Data Structures and Algorithms For Approximate String Matching

Zvi Galil, Raffaele Giancarlo

Technical Report
CUCS-299-87

DATA STRUCTURES AND ALGORITHMS FOR APPROXIMATE STRING MATCHING

Z. Galil[1,2] and R. Giancarlo[1,3]

Computer Science Department[1]
Columbia University, New York, NY 10027.

Computer Science Department[2]
Tel Aviv University, Tel Aviv, Israel.

Dipartimento di Informatica ed Applicazioni[3]
Universita' di Salerno, Salerno, Italy

ABSTRACT

This paper surveys techniques for designing efficient sequential and parallel approximate string matching algorithms. Special attention is given to the methods for the construction of data structures that efficiently support primitive operations needed in approximate string matching.

Key Words and Phrases: Analysis of sequential and parallel algorithms, data structures, finite automata, string to string correction problem, string matching with k mismatches, string matching with k differences, suffix tree.

Work supported in part by NSF Grants DCR-85-11713, MCS-830-3139, CCR-86-05353 and by the Italian Ministry of Education, Project "Teoria degli Algoritmi".

1. Introduction

The classic algorithms for string matching (see for example [KMP76] and [BM77]) provide linear time solutions for the following problem: Given a text string $text = t[0, n-1]$ and a pattern string $pattern = p[0, m-1]$, $0 \leq m \leq n-1$, find all occurrences of the pattern in the text, i.e. all positions $i, 0 \leq i \leq n-m$ in *text* such that $t[i, i+m-1] = p[0, m-1]$. However in many applications, such as molecular biology, text editing and speech recognition, it is desirable to find substrings of the text that are at most k units apart from the pattern according to a given distance d and for a given integer $k \leq m$. We refer to the following problem as *approximate string matching*: Given strings *text*, *pattern*, an integer $k \leq m$ and a distance d, find all substrings $x_1, x_2, ..., x_s$ of *text* such that $d(x_i, pattern) \leq k$, for $1 \leq i \leq s$.

In this paper we will consider two distances: Hamming distance and Levenshtein distance.

The *Hamming distance* between two strings x and y of equal length is defined as the number of positions with mismatching characters in the two strings. We refer to *approximate string matching* as *string matching with k mismatches* whenever d is the Hamming distance.

The *Levenshtein distance* [LE66], or *edit distance*, between strings x and y, not necessarily of the same length, is defined as the minimal number of *differences* between the two strings, a *difference* being one of the following:

1. A character of the pattern corresponds to a different character of the text.
2. A character of the pattern corresponds to no character in the text.
3. A character of the text corresponds to no character in the pattern.

We refer to *approximate string matching* as *string matching with k differences* whenever d is the Levenshtein distance.

We recall that the edit distance between the pattern and the text can be efficiently computed by means of several algorithms (see for instance [FW74, MP80, UKK83]). Here we are dealing with a more general problem because we want to locate all substrings of the text that are at edit distance of at most k from the pattern. Indeed, it is easily seen that string matching with k differences reduces to this latter problem since a difference of type (1), i.e. $p[i] \neq t[j]$, can be thought of as a substitution of a character ($p[i]$) of the pattern with a character of the text ($t[j]$). Similarly, a difference of type (2) can be thought of as a deletion of a character from the pattern and a difference of type (3) can be thought of as an insertion of a character in the pattern.

In recent years, several efficient parallel and serial algorithms have been devised for approximate string matching. Algorithms for the k mismatches problem are reported in [LV85a, LV85b, GG86a,GG86b ,I84], whereas algorithms for the

k differences problem are reported in [BLLP87,LV85a. LV86. MY86. UKK85.]. Although all the quoted algorithms differ considerably from algorithms for string matching, for the edit distance problem [FW74. UKK83] and for the longest common subsequence [HI75, HS77, AG85], they all exploit basic algorithmic tools that have proved to be useful for exact string matching and for the edit distance algorithms as, for instance, the *suffix tree* [Mc76.W74] and the space efficient computation of the edit distance between two strings devised by [UKK83].

We remark that an efficient algorithm for the k mismatches problem can be obtained as a byproduct of Fischer and Paterson reduction of string matching with don't care characters to fast integer multiplication [FP74] and we also point out that an algorithm for the k differences problem can be obtained by modifying one of the algorithms reported in [FW74, MP80, UKK83] for the computation of the edit distance between two strings. In particular, when the size of the alphabet is constant and k is large, the clever algorithm by Masek and Paterson [MP80] performs quite well.

All the algorithms, both parallel and sequential, that have been recently developed for approximate string matching fit nicely into the following paradigm:

a. Preprocessing of the pattern and the text.
b. Analysis of the text.

The preprocessing step consists either of gathering some information about the pattern and the text which can be used for a fast implementation of primitive operations in the analysis step [LV85a,LV85b, LV86,GG86a,GG86b,MY86,BLLP87] or of constructing a finite automaton that accepts all strings at a distance at most k from the pattern [UKK85, I84]. In order to locate all approximate occurrences of the pattern in the text, the analysis of the text consists either of scanning the text [LV85a,LV85b, LV86,GG86a,GG86b,UKK85,I84] or of the construction of a table [LV85a, LV86, MY86, BLLP87].

The algorithms in [LV85a,LV85b,LV86,GG86a,GG86b, MY86,BLLP87] use the primitive operation that finds on line the leftmost mismatch between any two given suffixes of the pattern and the text. In order to implement such an operation in $O(1)$ time, the preprocessing algorithm must construct data structures that readily support such on line queries. As will be shown later, there are various preprocessing algorithms that set up such structures in $O(m)$ time (sequential algorithms) [LV85a,LV85b, GG86a] or in $O(\log m)$ with $O(m^2+n)$ processors [LV86] or in time $O(\log n)$ with $O(n)$ processors [AILSV87] (a parallel algorithm). We also present new preprocessing algorithms that are improvements of extant ones in various respects. All these algorithms are based on the construction of the suffix tree T for either the string *pattern* or the string *text*\$*pattern*. T has to be modified to efficiently support the *Lowest Common Ancestor* algorithm by [HT84] or the simplified version devised by [SV86]. It is worth to point out that such preprocessing algorithms are interesting in their own right. For instance, the parallel ones efficiently construct a fundamental data structure for most string matching algorithms such

as the suffix tree [A84]. It was an open problem to efficiently compute such data structure in parallel [GA85].

The text analysis algorithms in [LV85a,LV85b,GG86a,GG86b] test, in increasing order, each substring $t[i, i+m-1]$ of the text in an attempt to locate (up to) k mismatches between $t[i, i+m-1]$ and the pattern. Such a process is a natural extension of classic string matching strategies. On the other hand, the analysis algorithms in [LV85a, LV86, MY86, BLLP87] locate all occurrences, with at most k differences, of the pattern in the text through a table obtained by dynamic programming techniques. Such a process is a natural extension of classic algorithms for the computation of the edit distance between two strings. The main difference between the quoted analysis algorithms for the k mismatches problem and the quoted ones for the k differences is that the latter ones are based on a rigid dynamic programming scheme whereas the former ones have more degrees of freedom. Since the most efficient sequential algorithms for the k differences problem [LV86, MY86, BLLP87] run in time $O(nk)$ as well as the most efficient sequential ones for the k mismatches problem [LV85a,LV85b, LV86, GG86a] do, this difference does not seem to be significant in the context of sequential algorithms but it seems to play some role in the context of parallel algorithms.

The preprocessing algorithm by [UKK85] constructs a finite automaton that recognizes all substrings of the text at edit distance at most k from the pattern. Such a construction can be performed in $O(m\sigma K)$ time, where $K = min(3^m, 2^k\sigma^k m^{k+1})$ and σ denotes the size of the input alphabet. Then, the analysis of the text is easily completed in $O(n)$ time. Obviously, such an algorithm is practical only when m is very small compared to n. A similar method is used in [I84] for the k mismatches problem in order to obtain a real time algorithm for the analysis of the text.

The model of computation that we assume for the sequential algorithms is the Random Access Machine (RAM) [AHU74]. Informally, a RAM is a processor with a random access memory. The model of computation that we assume for the parallel algorithms is parallel random access machine (PRAM) [FW78]. Informally, a PRAM is composed of t synchronous processors all having access to a common memory for read or write operations. In an Exclusive Read Exclusive Write PRAM no two processors are allowed to access the same memory location for reading or writing operations. That is, read and write conflicts are forbidden. In a Concurrent Read Concurrent Write PRAM processors are allowed to access the same memory location for read or write operations. However, when a write conflict occurs among processors, only one of them succeeds in writing into the contended memory location.

This paper is organized as follows. In Section 2 we present basic methods that allow string matching algorithms to efficiently deal with input alphabets of arbitrary size. Then, in Section 3 data structures and algorithms for the preprocessing step are considered. Section 4 is dedicated to the presentation of approximate string matching algorithms. Finally, in Section 5 we give some open problems relevant to

approximate string matching.

2. On The Size Of The Alphabet

Let Σ denote the input alphabet and let σ denote its size. In this section we overview basic methods that allow string matching algorithms to deal efficiently with input alphabets of arbitrary size. The only assumption that we make in this section is that each character of Σ can be contained in a RAM register and that characters can be compared in constant time. This is equivalent to assuming the *uniform cost criterion* for RAM instructions [AHU74].

In string matching algorithms, the magnitude of σ plays a crucial role in two different contexts: In the construction of data structures such as digital search trees and finite automata and in the assignment of names to strings. The first context is relevant to RAM algorithms whereas the second one is relevant to PRAM algorithms.

We consider first the impact of σ on RAM algorithms by means of a digital search tree T of n nodes. Recall that a digital search tree for strings $x_1, x_2, ..., x_l$ is a tree in which the path from the root to a leaf is uniquely associated with a string x_i. Any two leaves having a common ancestor other than the root are associated to strings which have a prefix in common. Each internal node may have as many as σ outgoing edges, each associated with a distinct character. Assume that there is an edge $e = (u, v)$ between nodes u and v labeled a, $a \in \Sigma$. We have to organize the information in node u in such a way that, given u and a, node v can be quickly found. The obvious way to organize this information is to maintain a table $NEXT_u$ of σ entries such that $NEXT_u(a) = v$ iff (u, v) is an edge labeled a in T. This would yield a performance of $O(1)$ time per insertion, deletion and find operation and a total space of $O(n\sigma)$. Such an organization for $NEXT_u$ has a major drawback: σ may be not known *a priori* or it may be large, i.e. $\sigma > n$. There are basically two ways of solving this problem: Organize $NEXT_u$ as an hashing table or as a binary search tree.

Recall that hashing scheme is the following: A hash table $A[0, s-1]$ and a hash function $h : U \rightarrow A$. U is the name space and its cardinality is assumed to be much larger than s. Such an assumption implies that there exist elements a_1 and a_2 such that $h(a_1) = h(a_2)$.Thus, an attempt to store a_1 in $A[h(a_1)]$ can result in a collision if a_2 is already there. The storage and retrieval of information in A depends on how collisions are handled. In what follows, we denote the find operation by $A[h(a)]$ assuming that such an operation is consistent with the chosen collision method for the hashing scheme at hand.

For well chosen functions h and array size s, a hashing scheme supports insertion, deletion and find operations in $O(1)$ expected time [KNU73, MEH84] whereas, in the worst case, each operation takes $O(s)$.

In our case, since $\sigma > n$ the set U is the set of distinct characters in $x_1, x_2, ..., x_l$. A hash table $NEXT_u$ and a hash function h_u can be assigned to each node u of T. Then, an edge (u, v) labeled a is represented by storing v in $NEXT_u[h_u(a)]$. The dimension of $NEXT_u$ should be chosen to be big if u is close to the root, otherwise it can be small. In any case, the dimension of $NEXT_u$ is a multiple of the number of children of u. When $NEXT_u$ is organized with hashing tables, we obtain a performance of $O(1)$ expected time per operation and a total space of $O(n)$.

The other basic method to organize the adjacency list for a node u in T is to use binary search trees (see [KNU73,MEH84]). Each node in binary search tree $NEXT_u$ has a key field a_i and an info field v iff there is an edge (u, v) labeled a_i in T. $NEXT_u$ can support insertion, deletion and find operations in $O(\log s)$ time, where s is the number of elements in $NEXT_u$.

Since this organization of $NEXT_u$ can be applied either when $\sigma < n$ or $\sigma \geq n$, we have that $s = \min(\sigma, n)$. Thus, we obtain a performance of $\log \tilde{n} = \log \min(\sigma, n)$ per operation and a total space of $O(n)$. In the remaining part of this paper, we assume that digital search trees and finite automata have adjacency lists organized as binary search trees.

In the context of parallel computation the size of the alphabet is relevant when one wants to assign names to substrings of length l of a given substring $x = x[0, n-1]$. A name for a substring $x[i, i+l]$ of x is simply an integer $k, 0 \leq k \leq n-1$. Every occurrence of $x[i, i+l]$ in x has the same name and different substrings of length l have different names. In the next section we consider efficient parallel algorithms for the naming of strings. Here we limit ourselves to consider the case in which we want to assign names to characters. For this purpose, we use n processors and assume that processors can concurrently access the same memory location. Processor p_i is in charge of the character in position i of x.

If $\sigma < n$, we can assign names to characters as follows. The alphabet Σ is sorted by means of σ processors and the result is stored in an array $A[0, \sigma - 1]$. Then, each processor p_i performs binary search on array A looking for character $x[i]$. Processor p_i assigns name j to $x[i]$ iff $A(j) = x[i]$. This procedure takes $O(\log \sigma)$ on a PRAM. When the alphabet size is larger than n, we can use the same procedure just outlined but we sort the characters in x. In this case, we obtain a naming process that takes $O(\log n)$ time on a PRAM. Thus, we obtain a time performance of $\log \tilde{n}$ with $min(\sigma, n)$ processors for the naming process.

3. Preprocessing Algorithms

In this section we present preprocessing algorithms that efficiently support the analysis of the text reported in [LV85a,LV85b,LV86,GG86a,GG86b,MY86,BLLP87].

The presentation of the preprocessing algorithm by [UKK85] is deferred to next section since it exploits ideas discussed in that section.

We recall that the algorithms discussed here must set up data structures to find, on line, the leftmost mismatch between any suffixes of the pattern and the text in $O(1)$ time. Given suffixes $t[i, n-1]$ and $p[j, m-1]$ of the text and the pattern, respectively, the leftmost mismatch between them can be easily located by finding the length of their longest common prefix.

