Dynamic all scores matrices for LCS score

Amir Carmel[*] Dekel Tsur[*] Michal Ziv-Ukelson[*]

Abstract

The problem of aligning two strings A, B in order to determine their similarity is fundamental in the field of pattern matching. An important concept in this domain is the "all scores matrix" that encodes the local alignment comparison of two strings. Namely, let $\mathcal{K}$ denote the all scores matrix containing the alignment score of every substring of B with A, and let $\mathcal{J}$ denote the all scores matrix containing the alignment score of every suffix of B with every prefix of A.

In this paper we consider the problem of maintaining an all scores matrix where the scoring function is the LCS score, while supporting single character prepend and append operations to A and B. Our algorithms exploit the sparsity parameters $L = LCS(A, B)$ and $\Delta = |B| - L$. For the matrix $\mathcal{K}$ we propose an algorithm that supports incremental operations to both ends of A in $O(\Delta)$ time. Whilst for the matrix $\mathcal{J}$ we propose an algorithm that supports a single type of incremental operation, either a prepend operation to A or an append operation to B, in $O(L)$ time. This structure can also be extended to support both operations simultaneously in $O(L \log \log L)$ time.

1 Introduction

In the classical problem of sequence alignment, two strings are aligned according to some predifined scoring function. The common scoring schemes are the Edit Distance (ED) and the Longest Common Subsequence (LCS). This problem is fundamental in the field of pattern matching. It has broad applications to many different fields in computer science, among others: computer-vision, bioinformatics and natural language processing. Hence, it is of no suprise that this problem has attracted a vast amount of research and publications over the years.

Given two strings A and B of lengths m and n, respectively. The alignment problem of A and B can be naturally viewed as a shortest path problem on the *alignment graph* of A and B. That is, an $(n+1) \times (m+1)$ grid graph, in which, horizontal (respectively, vertical) edges correspond to alignment of a character in A (respectively, B) with a gap, and diagonal edges correspond to alignment of two characters in A and B (see Figure 1).

Landau et al. [23] introduced the problem of incremental string comparison, i.e. given an encoding of the global comparison of two strings A and B, to efficiently

[*]Department of Computer Science, Ben-Gurion University of the Negev.

compute the answer for A and σB, and the answer for σA and B. Furthermore, they show how incremental string comparison can be used to obtain more efficient algorithms for various problems in pattern matching, such as: the longest prefix approximate matching problem, the approximate overlap problem and cyclic string comparison. Incremental string comparison has also been applied to finding all approximate gapped palindromes [14], approximate regularities in strings [8, 38], consecutive suffix alignment [15, 22] and more [21, 33]. Several improvements to the algorithm of Landau et al. [23] have been proposed over the years [16, 17, 18, 20].

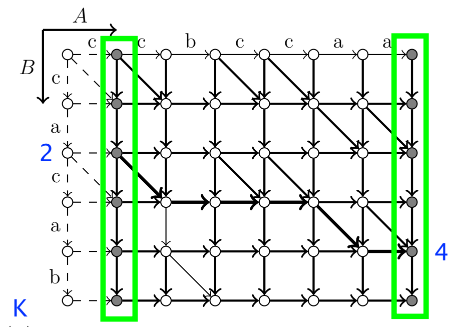
All scores matrices were introduced by Apostolico et al. [2] in order to obtain fast parallel algorithms for LCS computation. An *all scores matrix* is a matrix that stores the optimal alignment scores of one or more types from the following types: (I) B against every substring of A, (II) A against every substring of B, (III) every suffix of A against every prefix of B, and (IV) every suffix of B against every prefix of A. We denote by $\mathcal{L}$ the all scores matrix containing the optimal alignment scores of all the types defined above. We denote by $\mathcal{K}$ the all scores matrix containing the optimal alignment scores of type (II), and denote by $\mathcal{J}$ the all scores matrix containing the optimal alignment scores of type (IV) (see Figure 1). Due to symmetry, we will ignore the all scores matrices that contain optimal alignment scores of type (I) or of type (III). All scores matrices are also called DIST matrices [2, 32] or highest-score matrices [35].

The problem of constructing all scores matrices has been studied in [1, 7, 27, 31, 32, 35, 36]. We note that this problem is a special case of the more general problem of computing all pairs shortest paths in planar graphs [5, 6, 9, 10, 11, 28].

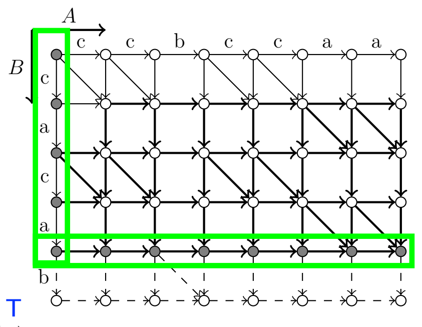
For an $n \times n$ matrix $\mathcal{D}$, its *density* matrix $\mathcal{D}^\square$ is an $(n-1)\times(n-1)$ matrix, where $\mathcal{D}^\square[i,j] = \mathcal{D}[i-1,j-1] + \mathcal{D}[i,j] - \mathcal{D}[i-1,j] - \mathcal{D}[i,j-1]$. A matrix is called *Monge* (resp. *anti-Monge*) if its density matrix is non-negative (resp. non-positive), and *sub-unit Monge* (resp. *sub-unit anti-Monge*) if every row or column of the density matrix contains at most one non-zero element, and all the non-zero elements are equal to 1 (resp. -1). All scores matrices are known to be sub-unit Monge or sub-unit anti-Monge [35]. Hence, all scores matrices can be encoded in linear space. Consequently, supporting incremental operations for all scores matrices can also serve to further reduce the space complexity for Incremental String Comparison, which was noted in [17] to be the main practical limitation.

We next consider the problem of incremental construction of all scores matrices. That is, we wish to maintain an all scores matrix of two strings A and B while supporting all or some of the following operations: appending a character to A, prepending a character to A, appending a character to B, and prepending a character to B. If the number of supported operations from the four operations above is k, we refer to the problem as the *k-sided incremental all scores matrix problem (k-IASM)*.

The algorithm of Schmidt [32] for all scores matrix construction also solves the 2-IASM problem on the matrix $\mathcal{L}$. For discrete score functions, including LCS scores, the algorithm of Schmidt requires $O(m+n)$ space and it supports incremental operations to the right ends of A and B in $O(n)$ and $O(m)$ time, respectively. For LCS scores, Tiskin [34, Chapter 5.3] utilized the property that the all scores matrix $\mathcal{L}$ is unit-Monge. His algorithm solves the 4-IASM problem on $\mathcal{L}$. The algorithm uses $O(m+n)$ space and supports incremental operations in either $O(n)$ or $O(m)$



(a) The all substring-string LCS problem 如：S-table



(b) The all prefix-suffix LCS problem

Figure 1: The alignment graph for the strings $A = ccbccaa$ and $B = cacab$. Figure (a) illustrates a prepend operation of character c to A. The matrix $\mathcal{K}$ contains the optimal scores of paths from every vertex on the left border to every vertex on the right border (these vertices are colored gray). The optimal path that corresponds to $\mathcal{K}[2, 4]$ is marked using thicker edges. Figure (b) illustrates an append operation of character b to B. The matrix $\mathcal{J}$ contains the optimal scores of paths from every vertex on the left border to every vertex on the bottom border (these vertices are colored gray).

time. Restricting the set of supported incremental operations and maintaining either the matrix $\mathcal{J}$ or $\mathcal{K}$ rather than the full matrix $\mathcal{L}$ allows for faster algorithms. Such an algorithm was given in Landau et al. [22]. The algorithm of Landau et al. allows prepend operations to the string A while maintaining the alignment score between every suffix of B to every prefix of A, thus it solves 1-IASM on the matrix $\mathcal{J}$. Amortized on m incremental operations, the algorithm runs in $O(L)$ time per operation, where L is the length of the longest common subsequence of A and B. The space complexity of the algorithm is $O(n)$.

