Linear-Space Computation of the Edit-Distance between a String and a Finite Automaton

Cyril Allauzen[1] and Mehryar Mohri[2,1]

[1] Google Research
76 Ninth Avenue, New York, NY 10011, US.
[2] Courant Institute of Mathematical Sciences
251 Mercer Street, New York, NY 10012, US.

Abstract. The problem of computing the edit-distance between a string and a finite automaton arises in a variety of applications in computational biology, text processing, and speech recognition. This paper presents linear-space algorithms for computing the edit-distance between a string and an arbitrary weighted automaton over the tropical semiring, or an unambiguous weighted automaton over an arbitrary semiring. It also gives an efficient linear-space algorithm for finding an optimal alignment of a string and such a weighted automaton.

1 Introduction

The problem of computing the edit-distance between a string and a finite automaton arises in a variety of applications in computational biology, text processing, and speech recognition [8, 10, 18, 21, 14]. This may be to compute the edit-distance between a protein sequence and a family of protein sequences compactly represented by a finite automaton [8, 10, 21], or to compute the error rate of a word lattice output by a speech recognition with respect to a reference transcription [14]. A word lattice is a weighted automaton, thus this further motivates the need for computing the edit-distance between a string and a weighted automaton. In all these cases, an optimal alignment is also typically sought. In computational biology, this may be to infer the function and various properties of the original protein sequence from the one it is best aligned with. In speech recognition, this determines the best transcription hypothesis contained in the lattice.

This paper presents linear-space algorithms for computing the edit-distance between a string and an arbitrary weighted automaton over the tropical semiring, or an unambiguous weighted automaton over an arbitrary semiring. It also gives an efficient linear-space algorithm for finding an optimal alignment of a string and such a weighted automaton. Our linear-space algorithms are obtained by using the same generic shortest-distance algorithm but by carefully defining different queue disciplines. More precisely, our meta-queue disciplines are derived in the same way from an underling queue discipline defined over states with the same level.

The connection between the edit-distance and the shortest distance in a directed graph was made very early on (see [10, 4–6] for a survey of string algorithms). This paper revisits some of these algorithms and shows that they are all special instances of the same generic shortest-distance algorithm using different queue disciplines. We also show that the linear-space algorithms all correspond to using the same meta-queue discipline using different underlying queues. Our approach thus provides a better understanding of these classical algorithms and makes it possible to easily generalize them, in particular to weighted automata.

The first algorithm to compute the edit-distance between a string x and a finite automaton A as well as their alignment was due to Wagner [25] (see also [26]). Its time complexity was in $O(|x||A|_Q^2)$ and its space complexity in $O(|A|_Q^2|\Sigma| + |x||A|_Q)$, where Σ denotes the alphabet and $|A|_Q$ the number of states of A. Sankoff and Kruskal [23] pointed out that the time and space complexity $O(|x||A|)$ can be achieved when the automaton A is acyclic. Myers and Miller [17] significantly improved on previous results. They showed that when A is acyclic or when it is a *Thompson automaton*, that is an automaton obtained from a regular expression using Thompson's construction [24], the edit-distance between x and A can be computed in $O(|x||A|)$ time and $O(|x|+|A|)$ space. They also showed, using a technique due to Hirschberg [11], that the optimal alignment between x and A can be obtained in $O(|x| + |A|)$ space, and in $O(|x||A|)$ time if A is acyclic, and in $O(|x||A| \log |x|)$ time when A is a Thompson automaton.

The remainder of the paper is organized as follows. Section 2 introduces the definition of semirings, and weighted automata and transducers. In Section 3, we give a formal definition of the edit-distance between a string and a finite automaton, or a weighted automaton. Section 4 presents our linear-space algorithms, including the proof of their space and time complexity and a discussion of an improvement of the time complexity for automata with some favorable graph structure property.

2 Preliminaries

This section gives the standard definition and specifies the notation used for weighted transducers and automata which we use in our computation of the edit-distance.

