Adaptive Computation of the Swap-Insert Correction Distance

JÉRÉMY BARBAY, Departamento de Ciencias de la Computación, Universidad de Chile, Chile
PABLO PÉREZ-LANTERO, Departamento de Matemática y Ciencia de la Computación, Universidad de Santiago, Chile

The Swap-Insert Correction distance from a string S of length n to another string L of length $m \geq n$ on the alphabet $[1..\sigma]$ 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 σ of the alphabet. We describe an algorithm computing this distance in time within $O(\sigma^2 nmg^{\sigma-1})$, where for each $\alpha \in [1..\sigma]$ there are n_α occurrences of α in S, m_α occurrences of α in L, and where $g = \max_{\alpha \in [1..\sigma]} \min\{n_\alpha, m_\alpha - n_\alpha\}$ is a new parameter of the analysis, measuring one aspect of the difficulty of the instance. The difficulty g is bounded by above by various terms, such as the length n of the shortest string S, and by the maximum number of occurrences of a single character in S ($\max_{\alpha \in [1..\sigma]} n_\alpha$). This result illustrates how, in many cases, the correction distance between two strings can be easier to compute than in the worst case scenario.

CCS Concepts: • **Theory of computation** → *Branch-and-bound*; *Dynamic programming*;

Additional Key Words and Phrases: Adaptive, dynamic programming, edit distance, insert, swap

ACM Reference format:
Jérémy Barbay and Pablo Pérez-Lantero. 2018. Adaptive Computation of the Swap-Insert Correction Distance. *ACM Trans. Algorithms* 14, 4, Article 49 (August 2018), 16 pages.
https://doi.org/10.1145/3232057

1 INTRODUCTION

Given two strings S (for "short") and L (for "long") on the alphabet $\Sigma = [1..\sigma]$ 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 (1974), 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 such as `cvs`, `svn`, or `git`.

Each distinct set of correction operators yields a distinct correction distance on strings. For instance, Wagner and Fischer (1974) 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

This work is supported by the projects CONICYT FONDECYT/Regular No. 1170366 and No. 1160543. Part of this work was presented at the conference SPIRE 2015 (Barbay and Pérez-Lantero 2015).
Authors' addresses: J. Barbay, Departamento de Ciencias de la Computación (DCC), Avenida Beauchef 851, Universidad de Chile. 837-0456 Santiago, Chile; email: jeremy@barbay.cl; P. Pérez-Lantero, Departamento de Matemática y Ciencia de la Computación (DMCC), Las Sophoras 173, Universidad de Santiago (USACH), 917-0020 Santiago, Chile; email: pablo.perez.l@usach.cl.

https://doi.org/10.1145/3232057

$$L = \text{“HORSE”}$$
$$S = \text{“BABY”}$$
$$\Sigma = \{A, B, E, H, O, R, S, Y\},\ \sigma = 8$$

α	n_α	m_α	g_α
A	1	0	0
B	2	0	0
E	0	1	0
H	0	1	0
O	0	1	0
R	0	1	0
S	0	1	0
Y	1	0	0

Fig. 1. Degenerated case where the symbol sets of the strings are disjoint, so that the correction distance is infinite: the distance can be computed in time within $O(n + \sigma)$ by counting the numbers n_α and m_α of each symbol α in each string and comparing them.

$$S = \text{“TOC._EYRRASE._TOC.”}$$
$$L = \text{“EASY_TO_CORRECT...”}$$
$$\Sigma = \{A, C, E, R, S, T, Y, .\},\ \sigma = 8$$

α	n_α	m_α	g_α
A	1	1	0
C	2	2	0
E	1	1	0
R	2	2	0
S	1	1	0
T	2	2	0
Y	1	1	0
.	3	3	0

Fig. 2. Degenerated case where only swaps are required: The distance can be computed in time within $O(\sigma n)$, as will be seen in the next sections, mainly in Section 4.1.

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 (1975) proved the NP-hardness of the Swap-Insert Correction distance, denoted $\delta(S, L)$ through this article, i.e., the correction distance when restricted to the operators `insertion` and `swap` (or, by symmetry, to the operators `deletion` and `swap`), where the `swap` operator exchanges the positions of two contiguous symbols.

The Swap-Insert Correction distance's difficulty attracted special interest, with two results of importance: Abu-Khzam et al. (2011) described an algorithm computing $\delta(S, L)$ in time within $O(1.6181^{\delta(S,L)}m)$ and Meister (2015) described an algorithm computing $\delta(S, L)$ in time polynomial in the input size when S and L are strings on a finite alphabet: its running time is $(m + 1)^{2\sigma+1} \cdot (n + 1)^2$ times some polynomial function on n and m.

