Indexed Dynamic Programming to boost Edit Distance and LCSS Computation[*]

Jérémy Barbay[0000−0002−3392−8353][**] and Andrés Olivares

Departamento de Ciencias de la Computación, University of Chile,
jeremy@barbay.cl, aolivare@dcc.uchile.cl

Abstract. There are efficient dynamic programming solutions to the computation of the Edit Distance from $S \in [1..\sigma]^n$ to $T \in [1..\sigma]^m$, for many natural subsets of edit operations, typically in time within $O(nm)$ in the worst-case over strings of respective lengths n and m (which is likely to be optimal), and in time within $O(n+m)$ in some special cases (e.g., disjoint alphabets). We describe how indexing the strings (in linear time), and using such an index to refine the recurrence formulas underlying the dynamic programs, yield faster algorithms in a variety of models, on a continuum of classes of instances of intermediate difficulty between the worst and the best case, thus refining the analysis beyond the worst case analysis. As a side result, we describe similar properties for the computation of the Longest Common Sub Sequence $\texttt{LCSS}(S,T)$ between S and T, since it is a particular case of Edit Distance, and we discuss the application of similar algorithmic and analysis techniques for other dynamic programming solutions. More formally, we propose a parameterized analysis of the computational complexity of the Edit Distance for various sets of operators and of the Longest Common Sub Sequence in function of the area of the dynamic program matrix relevant to the computation.

Keywords: Adaptive Algorithm · Dynamic Programming · Edit Distance · Longest Common Sub-Sequence.

1 Introduction

Given a set of edition operators on strings, a source string $S \in [1..\sigma]^n$ and a target string $T \in [1..\sigma]^m$ of respective lengths n and m on the alphabet $[1..\sigma]$, the EDIT DISTANCE is the minimum number of such operations required to transform the string S into the string T. Introduced in 1974 by Wagner and Fischer [15], such computation is a fundamental problem in Computer Science, with a wide range of applications, from text processing and information retrieval to computational biology. The typical edit distance between two strings is defined by the minimum number of insertions, deletions (in both cases, of a character at an arbitrary

---

Operators	(n,m)-Worst Case Complexity	Finer Results	
		Distance	Parikh vectors
Swap	$O(n^2)$ [14]	$O(n(1+\lg d/n)\lg n)$	DNA
Delete, Insert	$O(nm)$ [6]	$O(d^2)$	$O(\sum n_\alpha m_\alpha \lg nm)$
Delete, Replace	$O(nm)$ [17]	$O(d^2)$	$O(\sum n_\alpha m_\alpha \lg nm)$
Delete, Swap	NP-complete [15]	$O(1.6181^d m)$ [1]	$O(\sigma^2 nm\gamma^{\sigma-1})$ [3]
Replace, Swap	$O(nm)$ [17]	$O(d^2)$	$O(\sum n_\alpha m_\alpha \lg nm)$
Delete, Insert, Replace	$O(nm)$ [6]	$O(d^2)$	$O(\sum n_\alpha m_\alpha \lg nm)$
Delete, Insert, Swap	$O(nm)$ [15]	$O(d^2)$	
Delete, Replace, Swap	$O(nm)$ [15]	$O(d^2)$	
Insert, Replace, Swap	$O(nm)$ [15]	$O(d^2)$	
Delete, Insert, Replace, Swap	$O(nm)$ [6]	$O(d^2)$	

Fig. 1. Summary of results for various combinations of operators from the basic set {`Insert`, `Delete`, `Replace`, `Swap`}. The column labeled "(n,m)-Worst Case Complexity" presents results in the worst case over instances of fixed sizes n and m, while the columns labeled "Finer Results" present results where the analysis was refined by various parameters: the distance d, the size σ of the alphabet, and some form of imbalance $\gamma = \max_{\alpha\in[1..\sigma]}\min\{n_\alpha, m_\alpha - n_\alpha\}$ between the `Parikh vectors` of S and T. For brevity, the only distance based on a single operator presented is the SWAP EDIT DISTANCE, as the computation of the others is always linear in the size of the input.

position of S) and `replacement` (of one character of S by some other) needed to transform the string S into T. Many generalizations have been defined in the literature, including weighted costs for the edit operations, and different sets of edit operations – the standard set is {`insertion`, `deletion`, `replacement`}.

