KNUTH - MORRIS - PRATT
ALGORITHM
1977

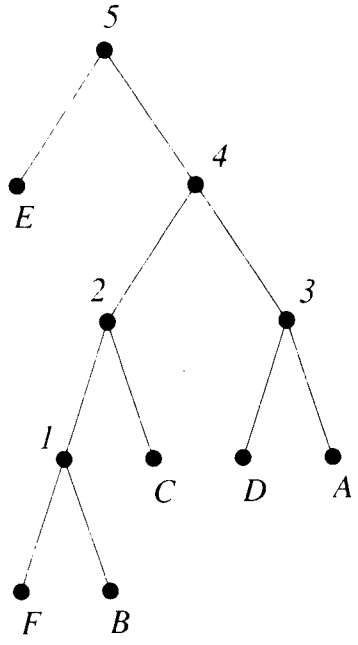

Figure 6.19 The Huffman tree for example 6.1.

The Problem Given two strings A and B, find the first occurrence (if any) of B in A. In other words, find the smallest k such that, for all i, $1 \le i \le m$, we have $a_{k+i} = b_i$.

The most obvious example of this problem is a search for a certain word or pattern in a text file.[1] Any text editor must contain commands to find patterns. The problem also has applications to other areas — including molecular biology, where it is useful to find certain patterns inside large RNA or DNA molecules.

This problem seems simple at first. We can try to match B inside A by starting at the first character of A that matches b_1 and continuing (comparing to b_2 and so on) until we either complete the match or find a mismatch. In the latter case, however, we must go back to the place from which we started and start again. This process is illustrated in Fig. 6.20 by an example that we will use throughout this section. In this example, $A = xyxxyxyxyyxyxyxyyxyxyxx$, and $B = xyxyyxyxyxx$. The first mismatch occurs at a_4 since $b_4 \neq a_4$. We now must start comparing b_1 to a_2, which leads to a mismatch right away. Next, we start at a_3, which is a match, but $a_4 \neq b_2$. The next attempt is more promising: We have a match from a_4 to a_7, only to have a mismatch at a_8. Now, we need to backtrack several steps and to compare b_1 to a_5 (mismatch), then b_1 to a_6, and so on. Eventually, we find a match starting at a_{13}. We may have to backtrack and compare again a substantial number of times, leading to $O(mn)$ number of comparisons

[1] At least, that is the most obvious one to me, as I am currently editing a text file.

