Introduction

The algorithm of Knuth, Morris, and Pratt solves the problem of substring search, where we must find a pattern of length $M$ within text of length $N$ in some alphabet of size $R$. In lecture, you covered an implementation of the algorithm with a full DFA. This algorithm runs in at most $\sim 2N$ operations (typical $\sim 1.1N$) with $\Theta(MR)$ extra space. In fact, by modifying the algorithm to compute mismatch transitions only, we have at most $\sim 3N$ (typical $\sim 1.1N$) operations with only $\Theta(M)$ space. We will now explore this optimization in order to uncover some of the additional motivations in this algorithm.

Background

Definition (Prefix of a string). A string $u$ is a prefix of a string $x$, denoted by $u \sqsubseteq x$, if $x = uv$.

Definition (Suffix of a string). A string $v$ is suffix of a string $x$, denoted by $v \sqsupseteq x$, if $x = uv$.

Definition (Border of a string). A string $r$ is a border of a string $x$ if $r$ is both a prefix and a suffix of $x$.

These definitions should be rather straightforward. We will consider the empty string $\epsilon$ as a proper prefix, suffix, and border.

Theorem 1. Let $r$ and $s$ be borders of a string $x$ with $|r| < |s|$. Then $r$ is a a border of $s$.

This theorem is rather easily visualized with if you draw a picture. Simply compare

```
  r   s   x   s   r
```

with

```
  r   s   r
```

The formal details are left as an exercise to the reader.

Why spend all this effort formalizing the notion of a border? We will explore this next.

Idea—borders and mismatch transitions

Recall that one key property of the Knuth-Morris-Pratt algorithm is that it never backtracks the input text to searched. The algorithm constructs a deterministic finite automaton (DFA), where the states represent the length of the prefix that was matched on the pattern. This DFA encapsulates how to make use of the information gained from previous symbol comparisons, so the algorithm never re-compares a text symbol that has already matched a pattern symbol.

Formalizing the problem more generally, consider a pattern $P[1 \ldots m]$ to be matched against text characters $T[1 \ldots n]$. We are aligned at position $s$ in the source, and have matched $q$ characters. This means $P[1 \ldots q]$ matches $T[s \ldots s + q]$, with $T[s + q + 1]$ being a mismatched text character.

Examine the example of [Figure 1] with pattern characters Suppose we try a shift of our pattern from position $s$ to $s + 1$. This is a bad shift to make, as the first character $a$ would be aligned with a nonmatching text character $b$. Furthermore, we know in advance that such a shift is suboptimal, since we already compared...
the text character at \( T[s+1] = b \) with the original pattern character \( P[2] = b \). At this point, we should think to ourselves that there really ought to be a way to somehow capture this information.

In general, we do not want to shift the pattern too far, lest we skip a potential match. If we shift the pattern too slowly, then we will most surely have a correct answer while sacrificing running time in the process. Since we matched \( q \) characters of the pattern, \( P[1..q] \) must be a suffix of \( T[s..s+q] \). Thus for an optimal shift, we move \( P \) into position where some prefix \( P[1..k] \) of \( P[1..q] \) that is also a suffix of \( T[s..s+q] \). In other words, we need to find the longest border of the partial match \( P[1..q] \).

Observe that on an optimal shift, we do not need to recompare pattern characters \( P[1], P[2], ..., P[k] \), so we can pick up with comparing \( P[k+1] \) with \( T[s+q+1] \). No text backtracking is performed! An example is case (b) of Figure 1.

### Implementation of Knuth-Morris-Pratt with a partial match table

We know that we need to be able to find the longest border for a partial match \( P[1..q] \). Thus, we can preprocess our pattern string to produce a table \( \pi \) for the longest border.

