Faster Algorithms for Computing the Hairpin Completion Distance and Minimum Ancestor

Itai Boneh
Reichman University, Herzliya, Israel

Dvir Fried
Bar-Ilan University, Ramat-Gan, Israel

Adrian Miclăuş
Faculty of Mathematics and Computer Science, University of Bucharest, Romania

Alexandru Popa
Faculty of Mathematics and Computer Science, University of Bucharest, Romania

Abstract

Hairpin completion is an operation on formal languages that has been inspired by hairpin formation in DNA biochemistry and has many applications especially in DNA computing. Consider s to be a string over the alphabet $\{A, C, G, T\}$ such that a prefix/suffix of it matches the reversed complement of a substring of s. Then, in a hairpin completion operation the reversed complement of this prefix/suffix is added to the start/end of s forming a new string.

In this paper we study two problems related to the hairpin completion. The first problem asks the minimum number of hairpin operations necessary to transform one string into another, number that is called *the hairpin completion distance*. For this problem we show an algorithm of running time $O(n^2)$, where n is the maximum length of the two strings. Our algorithm improves on the algorithm of Manea (TCS 2010), that has running time $O(n^2 \log n)$.

In *the minimum distance common hairpin completion ancestor* problem we want to find, for two input strings x and y, a string w that minimizes the sum of the hairpin completion distances to x and y. Similarly, we present an algorithm with running time $O(n^2)$ that improves by a $O(\log n)$ factor the algorithm of Manea (TCS 2010).

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases dynamic programming, incremental trees, exact algorithm

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.5

Funding This work was supported by a grant of the Ministry of Research, Innovation and Digitization, CNCS - UEFISCDI, project number PN-III-P1-1.1-TE-2021-0253, within PNCDI III.
Itai Boneh: Israel Science Foundation grant 810/21.
Dvir Fried: Supported in part by ISF grant no. 1926/19, by a BSF grant no. 2018364, and by an ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).

1 Introduction

1.1 Motivation and informal problem definition

Hairpin completion is an operation on formal languages that has been inspired by hairpin formation in DNA biochemistry and has many applications especially in DNA computing [11, 12, 14, 15]. This operation has been inspired by three biological principles: Watson-Crick complementarity, DNA annealing and DNA lengthening through polymerases. The DNA chain is a molecule consisting of two intertwined strands, each strand being composed by nucleotides: A(Adenine), C(cytosine), G(guanine) and T(thymine). The two strands which form the DNA molecule are kept together by the hydrogen bond between the bases: A bonds with T and C with G. This paradigm is usually referred to as the Watson-Crick complementarity [25].

Another important bio-chemical principle is annealing, the process of fusing two single stranded molecules by complementary base. DNA lengthening through polymerases is a phenomenon that produces a complete double stranded DNA molecule as follows: one starts with two single strands such that one (called primer) is bonded to a part of the other (called template) through Watson-Crick complementarity and a polymerization buffer with many copies of the four nucleotides. The polymerases will then concatenate to the primer by complementing the template [22].

We now begin to informally explain the hairpin completion operation and how it can be related to the biological concepts presented above. Consider s to be a string over the alphabet $\{A, C, G, T\}$ such that a prefix/suffix of it matches to the reversed complement of a substring of s. Then, the reversed complement of this prefix/suffix is added to the beginning/ending of s forming a new string as can be visualized in Figure 1. The mathematical expression of this hypothetical situation defines the hairpin completion operation. Starting with a single string, one can generate a set of strings using this formal operation: via hairpin completion, a new string can be created for each possible pairing between a prefix or suffix and a complementary substring. In addition, one could be interested in knowing how many iterations of hairpin completion are required to transform one string into another. In this way, the hairpin completion distance between two strings was defined as the minimum number of times we must iterate the hairpin completion operation, starting from one of the two string, in order to obtain the other. Further, one can also be interested in finding for two strings, a common ancestor that minimizes the sum of the hairpin completion distances to those strings. This ancestor is called minimum distance common hairpin completion ancestor.

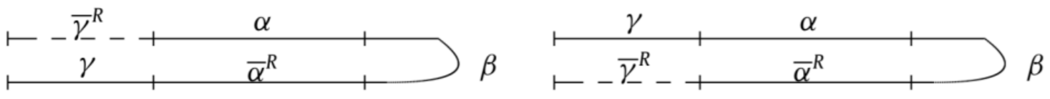

Figure 1 An illustration of the left and right hairpin completion operations.

1.2 Previous and related work

The hairpin completion operation has been introduced by Cheptea, Martin-Vide and Mitrana [4]. In several papers, the hairpin completion and other familiar operations have been studied [3, 5, 6, 7, 8, 9, 13, 17, 19, 20, 21, 22, 23, 24].

Hairpin reduction [3, 22, 23] was introduced as an inverse operation for hairpin completion. The hairpin reduction of a string x consists of all strings y such that x can be obtained from y by hairpin completion. Further, two variants of hairpin completion were considered, as they seem more appropriate for practical implementation: hairpin lengthening and bounded hairpin completion [9, 19, 21]. The first variant consist of adding a prefix or a suffix of γ. The second variant assumes that the length of the added prefix or suffix is bounded by a constant. Besides the algorithmic aspects, hairpin completion operation has been studied from the language theory point of view in several papers [5, 6, 8, 13, 17].

Manea and Mitrana introduced the minimum distance common k-hairpin completion ancestor of two strings in [22] where they presented a cubic time algorithm to compute the ancestor. Afterwards, Manea, Martin-Vide, and Mitrana [20] suggested a cubic time algorithm to tackle the k-hairpin completion distance problem. In addition, in [18] improved the time complexity to $O(n^2 \log n)$ to both problems, where n is the length of the longest string.

1.3 Our results

The focus of this paper is on two algorithmic problems related to iterated hairpin completion: k-hairpin completion distance and minimum distance common k-hairpin completion ancestor. Our main results are improving the upper bound on both problems with a $\log n$ factor, from $O(n^2 \log n)$ to $O(n^2)$. For the k-hairpin completion distance, our speedup is based on using incremental tree, a data structure proposed by Kaplan and Shafrir [10] which can support in constant time the following operations in a weighted tree: return the edge with minimum weight on a path and add a leaf to the tree. Our algorithm for finding a minimum distance k-hairpin completion ancestor of two strings (x, y) is based on dynamic programming technique presented in [18]. As in [18], we are interested in constructing the table DP_x, where $DP_x[i][j]$ represents the minimum number of k-hairpin completion operations to transform $x[i \dots j]$ into x. Similarly, we would like to compute a table DP_y. Our speedup relies in an $O(n^2)$ time algorithm for computing these tables by rephrasing the problem of computing DP_x in terms of shortest distances in a graph and replacing the segment tree used in [18] with doubly linked list and changing the order we process the cells in the matrix.

2 Preliminaries

We start with basic notations related to strings. An alphabet Σ is a finite, non-empty set of symbols. Throughout this paper, we mostly discuss strings over the alphabet $\Sigma = \{A, C, G, T\}$. For a letter $x \in \Sigma$, we denote as $\overline{x}$ the letter in Σ that is complementary to x. For the previously mentioned alphabet, we have $\overline{A} = T$ and $\overline{C} = G$. The set of all strings over an alphabet Σ is denoted by Σ^*. The empty string is denoted as λ, and $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$. Given a string $w \in \Sigma^*$, we denote by $|w|$ its length. If $w = xy$, $x, y \in \Sigma^*$ then x is called prefix and y a suffix. For a string w, $w[i \dots j]$ denotes the substring of w starting at position i and ending at position j, $1 \le i \le j \le |w|$. Given a string $s \in \Sigma^+$, we denote by $\overline{s} = \overline{s_1}\ \overline{s_2} \dots \overline{s_{|s|}}$ the complement of the string s and s^R the reversed string of s, i.e. $s^R = s_{|s|} s_{|s|-1} \dots s_1$.