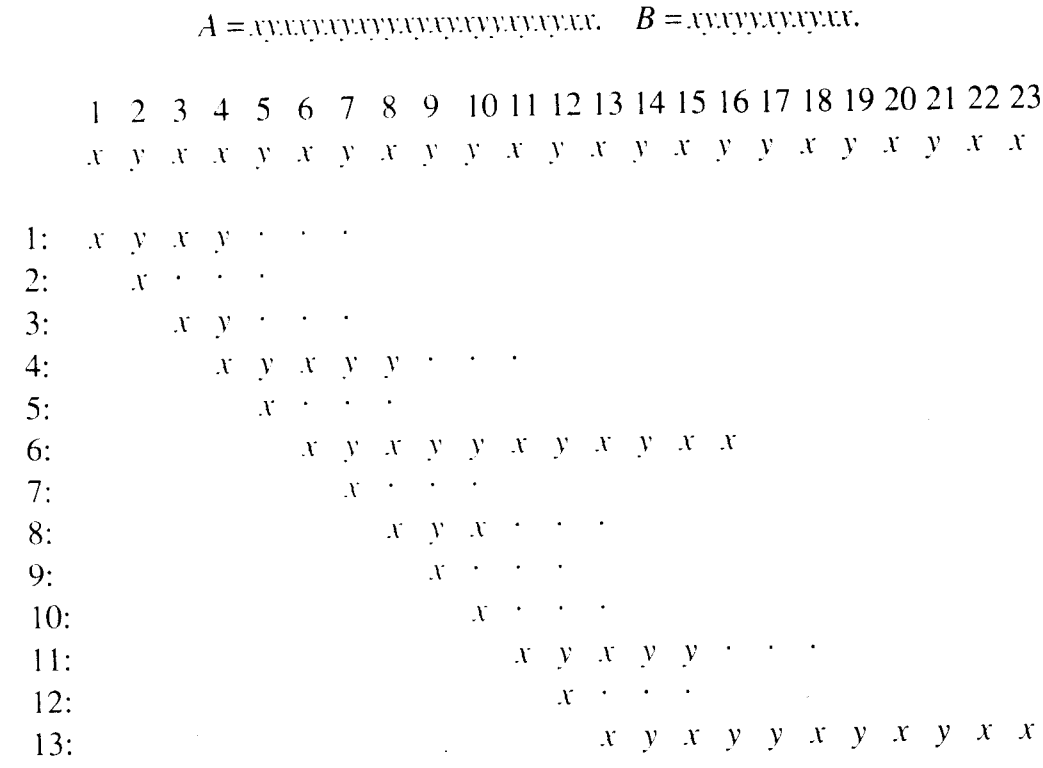

Figure 6.20 An example of a straightforward string matching.

in the worst case. Notice that a lot of the work is redundant. For example, we find twice that the subpattern $xyxy$ fits inside A starting at a_{11} (lines 6 and 11). In the example of finding a word in a text file, the number of backtracking steps will be very small, since most of the time the mismatch will occur early on. This simple algorithm is fairly good for such applications. In other cases, where the alphabet is small and the patterns have many repetitions, the number of backtracking steps may be large. The algorithm above may compare the same subpattern to the same place in the text many times. We would like to find an algorithm that avoids such worst cases. The problem is to arrange the information we learn throughout the algorithm such that it can be used efficiently later on when the same matches occur in other places.

To improve the straightforward algorithm we must first understand the reasons for its inefficiency. The bad case we discussed was caused by the need to backtrack. A particular bad case will occur if the pattern is $yyyyyx$ and the text is $yyyyyyyyyyyyx$. We will compare the five ys in the pattern to the text, find the mismatch with the x, move one step to the right, and make four redundant comparisons again and again. (This simple case is easy to handle, but it illustrates the general problem.) On the other hand, consider the pattern $xyyyyy$. To match this pattern in the text, we look for occurrences of x followed by five ys. If the number of ys is not sufficient, there is no need to backtrack. We will need to find the next x, and all the matched ys will not help. The straightforward algorithm, adapted to the pattern $xyyyyy$, runs in linear time since no backtracking is needed.

Let's return now to the original pattern $B = xyxyyxyxyxx$. Suppose that a mismatch occurs when the fifth character of B is scanned (as it is when a_8 is compared to it in line 4 of Fig. 6.20). The preceding two characters in A must have been xy (since they matched). But, xy are also the first characters of B. We now want to "slide" B to the right and compare the current character in A to some character in the middle of B (taking into account the previous matches). We would like to slide B as far to the right as possible (to save comparisons) without bypassing potential matches. In this case, we can slide B two steps to the right. We continue the match by comparing the same character in A that caused the mismatch (a_8 in the example) to b_3, since we already know that b_1 and b_2 matched. (In fact, that is exactly what we did later on, in line 6 of Fig. 6.20, except that it took us three more redundant comparisons — x in line 5, and xy at the beginning of line 6 — to get there.) Notice that this whole discussion is completely independent of A! We know the last few characters in A since they have matched B so far.

In the following discussion, we will not assume that there are only two characters in the text (and pattern), even though, for simplicity, the examples will contain only two characters. It is possible (and that is the subject of Exercise 6.45) to make the algorithm even more efficient in this case.

Let's look at another example by continuing the match. The mismatch at line 6 of Fig. 6.20 is at the last character of B, b_{11}. We can now do a lot more sliding. Consider the subpattern $B(10) = b_1 b_2 \cdots b_{10}$. We know that $B(10)$ is exactly the same as the preceding 10 characters in A; that is, $B(10) = A[6..15]$, because they matched. We want to determine exactly how many steps B can be shifted to the right until there is some hope of another match. We determine this number by looking for a maximum *suffix* of $B(10)$ that is equal to a *prefix* of B. In this case, that suffix is of length 3 (xyx), as is illustrated in Fig. 6.21. In the figure, $B(10)$ is shifted, one step at a time, and is compared to itself, until a prefix matches a suffix. (The last character, b_{11}, is ignored since it is the cause of the mismatch.) Since we know that $B[1..3] = B[8..10]$, we can continue by comparing a_{16} to b_4, and so on, until the complete match occurs. We save all the comparisons on lines 7 to 12 and half those on line 13. The only difference between Fig. 6.21 and Fig. 6.20 is that the information in Fig. 6.21 depends only on B. This is important because we can preprocess B once, and find all the relevant information about it regardless of the text A. We now can take advantage of all the matches done in line 6 of Fig. 6.20; none of them will be repeated.