Each distinct set of correction operators yields a distinct correction distance on strings (see Figure 1 for a summary). For instance, Wagner and Fischer [15] showed that for the three following operations, the `insertion` of a symbol at some arbitrary position, the `deletion` of a symbol at some arbitrary position, and the `replacement` of a symbol at some arbitrary position, the EDIT DISTANCE can be computed in time within $O(nm)$ and space within $O(n+m)$ using traditional dynamic programming techniques. As another variant of interest, Wagner and Lowrance [16] introduced the `Swap` operator (`S`), which exchanges the positions of two contiguous symbols. For two of the newly defined distances, the INSERT SWAP EDIT DISTANCE and the DELETE SWAP EDIT DISTANCE (equivalent by symmetry), the best known algorithms take time exponential in the input size [3,4], which is likely to be optimal [15]. The EDIT DISTANCE itself is linked to many other problems: for instance, given the two same strings $S \in [1..\sigma]^n$ and $T \in [1..\sigma]^m$, the computation of the LONGEST COMMON SUBSEQUENCE (LCSS) L between S and T is equivalent to the computation of the DELETE INSERT EDIT DISTANCE d, as the symbols deleted from S and inserted from T in order to "edit" S into T are exactly the same as the symbols deleted from S and T in order to produce L. Hence, the LCSS between S and T can

be computed in time within $O(nm)$ and space within $O(n+m)$ using traditional dynamic programming techniques.

Most of these computational complexities are likely to be optimal in the worst case over instances of size (n, m): the algorithms computing the three basic distances in linear time are optimal as any algorithm must read the whole strings; the INSERT SWAP EDIT DISTANCE and its symmetric the DELETE SWAP EDIT DISTANCE are NP-hard to compute [17]; and in 2015 Backurs and Indyk [2] showed that the $O(n^2)$ upper bound for the computation of the DELETE INSERT REPLACE EDIT DISTANCE is optimal up to a sub-polynomial factor unless the *Strong Exponential Time Hypothesis* (SETH) is false.

More recently, Barbay and Pérez-Lantero [3,4], complementing Meister's previous results [12] by the use of an index supporting the operators `rank` and `select` on strings, described an algorithm computing this distance in time within $O(\sigma^2 nm\gamma^{\sigma-1})$ in the worst case over instances where σ, n, m and γ are fixed, where $\gamma = \max_{\alpha \in [1..\sigma]} \min\{n_\alpha, m_\alpha - n_\alpha\}$ measures a form of imbalance between the frequency distributions of each string.

Hypothesis: Given this situation, is it possible to **take advantage of** indexing techniques supporting **rank and select** in order **to speed up the computation of** other **edit distances**? Can a similar analysis to that of Barbay and Pérez-Lantero [3,4] be applied to other edit distances? **Are there instances for which the edit distance is easier to compute**, and **do such instances occur in real applications** of the computation of the edit distance?

Our Results: We answer all those questions positively, and describe general techniques to refine the analysis of dynamic programs beyond the traditional analysis in the worst case over inputs of fixed size. More specifically, we analyze the computational cost of four EDIT DISTANCES using various `rank` and `select` text indices, as a function of the `Parikh vector` [13] of the source S and target T strings. As a side result, this yields similar properties for the computation of the LONGEST COMMON SUB SEQUENCE `LCSS`(S, T) between S and T, and definitions and techniques which can be applied to other dynamic programs. After defining formally the notion of `Parikh's vector` and various index data structures supporting `rank` and `select` on strings in Section 2, we describe the algorithms taking advantage of such techniques in Section 3: for the LONGEST COMMON SUB SEQUENCE and DELETE-INSERT EDIT DISTANCE (Section 3.1), the DELETE INSERT REPLACE EDIT DISTANCE (Section 3.2), and finally for the DELETE-REPLACE EDIT DISTANCE and its dual, the INSERT-REPLACE EDIT DISTANCE (Section 3.3). We describe some preliminary experiments and their results, which seem to indicate that those instances are not totally artificial and occur naturally in practical applications in Section 4. We conclude in Section 5 with a discussion of other potential refinement of the analysis, and the extension of our results to other EDIT DISTANCES.

2   Preliminaries

Before describing our proposed algorithms to compute various EDIT DISTANCES, we describe formally in Section 2.1 the notion of `Parikh vector` which is essential to our analysis technique; and in Section 2.2 two key implementations of indices supporting the `rank` and `select` operators on strings.

2.1   Parikh vector

Given positive integers σ and n, a string $S \in [1..\sigma]^n$, and the integers $n_1, \ldots, n_\sigma$ such that n_α denotes the number of occurrences of the letter $\alpha \in [1..\sigma]$ in the string S, the `Parikh vector` of S is defined [13] as $p(S) = (n_1, \ldots, n_\sigma)$.

