Symposium on Theoretical Aspects of Computer Science 2009 (Freiburg), pp. 529–540
www.stacs-conf.org

A UNIFIED ALGORITHM FOR ACCELERATING EDIT-DISTANCE COMPUTATION VIA TEXT-COMPRESSION

DANNY HERMELIN[*] [1] AND GAD M. LANDAU[†] [1,2] AND SHIR LANDAU [3] AND OREN WEIMANN[‡] [4]

[1] Department of Computer Science, University of Haifa, Haifa, Israel.
E-mail address: danny@cri.haifa.ac.il,landau@cs.haifa.ac.il

[2] Department of Computer and Information Science, Polytechnic Institute of NYU, NY, USA.

[3] Department of Computer Science,, Bar-Ilan University, Ramat Gan, Israel.
E-mail address: shir.landau@live.biu.ac.il

[4] MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA.
E-mail address: oweimann@mit.edu

Abstract. The edit distance problem is a classical fundamental problem in computer science in general, and in combinatorial pattern matching in particular. The standard dynamic-programming solution for this problem computes the edit-distance between a pair of strings of total length $O(N)$ in $O(N^2)$ time. To this date, this quadratic upper-bound has never been substantially improved for general strings. However, there are known techniques for breaking this bound in case the strings are known to compress well under a particular compression scheme. The basic idea is to first compress the strings, and then to compute the edit distance between the compressed strings.

As it turns out, practically all known $o(N^2)$ edit-distance algorithms work, in some sense, under the same paradigm described above. It is therefore natural to ask whether there is a single edit-distance algorithm that works for strings which are compressed under any compression scheme. A rephrasing of this question is to ask whether a single algorithm can exploit the compressibility properties of strings under any compression method, even if each string is compressed using a different compression. In this paper we set out to answer this question by using *straight-line programs*. These provide a generic platform for representing many popular compression schemes including the LZ-family, Run-Length Encoding, Byte-Pair Encoding, and dictionary methods.

For two strings of total length N having straight-line program representations of total size n, we present an algorithm running in $O(n^{1.4}N^{1.2})$ time for computing the edit-distance of these two strings under any rational scoring function, and an $O(n^{1.34}N^{1.34})$-time algorithm for arbitrary scoring functions. This improves on a recent algorithm of Tiskin that runs in $O(nN^{1.5})$ time, and works only for rational scoring functions.

Key words and phrases: edit distance, straight-line programs, dynamic programming acceleration via compression, combinatorial pattern matching.

[*] Supported by the Adams Fellowship of the Israel Academy of Sciences and Humanities.

[†] Partially supported by the Israel Science Foundation grant 35/05 and the Israel-Korea Scientific Research Cooperation.

‡ Partially supported by the Israel-Korea Scientific Research Cooperation and by the Center for Massive Data Algorithmics (MADALGO) – a center of the Danish National Research Foundation.

© Danny Hermelin, Gad M. Landau, Shir Landau, and Oren Weimann
Creative Commons Attribution-NoDerivs License

STACS 2009
Symposium on Theoretical Aspects of Computer Science 2009 (Freiburg), pp. 561-572
http://drops.dagstuhl.de/opus/volltexte/2009/1804

1. Introduction

The *edit distance* between two strings over a fixed alphabet Σ is the minimum cost of transforming one string into the other via a sequence of character deletion, insertion, and replacement operations [31]. The cost of these elementary editing operations is given by some scoring function which induces a metric on strings over Σ. The simplest and most common scoring function is the Levenshtein distance [16] which assigns a uniform score of 1 for every operation. Determining the edit-distance between a pair of strings is a fundamental problem in computer science in general, and in combinatorial pattern matching in particular, with applications ranging from database indexing and word processing, to bioinformatics [11].

The standard dynamic programming solution for computing the edit distance between a pair of strings $A = a_1a_2 \cdots a_N$ and $B = b_1b_2 \cdots b_N$ involves filling in an $(N+1)\times(N+1)$ table T, with $T[i, j]$ storing the edit distance between $a_1a_2 \cdots a_i$ and $b_1b_2 \cdots b_j$. The computation is done according to the base-case rules given by $T[0, 0] = 0$, $T[i, 0] = T[i-1, 0]+$ the cost of deleting a_i, and $T[0, j] = T[0, j-1]+$ the cost of inserting b_j, and according to the following dynamic programming step:

$$T[i,j] = \min \begin{cases} T[i-1,j] + \text{the cost of deleting } a_i \\ T[i,j-1] + \text{the cost of inserting } b_j \\ T[i-1,j-1] + \text{the cost of replacing } a_i \text{ with } b_j \end{cases} \tag{1.1}$$

Note that as T has $(N+1)^2$ entries, the time-complexity of the algorithm above is $O(N^2)$.