Thus, in general, the problem we are dealing with is the following : Given strings $x = x[0, n-1]$ and $y = y[0, m-1]$, $m \leq n$, devise data structures and algorithms that allow a fast answer ($O(1)$ time) to the queries $Prefix(i, j)$, where $Prefix(i, j)$ is the length of the longest common prefix between $x[i, n-1]$ and $y[j, m-1]$.

3.1 *Preprocessing on a RAM*

On a RAM, a solution to this problem can be easily obtained by using as a data structure the suffix tree of a string [Mc76, W73] modified in order to support the static lowest common ancestor algorithm (LCA for short) given in [HT84] or in [SV86]. A *suffix tree* T of a string $z = z[0, k-1]\$$, shown in Fig. 1, is a digital search tree of at most $2k$ nodes containing all suffixes of that string. The character \$ is a right end marker that matches no character of z. Its function is to separate (in T) suffix $z[i, k-1]$ from $z[j, k-1]$ whenever the former is a prefix of the latter. Thus, each leaf of T can be labeled with a distinct integer j so that the path from the root of T to leaf (labeled) j corresponds to suffix $z[j, k-1]\$$. Moreover, the path from the root of T to an internal node u corresponds to a substring of z. We refer to such string as $w(u)$. Fast sequential algorithms for the construction of T are reported in [Mc76, W73] and both algorithms take $O(k \log \tilde{k})$. Notice that such time bound becomes $O(k)$ whenever σ is a constant.

Given suffixes $z[i, k]$ and $z[j, k]$, the length of their longest common prefix is given by $|w(u)|$, where u is the lowest common ancestor between leaves i and j of T. Node u can be found in $O(1)$ time by means of the LCA algorithm provided that T has been modified as described in [HT84] or in [SV86]. This transformation can be performed in $O(k)$ time. In what follows, we assume without loss of generality that $LCA(i, j)$ returns the length of the longest common prefix between suffixes $z[i, k]$ and $z[j, k]$.

Algorithm P1: Construct the suffix tree T for the string $x\hat{\$}y\$$ ($\hat{\$}, \$$ being end markers) and modify it to efficiently support the LCA algorithm. Then, $Prefix(i, j)$ is computed in $O(1)$ by finding $LCA(i, j+n+1)$.

Algorithm **P1** takes $O((n+m) \log \tilde{m})$ time and $O(n+m)$ space.

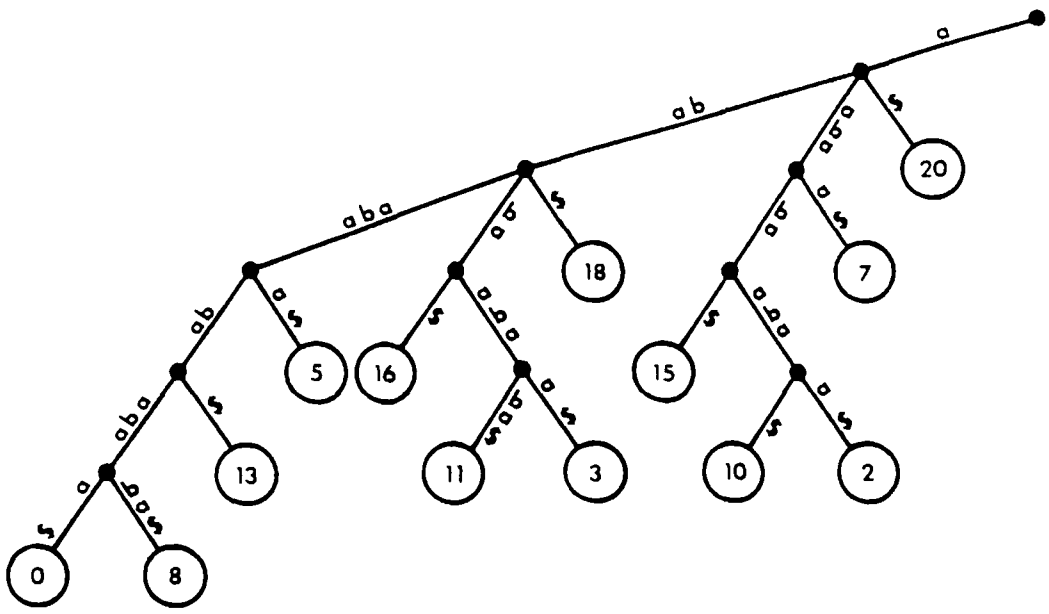

Figure 1

A partial view (all suffixes starting with a) of the suffix tree of the string abaababaabaababaababa$

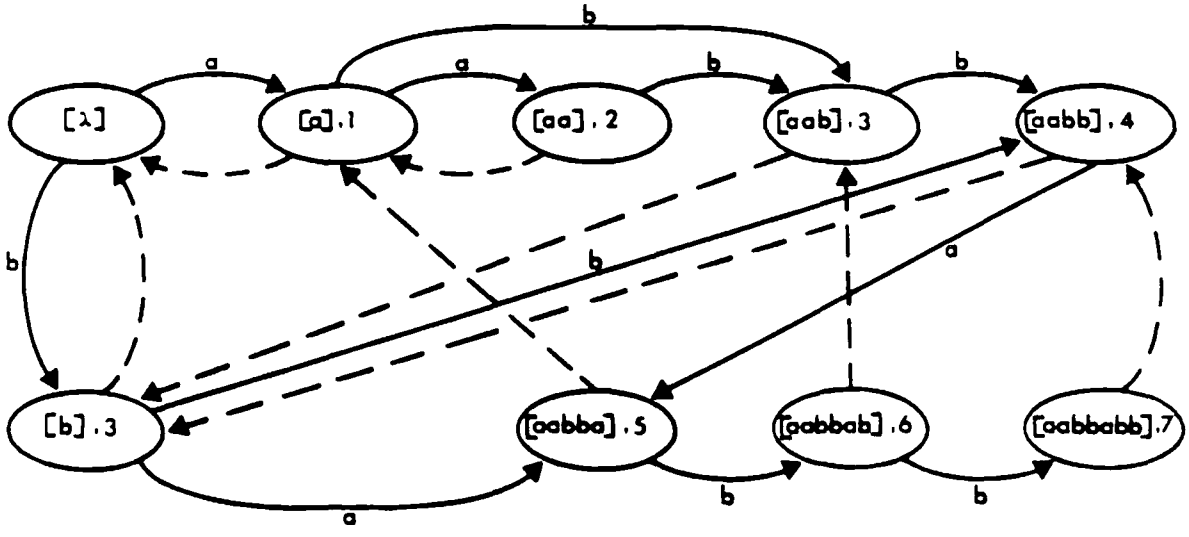

Figure 2

A DFA $F(x)$ for string x = aabbabb. Solid lines denote *next* transitions and and dotted lines denote *fail* transitions. For each state q, it is also shown $p(q)$

Such an algorithm has been devised by Landau and Vishkin in order to obtain efficient algorithms for string matching with k mismatches and string matching with k differences [LV86]. Its major drawback is that the constants hidden in its asymptotic time and space bounds are quite large. This is mainly due to the construction of the suffix tree for $x\$y\$$ and to the need of adapting it to support the LCA algorithm.

We can save on time and space by pursuing a different approach to the solution of the posed problem. Indeed, we can obtain an algorithm whose time performance is asymptotically the same as in algorithm **P1**, but with much smaller constants hidden in the big-O notation since it constructs the suffix tree only for string $y\$$. Moreover, it never uses more than $2n + O(m)$ space.

Let Σ^* denote the Kleene closure of the input alphabet Σ. Then, let λ denote the empty string and let Σ^+ denote $\Sigma^* - \{\lambda\}$. Recall that a *substring* $u \in \Sigma^*$ of y is any string such that $y = vuw$, v and w in Σ^*. Let $S(y)$ denote the set of substrings of y .

Let $BEST - FIT[0, n-1]$ be an array such that $BEST - FIT[i] = (g, l)$ iff $y[g, g+l-1]$ is a longest substring of y occurring at position i of x. We later show that $BEST - FIT[0, n-1]$ can be computed in $O((n+m)\log \tilde{m})$ time.

Algorithm P2: Construct $BEST - FIT[0, n-1]$ and the suffix tree T for string $y\$$ adapting it to efficiently support the LCA algorithm. Then, $Prefix(i, j) = min(l, q)$ where $BEST - FIT[i] = (g, l)$ and $q = LCA(g, j)$.

In order to construct $BEST-FIT[0, n-1]$ we can compute a parse of x in terms of the longest substrings of y occurring in x. Such a parse can be represented as a sequence of triples of integers $(l_g, r_g, p_g), 0 \leq g \leq s$, with $(l_0, r_0, p_0) = (-1, -1, -1)$ and $r_s \leq n-1$. Each $(l_g, r_g, p_g), 0 \leq g \leq s$, denotes the fact that $x[l_g, r_g]$ matches a substring w of y of maximal length which starts at position p_g of y. The substrings of y giving such a parse of x may overlap in x.

For $g > 0$, (l_g, r_g, p_g) is defined in terms of $x[l_{g-1}, r_{g-1}]$ as follows.

If there exist positions $i, l_{g-1} < i \leq r_{g-1}$, such that $x[i, r_{g-1}+1]$ matches a substring of y, l_g is set equal to the minimal such i. That is, $x[l_g, r_{g-1}]$ is the longest suffix of $x[l_{g-1}, r_{g-1}]$ that can be extended to match a substring of y of length greater than $r_{g-1} - l_g + 1$. On the other hand, if there is no such position i, l_g is set equal to the minimal position of $x[r_{g-1}+1, n-1]$ such that $x[l_g]$ matches a character of y. In any case, r_g is the maximal position of $x[l_g, n-1]$ such that $x[l_g, r_g]$ matches a substring w of y and p_g is a starting position of w in y.

Assuming that the sequence $(l_g, r_g, p_g), 0 \leq g \leq s$, is known, $BEST - FIT[i]$ can be easily computed. Indeed, let l_c be the largest integer in $l_0, ..., l_s$ such that $l_c \leq i$. Then, if $i \leq r_c$, $BEST - FIT[i] = (p_c + i - l_c, r_c - i + 1)$; otherwise $BEST - FIT[i] = (0, 0)$. Since the sequence $l_0, ..., l_s$ is increasing, we can compute $BEST - FIT[0, n-1]$ in $O(n)$ time starting from $i = 0$.

In order to obtain the sequence $(l_g, r_g, p_g), 1 \leq g \leq s$, we can construct a machine that, on input x, outputs a parse of x in terms of the longest substrings of y occurring in x. Such a machine is easily obtained by suitably modifying the minimal DFA $\mathbf{F}(y)$ recognizing all substrings of y. However, for ease of presentation, we construct $BEST - FIT[0, n-1]$ by using two automata $\mathbf{F}(y)$ and $\mathbf{F}(y^r)$ for strings y and y^r (y reverse), respectively.

On a generic string z, $\mathbf{F}(z)$ can be defined by means of the following equivalence relation on Σ^* denoted by $\equiv_z$ (or $\equiv$ when z is understood):

$u \equiv v$ iff $\forall w \in \Sigma^* : uw \in \mathbf{S}(z) \leftrightarrow vw \in \mathbf{S}(z)$.

The states of $\mathbf{F}(z)$ are then the equivalence classes, denoted by $[u]_z$ (or $[u]$ when z is understood), of the strings $u \in \mathbf{S}(z)$. The transition function $next$ of $\mathbf{F}(z)$ is defined as: $next([u], a) = [ua], u \in \mathbf{S}(z)$ and $a \in \Sigma$. For each state $q = [w]_z$ we define $p(q) =$ the length of the shortest prefix of z having w as suffix. Notice that $p(q)$ is well defined since if two strings are in the same class then one is a suffix of the other. We also define failure transitions as follows. For $u \in \mathbf{S}(z)$, let v be the longest suffix of u such that $v \not\equiv u$. Then, $fail([u]) = [v]$.

In Fig. 2 a minimal DFA $\mathbf{F}(z)$ for string $z = aabbabb$ is shown.

In what follows, we denote by $next^+$ the application of function $next$ to a string of characters.

An on line algorithm for the construction of $\mathbf{F}(z)$ is reported in [CR84]. Its time performance is $O(|z| \log |\tilde{z}|)$.

Assume that $\mathbf{F}(y)$ and $\mathbf{F}(y^r)$ have been constructed. The sequence $(l_g, r_g, p_g), 1 \leq g \leq s$, can be computed in two phases as follows. We compute first the sequence $r_1, r_2, ..., r_s$ by performing string matching between x and y by means of $\mathbf{F}(y)$ (phase one). Then, the sequences $l_1, l_2, ..., l_s$ and $p_1, p_2, ..., p_s$ are computed by performing string matching between x^r and y^r by means of $\mathbf{F}(y^r)$ (phase two).

At the start of phase one, $\mathbf{F}(y)$ is in its initial state $q = [\lambda]$ reading input $x[0]$ and string $x[0, n-1]$ has to be processed.

As long as $next([\lambda], x[j]) = undefined$, $\mathbf{F}(y)$ does not change state. Let $x[i]$ be the leftmost character of x such that $next([\lambda], x[i])$ is defined and let k be the smallest integer such that $next^+([\lambda], x[i, i+k-1]) = q$ and $next(q, x[i+k]) = undefined$, for some state q of $\mathbf{F}(y)$. That is, $x[i, i+k-1] = y[p(q)-k, p(q)-1]$ and this is a match of maximal length among all substrings of y. Then, r_1 can be set equal to $i+k-1$.

Since a mismatch has been detected, $\mathbf{F}(y)$ changes state by means of $fail$ transitions reaching, from q, the closest state q' such that either ($next(q', x[i+k]) = defined$ and $q' \neq [\lambda]$) or ($q' = [\lambda]$). It can be easily shown that if $next(q', x[i+k])$ is defined and $q' \neq [\lambda]$, there exists a maximal length suffix w of $x[i, i+k-1]$ that

can be extended to match a factor of y of length at least $|w|+1$. Moreover, the equivalence class of w is represented by q' in $\mathbf{F}(y)$. On the other hand, if $q' = [\lambda]$ there is no such suffix w of $x[i, i+k-1]$. In any case, $\mathbf{F}(y)$ reaches the correct state for the computation of r_2. Thus, the string matching algorithm is resumed with $\mathbf{F}(y)$ in state q' reading input $x[i+k]$ and string $x[i+k, n-1]$ remains to be processed.

The output of the algorithm outlined above is a sequence $r_1, r_2, ..., r_s$ of positions of x in which the maximal substrings of y occurring in x end. During phase two, we find the sequence $l_1, l_2, ..., l_s$ of the starting points of those substrings in x. Moreover, for each (l_g, r_g) we also find a position p_g of y where an occurrence of $x[l_g, r_g]$ starts.

