Sequential and Parallel Algorithms for the All-Substrings Longest Common Subsequence Problem*

C. E. R. Alves
Faculdade de Tecnologia e Ciências Exatas
Universidade São Judas Tadeu
São Paulo-SP-Brazil
prof.carlos_r_alves@usjt.br

E. N. Cáceres
Dept. of Computação e Estatística
Universidade Federal de Mato Grosso do Sul
Campo Grande-MS-Brazil
edson@dct.ufms.br
http://www.dct.ufms.br/∼edson

S. W. Song
Dept. de Ciência da Computação - IME
Universidade de São Paulo
São Paulo-SP-Brazil
song@ime.usp.br
http://www.ime.usp.br/∼song

Abstract

Given two strings A and B of lengths n_a and n_b, respectively, the All-substrings Longest Common Subsequence (ALCS) problem obtains, for any substring B' of B, the length of the longest string that is a subsequence of both A and B'. The sequential algorithm takes $O(n_a n_b)$ time and $O(n_b)$ space. We present a parallel algorithm for the ALCS on the Coarse Grained Multicomputer (BSP/CGM) model with $p < \sqrt{n_a}$ processors, that takes $O(n_a n_b/p)$ time and $O(n_b\sqrt{n_a})$ space per processor, with $O(\log p)$ communication rounds. The proposed algorithm also solves the basic Longest Common Subsequence (LCS) Problem that finds the longest string (and not only its length) that is a subsequence of both A and B. To our knowledge, this is the best BSP/CGM algorithm for the LCS and ALCS problems in the literature.

*Partially supported by FINEP-PRONEX-SAI Proc. No. 76.97.1022.00, CNPq Proc. No. 52.3778/96-1 and 52.2028/02-9.

1 Introduction

Given two strings, obtention of the longest subsequence common to both strings is an important problem with applications in DNA sequence comparison, data compression, pattern matching, etc [16, 19]. In this report we consider the more general all-substring longest common subsequence problem and present a time and space efficient parallel algorithm.

1.1 The Longest Common Subsequence Problem

Consider a string of symbols from a finite alphabet. A *substring* of a string is any fragment of contiguous symbols of the string. In other words, a substring is obtained from a string by the deletion of zero or more symbols from its beginning or its end. A *subsequence* of a string, on the other hand, is obtained by deleting zero or more symbols from the original string, from any position. The remaining symbols preserve the order they had on the original string.

Before we proceed, let us define some notation for strings, substrings and symbols: Strings of symbols will be denoted by upper-case letters, like A and B. Isolated symbols of a string will be identified by the string name (lower-case) with a subscripted index. The indices start at 1, for the first symbol. The length of the string will be denoted by n with the lower-case string name subscripted. For example,

$$A = a_1 a_2 a_3 \ldots a_{n_a - 1} a_{n_a}.$$

A substring is indicated by the upper-case letter of the original string and indices that indicate the initial (subscripted) and final (superscripted) positions of the symbols in the original string. For example,

$$A_3^6 = a_3 a_4 a_5 a_6.$$

We define the Longest Common Subsequence (LCS) Problem as follows:

Definition 1.1 (Longest Common Subsequence Problem) *Given two strings A and B, find the longest string C that is a subsequence of A and B.*

Figure 1 illustrates one instance of this problem and its solution.

A	x	y	w	w	y	x	–	–	w	
B	x	–	w	w	y	x	y	z	–	
value	1	0	1	1	1	1	0	0	0	5

Figure 1: Example of a solution to the LCS Problem. For strings xywwyxw and xwwyxyz the longest common subsequence is xwwyx.

In this figure, we illustrate the concept of *alignment* of strings. The strings under comparison are placed in an array of two lines in such a way that no column contains two different symbols. Each column may contain one blank space, though. Columns with equal symbols have a value of 1, the others have value 0. The array with the greatest possible sum of column values gives the longest common subsequence.

In several applications, we are more interested in the length of the longest common subsequence than in the actual subsequence.

There are several algorithms for the LCS problem. Some are based on Dynamic Programming, others on primal-dual algorithms or other techniques, but in the general case all these algorithms have a worst-case time complexity that is $O(n_a n_b)$, where n_a and n_b are the lengths of strings A and B, respectively. See [7, 16, 17, 19] for surveys. Here we will comment very briefly the Dynamic Programming algorithm for the LCS, when we present the characteristic Grid Directed Acyclic Graph (GDAG for short) that is used throughout this report.

There are also algorithms that exploit the finitude of the alphabet to decompose the LCS into repetitive subproblems that can be solved separately and then merged, using what is now known as the Four Russians technique (see [14, 15]). Unfortunately, although the time complexity of these algorithms is subquadratic, they are not for practical use, as they are better than the others previously mentioned only when the strings involved are very large, much larger than the ones that occur in real problems [7].

1.2 The All-substrings Longest Common Subsequence Problem

The problem that will be covered in this report is a generalization of the LCS problem, named the *All-substrings Longest Common Subsequence* (ALCS) Problem, defined as follows

Definition 1.2 (All-substrings Longest Common Subsequence Problem) *For two strings A and B, solve the LCS problem for A and all substrings of B, that is, solve the LCS for all pairs (A, B_i^j), where $1 \leq i \leq j \leq n_b$.*

In fact, we are more interested in the *lengths* of the longest common subsequences. The determination of all these lengths for the ALCS can be done in $O(n_a n_b)$ time, as will be shown in this report. Asymptotically, this is the same complexity we have for the basic LCS problem, and the implementation of the algorithm is quite simple.

The ALCS problem is a restricted form of the All-Substrings Alignment Problem [3, 2, 18], and has several applications, like finding approximate tandem repeats in strings [18], solving the circular alignment of two strings [13, 18] and finding the alignment of one string with several others that have a common substring [10].

In this report, a parallel algorithm for the ALCS problem is presented. It is based on a previous PRAM algorithm by Mi Lu and Hua Lin [12] and is adapted here for the BSP/CGM Model of computation, to be explained in the next section. This model is quite important for practical applications, being more suited to development of parallel algorithms to distributed systems [6].

1.3 The BSP/CGM Model

In this report we use the *Coarse Grained Computer* model [5, 6], that is based on the *Bulk Synchronous Parallel* (BSP) model [20, 21]. A BSP/CGM consists of a set of p processors $P_1, \ldots, P_p$, each with its own local memory. Each processor is connected to a router that can send messages in a point-to-point fashion.

A BSP/CGM algorithm consists of alternating rounds of local computation and global communication that are separated by barrier synchronizations. A round is equivalent to a superstep in the BSP model. Each communication round consists of routing a single h-relation with $h = O(M)$, being M the size of the local memory of each processor.

In the BSP/CGM model, the communication cost is modeled by the number of communication rounds. The main advantage of BSP/CGM algorithms is that they map very well

to standard parallel hardware, like Beowulf type processor clusters [6]. So, our goal is to achieve a linear speed-up with a low number of communication rounds, if possible depending only on the number of processors, not on the size of the instance of the problem. Of course, minimizing the space required in the local memory and size of the communication rounds are important. One space efficient algorithm should have $M = O(N/p)$, where N is the space required by reasonable sequential algorithm for the problem at hand.

1.4 Monotone and Totally Monotone Matrices

We present now the key ideas about Monotone and Totally Monotone Matrices, that are very important to our CGM algorithm.

Monotone and totally monotone matrices arise in several applications. The main reference to this kind of matrices is the work of Aggarwal et al. in [1], where these matrices are used in a geometric problem. The definitions and results presented here are adapted from this reference[1].

In Aggarwal et al. [1], several algorithms related to these kind of matrices were presented. To make this report more self-contained, this algorithms (already adapted) are presented in the final appendix. Readers that are already familiar with these matrices may skip that, but it is important to notice that we had to make some adaptations in the original results to make them more suited to our needs.

The characteristic properties of monotone matrices and totally monotone matrices are related to the location of the minimum element of each column. Before we describe these properties, some terms have to be defined.

Definition 1.3 *[Imin[M] and Cmin[M]] Given an $n \times m$ matrix M of integer numbers, vector Imin[M] is such that for all j, $1 \leq j \leq m$, Imin[M](j) is the smallest value of i such that $M(i, j)$ is the minimum of the column j of M. Vector Cmin[M] contains the minima of the columns of M, that is, Cmin[M](j) = M(Imin[M](j), j).*

So, *Imin*[M] contains the *locations* of the column minima, while *Cmin*[M] contains the *values* proper of these column minima. Now we proceed to the main definitions of this section:

Definition 1.4 (Monotone Matrix) *Let M be an $n \times m$ matrix of integer numbers. M is a* monotone *matrix if and only if, for all $1 \leq j_1 < j_2 \leq m$, Imin[M](j_1) $\leq$ Imin[M](j_2).*

Definition 1.5 (Totally Monotone Matrix) *Let M be a $n \times m$ matrix of integer numbers. M is a* totally monotone *matrix if and only if every 2×2 submatrix of M is monotone.*

The following lemma is very simple to prove.

Lemma 1.1 *Every totally monotone matrix is also monotone.*

Proof. If M is not a monotone matrix then there are j_1 and j_2 such that $j_1 < j_2$ and *Imin*[M](j_1) > *Imin*[M](j_2). The 2×2 matrix formed by $M(i_1, j_1)$, $M(i_1, j_2)$, $M(i_2, j_1)$ and $M(i_2, j_2)$ is not monotone, so M cannot be totally monotone. □

Here we will be interested in finding the minima of *all* the columns of monotone or totally monotone matrices. We state this problem more formally now.

[1]For those familiar with the reference, we had to exchange the roles of matrix columns and matrix lines, and also changed "maxima" to "minima".

Definition 1.6 (Column Minima Problem) *Given an $n \times m$ matrix M of integers, determine vector $Imin[M]$.*

The problem is stated here in terms of the locations of the minima. Efficient algorithms to solve this problem were already presented in [1] and are commented in the final appendix.

An important operation with totally monotone matrices is the *contraction of lines*. This operation was used implicitly in [12], for the solution of LCS in the CREW-PRAM model. We give now an explicit definition.

Definition 1.7 *[Contraction of Contiguous Lines of a Matrix (with respect to the Column Minima Problem)] The* contraction of lines *applied to a set of* contiguous *lines of a matrix M is the substitution of these lines in M by a single new line. Element i of this new line is the minimum of all elements in column i of the replaced lines.*

So, if $S = \{l_1, l_1 + 1, l_1 + 2, \ldots, l_2\}$, we will denote by $Cont_S(M)$ the matrix resulting from the contraction of matrix M applied to lines from l_1 to l_2. If M is an $n \times m$ matrix, let M^i denote the ith row of M, then $N = Cont_S(M)$ is an $(n - l_2 + l_1) \times m$ matrix such that $N^{l_1} = Cmin[M(i, j)[l_1 \leq i \leq l_2]]$, $N^i = M^i$ for $i < l_1$ and $N^i = M^{i+l_2-l_1}$ for $i > l_1$.

Theorem 1.2 *If M is a totally monotone matrix and S is a set of consecutive indices of lines of M, $Cont_S(M)$ is also a totally monotone matrix and $Cmin[M] = Cmin[Cont_S(M)]$.*

Proof. $Cmin[M] = Cmin[Cont_S(M)]$ follows directly from Definition 1.7, for any kind of matrix. We need to prove that $Cont_S(M)$ is totally monotone.

Let $Cont_S(M)$ be defined for some totally monotone matrix M and set S. For any 2×2 submatrix M' of $Cont_S(M)$, with elements in columns j_1 and j_2 ($j_1 < j_2$), we will have only two possibilities: none of the lines results from the contraction or one of them does.

In the first case M' is clearly monotone, because it is also a submatrix of M. In the second case we have two subcases: both elements of the contracted line are in the same line of M or not.

Again, in the first case M' is also a submatrix of M. In the second case, the elements of the contracted line may be denoted by $M(i_1, j_1)$ and $M(i_2, j_2)$ with $i_1 < i_2$, because M is (totally) monotone.

The other two elements of M' may be denoted by $M(i_0, j_1)$ and $M(i_0, j_2)$. Let us suppose that $i_0 < i_1$. The case where $i_0 > i_2$ can be treated in an analogous way.

To prove that this submatrix is monotone, we need to show that if $M(i_1, j_1) < M(i_0, j_1)$ then $M(i_2, j_2) < M(i_0, j_2)$. The total monotonicity of M assures that if $M(i_1, j_1) < M(i_0, j_1)$ then $M(i_1, j_2) < M(i_0, j_2)$. Since $M(i_2, j_2) < M(i_1, j_2)$ (because $M(i_1, j_2)$ is the minimum of column j_2 among the lines S) we will have $M(i_2, j_2) < M(i_0, j_2)$. □

2 Properties of the ALCS Problem

We first introduce the characteristic GDAG (Grid Directed Acyclic Graph) for the ALCS problem. Let A and B be two strings, respectively of lengths n_a and n_b. The GDAG has $n_a + 1$ lines and $n_b + 1$ columns of vertices. Labeling lines from 0 to n_a and the columns from 0 to n_b, each vertex of a GDAG G is identified by $G(i, j)$, where $0 \leq i \leq n_a$ and $0 \leq j \leq n_b$.

The arcs in this GDAG are weighted with values 0 or 1 as follows:

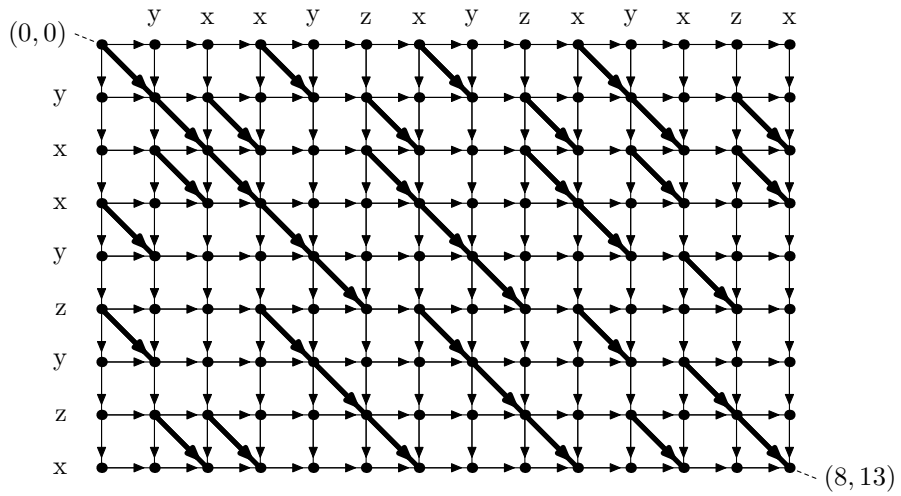

Figure 2: GDAG for the ALCS problem. The strings being compared are A = yxxyzyzx and B = yxxyzxyzxyxzx. The thick diagonal arcs have weight 1, the others have weight 0.

- (diagonal arcs) for $1 \leq i \leq n_a$ and $1 \leq j \leq n_b$ there is an arc from $G(i-1, j-1)$ to $G(i, j)$ with weight 1 if $a_i = b_j$, 0 otherwise.
- (vertical arcs) for $0 \leq i \leq n_a$ and $1 \leq j \leq n_b$ there is an arc from $G(i, j-1)$ to $G(i, j)$ with weight 0.
- (horizontal arcs) for $1 \leq i \leq n_a$ and $0 \leq j \leq n_b$ there is an arc from $G(i-1, j)$ to $G(i, j)$ with weight 0.

An example can be seen in Figure 2. The diagonal arcs with weight 0 will be ignored.

The vertices at the top (superior line) of G will be called $T_G(i)$ and the the ones at the bottom (inferior line) will be called $F_G(i)$[2],$0 \leq i \leq n_b$.

To solve the LCS problem, one may look for the *highest weighting path* from $G(0, 0)$ to $G(n_a, n_b)$. The total weight of this path is the length of the longest common subsequence between A and B, and the actual path gives us the actual subsequence: each vertical arc taken represents a deletion in A, each horizontal arc represents a deletion in B and each diagonal arc (of weight 1) represents matched symbols in A and B.

Dynamic Programming techniques based on this principle are of common use and have been applied in parallel and sequential algorithms. The knowledge of this algorithm is not necessary for the comprehension of this report, so it will not be presented here. In fact, this algorithm is a restricted form of the one that solves the more general *string editing* problem. We suggest that the reader look for the algorithm by Hirschberg [8], that can be found in [7, 16, 19]. With this algorithm, the basic LCS problem can be solved in $O(n_a n_b)$ time and $O(n_a + n_b)$ space.

For the ALCS problem we are interested in more paths than just the one from $G(0, 0)$ to $G(n_a, n_b)$. We have then the following definition:

Definition 2.1 ($C_G(i, j)$) *For $0 \leq i \leq j \leq n_b$, $C_G(i, j)$ is the largest total weight of a path from $T_G(i)$ to $F_G(j)$ in G. If $i > j$ (there is no path between $T_G(i)$ and $F_G(j)$), $C_G(i, j) = 0$.*

[2]We use F (*Floor*) and not B (*Bottom*) to avoid confusion with string B.

$C_G(i,j)$ represents the length of the longest common subsequence of A and the substring B_{i+1}^{j}. When $i = j$, there is no such substring and the only path from $T_G(i)$ to $F_G(j)$ is formed exclusively by vertical arcs and has total weight 0. When $i > j$, B_{i+1}^{j} also does not exist and we adopt $C_G(i,j) = 0$.

The values of $C_G(i,j)$ have the following properties:

Properties 2.1

1. *For all i ($0 \leq i \leq n_b$) and all j ($1 \leq j \leq n_b$) we have $C_G(i,j) = C_G(i,j-1)$ or $C_G(i,j) = C_G(i,j-1) + 1$.*
2. *For all i ($1 \leq i \leq n_b$) and all j ($0 \leq j \leq n_b$) we have $C_G(i-1,j) = C_G(i,j)$ or $C_G(i-1,j) = C_G(i,j) + 1$.*
3. *For all i and all j ($1 \leq i < j \leq n_b$) we have that if $C_G(i-1,j) = C_G(i-1,j-1) + 1$ then $C_G(i,j) = C_G(i,j-1) + 1$.*

Proof. We demonstrate the property 2.1(1). The demonstration of 2.1(2) is similar.

