Chapter 12

Approximate pattern matching

In practical pattern-matching applications, the exact matching is not always pertinent. It is often more important to find objects that match a given pattern in a reasonably approximate way. In this chapter, approximation is measured mainly by the so-called edit distance: the minimal number of local edit operations needed to transform one object into another. The analogue for DNA sequences is called the *alignment problem* (see Figure 12.1). Algorithms are mainly based on the algorithmic method called *dynamic programming*. We also present problems strongly related to the notion of edit distance, namely, the computation of longest common subsequences, string matching allowing errors, and string matching with don't care symbols.

12.1 Edit distance

An immediate question arising in applications is how to test the equality of two strings allowing some errors. The errors correspond to differences between

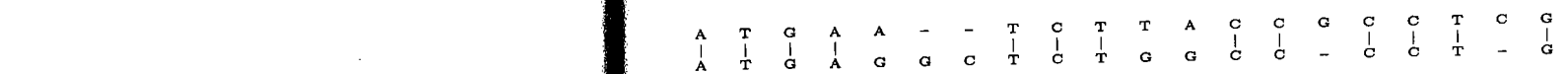

Figure 12.1: Alignment of two DNA sequences showing the operations of changes, insertions ("–" in top line), and deletions ("–" in bottom line).

the two words. We consider in this section three types of differences between two strings x and y:

change: symbols at corresponding positions are distinct,

insertion: a symbol of y is missing in x at a corresponding position,

deletion: a symbol of x is missing in y at a corresponding position.

We require the minimum number of differences between x and y. We translate this as the smallest possible number of operations (change, deletion, insertion) to transform x into y. This is called the *edit distance* between x and y, and denoted by $edit(x, y)$. It is clear that it is a distance between words, in the mathematical sense. This means that the following properties are satisfied:

- $edit(x, y) \geq 0$,
- $edit(x, y) = 0$ iff $x = y$,
- $edit(x, y) = edit(y, x)$ (symmetry),
- $edit(x, y) \leq edit(x, z) + edit(z, y)$ (triangle inequality).

The symmetry of *edit* comes from the duality between deletions and insertions: a deletion of the letter a of x in order to get y corresponds to an insertion of a into y to get x.

Example. The text $x = wojtk$ can be transformed into $y = wjeek$ using one deletion, one change and one insertion. This shows that $edit(wojtk, wjeek) \leq 3$, because it uses three operations. In fact, this is the minimum number of edit operations to transform $wojtk$ into $wjeek$.

From now on, we consider that words x and y are fixed. The length of x is m, and the length of y is n, and we assume that $n \geq m$. We define the table $EDIT$ by:

$$EDIT[i, j] = edit(x[1..i], y[1..j])$$

for $0 < i \leq m$ and $0 < j \leq n$. The boundary values are defined as follows (for $0 \leq i \leq m$, $0 \leq j \leq n$):

$$EDIT[0, j] = j, \ EDIT[i, 0] = i.$$

There is a simple formula for computing other elements.

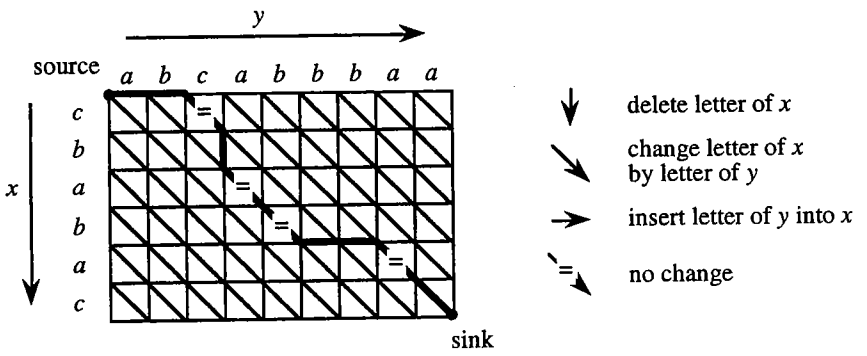

Figure 12.2: The path corresponds to the sequence of edit operations: $insert(a)$, $insert(b)$, $delete(b)$, $insert(b)$, $insert(b)$, $change(c, a)$.

