7
Fast Parallel and Serial Approximate String Matching

Consider the string matching problem, where differences between characters of the pattern and characters of the text are allowed. Each difference is due to either a mismatch between a character of the text and a character of the pattern, or a superfluous character in the text, or a superfluous character in the pattern. Given a text of length n, a pattern of length m and an integer k, serial and parallel algorithms for finding all occurrences of the pattern in the text with at most k differences are presented. For completeness we also describe an efficient algorithm for preprocessing a rooted tree, so that queries requesting the *lowest common ancestor* of every pair of vertices in the tree can be processed quickly.

Problems:

Input form. Two arrays: $A = a_1, ..., a_m$ - the pattern, $T = t_1, ..., t_n$ - the text and an integer k (≥ 1).

In the present chapter we will be interested in finding all occurrences of the pattern string in the text string with at most k differences.

Three types of differences are distinguished:

(a) A character of the pattern corresponds to a different character of the text - a *mismatch* between the two characters. (Item 2 in Example 1, below.)

(b) A character of the pattern corresponds to "no character" in the text. (Item 4).

(c) A character of the text corresponds to "no character" in the pattern. (Item 6).

Example 1. Let the text be $abcdefghi$, the pattern $bxdyegh$ and $k = 3$. Let us see whether there is an occurrence with $\leq k$ differences that ends at the eighth location of the text. For this the following correspondence between $bcdefgh$ and $bxdyegh$ is proposed. 1. b (of the text) corresponds to b (of the pattern). 2. c to x. 3. d to d. 4. Nothing to y. 5. e to e. 6. f to nothing. 7. g to g. 8. h to h. The correspondence can be illustrated as

$$\begin{array}{l} b\ x\ d\ y\ e\quad g\ h \\ b\ c\ d\quad e\ f\ \ g\ h \end{array}$$

In only three places the correspondence is between non-equal characters. This implies that there is an occurrence of the pattern that ends at the eighth location of the text with 3 differences as required.

So, the main problem we consider is:
String matching with k differences (the $k-differences$ problem, for short): Find all occurrences of the pattern in the text with at most k differences of type (a),(b) and (c).

The case $k = 0$ in the both problems is the string matching problem, which is discussed in Chapter 2. In this Chapter algorithms for the k differences problem are given. The "k mismatches problem" is simpler than the k differences problem (there, occurrences of the pattern in the text with at most k differences of type (a) only are allowed); however, there are no known algorithms for the k mismatches problem that are faster than the algorithms for the k differences problem, on the other hand the algorithms for the k differences problem solve the k mismatches problem as well.

The model of computation used in this chapter is the random-access-machine (RAM) for the serial algorithms, and the concurrent-read concurrent-write (CRCW) parallel random access machine (PRAM) for the parallel algorithms. A PRAM employs p synchronous processors, all having access to a common memory. A CRCW PRAM allows simultaneous access by more than one processor to the same memory location for read and write purposes. In case several processor seek to write simultaneously at the same memory location, one of them succeeds and it is not known in advance which one.

The k-differences problem is not only a basic theoretical problem. It also has a strong pragmatic flavor. In practice, one often needs to analyze situations where the data is not completely reliable. Specifically, consider a situation where the strings that are the input for the problem contain errors, as in reality, and one still needs to find all possible occurrences of the pattern in the text. The errors may include a character being replaced by another character, a character being omitted, or a superfluous character being inserted. Assuming some bound on the number of errors would clearly yield the k-differences problem.

Note that the measure of the quality of a match between the pattern and a substring of the text depends on the application. The k differences problem defines one possible measure. In many applications in molecular biology a penalty table is given. This table assigns a penalty value for the deletion and insertion of each letter of the alphabet, as well as a value for matching any pair of characters. In the simplest case the score of a match is simply the sum of the corresponding values in the penalty matrix. In some cases however gaps (successive insertions or deletions) get penalties

that are different from the sum of the penalties of each insertion (deletion).

The serial algorithm is given in Section 7.1. The parallel algorithm is described in Section 7.2. Both, the serial and parallel, algorithms use, as a procedure, an algorithm for the LCA problem. The problem and the algorithm are given in Section 7.3.

7.1 The serial algorithm

In this section, an efficient algorithm for the k-differences problem is presented. As a warm-up, the section starts with two serial $O(mn)$ time algorithms for this problem. The first one is a simple dynamic programming algorithm. The second algorithm follows the same dynamic programming computation in a slightly different way, that will help explain the efficient algorithm. Subsection 7.1.3 gives the efficient serial algorithm.

