String matching (Knuth-Morris-Pratt Algorithm)*

November 8, 2011

In this and the next two sections we will talk about strings, which are really just arrays. The elements of the array come from a constant size set \( \Sigma \) called the alphabet; the elements themselves are called characters. Common examples are English text whose alphabet consists of 26 letters plus special characters, strands of DNA constructed from an alphabet of four nucleotides, and proteins constructed from an alphabet of twenty amino acids.

**Text and pattern.** The problem we want to solve is the following. Given two strings, a text \( T[1..n] \) and a pattern \( P[1..m] \), find the first substring of the text that is the same as the pattern. Here a substring is just a contiguous subarray. For any shift \( 0 \leq s \leq n - m \) let \( T_s \) denote the substring \( T[s+1..s+m] \). More formally, we want to find the smallest shift \( s \) such that \( T_s = P \), or report that there is no match.

The straightforward approach to solving this problem uses two nested for-loops. The outer loop enumerates the \( T_s \) and the inner loop compares \( T_s \) with \( P \). We improve this algorithm by exiting the inner loop as soon as we find the first mismatch.

```plaintext
for s = 0 to n - m do j = 1;
  while j \leq m and T[s + j] = P[j] do
    j++
  endwhile;
  if j = m + 1 then
    print "\( T_s = P \)"; stop
  endif
endfor; print "there is no match".
```

In the worst case, we test \( P \) against \( n - m + 1 \) substrings comparing \( m \) pairs of characters in each test, which takes time \( O(nm) \). We can in fact create an input where the running time is as high as this pessimistic estimate: take a text consisting of \( n \) A’s and search with a pattern \( \text{AA...AB} \) of \( m - 1 \) A’s followed by one B. On the other hand, breaking out of the inner loop at the first mismatch makes the algorithm quite practical. Certainly for random strings, the probability of having long common substrings is rather small. But then again, text is typically not random.

*These notes are extracted from the lecture notes on “Design and Analysis of Algorithms”, Fall 2005 by Herbert Edelsbrunner.
Redundant comparisons. Suppose we are looking for the pattern ABRACADABRA in some longer text using the above straightforward algorithm. Consider the case shown in Figure 1 in which, for shift $s = 10$, the substring comparison fails at the fifth position. At this point the algorithm increments the shift $s$ and starts the substring

HOCUSPOCUSABRA|BRACADABRA...

<table>
<thead>
<tr>
<th></th>
<th>ABR</th>
<th>ADABRA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AB</td>
<td>RACADABRA</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>BRACADABRA</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>BRACADABRA</td>
</tr>
</tbody>
</table>

Figure 1: Action of the straightforward algorithm. The box marks the leftmost mismatch for each shift.

comparison from scratch. Note, however, that there is no point in looking at the shift $s = 11$. We already know that $T[12] = B$ because it matched $P[2]$ during the previous comparison. Likewise, we already know that the next shift $s = 12$ also fail, so why bother looking there? Finally, when we get to $s = 13$ we cannot immediately rule out a match based on earlier considerations. However, since we already know that $T[14] = P[4] = A$, we should not start the substring comparison from scratch. Instead, we should start the substring comparison at the second character of the pattern, since we do not yet know whether or not it matches the corresponding text character.

Notice that with these improvements the character comparison should always advance through the text. More precisely, once we have found a match for a text character, we never need to do another comparison with that character again. In other words, we should improve the straightforward algorithm so that it always advances through the text. We also need a good rule for finding the next shift. Remember that a prefix of a string is a substring that includes the first character. Symmetrically, a suffix is a substring that includes the last character. A prefix or suffix is proper if it is not the entire string. Suppose that we have just discovered that $T[i] \neq P[j]$. The next reasonable shift is the smallest value of $s$ such that $T[s+1..i-1]$ is a suffix of the previously read text, which is also a proper prefix of the pattern; see Figure 2.

<table>
<thead>
<tr>
<th></th>
<th>old</th>
<th>new</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>ABR</td>
<td>ABR</td>
</tr>
<tr>
<td>$P$</td>
<td>ABR</td>
<td>ABR</td>
</tr>
</tbody>
</table>

Figure 2: The new position, $s$, precedes the first character of the largest suffix that is a proper prefix of the pattern.

Finite state machines. If we have a string matching algorithm that always advances through the text, we can interpret it as feeding the text through a finite state machine,
which is a directed graph with labeled vertices. Each vertex is called a state and is labeled with a character from the pattern, except for two special states which are labeled $ and !. Success edges connect the characters in sequence while the (thin) failure edges return to earlier positions in the string.

Figure 3: The finite state machine for ABRACADABRA. The (thick) success edges connect the characters in sequence while the (thin) failure edges return to earlier positions in the string.

$ and !. Success edges connect the characters in the sequence of the pattern beginning at $ and ending at !. Failure edges point back to earlier characters. Figure 3 illustrates the idea for the pattern ABRACADABRA which we considered earlier.

We use the finite state machine to search for the pattern as follows. At all times, we have a current text character $T[i]$ and a current state, which is usually labeled by some pattern character $P[j]$. Initially, $i = 0$ and the current state is the one labeled $$. We iterate the following two rules:

- if $T[i] = P[j]$ or the current label is $$, then we follow the success edge and increment $i$;
- if $T[i] \neq P[j]$, then we follow the failure edge back to an earlier state and we keep $i$ unchanged.

