Faster Algorithm for Computing the Edit Distance between SLP-Compressed Strings

Paweł Gawrychowski*

Institute of Computer Science, University of Wrocław, Poland
Max-Planck-Institute für Informatik, Saarbrücken, Germany
gawry@cs.uni.wroc.pl

Abstract. Given two strings described by SLPs of total size n, we show how to compute their edit distance in $\mathcal{O}(nN\sqrt{\log \frac{N}{n}})$ time, where N is the sum of the strings length. The result can be generalized to any rational scoring function, hence we improve the existing $\mathcal{O}(nN \log N)$ [10] and $\mathcal{O}(nN \log \frac{N}{n})$ [4] time solutions. This gets us even closer to the $\mathcal{O}(nN)$ complexity conjectured by Lifshits [7]. The basic tool in our solution is a linear time procedure for computing the max-product of a vector and a unit-Monge matrix, which might be of independent interest.

1 Introduction

The edit distance is a basic measure of similarity between strings, commonly used in real-life applications. The dynamic programming algorithm for computing this distance is usually among the very first examples covered in an algorithms and data structures course. Unfortunately, the quadratic running time of such algorithm makes it useless when we have to deal with really large data. While it is possible to achieve better running times in some specific cases [6], by exploiting the RAM model [8], or by allowing approximate solutions [1], it seems that there is still some room for improvement here. One promising direction is to consider strings which are given in a compressed representation, with the hope that if the data is really big, it might be, in some sense, somehow redundant. Hence if we manage to bound the running time in terms of the size of this compressed representation, we might hope to get a substantial speed-up in some situations.

A natural and very powerful method of representing compressed strings are straight-line programs. Computing the edit distance between strings defined by straight-line programs was already considered a number of times, with [10] giving $\mathcal{O}(nN \log N)$ time solution, and [4] (improved version of [3]) decreasing the complexity to $\mathcal{O}(nN \log \frac{N}{n})$. In this paper we give a faster algorithm based on a similar idea. In order to achieve a better running time, we prove that max-multiplication of vectors and unit-Monge matrices requires just linear time, hence improving the $\mathcal{O}(n \log \log n)$ time solution due to Tiskin [9]. This tool might be of independent interest, as it could find further uses in the approximate pattern matching area.

* Supported by MNiSW grant number N N206 492638, 2010–2012.

L. Calderón-Benavides et al. (Eds.): SPIRE 2012, LNCS 7608, pp. 229–236, 2012.

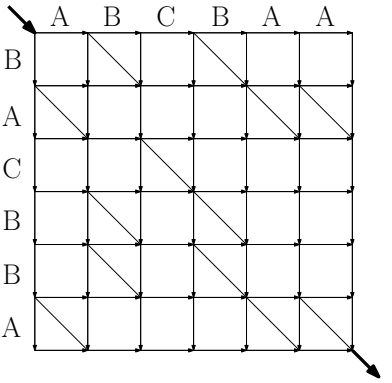

Fig. 1. Interpreting LCS as a highest score path in a grid graph

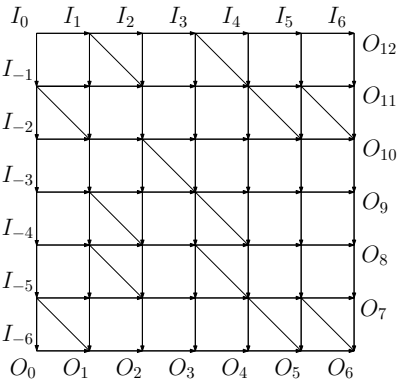

Fig. 2. Input and output vertices. Some vertices are both input and output.

2 Preliminaries

We will consider strings over a fixed finite alphabet Σ. The strings will be described using *straight-line programs*, which are context-free grammars in Chomsky normal form with exactly one production for each nonterminal, hence describing exactly one word. The size of such SLP is simply the number of rules. The edit distance between two strings $a, b \in \Sigma^*$ is the smallest number of operations required to transform a into b, assuming that in a single step we can delete, insert or change a single character. A basic fact concerning the edit distance is that computing it reduces to finding the longest common subsequence. Sometimes we are interested in the *weighted* edit distance, where all operations have costs depending on the characters involved. In this paper we will consider only the case when those costs are rational, which is usually called the *rational scoring function* case.

We are interested in computing the edit distance between two strings a and b of total length N defined by SLPs of total size n. We will show how to compute their longest common subsequence in $\mathcal{O}(nN\sqrt{\log \frac{N}{n}})$ time. Using the blow-up technique of Tiskin [9], this can be generalized to computing the edit distance for any rational scoring function.