Naturally, $C_G(i,j)$ and $C_G(i,j-1)$ are integers. When $i \geq j$ we have $C_G(i,j) = C_G(i,j-1) = 0$. When $i < j$, looking at the paths that start at vertex $T_G(i)$ it becomes clear that $C_G(i,j) \geq C_G(i,j-1)$, since the best path to vertex $F_G(j-1)$ can be extended to $F_G(j)$ by a horizontal arc with weight 0. On the other hand, as all arcs with weight 1 are diagonal arcs, it is clear that $C_G(i,j) - C_G(i,j-1) \leq 1$. That is because a path from $T_G(i)$ to $F_G(j-1)$ can be created based on the best path to $F_G(j)$, eliminating the vertices in column j and completing the path to $F_G(j-1)$ with vertical arcs. One diagonal is eliminated at the most.

To demonstrate 2.1(3), we suppose that $C_G(i-1,j) = C_G(i-1,j-1) + 1$ for a certain pair (i,j), $1 \leq i < j \leq n_b$. Let $C1$ be the *leftmost* path from $T_G(i-1)$ to $F_G(j)$ with total weight $C_G(i-1,j)$ ($C1$ is such that no path from $T_G(i-1)$ to $F_G(j)$ with the same weight has any vertex to the left of a vertex of $C1$ in the same line). Let $C2$ be the *leftmost* path from $T_G(i)$ to $F_G(j-1)$ with total weight $C_G(i,j-1)$. It is easy to show that such *leftmost* paths shall exist and that they have a unique common subpath with at least one vertex. Let $C1_a$ and $C1_b$ be the subpaths of $C1$ that do not belong in $C2$, $C1_a$ is the subpath starting at $T_G(i-1)$ and $C1_b$ is the subpath ending at $F_G(j)$. Let $C2_a$ and $C2_b$ be subpaths of $C2$ defined analogously. The common subpath is denoted C. See Figure 3.

Naturally, the weight of $C1_b$ must be equal to the weight of $C2_b$ plus 1. Otherwise we would have a path from $T_G(i-1)$ to $F_G(j-1)$ (formed by $C1_a$, C and $C2_b$) with the same weight as $C1$, which contradicts our hypothesis. So, the path composed by $C2_a$, C and $C1_b$ will have a total weight equal to the weight of $C2$ plus 1, which implies $C_G(i,j) > C_G(i,j-1)$. By 2.1(1) we conclude that $C_G(i,j) = C_G(i,j-1) + 1$.

□

Property 2.1(3) is a variation of the *Monge properties*. In [18] this kind of properties is extensively used to solve the All-Substrings Alignment problem [2], specially in some special cases related to the ALCS problem. These results will be commented in Section 3, that contains the exposition of the sequential algorithm for the ALCS problem.

Property 2.1(1) is important because it indicates that for a fixed value of i, the values of $C_G(i,0)$, $C_G(i,1)$, $C_G(i,2)$, $\ldots C_G(i,n_b)$ form a non-decreasing sequence that can be defined in an implicit way, indicating just the values of j for which $C_G(i,j) > C_G(i,j-1)$.

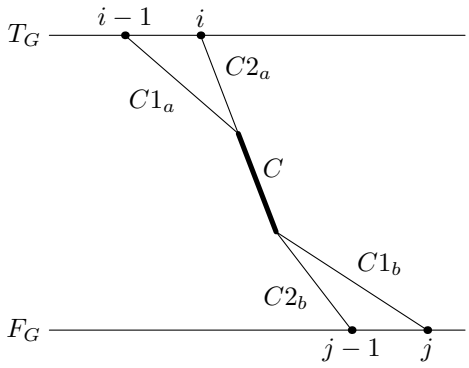

Figure 3: Demonstration of Property 2.1(3).

This fact was used in several sequential algorithms for the LCS (see [4, 9, 17]). It is also important for the CREW-PRAM algorithm presented by H. Lin and M. Lu [12], that is the basis for the algorithm described in this report and for the following definition:

Definition 2.2 (Matrix D_G) *Let G be the GDAG for the ALCS problem with strings A and B. For $0 \leq i \leq n_b$, $D_G(i,0) = i$ and for $1 \leq k \leq n_a$, $D_G(i,k)$ indicates the value of j such that $C_G(i,j) = k$ and $C_G(i,j-1) = k-1$. If there is no such value, $D_G(i,k) = \infty$.*

This definition implies $C(i,j) \leq n_a$. We define $D_G(i,0) = i$ and not 0 because this simplifies the exposition of the parallel algorithm, as will soon become clear. Notice that, for notational simplicity, D_G will be treated as a matrix with indices starting at 0. We denote D_G^i the line i of D_G, that is, the vector composed by $D_G(i,0)$, $D_G(i,1)$, ..., $D_G(i,n_a)$. Matrix D_G is essential to the CGM algorithm that will be presented, being the main form of data representation used during its execution.

For an example, let us take the GDAG from Figure 2. The values of $C_G(i,j)$ and $D_G(i,k)$ are displayed in tables 1 and 2 (page 10).

From the values of $D_G(i,k)$ we can obtain the values of $C_G(i,k)$. Furthermore, the following results [11, 12] indicate that the information contained in D_G can be represented in space $O(n_a + n_b)$, instead of the $O(n_a n_b)$ space necessary for the direct representation.

Properties 2.2

1. *If $k_1 < k_2$ and $D_G(i,k_1) \neq \infty$ then $D_G(i,k_1) < D_G(i,k_2)$.*
2. *If $i_1 < i_2$ then $D_G(i_1,k) \leq D_G(i_2,k)$.*
3. *If $D_G(i_1,k_1) = j_1$ and $j_1 \neq \infty$ then for all i, $i_1 \leq i \leq j_1$, there is k such that $D_G(i,k) = j_1$. In other words, if component j_1 appears in line D_G^i then it also appears in all the lines up to line $D_G^{j_1}$.*
4. *If $D_G(i_1,k_1) = D_G(i_2,k_2) = j_1$ then for all i, $i_1 \leq i \leq i_2$, there is k such that $D_G(i,k) = j_1$. In other words, if j_1 appears in lines $D_G^{i_1}$ and $D_G^{i_2}$ then it also appears in lines between $D_G^{i_1}$ and $D_G^{i_2}$.*

Proof. If $k_1 < k_2$ and $j_1 = D_G(i,k_1) \neq \infty$ then $C_G(i,j_1) = k_1$. $C_G(i,j)$ does not decrease with j, so if there is j such that $C_G(i,j) = k_2$ then $j > j_1$. If there is no such j then $D_G(i,k_2) = \infty$. In either case we have $D_G(i,k_1) < D_G(i,k_2)$, proving (1).

If $i_1 < i_2$, by Property 2.1(2) $C_G(i_2, j) \leq C_G(i_1, j)$ for all j. If $D_G(i_1, k) > D_G(i_2, k) \neq \infty$ for some value of k, then for $j = D_G(i_2, k)$ we have $C_G(i_2, j) = k > C_G(i_1, j)$, a contradiction. Thus (2) follows.

The demonstration of (3) is done by induction on i. The presence of j_1 in $D_G^{j_1}$ follows from Definition 2.2. Now let us suppose that for a certain value of i ($i_1 \leq i < j_1 \neq \infty$) we have $D_G(i, k) = j_1$ for some value of k (the specific value of k is not important). This means that $C_G(i, j_1) = k = C_G(i, j_1 - 1) + 1$. By Property 2.1(3) we have $C_G(i+1, j_1) = C_G(i+1, j_1-1)+1$, which means that there is some k such that $D_G(i+1, k) = j_1$, concluding the inductive step. This step applies while the condition for Property 2.1(3) ($i + 1 < j_1$) is true. So, j_1 appears in every line from $D_G^{i_1}$ to $D_G^{j_1-1}$.

Property (4) can be easily demonstrated with (3), since $D_G(i, k)$ can only be j_1 if $i \leq j_1$. □

Properties 2.2(2) and 2.2(3) are used to reduce the space necessary to represent D_G, because they show that two consecutive lines of D_G are very similar. We state this in a more precise way now.

Properties 2.3 *For $0 \leq i < n_b$,*

1. *There is one unique finite component of D_G^i that does not appear in D_G^{i+1}, which is $D_G(i, 0) = i$.*
2. *There is at most one finite component of D_G^{i+1} that does not appear in D_G^i.*

Proof. Property (1) follows directly from Property 2.2(3). If j_1 appears in line D_G^i then it appears in all the following lines, up to line j_1. As $D_G(i, 0) = i$, this is the last line in which i appears. All the other components from D_G^i are larger than i (by Property 2.2(1)) and also appear in D_G^{i+1}.

By Property 2.2(2) we have that D_G^{i+1} cannot have more finite components than D_G^i. As only one finite component from D_G^i does not appear in D_G^{i+1}, there is at the most one new component in D_G^{i+1}. □

Properties 2.3(1) and 2.3(2) indicate that we can transform one line of D_G in the following line through the removal of one component (the first one) and the insertion of another component (finite or not). So we say that any two consecutive lines of D_G are *1-variant*. Any $r + 1$ consecutive lines of D_G we have that they are *r-variant*, because from any of these lines we can obtain another one by the removal and insertion of r components at the most.

This suggests a representation for D_G in space $O(n_a + n_b)$. This representation is composed by D_G^0 (the first line of D_G), that has size $n_a + 1$, and a vector that has size n_b, defined below.

Definition 2.3 (Vector V_G) *For $1 \leq i \leq n_b$, $V_G(i)$ is the value of the finite component that is present in line D_G^i but not in line D_G^{i-1}. If there is no such component, then $V_G(i) = \infty$.*

Table 3 (page 10) shows V_G for the GDAG of Figure 2. It is worthy to compare the three representations already commented: by C_G (Table 1), by D_G (Table 2) and this last one.

The structure formed by V_G and D_G^0 is important to the parallel algorithm, since the main strategy involves the division of the original GDAG in strips and the resolution of the ALCS in each strip. The strips are then united, which requires the transmission of the

	j													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$C_G(0,j)$	0	1	2	3	4	5	6	6	7	8	8	8	8	8
$C_G(1,j)$	0	0	1	2	3	4	5	5	6	7	7	7	7	7
$C_G(2,j)$	0	0	0	1	2	3	4	4	5	6	6	6	6	7
$C_G(3,j)$	0	0	0	0	1	2	3	3	4	5	5	6	6	7
$C_G(4,j)$	0	0	0	0	0	1	2	2	3	4	4	5	5	6
$C_G(5,j)$	0	0	0	0	0	0	1	2	3	4	4	5	5	6
$C_G(6,j)$	0	0	0	0	0	0	0	1	2	3	3	4	4	5
$C_G(7,j)$	0	0	0	0	0	0	0	0	1	2	2	3	3	4
$C_G(8,j)$	0	0	0	0	0	0	0	0	0	1	2	3	3	4
$C_G(9,j)$	0	0	0	0	0	0	0	0	0	0	1	2	3	4
$C_G(10,j)$	0	0	0	0	0	0	0	0	0	0	0	1	2	3
$C_G(11,j)$	0	0	0	0	0	0	0	0	0	0	0	0	1	2
$C_G(12,j)$	0	0	0	0	0	0	0	0	0	0	0	0	0	1
$C_G(13,j)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 1: C_G corresponding to the GDAG of Figure 2.

	k								
	0	1	2	3	4	5	6	7	8
$D_G(0,k)$	0	1	2	3	4	5	6	8	9
$D_G(1,k)$	1	2	3	4	5	6	8	9	∞
$D_G(2,k)$	2	3	4	5	6	8	9	13	∞
$D_G(3,k)$	3	4	5	6	8	9	11	13	∞
$D_G(4,k)$	4	5	6	8	9	11	13	∞	∞
$D_G(5,k)$	5	6	7	8	9	11	13	∞	∞
$D_G(6,k)$	6	7	8	9	11	13	∞	∞	∞
$D_G(7,k)$	7	8	9	11	13	∞	∞	∞	∞
$D_G(8,k)$	8	9	10	11	13	∞	∞	∞	∞
$D_G(9,k)$	9	10	11	12	13	∞	∞	∞	∞
$D_G(10,k)$	10	11	12	13	∞	∞	∞	∞	∞
$D_G(11,k)$	11	12	13	∞	∞	∞	∞	∞	∞
$D_G(12,k)$	12	13	∞	∞	∞	∞	∞	∞	∞
$D_G(13,k)$	13	∞	∞	∞	∞	∞	∞	∞	∞

Table 2: D_G corresponding to the GDAG of Figure 2.

	k												
	1	2	3	4	5	6	7	8	9	10	11	12	13
$V_G(k)$	∞	13	11	∞	7	∞	∞	10	12	∞	∞	∞	∞

Table 3: V_G corresponding to the GDAG of Figure 2. V_G and D_G^0 (first line from Table 2) can be used together to reconstruct D_G.

solutions of the partial ALCS problems between processors. A compact representation for these solutions makes the communication steps smaller.

In the following section we present the sequential algorithm for the ALCS.

3 Sequential Algorithm for the ALCS

The algorithm presented in this section is based on another one by Schmidt [18] that deals with a similar problem, but with more flexibility with the weights of the arcs in the GDAG.

Our presentation is more complete and deals with the particular case of the ALCS. For this presentation, several additional properties will be shown. These properties are used only in this section, so the reading of this section is not needed to the comprehension of the rest of the text. The one thing that must be known is that the ALCS problem can be solved sequentially for strings of lengths n_a and n_b in $O(n_a n_b)$ time and $O(n_a + n_b)$ space (or $O(n_a n_b)$ space, if the actual common sequence is required besides its length).

3.1 Additional Properties of the ALCS Problem

For each vertex of the GDAG G we need to determine some information about the distance between this vertex and each vertex on the top row of G. For this we have the following definition:

Definition 3.1 ($C_G^l(i,j)$) *For $0 \leq l \leq n_a$, $0 \leq i \leq n_b$ and $0 \leq j \leq n_b$, $C_G^l(i,j)$ is the total weight of the largest path between vertices $(0,i)$ and (l,j), if there is such path. If there is no such path (in the case that $i > j$), $C_G^l(i,j) = 0$.*

This definition is an extension of Definition 2.1 that deals with the internal vertices of G (notice that $C_G(i,j) = C_G^{n_a}(i,j)$). Now we have the following properties, that deal with neighboring vertices in G.

Properties 3.1 *For all l ($0 \leq l \leq n_a$) we have:*

1. *For all i ($0 \leq i \leq n_b$) and all j ($1 \leq j \leq n_b$) we have $C_G^l(i,j) = C_G^l(i,j-1)$ or $C_G^l(i,j) = C_G^l(i,j-1) + 1$.*
2. *For all i and all j ($0 < i < j \leq n_b$) we have that if $C_G^l(i-1,j) = C_G^l(i-1,j-1) + 1$ then $C_G^l(i,j) = C_G^l(i,j-1) + 1$.*

The demonstration of these properties is omitted, for it is similar to the one already presented to Property 2.1. The inclusion of l in these properties does not affect their nature.

Properties 3.1 are related to the differences between distances to two internal vertices of G that are neighbors in the same *line*. The following properties are related to internal vertices that are neighbors in the same *column*.

Properties 3.2 *For all l ($1 \leq l \leq n_a$) we have:*

1. *For all i ($0 \leq i \leq n_b$) and all j ($0 \leq j \leq n_b$) we have $C_G^l(i,j) = C_G^{l-1}(i,j)$ or $C_G^l(i,j) = C_G^{l-1}(i,j) + 1$.*
2. *For all i and all j ($0 < i \leq j \leq n_b$) we have that if $C_G^l(i-1,j) = C_G^{l-1}(i-1,j)$ then $C_G^l(i,j) = C_G^{l-1}(i,j)$.*

The demonstration of these properties is also omitted.

A consequence of Properties 3.1 and 3.2 is that it is possible to encode *variations* in the weights of the paths to internal vertices of the GDAG in an efficient way. If we pick a vertex in the superior line of G, say $T_G(i)$, and determine the best possible paths from it to two neighboring vertices in a line, say $(l, j-1)$ and (l, j), these paths will have the same weight if i is below a certain limit. For values of i equal or above this limit, the weight of the path to (l, j) will be larger (by 1) than the weight of the path to $(l, j-1)$. The value of this limit depends on l and j and will be called $i_h(l, j)$.

In a similar way, the weights of the best paths from $T_G(i)$ to neighboring vertices in a column, $(l-1, j)$ and (l, j), differ by 1 if i is *below* a certain limit $i_v(l, j)$. In the limit or above it, the paths have the same weight.

Formally, we define:

Definition 3.2 ($i_h(l, j)$) *For all (l, j), $0 \leq l \leq n_a$ and $1 \leq j \leq n_b$, $i_h(l, j)$ is the smallest value of $i < j$ such that $C^l_G(i, j) = C^l_G(i, j-1) + 1$. If there is no such i, $i_h(l, j) = j$.*

Definition 3.3 ($i_v(l, j)$) *For all (l, j), $1 \leq l \leq n_a$ and $1 \leq j \leq n_b$, $i_v(l, j)$ is the smallest value of $i \leq j$ such that $C^l_G(i, j) = C^{l-1}_G(i, j)$.*

The algorithm for the ALCS is based on the determination of the limits i_h and i_v for all the vertices of the GDAG. Before we proceed, we make an observation on i_h: the existence of this limit was already hinted in Property 2.2(3). The first value of i for which there is a k such that $D_G(i, k) = j$ is $i_h(n_a, j)$. For values of i above this limit, j continues to appear in D^i_G, which means that there is a difference between the best paths from $T_G(i)$ to $(n_a, j-1)$ and (n_a, j).

Besides i_h and i_v, for each vertex we need to determine two limits, i_1 and i_2, explained below. Given a vertex (l, j), with $0 < l \leq n_a$ and $0 < j \leq n_b$, the best paths from the top of the GDAG to it must have $(l, j-1)$, $(l-1, j-1)$ or $(l-1, j)$ as the next-to-last vertex. If we look for the leftmost paths, we see that they have the following property:

Property 3.3 *For all (l, j), $1 \leq l \leq n_a$ and $0 < j \leq n_b$, there are values i_1 and i_2 such that the best (leftmost) path from $T_G(i)$ to (l, j) has as the next-to-last vertex:*

- $(l, j-1)$ *if* $0 \leq i < i_1$,
- $(l-1, j-1)$ *if* $i_1 \leq i < i_2$,
- $(l-1, j)$ *if* $i_2 \leq i < j$.

Figure 4 ilustrates Property 3.3.

