Adaptive Computation of the Swap-Insert Edition Distance

Jérémry Barbay\textsuperscript{1} and Pablo Pérez-Lantero\textsuperscript{2}

\textsuperscript{1} Departamento de Ciencias de la Computación, Universidad de Chile, Chile
\textsuperscript{2} Escuela de Ingeniería Civil en Informática, Universidad de Valparaíso, Chile

Abstract. The Swap-Insert String-to-String Correction distance from a string $S$ to another string $L$ on the alphabet $[1..d]$ is the minimum number of insertions and swaps of pairs of adjacent symbols converting $S$ into $L$. In 1975, Wagner proved that its computation is NP-hard for unbounded alphabet size $d$. In 2014, Meister described a polynomial solution for bounded alphabet size, without giving its exact complexity. We describe a simpler algorithm computing this distance in time polynomial in the lengths $n$ of $S$ and $m$ of $L$, within $O(d(n + m) + d^2n \cdot \sum_{\alpha=1}^{d}(m_{\alpha} - g_{\alpha}) \cdot \prod_{\beta=2}^{d}(g_{\beta} + 1))$, when for each symbol $\alpha \in [1..d]$ there are $n_{\alpha}$ instances of $\alpha$ in $S$, $m_{\alpha}$ instances of $\alpha$ in $L$, and the vector of values $g_{\alpha} = \min\{n_{\alpha}, m_{\alpha} - n_{\alpha}\}$, assuming that $g_1 = \min_{\alpha \in [1..d]} g_{\alpha}$, measures the difficulty to compute the distance between $S$ and $L$. This is 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 simplifies to within $O(n^d(m - n) + n + m)$ when $d$ is fixed. Our solution shows adaptivity to the easier cases where, separately for each symbol $\alpha \in [1..d]$, mostly swaps are required (e.g. $m_{\alpha} - n_{\alpha}$ is small) or mostly insertions are required (e.g. $n_{\alpha}$ is small), two cases of figure not considered by previous results.

Keywords: Edit Distance, Insert, Swap, Dynamic Programming.

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 \cite{wagner1974}, 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.

Each distinct set of correction operators yields a distinct correction distance on strings. For instance, Wagner and Fischer \cite{wagner1974} showed that for the three following operations, the insertion of a symbol at some arbitrary position, the

\textsuperscript{*} Partially supported by Millennium Nucleus Information and Coordination in Networks ICM/FIC RC130003.
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 that computing the Swap-Insert String-to-String Correction distance, i.e. the correction distance when restricted to the operators insertion and swap (or, by symmetry, to the operators deletion and swap), is NP-hard.

Let $\delta(S, L)$ be the Swap-Insert String-To-String Correction distance from $S$ to $L$. This distance is infinite when $L$ is unreachable from $S$, that is, when some symbol $\alpha \in \Sigma$ occurs more times in $S$ than in $L$. Spreen [5] observed in 2013 that Wagner’s reduction [6] from the Minimum Set Cover problem [3] supposed an infinite alphabet, and conjectured that bounding the size of the alphabet would yield a polynomial-time solution. One year later, Meister [4] proved Spreen’s conjecture by describing an algorithm computing $\delta(S, L)$ in time polynomial in the input size when $S$ and $L$ are strings on a finite alphabet.

While Meister [4] closed the question of whether $\delta(S, L)$ could be computed in polynomial time, the analysis of his algorithm does not yield an explicit formula for its time complexity. The running time given is $(m + 1)^{2d+1} \cdot (n + 1)^2$ times some polynomial function on $n$ and $m$. Furthermore, the algorithm does not take advantage of cases where $\delta(S, L)$ can be computed in linear time, such as when only swaps are involved (i.e. $S$ and $L$ have the same lengths), or when many symbols $\alpha \in \Sigma$ present in $L$ are absent from $S$ (mostly insertions are involved).