During phase two we perform string matching between y^r and x^r starting with $\mathbf{F}(y^r)$ in its initial state $q = [\lambda]$ reading input $x^r[0]$. In what follows $\hat{next}$ denotes the transition function of $\mathbf{F}(y^r)$.

As long as $\hat{next}([\lambda], x^r[j]) = \mathit{undefined}$, $\mathbf{F}(y^r)$ does not change state. Let $x^r[i]$ be the leftmost character of x^r such that $\hat{next}([\lambda], x^r[i])$ is defined. Notice that r_s must be equal to $n-i-1$ otherwise $x[r_s]$ could not be the last character of x that matched a character of y. Let k be the smallest integer such that $\hat{next}^+([\lambda], x^r[i, i+k-1]) = q$ and $\hat{next}(q, x^r[i+k]) = \mathit{undefined}$, for some state q of $\mathbf{F}(y)$. In terms of the strings x and y, we have that $x[n-i-k, n-i-1] = y[m-p(q), m-p(q)+k-1]$ and this is a match of maximal length among all substrings of y. Since $r_s = n-i-1$, l_s must be equal to $n-i-k$ and p_s must be equal to $m-p(q)$.

Since a mismatch has been detected, $\mathbf{F}(y^r)$ changes state by means of *fail* transitions reaching, from q, the closest state $\hat{q}$ such that either $(\hat{next}(\hat{q}, x^r[i+k]) = \mathit{defined}$ and $\hat{q} \neq [\lambda])$ or $(\hat{q} = [\lambda])$. By using the same arguments as for $\mathbf{F}(y)$ and state q' above, it can be shown that state $\hat{q}$ of $\mathbf{F}(y^r)$ is the correct state for the computation of l_{s-1}. Thus, the string matching algorithm is resumed with $\mathbf{F}(y^r)$ in state $\hat{q}$ reading input $x^r[i+k]$ and string $x^r[i+k, n-1]$ remains to be processed.

The method outlined above for the computation of $BEST-FIT[0, n-1]$ is essentially based on a string matching algorithm by means of an automaton $\mathbf{F}$, hence we conclude that the computation of $BEST-FIT[0, n-1]$ can be performed in $O((n+m)\log\tilde{m})$ time.

Comparing algorithm **P2** with **P1**, it is easily seen that an implementation of **P2** needs the construction of the automata $\mathbf{F}(y)$, $\mathbf{F}(y^r)$ and of the suffix tree T for y adapted to efficiently support the LCA algorithm. Then, $BEST-FIT[0, n-1]$ is quickly computed by means of a simple scan of the string x. Thus, the most time consuming tasks, i.e. suffix tree construction and LCA preprocessing algorithm are performed on the string y which is usually much shorter than x. This should be contrasted with algorithm **P1**. Based on these considerations, we claim that algorithm **P2** is less time consuming than algorithm **P1**, although they both have the same asymptotic time performances. Moreover, algorithm **P2** uses only $2n+$

$O(m)$ space. This should also be contrasted with algorithm **P1** which uses $O(n+m)$ space.

3.2 *Preprocessing on a PRAM*

We can now turn our attention to an implementation of algorithms **P1** and **P2** on a PRAM. Since both these algorithms revolve around the suffix tree T adapted to efficiently support the LCA algorithm, we need to show how such a data structure can be efficiently constructed on a PRAM. In what follows, we present an efficient parallel algorithm for the construction of the suffix tree devised by [AILSV87]. The time performance of this algorithm is evaluated assuming the Concurrent Read Concurrent Write (CRCW) PRAM model of computation.

We first present an algorithm that constructs the suffix tree T_z for a string $z = z[0, k-1]\$$ in $O(\log k)$ time by using k processors and $O(k^2)$ space. Then, we show how to reduce the space from $O(k^2)$ to $O(k^{1+\epsilon(k)})$, $0 < \epsilon(k) \leq 1$.

The algorithm consists of two main parts. In the first part, an approximate version D_z of the tree is constructed, called the *skeleton*. In the second part, the skeleton is refined to obtain T_z. In the following presentation we assume without loss of generality that $k = 2^q$ and that the input alphabet is binary. We also extend z by appending to it $k-1$ instances of the symbol \$. We refer to such an extension as $z\$^{k-1}$.

The skeleton D_z is constructed in $\log k$ stages by successively creating nodes at stage $q, 0 \leq q \leq \log k$. The main features of D_z are now reported. Each node u of D_z has a descriptor (i, l) associated with it, $l = 2^q$ for some $q < \log k$. We refer to integer q as the stage number of u. Such a descriptor indicates that the path from the root of D_z to u is associated with string $z[i, i+l-1] = w(u)$. Links in D_z go from children to parents. Any node u in D_z is allowed to have c children if and only if there exist $c \geq 2$ distinct substrings $s_1, ..., s_c$ of $z\$^{k-1}$ all of which have $w(u)$ as prefix and such that $|s_t| = 2|w(u)|, 1 \leq t \leq c$. This constraint rules out non-branching nodes from D_z.

For the computation of D_z, we use k processors $p_0, p_1, ..., p_{k-1}$, a *Bulletin Board* BB of $k(k+1)$ locations and $2k$ arrays ID_i and $NODE_i$ of $\log k + 1$ cells each . The access to BB is ruled by allowing each processor to attempt to write in the same location, but only one processor can succeed. Let $winner(i)$ be the processor which succeeds in writing in the location attempted by p_i. The role played by ID_i and $NODE_i$ in the algorithm is best seen in terms of their contents at the end of the computation. At that time, $ID_i[q] = j$, $0 \leq q \leq \log k$, if and only if $z[i, i+2^q-1] = z[j, j+2^q-1]$ and j is the first component in the descriptor of all occurrences of $w = z[i, i+2^q-1]$ in $z\$^{k-1}$. That is, j is the "nickname" of w in $z\$^{k-1}$. Array $NODE_i$ records nodes of D_z. Indeed, if for some value of

$q < \log k$. $NODE_i[q]$ is not empty, then it represents a node u as follows: the field $NODE_i[q].LABEL$ is a replica of $ID_i[q]$, and the field $NODE_i[q].PARENT$ points to the location of the parent of u. Finally, $NODE_i[\log k]$ stores the leaf labeled (i, k). It is convenient to extend the notion of ID to all positions $i \geq k$ through the convention : $ID_i[q] = k$, $i \geq k$.

At the beginning of the computation, processor p_i is assigned to position i of z. Assuming that each character of z has been assigned a name as described in Section 2., the values of $ID_i[0]$ and $NODE_i[0]$ can be computed as follows. Processor p_i attempts to write in $BB[1, s]$, if $z[i] = s \in \Sigma$. Then, each processor p_i sets $ID_i[0] = winner(i)$. In this way, nicknames for the characters of z are computed. Each winning processor p_i sets $NODE_i[0].PARENT = ROOT$ and $NODE_i[0].LABEL = ID_i[0]$. We can now describe stage $q \leq \log k - 1$ for the construction of D_z, assuming that $ID_i[j]$ and $NODE_i[j]$ have been correctly computed, for $0 \leq j < q$ and for $0 \leq i < k$.

Each processor p_i creates a composite label $TID_i = (ID_i[q], ID_{i+2^q}[q])$ and attempts to write i in $BB[TID_i]$. Thus, all processors with the same TID attempt to write in the same location of BB. Then processors read: if $BB[TID_i] = j$ then p_i sets $ID_i[q+1] = j$. That is, a nickname j has been created for all occurrences of $w = z[i, i + 2^{q+1} - 1]$ in $z\$^{k-1}$. Formally, the assignment of nicknames is performed in parallel as follows: $ID_i[q+1] = winner(i)$.

Now the winners can create new nodes while the losers are idle. A successful processor p_i creates a new node by setting a link to $NODE_{ID_i[q]}[q]$ in $NODE_i[q+1].PARENT$. We denote such an operation as $NODE_i[q+1].PARENT \leftarrow NODE_{ID_i[q]}[q]$. Moreover, $NODE_i[q+1].LABEL$ is set to $ID_i[q+1]$, i.e. $NODE_i[q+1].LABEL \leftarrow ID_i[q+1]$. However, a newly created node u can be a non-branching node. This condition is easily checked by keeping track of how many winners accessed row $ID_i[q]$ of BB. If exactly one winner accessed row $ID_i[q]$, then u is a non-branching node. In such a case, the parent v of u is removed and u is hooked to the parent of v. In the formalism above: $NODE_i[q+1].PARENT \leftarrow NODE_{ID_i[q]}[q].PARENT$; $NODE_{ID_i[q]}[q].PARENT \leftarrow \lambda$; $NODE_{ID_i[q]}[q].LABEL \leftarrow \lambda$. The value of $NODE_i[q+1].LABEL$ need not be changed because it is correctly set to the nickname of the string associated with $NODE_i[q+1]$.

Stage q can be implemented in constant time on a $CRCW\ PRAM$. Thus, on such a $PRAM$, D_z can be constructed in $O(\log k)$ time by k processors.

Prior to the transformation of D_z in T_z, the labels of the nodes in D_z are modified as follows. Let $w(u) = w(parent(u))v$ be the string associated with node u in D_z. The modified label (mlabel) for u is any pair (i, l) such that $l = |v|$ and i is a starting position of v in $z\$^{k-1}$. A processor can trivially compute the mlabel of u in $O(1)$ time knowing the $LABEL$ of u and the stagenumbers q and q' of u and $parent(u)$, respectively. Indeed, if j is the $LABEL$ of u, $(j + 2^{q'}, 2^q - 2^{q'})$ is the mlabel of u. During the first step of the transformation, all processors occupying

leaves pointing to nodes of stagenumber $\log k - 1$ change the labels of these leaves into mlabels. Then, processors compete for the common parent node by attempting to simultaneously write on it the label of the nodes which they currently occupy. The winners are marked *free*: they ascend to the parent node where they will compute mlabel for that node at the appropriate stage. The losers record the (old) label used by the winner. The $(q-1)$-th step of the transformation involves all free processors on nodes with stagenumber q or higher. The operation is the same as above. From now on, we refer to mlabels as labels.

A byproduct of this transformation is an assignment of nodes to processors satisfying the following property:

PROPERTY 1 [AILSV87]: If a node other than *ROOT* has c children, then precisely $c-1$ children have been assigned to processors. Moreover, each one of the $c-1$ processors knows the address of the unique sibling without processor.

Any assignment of nodes of a generic tree to processors enjoying PROPERTY 1 is referred to as a *legal assignment*.

The main difference between T_z and the m-labeled version of D_z is that in T_z there cannot be two siblings such that their labels describe strings of z having a common prefix. This difference is eliminated by producing $\log k - 1$ refinements $D^{(h)}$ of $D_z = D^{(\log k-1)}$, $h = \log k - 1, ..., 0$. $D^{(0)}$ is T_z with the direction of the arcs inverted.

In order to characterize $D^{(h)}$, let a *nest* be any set of children of a node in $D^{(h)}$ and let (i,l) and (j,r) be the labels of nodes in some nest of $D^{(h)}$. An integer t is a *refiner* for (i,l) and (j,r) of size t if and only if $z[i, i+t-1] = z[j, j+t-1]$. The labeling of $D^{(h)}$ is such that no two pairs of labels of nodes in the same nest admits a refiner of size 2^h. Obviously, the labeling of $D^{(\log k-1)}$ satisfies the above condition.

Given a legal assignment of processors to $D^{(h)}$, $D^{(h-1)}$ can be computed in constant time by synchronously refining all *eligible nests* of $D^{(h)}$. An *eligible nest* of $D^{(h)}$ is a nest which might admit of a refiner of size 2^{h-1}. Let $(i_1,l_1),(i_2,l_2),...,(i_m,l_m)$ be the set of labels in the eligible nest at node v of $D^{(h)}$. The refinement of this nest is performed in two steps.

STEP1: Compute *split*-labels $(ID_{i_j}[h-1], ID_{i_j+2^{h-1}}[h-1])$. Partition the children of v into equivalence classes, putting in the same class all nodes with the same first component in their split-label. For each non singleton equivalence class perform the following operations.

(1) Create a new parent node u for all nodes in the class and make u a child of v. Set the $LABEL$ of u to $(i, 2^{h-1})$, where i is the first component of the split label of the nodes in the class.

(2) Consider each child of u. For the child whose current $LABEL$ is (i_j, l_j), change $LABEL$ to $(i_j + 2^{h-1}, l_j - 2^{h-1})$.

STEP2: If more than one class resulted from the partition, STOP. Otherwise, v is a unary node with child u (created during step 1). Make u a child of the parent of v and set the $LABEL$ of u to $(i, l + 2^{h-1})$, where (i, l) is the label of the parent of v. Remove v.

The non-trivial point in the parallel implementation of steps 1 and 2 is that the $m - 1$ processors assigned to the nest are sufficient to perform the refinement in $O(1)$ time. This is achieved by letting the processors in the nest elect among themselves a substitute for the missing processor. Then, the substitute processor works for itself and for the missing processor. This election is performed by letting the processors in the nest concurrently write on the parent node of the nest. The winner becomes the substitute processor.

A legal assignment of processors to $D^{(h-1)}$ can be generated in constant time from the legal assignment of $D^{(h)}$ as follows. Let $u_1, u_2, ..., u_s$ be the new nodes generated by the synchronous application of step 1 to $D^{(h)}$. Each processor that was assigned (in $D^{(h)}$) to the children of u_i competes for u_i. The winner is assigned to u_i. Moreover, for each node v removed during step 2 and such that it has a processor assigned to it, such processor is assigned to its (unique) child. It can be easily proved by straightforward case analysis that the resulting assignment of processors to $D^{(h-1)}$ is a legal assignment.

Since $D^{(h-1)}$ can be obtained from $D^{(h)}$ in $O(1)$ time, we may conclude that $D^{(0)}$ can be computed in $O(\log k)$ time on a CRCW PRAM. The direction of the arcs in $D^{(0)}$ is inverted in order to obtain T_x. This task can be easily performed in constant time by k processors, given a legal assignment of processors to $D^{(0)}$.

Hence, suffix tree T_x can be computed in $O(\log k)$ time by using k processors and $O(k^2)$ space on a CRCW PRAM [AILSV87].

One final remark is in order. The space can be reduced from $O(k^2)$ to $O(k^{1+\epsilon(k)})$, $0 < \epsilon(k) \leq 1$ [AILSV87]. Recall that during the naming process a new name e is generated by combining a pair (i, j) of shorter names. If the second component of (i, j) is represented by means of a sequence of integers $(a_1, a_2, ..., a_{\frac{1}{\epsilon(k)}})$ in base $k^{\epsilon(k)}$, then we need only $O(k^{1+\epsilon(k)})$ space for the naming process. This results in a slow down in time by a factor of $\frac{1}{\epsilon(k)}$.

