

## Exact Matching: Classical Comparison-Based Methods

---

### 2.1. Introduction

This chapter develops a number of classical comparison-based matching algorithms for the exact matching problem. With suitable extensions, all of these algorithms can be implemented to run in linear worst-case time, and all achieve this performance by preprocessing pattern  $P$ . (Methods that preprocess  $T$  will be considered in Part II of the book.) The original preprocessing methods for these various algorithms are related in spirit but are quite different in conceptual difficulty. Some of the original preprocessing methods are quite difficult.<sup>1</sup> This chapter does not follow the original preprocessing methods but instead exploits fundamental preprocessing, developed in the previous chapter, to implement the needed preprocessing for each specific matching algorithm.

Also, in contrast to previous expositions, we emphasize the Boyer–Moore method over the Knuth–Morris–Pratt method, since Boyer–Moore is the practical method of choice for exact matching. Knuth–Morris–Pratt is nonetheless completely developed, partly for historical reasons, but mostly because it generalizes to problems such as real-time string matching and matching against a set of patterns more easily than Boyer–Moore does. These two topics will be described in this chapter and the next.

### 2.2. The Boyer–Moore Algorithm

As in the naive algorithm, the Boyer–Moore algorithm successively aligns  $P$  with  $T$  and then checks whether  $P$  matches the opposing characters of  $T$ . Further, after the check is complete,  $P$  is shifted right relative to  $T$  just as in the naive algorithm. However, the Boyer–Moore algorithm contains three clever ideas not contained in the naive algorithm: the right-to-left scan, the bad character shift rule, and the good suffix shift rule. Together, these ideas lead to a method that typically examines fewer than  $m + n$  characters (an expected sublinear-time method) and that (with a certain extension) runs in linear worst-case time. Our discussion of the Boyer–Moore algorithm, and extensions of it, concentrates on *provable* aspects of its behavior. Extensive experimental and practical studies of Boyer–Moore and variants have been reported in [229], [237], [409], [410], and [425].

#### 2.2.1. Right-to-left scan

For any alignment of  $P$  with  $T$  the Boyer–Moore algorithm checks for an occurrence of  $P$  by scanning characters from *right to left* rather than from left to right as in the naive

<sup>1</sup> Sedgewick [401] writes “Both the Knuth–Morris–Pratt and the Boyer–Moore algorithms require some complicated preprocessing on the pattern that is difficult to understand and has limited the extent to which they are used”. In agreement with Sedgewick, I still do not understand the original Boyer–Moore preprocessing method for the *strong* good suffix rule.

algorithm. For example, consider the alignment of  $P$  against  $T$  shown below:

```

          1           2
          12345678901234567
T:      xpbcttxabpqxctbpq
P:      tpabxab

```

To check whether  $P$  occurs in  $T$  at this position, the Boyer–Moore algorithm starts at the *right* end of  $P$ , first comparing  $T(9)$  with  $P(7)$ . Finding a match, it then compares  $T(8)$  with  $P(6)$ , etc., moving right to left until it finds a mismatch when comparing  $T(5)$  with  $P(3)$ . At that point  $P$  is shifted *right* relative to  $T$  (the amount for the shift will be discussed below) and the comparisons begin again at the right end of  $P$ .

Clearly, if  $P$  is shifted right by one place after each mismatch, or after an occurrence of  $P$  is found, then the worst-case running time of this approach is  $O(nm)$  just as in the naive algorithm. So at this point it isn't clear why comparing characters from right to left is any better than checking from left to right. However, with two additional ideas (the *bad character* and the *good suffix* rules), shifts of more than one position often occur, and in typical situations large shifts are common. We next examine these two ideas.

### 2.2.2. Bad character rule

To get the idea of the bad character rule, suppose that the last (right-most) character of  $P$  is  $y$  and the character in  $T$  it aligns with is  $x \neq y$ . When this initial mismatch occurs, if we know the right-most position in  $P$  of character  $x$ , we can safely shift  $P$  to the right so that the right-most  $x$  in  $P$  is below the mismatched  $x$  in  $T$ . Any shorter shift would only result in an immediate mismatch. Thus, the longer shift is correct (i.e., it will not shift past any occurrence of  $P$  in  $T$ ). Further, if  $x$  never occurs in  $P$ , then we can shift  $P$  completely past the point of mismatch in  $T$ . In these cases, some characters of  $T$  will never be examined and the method will actually run in “sublinear” time. This observation is formalized below.

**Definition** For each character  $x$  in the alphabet, let  $R(x)$  be the position of right-most occurrence of character  $x$  in  $P$ .  $R(x)$  is defined to be zero if  $x$  does not occur in  $P$ .

It is easy to preprocess  $P$  in  $O(n)$  time to collect the  $R(x)$  values, and we leave that as an exercise. Note that this preprocessing does not require the fundamental preprocessing discussed in Chapter 1 (that will be needed for the more complex shift rule, the good suffix rule).