We describe in Section 3 a new algorithm, with an approach different to that of Meister, to compute the Swap-Insert String-To-String Correction distance $\delta(S, L)$ in time polynomial in $n$ and $m$ (but exponential in only $d$), which takes advantage of cases where, separately for each symbol $\alpha \in \Sigma$, mostly swaps are required (e.g. $m_\alpha = n_\alpha$ is small) or mostly insertions are required (e.g. $n_\alpha$ is small), a condition measured separately for each symbol by the parameter $g_\alpha = \min\{n_\alpha, m_\alpha - n_\alpha\}$, and globally by the parameter $g = \max_{\alpha \in \Sigma} g_\alpha$.

Our solution is simpler and faster than the one described by Meister [4], and it yields an explicit analysis of the complexity, in the worst case over instances with fixed $d, n_1, \ldots, n_d, m_1, \ldots, m_d$. In turn, this analysis yields an explicit computational analysis in the worst case over instances over strings of sizes $n$ and $m$, which in turn yields an explicit computational analysis in the worst case over instances with fixed input size $n + m$.

More formally, we show (see Theorem 1) that the dynamic program described computes the Swap-Insert String-To-String Correction distance $\delta(S, L)$ in time within $O(d(n + m) + d^2 n \cdot \sum_{\alpha=1}^{d}(m_\alpha - g_\alpha) \cdot \prod_{\beta=2}^{d}(g_\beta + 1))$ assuming that $g_1 = \min_{\alpha \in [1..d]} g_\alpha$, being within $O(d(n + m) + (d/(d-1))^{d-2} \cdot n^d(m - n))$, and within $O((n + m) + n^d(m - n)$ when $d$ is fixed. This complexity is not only polynomial in $n$ and $m$, it also shows that it is possible to take advantage of “easy” instances where for each symbol there are few insertions or swaps, which results in a running time within $O(d^2 g^{d-1} nm)$ in the worst case for $d, n, n$ and $g$ fixed. We further show (Corollary 1) that in the worst case over instances
with \(d, n\) and \(m\) fixed, the above running time is within both \(O(n+m+n^d(m-n))\) and \(O(n + m + n^2(m - n)^{d-1})\).

## 2 Previous Work

In 1974, motivated by application to correct typing and transmission errors, Wagner and Fischer \[7\] introduced the \textit{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 \textbf{insertion} of a symbol at some arbitrary position,
- the \textbf{deletion} of a symbol at some arbitrary position, and
- the \textbf{substitution} of a symbol at some arbitrary position.

They gave 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, Wagner \[6\] and Lowrance and Wagner \[8\] extended the \textit{String-to-String Correction} problem (calling it the \textit{Extended String-to-String Correction} problem), where one considers not only \textit{insertions}, \textit{deletions}, and \textit{substitution} of symbols, but also the \textbf{swap} of two contiguous symbols.

The \textbf{Swap-Insert String-to-String Correction} problem is the variant of the \textit{Extended String-to-String Correction} problem when only \textit{insertions} and \textit{swaps} are considered, and it is among the fifteen different variants of the \textit{Extended String-to-String Correction} problem that arise when considering a given subset of the four correction operators. Most of them can be computed in polynomial time \([6,7,8]\), the only exception being the \textbf{Swap-Insert String-to-String Correction} problem, and its symmetric variant where the operators are limited to \textbf{deletion} and \textbf{swap}: Wagner \[6\] proved the \textit{NP-hardness} of computing those, by reduction from the \textit{Minimum Set Cover} problem \[3\].

In 2011, Abu-Khzam et al. \[1\] described an algorithm for the case where the operators are limited to \textbf{swaps} and \textbf{deletions}. Their algorithm decides if the \textbf{Swap-Deletion String-to-String Correction} distance from \(L\) to \(S\) is smaller than a parameter \(k\), in time within \(O(1.6181^k m)\). This directly yields an algorithm to decide if the \textbf{Swap-Insert String-to-String Correction} distance is smaller than a parameter \(k\) within the same complexity, as the \textbf{Swap-Deletion String-to-String Correction} distance from \(L\) to \(S\) is exactly the \textbf{Swap-Insert String-to-String Correction} distance from \(S\) to \(L\). Furthermore, it 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 \(O(\sum_{k=0}^{\delta(S,L)} 1.6181^k) \subset O(1.6181^{\delta(S,L)} m)\). Since any correct algorithm must verify the correctness of its output, such algorithm implies the existence of an algorithm with the same running time which outputs a minimum sequence of corrections from \(S\) to \(L\).
In 2013, Spreen [5] observed that Wagner’s NP-hardness proof [6] was based on unbounded alphabets (i.e. the Swap-Insert String-to-String 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 alphabets. He described some polynomial-time algorithms for various special cases when $d = 2$, and described some more general properties (that we describe further in Section 3.1).