Compression is traditionally used to efficiently store data. In this paper, we focus on using compression to accelerate the dynamic-programming solution for the edit-distance problem described above. The basic idea is to first compress the strings, and then compute the edit distance between the compressed strings. Note that the "acceleration via compression" approach has been successfully applied also to other classical problems on strings. Various compression schemes, such as LZ77 [33], LZW-LZ78 [32], Huffman coding, Byte-Pair Encoding (BPE) [27], Run-Length Encoding (RLE), were employed to accelerate exact string matching [3, 13, 17, 20, 28], subsequence matching [9], approximate pattern matching [2, 12, 13, 24], and more [23].

Regarding edit-distance computation, Bunke and Csirik presented a simple algorithm for computing the edit-distance of strings that compress well under RLE [8]. This algorithm was later improved in a sequence of papers [5, 6, 10, 19] to an algorithm running in time $O(nN)$, for strings of total length N that encode into run-length strings of total length n. In [10], an algorithm with the same time complexity was given for strings that are compressed under LZW-LZ78, where n again is the length of the compressed strings. Note that this algorithm is also $O(N^2/\lg N)$ in the worst-case for any strings over constant-size alphabets.

The first paper to break the quadratic time-barrier of edit-distance computation was the seminal paper of Masek and Paterson [21], who applied the "Four-Russians technique" to obtain a running-time of $O(N^2/\lg N)$ for any pair of strings, and of $O(N^2/\lg^2 N)$ assuming a unit-cost RAM model. Their algorithm essentially exploits repetitions in the strings to obtain the speed-up, and so in many ways it can also be viewed as compression-based. In fact, one can say that their algorithm works on the "naive compression" that all strings over constant-sized alphabets have. A drawback of the the Masek and Paterson algorithm is that it can only be applied when the given scoring function is rational. That is, when all costs of

editing operations are rational numbers. Note that this restriction is indeed a limitation in biological applications, where PAM and evolutionary distance similarity matrices are used for scoring [10, 21]. For this reason, the algorithm in [10] mentioned above was designed specifically to work for arbitrary scoring functions. We mentioned also Bille and Farach-Colton [7] who extend the Masek and Paterson algorithm to general alphabets.

There are two important things to observe from the above: First, all known techniques for improving on the $O(N^2)$ time bound of edit-distance computation, essentially apply acceleration via compression. Second, apart from RLE, LZW-LZ78, and the naive compression of the Four-Russians technique, we do not know how to efficiently compute edit-distance under other compression schemes. For example, no algorithm is known which substantially improves $O(N^2)$ on strings which compress well under LZ77. Such an algorithm would be interesting since there are various types of strings that compress much better under LZ77 than under RLE or LZW-LZ78. In light of this, and due to the practical and theoretical importance of substantially improving on the quadratic lower bound of string edit-distance computation, we set out to answer the following question:

> "Is there a general compression-based edit-distance algorithm that can exploit the compressibility of two strings under *any* compression scheme?"

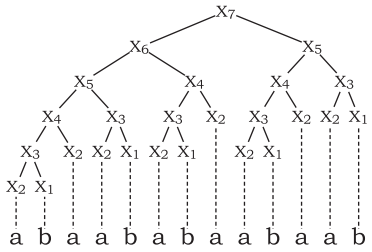
A key ingredient to answering this question, we believe, lies in a notion borrowed from the world of formal languages: The notion of straight-line programs.

1.1. Straight-line programs

A *straight-line program* (SLP) is a context-free grammar generating exactly one string. Moreover, only two types of productions are allowed: $X_i \to a$ where a is a unique terminal, and $X_i \to X_p X_q$ with $i > p, q$ where $X_1, \ldots, X_n$ are the grammar variables. Each variable appears exactly once on the left hand side of a production. The string represented by a given SLP is a unique string corresponding to the last nonterminal X_n. We define the size of an SLP to be n, the number of variables (or productions) it has. The length of the strings that is generated by the SLP is denoted by N. It is important to observe that many SLPs can be exponentially smaller than the string they generate.

Example 1.1. Consider the string *abaababaabaab*. It could be generated by the following SLP, also known as the *Fibonacci SLP*:

$$
\begin{aligned}
X_1 &\to b\\
X_2 &\to a\\
X_3 &\to X_2X_1\\
X_4 &\to X_3X_2\\
X_5 &\to X_4X_3\\
X_6 &\to X_5X_4\\
X_7 &\to X_6X_5
\end{aligned}
$$



Rytter [25] proved that the resulting encoding of most compression schemes including the LZ-family, RLE, Byte-Pair Encoding, and dictionary methods, can be transformed to straight-line programs quickly and without large expansion[1]. In particular, consider an

[1]Important exceptions of this list are statistical compressors such as Huffman or arithmetic coding, as well as compressions that are applied after a Burrows-Wheeler transformation.

LZ77 encoding [33] with n' blocks for a string of length N. Rytter's algorithm produces an SLP-representation with size $n = O(n' \log N)$ of the same string, in $O(n)$ time. Moreover, n lies within a $\log N$ factor from the size of a *minimal* SLP describing the same string. This gives us an efficient logarithmic approximation of minimal SLPs, since computing the LZ77 encoding of a string can be done in linear-time. Note also that any string compressed by the LZ78-LZW encoding can be transformed directly into a straight-line program within a constant factor.