The complexity of Meister's result (Meister 2015), polynomial in m of degree at least $2\sigma + 1$, is a very pessimistic upper bound of the computational complexity of the distance. At one extreme, the Swap-Insert Correction distance of strings that are completely different (e.g., their effective alphabets are disjoint, see Figure 1 for an example) can be computed in linear time (it is then infinite or close to $n + m$). At the other extreme, the Swap-Insert Correction distance between two strings that are very similar (e.g., only a finite number of symbols need to be swapped or inserted, see Figures 2 and 3 for examples) can be computed in time linear in n and σ (Abu-Khzam et al. 2011). 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., when S and L are almost of the same length, see Figure 2 for an example), and when mostly insertions are involved to transform S into L (i.e., when many symbols present in L are absent from S, see Figure 3 for an example).

$$S = \text{“CORRECT”}$$
$$L = \text{“INCORRECT”}$$
$$\Sigma = \{C, E, I, N, O, R, T\},\ \sigma = 7$$

α	n_α	m_α	g_α
C	2	2	0
E	1	1	0
I	0	1	0
N	0	1	0
O	1	1	0
R	2	2	0
T	1	1	0

Fig. 3. Degenerated case where only `insertions` are required: The distance can be computed in time within $O(n + \sigma)$, if such a situation is checked beforehand by counting the numbers n_α and m_α for each symbol α, and verifying that either $n_\alpha = 0$ or $n_\alpha = m_\alpha$ for each α.

$$S = \text{“FIRE MOTH.”}$$
$$L = \text{“MORE DIFFICULT. MUCH MORE DIFFICULT.”}$$
$$\Sigma = \{C, D, E, F, H, I, L, M, O, R, T, U, .\},\ \sigma = 13$$
$$g = \max_\alpha g_\alpha = 1$$

α	n_α	m_α	g_α
C	0	3	0
D	0	2	0
E	1	2	1
F	1	4	1
H	1	1	0
I	1	4	1
L	0	2	0
M	1	3	1
O	1	2	1
R	1	2	1
T	1	2	0
U	0	1	0
.	1	2	1

Fig. 4. A short example of the worst case where, for most symbols of the alphabet (7 over 13), 1 is moved by swap operations and 1 or 2 are `inserted`.

Hypothesis: We consider whether the Swap-Insert Correction distance $\delta(S, L)$ can be computed in time polynomial in the lengths $n = |S|$ and $m = |L|$ 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 review of previous results and techniques in Section 2, we present such an algorithm in Section 3, in several steps: the intuition behind the algorithm in Section 3.1, the formal description of the dynamic program in Section 3.2, the implementation details of the algorithm in Section 3.3, and the analysis of its complexity in Section 3.4. 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

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

in the worst case over all instances of fixed sizes n and m, with imbalance vector $(g_1, \ldots, g_\sigma)$; where $\Sigma_+ = \Sigma \setminus \{\arg\min_{\alpha \in \Sigma} g_\alpha\}$. This simplifies to within $O(\sigma^2 nmg^{\sigma-1})$ in the worst case over instances where σ, n, m, and g are fixed, assuming without loss of generality that $n \leq m$ (as the Swap-Insert Correction distance is infinite when $n > m$). See Figure 4 for an example where the imbalance g is maximized.

Operators	Complexity
Deletion	$O(m)$ [Spreen 2013]
Insertion	$O(n)$ [Spreen 2013]
Substitution	$O(n)$ [Spreen 2013]
Swap	$O(n^2)$ [Spreen 2013]
Deletion, Insertion	$O(nm)$ [Bergroth et al. 2000]
Deletion, Substitution	$O(nm)$ [Wagner 1975]
Deletion, Swap	NP-complete [Wagner and Fischer 1974]
Insertion, Substitution	$O(nm)$ [Wagner 1975]
Insertion, Swap	NP-complete [Wagner and Fischer 1974]
Substitution, Swap	$O(nm)$ [Wagner 1975]
Deletion, Insertion, Substitution	$O(nm)$ [Bergroth et al. 2000]
Deletion, Insertion, Swap	$O(nm)$ [Wagner and Fischer 1974]
Deletion, Substitution, Swap	$O(nm)$ [Wagner and Fischer 1974]
Insertion, Substitution, Swap	$O(nm)$ [Wagner and Fischer 1974]
Deletion, Insertion, Substitution, Swap	$O(nm)$ [Bergroth et al. 2000]

Fig. 5. Summary of previous results in the worst case over instances of fixed sizes, for various combinations of operators from the basic set {`Insertion`, `Deletion`, `Substitution`, `Swap`}. The fact that the Swap-Insert and Swap-Delete correction distances are the only ones being NP-complete motivated their choice as the topic of this study.

We discuss in Section 4 some implied results, such as when the operators are assigned asymmetric costs, when the algorithm is required to output the sequence of corrections, when only swaps are allowed, or when the distribution of the frequencies of the symbols is very unbalanced, and some questions left open, such as further improvements of the algorithm, of the analysis, and similar work for other existing distances.

2 BACKGROUND

In 1974, motivated by the problem of correcting typing and transmission errors, Wagner and Fischer (1974) 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.

