity of this algorithm is equal to $(m + 1)^{2d+1} \cdot (n + 1)^2$ times some function polynomial in n and m.

3 Algorithm

We describe the intuition behind our algorithm in Section 3.1, the high level description of the dynamic program in Section 3.2, and the formal analysis of its complexity in Section 3.3.

Author Proof

3.1 High Level Description

The algorithm runs through S and L from left to right, building a mapping from the characters of S to a subset of the characters of L, using the fact that, for each distinct character, the mapping function on positions is monotone. The Dynamic Programming matrix has size $n_1 \times \cdots \times n_d < n^d$.

For every string $X \in \{S, L\}$, let $X[i]$ denote the i-th symbol of X from left to right ($i \in [1..|X|]$), and $X[i..j]$ denote the substring of X from the i-th symbol to the j-th symbol ($1 \leq i \leq j \leq |X|$). For every $1 \leq j < i \leq n$, let $X[i..j]$ denote the empty string. Given any symbol $\alpha \in \Sigma$, let $rank(X, i, \alpha)$ denote the number of occurrences of the symbol α in the substring $X[1..i]$, and $select(X, k, \alpha)$ denote the value $j \in [1..|X|]$ such that the k-th occurrence of α in X is precisely at position j, if j exists. If j does not exist, then $select(X, k, \alpha)$ is $null$.

The algorithm runs through S and L simultaneously from left to right, skipping positions where the current symbol of S equals the current symbol of L, and otherwise branching out between two options to correct the current symbol of S: inserting a symbol equal to the current symbol of L in the current position of S, or moving (by applying many swaps) the first symbol of the part not scanned of S equal to the current symbol of L, to the current position in S.

More formally, the computation of $\delta(S, L)$ can be reduced to the application of four rules:

- **if S is empty**: We just return the length $|L|$ of L, since insertions are the only possible operations to perform in S.
- **if some $\alpha \in \Sigma$ appears more times in S than in L**: We return $+\infty$, since `delete` operations are not allowed to make S and L match.
- **if S and L are not empty, $S[1] = L[1]$**: We return $\delta(S[2..|S|], L[2..|L|])$.
- **if S and L are not empty, $S[1] \neq L[1]$**: We compute two distances: the distance $d_{ins} = 1 + \delta(S, L[2..|L|])$ corresponding to an `insertion` of the symbol $L[1]$ at the first position of S, and the distance $d_{swaps} = (r - 1) + \delta(S', L[2..|L|])$ corresponding to perform $r - 1$ `swaps` to bring to the first position of S the first symbol of S equal to $L[1]$. In this case, r denotes the position of such a symbol, and S' the string resulting from S by removing that symbol. We then return $\min\{d_{ins}, d_{swaps}\}$.

There can be several overlapping subproblems in the recursive definition of $\delta(S, L)$ described above, which calls for *dynamic programming* [3] and *memoization*. In any call $\delta(S', L')$ in the recursive computation of $\delta(S, L)$, the string L' is always a substring $L[j..|J|]$ for some $j \in [1..|J|]$, and can thus be replaced by such an index j, but this is not always the case for the string S'. Observe that S' is a substring $S[i..|S|]$ for some $i \in [1..|S|]$ with (eventually) some symbols removed. Furthermore, if for some symbol $\alpha \in \Sigma$ precisely c_α symbols α of $S[i..|S|]$ have been removed, then those symbols are precisely the first c_α symbols α from left to right. We can then represent S' by the index i and a counter c_α for each symbol $\alpha \in \Sigma$ of how many symbols α of $S[i..|S|]$ are removed (i.e. ignored). In the above fourth rule, the position r is equivalent to the position of

Author Proof

the $(c_{L[1]}+1)$-th occurrence of the symbol $L[1]$ in $S[i..|S|]$. To quickly compute r, the functions *rank* and *select* will be used.

Let $\mathbb{W} = \prod_{\alpha=1}^{d}[0..n_\alpha]$ denote the domain of such vectors of counters, where for any $\overline{c} = (c_1, c_2, \ldots, c_d) \in \mathbb{W}$, c_α denotes the counter for $\alpha \in \Sigma$. Using the ideas described above, the algorithm recursively computes the extension $DIST(i, j, \overline{c})$ of $\delta(S, L)$, defined for each $i \in [1..n+1]$, $j \in [1..m+1]$, and $\overline{c} = (c_1, c_2, \ldots, c_d) \in \mathbb{W}$, as the value of $\delta(S[i..n]_{\overline{c}}, L[j..m])$, where $S[i..n]_{\overline{c}}$ is the string obtained from $S[i..n]$ by removing (i.e. ignoring) for each $\alpha \in \Sigma$ the first c_α occurrences of α from left to right.