1.2. Our results

Due to Rytter's results, SLPs are perfect candidates for achieving our goal of generalizing compression-based edit-distance algorithms. Indeed, a fast edit-distance algorithm for strings that have small SLP representations, would give a fast algorithm for strings which compress well under the compression schemes generalized by SLPs. Note that since constructing the strings generated by the SLPs requires linear-time in the length of the strings, an $O(N^2)$ algorithm is available via the standard dynamic-programming formulation (1.1). The main result of this paper gives an algorithm which beats this bound:

Theorem 1.2. *Let $\mathcal{A}$ and $\mathcal{B}$ be two SLPs of total size n that respectively generate two string A and B of total size N. Then, given $\mathcal{A}$ and $\mathcal{B}$, one can compute the edit-distance between A and B in $O(n^{1.4}N^{1.2})$ time for any rational scoring function.*

We can remove the dependency of rational scoring schemes in Theorem 1.2, recalling that arbitrary scoring schemes are important for biological applications. We obtain the following secondary result for arbitrary scoring functions:

Theorem 1.3. *Let $\mathcal{A}$ and $\mathcal{B}$ be two SLPs of total size n that respectively generate two string A and B of total size N. Then, given $\mathcal{A}$ and $\mathcal{B}$, one can compute the edit-distance between A and B in $O(n^{1.34}N^{1.34})$ time for any arbitrary scoring function.*

In the last part of the paper, we explain how the four-russians technique can also be incorporated into our SLP edit-distance scheme. We obtain a very simple algorithm that matches the performance of [10] in the worst-case. That is, we obtain a four-russian like algorithm with an $\Omega(\lg N)$ speed-up which can handle arbitrary scoring functions, unlike the Masek and Paterson algorithm which works only for rational functions. We add this algorithm to our presentation not only for its practical importance, but also to emphasize the fact that SLPs provide a framework which allows an almost perfect generalization of compression-based edit-distance algorithms.

1.3. Related Work

Rytter *et al.* [14] was the first to consider SLPs in the context of pattern matching, and other subsequent papers also followed this line [15, 22]. In [25] and [17] Rytter and Lifshits took this work one step further by proposing SLPs as a general framework for dealing with pattern matching algorithms that are accelerated via compression. However, the focus of Lifshits was on determining whether or not these problems are polynomial in n or not. In particular, he gave an $O(n^3)$-time algorithm to determine equality of SLPs [17], and he established hardness for the edit distance [18], and even for the hamming distance problems [17]. Nevertheless, Lifshits posed as an open problem the question of whether or not there is an $O(nN)$ edit-distance algorithm for SLPs. Here, our focus is on algorithms

which break the quadratic $O(N^2)$ time-barrier, and therefore all algorithms with running-times between $O(nN)$ and $O(N^2)$ are interesting for us.

Recently, Tiskin [29] gave an $O(nN^{1.5})$ algorithm for computing the longest common subsequence between two SLPs, an algorithm which can be extended at constant-factor cost to compute the edit-distance between the SLPs under any rational scoring function. Observe that our algorithm for arbitrary scoring functions in Theorem 1.3 is already faster than Tiskin's algorithm for most values of N and n. Also, it has the advantage of being much more simpler to implement. As for our main algorithm of Theorem 1.2, our faster running-time is achieved also by utilizing some of the techniques used by Tiskin in a more elaborate way.

2. The $DIST$ Table

The central dynamic-programming tool we use in our algorithms is the $DIST$ table, a simple and handy data-structure which was originally introduced by Apostolico *et al.* [4], and then further developed by others in [10, 26]. In the following section we briefly review basic facts about this tool that are essential for understanding our results, following mostly the presentation in [10]. We begin with the so-called dynamic-programming grid, a graph representation of edit-distance computation on which $DIST$ tables are defined.

Consider the standard dynamic programming formulation (1.1) for computing the edit-distance between two strings $A = a_1 a_2 \cdots a_N$ and $B = b_1 b_2 \cdots b_N$. The *dynamic-programming grid* associated with this program, is an acyclic-directed graph which has a vertex for each entry of T (see Figure 1). The vertex corresponding to $T[i, j]$ is associated with a_i and b_j, and has incoming edges according to (1.1) – an edge from $T[i-1, j]$ whose weight is the cost of deleting a_i, an edge from $T[i, j-1]$ whose weight is the cost of inserting b_j, and an edge from $T[i-1, j-1]$ whose weight is the cost of replacing a_i with b_j. The *value* at the vertex corresponding to $T[i, j]$ is the value stored in $T[i, j]$, *i.e.* the edit-distance between the length i prefix of A and the length j prefix of B. Using the dynamic-programming grid G, we reduce the problem of computing the edit-distance between A and B to the problem of computing the weight of the lightest path from the upper-left corner to bottom-right corner in G.