In 1975, Wagner and Lowrance (1975) 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, 15 different variants arise when considering any given subset of those four correction operators (see Figure 5 for an array summarizing those results). Thirteen of those variants can be computed in polynomial time (Wagner 1975; Wagner and Fischer 1974; Wagner and Lowrance 1975). Concerning 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 (Wagner 1975), hence our interest. Any result (including the ones presented here) 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 2011, Abu-Khzam et al. (2011) described an algorithm computing the Swap-Delete Correction distance from a string L to a string S (and hence the Swap-Insert Correction from S to L). Their algorithm decides if this distance is at most a given parameter k, in time within $O(1.6181^k m)$. As $\sum_{k=0}^{K} \alpha^k = \frac{\alpha^{K+1}-1}{\alpha-1} \in O(\alpha^k)$, this indirectly yields an algorithm computing both distances in time within $O(1.6181^{\delta(S,L)} m)$: testing values of k from 0 to infinity in increasing order yields an algorithm computing the distance in time within $O(\sum_{k=0}^{\delta(S,L)} 1.6181^k m) = O(1.6181^{\delta(S,L)} m)$. Since any correct algorithm must verify the correctness of its output, such an algorithm implies the existence of an algorithm with the same running time, which outputs a minimum sequence of corrections from S to L. Later in 2013, Watt (2013) showed that computing the Swap-Deletion Correction distance has a kernel size within $O(k^4)$.

In 2013, Spreen (2013) observed that Wagner's NP-hardness proof (Wagner 1975) was based on unbounded alphabet sizes (i.e., the Swap-Insert Correction problem is NP-hard when the size σ 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 (2015) extended Spreen's work (2013) 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 $\sigma \geq 2$, in time polynomial in n and m. Meister proved that the time complexity of this algorithm is equal to $(m+1)^{2\sigma+1} \cdot (n+1)^2$ times some function polynomial in n and m.

We describe in the following section an algorithm computing the Swap-Insert Correction distance in explicit polynomial time, and where the running time goes gradually down to linear for easier cases.

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, the full code of the algorithm in Section 3.3, and the formal analysis of its complexity in Section 3.4.

3.1 High-Level Description

The algorithm runs through S and L from left to right, building a mapping from the symbols of S to a subset of the symbols of L, using the fact that, for each distinct symbol, the mapping function on positions is monotone.

For every string $X \in \{S, L\}$ and integer $i \in [1..|X|]$, $X[i]$ denotes the i-th symbol of X from left to right. For every pair of integers $i, j \in [1..|X|]$ such that $i \leq j$, $X[i..j]$ denotes the substring of X from the i-th symbol to the j-th symbol, and for every pair of integers $i, j \in [1..|X|]$ such that $j < i$, $X[i..j]$ denotes the empty string. Given a symbol $\alpha \in \Sigma$, an integer $i \in [1..|X|]$ and an integer $k > 0$, $rank(X, i, \alpha)$ denotes the number of occurrences of the symbol α in the substring $X[1..i]$, and $select(X, k, \alpha)$ denotes 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 swaps) the first symbol of the part of S that is not scanned and is 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, described in Figure 6.

(1) **if** S **is empty**: return the length $|L|$ of L (only insertions are performed in S);
(2) **if some** $\alpha \in \Sigma$ **appears more times in** S **than in** L: return $+\infty$ (the superfluous occurrences cannot be removed from S);
(3) **if** S **and** L **are not empty and** $S[1] = L[1]$: return $\delta(S[2..|S|], L[2..|L|])$;
(4) **if** S **and** L **are not empty and** $S[1] \neq L[1]$:
— 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 $L[1]$, where r denotes the position of such a symbol, and S' the string resulting from S by removing that symbol.
— return $\min\{d_{ins}, d_{swaps}\}$.

Fig. 6. The essence of the dynamic program in four rules.

There can be several overlapping subproblems in the recursive definition of $\delta(S, L)$ described above, which calls for *dynamic programming* (Cormen et al. 2009) and *memoization*.[1] As is usual in *dynamic programming*, the implementation is separated between the function $\texttt{DIST}(i, j)$ (described in Figure 8), corresponding to the dynamic programming aspect of the computation, and the function $\texttt{DISTREC}(i, j, c)$ (described in Figure 9), maintaining the counters c of symbols already "used" in previous recursive calls. In any call $\delta(S', L')$ during the recursive computation of $\delta(S, L)$, the string L' is always a suffix $L[j..|L|]$ for some $j \in [1..|L|]$, and can thus be replaced by such an index j, but this is not always the case for the string S', which can be any substring of S. 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 in $S[i..|S|]$. 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 (see Figure 6 for the list of rules), the position r is equivalent to the position of the $(c_{L[1]} + 1)$-th occurrence of the symbol $L[1]$ in $S[i..|S|]$. The index supporting the functions *rank* and *select* permits us to compute r efficiently.