(*) *Dynamic Programming Recurrence:*

$$EDIT[i, j] = \min(EDIT[i-1, j] + 1, EDIT[i, j-1] + 1, \\ EDIT[i-1, j-1] + \partial(x[i], y[j]),$$

where $\partial(a, b) = 0$ if $a = b$, and $\partial(a, b) = 1$ otherwise.

The formula reflects the three operations, deletion, insertion, and change, in that order.

There is a graph theoretical formulation of the editing problem. We consider the *grid graph* also called the *alignment graph*, denoted by G. It is composed of nodes (i, j) (for $0 \leq i \leq m$, $0 \leq j \leq n$).

The node $(i-1, j-1)$ is connected to the three nodes $(i-1, j), (i, j-1), (i, j)$ when they are defined (i.e., when $i \leq m$, or $j \leq n$).

Each edge of the grid graph has a weight corresponding to the recurrence $(*)$. The edges from $(i-1, j-1)$ to $(i-1, j)$ and $(i, j-1)$ have weight 1, as they correspond to the insertion and deletion of a single symbol, respectively. The edge from $(i-1, j-1)$ to (i, j) has weight $\partial(x[i], y[j])$.

Figure 12.2 shows an example of grid graph for words *cbabac* and *abcabbbaa*.

The edit distance between words x and y equals the length of a least weighted path in this graph from the source $(0, 0)$, left upper corner, to the sink (m, n), right bottom corner.

```
function edit(x, y) { computation of edit distance }
{ |x| = m, |y| = n, EDIT is a matrix of integers }
    for i := 0 to m do EDIT[i, 0] := i;
    for j := 1 to n do EDIT[0, j] := j;
    for i := 1 to m do
        for j := 1 to n do
            EDIT[i, j] = min(EDIT[i − 1, j] + 1, EDIT[i, j − 1] + 1,
                             EDIT[i − 1, j − 1] + ∂(x[i], y[j]);
    return EDIT[m, n];
```

The algorithm above computes the edit distance of strings x and y. It stores and computes all values of the matrix $EDIT$, although only one entry, $EDIT[m, n]$, is required. This serves to save time, and this feature is called the *dynamic programming method.* Another possible algorithm to compute $edit(x, y)$ could be to use the classical Dijkstra's algorithm for shortest paths in the grid graph.

Theorem 12.1 *Edit distance of two words of lengths m and n can be computed in $O(mn)$ time using $O(\min\{m, n\})$ additional memory.*

Proof. The time complexity of the algorithm above is obvious. The space complexity results from the fact that we do not have to store the entire table. The current and the previous columns (or lines) are sufficient for carrying out computations. □

We can assign specific costs to edit operations, depending on the type of operations and on the type of symbols involved. Such a generalized edit distance can be computed using a formula analogous to equation (*).

As noted in the proof of the previous theorem, the whole matrix $EDIT$ does not need to be stored (only two columns are sufficient at a given step) in order to compute only $edit(x, y)$. However, we can keep it in memory if we want to compute a shortest sequence of edit operations transforming x into y. This is essentially done by tracing back in the matrix how each value has been obtained.