Barbay and Pérez-Lantero [3] refined the analysis of the INSERT SWAP EDIT DISTANCE from a string $S \in [1..\sigma]^n$ to a string $T \in [1..\sigma]^m$ via a function of the `Parikh vectors` $(n_1, \ldots, n_\sigma)$ of S and $(m_1, \ldots, m_\sigma)$ of T, the local imbalance $\gamma_\alpha = \min\{n_\alpha, m_\alpha - n_\alpha\}$ for each symbol $\alpha \in [1..\sigma]$, projected to a global measure of imbalance, $\gamma = \max_{\alpha \in [1..\sigma]} \gamma_\alpha$. In the worst case among instances of fixed `Parikh vector`, they describe an algorithm to compute the INSERT SWAP EDIT DISTANCE in time within $O(\sigma^2 nm\gamma^{\sigma-1})$ in the worst case over instances where σ, n, m and γ are fixed.

Such a vector is essential to the fine analysis of dynamic programs for computing EDIT DISTANCES when using operators whose running time depends on the number of occurrence of each symbol, such as for the `rank` and `select` operators described in the next section.

2.2   Rank and Select in Strings

Given a symbol $\alpha \in [1..\sigma]$, an integer $i \in [1..|X|]$ and an integer $k > 0$, the operation $rank(X, i, \alpha)$ denotes the number of occurrences of the symbol α in the substring $X[1..i]$, and the operation $select(X, k, \alpha)$ 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*.

A simple way to support the `rank` and `select` operators in reasonably good time consists in, for each symbol $\alpha \in [1..\sigma]$, listing all the occurrences of α in a sorted array (called a "Posting List" [18]): supporting the `select` operator reduces to a simple access to the sorted array corresponding to the symbol α; while supporting the `rank` operator reduces to a SORTED SEARCH in the same array, which can be simply implemented by a `Binary Search`, or more efficiently in practice by a `Doubling Search` [5] in time within $O(q_\alpha \lg(n_\alpha/q_\alpha))$ when supporting q_α monotone queries in a posting list of size n_α (for a given symbol $\alpha \in [1..\sigma]$.

Golynski *et al.* [9] described a more sophisticated (but asymptotically more efficient) way to support the `rank` and `select` operators in the RAM model, via a clever reduction to Y-Fast Trees on permutations supporting the operators in time within $O(\lg \lg \sigma)$. We describe how to use those techniques to speed up the computation of various EDIT DISTANCES in the following sections.

3   Adaptive Dynamic Programs

For each of the problems considered, we describe how to compute a subset of the values computed by classical dynamic programs. We start with the computation of the LONGEST COMMON SUB SEQUENCE (LCSS) and the DELETE INSERT (DI) EDIT DISTANCE (Section 3.1) because it is the simplest; extend its results to the computation of the LEVENSHTEIN EDIT DISTANCE (Section 3.2); and project those to the computation of the DELETE REPLACE (DR) EDIT DISTANCE and its symmetric INSERT REPLACE (IR) EDIT DISTANCE (Section 3.3).

3.1   LCSS and DI-Edit Distance

The DELETE INSERT EDIT DISTANCE is a classical problem in Stringology [6], if only as a variant of the LONGEST COMMON SUB SEQUENCE problem. It is classically computed using dynamic programming: we describe the classical solution first, which we then refine.

Classical solution: Given two strings $S \in [1..\sigma]^n$ and $T \in [1..\sigma]^m$, we note $d_{DI}(n, m)$ the DELETE INSERT EDIT DISTANCE from S to T. If the last symbols of S and T match, the edit distance is the same as the edit distance between the prefixes of respective lengths $n - 1$ and $m - 1$ of S and T. Otherwise, the edit distance is the minimum between the edit distance when inserting a copy of the last symbol of T in S (i.e. deleting this symbol in T) and the edit distance when deleting the mismatching symbol in T. More formally:

$$d_{DI}(S[1..n], T[1..m]) = \begin{cases} n \text{ if } m == 0; \\ m \text{ if } n == 0; \\ d_{DI}(S[1..n-1], T[1..m-1]) \text{ if } S[n] == T[m]; \text{ and} \\ 1 + \min \left\{ \begin{array}{l} d_{DI}(S[1..n-1], T[1..m]), \\ d_{DI}(S[1..n], T[1..m-1]) \end{array} \right\} \text{ otherwise.} \end{cases}$$

This recursive definition directly yields an algorithm to compute the DELETE INSERT EDIT DISTANCE from S to T in time within $O(nm)$ and space within $O(n + m)$. We describe in the next section a technique taking advantage of the discrepancies between the `Parikh vectors` of S and T.