Let $\mathbb{W} = \prod_{\alpha=1}^{\sigma} [0..n_\alpha]$ denote the domain of such vectors of counters, where for any $\mathbf{c} = (c_1, c_2, \ldots, c_\sigma) \in \mathbb{W}$, c_α denotes the counter for $\alpha \in \Sigma$. Using the ideas described above, the algorithm recursively computes the function $\texttt{DISTREC}(i, j, \mathbf{c})$ defined for each $i \in [1..n+1]$, $j \in [1..m+1]$, and $\mathbf{c} = (c_1, c_2, \ldots, c_\sigma) \in \mathbb{W}$, as the value of $\delta(S[i..n]_{\mathbf{c}}, L[j..m])$, where $S[i..n]_{\mathbf{c}}$ 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, $\delta(S, L) = \texttt{DISTREC}(1, 1, \mathbf{0})$, where $\mathbf{0}$ denotes the vector $(0, \ldots, 0) \in \mathbb{W}$. Given i, j, and $\mathbf{c}$, $\texttt{DISTREC}(i, j, \mathbf{c}) < +\infty$ if and only if for each symbol $\alpha \in \Sigma$ the number of α symbols in $S[i..n]$ considered (that is symbols that are neither removed nor ignored) 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 $\texttt{DISTREC}(i, j, \mathbf{c})$ recursively for every i, j,

[1]Cormen et al. (2009) explain that *memoization* comes from *memo*, referring to the fact that the technique consists of recording a value so that we can look it up later.

and $\mathbf{c}$. For a given $\alpha \in \Sigma$, let $\mathbf{u}_\alpha \in \mathbb{W}$ be the vector whose components are all equal to zero except the α-th component, which is equal to 1.

3.2 Recursive Computation of DISTREC($i, j, \mathbf{c}$)

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 ((Abu-Khzam et al. 2011; Spreen 2013)). *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.*

This observation is essential to the design of the algorithm: while a text index over S permits us to decide if the "missing" symbol at the end of if can be "brought" via swaps to the current position, the observation above permits us to "schedule" exactly which occurrence of this symbol (the closest unused one on the left of the current position) to "schedule" for such a sequence of swaps.

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 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 3.1. *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 $\mathbf{c} = (c_1, \ldots, c_\sigma) \in \mathbb{W}$ and for any symbol $\alpha \in \Sigma$,*

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

Proof. Given strings X, Y in the alphabet Σ, and an integer k, Abu-Khzam et al. (2011) proved that if $X[1] = Y[1]$, then

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

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

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

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 $X = S[i..n]_{\mathbf{c}}$ and $Y = L[j..m]$ to obtain the next equalities:

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

The result thus follows. □

The second simplest case is when the first symbol $S[i]$ 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 3.2. *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 $\mathbf{c} = (c_1, \ldots, c_\sigma) \in \mathbb{W}$ and for any symbol $\alpha \in \Sigma$,*

$$\left.\begin{array}{l} S[i] = \alpha \\ c_\alpha > 0 \end{array}\right\} \Rightarrow \texttt{DISTREC}(i, j, \mathbf{c}) = \texttt{DISTREC}(i+1, j, \mathbf{c} - \mathbf{u}_\alpha).$$

Proof. Since $c_\alpha > 0$, the first c_α symbols α of $S[i..n]$ have been ignored, thus $S[i]$ is ignored. Then, $\texttt{DISTREC}(i, j, \mathbf{c})$ must be equal to $\texttt{DISTREC}(i+1, j, \mathbf{c} - \mathbf{u}_\alpha)$, 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 3.3. *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 $\mathbf{c} = (c_1, \ldots, c_\sigma) \in \mathbb{W}$, 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 $\sum_{\theta=1}^{\sigma} \min\{c_\theta, rank(S, r, \theta) - rank(S, i-1, \theta)\}$ of symbols ignored in $S[i..r]$.*

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

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

and

$$d_{swaps} = \begin{cases} (r - i) - \Delta + \texttt{DISTREC}(i, j+1, \mathbf{c} + \mathbf{u}_\beta) & \text{if } r \neq null \\ +\infty & \text{if } r = null. \end{cases}$$

Proof. Let $S'[1..n'] = S[i..n]_{\mathbf{c}}$. Given that $\alpha \neq \beta$ and $c_\alpha = 0$, there are at most two possibilities for $\texttt{DISTREC}(i, j, \mathbf{c})$: (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 $\texttt{DISTREC}(i, j, \mathbf{c}) = \texttt{DISTREC}(i, j+1, \mathbf{c}) + 1$. 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 $\Delta = \sum_{\theta=1}^{\sigma} \min\{c_\theta, rank(S, r, \theta) - rank(S, i-1, \theta)\}$ 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. Given a call $\texttt{DISTREC}(i, j, \mathbf{c})$, we say that L has been *completely processed* if $j = m + 1$, that is, j reached the end of L. Similarly, we say that S has been *completely processed* if $i = n + 1$.