12.2 Longest common subsequence problem

In this section, we consider a problem that illustrates a particular case of the edit distance problem of the previous section. This is the example of computing

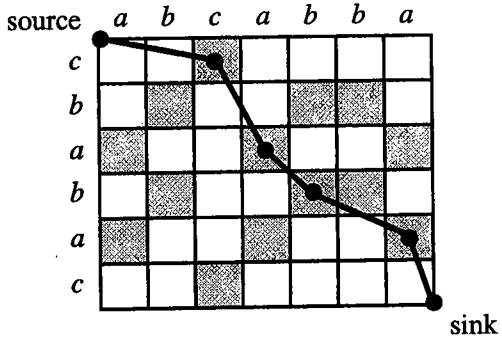

Figure 12.3: The size of the longest subsequences is the maximal number of shaded boxes (associated with matches) on a monotonically decreasing path from source to sink. Compare with Figure 12.4.

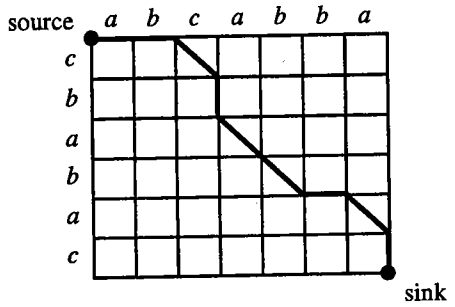

Figure 12.4: Assigning cost 2 diagonal edges, the length of the path is $m + n$. Diagonal edges correspond to equal letters, other edges to edit operations.

longest common subsequences (see Figure 12.3). We denote by $lcs(x, y)$ the maximal length of subsequences common to x and y. For fixed x and y, we denote by $LCS[i, j]$ the length of a longest common subsequence of $x[1..i]$ and $y[1..j]$.

There is a strong relationship between the longest common subsequence problem and a particular edit distance computation. Let $edit_{di}(x, y)$ be the minimal number of operations *delete* and *insert* necessary to transform x into y. This corresponds to a restricted edit distance where changes are not allowed. A change in the edit distance can be replaced by a pair of operations, deletion and insertion. The following lemma shows that the computation of $lcs(x, y)$ is equivalent to the evaluation of $edit_{di}(x, y)$. The statement is not true in general for all edit operations, or for edit operations having distinct costs. However, the restricted edit distance $edit_{di}$ remains to give weight 2 to changes, and weight 1 to both deletions and insertions. Recall that x and y have respective length m and n, and let $EDIT_{di}[i, j]$ be $edit_{di}(x[1..i], [1..j])$.

Lemma 12.1 *We have* $2 \cdot lcs(x, y) = m + n - edit_{di}(x, y)$, *and* $2 \cdot LCS[i, j] = i + j - EDlT_{di}[i, j]$, *for* $0 \leq i \leq m$ *and* $0 \leq j \leq n$.

Proof. The equation can be easily proved by induction on i and j. This is also apparent on the graphical representation in Figure 12.4. Consider a path from the source to the sink in which diagonal edges are only allowed when the corresponding symbols are equal. The number of diagonal edges in the path is the length of a subsequence common to x and y. Horizontal and vertical edges correspond to a sequence of edit operations (delete, insert) to transform x into y. If we assign cost 2 to the diagonal edges, the length of the path is exactly the sum of the lengths of words, $m + n$. □

As a consequence of Lemma 12.1 computing $lcs(x, y)$ takes the same amount of time as computing the edit distance of the strings. A longest common subsequence can even be found with linear extra space (see bibliographic references).

Theorem 12.2 *A longest common subsequence of two strings of lengths* m, n *can be computed in* $O(mn)$ *time using* $O(mn)$ *additional memory.*

Proof. Assume that the table $EDIT_{di}$ is computed for zero-one costs of edges. Table LCS can be precomputed from Lemma 12.1. After that, a longest common subsequence can be constructed from the table LCS. □

Let r be the number of shaded boxes in Figure 12.3. More formally, it is the number of pairs (i, j) such that $x[i] = y[j]$. If r is small (which happens often in practice) compared to mn, then there is an algorithm to compute longest common subsequence that is faster than the previous algorithm.

