XVII

String Matching

- **String matching**: Given two strings P, T over some alphabet Σ

Pattern $|P| = m$
Text $|T| = n$

Find the first (all) occurrences of P in T

- Web searches
- bibliographic search
- text editor

- **Straight forward approach**

Comparisons are done (in left-to-right order) on the pairs of characters indicated by arrows

P: A B A B C
T A B A B A B C C A
mismatch ↑

A B A B C
A B A B A B C C A
Successful match

When a mismatch occurs, we slide the pattern to the right with respect to the text.

But by how much?

XVII

String Matching - A Straightforward Approach

Algorithm 5.1 Straightforward String-matching Algorithm

Input: P and T, the pattern and text strings. If *P.length* and *T.length* are not known in advance, their explicit use in the algorithm can be replaced by end-of-string tests.

Output: The index in T where a copy of P begins. The index will be *T.length*+1 if no match for P is found.

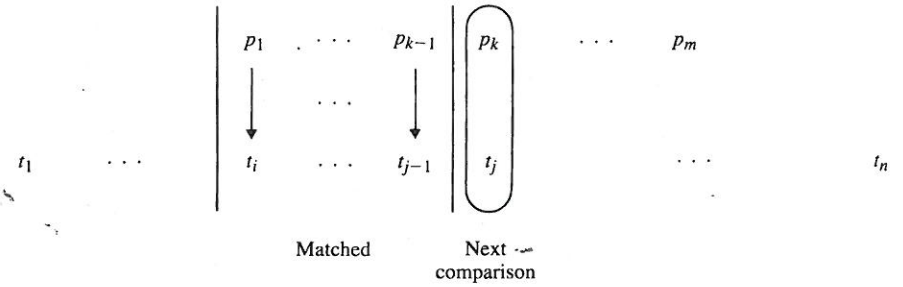


Figure 5.1 The general picture for Algorithm 5.1.

Comment: The general picture is shown in Fig. 5.1. The index variable i is not really needed in the algorithm since it can be computed from j and k (that is, $i = j-k+1$).

```
function Match (P, T : String) : index;
var
    i, j, k : index;
    { i is the current guess at where P begins in T;
      j is the index of the current character in T;
      k is the index of the current character in P.    }
begin
    i := 1; j := 1;  k := 1;
    while j≤T.length and k≤P.length do
        if tj = pk then
            j := j+1;  k := k+1
        else { slide pattern forward and start over }
            i := i+1;  j := i;  k := 1
        end { if }
    end { while };
    if k>P.length then Match := i  { match found }
    else Match := j  { j = T.length+1, no match }
    end { if }
end { Match }
```

String Matching - A Straightforward Approach

- time complexity of straightforward approach

counting character comparisons

- If P occurs at the beginning of the text - n comparisons are made.
- If the first pattern character, p_1 is not in the Text, then n comparisons occur.
- In the worst case $m \cdot n$ comparisons are made. The algorithm is $\Theta(m \cdot n)$

When does this worst case occur?

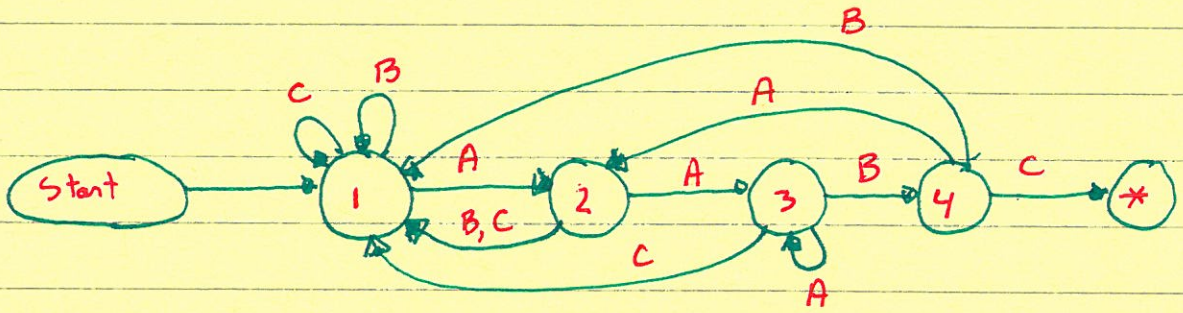
- This algorithm has an undesirable property
- The assignment "$j := i$" in the while loop requires that we back up in the text string
- When text is long, overhead may be costly.

The next approach requires no backing up.

Pattern Matching with Finite Automata

- Given a pattern P, construct a finite automaton.

Let P = 'AABC'



(*) : stop node. A match was found.