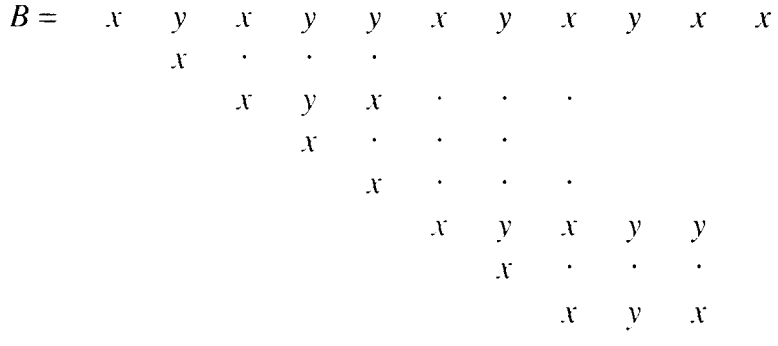

Figure 6.21 Matching the pattern against itself.

The preprocessing of B is the essence of the improved algorithm. We will study all the repeating patterns of B and devise a way to handle mismatches when they occur without backtracking. Our scheme is the following. The string A is always scanned forward; there is no backtracking in A, although the same character of A may be compared to several characters of B (when there are mismatches). When a mismatch is encountered, we consult a table to find how far in B we must backtrack. There is an entry in the table for each character in B corresponding to the amount of backtracking (or the number of shifts) required when there is a mismatch involving this character. In a moment, we will show how to construct this table efficiently. We first define the table precisely and show how we use it for the string-matching problem.

The idea behind the table should be clear now. For each b_i we want to find the largest suffix of $B(i-1)$ that is equal to a prefix of $B(i-1)$. If the length of this suffix is j, then the mismatched character in A can be matched against b_{j+1} directly, without going through all the other redundant matches. We already know that the most recent j characters in A match the beginning of B. Furthermore, since this suffix is the *largest* among those that are equal to a prefix, we know that B cannot fit into A any farther to the left. The table is called *next*, and here is a precise definition of the values of its entries:

> $next(i)$ = the maximum j $(0 < j < i-1)$ such that $b_{i-j}\, b_{i-j+1} \cdots b_{i-1} = B(j)$, and 0 if no such j exists.

For convenience we define $next(1) = -1$ to distinguish this case. It is clear that $next(2)$ is always equal to 0 (since there is no j satisfying $0 < j < 2-1$). The values of the *next* table for the pattern B in Fig. 6.21 are given in Fig. 6.22. These values can be computed in a brute force way, as was done in Fig. 6.22. However, there is an elegant way to compute all these values in time $O(m)$. Let's first assume that the values of *next* are given to us, and see how to perform the matching. Afterwards, we will describe how to compute *next*.

The matching proceeds as follows. The characters in A are compared to those in B until there is a mismatch. At that point, say at b_i, the *next* table is consulted and the same character in A is compared against $b_{next(i)+1}$ (since the first $next(i)$ characters already match). If this is a mismatch too, then the next comparison is against $b_{next(next(i)+1)+1}$, and so on. The only exception to this rule is when the mismatch is against b_1; in this case,

$i =$	1	2	3	4	5	6	7	8	9	10	11
$B =$	x	y	x	y	y	x	y	x	y	x	x
$next =$	-1	0	0	1	2	0	1	2	3	4	3

Figure 6.22 The values of *next*.

we want to proceed in A. This case can be determined by the special value of $next(1)$, which is -1. The program for string matching is given in Fig. 6.23.

Algorithm String_Match *(A, n, B, m)* ;
Input: *A* (a string of size *n*), and *B* (a string of size *m*).
{ We assume that *next* has been computed; see Fig. 6.25 }
Output: *Start* (the first index such that *B* is a substring of *A* starting at *A* [*Start*]).

```
begin
    j := 1 ; i := 1 ;
    Start := 0 ;
    while Start = 0 and i ≤ n do
        if B [j ] = A [i ] then
            j := j + 1 ;
            i := i + 1
        else
            j := next [j ] + 1 ;
            if j = 0 then
                j := 1 ;
                i := i + 1 ;
        if j = m + 1 then Start := i - m
end
```

Figure 6.23 Algorithm *String_Match*.

