The Knuth-Morris-Pratt algorithm

We now present a linear-time string-matching algorithm due to Knuth, Morris, and Pratt. Their algorithm achieves a $\Theta(n + m)$ running time by avoiding the computation of the transition function δ altogether, and it does the pattern matching using just an auxiliary function $\pi[1 . . m]$ precomputed from the pattern in time $O(m)$. The array π allows the transition function δ to be computed efficiently (in an amortized sense) "on the fly" as needed. Roughly speaking, for any state $q = 0, 1, . . . , m$,and any character $a \in \Sigma$, the value $\pi[q]$ contains the information that is independent of a and is needed to compute $\delta(q, a)$. (This remark will be clarified shortly.) Since the array π has only m entries, whereas δ has $O(m|\Sigma|)$ entries, we save a factor of Σ in the preprocessing by computing π rather than δ.

The prefix function for a pattern

The prefix function for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. This information can be used to avoid testing useless shifts in the naive pattern-matching algorithm or to avoid the precomputation of δ for a string-matching automaton.

Consider the operation of the naive string matcher. Figure 34.9(a) shows a particular shift s of a template containing the pattern P = ababaca against a text T. For this example, $q = 5$ of the characters have matched successfully, but the 6th pattern character fails to match the corresponding text character. The information that q characters have matched successfully determines the corresponding text characters. Knowing these q text characters allows us to determine immediately that certain shifts are invalid. In the example of the figure, the shift $s + 1$ is necessarily invalid, since the first pattern character, an a, would be aligned with a text character that is known to match with the second pattern character, a b. The shift $s + 2$ shown in part (b) of the figure, however, aligns the first three pattern characters with three text characters that must necessarily match. In general, it is useful to know the answer to the following question:

Given that pattern characters $P[1 . . q]$ match text characters $T[s + 1 . . s + q]$, what is the least shift $s' > s$ such that

$$P[1 . . k] = T[s' 1 . . s' k],$$

(34.5)

where $s' + k = s + q$?

Such a shift s' is the first shift greater than s that is not necessarily invalid due to our knowledge of $T[s + 1 . . s + q]$. In the best case, we have that $s' = s + q$, and

shifts $s + 1, s + 2, \ldots, s + q - 1$ are all immediately ruled out. In any case, at the new shift s' we don't need to compare the first k characters of P with the corresponding characters of T, since we are guaranteed that they match by equation (34.5).

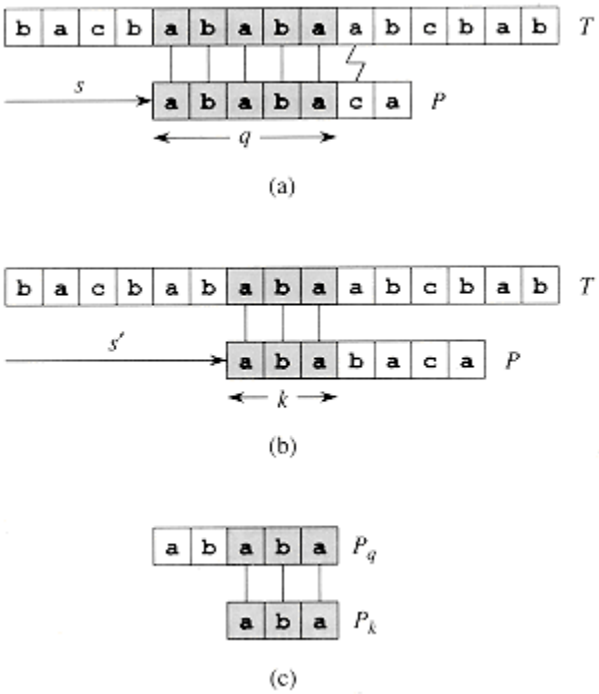


Figure 34.9 The prefix function π. (a) The pattern P = ababaca is aligned with a text T so that the first q = 5 characters match. Matching characters, shown shaded, are connected by vertical lines. (b) Using only our knowledge of the 5 matched characters, we can deduce that a shift of s + 1 is invalid, but that a shift of s' = s + 2 is consistent with everything we know about the text and therefore is potentially valid. (c) The useful information for such deductions can be precomputed by comparing the pattern with itself. Here, we see that the longest prefix of P that is also a suffix of P_5 is P_3. This infomation is precomputed and represented in the array π, so that $\pi[5] = 3$. Given that q characters have matched successfully at shift s, the next potentially valid shift is at s' = s + (q - π[q]).