In 2014, Meister [4] extended Spreen’s work [5] to an involved algorithm computing the Swap-Insertion String-to-String 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 $\varphi: [1..n] \rightarrow [1..m]$ 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 (see the transformation mapping of Section 3.1), and the number of insertions is always equal to $m - n$. Meister does not explicitly give the time complexity of the algorithm described, he proved that the time complexity is polynomial, being equal to $(m + 1)^{2d+1} \cdot (n + 1)^2$ times some polynomial function on $n$ and $m$.

3 Algorithm

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

3.1 Distance Properties

We list the following properties of the optimal transformation from a short string $S$ of length $n$ to a larger string $L$ of length $m$. These properties will be useful to prove the correctness of the recursive computation of $\delta(S, L)$.

1. The number of insertions is always equal to $m - n$, thus $\delta(S, L)$ equals $m - n$ plus the minimum number of swaps.
2. The swap operations used in any optimal solution satisfy the following properties [5]: two contiguous 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 other one by performing swaps operations, then no symbol equal to it can be inserted afterwards between these two positions.
3. There always exists an optimal transformation in which all swap operations are performed before any insertion \( \Pi \). Therefore, we can see an optimal transformation from \( S \) to \( L \), consisting of \( s \) swaps and \( m-n \) insertions, as a mapping \( f \) (called transformation mapping) from \([1..n]\) to a \( n \)-cardinality subset \( \{ j_1 < j_2 < \ldots < j_n \} \subseteq [1..m] \) such that \( S[i] = L[f(i)] \) for all \( i \in [1..n] \). The mapping \( f \) induces the string \( S_f = L[j_1]L[j_2] \ldots L[j_n] \) obtained by permuting the symbols of \( S \) using precisely \( s \) swaps. Then, \( L \) is obtained by inserting \( m-n \) symbols in the string \( S_f \).

### 3.2 Idea and variables

The overall algorithm is as follows: We “scan” \( S \) and \( L \) simultaneously from left to right, correcting \( S \) at the same time, and applying the next rules. If the current symbol of \( S \) equals the current symbol of \( L \), then we continue with the scan. Otherwise, we decide between two options: insert a symbol in \( S \), or move (by applying many swaps) the first symbol of the part not scanned of \( S \) equal to the current position in \( S \); and resume the scan.

To avoid modifying the string \( S \) during the scanning process, the algorithm maintains a counter for each symbol \( \alpha \in \Sigma \) of how many symbols of \( S \) were “swapped” to earlier positions in order to match an already scanned symbol of \( L \). Let \( \mathbb{W} = \prod_{\alpha=1}^{q} [0..n_\alpha] \) denote the domain of such a vector of counters, and \( \overline{c} = (c_1, c_2, \ldots , c_q) \in \mathbb{W} \) be such a vector of counters, where \( c_\alpha \) denotes the counter for \( \alpha \in \Sigma \). To know the position of the first symbol of the part not scanned of \( S \) equal to the current symbol of \( L \), the functions rank and select, parametrized with \( \overline{c} \), will be used.

The algorithm computes the extension \( \text{DIST}(i,j,\overline{c}) \) of the Swap-Insert String-to-String Correction distance \( \delta(S,L) \), defined for each integers \( i \in [1..n+1] \) and \( j \in [1..m+1] \), 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) each \( \alpha \in \Sigma \) that first \( c_\alpha \) occurrences of \( \alpha \) from left to right. Such an “ignored” symbol \( S[k] \) for any position \( k \in [i..n] \), means that a symbol have been swapped with some symbol within \( S[1..i-1] \) in a previous step of the algorithm.