Refined Analysis: It is natural to wonder if techniques described in the previous section can be used in to take advantage of cases where a symbol occurs many times in one string, but occurs only once in the other: at some point, the dynamic program will reduce to the case described in the previous section. To be able to notice when this happens, one would need to maintain dynamically the counters of occurrences of each symbol during the execution of the dynamic program, or more simply pre-compute an index on S and T supporting the operators `rank` and `select` on it.

Given the support for the `rank` and `select` operators on both S and T, we can refine the dynamic program to compute the distance $d_{DI}(n, m)$ as follows:

$$\begin{cases} n \text{ if } m == 0; \\ m \text{ if } n == 0; \\ d_{DI}(n-1, m-1) \text{ if } S[n] == T[m]; \\ 1 + d_{DI}(n-1, m) \text{ if } \mathtt{rank}(S, T[m]) == 0; \\ 1 + d_{DI}(n, m-1) \text{ if } \mathtt{rank}(T, S[n]) == 0; \\ \min \left\{ \begin{array}{l} 1 + d_{DI}(n-1, m-1), \\ n - \mathtt{select}(S, T[m]) \\ \quad + d_{DI}(\mathtt{select}(S, T[m], \mathtt{rank}(S, T[m]) - 1) - 1, m-1), \\ m - \mathtt{select}(T, S[n]) \\ \quad + d_{DI}(n-1, \mathtt{select}(T, S[n], \mathtt{rank}(T, S[n]) - 1)) - 1) \end{array} \right\} \text{ otherwise.} \end{cases}$$

The running time of the algorithm can then be expressed as a function of the number of recursive calls, the number of `rank` and `select` operations performed on the strings, in order to yield various running times depending upon the solution used to support the `rank` and `select` operators.

Theorem 1. *Given two strings $S \in [1..\sigma]^n$ and $T \in [1..\sigma]^m$ of respective **Parikh vectors** $(n_a)_{a \in [1..\sigma]}$ and $(m_a)_{a \in [1..\sigma]}$, the dynamic program above computes the* DELETE INSERT EDIT DISTANCE *from S to T and the* LONGEST COMMON SUB SEQUENCE *between S and T*

1. *through at most $4 \sum_{a \in [1..\sigma]} n_a m_a$ recursive calls;*
2. *within $O(\sum_{a \in [1..\sigma]} n_a m_a)$ operations `rank` or `select`;*
3. *in time within $O(\sum_{a \in [1..\sigma]} n_a m_a \times \lg(\max_a\{n_a, m_a\}) \times \lg(nm))$ in the comparison model; and*
4. *in time within $O(\sum_{a \in [1..\sigma]} n_a m_a \times \lg\lg\sigma \times \lg(nm))$ in the RAM memory model.*

Proof. We prove point (1) by an amortization argument. Point (2) is a direct consequence of point (1), given that each recursive call performs a finite number of calls to the `rank` and `select` operators. Point (3) is a simple combination of Point (2) with the classical *inverted posting list* implementation [18] of an index supporting the `select` operator in constant time and the `rank` operator via *doubling search* [5]; while point (4) is a simple combination of Point (2) with the index described by Golynski *et al.* [9] to support the `rank` and `select` operators.

Albeit quite simple, this results corresponds to real improvement in practice: see in Figure 2 how the number of recursive calls is reduced by using such indexes. Moreover, such a refinement of the analysis and optimization of the computation can be applied to more than the DELETE INSERT EDIT DISTANCE: in the next sections, we describe a similar one for computing the LEVENSHTEIN DISTANCE (Section 3.2) and the DELETE REPLACE and INSERT REPLACE EDIT DISTANCE (Section 3.3).

3.2   Levenshtein Distance, or DIR-Edit Distance

In information theory, linguistics and computer science, the LEVENSHTEIN DISTANCE is a string metric for measuring the difference between two sequences [6]. It generalizes the DELETE INSERT EDIT DISTANCE explored in the previous section by adding the `Replace` operator to the operators `Delete` and `Insert` (so that it can be also called the DELETE INSERT REPLACE EDIT DISTANCE, or DIR for short). The recursion traditionally used is a mere extension from the one described in the previous section, and the adaptive version only a technical extension of the one for the DELETE INSERT EDIT DISTANCE: we will compute the distance $d_{DIR}(n, m)$ as