The very basic method of computing the longest common subsequence of a and b uses dynamic programming to compute the LCS of all possible pairs of prefixes in $\mathcal{O}(|a||b|)$ time, which is usually seen as calculating the highest score path between the two opposite corner vertices in the corresponding grid graph, see Figure 1. It turns out that if one is interested in computing the paths between all pairs of boundary vertices, namely in calculating $H_{a,b}(i, j)$ being the best path between the i-th input and j-th output (input being the left and top boundary, and output being the right and bottom boundary, see Figure 2), the matrix $H_{a,b}$ has a very special structure, namely it is unit-anti-Monge. It means that if we number the input and output vertices as shown on

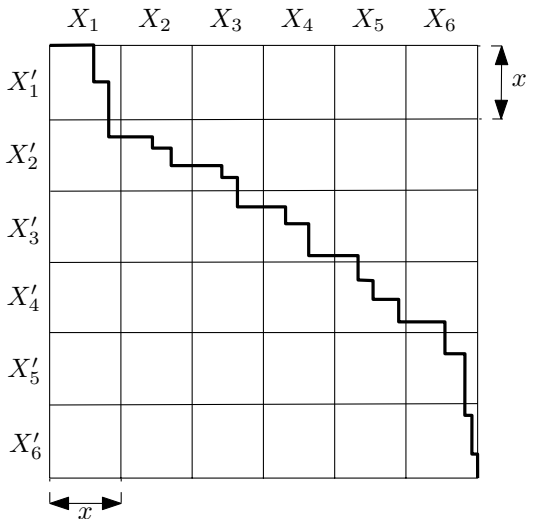

Fig. 3. Cutting the table into $x \times x$ blocks. We need the values on all boundaries.

Figure 2, and let $H_{a,b}(i,j) = j - i < 0$ if $j < i$, the matrix can be represented as $H_{a,b}(i,j) = j - i - P^{\Sigma}(i,j)$, where P is a permutation matrix (meaning that it contains at most one in each row and column, and zeroes everywhere else), and $P^{\Sigma}(i,j) = \sum_{i' \geq i, j' \leq j} P(i',j')$. The reader is kindly requested to consult Section 3.2 of [9] for an example and a more detailed explanation. It turns out that the max-product of such matrices can be computed very efficiently using a surprising result of Tiskin [10], where the max-product of two matrices A and B is a matrix C such that $C(i,k) = \max_j A(i,j) + B(j,k)$ (similarly, the min-product is C such that $C(i,k) = \min_j A(i,j) + B(j,k)$).

Theorem 1 ([10], Theorem 3.3). *Given two $x \times x$ permutation matrices P_1 and P_2, we can compute P_3 such that P_3^{Σ} is the min-product of P_1^{Σ} and P_2^{Σ} in $\mathcal{O}(x \log x)$ time.*

The above theorem can be directly used to compute the representation of $H_{a'a'',b}$ given the representations of $H_{a',b}$ and $H_{a'',b}$. If the lengths of a', a'', b are all bounded by x, the running time of such computation is $\mathcal{O}(x \log x)$.

Throughout the paper, we assume the usual unit-cost word RAM model with word size $\Omega(\log N)$.

3 The Algorithm

The high-level idea is the same as in the previous solutions [4,10]. We would like to compute the whole $N \times N$ table used by the naive dynamic programming solution. This is clearly infeasible, but we will show that one can cut it into fragments of sizes roughly $x \times x$ so that all $2\frac{N}{x}$ values on their boundaries can

be computed efficiently, see Figure 3. More precisely, for each such fragment we will precompute a function $H(i,j)$ equal to the best scoring path between the i-th input and j-th ouptut. This functions depends only on the corresponding substrings of a and b, so whenever both substrings are the same, we can reuse the representation of H. The partition will be chosen so that the number of non-equivalent fragments will be roughly n^2 and we will be able to compute the representations of all corresponding matrices in $\mathcal{O}(n^2 x \log x)$ time. Then we will repeatedly max-multiply the vector representing all values on the left and top boundary of the next fragment with its corresponding matrix to get the values on its right and bottom boundary. We will show how to perform each such multiplication in $\mathcal{O}(x)$ time, hence achieving the total complexity $\mathcal{O}(n^2 x \log x + (\frac{N}{x})^2 x)$.