7.1.1 THE DYNAMIC PROGRAMMING ALGORITHM.

A matrix $D_{[0,\ldots,m;0,\ldots,n]}$ is constructed, where $D_{i,\ell}$ is the minimum number of differences between $a_1, \ldots, a_i$ and any contiguous substring of the text ending at t_ℓ.

If $D_{m,\ell} \leq k$ then there must be an occurrence of the pattern in the text with at most k differences that ends at t_ℓ.

Example 2.

Let the text be $GGGTCTA$, the pattern $GTTC$ and $k = 2$. The matrix $D_{[0,\ldots 4;0,\ldots,7]}$ (Table 7.1) is computed to check whether there are occurrences of the pattern in the text with $\leq k$ differences.

		G	G	G	T	C	T	A
	0	0	0	0	0	0	0	0
G	1	0	0	0	1	1	1	1
T	2	1	1	1	0	1	1	2
T	3	2	2	2	1	1	1	2
C	4	3	3	3	2	1	2	2

Table 7.1.

There are occurrences of the pattern in the text with $\leq k$ differences ending at t_4, t_5, t_6 and t_7.

The following algorithm computes the matrix $D_{[0,\ldots,m;0,\ldots,n]}$

Initialization
  for all $\ell, 0 \leq \ell \leq n,\ D_{0,\ell} := 0$
  for all $i, 1 \leq i \leq m,\ D_{i,0} := i$
for $i := 1$ to m do
  for $\ell := 1$ to n do

$D_{i,\ell} := \min$ ($D_{i-1,\ell} + 1$, $D_{i,\ell-1} + 1$, $D_{i-1,\ell-1}$ if $a_i = t_\ell$ or $D_{i-1,\ell-1} + 1$ otherwise)
($D_{i,\ell}$ is the minimum of three numbers. These three numbers are obtained from the predecessors of $D_{i,\ell}$ on its column, row and diagonal, respectively.)

Complexity. The algorithm clearly runs in $O(mn)$ time.

7.1.2 AN ALTERNATIVE DYNAMIC PROGRAMMING COMPUTATION

The algorithm computes the same information as in the matrix D of the dynamic programming algorithm, using the diagonals of the matrix. A *diagonal* d of the matrix consists of all $D_{i,\ell}$'s such that $\ell - i = d$.

For a number of differences e and a diagonal d, let $L_{d,e}$ denote the largest row i such that $D_{i,\ell} = e$ and $D_{i,\ell}$ is on diagonal d. The definition of $L_{d,e}$ clearly implies that e is the minimum number of differences between $a_1, ..., a_{L_{d,e}}$ and any substring of the text ending at $t_{L_{d,e}+d}$. It also implies that $a_{L_{d,e}+1} \neq t_{L_{d,e}+d+1}$. For the k-differences problem one needs only the values of $L_{d,e}$'s, where e satisfies $e \leq k$.

Example 2 (continued)

Let us demonstrate the $L_{d,e}$ values for diagonal 3 (Table 7.2).

		G	G	G	T	C	T	A
				0				
G					1			
T						1		
T							1	
C								2

Table 7.2.

$L_{3,0} = 0$, $L_{3,1} = 3$ and $L_{3,2} = 4$.

If one of the $L_{d,e}$'s equals m, for $e \leq k$, it means that there is an occurrence of the pattern in the text with at most k differences that ends at t_{d+m}.

The $L_{d,e}$'s are computed by induction on e. Given d and e it will be shown how to compute $L_{d,e}$ using its definition. Suppose that for all $x < e$ and all diagonals y, $L_{y,x}$ was already computed. Suppose $L_{d,e}$ should get the value i. That is, i is the largest row such that $D_{i,\ell} = e$, and $D_{i,\ell}$ is on the diagonal d. The algorithm of the previous subsection reveals that $D_{i,\ell}$ could have been assigned its value e using one (or more) of the following data:

(a) $D_{i-1,\ell-1}$ (which is the predecessor of $D_{i,\ell}$ on the diagonal d) is $e - 1$ and $a_i \neq t_\ell$. Or, $D_{i,\ell-1}$ (the predecessor of $D_{i,\ell}$ on row i which is also on

the diagonal "below" d) is $e-1$. Or, $D_{i-1,\ell}$ (the predecessor of $D_{i,\ell}$ on column ℓ which is also on the diagonal "above" d) is $e-1$.
(b) $D_{i-1,\ell-1}$ is also e and $a_i = t_\ell$.