1.1 Our Contribution

In this paper we study the k-IASM problem under LCS score. We exploit the sparsity parameters L and $\Delta = n - L$, to design faster algorithms. These parameters have already been extensively used to give faster LCS algorithms [3, 4, 13, 29, 37]. In order to obtain faster running times, we restrict the set of supported incremental operations (namely k is either 1 or 2), and we also maintain either $\mathcal{J}$ or $\mathcal{K}$ and not the full matrix $\mathcal{L}$.

We note that various applications that utilize incremental all scores matrices require a specific type of all scores matrix, and also use restricted sets of update operations (see for example [14, 15, 19, 22, 24, 25, 26, 30, 32, 36]). Thus, our algorithms are advantageous to these applications.

Our results are as follows (see also Table 1). In Section 3 we give an algorithm for maintaining the matrix $\mathcal{K}$ while supporting incremental operations to both sides of string A in $O(\Delta)$ time, using $O(n)$ space. In Section 4 we give an algorithm for maintaining the matrix $\mathcal{J}$ and supporting one type of update operations: either prepending a character to A or appending a character to B. The algorithm uses

	Problem	Supported operations	Time	Space
Landau et al. [22]	1-sided $\mathcal{J}$	Prepend a character to A	$O(L)$ amortized	$O(n)$
Schmidt [32]	2-sided $\mathcal{L}$	Append a character to A or B	$O(n)/O(m)$	$O(m+n)$
Tiskin [34]	4-sided $\mathcal{L}$	Prepend/append a character to A or B	$O(n)/O(m)$	$O(m+n)$
Ours (Thm. 3.8)	2-sided $\mathcal{K}$	Prepend and append a character to A	$O(\Delta)$	$O(n)$
Ours (Thm. 4.7)	1-sided $\mathcal{J}$	Prepend a character to A, or append a character to B	$O(L)$	$O(n)$ $O(m)$
Ours (Thm. 4.8)	2-sided $\mathcal{J}$	Prepend a character to A, and append a character to B	$O(L \log\log L)$	$O(m+n)$
Ours (Thm. 5.3)	2-sided $\mathcal{K}, \mathcal{J}$	Append a character to A or B	$O(\Delta)/O(L)$	$O(m+n)$

Table 1: Comparison of known and new results for k-IASM. A and B denote two strings of lengths m and n, respectively, over a constant alphabet. In this setting, $L = LCS(A, B)$ and $\Delta = n - L$.

either $O(n)$ or $O(m)$ space, and the worst-case time complexity of an update operation is $O(L)$. We also show how to support both update operations simultaneously. This increases the space complexity to $O(m+n)$ and the worst-case time complexity of an update operation to $O(L \log\log L)$. Our result improves the result of Landau et al. [22] since the update time in Landau et al. is $O(L)$ amortized. Additionally, the proof of correctness of our algorithm is simpler. Finally, in Section 5, we give an algorithm for maintaining both matrices $\mathcal{K}$ and $\mathcal{J}$ while supporting append operations to either string. Here, the space complexity is $O(m + n)$ and the update time is $O(\Delta)$ or $O(L)$.

We note that our proposed approach could also be applied to yield a solution to the 4-sided problem with the same time and space complexity as the algorithm of Tiskin [34].

2 Preliminaries

We denote $[i : j] = \{i, i + 1, \dots, j\}$. Let A, B be two strings of lengths m, n, respectively, over an alphabet Σ of constant size. Denote by $G_{A,B}$ the alignment graph of A and B. $G_{A,B}$ is a grid graph over a vertex set $[0 : n] \times [0 : m]$, where all vertical and horizontal edges are present, and a diagonal edge between $(i-1, j-1)$ to (i, j) is present if and only if $B[i] = A[j]$, in which case we say that (i, j) is a *match point* in $G_{A,B}$. Diagonal edges have score 1, and horizontal and vertical edges have a score 0. We denote $L = LCS(A, B)$ and $\Delta = n - L$.

For a string $S = \sigma_1\sigma_2 \cdots \sigma_n$, let $S[i..j] = \sigma_{i+1} \cdots \sigma_j$ denote the substring of S from $i + 1$ to j. Consequently, $S[i..i]$ denotes the empty string, for which we use the symbol ϵ. For a character σ and a string S, let $\text{NextMatch}_S(i, \sigma)$ denote the minimum index $i' > i$, such that $\sigma = S[i']$ (and ∞ if no such i' exists), and let $\text{PrevMatch}_S(j, \sigma)$ denote the maximum index $j' \leq j$, such that $\sigma = S[j']$ (and $-\infty$ if no such j' exists).

We consider two types of all scores matrices (see Figure 1). Let $\mathcal{J}$ denote the matrix containing the scores of the optimal paths between every vertex on the left border of $G_{A,B}$ to every vertex on its bottom border, that is, $\mathcal{J}[i, j] = LCS(B[i..n], A[0..j])$. Also, let $\mathcal{K}$ denote the matrix containing the scores of the optimal paths between every vertex on the left border of $G_{A,B}$ to every vertex on its right border, namely $\mathcal{K}[i,j] = LCS(B[i..j], A)$. If $i > j$ then no such path exists and we define $\mathcal{K}[i,j] = j - i$.

For a matrix $\mathcal{D}$ over index set $[0 : n] \times [0 : m]$, we define its *density matrix*, denoted by $\mathcal{D}^{\Box}$, to be a matrix over the index set $[1 : n] \times [1 : m]$ such that Monge $\mathcal{D}^{\Box}[i,j] = (\mathcal{D}[i,j] + \mathcal{D}[i-1,j-1]) - (\mathcal{D}[i-1,j] + \mathcal{D}[i,j-1])$. The next property follows immediately from the above definition. (i−1, j−1) (i−1, j) 斜角−反斜角
(i. , j−1) (i , j)

Proposition 2.1. $\mathcal{D}[i,j] = \sum_{\substack{1 \le i' \le i \\ 1 \le j' \le j}} \mathcal{D}^{\Box}[i',j'] - \mathcal{D}[0,0] + \mathcal{D}[0,j] + \mathcal{D}[i,0]$, for every $0 \le i \le n$ and $0 \le j \le m$.

We say that a matrix is *sub-unit Monge* (resp., *sub-unit anti-Monge*), if every row and column of its density matrix contains at most one non-zero element, and all the non-zero elements are equal to 1 (resp., -1). It is well established that the matrices $\mathcal{K}$ and $\mathcal{J}$ are sub-unit Monge and sub-unit anti-Monge, respectively (cf. [35]). This implies the next corollary regarding the size of the encoding of $\mathcal{K}$ and $\mathcal{J}$.

Corollary 2.2. The density matrix of $\mathcal{J}$ has exactly L non-zero elements, and the density matrix of $\mathcal{K}$ has exactly Δ non-zero elements.

Proof. For the first part, note that $\mathcal{J}[n,m] = LCS(B[n..n], A) = 0$. However, by Proposition 2.1, $\mathcal{J}[n,m] = \sum_{\substack{1 \le i' \le n \\ 1 \le j' \le m}} \mathcal{J}^{\Box}[i',j'] - \mathcal{J}[0,0] + \mathcal{J}[0,m] + \mathcal{J}[n,0]$. Hence, $\mathcal{J}[0,m] + \mathcal{J}[n,0] - \mathcal{J}[0,0] = -\sum_{\substack{1 \le i' \le n \\ 1 \le j' \le m}} \mathcal{J}^{\Box}[i',j']$. By the definition of the matrix $\mathcal{J}$, $\mathcal{J}[0,0] = LCS(B,\epsilon) = 0$, $\mathcal{J}[0,m] = LCS(B,A) = L$ and $\mathcal{J}[n,0] = LCS(\epsilon,\epsilon) = 0$. Therefore, $\sum_{\substack{1 \le i' \le n \\ 1 \le j' \le m}} -\mathcal{J}^{\Box}[i',j'] = L$. The second part of the corollary can be obtained similarly, by the definition of the matrix $\mathcal{K}$. $\Box$

In what follows the non-zero cells of a density matrix are called *pivotal points*. We encode an all scores matrix by storing the pivotal points of its density matrix. This requires $O(L)$ space for the $\mathcal{J}$ matrix, and $O(\Delta)$ space for $\mathcal{K}$ (by Corollary 2.2). In the subsequent sections we handle the two matrices $\mathcal{K}$ and $\mathcal{J}$ separately, and for each matrix we consider both the 1-sided and the 2-sided problems.