It remains to find an algorithm to compute the values of the *next* table. We use induction. As we mentioned, $next(2)=0$, which takes care of the base case. We assume that the values of *next* for 1, 2, ..., $i-1$ have been computed, and we consider $next(i)$. At best, $next(i)$ can be $next(i-1)+1$, which will happen if $b_{i-1}=b_{next(i-1)+1}$. In other words, the largest suffix that is equal to a prefix is extended by b_{i-1}. This is the easy case. The difficult case is when $b_{i-1} \neq b_{next(i-1)+1}$. We need to find a new suffix that is equal to a prefix. However, we already know how to fit the largest suffix of $B(i-2)$: It fits in $b_1 b_2 \cdots b_{next(i-1)}$ (see Fig. 6.24). But having $b_{i-1} \neq b_{next(i-1)+1}$ is exactly the same as having a regular mismatch at $b_{next(i-1)+1}$! And we already know what to do about that. If there is a mismatch at index j, we go to $next(j)$. So, we have a mismatch at index $next(i-1)+1$, and we go to $next(next(i-1)+1)$. That is, we try to match b_{i-1} to $b_{next(next(i-1)+1)+1}$. If they match, we set $next(i) = next(next(i-1)+1)+1$. Otherwise, we continue in the same fashion until we either get a match or we return to the beginning.

□ Example 6.2

Let $B = xyxyyxyxyxx$ (the same as in Fig. 6.21), and consider $next(11)$. We first look at $next(10)$, which is 4, and compare b_{10} to b_5. If they had been the same, then the largest

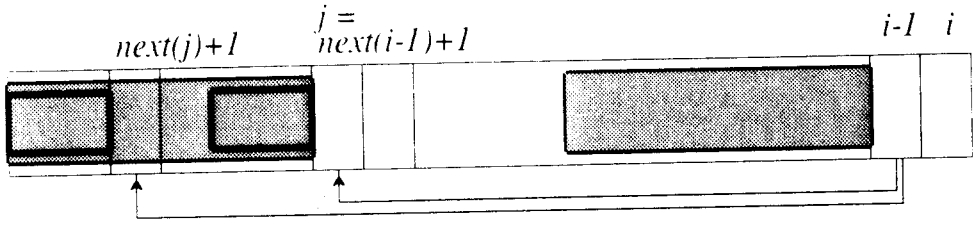

Figure 6.24 Computing next(i).

prefix that is equal to a suffix would have been 5, but they are not. So, we have a mismatch at b_5, and we look at *next* (5) which is 2. We now compare b_{10} to b_3, and they happen to be the same. Hence, *next* (11) = 3, which can easily be verified by hand. □

The algorithm for computing the *next* table is difficult to understand, but it is not difficult to implement. The program is given in Fig. 6.25.

Algorithm Compute_Next *(B, m)* ;
Input: *B* (a string of size *m*).
Output: *next* (an array of size *m*).

```
begin
    next(1) := -1 ;
    next(2) := 0 ;
    for i := 3 to m do
        j := next(i - 1) + 1 ;
        while b_{i-1} ≠ b_j and j > 0 do
            j := next(j) + 1 ;
        next(i) := j
end
```

Figure 6.25 Algorithm *Compute_Next*.

Complexity A character of *A* may be compared against many characters of *B*. If there is a mismatch, then the same character of *A* is compared against the character of *B* pointed to by the *next* table. If there is another mismatch, then we continue comparing against the same character of *A* until there is either a match or we reach the beginning of *B*. Nevertheless, we claim that the running time of this algorithm is still $O(n)$. How many times can we backtrack for one character from *A*, say a_i? Let's assume that the first mismatch involved b_k. Since each backtrack leads us to a smaller index in *B*, we can backtrack only k times. However, to reach b_k we must have gone *forward* k times without any backtracking! If we assign the costs of backtracking to the forward moves, then we at most double the cost of the forward moves. But there are exactly n forward moves, so the number of comparisons is $O(n)$.