In order to obtain a parallel implementation of algorithm **P1** we also need to show how T_x can be modified to efficiently support the LCA algorithm. Such a transformation can be performed in $O(\log k)$ time by $\frac{k}{\log k}$ processors [SV86]. We omit the presentation of such algorithm referring the reader to the original paper. However, we report here that a query $LCA(i, j)$ can still be answered on line in $O(1)$ time by a single processor. Thus, we obtain the following parallel implementation of algorithm **P1**.

Algorithm PP1: Construct the suffix tree T for string $x\$y\$$ by using the algorithm by [AILSV87]. Using the algorithm by [SV86], transform T so that it

can efficiently support the LCA algorithm. Then, $Prefix(i,j)$ is computed in $O(1)$ time (by a single processor) by finding $LCA(i, j+n+1)$.

Algorithm **PP1** takes $O(\log n)$ time with $n+m+1$ processors and $O((n+m)^2)$ space on a CRCW PRAM.

An alternative to algorithm **PP1** can be obtained by an efficient parallel implementation of algorithm **P2**. Indeed, we can modify the algorithm by [AILSV87] so that it can construct T_y and, as a side effect, it can also compute $BEST-FIT[0, n-1]$. This modified algorithm works on the input string $z = y\hat{\$}x\$$ rather than on the input string $y\hat{\$}$.

Roughly speaking, the basic idea underlying this algorithm is the following. Assume that we have obtained suffix tree T_z. Let us assign color black to the leaves corresponding to positions $0, ..., m$ of z and color red to the remaining leaves. An internal node u is assigned color black if u is an ancestor of at least one black leaf, otherwise u is assigned color red. Our algorithm constructs a sequence of trees $\tilde{T}_z^{(i)}$ such that no red internal node is explicitly created and such that red leaves point directly to their black ancestor nodes. That is, paths involving only red nodes in T_z are represented in compressed form in $\tilde{T}_z^{(i)}$. Moreover, nodes are removed from $\tilde{T}_z^{(i)}$ according to the following rule: Any red leaf is removed from $\tilde{T}_z^{(i)}$ whenever its black ancestor node u becomes a non-branching node. In other words, if a black node u has only one black child in $\tilde{T}_z^{(i)}$, then all red leaves pointing to u are removed from $\tilde{T}_z^{(i)}$. In what follows, we refer to u as a superfluous node. Superfluous black nodes are also removed from $\tilde{T}_z^{(i)}$.

The sequence of trees $\tilde{T}_z^{(i)}$ converges to a tree $\tilde{T}_z$ which is composed of black nodes and, possibly, red leaves that have not been removed yet. $\tilde{T}_z$ can be transformed into T_y by deleting the red leaves of $\tilde{T}_z$. Moreover, we gain enough information to compute $BEST - FIT$ in $O(\log m)$ additional steps by keeping record of the deletion of superfluous nodes and their associated red leaves in each $\tilde{T}_z^{(i)}$.

The computation of $\tilde{T}_z$ is performed by first constructing its skeleton $\tilde{D}_z$ and then by refining it. The algorithm supporting this computation is a variation of the one by [AILSV87] for the construction of the suffix tree.

We use $n+m+1$ processors $p_0, p_1, ..., p_{n+m}$ initially assigned to positions $0, ..., n+m$ of z, respectively. A processor p_i is assigned color red if $i > m$ and color black otherwise. Red processors may not create write conflicts and none of them may be a winner. This allows to assign names to substrings of x in terms of substrings of y. During the computation of $\tilde{T}_z$, each red leaf labeled $i, i > m$ is permanently assigned to red processor p_i. Red processors are transparent to black processors.

For $i \leq m$, $ID_i[q]$ is equal j if j is the nickname of string $y[i, i+2^q-1]$ in y. For $i > m$ we define $ID_i[q] = j$ if $x[i, i+2^q-1] = y[s, s+2^q-1]$ and j is the nickname of $y[s, s+2^q-1]$ in y. $ID_i[q] = m+1$ otherwise. Note that $ID_i[0], 0 \leq i \leq n+m$,

can be correctly computed by letting black processors attempt to write in an array $INIT$ of $n+m+1$ locations initially set at -1. That is, black processor p_i attempts to write in $INIT[a], a = y_i$. The name of the winners is stored in $INIT$. Then each processor $p_i, 0 \leq i \leq n+m$ sets $ID_i[0] = INIT[z_i]$. If a red processor reads -1 it sets its $ID[0] = m+1$. For $0 < q$ and $0 \leq i \leq n+m$, $ID_i[q]$ can be correctly computed as in [AILSV87] using a BB of $(m+2) \times (m+2)$ locations. The entries of the last row and column of BB are set to $m+1$.

The structure of $\tilde{D}_x$ is the same as D for string y with the following added features. Each black node u has one more label, *mnewlabel*, (i, l_1) , with $l_1 = |w(u)|$ and i being the starting position of an occurrence of $w(u)$ in y. The mnewlabels can be computed in essentially the same way as the labels. Moreover, $\tilde{D}_x$ has red leaves pointing to black nodes.

During the construction and refinement of $\tilde{D}_x$ no generation of red internal nodes can take place. This is an obvious consequence of the writing restriction imposed on the red processors. A red processor p_i, $i > m$, stores in a register F the address of the black node that is father of the red leaf labeled i. Initially, F is set to $ROOT$. During the construction and refinement of $\tilde{D}_x$, p_i updates F by storing in it the value of $NODE.FATHER$ link owned by the black node associated with its last winner.

We next describe briefly the modification needed in the algorithm by [AILSV87]. We need only specify the actions taken by the red processors during the various stages of the computation of $\tilde{T}_x$ since black processors closely follow the algorithm by [AILSV87]. We recall that such algorithm is composed of two parts: The construction of a skeleton (part one) and then the refinement of it (part two).

Consider stage q of part one, i.e. stage q of the construction of $\tilde{D}_x$.

An active red processor p_i, $i = m+1+s$, sets $ID_i[q+1] = winner(i)$ by accessing $BB[ID_i[q], ID_{i+2^q}[q]]$. If $winner(i) < m+1$, p_i stores the value of $NODE_{winner(i)}[q+1].FATHER$ in register F. That is, p_i records the fact that the red leaf i has a new black father.

Assume that a red processor p_i , associated with suffix $x[s, n-1]$ of x, sets $ID_i[q+1] = m+1$, with $ID_i[q] < m+1$, and assume that its black parent node u is not superfluous. This means that $w(u) = x[s, s+2^q-1]$ occurs at least twice in y whereas $w(u)x[s+2^q, s+2^{q+1}-1]$ is not a substring of y. Let $v_1, v_2, ..., v_c$ be the black children of u. Since $x[s+2^q, s+2^{q+1}-1]$ is not a suffix of any of the strings $w(v_1), w(v_2), ..., w(v_c)$, it follows that $x[s+2^q, s+2^{q+1}-1]$ and the suffix of length 2^q of $w(v_j), 1 \leq j \leq c$, do not admit of a refiner of size 2^q. Thus, during the refinement of $\tilde{D}_x$, p_i needs be active only starting at stage $k = q-1$. Red processor p_i records this fact and sets $ID_i[d] = m+1$, for $q < d \leq \log m$. Such processor does not participate anymore in the construction of $\tilde{D}_x$.

Superfluous nodes can be identified by black processors as non-branching nodes. Each non-branching node is marked. The action taken by each of the red processors

$p_{r_1}, ..., p_{r_k}$, children of marked nodes, is to store in a register PD the descriptor of the black node whose address is in F. Register PD is local to each processor. Moreover, $p_{r_1}, ..., p_{r_k}$ clear their F registers and, from now on, $p_{r_1}, ..., p_{r_k}$ are idle. This has the effect of deleting the red leaves associated with $p_{r_1}, ..., p_{r_k}$. Then, black processors remove non-branching nodes, including the superfluous ones.

Recall that part two of the construction of $\tilde{T}_z$ consists of $\log m$ stages in which $\tilde{D}_z$ is refined. Each stage is composed of two steps. We now outline the actions taken by red processors during STEP 1 of the k-th stage for the refinement of $\tilde{D}_z$.

An active red processor p_i finds its equivalence class by means of $ID_i[k-1]$ and updates its parent accordingly. If $ID_i[k-1] = m+1$, p_i does nothing. Superfluous nodes can be identified as in stage q of part 1. A red processor, child of a superfluous node, stores the mnewlabel of its parent in PD, clears its F register and then it becomes idle.

Superfluous nodes are removed at STEP 2 by black processors.

At the end of the construction of $\tilde{T}_z$, each non idle red processor stores the mnewlabel of its black parent in PD and then it clears its F register. This latter action has the effect of transforming $\tilde{T}_z$ in T_y. Then, table $BEST-FIT$ is computed as follows.

Each idle red processor becomes active. Consider red processor p_k, $k = m+j+1$, and let its PD be (i, l). In terms of strings, processor p_k knows that $x[j, j+l-1] = y[i, i+l-1]$ and that $y[i, i+t]$ is the longest substring of y starting at position j of x, for some $t \geq l-1$. In order to find such t, p_k keeps comparing $ID_{k+l}[q]$ vs $ID_{i+l}[q]$, starting with $q = 0$ and increasing it as long as the two ID's are equal. If $ID_{k+l}[q'] \neq ID_{i+l}[q']$, for some q', p_k performs binary search between strings $x[j+l-1+2^{q'-1}, j+l-1+2^{q'}]$ and $y[i+l-1+2^{q'-1}, i+l-1+2^{q'}]$ by means of ID's as shown above. Thus, p_k computes t in at most $O(\log m)$ time. Then it sets $BEST-FIT[j] = (i, t)$.

Hence, $BEST-FIT[0, n-1]$ can be computed in $O(\log m)$ time as a byproduct of the modified construction of T_y.

Finally, we obtain the following parallel implementation of algorithm **P2**.

Algorithm PP2: Compute T_y and $BEST-FIT$ via string $z = y\hat{\$}x\$$. Using the algorithm by [SV86], transform T_y so that it can efficiently support the LCA algorithm. Then, $Prefix(i, j) = min(l, q)$ where $BEST-FIT[i] = (g, l)$ and $q = LCA(g, j)$.

Algorithm **PP2** takes $O(\log m)$ time with $n+m$ processors and $O(n \log n + m^2)$ space on a CRCW PRAM.

Comparing the performances of **PP1** and **PP2**, it is easily seen that **PP2** performs better than **PP1** both in terms of time and space.

We remark that another parallel implementation of algorithm **PP2** was reported in [LV86]. It takes $O(\log m)$ time with $O(n+m^2)$ processors and $O(n+m)$ space on an Exclusive Read Exclusive Write PRAM. The performance of this latter algorithm is incomparable with the performance of algorithms **PP1** and **PP2**, since the underlying models of computation are different.

4. Approximate String Matching Algorithms

In this section we consider serial and parallel algorithms for approximate string matching. We start by reviewing the basic algorithms for string matching with k mismatches. Then we consider the more general algorithms for string matching with k differences.

4.1 String Matching Algorithms with k Mismatches

Given a text string $text = t[0, n-1]$, a pattern string $pattern = p[0, m-1]$ and an integer k, $k \leq m \leq n$, we are interested in finding all occurrences of the pattern in the text with at most k mismatches, i.e. with at most k locations in which the pattern and the text have different symbols.

We first present two sequential algorithms **SM1** and **SM2**, respectively. Then, we consider parallel algorithms.

Algorithm **SM1**, reported in [GG86a], is a variation of an earlier algorithm by [LV85a]. The preprocessing algorithm associated with **SM1** consists of the construction of $T_{pattern}$, the suffix tree of the pattern, and of its modification in order to support the LCA algorithm.

Procedures implementing **SM1** are reported in Fig. 3.

SM1 tests, in increasing order, all positions of the text in order to locate occurrences of the pattern in the text with at most k mismatches. Let i_{new} be the current position to be tested and let $A_{new}[1, k+1]$ be an array in which the (up to) first $k+1$ mismatching text positions are stored. Namely, $A_{new}[s] = r$ if $t[r] \neq p[r - i_{new}]$ is the s-th mismatch between $t[i_{new}, i_{new} + m - 1]$ and $p[0, m-1]$. Let $i_{old} < i_{new}$ be such that (up to at most $k+1$ mismatches) $t[i_{old}, j] = p[0, j - i_{old}]$, with j maximal. Let $A_{old}[1, k+1]$ be an array containing, in increasing order, the S mismatching positions between $t[i_{old}, j]$ and $p[0, j - i_{old}]$. A_{old} for i_{old} is as A_{new} for i_{new}.

In order to test an occurrence of $p[0, m-1]$ at i_{new}, we first compare the string $t[i_{new}, j]$ with $p[0, j - i_{new}]$ by using $T_{pattern}$ and the S positions in A_{old} (**Procedure MERGE**). Indeed, assume that the first $q \leq k$ mismatches between $t[i_{new}, j]$ and $p[0, j - i_{new}]$ have been found by using the first $s - 1 < S$ entries of A_{old}, i.e. it has been found that prefixes $t[i_{new}, i-1]$ and $p[0, i - i_{new} - 1]$

```
Procedure SM1
begin
 i_old = 0; j = 0; S = 0;
 for i_new = 0 to n-m do
    begin
     q = 0; if i_new < j then  MERGE(i_new,i_old,j,S,q);
     if q < k+1 then  [EXTEND(i_new,j,q); i_old = i_new; A_old = A_new; S = q;]
     if q < k+1 then print(i_new,A_old);
                 {i_new is an occurrence of pattern in text; mismatching positions are in A_old}
    end
end {SM1}

Procedure EXTEND(i_new,j,q)
begin
 while (q < k+1)  and  (j-i_new < m)  do
  begin
  j = j+1;  if  t[j] ≠ p[j-i_new]  then q = q+1; A_new[q] = j;
  end
end {EXTEND}


Procedure MERGE(i_new,i_old,j,S,q)
begin
s = 1; i = i_new;
while  i ≤ A_old[S]  and  q ≤ k  do
  begin
  l = LCA(T,i-i_new,i-i_old);
    begin-case
    1. i+l < A_old[s] :   q = q+1; A_new[q] = i+l; i = i+l+1;

    2. i+l = A_old[s] :   if  t[A_old[s]] ≠ p[i-i_new+l]  then  [q = q+1; A_new[q] = A_old[s]]
                          i = A_old[s]+1; s = s+1;


    3. i+l > A_old[s] :  q = q+1; A_new[q] = A_old[s]; i = A_old[s]+1; s = s+1;

    end-case

  end

while  q ≤ k  do
  begin
   l = LCA(T,i-i_new,i-i_old)
   if  i+l ≤ j  then  q = q+1;  A_new[q] = i+l;  i = i+l+1;
               else return
  end

end {MERGE}
```

Figure 3

Algorithm SM1

have q mismatching positions. Then, $A_{old}[s]$ can be used to locate the $q+1-st$ mismatch, i.e. the first mismatch between suffixes $t[i,j]$ and $p[i-i_{new}, j-i_{new}]$. Let $l = LCA(T_{pattern}, i-i_{new}, i-i_{old})$ and notice that $i \leq A_{old}[s]$. The following three cases may arise:

1. $i+l < A_{old}[s]$. By noting that $A_{old}[s]$ is the first mismatch between $t[i,j]$ and $p[i-i_{old}, j-i_{old}]$ and that $p[i-i_{new}+l] \neq p[i-i_{old}+l]$, with $p[i-i_{new}, i-i_{new}+l-1] = p[i-i_{old}, i-i_{old}+l-1]$, we can conclude that $t[i+l] \neq p[i-i_{new}+l]$. Thus, $i+l$ can be stored in $A_{new}[q+1]$ and $A_{old}[s]$ is still usable to detect mismatches (and we do not change s).

