The Homo-Edit Distance Problem

Maren Brand[1]
Algorithmic Bioinformatics, Heinrich Heine University Düsseldorf, Germany
maren.brand@hhu.de

Nguyen Khoa Tran[1]
Algorithmic Bioinformatics, Heinrich Heine University Düsseldorf, Germany
nguyen.tran@hhu.de

Philipp Spohr
Algorithmic Bioinformatics, Heinrich Heine University Düsseldorf, Germany
philipp.spohr@hhu.de

Sven Schrinner
Algorithmic Bioinformatics, Heinrich Heine University Düsseldorf, Germany
sven.schrinner@hhu.de

Gunnar W. Klau[2]
Algorithmic Bioinformatics, Heinrich Heine University Düsseldorf, Germany
Cluster of Excellence on Plant Sciences (CEPLAS), Heinrich Heine University Düsseldorf, Germany
gunnar.klau@hhu.de

Abstract

We consider the homo-edit distance problem, which is the minimum number of homo-deletions or homo-insertions to convert one string into another. A homo-insertion is the insertion of a string of equal characters into another string, while a homo-deletion is the inverse operation. We show how to compute the homo-edit distance of two strings in polynomial time: We first demonstrate that the problem is equivalent to computing a common subsequence of the two input strings with a minimum number of homo-deletions and then present a dynamic programming solution for the reformulated problem.

2012 ACM Subject Classification Applied computing → Bioinformatics; Applied computing → Molecular sequence analysis; Theory of computation → Dynamic programming

Keywords and phrases sequence analysis, homo-edit distance, string distance, sequence alignment, common subsequence

Supplement Material Source code available at `https://github.com/AlBi-HHU/homo-edit-distance`.

Funding Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2048/1 – projectID 390686111.

Acknowledgements The authors thank Max Ried for supporting the release of the code as a Python package.

1 Introduction

A homo-insertion is an insertion of a string of equal characters, which we also call a *block*, into another string. A homo-deletion is the inverse operation, that is, the deletion of such a block. We consider the following problem: Given two strings, what is the minimum number of homo-insertions or homo-deletions needed to convert one into the other? We refer to this number as the homo-edit distance. This distance is a generalization of the edit distance

[1] Shared first authors.
[2] Corresponding author.

between two strings, where only insertions and deletions are possible, which is also known as the longest common subsequence distance [4, 1]. Unlike in the classic special case, where blocks consist only of single characters, two blocks may merge to one after a homo-deletion. For example, the homo-edit distance of `ATA` and the empty string is 2 and is achieved by first deleting `T` and then the block `AA`. This property makes the homo-edit distance more difficult to compute than the longest common subsequence distance.

We became aware of this problem as an exercise (6.40) in the classic textbook on bioinformatics algorithms by Jones and Pevzner [3]. As this exercise caused our students a lot of trouble we decided to look at it more closely within a thesis project [2]. We show how to compute the homo-edit distance of two strings in polynomial time: We first demonstrate that the problem is equivalent to computing a common subsequence of the two input strings with a minimum number of homo-deletions and then present a dynamic programming solution for the reformulated problem.

2 Problem Formulation

Let Σ be a finite alphabet. A string of length $n \in \mathbb{N}_0$ is defined as $s = s_1 s_2 \dots s_n \in \Sigma^n$. The empty string is denoted as ε. We also write the length of a string s as $|s|$. A *block* is a string consisting of identical characters, and we write a^k for a block of length k and some $a \in \Sigma$. We refer to substrings of s by

$$s(i,j) = \begin{cases} s_i s_{i+1} \dots s_j & \text{if } 1 \leq i \leq j \leq n, \\ \varepsilon & \text{otherwise.} \end{cases}$$

Subsequences $s_{k_1} s_{k_2} \dots s_{k_l}$ of s are characterized by their indices $k_1, k_2, \dots, k_l$, where $1 \leq k_1 < k_2 < \dots < k_l \leq n$.