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 3.4. *Given S and L over the alphabet Σ, for any position $i \in [1..n+1]$ in S, and for any vector of counters $\mathbf{c} = (c_1, \ldots, c_\sigma) \in \mathbb{W}$,*

$$\texttt{DISTREC}(i, m+1, \mathbf{c}) = \begin{cases} 0 & \text{if } c_1 + \cdots + c_\sigma = n - i + 1 \text{ and} \\ +\infty & \text{otherwise.} \end{cases}$$

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

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

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

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

3.3 Complete Algorithm

In the following, we describe the formal algorithm to compute $\texttt{DISTREC}(i, j, \mathbf{0})$. We consider the worst scenario for the running time of Theorem 3.6, where for each symbol $\alpha \in \Sigma$ we have $g_\alpha > 0$. The other cases in which $g_\alpha > 0$ is not satisfied for all $\alpha \in \Sigma$ are easier to implement. Note that line 3 of the algorithm `Compute` (Figure 8) and line 26 of the algorithm `DIST` (Figure 7) guarantee that $\texttt{DIST}(i, j) = \texttt{DISTREC}(i, j, \mathbf{c}) < +\infty$ in every call of `DIST`. Further, the counters $(c_1, c_2, \ldots, c_\sigma)$ are global variables to the recursive algorithm `DIST`.

3.4 Complexity Analysis

Combining Lemmas 3.1 to 3.5, the value of $\texttt{DISTREC}(1, 1, \mathbf{0})$ can be computed recursively, as shown in the algorithm of Figure 7. We analyze the formal complexity of this algorithm in Theorem 3.6, in the finest model that we propose, 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 3.6. *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 $n = n_1 + \cdots + n_\sigma$ and $m = m_1 + \cdots + m_\sigma$, 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 within $O(\sigma + m)$ if S and L have no symbol in common, and otherwise in time within*

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

where $\Sigma_+ = \Sigma \setminus \{\arg\min_{\alpha \in \Sigma} g_\alpha\}$.

Proof. 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 $O(\sigma + m)$.

For the general case, observe first that there is a reordering of $\Sigma = [1..\sigma]$ such that $0 < g_1 \leq g_2 \leq \cdots \leq g_s$ and $g_{s+1} = g_{s+2} = \cdots = g_\sigma = 0$ for some index $s \in [0..\sigma]$, and we assume such an ordering from now on. Note also that given any string $X \in \{S, L\}$, a simple two-dimensional array using space within $O(\sigma \cdot |X|)$ can be computed in time within $O(\sigma \cdot |X|)$, 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 $\alpha \in \Sigma$.

Algorithm DISTREC$(i, j, \mathbf{c} = (c_1, \ldots, c_\sigma))$
1. **if** $(j = m + 1$ **and** $c_1 + \cdots + c_\sigma < n - i + 1)$ **or** $(i = n + 1$ **and** $\mathbf{c} \neq \mathbf{0})$ **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** DISTREC$(i + 1, j, \mathbf{c} - \mathbf{u}_\alpha)$
14. **else if** $\alpha = \beta$ **then**
15. (* $S[i]$ and $L[j]$ match *)
16. **return** DISTREC$(i + 1, j + 1, \mathbf{c})$
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. $d_{ins} \leftarrow 1 +$ DISTREC$(i, j + 1, \mathbf{c})$
22. $r \leftarrow select(S, rank(S, i, \beta) + c_\beta + 1, \beta)$
23. **if** $r \neq null$ **then**
24. $\Delta \leftarrow \sum_{\theta=1}^{\sigma} \min\{c_\theta, rank(S, r, \theta) - rank(S, i - 1, \theta)\}$
25. (* swaps *)
26. $d_{swaps} \leftarrow (r - i) - \Delta +$ DISTREC$(i, j + 1, \mathbf{c} + \mathbf{u}_\beta)$
27. **return** $\min\{d_{ins}, d_{swaps}\}$

Fig. 7. Informal algorithm to compute DISTREC$(i, j, \mathbf{c})$: Lemmas 3.4 and 3.5 guarantee the correctness of lines 1 to 8; Lemma 3.2 guarantees the correctness of lines 11 to 13; Lemma 3.1 guarantees the correctness of lines 14 to 16; and Lemma 3.3 guarantees the correctness of lines 18 to 27.

Algorithm Compute $\delta(S, L)$:
1. preprocess each of S and L for $rank$ and $select$
2. $(c_1, c_2, \ldots, c_\sigma) \leftarrow \mathbf{0}$
3. **return if** DISTREC$(1, 1, \mathbf{0}) < +\infty$ **then** DIST$(1, 1)$ **else** $+\infty$

Fig. 8. Calling algorithm to compute DISTREC$(i, j, \mathbf{0})$, filtering degenerated cases before launching the real computation of the distance.

Consider the algorithm of Figure 7, and let $i \in [1..n]$, $j \in [1..m]$, and $\mathbf{c} = (c_1, \ldots, c_\sigma)$ be parameters such that DISTREC$(i, j, \mathbf{c}) < +\infty$.