- (1) ... (4) read nodes. Read the next text character. If there are no further characters in the text string, halt, there is no match.

- If M is in state 3, the last two characters have been AA.

suffix / prefix comparisons of the pattern with itself.

- We can use M to search through text to find occurrences of pattern in O(n) time.

We never back up within the text. Characters only considered once. // However too many arcs.

Knuth-Morris Pratt Flowchart

6 Put arrows that correspond to a successful match

There are called success links.

- The kmp flowchart places labels on the nodes rather than on the arrows.
- If a success link has been followed, then the next text character is read.
- If a failure link is followed, then the same text character is reconsidered.
- As before, if the * is reached, a copy of the pattern has been found.
- Once the flowchart is completed, the text can be examined for an occurrence of the pattern in $O(n)$ time.

Each text character need only be examined once.

KMP flowchart

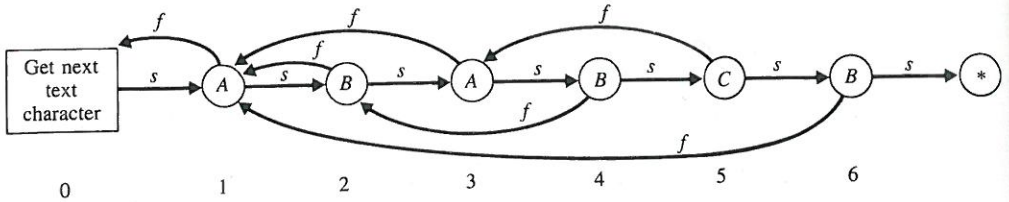
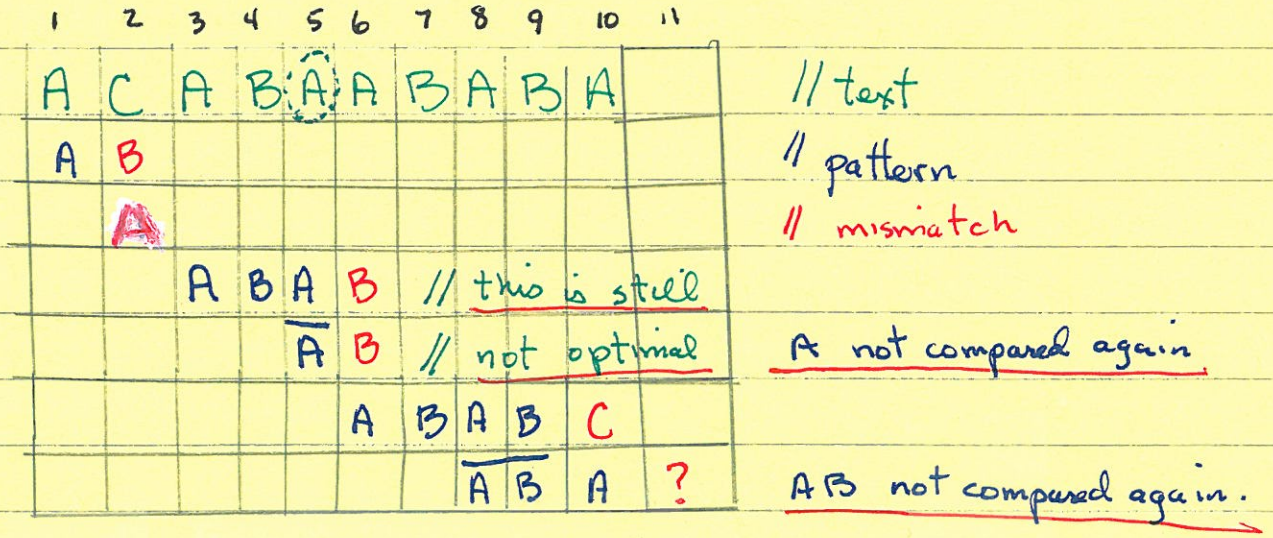


Figure 5.3 The KMP flowchart for P = '*ABABCB*'.