Incremental tree is a data structure introduced by Kaplan and Shafrir [10] based on a similar structure of Alstrup and Holm [1] for the level ancestor problem, to maintain a rooted tree T, with an integer weight on each edge, such that the following operations are supported in $O(1)$ amortized time:

- add-leaf$_\mathrm{T}$(v, w, c): Add a new leaf v with parent w to T. The weight of the edge (v, w) is c.
- add-root$_\mathrm{T}$(v, c): Add a new root v to T. The old root (r) becomes a child of v and the weight of edge (r, v) is c.
- min-edge$_\mathrm{T}$(v, w): Returns the edge with minimum weight on the path from v to w.
- change-weight$_\mathrm{T}$(v, c): v is a leaf or v's parent is the root of T . Changes the weight of the edge between v and its parent to c.

From this data structure we will use just add-leaf$_\mathrm{T}$ and min-edge$_\mathrm{T}$ operations.

2.1 Hairpin Operations

For a string $x \in \Sigma^+$ and a positive integer $k \in \mathbb{N}$, k-hairpin completion is a family of transformations that can be applied to x. When applying a *left k-hairpin completion*, we select a non-empty suffix γ of x such that x can be partitioned into $x = \alpha\beta\overline{\alpha^R}\gamma$ with

$\alpha, \beta, \gamma \in \Sigma^+$ and $|\alpha| = k$. We execute the left hairpin operation by appending $\overline{\gamma^R}$ to the beginning of s. Formally, the set of strings that can be obtained from x by applying a single left k-hairpin operation is denoted as

$$HCL_k(x) = \{\overline{\gamma^R}x | x = \alpha\beta\overline{\alpha^R}\gamma, |\alpha| = k, \alpha, \beta, \gamma \in \Sigma^+\}$$

A *right k-hairpin completion* is defined in a symmetrical manner and the set of strings that can be obtained from s by applying a single right k-hairpin completion operation is denoted as

$$HCR_k(x) = \{x\overline{\gamma^R} | x = \gamma\alpha\beta\overline{\alpha^R}, |\alpha| = k, \alpha, \beta, \gamma \in \Sigma^+\}$$

▶ **Example 1.** The string $s = GAATCT$ can be partitioned into $\alpha = GA$, $\beta = A$, $\overline{\alpha^R} = TC$ and $\gamma = T$. Applying the left hairpin completion operation on s with this partitioning yields the string $AGAATCT$. Also, s can be partitioned into $\gamma = GA$, $\alpha = A$, $\beta = TC$, $\overline{\alpha^R} = T$ and by applying right hairpin completion operation we obtain $GAATCTTC$.

Collectively, the set of strings that can be obtained from x either by applying a right or a left k-hairpin completion operation is denoted as

$$HC_k(x) = HCL_k(x) \cup HCR_k(x)$$

The hairpin completion is the variant of the k-hairpin completion where we do not place a bound on the length of prefix. The hairpin completion of x is defined by:

$$HC(x) = \bigcup_{k \geq 1} HC_k(x)$$

We extend the notation of hairpin completion to sets of strings in the following way, for a set $L \subseteq \Sigma^*$ and a positive integer k,

$$HC_k(L) = \bigcup_{x \in L} HC_k(x) \qquad HC(L) = \bigcup_{x \in L} HC(x)$$

For every non negative integers k, i and string $x \in \Sigma^+$, we denote as $HC_k^i(x)$ the set of strings that can be obtained from x using exactly i k-hairpin completion operations and $HC_k^*(x)$ as the set of strings that are obtainable from x using any number of k-hairpin completion operations. Similarly, we denote as $HC^i(x)$ and $HC^*(x)$ the sets of strings obtainable from x by applying i (resp. any number) of hairpin operations, respectively. Formally,

$$HC_k^0(x) = \{x\} \qquad HC_k^{i+1}(x) = HC_k(HC_k^i(x)) \qquad HC_k^*(x) = \bigcup_{i \geq 0} HC_k^i(x)$$

$$HC^0(x) = \{x\} \qquad HC^{i+1}(x) = HC(HC^i(x)) \qquad HC^*(x) = \bigcup_{i \geq 0} HC^i(x)$$

$$HC_k^*(L) = \bigcup_{x \in L} HC_k^*(x) \qquad HC^*(L) = \bigcup_{x \in L} HC^*(x)$$

▶ **Definition 2** (k-Hairpin Completion Common Ancestor). *A string w is a common k-hairpin completion ancestor of two strings x and y if $\{x, y\} \subseteq HC_k^*(w)$. We denote the set of common k-hairpin ancestors of x and y as $HCA_k(x, y)$.*

▶ **Definition 3** (k-Hairpin Completion Distance)**.** *Given two strings x and y such that $|x| \leq |y|$, the k-hairpin completion distance between x and y is the minimal number of k-hairpin operations required to obtain y from x. Formally*

$$HCD_k(x, y) = \begin{cases} \min\{p | x \in HC_k^p(y)\} \\ \infty, x \notin HC_k^*(y) \end{cases}$$

▶ **Definition 4** (Minimum Distance k-hairpin Completion Ancestor)**.** *For two strings $x, y \in \Sigma^*$, a k-hairpin completion ancestor $w \in HCA_k(x, y)$ is a minimum distance k-hairpin completion ancestor of x and y if $\forall w' \in HCA_k(x, y)$ it holds that $HCD_k(w, x) + HCD_k(w, y) \leq HCD_k(w', x) + HCD_k(w', y)$, i.e. w minimizes the sum of the k-hairpin completion distances from x and from y.*

▶ **Definition 5** (Border)**.** *Given a string $s[1 \dots n] \in \Sigma^+$, $Border(s)$ is the length of the longest prefix of the string s which is also a complemented reversed suffix of this string. Formally, $Border(s) = \max(\{t | s[1 \dots 1 + t - 1] = \overline{s[n - t + 1 \dots n]^R}\} \cup \{0\})$. This definition can be easily extended for any substring $s[i \dots j]$ in the following way: $Border(s[i \dots j]) = \max(\{t | s[i \dots i + t - 1] = \overline{s[j - t + 1 \dots j]^R}\} \cup \{0\})$*

▶ Remark 6. Note that the above definition for border is different than the common definition, which is usually the largest prefix of x which is also a suffix of x.

Since in the k-hairpin completion operation we have to make sure that $|\alpha| = k$, we introduce the definition of k-Border.