We define two string operations which we subsume as *homo-operations*: The first operation inserts a block of length k into a string at a certain position. Let $a \in \Sigma$ and let $u = a^k$ be the block that is to be inserted into string s at position i, where $1 \leq i \leq n+1$. We define this *homo-insertion* as the string

$$I_{i,u}(s) = s_1 s_2 \dots s_{i-1} u_1 \dots u_k s_i \dots s_n.$$

The second operation deletes a block $s(i,j) = a \dots a$ with $a \in \Sigma$ and $1 \leq i \leq j \leq n$. We define this *homo-deletion* as the string

$$D_{i,j}(s) = s_1 s_2 \dots s_{i-1} s_{j+1} \dots s_n.$$

Note that both operations are reversible, that is, for each homo-insertion there is a homo-deletion that can be applied to obtain the original string, and vice versa. For an operation O we denote the corresponding reverse operation by $\overline{O}$. Reversibility also holds for chains of operations as the following lemma shows.

▶ **Lemma 1.** *Consider a series of homo-operations $O_1, O_2, \dots, O_k$ to convert a string s into another string t, that is, $t = (O_k \circ O_{k-1} \circ \dots \circ O_1)(s)$. Then, also $s = (\overline{O_1} \circ \overline{O_2} \circ \dots \circ \overline{O_k})(t)$ holds.*

Proof by induction.

- Base case: $k = 1$
  Case 1: $t = O_1(s) = I_{i,u}(s)$ is obtained by a homo-insertion of a string $u = a^{j-i+1}$ into s at position i, where $a \in \Sigma$ and $j \geq i$. We reverse this homo-insertion by using a homo-deletion of substring $s(i,j) = u$ from t, i.e., $D_{i,j}(t) = \overline{O_1}(t) = \overline{O_1}(O_1(s)) = s$.
  Case 2: $t = O_1(s) = D_{i,j}(s)$ is obtained by a homo-deletion of a substring $u = s(i,j)$ from s. We reverse this homo-deletion by using a homo-insertion of u into t at position i, i.e., $I_{i,u}(t) = \overline{O_1}(t) = \overline{O_1}(O_1(s)) = s$.
- Induction step: $k \to k+1$
  Let $s' = (O_k \circ O_{k-1} \circ \ldots \circ O_1)(s)$ such that $O_{k+1}(s') = t$.
  Case 1: $t = O_{k+1}(s') = I_{i,u}(s')$ is obtained by a homo-insertion of a string $u = a^{j-i}$ into s' at position i, where $a \in \Sigma$ and $j \geq i$. We reverse this homo-insertion by using a homo-deletion of substring $s'(i,j) = u$ from t, i.e., $D_{i,j}(t) = \overline{O_{k+1}}(t) = \overline{O_{k+1}}(O_{k+1}(s')) = s'$.
  Case 2: $t = O_{k+1}(s) = D_{i,j}(s')$ is obtained by a homo-deletion of a substring $u = s'(i,j)$ from s'. We reverse this homo-deletion by using a homo-insertion of u into t at position i, i.e., $I_{i,u}(t) = \overline{O_{k+1}}(t) = \overline{O_{k+1}}(O_{k+1}(s')) = s'$. ◀

We define the *homo-edit distance* $H(s,t)$ between two strings s and t as the minimum number of homo-operations to convert s into t. From Lemma 1 it follows that the homo-edit distance is symmetric, that is, $H(s,t) = H(t,s)$. We can now define the homo-edit distance problem formally as follows:

▶ **Problem 1** (Homo-Edit Distance Problem)**.** Given two strings s and t, compute their homo-edit distance $H(s,t)$.

3 Problem Reformulation

In this section we point out that the homo-edit distance between two strings s and t can be computed by considering homo-deletions only. For this we show that there exists a common subsequence v of both strings such that converting both s and t into v needs a total of $H(s,t)$ homo-deletions.

▶ **Lemma 2.** *Let s and t be two strings and let $H(s,t) = k$. Then there exists an optimal series of homo-operations $O_1, O_2, \ldots, O_k$ to convert s into t, such that the first part of the series contains only homo-deletions and the second part only homo-insertions.*

Proof. Let $O'_1, O'_2, \ldots, O'_k$ be any optimal series of homo-operations without the property stated in Lemma 2, let O'_i be a homo-insertion followed by a homo-deletion O'_{i+1}, where $1 \leq i \leq k-1$, and let $s[i] = (O'_{i-1} \circ O'_{i-2} \circ \ldots \circ O'_1)(s)$. If there exists a homo-deletion O''_i followed by a homo-insertion O''_{i+1} such that $(O'_{i+1} \circ O'_i)(s[i]) = (O''_i \circ O''_{i+1})(s[i])$, then the series $O_1, O_2, \ldots, O_k$ must exist as well, because we can repeatedly replace each homo-insertion followed by a homo-deletion with a homo-deletion followed by a homo-insertion, resulting in the same string.