Use this flowchart to search for an occurrence of the pattern within the text ACABAABABA



KMP cell number	Text character being scanned: Index	Character	Success (s) or failure (f)
1	1	*A*	s
2	2	*C*	f
1	2	*C*	f
0	2	*C*	get next char.
1	3	*A*	s
2	4	*B*	s
3	5	*A*	s
4	6	*A*	f
2	6	*A*	f
1	6	*A*	s
2	7	*B*	s
3	8	*A*	s
4	9	*B*	s
5	10	*A*	f
3	10	*A*	s
4	11	none	failure

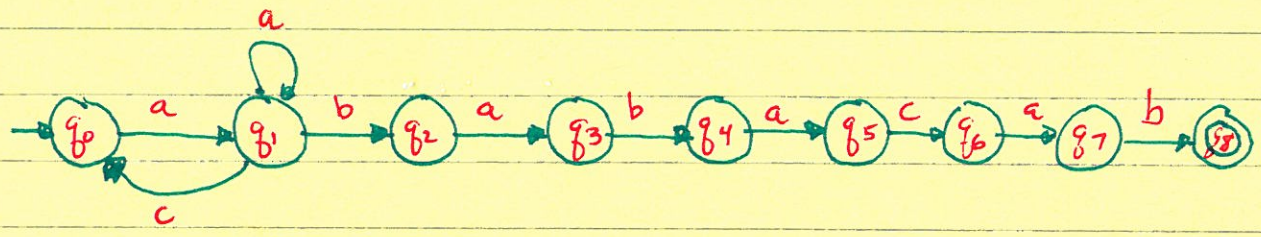
KMP - con't.

- Review

Let pattern = 'abababacab'

- Build a finite automaton for this pattern string.

M

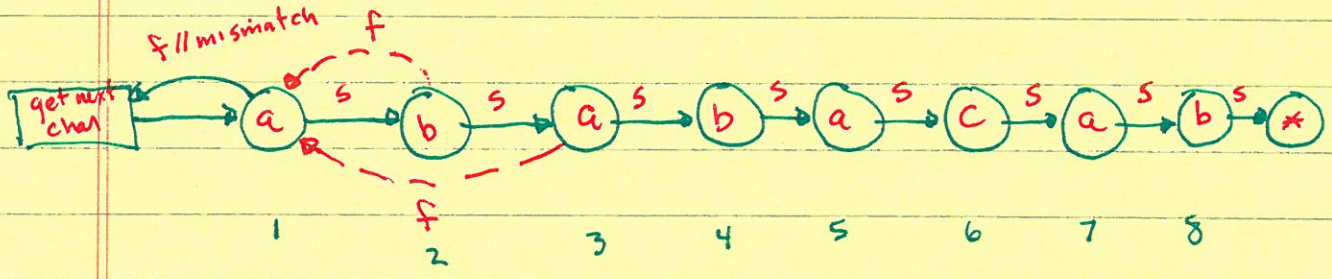


complete this dfa

Process text with M.

We avoid $\Theta(mn)$ time - however space & time cumbersome

- KMP flowchart

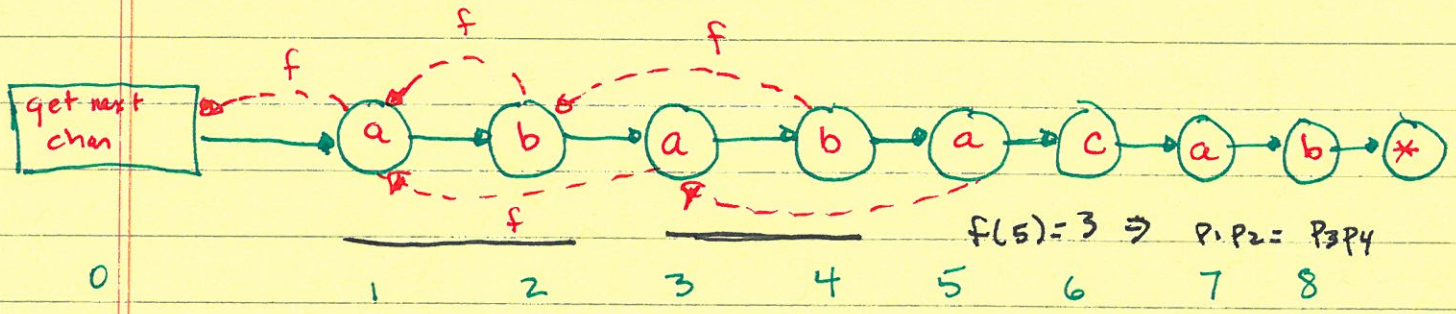


s: success links

f: failure links.

How are these failure links filled in?

Kmp - con't.