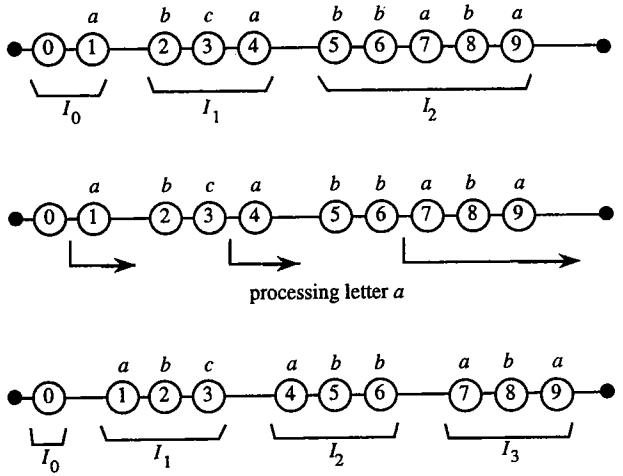

Figure 12.5: Hunt-Szymanski strategy on the word $y = abcabbaba$. Partitions of positions just before processing letter a of $x = cba$ (top), and just after (bottom).

The algorithm is given below. It processes the word x sequentially from left to right. Consider the situation in which $x[1..i-1]$ has just been processed. The algorithm maintains a partition of positions on the word y into intervals $I_0, I_1, \ldots, I_k, \ldots$ that are defined by

$$I_k = \{j : lcs(x[1..i-1], y[1..j]) = k\}.$$

In other words, positions in a class I_k correspond to prefixes of y having the same maximal length of common subsequence with $x[1..i-1]$. Consider, for instance, $y = abcabbaba$ and $x = cb\ldots$. Figure 12.5 (top) shows the partition $\{I_0, I_1, I_2\}$ of positions on y. For the next symbol of x, letter a, the figure (bottom) displays the modified partition $\{I_0, I_1, I_2, I_3\}$. The computation reduces to shifting to the right, like bowls on a wire, positions corresponding to occurrences of a inside y.

The algorithm *lcs* below implements this strategy. It makes use of operations on intervals of positions: *CLASS*, *SPLIT*, and *UNION*. They are defined as follows. For a position p on y, $CLASS(p)$ is the index k of the interval I_k to which it belongs. When p is in the interval $[f, f+1, \ldots, g]$, and $p \neq f$, then $SPLIT(I_k, p)$ is the pair of intervals $([f, f+1, \ldots, p-1], [p, p+1, \ldots, g])$. Finally, *UNION* is the union of two intervals; in the algorithm, only unions of disjoint consecutive intervals are performed.

Theorem 12.3 *Algorithm lcs computes the length of a longest common subsequence of words of length m and n $(m < n)$ in $O((n + r)\log n)$ time, where $r = |\{(i,j) : x[i] = y[j]\}|$.*

Proof. The correctness of the algorithm is left as an exercise. The time complexity of the algorithm strongly relies on an efficient implementation of intervals I_k's. Using an implementation with B-trees, it can be shown that each operation *CLASS*, *SPLIT*, and *UNION* takes $O(\log n)$ time. Preprocessing lists of occurrences of letters of the word y takes $O(n \log n)$ time. The rest of the algorithm takes $O(r \log n)$ time. □

```
function lcs(x, y): integer; { Hunt-Szymanski algorithm }
{ m = |x| and n = |y| }
   I_0 := {0, 1, ..., n}; for k := 1 to n do I_k = ∅;
   for i := 1 to m do
       for each p position of x[i] inside y
                in decreasing order do begin
           k := CLASS(p);
           if k = CLASS(p − 1) then begin
               (I_k, X) := SPLIT(I_k, p);
               I_{k+1} := UNION(X, I_{k+1});
           end;
       end;
   return CLASS(n);
```

According to Theorem 12.3, if r is small, the computation of *lcs* by the last algorithm takes $O(n \log n)$ time, which is faster than with the dynamic programming algorithm. But, r can be of order mn (in the trivial case where $x = a^m, y = a^n$, for example), and then the time complexity becomes $O(mn \log n)$, which is larger than the running time of the dynamic programming algorithm.