$$
\begin{cases}
n \text{ if } m == 0; \\
m \text{ if } n == 0; \\
d_{DIR}(n-1, m-1) \text{ if } S[n] == T[m]; \\
1 + d_{DIR}(n-1, m-1) \quad \begin{array}{l} \text{if } \mathtt{rank}(S, T[m]) == 0 \\ \text{and } \mathtt{rank}(T, S[n]) == 0 \end{array}; \\
1 + d_{DIR}(n-1, m) \quad \begin{array}{l} \text{if } \mathtt{rank}(S, T[m]) == 0 \\ \text{but } \mathtt{rank}(T, S[n]) > 0 \end{array}; \\
1 + d_{DIR}(n, m-1) \quad \begin{array}{l} \text{if } \mathtt{rank}(T, S[n]) == 0 \\ \text{but } \mathtt{rank}(S, T[m]) > 0 \end{array}; \\
\min \begin{cases} n - \mathtt{select}(S, T[m]) \\ \quad + d_{DIR}(\mathtt{select}(S, T[m], \mathtt{rank}(S, T[m]) - 1) - 1, m - 1), \\ m - \mathtt{select}(T, S[n]) \\ \quad + d_{DIR}(n - 1, \mathtt{select}(T, S[n], \mathtt{rank}(T, S[n]) - 1)) - 1), \\ 1 + d_{DIR}(n-1, m-1) \end{cases} \quad \text{otherwise.}
\end{cases}
$$

The refined analysis yields similar results (we omit the proof for lack of space):

Theorem 2. *Given two strings $S \in [1..\sigma]^n$ and $T \in [1..\sigma]^m$ of respective* `Parikh` *vectors* $(n_a)_{a \in [1..\sigma]}$ *and* $(m_a)_{a \in [1..\sigma]}$, *the dynamic program above computes the* LEVENSHTEIN EDIT DISTANCE *from S to T*

1. *through at most $4 \sum_{a \in [1..\sigma]} n_a m_a$ recursive calls;*
2. *within $O(\sum_{a \in [1..\sigma]} n_a m_a)$ operations* `rank` *or* `select`;
3. *in time within $O(\sum_{a \in [1..\sigma]} n_a m_a \times \lg(\max_a\{n_a, m_a\}) \times \lg(nm))$ in the comparison model; and*
4. *in time within $O(\sum_{a \in [1..\sigma]} n_a m_a \times \lg \lg \sigma \times \lg(nm))$ in the RAM memory model.*

It is important to note that for two strings S and T, the computation of the LEVENSHTEIN EDIT DISTANCE from S to T actually generates more recursive calls than the computation of the DELETE INSERT EDIT DISTANCE from S to T, but that the analysis above does not capture this difference. In the following section, we project this result to two equivalent edit distances, the DELETE REPLACE and INSERT REPLACE EDIT DISTANCES, for which the dynamic program explores only half of the position in the dynamic program matrix compared to the LEVENSHTEIN EDIT DISTANCE or DELETE INSERT EDIT DISTANCE.

3.3   DR-Edit Distance and IR-Edit Distance

Given a source string $S \in [1..\sigma]^n$ and a target string $T \in [1..\sigma]^m$, the DELETE REPLACE EDIT DISTANCE from S to T and the INSERT-REPLACE EDIT DISTANCE from T to S are the same, as the sequence of `Insert` or `Replace` operations transforming S into T is the symmetric to the sequence of `Delete` or `Replace` operations transforming T back into S.

   As before, if the last symbols of S and T match, the edit distance is the same as the edit distance between the prefixes of respective lengths $n-1$ and $m-1$ of S and T. Otherwise, the edit distance is the minimum between the edit distance when inserting a copy of the last symbol of T in S (i.e. deleting this symbol in T) and the edit distance when replacing the mismatching symbol in S by the corresponding one in T.

   As in the two previous sections, given the support for the `rank` and `select` operators on both S and T, we can refine the dynamic program to compute the DELETE REPLACE EDIT DISTANCE $d_{DR}(n, m)$ as follows:

$$
\begin{cases}
n \text{ if } m == 0; \\
\infty \text{ if } n < m; \\
d_{DR}(n-1, m-1) \text{ if } S[n] == T[m]; \\
1 + d_{DR}(n-1, m) \text{ if } \mathtt{rank}(T, S[n]) == 0 \text{ ;} \\
1 + d_{DR}(n-1, m-1) \text{ if } \mathtt{rank}(S, T[n]) == 0 \text{ ;} \\
\min \begin{cases} n - \mathtt{select}(S, T[m]) \\ \quad + d_{DR}(\mathtt{select}(S, T[m], \mathtt{rank}(S, T[m]) - 1) - 1, m - 1), \\ 1 + d_{DR}(n-1, m-1) \end{cases} \quad \text{otherwise.}
\end{cases}
$$

   The analysis from the two previous sections projects to a similar result.