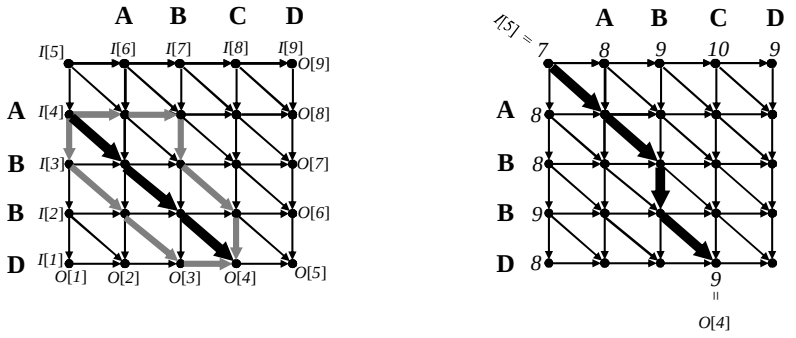


Figure 1: A subgraph of a Levenshtein distance dynamic program graph. On the left, $DIST[4, 4]$ (in bold) gives the minimum-weight path from $I[4]$ to $O[4]$. On the right, the value 9 of $O[4]$ is computed by $min_i I[i] + DIST[i, 4]$.

We will work with sub-grids of the dynamic-programming grid that will be referred to as *blocks*. The *input vertices* of a block are all vertices in the first row and column of the block, while its *output vertices* are all vertices in the last row and column. Together, the input and output vertices are referred to as the *boundary* of the block. The substrings of A and B associated with the block are defined in the straightforward manner according to its first row and column. Also, for convenience purposes, we will order the input and output vertices, with both orderings starting from the vertex in bottom-leftmost corner of the block, and ending at the vertex in the upper-rightmost corner. The ith input vertex and jth output vertex are the ith and jth vertices in these orderings. We next give the definition of $DIST$ tables, defined over blocks of G.

Definition 2.1 ($DIST$ [4])**.** Let G' be a block in G with x input vertices and x output vertices. The *DIST table* corresponding to G' is an $x \times x$ matrix, with $DIST[i, j]$ storing the weight of the minimum-weight path from the ith input to the jth output in G, and otherwise ∞ if no such paths exists.

It is important to notice that the values at the output vertices of a block are completely determined by that values at its input and its corresponding $DIST$ table. In particular, if $I[i]$ and $O[j]$ are the values at the ith input vertex and jth output vertex of a block G' of G, then

$$O[j] = \min_{1 \le i \le x} I[i] + DIST[i, j]. \tag{2.1}$$

Equation 2.1 implies not only the input-output relation of the dynamic-programming values of a block, but also that the values at the output vertices can be computed in linear time from the values at the input vertices. Indeed, by (2.1), the values at the output vertices of G' are given by the column minima of the matrix $I+DIST$. Furthermore, by a simple modification of all ∞ values in $I+DIST$, we get what is known as a *totally-monotone matrix* [10]. Now, Aggarwal *et al.* [1] gave a simple recursive algorithm, nicknamed SMAWK in the literature, that computes all column minima of an $x \times x$ totally-monotone matrix by querying only $O(x)$ elements of the matrix. It follows that using SMAWK we can compute the output values of G' in $O(x)$ time.

Let us now discuss how to efficiently construct the $DIST$ table corresponding to a block in G. Observe that this can be done quite easily in $O(x^3)$ time, for blocks with boundary size $O(x)$, by computing the standard dynamic-programming table between every prefix of A against B and every prefix of B against A. Each of these dynamic-programming tables contains all values of a particular row in the $DIST$ table. In [4], Apostolico *et al.* show an elegant way to reduce the time-complexity of this construction to $O(x^2 \lg x)$. In the case of rational scoring functions, the complexity can be further reduced to $O(x^2)$ as shown by Schmidt [26].

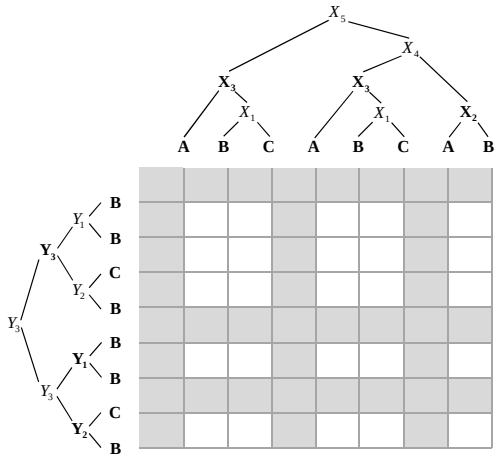
3. Acceleration via Straight-Line Programs

In the following section we describe a generic framework for accelerating the edit distance computation of two strings which are given by their SLP representation. This framework will later be used for explaining all our algorithms. We will refer throughout the paper to this framework as the *block edit-distance* procedure.