The demonstration of this property is based on the impossibility of crossing of two best-leftmost paths to a single vertex. The details are left to the reader.

3.2 Description and Analysis of the Sequential Algorithm

The determination of the four forementioned limits (i_h, i_v, i_1 and i_2) is done in a vertex-by vertex basis, sweeping the GDAG by lines (or columns). To calculate the limits for a vertex (l, j) it is necessary to know only $i_h(l-1, j)$ and $i_v(l, j-1)$, for these values indicate differences between paths to vertices adjacent to (l, j). We will call these vertices *candidate vertices*. Two distinct cases must be considered separately.

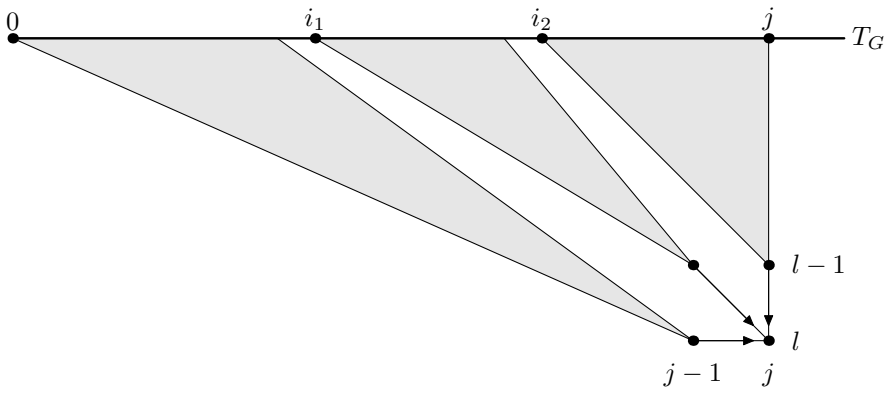

Figure 4: Next-to-last vertex of each path in a GDAG. From the top of the GDAG, the paths that arrive in an interior vertex pass through the adjacent vertices as shown.

Case 1 If $a_l \neq b_j$, that is, the arc from $(l-1, j-1)$ to (l, j) has weight 0, this arc can be ignored and the next-to-last vertex of any path to (l, j) cannot be $(l-1, j-1)$. In this case, $i_1 = i_2$. There are two possibilities from this point:

Case 1.i $i_v(l, j-1) \leq i_h(l-1, j)$: there are three ranges of values to consider for i in the choice of the next-to-last vertex in the path to (l, j):

- for $0 \leq i < i_v(l, j-1)$ the best path from $T_G(i)$ to $(l, j-1)$ is better than the best path to the other two candidates, so the chosen vertex is $(l, j-1)$.
- $i_v(l, j-1) \leq i < i_h(l-1, j)$: the best paths to each of the three candidates have the same weight, so the chosen candidate is again $(l, j-1)$ (which gives the leftmost path).
- $i_h(l-1, j) \leq i \leq j$: the best path from $T_G(i)$ to $(l-1, j)$ is better than the path to the other two candidates, so $(l-1, j)$ is chosen.

So, $i_1 = i_2 = i_h(l-1, j)$, $i_h(l, j) = i_h(l-1, j)$ and $i_v(l, j) = i_v(l, j-1)$.

Case 1.ii $i_v(l, j-1) > i_h(l-1, j)$: there are again three ranges of values to consider for i.

- for $0 \leq i < i_h(l-1, j)$: the chosen vertex is $(l, j-1)$.
- $i_h(l-1, j) \leq i < i_v(l, j-1)$: the best paths from $T_G(i)$ to $(l, j-1)$ and to $(l-1, j)$ have the same weight (the path to $(l-1, j-1)$ has a lower weight). The leftmost vertex $(l, j-1)$ is chosen.
- $i_v(l, j-1) \leq i \leq j$: the chosen vertex is $(l-1, j)$.

So we have $i_1 = i_2 = i_v(l, j-1)$, $i_h(l, j) = i_v(l, j-1)$ and $i_v(l, j) = i_h(l-1, j)$.

In Case 1 we have $i_1 = i_2 = i_h(l, j) = \max(i_v(l, j-1), i_h(l-1, j))$ and $i_v(l, j) = \min(i_v(l, j-1), i_h(l-1, j))$.

Case 2 If $a_l = b_j$, that is, the arc from $(l-1, j-1)$ to (l, j) has weight 1, We have to consider the three candidates, but $(l-1, j-1)$ has the advantage of being connected to (i, j) through an arc of weight 1. In fact, as we search for the leftmost path among the best

ones to (i, j), no path will pass through $(l-1, j)$ except the one from $T_G(j)$, and so we have $i_2 = j$.

When $i < i_v(l, j-1)$ the best path from $T_G(i)$ to $(l, j-1)$ has a lower weight than the one to $(l-1, j-1)$, compensating the weight 1 of the arc from $(l-1, j-1)$ to (l, j). So, $i_1 = i_v(l, j-1)$ and $i_h(l, j) = i_v(l, j-1)$. For similar reasons, we have $i_v(l, j) = i_h(l-1, j)$.

The preceding calculi apply to the vertices that are not in the upper and left borders of the GDAG. For the vertices in the upper border we have $i_h(0, j) = j$ $(1 \leq j \leq n_b)$ and for the vertices in the left border we have $i_v(l, 0) = 0$ $(1 \leq l \leq n_a)$. The other values are not important.

From the values of $i_h(n_a, j)$ with $1 \leq j < n_b$ (related to the vertices in the lower border of the GDAG) it is possible to determine the solution for the ALCS in the encoding that we are using in this report (based on D^0_G and V_G).

At first we make $D^0_G(0) = 0$. Then, if for a certain j we have $i_h(n_a, j) = 0$, this means that j is a component of D^0_G. If $i_h(n_a, j) = i, 1 \leq i \leq j$, this means that the first line of D_G in which j appears is D^i_G, so $V_G(i) = j$.

Algorithm 3.1 makes this procedure explicit. Notice that the limits i_1 and i_2 are not necessary for the determination of D^0_G and V_G, so these limits are not included in the algorithm. However, if it is necessary to make possible the fast retrieval of the actual paths between $T_G(i)$ and $F_G(j)$ (for any i and j), then the determination and storage of the limits i_1 and i_2 are necessary. These limits allow the reconstruction of the path, starting at $F_G(j)$ and going back to $T_G(i)$, in time $O(n_a + n_b)$.

Algorithm 3.1: **Sequential ALCS.**
Input: Strings $A = a_1 a_2 \ldots a_{n_a}$ and $B = b_1 b_2 \ldots b_{n_b}$.
Output: Vectors D^0_G and V_G related to the strings A and B.

1 For $j \leftarrow 0$ to n_b do
1.1 $i_h(0, j) \leftarrow j$
2 For $l \leftarrow 0$ to n_a do
2.1 $i_v(l, 0) \leftarrow 0$
3 For $l \leftarrow 1$ to n_a do
3.1 For $j \leftarrow 1$ to n_b do
3.1.1 If $a_l \neq b_j$ then
3.1.1.1 $i_h(l, j) \leftarrow \max(i_v(l, j-1), i_h(l-1, j))$
3.1.1.2 $i_v(l, j) \leftarrow \min(i_v(l, j-1), i_h(l-1, j))$
3.1.2 Else
3.1.2.1 $i_h(l, j) \leftarrow i_v(l, j-1)$
3.1.2.2 $i_v(l, j) \leftarrow i_h(l-1, j)$
4 For $j \leftarrow 1$ to n_b do
4.1 $V_G(j) \leftarrow \infty$
5 $D^0_G(0) \leftarrow 0$
6 $i \leftarrow 1$
7 For $j \leftarrow 1$ to n_b do
7.1 If $i_h(n_a, j) = 0$ then

7.1.1 $D^0_G(i) \leftarrow j$

7.1.2 $i \leftarrow i + 1$

7.2 Else

7.2.1 $V_G(i_h(n_a, j)) \leftarrow j$

8 For $l \leftarrow i$ to n_a do

8.1 $D^0_G(l) \leftarrow \infty$

end of the algorithm.

An example of the results of this algorithm can be seen in Table 4, where the values of i_h and i_v are shown for each vertex of the GDAG of Figure 2 (page 6). The results in the last line of the table (related to $i_h(n_a, j)$) lead correctly to the results shown in Tables 2 and 3 (page 10).

		j													
		b_j													
		0	1	2	3	4	5	6	7	8	9	10	11	12	13
			y	x	x	y	z	x	y	z	x	y	x	z	x
l	a_l														
	0	—	—	—	—	—	—	—	—	—	—	—	—	—	—
		—	1	2	3	4	5	6	7	8	9	10	11	12	13
	1 y	0	1	1	1	4	4	4	7	7	7	10	10	10	10
		—	0	2	3	1	5	6	4	8	9	7	11	12	13
	2 x	0	0	2	3	1	1	6	4	4	9	7	11	11	13
		—	0	0	2	3	5	1	6	8	4	9	7	12	11
	3 x	0	0	0	2	2	2	1	1	1	4	4	7	7	11
		—	0	0	0	3	5	2	6	8	1	9	4	12	7
	4 y	0	0	0	0	3	3	2	6	6	1	9	4	4	4
		—	0	0	0	0	5	3	2	8	6	1	9	12	7
	5 z	0	0	0	0	0	5	3	2	8	6	1	1	12	7
		—	0	0	0	0	0	5	3	2	8	6	9	1	12
	6 y	0	0	0	0	0	0	0	3	2	2	6	6	1	1
		—	0	0	0	0	0	5	0	3	8	2	9	6	12
	7 z	0	0	0	0	0	0	0	0	3	3	2	2	6	6
		—	0	0	0	0	0	5	0	0	8	3	9	2	12
	8 x	0	0	0	0	0	0	5	0	0	8	3	9	2	12
		—	0	0	0	0	0	0	5	0	0	8	3	9	2

Table 4: Partial results of the execution of Algorithm 3.1 for the GDAG in Figure 2. In each cell, the upper number represents $i_v(l, j)$ and the lower $i_h(l, j)$.

Theorem 3.1 *Given two strings A and B, of lengths n_a and n_b respectively, it is possible to solve (sequentially) the ALCS problem for A and B in time $O(n_a n_b)$ and space $O(n_b)$ (or $O(n_a n_b)$ to allow the retrieval of the actual best paths).*

Proof. The execution time is clearly defined by the nested loops in lines 3 and 3.1, being $O(n_a n_b)$. In the case where only the distances are important, we can discard the values of $i_h(l, j)$ after the line $l + 1$ has been processed. So, to use the values $i_h(l, j)$, $O(n_b)$ space

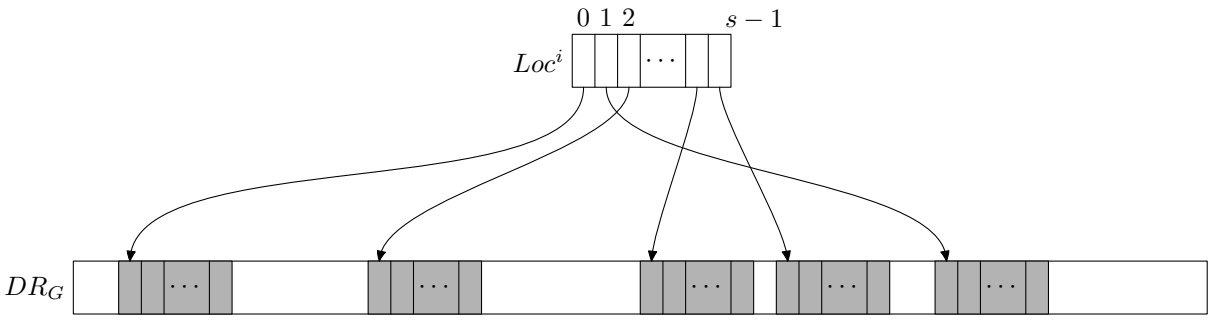

Figure 5: Storage of data from D_G^i in DR_G.

is required. The value of $i_v(l, j)$ must be kept only to the calculation of $i_v(l, j+1)$, so the space required by the values of $i_v(l, j)$ is only $O(1)$.

The storage of the resulting vectors D_G^0 (space $O(n_a)$) and V_G (space $O(n_b)$) requires $O(n_a + n_b)$ space. In fact, D_G^0 can be stored in $O(n_b)$ space, because if $n_a > n_b$ all values of $D_G^0(k)$ are infinite for $k > n_b$. For simplicity, we will consider $n_a \leq n_b$.

If the further retrieval of the paths is necessary, the necessary space becomes $O(n_a n_b)$, for the storage of i_1 and i_2 for all the vertices. □

The presentation of the sequential algorithm is concluded. Before we proceed to the presentation of the CGM algorithm we will show a data structure that allows fast queries of the components of D_G and occupies less storage space than D_G.

4 Compact Structure for D_G

Until now we are considering two possible representations for matrix D_G. The "direct" representation has too much redundancy and so uses too much space ($O(n_a n_b)$), but given i and k $D_G(i, k)$ may be obtained in time $O(1)$. On the other hand, the "incremental" representation, given by vectors D_G^0 and V_G, uses only $O(n_b)$ space but does not allow fast accesses.

In this section we present an efficient data structure for the storage of D_G. This structure uses space $O\left(\sqrt{n_a} n_b\right)$, allows data retrieval in time $O(1)$ and can be constructed in time $O\left(\sqrt{n_a} n_b\right)$ given D_G^0 and V_G. This structure is essential to the CGM algorithm presented later.

One of the main parameters of this data structure is $s = \lceil \sqrt{n_a + 1} \rceil$.

The data from D_G is distributed in a vector DR_G (*reduced* D_G) of size $O(n_b s)$. Let us consider first one single line of D_G. D_G^i $(0 \leq i \leq n_b)$ has size $n_a + 1$. It is divided in s subvectors, all of size s with the possible exception of the last one. These subvectors are stored in separate locations of DR_G, so an additional vector of size s is necessary to keep the location of each subvector (more exactly, the position of the first element of each subvector). Let us call this additional (line) vector Loc^i. The $(n_b + 1) \times s$ matrix formed by all vectors Loc^i (one for each line D_G^i) is called *Loc*. The indices of *Loc* start at 0. This structure is shown in Figure 5.

The complete data structure is formed by vector DR_G and matrix *Loc*. Before we proceed with the description, we state the following lemma based on what was already presented:

Lemma 4.1 *Through DR_G and Loc any element of D_G may be read in time $O(1)$.*

Proof. To locate one element of $D_G(i,k)$ in DR, first we calculate in which subvector it is stored. It is easy to see that the initial position of the subvector is given by $Loc(i, \lfloor k/s \rfloor)$. The displacement of the element inside the vector is (k mod s), so $D_G(i,k) = DR_G(Loc(i, \lfloor k/s \rfloor) + k \bmod s)$. All the operations may be done in time $O(1)$. □

Now we describe the construction of DR_G and Loc. D_G^0 is the first line of D_G to be included in DR_G. Each subvector of size s of D_G^0 is allocated in a space of size $2s$ in DR_G, leaving a free extra space of size s after the stored data of each subvector. This extra space is used by the next lines of D_G. So, vector Loc^0 is filled with the first multiples of $2s$.

The construction is done incrementally, including the data of one line of D_G at a time. What makes possible for DR_G to keep the data from D_G in just $O(n_b s)$ space is that the subvectors of different lines of D_G may overlap. If the data from line D_G^i are already in the structure, the inclusion of data from D_G^{i+1} is done by the insertion of just a few elements.

As already shown in Property 2.3 and in Definition 2.3, the only difference between lines D_G^i and D_G^{i+1} is that D_G^{i+1} does not have the value $D_G^i(0) = i$ but may have a value that is not in D_G^i, given by $V_G(i+1)$. The construction of the representation of D_G^{i+1} is as follows:

1. We determine which of the s subvectors of D_G^i should receive element $V_G(i+1)$ to transform D_G^i into D_G^{i+1}. Let v be the index of this subvector.

2. We determine the position in this subvector where $V_G(i+1)$ should be inserted.

3. All suvectors of D_G^{i+1} with index *above* v are equal to the ones of D_G^i, so there is no need to rewrite them in DR_G. It is sufficient to make $Loc(i+1,j) = Loc(i,j)$ for $v < j < s$.

4. All subvectors of D_G^{i+1} with index *below* v are equal to the ones in D_G^i, but for a shift to the left and the inclusion of one new element at the last position (the element that was "thrown out" from the next subvector). The simplest way to shift the subvectors to the left is to make $Loc(i+1,j) = Loc(i,j) + 1$ for $0 \leq j < v$. The new element of each subvector can be copied just to the right of the subvector, in position $Loc(i+1,j) + s - 1$ of DR_G, provided that this position is still free. The details of this operation will be presented shortly.

5. The subvector v of D_G^i would have to be changed in a more complex way, so a new space is allocated in DR_G, and subvector v of D_G^{i+1} is constructed there, with $V_G(i+1)$ already in position. $Loc(i+1,v)$ indicates the position of this new space.

This procedure is shown in Figure 6.

Each time a new line is included in DR_G, a new free space of size between 0 and s is left to the right of each of the s subvectors. In the inclusion of the next line, part of this free space may be consumed, meaning that there will be less space for the next lines to be included. Eventually, the previously described procedure will not work due to the lack of space to perform step 4. When this happens, a totally new space must be allocated in DR_G and the data of the subvector must be copied there.

So, in the inclusion of D_G^{i+1} there are two cases where the algorithm requires the allocation of a new space and copy of a subvector:

1. The subvector is the one that should contain $V_G(i+1)$. It is copied into a new space with $V_G(i+1)$ already inserted.