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

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

$$j \;=\; i + (c_1 + \cdots + c_\sigma) + (t_1 + \cdots + t_\sigma),$$

Algorithm DIST(i, j):
1. $p \leftarrow$ the first index in $[1..\sigma]$ so that $c_p = 0$
2. **for** $\alpha = 1$ **to** σ **do**
3. **if** $n_\alpha \leq m_\alpha - n_\alpha$ **then**
4. $x_\alpha \leftarrow c_\alpha$
5. **else**
6. $x_\alpha \leftarrow rank(L, j-1, \alpha) - rank(S, i-1, \alpha) - c_\alpha$
7. $(r_1, \ldots, r_{\sigma-1}) \leftarrow (x_1, \ldots, x_{p-1}, x_{p+1}, \ldots, x_\sigma)$
8. $k \leftarrow j - i - (r_1 + \ldots + r_{\sigma-1})$
9. **if** $T[p, i, k, r_1, \ldots, r_{\sigma-1}] \neq undefined$ **then**
10. **return** $T[p, i, k, r_1, \ldots, r_{\sigma-1}]$
11. **else**
12. **if** $i = n + 1$ **then**
13. $T[p, i, k, r_1, \ldots, r_{\sigma-1}] \leftarrow m - j + 1$
14. **else if** $j = m + 1$ **then**
15. $T[p, i, k, r_1, \ldots, r_{\sigma-1}] \leftarrow 0$
16. **else**
17. $\alpha \leftarrow S[i]$, $\beta \leftarrow L[j]$
18. **if** $c_\alpha > 0$ **then**
19. $c_\alpha \leftarrow c_\alpha - 1$
20. $T[p, i, k, r_1, \ldots, r_{\sigma-1}] \leftarrow$ DIST$(i + 1, j)$
21. $c_\alpha \leftarrow c_\alpha + 1$
22. **else if** $\alpha = \beta$ **then**
23. $T[p, i, k, r_1, \ldots, r_{\sigma-1}] \leftarrow$ DIST$(i + 1, j + 1)$
24. **else**
25. $d_{ins} \leftarrow +\infty$, $d_{swaps} \leftarrow +\infty$
26. **if** $c_\beta = 0$ **and** $count(S, i, \beta) < count(L, j, \beta)$ **then**
27. $d_{ins} \leftarrow 1 +$ DIST$(i, j + 1)$
28. $r \leftarrow select(S, rank(S, i, \beta) + c_\beta + 1, \beta)$
29. **if** $r \neq null$ **then**
30. $\Delta \leftarrow \sum_{\theta=1}^{\sigma} \min\{c_\theta, rank(S, r, \theta) - rank(S, i-1, \theta)\}$
31. $c_\beta \leftarrow c_\beta + 1$
32. $d_{swaps} \leftarrow (r - i) - \Delta +$ DIST$(i, j + 1)$
33. $c_\beta \leftarrow c_\beta - 1$
34. $T[p, i, k, r_1, \ldots, r_{\sigma-1}] \leftarrow \min\{d_{ins}, d_{swaps}\}$
35. **return** $T[p, i, k, r_1, \ldots, r_{\sigma-1}]$

Fig. 9. Formal algorithm to compute DISTREC$(i, j, \mathbf{0})$, using dynamic programming with memoization. Note that line 26 of algorithm Compute and line 3 of algorithm DIST guarantee that DIST(i, j) = DISTREC$(i, j, \mathbf{c}) < +\infty$ in every call.

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 DISTREC$(i, j, \mathbf{c})$, for i, j, and $\mathbf{c}$ such that DISTREC$(i, j, \mathbf{c}) < +\infty$, into the following table T of $s + 2 \leq \sigma + 2$ dimensions. If we have $s = \sigma$, then

$$T[p, i, k, r_1, \ldots, r_{\sigma-1}] \;=\; \text{DISTREC}(i, j, \mathbf{c} = (c_1, \ldots, c_\sigma)),$$

where

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

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

$$T[i, k, r_1, \ldots, r_s] \;=\; \texttt{DISTREC}(i, j, \mathbf{c} = (c_1, \ldots, c_\sigma)),$$

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

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

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

Otherwise, if $0 \leq s < \sigma$, the time to compute $\texttt{DISTREC}(1, 1, \mathbf{0})$ is within

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

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

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

Corollary 3.7. *Given two strings S and L over the alphabet Σ, of respective sizes n and m, the algorithm analyzed in Theorem 3.6 computes the Swap-Insert Correction distance $\delta(S, L)$ in time within*

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

which is within $O(n^\sigma m)$ for alphabets of fixed size σ; and within

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

which is within $O(n + m + nm(m-n)^{\sigma-1})$ for alphabets of fixed size σ.

Proof. We use the following two claims. The first one is: If $a \geq 1$ and $x \leq y$, then $(a+y)(x+1) \leq (a+x)(y+1)$. It can be proved as follows:

$$\begin{aligned}(a-1)x &\leq (a-1)y\\ ax+y &\leq ay+x\\ ax+y+a+xy &\leq ay+x+a+xy\\ (a+y)(x+1) &\leq (a+x)(y+1).\end{aligned}$$

The second claim is: For $a \geq 1$ and $0 \leq x_i \leq y_i$ for $i \in [1..k]$, then

$$\left(a+\sum_{i=1}^{k} y_i\right)\prod_{i=1}^{k}(x_i+1) \leq \left(a+\sum_{i=1}^{k} x_i\right)\prod_{i=1}^{k}(y_i+1)\,.$$

It can be proved by using induction and the first claim.