This implies that one can start from $D_{i,\ell}$ and follow its predecessors on diagonal d by possibility (b) till the first time possibility (a) occurs.

The following algorithm "inverts" this description in order to compute the $L_{d,e}$'s. $L_{d,e-1}$, $L_{d-1,e-1}$, and $L_{d+1,e-1}$ are used to initialize the variable row, which is then increased by one at a time till it hits the correct value of $L_{d,e}$.

The following algorithm computes the $L'_{d,e}s$

Initialization
for all $d, 0 \leq d \leq n$, $L_{d,-1} := -1$
for all $d, -(k+1) \leq d \leq -1$, $L_{d,|d|-1} := |d|-1$, $L_{d,|d|-2} := |d|-2$
for all $e, -1 \leq e \leq k$, $L_{n+1,e} := -1$
2. for *e:=0* to k do
for *d:=-e* to n do
3. $row := \max [L_{d,e-1} + 1,\ L_{d-1,e-1},\ L_{d+1,e-1} + 1]$
$row := \min(row, m)$
4. while $row < m$ and $row+d < n$ and $a_{row+1} = t_{row+1+d}$ do
$row := row + 1$
5. $L_{d,e} := row$
6. *if* $L_{d,e} = m$ *then*
print *THERE IS AN OCCURRENCE ENDING AT t_{d+m}*

Remarks. (a) For every i, ℓ, $D_{i,\ell} - D_{i-1,\ell-1}$ is either zero or one.
(b) The values of the matrix D on diagonals d, such that $d > n-m+k+1$ or $d < -k$ will be useless for the solution of the k-differences problem.
(c) The Initialization step is given for completeness of this presentation. The values entered in this step are meaningless. It is easy to check that these values properly initialize the $L_{d,e}$ values on the boundary of the matrix.

Correctness of the algorithm

Claim. $L_{d,e}$ gets its correct value.

Proof. By induction on e. Let $e = 0$. Consider the computation of $L_{d,0}$, $(d \geq 0)$. Instruction 3 starts by initializing row to 0. Instructions 4 and 5 find that $a_1, ..., a_{L_{d,0}}$ is equal to $t_{d+1}, ..., t_{d+L_{d,0}}$, and $a_{L_{d,0}+1} \neq t_{d+L_{d,0}+1}$. Therefore, $L_{d,0}$ gets its correct value. To finish the base of the induction the reader can see that for $d < 0$, $L_{d,|d|-1}$ and $L_{d,|d|-2}$ get correct values in the Initialization.

Let $e = \ell$. Assume that all $L_{d,\ell-1}$ are correct. Consider the computation of $L_{d,e}$, $(d \geq -e)$. Following Instruction 3, row is the largest row on

diagonal d such that $D_{row,d+row}$ can get value e by possibility (a). Then Instruction 4 finds $L_{d,e}$. □

Complexity. The $L_{d,e}$'s for $n + k + 1$ diagonals are evaluated. For each diagonal the variable row can get at most m different values. Therefore, the computation takes $O(mn)$ time.

7.1.3 THE EFFICIENT ALGORITHM

The efficient algorithm has two steps:
Step I. Concatenate the text and the pattern to one string $t_1, ..., t_n$ $a_1, ..., a_m$. Compute the "suffix tree" of this string.
Step II. Find all occurrences of the pattern in the text with at most k differences.

Step I. The construction of the suffix tree is given in Section 4.

Upon construction of the suffix tree the following is required. For each node v of the tree, a contiguous substring $c_{i+1}, ..., c_{i+f}$ that defines it will be stored as follows: $START(v) := i$ and $LENGTH(v) := f$.
Complexity. The computation of the suffix tree is done in $O(n)$ time when the size of the alphabet is fixed. This is also the running time of Step I for fixed size alphabet. If the alphabet of the pattern contains x letters then it is easy to adapt the algorithm (and thus Step I) to run in time $O(n \log x)$. In both cases the space requirement of Step I is $O(n)$.

Step II. The matrix D and the $L_{d,e}$'s are exactly as in the alternative dynamic programming algorithm. This alternative algorithm is used with a very substantial change. The change is in Instruction 4, where instead of increasing variable row by one at a time until it reaches $L_{d,e}$, one finds $L_{d,e}$ in $O(1)$ time!