Given this definition, $\delta(S, L) = DIST(1, 1, \overline{0})$, where $\overline{0}$ denotes the vector $(0, \ldots, 0) \in \mathbb{W}$. Given i, j, and $\overline{c}$, $DIST(i, j, \overline{c}) < +\infty$ if and only if for each symbol $\alpha \in \Sigma$ the number of considered (i.e. not removed or ignored) α symbols in $S[i..n]$ is at most the number of α symbols in $L[j..m]$. That is, $count(S, i, \alpha) - c_\alpha \leq count(L, j, \alpha)$ for all $\alpha \in \Sigma$, where $count(X, i, \alpha) = rank(X, |X|, \alpha) - rank(|X|, i-1, \alpha)$ is the number of symbols α in the string $X[i..|X|]$. In the following, we show how to compute $DIST(i, j, \overline{c})$ recursively for every i, j, and $\overline{c}$. For a given $\alpha \in \Sigma$, let $\overline{w}_\alpha \in \mathbb{W}$ be the vector whose components are all equal to zero except the α-th component that is equal to 1.

3.2 Recursive Computation of $DIST(i, j, \overline{c})$

We will use the following observation which considers the `swap` operations performed in the optimal transformation from a short string S of length n to a larger string L of length m.

Observation 1 ([1, 5]). *The `swap` operations used in any optimal solution satisfy the following properties: two equal symbols cannot be swapped; each symbol is always swapped in the same direction in the string; and if some symbol is moved from some position to another by performing `swaps` operations, then no symbol equal to it can be inserted afterwards between these two positions.*

The following lemma deals with the basic case where $S[i..n]$ and $L[j..m]$ start with the same symbol, i.e. $S[i] = L[j]$. When the beginnings of both strings are the same, matching those two symbols seems like an obvious choice in order to minimize the distance, but one must be careful to check first if the first symbol from $S[i..n]$ has not been scheduled to be "swapped" to an earlier position, in which case it must be ignored and skipped:

Lemma 1. *Given two strings S and L over the alphabet Σ, for any positions $i \in [1..n]$ in S and $j \in [1..m]$ in L, for any vector of counters $\overline{c} = (c_1, \ldots, c_d) \in \mathbb{W}$ and for any symbol $\alpha \in \Sigma$,*

$$\left.\begin{array}{l} S[i] = L[j] = \alpha \\ c_\alpha = 0 \end{array}\right\} \implies DIST(i, j, \overline{c}) = DIST(i+1, j+1, \overline{c}).$$

Proof. Given strings X, Y in the alphabet Σ, and an integer k, Abu-Khzam et al. [1, Corollary1] proved that if $X[1] = Y[1]$, then:

$$\delta(X, Y) \leq k \text{ if and only if } \delta(X[2..|X|], Y[2..|Y|]) \leq k.$$

Author Proof

Given that one option to transform X into Y with the minimum number of operations is to transform $X[2..|X|]$ into $Y[2..|Y|]$ with the minimum number of operations (matching $X[1]$ with $Y[1]$), we have:

$$\delta(X,Y) \;\leq\; \delta(X[2..|X|],Y[2..|Y|]).$$

By selecting $k = \delta(X,Y)$, we obtain the equality

$$\delta(X,Y) \;=\; \delta(X[2..|X|],Y[2..|Y|]).$$

Then, since the symbol $\alpha = S[i]$ must be considered (because $c_\alpha = 0$), and $S[i] = L[j]$, we can apply the above statement for $X = S[i..n]_{\overline{c}}$ and $Y = L[j..m]$ to obtain the next equalities:

$$DIST(i,j,\overline{c}) = \delta(X,Y) \;=\; \delta(X[2..|X|],Y[2..|Y|]) = DIST(i+1,j+1,\overline{c}).$$

The result thus follows. □

The second simplest case is when the first available symbol of $S[i..n]$ is already matched (through `swaps`) to a symbol from $L[1..j-1]$. The following lemma shows how to simply skip such a symbol:

Lemma 2. *Given S and L over the alphabet Σ, for any positions $i \in [1..n]$ in S and $j \in [1..m]$ in L, and for any vector of counters $\overline{c} = (c_1,\ldots,c_d) \in \mathbb{W}$ and for any symbol $\alpha \in \Sigma$,*

$$\left.\begin{array}{l} S[i] = \alpha \\ c_\alpha > 0 \end{array}\right\} \implies DIST(i,j,\overline{c}) = DIST(i+1,j,\overline{c} - \overline{w}_\alpha).$$

Proof. Since $c_\alpha > 0$, the first c_α symbols α of $S[i..n]$ have been ignored, thus $S[i]$ is ignored. Then, $DIST(i,j,\overline{c})$ must be equal to $DIST(i+1,j,\overline{c}-\overline{w}_\alpha)$, case in which $c_\alpha - 1$ symbols α of $S[i+1..n]$ are ignored. □

The most important case is when the first symbols of $S[i..n]$ and $L[j..m]$ do not match: the minimum "path" from S to L can then start either by an `insertion` or a `swap` operation.

Lemma 3. *Given S and L over the alphabet Σ, for any positions $i \in [1..n]$ in S and $j \in [1..m]$ in L, and for any vector of counters $\overline{c} = (c_1,\ldots,c_d) \in \mathbb{W}$, note $\alpha,\beta \in \Sigma$ the symbols $\alpha = S[i]$ and $\beta = L[j]$, r the position $r = select(S, rank(S,i,\beta) + c_\beta + 1, \beta)$ in S of the $(c_\beta+1)$-th symbol β of $S[i..n]$, and Δ the number $\sum_{\theta=1}^{d} \min\{c_\theta, rank(S,r,\theta) - rank(S,i-1,\theta)\}$ of symbols ignored in $S[i..r]$.*

If $\alpha \neq \beta$ and $c_\alpha = 0$, then $DIST(i,j,\overline{c}) = \min\{d_{ins}, d_{swaps}\}$, where

$$d_{ins} \;=\; \begin{cases} DIST(i,j+1,\overline{c}) + 1 & \text{if } c_\beta = 0 \\ +\infty & \text{if } c_\beta > 0 \end{cases}$$

and

$$d_{swaps} \;=\; \begin{cases} (r-i) - \Delta + DIST(i,j+1,\overline{c} + \overline{w}_\beta) & \text{if } r \neq 0 \\ +\infty & \text{if } r = 0. \end{cases}$$

Author Proof

Proof. Let $S'[1..n'] = S[i..n]_{\overline{c}}$. Given that $\alpha \neq \beta$ and $c_\alpha = 0$, there are two possibilities for $DIST(i, j, \overline{c})$: (1) transform $S'[1..n']$ into $L[j+1..m]$ with the minimum number of operations, and after that insert a symbol β at the first position of the resulting $S'[1..n']$; or (2) swap the first symbol β in $S'[2..n']$ from left to right from its current position r' to the position 1 performing $r'-1$ `swaps`, and then transform the resulting $S'[2..n']$ into $L[j+1..m]$ with the minimum number of operations. Observe that option (1) can be performed if and only if there is no symbol β ignored in $S[i..n]$ (see Observation 1). If this is the case, then $DIST(i, j, \overline{c}) = DIST(i, j+1, \overline{c}) + 1$. Option (2) can be used if and only if there is a non-ignored symbol β in $S[i..n]$, where the first one from left to right is precisely at position $r = select(S, rank(S, i, \beta) + c_\beta + 1, \beta)$. In such a case $r' = (r - i + 1) - \Delta$, where $\Delta = \sum_{\theta=1}^{d} \min\{c_\theta, rank(S, r, \theta) - rank(S, i-1, \theta)\}$ is the total number of ignored symbols in the string $S[i..r]$. Hence, the number of swaps counts to $r' - 1 = (r - i) - \Delta$. Then, the correctness of d_{ins}, d_{swaps}, and the result follow. □

The next two lemmas deal with the cases where one string is completely processed. When L has been completely processed, either the remaining symbols in S have all previously been matched via `swaps` and the distance equals zero, or there is no sequence of operations correcting S into L:

Lemma 4. *Given S and L over the alphabet Σ, for any positions $i \in [1..n+1]$ in S and $j \in [1..m]$ in L, for any vector of counters $\overline{c} = (c_1, \ldots, c_d) \in \mathbb{W}$,*

$$DIST(i, m+1, \overline{c}) = \begin{cases} 0 & \text{if } c_1 + \ldots + c_d = n - i + 1 \text{ and} \\ +\infty & \text{otherwise.} \end{cases}$$

Proof. Note that $DIST(i, m+1, \overline{c})$ is the minimum number of operations to transform the string $S[i..n]$ into the empty string $L[m+1..m]$. This number is null if and only if all the $n - i + 1$ symbols of $S[i..n]$ have been ignored, that is, $c_1 + \ldots + c_d = n - i + 1$. If not all the symbols have been ignored, then such a transformation does not exist and $DIST(i, m+1, \overline{c}) = +\infty$. □

When S has been completely processed, there are only insertions left to perform: the distance can be computed in constant time, and the list of corrections in linear time.

Lemma 5. *Given S and L over the alphabet Σ, for any position $j \in [1..m+1]$ in L, and for any vector of counters $\overline{c} = (c_1, \ldots, c_d) \in \mathbb{W}$,*

$$DIST(n+1, j, \overline{c}) = \begin{cases} m - j + 1 & \text{if } \overline{c} = \overline{0} \text{ and} \\ +\infty & \text{otherwise.} \end{cases}$$

Proof. Note that $DIST(i, m+1, \overline{c})$ is the minimum number of operations to transform the empty string $S[n+1..n]$ into the string $L[j..m]$. If $\overline{c} = \overline{0}$, then $DIST(n+1, j, \overline{c}) < +\infty$ and the transformation consists of only insertions which are $m - j + 1$. If $\overline{c} \neq \overline{0}$, then $DIST(n+1, j, \overline{c}) = +\infty$. □

Author Proof

Algorithm $DIST(i, j, \overline{c} = (c_1, \ldots, c_d))$
1. **if** $DIST(i, j, \overline{c}) = +\infty$ **then**
2. **return** $+\infty$
3. **else if** $i = n + 1$ **then**
4. (* insertions *)
5. **return** $m - j + 1$
6. **else if** $j = m + 1$ **then**
7. (* skip all symbols since they were ignored *)
8. **return** 0
9. **else**
10. $\alpha \leftarrow S[i]$, $\beta \leftarrow L[j]$
11. **if** $c_\alpha > 0$ **then**
12. (* skip $S[i]$, it was ignored *)
13. **return** $DIST(i + 1, j, \overline{c} - \overline{w}_\alpha)$
14. **else if** $\alpha = \beta$ **then**
15. (* $S[i]$ and $L[j]$ match *)
16. **return** $DIST(i + 1, j + 1, \overline{c})$
17. **else**
18. $d_{ins} \leftarrow +\infty$, $d_{swaps} \leftarrow +\infty$
19. **if** $c_\beta = 0$ **then**
20. (* insert a β at index i *)
21. $d_{ins} \leftarrow 1 + DIST(i, j + 1, \overline{c})$
22. $r \leftarrow select(S, rank(S, i, \beta) + c_\beta + 1, \beta)$
23. **if** $r \neq null$ **then**
24. $\Delta \leftarrow \sum_{\theta=1}^{d} \min\{c_\theta, rank(S, r, \theta) - rank(S, i - 1, \theta)\}$
25. (* swaps *)
26. $d_{swaps} \leftarrow (r - i) - \Delta + DIST(i, j + 1, \overline{c} + \overline{w}_\beta)$
27. **return** $\min\{d_{ins}, d_{swaps}\}$

Fig. 1. Informal algorithm to compute $DIST(i, j, \overline{c})$: Lemma 4 and Lemma 5 guarantee the correctness of lines 1 to 8; Lemma 2 guarantees the correctness of lines 11 to 13; Lemma 1 guarantees the correctness of lines 14 to 16; and Lemma 3 guarantees the correctness of lines 18 to 27.

3.3 Complexity Analysis

Combining Lemmas 1 to 5, the value of $DIST(1, 1, \overline{0})$ can be computed recursively, as shown in the algorithm of Figure 1. We analyze the formal complexity of this algorithm in Theorem 1, in the finest model that we can define, taking into account the relation for each symbol $\alpha \in \Sigma$ between the number n_α of occurrences of α in S and the number m_α of occurrences of α in L.