▶ **Definition 7** (k-Border)**.** *Given a string $s \in \Sigma^+$, k-$Border(s) = \max(Border(s) - k, 0)$.*

Hairpin reduction is the inverse operation of hairpin completion. The hairpin reduction of a string x consists of all strings y such that x can be obtained from y by hairpin completion. For a string $x \in \Sigma^+$ and a positive integer $k \in \mathbb{N}$, k-hairpin reduction is a family of transformations that can be applied to x. When applying a *left hairpin reduction*, we select a non-empty prefix γ of x such that x can be partitioned into $\gamma\alpha\beta\overline{\alpha^R\gamma^R}$ with $\alpha, \beta, \gamma \in \Sigma^+$ and $|\alpha| = k$. We execute the left hairpin reduction operation by deleting γ. Formally, the set of strings that can be obtained from x by applying a single left k-hairpin reduction operation is denoted as

$$HRL_k(x) = \{\alpha\beta\overline{\alpha^R\gamma^R} | x = \gamma\alpha\beta\overline{\alpha^R\gamma^R}, |\alpha| = k, \alpha, \beta, \gamma \in \Sigma^+\}$$

A *right k-hairpin reduction* operation is defined in a symmetrical manner and the set of strings that can be obtained from x by applying a single right k-hairpin reduction operation is denoted as

$$HRR_k(x) = \{\gamma\alpha\beta\overline{\alpha^R} | x = \gamma\alpha\beta\overline{\alpha^R\gamma^R}, |\alpha| = k, \alpha, \beta, \gamma \in \Sigma^+\}$$

The set of strings that can be obtained from x either by applying a left or a right k-hairpin reduction operation is denoted as

$$HR_k(x) = HRL_k(x) \cup HRR_k(x)$$

The hairpin reduction is the variant of the k-hairpin reduction where we do not place a bound on the length of prefix. The hairpin reduction of x is defined by:

$$HR(x) = \bigcup_{k \geq 1} HR_k(x)$$

We make the following observation.

▶ **Observation 8.** *Let $x[1 \dots n]$ be a string with k-Border l.*

$$HRL_k(x) = \bigcup_{j \in [1 \dots l]} \{x[j+1 \dots n]\} \qquad HRR_k(x) = \bigcup_{j \in [1 \dots l]} \{x[1 \dots n-j]\}$$

$$HR_k(x) = \bigcup_{j \in [1 \dots l]} \{x[j+1 \dots n], x[1 \dots n-j]\}$$

Now we are ready to introduce the problems that we study in this paper.

▶ **Problem 1** (Hairpin completion distance). *Let Σ be the alphabet and $x, y \in \Sigma^+$. Compute $HCD_k(x, y)$.*

▶ **Problem 2** (Minimum distance common hairpin completion ancestor). *Let Σ be the alphabet and $x, y \in \Sigma^+$. Compute a minimum-distance common k-hairpin completion ancestor of x, y.*

2.2 Suffix Tree and Extension queries

The *suffix tree* [26] is a useful string data structure.

▶ **Definition 9.** *Let $S_1, \dots, S_k$ be strings over alphabet Σ and let $\$ \notin \Sigma$.*

A trie *of strings $S_1, \dots, S_k$ is an edge-labeled tree with k leaves. Every path from the root to a leaf corresponds to a string S_i with a \$ symbol appended to its end. The edges on this path are labeled by the symbols of S_i. Strings with a common prefix start at the root and follow the same path of the prefix, the paths split where the strings differ.*

A compacted trie *is a trie with every chain of edges connected by degree-2 nodes contracted to a single edge whose label is the concatenation of the symbols on the edges of the chain.*

Let $S = S[1], \dots, S[n]$ be a string over alphabet Σ. Let $\{S_1, \dots, S_n\}$ be the set of suffixes of S, where $S_i = S[i \dots n]$, $i = 1, \dots, n$. A suffix tree *of S is the compacted trie of the suffixes $S_1, \dots, S_n$.*

For every node u, we call the concatenation of the labels on the path from the root to u the *locus* of u denoted as $\mathcal{L}(u)$. For an edge e in the compact trie, we use the same notation $\mathcal{L}(e)$ to denote the label (or the locus) of e. Finally, for a downwards path P in the compact trie, the locus $\mathcal{L}(P)$ is the concatenation of the loci of the edges in P. In a compact trie, an edge e can have label s.t. $|\mathcal{L}(e)| > 1$. We refer to the symbol $\mathcal{L}(e)[1]$ as the symbol of e.

▶ **Theorem 10** (Weiner [26]). *For finite alphabet Σ, the suffix tree of a length-n string can be constructed in time $O(n)$. For general alphabets it can be constructed in time $O(n \log \sigma)$, where $\sigma = \min(|\Sigma|, n)$.*

For two strings $S[1 \dots n]$ and $T[1 \dots m]$, a string $P[1 \dots p]$ is a *common prefix* of S and T if $S[1 \dots p] = T[1 \dots p] = P$. We say that P is the *longest common prefix* (LCP) of S and T if P is a common prefix and $m = p$ or $n = p$ or $S[p+1] \neq T[p+1]$. Similarly, a string $A[1 \dots a]$ is a common suffix of S and T if $S[n-a+1 \dots n] = T[m-a+1 \dots m] = A$. A is the longest common suffix (LCS) if $n = a$ or $m = a$ or $S[n-a] \neq T[m-a]$. Collectively, we refer to LCP and LCS as longest common extensions (LCE).

By preprocessing the suffix tree of a string S for level ancestor queries [2], we can obtain the following.

▶ **Lemma 11** (Longest Common Extension Data Structure). *A string S can be preprocessed in $O(n)$ time to support the following queries in $O(1)$ time.*

1. *$LCP(i, j)$ - return the length of the longest common prefix of $S[i \dots n]$ and $S[j \dots n]$*
2. *$LCS(i, j)$ - return the length of the longest common suffix of $S[1 \dots i]$ and $S[1 \dots j]$*

By constructing the above data structure for the string $x\$\overline{x^R}$ with $\$ \notin \Sigma$, we obtain the following.

▶ **Corollary 12.** *We can process a string $S[1\dots n]$ in linear time to construct a data structure for answering the following query in $O(1)$ time.*

k-Border$(S[i\dots j])$ – Return the length of the k-Border of $S[i\dots j]$.

3 Hairpin completion distance

In this section we study Problem 1.

Our algorithm is based on the dynamic programming technique presented in [18]. For the sake of clarity, we briefly describe this technique. Without loss of generality, we assume that $|x| \leq |y|$ and $n = |y|, m = |x|$. We are interested in computing a dynamic programming table $DP[n][n]$ with dimensions $n \times n$. For every two indices $1 \leq i \leq j \leq n$, we define $DP[i][j]$ to be the minimum number of k-hairpin completion operations to transform x into $y[i\dots j]$. Formally, $DP[i][j] = HCD_k(x, y[i\dots j])$.

▶ **Definition 13.** *Given two non-negative integers i, j $(1 \leq i \leq j \leq n)$, L_j represents the DP values of all strings that can generate the substring $y[i\dots j]$ through a single left k-hairpin completion operation (elements of the set $HRL_k(y[i\dots j])$).*

▷ Claim 14. $L_j = \{DP[i+1][j], \dots, DP[i+l][j]\}$ where l is the k-Border of $y[i\dots j]$.

▶ **Definition 15.** *Given two non-negative integers i, j $(1 \leq i \leq j \leq n)$, R_i represents the DP values of all strings that can generate the substring $y[i\dots j]$ through a single right k-hairpin completion operation (elements of the set $HRR_k(y[i\dots j])$).*

▷ Claim 16. $R_i = \{DP[i][j-l], \dots, DP[i][j-1]\}$ where l is the k-Border of $y[i\dots j]$.

Correctness of Claim 14 and Claim 16 is based on Observation 8.

▶ **Lemma 17.** *Given two non-negative integers i, j $(1 \leq i \leq j \leq n)$, we have that $DP[i][j] = \min(\min L_j, \min R_i) + 1$.*

For the proof of Lemma 17 we refer to [18].

All positions in DP are initialized with ∞. We start by considering the base cases. These are represented by all subsequences $y[i\dots j] = x$. To determine them, we use any pattern matching algorithm which runs in linear time, for example KMP [16] and set $DP[i][j] = 0$. Analyzing the elements of the sets L_j and R_i, it can be seen that they actually represent continuous blocks from line i or column j. Thus, determining the minimum values from each of those sets is a range minimum query.

