Knuth-Morris-Pratt & Boyer-Moore Algorithms
by Robert C. St.Pierre

Overview
- Notation review
- Knuth-Morris-Pratt algorithm
  - Discussion of the Algorithm
  - Example
- Boyer-Moore algorithm
  - Discussion of the Algorithm
  - Example
- Applications in automatic proving

Notation Review (1)
- $T[1..n]$ – text to search, length $n$
  - Characters from $\Sigma$
- $P[1..m]$ – pattern text, length $m$
  - Characters from $\Sigma$
  - $m \leq n$
- $\Sigma$ – finite alphabet
- $\delta$ – transition function in a FA

Notation Review (2)
- $P_k$ – $k$-character prefix $P[1..k]$ of $P[1..m]$
- $w$ is a prefix of $x$, denoted $w \subset x$, if $x = wy$ for some string $y \in \Sigma^*$
- $w$ is a suffix of $x$, denoted $w \supset x$, if $x = yw$ for some string $y \in \Sigma^*$
- $\varepsilon$ - empty string

Notation Review (3)
- $s$ – shift
  - $0 \leq s \leq n - m$
- If $T[s+1..s+m] = P[1..m]$ then $s$ is called a valid shift else $s$ is called an invalid shift.
- The string-matching problem is to find all valid shifts.

The Kunth-Morris-Pratt (KMP) Algorithm
- $\Theta(m)$ preprocessing time
  - Saves a factor of $|\Sigma|$ over the preprocessing time for the FA
    - Done by using an auxiliary function $\pi$, instead of the transition function $\delta$.
- $\Theta(n)$ matching time
  - Section 32.4 in text.
The prefix function $\pi$ 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 $\delta$ for a string-matching automation.

Formally:
- Given a pattern $P[1..m]$, the prefix function for the pattern $P$ is the function $\pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m-1\}$ such that $\pi[q] = \max\{k : k < q \text{ and } P[k+1] \supseteq P[q]\}$.
- $\pi[q]$ is the length of the longest prefix of $P$ that is a proper suffix of $P[q]$.

KMP Algorithm

**Discussion – Prefix Function (1)**

**KMP Algorithm**

**Discussion – Prefix Function (2)**

**KMP Algorithm**

**Discussion – Prefix Function (3)**

**KMP Algorithm**

**Discussion – Prefix Function (4)**

**KMP Algorithm**

**Discussion – Pseudocode (1)**

**KMP Algorithm**

**Discussion – Pseudocode (2)**

**KMP Algorithm**

**Discussion – Pseudocode (1)**

**KMP Algorithm**

**Discussion – Pseudocode (2)**
KMP Algorithm Discussion – Correctness (1)

- Lemma 32.5 (Prefix-function iteration lemma)
  Let P be a pattern of length m with prefix function \( \pi \). Then, for \( q = 1, 2, \ldots, m \), we have 
  \[ \pi^\ast[q] = \{ k : k < q \text{ and } P_k \supseteq P_q \} \].

- Lemma 32.6
  Let P be a pattern of length m, and let \( \pi \) be the prefix function for P. For \( q = 1, 2, \ldots, m \), if \( \pi[q] > 0 \), then \( \pi[q] - 1 \in \pi^\ast[q - 1] \).

KMP Algorithm Examples (HTML based)


The Boyer-Moore Algorithm

- “If the pattern P is relatively long and the alphabet \( \Sigma \) is reasonably large, then [this algorithm] is likely to be the most efficient string-matching algorithm.”
- Matches right to left, unlike KMP.
- This algorithm is NOT in the second edition of our book, but is in section 34.5 of the first edition.