We start with showing how one can transform a SLP in order to cut the original string into fragments of roughly the same size which can be derived from single nonterminals. This is very similar to the x-partitions of [4], but allows us to directly bound the number of nonterminals in the new SLP. It might be possible to also derive such transformation from the construction of Charikar *et al.* [2], who showed how one can make a SLP balanced, in a certain sense. We prefer to give a simple direct proof, though. Note that in the statement below by SLP we actually mean a collection of SLPs with shared rules, each describing a single string.

Lemma 1. *Given an SLP of size n describing a string of length N and a parameter x, we can construct in $\mathcal{O}(n + \frac{N}{x})$ time a new SLP of size $\mathcal{O}(n)$ with all nonterminals describing strings of length at most x and a representation of the original string as a concatenation of $\mathcal{O}(\frac{N}{x})$ new nonterminals.*

Proof. Call a nonterminal (from the original program) *small* if it describes a string of length at most x, and *big* otherwise. Each small nonterminal is directly copied into the new program. Then we run the following rewriting process: start with $t = S$, where S is the starting symbol. As long as t contains a big nonterminal $A \to BC$, where B, C are also big, replace A with BC. As a result we get t of length at most $\frac{N}{x}$ describing the original string in which each nonterminal A is either small or derives $A \to BC$ with exactly one of B, C small. We would like to somehow rewrite those remaining big nonterminals. Doing it naively might create an excessive increase in the length.

We define the *right graph* as follows: each big nonterminal is a vertex, and if $A \to BC$ with B big and C small, we create an edge $A \xrightarrow{C} B$. Symmetrically, we define the *left graph*, where for each $A \to BC$ with B small and C big we create an edge $A \xrightarrow{B} C$. Note that both graphs are in fact trees. The *core* of a nonterminal A is defined recursively as follows:

1. if $A \to BC$ with both B and C small, then the core of A is BC,
2. if $A \to BC$ with B small and C big, then the core of A is the core of C,
3. if $A \to BC$ with B big and C small, then the core of A is the core of B.

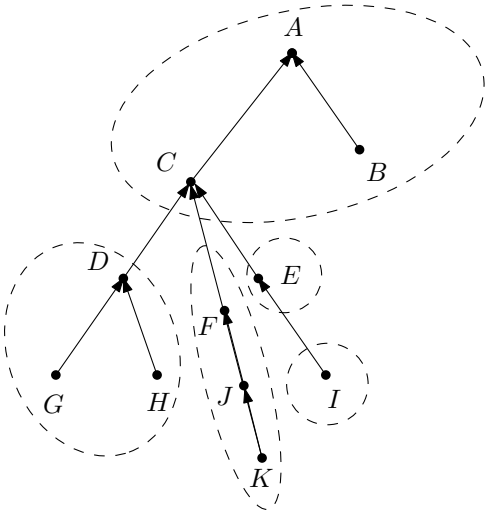

Fig. 4. A sample right graph and its partition into chunks. A, D, F, E and I are the frontiers. Then, for example, the path from J to the root is $\mathrm{path}(C)\,\mathrm{path}(J)$.

Then for any remaining big nonterminal A we would like to replace it with the label of the path to the root in the left graph, its *core*, and the label of the path from the root in the right graph. Because of the symmetry, it is enough to show how to construct a short description of each path in the right graph. We could simply define a new nonterminal $\mathrm{path}(A)$ for each vertex A by adding a production $\mathrm{path}(A) \to \mathrm{path}(B)C$, where B is the parent of A, but then those new nonterminals could derive strings of length vastly exceeding x. We use a procedure which greedily partitions the trees into connected fragments called *chunks*. The procedure works as follows: if A is connected to its parent B with an edge labeled by C, check if $\mathrm{path}(B)C$ derives a string of length at most x. If so, A belongs to the same chunk as B, and we add a production $\mathrm{path}(A) \to \mathrm{path}(B)C$. Otherwise, create a new chunk, initially containing just A, which we call its *frontier*, and add a production $\mathrm{path}(A) \to C$, see Figure 4. The number of new nonterminals (and hence also productions) is clearly at most n. To describe the label of the path from A to the root, we concatenate all nonterminals $\mathrm{path}(B)$ where B is either A or a parent of a frontier on the path. As a result we get a sequence of nonterminals $Y_1 Y_2 \dots Y_\ell$ such that the length of the string described by any pair of neighbors $Y_i Y_{i+1}$ exceeds x. Hence after the final rewriting step the length of t will be at most $\mathcal{O}(\frac{N}{x})$. □