Let u be the string that we want to insert by applying O'_i and let w be the string that we want to delete by applying O'_{i+1}. We consider two cases:

- Case 1: w consists of a substring of $s[i]$ only.
  Let p_1 be the position in $s[i]$ where we want to insert u, and let p_2 be the position of w_1 in $s[i]$. We can safely delete w first and then insert u either at position p_1, if $p_1 \leq p_2$, or at position $p_1 - |w|$, if $p_1 > p_2$.

- Case 2: w consists of a substring of u as well as a substring of $s[i]$.
  Let $a \in \Sigma$, let $u = a^{c_1}$, and let $w = a^{c_2}$, such that after applying both homo-operations, we either inserted or deleted $a^{c_1 - c_2}$, depending on whether $c_1 > c_2$ or $c_1 \leq c_2$. This means we could use one instead of two homo-operations for inserting or deleting $a^{c_1 - c_2}$, or even zero if $c_1 = c_2$. Thus, the series $O'_1, O'_2, \ldots, O'_k$ would not be optimal, which is a contradiction. ◀

▶ **Lemma 3.** *Let s and t be two strings. Then $H(s,t) = k$ if and only if there is a common subsequence v of s and t, such that it takes a total of k homo-deletions to convert s and t into v.*

Proof. From Lemma 2 we know that for converting s into t, there exists a series of homo-operations $O_1, O_2, \ldots O_k$ where the first i homo-operations of this series include homo-deletions only and where the last $k - i$ homo-operations include homo-insertions only, where $0 \leq i \leq k$. Let v be the string that we obtain by performing these homo-deletions on s, that is, $v = (O_i \circ O_{i-1} \circ \ldots \circ O_1)(s)$. Then v is a subsequence of s by definition. From Lemma 1 we know that we can reverse the homo-insertions of the series $t = (O_k \circ O_{k-1} \circ \ldots \circ O_{i+1})(v)$ such that $v = (\overline{O_{i+1}} \circ \overline{O_{i+2}} \circ \ldots \circ \overline{O_k})(t)$. Thus, v is also a subsequence of t, and we can obtain v by a total of k homo-deletions. ◀

Lemma 3 implies that we can safely disregard homo-insertions for computing homo-edit distances. In the next section we present an algorithm that computes the homo-edit distance of two strings by finding the minimum number of homo-deletions to convert both into a common subsequence.

4 Dynamic Programming Algorithm

This section contains our algorithmic contributions to the problem, their correctness proofs, a note on backtracking and a running time analysis.

4.1 Algorithms

We compute the homo-edit distance between two strings s and t with a two-part dynamic programming (DP) algorithm: The first part is a precomputation step that computes and stores the homo-edit distance between every substring of both s and t and the empty string ε. The second part is the main algorithm that, similar to classic textbook approaches for sequence alignment, computes a DP matrix containing the homo-edit distances between all prefixes of s and t. For better understanding we explain the main algorithm first.

Given two strings s and t, let v be an optimal common subsequence, that is, v satisfies the conditions of Lemma 3. Let $m = |s|$ and $n = |t|$. We compute an $(m+1) \times (n+1)$ matrix d, where each entry $d_{i,j}$ corresponds to the homo-edit distance between the prefixes $s(1,i)$ and $t(1,j)$, with the following recurrence:

$$d_{0,0} = 0$$

$$d_{i,j} = \min \left\{ \begin{array}{ll} d_{i-1,j-1} & \text{if } s_i = t_j, \\ \min\limits_{0 \leq k < i} \{d_{k,j} + H(s(k+1,i),\varepsilon)\}, & \\ \min\limits_{0 \leq l < j} \{d_{i,l} + H(t(l+1,j),\varepsilon)\} & \end{array} \right\} \tag{1}$$