Let $\mathcal{A}$ and $\mathcal{B}$ be two SLP representations of a pair of strings A and B, and for ease of presentation assume that $|\mathcal{A}| = |\mathcal{B}| = n$ and $|A| = |B| = N$. Recall the definition in

Section 2 for the dynamic-programming grid corresponding to A and B. The general idea behind the block edit-distance procedure is to partition this grid into disjoint blocks, and then to compute the edit-distance between A and B at the cost of computing the values at the boundary vertices of each block. This is achieved by building in advance a repository containing all $DIST$ tables corresponding to blocks in the partition. To efficiently construct this repository, we show how to partition the grid in a way which induces many block repeats. This is possible by utilizing substring repeats in A and B that are captured in $\mathcal{A}$ and $\mathcal{B}$, and imply block repeats in the partitioning of G. The edit-distance of A and B is then computed by propagating the dynamic programming values at the boundary vertices of the blocks using the $DIST$ tables in the repository and SMAWK. Before giving a complete description of this algorithm, we need to introduce the notion of xy-partition.

Definition 3.1 (xy-partition)**.** An xy-partition is a partitioning of G into disjoint blocks such that every block has boundary of size $O(x)$, and there are $O(y)$ blocks in each row and column. In addition, we require each pair of substrings of A and B associated with a block to be generated by a pair of SLP variables in $\mathcal{A}$ and $\mathcal{B}$.



An xy-partition of an edit distance graph for two SLPs generating the strings "ABCABCAB" and "BBCBBBCB". The white blocks are the ones of the partition and their corresponding SLP variables are marked in bold. Notice that there are nine blocks in the partition but only six of them are distinct.

Figure 2: An xy-partition.

An xy-partition of G is a partition with a specific structure, but more importantly, one where each substring is generated by a unique SLP variable of $\mathcal{A}$ and $\mathcal{B}$. This latter requirement allows us to exploit the repetitions of A and B captured by their SLPs. We next give a complete description of the block edit distance procedure. It assumes an xy-partition of G has already been constructed. Section 4 explains how to construct such partitions.

Block Edit Distance

(1) Construct a repository with the $DIST$ tables corresponding to each block in the xy-partition.
(2) Fill-in the first row and column of G using the standard base-case rules.
(3) In top-to-bottom and left-to-right manner, identify the next block in the partition of G and use its input and the repository to compute its output using (2.1).
(4) Use the outputs in order to compute the inputs of the next blocks using (1.1).
(5) The value in the bottom-rightmost cell is the edit distance of A and B.

Apart from the repository construction in step 1, all details necessary for implementing the block edit-distance procedure are by now clear. Indeed, steps 2 and 5 are trivial, and step 4 is done via the standard dynamic-programming formulation of (1.1). Furthermore, the SMAWK computation of output values of a block, given its input values plus its corresponding $DIST$ table (step 3), is explained in Section 2. We next show that, as we are working with xy-partitions where each block is associated with an SLP variable, we can compute a repository containing all $DIST$ necessary which is rather small.

The first crucial observation for this, is that any two blocks associated with the same pair of substrings A' and B' have the same $DIST$ table. This is immediate since any such pair of blocks have identical edge-weights.

Observation 3.2. A pair of substrings A', B' uniquely identify the $DIST$ table of a block.

Since we required each substring in the xy-partition of G to be generated by some SLP variable, the above observation actually suggests that the number of different $DIST$ tables is bounded by the number of variable pairs $X \in \mathcal{A}$ and $Y \in \mathcal{B}$:

Observation 3.3. The number of different $DIST$ tables corresponding to any xy-partition is $O(n^2)$.

Therefore, combining the two observations above, we know that a repository containing a $DIST$ tables for each SLP variable pair $X \in \mathcal{A}$ and $Y \in \mathcal{B}$ will not be too large, and that it will contain a table corresponding to each block in our given xy-partition at hand. We can therefore state the following lemma:

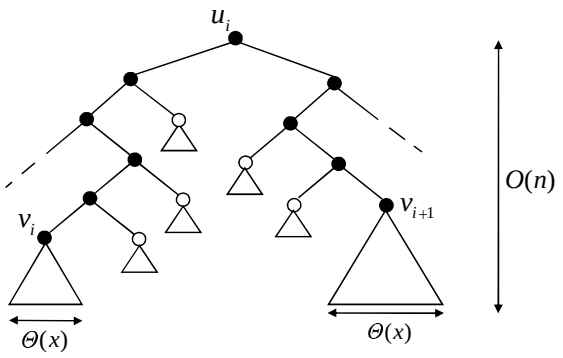
Lemma 3.4. *The block edit-distance procedure runs in $O(n^2x^2 \lg x + Ny)$ time.*

Proof. We analyze the time complexity of each step in the block edit-distance procedure separately. Step 1 can be performed in $O(n^2x^2 \lg x)$ time, as we can construct every $DIST$ table in $O(x^2 \lg x)$ time (see Section 2), and the total number of such distinct matrices is $O(n^2)$. Step 2 can be done trivially in $O(N)$ time. Then, step 3 takes $O(x)$ time per block by using the SMAWK algorithm as explained in Section 2. Step 4 also takes $O(x)$ time per block as it only computes the values in the $O(x)$ vertices adjacent to the output vertices. The total time complexity of steps 3 and 4 is thus equal to the total number of boundary vertices in the xy-partition of G, and therefore to $O(Ny)$. Accounting for all steps together, this gives us the time complexity stated in the lemma. ∎