We use the  $R$  values in the following way, called the *bad character shift rule*:

Suppose for a particular alignment of  $P$  against  $T$ , the right-most  $n - i$  characters of  $P$  match their counterparts in  $T$ , but the next character to the left,  $P(i)$ , mismatches with its counterpart, say in position  $k$  of  $T$ . The *bad character rule* says that  $P$  should be shifted right by  $\max[1, i - R(T(k))]$  places. That is, if the right-most occurrence in  $P$  of character  $T(k)$  is in position  $j < i$  (including the possibility that  $j = 0$ ), then shift  $P$  so that character  $j$  of  $P$  is below character  $k$  of  $T$ . Otherwise, shift  $P$  by one position.

The point of this shift rule is to shift  $P$  by more than one character when possible. In the above example,  $T(5) = t$  mismatches with  $P(3)$  and  $R(t) = 1$ , so  $P$  can be shifted right by two positions. After the shift, the comparison of  $P$  and  $T$  begins again at the right end of  $P$ .

### Extended bad character rule

The bad character rule is a useful heuristic for mismatches near the right end of  $P$ , but it has no effect if the mismatching character from  $T$  occurs in  $P$  to the right of the mismatch point. This may be common when the alphabet is small and the text contains many similar, but not exact, substrings. That situation is typical of DNA, which has an alphabet of size four, and even protein, which has an alphabet of size twenty, often contains different regions of high similarity. In such cases, the following *extended bad character rule* is more robust:

When a mismatch occurs at position  $i$  of  $P$  and the mismatched character in  $T$  is  $x$ , then shift  $P$  to the right so that the closest  $x$  to the left of position  $i$  in  $P$  is below the mismatched  $x$  in  $T$ .

Because the extended rule gives larger shifts, the only reason to prefer the simpler rule is to avoid the added implementation expense of the extended rule. The simpler rule uses only  $O(|\Sigma|)$  space ( $\Sigma$  is the alphabet) for array  $R$ , and one table lookup for each mismatch. As we will see, the extended rule can be implemented to take only  $O(n)$  space and at most one extra step per character comparison. That amount of added space is not often a critical issue, but it is an empirical question whether the longer shifts make up for the added time used by the extended rule. The original Boyer–Moore algorithm only uses the simpler bad character rule.

### Implementing the extended bad character rule

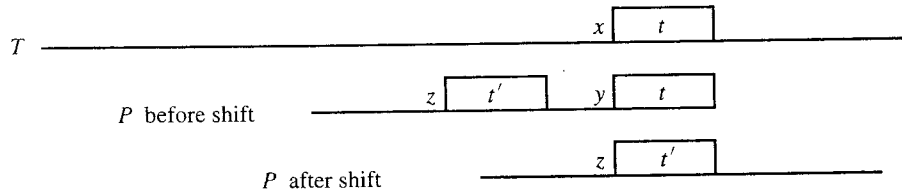
We preprocess  $P$  so that the extended bad character rule can be implemented efficiently in both time and space. The preprocessing should discover, for each position  $i$  in  $P$  and for each character  $x$  in the alphabet, the position of the closest occurrence of  $x$  in  $P$  to the left of  $i$ . The obvious approach is to use a two-dimensional array of size  $n$  by  $|\Sigma|$  to store this information. Then, when a mismatch occurs at position  $i$  of  $P$  and the mismatching character in  $T$  is  $x$ , we look up the  $(i, x)$  entry in the array. The lookup is fast, but the size of the array, and the time to build it, may be excessive. A better compromise, below, is possible.

During preprocessing, scan  $P$  from right to left collecting, for each character  $x$  in the alphabet, a list of the positions where  $x$  occurs in  $P$ . Since the scan is right to left, each list will be in decreasing order. For example, if  $P = abacbabc$  then the list for character  $a$  is 6, 3, 1. These lists are accumulated in  $O(n)$  time and of course take only  $O(n)$  space. During the search stage of the Boyer–Moore algorithm if there is a mismatch at position  $i$  of  $P$  and the mismatching character in  $T$  is  $x$ , scan  $x$ 's list from the top until we reach the first number less than  $i$  or discover there is none. If there is none then there is no occurrence of  $x$  before  $i$ , and all of  $P$  is shifted past the  $x$  in  $T$ . Otherwise, the found entry gives the desired position of  $x$ .

After a mismatch at position  $i$  of  $P$  the time to scan the list is at most  $n - i$ , which is roughly the number of characters that matched. So in worst case, this approach at most doubles the running time of the Boyer–Moore algorithm. However, in most problem settings the added time will be vastly less than double. One could also do binary search on the list in circumstances that warrant it.

### 2.2.3. The (strong) good suffix rule

The bad character rule by itself is reputed to be highly effective in practice, particularly for English text [229], but proves less effective for small alphabets and it does not lead to a linear worst-case running time. For that, we introduce another rule called the *strong*



**Figure 2.1:** Good suffix shift rule, where character  $x$  of  $T$  mismatches with character  $y$  of  $P$ . Characters  $y$  and  $z$  of  $P$  are guaranteed to be distinct by the good suffix rule, so  $z$  has a chance of matching  $x$ .