3 Incremental $\mathcal{K}$ matrix 左右兩邊A對B的所有子字串

Recall that we need to maintain the all scores matrix, denoted $\mathcal{K}$, of the strings A and B. Our algorithm encodes the matrix $\mathcal{K}$ by storing the pivotal points of its density matrix. Consider an incremental operation that prepends a character σ to A, that is $A' = \sigma A$. We need to compute the pivotal points of $\mathcal{K}'^{\Box}$, where $\mathcal{K}'$ is the all scores matrix of A' and B. Define $C = \mathcal{K}' - \mathcal{K}$. By definition, $\mathcal{K}'^{\Box} = \mathcal{K}^{\Box} + C^{\Box}$. We will next show how to compute the pivotal points of $C^{\Box}$. This will allow the algorithm to compute the pivotal points of $\mathcal{K}'^{\Box}$. C contains either 0 or 1 values, since the new character σ can increase the length of the LCS of each alignment by at most 1. See Figure 2 for an example of matrices $\mathcal{K}', \mathcal{K}, C$ and $C^{\square}$.

The optimal path from the i'th vertex on the first column to the j'th vertex on the last column of $G_{\sigma A,B}$ either utilizes a match point that was generated due to the new prepended character σ, or not. In the former case we can assume the optimal path uses the topmost such match point, and in the latter case the score of this path is equal to $\mathcal{K}[i,j]$. Hence, we have $\mathcal{K}'[i,j] = \max\{\mathcal{K}[\text{NextMatch}_B(i,\sigma), j]+1, \mathcal{K}[i,j]\}$. This leads to the following proposition.

Proposition 3.1. $\mathcal{K}'[i,j] = \mathcal{K}[i,j]+1$ if and only if $\text{NextMatch}_B(i,\sigma) < \infty$ and $\mathcal{K}[i,j] = \mathcal{K}[\text{NextMatch}_B(i,\sigma), j]$.
存在

We now describe our main lemma required for the incremental step.
存在

Lemma 3.2. $C[i,j] = 1$ if and only if $\text{NextMatch}_B(i,\sigma) < \infty$ and every row of $\mathcal{K}^{\square}[i+1..\text{NextMatch}_B(i,\sigma), 1..j]$ has exactly one pivotal point.

Proof. Let $\text{NextMatch}_B(i,\sigma) = k$. By Proposition 2.1, $\mathcal{K}[i,j] = \mathcal{K}[k,j]$ if and only if:

$$\sum_{\substack{1\le i'\le i\\1\le j'\le j}} \mathcal{K}^{\square}[i',j'] - \mathcal{K}[0,0] + \mathcal{K}[0,j] + \mathcal{K}[i,0] = \sum_{\substack{1\le i'\le k\\1\le j'\le j}} \mathcal{K}^{\square}[i',j'] - \mathcal{K}[0,0] + \mathcal{K}[0,j] + \mathcal{K}[k,0].$$

After canceling the terms that appear in both sides, we obtain the equality

移項化簡
$$\sum_{\substack{i+1\le i'\le k\\1\le j'\le j}} \mathcal{K}^{\square}[i',j'] = \mathcal{K}[i,0] - \mathcal{K}[k,0].$$

By the definition of the matrix $\mathcal{K}$, $\mathcal{K}[i,0] = -i$ and $\mathcal{K}[k,0] = -k$, thus obtaining

$$\sum_{\substack{i+1\le i'\le k\\1\le j'\le j}} \mathcal{K}^{\square}[i',j'] = k - i.$$

Recall that $\mathcal{K}$ is a sub-unit Monge matrix and hence there is at most one pivotal point in every row of $\mathcal{K}^{\square}$. Therefore, $\sum_{\substack{i+1\le i'\le k\\1\le j'\le j}} \mathcal{K}^{\square}[i',j'] = k-i$ if and only if every row of $\mathcal{K}^{\square}[i+1..\text{NextMatch}_B(i,\sigma), 1..j]$ has exactly one pivotal point. The lemma now follows from Proposition 3.1. □

The following corollary follows immediatly from Lemma 3.2.

Corollary 3.3. If $C[i,j] = 1$ then $C[i,j'] = 1$ for every $j' \ge j$. 自己為1，則右邊皆為1

We say that (i,j) is a *step index* in C if $C[i,j] \ne C[i,j-1]$ (see Figure 2). The following lemma shows that the pivotal points of $C^{\square}$ can be obtained from the step indices of C. Therefore, the computation of these indices will be the focus of our algorithm.
自己為1且左邊為0，則自己是step

Lemma 3.4. For every cell (i,j) in $C^{\square}$,

0	1	2	2	2	3	4	4	4	4
-1	0	1	2	2	3	4	4	4	4
-2	-1	0	1	2	3	4	4	4	4
-3	-2	-1	0	1	2	3	3	3	3
-4	-3	-2	-1	0	1	2	3	3	3
-5	-4	-3	-2	-1	0	1	2	2	2
-6	-5	-4	-3	-2	-1	0	1	2	2
-7	-6	-5	-4	-3	-2	-1	0	1	2
-8	-7	-6	-5	-4	-3	-2	-1	0	1
-9	-8	-7	-6	-5	-4	-3	-2	-1	0

(a) $\mathcal{K}'$

		c	c	a	b	a	c	c	a	a
	0	1	2	2	2	2	3	3	3	3
c	-1	0	1	2	2	2	3	3	3	3
c	-2	-1	0	1	1	2	3	3	3	3
a	-3	-2	-1	0	1	2	3	3	3	3
b	-4	-3	-2	-1	0	1	2	2	2	2
a	-5	-4	-3	-2	-1	0	1	2	2	2
c	-6	-5	-4	-3	-2	-1	0	1	2	2
c	-7	-6	-5	-4	-3	-2	-1	0	1	1
a	-8	-7	-6	-5	-4	-3	-2	-1	0	1
a	-9	-8	-7	-6	-5	-4	-3	-2	-1	0

(b) $\mathcal{K}$

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	1	1	1	1	1
0	0	0	0	0	1	1	1	1	1
0	0	0	0	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

(c) C =K' – K

0	0	0	0	0	0	0	0	0
0	0	0	1	-1	0	0	0	0
0	0	0	-1	0	0	0	0	0
0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	-1	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	-1
0	0	0	0	0	0	0	0	0

(d) $C^{\square}$

Figure 2: An example of the matrices $\mathcal{K}, \mathcal{K}', C$ and $C^{\square}$ for $A = bbcac$, $B = ccabaccaa$, and a prepend of the character a to A. Pivotal points are colored dark gray and step indices in C are colored light gray.

自己step且上面不是step

- $C^{\square}[i,j] = 1$ if and only if (i,j) is a step index in C and $(i-1,j)$ is not a step index.
- $C^{\square}[i,j] = -1$ if and only if (i,j) is not a step index in C and $(i-1,j)$ is a step index.