2. $i+l = A_{old}[s]$. An analysis similar to (1) shows that $t[A_{old}[s]]$ must be compared with $p[i-i_{new}+l]$. $A_{old}[s]$ is stored in $A_{new}[q+1]$, if there is a mismatch. In any case, we increment s ($A_{old}[s]$ is useless from now on) and we continue looking for mismatches starting at $i = A_{old}[s]+1$.

3. $i+l > A_{old}[s]$. An analysis similar to (1) and (2) shows that $t[A_{old}[s]] \neq p[A_{old}[s]-i_{new}]$. Thus, $A_{old}[s]$ can be stored in $A_{new}[q+1]$ and we increment s ($A_{old}[s]$ is useless from now on).

Assume now that $Q \leq k$ mismatches have been detected at the time $A_{old}[S]$ becomes useless. If $A_{old}[S] < j$, we still have to compare $t[A_{old}[S]+1, j]$ with $p[A_{old}[S]-i_{new}+1, j-i_{new}]$. Observing that $t[A_{old}[S]+1, j]$ matches a suffix of $p[0, j-i_{old}]$, this can be efficiently done by using $T_{pattern}$ and the LCA algorithm (second **while** loop in **Procedure MERGE**).

As soon as it is guaranteed that only $Q \leq k$ mismatches exist between $t[i_{new}, j]$ and $p[0, j-i_{new}]$, it can be directly checked whether $t[j+1, i_{new}+m-1]$ and $p[j-i_{new}+1, m-1]$ have at most $k-Q$ mismatching positions (**Procedure EXTEND**). In such a case i_{new} is an occurrence of the pattern in the text.

The performance of algorithm **SM1** is evaluated as follows. **Procedure EXTEND** scans each character of the text at most once contributing $O(n)$ time. On the other hand, **Procedure MERGE** takes time proportional to the size of $A_{old} + A_{new}$, that is at most $2k+2$. Since it can be called at most $O(n)$ times, **Procedure MERGE** contributes $O(kn)$ time. The construction of the suffix tree $T_{pattern}$ takes $O(m \log \bar{m})$. Thus, the total time is $O(m \log \bar{m} + kn)$. The space required by the algorithm is $O(m+k) = O(m)$.

We can obtain a more compact version of algorithm **SM1** by resorting to one of the preprocessing algorithms presented in Section 3. Indeed, assume that we can compute $Prefix(i,j)$ in constant time. Then, we can find the first (leftmost) mismatch between $p[0, m-1]$ and $t[i, i+m-1]$ in $O(1)$ time. If we keep track of where this mismatch occurs, say at position l of *pattern*, we can locate the second mismatch, in $O(1)$ time, by finding the leftmost mismatch between $p[l+1, m-1]$ and $t[i+l+1, i+m-1]$. In general, the q-th mismatch between $p[0, m-1]$ and

$t[i, i+m-1]$ can be found in $O(1)$ time by knowing the location of the $(q-1)$-th mismatch. Algorithm **SM2**, reported in Fig. 4, is based on the method just outlined .

The time complexity of algorithm **SM2** is obviously $O(nk)$, since the innermost while loop takes $O(k)$ time. Accounting for the preprocessing phase, we obtain the time bound of $O(nk + (n+m)\log m)$. The space required by algorithm **SM2** is $O(n+m)$.

For alphabets of fixed size, the main advantage of **SM2** with respect to **SM1** is its simplicity of the analysis of the text that yields a speed-up of the same phase in **SM1**. This result is obtained at the expense of a slight increase in the complexity of the preprocessing phase . However, both algorithms have the same asymptotic time performance of $O(nk + m)$. For general alphabets, such an advantage is preserved provided that $k > \log m$. Otherwise, **SM1** performs better than **SM2**.

There are also some heuristics that might speed up the identification of an occurrence of the pattern in the text with at most k mismatches. For instance, we can identify all distinct substrings of length m of the *text* by means of the suffix tree T_{text}. Let $s \leq n$ be the number of such distinct substrings. Then, we can identify the first k mismatches between a distinct substring $t[i, i+m-1]$ and *pattern* in $O(k)$ time. In this way, we can find all occurrences of the pattern in the text in $O(sk)$ time.

We now present two efficient parallel algorithms for string matching with k mismatches devised by [GG86b]. One algorithm, referred to as **PM1**, is a parallel implementation of Schönhage and Strassen integer multiplication algorithm adapted to compute the Hamming distances between a binary pattern and its potential occurrences in a binary text. Such an algorithm uses $O(nq \log m \log\log n)$ processors and it takes $O(\log n)$ time, where $q = min(\sigma, m)$. We remark that the use of fast integer multiplication algorithms to efficiently solve a wide class of string matching problems is not new [FP74] (see also [HM85]), and we discuss **PM1** for the sake of completeness.

The other algorithm, referred to as **PM2**, assigns $2k$ processors to each position of the text and then it locates (up to) $k+1$ mismatches between the pattern and such a substring of the text. It consists of two major steps: (a) preprocessing of the pattern and the text and (b) finding all occurrences of the pattern in the text with at most k mismatches. Step (b) can be implemented in time $O(\frac{\log(m/k)\log k}{\log\log m + \log\log k})$ by using $O(nk)$ processors. The preprocessing step can be implemented by means of either **PP1** or **PP2** reported in Section 3.2 .

We remark that the k mismatches problem can also be solved by the parallel algorithm for the k differences problem reported in Section 4.2. As it is shown later, that algorithm runs in $O(k + \log m)$ time with n processors. Thus, for the special case of string matching with k mismatches, its performance is comparable with **PM1** and **PM2** only when $k \leq \log m$.

```
Procedure SM2
begin
   for i = 1 to n-1 do
    begin
      pt := i; v := 1; nms := 0
   **initially t[i,i+m-1] is aligned with p[0,m-1]**
   **and no mismatch has been found**
      while v < m and nms ≤ k do
       begin
        **find leftmost mismatch between t[pt,pt+m-1] and**
        **p[v,m-1]**
          ll := PREFIX(pt,v;)
        **t[pt+ll] ≠ p[v+ll] is leftmost mismatch**
         if v+ll < m then nms := nms+1;
         pt := pt+ll+1; v := v+ll+1;
       end
      if nms ≤ k then found match;

    end
end {SM2}
```

Figure 4

Algorithm SM2

Assume that *text* and *pattern* are binary strings. Algorithm **PM1** finds the occurrences of *pattern* in *text* with at most k mismatches by computing the Hamming distance H between *pattern* and $t[i, i+m-1]$, $0 \leq i \leq n-m$. If $H(pattern, t[i, i+m-1]) \leq k$ then i is an occurrence of the pattern in the text.

The Hamming distance between two binary strings a and b of length m is given by

$$H(a,b) = \sum_{j=0}^{m-1} a[j] \oplus b[j].$$

Let $b^r = b[m-1, 0]$. Since $a[j] \oplus b[j] = (a[j]\overline{b[j]}) + (\overline{a[j]}b[j])$, $H(a,b)$ can be rewritten as

$$H(a,b) = \sum_{j=0}^{m-1} (a[j]\overline{b^r[m-j]}) + \sum_{j=0}^{m-1} (\overline{a[j]}b^r[m-j]).$$

$H(a,b)$ can be computed by first inserting $\log m$ 0's between each bit of a and each bit of b thus obtaining two strings a' and b' of length $m(\log m + 1)$ each. Then, the products $c = a'\overline{(b')^r}$ and $d = \overline{a'}(b')^r$ are computed. Finally, $H(a,b)$ is given by the sum of the two binary numbers $c_{(m-1)(logm+1)+logm} \cdots c_{(m-1)(logm+1)}$ and $d_{(m-1)(logm+1)+logm} \cdots d_{(m-1)(logm+1)}$ extracted from c and d, respectively. The role of the blocks of 0's is to separate the result from the other carries.

The above method can be easily extended to compute concurrently $H(pattern, t[i, i+m-1])$, for all i, $0 \leq i \leq n-m$. Indeed, both the text and the pattern are transformed into strings $text'$ of length $n(\log m + 1)$ and $pattern'$ of length $m(logm + 1)$. Then, the products

$c = (text')(\overline{(pattern')^r})$ and $d = (\overline{text'})((pattern')^r)$ are computed. Now, $H(pattern, t[i, i+m-1]) = c_i + d_i$, where $c_i = c_{(m-1+i+1)(\log m+1)-1} \cdots c_{(m-1+i)(\log m+1)}$ and $d_i = d_{(m-1+i+1)(\log m+1)-1} \cdots d_{(m-1+i)(\log m+1)}$.

It has been shown in [GP83] that a parallel integer multiplication of two s-bit numbers can be performed in time $O(\log s)$ with $O(s \log\log s)$ processors. Thus, parallel string matching with k mismatches can be performed in time $O(\log n)$ with $O(n \log m \log\log n)$ processors provided that the input alphabet is binary. If the size σ of the input alphabet is greater than two then each character can be represented by $q = min(\sigma, m)$ bits, i.e. the i-th character is represented by a bit vector with the i-th bit set to 1 and the remaining ones set to 0. Thus, the performance of algorithm **PM1** is $O(\log n)$ time with $O(nq \log m \log\log n)$ processors. Notice that the time bound of **PM1** has been obtained by assuming the bitwise computational model [AHU74]. A sequential implementation of **PM1** would yield an $O(nq \log n \log m \log\log n)$ algorithm for the k mismatches problem. This sequential algorithm is better than **SM1** and **SM2** when the size of the alphabet is small and k is very large.

Algorithm **PM2** is composed of two major steps:

1. Preprocessing of the pattern and the text.
2. Detection of all occurrences of the pattern in the text with at most k mismatches.

As it has been shown earlier, a basic operation in pattern matching algorithms with k mismatches is the detection of the leftmost mismatch between any suffix of the text and any suffix of the pattern. In what follows, we denote by FIND-MISMATCH(i,j) a function that gives the text position of the leftmost mismatch between $t[i,n-1]$ and $p[0,m-1]$. FIND-MISMATCH(i,j) is equal to $i + Prefix(i,j)$. Thus, it can be computed in constant time provided that the preprocessing algorithm is either **PP1** or **PP2**.

In order to test whether an occurrence of the pattern starts at position i of the text (**Procedure OCCURRENCE** in Fig. 5) $2k$ processors are used to detect (up to $k+1$) mismatches between $t[i,i+m-1]$ and $p[0,m-1]$. Processors may be active or idle depending on whether or not they are assigned to a substring of $t[i,i+m-1]$. Each active processor is assigned to a substring $t[i+j,i+s], 0 \leq j \leq s \leq m-1$ of the text and its task is to find up to the first $\log k$ (without loss of generality $k = 2^l$) mismatches between such a substring and $p[j,s]$ (**Procedure MISMATCH**). We remark that the union of the strings assigned to the active processors need not be a contiguous substring of the text. As soon as an active processor completes its task, it can be in one of two states: busy or free.

A processor, assigned to $t[i+j,i+s]$, is in the *busy* state if it detects the $\log k$-th mismatch in a position $q-1 < i+s$. That is, substring $t[q,i+s]$ has not been processed yet and it must be tested for possible mismatches. Otherwise, a processor is *free* . A busy processor reports the end points (q,s) of the substring that remains to be tested.

As soon as active processors finish their task, the number of mismatches found so far is updated by means of a parallel addition performed by the active processors. Then, the $2k$ processors are again assigned to substrings of $t[i,i+m-1]$ that have not been processed yet and each processor performs its task on the given substring. We remark that, at this stage, some processors may turn out to be idle. When such an assignment takes place we say that a new iteration is started. Initially, $t[i,i+m-1]$ is divided into k contiguous substrings of length at least $\lfloor m/k \rfloor$ and at most $\lceil m/k \rceil$. Then, k processor are assigned to each one of such strings. Subsequently, processors are assigned to strings of almost the same length as follows.

Assume that at the end of iteration j, c processors report that substrings $x_1, x_2, ..., x_c$, with endpoints $(q_1,s_1)...(q_c,s_c)$, of $t[i,i+m-1]$ remain to be tested. Notice that $c < \frac{k}{\log k}$, since each of these processors found $\log k$ mismatches. Let $z = \sum_{i=1}^{c} z_i$, $z_i = s_i - q_i + 1$. We assign $v_i = \lceil \frac{k z_i}{z} \rceil$ processors to substrings x_i. The v_i processors can be assigned to substrings x_i, for all $i, 1 \leq i \leq c < k$, by sorting the triples (q_i, s_i, k_i). Thus, at the beginning of iteration $j+1$, $\bar{k} \leq 2k$ processors

```
 Procedure OCCURRENCE
begin
   v := 0;
  **v is equal to the number of mismatches found so far**

   partition t[i,i+m-1] in k contiguous strings of roughly
   the same lenght and assign them to k processors

   while v≤k and (number active processors) > 0 do
     begin
       foreach active processor q pardo MISMATCH(q);
       update v;
       if v≤k then assign substring not processed to processors;
                 else stop;

     end

  if v≤k then print "position i is an occurrence of the pattern in the text"
end {OCCURRENCE}


Procedure MISMATCH(q)

**assume that string t[i+j,i+s] has been assigned to processor q
begin
  vv :=0; free := false; tp := i+j; pp := j; mp := 0;

**tp and pp denote current text and pattern positions, respectively.**
**mp denotes a mismatching position in the text.**

 while vv < logk or free = false do
   begin
     mp := FIND-MISMATCH(tp,pp);
     begin-case

     mp < i+s :  vv := vv+1;

     mp = i+s :  vv := vv+1; free := true;

     mp > i+s : free := true;


     end-case

     tp := tp+mp+1; pp := pp+mp+1;

   end
end {MISMATCH}
```

Figure 5

Procedures OCCURRENCE and MISMATCH of algorithm PM2

are active and each active processor is assigned to a substring of length at most $\lceil \frac{z_i}{v_i} \rceil \leq \frac{z}{k}$.