*good suffix rule.* The original preprocessing method [278] for the strong good suffix rule is generally considered quite difficult and somewhat mysterious (although a weaker version of it is easy to understand). In fact, the preprocessing for the strong rule was given incorrectly in [278] and corrected, without much explanation, in [384]. Code based on [384] is given without real explanation in the text by Baase [32], but there are no published sources that try to fully explain the method.<sup>2</sup> Pascal code for strong preprocessing, based on an outline by Richard Cole [107], is shown in Exercise 24 at the end of this chapter.

In contrast, the fundamental preprocessing of  $P$  discussed in Chapter 1 makes the needed preprocessing very simple. That is the approach we take here. The *strong good suffix rule* is:

Suppose for a given alignment of  $P$  and  $T$ , a substring  $t$  of  $T$  matches a suffix of  $P$ , but a mismatch occurs at the next comparison to the left. Then find, if it exists, the right-most copy  $t'$  of  $t$  in  $P$  such that  $t'$  is not a suffix of  $P$  and *the character to the left of  $t'$  in  $P$  differs from the character to the left of  $t$  in  $P$* . Shift  $P$  to the right so that substring  $t'$  in  $P$  is below substring  $t$  in  $T$  (see Figure 2.1). If  $t'$  does not exist, then shift the left end of  $P$  past the left end of  $t$  in  $T$  by the least amount so that a prefix of the shifted pattern matches a suffix of  $t$  in  $T$ . If no such shift is possible, then shift  $P$  by  $n$  places to the right. If an occurrence of  $P$  is found, then shift  $P$  by the least amount so that a *proper* prefix of the shifted  $P$  matches a suffix of the occurrence of  $P$  in  $T$ . If no such shift is possible, then shift  $P$  by  $n$  places, that is, shift  $P$  past  $t$  in  $T$ .

For a specific example consider the alignment of  $P$  and  $T$  given below:

```

      0         1
      123456789012345678
T:  prstabstuabvqxrst
      *
P:   qcabdabdab
      1234567890

```

When the mismatch occurs at position 8 of  $P$  and position 10 of  $T$ ,  $t = ab$  and  $t'$  occurs in  $P$  starting at position 3. Hence  $P$  is shifted right by six places, resulting in the following alignment:

<sup>2</sup> A recent plea appeared on the internet newsgroup comp. theory:

I am looking for an elegant (easily understandable) proof of correctness for a part of the Boyer-Moore string matching algorithm. The difficult-to-prove part here is the algorithm that computes the  $dd_2$  (good-suffix) table. I didn't find much of an understandable proof yet, so I'd much appreciate any help!

0	1
123456789012345678	
T: prstabst <sup>0</sup> ubabvq <sup>1</sup> xrst	
P:                   qcabdab <sup>1</sup> dab	

Note that the extended bad character rule would only have shifted  $P$  by only one place in this example.

**Theorem 2.2.1.** *The use of the good suffix rule never shifts  $P$  past an occurrence in  $T$ .*

**PROOF** Suppose the right end of  $P$  is aligned with character  $k$  of  $T$  before the shift, and suppose that the good suffix rule shifts  $P$  so its right end aligns with character  $k' > k$ . Any occurrence of  $P$  ending at a position  $l$  strictly between  $k$  and  $k'$  would immediately violate the selection rule for  $k'$ , since it would imply either that a closer copy of  $t$  occurs in  $P$  or that a longer prefix of  $P$  matches a suffix of  $t$ .  $\square$

The original published Boyer–Moore algorithm [75] uses a simpler, weaker, version of the good suffix rule. That version just requires that the shifted  $P$  agree with the  $t$  and does not specify that the next characters to the left of those occurrences of  $t$  be different. An explicit statement of the weaker rule can be obtained by deleting the italics phrase in the first paragraph of the statement of the strong good suffix rule. In the previous example, the weaker shift rule shifts  $P$  by three places rather than six. When we need to distinguish the two rules, we will call the simpler rule the *weak* good suffix rule and the rule stated above the *strong* good suffix rule. For the purpose of proving that the search part of Boyer–Moore runs in linear worst-case time, the weak rule is not sufficient, and in this book the strong version is assumed unless stated otherwise.

### 2.2.4. Preprocessing for the good suffix rule

We now formalize the preprocessing needed for the Boyer–Moore algorithm.

**Definition** For each  $i$ ,  $L(i)$  is the largest position less than  $n$  such that string  $P[i..n]$  matches a suffix of  $P[1..L(i)]$ .  $L(i)$  is defined to be zero if there is no position satisfying the conditions. For each  $i$ ,  $L'(i)$  is the largest position less than  $n$  such that string  $P[i..n]$  matches a suffix of  $P[1..L'(i)]$  and such that the character preceding that suffix is not equal to  $P(i-1)$ .  $L'(i)$  is defined to be zero if there is no position satisfying the conditions.

For example, if  $P = \overset{7}{c}\overset{6}{a}\overset{5}{b}\overset{4}{d}\overset{3}{a}\overset{2}{b}\overset{1}{d}\overset{0}{a}\overset{-1}{b}$ , then  $L(8) = 6$  and  $L'(8) = 3$ .