For a diagonal d, the situation following Instruction 3 is that $a_1, ..., a_{row}$ of the pattern is matched (with e differences) with some substring of the text that ends at t_{row+d}. One wants to find the largest q for which $a_{row+1}, ..., a_{row+q}$ equals $t_{row+d+1}, ..., t_{row+d+q}$. Let $LCA_{row,d}$ be the lowest common ancestor (in short LCA) of the leaves of the suffixes $t_{row+d+1}, ...$ and $a_{row+1}, ...$ in the suffix tree. The desired q is simply $LENGTH(LCA_{row,d})$. Thus, the problem of finding this q is reduced to finding $LCA_{row,d}$. An algorithm for the LCA problem is described in Section 7.3.
Example 2 (continued).

Let us explain how one computes $L_{3,1}$ (Table 7.3). For this, $L_{2,0}$, $L_{3,0}$ and $L_{4,0}$ are used. Specifically $L_{2,0} = 2$, $L_{3,0} = 0$ and $L_{4,0} = 0$.
The algorithm (Instruction 3) initializes row to $max(L_{2,0}, L_{3,0} + 1, L_{4,0} + 1)$ $= 2$. This is reflected in the box in which "Initially $row = 2$" is written. From the suffix tree one gets that $q = 1$. (Since $a_3 = t_6 = T$ and $a_4 \neq t_7$.) Therefore, $L_{3,1} := 3$.

		G	G	G	T	C	T	A
				0 ($L_{3,0}$)	0 ($L_{4,0}$)			
G								
T					0 ($L_{2,0}$)	(Initially $row = 2$)		
T							1 ($L_{3,1}$)	
C								

Table 7.3.

Complexity. In this section we are interested in the *static lowest common ancestors* problem; where the tree is static, but queries for lowest common ancestors of pair of vertices are given on line. That is, each query must be answered before the next one is known. The suffix tree has $O(n)$ nodes. In Section 7.3 an algorithm for the LCA problem is described. It computes LCA queries as follows. First it preprocesses the suffix tree in $O(n)$ time. Then, given an LCA query it responds in $O(1)$ time. For each of the $n+k+1$ diagonals, $k+1$ $L_{d,e}$'s are evaluated. Therefore, there are $O(nk)$ LCA Queries. It will take $O(nk)$ time to process them. This time dominates the running time of Step II.
Complexity of the serial algorithm. The total time for the serial algorithm is $O(nk)$ time for an alphabet whose size is fixed and $O(n(\log m + k))$ time for general input.

7.2 The parallel algorithm

The parallel algorithm described below runs in $O(\log n + k)$ time. At the end of this section, an explanation how to modify it to run in $O(\log m + k)$ time is given. The parallel algorithm has the same two steps as the efficient serial algorithm. Specifically:
Step I. Concatenate the text and the pattern to one string $(t_1, ..., t_n a_1, ..., a_m)$. Then, compute, in parallel, the suffix tree of this string (see Chapter 4).
Step II. Find all occurrences of the pattern in the text with at most k differences. This step is done in a similar way to Step II in the serial algorithm.

The matrix D and the $L_{d,e}$'s are exactly as in the serial algorithm. The parallel algorithm employs $n+k+1$ processors. Each processor is assigned to a diagonal d, $-k \leq d \leq n$. The parallel treatment of the diagonals is the source of parallelism in the algorithm.

For a diagonal d the situation following Instruction 3 is that $a_1, ..., a_{row}$ of the pattern is matched (with e differences) with some substring of the text that ends at t_{row+d}. One wants to find the largest q for which $a_{row+1}, ..., a_{row+q}$ equals $t_{row+d+1}, ..., t_{row+d+q}$. As in the serial algorithm one gets this q from the suffix tree. Let $LCA_{row,d}$ be the lowest common ancestor (in short LCA) of the leaves of the suffixes $t_{row+d+1}, ...$ and $a_{row+1}, ...$ in the suffix tree. The desired q is simply $LENGTH(LCA_{row,d})$. Thus, the problem of finding this q is reduced to finding $LCA_{row,d}$.

The parameter d is used and the *pardo* command for the purpose of guiding each processor to its instruction.