We start by initializing $d_{0,0}$ with 0. For all other entries we proceed, e.g., from top to bottom ($i = 0, 1, \ldots, m$) and from left to right ($j = 0, 1, \ldots, n$), and consider three cases for the homo-edit distance between $s(1,i)$ and $t(1,j)$, among which we pick the minimum:

1. The first case is given if we have a match, i.e., $s_i = t_j$. In this case, the common character could be part of an optimal common subsequence v. As we would neither delete s_i nor t_j by a homo-deletion, we have $d_{i,j} = d_{i-1,j-1}$.
2. The next case comprises all possibilities that involve deleting s_i from s, meaning that this character would not be part of an optimal subsequence v. More precisely, for $d_{i,j}$ we consider each entry $d_{k,j}$ of the same column j in a row k from above plus the cost of deleting $s(k+1, i)$. We will show how to compute the homo-edit distances between all substrings of a string and the empty string ε later.
3. The last case consists of all possibilities where we delete t_j from t. More precisely, for $d_{i,j}$ we consider each entry $d_{i,l}$ of the same row i in a column l from left plus the cost of deleting $t(l+1, j)$.

Eventually, $d_{m,n}$ contains the homo-edit distance between s and t. We can obtain an optimal subsequence v and thereby an optimal series of operations to obtain s from t or vice versa by backtracking the cases from $d_{m,n}$ to $d_{0,0}$. Note that there can be multiple possibilities for v. See Algorithm 1 and the paragraph about backtracking below for more details.

Algorithm 1 Main dynamic programming algorithm to compute the homo-edit distance between two strings s and t.

```
1:  function INT HOMOEDITDISTANCE($s, t$)
2:      let $H$ be a dictionary, which holds all entries for DISTANCESTOEMPTYSTRING of $s$
    and $t$, with substrings as keys and the corresponding homo-edit distances to $\varepsilon$ as values
3:      $m \leftarrow |s|$
4:      $n \leftarrow |t|$
5:      initialize $d$ as an $(m+1) \times (n+1)$ matrix
6:      for $i \leftarrow 0, 1, \ldots, m$ do
7:          for $j \leftarrow 0, 1, \ldots, n$ do
8:              if $i = j = 0$ then
9:                  $d_{i,j} \leftarrow 0$
10:                 continue
11:             $C \leftarrow \{\}$                                    ▷ candidate values
12:             if $s_i = t_j$ then
13:                 append $d_{i-1,j-1}$ to $C$
14:             for $k \leftarrow 0, 1, \ldots, i-1$ do
15:                 append $d_{k,j} + H[s(k+1, i)]$ to $C$
16:             for $l \leftarrow 0, 1, \ldots, j-1$ do
17:                 append $d_{i,l} + H[t(l+1, j)]$ to $C$
18:             $d_{i,j} \leftarrow \min(C)$
19:     return $d_{m,n}$
```

We now show how to precompute the homo-edit distances $H(s(i,j), \varepsilon)$ between all substrings of a string s of length n and the empty string. Again, we use dynamic programming, filling an $n \times n$ matrix $h(s)$, with the following recurrence:

$$h_{i,j}(s) = \begin{cases} 1 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{h_{i,k}(s) + h_{k+1,j}(s) - [s_i = s_j]\} & \text{otherwise.} \end{cases} \tag{2}$$

We start by initializing all homo-edit distances between ε and every substring $s(i,i)$ of length one to $h_{i,i}(s) = H(s(i,i),\varepsilon) = 1$ for all $1 \leq i \leq n$. Then we loop over all substrings of length two and compute their homo-edit distances to ε, and repeat the same procedure for all substrings of increasing length up to length n: To compute $h_{i,j}(s)$, we partition substring $s(i,j)$ into all possible pairs of shorter substrings $s(i,k)$ and $s(k+1,j)$, where $i \leq k < j$. For each partition we compute the cost to delete it, and choose the minimum of these costs. If $s_i \neq s_j$ the cost of deleting a partition is the sum of the costs to delete either substring. If, however, $s_i = s_j$ the cost decreases by one, which we notate using the Iverson bracket. The reason is that all partitions delete s_i in $s(i,k)$ and s_j in $s(k+1,j)$ separately by two homo-deletions, but it is always possible to delete the characters at the first and last index together with one homo-deletion. In the end, $h_{i,j}(s) = H(s(i,j),\varepsilon)$ for all $1 \leq i < j \leq n$. See Algorithm 2 and the correctness proof below for more details.