Finite-state transducers are finite automata [20] in which each transition is augmented with an output label in addition to the familiar input label [2, 9]. Output labels are concatenated along a path to form an output sequence and similarly input labels define an input sequence. *Weighted transducers* are finite-state transducers in which each transition carries some weight in addition to the input and output labels [22, 12]. Similarly, *weighted automata* are finite automata in which each transition carries some weight in addition to the input label. A path from an initial state to a final state is called an *accepting path*. A weighted transducer or weighted automaton is said to be *unambiguous* if it admits no two accepting paths with the same input sequence.

The weights are elements of a semiring $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$, that is a ring that may lack negation [12]. Some familiar semirings are the tropical semiring $(\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$ and the probability semiring $(\mathbb{R}_+ \cup \{\infty\}, +, \times, 0, 1)$, where $\mathbb{R}_+$ denotes the set of non-negative real numbers. In the following, we will only consider weighted automata and transducers over the tropical semiring. However, all the results of section 4.2 hold for an unambiguous weighted automaton A over an arbitrary semiring.

The following gives a formal definition of weighted transducers.

Definition 1. *A* weighted finite-state transducer T *over the tropical semiring* $(\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$ *is an 8-tuple* $T = (\Sigma, \Delta, Q, I, F, E, \lambda, \rho)$ *where* Σ *is the finite input alphabet of the transducer,* Δ *its finite output alphabet,* Q *is a finite set of states,* $I \subseteq Q$ *the set of initial states,* $F \subseteq Q$ *the set of final states,* $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Delta \cup \{\epsilon\}) \times (\mathbb{R}_+ \cup \{\infty\}) \times Q$ *a finite set of transitions,* $\lambda : I \to \mathbb{R}_+ \cup \{\infty\}$ *the initial weight function, and* $\rho : F \to \mathbb{R}_+ \cup \{\infty\}$ *the final weight function mapping* F *to* $\mathbb{R}_+ \cup \{\infty\}$.

We define the *size* of T as $|T| = |T|_Q + |T|_E$ where $|T|_Q = |Q|$ is the number of states and $|T|_E = |E|$ the number of transitions of T.

The weight of a path π in T is obtained by summing the weights of its constituent transitions and is denoted by $w[\pi]$. The weight of a pair of input and output strings (x, y) is obtained by taking the minimum of the weights of the paths labeled with (x, y) from an initial state to a final state.

For a path π, we denote by $p[\pi]$ its origin state and by $n[\pi]$ its destination state. We also denote by $P(I, x, y, F)$ the set of paths from the initial states I to the final states F labeled with input string x and output string y. The weight $T(x, y)$ associated by T to a pair of strings (x, y) is defined by:

$$T(x, y) = \min_{\pi \in P(I,x,y,F)} \lambda(p[\pi]) + w[\pi] + \rho[n[\pi]]. \tag{1}$$

Figure 1(a) shows an example of weighted transducer over the tropical semiring.

Weighted automata can be defined as weighted transducers A with identical input and output labels, for any transition. Thus, only pairs of the form (x, x) can have a non-zero weight by A, which is why the weight associated by A to (x, x) is abusively denoted by $A(x)$ and identified with the *weight associated by* A *to* x. Similarly, in the graph representation of weighted automata, the output (or input) label is omitted. Figure 1(b) shows an example.

3 Edit-distance

We first give the definition of the edit-distance between a string and a finite automaton.

Let Σ be a finite alphabet, and let Ω be defined by $\Omega = (\Sigma \cup \{\epsilon\}) \times (\Sigma \cup \{\epsilon\}) - \{(\epsilon, \epsilon)\}$. An element of Ω can be seen as a symbol edit operation: (a, ϵ) is a deletion, (ϵ, a) an insertion, and (a, b) with $a \neq b$ a substitution. We will denote by h the natural morphism between Ω^* and $\Sigma^* \times \Sigma^*$ defined by

Fig. 1. (a) Example of a weighted transducer T. (b) Example of a weighted automaton A. $T(aab, bba) = A(aab) = \min(.1+.2+.6+.8, .2+.4+.5+.8)$. A bold circle indicates an initial state and a double-circle a final state. The final weight $\rho[q]$ of a final state q is indicated after the slash symbol representing q.