The parallel algorithm

1. *Initialization* (as in Subsection 7.1.2)
2. for $e := 0$ to k do
   for $d := -e$ to n pardo
   3. $row := \max[(L_{d,e-1}+1), (L_{d-1,e-1}), (L_{d+1,e-1}+1)]$
      $row := \min(row, m)$
   4. $L_{d,e} := row + LENGTH(LCA_{row,d})$
   5. if $L_{d,e} = m$ and $d + m \leq n$ then
      print *THERE IS AN OCCURRENCE ENDING AT t_{d+m}*

Complexity. In Chapter 4 it is shown how one may compute the suffix tree in $O(\log n)$ time using n processors. This suffix tree algorithm has the same time complexity for fixed alphabet and for general alphabet. This is also the running time of Step I. As in the serial case, one is interested in the *static lowest common ancestors* problem: where the tree is static, but queries for lowest common ancestors of pair of vertices are given on line. That is, each query must be answered before the next one is known. The suffix tree has $O(n)$ nodes. The parallel version of the serial algorithm, which is given in Section 7.3, for the LCA problem works as follows. It preprocesses the suffix tree in $O(\log n)$ time using $n/\log n$ processors. Then, an LCA query can be processed in $O(1)$ time using a single processor. Therefore, x parallel queries can be processed in $O(1)$ time using x processors. In the second step $n + k + 1$ processors (one per diagonal) are employed. Each processor computes at most $k + 1$ $L_{d,e}$'s. Computing each $L_{d,e}$ takes $O(1)$ time. Therefore, the second step takes $O(k)$ time using $n+k+1$ processors. Simulating the algorithm by n processors, instead of $n + k + 1$ still gives $O(k)$ time. The total time for the parallel algorithm is $O(\log n + k)$ time, using n processors.

Lastly, an explanation how one can modify the algorithm to get $O(\log m + k)$ time using $O(n)$ processors is given. Instead of the above problem $\lceil n/m \rceil$ smaller problems will be solved, in parallel. The first subproblem will be as follows. Find all occurrences of the pattern that end in locations $t_1, ..., t_m$ of the text. Subproblem $i, 1 \leq i \leq \lceil n/m \rceil$ will be:

Find all occurrences of the pattern that end in locations $t_{(i-1)m+1}, ..., t_{im}$ of the text. The input for the first subproblem will consist of the substring $t_1, ..., t_m$ of the text and the pattern. The input for subproblem i will consist of the substring $t_{(i-2)m-k+2}, ..., t_{im}$ of the text and the pattern. Clearly, the solution for all these subproblems give a solution for the above problem. Finally, note that one can apply the parallel algorithm of this section to solve each subproblem in $O(\log m + k)$ time using $O(m)$ processors, and all $\lceil n/m \rceil$ subproblems in $O(\log m + k)$ time using $O(n)$ processors. Simulating this algorithm by n processors still gives $O(\log m + k)$ time.

7.3 An algorithm for the LCA problem

The lowest-common-ancestor (LCA) problem

Suppose a rooted tree T is given for preprocessing. The preprocessing should enable to process quickly queries of the following form. Given two vertices u and v, find their lowest common ancestor in T.
The input to this problem is a rooted tree $T = (V, E)$, whose root is some vertex r. The *Euler tour technique* enables efficient parallel computation of several problems on trees. We summarize only those elements of the technique which are needed for presenting the serial lowest common ancestor algorithm below. Let H be a graph which is obtained from T as follows: For each edge $(v \rightarrow u)$ in T we add its anti-parallel edge $(u \rightarrow v)$. Since the in-degree and out-degree of each vertex in H are the same, H has an Euler path that starts and ends in the root r of T. This path can be computed, in linear time, into a vector of pointers D of size $2|E|$, where for each edge e of H, $D(e)$ gives the successor edge of e in the Euler path.

Let $n = 2|V| - 1$. We assume that we are given a sequence of n vertices $A = [a_1, \ldots, a_n]$, which is a slightly different representation of the Euler tour of T, and that we know for each vertex v its level, $LEVEL(v)$, in the tree.

The *range-minima problem* is defined as follows:
Given an array A of n real numbers, preprocess the array so that for any interval $[a_i, a_{i+1}, \ldots, a_j]$, the minimum over the interval can be retrieved in constant time.

Below we give a simple reduction from the LCA problem to a restricted-domain range-minima problem, which is an instance of the range-minima problem where *the difference between each two successive numbers for the range-minima problem is exactly one.* The reduction takes $O(n)$ time. An algorithm for the restricted-domain range-minima problem is given later, implying an algorithm for the LCA problem.

7.3.1 REDUCING THE LCA PROBLEM TO A RESTRICTED-DOMAIN RANGE-MINIMA PROBLEM

Let v be a vertex in T. Denote by $l(v)$ the index of the leftmost appearance of v in A and by $r(v)$ the index of its rightmost appearance. For each vertex v in T, it is easy to find $l(v)$ and $r(v)$ in $O(n)$ time using the following (trivial) observation:
$l(v)$ is where $a_{l(v)} = v$ and $LEVEL(a_{l(v)-1}) = LEVEL(v) - 1$.
$r(v)$ is where $a_{r(v)} = v$ and $LEVEL(a_{r(v)+1}) = LEVEL(v) - 1$.