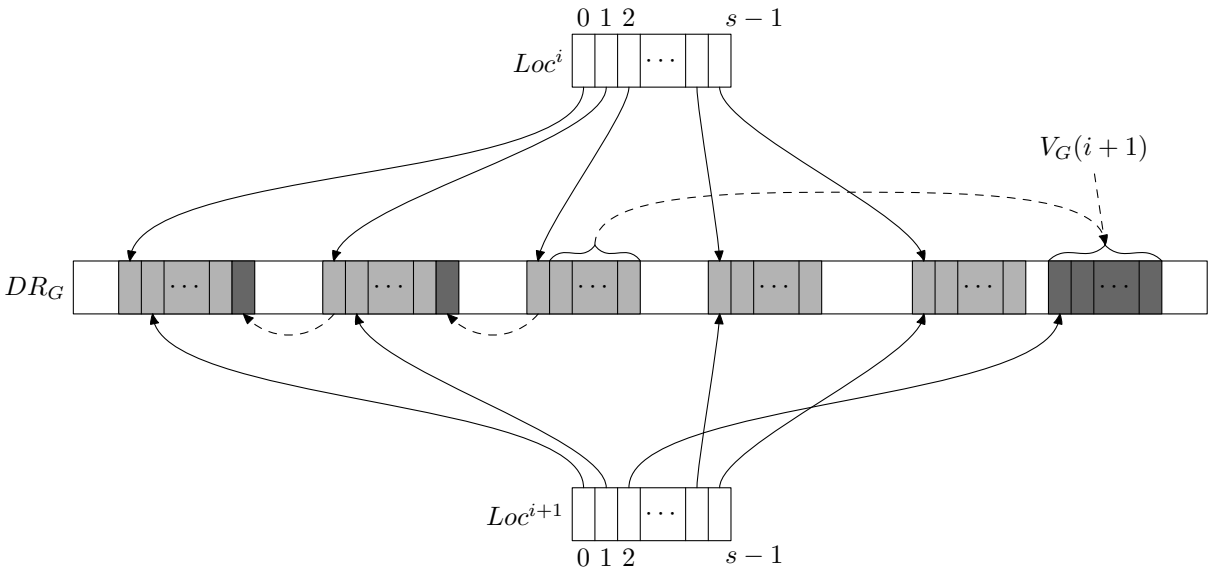

Figure 6: Inclusion of D_G^{i+1} in DR_G using data related to D_G^i. In this example, element $V_G(i+1)$ must be inserted in the subvector of index 2. The darker areas represent the data written in DR_G in this step. The dashed arrows indicate data copying.

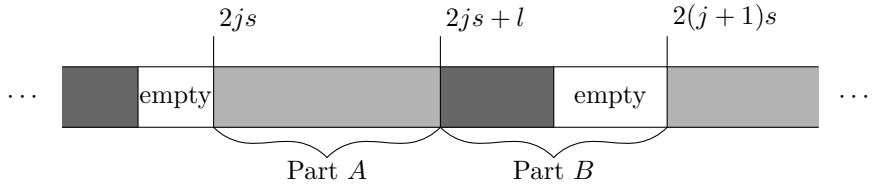

Figure 7: Allocation unit in DR_G. Part A (size s) is filled with data right after the unit is allocated. Part B (also of size s) is filled with data one position at a time as new lines of D_G are included. When filled, Part B is copied into Part A of a new allocation unit.

2. It is not possible to make a left shift of a subvector of D_G^i due to the lack of space for the inclusion of the extra element at its end. The subvector of D_G^{i+1}, already shifted and with the new element, is copied into a new space.

Each new space allocated has size $2s$, and the first s receive the copied subvector. The extra space is used for the inclusion of the next lines. Each new line may require a position in this extra space. The extra space is totally ocuppied when the next position to be used has index multiple of $2s$. This allocation scheme is shown in Figure 7.

Algorithm 4.1 contains the complete description of the construction of the structure. Variable *new* indicates the next space of length $2s$ that will be allocated in DR_G.

Algorithm 4.1: **Construction of the compact representation of D_G.**
Input: Vectors D_G^0 and V_G.
Output: Vector DR_G and matrix *Loc*.

1 $new \leftarrow 0$

2 (* Insert the data from D_G^0 in the structure *)
For $v \leftarrow 0$ to $s-1$ do

2.1 $Loc^0(v) \leftarrow new$

2.2 (* Insert the data of the subvector of index v *)
For $k \leftarrow vs$ to $\min\{n_a, (v+1)s - 1\}$ do

2.2.1 $DR_G(Loc^0(v) + k - vs) \leftarrow D_G^0(k)$

2.3 $new \leftarrow new + 2s$

3 (* Insert the other lines of D_G *)
For $i \leftarrow 0$ to $n_b - 1$ do

3.1 (* Find the subvector that must receive $V_G(i+1)$ *)
$v \leftarrow \min\limits_{1 \le j \le s} \{j | j = s \text{ or } DR_G(Loc^i(j)) > V_G(i+1)\} - 1$

3.2 (* Find the insertion position of $V_G(i+1)$ in the subvector *)
$r \leftarrow \max\limits_{1 \le j < s} \{j | DR_G(Loc^i(v) + j) < V_G(i+1)\}$

3.3 (* allocates a new subvector *)
$Loc^{i+1}(v) \leftarrow new$

3.4 (* transfer the data that is to the left of position r, shifting to the left *)
For $j \leftarrow 0$ to $r - 1$ do

3.4.1 $DR_G(new + j) \leftarrow DR_G(Loc^i(v) + j + 1)$

3.5 (* Insert $V_G(i+1)$ *)
$DR_G(new + r) \leftarrow V_G(i+1)$

3.6 (* transfer the data that is to the right of position r *)
For $j \leftarrow r + 1$ to $s - 1$ do

3.6.1 $DR_G(new + j) \leftarrow DR_G(Loc^i(v) + j)$

3.7 $new \leftarrow new + 2s$

3.8 (* sharing subvectors of D_G^i and D_G^{i+1} above v *)
For $j \leftarrow v + 1$ to $s - 1$ do

3.8.1 $Loc^{i+1}(v) \leftarrow Loc^i(v)$

3.9 (* below v, share subvectors if possible *)
For $j \leftarrow 0$ to $v - 1$ do

3.9.1 (* verify if the extra space after the subvector is over *)
If $Loc^i(j) + s \equiv 0 \pmod{2s}$ then

3.9.1.1 (* allocate new subvector and copy data, shifting to the left *)
$Loc^{i+1}(j) \leftarrow new$

3.9.1.2 For $t \leftarrow 0$ to $s - 2$ do

3.9.1.2.1 $DR_G(new + t) \leftarrow DR_G(Loc^i(j) + t + 1)$

3.9.1.3 $new \leftarrow new + 2s$

3.9.2 Else

3.9.2.1 $Loc^{i+1}(j) \leftarrow Loc^i(j) + 1$

3.9.3 (* copy the first element of the next subvector to the end of this one *)
$DR_G(Loc^{i+1}(j) + s - 1) \leftarrow DR_G(Loc^i(j))$

end of the algorithm.

Lemma 4.2 *From D_G^0 and V_G, a representation of D_G based on DR_G and Loc can be constructed in $O(n_b s)$ time using $O(n_b s)$ space.*

Proof. Since there are $n_b + 1$ lines in D_G and each one requires a line of size s in *Loc*, it is clear that *Loc* uses $O(n_b s)$ space. Let us concentrate on the space used by DR_G.

The information contained in D_G^0 is processed in step 2 in $O(s^2)$ time and uses $O(s^2)$ space. Then, the loop in step 3 inserts the data of each additional line of D_G in n_b iteractions. It remains to be proved that on average each iteration adds $O(s)$ data to DR_G and executes in $O(s)$ time. Let us analyze each step of this loop.

Steps 3.1 and 3.2 can be done with binary searches in $O(\log s)$ time. The steps 3.3 to 3.6 involve only loops that execute $O(s)$ iteractions of constant time, so they take $O(s)$ time. A new allocation unit of size $2s$ is allocated and s data are copied. Step 3.8 takes $O(s)$ time and does not add new data to DR_G.

The analysis of step 3.9 is a little different. In fact, the loops of lines 3.9 and 3.9.1.2 together may need time and space larger than $O(s)$. This may occur if several subvectors get out of extra space at the same time.

We proceed with an amortized analysis. Every time it is needed to copy a subvector into another in step 3.9.1.2, the copy is done from Part B of an allocation unit to Part A of another allocation unit that was recently allocated (see Figure 7). If the cost of the copy of each element is counted "previously" at the moment that the element was first written in Part B (in step 3.9.3), we may consider the cost of step 3.9.1.2 as null. On the other hand, step 3.9.3 will have its cost doubled, but this does not affects our results. So, the (amortized) time of step 3.9 is $O(s)$. The (amortized) quantity of added data is also $O(s)$.

So, all the steps inside loop 3 require $O(s)$ time and space. The whole loop requires $O(n_b s)$ time and space. □

Some improvements can be made in Algorithm 4.1. For example, the case where $V_G(i+1) = \infty$ may be treated in a special way. Anyway, these improvements would not affect the results of our analysis, so they are ignored for simplicity.

The results of this section may be summarized in the following theorem, a direct consequence of Lemmas 4.1 and 4.2:

Theorem 4.3 *Let G be the GDAG of the ALCS problem for strings A and B. From D_G^0 and V_G it is possible to construct a representation of D_G in $O\left(\sqrt{n_a} n_b\right)$ time and space such that any value of D_G may be read in constant time.*

5 Outline of the CGM Algorithm for the ALCS

In this section a parallel CGM algorithm for the ALCS will be shown. With p processors, the algorithm requires $O(\frac{n_a n_b}{p})$ time and $O(C \log p)$ communication steps. During each communication step, $O(n_a p^{1/C} + n_b)$ data needs to be sent/received by each processor. In these expressions, C is a positive integer constant to be chosen (small values like 1 or 2 are the most probable choices).

The time complexity of this algorithm is adequate to the solution of the basic LCS problem. Let us suppose, for simplicity and with no loss of generality in the following complexity analysis, that n_a is a multiple of p and p is a power of 2.

The algorithm involves the division of string A in p non-overlapping substrings of size $\frac{n_a}{p}$. For $1 \leq t \leq p$, processor P_t sequentially solves the ALCS problem for the strings $A_{n_a(t-1)/p+1}^{n_a t/p}$ and B. The GDAG of the original problem is divided horizontally in strips, generating p GDAGs with $\frac{n_a}{p} + 1$ lines. Two contiguos strips share a line.

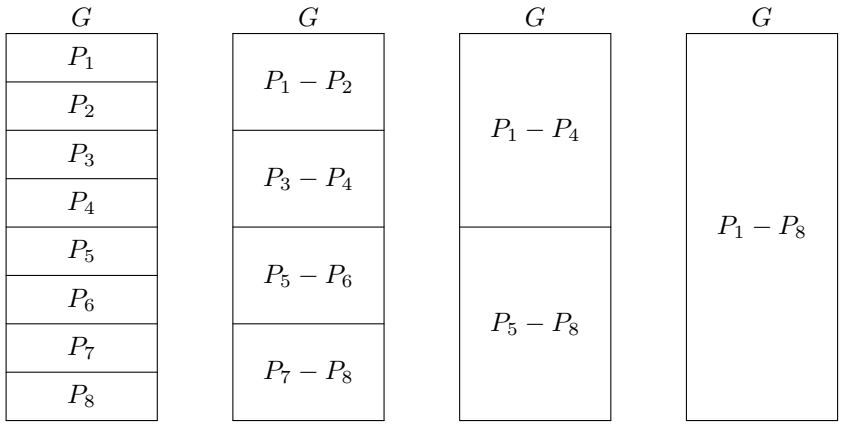

Figure 8: Union of partial solutions for the ALCS, with $p = 8$. In each strip of GDAG G, the processors used in the solution are shown.

The sequential algorithm used in this initial step is shown in Section 3. The time needed for the p processors to solve all the subproblems in parallel is $O\left(\frac{n_a n_b}{p}\right)$, as shown in Theorem 3.1.

Next, the p partial solutions are united in a binary tree fashion, using $\log p$ parallel steps. The basic operation is the union of two neighboring strips to form a merged strip. More exactly, the solution for the ALCS problem in the original strips are used to generate the solution for the merged strip. At each step, several unions are done in parallel. Each union is itself a parallel process. At each step, the number of strips are halved, the size of the strips are doubled and the number of processors allocated to each union is also doubled. After $\log p$ union steps we obtain the solution for the whole problem. The total parallel time of all this steps is $O\left(n_b\sqrt{n_a}\left(1+\frac{\log n_a}{\sqrt{p}}\right)\right)$, as will be shown in Section 7. Figure 8 illustrates this process.

The most complex part of algorithm involves the union process of two strips. This process is explained in Section 6.

6 Union of Partial Solutions

In this section we will see how to solve the ALCS problem for two strings A_1^{2m} and B, of lengths $2m$ and $n = n_b$ respectively, from the ALCS solutions for A_1^m and B and for A_{m+1}^{2m} and B. The GDAGs related to the first and second subproblems will be called S (superior) and I (inferior), respectively. The GDAG formed by the union of these two GDAGs (by making $F_S = T_I$) will be called U.

Naturally, all the properties already described for the GDAG G also apply to these partial GDAGs. The notation introduced for G will be adapted for these GDAGs in the following descriptions.

As explained in Section 2, the data related to a GDAG G can be stored in a reduced space using vectors D_G^0 and V_G. At the beginning of the union process the solutions for the partial GDAGs will be stored in this compact representation: vectors D_S^0, D_I^0 (with indices between 0 and m), V_S and V_I (with indices between 1 and n).

We will consider that q processors ($2 \leq q \leq p$) will be involved in the determination of D_U^0 (indices between 0 and $2m$) and V_U (indices between 1 and n). Thus, the parameters

for the analysis of the union process are m, n and q. For simplicity, assume $m \leq n$. If $m > n$, the lines of D_S and D_I would contain ∞ in all positions above n, and this would be used to reduce the time and space requirements of the union process.

As shown in Section 4, it is possible to access any element from D_S or D_I in constant time, if we construct (in $O(\sqrt{mn})$ time and space) the data structure of Section 4. Each of the q processors constructs its own copy of these data structures in its local memory for fast accesses.

6.1 Basic Principles of the Union Process

The main idea in use here is the same that was used in [12].

For each i, $0 \leq i \leq n$, we build D_U^i from the available data. As already seen, $D_U^i(k) = D_U(i,k)$ represents the smallest value of j such that $C_U(i,j)$, the weight of the best path from $T_U(i)$ (vertex i at the top of GDAG U) and $F_U(j)$ (vertex j at the bottom (floor) of GDAG U), is k. All paths from $T_U(i) = T_S(i)$ to $F_U(j) = F_I(j)$ have to cross the common border $F_S = T_I$ at some vertex and the total weight of the path is the sum of the weights of the path segments in S and I. So, if we want to determine $D_U(i,k)$ we need to consider for all l, $0 \leq l \leq m$, the paths that cross S with weight l and then cross I with weight $k-l$.

Fixing l, the smallest value of j such that there is a path from $T_U(i)$ to $F_U(j)$ with weight l in S and weight $k-l$ in I is given by $D_I(D_S(i,l), k-l)$, because $D_S(i,l)$ is the first vertex in the common border that is at a distance of l from $T_U(i)$, and $D_I(D_S(i,l), k-l)$ is the first vertex that is at a distance $k-l$ from the chosen border vertex. Properties 2.2 justify this choice.

By the considerations above we have the following:

$$D_U(i,k) = \min_{0 \leq l \leq m} \{D_I(D_S(i,l), k-l)\} \tag{1}$$

It should be noticed that if we keep i fixed and variate k, the lines of D_I that are used are always the same. The variation of k changes just the element that is used in each line.

For each line of D_I used we get an element at a different position, due to the $-l$ term. This "shift" suggests the following definition:

Definition 6.1 (*Shift*$[l, W, c]$) *Given a vector W of length $s+1$ (indices from 0 to s) and an integer l ($0 \leq l \leq c-s$), Shift$[l, W, c]$ is the vector of length $c+1$ (indices from 0 to c) such that:*

$$Shift[l,W,c](i) = \begin{cases} \infty & \text{if } 0 \leq i < l \\ W(i+l) & \text{if } l \leq i \leq s+l \\ \infty & \text{if } s+l < i \leq c \end{cases}$$

In other words, Shift$[l, W, c]$ is vector W shifted to the right by l positions, expanded to size $c+1$ and completed with ∞.

We this definition we may rewrite Equation 1:

$$D_U(i,k) = \min_{0 \leq l \leq m} \{Shift[D_I^{D_S(i,l)}, l, 2m](k)\} \tag{2}$$

Taking all the lines related to a certain value of i we can, using the following definitions, build a matrix that will be used to find *all* elements of D_U^i.

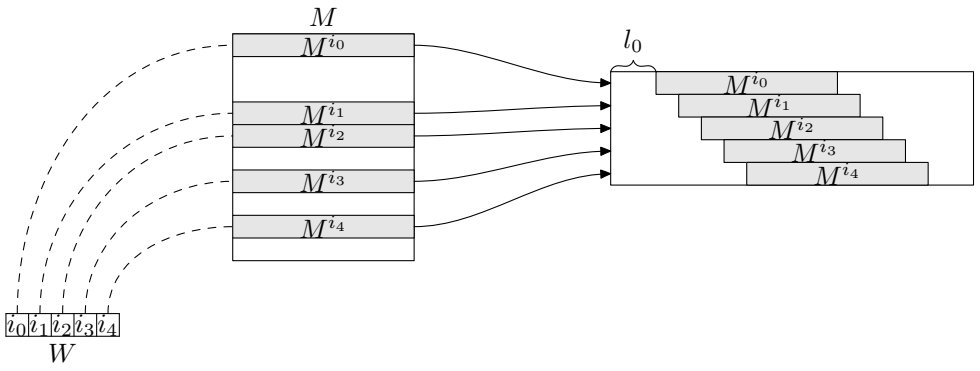

Figure 9: Construction of $Diag[W, M, l_0]$.

	k											
	0	1	2	3	4	5	6	7	8	9	10	11-16
$MD[2](0,k)$	2	3	4	5	6	8	9	13	∞	∞	∞	∞
$MD[2](1,k)$	∞	3	4	5	6	8	9	11	13	∞	∞	∞
$MD[2](2,k)$	∞	∞	4	5	6	8	9	11	13	∞	∞	∞
$MD[2](3,k)$	∞	∞	∞	5	6	7	8	9	11	13	∞	∞
$MD[2](4,k)$	∞	∞	∞	∞	6	7	8	9	11	13	∞	∞
$MD[2](5,k)$	∞	∞	∞	∞	∞	8	9	10	11	13	∞	∞
$MD[2](6,k)$	∞	∞	∞	∞	∞	∞	9	10	11	12	13	∞
$MD[2](7,k)$	∞	∞	∞	∞	∞	∞	∞	13	∞	∞	∞	∞

Table 5: Example MD matrix. Here we have $MD[2]$ considering a GDAG formed by the union of two identical GDAGs as the one in Figure 2.