Theorem 3. *Given two strings $S \in [1..\sigma]^n$ and $T \in [1..\sigma]^m$ of respective `Parikh` vectors $(n_a)_{a \in [1..\sigma]}$ and $(m_a)_{a \in [1..\sigma]}$, the dynamic program above computes the* DELETE REPLACE EDIT DISTANCE *from S to T*

1. *through at most $4 \sum_{a \in [1..\sigma]} n_a m_a$ recursive calls;*
2. *within $O(\sum_{a \in [1..\sigma]} n_a m_a)$ operations `rank` or `select`;*
3. *in time within $O(\sum_{a \in [1..\sigma]} n_a m_a \times \lg(\max_a\{n_a, m_a\}) \times \lg(nm))$ in the comparison model; and*
4. *in time within $O(\sum_{a \in [1..\sigma]} n_a m_a \times \lg \lg \sigma \times \lg(nm))$ in the RAM memory model.*

   Parameterizing the analysis of the computation of the LONGEST COMMON SUB SEQUENCE, of the LEVENSHTEIN EDIT DISTANCE and of the DELETE REPLACE or INSERT REPLACE EDIT DISTANCE would be only of moderate theoretical interest, if it did not correspond to some correspondingly "easy" instances in practice. In the next section we describe some preliminary experimental results which seem to indicate the existence of such "easy" instances in information retrieval.

4   Experiments

In order to test the practicality of the parameterization and algorithms described in the previous section, we performed some preliminary experiments on some public data sets from the GUTENBERG project [11]. We considered each text as a sequence of words (hence considering as equivalent all the word separations, from blank spaces to punctuations and line jumps), which results in large alphabets. Due to some problems with the implementation, we could not run the algorithms for texts larger than 32kB (a memory issue with a library in `Python`), so we extracted the first 32kB of the texts "Romeo & Juliet" (English), "Romeo & Julia" (German), "Hamlet" (German), and "The hound of the Baskervilles" (English), "The war of the worolds" (English); the last two texts being randomly picked non Shakespeare texts.

Figures 2, 3 and 4 show the runtime for five pairs of texts for each algorithm described in Section 3, the main difference is the use of constant time `rank` and `select`: "Hamlet" (English) vs "Hamlet" (German), "The hound of the Baskervilles" (English) vs "The war of the worlds" (English), "The hound of the Baskervilles" (English) vs "Romeo and Juliet" (English), "Romeo & Juliet" (English) vs "Romeo & Julia" (German), and "Romeo & Juliet" (English) vs "Hamlet" (German).

For the two types of EDIT DISTANCES and the five pairs of texts, the adaptive variant is faster. In the case of the DELETE REPLACE EDIT DISTANCE, since the `rank` and `select` structures take from 15 to 18 seconds of preprocessing time, the adaptive variant is slower in 3 cases. For the three types of EDIT DISTANCES, the speedup of the adaptive variant is less between two texts from the same author (i.e. "Romeo & Juliet" (English) vs "Hamlet" (English))), because the vocabulary is the same, and is the most between texts of distinct languages (i.e. "Romeo & Juliet" (English) vs "Romeo & Julia" (German) and "Hamlet" (English) vs "Hamlet" (German)), because the vocabulary (i.e. the alphabet) is mostly distinct, there seems to be a correlation between alphabet intersection and speedup. Still, for two texts in the same language, but from distinct authors (i.e. "The hound of the Baskervilles" (English) vs "The war of the worlds" (English)), the difference is quite sensible for DELETE INSERT and DELETE INSERT REPLACE edit distances. Obviously, those experimental results are only preliminary, and a more thorough study is needed (and underway), both with a larger data set and with a larger range of measures, from the *running time* with various indexing data structures supporting the operators `rank` and `select`, to the number of entries of the dynamic program matrix being effectively computed. We discuss additional perspectives for future work in the next section.