4. Constructing an xy-partition

In this section we discuss the missing component of Section 3, namely the construction of xy-partitions. In particular, we complete the proof of Theorem 1.3 by showing how to efficiently construct an xy-partition where $y = O(nN/x)$ for every $x \leq N$. Together with Lemma 3.4, this implies an $O(n^{\frac{4}{3}}N^{\frac{4}{3}} \lg^{\frac{1}{3}} N) = O(n^{1.34}N^{1.34})$ time algorithm for arbitrary scoring functions by considering $x = N^{\frac{2}{3}}/(n \lg N)^{\frac{1}{3}}$. In the remainder of this section we prove the following lemma.

Lemma 4.1. *For every $x \leq N$ there exists an xy-partition with $y = O(nN/x)$. Moreover, this partition can be found in $O(N)$ time.*



The key-vertices v_i and v_{i+1} both generate substrings of length $\Theta(x)$, and their least common ancestor is u_i. The white vertices "hanging" of the v_i-to-v_{i+1} path are the added key-vertices. Together, they generate the substring that lies between the substrings of v_i and v_{i+1}.

Figure 3: A closer look on the parse tree of an SLP $\mathcal{A}$.

To prove the lemma, we show that for every SLP $\mathcal{A}$ generating a string A and every $x \leq N$, one can partition A into $O(nN/x)$ disjoint substrings, each of length $O(x)$, such that every substring is generated by some variable in $\mathcal{A}$. This defines a subset of variables in both input SLPs which together defined our desired xy-partition. To partition A, we first identify $O(N/x)$ grammar variables in $\mathcal{A}$ each generating a disjoint substring of length between x and $2x$. We use these variables to partition A. We then show that the substrings of A that are still not associated with a variable can each be generated by $O(n)$ additional variables. Furthermore, these $O(n)$ variables each generate a string of length bounded by x. We add all such variables to our partition of A for a total of $O(nN/x)$ variables.

Consider the parse tree of $\mathcal{A}$. We want to identify $O(nN/x)$ *key-vertices* such that every key-vertex generates a substring of length $O(x)$, and A is a concatenation of substrings generated by key-vertices. We start by marking every vertex v that generates a substring of length greater than x as a key-vertex iff both children of v generate substrings of length smaller than x. This gives us $\ell \leq N/x$ key-vertices so far, each generating a substring of length $\Theta(x)$ (see Figure 3). But we are still not guaranteed that these vertices cover A entirely.

To fix this, consider the ordering $v_1, v_2, \ldots, v_\ell$ on the current key-vertices induced by a left-to-right postorder traversal of the parse tree. This way, v_{i+1} is "to the right of" v_i. If every v_i generates the substring A_i then $A = A'_1 A_1 A'_2 A_2 \cdots A'_\ell A_\ell A'_{\ell+1}$, where every A_i is of length $\Theta(x)$, and every A'_i is the "missing" substring of A that lies between A_{i-1} and A_i. We now show that every A'_i is a concatenation of substrings of length smaller than x generated by at most $O(n)$ vertices.

Let u_i be the lowest common ancestor of v_i and v_{i+1} and let P_i (resp. P_{i+1}) be the unique path between u_i and v_i (resp. v_{i+1}). For every vertex $v \in P_i - \{u_i\}$ such that v's left child is also in P_i mark v's right child as a key-vertex. Similarly, for every vertex $v \in P_{i+1} - \{u_i\}$ such that v's right child is also in P_{i+1} mark v's left child as a key-vertex. It is easy to verify that A'_i is the concatenation of substrings generated by these newly marked key-vertices. There are at most $2n$ of these key-vertices since the depth of the parse tree is bounded by the number of different SLP variables. Moreover, they each generate a substring of length smaller than x for the following reason. Assume for contradiction that one of them generates a string of length greater than x. This would imply the existence of some vertex between v_i and v_{i+1} in the $v_1, v_2, \ldots, v_\ell$ ordering.

To conclude, we showed that $A = A'_1 A_1 A'_2 A_2 \cdots A'_\ell A_\ell A'_{\ell+1}$ where $\ell \leq N/x$, every A_i is of length $\Theta(x)$ and is generated by one vertex, and every A'_i is a concatenation of $O(n)$ substrings each of length smaller than x and generated by one vertex. Overall, we get that $y = O(n\ell) = O(nN/x)$ vertices suffice to generate A for every $x \leq N$. It is easy to see that we can identify these vertices in $O(N)$ time thus proving Lemma 4.1. By choosing $x = N^{\frac{2}{3}}/(n \lg N)^{\frac{1}{3}}$, and using the block edit distance time complexity of Lemma 3.4, this implies an $O(n^{1.34}N^{1.34})$ time algorithm for arbitrary scoring functions.