The problem of computing *lcs* can be reduced to the computation of the longest increasing subsequence of a given string of elements belonging to a linearly ordered set. Let us write the coordinates of shaded boxes (as in Figure 12.3) from the first to the last row, and from left to right within rows. By doing so, we get a string w. No matrix table is needed to build w, the words x and y themselves suffice. For example, for the words of Figure 12.4 we get the sequence

$$((1,3), (2,2), (2,5), (2,6), (3,1), (3,4), (3,7),$$
$$(4,2), (4,5), (4,6), (5,1), (5,4), (5,7), (6,3)).$$

Define the following linear order on pairs of positions:

$$(i,j) \ll (k,l) \text{ iff } ((i = k) \text{ and } (j > l)) \text{ or } ((i < k) \text{ and } (j < l)).$$

Then a longest increasing (according to $\ll$) subsequence of the string w gives a longest common subsequence of the words x and y. There is an elegant algorithm to compute such increasing subsequences running is $O(r \log r)$ time. It presents an alternative to the above algorithm.

12.3 String matching with errors

String matching with errors differs only slightly from the edit distance problem. Here, we are given pattern *pat* and text *text* and we want to compute $\min(edit(pat, y) : y \in \mathcal{F}(text))$. Simultaneously, we want to find a factor y of *text* realizing the minimum and the position of one of its occurrences on *text*. We consider the table SE (that stands for String matching with Errors), of the same type as table *EDIT*:

$$SE[i,j] = \min(edit(pat[1..i], y) : y \in \mathcal{F}(text[1..j])).$$

The computation of table SE can be executed with dynamic programming. It is very similar to the computation of table *EDIT*.

Theorem 12.4 *The problem of string matching with errors can be solved in $O(mn)$ time.*

Proof. Surprisingly the algorithm is almost the same as that for computing edit distances. The only difference is that we initialize $SE[0,j]$ to 0, instead of j for *EDIT*. This is because the empty prefix of *pat* matches an empty factor of *text* (no error). The formula $(*)$ also works for SE. Then, $SE[m,n]$ is the distance between *pat* and one of its best matches y in the text. To find an occurrence of y and its position in *text*, we can use the same graph-theoretic approach as we used for the computation of longest common subsequences. It is recovered by a trace back inside the computed table SE from an extremal value. This completes the proof. □

One of the most interesting problems related to string matching with errors concerns the case in which the allowed number of errors is bound by a constant k. The number k is usually understood as a small fixed constant. We show that this problem can be solved in $O(n)$ time, or more exactly in $O(kn)$ time, if k is not fixed. For a fixed value of the parameter k, this gives an algorithm

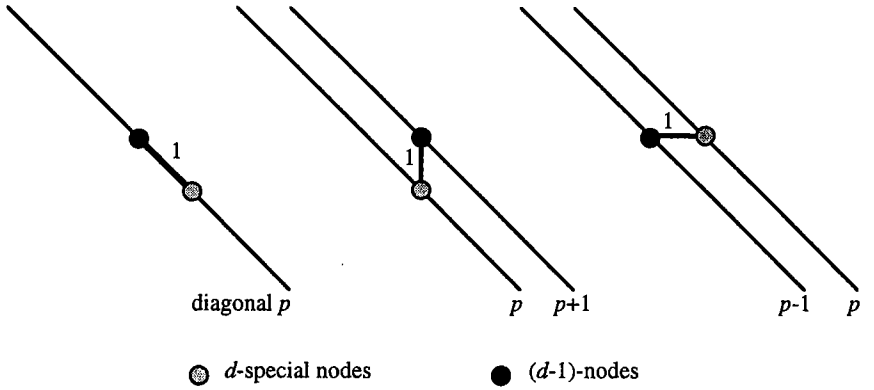

Figure 12.6: The computation of d-special nodes: nodes reachable by one edge of weight 1 from a $(d-1)$-node.

having an optimal asymptotic time complexity. Recall that the string edit table SE is computed according to the recurrence:

$$(*)\ \ SE[i,j] = \min(SE[i-1,j]+1, SE[i,j-1]+1, SE[i-1,j-1]+\partial(x[i],y[j])),$$

for words x and y.