Algorithm 2 Auxiliary dynamic programming algorithm to compute the homo-edit distance between every substring of a string s and the empty string.

```
1:  function DICTIONARY DISTANCESTOEMPTYSTRING($s$)
2:      $n \leftarrow |s|$
3:      initialize $H$ as an empty DICTIONARY
4:      for $l \leftarrow 0, 1, \ldots, n-1$ do
5:          for $i \leftarrow 1, 2, \ldots, n-l$ do
6:              $j \leftarrow i + l$
7:              if $i = j$ then
8:                  $H[s(i,j)] \leftarrow 1$
9:                  continue
10:             $C \leftarrow \{\}$                                          ▷ candidate values
11:             for $k \leftarrow i, i+1, \ldots, j-1$ do
12:                 append $H[s(i,k)] + H[s(k+1,j)] - [s_i = s_j]$ to $C$
13:             $H[s(i,j)] \leftarrow \min(C)$
14:     return $H$
```

The example in Fig. 1 illustrates how the algorithms compute the homo-edit distance for the input strings $s =$ `CTGCA` and $t =$ `AGAAC`.

4.2 Correctness

▶ **Lemma 4.** *Given a string $s = s_1 s_2 \ldots s_n$, Recurrence* (2) *computes the homo-edit distance between every substring of s and the empty string ε.*

Proof by induction.

- *Base case:* $n = 1$.
  We need exactly one homo-deletion for one character, thus we have a homo-edit distance of 1, which is consistent with Recurrence (2).
- *Induction step:* $n \to n + 1$.
  We consider two cases:
  - Case 1: There exists an index k where $1 \leq k < n+1$ such that $s(1,k)$ and $s(k+1, n+1)$ can be deleted independently from one another, i.e., we do not perform a homo-deletion that involves both substrings at once. This means the induction holds since we can reduce this problem to two subproblems.

(a) $s =$ `CTGCA`

j	substrings	deletion costs
1	`C, T, G, C, A`	1, 1, 1, 1, 1
2	`CT, TG, GC, CA`	2, 2, 2, 2
3	`CTG, TGC, GCA`	3, 3, 3
4	`CTGC, TGCA`	3, 4
5	`CTGCA`	4

$t =$ `AGAAG`

j	substrings	deletion costs
1	`A, G, A, A, G`	1, 1, 1, 1, 1
2	`AG, GA, AA, AG`	2, 2, 1, 2
3	`AGA, GAA, AAG`	2, 2, 2
4	`AGAA, GAAG`	2, 2
5	`AGAAG`	3

(b)

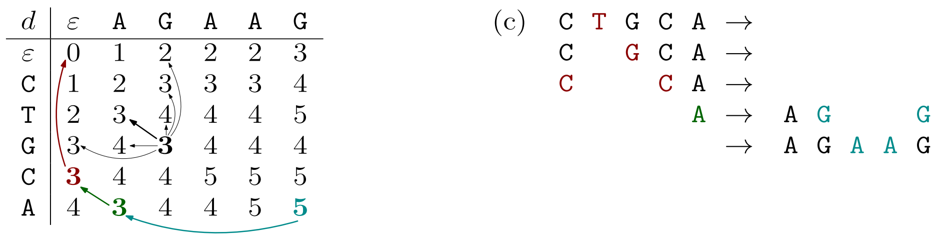

Figure 1 Example illustrating the algorithm. (a) Input strings s and t and the costs computed by the auxiliary DP. (b) Main DP matrix. An optimal backtracking path is colorized. Black arrows indicate how the entry for prefixes `CTG` and `AG` (shown in bold face) is computed. Here, the minimum is determined by the first case of Recurrence (1) and indicated by a bold arrow. (c) A transition in five steps as given by backtracking the colored path in (b).