$L(i)$  gives the right end-position of the right-most copy of  $P[i..n]$  that is not a suffix of  $P$ , whereas  $L'(i)$  gives the right end-position of the right-most copy of  $P[i..n]$  that is not a suffix of  $P$ , with the stronger, added condition that its preceding character is unequal to  $P(i-1)$ . So, in the strong-shift version of the Boyer–Moore algorithm, if character  $i-1$  of  $P$  is involved in a mismatch and  $L'(i) > 0$ , then  $P$  is shifted right by  $n - L'(i)$  positions. The result is that if the right end of  $P$  was aligned with position  $k$  of  $T$  before the shift, then position  $L'(i)$  is now aligned with position  $k$ .

During the preprocessing stage of the Boyer–Moore algorithm  $L'(i)$  (and  $L(i)$ , if desired) will be computed for each position  $i$  in  $P$ . This is done in  $O(n)$  time via the following definition and theorem.

**Definition** For string  $P$ ,  $N_j(P)$  is the length of the longest *suffix* of the substring  $P[1..j]$  that is also a *suffix* of the full string  $P$ .

For example, if  $P = \overset{3}{c}\overset{6}{abdabdab}$ , then  $N_3(P) = 2$  and  $N_6(P) = 5$ .

Recall that  $Z_i(S)$  is the length of the longest substring of  $S$  that starts at  $i$  and matches a prefix of  $S$ . Clearly,  $N$  is the reverse of  $Z$ , that is, if  $P^r$  denotes the string obtained by reversing  $P$ , then  $N_j(P) = Z_{n-j+1}(P^r)$ . Hence the  $N_j(P)$  values can be obtained in  $O(n)$  time by using *Algorithm Z* on  $P^r$ . The following theorem is then immediate.

**Theorem 2.2.2.**  $L(i)$  is the largest index  $j$  less than  $n$  such that  $N_j(P) \geq |P[i..n]|$  (which is  $n-i+1$ ).  $L'(i)$  is the largest index  $j$  less than  $n$  such that  $N_j(P) = |P[i..n]| = (n-i+1)$ .

Given Theorem 2.2.2, it follows immediately that all the  $L'(i)$  values can be accumulated in linear time from the  $N$  values using the following algorithm:

### Z-based Boyer-Moore

```

for  $i := 1$  to  $n$  do  $L'(i) := 0$ ;
for  $j := 1$  to  $n - 1$  do
  begin
     $i := n - N_j(P) + 1$ ;
     $L'(i) := j$ ;
  end;

```

The  $L(i)$  values (if desired) can be obtained by adding the following lines to the above pseudocode:

```

 $L(2) := L'(2)$ ;
for  $i := 3$  to  $n$  do  $L(i) := \max[L(i-1), L'(i)]$ ;

```

**Theorem 2.2.3.** *The above method correctly computes the  $L$  values.*

**PROOF**  $L(i)$  marks the right end-position of the right-most substring of  $P$  that matches  $P[i..n]$  and is not a suffix of  $P[1..n]$ . Therefore, that substring begins at position  $L(i)-n+i$ , which we will denote by  $j$ . We will prove that  $L(i) = \max[L(i-1), L'(i)]$  by considering what character  $j-1$  is. First, if  $j=1$  then character  $j-1$  doesn't exist, so  $L(i-1) = 0$  and  $L'(i) = 1$ . So suppose that  $j > 1$ . If character  $j-1$  equals character  $i-1$  then  $L(i) = L(i-1)$ . If character  $j-1$  does not equal character  $i-1$  then  $L(i) = L'(i)$ . Thus, in all cases,  $L(i)$  must either be  $L'(i)$  or  $L(i-1)$ .

However,  $L(i)$  must certainly be greater than or equal to both  $L'(i)$  and  $L(i-1)$ . In summary,  $L(i)$  must either be  $L'(i)$  or  $L(i-1)$ , and yet it must be greater or equal to both of them; hence  $L(i)$  must be the maximum of  $L'(i)$  and  $L(i-1)$ .  $\square$

### Final preprocessing detail

The preprocessing stage must also prepare for the case when  $L'(i) = 0$  or when an occurrence of  $P$  is found. The following definition and theorem accomplish that.

**Definition** Let  $l'(i)$  denote the length of the largest suffix of  $P[i..n]$  that is also a prefix of  $P$ , if one exists. If none exists, then let  $l'(i)$  be zero.

**Theorem 2.2.4.**  $l'(i)$  equals the largest  $j \leq |P[i..n]|$ , which is  $n-i+1$ , such that  $N_j(P) = j$ .

We leave the proof, as well as the problem of how to accumulate the  $l'(i)$  values in linear time, as a simple exercise. (Exercise 9 of this chapter)

### 2.2.5. The good suffix rule in the search stage of Boyer–Moore

Having computed  $L'(i)$  and  $l'(i)$  for each position  $i$  in  $P$ , these preprocessed values are used during the search stage of the algorithm to achieve larger shifts. If, during the search stage, a mismatch occurs at position  $i - 1$  of  $P$  and  $L'(i) > 0$ , then the good suffix rule shifts  $P$  by  $n - L'(i)$  places to the right, so that the  $L'(i)$ -length prefix of the shifted  $P$  aligns with the  $L'(i)$ -length suffix of the unshifted  $P$ . In the case that  $L'(i) = 0$ , the good suffix rule shifts  $P$  by  $n - l'(i)$  places. When an occurrence of  $P$  is found, then the rule shifts  $P$  by  $n - l'(2)$  places. Note that the rules work correctly even when  $l'(i) = 0$ .