Proof. Let $\bar{C}$ be a matrix containing the columns differences of the matrix C, that is, $\bar{C}[i,j] = C[i,j] - C[i,j-1]$. 自己-左邊 From Corollary 3.3 we have that $\bar{C}[i,j] = 1$ if and only if (i,j) is a step index in C (note that if (i,j) is a step index then $j > 0$). The lemma follows due to the equality $C^{\square}[i,j] = \bar{C}[i,j] - \bar{C}[i-1,j]$. □

自己step且上面不是step = 1 - 0 = 1

Lemma 3.2 gives the following corollary.

Corollary 3.5. (i,j) is a step index in C if and only if j is equal to the maximum column index among all pivotal points in rows $i+1,\ldots,\mathrm{NextMatch}_B(i,\sigma)$ of the matrix $\mathcal{K}^{\square}$.

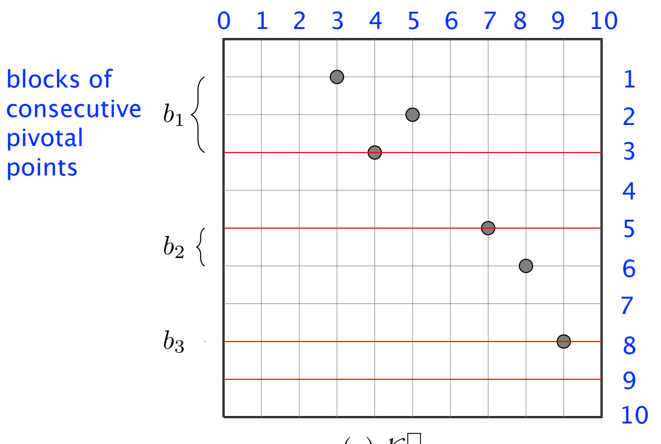
3.1 1-sided incremental $\mathcal{K}$ matrix

We now describe our algorithm that supports 1-sided operations to the string A. We consider the incremental operation of prepending a character σ to A (an appending operation to A can be carried out in the same manner). In our proposed solution, we assume that there is an auxilary data structure that encodes the string B and allows access operations to any position in B in constant time. We do not take into account the space occupied by this data structure in the space complexity of the algorithm. This leads to an $O(\Delta)$ space data structure that supports prepend (or append) operations to the string A in $O(\Delta)$ time.

P : pivotal

Recall that the all scores matrix $\mathcal{K}$ is encoded via the pivotal points of its density matrix. These points are stored in a list P, sorted by increasing row indices. Our algorithm computes the step indices of C, from which the pivotal points of $C^{\square}$, and hence the pivotal points of $\mathcal{K}'^{\square}$, can be obtained (Lemma 3.4). Note that the number of step indices in C is bounded by Δ. This is due to the fact that each such step index corresponds to a unique pivotal point of $\mathcal{K}^{\square}$ (Lemma 3.2).

The step indices are computed as follows (see Figure 3). Denote by i_1+1 the minimum row index of a pivotal point of $\mathcal{K}$ (the value of i_1+1 is obtained by accessing the first element of P) and let $k_1 = \mathrm{NextMatch}_B(i_1,\sigma)$. Note that by Lemma 3.2, there is a step index in the i_1'th row of C only if there are $k_1 - i_1$ consecutive pivotal points in P with row indices i_1+1 to k_1. Therefore we scan simultaneously the list P and the string B starting from index $i = i_1+1$. At each step we check whether the current examined element in P is a pivotal point in row i, and if it is, we move to the next element of P and increase i by 1. This scan is stopped when reaching the index k_1 for which $B[k_1] = \sigma$. If such k_1 is found then by Lemma 3.2, every row from rows $i_1,\ldots,k_1-1$ contains a step index. In order to compute these step indices, go over $i = k_1-1, k_1-2,\ldots,i_1$. For each such i, compute the maximum column index j_i among all pivotal points in rows $i+1,\ldots,k_1$. j_i can be computed in constant time since j_i is equal to the maximum of j_{i+1} and the column of the pivotal point of row $i+1$ (this column is obtain from P, which is scanned in reverse order, K-> $\boxed{K}$ -> C(step) -> $\boxed{C}$



(a) $\mathcal{K}^{\square}$

A = bbcac, B = ccabaccaa

i	$\text{NextMatch}_B(i,a)$
0	3
1	3
2	3
3	5
4	5
5	8
6	8
7	8
8	9
9	∞

$P = \{(1,3),(2,5),(3,4),(5,7),(6,8),(8,9)\}$

P : pivotal

Figure 3: A run of the algorithm on the strings A and B of Figure 2, and a prepend of the character a to A. Figure (a) shows the pivotal points of $\mathcal{K}^{\square}$. The red lines correspond to the different values of $\text{NextMatch}_B(i,a)$. b_1, b_2 and b_3 denote blocks of consecutive pivotal points as defined in Section 3.2. The algorithm begins with the pivotal point at row $i_1 + 1 = 1$. It then scans for the next match, denoted by a red line at index $k_1 = 3$, and verifying at each step that there is a pivotal point in every row. Once reaching index 3, the algorithm backward scans the previously traversed pivotal points starting with the pivotal point at row 3. At each step the algorithm computes the maximum column index among all scanned pivotal points. The first column maxima is $j_2 = 4$, hence $(2,4)$ is a step index in C. The following pivotal point is $(2,5)$, thus the column maxima is $j_1 = 5$, and $(1,5)$ is identified as a step index in C. The last scanned pivotal point is $(1,3)$, thus the column maxima remains 5 and the step index identified is $(0,5)$. The next scanned block is b_2 starting with pivotal point $(5,7)$. The next match point is 5, and thus $(4,7)$ is a step index. Pivotal point $(6,8)$ does not yield a step index since there is no match point at index 6 and also there is no pivotal point at row 7, hence the search stops after the first iteration. The last pivotal point is handled similarly.

黄色是step indices in C

step : (0, 5) (1,5) (2,4) | (4,7) | (7,9) starting from the element holding the pivotal point of row k_1). By Corollary 3.5, (i, j_i) is the step index of row i.

The algorithm processes the remaining pivotal points (starting from the topmost pivotal point below row k_i) similarly, until exhausting all Δ pivotal points.

K-> $\boxed{K}$ -> C(step) -> $\boxed{C}$

Complexity analysis To obtain the set of step indices, we traverse the list P of pivotal points, and examine each pivotal point at most twice (once during a forward scan on P and once during a backward scan on P). We also scan the string B. The number of access operation to B is bounded by the number of pivotal points.

Once the set of step indices has been obtained, the computation of $C^{\square}$ (using Lemma 3.4) and the pivotal points of $\mathcal{K}'$ is done in $O(\Delta)$ time. Hence the total running time is $O(\Delta)$ and the space complexity is $O(\Delta)$.

Appending a character to B Appending character σ to A follows the same paradigm. Note that now $\mathcal{K}'[i,j] = \max\{\mathcal{K}[i, \text{PrevMatch}_B(j,\sigma) - 1] + 1, \mathcal{K}[i,j]\}$.

The next lemma summarizes the properties required to carry out an append operation to A.

Lemma 3.6. Let $k = \text{PrevMatch}_B(j,\sigma)$. $C[i,j] = 1$ if and only if there is one pivotal point in each column from columns $k, \ldots, j$ of the matrix $\mathcal{K}^{\square}$, and i is less than the minimum row index among all pivotal points in the submatrix $\mathcal{K}^{\square}[1..n, k..j]$.