The claims and corollaries below provide guidelines for the reduction.

Claim 1: Vertex u is an ancestor of vertex v iff $l(u) < l(v) < r(u)$.

Corollary 1: Given two vertices u and v, one can find in constant time whether u is an ancestor of v.

Vertices u and v are unrelated (namely, neither u is an ancestor of v nor v is an ancestor of u) iff either $r(u) < l(v)$ or $r(v) < l(u)$.

Claim 2 . Let u and v be two unrelated vertices. (By Corollary 2, we may assume without loss of generality that $r(u) < l(v)$.) Then, the LCA of u and v is the vertex whose level is minimal over the interval $[r(u), l(v)]$ in A.

The reduction. Let $LEVEL(A) = [LEVEL(a_1), LEVEL(a_2), \ldots, LEVEL(a_n)]$. Claim 2 shows that after performing the range-minima preprocessing algorithm with respect to $LEVEL(A)$, a query of the form $LCA(u, v)$ becomes a range minimum query. Observe that the difference between the level of each pair of successive vertices in the Euler tour (and thus each pair of successive entries in $LEVEL(A)$) is exactly one and therefore the reduction is to the restricted-domain range-minima problem as required.

Remark. The observation that the problem of preprocessing an array so that each range-minimum query can be answered in constant time is equivalent to the LCA problem was known. This observation has led to a linear time algorithm for the former problem using an algorithm for the latter. This does not look very helpful: we know to solve the range-minima problem based on the LCA problem, and conversely, we know to solve the LCA problem based on the range-minima problem. Nevertheless, using the restricted domain properties of our range-minima problem we show that this cyclic relationship between the two problems can be broken and thereby, lead to a new algorithm.

7.3.2 A SIMPLE SEQUENTIAL LCA ALGORITHM

In this subsection we outline a sequential variant of the restricted-domain range-minima problem where k, the difference between adjacent elements, is one. Together with the reduction of Section 7.3.1, this gives a sequential algorithm for the LCA problem.

We first describe two preprocessing procedures for the range-minima

problem: (i) Procedure I takes $O(n \log n)$ time, for an input array of length n. No assumptions are needed regarding the difference between adjacent elements. (ii) Procedure II takes exponential time. Following each of these preprocessing procedures, query retrieval takes constant-time. Second, the sequential linear-time range-minima preprocessing algorithm is described. Finally, we show how to retrieve a range-minimum query in constant time.
Procedure I. Build a complete binary tree whose leaves are the elements of the input array A. Compute (and keep) for each internal node all prefix minima and all suffix minima with respect to its leaves.

Procedure I clearly runs in $O(n \log n)$ time. Given any range $[i, j]$, the range-minimum query with respect to $[i, j]$ can be processed in constant time, as follows. (1) Find the lowest node u of the binary tree such that the range $[i, j]$ falls within its leaves. This range is the union of a suffix of the left child of u and a prefix of the right child of u. The minima over these suffix and prefix was computed by Procedure I. (2) The answer to the query is the minimum among these two minima.
Procedure II. We use the assumption that the difference between any two adjacent elements of the input array A is exactly one. A table is built as follows. We assume without loss of generality that the value of the first element of A is zero (since, otherwise, we can subtract from every element in A the value of the first element without affecting the answers to range-minima queries). Then, the number of different possible input arrays A is 2^{n-1}. The table will have a subtable for each of these 2^{n-1} possible arrays. For each possible array, the subtable will store the answer to each of the $n(n-1)/2$ possible range queries. The time to build the table is $O(2^n n^2)$ and $O(2^n n^2)$ space is needed.

The linear-time range-minima preprocessing algorithm follows.

- For each of the subsets $a_{i \log n+1}, \ldots, a_{(i+1) \log n}$ for $0 \leq i \leq n/\log n - 1$ find its minimum and apply Procedure I to an array of these $n/\log n$ minima.
- Separately for each of the subsets $a_{i \log n+1}, \ldots, a_{(i+1) \log n}$ for $0 \leq i \leq n/\log n - 1$ do the following. Partition such subset to smaller subsets of size $\log \log n$ each, and find the minimum in each smaller subset; apply Procedure I to these $\log n/\log \log n$ minima.
- Run Procedure II to build the table required for an (any) array of size $\log \log n$. For each of the subsets $a_{i \log \log n+1}, \ldots, a_{(i+1) \log \log n}$ for $0 \leq i \leq n/\log \log n - 1$ identify its subtable.