Theorem 1. *Given two strings S and L over the alphabet Σ, for each symbol $\alpha \in \Sigma$, note n_α the number of occurrences of α in S and m_α the number of occurrences of m in L, their sums $n = n_1 + \cdots + n_d$ and $m = m_1 + \cdots + m_d$, and $g_\alpha = \min\{n_\alpha, m_\alpha - n_\alpha\}$ a measure of how far n_α is from $m_\alpha/2$. There is an algorithm computing the* Swap-Insert Correction *distance $\delta(S, L)$ in time*

Author Proof

within $O(d+m)$ if S and L have no symbol in common, and otherwise in time within

$$O\left(d(n+m) + d^2 n \cdot \sum_{\alpha=1}^{d}(m_\alpha - g_\alpha) \cdot \prod_{\alpha \in \Sigma_+}(g_\alpha + 1)\right),$$

where $\Sigma_+ = \{\alpha \in \Sigma : g_\alpha > 0\}$ if $g_\alpha = 0$ for any $\alpha \in \Sigma$, and $\Sigma_+ = \Sigma \setminus \{\arg\min_{\alpha\in\Sigma} g_\alpha\}$ otherwise.

Proof. Observe first that there is a reordering of $\Sigma = [1..d]$ such that $0 < g_1 \leq g_2 \leq \cdots \leq g_s$ and $g_{s+1} = g_{s+2} = \cdots = g_d$ for some index $s \in [0..d]$, and we assume such an ordering from now on. Note also that given any string $X \in \{S, L\}$, a simple 2-dimensional array using space within $O(d \cdot |X|)$ can be computed in time within $O(d \cdot |X|)$, to support the queries $rank(X, i, \alpha)$ and $select(X, k, \alpha)$ in constant time for all values of $i \in [1..n]$, $k \in [1..|X|]$, and $\alpha \in \Sigma$.

The case where the two strings S and L have no symbol in common is easy: the distance is then $+\infty$. The algorithm detects this case by testing if $g_\alpha = 0$ for all $\alpha \in \Sigma$, in time within $O(d+m)$.

Consider the algorithm of Figure 1, and let $i \in [1..n]$, $j \in [1..m]$, and $\overline{c} = (c_1, \ldots, c_d)$ be parameters such that $DIST(i, j, \overline{c}) < +\infty$.

At least one of the $c_1, \ldots, c_d$ is equal to zero: in the first entry $DIST(1, 1, \overline{0})$ all the counters $c_1, c_2, \ldots, c_d$ are equal to zero, and any counter is incremented only at line 26, in which another counter must be equal to zero because of the lines 11 and 14.

The number of `insertions` counted in line 21, in previous calls to the function $DIST$ in the recursion path from $DIST(1, 1, \overline{0})$ to $DIST(i, j, \overline{c})$, is equal to $j - i - (c_1 + \cdots + c_d)$. Let t_α denote the number of such insertions for the symbol $\alpha \in \Sigma$. Then, we have

$$j \;=\; i + (c_1 + \cdots + c_d) + (t_1 + \cdots + t_d),$$

and for all $\alpha \in \Sigma$, $c_\alpha \leq n_\alpha$, $t_\alpha \leq m_\alpha - n_\alpha$, and

$$c_\alpha + t_\alpha \;=\; rank(L, j-1, \alpha) - rank(S, i-1, \alpha).$$

Using the above observations, we encode all entries $DIST(i, j, \overline{c})$, for i, j and $\overline{c}$ such that $DIST(i, j, \overline{c}) < +\infty$, into the following table T of $s + 2 \leq d + 2$ dimensions. If we have $s = d$, then

$$T[p, i, k, r_1, \ldots, r_{d-1}] \;=\; DIST(i, j, \overline{c} = (c_1, \ldots, c_d)),$$

where

$$\begin{aligned}
c_p &= 0, \\
(r_1, \ldots, r_{d-1}) &= (x_1, \ldots, x_{p-1}, x_{p+1}, \ldots, x_d) \\
x_\alpha &= \begin{cases} c_\alpha \text{ if } & n_\alpha \leq m_\alpha - n_\alpha \\ t_\alpha \text{ if } & m_\alpha - n_\alpha < n_\alpha \end{cases} \text{ for every } \alpha \in \Sigma, \text{ and} \\
k &= (c_1 + \cdots + c_d) + (t_1 + \cdots + t_d) - (r_1 + \cdots + r_{d-1}).
\end{aligned}$$