Proof. Following the same steps as in Lemma 3.2 we obtain that $C[i,j] = 1$ if and only if:

$$\sum_{\substack{1 \le i' \le i \\ k \le j' \le j}} \mathcal{K}^{\square}[i', j'] = \mathcal{K}[0, k-1] - \mathcal{K}[0, j]. \tag{1}$$

Note that $\sum_{\substack{1 \le i' \le i \\ k \le j' \le j}} \mathcal{K}^{\square}[i', j'] \ge 0$ and $\mathcal{K}[0, k-1] \le \mathcal{K}[0, j]$. Hence, Equation (1) holds if and only if, (i) $\mathcal{K}[0, k-1] = \mathcal{K}[0, j]$, and (ii) $\sum_{\substack{1 \le i' \le i \\ k \le j' \le j}} \mathcal{K}^{\square}[i', j'] = 0$. The first item holds if and only if $\sum_{\substack{1 \le i' \le n \\ k \le j' \le j}} \mathcal{K}^{\square}[i', j'] = j - k + 1$, that is, there is a pivotal point in every column from columns $k, \ldots, j$. The second item holds if and only if i is less than the minimum row index among all pivotal points in the submatrix $\mathcal{K}^{\square}[1..n, k..j]$. $\square$

Due to Lemma 3.6 we need to use a different definition of step indices in this case. We say that (i,j) is a *column step index* if $C[i,j] \ne C[i-1,j]$. Hence, the main difference to the algorithm is that now we need to store a list P' containing the pivotal points of $\mathcal{K}^{\square}$ sorted by decreasing columns indices.

We obtain the following theorem.

Theorem 3.7. Given strings A and B of lengths m and n, respectively, over alphabet Σ with $L = LCS(A,B)$ and $\Delta = n - L$. We can construct an $O(\Delta)$ space data structure that encodes the LCS score between A and every substring of B and supports 1-sided incremental operations to A in $O(\Delta)$ time.

3.2 2-sided incremental $\mathcal{K}$ matrix

In this section we show how to extend the previously discussed data structure to support both prepend and append operations to A. This extension requires $O(n)$ space and $O(\Delta)$ time per operation.

In order to support incremental operations for both sides of the string A, one possible solution is to sort the elements of P by their columns indices to obtain a list P' when an append operation is performed. This sorting takes $O(\Delta \log\log \Delta)$ time using the sorting algorithm of [12]. To obtain $O(\Delta)$ worst-case running time we use the observation that we don't need the lists P and P' to be sorted, and instead we will use a weaker requirement on the order of the elements.

We say that $[j_1 : j_2]$ is a *column block* if there is a pivotal point in column j for every $j \in [j_1 : j_2]$, and there are no pivotal points in column $j_1 - 1$ and column $j_2 + 1$. We now store P' in the following order. For every column block $[j_1 : j_2]$, the pivotal points in columns $[j_1 : j_2]$ appear consecutively in P', and ordered by decreasing column indices (namely, the pivotal point in column j_2 appears first, then the pivotal point in column $j_2 - 1$ an so on). This corresponds to the sorting required for the appending operation to A as described in Section 3.1. Note that we do not require a specific order between pivotal points of different blocks. We define *row block* analogously and we store P ordered according to the row blocks. It is easy to verify that the algorithm described in Section 3.1 remains correct when P and P' are stored in block order.

We now describe how to construct the list P' from P. The process requires an auxiliary array BA of size n, in which each cell is initialized with 0. Let $(i_1, j_1), \ldots, (i_\Delta, j_\Delta)$ be the pivotal points. We traverse the set of pivotal points as stored in P and set $BA[j_t] \leftarrow i_t$ for every $1 \le t \le \Delta$. Note that after the step above, every column block corresponds to a maximal sub-array of BA with non-zero elements. We next traverse again the set of pivotal points in order to extract the column blocks. For each pivotal point (i_t, j_t) we examine the value $BA[j_t]$. If $BA[j_t] \neq 0$, we scan the array BA starting at index j_t to find the minimum index $j' > j_t$ such that $BA[j'] = 0$. Then, for $j = j' - 1, j' - 2, \ldots$ we add the pivotal point $(BA[j], j)$ to the end of P', and set $BA[j]$ to 0. This loop is stopped when $BA[j] = 0$. We then move to the next pivotal point (i_{t+1}, j_{t+1}). The running time of this algorithm is $O(\Delta)$.

We obtain the following theorem.

Theorem 3.8. Given strings A and B of lengths m and n, respectively, over alphabet Σ with $L = LCS(A, B)$ and $\Delta = n - L$. We can construct an $O(n)$ space data structure that encodes the LCS score between A and every substring of B and supports 2-sided incremental operations to both ends of A in $O(\Delta)$ time.

We note that our approach cannot support prepend or append operations on B. The reason is that such operations increase the size of the matrix $\mathcal{K}$. The new column in $\mathcal{K}^\square$ may contain a new pivotal point. However, the matrix $\mathcal{K}^\square$ does not have the information required for computing this new pivotal point.

4 Incremental $\mathcal{J}$ matrix 左邊對下面，A前綴對B後綴

Our method introduced above can be modified to maintain the matrix $\mathcal{J}$ and supports prepend operations to A and append operations to B. Note that the size of the matrix $\mathcal{J}$ is $n \times m$, thus it increases when such operations are performed. We start by considering an incremental operation that prepends a character σ to A, that is $A' = \sigma A$. We assume that $\mathcal{J}$ is over the index set $[0:n] \times [0:m]$ and $\mathcal{J}'$ is over the index set $[0:n] \times [-1:m]$. We define $C[i,j] = \mathcal{J}'[i,j] - \mathcal{J}[i,j]$ for $j \geq 0$, and $C[i,j] = 0$ otherwise. J會隨著A前加或B後加增長

In this case we have that $\mathcal{J}'[i,j] = \max\{\mathcal{J}[i,j], \mathcal{J}[\text{NextMatch}_B(i,\sigma), j] + 1\}$ for $j \geq 0$. The following lemma is the analogous of Lemma 3.2.

存在

Lemma 4.1. For $j \geq 0$, $C[i,j] = 1$ if and only if $\text{NextMatch}_B(i,\sigma) < \infty$ and there are no pivotal points in the submatrix $\mathcal{J}^\Box[i+1..\text{NextMatch}_B(i,\sigma), 1..j]$.

Proof. Let $\text{NextMatch}_B(i,\sigma) = k$. Similarly to Lemma 3.2, $C[i,j] = 1$ if and only if $\sum_{\substack{i+1 \leq i' \leq k \\ 1 \leq j' \leq j}} \mathcal{J}^\Box[i',j'] = \mathcal{J}[i,0] - \mathcal{J}[k,0]$. The lemma follows due to the definition of $\mathcal{J}$, since $\mathcal{J}[i,0] = \mathcal{J}[k,0] = 0$. $\square$

This leads to the following corollary (analogous to Corollary 3.3).

Corollary 4.2. If $C[i,j] = 1$ then $C[i,j'] = 1$ for every $0 \leq j' \leq j$. 自己為1，則左邊也為1

Following Corollary 4.2 we say that (i,j) is a *step index* if $j \geq 1$ and $C[i,j] \neq C[i,j-1]$ (see Figure 4). We then obtain the following corollary (analogous to Corollary 3.5).

在J中的step為，自己和上面不一樣

存在

Corollary 4.3. (i,j) is a step index in C if and only if $\text{NextMatch}_B(i,\sigma) < \infty$ and j is equal to the minimum column index among all pivotal points in rows $i+1,\ldots,\text{NextMatch}_B(i,\sigma)$ of the matrix $\mathcal{J}^\Box$. 這裡是最小值喔喔

Lemma 4.1 introduces several challenges compared to Section 3. First, the number of step indices is not bounded by the number of pivotal points (see for example Figure 4). Consequently, the number of step indices may be $\Omega(n)$. Thus, to obtain $O(L)$ running time, we show in Lemma 4.6 how to compute $C^\Box$ directly, without computing the entire set of step indices explicitly. Furthermore, the property of consecutive pivotal points, which was exploited in Section 3 to bound the time complexity of scanning for the next character matching to σ, no longer holds. Thus, we need to use a data structure that supports $\text{NextMatch}_B(i,\sigma)$ or $\text{PrevMatch}_B(j,\sigma)$ queries in constant time.