Definition 6.2 (*$Diag[W, M, l_0]$*) *Let W be a vector (initial index 0) of integers in ascending order such that the first $m'+1$ elements are finite and $W(m') \leq n$. Let M be an $(n+1) \times (m+1)$ matrix (indices starting at 0). $Diag[W, M, l_0]$ is an $(m'+1) \times (2m+1)$ matrix such that its line of index l is $Shift[M^{W(l)}, l + l_0, 2m]$.*

$Diag[W, M, l_0]$ has its lines copied from matrix M. The selection of lines to be copied is guided by vector W. The copies are shifted, in a way that the first element of each line is one column to the right from the first element of the preceding line. The shift amount of the first line is given by l_0. Figure 9 illustrates this construction.

Definition 6.3 (*$MD[i]$*) *Let U be a GDAG for the ALCS problem, formed by the union of GDAGs S (superior) and I (inferior). $MD[U, i] = Diag[D^i_S, D_I, 0]$. If it is clear, from the context, that we refer to U, we will write just $MD[i]$.*

As an example, supposing that a GDAG U is formed from two copies of the same GDAG of Figure 2, then D_S and D_I are equal and given by Table 2. $MD[2]$ has its rows defined by $D^2_I = (2, 3, 4, 5, 6, 8, 9, 13, \infty)$ and its complete structure is presented in Table 5. Notice there is no row defined from $D_I(2, 8) = \infty$.

By the preceding definitions, by solving the Column Minima Problem (see Definition 1.6 in Section 1.4) in $MD[i]$ we find D^i_U. More exactly, if $Cmin[M]$ is the vector of the minima of the columns of a matrix M (Definition 1.3, page 4), we have:

$$D_U^i(k) = Cmin[MD[i]](k) \tag{3}$$

Continuing the example of Table 5, vector D_U^2 would be $(2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, \infty)$.

Theorem 6.1, to be presented shortly, shows that the matrices $MD[i]$ belong to the class of totally monotone matrices (Definition 1.5). In Section 9 the Column Minima Problem (Definition 1.6) for totally monotone matrices is covered, and efficient procedures to solve this problem, originally developed by Aggarwal et al. [1], are shown.

Before we prove that the matrices $MD[i]$ are totally monotone, some details must be given about the infinite elements of these matrices, more exactly were they are located and how we compare two of them. These details were also covered in [12].

Observation 6.1 *Given two infinite elements of a column j of $MD[i]$, $MD[i](l_1, j)$ and $MD[i](l_2, j)$ with $l_1 < l_2$, we will consider that $MD[i](l_1, j) > MD[i](l_2, j)$ if and only if $l_2 < j$, that is, the two elements are above the main diagonal, otherwise, we will consider that $MD[i](l_1, j) < MD[i](l_2, j)$.*

Observation 6.2 *Since the first element of all lines of D_I are finite ($D_I(k, 0) = k$), all the main diagonal of $MD[i]$ is finite. By construction, all elements below this main diagonal are infinite. Also, all finite elements in any line of $MD[i]$ occupy adjacent positions, with eventual infinite elements to the left and to the right.*

Theorem 6.1 *If we do the comparisons according to Observation 6.1, matrix $MD[i]$ is totally monotone.*

Proof. We need to prove that every 2×2 submatrix of $MD[i]$ is monotone, that is, that for any $l_1 < l_2$ and $j_1 < j_2$, if $MD[i](l_2, j_1) < MD[i](l_1, j_1)$ then $MD[i](l_2, j_2) < MD[i](l_1, j_2)$.

Let us see the case where $j_2 = j_1 + 1$ (using two adjacent columns of $MD[i]$). The more general case can be easily proved by induction from this case.

Let us initially suppose that all elements involved in the submatrix are finite. Let $v_1 = D_S(i, l_1)$ and $v_2 = D_S(i, l_2)$. By Property 2.2(1) we have $v_1 < v_2$. Using Definitions 6.1 and 6.3, we need to prove that

$$D_I(v_2, j_1 - l_2) < D_I(v_1, j_1 - l_1) \Rightarrow D_I(v_2, j_1 - l_2 + 1) < D_I(v_1, j_1 - l_1 + 1) \ .$$

Supposing that $D_I(v_2, j_1 - l_2) < D_I(v_1, j_1 - l_1)$ and recalling that the elements of $D_I^{v_2}$ are greater than or equal to v_2 (this follows directly from the definition) we have that $D_I(v_1, j_1 - l_1) > v_2$. So, Property 2.2(3) states that $t = D_I(v_1, j_1 - l_1)$ is a common element of $D_I^{v_1}$ and $D_I^{v_2}$.

It should be noticed that Property 2.2(1) indicates that $D_I^{v_2}$ is in increasing order, so there should be no element in $D_I^{v_2}$ with value between the adjacent elements $D_I(v_2, j_1 - l_2)$ and $D_I(v_2, j_1 - l_2 + 1)$. Since $D_I(v_2, j_1 - l_2) < t$ we have $D_I(v_2, j_1 - l_2 + 1) < t$. Since $t < D_I(v_1, j_1 - l_1 + 1)$ (again by Property 2.2(1)), we conclude that $D_I(v_2, j_1 - l_2 + 1) < D_I(v_1, j_1 - l_1 + 1)$ as expected. Figure 10 illustrates this discussion.

To study the case where there is at least one infinite element in the submatrix, we utilize Observations 6.1 and 6.2. Considering the 4 elements of the submatrix, there are 16 possible arrangements of infinite and finite elements. We have already studied the case where there is no infinite element. We eliminate all the cases where the submatrix is clearly monotone. The remaining cases are shown below.

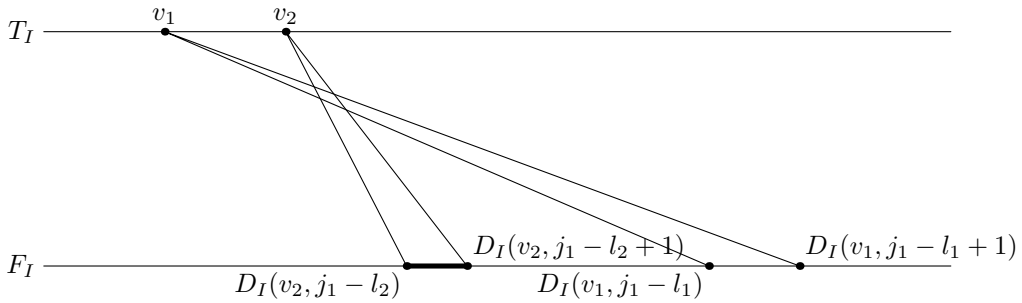

Figure 10: Proof of the monotonicity of a 2×2 submatrix of $MD[i]$ when all elements are finite. This figure illustrates the GDAG I only. The marked interval in F_I is comprehended between two adjacent elements of $D_I^{v_2}$, which prohibits the existence in it of any breaking point to any vertex preceding v_2.

$$\underset{\text{(A)}}{\begin{bmatrix} \infty & a \\ b & c \end{bmatrix}} \quad \underset{\text{(B)}}{\begin{bmatrix} \infty & a \\ b & \infty \end{bmatrix}} \quad \underset{\text{(C)}}{\begin{bmatrix} \infty & \infty \\ b & \infty \end{bmatrix}} \quad \underset{\text{(D)}}{\begin{bmatrix} \infty & a \\ \infty & \infty \end{bmatrix}} \quad \underset{\text{(E)}}{\begin{bmatrix} \infty & a \\ \infty & c \end{bmatrix}} \quad \underset{\text{(F)}}{\begin{bmatrix} \infty & \infty \\ \infty & \infty \end{bmatrix}}$$

By Observation 6.2 cases (A) and (B) cannot occur. Case (C) can only occur if the second column of the submatrix is formed by elements that are above the main diagonal of $MD[i]$, so the minimum of the second column is in the second line (Observation 6.1) and the submatrix is monotone.

The following cases also involve Observation 6.1. In cases (D) and (E) the first column contains elements that are below the main diagonal of $MD[i]$, so the minimum of this column is in the first line and the submatrix is monotone (in fact, case (E) is not possible when we consider only submatrices that are taken from adjacent columns of $MD[i]$). In case (F) the submatrix is monotone because if the first column contains elements that are above the main diagonal of $MD[i]$ the same will happen with the elements of the second column.

So, all 2×2 submatrices of $MD[i]$ are monotone and $MD[i]$ is totally monotone □

Given that all matrices of $MD[i]$ are totally monotone, the union of the GDAGs may be done through the search of the minima of their columns through Algorithm 9.4 (see Section 9.1). This algorithm can make these searches in $O(m)$ time for each of these $n+1$ matrices, totaling $O(nm)$ time. This is not good enough.

To solve the problem of the union in a better time we need to notice that, since adjacent vectors of D_S are very similar, matrices $MD[i]$ for close values of i are also very similar. This will be explored in the following section.

6.2 Elimination of the Redundancy Among Subproblems

Still exploring the ideas presented in [12], we use the fact that a set of $r+1$ adjacent lines of D_G are r-variant: it is possible to obtain one line from another through at most r insertions and r deletions of elements. One important fact is that for r adjacent lines, all with length m, it is possible to find a vector of $m-r$ common elements, that we will call simply *common vector*. We will denote $D_G^{i_0,r}$ the common vector of lines from $D_G^{i_0}$ to $D_G^{i_0+r}$ related to a GDAG G.

By Property 2.2(3), all elements of a vector $D_G^{i_0}$ will be present in vector $D_G^{i_0+r}$, except those that are less than $i_0 + r$. For example, using data from Table 2, we have that the

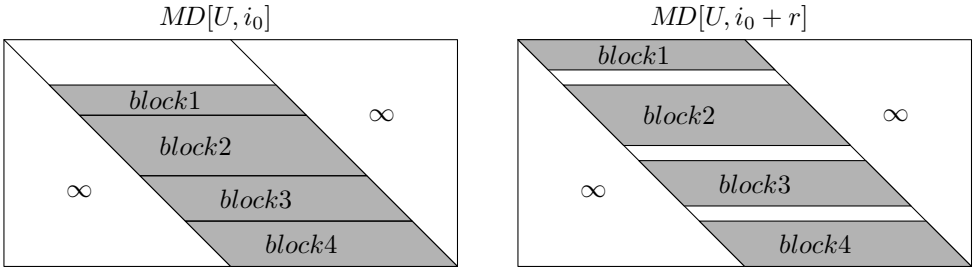

Figure 11: Structure of matrices $MD[i_0]$ and $MD[i_0 + r]$, highlighting r common blocks (in this case, $r = 3$). These blocks are also present in matrices between $MD[i_0]$ and $MD[i_0 + r]$.

common elements of $D_G^0 = (0, 1, 2, 3, 4, 5, 6, 8, 9)$ and $D_G^6 = (6, 7, 8, 9, 11, 13, \infty, \infty, \infty)$ are 6, 8 and 9 (the last ones of D_G^0). Using larger matrices and values of r that are significantly smaller than m, the number of common elements become close to m. In [12] two distinct algorithms use $r = \log^2 m$ and $r = \log^4 m$. The value used in this report is $r = \lceil \sqrt{m} \rceil$.

By Property 2.2(4), all the elements present in both vectors $D_G^{i_0}$ and $D_G^{i_0+r}$ will be present also in vectors D_G^i for $i_0 < i < i_0 + r$. So we have a simple way to determine the common vector: we just have to read and copy all the elements of $D_G^{i_0}$, discarding those that are smaller than $i_0 + r$. This copy takes $O(m)$ time. In the preceding example we would have $D_G^{i_0,r} = (6, 8, 9)$.

Besides that, it is important to determine which elements are adjacent in all vectors, forming "indivisible groups" that we will call *common groups*. In the preceding example, the common groups of $D_G^{i_0,r}$ are $(6, 8)$ and (9).

The determination of the common groups can be done based on vector V_G. From $V_G(i_0 + 1)$ to $V_G(i_0 + r)$ we have all the elements that do not appear in the common vector and can divide it (notice that the elements removed from $D_G^{i_0}$ also does not appear in the common vector but are all smaller than its elements). We determine the divisions between groups searching for the insertion point of each element of V_G, in $O(\log m)$ time per element or $O(r \log m)$ in total.

With r possible division points, it is clear that we will have at most $r + 1$ groups in $D_G^{i_0,r}$. These groups will be enumerated from 0 to r and group t will be called $D_G^{i_0,r}[t]$.

The operations of common vector and common group determinations will be applied to matrix D_S in each union step of the algorithm. The lines from $D_S^{i_0}$ to $D_S^{i_0+r}$ lead the construction of matrices from $MD[i_0]$ to $MD[i_0 + r]$ and, as already mentioned, the similarities between neighboring lines in D_S lead to similarities between matrices $MD[i]$ for close values of i. The common vector $D_S^{i_0,r}$ contains indices of lines of D_I that are present in all matrices from $MD[i_0]$ to $MD[i_0 + r]$. Each group $D_S^{i_0,r}[t]$ indicates a set of lines of D_I that will be used in *adjacent* lines in all these matrices.

Let us consider a common group $D_S^{i_0,r}[t]$, $0 \leq t \leq r$. All matrices $MD[i]$ with $i_0 \leq i \leq i_0 + r$ will contain $Diag[D_S^{i_0,r}[t], D_I, l_{i,t}]$ as a block of contiguous lines starting at line $l_{i,t}$ (the value of $l_{i,t}$ depends on the matrix and the common group). This is illustrated in Figure 11.

These common blocks cover the greatest part of matrices $MD[i]$ when $r \ll m$. Since we will have to solve the Column Minima Problem in each matrix, the determination of the column minima of the common blocks may avoid the repetitive processing of these blocks. So we have the following definition:

Definition 6.4 ($ContBl[i_0, r, t]$)

$$ContBl[i_0, r, t] = Cmin[Diag[D_S^{i_0,r}[t], D_I, 0]]$$

In other words, $ContBl[i_0, r, t]$ is the vector that contains the minima of the columns of block t, common to the matrices from $MD[i_0]$ to $MD[i_0 + r]$.

The determination of the column minima of the blocks will be covered in detail in Section 6.3.

We recall the definition of the *contraction of lines* (Definition 1.7, page 5). If for a matrix $MD[i]$ we perform successive *contractions of lines*, one for each common block, the result will be another totally monotone matrix that will be called $ContMD[i]$. This matrix is such that $Cmin[MD[i]] = Cmin[ContMD[i]]$ (see Theorem 1.2).

The common blocks appear in different matrices $MD[i]$ in different positions, but the results of the search for the column minima in these blocks can be used in all matrices. More precisely, if a common block t appears in line $l_{i,t}$ of matrix $MD[i]$, the contraction of this block in this matrix is done by the simple substitution of the block by the vector $Shift[l_{i,t}, ContBl[i_0, r, t], 2m]$. In Section 6.4 we show how this substitution is done.

Each matrix $ContMD[i]$ will contain, besides the $r+1$ lines generated by the contraction of the common blocks, at most r additional lines. As already mentioned, $r = \lceil \sqrt{m} \rceil$ and so the contraction of the matrices reduce their height to $O(\sqrt{m})$. The benefit of operating over $r + 1 = \lceil \sqrt{m} \rceil + 1$ matrices of reduced height more than compensates the additional cost of the contraction of the common blocks.

The next sections will contain the details and analysis of each step of this procedure.

6.3 Determination of the Column Minima for the Common Blocks

In this section we cover the determination of vectors $ContBl[i_0, r, t]$, $0 \le t \le r$. Each of these vectors contains the solution for the Column Minima Problem for a matrix $Diag[D_S^{i_0,r}[t], D_I, 0]$. Each matrix has width $\Theta(m)$ and the sum of the heights of all of them is $O(m)$ (the length of the common vector $D_S^{i_0,r}$).

As already mentioned, the $r + 1$ common groups of $D_S^{i_0,r}$ may be determined in $O(m)$ time. We call m_t the length of the common group t, that defines the height of matrix $Diag[D_S^{i_0,r}[t], D_I, 0]$.

The elements of the groups may be accessed in $O(1)$ time, just like the elements of D_S and D_I, so we may consider that the elements of the matrices are (indirectly) accessible in constant time.

In Section 9.2 an algorithm for the Column Minima Problem for totally monotone matrices is shown (Algorithm 9.4). This algorithm finds the vector of solutions for block t in $O(m_t+m+m_t \log(m/m_t)))$ time (Theorem 9.5). With this, the time for the determination of all r vectors $ContBl[i_0, r, t]$ is $\Theta(mr) = \Theta(m\sqrt{m})$, too much for our objectives.

Algorithm 9.4 may be adapted as suggested in Section 9.2 (Corollary 9.6) to build a data structure that allows queries on the column minima of the blocks in $O(\log m_t) = O(\log m)$ time. The larger access time for these minima is compensated by a smaller construction time of $O(m_t \log(m/m_t))$. The time for the construction of the structures for all the matrices is $O(m \log m)$.

At the end of these constructions, vectors $ContBl[i_0, r, t]$ will be available for queries in $O(\log m)$ time per element. The space utilized in the construction and storage of all these vectors is $O(m \log m)$ (Corollary 9.6).

6.4 Representation of Matrices $ContMD[i]$

Once the column minima of the common blocks are determined, we can proceed with the determination of the column minima of all matrices $ContMD[i]$. These minima are the elements of the matrix we are trying to build, D_U.

To describe the solution of these minimization problems we initially need to describe how $ContMD[i]$ ($i_0 \leq i \leq i_0 + r$) is represented and accessed. All the lines of these matrices are already available, some in a direct form (through the lines of D_I), some indirectly (through the representation of vectors $ContBl[i_0, r, t]$, as seen in the previous section). To build a particular matrix it is necessary to specify which lines will be used, and in which order. Each line has to be shifted to the right by a certain amount, and this amount changes from matrix to matrix.

Since matrices $ContMD[i]$ will be processed sequentially, we can build a structure that is capable of representing just one matrix at a time, modifying it at each step to represent the next matrix; Two vectors will be used to accomplish this, both of size $2r + 1$ (indices from 0 to $2r$). Vector *Lin* will indicate the *source* of the data of each line of $ContMD[i]$ (this source can be a line of D_I or a vector $ContBl[i_0, r, t]$). Vector *Shf* will indicate how much each *source* has to be shifted to the right.

The construction of $ContMD[i_0]$, the first matrix in the range, is done directly with $D_S^{i_0}$. This line of D_S is read element by element and the vectors *Lin* and *Shf* are filled starting at index 0. Each element of $D_S^{i_0}$ that is not part of the common vector is inserted in the next position of *Lin*, representing one single line of D_I. When an element that is part of the common vector is found in $D_S^{i_0}$, it and all the elements of the same common group are substituted in *Lin* by a reference to the corresponding vector $ContBl[i_0, r, t]$, representing an already contracted block of lines.