$h((a_1, b_1) \cdots (a_n, b_n)) = (a_1 \cdots a_n, b_1 \cdots b_n)$. An *alignment* ω between two strings x and y is an element of Ω^* such that $h(\omega) = (x, y)$.

Let $c : \Omega \to \mathbb{R}_+$ be a function associating a non-negative cost to each edit operation. The cost of an alignment $\omega = \omega_1 \cdots \omega_n$ is defined as $c(\omega) = \sum_{i=1}^{n} c(\omega_i)$.

Definition 2. *The* edit-distance $d(x, y)$ *of two strings x and y is the minimal cost of a sequence of symbols insertions, deletions or substitutions transforming one string into the other:*

$$d(x, y) = \min_{h(\omega)=(x,y)} c(\omega). \tag{2}$$

When c is the function defined by $c(a, a) = 0$ and $c(a, \epsilon) = c(\epsilon, a) = c(a, b) = 1$ for all a, b in Σ such that $a \neq b$, the edit-distance is also known as the Levenshtein distance. The edit-distance $d(x, A)$ between a string x and a finite automaton A *can then be defined as*

$$d(x, A) = \min_{y \in L(A)} d(x, y), \tag{3}$$

where $L(A)$ denotes the regular language accepted by A. The edit-distance $d(x, A)$ between a string x and a weighted automaton A over the tropical semiring *is defined as:*

$$d(x, A) = \min_{y \in \Sigma^*} \big(A(y) + d(x, y)\big). \tag{4}$$

4 Algorithms

In this section, we present linear-space algorithms both for computing the edit-distance $d(x, A)$ between an arbitrary string x and an automaton A, and an optimal alignment between x and A, that is an alignment ω such that $c(\omega) = d(x, A)$.

We first briefly describe two general algorithms that we will use as subroutines.

4.1 General algorithms

Composition. The *composition* of two weighted transducers T_1 and T_2 over the tropical semiring with matching input and output alphabets Σ, is a weighted transducer denoted by $T_1 \circ T_2$ defined by:

$$(T_1 \circ T_2)(x, y) = \min_{z \in \Sigma^*} T_1(x, z) + T_2(z, y). \tag{5}$$

$T_1 \circ T_2$ can be computed from T_1 and T_2 using the composition algorithm for weighted transducers [19, 15]. States in the composition $T_1 \circ T_2$ are identified with pairs of a state of T_1 and a state of T_2. In the absence of transitions with ϵ inputs or outputs, the transitions of $T_1 \circ T_2$ are obtained as a result of the following matching operation applied to the transitions of T_1 and T_2:

$$(q_1, a, b, w_1, q_1') \text{ and } (q_2, b, c, w_2, q_2') \rightarrow ((q_1, q_1'), a, c, w_1 + w_2, (q_2, q_2')). \tag{6}$$

A state (q_1, q_2) of $T_1 \circ T_2$ is initial (resp. final) iff q_1 and q_2 are initial (resp. final) and, when it is final, its initial (resp.final) weight is the sum of the initial (resp. final) weights of q_1 and q_2. In the worst case, all transitions of T_1 leaving a state q_1 match all those of T_2 leaving state q_2, thus the space and time complexity of composition is quadratic, that is $O(|T_1||T_2|)$.

Shortest distance. Let A be a weighted automaton over the tropical semiring. The *shortest distance* from p to q is defined as

$$d[p, q] = \min_{\pi \in P(p,q)} w[\pi]. \tag{7}$$

It can be computed using the generic single-source shortest-distance algorithm of [13], a generalization of the classical shortest-distance algorithms. This generic shortest-distance algorithm works with an arbitrary *queue discipline*, that is the order according to which elements are extracted from a queue. We shall make use of this key property in our algorithms. The pseudocode of a simplified version of the generic algorithm for the tropical semiring is given in Figure 2.