In general, $f(i) = r$ where $r < i$

this implies $p_1 p_2 \cdots p_{r-1} \; . \; = p_{i-r+1} \cdots p_{i-1}$

$f(i+1) = ?$

$f(1) = 0$ // mismatch

$f(2) = 1$

$f(3) = 1$

$f(4) = ?$

We ask if $p_{3+1} = p_{f(3)+1}$

$p_4 = p_2$

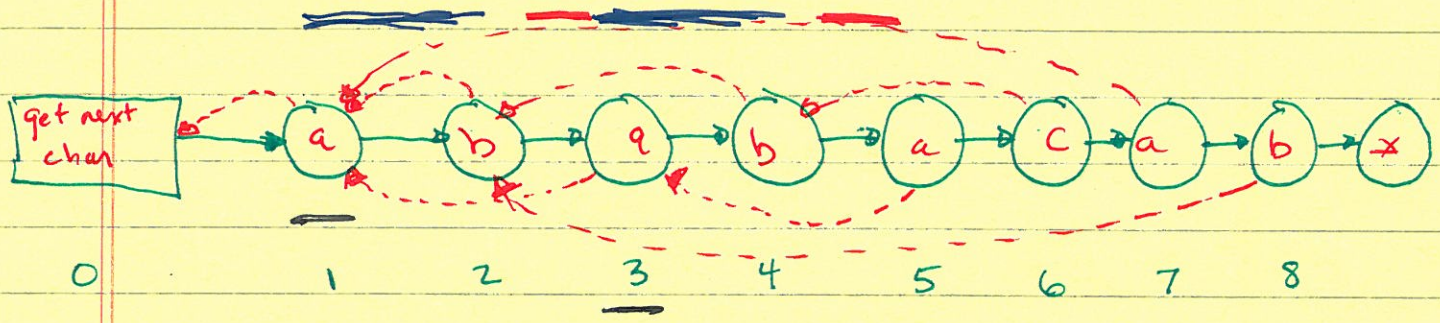
$b = b$ // yes.

$\therefore f(4) = f(3) + 1$

and $p_3 \; . = p_1 \; .$

$a \; . = a \; .$

f(5)=3 f(6)=4 KMP- con't.



$f_4 = 2$ ∴ $p_3 = p_1$

a = a

---

f(5) = ?

$p_{i+1} = p_{f(i)+1}$?

$p_5 = p_{2+1}$

a = a ? yes

∴ f(5) = f(4) + 1 = 2 + 1 = 3

$p_3 p_4$ $p_1 p_2$

ab = ab

---

f(6) = ? //see above

f(6) = 4

∴ $p_1 p_2 p_3 = p_3 p_4 p_5$

'aba' = 'aba'

f(7) = ? Is $p_6 = p_{f(6)}$?

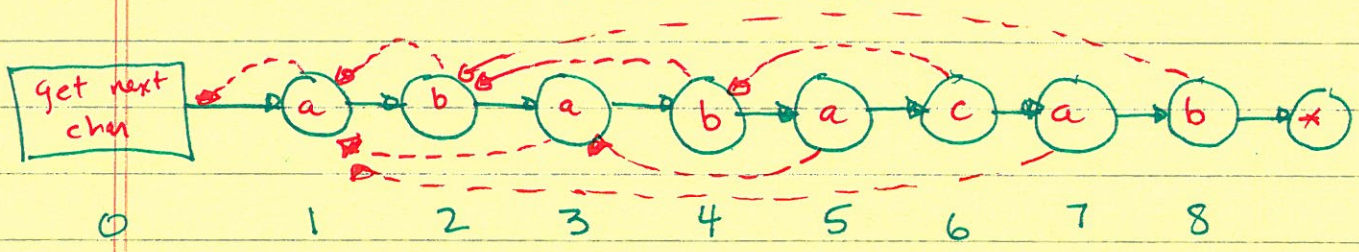
c = b // NO

Is $p_6 = p_{f^2(6)}$

c = $p_{f(4)}$

c = b // NO

Kmp - con't.



f(7) continued

$p_6 = p_{f^3(6)}$

$= p_{f^2(4)}$

$= p_{f(2)}$

$p_6 = p_1$

$c = a$ // still no.

∴ set $f(7) = 1$

no suffix - prefix relationship

Finally, confirm that $f(8) = 2$

indicating that $p_7 = p_1$

$a = a$

XVII

KMP- con't.

Failure function table

	a	b	a	b	a	c	a	b
i	1	2	3	4	5	6	7	8
f(i)	0	1	1	2	3	4	1	2

Find all occurrences of this pattern in the text

T = abbaabbabbabacab

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

text a b b a b a b b a b a b a c a b

a b a // f(3) = 1, slide i - f(i) = 3 - 1 = 2 places

a // f(1) = 0, slide i - f(i) = 1 - 0 = 1 place

a b a b a // slide i - f(i) = 5 - 3 = 2 places

no comparisons ... a b a // slide i - f(i) = 3 - 1 = 2 places

a // slide i - f(i) = 1 - 0 = 1 place

a b a b a c a b // match

starting at location 16 - 8 + 1 = 9

5.2.3 Construction of the Knuth-Morris-Pratt Flowchart

This flowchart representation is quite simple; it uses two arrays, one containing the characters of the pattern and one containing the failure links. The success links are implicit in the ordering of the array entries.