The necessary information can be precomputed by comparing the pattern against itself, as illustrated in Figure 34.9(*c*). Since $T[s' + 1 \ldots s' + k]$ is part of the known portion of the text, it is a suffix of the string P_q. Equation (34.5) can therefore be interpreted as asking for the largest $k < q$ such that $P_k \supset P_q$. Then, $s' = s + (q - k)$ is the next potentially valid shift. It turns out to be convenient to store the number k of matching characters at the new shift **s**′, rather than storing, say, $s' - s$. This information can be used to speed up both the naive string-matching algorithm and the finite-automaton matcher.

We formalize the precomputation required as follows. Given a pattern $P[1 \ldots m]$, the ***prefix function*** for the pattern P is the function $\pi : \{1, 2, \ldots, m\} \to \{0, 1, \ldots, m-1\}$ such that

$$\pi[q] = \max \{k : k < q \text{ and } P_k \sqsupset P_q\} .$$

That is, $\pi[q]$ is the length of the longest prefix of P that is a proper suffix of P_q. As another example, Figure 34.10(a) gives the complete prefix function π for the pattern `abababababca`.

The Knuth-Morris-Pratt matching algorithm is given in pseudocode below as the procedure KMP-MATCHER. It is mostly modeled after FINITE-AUTOMATON-MATCHER, as we shall see. KMP-MATCHER calls the auxiliary procedure COMPUTE-PREFIX-FUNCTION to compute π.

```
KMP-MATCHER(T, P)
1 n ← length[T]
2 m ← length[P]
3 Π ← COMPUTE-PREFIX-FUNCTION(P)
4 q ← 0
5 for i ← 1 to n
6       do while q > 0 and P[q + 1] ≠ T[i]
7             do q ← Π[q]
8          if P[q + 1] = T[i]
9             then q ← q + 1
10         if q = m
11            then print "Pattern occurs with shift" i - m
12                 q ← Π[q]
COMPUTE-PREFIX-FUNCTION(P)
1 m ← length[P]
2 Π [1] ← 0
3 k ← 0
4 for q ← 2 to m
5       do while k > 0 and P[k + 1] ≠ P[q]
6             do k ← Π[k]
7          if P[k + 1] = P[q]
8             then k ← k + 1
9          Π[q] ← k
10 return Π
```

We begin with an analysis of the running times of these procedures. Proving these procedures correct will be more complicated.

Running-time analysis

The running time of COMPUTE-PREFIX-FUNCTION is $O(m)$, using an amortized analysis (see Chapter 18). We associate a potential of k with the current state k of the algorithm. This potential has an initial value of 0, by line 3. Line 6 decreases k whenever it is executed, since $\pi[k] < k$. Since $\pi[k] \geq 0$ for all k, however, k can never become negative. The only other line that affects k is line 8, which increases k by at most one during each execution of the **for** loop body.

Since $k < q$ upon entering the **for** loop, and since q is incremented in each iteration of the **for** loop body, $k < q$ always holds. (This justifies the claim that $\pi[q] < q$ as well, by line 9.) We can pay for each execution of the **while** loop body on line 6 with the corresponding decrease in the potential function, since $\pi[k] < k$. Line 8 increases the potential function by at most one, so that the amortized cost of the loop body on lines 5-9 is $O(1)$. Since the number of outer-loop iterations is $O(m)$, and since the final potential function is at least as great as the initial potential function, the total actual worst-case running time of COMPUTE-PREFIX-FUNCTION is $O(m)$.

The Knuth-Morris-Pratt algorithm runs in time $O(m + n)$. The call of COMPUTE-PREFIX-FUNCTION takes $O(m)$ time as we have just seen, and a similar amortized analysis, using the value of q as the potential function, shows that the remainder of KMP-MATCHER takes $O(n)$ time.