5   Discussion

We have shown how the computation of other EDIT DISTANCES than the INSERT SWAP and DELETE SWAP EDIT DISTANCE is also sensitive to the `Parikh vectors` of the input. We discuss here various directions in which these results
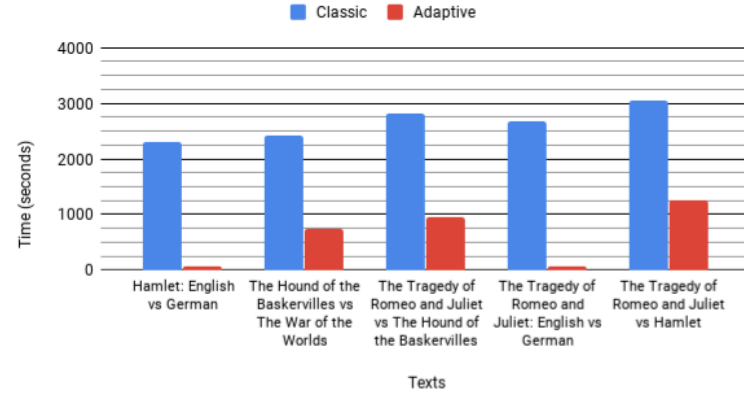
Delete - Insert Edit Distance For Texts

Fig. 2. Experimental results for the Delete Insert Edit Distance with constant `rank` and `select`.

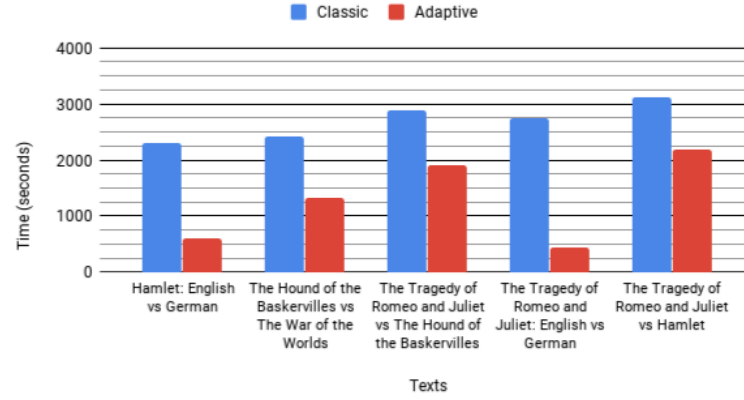**Delete - Insert - Replace Edit Distance For Texts**

Fig. 3. Experimental results for the Levenshtein Edit Distance with constant `rank` and `select`.

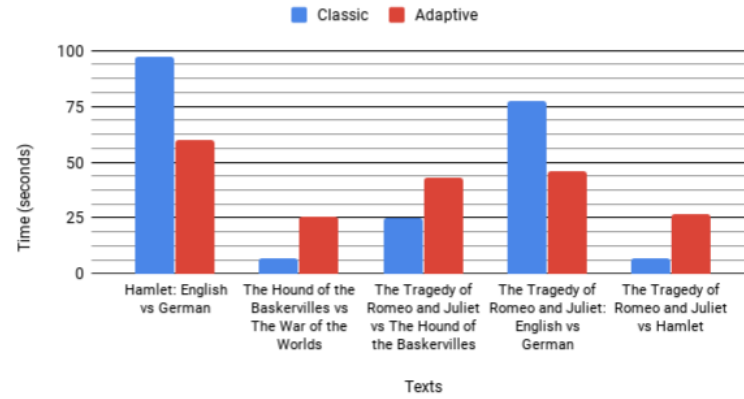**Delete - Replace Edit Distance For Texts**

Fig. 4. Experimental results for the Delete Replace Edit Distance with constant `rank` and `select`.

can be extended, from the possibility of proving conditional lower bounds in the refined analysis model, to further refinements of the analysis for these same EDIT DISTANCES, and to the analysis of other dynamic programs.

Adaptive Conditional Lower Bounds: Backurs and Indyk [2] showed that the $O(n^2)$ upper bound for the computation of the DELETE INSERT REPLACE EDIT DISTANCE is optimal unless the *Strong Exponential Time Hypothesis* (SETH) is false, and since then the technique has been applied to various other related problems. Should the reduction from the SETH to the EDIT DISTANCE computation be refined as shown here for the upper bound, it would speak in favor of the optimality of the analysis.

Other measures of difficulty: Abu-Khzam et al. [1] described an algorithm computing the INSERT SWAP EDIT DISTANCE d from S to T in time within $O(1.6181^d m)$, which is polynomial in the size of the input and exponential in the output size d. The output distance d itself can be as large as n, but such instances are not necessarily difficult: Barbay and Pérez-Lantero [3] showed that the gap vector between the `Parikh vectors` separate the hard instances from the easy ones, and we showed that the same can be applied to other EDIT DISTANCES.