Let *fail* be the array of failure links; *fail*[*k*] will be the index of the node pointed to by the failure link at the *k*th node, for $1 \le k \le m$. The special node that merely forces the next text character to be read is considered to be the zero-th node; *fail*[1] = 0. To see how to set the other failure links, we consider an example.

Example 5.2 Setting failure links for the KMP algorithm

Let *P* = '*ABABABCB*' and suppose that the first six characters have matched six consecutive text characters as indicated:

```
P :        | A B A B A B | C B
           | ↓ ↓ ↓ ↓ ↓ ↓ |
T : . . .  | A B A B A B | x . . .
```

Suppose that the next text character, *x*, is not a '*C*'. The next possible place where the pattern could begin in the text is at the third position shown, as follows:

```
P :          | A B A B | A B C B
             |         |
T : . . . A B | A B A B | x . . .
```

The pattern is moved forward so that the longest initial segment that matches part of the text preceding *x* is lined up with that part of the text. Now *x* should be tested to see if it is an *A* to match the third *A* of the pattern. Thus the failure link for the node containing the *C* should point to the node containing the third *A*. ■

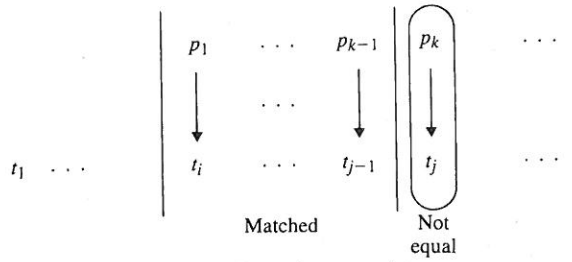
The general picture is shown in Fig. 5.4. When a mismatch occurs, we want to slide *P* forward, but maintain the longest overlap of a prefix of *P* with a suffix of the part of the text that has matched the pattern so far. Thus, the current text character should be compared to p_r next; that is, *fail*[*k*] should be *r*. But we want to construct the flowchart before we ever see *T*. How do we determine *r* without knowing *T*? The key observation is that when we do scan *T*, the part of *T* just scanned will have matched the part of *P* just scanned, so we need only find the longest overlap of a prefix of *P* with a suffix of the part of *P* just scanned. More precisely,

fail[*k*] is the largest *r* (with $r < k$) such that $p_1 \ldots p_{r-1}$ matches $p_{k-r+1} \ldots p_{k-1}$.

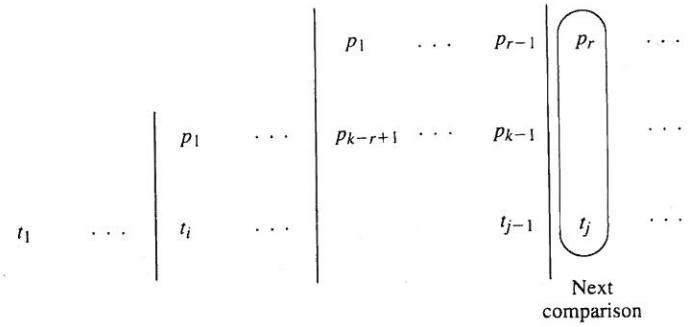
Thus the failure links are determined by repetition within *P* itself.

An occurrence of the pattern could be missed if *r* were chosen too small. (Consider what would happen if in Example 5.2 the failure link for *C* were set to point to the second *A*, and if $x = A$ and is followed by *BCB* in the text.)

Although we have described the correct values for the failure links, we still do not have an algorithm to compute them efficiently. Suppose that the first $k-1$ failure



(a) Mismatch at p_k and t_j.



(b) Slide p to line up the longest prefix that matches a suffix of the scanned characters.

Figure 5.4 Sliding the pattern for the KMP algorithm.

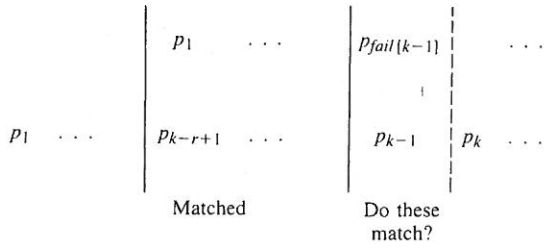
links have been computed. Then we have the picture in Fig. 5.5(a). To assign $fail[k]$ we need to match a suffix ending at p_{k-1}. The easy case is when $p_{k-1} = p_{fail[k-1]}$. Then the two matching sequences in Fig. 5.5(a) can be extended by one more character, so in this case $fail[k]$ is assigned $fail[k-1]+1$. In the example in Fig. 5.6, $fail[6] = 4$ because $p_1 p_2 p_3$ matches $p_3 p_4 p_5$. Since $p_6 = p_4$, $fail[7]$ is assigned 5.