It is worth to point out that, whenever $z < 2k$, the string matching process for position i of the text is concluded as soon as active processors complete their task. This process for position i of the text is also concluded as soon as the total number of mismatches detected exceedes k.

Procedures **OCCURRENCE(i)** and **MISMATCH** implement the algorithm outlined above for the detection of an occurrence of the pattern in the text at position i.

The time complexity of **Procedure OCCURRENCE(i)** can be derived as follows. Each iteration of the **while** loop in **Procedure OCCURRENCE(i)** takes $O(\log k)$ time. Indeed, a call to **Procedure MISMATCH** takes $O(\log k)$ time since it finds up to $\log k$ mismatches by using function **FIND-MISMATCH**. The operation **update v** is a parallel addition of all the mismatches found during the current iteration and thus it can be performed in $O(\log k)$ time by $2k$ processors. Finally, the assignment of processors to substrings takes $O(logk)$ since it essentially reduces to sorting at most $\frac{k}{\log k}$ triples [C86].

The number of iterations sufficient to test whether $t[i, i+m-1]$ is an occurrence of the pattern with at most k mismatches can be derived as follows.

Let k_i and l_i be the total number of mismatches found and the total length of the substrings that remain to be processed, respectively, when iteration i is started. Initially $l_0 = m$ and $k_0 = 0$. Let $\alpha_i k$ be the number of busy processors after iteration i. Then, at completion of such iteration , $k_{i+1} \geq k_i + \alpha_i k \log k$ since at least $\alpha_i k \log k$ mismatches have been found. Moreover, $l_{i+1} \leq \frac{\alpha_i k l_i}{k} = \alpha_i l_i$ since each active processor is assigned to a string of length at most $\frac{l_i}{k}$ at the start of iteration i and at least $\alpha_i k \log k$ mismatches have been found during that iteration. The algorithm halts when either $l_i \leq 2k$ or $k_i > k$. Thus, the maximum number of iterations is given by the *maximal* s such that:

$$m \prod_{j=1}^{s} \alpha_j \geq k$$

$$\sum_{j=1}^{s} \alpha_j k \log k \leq k$$

Now, the *maximal* s is achieved when all the α_j's are equal, that is $\alpha_j = \frac{1}{s \log k}$. Thus, we obtain that $s \approx \frac{\log(m/k)}{\log\log(m/k)+\log\log k}$.

Hence, **Procedure OCCURRENCE** takes $O(\frac{\log(m/k)\log k}{\log log(m/k)+\log\log k})$.

It follows from the analysis of Procedure OCCURRENCE that the overall time complexity of algorithm PM2 is

$$O(\frac{\log(m/k)\log k}{\log logm + \log\log k} + time\ preprocessing)$$

where *time preprocessing* can be either $O(\log m)$ (**PP2**) or $O(\log n)$ (**PM1**). The number of processors needed is $O(nk)$, .

Algorithm **PM2** has a very good time performance for random strings. Indeed, consider the following restricted version of **PM2**. We define a processor to be free if and only if it finds at most 1 mismatch in the string assigned to it. Obviously, at any stage, there cannot be more than $\frac{k}{2}$ busy processors.

Letting $q \leq 1/2$ be the probability of a mismatch we find that the probability of a processor being free after the first step is $q^{(m/k)-1}$. Thus, the average number of free processors after the first step is $kq^{(m/k)-1}$ which is less than $k/2$ for $k \leq m/2$. Hence, after the first step, the number of busy processors is larger than $k/2$ on the avarage. This immediately establishes an $O(\log k)$ time bound for the algorithm.

A comparison of the performances of **PM1** and **PM2** is in order.

Algorithm **PM1** guarantees a good time performance irrespective of the order of magnitude of k. However, it has two major drawbacks: the number of processors depends linearly on q, and thus on the alphabet size, and the constant hidden in the big-O notation is quite large. Moreover, its worst case time bound is achieved by any instance of the problem. The worst case time bound of **PM2** is $O(\log m)$ whenever $k = O(\log^c m)$ or $k \geq \frac{m}{\log^c m}$, c constant, and is never worse than $O(\frac{\log^2 m}{\log\log m})$. Moreover, its time performance depends on the input strings and thus **PM2** may behave better than its worst case time bound. The major drawback of **PM2** is that if $k \approx m$ it uses essentially the same number of processors as the naive algorithm achieving the same time performance.

4.2 String Matching Algorithms with k Differences

In this section we consider the following problem:

Given strings $text = t[1,n]$, $pattern = p[1,m]$ and an integer $k, k \leq m$, find all occurrences of *pattern* in *text* with at most k differences. Three different kinds of differences are allowed:

(a) A symbol of the pattern corresponds to a different symbol of the text.

(b) A symbol of the pattern corresponds to no symbol in the text.

(c) A symbol of the text corresponds to no symbol in the pattern.

As stated in Section 1., differences (a)-(c) can be restated in terms of edit operations. Indeed, a difference of type (a), i.e. $p[i] \neq t[j]$, can be thought of as a

substitution of a character ($p[i]$) of the pattern with a character of the text ($t[j]$), a difference of type (b) can be thought of as a deletion of a character from the pattern and a difference of type (c) can be thought of as an insertion of a character in the pattern.

We present five serial algorithms that efficiently find all occurrences of *pattern* in *text* with k differences. Those algorithms, referred to as **SD1**, **SD2**, **SD3**, **SD4** and **SD5**, are based on algorithms for the computation of the edit distance between two strings [FW74,UKK83] and they allow all three differences (a)-(c). We also consider a special case of string matching with k differences where differences of type (a) are not allowed.

The first algorithm that we consider, **SD1**, has been independently discovered and published by many researchers working in various areas of science. This remarkable history is reported in [L86]. **SD1** computes a $(m+1) \times (n+1)$ matrix A according to the following recurrence relation:

$$A_{0,j} = 0, \quad 0 \leq j < n, \quad A_{i,0} = i, \quad 0 \leq i < m.$$

$$A_{i,j} = min(A_{i-1,j} + 1, A_{i,j-1} + 1, \ if\ p[i] = t[j]\ then\ A_{i-1,j-1}\ else\ A_{i-1,j-1} + 1).$$

Matrix A can be computed row by row, or column by column, in $O(nm)$ time.

It can be easily shown that $A_{i,j}$ is the minimal distance between $p[1,i]$ and a substring of *text* ending at position j of *text*. Thus, it follows that there is an occurrence of the pattern in the text ending at position j of the text if and only if $A_{m,j} \leq k$.

In what follows we refer to matrix A as the ***edit distance*** matrix between strings *pattern* and *text*.

SD1 can be improved by observing that, for any i and j, either $A_{i+1,j+1} = A_{i,j}$ or $A_{i+1,j+1} = A_{i,j} + 1$. That is, the elements along any diagonal of A form a non-decreasing sequence of integers. Thus, the computation of A can be performed by finding, for all diagonals, the rows in which $A_{i+1,j+1} = A_{i,j} + 1 \leq k$. Such an observation was exploited by [UKK83] in order to obtain a space efficient algorithm for the computation of the edit distance between two strings. Recently, Landau and Vishkin [LV86] cleverly extended the method by [UKK83] to obtain efficient algorithms, here referred to as **SD2** and **SD3**, that handle the more general problem of string matching with k differences. They also derived preprocessing algorithms that are critical for the efficiency of **SD2** and **SD3**. We start by reviewing the algorithm by [UKK83] for the computation of the edit distance between two strings since both **SD2** and **SD3** are extensions of it.

Consider strings $y = y[1,m]$ and $x = x[1,m+k]$ and consider an $(m+1) \times (m+k+1)$ matrix B defined analogously to matrix A except for the following initial

conditions:

$$B_{0,j} = j, \quad 0 \leq j \leq m + k, \quad B_{i,0} = i, \quad 0 \leq i \leq m.$$

Let a diagonal d of the matrix consists of all the $B_{i,j}$'s such that $j - i = d$.

Assume that we are interested in testing whether y is at distance at most k from x. This can be done by computing the elements of B along diagonal d, $d = -k, ..., k$. Actually, for a given diagonal d and a number of differences $e \leq k$, we compute the largest row i of B such that $B_{i,j} = e$ and $d = j - i$.

Formally, let $L_{d,e}$ denote the largest row i such that $B_{i,j} = e$ and $j - i = d$. The definition of $L_{d,e}$ implies that e is the minimal number of differences between $y[1, L_{d,e}]$ and $x[1, L_{d,e} + d]$, with $y[L_{d,e} + 1] \neq x[L_{d,e} + d + 1]$. In order to test whether y is at distance at most k from x we need to compute the values of $L_{d,e}$ that satisfy $e \leq k$.

Assuming that $L_{d+1,e-1}$, $L_{d-1,e-1}$ and $L_{d,e-1}$ have been correctly computed, $L_{d,e}$ is computed as follows. Let $row = max(L_{d+1,e-1} + 1, L_{d-1,e-1}, L_{d,e-1} + 1)$ and let j be the largest integer such that $y[row + 1, row + j] = x[d + row + 1, d + row + j]$. Then, $L_{d,e} = row + j$. The correctness of the computation of $L_{d,e}$ is derived by induction on e and the recursion satisfied by the elements of matrix B.

Let $c = row + j$ be the computed value of $L_{d,e}$. From the correctness of $L_{d',e-1}$, for $d' \in \{d - 1, d, d + 1\}$ and the recursive definition of B, it follows that $B_{row,row+d} = e$, $B_{row+i,row+i+d} = e$ for $i = 1, ..., c - row$, and $B_{c+1,c+1+d} > e$. Hence $c = L_{d,e}$.

Procedure EDIT DISTANCE in Fig. 6 is an implementation of such a method for the computation of B . As it is easily seen, its time complexity is $O(km)$ and it requires $O(k^2)$ space.

Algorithm **SD2**, devised by [LV85a], tests, in increasing order, all positions of the text in order to locate all occurrences of the pattern in the text with at most k differences. The preprocessing algorithm associated with **SD2** consists of the construction of the suffix tree $T_{pattern}$, the suffix tree of the pattern, and of its modification to support the LCA algorithm.

SD2 performs $n - m + k + 1$ iterations. During iteration i , position $i+1$ is tested by computing , as outlined above, the elements along diagonal d, $d = -k, ..., k$, of matrix B for strings $y = pattern$ and $x = t[i + 1, i + m + k + 1]$. The computation of such elements of B is sped up by using information about which substrings of the pattern matched which substrings of the text during previous iterations. In order to see how this speed up is accomplished, let j be the rightmost position of the text reached at an iteration prior to the i-th. Let $r < i$ be the first iteration in which position j was reached. That is, when the algorithm tested position $r + 1$ it inspected $t[r + 1, j]$ and, since then, no character of the text beyond $t[j]$ has been inspected.

Notice that $t[r+1, j]$ is at distance at most $k+1$ from a prefix $p[1, l]$ of the pattern. Thus, $p[1, l]$ can be transformed into $t[r+1, j]$ by means of a sequence of at most $k+1$ insert, delete and substitution operations. Such a sequence of operations establishes a correspondence between substrings of $p[1, l]$ and $t[r+1, j]$. This correspondence is such that there are at most $k+1$ successive substrings of $t[1, j]$ that match at most $k+1$ substrings of $p[1, l]$ and there are at most $k+1$ characters of $t[1, l]$ that do not match characters of $p[1, l]$ in specific positions. We encode this correspondence by means of a set S of at most $2k+2$ triples. A triple $(q, c, f) \in S$ if substring $t[q+1, q+f]$ matches substring $p[c+1, c+f]$. In particular, $(q, 0, 0) \in S$ if $t[q+1]$ does not match a specific character of $p[1, l]$. We later show how S can be computed.

Iteration i of **SD2** consists of **Procedure EDIT DISTANCE** modified in order to exploit the information contained in S for the computation of $L_{d,e}$. Recall that, given $row = max(L_{d+1,e-1}+1, L_{d-1,e-1}, L_{d,e-1}+1)$, $L_{d,e}$ can be computed by finding the leftmost mismatch between $p[row+1, m]$ and $t[i+row+d+1, n]$. We next show how **SD2** computes a single $L_{d,e}$.

As long as $i+row+d+1 \leq j$, **SD2** performs the following. It extracts from S a triple (q, c, f) such that $q \leq i+row+d \leq q+f$. This operation can be performed in $O(1)$ time [LV85a]. Then, two cases are considered:

1 . $f > 0$. $Prefix(c+1, row+1)$ is computed. Let g be its value. We now know that $p[row+1, row+g]$ matches $p[c+1, c+g]$ and that $p[c+1, c+f]$ matches $t[i+row+d+1, i+row+d+f]$. The algorithm sets variable row equal to $row+min(f, g)$. If $f \neq g$, the mismatch we are looking for has been found and $L_{d,e}$ can be set equal to $row+min(f, g)$. On the other hand, if $f = g$ no mismatch has been found and the process for the computation of $L_{d,e}$ must continue.

2 . $f = 0$. If $t[i+row+d+1] \neq p[row+1]$, the mismatch we are looking for has been found and $L_{d,e}$ can be set equal to row. If $t[i+row+d+1] = p[row+1]$, no mismatch has been found and the process for the computation of $L_{d,e}$ must continue. Variable row is set equal to $row+1$.

As soon as $i+row+d+1 > j$, **SD2** cannot use S any more for the computation of $L_{d,e}$. Thus, it directly compares $t[i+row+d+1]$ vs. $p[row+1]$ incrementing row by 1 for each match. When a mismatch is found, $L_{d,e}$ is set equal to row.