The first position of *Shf* receives 0. For $l > 0$, $Shf(l)$ receives $Shf(l-1)+1$, if the item registered in $Lin(l-1)$ is a single line, or $Shf(l-1)+m_t$, if the item is a contracted block of height m_t.

This construction can be done in time $O(m)$. For reading any element of $ContMD[i_0](l, k)$ we proceed in this way: $Lin(l)$ indicates the source of the data, a vector from which we get the element of index $k - Shf(l)$. In the case that this index is not usable, the element returned is ∞. The total access time is $O(1)$ for single lines, $O(\log m)$ for contracted blocks.

When line $D_U^{i_0} = Cmin[ContMD[i_0]]$ is determined and matrix $ContMD[i_0]$ is no longer necessary, we may proceed to the next matrices. The changes in *Lin* and *Shf* for each matrix $ContMD[i]$, $i_0 < i \leq i_0 + r$, is done by the insertion of $V_S(i)$ in *Lin*. All the elements smaller than $V_S(i)$ (including those that represent blocks of lines with indices smaller than $V_S(i)$) are shifted one position to the left. The one that was in position 0 is discarded. The corresponding positions in *Shf* are decremented and the one that corresponds to $V_S(i)$ is calculated. All this can be done in $O(r) = O(\sqrt{m})$ time.

The additional space for the representation of the matrices is $O(\sqrt{m})$ and the total time for the maintenance of this representation during the processing of $r + 1$ matrices is $O(m)$.

We will now see how these matrices are actually used. From now on, we will consider that the access time to $ContMD[i](l, k)$ is $O(\log m)$, ignoring the implementation details.

6.5 Determination of lines from i_0 to $i_0 + r$ of D_U

The determination of line $D_U^{i_0}$ is done based on $ContMD[i_0]$, using Algorithm 9.4. This determination takes $O(\sqrt{m} \log^2 m)$ time, by Corollary 9.6, where the extra $\log m$ factor is

due to the access time of each item of $ContMD[i_0]$. Throughout this section, this $\log m$ factor will appear in all time estimates. An extra $O(m)$ time is necessary to make explicit all elements of $D_U^{i_0}$.

Each of the following lines of D_U is generated based on the preceding one. Here we have a huge difference from the procedure of Lu & Lin ([12]), as the parallel model in consideration in that paper is the PRAM, requiring the simultaneous determination of all lines of D_U.

Using Properties 2.3, if we have D_U^i we can obtain D_U^{i+1} through the deletion of the first element and the insertion of a new one, $V_U(i+1)$, in order with the other elements. All elements of D_U^i that are smaller than $V_U(i+1)$ are shifted one position to the left (the first one, as already said, is removed). The other elements are kept in their positions.

So, the determination of D_U^{i+1} may be done simply by determining $V_U(i+1)$, avoiding the search for the minima of *all* the columns of $ContMD[i+1]$. If the minimum of a certain column k is equal to $D_U^i(k)$, this means that the insertion position of $V_U(i+1)$ is less than k. If the minima of column k is larger than $D_U(i,k)$, this means that the insertion position of $V_U(i+1)$ is k or greater.

There are $2m+1$ columns in $ContMD[i+1]$. We take the columns of index multiple of $r = \lceil \sqrt{m} \rceil$ (except 0) and determine the minimum in each of them. This can be done through Algorithm 9.4 for the submatrix $ContMD[i+1](l,k)[k = rs \leq 2m, s = 1, 2, \ldots, \lfloor 2m/r \rfloor]$. This submatrix has less than $2\sqrt{m}$ columns and approximately $\sqrt{m}$ lines, so Algorithm 9.4 executes in $O(\sqrt{m} \log m)$ time.

We then compare the minima of the selected columns with the corresponding elements of D_U^i (the indices are multiple of r). Let s_0 be the smallest value of s such that $D_U(i, rs)$ is equal to the minimum of column rs of $ContMD[i+1]$. We know that $V_U(i+1)$ has to be in a column in the range from $k_1 = r(s_0 - 1)$ to $k_2 = rs_0$. If there is no such s_0, $V_U(i+1)$ has to be in a column in the range from $k_1 = r\lfloor 2m/r \rfloor$ and $k_2 = 2m$. The determination of s_0 may be done in $O(\log m)$ time by a binary search.

We obtain the value and insertion position of $V_U(i+1)$ using again Algorithm 9.4, now for the submatrix $ContMD[i+1](l,k)[k_1 \leq k \leq k_2]$, and comparing the column minima with the corresponding values of D_U^i. This step also requires $O(\sqrt{m} \log m)$ time.

We conclude that the determination of lines from $i_0 + 1$ to $i_0 + r$ of D_U may be done in $O(r\sqrt{m} \log m) = O(m \log m)$ time. Including the determination of line i_0, we still have $O(m \log m)$ time.

An important observation: this procedure demands that the line D_U^i is available for queries in constant time during the determination of $V_U(i+1)$. This can be accomplished using the structure presented in Section 4 for the representation of matrix D_U. Initially we store $D_U^{i_0}$. Then, for each new element of V_U the structure is updated in time $O(\sqrt{m})$ to represent the new determined line of D_U.

Since we are interested in producing only D_U^0 and V_U (the compact representation of D_U), the other lines of D_U are used only to determine the elements of V_U and can be discarded when they are no longer necessary. The structure of Section 4 can be easily modified to occupy only $O(m)$ space, enough to contain the elements of the last determined line of D_U.

6.6 Complete Analysis of the Union Process

Using the redundancies among matrices $MD[i]$ it is possible to determine $D_U^{i_0}$ and $V_U(i)$ for $i_0 \leq i \leq i_0 + r$ in $O(m \log m)$ time: $O(m \log m)$ is spent on the contraction of the groups of

common lines of matrices $MD[i]$, $O(m)$ is spent on the maintenance of matrices $ContMD[i]$ and $O(m \log m)$ is spent on the determination of results. Considering that there are a total of $(n+1)/(r+1)$ groups of $r+1$ matrices $MD[i]$ to process in this way, the total time for the determination of D_U^0 and V_U from D_S and D_I is $O((nm/r) \log m) = O(n\sqrt{m} \log m)$.

We may split this work among q processors through the division of D_S among them. Each block of r lines of D_S is used to determine r lines of D_U. The processing of each block is independent of the others. With this, the time spent in the union drops to $O((n\sqrt{m} \log m)/q)$.

We still need to consider the time spent (before the union itself) on the construction of the compact representations of D_S and D_I, as shown in Section 4. The biggest problem is the representation of D_I, that has to be available in the local memory of all processors. This construction takes $O(n\sqrt{m})$ time and is done by all processors in a redundant manner.

With these results we may present the following lemma:

Lemma 6.2 *Let U be a GDAG $(2m+1) \times (n+1)$ for the ALCS problem, formed by the union of the $(m+1) \times (n+1)$ GDAGs S (superior) and I (inferior). The determination of D_U^0 and V_U from D_S^0, V_S, D_I^0 and V_I can be done by q processors in $O\left(n\sqrt{m}\left(1 + \frac{\log m}{q}\right)\right)$ time and $O(n\sqrt{m})$ space.*

7 Analysis of the CGM Algorithm for the ALCS Problem

The parameters of the complete problem are the lengths of the strings A and B and the number of processors, respectively n_a, n_b and p. As already mentioned, the resolution of the ALCS in the p GDAGs defined by p substrings of A takes $O\left(\frac{n_a n_b}{p}\right)$ time.

The $\log p$ union steps that follow take $O\left(n\sqrt{m}\left(1 + \frac{\log m}{q}\right)\right)$ time each one, as seen in Lemma 6.2. The parameters q, n and m are such that $n = n_b$ and $m = \frac{n_a q}{2p}$, turning the time expression into

$$O\left(\frac{n_b\sqrt{n_a q}}{\sqrt{2p}}\left(1 + \frac{\log \frac{n_a q}{2p}}{q}\right)\right) = O\left(\frac{n_b\sqrt{n_a}}{\sqrt{p}}\left(\sqrt{q} + \frac{\log n_a}{\sqrt{q}}\right)\right) .$$

So, the time for each union step is defined by a single variable, that is the number of processors involved with each new GDAG produced, q. This number doubles at each step, so the sum of the times of all the union steps is

$$O\left(\frac{n_b\sqrt{n_a}}{\sqrt{p}}\left(\sum_{i=1}^{\log p}(\sqrt{2})^i + \sum_{i=1}^{\log p}\frac{\log n_a}{(\sqrt{2})^i}\right)\right) = O\left(\frac{n_b\sqrt{n_a}}{\sqrt{p}}\left(\sqrt{p} + \log n_a\right)\right) =$$

$$O\left(n_b\sqrt{n_a}\left(1 + \frac{\log n_a}{\sqrt{p}}\right)\right) .$$

In order to guarantee a linear speed-up for the CGM algorithm, the time above must be $O\left(\frac{n_a n_b}{p}\right)$. This is achieved if $p < \sqrt{n_a}$, so we choose this to be the limit for the number of processors to be used in our algorithm.

We now study the communication complexity. When there are q processors working in a union of GDAGs, each one determines n_b/q elements of V_U and needs to send this to other

$2q-1$ processors that will work in the next union step. So, for the transmission of V_U we have $O(n_b)$ data sent/received per processor in a communication round.

The processor that determines D_U^0 needs to transfer the $\frac{n_a q}{p}$ elements of these vector to other $2q-1$ processors, so it transfers $O(\frac{n_a q^2}{p})$ data in one communication round. Our computational model does not consider communication broadcasts.

For some constant C, the transfer of D_U^0 may also be done in C communication rounds, with only $O\left(\frac{n_a q^{1+1/C}}{p}\right)$ data transfered by each processor: in the first round the processor that determined D_U^0 broadcast this vector to $\lfloor q^{1/C} \rfloor$ other processors, that in the next round transmit to other $\lfloor q^{2/C} \rfloor$ processors and so on. There are other schemes for reducing the amount each processor has to send, utilizing a larger number of rounds. We will use this one, that keeps the number of rounds constant for each union step.

The union step with the largest amount of data to be transfered is the last one, when all processor need to send $O(n_a p^{1/C} + n_b)$ data, already considering D_U^0 and V_U.

In the last union step, vectors D_G^0 and V_G are determined for the complete GDAG G, that represents the full ALCS problem. With a little change in the algorithm, each processor can generate and keep part of D_G in a compact structure. In this way, processor P_t $(1 \leq t \leq p)$ will have the data of the lines from $D_G^{\lfloor n_b(t-1)/p \rfloor}$ to $D_G^{\lfloor n_b t/p \rfloor - 1}$.

These results are summarized in the following theorem:

Theorem 7.1 *Given two strings A and B, respectively of lengths n_a and n_b, let G be the GDAG for the ALCS Problem for A and B. The construction of D_G can be done by $p < \sqrt{n_a}$ processors in $O\left(\frac{n_a n_b}{p}\right)$ time, $O(n_b\sqrt{n_a})$ space per processor and $O(C \log p)$ communication rounds with $O(n_a p^{1/C} + n_b)$ data transfered from/to each processor.*

Matrix D_G allows the determination of the similarity of string A and any substring of B, but not in $O(1)$ time. For this it would be necessary to build matrix C_G (Definition 2.1). One access to $C_G(i,j)$ may be simulated with D_G (compact representation or not) in $O(\log n_a)$ time, using a binary search in D_G^i to find the smallest value of k such that $D_G^i(k)$ is larger than j. The value of $C_G(i,j)$ is $k-1$.

If a quicker access is desirable, a representation of C_G can be constructed. The construction is very simple: each line of D_G is used to build a line of C_G in time $O(n_b)$. The complete construction, done by p processors, takes $O(n_b^2/p)$ time. The space required is also $O(n_b^2/p)$ per processor.

These results are summarized below:

Theorem 7.2 *Given two strings A and B, respectively of lengths n_a and n_b, let G be the GDAG for the ALCS Problem for A and B. The construction of C_G can be done by $p < \sqrt{n_a}$ processors in $O\left(\frac{(n_a+n_b)n_b}{p}\right)$ time, $O(n_b \max\{\sqrt{n_a}, n_b/p\})$ space per processor and $O(C \log p)$ communication rounds with $O(n_a p^{1/C} + n_b)$ data transfered from/to each processor.*

It should be noticed that to get the similarity of A and B (basic LCS Problem) only D_G^0 is needed. The similarity is equal to the index of the last finite element of D_G.

8 Obtention of the Longest Common Subsequence

Normally there is no interest in the obtention of the actual longest common subsequence for string A and *all* substrings of B. The *lengths* of these subsequences are determined as

previously shown and if an interest in any of the subsequences arises it can be determined later. To keep a data structure that allows the fast obtention of any common subsequence requires too much space, so our objective here is to maintain some data that allow the obtention of a common subsequence in a reasonable time.

Particularly, the algorithm presented in this report for the ALCS can be used for the solution of basic LCS, since the complexity of this algorithm is satisfactory also for the LCS. In other words, this algorithm may be applied in situations where a particular longest common subsequence (the one for string A and the complete string B) is desired.

8.1 Basic Procedure

We present now some extensions to the CGM algorithm for the ALCS that allow the obtention of the longest common subsequence of A and *any* substring of B in $O(n_b \log p + n_a \log n_a)$ time and $O(\log p)$ communication rounds.

A recursive procedure is used, decomposing a GDAG in smaller GDAGs at each call. The union steps are followed "backwards", utilizing data collected during these steps.

In a union step, two GDAGs S and I are united to determine a new GDAG U. In the sequence determination phase, the best path from $T_U(i) = T_S(i)$ to $F_U(j) = F_I(j)$ must be determined. A vertex in $F_S = T_I$ (the common border) that belongs to this path is determined, say $F_S(x) = T_I(x)$. There may be several vertices of the path in the border, and $F_S(x) = T_I(x)$ will be the leftmost one. This vertex will be called the *crossing vertex* between S and I.

After that, the best path from $T_S(i)$ to $F_S(x)$ and the best path from $T_I(x)$ to $F_I(j)$ are determined, which is done recursively, that is, S and I will be decomposed in smaller GDAGs and new crossing vertices will be found inside them. This recursion stops when a path in a basic GDAG (whose matrix D_G were determined by a single processor and not in a union step) is to be determined.

Then, the p GDAGs that were used in the initial step of the algorithm are used again. The paths in these GDAGs are determined by the p processors in parallel, in $O(n_a n_b/p)$ time and $O(n_a/p + n_b)$ space per processor using the algorithm by Hirschberg [8]. The complete path is formed by the union of the paths in the basic GDAGs. Figure 12 illustrates the process.

Figure 12 also shows which processor will contain the data for a particular GDAG. Each processor keeps the data of at most two GDAGs, neighbors is some union step. The data that processor P_t stores are generated at step $w(t) = \max\{x \mid t \equiv 0 \pmod{2^x}\}$. If $w(t) = 0$, processor P_t stores data from two of the basic p GDAGs.

Each processor uses these data in the determination of one of the vertices in the desired path. This data distribution allows the determination of the path with very little communication.

8.2 Determination of the Crossing Vertices

As always, let U be a $(2m+1) \times (n+1)$ GDAG resulting from the union of $(m+1) \times (n+1)$ GDAGs S and I. The data structures used to unite these GDAGs are vectors D_S^0, V_S, D_I^0 and V_I. These structures may be kept in memory after the union steps, for they do not occupy too much space. They contain the data that are transfered during the union steps and are later used to build the compact representation of Section 4. This compact

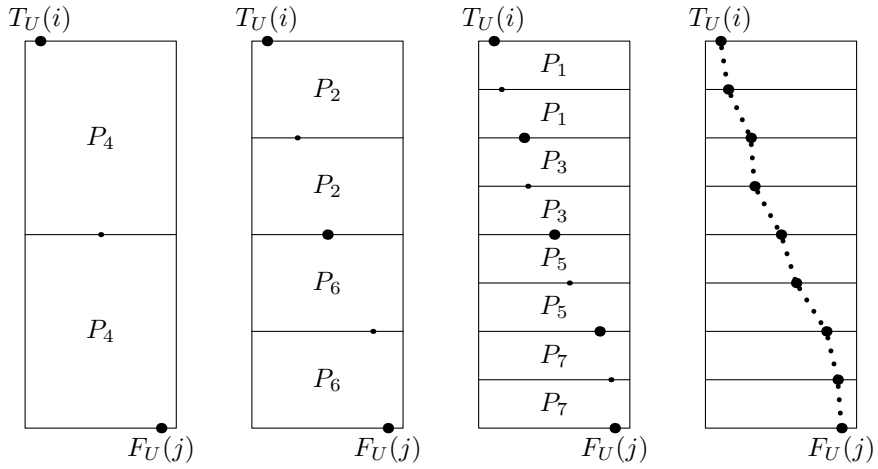

Figure 12: Obtention of the path from $T_U(i)$ to $F_U(j)$. At each step, the data that was determined in the union steps are used in the reverse order to obtain the crossing points between GDAGs. In the last step, the subpaths inside the GDAGs are determined. In this figure, each GDAG indicates the processor that contain the data about it.

representation allows fast retrieval of data but occupies too much space, so no data are kept in this format after the union steps.

To obtain the path from $T_U(i)$ to $F_U(j)$ two vectors are needed: D^i_S, that gives information about the paths from $T_U(i)$ to the common line $F_U = T_I$, and $D^j_{I^{rev}}$, that gives information about the paths from the common line to $F_I(j)$. This last vector is equivalent to D^j_I, if all the arcs of I were *reversed*. Its definition is given below:

Definition 8.1 (Vector $D^j_{I^{rev}}$) *Let I be a GDAG $(m+1)\times(n+1)$ for the ALCS problem. For $0 \le j \le m$, $D^j_{I^{rev}}(0) = j$ and for $1 \le k \le n$, $D^j_{I^{rev}}(k)$ indicates the value i such that $C_I(i,j) = k$ and $C_I(i+1,j) = k-1$. If there is no such value, $D_{I^{rev}}(i,j) = -\infty$.*

It is convenient to compare definitions 2.2 (page 8) and 8.1 now. Both definitions are very similar, with exchanged roles for variables i and j. We recall that $C_I(i,j)$ indicates the weight of the best path from $T_I(i)$ to $F_I(j)$. Notice that the elements of $D^j_{I^{rev}}$ are sorted *decreasingly*.