The time (and space) for each step of the preprocessing algorithm is $O(n)$.

Consider a query requesting the minimum over a range $[i, j]$. We show how to process it in constant time. The range $[i, j]$ can easily be presented as the union of the following (at most) five ranges: $[i, x_1], [x_1 + 1, y_1], [y_1 + 1, y_2], [y_2+1, x_2]$ and $[x_2+1, j]$; where: (1) $[i, x_1]$ (and $[x_2+1, j]$) falls within

a single subset of size $\log\log n$ – its minimum is available in its subtable, (2) $[x_1+1, y_1]$ (and $[y_2+1, x_2]$) is the union of subsets of size $\log\log n$ and falls within a single subset of size $\log n$ – its minimum is available from the application of Procedure I to the subset of size $\log n$, and (3) $[y_1+1, y_2]$ is the union of subsets of size $\log n$ – its minimum is available from the first application of Procedure I. So, the minimum over range $[i, j]$ is simply the minimum of these five minima.

7.4 Bibliographic notes

Levenshtein [1966] was the first to define the three types of differences. The random-access-machine (RAM)is described in Aho et al. [1974]. Several books, AKL [1989], Gibbons and Rytter [1988], JáJá [1992], and Reif [1992], and a few review papers, Eppstein and Galil [1988], Karp and Ramachandran [1990], Kruskal et al. [1990], Vishkin [1991], can be used as references for PRAM algorithms. A discussion on gaps is given in Galil and Giancarlo [1989] and Myers and Miller [1988].

The reader is referred to Sankoff and Kruskal [1983], a book which is essentially devoted to various instances of the k-differences problem. The book gives a comprehensive review of applications of the problem in a variety of fields, including: computer science, molecular biology and speech recognition. Quite a few problems in Molecular Biology are similar to the k difference problem. Definitions of the problems and algorithms that solve these problems can be found, for example, in Doolittle [1990] and Waterman [1989].

The dynamic programming algorithm (Section 7.1.1) was given independently by 9 different papers; a list of these papers can be found in Sankoff and Kruskal [1983]. The algorithm given in Section 7.1.2 was presented by Ukkonen [1983]. The algorithms given in Sections 7.1.3 and 7.2 were presented in Landau and Vishkin [1989].

The serial algorithm of Harel and Tarjan [1984] was the first to solve the LCA problem. It preprocesses the tree in linear time and then responses to each query in $O(1)$ time. The algorithms of Schieber and Vishkin [1988] and Berkman and Vishkin [1989] compute it in parallel; these algorithms can be used in the serial case, as well, and are simpler than the one of Harel and Tarjan [1984]. The serial algorithm in Section 7.3 was presented in Berkman and Vishkin [1989] where one can find the parallel version of it. The remark in Section 7.3 was observed in Gabow et al. [1984]. A procedure similar to Procedure I in Section 7.3 was used in Alon and Schieber [1987]. For more on the Euler tour technique see Tarjan and Vishkin [1985] and Vishkin [1985].

Other algorithms for the k mismatches problem were given in Galil and Giancarlo [1986], Galil and Giancarlo [1987] and Landau and Vishkin [1986], and for the k-differences problem in Galil and Park [1990], Landau,

Myers and Schmidt [1996], Landau and Vishkin [1988], Ukkonen [1985] and Wu and Manber [1992]. In Galil and Giancarlo [1988] a survey was given. Algorithms for approximate multi-dimensional array matching are given in Amir and Landau [1991].

7.5 Bibliography

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

Akl, S.G. [1989], *The Design and Analysis of Parallel Algorithms*, Prentice Hall, Engelwood Cliffs, New Jersey.

Amir, A., and G.M. Landau [1991], "Fast parallel and serial multi dimensional approximate array matching", *Theoretical Computer Science*, **81**, 97–115.

Alon, N., and B. Schieber[1987], "Optimal preprocessing for answering on-line product queries," approximate array matching," TR 71/87, The Moise and Frida Eskenasy Institute of Computer Science, Tel Aviv University.

Berkman, O. and U. Vishkin [1989], "Recursive star-tree parallel data-structure," *SIAM J. Computing*, **22**, 221–242.

Eppstein, D. and Z. Galil [1988], "Parallel algorithmic techniques for combinatorial computation," *Ann. Rev. Comput. Sci.*, **3**, 233–283.