At the end of iteration i, the triples in S may become useless if characters of the text beyond $t[j]$ have been inspected and a new set of triples must be computed. In order to efficiently compute the new set of triples S when there is need to, we maintain a sequence of triples $S_{d,e}$ for each $L_{d,e}$ during itaration i. Initially, each $S_{d,e}$ is empty. Now assume that we have to compute the value of $L_{d,e}$ based on the values of $L_{d-1,e-1}$, $L_{d,e-1}$ and $L_{d+1,e-1}$. Then, we can compute $S_{d,e}$ based on $S_{d-1,e-1}$, $S_{d,e-1}$ and $S_{d+1,e-1}$. Indeed, when the computation of $L_{d,e}$ is started, $S_{d,e}$ is set equal to $S_{g,h}$, where $L_{g,h}$ achieves the maximum in the expression $r = (L_{d+1,e-1}+1, L_{d-1,e-1}, L_{d,e-1}+1)$. Actually, a link is set between $S_{d,e}$ and

```
Procedure SD2
begin
j := 0
for i = 0 to n-m+k do
 begin
 initialization as in Procedure EDIT DISTANCE
 for e = 0 to k do
  begin
    for d = -e to e do
     begin
       row := max(L_{d,e-1}+1, L_{d-1,e-1}, L_{d+1,e-1}+1);
       while i+row+d+1 ≤ j do
        begin
          extract from S a triple (q,c,f) such that q ≤ i+row+d+1 ≤ q+f;
          begin-case


          1. f ≥ 1:   row := row+min(f,PREFIX(c+1,row+1))
                      if f ≠ PREFIX(c+1,row+1) then goto Label;



          2. f = 0:       if t[i+row+d+1] = p[row+1] then row := row+1;
                                                     else goto Label;

          end-case

        end
Label:      while p[row+1] = t[row+d+1] do row := row+1;
            L_{d,e} := row;
            if L_{d,e} = m then Print"position i+1 is an occurrence of the pattern in the text";
    end
  end
 If new characters in the text have been examined, update S
 end
end {SD2}
```

Figure 7

Algorithm SD3

$S_{g,h}$. If $(g,h) = (d-1, e-1)$, or $(d, e-1)$, the triple $(i+r+d-1, 0, 0)$ is added to $S_{d,e}$ meaning that for $t[i+r+d]$ there is no match in the pattern. When $L_{d,e}$ has been computed and its value is greater that r, the triple $(i+r+d, r, L_{d,e})$ is added to $S_{d,e}$ meaning that there is a match between $t[i+r+d+1, i+L_{d,e}+d]$ and $p[r+1, L_{d,e}]$. At the end of iteration i, we check which of the $2k+1$ sequences $S_{d,k}$ reached the rightmost character in the text. If the index of this character is greater than j we take its sequence of triples as the new set S.

Procedure **SD2** reported in Fig. 7 is an implementation of the method just presented. Its time complexity can be established as follows. Observe that **SD2** maintains $2k+1$ diagonals at any time during the text analysis. For each of these diagonals, we may need to inspect a new character of the text when S becomes useless for that diagonal. Thus, **SD2** performs a total of $n(2k+1)$ character comparisons. This accounts for the cost of all character comparisons in all iterations. However, during iteration i, **SD2** also uses S to maintain the $2k+1$ diagonals. For each one of those diagonals, we may charge each operation performed to either a difference being discovered or to an element of S being examined. Since there can be at most $k+1$ differences to be discovered and at most $2k+2$ triples to be examined, we have a cost of $O(k)$ per diagonal. Thus, the total cost of the text analysis performed by **SD2** is $O(nk^2)$ since there are $2k+1$ diagonals and $n-m+k+1$ iterations. Accounting for the preprocessing phase, we obtain a $O(nk^2 + m\log\tilde{m})$ time bound for **SD2**.

Algorithm **SD3** [LV86] finds all occurrences of the pattern in the text with at most k differences by efficiently computing matrix A previously defined. The main feature of **SD3** is that it discovers the positions of the text where an occurrence of the pattern ends. This should be contrasted with the approach followed by **SD2**.

Recall that $L_{d,e}$ denotes the largest row i of A such that $A_{i,j} = e$ and $j-i = d$. The definition of $L_{d,e}$ implies that e is the minimal number of differences between $p[1, L_{d,e}]$ and the substrings of the text ending at $t[L_{d,e}+d]$, with $p[L_{d,e}+1] \neq t[L_{d,e}+d+1]$. In order to solve the k differences problem we need to compute the values of $L_{d,e}$ that satisfy $e \leq k$.

Assuming that $L_{d+1,e-1}$, $L_{d-1,e-1}$ and $L_{d,e-1}$ have been correctly computed, $L_{d,e}$ is computed as follows. Let $row = max(L_{d+1,e-1}+1, L_{d-1,e-1}, L_{d,e-1}+1)$ and let j be the largest integer such that $p[row+1, row+j] = t[d+row+1, d+row+j]$. Then, $L_{d,e} = row + j$. Once again, the correctness of such a computation can be easily shown.

If one makes use of the preprocessing algorithms presented in Section 3.1, $L_{d,e}$ can be computed in $O(1)$ time as follows: $L_{d,e} = row + Prefix(row+1, row+d+1)$.

Procedure **SD3** in Fig. 8 is a formalization of the ideas outlined above. Its time performance is $O(nk + (n+m)\log\tilde{m})$, where the term $(n+m)\log\tilde{m}$ accounts for the preprocessing. The space complexity of **SD3** is $O(n+m)$.

We remark that, based on **SD3**, one can obtain an algorithm for the k differences problem that has the same time complexity of **SD3** but uses only $O(m)$ space. Such an algorith may be relevant for practical purposes. In what follows we briefly outline this space efficient algorithm.

Let the preprocessing step consist of the computation of $T_{pattern}$ and of the automata $\mathrm{F}(pattern)$ and $\mathrm{F}(pattern^r)$ introduced in Section 3. Assume that the text has been divided in $\lfloor n/m \rfloor$ parts $chunk_0, chunk_1, ..., chunk_{\lfloor n/m \rfloor - 1}$ where $chunk_q = text[mq+1, mq+3m-1]$. Each $chunk_q$, $0 \leq q < \lfloor n/m \rfloor$ is processed separately as follows: Table $BEST-FIT$ for $chunk_q$ is computed and then algorithm **SD3** is applied to strings $chunk_q$ and *pattern*, $chunk_q$ being the text. Obviously, the time and space bounds are $O(nk + (n+m)\log \tilde{m})$ and $O(m)$, respectively. It should be noticed that this space efficient algorithm may detect more than once that an occurrence of *pattern* with at most k differences ends in the same position of the text.

It is worth to point out that **SD3** can be modified so that it can solve a version of the k differences problem more general than the one considered here. The generalization of the problem as well as the new algorithm are reported in [BLLP87].

A comparison of the time performances of **SD1**,**SD2** and **SD3** is in order. **SD3** is faster than both **SD1** and **SD2** whenever the size of the alphabet is small. However, for k large, i.e. $k = \Theta(m)$, **SD1** and **SD3** have the same time bound of $O(nm)$. This is true irrespective of the size of the alphabet. When the size of the alphabet is large and $k \leq \sqrt{\log m}$, **SD2** is faster than **SD3**.

Algorithm **SD3** can be efficiently implemented on a PRAM. Indeed, assume that the preprocessing phase is implemented by means of algorithm **PP2** presented in Section 3.2 . If we use n processors, the second **for loop** of **procedure SD3** can be executed in $O(1)$ time. Thus, a parallel implementation of **SD3** performs in time $O(k + \log m)$, where the term $\log m$ accounts for the preprocessing. This algorithm, due to [LV86], is so far the most efficient parallel algorithm for the k differences problem that uses only a linear number of processors. Notice that, for $k > \log m$, the advantage of parallelism is lost. It is worth pointing out that the k differences problem can be solved in $O(\log m \log k)$ time and $O(kmn)$ space by k^2mn processors. Indeed, let $M^{(s)}$ be a $(mn) \times (2k+1)$ matrix such that $M^{(s)}[(i,j),l]$ is equal to the minimal number of differences between $p[i, i+2^s-1]$ and $t[j, j+2^s+l-1]$, for $1 \leq i \leq m$, $1 \leq j \leq n$, $-k \leq l \leq k$. In order to solve the k differences problem we need to compute $M^{(\log m)}$ starting from $M^{(0)}$. Matrix $M^{(0)}$ can be easily computed in $O(1)$ time by kmn processors. Given $M^{(s)}$, $M^{(s+1)}$ can be computed in $O(\log k)$ time with k^2mn processors by computing each entry as follows:

$$M^{(s+1)}[(i,j),l] = \min_{-k \leq l' \leq k} (M^{(s)}[(i,j),l'] + M^{(s)}[(i+2^s, j+2^s+l'), l-l']).$$

We assume that undefined entries (when $l - l'$ is out of range) are infinity.

```
Procedure EDIT DISTANCE
begin
 for d = -(k+1) to (k+1) do
  begin
   L_{d,|d|-2} := -∞;
   if d < 0 then L_{d,|d-1|} := |d|-1;
            else L_{d,|d-1|} := -1;
  end
 for e = 0 to k do
  begin
   for d = -e to e do
    begin
     row := max(L_{d,e-1}+1, L_{d-1,e-1}, L_{d+1,e-1}+1);
     while y[row+1] = x[row+d+1] do row := row+1;
     L_{d,e} := row;
     if L_{d,e} = m then "Print Yes and Stop";
    end
  end
end {EDIT DISTANCE}
```

Figure 6

A space efficient algorithm for the computation of the edit distance between two strings

```
Procedure SD3
begin
for d = 0 to n+1 do L_{d,-1} := -1;
for d = -(k+1) to -1 do L_{d,|d-2|} := -∞; L_{d,|d-1|} := |d-1|;

for e = 0 to k do
 begin
  for d = -e to n do
   begin
    row := max(L_{d,e-1}+1, L_{d-1,e-1}, L_{d+1,e-1}+1);
    L_{d,e} := row+PREFIX(row+1, row+1+d);
    if L_{d,e} = m then "there is an occurrence ending at t_{m+d}";
   end
 end
end {SD3}
```

Figure 8

Algorithm SD3

Thus, we can find all occurrences of the pattern in the text with at most k differences in $O(\log m \log k)$ time with k^2mn processors.

When the size of the alphabet is constant, the k differences problem can be solved in $O(\frac{nm}{\log n})$ time by using a variation of the algorithm by [MP80] for the computation of the edit distance matrix A between strings *text* and *pattern*. This algorithm, here referred to as **SD4**, follows a strategy analogous to the "four Russians algorithm" for the computation of the transitive closure of a directed graph [ADKF70]. Algorithm **SD4** is composed of two main parts. The first part is devoted to the computation of all possible $(s+1) \times (s+1)$ submatrices which can occurr in A, for a suitably chosen parameter s. Then, the second part combines $(s+1) \times (s+1)$ submatrices to form A. The following observations establish a relationship between the first and second part of **SD4**.

Let (i,j,s) denote a $(s+1) \times (s+1)$ submatrix of the edit distance matrix A whose upper left corner entry is (i,j). It is easy to prove that the values in an (i,j,s) submatrix depend solely by its initial vectors $(A_{i,j}, A_{i,j+1}, ..., A_{i,j+s})$; $(A_{i,j}, A_{i+1,j}, ..., A_{i+s,j})$ along with the two strings $p[i+1, i+s]$ and $t[j+1, j+s]$. Moreover, for each (i,j,s) submatrix, the algorithm needs to remember only its pair of final vectors $(A_{i+s,j}, A_{i+s,j+1}, ..., A_{i+s,j+s})$ and $(A_{i,j+s}, A_{i+1,j+s}, ..., A_{i+s,j+s})$ for the computation of A.

In order to compute all possible (i,j,s) submatrices and their pair of final vectors, we need to enumerate all possible strings of length s over Σ and all possible pairs of initial vectors. However, the enumeration of all possible pairs of initial vectors can be uneconomical since the values in such vectors can be integers in the range $[0,m]$.

An efficient approach to the computation of all possible (i,j,s) submatrices is the following. Let a *step* be a difference between any two horizontally or vertically adjacent matrix elements and let a *step vector* be a vector of steps. It is easily seen that an (i,j,s) submatrix may be determined by an initial value $A_{i,j}$ and two initial step vectors $(A_{i,j+1} - A_{i,j}, ..., A_{i,j+s} - A_{i,j+s-1})$ and $(A_{i+1,j} - A_{i,j}, ..., A_{i+s,j} - A_{i+s-1,j})$ along with the two strings $p[i+1, i+s]$ and $t[j+1, j+s]$. Let (H, V, x, y) denote two initial step vectors (H and V) and two strings of length s (x and y). For each possible (H, V, x, y), the first part of algorithm **SD4** computes two $(s+1)\times(s+1)$ step matrices and stores the resulting pair of final step vectors. Notice that the number of pairs of initial step vectors is $O(3^s)$ since $-1 \leq A_{i,j} - A_{i-1,j} \leq 1$ and $-1 \leq A_{i,j} - A_{i,j-1} \leq 1$.

Given a pair of initial step vectors $H = (h_{0,1}, h_{0,2}, ..., h_{0,s})$, $-1 \leq h_{0,j} \leq 1$, and $V = (v_{1,0}, v_{2,0}, ..., v_{s,0})$, $-1 \leq v_{i,0} \leq -1$, along with two strings $x[1,s]$ and $y[1,s]$, we can obtain two $(s+1) \times (s+1)$ step matrices, one horizontal and the other vertical, by iteratively computing the following recurrence relations:

$$h_{i,j} = min(R_{x[i],y[j]} - v_{i,j-1}, h_{i-1,j} - v_{i,j-1} + 1, 1),$$

$$v_{i,j} = min(R_{x[i],y[j]} - h_{i-1,j}, 1, v_{i,j-1} - h_{i-1,j} + 1),$$

where $R_{x[i],y[j]} = 0$ if $x[i] = y[j]$ and it is equal to 1 otherwise.

The resulting pair of final step vectors is given by $H' = (h_{s,1}, h_{s,2}, ..., h_{s,s})$ and $V' = (v_{1,s}, v_{2,s}, ..., v_{s,s})$.

It can be shown [MP80] that the computation of the two step matrices, given (H, V, x, y), can be performed in $O(s^2 \log s)$ time by assuming the logarithmic cost criterion for RAM [AHU74]. Such a computation must be performed for all quadruples (H, V, x, y). Since there are $O(3^s)$ initial step vectors and $O(\sigma^s)$ strings of length s, it follows that the first part of algorithm **SD4** takes $O(\sigma^s s^2 \log s) = O(c^s)$, for some constant c independent of s.

The second part of algorithm **SD4** puts together the (i, j, s) submatrices, generated by the first part of it, to form two edit matrices of steps **P** and **Q**. Then, the last row of matrix A can be easily computed.

Each entry in **P** and **Q** is a vector of length s. **P** is a matrix of initial and final column step vectors of s by s submatrices and **Q** is the corresponding matrix of row step vectors. Let $FETCH(H, V, x, y)$ be an operation that returns the pair of final step vectors corresponding to the quadruple (H, V, x, y) and let $SUM(vector)$ be a function that computes the sum of a vector's components. Without loss of generality, assume that s divides n and m.