Suppose that we have a fixed bound k on the number of errors. We require that the complexity is $O(kn)$. Only $O(kn)$ entries of the table must be considered. The basic algorithmic trick is to consider only so-called *d-nodes*, which are entries of the table SE satisfying special conditions. The d-nodes are defined in such a way that altogether we have$O(kn)$ such nodes.

We consider diagonals of the table SE. Each diagonal is oriented top-down, left-to-right. We define a d-node as the last pair (i,j) on a given diagonal satisfying $SE[i,j] = d$. Note that it is possible that a diagonal has no d-node. The approximate string-matching problem reduces to the computation of d-nodes. And it is clear that there is an occurrence of the pattern with d errors ending at position j on *text* iff (m,j) is a d-node.

Computation of d-nodes is executed for $d = 0, 1, \ldots, k$ in this order. Computation of 0-nodes is equivalent to string matching without errors. Assume that we have already computed the $(d-1)$-nodes. To compute d-nodes we need two auxiliary concepts: d-special nodes, and maximal subpaths of zero-weight on a given diagonal. For a node (i,j), define the node $NEXT(i,j) = (i+t, j+t)$ as the lowest node on the same diagonal as (i,j), reachable from (i,j) by a subpath of zero weight. The subpath can be of zero length, and in this case $NEXT(i,j) = (i,j)$. A *d-special node* is a node reachable from a $(d-1)$-node

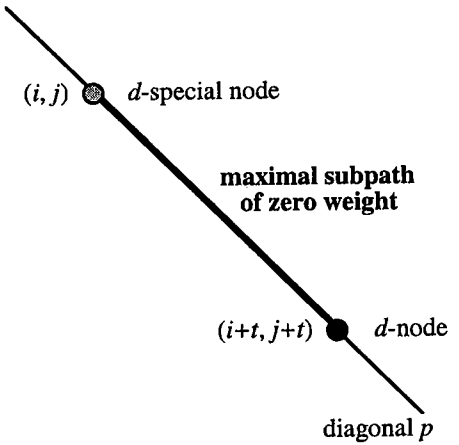

Figure 12.7: The computation of d-nodes from d-special nodes.

by one edge. Once d-special nodes are computed, the d-nodes can be easily found, as suggested in Figure 12.7. The structure of the algorithm is given below.

Theorem 12.5 *Assume that the alphabet is of a constant size. Approximate string matching with k errors can be achieved in $O(kn)$ time.*

Proof. It is sufficient to prove that we can run $O(kn)$ calls of the function $NEXT$ in $O(kn)$ time. The equation $NEXT(i,j) = (i+t, j+t)$ means that t is the size of the longest common prefix of $pat[i..m]$ and $text[j..n]$. Assume that we have computed the common suffix tree for words *pat* and *text*. The computation of the longest common prefix of two suffixes is equivalent to the computation of their lowest common ancestor LCA in the tree. There is an algorithm (mentioned in Chapter 5) that preprocesses any tree in linear time in order to allow further LCA queries in constant time. This completes the proof. □

Algorithm *Approximate string-matching with at most k errors*;
compute 0-nodes { exact string-matching }
for $d := 1$ **to** k **do begin**
compute d-special nodes; { see Figure 12.6 }
{ computation of d-nodes }
$S = \{NEXT(i,j) : (i,j)$ is a d-special node $\}$;
for each diagonal p **do**
select on p, the lowest node that is in the set S;
{ selected nodes form the set of d-nodes }