Compared to FINITE-AUTOMATON-MATCHER, by using π rather than δ, we have reduced the time for preprocessing the pattern from $O(m\,|\Sigma|)$ to $O(m)$, while keeping the actual matching time bounded by $O(m + n)$.

Correctness of the prefix-function computation

We start with an essential lemma showing that by iterating the prefix function π, we can enumerate all the prefixes P_k that are suffixes of a given prefix P_q. Let

$$\pi^*[q] = \{q, \pi[q], \pi^2[q], \pi^3[q], \ldots, \pi^t[q]\},$$

where $\pi^i[q]$ is defined in terms of functional composition, so that $\pi^0[q] = q$ and $\pi^{i+1}[q] = \pi[\pi^i[q]]$ for $i > 1$, and where it is understood that the sequence in $\pi^*[q]$ stops when $\pi^t[q] = 0$ is reached.

Lemma 34.5

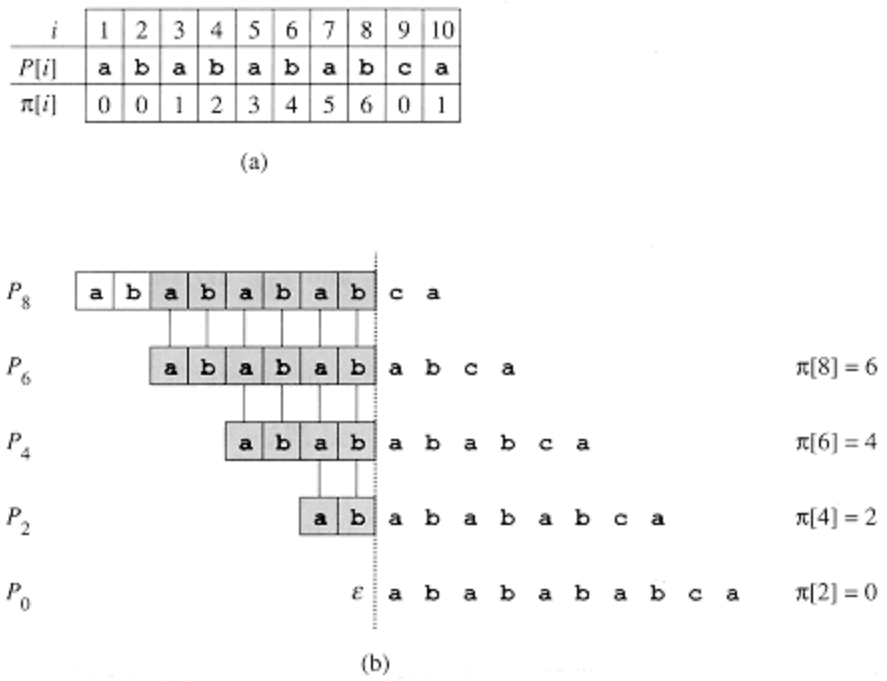
Let P be a pattern of length m with prefix function π. Then, for $q = 1, 2, \ldots, m$, we have $\pi^*[q] = \{k : P_k \supset P_q\}$.

Proof We first prove that

$$i \in \pi^*[q] \text{ implies } P_i \supset P_q . \tag{34.6}$$

If $i \in \pi^*[q]$, then $i = \pi^u[q]$ for some u. We prove equation (34.6) by induction on u. For $u = 0$, we have $i = q$, and the claim follows since $P_q \supset P_q$. Using the

relation $P_{\pi[i]} \sqsupset P_i$ and the transitivity of $\sqsupset$ establishes the claim for all i in $\Pi^*[q]$. Therefore, $\pi^*[q] \subseteq \{k : P_k \sqsupset P_q\}$.



i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

(a)

(b)