The complexity of the algorithm depends on the queue discipline selected for S. Its general expression is

$$O(|Q| + \mathsf{C(A)} \max_{q \in Q} \mathsf{N(q)}|E| + (\mathsf{C(I)} + \mathsf{C(X)}) \sum_{q \in Q} \mathsf{N(q)}), \tag{8}$$

where $\mathsf{N(q)}$ denotes the number of times state q is extracted from queue S, $\mathsf{C(X)}$ the cost of extracting a state from S, $\mathsf{C(I)}$ the cost of inserting a state in S, and $\mathsf{C(A)}$ the cost of an assignment.

With a shortest-first queue discipline implemented using a heap, the algorithm coincides with Dijkstra's algorithm [7] and its complexity is $O((|E| + |Q|) \log |Q|)$. For an acyclic automaton and with the topological order queue discipline, the algorithm coincides with the standard linear-time ($O(|Q| + |E|)$) shortest-distance algorithm [3].

SHORTEST-DISTANCE(A, s)

```
1  for each p ∈ Q do
2      d[p] ← ∞
3  d[s] ← 0
4  S ← {s}
5  while S ≠ ∅ do
6      q ← HEAD(S)
7      DEQUEUE(S)
8      for each e ∈ E[q] do
9          if (d[s] + w[e] < d[n[e]]) then
10             d[n[e]] ← d[s] + w[e]
11             if (n[e] ∉ S) then
12                 ENQUEUE(S, n[e])
```

Fig. 2. Pseudocode of the generic shortest-distance algorithm.

4.2 Edit-distance algorithms

The edit cost function c can be naturally represented by a one-state weighted transducer over the tropical semiring $T_c = (\Sigma, \Sigma, \{0\}, \{0\}, \{0\}, E_c, \overline{1}, \overline{1})$, or T in the absence of ambiguity, with each transition corresponding to an edit operation: $E_c = \{(0, a, b, c(a, b), 0) | (a, b) \in \Omega\}$.

Lemma 1. *Let A be a weighted automaton over the tropical semiring and let X be the finite automaton representing a string x. Then, the edit-distance between x and A is the shortest-distance from the initial state to a final state in the weighted transducer $U = X \circ T \circ A$.*

Proof. Each transition e in T corresponds to an edit operation $(i[e], o[e]) \in \Omega$, and each path π corresponds to an alignment ω between $i[\pi]$ and $o[\pi]$. The cost of that alignment is, by definition of T, $c(\omega) = w[\pi]$. Thus, T defines the function:

$$T(u, v) = \min_{\omega \in \Omega^*} \{c(\omega) \colon h(\omega) = (u, v)\} = d(u, v), \tag{9}$$

for any strings u, v in Σ^*. Since A is an automaton and x is the only string accepted by X, it follows from the definition of composition that $U(x, y) = T(x, y) + A(y) = d(x, y) + A(y)$. The shortest-distance from the initial state to a final state in U is then:

$$\min_{\pi \in P_U(I,F)} w[\pi] = \min_{y \in \Sigma^*} \min_{\pi \in P_U(I,x,y,F)} w[\pi] = \min_{y \in \Sigma^*} U(x, y) \tag{10}$$