12.4 String matching with don't care symbols

In this section, we assume that pattern *pat* and *text* can contain occurrences of the symbol $\oslash$, called the *don't care symbol.* Several different don't care symbols can be considered instead of only one, but the assumption is that they are all indistinguishable from the point of view of string matching. These symbols match any other symbol of the alphabet. We define an associated notion of *matching* on words as follows. We say that two symbols a, b match if they are equal, or if one of them is a don't care symbol (see Figure 12.8). We write $a \approx b$ in this case. We say that two strings u and v (of same length) match if $u[i] \approx v[i]$ for any position i. String matching with don't care symbols entails the problem of finding a factor of *text* that matches the pattern *pat* according to the present relation $\approx$.

$a\ a\ a\ \oslash\ b\ a\ b\ b\ \oslash\ \oslash\ a\ a\ b\ \oslash$

$a\ \oslash\ a\ b\ b\ \oslash\ \oslash\ b\ b\ a\ a\ a\ b\ a$

Figure 12.8: Two strings, with don't care symbols, that match.

String matching with don't care symbols does not use any of the techniques developed for other string-matching questions. This is because the relation $\approx$ is not transitive. Moreover, if symbol comparisons (involving only the relation $\approx$) are the only access to input texts, then there is a quadratic lower bound for the problem, which additionally proves that the problem is quite different from other string-matching problems. The algorithm presented later is an interesting example of a reduction of a textual problem to a problem in arithmetics.

Theorem 12.6 *If symbol comparisons are the only access to input texts, then $\Omega(n^2)$ such comparisons are necessary to solve the string-matching problem with don't care symbols.*

Proof. Consider a pattern of length m, and a text of length $n = 2m$, both consisting entirely of don't care symbols $\oslash$. Occurrences of the pattern start at all positions $1, \ldots, m$. If the comparison "$pat[j] \approx text[i]$," for $1 \leq j \leq m$ and $1 \leq j \leq n$, is not done, then we can replace $pat[j]$ and $text[i]$ by two distinct symbols a, b (that are not don't care symbols). The output then remains unchanged, but one of the occurrences is disqualified. Hence, the algorithm is not correct. This proves that all comparisons "$pat[j] \approx text[i]$" for $1 \leq j \leq m$ and $m < i \leq n$, must be executed. □

Contrary to what occurs elsewhere, we temporarily assume that positions in *pat* and *text* are numbered from zero (and not from one). We start with an algorithm that "multiplies" two words in a manner similar to how two binary numbers are multiplied, but ignoring the carry. We define the product operation $\bullet$ as the composition of $\approx$ and the logical "and" in the following sense. If x, y are two strings, then $z = x \bullet y$ is defined by

$$z[k] = \text{AND}(x[i] \approx y[j] : i + j = k),$$

for $k = 0, 1, \ldots, m + n - 2$. In other words, it is the logical "and" of all values $x[i] \approx y[j]$ taken over all i, j such that both $i + j = k$ and $x[i], y[j]$ are defined: We can write symbolically $\bullet = (\approx,$ and).

Let p be the reverse of pattern *pat*, and consider $z = p \bullet text$. Let us examine the value of $z[k]$. We have $z[k] =$ true iff $(p[m-1] \approx text[k-m+1])$ and $p[m-2] \approx text[k-m+2]$ and ... and $p[0] \approx text[k])$. Therefore, "$z[k] =$ true" exactly means that there is an occurrence of *pat* ending at position k in *text*. Hence, the string matching with don't care symbols reduces to the computation of the product $\bullet$.