▶ **Definition 18.** *Given two non-negative integers i, j $(1 \leq i \leq j \leq n)$, DSL_j represents the data structure that keeps the DP values of column j and DSR_i represents the data structure that keeps the DP values of row i. (Note that we don't have to keep the values below the main diagonal)*

Naively, DSL_j and DSR_i could be arrays, which leads to constant update time, but linear query time. The overall complexity of the algorithm with this naive implementation is $O(n^3)$. In [18], the algorithm is implemented using segment trees, which leads to a logarithmic time for queries and for updates.

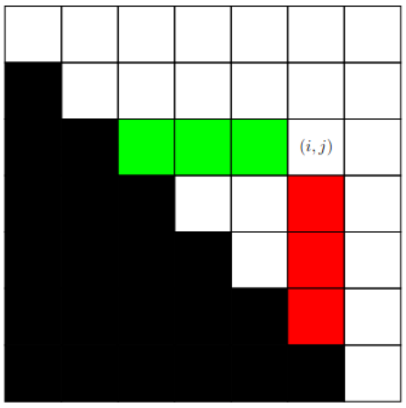

Figure 2 We compute the DP matrix in increasing order of difference $j - i$ (parallel with the main diagonal). Red line represents DSL_j and the green line DSR_i.

Considering that the update operations are append-like, i.e. they are only done after the first/last index of DSL_j and DSR_i, we propose using an incremental tree. The advantages of this approach consist in the fact that this structure can perform query and update operations in constant time. Practically, we keep an incremental tree for each row and column. A row or a column in the matrix represents a particular case of a tree, more precisely a chain. For the range minimum query needed in the computation of $DP[i][j]$ we use incremental tree's min-edge$_\text{T}$ operation. After we compute the $DP[i][j]$ value, we have to add to DSR_i and DSL_j. This can be done by using the add-leaf$_\text{T}$ operation.

Algorithm 1 An $O(n^2)$ algorithm for Problem 1.

```
    Input: $x, y \in \Sigma^+$
    Output: $HCD_k(x, y)$
 1: $DP[i][j] = \infty, \forall\ 1 \le i \le j \le n$
 2: Find all pairs $(i, j)$ such that $x = y[i \dots j]$ and set $DP[i][j] = 0$.
 3: for $len \leftarrow m + 1$ to $n$ do
 4:     for $i \leftarrow 1$ to $n - len + 1$ do
 5:         $j \leftarrow i + len - 1$
 6:         if $DP[i][j] = \infty$ then
 7:             $x \leftarrow min\text{-}edge_{DSR_i}(j - k\text{-}Border(s[i \dots j]), j - 1)$
 8:             $y \leftarrow min\text{-}edge_{DSL_j}(i + 1, i + k\text{-}Border(s[i \dots j]))$
 9:             $DP[i][j] = min(x, y) + 1$
10:         end if
11:         $add\text{-}leaf_{DSR_i}(j, j - 1, DP[i][j])$
12:         $add\text{-}leaf_{DSL_j}(i, i + 1, DP[i][j])$
13:     end for
14: end for
15: return $DP[1][n]$
```

▶ **Theorem 19.** *Algorithm 1 solves Problem 1 in $O(n^2)$.*

Proof.

Correctness. We prove the correctness of the algorithm by induction over the algorithm execution. The base cases correspond to the substrings $y[i \dots j] = x$. In these cases, $DP[i][j] = 0$ because no operation is needed to convert x to $y[i \dots j]$. Suppose we want to calculate the value of $DP[i][j]$. We remind that $DP[i][j] = \min(\min L_j, \min R_i) + 1$. We can rewrite the elements of the set R_i in the following form $DP[i][p]$ with $j - Border(i,j) + k \leq p < j$ and the elements of the set L_j in the form $DP[s][j]$ with $i < s \leq i + Border(i,j) - k$. Taking into account the iteration order (increasing according to the difference $j - i$) and $j > i$, we obtain the following inequalities: $j - i > j - s$ and $j - i > p - i$. Thus, it is guaranteed that when we want to calculate $DP[i][j]$ all the necessary values are already calculated.

Complexity. Line 1 runs in $O(n^2)$ and Line 2 in $O(n+m)$. For each cell above the main diagonal we have two queries and two updates both done in $O(1)$ amortized time. The overall time complexity is therefore $O(n^2)$. ◀

4 Minimum distance common hairpin completion ancestor

In this section we study Problem 2.

Our algorithm is based on the dynamic programming technique described in [18], but we replace the segment tree with a linked list and change the order of processing the cells in the matrix. Without loss of generality, we assume that $|x| \geq |y|$ and $n = |x|, m = |y|$. As in [18], we are interested in constructing the table $DP_x[1 \dots n][1 \dots n]$ with $DP_x[i][j] = HCD_k(x[i \dots j], x)$. Similarly, we would like to compute a table $DP_y[1 \dots m][1 \dots m]$ with $DP_y[i][j] = HCD_k(y[i \dots j], y)$. Our speedup relies on an $O(n^2)$ time algorithm for computing DP_x and DP_y.

We are interested in rephrasing the problem of computing DP_x in terms of shortest distances in a graph. We present the following definition.

▶ **Definition 20** (Hairpin Deletion Graph). *For a string $x[1 \dots n]$, we define the Hairpin Deletion Graph $G_h(x) = (V, E)$ of x as follows.*

- $V = \{x[i \dots j] | 1 \leq i \leq j \leq n\}$ *is the set of substrings of x.*
- $E = \{(x[i \dots j], x[a \dots b]) | x[i \dots j] \in HR_k(x[a \dots b])\}$ *I.e. there is a directed edge from substring A to the substring B if A can be obtained from B by applying a single hairpin completion operation.*

It is easy to see that $HCD_k(x[i \dots j], x)$ is exactly the length of the shortest path from $x[1 \dots n]$ to $x[i \dots j]$ in $G_h(x)$. Following Observation 8, we present the following characterization of the edges in $G_h(x)$.

▶ **Corollary 21.** *Let $x[1 \dots n]$ be a string and let $A = x[i \dots j]$ be a substring of x with k-Border length l. The set of edges emerging from A in $G_h(x)$ is*

$$E_A = \bigcup_{p \in [1 \dots l]} \{(A, x[i+p \dots j]), (A, x[i \dots j-p])\}$$

We call edges from $A = x[i \dots j]$ to a suffix $[i+p \dots j]$ a downward edge and an edge from A to a prefix $x[i \dots j-p]$ a leftward edge. When a path in $G_h(x)$ uses a downward (resp. left) edge, we say that it takes a step down (resp. leftward). For example, if $P = (v_1, v_2 \dots v_z)$ is a path in $G_h(x)$, and the edge (v_{z-1}, v_z) is a downward (resp. leftward) edge, we say that P ends with a step left (resp. down).

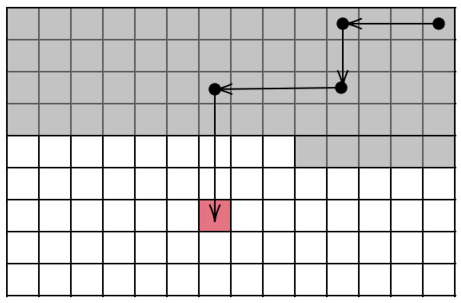

Figure 3 A demonstration of a restricted path to the red square. The grey squares resemble cells that precede the cell (i, j).

Then, the algorithm computes the cells of DP_x row by row from top to bottom, iterating a row in decreasing order of the columns. Formally, when iterating the cell $DP_x[i][j]$, the algorithm have already computed the cells $DP_x[a][b]$ with $a < i$ and the cells $DP_x[i][b]$ with $b > j$. The order of the iteration implies a total order on the pairs $i, j \in [n] \times [n]$.