$$= \min_{y \in \Sigma^*} \big(d(x, y) + A(y)\big) = d(x, A), \tag{11}$$

that is the edit-distance between x and A. □

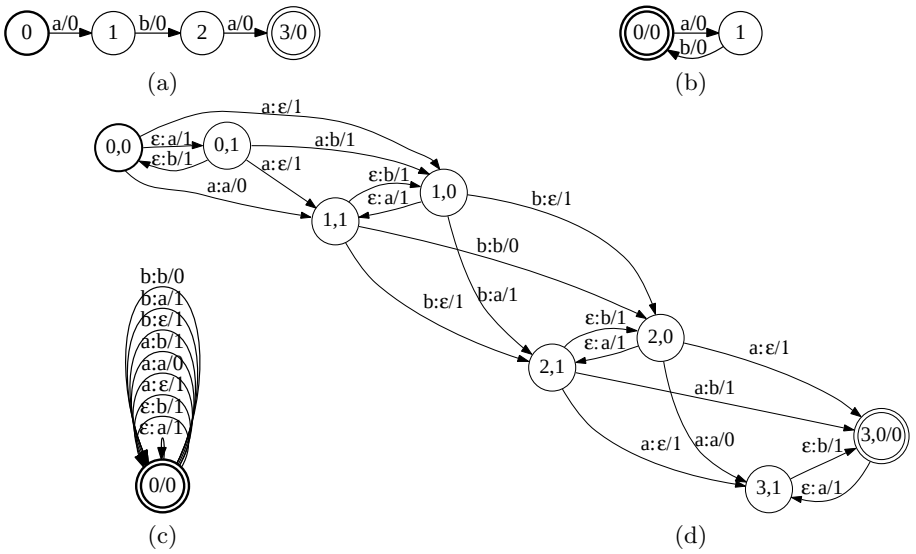

Fig. 3. (a) Finite automaton X representing the string $x = aba$. (b) Finite automaton A. (c) Edit transducer T over the alphabet $\{a, b\}$ where the cost of any insertion, deletion and substitution is 1. (d) Weighted transducer $U = X \circ T \circ A$.

Figure 3 shows an example illustrating Lemma 1. Using the lateral strategy of the 3-way composition algorithm of [1] or an *ad hoc* algorithm exploiting the structure of T, $U = X \circ T \circ A$ can be computed in $O(|x||A|)$ time. The shortest-distance algorithm presented in Section 4.1 can then be used to compute the shortest distance from an initial state of U to a final state and thus the edit distance of x and A. Let us point out that different queue disciplines in the computation of that shortest distance lead to different algorithms and complexities. In the next section, we shall give a queue discipline enabling us to achieve a linear-space complexity.

4.3 Edit-distance computation in linear space

Using the shortest-distance algorithm described in Section 4.1 leads to an algorithm with space complexity linear in the size of U, i.e. in $O(|x||A|)$. However, taking advantage of the topology of U, it is possible to design a queue discipline that leads to a linear space complexity $O(|x| + |A|)$.

We assume that the finite automaton X representing the string x is topologically sorted. A state q in the composition $U = X \circ T \circ A$ can be identified with a triplet $(i, 0, j)$ where i is a state of X, 0 the unique state of T, and j a state of A. Since T has a unique state, we further simplify the notation by identifying each state q with a pair (i, j). For a state $q = (i, j)$ of U, we will refer to i by the *level of* q. A key property of the levels is that there is a transition in U from q to q'

iff $\text{level}(q') = \text{level}(q)$ or $\text{level}(q') = \text{level}(q) + 1$. Indeed, a transition from (i, j) to (i', j') in U corresponds to taking a transition in X (in that case $i' = i + 1$ since X is topologically sorted) or staying at the same state in X and taking an input-ϵ transition in T (in that case $i' = i$).

From any queue discipline $\prec$ on the states of U, we can derive a new queue discipline $\prec_l$ over U defined for all q, q' in U as follows:

$$q \prec_l q' \text{ iff } \big(\text{level}(q) < \text{level}(q')\big) \text{ or } \big(\text{level}(q) = \text{level}(q') \text{ and } q \prec q'\big). \tag{12}$$

Proposition 1. *Let $\prec$ be a queue discipline that requires at most $O(|V|)$ space to maintain a queue over any set of states V. Then, the edit-distance between x and A can be computed in linear space, $O(|x| + |A|)$, using the queue discipline $\prec_l$.*

Proof. The benefit of the queue discipline $\prec_l$ is that when computing the shortest distance to $q = (i, j)$ in U, only the shortest distances to the states in U of level i and $i - 1$ need to be stored in memory. The shortest distances to the states of level strictly less than $i - 1$ can be safely discarded. Thus, the space required to store the shortest distances is in $O(|A|_Q)$.