Let us define an operation on logical vectors similar to $\bullet$. If x, y are two logical vectors, then $z = x \diamondsuit y$ is defined by

$$z[k] = \text{OR}(x[i] \text{ and } y[j] : i + j = k).$$

For a word x and a symbol a, denote by $logical(a, x)$ the logical vector in which the i-th component is *true* iff $x[i] = a$. Define also

$$LOGICAL_{a,b}(x, y) = logical(a, x) \diamondsuit logical(b, y).$$

The following fact is now apparent: for two words x, y, the vector $x \bullet y$ equals the negation of logical OR of all vectors $LOGICAL_{a,b}(x, y)$ over all distinct symbols a, b that are not don't care symbols.

A consequence of the above fact is that, for a fixed-size alphabet, the complexity of evaluating the product $\bullet$ is of the same order as that of computing the operation $\diamondsuit$. Now, we show that the computation of the operation $\diamondsuit$ can be reduced to the computation of the ordinary product $*$ of two integers. Let x, y be two logical vectors of size n. Let $k = \log n$. Replace logical values *true* and *false* by ones and zeros, respectively. Next, insert an additional group of k zeros between each two consecutive digits. We obtain binary representations of two numbers x', y'. Let z' be the integer $x * y$. The vector $z = x \diamondsuit y$ can be recovered from z' as follows. Take the first digit (starting at position 0), and then each $(k+1)$-th digit of z'; convert one and zero into *true* and *false*, respectively. In this way, we have proven the following statement.

Theorem 12.7 *The string-matching problem with don't care symbols can be solved $IM(n \log n)$ time, where $IM(r)$ denotes the complexity of multiplying two integers of size r.*

The value $IM(r)$ depends heavily on the model of computations considered. If bit operations are counted, then the best known algorithm for the problem is given by the Schönhage-Strassen multiplication, which works in time only slightly larger than $O(r \log r)$. No linear-time algorithm for the problem is known. This gives an $O(n \log^2 n)$-time algorithm for the string-matching problem with don't care symbols. String matching with don't care symbols generates a methodological interest because of its relationship to arithmetics. It would be also interesting to find relationship between some other typical textual problems to arithmetics.

Bibliographic notes

The edit distance computation can be attributed to Needleman and Wunsch [NW 70] and to Wagner and Fischer [WF 74]. Various applications of sequence comparisons are presented in a book edited by Sankoff and Kruskal [SK 83]. Approximate string-matching algorithms are widely used for molecular sequence comparisons, for which a large number of variants have been developed (see, of example [GG 89]). Computation of a longest common subsequence (not only its length) in linear space is from Hirschberg [Hi 75]. The last algorithm of Section 12.2 is from Hunt and Szymanski [HS 77]. It is the base of the "diff" command of UNIX system. An improvement on it is due to Apostolico and Guerra [AG 87].

An algorithm for the longest increasing subsequence can be found in [Ma 89].

There have been numerous substantial contributions to the problem, among them are those by Hirschberg [Hi 77], Nakatsu, Kambayashi, and Yajima [NKY 82], Hsu and Du [HD 84], Ukkonen [Uk 85b], Apostolico [Ap 85] and [Ap 87], Myers [My 86], Apostolico and Guerra [AG 87], Landau and Vishkin [LV 89], Apostolico, Browne, and Guerra [ABG 92], Ukkonen and Wood [UW 93].

A subquadratic solution (in $O(n^2/\log n)$ time) to the computation of edit distances has been given by Masek and Paterson [MP 80] for fixed-size alphabets. A subquadratic solution for unrestricted cost functions may be found in [CLU 02].

The first efficient string matching with errors is by Landau and Vishkin [LV 86b]. The computation of lowest common ancestors (LCA) is discussed by Schieber and Vishkin in [SV 88] (see the bibliographic notes at the end of Chapter 5). The best asymptotic time complexity of the string matching with don't care symbols is achieved by the algorithm of Fischer and Paterson [FP 74]. Practical approximate string matching is discussed by Baeza-Yates and Gonnet [BG 92], and by Wu and Manber [WM 92]. These solutions are close to each others. The second algorithm is implemented under UNIX as command "agrep."