Let $\Sigma_+ \subset \Sigma$ be as defined in Theorem 3.6, and assume w.l.o.g. that $\Sigma \setminus \Sigma_+ = \{1\}$. Let $B = \{\beta \in \Sigma_+ \mid m_\beta - n_\beta < n_\beta\}$, and note that $g_\beta = m_\beta - n_\beta$ for all $\beta \in B$. Define a and b such that

$$a \;=\; \sum_{\alpha\in\Sigma\setminus B}(m_\alpha - g_\alpha) \geq 1 \;\text{ and }\; b = \prod_{\alpha\in\Sigma_+\setminus B}(g_\alpha+1).$$

Then, we have

$$\begin{aligned}\sum_{\alpha=1}^{\sigma}(m_\alpha-g_\alpha)\cdot\prod_{\alpha\in\Sigma_+}(g_\alpha+1) &= \left(a+\sum_{\beta\in B}(m_\beta-g_\beta)\right)\cdot b\cdot\prod_{\beta\in B}(g_\beta+1)\\ &= \left(a+\sum_{\beta\in B} n_\beta\right)\cdot b\cdot\prod_{\beta\in B}(m_\beta-n_\beta+1)\\ &\leq \left(a+\sum_{\beta\in B}(m_\beta-n_\beta)\right)\cdot b\cdot\prod_{\beta\in B}(n_\beta+1)\\ &= \left(m_1-g_1+\sum_{\alpha=2}^{\sigma}(m_\alpha-n_\alpha)\right)\cdot\prod_{\alpha\in\Sigma_+}(n_\alpha+1),\end{aligned}$$

which immediately implies

$$\sum_{\alpha=1}^{\sigma}(m_\alpha-g_\alpha)\cdot\prod_{\alpha\in\Sigma_+}(g_\alpha+1) \;\subseteq\; O\left(m\prod_{\alpha\in\Sigma_+}(n_\alpha+1)\right).$$

Then,

$$\begin{aligned}O\left(\sigma^2 n\cdot\sum_{\alpha=1}^{\sigma}(m_\alpha-g_\alpha)\cdot\prod_{\alpha\in\Sigma_+}(g_\alpha+1)\right) &\subseteq O\left(\sigma^2 nm\cdot(n_2+1)\cdot\cdots\cdot(n_\sigma+1)\right)\\ &\subseteq O\left(\sigma^2 nm\cdot\left(\frac{n_2+\cdots+n_\sigma}{\sigma-1}+1\right)^{\sigma-1}\right)\\ &\subseteq O\left(\sigma^2 nm\cdot\left(\frac{n}{\sigma-1}+1\right)^{\sigma-1}\right).\end{aligned}$$

Similarly,

$$\sum_{\alpha=1}^{\sigma}(m_\alpha - g_\alpha)\cdot\prod_{\alpha\in\Sigma_+}(g_\alpha+1) \;\subseteq\; O\left(m\prod_{\alpha\in\Sigma_+}(m_\alpha - n_\alpha+1)\right),$$

which implies the second part of the result. □

4 DISCUSSION

In 2014, Meister (2015) described an algorithm computing the Swap-Insert Correction distance from a string $S\in[1..\sigma]^n$ to another string $L\in[1..\sigma]^m$ on any fixed alphabet size $\sigma\geq 2$, in time polynomial in n and m. The algorithm that we described takes advantage of instances where for all symbols $\alpha\in\Sigma$, the number n_α of occurrences of α in S is either close to zero (i.e., most α symbols from L are placed in S through `insertions`) or close to the number m_α of occurrences of α in L (i.e., most α symbols from L are matched to symbols in S through `swaps`), while still running in time within $O(m+\min\left\{n^\sigma(m-n), n^2(m-n)^{\sigma-1}\right\})$ when the alphabet size σ is a constant, in the worst case over instances composed of strings of sizes n and m. The exact running time of our algorithm is within

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

where n_α and m_α are the respective number of occurrences of symbol $\alpha\in[1..\sigma]$ 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 $\Sigma_+=\Sigma\setminus\{\arg\min_{\alpha\in\Sigma} g_\alpha\}$.

Summarizing the imbalance 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 time complexity within $O(\sigma^2 nmg^{\sigma-1})$, which is polynomial in n and m, and exponential only in σ of base g.

This result is improving upon the state of the art (Meister 2015) even in the coarser context of the worst case over instances formed by strings of lengths n and m over an alphabet of size σ. The imbalance g is smaller than the difference $m-n$ between the lengths of the input strings, and hence than the length m of the largest string L. This implies a worst case complexity within $O(\sigma^2 nm^\sigma)$ over instances formed by strings of lengths n and m over an alphabet of size σ, which improves by several factors over the previous known complexity (Meister 2015) of $(m+1)^{2\sigma+1}\cdot(n+1)^2$ times some polynomial function on n and m.