Author Proof

Furthermore, given any combination of values $i, j, c_1, \ldots, c_d$ we can switch to the values $p, i, k, r_1, \ldots, r_{d-1}$, and vice versa, in time within $O(d)$. Otherwise, if $s < d$, then

$$T[i, k, r_1, \ldots, r_s] \;=\; DIST(i, j, \overline{c} = (c_1, \ldots, c_d)),$$

where $(r_1, \ldots, r_s) = (x_1, \ldots, x_s)$. Again, given the values $i, j, c_1, \ldots, c_d$ we can switch to the values $i, k, r_1, \ldots, r_s$, and vice versa, in $O(d)$ time.

Since $p \in [1..d]$, $i \in [1..n+1]$, $k \in [0..\sum_{\alpha=1}^{d}(m_\alpha - g_\alpha)]$, and $r_\alpha \in [0..g_\alpha]$ for every α, the table T can be as large as $d \times (n+1) \times (1+\sum_{\alpha=1}^{d}(m_\alpha - g_\alpha)) \times (g_2+1) \times \cdots \times (g_d+1)$ if $s = d$, and as large as $(n+1) \times (1+\sum_{\alpha=1}^{d}(m_\alpha - g_\alpha)) \times (g_1+1) \times \cdots \times (g_s+1)$ if $0 < s < d$. For $s = 0$, no table is needed. The running time of this new algorithm includes the $O(d(n+m)) = O(dm)$ time for processing each of S and L for *rank* and *select*, and the time to compute $DIST(1, 1, \overline{0})$ which is within $O(d)$ times $n+m$ plus the number of cells of the table T. If $s = d$, the time to compute $DIST(1, 1, \overline{0})$ is within

$$O\left(d(n+m) + d^2 n \cdot \sum_{\alpha=1}^{d}(m_\alpha - g_\alpha) \cdot (g_2+1) \cdot \cdots \cdot (g_d+1)\right).$$

Otherwise, if $0 \leq s < d$, the time to compute $DIST(1, 1, \overline{0})$ is within

$$O\left(d(n+m) + dn \cdot \sum_{\alpha=1}^{d}(m_\alpha - g_\alpha) \cdot (g_1+1) \cdot \cdots \cdot (g_s+1)\right).$$

The result follows by noting that: if $s = d$, then $\Sigma_+ = \{2, \ldots, d\}$. Otherwise, if $s < d$, then $\Sigma_+ = \{1, \ldots, s\}$. □

The result above, about the complexity in the worst case over instances with $d, n_1, \ldots, n_d, m_1, \ldots, m_d$ fixed, implies results in less precise models, such as in the worst case over instances for d, n, m fixed:

Corollary 1. *Given two strings S and L over the alphabet Σ, of respective sizes n and m, the algorithm analyzed in Theorem 1 computes the* Swap-Insert Correction *distance $\delta(S, L)$ in time within*

$$O\left(d(n+m) + d^2 n(m-n)\left(\frac{n}{d-1}+1\right)^{d-1}\right),$$

which is within $O\left(n+m+n^d(m-n)\right)$ for alphabets of fixed size d; and within

$$O\left(d(n+m) + d^2 n^2\left(\frac{m-n}{d-1}+1\right)^{d-1}\right),$$

which is within $O\left(n+m+n^2(m-n)^{d-1}\right)$ for alphabets of fixed size d.

Author Proof

4 Discussion

The exact running time of our algorithm is within

$$O\left(d(n+m) + d^2 n \cdot \sum_{\alpha=1}^{d} (m_\alpha - g_\alpha) \cdot \prod_{\alpha \in \Sigma_+} (g_\alpha + 1)\right),$$

where n_α and m_α are the respective number of occurrences of symbol $\alpha \in [1..d]$ in S and L respectively; where the vector formed by the values $g_\alpha = \min\{n_\alpha, m_\alpha - n_\alpha\}$ measures the distance between $(n_1, \ldots, n_\sigma)$ and $(m_1, \ldots, m_\sigma)$; and where $\Sigma_+ = \{\alpha \in \Sigma : g_\alpha > 0\}$ if $g_\alpha = 0$ for any $\alpha \in \Sigma$, and $\Sigma_+ = \Sigma \setminus \{\arg\min_{\alpha \in \Sigma} g_\alpha\}$ otherwise.