5. Improvement for Rational Scoring Functions

In this section we show that in the case of rational scoring functions, the time complexity of the block edit distance procedure can be reduced substantially by using a recursive construction of the $DIST$ tables. In particular, we complete the proof of Theorem 1.2 by showing that in this case the repository of $DIST$ tables can be computed in $O(n^2x^{1.5})$ time. This implies an $O(n^{1.4}N^{1.2})$ time algorithm for rational scoring functions by considering $x = N^{0.8}/n^{0.4}$ and the xy-partition with $y = nN/x$.

Before we describe how to compute the repository in $O(n^2x^{1.5})$ time, we need to introduce some features that $DIST$ tables over rational scoring functions have. The first property, discovered by Schmidt [26], is what is known as the succinct representation property: Any $x \times x$ $DIST$ table can be succinctly stored using only $O(x)$ space. This follows from considering the vector obtained by subtracting a $DIST$ column from the column to its right, and observing that this vector has only a constant number of value changes. The second property is that succinct representations allow to efficiently merge two $DIST$ tables. That is, if D_1 and D_2 are two $DIST$ tables, one between a pair of substrings A' and B' and the other between A' and B'', then we refer to the $DIST$ table between A' and $B'B''$ as the product of *merging* D_1 and D_2. A recent important result of Tiskin [30] shows how to utilize the succinct representation of $DIST$ tables in order to merge two succinct $x \times x$ $DIST$ tables in $O(x^{1.5})$ time.

Lemma 5.1. *The block edit distance algorithm runs in $O(n^2x^{1.5} + Ny)$ time in case the underlying scoring function is rational.*

Proof. To prove the lemma it suffices to show how to compute the repository of $DIST$ tables in step 1 of the block edit-distance procedure in $O(n^2x^{1.5})$ time, in case the underlying scoring function is rational. We will work with succinct representations of the $DIST$ tables as described above. Say $X \to X_pX_q$ and $Y \to Y_sY_t$ are two rules in the SLPs $\mathcal{A}$ and $\mathcal{B}$ respectively. To compute the $DIST$ table that corresponds to the strings generated by X and Y, we first recursively compute the four $DIST$ tables that correspond to the pairs (X_p, Y_s), (X_p, Y_t), (X_q, Y_s), and (X_q, Y_t). We then merge these four tables to obtain the $DIST$ table that corresponds to (X, Y). To do so we use Tiskin's procedure to merge (X_p, Y_s) with (X_p, Y_t) into (X_p, Y_sT_t), then merge (X_q, Y_s) with (X_q, Y_t) into (X_q, Y_sT_t), and finally we merge (X_p, Y_sT_t) and (X_q, Y_sT_t) into $(X_pX_q, Y_sT_t) = (X, Y)$. This recursive procedure computes each succinct $DIST$ table by three merge operations, each taking $O(x^{1.5})$ time and $O(x)$ space. Since the number of different $DIST$ tables is bounded by $O(n^2)$, the $O(n^2x^{1.5})$ time for constructing the repository follows. ∎

To conclude, we have shown an $O(n^2x^{1.5} + Ny)$ time algorithm for computing the edit distance. Using the xy-partition from Lemma 4.1 with $x = N^{0.8}/n^{0.4}$ and $y = nN/x$, we get a time complexity of $O(n^{1.4}N^{1.2})$.

6. Four-Russian Interpretation

In the previous sections we showed how SLPs can be used to speed up the edit distance computation of strings that compress well under some compression scheme. In this section, we conclude the presentation of our SLP framework by presenting an $\Omega(\lg N)$ speed-up for strings that do not compress well under any compression scheme. To do so, we adopt the Four Russions approach of Masek and Paterson [21] that utilizes a naive property that every string over a fixed alphabet has. Namely, that short enough substrings must appear many times. However, while the Masek and Paterson algorithm can only handle rational scoring functions, the SLP version that we propose can handle arbitrary scoring functions.

Consider a string A of length N over an alphabet Σ. The parse tree of the naive SLP $\mathcal{A}$ is a complete binary tree with N leaves[2]. This way, for every $x \leq N$ we get that A is the concatenation of $O(N/x)$ substrings each of length $\Theta(x)$ and each can be generated by some variable in $\mathcal{A}$. This partition of A suggests an xy-partition in which $y = N/x$. At first glance, this might seem better than the partition guarantee of Lemma 4.1 in which $y = nN/x$. However, notice that in the naive SLP we have $n \geq N$ so we can not afford to compute a repository of $O(n^2)$ $DIST$ tables.

To overcome this problem, we choose x small enough so that $|\Sigma|^x$, the number of possible substrings of length x, is small. In particular, by taking $x = \frac{1}{2}\log_{|\Sigma|} N$ we get that the number of possible substrings of length x is bounded by $|\Sigma|^x = \sqrt{N}$. This implies an xy-partition in which $x = \frac{1}{2}\log_{|\Sigma|} N$, $y = N/x$, and the number of distinct blocks n' is $O(N)$. Using this partition, we get that the total construction time of the $DIST$ repository is $O(n'x^2 \lg x)$. Similar to Lemma 3.4, we get that the total running time of the block edit distance algorithm is $O(n'x^2 \lg x + Ny)$ which gives $O(N^2/\lg N)$.