▶ **Definition 22** (Iteration Order). *For two pairs of integers $(i_1, j_1), (i_2, j_2) \in [n] \times [n]$, we say that (i_1, j_1) precedes (i_2, j_2) (denoted as $(i_1, j_1) < (i_2, j_2)$) if the cell $DP_x[i_1][j_1]$ is iterated before $DP_x[i_2][j_2]$ by our algorithm. Similarly, we say (i_2, j_2) proceeds (i_1, j_1)*

We proceed to introduce a useful concept used by the algorithm.

▶ **Definition 23** (Restricted Path). *For a string $x[1 \dots n]$ and integers $i, j \in [n]$, a path $P = (x, v_1, v_2 \dots, v_z, A)$ from x to A in $G_h(x)$ is (i, j)-restricted if for every $r \in [z]$ we have $v_r = x[a_r \dots b_r]$ such that (a_r, b_r) precedes (i, j). For $(i, j) = (0, 0)$, we say that there is no $(0, 0)$-restricted path.*

For integer pairs $(i, j), (a, b) \in [n][n]$ such that $(i, j) < (a, b)$, we denote as $ResL_{(i,j)}[(a, b)]$ the length of the shortest (i, j)-restricted path from x to $x[a \dots b]$ in $G_h(x)$ that ends with a step left. Similarly, we denote as $ResD_{(i,j)}[(a, b)]$ the length of the shortest (i, j)-restricted path from x to $x[a \dots b]$ in $G_h(x)$ that ends with a step down.

We make the following observations regarding the structure of $ResL_{(i,j)}[(a, b)]$ and $ResD_{(i,j)}[(a, b)]$.

▶ **Lemma 24.** *For every $i, j \in [n]$ and $a, b \in [n] \times [n-1]$ such that $(i, j) < (a, b)$ it is satisfied that $ResL_{(i,j)}[(a, b + 1)] \le ResL_{(i,j)}[(a, b)]$.*

Proof. If there is no (i, j)-restricted path that ends with a step leftwards from $(0, 0)$ to (a, b), the claim is vacuously true. Otherwise, let $P = (1, n), (x_1, y_1), (x_2, y_2) \dots (x_d, y_d), (a, b)$ be the shortest (i, j)-restricted path from $(1, n)$ to (a, b) that ends with a step to the left. Since P ends with a step leftwards, we have $x_d = a$ and there is an edge from (a, y_d) to (a, b). According to Corollary 21, there is also an edge from (a, y_d) to $(a, b + 1)$. Therefore, we can replace (a, b) with $(a, b + 1)$ in P to obtain an (i, j)-restricted path P' with length $|P|$ from $(1, n)$ to $(a, b + 1)$. ◀

The following symmetric statement can be proven in a similar manner.

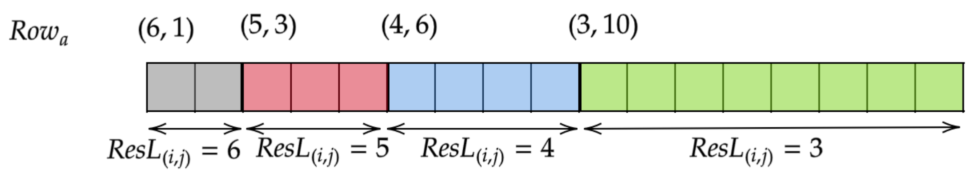

Figure 4 The list Row_a above the a'th row of DP_x. Every pair (δ, β) appears above the cell (a, β). The content of Row_a implies $ResL(i, j)$ for the cells in the a'th row. Every cell in the green region has the boundary predecessor $(3, 10)$ in Row_k. So for every b in the green region, $ResL_{(i,j)}[(a, b)] = 3$.

▶ **Lemma 25.** *For every $i, j \in [n]$ and $a, b \in [n-1] \times [n]$ such that $(i, j) < (a, b)$ it is satisfied that $ResD_{(i,j)}[(a, b)] \leq ResD_{(i,j)}[(a+1, b)]$.*

Lemma 24 and Lemma 25 suggest that the values of $ResL_{(i,j)}$ (resp. $ResD_{(i,j)}$) in every row (resp. column) are monotonic.

For every row $k \in [n]$ of DP_x, the algorithm maintains a corresponding double-sided linked list Row_k. Similarly, for every column $k \in [n]$ the algorithm maintains a list Col_k. Conceptually, Row_k (resp. Col_k) compactly represents the values $ResL_{(i,j)}[(a, b)]$ (resp. $ResD_{(i,j)}(a, b)$) for all the cells (a, b) in row k (resp. in column k). Every list stores a sequence of pairs of integers (δ, β). The first value δ is called the *distance* and the second value β is called the *boundary*. We call such pairs *boundary pairs*. The pairs are stored in increasing order of distances. For an integer x, we call the pair (δ, β) in a list the *boundary predecessor* (resp. boundary successor) of x in Row_k if β is the minimal (resp. maximal) boundary in the list that is at least (resp. at most) x.

When processing $DP_x[i][j]$, we are interested in maintaining the following invariant regarding the pairs stored in Row_a (for every $a \in [n]$):

Let $b \in [n]$ be an integer such that $(i, j) < (a, b)$ and let (δ, β) be the boundary predecessor of b in Row_a. It holds that $ResL_{(i,j)}[(a, b)] = \delta$. Equivalently: Let $(\delta_1, \beta_1), (\delta_2, \beta_2) \ldots (\delta_z, \beta_z)$ be the pairs in Row_a. Note that due to Lemma 24, the pairs in Row_k are naturally stored in decreasing order of their boundaries. If an integer b satisfies $b \in [\beta_r \ldots \beta_{r-1} - 1]$ for some $r \in [z]$, then $ResL_{(i,j)}[(a, b)] = \delta_r$. For a visualization, see Figure 4.

Essentially, the list Row_a stores an implicit representation of the shortest (i, j)-restricted paths that end with a left step to the cells in row a.

Similarly, the list Col_b stores an implicit representation of $ResD_{(i,j)}[(a, b)]$ for vertices in column b as follows. For every $a \in [n]$ such that $(i, j) < (a, b)$ with boundary successor (δ, β) in Col_b, it holds that $ResD_{(i,j)}[(a, b)] = \delta$.

Throughout the iterations, we maintain the pair $r = (\delta_r, \beta_r)$ such that when we iterate $DP_x[i][j]$, the pair r is the boundary predecessor j in Row_i. We also store n pairs $c_1, c_2 \ldots c_n$ such that when iterating $DP_x[i][j]$, the pair $c_j = (\delta_c^j, \beta_c^j)$ is the boundary successor of i in Col_j. We initialize every list Row_k with a single pair $(\infty, 1)$ and every list Col_k with a single pair (∞, n). For the sake of consistency, we treat the initialization of the algorithm as a phase in the iteration in which a dummy cell $(0, 0)$ is the currently iterated cell. Note that the initialization for the lists suggests that for every vertex, there is no $(0, 0)$-restricted path that ends with a step to downwards or leftwards.

Processing a cell. When processing $DP_x[i][j]$, we first obtain the distance to $DP_x[i][j]$ using r and c_j. The shortest path to (i,j) must end with a step to the left from a vertex in row i or with a step downwards from a vertex in column j. Note that all the cells to the right of (i,j) and above it have already been processed. It follows that the shortest path to (i,j) is an (i',j')-restricted path with (i',j') being the cell processed in the previous iteration. Let $r=(\delta_r,\beta_r)$ and $c_j=(\delta_c^j,\beta_c^j)$. r is the boundary predecessor of i in Row_i, so according to the invariant we have $ResL_{(i',j')}=\delta_r$. Similarly, $ResD_{(i',j')}=\delta_c^j$. We can therefore set $DP_x[i][j]=\min(\delta_r,\delta_c^j)$.