Following Lemma 4.1 we have that $C[i,0] = 1$ for every $0 \leq i \leq n$ such that $\text{NextMatch}_B(i,\sigma) < \infty$. Hence, if $C[i,0] = 1$ 自己 then $C[i',0] = 1$ 上面 for every $0 \leq i' \leq i$. We get that the maximal row index i for which $C[i,0] = 1$ yields a pivotal point in cell $C^\Box[i+1,0]$ of value -1 (see for example $C^\Box[7,0]$ in Figure 4 (d)). This yields the following corollary.

Corollary 4.4. $C^\Box[i,0] = -1$ for the minimum index i for which $\text{NextMatch}_B(i,\sigma) = \infty$, and this is the only pivotal point in the 0'th column of $C^\Box$. 不存在

From the definition of step indices we obtain the following lemma.

Lemma 4.5. For every cell (i, j) with $j > 0$ in $C^\square$,

- $C^\square[i, j] = -1$ if and only if (i, j) is a step index in C and $(i - 1, j)$ is not a step index. C=-1 <-> 自己是step且上面不是step
- $C^\square[i, j] = 1$ if and only if (i, j) is not a step index in C and $(i - 1, j)$ is a step index.

We would like to identify cases (i) and (ii) described above, to be able to compute the pivotal points of $C^\square$ directly.

Lemma 4.6. Let $j_{\min}$ be the minimum column index among the pivotal points in rows $i + 1, \ldots, \text{NextMatch}_B(i, \sigma)$. If there are no such pivotal points, $j_{\min} = \infty$. In what follows we consider only cells of $C^\square$ with column indices greater than 0.

- If $B[i] \neq \sigma$ then row i of $C^\square$ contains non zero elements if and only if $\text{NextMatch}_B(i, \sigma) < \infty$ and there is a pivotal point (i, j) in $\mathcal{J}^\square$ with $j < j_{\min}$. If these conditions hold, $C^\square[i, j] = 1$. Additionally, if $j_{\min} < \infty$, $C^\square[i, j_{\min}] = -1$. All other cells at row i of $C^\square$ contain zeros.
- If $B[i] = \sigma$ then $C^\square[i, j_{\min}] = -1$ if $\text{NextMatch}_B(i, \sigma) < \infty$ and $j_{\min} < \infty$. Additionally, if there is a pivotal point (i, j) in $\mathcal{J}^\square$ then $C^\square[i, j] = 1$. All other cells at row i of $C^\square$ contain zeros.

Proof. Row i of $C^\square$ contains non zero element if and only if there is a cell in this row for which the first or the second case of Lemma 4.5 holds.

1 For the first part of the lemma, since $B[i] \neq \sigma$, we get $\text{NextMatch}_B(i - 1, \sigma) = \text{NextMatch}_B(i, \sigma)$. If $\text{NextMatch}_B(i, \sigma) = \infty$ 不存在 then also $\text{NextMatch}_B(i - 1, \sigma) = \infty$ 不存在 and by Corollary 4.3 rows $i - 1$ and i do not have step indices. Therefore, by Lemma 4.5 row i does not contain non zero elements. Assume now that $\text{NextMatch}_B(i, \sigma) <$ 存在 ∞. If there is no pivotal point (i, j) with $j < j_{\min}$ then by Corollary 4.3, both rows $i - 1$ and i have step indices at column $j_{\min}$ if $j_{\min} < \infty$ 存在, and these rows do not have step indices if $j_{\min} = \infty$ 不存在. Hence, neither case of Lemma 4.5 can occur, so row i does not contain non zero elements.

Suppose now that there is a pivotal point (i, j) with $j < j_{\min}$. By Corollary 4.3, (i, j) is not a step index in C and $(i-1, j)$ is a step index. By Lemma 4.5, $C^\square[i, j] = 1$. Moreover, if $j_{\min} < \infty$ then $(i, j_{\min})$ is a step index in C and $(i-1, j_{\min})$ is not a step index, so $C^\square[i, j_{\min}] = -1$. Finally, if $j_{\min} = \infty$ then row i of C does not contain a step index and the first case of Lemma 4.5 cannot occur.

2 For the second part of the lemma, since $B[i] = \sigma$, we get $\text{NextMatch}_B(i-1, \sigma) = i$. Suppose that $\text{NextMatch}_B(i, \sigma) < \infty$ 存在 and $j_{\min} < \infty$ 存在. By Corollary 4.3, $(i, j_{\min})$ is a step index in C and $(i - 1, j_{\min})$ is not a step index (since the rows range $(i - 1) + 1, \ldots, \text{NextMatch}_B(i - 1, \sigma)$ consists of only row i, and row i cannot have a pivotal point in column $j_{\min}$). By Lemma 4.5, $C^\square[i, j_{\min}] = -1$. Additionally, if there is a pivotal point (i, j) in $\mathcal{J}^\square$ then (i, j) is not a step index in C and $(i - 1, j)$ is a a step index. Therefore, $C^\square[i, j] = 1$. □

0	1	2	3	4	5	6	7	8
0	1	2	3	3	3	4	5	5
0	1	2	3	3	3	4	5	5
0	1	2	3	3	3	3	4	4
0	1	2	3	3	3	3	4	4
0	1	1	2	2	2	2	3	3
0	1	1	1	1	1	1	2	2
0	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0

(a) $\mathcal{J}'$

		b	b	c	b	b	a	a
	0	1	2	2	2	3	4	4
a	0	1	2	2	2	3	4	4
a	0	1	2	2	2	3	4	4
b	0	1	2	2	2	2	3	3
a	0	1	2	2	2	2	3	3
b	0	1	1	1	1	1	2	2
b	0	0	0	0	0	0	1	1
a	0	0	0	0	0	0	0	0

(b) $\mathcal{J}$

	0	1	2	3	4	5	6	7	8
	0	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1
	0	1	1	1	1	1	0	0	0
	0	1	1	1	1	1	1	1	1
	0	1	0	0	0	0	0	0	0
	0	1	0	0	0	0	0	0	0
	0	1	1	1	1	1	1	0	0
	0	0	0	0	0	0	0	0	0

(c) C

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	-1	0	0
3	0	0	0	0	0	1	0	0
4	0	-1	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	1	0	0	0	0	-1	0
7	-1	0	0	0	0	0	1	0

(d) $C^{\square}$

Figure 4: An example of the matrices $\mathcal{J}', \mathcal{J}, C$ and $C^{\square}$ for $A = bbcbbaa$, $B = aacabba$, and a predend of the character a to A. Pivotal points are colored dark gray and step indices in C are colored light gray.

4.1 1-sided incremental $\mathcal{J}$ matrix

In this section we encode the matrix $\mathcal{J}$ such that prepend operations to the string A are supported. We encode the matrix $\mathcal{J}$ by the L pivotal points of its density matrix. These points are stored in a list P sorted by increasing row indices. Note that the string B remains constant, hence we can preprocess B in $O(n)$ time and compute two static look-up tables that contain all possible values of $\mathrm{NextMatch}_B(i,\sigma)$ and $\mathrm{PrevMatch}_B(i,\sigma)$. We then use Lemma 4.6 to compute $C^{\square}$ efficiently in $O(L)$ time, see Figure 4.1 for a running example.

P : pivotal

We begin iterating over P. Let (i_1+1, j_1) denote the first element of P. We compute $k_1^- = \mathrm{PrevMatch}_B(i_1,\sigma)$, and $k_1^+ = \mathrm{NextMatch}_B(i_1,\sigma)$. k_1^- and k_1^+ are the starting and ending indices of a block of pivotal points (in contrast to Section 3.2, here the pivotal points of a block do not necessarily have consecutive row indices).

這裡塊的關鍵點不一定具有連續的行索引

We scan the list P until reaching a pivotal point (i_2+1, j_2) for which $i_2+1 > k_1^+$ (forward scan). We then scan backward the block starting from the pivotal point preceding (i_2+1, j_2), and at each step we compute the minimum column index in the block so far and apply Lemma 4.6 to obtain the pivotal points of $C^{\square}$.