References

[1] A. Aggarwal, M.M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987.

[2] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Comp. and Sys. Sciences*, 52(2):299–307, 1996.

[3] A. Amir, G.M. Landau, and D. Sokol. Inplace 2d matching in compressed images. In *Proc. of the 14th annual ACM-SIAM Symposium On Discrete Algorithms, (SODA)*, pages 853–862, 2003.

[4] A. Apostolico, M.J. Atallah, L.L. Larmore, and S.McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM Journal on Computing*, 19(5):968–988, 1990.

[5] A. Apostolico, G.M. Landau, and S. Skiena. Matching for run length encoded strings. *Journal of Complexity*, 15(1):4–16, 1999.

[6] O. Arbell, G. M. Landau, and J. Mitchell. Edit distance of run-length encoded strings. *Information Processing Letters*, 83(6):307–314, 2001.

[7] P. Bille and M. Farach-Colton. Fast and compact regular expression matching. *CoRR*, 2005.

[8] H. Bunke and J. Csirik. An improved algorithm for computing the edit distance of run length coded strings. *Information Processing Letters*, 54:93–96, 1995.

[9] P. Cégielski, I. Guessarian, Y. Lifshits, and Y. Matiyasevich. Window subsequence problems for compressed texts. In *Proc. of the 1st symp. on Computer Science in Russia (CSR)*, pages 127–136, 2006.

[2]We assume without loss of generality that N is a power of 2.

[10] M. Crochemore, G.M. Landau, and M. Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM Journal on Computing*, 32:1654–1673, 2003.
[11] D. Gusfield. *Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology.* Cambridge University Press, 1997.
[12] J. Karkkainen, G. Navarro, and E. Ukkonen. Approximate string matching over Ziv-Lempel compressed text. In *Proc. of the 11th symposium on Combinatorial Pattern Matching (CPM)*, pages 195–209, 2000.
[13] J. Karkkainen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. of the 3rd South American Workshop on String Processing (WSP)*, pages 141–155, 1996.
[14] M. Karpinski, W. Rytter, and A. Shinohara. Pattern-matching for strings with short descriptions. In *Proc. of the 6th symposium on Combinatorial Pattern Matching (CPM)*, pages 205–214, 1995.
[15] E. Lehman and A. Shelat. Approximation algorithms for grammar-based compression. In *Proc. of the 13th annual ACM-SIAM Symposium On Discrete Algorithms, (SODA)*, pages 205–212, 2002.
[16] V.I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.
[17] Y. Lifshits. Processing compressed texts: A tractability border. In *Proc. of the 18th symposium on Combinatorial Pattern Matching (CPM)*, pages 228–240, 2007.
[18] Y. Lifshits and M. Lohrey. Querying and embedding compressed texts. In *Proc. of the 31st international symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 681–692, 2006.
[19] V. Makinen, G. Navarro, and E. Ukkonen. Approximate matching of run-length compressed strings. In *Proc. of the 12th Symposium On Combinatorial Pattern Matching (CPM)*, pages 1–13, 1999.
[20] U. Manber. A text compression scheme that allows fast searching directly in the compressed file. In *Proc of the 5th Symposium On Combinatorial Pattern Matching (CPM)*, pages 31–49, 1994.
[21] W.J. Masek and M.S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20, 1980.
[22] M. Miyazaki, A. Shinohara, and M. Takeda. An improved pattern matching algorithm for strings in terms of straight-line programs. In *Proc. of the 8th symposium on Combinatorial Pattern Matching (CPM)*, pages 1–11, 1997.
[23] S. Mozes, O. Weimann, and M. Ziv-Ukelson. Speeding up HMM decoding and training by exploiting sequence repetitions. In *Proc. of the 18th symposium on Combinatorial Pattern Matching (CPM)*, pages 4–15, 2007.
[24] G. Navarro, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Faster approximate string matching over compressed text. In *Proc. of the 11th Data Compression Conference (DCC)*, pages 459–468, 2001.
[25] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.
[26] J.P. Schmidt. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM Journal on Computing*, 27(4):972–992, 1998.
[27] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Byte Pair encoding: A text compression scheme that accelerates pattern matching. *Technical Report DOI-TR-161, Department of Informatics, Kyushu University*, 1999.
[28] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Speeding up pattern matching by text compression. In *Proc. of the 4th Italian Conference Algorithms and Complexity (CIAC)*, pages 306–315, 2000.
[29] A. Tiskin. Faster subsequence recognition in compressed strings. *J. of Mathematical Sciences, to appear.*
[30] A. Tiskin. All semi-local longest common subsequences in subquadratic time. In *Proc. of the 1st Computer Science symposium in Russia (CSR)*, pages 352–363, 2006.
[31] R. Wagner and M. Fischer. The string-to-string correction problem. *J. of the ACM*, 21(1):168–173, 1974.
[32] J. Ziv and A. Lempel. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.
[33] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

This work is licensed under the Creative Commons Attribution-NoDerivs License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/3.0/.