Summarizing the disequilibrium between the frequency distributions of the symbols in the two strings via the measure $g = \max_{\alpha \in \Sigma} g_\alpha \leq n$, this yields a complexity within $O(d^2 nmg^{d-1})$, which is polynomial in n and m, and exponential only in d of base g. Since this disequilibrium g is smaller than the length n of the smallest string S, this implies a worst case complexity within $O(d^2 mn^d)$ over instances formed by strings of lengths n and m over an alphabet of size d, a result matching the state of the art [4] for this problem.

4.1 Implicit Results

The result from Theorem 1 implies the following additional results:

Weighted Operators: Wagner and Fisher [7] considered variants where the cost c_{ins} of an `insertion` and the cost c_{swap} of an `swap` are distinct. In the Swap-Insert Correction problem, there are always $n - m$ `insertions`, and always $\delta(S, L) - n + m$ `swaps`, which implies the optimality of the algorithm we described in such variants.

Implied Improvements When Only Swaps Are Needed: Abu-Khzam et al. [1] mention an algorithm computing the Swap String-to-String Correction distance (i.e. only `swaps` are allowed) in time within $O(n^2)$. This is a particular case of the Swap-Insert Correction distance, which happens exactly when the two strings are of the same size $n = m$ (and no `insertion` is neither required nor allowed). In this particular case, our algorithm yields a solution running in time within $O(dm)$, hence improving on Abu-Khzam et al.'s solution [1].

Effective Alphabet: Let d' be the effective alphabet of the instance, i.e. the number of symbols α of $\Sigma = [1..d]$ such that the number of occurrences of α in S is a constant fraction of the number of occurrences of α in L (i.e. $n_\alpha \in \Theta(m_\alpha)$). Our result implies that the real difficulty is d' rather than d, i.e. that even for a large alphabet size d the distance can still be computed in reasonable time if d' is finite.

Author Proof

4.2 Perspectives

Those results suggest various directions for future research:

Further Improvements of the Algorithm: our algorithm can be improved further using a lazy evaluation of the min operator on line 27, so that the computation in the second branch of the execution stops any time the computed distance becomes larger than the distance computed in the first branch. This would save time in practice, but it would not improve the worst-case complexity in our analysis, in which both branches are fully explored: one would require a finer measure of difficulty to express how such a modification could improve the complexity of the algorithm

Further Improvements of the Analysis: The complexity of Abu-Khzam et al.'s algorithm [1], sensitive to the distance from S to L, is an orthogonal result to ours. An algorithm simulating both their algorithm and ours in parallel yields a solution adaptive to both measures, but an algorithm using both techniques in synergy would outperform both on some instances, while never performing worse on other instances.

Adaptivity for other Existing Distances: Can other String-to-String Correction distances be computed faster when the number of occurrences of symbols in both strings are similar for most symbols? Edit distances such as when only insertions or only deletions are allowed are linear anyway, but more complex combinations require further studies.

Acknowledgement. The authors would like to thank the anonymous referees of SPIRE 2015 for insightful comments.

References

1. Abu-Khzam, F.N., Fernau, H., Langston, M.A., Lee-Cultura, S., Stege, U.: Charge and reduce: A fixed-parameter algorithm for String-to-String Correction. Discrete Optimization (DO) **8**(1), 41–49 (2011)
2. Barbay, J., Pérez-Lantero, P.: Adaptive computation of the Swap-Insert Edition Distance. arXiv preprint arXiv:1504.07298 (2015)
3. Cormen, T.H., Leiserson, C.E., Rivest, R. L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press (2009)
4. Meister, Daniel: Using swaps and deletes to make strings match. Theoretical Computer Science (TCS) **562**, 606–620 (2015)
5. Spreen, T.D.: The Binary String-to-String Correction Problem. Master's thesis, University of Victoria, Canada (2013)
6. Wagner, R.A.: On the complexity of the extended String-to-String Correction Problem. In: Proceedings of the Seventh Annual ACM Symposium on Theory Of Computing (STOC), pp. 218–223. ACM (1975)
7. Wagner, R.A., Fischer, M.J.: The String-to-String Correction Problem. Journal of the ACM (JACM) **21**(1), 168–173 (1974)
8. Wagner, R.A., Lowrance, R.: An extension of the String-to-String Correction Problem. Journal of the ACM (JACM) **22**(2), 177–183 (1975)