**Compute-Prefix-Function** \((P)\)

1. \( m = P.\text{length} \)
2. let \( \pi[1..m] \) be a new array
3. \( \pi[1] = 0 \)
4. \( k = 0 \)
5. for \( q = 2 \) to \( m \)
6.  \( \text{while } k > 0 \text{ and } P[k+1] \neq P[q] \)
7.  \( k = \pi[k] \)
8.  if \( P[k+1] = P[q] \)
9.  \( k = k + 1 \)
10. \( \pi[q] = k \)
11. return \( \pi \)

Figure 2: Computation of a function \( \pi : [1,m] \rightarrow [1,m] \) defining the widest border for a particular partial pattern match of length \( q \). Compare this to the DFA implementation from lecture. The function \( \pi \) gives the failure transitions, which is a subset of the full DFA transition matrix. Notice that the full DFA transition matrix processing is now stripped down to the important part of determining how to proceed on mismatches.

In the preprocessing, we compute an array \( \pi[1..m] \). At the end of **Compute-Prefix-Function**, each entry \( \pi[k] \) contains the width of longest border for prefix match of length \( k \). We consider increasing substrings of \( P \) of length \( q \). At the beginning of each for loop iteration, \( k \) stores the border computed from the previous iteration. In order to compute the widest border for the next character in the pattern, we need to check that we can extend \( P[1..k] \) with \( P[k+1] = P[q] \). If we cannot extend the border to include \( P[k+1] \) (if the character under consideration \( P[q] \) does not match), then we have to potentially check \( \pi^n[k] = \pi[\pi[\cdots \pi[k]]] \). In other words, we need to examine the next-widest border and see if we can extend that, or potentially the next-next-widest, etc.

![Figure 3](image)

Figure 3: The state of **Compute-Prefix-Function** right before updating \( \pi[q] \) on line 10. The algorithm has verified that the border (in light gray) can be extended to \( P[k+1] \). Characters \( k+1 \) and \( q \) are in blue. Note that at the beginning of the iteration, \( k \) stores the widest border on the previous iteration.
The running time of \texttt{Compute-Prefix-Function} is $\Theta(m)$. The hard part of this analysis is understanding why the \texttt{while} loop executes $O(m)$ time \textit{altogether}. Let us think about how $k$ is modified. In each iteration of the \texttt{for} loop, the value of $k$ is increased at most once on line 9, so $k$ is increased at most $m - 1$ times. Each iteration of the \texttt{while} loop decreases the value of $k$ on line 7. Since $k$ never becomes negative, if we consider all iterations of the \texttt{for} loop, the \texttt{while} loop can only decrease $k$ at most $m - 1$ times, so \texttt{Compute-Prefix-Function} runs in $\Theta(m)$ time.

\section*{Putting it all together—the final Knuth-Morris-Pratt matcher}

Without further ado, here is the final search function.

\begin{verbatim}
KMP-Matcher(T, P)
  1  m = P.length
  2  n = T.length
  3  π = Compute-Prefix-Function(P) // number of characters matched
  4  q = 0 // scan the text from left to right
  5  for i = 1 to n // number of characters matched
  6         while q > 0 and P[q + 1] \neq T[i] // next character does not match
  7         q = π[q] // look for the next match
  8       if P[q + 1] = T[i] // next character matches
  9           q = q + 1 // is all of P matched?
 10       if q == m
 11           print "Pattern occurs with shift" i - m // look for the next match
 12     q = π[q] // number of characters matched
\end{verbatim}

Note that \texttt{KMP-Matcher} feels very similar to \texttt{Compute-Prefix-Function}. Both match a string against the pattern $P$. In particular, \texttt{KMP-Matcher} matches text $T$ against $P$, and \texttt{Compute-Prefix-Function} matches $P$ against itself.
Chapter 32  String Matching

Figure 32.10 The prefix function $\pi$. (a) The pattern $P = ababa$ aligns 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) We can precompute useful information for such deductions by comparing the pattern with itself. Here, we see that the longest prefix of $P$ that is also a proper suffix of $P_5$ is $P_3$. We represent this precomputed information in the array $\pi$, so that $\pi(5) = 3$. Given that $q$ characters have matched successfully at shift $s$, the next potentially valid shift is at $s' = s + (q - \pi(q))$ as shown in part (b).

Figure 1: Mismatch transitions