Initially, $\mathbf{P}(i, 0)$ is set to $(1, 1, ..., 1)$, for $1 \leq i \leq \frac{m}{s}$, and $\mathbf{Q}(0, j)$ is set to $(0, 0, ..., 0)$, for $1 \leq i \leq \frac{n}{s}$. These assignments render the boundary conditions

$$A_{i,0} = i,\ 0 \leq i \leq m, \quad A_{0,j} = 0,\ 0 \leq j \leq n,$$

in terms of length s step vectors. For $1 \leq i \leq \frac{m}{s}$ and $1 \leq j \leq \frac{n}{s}$, $\mathbf{P}(i, j)$ and $\mathbf{Q}(i, j)$ are computed as follows:

$$[\mathbf{P}(i, j), \mathbf{Q}(i, j)] = FETCH(\mathbf{P}(i, j-1), \mathbf{Q}(i-1, j), p[(i-1)s+1, is], t[(j-1)s+1, js]).$$

Once that we know matrices **P** and **Q**, we can easily compute the last row of matrix A. Indeed, $A_{m,0} = m$ and $A_{m,js} = A_{m,(j-1)s} + SUM(\mathbf{Q}(\frac{m}{s}, j))$, for $1 \leq j \leq \frac{n}{s}$. Since $\mathbf{Q}(\frac{m}{s}, j) = (A_{m,(j-1)s+1} - A_{m,(j-1)s}, A_{m,(j-1)s+2} - A_{m,(j-1)s+1}, ..., A_{m,js} - A_{m,(j-1)s+s-1})$ and since we know $A_{m,(j-1)s}$, we can also compute $A_{m,(j-1)s+1}, A_{m,(j-1)s+2}, ..., A_{m,(j-1)s+s-1}$.

It can be shown [MP80] that the second part of **SD4** can be implemented in time $O(\frac{nm(s+\log n)}{s^2})$ assuming the logarithmic cost criterion for RAM. If we choose $s = \lfloor \log_c n \rfloor$, we obtain an $O(\frac{nm(s+\log n)}{s^2} + c^s) = O(\frac{nm}{\log n})$ time bound for **SD4**.

Obviously, when the size of the alphabet is constant and $\frac{m}{\log n} \leq k$, **SD4** is faster than any of the algorithms presented so far.

Algorithm **SD5**, devised by [UKK85], constructs a finite automaton $M_{pattern} = (Q, \Sigma, h, q_0, F)$, where Q, h, q_0 and F denote the set of states, the transition function, the initial state and the set of final states, respectively. Such a construction is based on the a priori knowledge of *pattern* and of the largest allowed edit distance k. Then, any arbitrary string *text* can be scanned by $M_{pattern}$ and a final state is reached if and only if *text* contains a substring at edit distance at most k from *pattern*. The scanning process takes $O(n)$ time.

Intuitively, each state of $M_{pattern}$ represents a possible column that may occur in matrix A. Since the text is not known when $M_{pattern}$ is constructed, the states of $M_{pattern}$ must account for all possible configurations that matrix A can assume for any input text.

Formally, each state of $M_{pattern}$ is an $(m+1)$-tuple of integers $S = (S_0, S_1, ..., S_m)$. A state S is final, i.e. $S \in F$, if and only if $S_m \leq k$. The initial state q_0 is $(0, 1, ..., m)$.

Starting from q_0, the set of states Q, the transition function h and the set of final states F are computed as follows.

Let **NEW** be a list of states computed so far and not processed yet. Initially, **NEW** $= \{q_0\}$.

A state $S = (S_0, S_1, ..., S_m)$ is extracted from **NEW** and, for each $a \in \Sigma$, a state $S' = (S'_0, S'_1, ..., S'_m)$ reachable from S by means of character a is obtained as follows: S'_0 is set to zero and S'_i is set to $min(S'_{i-1}+1, S_i+1, if\ p[i] = a\ then\ S_{i-1}\ else\ S_{i-1}+1)$. The transition $h(S, a)$ is added to function h. If S' is not already in Q, it is added to Q and **NEW**. Moreover, S' is added to F if and only if $S'_m \leq k$.

It can be shown [UKK85] that $M_{pattern}$ can be constructed in $O(m\sigma K)$ time, where $K = min(3^m, 2^k \sigma^k m^{k+1})$. Obviously, such an algorithm is practical only when m and k are very small compared to the size of the text.

The last algorithm that we consider deals with a variation of the k differences problem. Indeed, assume that differences of type (a), i.e. mismatches between characters, are not allowed. We are interested in finding all occurrences of *pattern* in *text* with at most k differences of type (b) and (c) only. An efficient algorithm for the solution of this problem was devised by [MY86] and it is based on algorithms for the computation of the longest common subsequence between two strings (see [HI75,HS77,AG85]). In what follows we give a simplified version of the algorithm [MY86].

First observe that a mismatch $p[i] \neq t[j]$ can be thought of as a deletion of a character ($p[i]$) from the pattern and an insertion of a character ($t[j]$) in the pattern. Assuming that each deletion and insertion costs one (hence a mismatch costs two), the problem we are dealing with is the following: Find all substrings $x_1, x_2, ..., x_s$ of *text* such that *pattern* can be transformed into x_i by means of a sequence of deletions and insertions whose cost is at most k.

A naive algorithm for the solution of this problem computes row by row, or column by column, a $(m+1) \times (n+1)$ cost matrix A' defined by the recurrence relation:

$$A'_{0,j} = j, \quad 0 \leq j \leq n, \quad A'_{i,0} = i, \quad 0 \leq i \leq m$$

$$A'_{i,j} = min(A'_{i-1,j} + 1, A'_{i,j-1} + 1, if \ \ p[i] = t[j] \ then \ A'_{i-1,j-1} \ else \ A'_{i-1,j-1} + 2).$$

Again, we can obtain an efficient algorithm for the computation of matrix A' by observing that the elements on any diagonal d of A' form a nondecreasing sequence of integers such that either $A'_{i+1,j+1} = A'_{i,j}$ or $A'_{i+1,j+1} = A'_{i,j} + 2$.

Algorithm **SD3** can compute matrix A' provided that the computation of $L_{d,e}$ is based on the knowledge of $L_{d+1,e-1}$, $L_{d-1,e-1}$ and $L_{d,e-2}$. Obviously, the time bound is still $O(nk + (n+m)\log m)$. We remark that the algorithm reported in [MY86] has the same time bound of $O(nk + (n+m)\log \tilde{m})$. We also notice that, when the size of the alphabet is constant, **SD4** can compute matrix A' in $O(\frac{nm}{\log n})$ time.

5. Concluding Remarks and Open Problems

We considered serial and parallel algorithms for approximate string matching. The serial algorithms **SM1** and **SD3** are the ones that perform best by achieving a time bound of $O(nk + (n+m)\log \tilde{m})$. This time bound is quite good when k is small. However, in practical applications, k may be large, i.e. $k = \Theta(m)$. In such a case, **SM1** and **SD3** perform in time $O(nm)$. Thus, if we compare those algorithms with classic ones as **SD1** and **SD4**, we see that no substantial progress has been made toward efficient algorithms for the case of k large.

The same critique applies to parallel algorithm **PM1** and to the parallel implementation of **SD3**. Thus, the problem of finding efficient parallel and serial approximate string matching algorithms for large k is still open.

As for preprocessing algorithms, we have that all the serial preprocessing algorithms are optimal within a constant factor. When the size of the input alphabet is small, this is no longer true for the parallel preprocessing algorithms. The source of inefficiency there is the construction of the suffix tree which requires $O(\log n)$ time, n processors and $n^{1+\epsilon}$, $0 < \epsilon \leq 1$ space. Since such a data structure has many important applications in combinatorial algorithms on strings [A84], it would be desirable to obtain a parallel algorithm for the construction of suffix trees that performs in $O(\log n)$ time with $\frac{n}{\log n}$ processors and $O(n)$ space when the size of the input alphabet is small.

ACKNOWLEDGEMENTS

We would like to thank David Eppstein and Baruch Schieber for their helpful comments, Alberto Apostolico, Gadi Landau, Greg Wasilkowsky and Henryk Wozniakowsky for reading an early version of this paper.

REFERENCES

[A84] A. Apostolico, The myriad virtues of subword trees, *Combinatorial Algorithms on Words (NATO ASI Series F, Vol 12)*, A. Apostolico and Z. Galil eds. , Springer-Verlag, Berlin, (1984), pp. 85-96.

[ADKF70] V.L. Arlazarov, E.A. Dinic, M.A. Kronrod and I.A. Faradzev, On economic construction of the transitive closure of a directed graph, *Soviet Math. Dokl. 11*, (1970), pp. 1209-1210.

[AG85] A. Apostolico and C. Guerra, The longest common subsequence problem revisited, *Algorithmica, to appear.*

[AILSV87] A. Apostolico, C. Iliopoulis, G.M. Landau, B. Schieber and U. Vishkin, Parallel construction of a suffix tree, with applications, *Algorithmica to appear.*

[AHU74] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Ma, (1974).

[BM77] R.S. Boyer and J.S. Moore, A fast string searching algorithm, *Comm. of the ACM 20* (1977), pp. 762-772.

[BLLP87] A.A. Bertossi, E. Lodi, F. Luccio and L. Pagli, String matching with context dependent errors, *Tech. Report*, Dipartimento di Informatica, Universita' di Pisa, Pisa, I56100 Italy, (1987)

[C86] R. Cole, Parallel merge sort, *Proc.27-th IEEE FOCS* (1986), pp. 511-516.

[CR84] M. Crochemore, Optimal factor transducers, in: *Combinatorial Algorithms on Words (NATO ASI Series F, Vol. 12)*, A. Apostolico and Z. Galil eds., Springer-Verlag, Berlin, (1984), pp. 31-44.

[FL80] M.J. Fischer and L. Ladner, Parallel prefix computation, *J. of the ACM 27* (1980), pp. 831-838.

[FP74] M.J. Fischer and M.S. Paterson, String matching and other products, in: *Complexity of Computation (SIAM-AMS Proceedings 7)*, R.M. Karp ed., American Mathematical Society, Prividence, RI, (1974), pp. 113-125.

[FW74] M.J. Fischer and R. Wagner, The string to string correction problem, *J. of the ACM 21* (1974), pp. 168-178.

[FW78] S. Fortune and J. Wyllie, Parallelism in Random Access Machines,*Proc. 10-th ACM STOC* (1978), pp. 114-118.

[GA85] Z. Galil, Open problems in stringology, *Combinatorial Algorithms on Words (NATO ASI Series F, Vol 12)*, A. Apostolico and Z. Galil eds. , Springer-Verlag, Berlin, (1984), pp. 1-8.

[GG86a] Z. Galil and R. Giancarlo, Improved string matching with k mismatches, *Sigact News 17* (1986), pp. 52-54.

[GG86b] Z. Galil and R. Giancarlo, Parallel string matching with k mismatches, *Theoretical Computer Science, to appear.*

[GP83] Z. Galil and W.J. Paul, An efficient general purpose parallel computer,*J. of the ACM 30* (1983), pp. 360-387.

[HI75] D.S. Hirshberg, A linear space algorithm for computing maximal common subsequences, *Communications of the ACM 18* (1975), pp. 341-343.

[HS77] J.W. Hunt and T.G. Szymanski, A fast algorithm for computing longest common subsequences, *Communication of the ACM 20* (1977), pp. 350-353.

[HM85] J. Hershberger and E. Mayr, Fast sequential algorithms to find shuffle minimizing and shortest paths in a shuffle exchange network, *Tech. Rep. TR STAN-CS-85-1050,* Dept. of Computer Science, Stanford University, Stanford, CA, (1985).

[HT84] D. Harel and R.E. Tarjan, Fast algorithms for finding nearest common ancestors, *SIAM J. on Computing 13* (1984), pp. 338-355.

[I84] A.G. Ivanov, Distinguishing an approximative word's inclusion on Turing machine in real time, *Izvestia Academee Nauk USSR Seria Matematicheskiya 48*, (1984), pp. 520-568.

[KMP76] D.E. Knuth, J.H. Morris and V.B. Pratt, Fast pattern matching in strings, *SIAM J. on Computing 6* (1977), pp. 189-195.

[KNU73] D.E. Knuth, *The Art of Computer Programming, VOL. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, (1973).

[L86] G.M. Landau, String matching in erroneous input, *Ph.D. Thesis*, Dept. of Computer Science, Tel-Aviv University (1986), Tel-Aviv, Israel.

[LE66] V.I. Levenshtein, Binary codes capable of correcting deletions, insertions and reversals, *Soviet Phys. Dokl. 6* (1966), pp. 707-710.

[LV85a] G.M. Landau and U. Vishkin, Efficient string matching in the presence of errors, *Proc. 26-th IEEE FOCS* (1985), pp. 126-136.

[LV85b] G.M. Landau and U. Vishkin, Efficient string matching with k mismatches, *Theoretical Computer Science, to appear.*

[LV86] G.M. Landau and U. Vishkin, Introducing efficient parallelism into approximate string matching and a new serial algorithm, *Proc. 18-th ACM STOC* (1986), pp. 220-230.

[MEH84] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, EATCS Monographs on TCS, Springer-Verlag, Berlin, (1984).

[Mc76] E.M. McCreight. A space economical suffix tree construction algorithm. *J. of ACM 23* (1976). pp. 262-272.

[MP80] W.J. Masek and M.S. Paterson, A faster algorithm computing string edit distances, *J. of Computer and System Sciences 20*, (1980), pp. 18-31.

[MY86] E.W. Myers, Incremental alignment algorithms and their applications, *Tech. Rep.* , Dept. of Computer Science, University of Arizona, Tucson, AZ, (1986).

[SV86] B. Schieber and U. Vishkin, On finding lowest common ancestors: simplification and parallelization, *Tech. Rep.*, Dept. of Computer Science, School of Mathematical sciences, Tel-Aviv University, Tel-Aviv, Israel, (1986).

[UKK83] E. Ukkonen, On approximate string matching, *Proc. Int. Conf. on Foundations of Computer Science*, Lecture Notes in Computer Science, Springer-Verlag, 1983, pp. 487-495.

[UKK85] E. Ukkonen, Finding approximate patterns in strings, *J. of Algorithms 6* (1985), pp. 132-137.

[W73] P. Wiener, Linear Pattern Matching algorithms, *Proc. 14-th Symposium on Switching and Automata Theory* (1973), pp. 1-11.