More simply, the imbalance g is smaller than the length n of the smallest string S, which implies a worst case complexity within $O(\sigma^2 mn^\sigma)$ over instances formed by strings of lengths n and m over an alphabet of size σ.

4.1 Additional Implicit Results

The result from Theorem 3.6 implies the following additional results:

Weighted Operators. Wagner and Fischer (1974) considered variants where the cost c_{ins} of an `insertion` and the cost c_{swap} of a `swap` are distinct. The introduction of weights does not affect the solution of the Swap-Insert Correction problem, as there are always $n-m$ `insertions`, and always $\delta(S,L)-n+m$ `swaps`.

Computing the Sequence of Corrections. Since any correct algorithm must verify the correctness of its output, given a set C of correction operators, any correct algorithm computing the String-to-String Correction Distance when limited to the operators in C implies an algorithm computing a minimal sequence of corrections under the same constraints within the same asymptotic running time, albeit with an increased space usage.

Implied improvements when only swaps are needed. Abu-Khzam et al. (2011) 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 $O(\sigma n)$, hence improving on Abu-Khzam et al.'s solution (Abu-Khzam et al. 2011) when the alphabet size σ is asymptotically negligible compared to that of the smaller text (e.g., $\sigma \in o(n)$), and in particular when the alphabet size is constant.
Effective Alphabet. Let σ' be the effective alphabet of the instance, i.e., the number of symbols α of $\Sigma = [1..\sigma]$ 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 σ' rather than σ, i.e., that even for a large alphabet size σ the distance can still be computed in reasonable time if σ' is constant or very slowly growing.

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 (Abu-Khzam et al. 2011), sensitive to the distance from S to L, is an orthogonal result to the one described here. 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. As for the above, one would require a finer measure of difficulty to express the advantage of such a synergy.
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 or very different for most symbols? Edit distances when only insertions or only deletions are allowed can be computed in linear time anyway, but more complex combinations require further studies (Barbay and Olivares 2018).
Adaptive Dynamic Programming. Can similar techniques be applied to other dynamic programs? Some problems such as computing the Delete-Insert-Replace edit distance between two strings could benefit from an index to "skip" some recursive calls, in a similar way what we described for the Swap-Insert Correction distance. Some others, such as the computation of the Discrete Fréchet distance (Eiter and Mannila 1994) between two sequences of points, seem to require other techniques (Barbay 2018).

REFERENCES

F. N. Abu-Khzam, H. Fernau, M. A. Langston, S. Lee-Cultura, and U. Stege. 2011. Charge and reduce: A fixed-parameter algorithm for string-to-string Correction. *Discrete Optimization* 8, 1 (2011), 41–49.
J. Barbay. 2018. Adaptive computation of the discrete Fréchet Distance. arxiv:cs.CG/1806.01226

J. Barbay and A. Olivares. 2018. Indexed dynamic programming to boost edit distance and LCSS computation. arxiv:cs.IR/1806.04277

J. Barbay and P. Pérez-Lantero. 2015. Adaptive computation of the Swap-Insert correction distance. In *String Processing and Information Retrieval—Proceedings of the 22nd International Symposium, SPIRE 2015, London, UK, September 1-4, 2015*, Lecture Notes in Computer Science, Vol. 9309, Costas S. Iliopoulos, Simon J. Puglisi, and Emine Yilmaz (Eds.). Springer, 21–32. DOI: http://dx.doi.org/10.1007/978-3-319-23826-5_3

L. Bergroth, H. Hakonen, and T. Raita. 2000. A survey of longest common subsequence algorithms. In *Proceedings of the Symposium on String Processing and Information Retrieval (SPIRE'00)*. IEEE Computer Society, 39–48.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2009. *Introduction to Algorithms* (3rd ed.). The MIT Press.

T. Eiter and H. Mannila. 1994. *Computing Discrete Fréchet Distance*. Technical Report. Technical Report CD-TR 94/64, Christian Doppler Laboratory for Expert Systems, TU Vienna, Austria.

D. Meister. 2015. Using swaps and deletes to make strings match. *Theoretical Computer Science* 562, 0 (2015), 606–620. DOI: http://dx.doi.org/10.1016/j.tcs.2014.11.002

T. D. Spreen. 2013. *The Binary String-to-String Correction Problem*. Master's thesis. University of Victoria, Canada.

R. A. Wagner. 1975. On the complexity of the extended string-to-string correction problem. In *Proceedings of the 7th Annual ACM Symposium on Theory of Computing (STOC'75)*. ACM, 218–223.

R. A. Wagner and M. J. Fischer. 1974. The string-to-string correction problem. *Journal of the ACM* 21, 1 (1974), 168–173.

R. A. Wagner and R. Lowrance. 1975. An extension of the string-to-string correction problem. *Journal of the ACM* 22, 2 (1975), 177–183.

N. Watt. 2013. *String to String Correction Kernelization*. Master's thesis. University of Victoria, Canada.

Received November 2015; revised May 2018; accepted May 2018