Doolittle, R. F.,, (editor) [1990], *Methods in Enzymology*, **183:** Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences.

Gabow, H. N., J. L. Bentley and R. E. Tarjan [1984], "Scaling and related techniques for geometry problems," *Proc. 16th ACM Symposium on Theory of Computing*, pp. 135-143.

Galil, Z. and R. Giancarlo [1986], "Improved string matching with k mismatches," *SIGACT News*, **17**, 52–54.

Galil, Z. and R. Giancarlo [1987], "Parallel string matching with k mismatches," *Theoretical Computer Science*, **51**, 341–348.

Galil, Z. and R. Giancarlo [1988], "Data Structures and algorithms for approximate string matching," *J. Complexity*, **4**, 33–72.

Galil, Z. and R. Giancarlo [1989], "Speeding up dynamic programming with applications to molecular biology," *Theoretical Computer Science*, **64**, 107–118.

Galil, Z. and Q. Park [1990], "An improved algorithm for approximate string matching," *SIAM J. Computing*, **19**, 989–999.

Gibbons, A. and W. Rytter [1988], *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge.

Harel, D. and R.E. Tarjan [1984], "Fast algorithms for finding nearest common ancestors," *SIAM J. Computing*, **13**, 338–355.

Karp, R.M. and V. Ramachandran [1990], "A survey of parallel algo-

rithms for shared-memory machines," *Handbook of Theoretical Computer Science: Volume A, Algorithms and Complexity* (Editor J. van Leeuwen), MIT Press/Elsevier, 869–942.

KNUTH, D.E., J.H. MORRIS AND V.R. PRATT [1977] "Fast pattern matching in strings," *SIAM J. Computing,* **6**, 323–350.

KRUSKAL, C.P., L. RUDOLPH, AND M. SNIR [1990], "A complexity theory of efficient parallel algorithms," *Theoretical Computer Science*, **71**, 95–132.

JÁJÁ, J. [1992], *Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA.

LEVENSHTEIN, V. I. [1966], "Binary Codes Capable of Correcting, Deletions, Insertions and Reversals," *Soviet Phys. Dokl* **10**, *SIAM J. Computing,* to appear. 707–710.

LANDAU, G. M., E. W. MYERS, AND J. P. SCHMIDT [1996], "Incremental String Comparison." *SIAM J. Computing,* to appear.

LANDAU, G.M. AND U. VISHKIN [1986], "Efficient string matching with k mismatches," *Theoretical Computer Science,* **43**, 239–249.

LANDAU, G.M. AND U. VISHKIN [1988], "Fast string matching with k differences," *JCSS*, **37**, 63–78.

LANDAU, G.M. AND U. VISHKIN [1989], "Fast parallel and serial approximate string matching," *Journal of Algorithms*, **10**, 157–169.

MYERS, E. W., AND W. MILLER [1988], "Sequence Comparison with Concave Weighting Functions," *Bulletin of Mathematical Biology*, **50**, 97–120.

REIF, J.H. (editor) [1992], *Synthesis of Parallel Algorithms*, Morgan Kaufmann, San Mateo, California.

SANKOFF, D. AND J.B. KRUSKAL (editors) [1983], *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison,* Addison-Wesley, Reading, MA.

SCHIEBER, B. AND U. VISHKIN [1988], "On finding lowest common ancestors: simplification and parallelization," *SIAM J. Computing,* **17**, 1253–1262.

TARJAN, R. E. AND U. VISHKIN [1985], "An efficient parallel biconnectivity algorithm," *SIAM J. Computing,* **14**, 862–874.

UKKONEN, E. [1983], "On approximate string matching," *Proc. Int. Conf. Found. Comp. Theor.,* Lecture Notes in Computer Science 158, Springer-Verlag, pp. 487–495.

UKKONEN, E. [1985], "Finding approximate pattern in strings," *J. of Algorithms,* **6**, 132–137.

VISHKIN [1985], "On efficient parallel strong orientation," *Information Processing Letters,* **20**, 235–240.

VISHKIN, U. [1991], "Structural parallel algorithmics," *Proc. of the 18th Int. Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science 510, Springer-Verlag, pp. 363-380.

WATERMAN, M. S. (editor) [1989], *Mathematical Methods for DNA Sequences,* CRC Press.

WU, S. AND U. MANBER [1992], "Fast Text Searching Allowing Errors," *Comm. of the ACM*, **35**, 83–91.