Figure 34.10 An illustration of Lemma 34.5 for the pattern P = abababacba and q = 8. (a) The Π function for the given pattern. Since $\Pi[8] = 6$, $\Pi[6] = 4$, $\Pi[4] = 2$, and $\Pi[2] = 0$, by iterating Π we obtain $\Pi^*[8] = \{8, 6, 4, 2, 0\}$. (b) We slide the template containing the pattern P to the right and note when some prefix P_k of P matches up with some proper suffix of P_8; this happens for k = 6, 4, 2, and 0. In the figure, the first row gives P, and the dotted vertical line is drawn just after P_8 Successive rows show all the shifts of P that cause some prefix P_k of P to match some suffix of P_8. Successfully matched characters are shown shaded. Vertical lines connect aligned matching characters. Thus, $\{k : P_k \sqsupset P_q\} = \{8, 6, 4, 2, 0\}$. The lemma claims that $\pi^*[q] = \{k : P_k \sqsupset P_q\}$ for all q.

We prove that $\{k : P_k \sqsupset P_q\} \subseteq \pi^*[q]$ by contradiction. Suppose to the contrary that there is an integer in the set $\{k : P_k \sqsupset P_q\} - \pi^*[q]$, and let j be the largest such value. Because q is in $\{k : P_k \sqsupset P_q\} \cap \pi^*[q]$, we have $j < q$, and so we let j' denote the smallest integer in $\Pi^*[q]$ that is greater than j. (We can choose $j' = q$ if there is no other number in $\Pi^*[q]$ that is greater than j.) We have $P_j \sqsupset P_q$ because $j \in \{k : P_k \sqsupset P_q\}$, $P_{j'} \sqsupset P_q$ because $j' \in \Pi^*[q]$; thus, $P_j \sqsupset P_{j'}$ by Lemma 34.1. Moreover, j is the largest such value with this

property. Therefore, we must have $\pi[j'] = j$ and thus $j \in \pi^*[q]$. This contradiction proves the lemma.

Figure 34.10 illustrates this lemma.

The algorithm COMPUTE-PREFIX-FUNCTION computes $\pi[q]$ in order for $q = 1, 2, \ldots, m$. The computation of $\pi[1] = 0$ in line 2 of COMPUTE-PREFIX-FUNCTION is certainly correct, since $\pi[q] < q$ for all q. The following lemma and its corollary will be used to prove that COMPUTE-PREFIX-FUNCTION computes $\pi[q]$ correctly for $q > 1$.

Lemma 34.6

Let P be a pattern of length m, and let π *be the prefix function for* P. *For* q = *1, 2,* . . . , m, *if* $\pi[q] > 0$, then π[q*] - 1* $\in \pi^*$[q - *1].*

Proof If $k = \pi[q] > 0$, then $P_k \sqsupset P_q$, and thus $P_{k-1} \sqsupset P_{q-1}$ (by dropping the last character from P_k and P_q). By Lemma 34.5, therefore, $k - 1 \in \pi^*[q - 1]$.

For $q = 2, 3, \ldots, m$, define the subset $E_q\text{-}1 \subseteq \pi^*[q - 1]$ by

$$E_q\text{-}1 = \{k : k \in \pi^*[q - 1] \text{ and } P[k + 1] = P[q]\} .$$

The set E_q-1 consists of the values k for which $P_k \sqsupset P_{q-1}$ (by Lemma 34.5); because $P[k + 1] = P[q]$, it is also the case that for these values of k, $P_{k+1} \sqsupset P_q$. Intuitively, $E_q - 1$ consists of those values $k \in \pi^*[q - 1]$ such that we can extend P_k to P_k+1 and get a suffix of P_q.

Corollary 34.7

Let P be a pattern of length m, and let π be the prefix function for P. For $q = 2, 3, \ldots, m$,

$$\pi[q] = \begin{cases} 0 & \text{if } E_{q-1} = \emptyset , \\ 1 + \max\{k \in E_{q-1}\} & \text{if } E_{q-1} \neq \emptyset . \end{cases}$$

Proof If $r = \pi[q]$, then $P_r \sqsupset P_q$, and so $r \geq 1$ implies $P[r] = P[q]$. By Lemma 34.6, therefore, if $r \geq 1$, then

$$r = 1 + \max \{k \in \pi^*[q - 1]: P[k + 1] = P[q]\} .$$

But the set maximized over is just E_q-1, so that $r = 1 + \max \{k \in E_q\text{-}1\}$ and E_q-1 is nonempty. If $r = 0$, there is no $k \in \pi^*[q - 1]$ for which we can

extend P_k to P_{k+1} and get a suffix of P_q, since then we would have $\pi[q] > 0$. Thus, $E_{q-1} = \emptyset$.