One special case remains. When the first comparison is a mismatch (i.e.,  $P(n)$  mismatches) then  $P$  should be shifted one place to the right.

### 2.2.6. The complete Boyer–Moore algorithm

We have argued that neither the good suffix rule nor the bad character rule shift  $P$  so far as to miss any occurrence of  $P$ . So the Boyer–Moore algorithm shifts by the largest amount given by either of the rules. We can now present the complete algorithm.

#### The Boyer–Moore algorithm

{Preprocessing stage}

Given the pattern  $P$ ,

Compute  $L'(i)$  and  $l'(i)$  for each position  $i$  of  $P$ ,

and compute  $R(x)$  for each character  $x \in \Sigma$ .

{Search stage}

$k := n$ ;

while  $k \leq m$  do

begin

$i := n$ ;

$h := k$ ;

while  $i > 0$  and  $P(i) = T(h)$  do

begin

$i := i - 1$ ;

$h := h - 1$ ;

end;

if  $i = 0$  then

begin

report an occurrence of  $P$  in  $T$  ending at position  $k$ .

$k := k + n - l'(2)$ ;

end

else

shift  $P$  (increase  $k$ ) by the maximum amount determined by the (extended) bad character rule and the good suffix rule.

end;

Note that although we have always talked about “shifting  $P$ ”, and given rules to determine by how much  $P$  should be “shifted”, there is no shifting in the actual implementation. Rather, the index  $k$  is increased to the point where the right end of  $P$  would be “shifted”. Hence, each act of shifting  $P$  takes constant time.

We will later show, in Section 3.2, that by using the strong good suffix rule alone, the

Boyer–Moore method has a worst-case running time of  $O(m)$  provided that the pattern does not appear in the text. This was first proved by Knuth, Morris, and Pratt [278], and an alternate proof was given by Guibas and Odlyzko [196]. Both of these proofs were quite difficult and established worst-case time bounds no better than  $5m$  comparisons. Later, Richard Cole gave a much simpler proof [108] establishing a bound of  $4m$  comparisons and also gave a difficult proof establishing a tight bound of  $3m$  comparisons. We will present Cole’s proof of  $4m$  comparisons in Section 3.2.

When the pattern does appear in the text then the original Boyer–Moore method runs in  $\Theta(nm)$  worst-case time. However, several simple modifications to the method correct this problem, yielding an  $O(m)$  time bound in all cases. The first of these modifications was due to Galil [168]. After discussing Cole’s proof, in Section 3.2, for the case that  $P$  doesn’t occur in  $T$ , we use a variant of Galil’s idea to achieve the linear time bound in all cases.

At the other extreme, if we only use the bad character shift rule, then the worst-case running time is  $O(nm)$ , but assuming randomly generated strings, the expected running time is sublinear. Moreover, in typical string matching applications involving natural language text, a sublinear running time is almost always observed in practice. We won’t discuss random string analysis in this book but refer the reader to [184].

Although Cole’s proof for the linear worst case is vastly simpler than earlier proofs, and is important in order to complete the full story of Boyer–Moore, it is not trivial. However, a fairly simple extension of the Boyer–Moore algorithm, due to Apostolico and Giancarlo [26], gives a “Boyer–Moore–like” algorithm that allows a fairly direct proof of a  $2m$  worst-case bound on the number of comparisons. The Apostolico–Giancarlo variant of Boyer–Moore is discussed in Section 3.1.

## 2.3. The Knuth-Morris-Pratt algorithm

The best known linear-time algorithm for the exact matching problem is due to Knuth, Morris, and Pratt [278]. Although it is rarely the method of choice, and is often much inferior in practice to the Boyer–Moore method (and others), it can be simply explained, and its linear time bound is (fairly) easily proved. The algorithm also forms the basis of the well-known Aho–Corasick algorithm, which efficiently finds all occurrences in a text of any pattern from a *set* of patterns.<sup>3</sup>

### 2.3.1. The Knuth-Morris-Pratt shift idea

For a given alignment of  $P$  with  $T$ , suppose the naive algorithm matches the first  $i$  characters of  $P$  against their counterparts in  $T$  and then mismatches on the next comparison. The naive algorithm would shift  $P$  by just *one* place and begin comparing again from the left end of  $P$ . But a larger shift may often be possible. For example, if  $P = abcxabcde$  and, in the present alignment of  $P$  with  $T$ , the mismatch occurs in position 8 of  $P$ , then it is easily deduced (and we will prove below) that  $P$  can be shifted by four places without passing over any occurrences of  $P$  in  $T$ . Notice that this can be deduced without even knowing what string  $T$  is or exactly how  $P$  is aligned with  $T$ . Only the location of the mismatch in  $P$  must be known. The Knuth-Morris-Pratt algorithm is based on this kind of reasoning to make larger shifts than the naive algorithm makes. We now formalize this idea.