The remaining task is to update the lists and the pointers in a manner that preserves our invariants. We make the following claims.

▷ Claim 26. For $(i,j)\in[n]\times[n]$ and $(a,b)\in[n]\times[n]$ such that $(i,j)<(a,b)$, if an (i,j)-restricted path to (a,b) visits the vertex (i,j) - (i,j) must be the second to last vertex in the path (i.e. the next vertex is (a,b).

Proof. According to Corollary 21, every edge emerging from (i,j) enter a vertex $(i'j')$ such that $(i,j)<(i',j')$. The only vertex in an (i,j)-restricted path that is allowed to proceed (i,j) is the destination vertex. ◁

Claim 26 suggests the following.

► **Corollary 27.** *Let $(i',j')\in[n]\times[n]$ and let $(i,j)\in[n]\times[n]$ be the vertex immediately following (i',j') in the iteration order. Let $(a,b)\in[n]\times[n]$ such that $(i,j)<(a,b)$. If there is no edge from (i,j) to (a,b) we have $ResL_{(i,j)}[(a,b)]=ResL_{(i',j')}[(a,b)]$ and $ResD_{(i,j)}[(a,b)]=ResD_{(i',j')}[(a,b)]$*

Furthermore, by Corollary 27 and Corollary 21 together, we obtain the following.

► **Corollary 28.** *For $k\neq i$ (resp. $k\neq j$), the list Row_k (resp. Col_k) does not need to be updated after the cell (i,j) is processed in order to satisfy the invariant.*

It follows from Corollary 28 that we only need to update the lists Row_i and Col_j to represent shortest (i,j)-restricted paths instead of representing shortest (i',j')-restricted paths. In other words, we need to update Row_i and Col_j to consider paths that use the vertex (i,j). Specifically, paths that use (i,j) as a second to last vertex (Claim 26)

We update the lists as follows. Let l be the k-Border of $x[i\dots j]$ and let d be the recently calculated $d=DP_x[i][j]=\min(\delta_r,\delta_c^j)$. According to Corollary 21, there is an edge from (i,j) to $(i,j-z)$ with $z\in[1\dots l]$, and only to those vertices in the i'th row. We call these vertices the *contested* vertices. For every contested vertex, there is an (i,j)-restricted path that ends with a step to the left via the vertex (i,j). This path has length $d+1$. Our task is to update Row_i such that every contested vertex (a,b) in the list with $ResL_{i',j'}[(a,b)]>d+1$ is updated to have $ResL_{(i,j)}[(a,b)]=d+1$. Every $(a,b)\in Row_i$ with $ResL_{(i',j')}[(a,b)]\le d+1$ needs to keep its current distance. The distances to uncontested vertices in the i'th row do not require an update (Corollary 27).

Assume w.l.o.g that $d=\delta_r$ (the case in which $d=\delta_c^j$ is treated symmetrically). We may need to add the boundary pair $(d+1,j-l)$ to Row_i to represent the newly available (i,j)-restricted paths. First, observe that $r=(d,\beta_r)$ should not be removed from Row_i. This is due to the cost of the newly available paths via (i,j) being $d+1$ - longer than the paths already represented by Row_i for the vertices (i,b) with $b\in[\beta_r\dots j]$. We follow the list pointer from r to obtain its boundary predecessor $r'=(\delta_1,\beta_1)$ in Row_i with $\beta_1<\beta_r$ and $\delta_1>d$. We consider the following cases.

Case 1.a: $\delta_1 = d+1$ and $j-l \geq \beta_1$. In this case, Row_i already represents the shortest restricted paths with cost $d+1$ to the vertices (i,k) with $k \in [j-l \dots \beta_r - 1]$. Therefore, no update is required for Row_i.

Case 1.b: $\delta_1 > d+1$ and $j-l \geq \beta_1$. In this case, we need to add the boundary pair $(d+1, j-l)$ after the boundary border r in Row_i. The following pairs in Row_i have boundaries smaller than $j-l$ and therefore represent the shortest paths uncontested vertices and do not need to be changed. If $j-l = \beta_1$, we also remove r' from Row_i, as it is redundant.

Case 2 : $j-l < \beta_1$. In this case, adding the pair $(d+1, j-l)$ after the pair r to Row_i may be insufficient. We also need to remove every pair (δ, β) in Row_i with $\beta \in [j-l \dots \beta_1]$. All of those pairs are now redundant in Row_i - as they represent paths with a length at least $d+1$ to contested vertices. We execute the deletion of these pairs in a straightforward manner by following the links from r' until we reach a pair $r^* = (\delta, \beta)$ with $\beta < j-l$. When r^* is finally met, we insert $(d+1, j-l)$ to Row_i between the r and r^*.

We proceed to treat Col_j. If $\delta_c^j = d$, the treatment of Col_j is completely symmetric to the treatment of Row_i. Otherwise, δ_c^j. As in the treatment of Row_i, our task is to add a representation of the paths with length $d+1$ to vertices (a,j) with $a \in [i \dots i+l]$.

Namely, every pair (δ, β) in Col_j with $\delta < i+l$ should be removed (including c^j), as it represents a path with length at least $d+1$ to one of the vertices (a,j) with $a \in [i, i+l]$. We execute the required deletion in a straightforward manner. Starting from c^j, we proceed to the next pair in the list until a pair (δ, β) with $\beta > j+l$ is found. We then remove the pairs iterated in this process from Col_j and append $(d+1, i+l)$ to the beginning of the list. This concludes the updates to Row_i and to Col_j. We note that if r of c_j is removed from Row_i or from Col_j, respectively, the new pair $(d+1, j-l)$ (or respectively, $(d+1, i+l)$) is becoming the new boundary predecessor (resp. boundary successor) of j in Row_i (resp. of i in Col_j).

Finally, we need to update r and c_j to be the boundary predecessor and successors required for the next iterated cell. If $i < n$, the i value will remain the same on the next iteration. In this case, if $\beta_r = j$, we update r to be the next element in Row_i. If $\beta_r < j$, we do not need to update r is it is also the boundary predecessor of $j-1$ in Row_i.

If $i = n$, the next iteration is the first step in row $i+1$. It follows that Row_{i+1} is still in its initialized state, and we set the only pair in $(\infty, 1) \in Row_{i+1}$ to be r.

As for the c_j pointers, we need to update all of them every time a new row is met. When moving from row i to row $i+1$, every c_j needs to be updated from the predecessor of i in Col_j to the predecessor of $i+1$ in Col_j. This is done in a symmetric manner to the update of r.