Similarly, there is no need to store in memory the full transducer U. Instead, we can keep in memory the last two levels active in the shortest-distance algorithm. This is possible because the computation of the outgoing transitions of a state with level i only requires knowledge about the states with level i and $i + 1$. Therefore, the space used to store the active part of U is in $O(|A|_E + |A|_Q) = O(|A|)$. Thus, it follows that the space required to compute the edit-distance of x and A is linear, that is in $O(|x| + |A|)$. □

The time complexity of the algorithm depends on the underlying queue discipline $\prec$. A natural choice is for $\prec$ is the shortest-first queue discipline, that is the queue discipline used in Dijkstra's algorithm. This yields the following corollary.

Corollary 1. *The edit-distance between a string x and an automaton A can be computed in time $O(|x||A| \log |A|_Q)$ and space $O(|x| + |A|)$ using the queue discipline $\prec_l$.*

Proof. A shortest-first queue is maintained for each level and contains at most $|A|_Q$ states. The cost for the global queue of an insertion, $\mathsf{C}(\mathsf{I})$, or an assignment, $\mathsf{C}(\mathsf{A})$, is in $O(\log |A|_Q)$ since it corresponds to inserting in or updating one of the underlying level queues. Since $\mathsf{N}(\mathsf{q}) = 1$, the general expression of the complexity (8) leads to an overall time complexity of $O(|x||A| \log |A|_Q)$ for the shortest-distance algorithm. □

When the automaton A is acyclic, the time complexity can be further improved by using for $\prec$ the topological order queue discipline.

Corollary 2. *If the automaton A is acyclic, the edit-distance between x and A can be computed in time $O(|x||A|)$ and space $O(|x| + |A|)$ using the queue discipline $\prec_l$ with the topological order queue discipline for $\prec$.*

Proof. Computing the topological order for U would require $O(|U|)$ space. Instead, we use the topological order on A, which can be computed in $O(|A|)$, to define the underlying queue discipline. The order inferred by (12) is then a topological order on U. □

Myers and Miller [17] showed that when A is a Thompson automaton, the time complexity can be reduced to $O(|x||A|)$ even when A is not acyclic. This is possible because of the following observation: in a weighted automaton over the tropical semiring, there exists always a shortest path that is *simple*, that is with no cycle, since cycle weights cannot decrease path weight.

In general, it is not clear how to take advantage of this observation. However, a Thompson automaton has additionally the following structural property: a *loop-connectedness* of one. The *loop-connectedness* of A is k if in any depth-first search of A, a simple path goes through at most k back edges. [17] showed that this property, combined with the observation made previously, can be used to improve the time complexity of the algorithm. The results of [17] can be generalized as follows.

Corollary 3. *If the loop-connectedness of A is k, then the edit-distance between x and A can be computed in $O(|x||A|k)$ time and $O(|x| + |A|)$ space.*

Proof. We first use a depth-first search of A, identify back edges, and mark them as such. We then compute the topological order for A, ignoring these back edges. Our underlying queue discipline $\prec$ is defined such that a state $q = (i, j)$ is ordered first based on the number of times it has been enqueued and secondly based on the order of j in the topological order ignoring back edges. This underlying queue can be implemented in $O(|A|_Q)$ space with constant time costs for the insertion, extraction and updating operations. The order $\prec_l$ derived from $\prec$ is then not topological for a transition e iff e was obtained by matching a back edge in A and $\text{level}(p[e]) = \text{level}(n[e])$. When such a transition e is visited, $n[e]$ is reinserted in the queue.