What if $p_{k-1} \neq p_{fail[k-1]}$, as in Fig. 5.6 for $k = 8$? We must find an initial substring of P that matches a substring ending at p_{k-1}. In this case the match in Fig. 5.5(a) cannot be extended, so we look farther back. Let $r = fail[k-1]$ and let $r' = fail[r]$. By the properties of the failure links we have the matches shown in Fig. 5.5(b). If $p_{r'} = p_{k-1}$, we have an initial substring to match a substring ending at p_{k-1} and $fail[k]$ should be $r'+1$. If $p_{r'} \neq p_{k-1}$, we must follow the failure link from node r' and try again. This process is continued until we find a failure link r such that $p_r = p_{k-1}$ or (as in Fig. 5.6 for $k = 8$) $r = 0$. In either case, $fail[k]$ should be $r+1$.

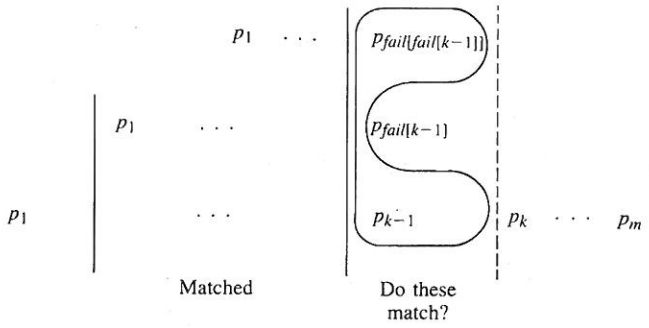
Algorithm 5.2 KMP flowchart construction

Input: P, a string of characters. If the length of P is not known in advance, its explicit use in the algorithm can be replaced by an end-of-string test.

Output: *fail*, the array of failure links.



(a) By definition of $fail[k-1]$.



(b) Looking back for a match for p_{k-1}.

Figure 5.5 Computing *fail* links.

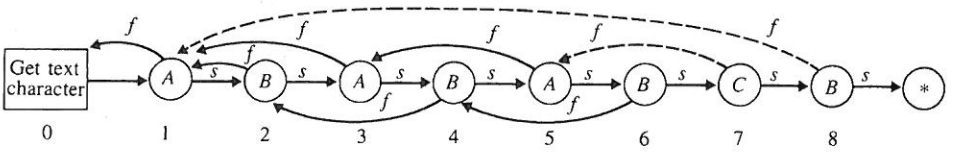


Figure 5.6 Computing *fail* links — an example.

```
procedure KMPsetup (P: String; var fail: IndexArray);
var
    k,r: integer;
begin
    fail[1] := 0;
    for k := 2 to P.length do
        r := fail[k-1];
        while r > 0 and p_r ≠ p_{k-1} do[1]
            r := fail[r]
        end { while };
```

[1] Reminder to Pascal programmers: By the semantics of our Boolean expressions, no reference to p_r is made if $r = 0$.

```
            fail[k] := r+1
        end { for }
    end { KMPsetup }
```

5.2.4 Analysis of the Flowchart Construction

Let $m = P.length$. It is easy to see that the complexity of Algorithm 5.2 is in $O(m^2)$. The body of the **for** loop is executed $m-1$ times, and each time, the body of the **while** loop is executed at most m times because r starts somewhere in P and "jumps" backward, at worst to zero. But this analysis is not careful enough.

We will count character comparisons as we did for Algorithm 5.1. Since the comparison is part of the condition of the **while** loop, the running time of the algorithm is bounded by a multiple of the number of comparisons. (Actually, since the character comparison is not executed when $r = 0$, we should also note that $r = 0$ cannot occur more than $m-1$ times.)

We call a comparison "successful" if $p_r = p_{k-1}$ and "unsuccessful" otherwise. A successful comparison terminates the **while** loop so at most $m-1$ successful comparisons are done (for k from 2 to $P.length$). After every unsuccessful comparison r is decreased (since $fail[r] < r$), so we can bound the number of unsuccessful comparisons by determining how many times r can decrease. Observe the following:

1. r is initially assigned $fail[1] = 0$ (the first time the first statement in the **for** loop is executed).
2. r is increased by exactly 1 each subsequent time that the first statement in the **for** loop is executed. The relevant steps are:

   $fail[k] := r+1$ (at the end of the previous pass through the **for** loop);
   k is incremented by the **for** loop control;
   $r := fail[k-1]$ (the first instruction in the **for** loop).