For better comprehension, Table 7 shows $D^j_{I^{rev}}$ for several values of j, using GDAG G of Figure 2 as I. The data from C_I (in this case C_G) for the same GDAG are shown in Table 6 (it is the same table from page 10).

The following observations can be easily deduced from the definitions:

Observation 8.1 *For any GDAG G, given two vertices $T_G(i)$ and $F_G(j)$, the weight of the best path between these vertices ($C_G(i,j)$) is equal to:*

1. *the number of elements of D^i_G that belong in the interval $]i,j]$.*
2. *the number of elements of $D^j_{G^{rev}}$ that belong in the interval $[i,j[$.*

These observations highlight the symmetry that exists between D_G and $D_{G^{rev}}$. Now, we will see how to obtain vectors D^i_S and $D^j_{I^{rev}}$.

	j													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$C_G(0,j)$	0	1	2	3	4	5	6	6	7	8	8	8	8	8
$C_G(1,j)$	0	0	1	2	3	4	5	5	6	7	7	7	7	7
$C_G(2,j)$	0	0	0	1	2	3	4	4	5	6	6	6	6	7
$C_G(3,j)$	0	0	0	0	1	2	3	3	4	5	5	6	6	7
$C_G(4,j)$	0	0	0	0	0	1	2	2	3	4	4	5	5	6
$C_G(5,j)$	0	0	0	0	0	0	1	2	3	4	4	5	5	6
$C_G(6,j)$	0	0	0	0	0	0	0	1	2	3	3	4	4	5
$C_G(7,j)$	0	0	0	0	0	0	0	0	1	2	2	3	3	4
$C_G(8,j)$	0	0	0	0	0	0	0	0	0	1	2	3	3	4
$C_G(9,j)$	0	0	0	0	0	0	0	0	0	0	1	2	3	4
$C_G(10,j)$	0	0	0	0	0	0	0	0	0	0	0	1	2	3
$C_G(11,j)$	0	0	0	0	0	0	0	0	0	0	0	0	1	2
$C_G(12,j)$	0	0	0	0	0	0	0	0	0	0	0	0	0	1
$C_G(13,j)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 6: C_G corresponding to the GDAG of Figure 2(reprise).

	k								
	0	1	2	3	4	5	6	7	8
$D_{G^{rev}}(13,k)$	13	12	11	10	9	6	5	3	0
$D_{G^{rev}}(12,k)$	12	11	10	9	6	5	3	1	0
$D_{G^{rev}}(11,k)$	11	10	9	8	6	5	3	1	0
$D_{G^{rev}}(10,k)$	10	9	8	6	5	3	2	1	0
$D_{G^{rev}}(9,k)$	9	8	7	6	5	3	2	1	0
$D_{G^{rev}}(8,k)$	8	7	6	5	3	2	1	0	$-\infty$
$D_{G^{rev}}(7,k)$	7	6	5	3	2	1	0	$-\infty$	$-\infty$
$D_{G^{rev}}(6,k)$	6	5	4	3	2	1	0	$-\infty$	$-\infty$
$D_{G^{rev}}(5,k)$	5	4	3	2	1	0	$-\infty$	$-\infty$	$-\infty$
$D_{G^{rev}}(4,k)$	4	3	2	1	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$D_{G^{rev}}(3,k)$	3	2	1	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$D_{G^{rev}}(2,k)$	2	1	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$D_{G^{rev}}(1,k)$	1	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$D_{G^{rev}}(0,k)$	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$

Table 7: $D_{G^{rev}}$ corresponding to the GDAG of Figure 2 (lines in reverse order).

	k												
	1	2	3	4	5	6	7	8	9	10	11	12	13
$V_G(k)$	∞	13	11	∞	7	∞	∞	10	12	∞	∞	∞	∞

Table 8: V_G corresponding to the GDAG of Figure 2. It can be used to determine a line of $D_{G^{rev}}$.

The determination of $D^j_{I^{rev}}$ can be done through V_I alone, D^0_I is not necessary. Table 8 shows V_I (in this case, V_G) in the same page with $D_{I^{rev}}$ to facilitate the comprehension of the following lemma:

Lemma 8.1 *For all $i' < j$, i' is an element of $D^j_{I^{rev}}$ if and only if $V_I(i'+1) > j$.*

Proof. If $i' < j$ and $V_I(i'+1) > j$ then by Definition 2.3, there is no element in $D^{i'+1}_I$ that is not in $D^{i'}_I$ and is less than or equal to j. However, element i' is in $D^{i'}_I$ but not in $D^{i'+1}_I$. By Observation 8.1(2) we conclude that $C_I(i', j) = C_I(i'+1, j) + 1$. By definition 8.1, i' is an element of $D^j_{I^{rev}}$.

On the other hand, if $i' < j$ and i' is an element of $D^j_{I^{rev}}$, by Definition 8.1 we have $C_I(i', j) = C_I(i'+1, j) + 1$. By observation 8.1(1), the number of elements of the interval $]i', j]$ in $D_I(i')$ has to be less than $D_I(i'+1)$. This can only occur if $V_I(i'+1) > j$ (the element that appears in $D_I(i'+1)$ is outside the interval $]i', j]$). □

Through this lemma, we have a simple way to construct vector $D^j_{I^{rev}}$ in $O(n)$ time. We make $D^j_{I^{rev}}(0) = j$ and verify vector V_I from the end to the beginning, inserting in $D^j_{I^{rev}}$ all i' such that $V_I(i'+1) > j$.

To construct D^i_S we need to insert all the elements from D^0_S and V_S (up to $V_S(i)$), discarding the ones that are smaller than i. This can be done in $O(m+n)$ time, obtaining a vector of size $O(m)$ that can be sorted in $O(m \log m)$ time.

So, both necessary vectors can be constructed in $O(n + m \log m)$ time and space.

Then we proceed with the search for the crossing vertex. Let l_0 be the number of elements of $D^j_{I^{rev}}$ in the interval $[i, j[$. This number may be determined in time $O(m)$ and is equal to $C_I(i, j)$ (Observation 8.1(2)).

To find the crossing vertex as the *leftmost* vertex in the best path, the natural candidates are those indicated by D^i_S. The first candidate is $F_S(i) = T_I(i)$, just below $T_S(i)$, because $D^i_S(0) = i$. The best path from $T_S(i)$ to $F_I(j)$ that pass through this vertex has weight l_0. For the following candidates, the weight of the path from $T_S(i)$ to the candidate is one plus the weight of the path to the preceding candidate, but the weight of the path from this candidate to $F_I(j)$ may be smaller.

If $Max(k)$ is the total weight of the best path that pass through candidate k (that is, through vertex $F_S(D^i_S(k))$) by Observation 8.1 we have:

$$Max(k) = l_0 + k - \text{number of elements of } D^j_{I^{rev}} \text{ in the interval } [i, D^i_S(k)[\ .$$

The determination of these values, for all candidates, can be done in $O(m)$ time. Figure 13 illustrates the procedure. Getting the largest of these values we find the crossing vertex.

So, from the data of D^0_S, V_S and V_I it is possible, in $O(n + m \log m)$ time, determine the crossing vertex in the path from $T_U(i)$ and $F_U(j)$.

8.3 Analysis of the Process of Obtention of the Longest Common Subsequence

To determine all the crossing vertices, $\log p$ steps are necessary. In each step, each processor sends/receives just $O(1)$ data, indicating vertices found in the previous step.

In a step where the GDAGs are $(m+1) \times (n+1)$, the time spent (in parallel) is $O(n + m \log m)$. At each step m is halved, being $n_a/2$ in the first one, and $n = n_b$ in all steps. From this we conclude that the time spent in all the steps is $O(n_b \log p + n_a \log n_a)$.

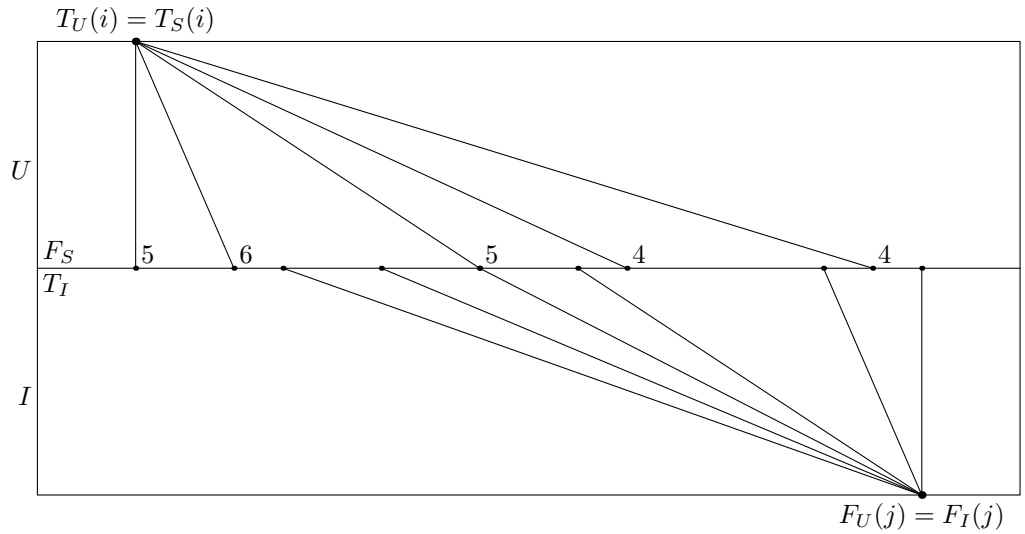

Figure 13: Obtention of the crossing vertex in the path between $T_U(i)$ and $F_U(j)$. The paths shown in S go from $T_U(i)$ to the vertices indicated in D_S^i. The paths in I go from the vertices indicated in $D_{I^{rev}}^j$ to $F_U(j)$. Next to each vertex indicated by D_S^i we have the total weight of the best path to $F_U(j)$ that pass through this vertex. The determination of this weight involves counting of the vertices indicated by $D_{I^{rev}}^j$ that are to the left of the vertex.

The subpaths contained in basic GDAGs may be determined in time $O(n_b + n_a/p)$ if the data used during the execution of Algorithm 3.1 (sequential ALCS) are kept, occupying space $O(n_a n_b/p)$. This is indicated in Theorem 3.1. However, it is more advantageous to keep the used space low. When a path between two crossing vertices is required in a basic GDAG, a sequential algorithm for the LCS that uses linear space is preferred (see [8]).

Each processor determines a subpath in the full GDAG, and from it a piece of the longest common subsequence of A and B is obtained. All pieces can be joined in one communication round of size $O(n_a/p)$

This leads to the following result, that unite the results of this section with Theorem 7.1:

Theorem 8.2 *Given two strings A and B, respectively of lengths n_a and n_b, the determination of the longest common subsequence of A and B may be done by $p < \sqrt{n_a}$ processors in $O\left(\frac{n_a n_b}{p}\right)$ time, with $O(n_b\sqrt{n_a})$ space per processor and $O(C \log p)$ rounds of communication with $O(n_a p^{1/C} + n_b)$ data transfered from/to each processor per round.*

9 Appendix: Algorithms for the Monotone and Totally Monotone Matrices

In this section we present the algorithms for the monotone and totally monotone matrices, already mentioned in the previous sections. This appendix is important because, although its results were already given by Aggarwal et al. [1], we need some further considerations to use them. More specifically, we need to know how to find the column minima of a "wide" matrix. Furthermore, this section makes this report more self-contained.

For those familiar with the results of the original article, this appendix may be skipped. However, it is important to notice that the definitions and algorithms presented here and in Section 1.4 exchange the roles of columns and lines of the matrices in the original article (The "rows" here are "columns" there and vice versa) and also change "maxima" for "minima".

Our special considerations are explicited in Algorithm 9.2, Theorem 9.2 and Corollary 9.6.

9.1 The Column Minima Problem in a Monotone Matrix

For monotone matrices, the (recursive) algorithm for the determination of the column minima is presented here. The algorithm is defined for submatrices to make the recursion explicit.

Algorithm 9.1: **Determination of the minima of the columns of a monotone matrix.**
Input: Submatrix $M(i,j)[i_1 \leq i \leq i_2, j_1 \leq j \leq j_2]$.
Output: $Imin[M](j)$, $j_1 \leq j \leq j_2$.

1 $aux \leftarrow i_1$
2 $middle \leftarrow \lfloor (j_1 + j_2)/2 \rfloor$
3 For $i \leftarrow i_1 + 1$ to i_2 do
3.1 If $M(i, middle) < M(aux, middle)$ then
3.1.1 $aux \leftarrow i$
4 $Imin[M](middle) \leftarrow aux$
5 If $middle > j_1$ solve the problem for submatrix $M(i,j)[i_1 \leq i \leq aux, j_1 \leq j < middle]$.
6 If $middle < j_2$ solve the problem for submatrix $M(i,j)[aux \leq i \leq i_2, meio < j \leq j_2]$.

end of the algorithm.

Figure 14 illustrates the operation of Algorithm 9.1.

Theorem 9.1 *The minima of all the columns of a monotone $n \times m$ matrix M can be found in $O(n \log m + m)$ time and $O(m)$ space (enough for the answer).*

Proof. The proof is based on Algorithm 9.1. Steps 1 and 3 determine the minimum of the central column of the submatrix while steps 5 and 6 deal with the columns respectively to the left and to the right of the central column. Steps 5 and 6 deal with submatrices of the original matrix. An element that is not in these submatrices cannot be the minimum of its column, due to the monotonicity of M. This assures that the algorithm works correctly.

Let us evaluate the time needed by the algorithm through the number of accesses to M that it has to do. Denoting as $t(n,m)$ the largest possible number of accesses required to solve the problem for a $n \times m$ submatrix , we can prove that $t(n,m) \leq f(n,m) = (\lfloor \log m \rfloor + 1)(n-1) + 2m - 1$. In fact, testing for small values of m we have $f(n,1) = n$, $f(n,2) = 2n + 1$, $f(n,3) = 2n + 3$ and $f(n,4) = 3n + 4$.

We make an induction in m. For a $n \times m$ matrix, we have n accesses related to steps 1 and 3, plus the accesses related to the recursive calls in steps 5 and 6. Let $x = i_{min}(\lceil m/2 \rceil)$. Since

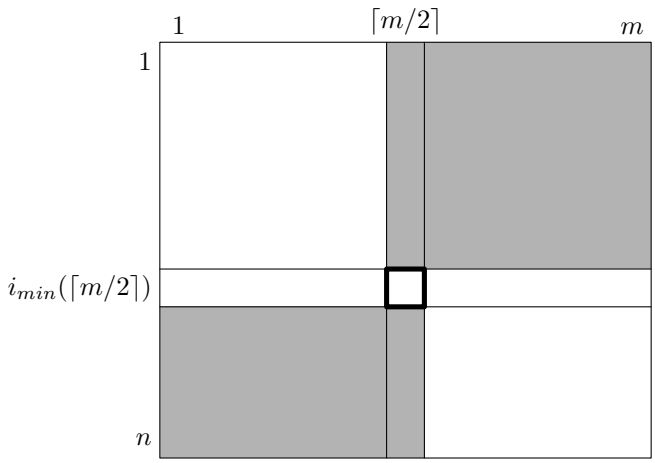

Figure 14: Determination of the minima of the columns of a monotone matrix. The minimum of the central column was determined, the shaded areas may be ignored. In the clear areas the algorithm will proceed recursively.

each recursive call will involve at most $\lfloor m/2 \rfloor$ columns and since $\lfloor \log \lfloor m/2 \rfloor \rfloor = \lfloor \log m \rfloor - 1$, by the induction hypothesis we have

$$
\begin{aligned}
t(n,m) \leq \; & n + \underbrace{f(x, \lfloor m/2 \rfloor)}_{step\ 5} + \underbrace{f(n-x+1, \lfloor m/2 \rfloor)}_{step\ 6} = \\
& n + \underbrace{\lfloor \log \lfloor m/2 \rfloor + 1 \rfloor (x-1) + 2\lfloor m/2 \rfloor - 1}_{step\ 5} + \underbrace{\lfloor \log \lfloor m/2 \rfloor + 1 \rfloor (n-x) + 2\lfloor m/2 \rfloor - 1}_{step\ 6} \leq \\
& n + \lfloor \log m \rfloor (x-1) + \lfloor \log m \rfloor (n-x) + 2m - 2 = n + \lfloor \log m \rfloor (n-1) + 2m - 2 = \\
& (\lfloor \log m \rfloor + 1)(n-1) + 1 + 2m - 2 = f(n,m).
\end{aligned}
$$

This proves the superior limit, that is $f(n,m) = O(n \log m + m)$, concluding the demonstration. Notice that the value of x, that is determined by the position of the minimum of the central column, has no influence in the determination of the superior limit. □

The term m in the execution time is due to the necessity of processing all the columns, even when the elimination of lines generates submatrices with only one line. Relaxing this requisite, the execution time can be reduced to $O(n \log m)$, which is better for "wide" matrices ($m \gg n$). On the other hand, the column minima cannot be made explicit (that is, written in a vector). Aggarwal et al. [1] demonstrated that the inferior limit for the time to solve the Column Minima problem is in fact $\Omega(n \log m)$ in the case of monotone matrices, but it is also said that the superior limit is also $O(n \log m)$, disregarding the problem of not making the solution explicit.

Algorithm 9.1 may be adapted to work in $O(n \log m)$ time when $m \gg n$, generating information about the minima. Unfortunately, this information does not allow the determination of these minima in constant time. The new algorithm produces a vector J of size $n+1$ (with indices starting at 1), such that $Imin[M](j) = i_1$ for all j, $J(i_1) \leq j < J(i_1+1)$. So, given j, to determine $Imin[M](j)$ we need to make a binary search in J, which takes $O(\log n)$ time. We can call value $J(i)$ the *starting point of line* i, because it indicates the column where the minima start to occur in line i (and stop occurring in line $i-1$). Notice that $J(1)$ is always 1 and we will define $J(n+1) = m+1$.

Now we present the adaptation of the Algorithm 9.1 that determines the starting points.

Algorithm 9.2: **Determination of the minima of the columns of a "wide" monotone matrix.**
Input: Submatrix $M(i,j)[i_1 \leq i \leq i_2, j_1 \leq j \leq j_2]$.
Output: $J(i)$, $i_1 + 1 \leq i \leq i_2$.