Given this definition, \( \delta(S,L) = \text{DIST}(1,1,\overline{u}) \), where \( \overline{u} \) denotes the vector \((0, \ldots , 0) \in \mathbb{W} \). Given \( i, j, \) and \( \overline{c} \), we have that \( \text{DIST}(i,j,\overline{c}) < +\infty \) if and only if for each symbol \( \alpha \in \Sigma \) the number of considered \( \alpha \) symbols in \( S[i..n] \) is at most the number of \( \alpha \) symbols in \( L[j..m] \), that is, \( \text{count}(S,i,\alpha) - c_\alpha \leq \text{count}(L,j,\alpha) \) for all \( \alpha \in \Sigma \), where \( \text{count}(X,i,\alpha) = \text{rank}(X,[X],[\alpha]) - \text{rank}([X],i-1,\alpha) \) is the number of symbols \( \alpha \) in the string \( X[i..|X|] \). In the following, we show how to compute \( \text{DIST}(i,j,\overline{c}) \) 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 all components are equal to zero except the oth component that is equal to 1.

### 3.3 Invariants

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 \( S[i..n] \) is not 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 \( \Sigma \), 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 \( \alpha \in \Sigma \),

\[
S[i] = L[j] = \alpha \\
c_\alpha = 0
\]

\[\implies \text{DIST}(i, j, \overline{c}) = \text{DIST}(i + 1, j + 1, \overline{c}).\]

Proof. Since \( c_\alpha = 0 \), no \( \alpha \) symbol of \( S[i..n] \) has been ignored, and the symbol \( \alpha = S[i] \) must be considered. Let \( S'[1..n'] = S[i..n] \overline{c} \). Given that two equal symbols cannot be swapped, we have two options to transform \( S'[1..n'] \) into \( L[j..m] \) with the minimum number of operations: (1) transform \( S'[1..n'] \) into \( L[j+1..m] \) with the minimum number of operations (matching \( S'[i] \) with \( S[j] \)); or (2) transform \( S'[1..n'] \) into \( L[j+1..m] \) with the minimum number of operations and then insert an \( \alpha \) symbol at the first position of the resulting \( S'[1..n'] \). Then, we have that

\[\text{DIST}(i, j, \overline{c}) = \min \{ \text{DIST}(i + 1, j + 1, \overline{c}), \text{DIST}(i, j + 1, \overline{c}) + 1 \}.\]

Note that the number \( I \) of insertions in \( \text{DIST}(i + 1, j + 1, \overline{c}) \) equals \((m - j + 1) - n'\), whereas the number of insertions in \( \text{DIST}(i, j + 1, \overline{c}) \) equals \( I - 1 \). Let \( \text{DIST}(i, j + 1, \overline{c}) = s_\alpha + s_{\overline{c}} + (I - 1) \), where \( s_\alpha \) denotes the number of swaps in which the symbol \( \alpha = S'[1] \) participates, and \( s_{\overline{c}} \) denotes the number of swaps in which that symbol does not participate. Let \( \text{DIST}(i + 1, j + 1, \overline{c}) = s + I \), where \( s \) denotes the total number of swaps.

Let \( f \) be the transformation mapping associated with \( \text{DIST}(i + 1, j + 1, \overline{c}) \), \( k = f(i) \), and \( f' \) be the transformation mapping associated with \( \text{DIST}(i + 1, j + 1, \overline{c}) \). Observe that \( f \) maps \([2..n']\) to a subset of \([j + 1..k - 1] \cup [k + 1..m]\) and that \( S'[2..n']_f \) is obtained from \( S'[2..n'] \) in \( s_{\overline{c}} \) swaps. Further, note that \( f' \) maps \([2..n']\) to a subset of \([j + 1..m]\) and that \( S'[2..n']_{f'} \) is obtained from \( S'[2..n'] \) in the (minimum) number \( s \) of swaps. Since \([j + 1..k - 1] \cup [k + 1..m] \subset [j + 1..m]\), it holds \( s \leq s_{\overline{c}} \). Then, we have

\[\text{DIST}(i, j + 1, \overline{c}) = s_\alpha + s_{\overline{c}} + (I - 1) \geq s_{\overline{c}} + I - 1 \geq s + I - 1 = \text{DIST}(i + 1, j + 1, \overline{c}) - 1, \]

which implies the result. \( \square \)

