A Simple Fast Hybrid Pattern-Matching Algorithm

Frantisek Franek*, Christopher G. Jennings†
& Bill Smyth*‡

The Knuth-Morris-Pratt (KMP) pattern-matching algorithm guarantees both independence from alphabet size and worst-case execution time linear in the pattern length; on the other hand, the Boyer-Moore (BM) algorithm provides near-optimal average-case and best-case behaviour, as well as executing very fast in practice. We describe a simple algorithm that employs the main ideas of KMP and BM (with a little help from Sunday) in an effort to combine these desirable features. Experiments indicate that in practice the new algorithm is consistently among the fastest exact pattern-matching algorithms discovered to date, apparently dominant for alphabet size 8 or more.

* Algorithms Research Group, McMaster University
† School of Computing Science, Simon Fraser University
‡ Department of Computing, Curtin University
We wish to compute all occurrences of a pattern $p = p[1..m]$ in a text string $x = x[1..n]$. We suppose that both strings are defined on an indexed alphabet $\Sigma$; for convenience, we suppose $\Sigma = \{1, 2, \ldots, k\}$.

Recall that Algorithm KMP resets $j \leftarrow \beta'[j]$ at each iteration. If $\beta[j]$ is one more than the length of the longest border of $p[1..j-1]$, then $\beta'[j]$ is defined as follows:

If $j = m+1$, $\beta'[j] = \beta[j]$. Otherwise, for $j \in 1..m$, $\beta'[j] = j'$ where $j'-1$ is the length of the longest border of $p[1..j-1]$ such that $p[j'] \neq p[j]$; if no such border exists, $\beta'[j] = 0$.

$\beta'$ is computable in $\Theta(m)$ time.
Recall that after each mismatch between $p[m]$ and $x[i']$, Algorithm BMS (Sunday) compares $p[m]$ and

$$x[i' + \Delta[x[i' + 1]]]$$

in the next iteration. For every letter $h \in 1..k$, $\Delta[h]$ is defined as follows:

$$\Delta[h] = m - j' + 1,$$

where $j'$ is the position of rightmost occurrence of the letter $h$ in $p$, if it exists; zero otherwise.

$\Delta$ is computable in $\Theta(m+k)$ time.
Main ideas of the algorithm:

(1) when \( p[m] \neq x[i'] \), do Sunday shifts until \( p[m] = x[i'] \);

(2) when \( p[m] = x[i'] \), do KMP matching of \( p[1..m-1] \) with \( x[i'-m+1..i'-1] \) — on completion, set \( j \leftarrow \beta'[j] \) (KMP shift), \( i' \leftarrow i+m-j \), and go to (1).

While no mismatch occurs, the algorithm does KMP matching in the order \( m, 1, 2, \ldots, m-1 \). The invariant \( i' = i+m-j \) is maintained, where \( i \) is the position in \( x \) that mismatches with position \( j \) in \( p \). The algorithm’s efficiency seems to depend on

- Markov independence of \( p[m] \) and \( p[1] \);

- avoidance of resetting \( j \leftarrow 1 \) during Sunday shifts.
Find all occurrences of $p[1..m]$ in $x[1..n]$

if $m < 1$ then return
$i' \leftarrow m; j \leftarrow 1; m' \leftarrow m - 1$
while $i' \leq n$ do

— BM (Sunday) shift if $p[m]$ fails to match
if $p[m] \neq x[i']$ then
repeat
  $i' \leftarrow i' + \Delta[x[i' + 1]]$
  if $i' > n$ then return
until $p[m] = x[i']$
$j \leftarrow 1$

— KMP matching if $p[m]$ matches
if $j \leq 1$ then
  $i \leftarrow i' - m'; j \leftarrow 1$
while $j < m$ and $x[i] = p[j]$ do
  $i \leftarrow i + 1; j \leftarrow j + 1$

— Restore invariant $i' = i + m - j$ for next shift
if $j = m$ then
  $i \leftarrow i + 1; j \leftarrow j + 1; \textbf{output } i - m$
  $j \leftarrow \beta'[j]; i' \leftarrow i + m - j$
**Letter Comparisons**

There can be at most $n-m+1$ comparisons in the Sunday portion. The KMP portion seeks to match a pattern of length $m' = m-1$ in a text of length $n' = n-1$ ($x[n]$ never tested). Since KMP performs at most $2n' - m'$ letter comparisons, therefore FJS performs at most

$$2(n-1) - (m-1) + (n - m + 1) = 3n - 2m$$

letter comparisons. The upper bound is attained by $x = a^n$, $p = a^{m-2}ba$. 
Efficiency in Practice

We compared Algorithm FJS with four algorithms that appear to be among the fastest in practice:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>BM Horspool (BMH)</td>
<td>$O(mn)$</td>
</tr>
<tr>
<td>BM Sunday (BMS)</td>
<td>$O(mn)$</td>
</tr>
<tr>
<td>Reverse Colussi (RC)</td>
<td>$O(mn)$</td>
</tr>
<tr>
<td>Turbo BM (TBM)</td>
<td>2n letter comparisons</td>
</tr>
</tbody>
</table>

Except for alphabet size less than 8, FJS was generally about 10% faster than its nearest competitor (usually either BMS or RC) — pre-processing time was always included.
Selection of Text Files

(1) We used Project Gutenberg with obsolete duplicates removed — altogether 2434 files, of which 1000 were selected at random for use. This was a total of 446M letters, with individual file sizes ranging from 10K to 4.8M.

(2) We used a version of the Human Genome Project’s map of the first human chromosome, filtered to remove non-DNA data — altogether 211M letters on \( \{A, T, C, G\} = \{00, 01, 10, 11\} \). Random substrings of the resulting binary string were used to construct strings on alphabets of sizes \( 2^k \), \( 1 \leq k \leq 6 \).
Gutenberg Patterns

(1) *High-Frequency Patterns*
We selected the seven most frequent six-letter patterns found in a random sample of 200 texts — occurring a total of 3.4M times in the complete corpus.

(2) *Moderate-Frequency Patterns*
We selected seven patterns of length 6 whose frequency in the 200-text sample ranked 185 or higher — occurring a total of 267K times in the whole corpus.

(3) *Variable Pattern Length*
We constructed sets of 9 patterns for each pattern length $3 \leq m \leq 9$, and we also tested 90 patterns of lengths 25-175, randomly selected from the corpus.
Gutenberg Results

(1) For high- and moderate-frequency patterns, execution time for all five algorithms increased linearly with text length; RC & BMS were virtually indistinguishable, and FJS was consistently faster than either of them by about 10%.

(2) For variable-length patterns, FJS was again fastest (over BMS) by about 10% for pattern-length $m \leq 9$. But not surprisingly, FJS & BMS became indistinguishable for $m \geq 100$. RC was burdened by its $O(m^2)$ preprocessing and was slowest for $m \geq 30$ — for $m = 175$, an order of magnitude slower than any other algorithm.
DNA-Based Results

For every $k \in 1..6$, a text of length $500K$ on an alphabet of size $2^k$ was formed and searched for 20 patterns of length 6, each pattern selected at random from the text.

For $k \leq 2$ (alphabet sizes 2 & 4), RC and BMS were much faster than FJS, but for $k \geq 3$ (alphabet size 8 or more), FJS was superior to both BMS (second fastest) and RC (third fastest), again by a margin increasing to 10%.
Conclusion

Over all pattern sizes and for alphabet size of 8 or more, Algorithm FJS appears to be the algorithm of choice, at least among the other algorithms tested.