1 If $j_1 \leq j_2$ then
1.1 $aux \leftarrow i_1$
1.2 $middle \leftarrow \lfloor (j_1 + j_2)/2 \rfloor$
1.3 For $i \leftarrow i_1 + 1$ to i_2 do
1.3.1 If $M(i, middle) < M(aux, middle)$ then
1.3.1.1 $aux \leftarrow i$
1.4 If $aux > i_1$ solve the problem for submatrix $M(i,j)[j_1 \leq j < middle, i_1 \leq i \leq aux]$.
1.5 If $aux < i_2$ solve the problem for submatrix $M(i,j)[middle < j \leq j_2, aux \leq i \leq i_2]$.
2 Else
2.1 For $i \leftarrow i_1 + 1$ to i_2 do
2.1.1 $J(i) \leftarrow j_1$
end of the algorithm.

Theorem 9.2 *For a monotone $n \times m$ matrix M with $n < m$ it is possible to construct a structure in $O(n \log m)$ time and $O(n+\log m)$ space that allows the retrieval of the minimum element of any column in $O(\log n)$ time.*

Proof. The proof is based on Algorithm 9.2. Instead of determining the minimum of the central column as in Algorithm 9.1, this algorithm just uses the position of this minimum to divide the problem in smaller problems, to determine the starting points.

To prove that the algorithm works correctly, we establish just for the sake of this proof that $Imin[M](0) = 1$ and $Imin[M](m+1) = n$. We also suppose that the initial call to Algorithm 9.2 is done for the complete matrix M as argument (that is, $i_1 = j_1 = 1$, $i_2 = n$ and $j_2 = m$) and that $n > 1$ (otherwise the algorithm would not be necessary).

We have the following invariants in the beginning of each call:

1. $Imin[M](j_1 - 1) = i_1$.
2. $Imin[M](j_2 + 1) = i_2$.
3. $i_1 < i_2$

It is easy to notice that the recursive calls on lines 1.4 and 1.5 maintain these invariants.

Each call establishes a value of $J(i)$ for $i_1 < i \leq i_2$. This can be proven by induction on the number of columns of the submatrix. When this number is 0 ($j_1 > j_2$, which implies $j_1 = j_2 + 1$), the starting point for all lines from $i_1 + 1$ to i_2 is j_1 and the loop in line 2.1 establishes the values of $J(i)$. When there is at least one column, line 1.4 establishes $J(i)$ for $i_1 < i \leq aux$ and line 1.5 establishes $J(i)$ for $aux < i \leq i_2$.

So, the algorithm establishes $J(i)$ correctly for $1 < i \leq m$. In the end, it is necessary to make $J(1) = 1$ and $J(m+1) = n+1$.

The used space is just $O(n + \log m)$, necessary to store the starting points and to keep the recursion stack. For the time analysis we will consider the *level* of each call of the algorithm as 0 for the call that involves the complete matrix and $l+1$ for each call made recursively by a call of level l. Each call involves a set of contiguous lines of the matrix, with at least two lines. It is easy to prove that two calls in a same level will involve, at the most, *one* common line. So, there are n calls at each level at the most.

Considering all the calls in a level, the sum of all the accesses done in the loop of line 1.3 is $O(n)$. Since the number of columns of the submatrices is halved at each new level, the number of levels is $O(\log m)$. So, the execution time of Algorithm 9.2 is $O(n \log m)$.

Finally, as already commented, to determine $Imin[M](j)$ for a certain j we do a binary search through the values of $J(i)$, searching for i that makes $J(i) \leq j < J(i+1)$. □

9.2 The Column Minima Problem in a Totally Monotone Matrix

For a totally monotone $n \times m$ matrix, the determination of the column minima can be done in $O(n)$ time, if the matrix is "narrow" ($n > m$). In this kind of matrix, it is possible to perform an initial step in time $O(n)$ where several lines of the matrix are eliminated. The resulting matrix is square and no minimum of any column is eliminated in the process. We will first see the elimination process, then we will see how the column minima problem is solved.

The reduction process is based on the comparison of two elements in a column. Based on the result of this comparison, we can state that certain elements of the matrix are *dead*, that is, they cannot be the minimum element of their respective columns. More precisely, we have the following lemma:

Lemma 9.3 *Let M be a $n \times m$ totally monotone matrix. For $1 \leq i_1 < i_2 \leq m$ and $1 \leq j \leq m$ we have that*

1. *If $M(i_1, j) \leq M(i_2, j)$ then all elements $M(i_2, j')$ with $1 \leq j' \leq j$ are dead, and*
2. *If $M(i_1, j) > M(i_2, j)$ then all elements $M(i_1, j')$ with $j \leq j' \leq m$ are dead.*

Proof. The proof is based on the monotonicity of certain 2×2 submatrices of M, more exactly the ones formed by elements $M(i_1, j)$, $M(i_2, j)$, $M(i_1, j')$ and $M(i_2, j')$. If $M(i_1, j) \leq M(i_2, j)$ and $1 \leq j' \leq j$ then $M(i_1, j') \leq M(i_2, j')$ and this proves the affirmation 1. If $M(i_1, j) > M(i_2, j)$ and $j \leq j' \leq m$ then $M(i_1, j') > M(i_2, j')$ and affirmation 2 is proved. □

Algorithm 9.3 performs the reduction on a $n \times m$ matrix M using Lemma 9.3. This algorithm determines complete lines that are dead and eliminates them from M. The result is matrix R, that is made initially equal to M. At each step, and index K is used to indicate that all elements $M(i, j)$ with $1 < i \leq k$ and $1 \leq j < i$ are dead. If $k = 1$ there are no dead elements in R. Figure 15 illustrates this procedure.

Algorithm 9.3:	**Reduction of a totally monotone matrix.**
Input:	Matrix M $(n \times m)$.
Output:	Matrix R $(m \times m)$ which has the same column minima as M.

1 $R \leftarrow M$

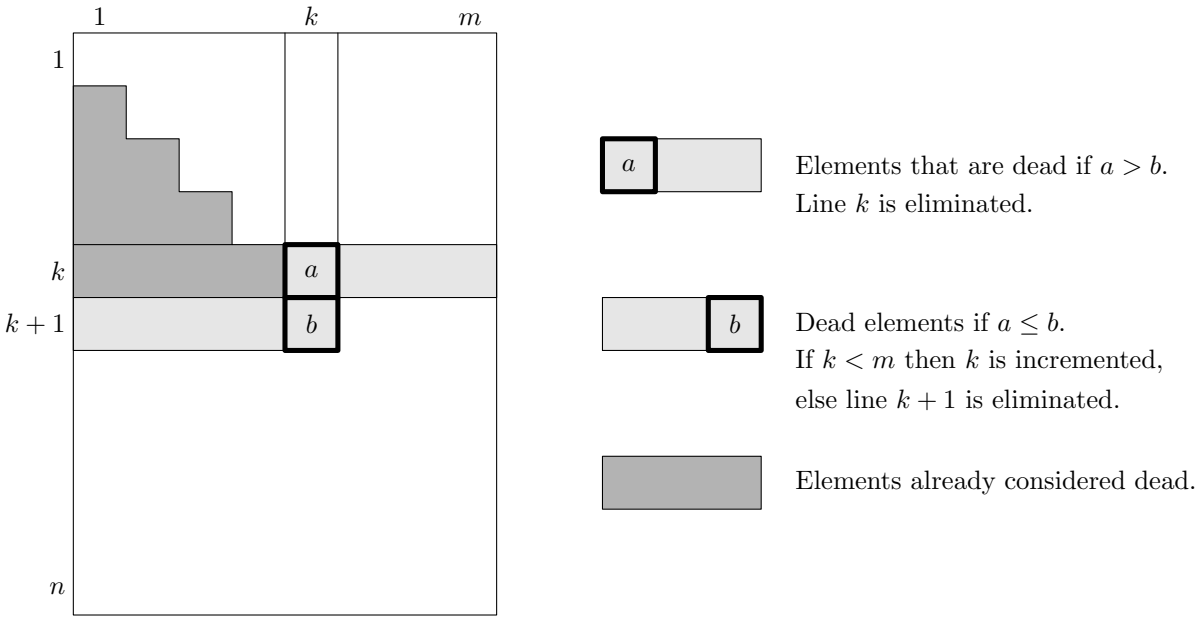

Figure 15: Reduction of a totally monotone matrix.

2 $k \leftarrow 1$
3 While R has more than m lines do
3.1 If $R(k,k) \leq R(k+1,k)$ then
3.1.1 If $k < m$ then
3.1.1.1 $k \leftarrow k+1$
3.1.2 Else
3.1.2.1 Eliminate line $k+1$ from R
3.2 Else
3.2.1 Eliminate line k from R
3.2.2 If $k > 1$ then
3.2.2.1 $k \leftarrow k-1$
end of the algorithm.

Lemma 9.4 *Given a $n \times m$ totally monotone matrix M $(n > m)$, Algorithm 9.3 produces, in $O(n)$ time and using $O(n)$ additional space, a $m \times m$ matrix R such that the minimum element of each column of R is equal to the minimum element of the corresponding column of M. The solution to the Column Minima Problem for M can be obtained from the solution of the same problem for R in $O(m)$ time.*

Proof. In the beginning of each iteration of the loop in line 3, it is alway true that the elements $R(i,j)$ with $1 < i \leq k$ and $1 \leq j < i$ are dead. In fact, with $k = 1$ no element is dead, as already said, and if the affirmation above is true in the beginning of an iteration it will be true in the end, because:

- If $R(k,k) \leq R(k+1,k)$ then by Lemma 9.3(1) all elements $R(k+1,j)$ with $1 \leq j \leq k$

are dead. k may be incremented, keeping the affirmation true, given that this does not make $k > m$. If $k > m$ then all the line $k+1$ would be dead and could be eliminated.

- If $R(k,k) > R(k+1,k)$ the by Lemma 9.3(2) all elements $R(k,j)$ with $k \leq j \leq m$ are dead. This makes all elements in line k dead. This line is eliminated and k is decremented, unless this makes $k < 1$.

The *invariant* affirmation just proved, and also the fact that the lines that are eliminated are those with only dead elements, can be seen in Figure 15.

It is not explicited in Algorithm 9.3 that matrix R can be defined from matrix M with the use of a linked list of lines, using just $O(n)$ space. The assignment of line 1 indicates the initialization of the linked list in time $O(n)$. Each line elimination can be done in constant time. Now it remains to be shown that the loop in line 3 is executed $O(n)$ times.

Let a, b and c be the number of times that the lines 3.1.1.1, 3.1.2.1 and 3.2.1 are executed, respectively. We have that $b + c = n - m$ because this is the number of eliminated lines. Since k starts at 1 and can grow up to m, we have that $a - c \leq m - 1$. So, the number of iterations of the loop in line 3 is $a+b+c \leq a+2b+c = a-c+2(b+c) \leq 2n-m-1 = O(n)$.

Finally, the linked list that defines which lines of M are in R can be copied in a vector in $O(m)$ time. The elements of R can then be accessed in $O(1)$ time. Once the Column Minima Problem is solved for R, the vector can be used to give the answer for M. □

Algorithm 9.4 solves the Column Minima Problem for totally monotone matrices using the reduction procedure.

Algorithm 9.4: **Determination of the column minima of a totally monotone matrix.**
Input: Matrix M $(n \times m)$.
Output: $Imin[M](j)$, $1 \leq j \leq m$.

1 Apply Algorithm 9.3 to M, generating a reduced matrix R $(m \times m)$.
2 If $m = 1$ then
2.1 Determine the minimum of the (only) column of M from R and stop.
3 Else
3.1 $M_2 \leftarrow R(i, 2j)[1 \leq i \leq m, 1 \leq 2j \leq m]$ (M_2 has the even columns of R).
3.2 Solve the Column Minima Problem for M_2 recursively. This solution gives the minima for the even columns of R, registered in $Imin[R](2j)$ for $1 \leq 2j \leq m$.
3.3 Determine $Imin[R](2j+1)$ for $1 \leq 2j+1 \leq m$ using the results of the preceding step.
3.4 Determine $Imin[M](j)$ from $Imin[R](j)$, for $1 \leq j \leq m$.
end of the algorithm.

Theorem 9.5 *The Column Minima Problem can be solved for a totally monotone $n \times m$ matrix in $O(n + m + n\log(m/n))$ time and $O(n + m\log m)$ space.*

Proof. Let us first consider the operation of Algorithm 9.4 for the case when $n \geq m$, then we will extend it to the general case. When $n \geq m$, the theorem affirms that the execution time is just $O(n)$.

By Lemma 9.4 step 1 reduces matrix M using $O(n)$ time and space. After the execution of this step, matrix R occupies $O(m)$ space, just enough to establish the relation between lines of M and R.

Step 3.1 indicates the construction of a matrix M_2 which has only the even columns of R. As this matrix is used in the recursive call to the algorithm, the actual construction of M_2 may be implemented by a new parameter of the algorithm, that indicates the distance between columns where the minimum is to be found. This parameter is doubled at each recursive call, so the construction of M_2 is actually done in constant time.

Step 3.3 can be solved in $O(m)$ time. The results for the even columns divide the m lines of R in $\lceil m/2 \rceil$ intervals, one for each minimum to be found in an odd column of R. The sum of the lengths of this intervals is at most $m + \lceil (m-1)/2 \rceil = O(m)$. The additional space required for this step is constant.

Step 3.4 uses the results of the reduction in step 1, stored in $O(m)$ space, to adapt the minima found to M.

So, the algorithm utilizes $O(n+m)$ time and space, not counting the recursive call in step 3.2. The space required by a recursive call before the next call is just $O(m)$, so the algorithm accumulates $O(m \log m)$ space, totaling $O(n + m \log m)$.

The time spent by the algorithm without considering the recursive call is $O(n+m) = O(n)$. In the first recursive call the size of the matrix is $m \times m/2$ and the time spent (not considering the following calls) is $O(m)$. At each new call the dimensions of the matrices are halved, which indicates that the total time for all the calls is $O(m)$. Since $n \geq m$, the total time spent is $O(n)$.

The case where $n < m$ may be treated normally by Algorithm 9.4. For the analysis of the time spent, it is simpler to consider the following variation: initially, the Column Minima Problem is solved for the submatrix $M(i, \lceil mj/n \rceil)[1 \leq i \leq n, 1 \leq j \leq n]$, formed by n equally spaced columns of M. Using Algorithm 9.4, this step takes $O(n)$ time and require $O(n \log n) = O(m \log m)$ space.

With the determination of the minima of the n selected columns, the minima of the remaining columns are restricted to n sumatrices. Each submatrix is comprehended between two selected columns and between the two lines where the minima of these columns are situated. The application of Algorithm 9.1 to all these submatrices determines the remaining minima of the matrix in $O(n \log(m/n) + m)$ time. The complete algorithm takes $O(n \log(m/n) + m)$ time and requires $O(n + m \log m)$ space. □

In Theorem 9.5, as in Theorem 9.1, the term m in the execution time is due to the necessity of making the minima explicit, which induces the search for the minima even in submatrices with just one line. If these term is discarded, the resulting time is $O(n \log(m/n))$. In [1] it is demonstrated that the lower limit for the time to find the minima of all the columns in a totally monotone $n \times m$ matrix is also $\Theta(n \log(m/n))$.

For "wide" submatrices ($m \gg n$) it is worthy to consider a new variation of the algorithm. Utilizing Algorithm 9.2 in place of Algorithm 9.1 (see the end of the proof of Theorem 9.5), we can construct a vector $J(i)(1 \leq i \leq n+1)$, as explained in Section 9.1, that allows queries for the minimum of any column in $O(\log n)$ time. This leads to the following result.

Corollary 9.6 *For a totally monotone $n \times m$ matrix M with $n < m$, we can build a a data structure in $O(n \log(m/n))$ time and $O(m \log m)$ space that allows queries to the minimum element of any column in $O(\log n)$ time.*

References

[1] Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter Shor, and Robert Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987.

[2] C. E. R. Alves. *Coarse Grained Parallel Algorithms for the String Alignment Problem (in Portuguese).* PhD thesis, Instituto de Matemática e Estatística, Universidade de São Paulo, Brazil, Dec 2002.

[3] C. E. R. Alves, E. N. Cáceres, F. Dehne, and S. W. Song. Parallel dynamic programming for solving the string editing problem on a CGM/BSP. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures - SPAA 2002*, pages 275–281. ACM Press, 2002.

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

[5] E. N. Cáceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S. W. Song. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Proceedings ICALP'97 - 24th International Colloquium on Automata, Languages, and Programming*, volume 1256 of *Lecture Notes in Computer Science*, pages 390–400. Springer Verlag, 1997.

[6] F. Dehne (Ed.). Coarse grained parallel algorithms. *Special Issue of Algorithmica*, 24(3/4):173–176, 1999.

[7] D. Gusfield. *Algorithms in Strings, Trees and Sequences. Computer Science and Computational Biology.* Cambridge University Press, 1997.

[8] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18:341–343, 1975.

[9] James W. Hunt and Thomas G. Szymansky. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, May 1977.

[10] Gad M. Landau and Michal Ziv-Ukelson. On the common substring alignment problem. *Journal of Algorithms*, 41:338–359, 2001.

[11] Mi Lu. Parallel computation of longest-common-subsequences. *Lecture Notes on Computer Science - International Conference on Computing and Information*, 468:385–394, 1990.

[12] Mi Lu and Hua Lin. Parallel algorithms for the longest common subsequence problem. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):835–848, 1994.

[13] M. Maes. On a cyclic string-to-string correction problem. *Information Processing Letters*, 35:73–78, 1990.

[14] W. J. Masek and M. S. Paterson. A faster algorithm for computing string edit distances. *J. Comp. Sys, Sci*, 20:18–31, 1980.

[15] W. J. Masek and M. S. Paterson. How to compute string-edit distances quickly. In D. Sankoff and J. Kruskal, editors, *Time warps, string edits, and macromolecules: the theory and practice of sequence comparison*, pages 337–349. Addison Wesley, 1983.

[16] Pavel A. Pevzner. *Computational Molecular Biology - An Algorithmic Approach*. The MIT Press, 2000.

[17] Claus Rick. New algorithms for the longest common subsequence problem. Technical Report 85123–CS, Institut für Informatik, Universität Bonn, 1994.

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

[19] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.

[20] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[21] Leslie G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 943–972. Elsevier/MIT Press, 1990.