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 \( \Sigma \), 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 \( \alpha \in \Sigma \),

\[
S[i] = L[j] = \alpha \\
c_\alpha > 0
\]

\[\implies \text{DIST}(i, j, \overline{c}) = \text{DIST}(i + 1, j, \overline{c} - \overline{c}_\alpha).\]
Proof. Since $c_α > 0$, the first $c_α$ symbols $α$ of $S[i..n]$ have been ignored, thus $S[i]$ is ignored. Then, $DIST(i, j, \overline{α})$ must be equal to $DIST(i + 1, j, \overline{α} - \overline{α}_β)$, case in which $c_α - 1$ 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 starts 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 $\overline{α} = (c_1, \ldots, c_d) \in \mathbb{W}$, note $α, β \in Σ$ the symbols $α = S[i]$ and $β = L[j]$, $r$ the position $r = select(S, rank(S, i, β) + c_β + 1, β)$ in $S$ of the $(c_β + 1)th$ symbol $β$ of $S[i..n]$, and $Δ$ the number $Δ = \sum_{θ=1}^{d} \min\{c_θ, rank(S, r, θ) - rank(S, i - 1, θ)\}$ of symbols ignored in $S[i..r]$.

If $α \neq β$ and $c_α = 0$, then $DIST(i, j, \overline{α}) = \min\{d_{ins}, d_{swaps}\}$, where

$$d_{ins} = \begin{cases} \begin{array}{ll} DIST(i, j + 1, \overline{α}) + 1 & \text{if } c_β = 0 \\ \infty & \text{if } c_β > 0 \end{array} \end{cases}$$

and

$$d_{swaps} = \begin{cases} \begin{array}{ll} (r - i) - Δ + DIST(i, j + 1, \overline{α} + \overline{α}_β) & \text{if } r \neq 0 \\ \infty & \text{if } r = 0. \end{array} \end{cases}$$

Proof. Let $S'[1..n'] = S[i..n]|_α$. Given that $α \neq β$ and $c_α = 0$, we have two possibilities for $DIST(i, j, \overline{α})$: (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 the property 2 of the optimal solutions). If this is the case, then $DIST(i, j, \overline{α}) = DIST(i, j + 1, \overline{α}) + 1$. Option (2) can be used if and only if there is a non-ignored symbol $β$ in $S[i..n]$, where the first one form left to right is precisely at position $r = select(S, rank(S, i, β) + c_β + 1, β)$. In such a case we have that $r' = (r - i + 1) - Δ$, where $Δ = \sum_{θ=1}^{d} \min\{c_θ, rank(S, r, θ) - rank(S, i - 1, θ)\}$ is the total number of ignored symbols in the string $S[i..r]$. Hence, the number of swaps counts to $r' - 1 = (r - i) - Δ$. 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 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 $\overline{α} = (c_1, \ldots, c_d) \in \mathbb{W}$,

$$DIST(i, m + 1, \overline{α}) = \begin{cases} 0 & \text{if } c_1 + \ldots + c_d = n - i + 1 \text{ and} \\ +\infty & \text{otherwise.} \end{cases}$$
Proof. Note that \( \text{DIST}(i, m+1, \bar{\tau}) \) is the minimum number of operations to transform the string \( S[i..n] \) into the empty string \( L[m+1..n] \). This number is equal zero 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, then \( \text{DIST}(i, m+1, \bar{\tau}) = +\infty \). □

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 \( \Sigma \), for any position \( j \in [1..m+1] \) in \( L \), and for any vector of counters \( \bar{\tau} = (c_1, \ldots, c_d) \in \mathbb{W}^d \),

\[
\text{DIST}(n+1, j, \bar{\tau}) = \begin{cases} 
    m - j + 1 & \text{if } \bar{\tau} = \bar{\tau'} \text{ and } \\
    +\infty & \text{otherwise.}
\end{cases}
\]

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

### 3.4 Complexity Analysis

Combining Lemmas 1 to 5 the value of \( \text{DIST}(1, 1, \bar{\tau}) \) 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 \( n_\alpha \) of occurrences of \( \alpha \) in \( S \) and the number \( m_\alpha \) of occurrences of \( \alpha \) in \( L \).