The algorithm processes the remaining pivotal points (starting from the pivotal point (i_2+1, j_2)) similarly, until exhausting all L pivotal points. At the end we compute $\mathrm{PrevMatch}_B(n,\sigma)$ to obtain the pivotal point at column 0 by Corollary 4.4.

Complexity analysis The preprocessing step takes $O(n)$ time and space. At each step of the algorithm we examine each pivotal point at most twice, hence we obtain $O(L)$ time and $O(n)$ space.

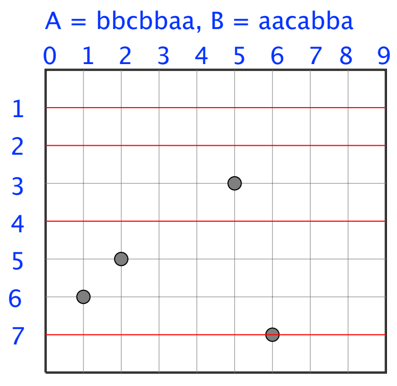
The following theorem concludes the data structure described in this section. We note that the append operation to B can be carried out similarly.

Theorem 4.7. *Given strings A and B of lengths m and n, respectively, over alphabet Σ with $L = LCS(A,B)$. We can construct an $O(n)$ (resp. $O(m)$) space data structure that encodes the LCS score between every suffix of B and every prefix of A and supports 1-sided prepend operations to A (resp. append operations to B) in $O(L)$ time.*

4.2 2-sided incremental $\mathcal{J}$ matrix

For the 2-sided case we note that both A and B are subjected to modifications. Thus, the look-up tables must be updated dynamically. We suggest the following data structure that supports $\mathrm{PrevMatch}_B$ and $\mathrm{NextMatch}_B$ queries in constant time, and append operations to B in constant time.

We start with a data structure for $\mathrm{PrevMatch}_B(i,\sigma)$. Note that the values of $\mathrm{PrevMatch}_B(i,\sigma)$ remain constant when append operations to B are applied. Thus, we store a dynamic array $\mathrm{PrevTable}_B$ in which $\mathrm{PrevTable}_B[i,\sigma] = \mathrm{PrevMatch}_B(i,\sigma)$ for all i and σ (a dynamic array is a data structure that stores an array and allows appending a constant number of cells to the end of the array in constant worst-case time). Assuming that B is of length n and a character τ is appended to B, we only need to update the cell $\mathrm{PrevTable}_B[n+1,\sigma]$ for every $\sigma \in \Sigma$. Clearly, $\mathrm{PrevTable}_B[n+1,\sigma] = \mathrm{PrevTable}_B[n,\sigma]$ if $\sigma \neq \tau$ and $\mathrm{PrevTable}_B[n+1,\tau] = n+1$.



(a) $\mathcal{J}^{\square}$

i	$\text{PrevMatch}_B(i,a)$	$\text{NextMatch}_B(i,a)$
0	$-\infty$	1
1	1	2
2	2	4
3	2	4
4	4	7
5	4	7
6	4	7
7	7	∞

$P = \{(3,5),(5,2),(6,1),(7,6)\}$

Figure 5: A run of the algorithm on the strings A and B of Figure 4, and a prepend of the character a to A. The algorithm begins with $i_1 + 1 = 3$, where $k_1^- = \text{PrevMatch}_B(i_1,\sigma) = 2$, and $k_1^+ = \text{NextMatch}_B(i_1,\sigma) = 4$. The only pivotal point in rows $2,3,4$ is $(3,5)$, hence $C^{\square}[3,5] = 1$ by part 1 of Lemma 4.6 and $C^{\square}[2,5] = -1$ by part 2 of Lemma 4.6. The following scanned pivotal point is $(5,2)$ for which $k_2^- = 4$ and $k_2^+ = 7$. The backward scan begins with the pivotal point $(7,6)$ and $j_{\min} = \infty$. By Lemma 4.6 set $C^{\square}[7,6] = 1$. The value of $j_{\min}$ is 6 when scanning the pivotal point $(6,1)$, thus we set $C^{\square}[6,1] = 1$ and $C^{\square}[6,6] = -1$. When scanning the pivotal point $(5,2)$ the value of $j_{\min}$ is 1, thus row 5 does not contain non zero elements. The block scan is complete after setting $C^{\square}[4,1] = -1$. Lastly, we set $C^{\square}[7,0] = 1$ by Corollary 4.4.

For the data structure that supports $\text{NextMatch}_B(i,\sigma)$ queries, it is no longer the case that the values remain constant. To handle this, we use a dynamic array NextTable_B that is defined as follows: $\text{NextTable}_B[i,\sigma] = \text{NextMatch}_B(i,\sigma)$ if $i = 0$ or $B[i] = \sigma$. Otherwise, $\text{NextTable}_B[i,\sigma]$ contains an arbitrary value. We can now compute $\text{NextMatch}_B(i,\sigma)$ using the following equality: $\text{NextMatch}_B(i,\sigma) = \text{NextTable}_B[i',\sigma]$ where $i' = \max(0, \text{PrevMatch}_B(i,\sigma))$. Assuming that B is of length n and a character τ is appended to B, we update NextTable_B by setting $\text{NextTable}_B[i,\tau] = n+1$ where $i = \text{PrevMatch}_B(n,\tau)$.

Now, using the dynamic data structures, the technique remains the same as described in Section 4.1. Note that when an append operation is preformed, the set of pivotal points needs to be sorted by the column indices. This requires $O(L \log\log L)$ time for sorting the set of L pivotal points using the sorting algorithm of [12]. This leads to the following theorem.

Theorem 4.8. Given strings A and B of lengths m and n, respectively, over alphabet Σ with $L = LCS(A,B)$. We can construct an $O(m+n)$ space data structure that encodes the LCS score between every suffix of B and every prefix of A and supports prepend operations to A and append operations to B in $O(L \log\log L)$ time.

5 Incremental $\mathcal{K}$ and $\mathcal{J}$ matrices

In this section we show how to maintain the matrices $\mathcal{K}$ and $\mathcal{J}$ and support append operations to either A or B.

Recall that $\mathcal{J}$ is over the index set $[0 : n] \times [0 : m]$ and $\mathcal{K}$ is over the index set $[0 : n] \times [0 : n]$. We store a list $P_{\mathcal{K}}$ for the matrix $\mathcal{K}$ as defined in Section 3, and a list $P_{\mathcal{J}}$ for the matrix $\mathcal{J}$ as defined in Section 4. Both lists are sorted by decreasing column indices.

First consider an append operation to B. This operation adds a new row to the matrix $\mathcal{J}$, and a new row and column to $\mathcal{K}$. Denote the modified matrices by $\mathcal{J}'$ and $\mathcal{K}'$. The modifications to the matrix $\mathcal{J}$ are carried as described in Section 4.1. The matrix $\mathcal{K}'$, on the other hand, does not vary much from the matrix $\mathcal{K}$, since the submatrix $\mathcal{K}'[0..n, 0..n]$ is not affected by the newly added symbol. However, note that a new pivotal point may be added to the matrix $\mathcal{K}'^{\square}$ at the newly appended column. In the following lemma we show how to compute this pivotal point.

Lemma 5.1. If (i, m) is a step index in $C = \mathcal{J}' - \mathcal{J}$ then $(i, n+1)$ is a pivotal point in $\mathcal{K}'^{\square}$. Otherwise, if $\text{PrevMatch}_A(m, \sigma) = -\infty$ (i.e. there is no new match-point) then $(n + 1, n + 1)$ is a pivotal point in $\mathcal{K}'^{\square}$.