<sup>3</sup> We will present several solutions to that set problem including the Aho–Corasick method in Section 3.4. For those reasons, and for its historical role in the field, we fully develop the Knuth-Morris-Pratt method here.



**Definition** For each position  $i$  in pattern  $P$ , define  $sp_i(P)$  to be the *length* of the longest proper *suffix* of  $P[1..i]$  that matches a prefix of  $P$ .

Stated differently,  $sp_i(P)$  is the length of the longest proper substring of  $P[1..i]$  that ends at  $i$  and that matches a prefix of  $P$ . When the string is clear by context we will use  $sp_i$  in place of the full notation.

For example, if  $P = abcaebcabd$ , then  $sp_2 = sp_3 = 0$ ,  $sp_4 = 1$ ,  $sp_8 = 3$ , and  $sp_{10} = 2$ . Note that by definition,  $sp_1 = 0$  for any string.

An optimized version of the Knuth-Morris-Pratt algorithm uses the following values.

**Definition** For each position  $i$  in pattern  $P$ , define  $sp'_i(P)$  to be the length of the longest proper suffix of  $P[1..i]$  that matches a prefix of  $P$ , with the added condition that characters  $P(i+1)$  and  $P(sp'_i+1)$  are unequal.

Clearly,  $sp'_i(P) \leq sp_i(P)$  for all positions  $i$  and any string  $P$ . As an example, if  $P = bbccaebcabd$ , then  $sp_8 = 2$  because string  $bb$  occurs both as a proper prefix of  $P[1..8]$  and as a suffix of  $P[1..8]$ . However, both copies of the string are followed by the same character  $c$ , and so  $sp'_8 < 2$ . In fact,  $sp'_8 = 1$  since the single character  $b$  occurs as both the first and last character of  $P[1..8]$  and is followed by character  $b$  in position 2 and by character  $c$  in position 9.

### The Knuth-Morris-Pratt shift rule

We will describe the algorithm in terms of the  $sp'$  values, and leave it to the reader to modify the algorithm if only the weaker  $sp$  values are used.<sup>4</sup> The Knuth-Morris-Pratt algorithm aligns  $P$  with  $T$  and then compares the aligned characters from *left to right*, as the naive algorithm does.

For any alignment of  $P$  and  $T$ , if the first mismatch (comparing from left to right) occurs in position  $i+1$  of  $P$  and position  $k$  of  $T$ , then shift  $P$  to the right (relative to  $T$ ) so that  $P[1..sp'_i]$  aligns with  $T[k-sp'_i..k-1]$ . In other words, shift  $P$  exactly  $i+1-(sp'_i+1) = i-sp'_i$  places to the right, so that character  $sp'_i+1$  of  $P$  will align with character  $k$  of  $T$ . In the case that an occurrence of  $P$  has been found (no mismatch), shift  $P$  by  $n-sp'_n$  places.

The shift rule guarantees that the prefix  $P[1..sp'_i]$  of the shifted  $P$  matches its opposing substring in  $T$ . The next comparison is then made between characters  $T(k)$  and  $P[sp'_i+1]$ . The use of the stronger shift rule based on  $sp'_i$  guarantees that the same mismatch will not occur again in the new alignment, but it does not guarantee that  $T(k) = P[sp'_i+1]$ .

In the above example, where  $P = abcxabcde$  and  $sp'_7 = 3$ , if character 8 of  $P$  mismatches then  $P$  will be shifted by  $7-3 = 4$  places. This is true even without knowing  $T$  or how  $P$  is positioned with  $T$ .

The advantage of the shift rule is twofold. First, it often shifts  $P$  by more than just a single character. Second, after a shift, the left-most  $sp'_i$  characters of  $P$  are guaranteed to match their counterparts in  $T$ . Thus, to determine whether the newly shifted  $P$  matches its counterpart in  $T$ , the algorithm can start comparing  $P$  and  $T$  at position  $sp'_i+1$  of  $P$  (and position  $k$  of  $T$ ). For example, suppose  $P = abcxabcde$  as above,  $T = xyabcbxabcxadcdqfeg$ , and the left end of  $P$  is aligned with character 3 of  $T$ . Then  $P$  and  $T$  will match for 7 characters but mismatch on character 8 of  $P$ , and  $P$  will be shifted

<sup>4</sup> The reader should be alerted that traditionally the Knuth-Morris-Pratt algorithm has been described in terms of *failure functions*, which are related to the  $sp_i$  values. Failure functions will be explicitly defined in Section 2.3.3.