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

\[
O \left( d(n + m) + d^2n \cdot \sum_{\alpha=1}^{d} (m_\alpha - g_\alpha) \cdot \prod_{\beta=2}^{d} (g_\beta + 1) \right).
\]

**Proof.** Observe first that there is a reordering of \( \Sigma = [1..d] \) such that \( g_1 \leq g_2 \leq \cdot \cdot \cdot \leq g_d \). Assuming that all symbols of \( \Sigma \) are used in \( L \), i.e. \( m_1, \ldots, m_d > 0 \), it implies that \( m_\alpha > g_\alpha \) for every \( \alpha \in \Sigma \).

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 \( O(d \cdot |X|) \), supporting \( \text{rank}(X, i, \alpha) \) and \( \text{select}(X, k, \alpha) \) queries in constant time for every values of \( i \in [1..n] \), \( k \in [1..|X|] \), and \( \alpha \in \Sigma \).
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 $DIST(i + 1, j, \overline{c} - w_\alpha)$
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 $\beta$ at index $i$ *)
   21. $d_{ins} \leftarrow 1 + DIST(i, j + 1, \overline{c})$
22. $r \leftarrow \text{select}(S, \text{rank}(S, i, \beta) + c_\beta + 1, \beta)$
23. if $r \neq \text{null}$ then
   24. $\Delta \leftarrow \sum_{\theta=1}^d \min\{c_\theta, \text{rank}(S, r, \theta) - \text{rank}(S, i - 1, \theta)\}$
   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.

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 $DIST(i, j, \overline{c})$, equals $j - i - (c_1 + \cdots + c_d)$. Let $t_\alpha$ denote the number of such insertions for the symbol $\alpha \in \Sigma$. Then, $j = i + (c_1 + \cdots + c_d) + (t_1 + \cdots + t_d)$, and observe that $c_\alpha \leq n_\alpha$, $t_\alpha \leq m_\alpha - n_\alpha$, and $c_\alpha + t_\alpha = \text{rank}(L, j - 1, \alpha) - \text{rank}(S, i - 1, \alpha)$ for all $\alpha \in \Sigma$. 

Using the above observations, we encode all entries $DIST(i, j, \bar{\sigma})$, for $i, j$ and $\bar{\sigma}$ such that $DIST(i, j, \bar{\sigma}) < +\infty$, into the following table $T$ of $d + 2$ dimensions, such that $T[p, i, k, r_1, \ldots, r_{d-1}] = DIST(i, j, \bar{\sigma} = (c_1, \ldots, c_d))$ where

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

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 $\alpha > 1$, the table $T$ can be as large as $d \times (n + 1) \times (\sum_{\alpha=1}^{d}(m_\alpha - g_\alpha) + 1) \times (g_2 + 1) \times \cdots \times (g_d + 1)$. Using a modified version of the algorithm of Figure 4 based on memoization on the table $T$, the goal entry $DIST(1, 1, \bar{\sigma})$ can be computed recursively, where each entry $DIST(i, j, \bar{\sigma})$ is computed in (amortized) $O(d)$ time. The running time of this new algorithm includes the $O(d(n + m))$ time for processing each of $S$ and $L$ for rank and select, and the time to compute $DIST(1, 1, \bar{\sigma})$ which is within $O(d)$ times the number of cells of the table $T$. The time to compute $DIST(1, 1, \bar{\sigma})$ is then within

\[
O \left( d \cdot d \cdot (n + 1) \cdot \left( \sum_{\alpha=1}^{d}(m_\alpha - g_\alpha) + 1 \right) \cdot (g_2 + 1) \cdots (g_d + 1) \right)
\]

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

The result thus follows.