But still, among instances of fixed input size, output distance, and imbalance between the `Parikh vectors`, there are instances easier than others (e.g. the computation of the INSERT SWAP EDIT DISTANCE on an instance where all the `insertions` are in the left part of S while all the `swaps` are in the right part of S). A measure which to refine the analysis would be the cost of encoding a *certificate* of the EDIT DISTANCE, one which is easier to check than recomputing the distance itself.

Indexed Dynamic Programming: Our results are close in spirit to those in fixed-parameter complexity, but with an important difference, namely, trying to spot one or more parameters that explain what makes an instance hard or easy. For the computation of the INSERT SWAP and DELETE SWAP EDIT DISTANCES, the size of the alphabet d makes the difference between polynomial time and NP-hardness. However, Barbay and Pérez-Lantero [3] showed that different instances of the same size can exhibit radically different costs-for a given fixed algorithm. The parameterized analysis captures in parameters the cause for such cost differences. We described how the same logic applies to other types of EDIT DISTANCES, and it is likely that similar situations happen with many other algorithms based on dynamic programming, such as the computation of the FRÉCHET DISTANCE [10], the DISCRETE FRÉCHET DISTANCE [8] and the decision problem ORTHOGONAL VECTOR [7].

References

1. Abu-Khzam, F.N., Fernau, H., Langston, M.A., Lee-Cultura, S., Stege, U.: Charge and reduce: A fixed-parameter algorithm for string-to-string correction. Discrete Optimization (DO) **8**(1), 41 – 49 (2011)
2. Backurs, A., Indyk, P.: Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In: Proceedings of the annual ACM Symposium on Theory Of Computing (STOC) (2015)
3. Barbay, J., Pérez-Lantero, P.: Adaptive computation of the swap-insert correction distance. In: Proceedings of the Annual Symposium on String Processing and Information Retrieval (SPIRE). pp. 21–32 (2015)
4. Barbay, J., Pérez-Lantero, P.: Adaptive computation of the swap-insert correction distance. In: ACM Transactions on Algorithms (TALG) (2018), accepted on [2018-05-25 Fri], to appear.
5. Bentley, J.L., Yao, A.C.C.: An almost optimal algorithm for unbounded searching. Information Processing Letters (IPL) **5**(3), 82–87 (1976)
6. Bergroth, L., Hakonen, H., Raita, T.: A survey of longest common subsequence algorithms. In: Proceedings of the 11th Symposium on String Processing and Information Retrieval (SPIRE). pp. 39–48 (2000)
7. Bringmann, K.: Why walking the dog takes time: Fréchet distance has no strongly subquadratic algorithms unless SETH fails. In: Proceedings of the 2014 IEEE 55th Annual Symposium on Foundations of Computer Science. pp. 661–670. FOCS '14, IEEE Computer Society, Washington, DC, USA (2014)
8. Eiter, T., Mannila, H.: Computing discrete Fréchet distance. Tech. rep., Christian Doppler Labor für Expertensyteme, Technische Universität Wien (1994)
9. Golynski, A., Munro, J.I., Rao, S.S.: Rank/select operations on large alphabets: A tool for text indexing. In: Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm (SODA). pp. 368–373. SODA '06, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2006)
10. H., A., M., G.: Computing the Fréchet distance between two polygonal curves. International Journal of Computational Geometry and Applications (IJCGA) **5**(1–2), 75–91 (1995)
11. Hart, M.: Gutenberg project. Online at `https://www.gutenberg.org/` (last accessed on [2018-05-27 Sun])
12. Meister, D.: Using swaps and deletes to make strings match. Theoretical Computer Science (TCS) **562**(0), 606 – 620 (2015)
13. Parikh, R.J.: On context-free languages. J. ACM **13**(4), 570–581 (Oct 1966). https://doi.org/10.1145/321356.321364, `http://doi.acm.org/10.1145/321356.321364`
14. Spreen, T.D.: The Binary String-to-String Correction Problem. Master's thesis, University of Victoria, Canada (2013)

15. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. Journal of the ACM (JACM) **21**(1), 168–173 (1974)
16. Wagner, R.A., Lowrance, R.: An extension of the string-to-string correction problem. Journal of the ACM (JACM) **22**(2), 177–183 (1975)
17. Wagner, R.A.: On the complexity of the extended string-to-string correction problem. In: Proceedings of the annual ACM Symposium on Theory Of Computing (STOC). pp. 218–223. STOC '75, ACM (1975)
18. Witten, I.H., Moffat, A., Bell, T.C.: Managing gigabytes : compressing and indexing documents and images. The Morgan Kaufmann series in multimedia information and systems, San Francisco, Calif. Morgan Kaufmann Publishers (1999)