When state q is dequeued for the lth time, the value of $d[q]$ is the weight of the shortest path from the initial state to q that goes through at most $l-1$ back edges. Thus, the inequality $\mathsf{N}(\mathsf{q}) \leq k+1$ holds for all q and, since the costs for managing the queue, $\mathsf{C}(\mathsf{I})$, $\mathsf{C}(\mathsf{A})$, and $\mathsf{C}(\mathsf{X})$, are constant, the time complexity of the algorithm is in $O(|x||A|k)$. □

4.4 Optimal alignment computation in linear space

The algorithm presented in the previous section can also be used to compute an optimal alignment by storing a back pointer at each state in U. However, this can increase the space complexity up to $O(|x||A|_Q)$. The use of back pointers to compute the best alignment can be avoided by using a technique due to Hirschberg [11], also used by [16, 17].

As pointed out in previous sections, an optimal alignment between x and A corresponds to a shortest path in $U = X \circ T \circ A$. We will say that a state q in U is a *midpoint* of an optimal alignment between x and A if q belongs to a shortest path in U and $\text{level}(q) = \lfloor |x|/2 \rfloor$.

Lemma 2. *Given a pair (x, A), a midpoint of the optimal alignment between x and A can be computed in $O(|x| + |A|)$ space with a time complexity in $O(|x||A|)$ if A is acyclic and in $O(|x||A| \log |A|_Q)$ otherwise.*

Proof. Let us consider $U = X \circ T \circ A$. For a state q in U let $d[q]$ denote the shortest distance from the initial state to q, and by $d^R[q]$ the shortest distance from q to a final state. For a given state $q = (i, j)$ in U, $d[(i, j)] + d^R[(i, j)]$ is the cost of the shortest path going through (i, j). Thus, for any i, the edit-distance between x and A is $d(x, A) = \min_j(d[(i, j)] + d^R[(i, j)])$.

For a fixed i_0, we can compute both $d[(i_0, j)]$ and $d^R[(i_0, j)]$ for all j in $O(|x||A| \log |A|_Q)$ time (or $O(|x||A|$ time if A is acyclic) and in linear space $O(|x| + |A|)$ using the algorithm from the previous section forward and backward and stopping at level i_0 in each case. Running the algorithm backward (exchanging initial and final states and permuting the origin and destination of every transition) can be seen as computing the edit-distance between x^R and A^R, the *mirror images* of x and A.

Let us now set $i_0 = \lfloor |x|/2 \rfloor$ and $j_0 = \operatorname{argmin}_j(d[(i_0, j)] + d^R[(i_0, j)])$. It then follows that (i_0, j_0) is a midpoint of the optimal alignment. Hence, for a pair (x, A), the running-time complexity of determining the midpoint of the alignment is in $O(|x||A|)$ if A is acyclic and $O(|x||A| \log |A|_Q)$ otherwise. □

The algorithm proceeds recursively by first determining the midpoint of the optimal alignment. At step 0 of the recursion, we first find the midpoint (i_0, j_0) between x and A. Let x^1 and x^2 be such that $x = x^1x^2$ and $|x^1| = i_0$, and let A^1 and A^2 be the automaton obtained from A by respectively changing the final state to j_0 in A^1 and the initial state to j_0 in A^2. We can now recursively find the alignment between x^1 and A^1 and between x^2 and A^2.

Theorem 1. *An optimal alignment between a string x and an automaton A can be computed in linear space $O(|x| + |A|)$ and in time $O(|x||A|)$ if A is acyclic, $O(|x||A| \log |x| \log |A|_Q)$ otherwise.*

Proof. We can assume without loss of generality that the length of x is a power of 2. At step k of the recursion, we need to compute the midpoints for 2^k string-automaton pairs $(x_k^i, A_k^i)_{1 \leq i \leq 2^k}$. Thus, the complexity of step k is in $O(\sum_{i=1}^{2^k} |x_k^i||A_k^i| \log |A_k^i|_Q) = O(\frac{|x|}{2^k} \sum_{i=1}^{2^k} |A_k^i| \log |A_k^i|_Q)$ since $|x_k^i| = |x|/2^k$ for all i. When A is acyclic, the log factor can be avoided and the equality $\sum_{i=1}^{2^k} |A_k^i| = O(|A|)$ holds, thus the time complexity of step k is in $O(|x||A|/2^k)$. In the general case, each $|A_k^i|$ can be in the order of $|A|$, thus the complexity of step k is in $O(|x||A| \log |A|_Q)$.