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 $\Sigma$, for each symbol $\alpha \in \Sigma$, note $n_\alpha$ the number of occurrences of $\alpha$ in $S$ and $m_\alpha$ 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 $n_\alpha$ is from $m_\alpha/2$. The algorithm analyzed in Theorem 4 computes the SWAP-INSERT STRING-TO-STRING CORRECTION distance $\delta(S, L)$ in time within

\[
O \left( d(n + m) + d^2 n(m - n) \left( \frac{n}{d - 1} + 1 \right)^{d-1} \right).
\]

\[^3\text{Cormen et al.}\ [2] \text{explain that memoization comes from memo, referring to the fact that the technique consists in recording a value so that we can look it up later.}\]
which is \( O(n + m + n^d(m-n)) \) for fixed \( d \); and within

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

which is \( O(n + m + n^2(m-n)^{d-1}) \) for fixed \( d \).

## 4 Discussion

In 2014, Meister [4] described an algorithm computing the Swap-Insertion String-to-String Correction distance from a string \( S \in [1..d]^n \) to another string \( L \in [1..d]^m \) on any fixed alphabet size \( d \geq 2 \), in time polynomial in \( n \) and \( m \). We describe a simpler algorithm running in time within

\[
O \left( n + m + n \frac{m-n}{d-1} \right),
\]

and even faster when for all symbols \( \alpha \in \Sigma \) the number \( n_\alpha \) of occurrences of \( \alpha \) in \( S \) is either close to zero (i.e. most \( \alpha \) symbols from \( L \) are placed in \( S \) through insertions) or close to the number \( m_\alpha \) of occurrences of \( \alpha \) in \( L \) (i.e. most \( \alpha \) symbols from \( L \) are matched to symbols in \( S \) through swaps). The exact running time of our algorithm is within

\[
O(d(n+m) + d^2n^2 \cdot \sum_{\alpha=1}^d (m_\alpha - g_\alpha) \cdot \prod_{\beta=2}^d (g_\beta + 1)),
\]

where \( g_\alpha = \min\{n_\alpha, m_\alpha - n_\alpha\} \). This simplifies to within \( O(d^2nmg^{d-1}) \), where \( g = \max_{\alpha \in \Sigma} g_\alpha \).

### 4.1 Implicit Results

The results from Theorem 1 imply further results, not quite important enough to figure in a corollary, but important enough to be mentioned:

**Weighted Operators** Wagner and Fisher [7] considered variants where the cost \( c_{\text{ins}} \) of an insertion and the cost \( c_{\text{swap}} \) of an swap are distinct. In the Swap-Insert String-to-String 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.

**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.

**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 String-to-String 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 \( O(d(n+m)) \), hence improving on Abu-Khzam et al.’s solution [1].
Large Alphabet Let \( d' \) be the number of symbols \( \alpha \) of \( \Sigma = [1..d] \) such that the number of occurrences of \( \alpha \) in \( S \) is a constant fraction of the number of occurrences of \( \alpha \) in \( L \) (i.e. \( n_\alpha \in \Theta(m_\alpha) \)). Our results imply 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.

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 the worst case, in which both branches are fully explored.

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 Abu-Khzam et al.’s algorithm and ours in parallel yields a solution adaptive to both measures, but there should be a more clever solution and analysis.

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? Some cases are easy (e.g. when only insertions or only deletions are allowed), and some others require further work.