▶ **Lemma 29.** *The time complexity of the algorithm is $O(n^2)$.*

Proof. When a cell (i,j) is processed, the value in $DP_x[i][j]$ is decided in constant time. In the process of maintaining the lists invariant, at most one pair is added to the list Row_i and to the list Col_j. Several pairs may be removed from these lists, but since every element can be removed at most once throughout the algorithm - the overall time complexity for treating the lists is $O(n^2)$. The pointer r is updated in constant time during the processing of $DP_x[i][j]$. The pointers $c^1, c^2 \dots c^n$ are all updated in $O(n)$ when a table row is visited for the first time, which happens n times throughout the algorithm. ◀

After we compute the tables DP_x and DP_y we have to find a common substring z of x, y such that $HCD_k(z,x) + HCD_k(z,y)$ is minimum among all common substrings of x, y. We use the algorithm *Compute_MDCA* presented in [18] which can return the answer in

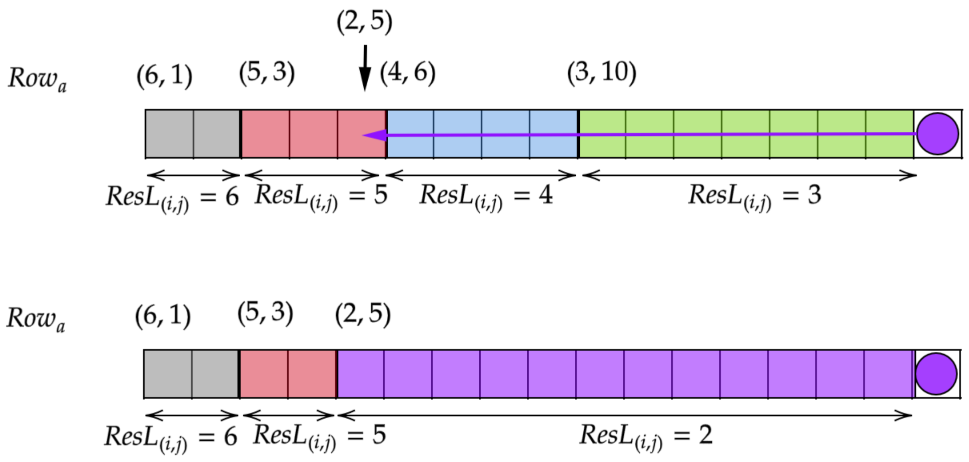

Figure 5 Case 2: The shortest path to the purple vertex is discovered to be 2. The k-Border of the substring corresponding to this vertex is 11 (denoted as the purple left arrow), thus creating new restricted paths to the vertices covered by the purple arrow (recall that in $G_h(x)$, there is a directed edge from the purple vertex to every one of the vertices covered by the purple arrow). The appropriate update to the list is removing the pairs $(3, 10)$ and $(4, 6)$ as the vertices in the green area and in the blue area are now accessible via a shorter path with length 2 via the purple vertex. This new paths are represented by the newly added pair $(2, 5)$.

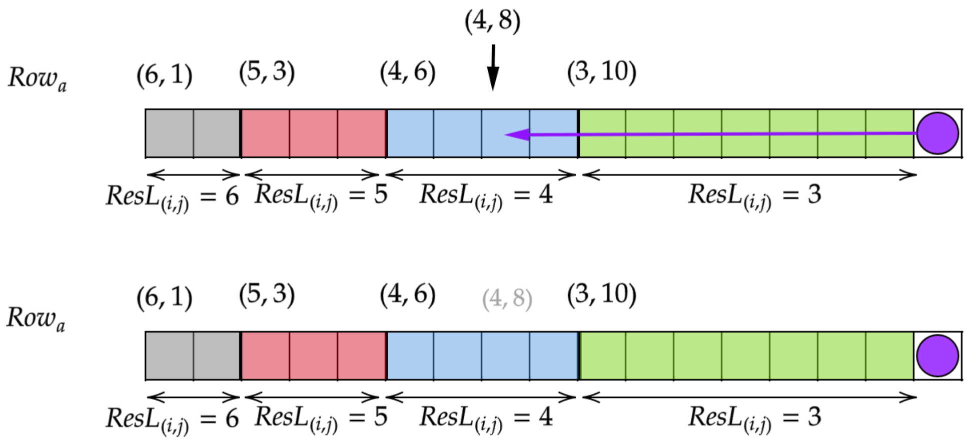

Figure 6 Case 1.a: The distance to the purple vertex is discovered to be 3, enabling new paths with length 4 to the vertices touched by the purple arrow (representing the length of the k-Border of the substring corresponding to the purple vertex). These new paths does not improve upon the restricted paths already represented in the list, so the pair $(4, 8)$ representing these new paths is simply not added to the list.

$O(n^2)$. To be clear, we provide a brief explanation of the algorithm. In the first stage, the algorithm builds a trie with all the suffixes of the string x. Then, it will traverse the trie for every suffix of y and at every match it will compute the sum of DP values. In short, this algorithm determines in quadratic time all the common substrings of x and y and keeps the one with the minimum sum of distances. We add the pseudocode for the algorithm described in this section in the appendix.

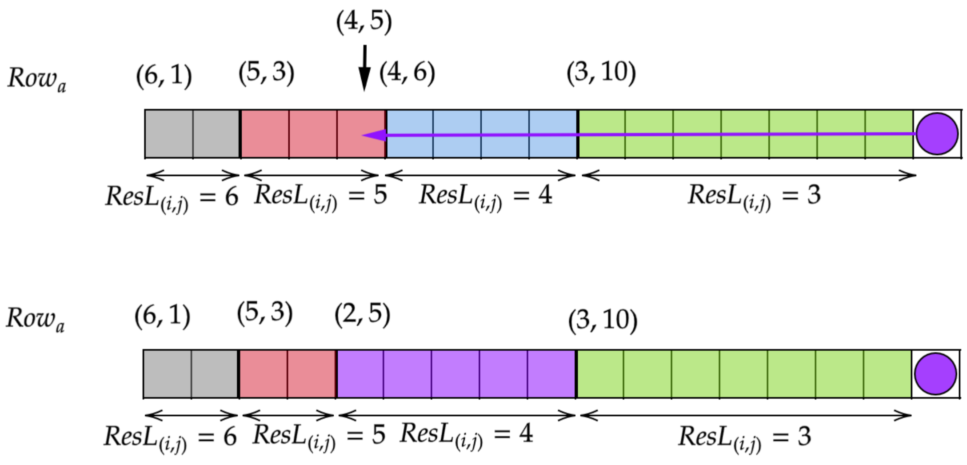

Figure 7 Case 2: The distance to the purple vertex is discovered to be 3. This creates new restricted paths with length 4 to the vertices touched by the purple arrow (representing the k-Border of the string corresponding to the purple vertex). For the vertices in the green area, this is not an improvement, as we already have a representation to a path with length 3 to those vertices. The distances to the vertices in the blue area and to the vertex in the red area touched by the purple arrow are longer or equal to 4. To represent this, we add the boundary pair $(4, 5)$ and remove the boundary pair $(4, 6)$ (as the k-Border $(4, 6)$ represented the distances to the vertices in the blue interval, which are now represented by $(4, 5)$.

5 Conclusions and future work

In this paper we study two problems related to the hairpin completion operation. We propose a quadratic time algorithm for solving these two problems, thus improving the runtime over previous work by Manea [18]. Notice that both our algorithms compute the dynamic programming table of the respective problem explicitly.

A question that arises from our work is can one find an algorithm that solves one of these problems by computing a small subset of cells in the dynamic programming table, which implies a runtime of $o(n^2)$. An interesting and challenging open problem is to provide an $o(n^2)$ algorithm for any of the two problems studied in this paper (not necessary with uses of the dynamic programming's formula), or present a lower bound matching with known problems.

For other variants of hairpin problems (see, e.g., [9, 20, 21]), we believe our techniques can help understand them better and help with designing efficient algorithms for these problems.

References

1 Stephen Alstrup and Jacob Holm. Improved algorithms for finding level ancestors in dynamic trees. In Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors, *Automata, Languages and Programming*, pages 73–84, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

2 Michael A Bender and Martın Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.

3 Henning Bordihn, Victor Mitrana, Andrei Păun, and Mihaela Păun. Hairpin completions and reductions: Semilinearity properties. *Natural Computing: An International Journal*, 20(2):193–203, June 2021.

4 Daniela Cheptea, Carlos Martin-Vide, and Victor Mitrana. A new operation on words suggested by DNA biochemistry: Hairpin completion. *Transgressive Computing*, January 2006.