Since there are at most $\log |x|$ steps in the recursion, this leads to an overall time complexity in $O(|x||A|)$ if A is acyclic and $O(|x||A| \log |A|_Q \log |x|)$ in general. □

When the loop-connectedness of A is k, the time complexity can be improved to $O(k|x||A| \log |x|)$ in the general case.

5 Conclusion

We presented general algorithms for computing in linear space both the edit-distance between a string and a finite automaton and their optimal alignment. Our algorithms are conceptually simple and make use of existing generic algorithms. Our results further provide a better understanding of previous algorithms for more restricted automata by relating them to shortest-distance algorithms and general queue disciplines.

References

1. C. Allauzen and M. Mohri. 3-way composition of weighted finite-state transducers. In O. Ibarra and B. Ravikumar, editors, *Proceedings of CIAA 2008*, volume 5148 of *Lecture Notes in Computer Science*, pages 262–273. Springer-Verlag Berlin Heidelberg, 2008.
2. J. Berstel. *Transductions and Context-Free Languages*. Teubner Studienbucher: Stuttgart, 1979.
3. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press: Cambridge, MA, 1992.
4. M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.
5. M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.
6. M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific, 2002.
7. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
8. R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probalistic Models of Proteins and Nucleic Acids*. Cambridge University Press, Cambridge, UK, 1998.
9. S. Eilenberg. *Automata, Languages and Machines*, volume A–B. Academic Press, 1974–1976.
10. D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, Cambridge, UK, 1997.
11. D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, June 1975.
12. W. Kuich and A. Salomaa. *Semirings, Automata, Languages*. Number 5 in EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1986.
13. M. Mohri. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3):321–350, 2002.
14. M. Mohri. Edit-distance of weighted automata: General definitions and algorithms. *International Journal of Foundations of Computer Science*, 14(6):957–982, 2003.
15. M. Mohri, F. C. N. Pereira, and M. Riley. Weighted automata in text and speech processing. In *Proceedings of the 12th biennial European Conference on Artificial Intelligence (ECAI-96), Workshop on Extended finite state models of language, Budapest, Hungary*. John Wiley and Sons, Chichester, 1996.
16. E. W. Myers and W. Miller. Optimal alignments in linear space. *CABIOS*, 4(1):11–17, 1988.
17. E. W. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51(1):5–37, 1989.

18. G. Navarro and M. Raffinot. *Flexible pattern matching*. Cambridge University Press, 2002.
19. F. Pereira and M. Riley. *Finite State Language Processing*, chapter Speech Recognition by Composition of Weighted Finite Automata. The MIT Press, 1997.
20. D. Perrin. Finite automata. In J. V. Leuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 1–57. Elsevier, Amsterdam, 1990.
21. P. A. Pevzner. *Computational Molecular Biology: an Algorithmic Approach*. MIT Press, 2000.
22. A. Salomaa and M. Soittola. *Automata-Theoretic Aspects of Formal Power Series*. Springer-Verlag, 1978.
23. D. Sankoff and J. B. Kruskal. *Time Wraps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, Reading, MA, 1983.
24. K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):365–375, 1968.
25. R. A. Wagner. Order-n correction for regular languages. *Communications of the ACM*, 17(5):265–268, May 1974.
26. R. A. Wagner and J. I. Seiferas. Correcting counter-automaton-recognizable languages. *SIAM Journal on Computing*, 7(3):357–375, August 1978.