References

1. F. N. Abu-Khzam, H. Fernau, M. A. Langston, S. Lee-Cultura, and U. Stege. Charge and reduce: A fixed-parameter algorithm for string-to-string correction. *Discrete Optimization (DO)*, 8(1):41 – 49, 2011.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
3. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.
4. D. Meister. Using swaps and deletes to make strings match. Technical report, Fachbereich IV, Universität Trier, 2014.
5. T. D. Spreen. The binary string-to-string correction problem. Master’s thesis, University of Victoria, Canada, 2013.
6. R. A. Wagner. On the complexity of the extended string-to-string correction problem. In *Proceedings of the seventh annual ACM Symposium on Theory Of Computing (STOC)*, pages 218–223. ACM, 1975.
7. R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.
8. R. A. Wagner and R. Lowrance. An extension of the string-to-string correction problem. *Journal of the ACM (JACM)*, 22(2):177–183, 1975.
A Proof of Corollary 1

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

\[(a - 1)x \leq (a - 1)y\]
\[ax + y \leq ay + x\]
\[ax + y + a + xy \leq ay + x + a + xy\]
\[(a + y)(x + 1) \leq (a + x)(y + 1)\]

Let $\beta \in [2,d] = \Sigma \setminus \{1\}$ be a symbol such that $m_\beta - n_\beta < n_\beta$, and define $a$ and $b$ such that:

\[a = \sum_{\alpha=1}^{d} (m_\alpha - g_\alpha) - (m_\beta - g_\beta) = \sum_{\alpha=1}^{d} (m_\alpha - g_\alpha) - n_\beta\]
\[= \sum_{\alpha \in \Sigma \setminus \{\beta\}} (m_\alpha - g_\alpha) \geq 1,\]

and

\[b = \prod_{\alpha \in \Sigma \setminus \{1,\beta\}} (g_\alpha + 1).\]

Note that:

\[\sum_{\alpha=1}^{d} (m_\alpha - g_\alpha) \cdot \prod_{\alpha=2}^{d} (g_\alpha + 1) = (a + n_\beta) \cdot b \cdot (m_\beta - n_\beta + 1)\]
\[\leq (a + m_\beta - n_\beta) \cdot b \cdot (n_\beta + 1),\]

which immediately implies

\[\sum_{\alpha=1}^{d} (m_\alpha - g_\alpha) \cdot \prod_{\alpha=2}^{d} (g_\alpha + 1) \leq (m - n) \prod_{\alpha=2}^{d} (n_\alpha + 1).\]

Then,

\[O\left(d^2n \cdot \sum_{\alpha=1}^{d} (m_\alpha - g_\alpha) \cdot \prod_{\alpha=2}^{d} (g_\alpha + 1)\right) \subseteq O\left(d^2n(m - n) \cdot (n_2 + 1) \cdots (n_d + 1)\right)\]
\[\subseteq O\left(d^2n(m - n) \cdot \left(\frac{n_2 + \cdots + n_d}{d - 1} + 1\right)^{d-1}\right)\]
\[\subseteq O\left(d^2n(m - n) \cdot \left(\frac{n}{d - 1} + 1\right)^{d-1}\right).\]

Using similar arguments, we can prove that

\[\sum_{\alpha=1}^{d} (m_\alpha - g_\alpha) \cdot \prod_{\alpha=2}^{d} (g_\alpha + 1) \leq n \prod_{\alpha=2}^{d} (m_\alpha - n_\alpha + 1)\]

which implies the second part of the result. \qed
B Complete algorithm

In the following, we describe the formal algorithm to compute \( DIST(i, j, \overline{u}) \). Note that the line 3 from the algorithm `Compute` and line 26 from the algorithm `DIST2` guarantee that \( DIST2(i, j) = DIST(i, j, \overline{u}) < +\infty \) in every call of `DIST2`. Further, the counters \( (c_1, c_2, \ldots, c_d) \) are global variables to the recursive `DIST2`. 
Algorithm Compute $\delta(S, L)$:
1. preprocess each of $S$ and $L$ for rank and select
2. $(c_1, c_2, \ldots, c_d) \leftarrow \emptyset$
3. return if $\text{DIST}(1, 1, 0) < +\infty$ then $\text{DIST}(1, 1)$ else $+\infty$

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

Fig. 2: Formal Algorithm to compute $\text{DIST}(i, j, 0)$, using dynamic programming with memorization. Note that the line 26 from the algorithm Compute and line 3 from the algorithm DIST2 guarantee that $\text{DIST}(2)(i, j) = \text{DIST}(i, j, 0) < +\infty$ in every call.