   So the first statement in the **for** loop is actually assigning $r+1$ to r.
3. r is incremented $m-2$ times.
4. r is never negative.

Since r starts at 0 and is incremented by 1 $m-2$ times and is never negative, r cannot be decreased more than $m-2$ times. Thus the number of unsuccessful comparisons is at most $m-2$ and the total number of character comparisons is at most $2m-3$. Observe that, to count character comparisons, we actually counted the number of times the index r changed. The latter is another good measure of the work done by the algorithm. The important conclusion is that the complexity of the construction of the flowchart is linear in the length of the pattern.

5.2.5 The Knuth-Morris-Pratt Scan Algorithm

We have already informally described the procedure for using the KMP flowchart to scan the text. The algorithm follows.

Algorithm 5.3 KMP Scan Algorithm

Input: P and T, the pattern and text strings; *fail*, the array of failure links set up in Algorithm 5.2. If the length of T is not known in advance, its explicit use in the algorithm can be replaced by an end-of-string test. The length of P would have been found when setting up the *fail* array.

Output: The index in T where a copy of P begins. The index will be $T.length+1$ if no match for P is found.

```
function KMPmatch (P, T : String; fail: IndexArray) : index;
var
   j, k : index;
   { j indexes text characters;
     k indexes the pattern and fail array. }
begin
   j := 1; k := 1;
   while j≤T.length and k≤P.length do
      if k = 0 or t_j = p_k then[2]
         j := j+1; k := k+1
      else { follow fail arrow }
         k := fail[k]
      end { if }
   end { while };
   if k>P.length then KMPmatch := j−P.length  { match found }
   else KMPmatch := j  { j = T.length+1, no match }
   end { if }
end { KMPmatch }
```

The analysis of the scan algorithm uses an argument very similar to that used to analyze the algorithm to set up the failure links, and it is left to the reader. The number of character comparisons done by Algorithm 5.3 is at most $2n$, where $n = T.length$. Thus, the Knuth-Morris-Pratt pattern-matching algorithm, which is comprised of Algorithms 5.2 and 5.3, does $\Theta(n+m)$ operations in the worst case, a significant improvement over the $\Theta(mn)$ worst-case complexity of Algorithm 5.1. Some empirical studies have shown that the two algorithms do roughly the same number of character comparisons on the average (for natural language text), but the KMP algorithm never has to back up in the text.

[2] Reminder to Pascal programmers: By the semantics of our Boolean expressions, no reference to p_k is made if $k = 0$.

Two extensions to the pattern-matching problem are often useful: Find *all* occurrences of the pattern in the text, and find any one of a finite set of patterns in the text.

Exercises

Section 5.1: The Problem and a Straightforward Solution

5.1. Rewrite Algorithm 5.1 eliminating the variable i.

5.2. Rewrite Algorithm 5.1 to work on inputs that are simply linked lists. Assume that each node has a *key* field and a *link* field, and assume that T and P are pointers to the first nodes of the two lists.

Section 5.2: The Knuth-Morris-Pratt Algorithm

5.3. Draw the finite automaton (flowchart) for the pattern '*ABAABA*', where $\Sigma = \{A, B, C\}$.

5.4. Give the fail indexes used by the KMP algorithm for the following patterns:

a) *AAAB*

b) *AABAACAABABA*

c) *ABRACADABRA*

d) *ASTRACASTRA*

5.5. Give a pattern beginning with an A and using only letters from $\{A, B, C\}$ that would have the following fail indexes (for the KMP algorithm):

0 1 1 2 3 4 2 2

5.6. Show that the KMP scan algorithm (Algorithm 5.3) does at most $2n$ character comparisons.

5.7. How will the KMP algorithms behave if the pattern and/or the text are null (have length zero)? Will they "crash"? If not, will their output be meaningful and correct?

5.8. Recall that the pattern P = '$A \ldots AB$' ($m-1$ A's followed by one B) and the text string T = '$A \ldots A$' (n A's) are a worst-case input for Algorithm 5.1.

a) Give the values of the *fail* indexes for P. Exactly how many character comparisons are done by *KMPsetup* (Algorithm 5.2) to compute them?

b) Exactly how many character comparisons are done by the KMP scan algorithm to scan T for an occurrence of P?

c) Given m, find a pattern Q with m letters such that *KMPsetup* does more character comparisons for Q than it does for the pattern P with m letters described above.