The finite state machine is a convenient metaphor for a simple type of algorithm.

Knuth-Morris-Pratt algorithm. In a real implementation, we would of course not construct the entire graph. Since the success edges go through the pattern in order, we only have to remember the failure edges. Each state has one failure edge (except for states labeled with the two special characters, which have none) and we encode them in an array $Fail[1..m]$ so that for each $j$ there is a failure edge from state $j$ to state $Fail[j]$. Following a failure edge back to an earlier state corresponds to shifting the pattern forward. For now, we assume that the failure edges are correctly computed and stored in the array $Fail$. The algorithm implementing the finite state machine then looks as follows.
\( j = 1; \)
\[
\text{for } i = 1 \text{ to } n \text{ do} \\
\text{while } j > 0 \text{ and } T[i] \neq P[j] \text{ do} \\
\quad j = \text{Fail}[j] \\
\text{endwhile; } j++; \\
\text{if } j = m + 1 \text{ then} \\
\quad \text{print } "T_{i-m} = P"; \text{ stop} \\
\text{endif} \\
\text{endfor; } \text{print } "\text{there is no match}".
\]

It is fairly easy to analyze the running time of the algorithm. At each character comparison, either we increase \( i \) and \( j \) by one each, or we decrease \( j \) and leave \( i \) unchanged. We can increment \( i \) at most \( n - 1 \) times before we run out of text, which implies that there are at most that many successful comparisons. Similarly there can be at most \( n - 1 \) failed comparisons, since the number of times we decrease \( j \) cannot exceed the number of times we increment \( j \). In other words, we amortize character mismatches over earlier character matches. The number of character comparisons performed by the Knuth-Morris-Pratt algorithm is less than \( 2n \), hence the running time is in \( O(n) \).

**Failure function.** We now rephrase our rule about how to choose a reasonable shift after a character mismatch \( T[i] \neq P[j] \): “\( P[1..\text{Fail}[j] - 1] \) is the longest proper prefix of \( P[1..j - 1] \) that is also a suffix of \( T[1..i - 1] \).” Notice, however, that if we compare \( T[i] \) with \( P[j] \), then we must have already matched the first \( j - 1 \) characters of the pattern. In other words, we already know that \( P[1..j - 1] \) is a suffix of \( T[1..i - 1] \). We can therefore substitute \( P[1..j - 1] \) for \( T[1..i - 1] \) in the above rule. We arrived at the definition of the Knuth-Morris-Pratt failure function \( \text{Fail}[j] \) for all \( j > 1 \).

By convention, we set \( \text{Fail}[1] = 0 \), which tells the algorithm that if the first pattern character does not match it should give up and try the next text character. Table 1 shows the failure function for our standard pattern example. The way we compute this failure function is essentially to use the Knuth-Morris-Pratt algorithm to look for the pattern inside itself. The variable \( k \) identifies \( P[1..k - 1] \) as the longest proper prefix of \( P[1..j - 1] \) that is also a suffix of \( P[1..j - 1] \). If \( P[j] \) matches \( P[k] \) we increment \( k \), else we try the previously computed next smaller prefix:

\[
\begin{align*}
\text{k} & = 0; \\
\text{for } j = 1 \text{ to } m \text{ do} \\
\quad \text{Fail}[j] & = k; \\
\quad \text{while } k > 0 \text{ and } P[j] \neq P[k] \text{ do} \\
\quad & k = \text{Fail}[k] \\
\quad \text{endwhile; } k++; \\
\text{endfor.}
\end{align*}
\]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>R</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 1: Failure function of the string ABRACADABRA.
Just as we did for the Knuth-Morris-Pratt algorithm, we can analyze the construction of the failure function by amortizing character mismatches over earlier character matches. Since there are at most $m$ character matches, the running time is in $O(m)$.

**Improvement.** We can speed up the algorithm by making one small change to the failure function. Recall that after comparing $T[i]$ with $P[j]$ and finding a mismatch, the algorithm compares $T[i]$ with $P[\text{Fail}[j]]$. With the current definition, it is possible that $P[j]$ and $P[\text{Fail}[j]]$ are the same character, in which case the next character comparison will automatically fail. We can improve the failure function by short-cutting these redundant comparisons using some simple post-processing.

```plaintext
for j = 2 to m do
    if P[j] = P[\text{Fail}[j]] then
        \text{Fail}[j] = \text{Fail}[\text{Fail}[j]]
    endif
endfor.
```

Alternatively, we can compute the improved failure function directly by substituting lines 2, 3, 4 for the line 1 in the earlier algorithm.

```plaintext
2  if P[j] = P[k] then \text{Fail}[j] = \text{Fail}[k]
3  else \text{Fail}[j] = k
4  endif;
```

The improved failure function is shown in Table 2 and the corresponding finite state machine is shown in Figure 4.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: The improved failure function of ABRACADABRA. The changed failure pointers are bold.

Figure 4: The improved finite state machine for the pattern ABRACADABRA. The changed failure edges are dotted.