Boyer-Moore Algorithm Discussion – Matcher Function (1)

Boyer-Moore-Matcher(T, P, \( \Sigma \))

```plaintext
1. n <- length(T)
2. m <- length(P)
3. \( \lambda \) <- Compute-Last-Occurrence-Function(P, m, \( \Sigma \))
4. \( \gamma \) <- Compute-Good-Suffix-Function(P, m)
5. s <- 0
6. while s <= n - m
7. do j <- m
8. while j > 0 and P[j] = T[s+j]
9. do j <- j – 1
10. if j = 0
11. then print "Pattern occurs at shift s"
12. s <- s + \( \gamma[0] \)
13. else s <- s + max(\( \lambda[0] \), j - \( \lambda[T[s+j]] \))
```

KMP Algorithm Discussion – Correctness (2)

For \( q = 2, 3, \ldots, m \), define the subset \( E_{q-1} \subseteq \pi^\ast[q - 1] \) by 
\[
E_{q-1} = \{ k : k \in \pi^\ast[q-1] : P(k+1) = P(q) \}
\]

\[
= \{ k : k < q - 1 \text{ and } P_k \supset P_{q-1} \text{ and } P(k+1) = P(q) \} \text{ (by Lemma 32.5)}
\]

\[
= \{ k : k < q - 1 \text{ and } P_{k+1} \supset P_{q-1} \} .
\]

Corollary 32.7
Let P be a pattern of length m, and let \( \pi \) be the prefix function for P. For \( q = 2, 3, \ldots, m \), 
\[
\pi[q] = \begin{cases} 
0 & \text{if } E_{q-1} = \emptyset, \\
1 + \max \{ k : k \in E_{q-1} \} & \text{if } E_{q-1} \neq \emptyset.
\end{cases}
\]
The function Boyer-Moore-Matcher(T, P, Σ) "looks remarkably like the naive string-matching algorithm." Indeed, commenting out lines 3-4 and changing lines 12-13 to $s <- s + 1$, results in a version of the naive string-matching algorithm.

The Boyer-Moore Algorithm uses the greater of two heuristics to determine how much to shift next by.

The first heuristic, is the bad-character heuristic.

In general, works as follows:

$P[j] \neq T[s+j]$ for some $j$, where $1 \leq j \leq m$. Let $k$ be the largest index in the range $1 \leq k \leq m$ such that $T[s+j] = P[k]$, if any such $k$ exists. Otherwise let $k = 0$.

We can safely increase by $j - k$, three cases to show this.

Case 1. $k = 0$, so increase by $j$.

Case 2. $k < j$, so increase by $j - k$.

Case 3. $k > j$, resulting in a negative shift, but the good-suffix heuristic recommendation is ignored.

Case 4. $k > j$, resulting in a negative shift, but the good-suffix heuristic recommendation is ignored.

Compute-Last-Occurrence-Function(P, m, Σ)

1. for each character $a \in \Sigma$
2. do $\lambda(a) = 0$
3. for $j \leftarrow 1$ to $m$
4. do $\lambda(P[j]) \leftarrow j$
5. return $\lambda$

The running time of this procedure is $O(|\Sigma| + m)$. 
Define the relation $Q \sim R$ for strings $Q$ and $R$ to mean that $Q \supset R$ or $R \supset Q$.

If two strings are similar, then we can align them with their rightmost characters matched, and no pair of aligned characters will disagree.

The relation "~" is symmetric.

$Q \sim R$ and $S \sim R$ imply $Q \sim S$.

"If $P[j] \neq T[s+j]$, where $j < m$, then the good-suffix heuristic says that we can safely advance by $\gamma[j] = m - \max\{k: 0 \leq k < m \text{ and } P[j+1..m] \sim P_k\}$."

$\gamma[j]$ is the least amount we can advance $s$ and not cause any characters in the "good suffix" $T[s+j+1..s+m]$ to be mismatched against the new alignment of the pattern.

$\gamma[j] > 0$ for all $j = 1..m$, which ensures that this algorithm makes progress.

This function has running time $O(m)$.

Worst case is $O((n - m + 1)m + |\Sigma|)$

- Compute-Last-Occurrence-Function takes time $O(m + |\Sigma|)$.
- Compute-Good-Suffix-Function takes time $O(m)$.
- $O(m)$ time is spent validating each valid shift $s$.

http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/strstrpos-example.html
Boyer-Moore Algorithm
Examples (Applets)

- http://www-sr.informatik.uni-tuebingen.de/~buehler/BM/BM.html

References