5.9. Prove that Algorithm 5.2 sets the KMP failure links so that

fail[k] is the largest r (with $r < k$) such that $p_1 \ldots p_{r-1}$ matches $p_{k-r+1} \ldots p_{k-1}$.

*5.10. The strategy for setting the fail links for the KMP algorithm has a flaw that is illustrated by Fig. 5.3. If a mismatch occurs at the fourth character, a B, *fail*[4] points us back to another B, which of course will not match the current text character either. Modify Algorithm 5.2 so that *fail* values satisfy the condition stated in Section 5.2.3

(and repeated in the previous exercise) *and also* the condition that $p_r \neq p_k$. (Be careful; a common first guess does not work.)

5.11. Rewrite the KMP algorithms to work on inputs that are simply linked lists. Assume that each node has a *key* field and a *link* field, and assume that T and P are pointers to the first nodes of the lists.

Section 5.3: The Boyer-Moore Algorithm

5.12. List the values in the *charJump* array for the Boyer-Moore algorithm for the following patterns assuming that the alphabet is $\{A, B, \ldots, Z\}$.

a) *ABRACADABRA*

b) *ASTRACASTRA*

5.13. List the values in the *matchJump* array for the Boyer-Moore algorithm for the following patterns.

a) *AAAB*

b) *AABAACAABABA*

c) *ABRACADABRA*

d) *ASTRACASTRA*

5.14. As Example 5.3 showed, just using the *charJump* values, without using *matchJump*, can give a very fast scan. However, the statement

$$j := j + \max(charJump[t_j], matchJump[k])$$

in Algorithm 5.6 cannot simply be replaced by

$$j := j + charJump[t_j]$$

Why not? What other (small) change is needed to make the scan algorithm work?

5.15. Recall that the pattern P = '$A \ldots AB$' ($m-1$ A's followed by one B) and the text string T = '$A \ldots A$' (n A's) are a worst-case input for Algorithm 5.1.

a) Give the values of the *charJump* and *matchJump* arrays for P.

b) Exactly how many character comparisons are done by the BM scan algorithm to scan T for an occurrence of P?

5.16. Give a formula relating *matchJump*[k] to the number of positions the pattern slides right when there is a mismatch at p_k.

5.17. Suppose that P and T are bitstrings.

a) Show the values in the *charJump* and *matchJump* arrays for the pattern 1101101011.

b) For bitstrings in general, which array, *charJump* or *matchJump*, will yield the longer "jumps"?

Additional Problems

5.18. Rewrite each of the three scan algorithms (Algorithms 5.1, 5.3, and 5.6) so that they find all occurrences of the pattern in the text.

5.19. P is a character string (of length m) consisting of letters and at most one asterisk ('*'). The asterisk is a "wild-card" character; it can match any sequence of zero or more characters. For example, if P = '*sun*day*' and T = '*happysundaemonday*', there is a match

beginning at the 's' and ending at the second 'y'; the asterisk "matches" *daemon*. Give an algorithm to find a match of P in a text string T (consisting of n characters), if there is one, and give an upper bound on the order of its worst-case time.

5.20. Let $X = x_1x_2 \ldots x_n$ and $Y = y_1y_2 \ldots y_n$ be two character strings. We say that X is a *cyclic shift* of Y if there is some r such that $X = y_{r+1} \ldots y_n y_1 \ldots y_r$. Give an $O(n)$ algorithm to determine whether X is a cyclic shift of Y.

5.21. a) Write an efficient algorithm to determine whether a (long) string of text contains 25 consecutive blanks. (Do not just give an exact copy of an algorithm in the text; customize it.)

*b) Construct a worst-case (or near worst-case) example for your algorithm. How many character comparisons are done in this case?

c) Suppose the text string contains ordinary English text in which blanks separate words and sentences but in which there is very rarely more than one blank together. If the text length is n, approximately how many character comparisons will your algorithm do?

*5.22. Investigate the problem of finding any one of a finite set of patterns in a text string. Can you extend any of the algorithms in this chapter to produce an algorithm that does better than scanning for each of the patterns separately?

Programs

Implement all three string-searching algorithms, including a counter for the number of character comparisons done; run a large set of test cases; and compare the results.

Notes and References

The main references for the algorithms presented here are Knuth, Morris, and Pratt (1977) and Boyer and Moore (1977). (The code in Algorithm 5.5 follows Smit (1982).) Guibas and Odlyzko (1977), Galil (1979), and Apostolico and Giancarlo (1986) present various worst-case linear versions of the Boyer-Moore algorithm. See also Aho and Corasick (1975). Boyer and Moore (1977) and Smit (1982) give empirical comparisons of the algorithms described in this chapter. The graph in Fig. 5.10 is from Smit.