Proof. Note that the $(n + 1)$'th column of $\mathcal{K}'$ is precisely the m'th column of the matrix $\mathcal{J}'$, and the n'th column of $\mathcal{K}$ is precisely the m'th column of $\mathcal{J}$. Hence if (i, m) is a step index in $C = \mathcal{J}' - \mathcal{J}$ then $\mathcal{K}'^{\square}[i, n+1] = (\mathcal{J}'[i, m] - \mathcal{J}[i, m]) - (\mathcal{J}'[i-1, m] - \mathcal{J}[i-1, m]) = C[i, m] - C[i-1, m] = 1$. Otherwise, if $\text{PrevMatch}_A(m, \sigma) = -\infty$, then $\mathcal{K}'[n, n+1] = 0$ and by definition we get $\mathcal{K}'^{\square}[n+1, n+1] = 1$. $\square$

The step index on the m'th column of $\mathcal{J}$ can be computed using Corollary 4.3 in $O(L)$ time. Hence the modifications to both $\mathcal{K}$ and $\mathcal{J}$ can be carried out in $O(L)$ time.

An append operation to A can be carried out analogously. In this case the matrix $\mathcal{K}$ is modified, and the list $P_{\mathcal{K}'}$ can be obtained as described in Section 3.1. A new column is also appended to $\mathcal{J}$. In this case we have the following lemma.

Lemma 5.2. If (i, n) is a step index in $C = \mathcal{K}' - \mathcal{K}$ then $(i, m+1)$ is a pivotal point in $\mathcal{J}'^{\square}$.

By applying Lemma 5.2 for the matrix $\mathcal{J}'^{\square}$ and using the approach described in Section 3.1 for the matrix $\mathcal{K}'^{\square}$, we can compute the lists $P_{\mathcal{J}'}$ and $P_{\mathcal{K}'}$ in $O(\Delta)$ time.

We derive the following theorem.

Theorem 5.3. Given strings A and B of lengths m and n, respectively, over alphabet Σ with $L = LCS(A, B)$ and $\Delta = n - L$. We can construct an $O(m + n)$ space data structure that encodes the LCS score between A and every substring of B, and between every suffix of B and every prefix of A. This data-structure supports append operations to A or B in $O(\Delta)$ and $O(L)$ time, respectively.

References

S-table [1] C. E. R. Alves, E. N. Cáceres, and S. W. Song. An all-substrings common subsequence algorithm. *Discrete Applied Mathematics*, 156(7):1025–1035, 2008.

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

[3] A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2(1-4):315–336, 1987.

[4] K. Bringman and M. Künnemann. Multivariate fine-grained complexity of longest common subsequence. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1216–1235. SIAM, 2018.

[5] S. Cabello. Many distances in planar graphs. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 1213–1220. Society for Industrial and Applied Mathematics, 2006.

[6] S. Cabello, E. W. Chambers, and J. Erickson. Multiple-source shortest paths in embedded graphs. *SIAM Journal on Computing*, 42(4):1542–1571, 2013.

[7] A. Carmel, D. Tsur, and M. Ziv-Ukelson. On almost monge all scores matrices. *Algorithmica*, 2018.

[8] M. Christodoulakis, C. S. Iliopoulos, K. Park, and J. S. Sim. Implementing approximate regularities. *Mathematical and computer modelling*, 42(7-8):855–866, 2005.

[9] V. Cohen-Addad, S. Dahlgaard, and C. Wulff-Nilsen. Fast and compact exact distance oracle for planar graphs. *arXiv preprint arXiv:1702.03259*, 2017.

[10] D. Eisenstat and P. N. Klein. Linear-time algorithms for max flow and multiple-source shortest paths in unit-weight planar graphs. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 735–744. ACM, 2013.

[11] P. Gawrychowski, S. Mozes, O. Weimann, and C. Wulff-Nilsen. Better tradeoffs for exact distance oracles in planar graphs. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 515–529. SIAM, 2018.

[12] Y. Han. Deterministic sorting in O(n log log n) time and linear space. *Journal of Algorithms*, 1(50):96–105, 2004.

[13] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM (JACM)*, 24(4):664–675, 1977.

[14] P.-H. Hsu, K.-Y. Chen, and K.-M. Chao. Finding all approximate gapped palindromes. *International Journal of Foundations of Computer Science*, 21(06):925–939, 2010.

[15] H. Hyyrö. An efficient linear space algorithm for consecutive suffix alignment under edit distance (short preliminary paper). In *International Symposium on String Processing and Information Retrieval*, pages 155–163. Springer, 2008.

[16] H. Hyyrö and S. Inenaga. Compacting a dynamic edit distance table by rle compression. In *International Conference on Current Trends in Theory and Practice of Informatics*, pages 302–313. Springer, 2016.

[17] H. Hyyrö, K. Narisawa, and S. Inenaga. Dynamic edit distance table under a general weighted cost function. *Journal of Discrete Algorithms*, 34:2–17, 2015.

[18] Y. Ishida, S. Inenaga, A. Shinohara, and M. Takeda. Fully incremental lcs computation. In *International Symposium on Fundamentals of Computation Theory*, pages 563–574. Springer, 2005.

[19] C. Kent, G. M. Landau, and M. Ziv-Ukelson. On the complexity of sparse exon assembly. *Journal of Computational Biology*, 13(5):1013–1027, 2006.

[20] S.-R. Kim and K. Park. A dynamic edit distance table. In *Annual Symposium on Combinatorial Pattern Matching*, pages 60–68. Springer, 2000.

[21] S. Lai, F. Yang, and T. Chen. Online pattern matching and prediction of incoming alarm floods. *Journal of Process Control*, 56:69–78, 2017.

[22] G. M. Landau, E. Myers, and M. Ziv-Ukelson. Two algorithms for lcs consecutive suffix alignment. *Journal of Computer and System Sciences*, 73(7):1095–1117, 2007.

[23] G. M. Landau, E. W. Myers, and J. P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582, 1998.

[24] G. M. Landau, B. Schieber, and M. Ziv-Ukelson. Sparse LCS common substring alignment. *Information Processing Letters*, 88(6):259–270, 2003.

[25] G. M. Landau, J. P. Schmidt, and D. Sokol. An algorithm for approximate tandem repeats. *J. of Computational Biology*, 8(1):1–18, 2001.

[26] G. M. Landau and M. Ziv-Ukelson. On the common substring alignment problem. *J. of Algorithms*, 41(2):338–359, 2001.

[27] U. Matarazzo, D. Tsur, and M. Ziv-Ukelson. Efficient all path score computations on grid graphs. *Theoretical Computer Science*, 525:138–149, 2014.

[28] S. Mozes and C. Sommer. Exact distance oracles for planar graphs. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 209–222. Society for Industrial and Applied Mathematics, 2012.

[29] E. W. Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[30] Y. Sakai. An almost quadratic time algorithm for sparse spliced alignment. *Theory of Computing Systems*, 48(1):189–210, 2011.

[31] Y. Sakai. A substring-substring lcs data structure. *Theoretical Computer Science*, 2018.

[32] J. P. Schmidt. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM J. of Computing*, 27(4):972–992, 1998.

[33] D. Sokol and J. Tojeira. Speeding up the detection of tandem repeats over the edit distance. *Theoretical Computer Science*, 525:103–110, 2014.

[34] A. Tiskin. Semi-local string comparison: algorithmic techniques and applications. arXiv:0707.3619v16.

[35] A. Tiskin. Semi-local longest common subsequences in subquadratic time. *Journal of Discrete Algorithms*, 6(4):570–581, 2008.

[36] A. Tiskin. Semi-local string comparison: Algorithmic techniques and applications. *Mathematics in Computer Science*, 1(4):571–603, 2008.

[37] S. Wu, U. Manber, G. Myers, and W. Miller. An o (np) sequence comparison algorithm. *Information Processing Letters*, 35(6):317–323, 1990.

[38] H. Zhang, Q. Guo, and C. S. Iliopoulos. Generalized approximate regularities in strings. *International Journal of Computer Mathematics*, 85(2):155–168, 2008.