5 Volker Diekert and Steffen Kopecki. Complexity results and the growths of hairpin completions of regular languages (extended abstract). In Michael Domaratzki and Kai Salomaa, editors, *Implementation and Application of Automata*, pages 105–114, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

6 Volker Diekert and Steffen Kopecki. It is NL-complete to decide whether a hairpin completion of regular languages is regular. *Computing Research Repository – CORR*, 22, January 2011. `arXiv:22`.

7 Volker Diekert, Steffen Kopecki, and Victor Mitrana. On the hairpin completion of regular languages. In Martin Leucker and Carroll Morgan, editors, *Theoretical Aspects of Computing – ICTAC 2009*, pages 170–184, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

8 Volker Diekert, Steffen Kopecki, and Victor Mitrana. Deciding regularity of hairpin completions of regular languages in polynomial time. *Information and Computation*, 217:12–30, 2012.

9 Masami Ito, Peter Leupold, Florin Manea, and Victor Mitrana. Bounded hairpin completion. *Information and Computation*, 209(3):471–485, 2011. Special Issue: 3rd International Conference on Language and Automata Theory and Applications (LATA 2009).

10 Haim Kaplan and Nira Shafrir. Path minima in incremental unrooted trees. In Dan Halperin and Kurt Mehlhorn, editors, *Algorithms – ESA 2008*, pages 565–576, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

11 Lila Kari, Stavros Konstantinidis, Elena Losseva, Petr Sosík, and Gabriel Thierrin. Hairpin structures in DNA words. In Alessandra Carbone and Niles A. Pierce, editors, *DNA Computing*, pages 158–170, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

12 Lila Kari, Stavros Konstantinidis, Petr Sosík, and Gabriel Thierrin. On hairpin-free words and languages. In Clelia De Felice and Antonio Restivo, editors, *Developments in Language Theory*, pages 296–307, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

13 Lila Kari, Steffen Kopecki, and Shinnosuke Seki. Iterated hairpin completions of non-crossing words. In Mária Bieliková, Gerhard Friedrich, Georg Gottlob, Stefan Katzenbeisser, and György Turán, editors, *SOFSEM 2012: Theory and Practice of Computer Science*, pages 337–348, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

14 Lila Kari, Elena Losseva, Stavros Konstantinidis, Petr Sosík, and Gabriel Thierrin. A formal language analysis of DNA hairpin structures. *Fundamenta Informaticae*, 71(4):453–475, 2006.

15 Lila Kari, Kalpana Mahalingam, and Gabriel Thierrin. The syntactic monoid of hairpin-free languages. *Acta Informatica*, 44(3):153–166, June 2007.

16 Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

17 Steffen Kopecki. On iterated hairpin completion. *Theoretical Computer Science*, 412(29):3629–3638, 2011.

18 Florin Manea. A series of algorithmic results related to the iterated hairpin completion. *Theoretical Computer Science*, 411(48):4162–4178, 2010.

19 Florin Manea, Carlos Martín-Vide, and Victor Mitrana. Hairpin lengthening. In Fernando Ferreira, Benedikt Löwe, Elvira Mayordomo, and Luís Mendes Gomes, editors, *Programs, Proofs, Processes*, pages 296–306, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

20 Florin Manea, Carlos Martín-Vide, and Victor Mitrana. On some algorithmic problems regarding the hairpin completion. *Discrete Applied Mathematics*, 157(9):2143–2152, 2009. Optimal Discrete Structures and Algorithms.

21 Florin Manea, Carlos Martín-Vide, and Victor Mitrana. Hairpin lengthening: language theoretic and algorithmic results. *Journal of Logic and Computation*, 25(4):987–1009, January 2013.

22 Florin Manea and Victor Mitrana. Hairpin completion versus hairpin reduction. In S. Barry Cooper, Benedikt Löwe, and Andrea Sorbi, editors, *Computation and Logic in the Real World*, pages 532–541, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

23 Florin Manea, Victor Mitrana, and Takashi Yokomori. Two complementary operations inspired by the DNA hairpin formation: Completion and reduction. *Theoretical Computer Science*, 410(4):417–425, 2009. Computational Paradigms from Nature.

24 Florin Manea, Victor Mitrana, and Takashi Yokomori. Some remarks on the hairpin completion. In *International Journal of Foundations of Computer Science*, 2010.

25 James D. Watson and Francis H. Crick. Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid. *Nature*, 171:737–738, 1953.

26 Peter Weiner. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

A Appendix

Algorithm 2 An $O(n^2)$ algorithm for Problem 2.

```
    Input: $x, y \in \Sigma^+$
    Output: a string $z$ such that $HCD_k(z,x) + HCD_k(z,y)$ is minimum
1: $DP_x = ComputeDP(x)$
2: $DP_y = ComputeDP(y)$
3: return $Compute\_MDCA(x, y, DP_x, DP_y)$
```

Algorithm 3 Updates the list Row_i.

```
1: procedure UPDATEROW$(i, j, r)$
2:     while $r$ is not NULL and $\beta_r > j - k\text{-}Border(s[i \dots j])$ and $\delta_r > DP_x[i][j]$ do
3:         delete $r$ from $Row_i$
4:         $r = r \rightarrow next$
5:     end while
6:     add $(j - k\text{-}Border(s[i \dots j]), DP_x[i][j] + 1)$ to $Row_i$
7: end procedure
```

Algorithm 4 Updates the list Col_j.

```
1: procedure UPDATECOL$(i, j, c_j)$
2:     while $c$ is not NULL and $\beta_c^j < i + k\text{-}Border(s[i \dots j])$ and $\delta_c^j > DP_x[i][j]$ do
3:         delete $c_j$ from $Col_j$
4:         $c_j = c_j \rightarrow next$
5:     end while
6:     add $(i + k\text{-}Border(s[i \dots j]), DP_x[i][j] + 1)$ to $Col_j$
7: end procedure
```

Algorithm 5 *ComputeDP*.

```
Input: $x \in \Sigma^+$
Output: $DP_x$
1:  $DP[i][j] = \infty, \forall\ 1 \le i \le j \le n$            ▷ $n$ is the length of the input string
2:  $DP_x[1][n] = 0$                                                 ▷ Base case
3:  add $(n - k\text{-}Border(s[1 \dots n]), 1)$ to $Row_1$
4:  for $i \leftarrow n - 1$ to $1$ do                               ▷ Compute the first line of $DP_x$
5:      if $i \le \beta_r$ then
6:          $DP_x[1][i] = \delta_r$
7:          if $j - k\text{-}Border(s[1 \dots i]) < \beta_r$ then
8:              updateRow$(1, i, r)$
9:          end if
10:     end if
11: end for
12: for $i \leftarrow 2$ to $n$ do
13:     for $j \leftarrow n$ to $1$ do
14:         if $\delta_r < \delta_c^j$ then
15:             if $j \ge \beta_r$ then
16:                 $DP_x[i][j] = \delta_r$
17:                 if $j - k\text{-}Border(s[i \dots j]) < \beta_r$ then
18:                     updateRow$(i, j, r)$
19:                     updateCol$(i, j, c_j)$
20:                 end if
21:             end if
22:         else
23:             if $i \le \beta_c^j$ then
24:                 $DP_x[i][j] = \delta_c^j$
25:                 if $i + k\text{-}Border(s[i \dots j]) > \beta_c^j$ then
26:                     updateCol$(i, j, c_j)$
27:                     updateRow$(i, j, r)$
28:                 end if
29:             end if
30:         end if
31:     end for
32: end for
33: return $DP_x$
```