We apply the above lemma to the SLPs describing a and b to represent them as $a = X_1 \dots X_\ell$ and $b = X'_1 \dots X'_{\ell'}$. By cutting the dynamic programming table along the boundaries between any two X_i and X_{i+1} or X'_i and X'_{i+1}, we split it into $\mathcal{O}(\frac{N^2}{x^2})$ fragments of size at most $x \times x$. Moreover, each fragment corresponds to exactly one pair of nonterminals from a SLP of size $\mathcal{O}(n)$. We will compute the values on the boundaries of the fragments in two steps. First we build (all) matrices corresponding to pairs of nonterminals. Then we go through

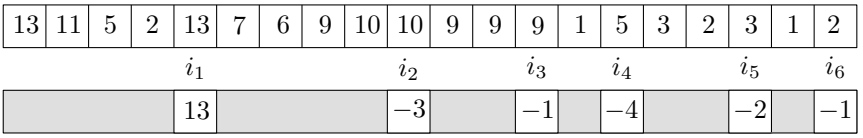

Fig. 5. Explicit (above) and implicit (below) description of the current t

the fragments one-by-one, and repeatedly multiply a vector describing values on the left and top boundary of the current block with the corresponding matrix, thus getting the values on the right and bottom boundary. We describe those two steps separately.

We compute the matrix corresponding to each pair of nonterminals in a bottom-up fashion. Assuming that we have the matrices corresponding to (A, D) and (B, D), we can compute the matrix corresponding to (C, D), where $C \to AB$, with a single max-product of two matrices in $\mathcal{O}(x \log x)$ time by Theorem 1. Hence the whole computation takes $\mathcal{O}(n^2 x \log x)$ time.

We compute the values on the boundaries fragment-by-fragment by constructing a new vector containing the values stored in the inputs of the current fragments, max-multiplying the vector by the corresponding H matrix, and thus computing the values which should be stored in the outputs. To multiply the vector and the matrix efficiently, we need the following lemma, which might be of independent interest.

Lemma 2. *Given a vector v of length x and an $x \times x$ matrix $H(i, j) = j - i - P^{\Sigma}(i, j)$, the max-product of v and H can be computed in $\mathcal{O}(x)$ time, assuming the matrix is given by the nonzeroes of P.*

Proof. We want to compute $u(j) = \max_i v(i) + H(i, j) = \max_i x(i) + j - i - P^{\Sigma}(i, j)$ for all j. Define $u'(j) = u(j) - j$ and $v'(i) = v(i) - i$, then $u'(j) = \max_i v'(i) - P^{\Sigma}(i, j)$. We will compute $u'(j)$ for $j = 1, 2, \ldots, x$ one-by-one.

For the current value of j we store an implicit description of all $t(i) = v'(i) - P^{\Sigma}(i, j)$. We start with $t(i) = v'(i)$. After increasing j by one we must decrease all $t(1), t(2), \ldots, t(k)$ by one, for some $k \in [1, x]$, and compute $\max_i t(i)$. Observe that if, at some point, $t(i) \leq t(i')$ for some $i < i'$, we can safely forget about i, as from this point on i' will always be a better (or equally good) choice than i. This motivates the following idea: we store a collection of *candidate indices* $i_1 < i_2 < \ldots < i_\ell$ such that $t(i_1) > t(i_2) > \ldots > t(i_\ell)$, chosen so that no matter what the future updates will be, the maximum value will be achieved on one of them. The initial choice is very simple, we take i_1 to be the rightmost maximum, i_2 the rightmost maximum on the remaining suffix, and so on. Such sequence of indices can be easily computed with a single sweep from right to left. We explicitly store $t(i_1)$ and, for each $t > 1$, $\delta(i_t) = t(i_t) - t(i_{t-1})$, see Figure 5.

To decrease $t(1), t(2), \ldots, t(k)$ we first locate the rightmost $i_t \leq k$ (if there is none, we terminate). Then we decrease $t(i_1)$ by one and increase $\delta(i_{t+1})$ by one (if $t = \ell$, we just decrease $t(i_1)$ and terminate). If as a result $\delta(i_{t+1})$ becomes zero, we consider two cases:

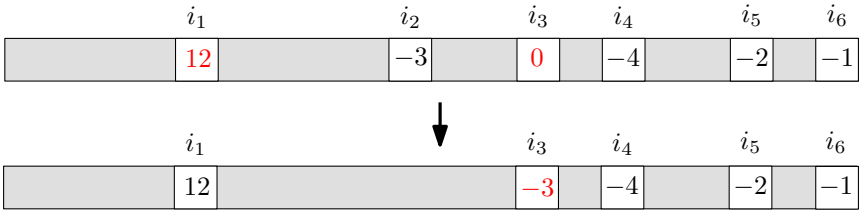

Fig. 6. Update with $t = 2$

1. $t = 1$, then we set $t(i_2) = t(i_1) + \delta(i_2)$ and remove i_1 from the list of candidate indices,
2. $t > 1$, then we set $\delta(i_{t+1}) = \delta(i_t)$ and remove i_t from the list of candidate indices.