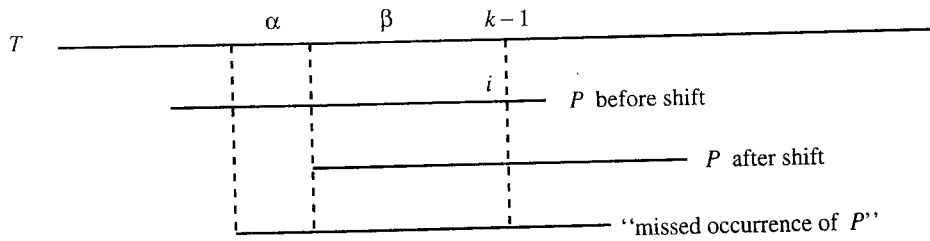


Figure 2.2: Assumed missed occurrence used in correctness proof for Knuth-Morris-Pratt.

by 4 places as shown below:

```

1           2
123456789012345678
xyabcxabcxadcdqfeg
  abcxabcde
    abcxabcde
    
```

As guaranteed, the first 3 characters of the shifted  $P$  match their counterparts in  $T$  (and their counterparts in the unshifted  $P$ ).

Summarizing, we have

**Theorem 2.3.1.** *After a mismatch at position  $i + 1$  of  $P$  and a shift of  $i - sp'_i$  places to the right, the left-most  $sp'_i$  characters of  $P$  are guaranteed to match their counterparts in  $T$ .*

Theorem 2.3.1 partially establishes the correctness of the Knuth-Morris-Pratt algorithm, but to fully prove correctness we have to show that the shift rule never shifts too far. That is, using the shift rule no occurrence of  $P$  will ever be overlooked.

**Theorem 2.3.2.** *For any alignment of  $P$  with  $T$ , if characters 1 through  $i$  of  $P$  match the opposing characters of  $T$  but character  $i + 1$  mismatches  $T(k)$ , then  $P$  can be shifted by  $i - sp'_i$  places to the right without passing any occurrence of  $P$  in  $T$ .*

**PROOF** Suppose not, so that there is an occurrence of  $P$  starting strictly to the left of the shifted  $P$  (see Figure 2.2), and let  $\alpha$  and  $\beta$  be the substrings shown in the figure. In particular,  $\beta$  is the prefix of  $P$  of length  $sp'_i$ , shown relative to the shifted position of  $P$ . The unshifted  $P$  matches  $T$  up through position  $i$  of  $P$  and position  $k - 1$  of  $T$ , and all characters in the (assumed) missed occurrence of  $P$  match their counterparts in  $T$ . Both of these matched regions contain the substrings  $\alpha$  and  $\beta$ , so the unshifted  $P$  and the assumed occurrence of  $P$  match on the entire substring  $\alpha\beta$ . Hence  $\alpha\beta$  is a suffix of  $P[1..i]$  that matches a proper prefix of  $P$ . Now let  $l = |\alpha\beta| + 1$  so that position  $l$  in the “missed occurrence” of  $P$  is opposite position  $k$  in  $T$ . Character  $P(l)$  cannot be equal to  $P(i + 1)$  since  $P(l)$  is assumed to match  $T(k)$  and  $P(i + 1)$  does not match  $T(k)$ . Thus  $\alpha\beta$  is a proper suffix of  $P[1..i]$  that matches a prefix of  $P$ , and the next character is unequal to  $P(i + 1)$ . But  $|\alpha| > 0$  due to the assumption that an occurrence of  $P$  starts strictly before the shifted  $P$ , so  $|\alpha\beta| > |\beta| = sp'_i$ , contradicting the definition of  $sp'_i$ . Hence the theorem is proved.  $\square$

Theorem 2.3.2 says that the Knuth-Morris-Pratt shift rule does not miss any occurrence of  $P$  in  $T$ , and so the Knuth-Morris-Pratt algorithm will correctly find all occurrences of  $P$  in  $T$ . The time analysis is equally simple.

× **Theorem 2.3.3.** *In the Knuth-Morris-Pratt method, the number of character comparisons is at most  $2m$ .*

**PROOF** Divide the algorithm into compare/shift phases, where a single phase consists of the comparisons done between successive shifts. After any shift, the comparisons in the phase go left to right and start either with the last character of  $T$  compared in the previous phase or with the character to its right. Since  $P$  is never shifted left, in any phase at most one comparison involves a character of  $T$  that was previously compared. Thus, the total number of character comparisons is bounded by  $m + s$ , where  $s$  is the number of shifts done in the algorithm. But  $s < m$  since after  $m$  shifts the right end of  $P$  is certainly to the right of the right end of  $T$ , so the number of comparisons done is bounded by  $2m$ .  $\square$

### 2.3.2. Preprocessing for Knuth-Morris-Pratt

The key to the speed up of the Knuth-Morris-Pratt algorithm over the naive algorithm is the use of  $sp'$  (or  $sp$ ) values. It is easy to see how to compute all the  $sp'$  and  $sp$  values from the  $Z$  values obtained during the fundamental preprocessing of  $P$ . We verify this below.

**Definition** Position  $j > 1$  maps to  $i$  if  $i = j + Z_j(P) - 1$ . That is,  $j$  maps to  $i$  if  $i$  is the right end of a  $Z$ -box starting at  $j$ .