- Case 2: There exists no such index k. Then s_1 and s_{n+1} are the same character and must be deleted together because otherwise Case 1 would apply. That is, the cost for deleting $s(1, n+1)$ are the same as for $s(1, n)$ because we can always delete s_1 and s_{n+1} (and perhaps other equal characters in between) together with the last homo-deletion before reaching ε. ◀

▶ **Lemma 5.** *Given two strings $s = s_1 s_2 \dots s_n$ and $t = t_1 t_2 \dots t_m$, Recurrence* (1) *computes and stores the homo-edit distance between s and t in $d_{m,n}$.*

Proof. The *edit distance problem* is to convert a string into another such that the sum of individual costs of the editing operations insertion, deletion, and substitution is minimized, where the mentioned editing operations can operate on exactly one character. Ukkonen [6] describes a generalization of this problem: Given two strings $s = s_1 s_2 \dots s_n$ and $t = t_1 t_2 \dots t_m$, we want to convert s into t such that the sum of individual costs of editing operations is minimized.

We can show that a problem is also a generalized edit distance problem by giving an *editing operation set* $E \subset \Sigma^* \times \Sigma^*$, where an element (x, y), $x \neq y$, represents an editing operation that replaces x with y, and a *recurrence* that defines a matrix d with *cost function* $\delta : E \to \mathbb{N}$ as follows:

$$d_{0,0} = 0,$$

$$d_{i,j} = \min \left\{ \begin{array}{ll} d_{i-1,j-1} & \text{if } s_i = t_j, \\ d_{i-k,j-r} + \delta((s_{i-k+1} \dots s_i, t_{j-r+1} \dots t_j)) & \text{if } (s_{i-k+1} \dots s_i, t_{j-r+1} \dots t_j) \in E \end{array} \right\}.$$

(Note that we rewrote Ukkonen's recurrence to fit our notations.) Hence, if the homo-edit distance problem is a generalized edit distance problem, Recurrence (1) works correctly.

For the homo-edit distance problem we can represent the editing operation set as

$$\begin{aligned} E =& \{(s(i-k+1,i), t(j+1,j)) \mid 1 \le i \le n \text{ and } 0 \le k \le i \text{ and } 1 \le j \le m\} \\ & \cup \{(s(i+1,i), t(j-r+1,j)) \mid 1 \le j \le m \text{ and } 0 \le r \le j \text{ and } 1 \le i \le n\}. \end{aligned}$$

The cost function can be defined as

$$\delta(s,t) = \begin{cases} H(s,\varepsilon) & \text{if } t = \varepsilon, \\ H(t,\varepsilon) & \text{if } s = \varepsilon. \end{cases}$$

As a result, Algorithm 1 works correctly as we can rewrite Recurrence (1) as Ukkonen's recurrence. ◀

4.3 Backtracking

From Lemma 1 and Lemma 3 we can deduce that an optimal series of operations needed for transforming s into t can be inferred from an optimal series of homo-deletions needed to transform both s and t into a common subsequence v with the property described in Lemma 3. Therefore, we disregard homo-insertions. Besides, we focus on backtracking one optimal series of homo-deletions that transform each input string into v. Note, however, that there might be multiple possible optimal series and subsequences.

In order to backtrack and thus generate an optimal series of homo-deletions as well as v, we augment our matrices d and h as follows: For each entry $d_{i,j}$, we additionally store the indices of any entry $d_{i',j'}$ from which we came from. For each entry $h_{i,j}(s)$, we additionally store the smallest index k that led to $h_{i,j}(s)$. We proceed analogously for $h(t)$.

Next, we backtrack a path from $d_{m,n}$ to $d_{0,0}$. Let d_{i_1,i_2} be the entry from where we obtained our current entry d_{j_1,j_2}. Let v' be an empty string ε that will eventually hold our desired v, and let L_s and L_t be initially empty lists in which we will store our indices denoting an optimal series of homo-deletions from s or t, respectively. Note that homo-deletions cause indices to shift such that the indices stored in L_s and/or L_t might need to be adjusted accordingly. We consider three cases:

1. If we obtained d_{j_1,j_2} from a match, we prepend s_{j_1} to v.
2. If we obtained d_{j_1,j_2} from an above entry that deletes $s(i_1, j_1)$ from s, we recursively split the deletion of $s(i_1, j_1)$ into the deletion of the two substrings $s(i_1, k)$ and $s(k+1, j_1)$, where k is the respective index obtained from backtracking $h(s)$. We abuse notation by using the same notation for any lower level of the recursion. The recursion adds a tuple (k, k) (or $(k+1, k+1)$) to L_s if a substring $s(k,k)$ (or $s(k+1,k+1)$) consists of one character only. Every time we move up one recursion level, we check whether the outer characters s_{i_1} and s_{j_1} are equal. If so, from L_s we remove the tuple that is returned first by $s(i_1, k)$, which contains i_1, as well as the tuple that is returned last by $s(k+1, j_1)$, which contains j_1. We then append (i_1, j_1) to L_s.
3. If we obtained d_{j_1,j_2} from a left entry that deletes $t(i_2, j_2)$ from t, we proceed analogously to the second case.

4.4 Running Time Analysis

Consider $s = s_1 s_2 \dots s_n$ and $t = t_1 t_2 \dots t_m$. For string s, Algorithm 2 considers $\frac{n(n+1)}{2}$ different substrings of s. For each of those substrings, we have up to $n-1$ different partitions into two substrings. Consequently, the running time is in $\mathcal{O}((n-1)\frac{n(n+1)}{2}) = \mathcal{O}(n^3)$. Analogously, we get $\mathcal{O}(m^3)$ to preprocess string t.

For each entry in d, Algorithm 1 regards up to m options from above, up to n options from left, and possibly one option from top-left. As we have $m \cdot n$ entries, the running time is in $\mathcal{O}(m \cdot n \cdot (m + n + 1)) = \mathcal{O}(\max\{n, m\}^3)$.

All in all, the running time is $\mathcal{O}(n^3) + \mathcal{O}(m^3) + \mathcal{O}(\max\{n, m\}^3) = \mathcal{O}(\max\{n, m\}^3)$ and thus cubic in the input length.

We can now state our main result:

▶ **Theorem 6.** *Algorithm 1 computes the homo-edit distance of two strings $s = s_1 s_2 \dots s_n$ and $t = t_1 t_2 \dots t_m$ in time $\mathcal{O}(\max\{n, m\}^3)$.*

Proof. Follows from Lemmas 4 and 5 and the above running time analysis. ◀

5 Conclusions

The focus of this paper is to introduce the homo-edit distance problem and to present a solution to compute this distance in polynomial time. We have not yet considered applications of this distance to specific problems in bioinformatics and leave this as future work.

We can, for example, imagine applications to sequence analysis problems that involve tandem repeats, in a similar way as done by Sammeth and Stoye [5] who analyzed coding regions of the *Staphylococcus aureus* protein A gene (spa). *S. aureus* is a major human pathogen, and the analysis of relations between antibiotics-resistant strains can have important implications for clinical practice. Here, the homo-edit distance could be a good starting point for an all-against-all comparison of the spa-regions of different strains with an alphabet given by the repeats or higher order repeat structures.

Another possible application is the analysis of homopolymer-rich DNA-regions. Basecalling in these regions is particularly difficult for pyro- and ion torrent-based sequencing technologies, where over- and undercalling are common errors in these regions. The challenge is to distinguish these sequencing artifacts from true genetic content where a homo-edit distance-based analysis of the reads falling in such regions may provide some help.

In general, we envision also more theoretical work on extensions of the homo-edit distance. For which combinations of additional biologically meaningful operations like, e.g., duplications or mutations, can the distance still be computed in polynomial time and which versions become intractable? These and related open questions provide challenging opportunities for the theoretical bioinformatics community.

References

1 A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2(1):315–336, 1987.
2 M. Brand. Das Homo-Edit-Distanz-Problem. Bachelor's thesis, Heinrich Heine University Düsseldorf, 2020.
3 N. Jones and P. Pevzner. *An Introduction to Bioinformatics Algorithms*. MIT Press, Cambridge, MA, 2004.
4 S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.
5 M. Sammeth and J. Stoye. Comparing tandem repeats with duplications and excisions of variable degree. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 3(4):395–407, 2006.
6 E. Ukkonen. Algorithms for approximate string matching. *Information and control*, 64(1-3):100–118, 1985.