See Figure 6 for an example of the second case.

The correctness of this procedure is immediate. Note that $\max_i t(i) = t(i_1)$, hence after each update we can compute the maximum in constant time. What is left is to show how quickly we can locate the rightmost $i_t \leq k$. We could simply store all candidate indices in a balanced search tree, and get $\mathcal{O}(\log x)$ update time. We can do better, though. Observe that what we really need is to store a partition of the whole $[1, x]$ into disjoint segments so that we can perform the following two operations efficiently:

1. locating the segment which a given k belongs to,
2. merging two adjacent segments.

A straightforward application of the standard union-find data structure allows us to achieve (amortized) $\mathcal{O}(\alpha(2x, x))$ complexity for both locating and merging. We can do even better, though. Notice that the segments are always contiguous, and we are actually dealing with an instance of the interval union-find problem. It is known that in this specific case, we can get (amortized) constant time per operation by splitting the whole universe into fragments of size $\Theta(\log x)$, storing a description of each such fragment in a single machine word (which assumes the RAM model), and building the usual union-find structure for the universe consisting of whole fragments [5]. Each description is constructed by simply marking the places a new segment starts at by setting the corresponding bit to 1. Then we can locate the fragment a given element belongs to and merge two segments in constant time using either bitwise operations, or by precomputing a few tables of size $o(x)$. In the latter case, the table contain, for each possible description, answer to any query, and the new description after each possible update. As each operation takes just constant time, we get the claimed total complexity. □

There are $\mathcal{O}(\frac{N^2}{x^2})$ blocks and for each of them we need $\mathcal{O}(x)$ time. Hence the total complexity is $\mathcal{O}(n^2 x \log x + \frac{N^2}{x})$. Let $f = \frac{N}{n}$ and set $x = \frac{f}{\sqrt{\log f}}$. Then the total time becomes $\mathcal{O}(n^2 f \sqrt{\log f} + \frac{N^2}{f}\sqrt{\log f}) = \mathcal{O}(nN\sqrt{\log \frac{N}{n}})$.

Theorem 2. *Edit distance between two strings of length N described by SLPs of total size n can be computed in $\mathcal{O}(nN\sqrt{\log \frac{N}{n}})$ time.*

Acknowledgments. The author would like to express his gratitude to Ela Babij, who explained the proof of Theorem 1 to him. He would also like to thank Alex Tiskin, who was kind enough to look at the proof of Lemma 2.

References

1. Andoni, A., Krauthgamer, R., Onak, K.: Polylogarithmic approximation for edit distance and the asymmetric query complexity. In: FOCS, pp. 377–386. IEEE Computer Society (2010)
2. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Transactions on Information Theory 51(7), 2554–2576 (2005)
3. Hermelin, D., Landau, G.M., Landau, S., Weimann, O.: A unified algorithm for accelerating edit-distance computation via text-compression. In: Albers, S., Marion, J.-Y. (eds.) STACS. LIPIcs, vol. 3, pp. 529–540. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2009)
4. Hermelin, D., Landau, G.M., Landau, S., Weimann, O.: Unified compression-based acceleration of edit-distance computation. CoRR, abs/1004.1194 (2010)
5. Itai, A.: Linear time restricted union/find (2006)
6. Landau, G.M., Vishkin, U.: Fast parallel and serial approximate string matching. J. Algorithms 10(2), 157–169 (1989)
7. Lifshits, Y.: Processing Compressed Texts: A Tractability Border. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 228–240. Springer, Heidelberg (2007)
8. Masek, W.J., Paterson, M.: A faster algorithm computing string edit distances. J. Comput. Syst. Sci. 20(1), 18–31 (1980)
9. Tiskin, A.: Semi-local string comparison: algorithmic techniques and applications. CoRR, abs/0707.3619 (2007)
10. Tiskin, A.: Fast distance multiplication of unit-Monge matrices. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, pp. 1287–1296. Society for Industrial and Applied Mathematics, Philadelphia (2010)