**Theorem 2.3.4.** *For any  $i > 1$ ,  $sp'_i(P) = Z_j = i - j + 1$ , where  $j > 1$  is the smallest position that maps to  $i$ . If there is no such  $j$  then  $sp'_i(P) = 0$ . For any  $i > 1$ ,  $sp_i(P) = i - j + 1$ , where  $j$  is the smallest position in the range  $1 < j \leq i$  that maps to  $i$  or beyond. If there is no such  $j$ , then  $sp_i(P) = 0$ .*

**PROOF** If  $sp'_i(P)$  is greater than zero, then there is a proper suffix  $\alpha$  of  $P[1..i]$  that matches a prefix of  $P$ , such that  $P[i + 1]$  does not match  $P[|\alpha| + 1]$ . Therefore, letting  $j$  denote the start of  $\alpha$ ,  $Z_j = |\alpha| = sp'_i(P)$  and  $j$  maps to  $i$ . Hence, if there is no  $j$  in the range  $1 < j \leq i$  that maps to  $i$ , then  $sp'_i(P)$  must be zero.

Now suppose  $sp'_i(P) > 0$  and let  $j$  be as defined above. We claim that  $j$  is the smallest position in the range 2 to  $i$  that maps to  $i$ . Suppose not, and let  $j^*$  be a position in the range  $1 < j^* < j$  that maps to  $i$ . Then  $P[j^*..i]$  would be a proper suffix of  $P[1..i]$  that matches a prefix (call it  $\beta$ ) of  $P$ . Moreover, by the definition of mapping,  $P(i + 1) \neq P(|\beta|)$ , so  $sp'_i(P) \geq |\beta| > |\alpha|$ , contradicting the assumption that  $sp'_i = \alpha$ .

The proofs of the claims for  $sp_i(P)$  are similar and are left as exercises.  $\square$

Given Theorem 2.3.4, all the  $sp'$  and  $sp$  values can be computed in linear time using the  $Z_i$  values as follows:

#### Z-based Knuth-Morris-Pratt

```

for  $i := 1$  to  $n$  do
     $sp'_i := 0$ ;
for  $j := n$  downto 2 do
begin
     $i := j + Z_j(P) - 1$ ;
     $sp'_i := Z_j$ ;
end;
```

The  $sp$  values are obtained by adding the following:

```

 $sp_n(P) := sp'_n(P);$ 
for  $i := n - 1$  downto 2 do
     $sp_i(P) := \max[sp_{i+1}(P) - 1, sp'_i(P)]$ 

```

### 2.3.3. A full implementation of Knuth-Morris-Pratt

We have described the Knuth-Morris-Pratt algorithm in terms of shifting  $P$ , but we never accounted for time needed to implement shifts. The reason is that shifting is only conceptual and  $P$  is never explicitly shifted. Rather, as in the case of Boyer-Moore, pointers to  $P$  and  $T$  are incremented. We use pointer  $p$  to point into  $P$  and one pointer  $c$  (for “current” character) to point into  $T$ .

**Definition** For each position  $i$  from 1 to  $n + 1$ , define the failure function  $F'(i)$  to be  $sp'_{i-1} + 1$  (and define  $F'(i) = sp_{i-1} + 1$ ), where  $sp'_0$  and  $sp_0$  are defined to be zero.

We will only use the (stronger) failure function  $F'(i)$  in this discussion but will refer to  $F(i)$  later.

After a mismatch in position  $i + 1 > 1$  of  $P$ , the Knuth-Morris-Pratt algorithm “shifts”  $P$  so that the next comparison is between the character in position  $c$  of  $T$  and the character in position  $sp'_i + 1$  of  $P$ . But  $sp'_i + 1 = F'(i + 1)$ , so a general “shift” can be implemented in constant time by just setting  $p$  to  $F'(i + 1)$ . Two special cases remain. When the mismatch occurs in position 1 of  $P$ , then  $p$  is set to  $F'(1) = 1$  and  $c$  is incremented by one. When an occurrence of  $P$  is found, then  $P$  is shifted right by  $n - sp'_n$  places. This is implemented by setting  $F'(n + 1)$  to  $sp'_n + 1$ .

Putting all the pieces together gives the full Knuth-Morris-Pratt algorithm.

#### Knuth-Morris-Pratt algorithm

```

begin
Preprocess  $P$  to find  $F'(k) = sp'_{k-1} + 1$  for  $k$  from 1 to  $n + 1$ .
     $c := 1;$ 
     $p := 1;$ 
    While  $c + (n - p) \leq m$ 
    do begin
        While  $P(p) = T(c)$  and  $p \leq n$ 
        do begin
             $p := p + 1;$ 
             $c := c + 1;$ 
        end;
        if  $p = n + 1$  then
            report an occurrence of  $P$  starting at position  $c - n$  of  $T$ .
        if  $p := 1$  then  $c := c + 1$ 
         $p := F'(p);$ 
    end;
end.

```

## 2.4. Real-time string matching

In the search stage of the Knuth-Morris-Pratt algorithm,  $P$  is aligned against a substring of  $T$  and the two strings are compared left to right until either all of  $P$  is exhausted (in which