We now finish the proof that COMPUTE-PREFIX-FUNCTION computes π correctly. In the procedure COMPUTE-PREFIX-FUNCTION, at the start of each iteration of the **for** loop of lines 4-9, we have that $k = \pi[q - 1]$. This condition is enforced by lines 2 and 3 when the loop is first entered, and it remains true in each successive iteration because of line 9. Lines 5-8 adjust k so that it now becomes the correct value of $\pi[q]$. The loop on lines 5-6 searches through all values $k \in \pi^*[q - 1]$ until one is found for which $P[k + 1] = P[q]$; at that point, we have that k is the largest value in the set E_q-1, so that, by Corollary 34.7, we can set $\pi[q]$ to $k + 1$. If no such k is found, $k = 0$ in lines 7-9, and $\pi[q]$ is set to 0. This completes our proof of the correctness of COMPUTE-PREFIX-FUNCTION.

Correctness of the KMP algorithm

The procedure KMP-MATCHER can be viewed as a reimplementation of the procedure FINITE-AUTOMATON-MATCHER. Specifically, we shall prove that the code on lines 6-9 of KMP-MATCHER is equivalent to line 4 of FINITE-AUTOMATON-MATCHER, which sets q to $\delta(q,T[i])$. Instead of using a stored value of $\delta(q, T[i])$, however, this value is recomputed as necessary from π. Once we have argued that KMP-MATCHER simulates the behavior of FINITE-AUTOMATON-MATCHER, the correctness of KMP-MATCHER follows from the correctness of FINITE-AUTOMATON-MATCHER (though we shall see in a moment why line 12 in KMP-MATCHER is necessary).

The correctness of KMP-MATCHER follows from the claim that either $\delta(q, T[i]) = 0$ or else $\delta(q, T[i]) - 1 \in \pi^*[q]$. To check this claim, let $k = \delta(q, T[i])$. Then, $P_k \supset P_q T[i]$ by the definitions of δ and Σ. Therefore, either $k = 0$ or else $k \geq 1$ and $P_{k-1} \supset P_q$ by dropping the last character from both P_k and $P_q T[i]$ (in which case $k - 1 \in \pi^*[q]$). Therefore, either $k = 0$ or $k - 1 \in \pi^*[q]$, proving the claim.

The claim is used as follows. Let q' denote the value of q when line 6 is entered. We use the equivalence $\pi^*[q] = \{k : P_k \supset P_q\}$ to justify the iteration $q \leftarrow \pi[q]$ that enumerates the elements of $\{k : P_k \supset P_{q'}\}$. Lines 6-9 determine $\delta(q', T[i])$ by examining the elements of $\pi^*[q']$ in decreasing order. The code uses the claim to begin with $q = \Phi(T_i - 1) = \Sigma(T_{i-1})$ and perform the iteration $q \leftarrow \pi[q]$ until a q is found such that $q = 0$ or $P[q + 1] = T[i]$. In the former case, $\delta(q', T[i]) = 0$; in the latter case, q is the maximum element in $E_{q'}$, so that $\delta(q', T[i]) = q + 1$ by Corollary 34.7.

Line 12 is necessary in KMP-MATCHER to avoid a possible reference to $P[m + 1]$ on line 6 after an occurrence of P has been found. (The argument that $q = \Sigma(T_i - 1)$ upon the next execution of line 6 remains valid by the hint given in Exercise 34.4-

6: $\delta(m, a) = \delta(\pi[m], a)$ or, equivalently, $\Sigma(Pa) = \Sigma(P\pi_{[m]_a})$ for any $a \in \Sigma$.) The remaining argument for the correctness of the Knuth-Morris-Pratt algorithm follows from the correctness of FINITE-AUTOMATON-MATCHER, since we now see that KMP-MATCHER simulates the behavior of FINITE-AUTOMATON-MATCHER.