A Simple Fast Hybrid Pattern-Matching Algorithm[⋆]

Frantisek Franek[1], Christopher G. Jennings[2], and William F. Smyth[1,3]

[1] Algorithms Research Group, Department of Computing & Software
McMaster University, Hamilton ON L8S 4K1, Canada
franek@mcmaster.ca
www.cas.mcmaster.ca/cas/research/groups.shtml

[2] School of Computing Science, Simon Fraser University
8888 University Drive, Burnaby BC V5A 1S6, Canada
cjennings@acm.org

[3] School of Computing, Curtin University, GPO Box U1987
Perth WA 6845, Australia
smyth@computing.edu.au

Abstract. 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 among the fastest exact pattern-matching algorithms discovered to date, perhaps dominant for alphabet size 8 or more.

1 Introduction

Since 1977, with the publication of both the Boyer-Moore [1] and Knuth-Morris-Pratt [16] pattern-matching algorithms, there have certainly been hundreds, if not thousands, of papers published that deal with exact pattern-matching, and in particular discuss and/or introduce variants of either BM or KMP. The literature has had two main foci:

– reducing the number of letter comparisons required in the worst/average case (for example, [3–5, 7, 10, 11]);
– reducing the time requirement in the worst/average case (for example, [6, 8, 13, 14, 21]).

This contribution resides in the second of these categories: in an effort to reduce processing time, we propose a mixture of Sunday's variant [21] of BM [1] with KMP [16, 19]. Our goal is to combine the best/average case advantages of Sunday's algorithm (BMS) with the worst case guarantees of KMP. Experiments

[⋆] Supported in part by grants from the Natural Sciences & Engineering Research Council of Canada.

suggest that our new algorithm (FJS) is among the fastest in practice for computating all occurrences of a pattern $\boldsymbol{p} = \boldsymbol{p}[1..m]$ in a text string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on an alphabet Σ of size k. Based on $\Theta(m+k)$-time preprocessing, it also guarantees that matching requires at most $3n-2m$ letter comparisons and $O(n)$ time.

Several comparative surveys of pattern-matching algorithms have been published over the years [9, 14, 17, 18, 20] and a useful website [2] is maintained that gives C code for the main algorithms and describes their important features.

In this paper we first introduce, in Section 2, the new algorithm, establish its asymptotic time and space requirements in the worst case, and demonstrate its correctness. Then in Section 3 we describe experiments that compare Algorithm FJS with algorithms that, from the available literature, appear to provide the best competition in practice; specifically: Horspool's algorithm (BMH) [13], Sunday's algorithm (BMS) [21], Reverse Colussi (RC) [6], and Turbo-BM (TBM) [8]. Finally, in Section 4, we draw conclusions from our experiments.

2 The New Algorithm

Algorithm FJS combines two well-known pattern-matching ideas:

(1) In accordance with the BM approach, FJS first compares $\boldsymbol{p}[m]$, the rightmost letter of the pattern, with the letter in the corresponding text position i'. If a mismatch occurs, a "Sunday shift" is implemented, moving $\boldsymbol{p}$ along $\boldsymbol{x}$ until the rightmost occurrence in $\boldsymbol{p}$ of the letter $h = \boldsymbol{x}[i'+1]$ is positioned at $i'+1$. At this new location, the rightmost letter of $\boldsymbol{p}$ is again matched with the corresponding text position. Only when a match is found does FJS invoke the next (KMP) step; otherwise, another Sunday shift occurs.

(2) If $\boldsymbol{p}[m] = \boldsymbol{x}[i']$, KMP pattern-matching begins, starting (as KMP does) from the lefthand end $\boldsymbol{p}[1]$ of the pattern and, if no mismatch occurs, extending as far as $\boldsymbol{p}[m-1]$. Then, whether or not a match for $\boldsymbol{p}$ is found, a KMP shift is eventually performed, followed by a return to step (1).

The pseudocode for Algorithm FJS is shown in the figure below.

As discussed in [15], the real rationale for the algorithm lies in the avoidance of Markov effects combined with efficient shifting: a match of $\boldsymbol{p}[m]$ with $\boldsymbol{x}[i']$ can in most circumstances be regarded as independent of a match (or mismatch) with $\boldsymbol{p}[1]$, where KMP matching begins; if $\boldsymbol{p}[m] \neq \boldsymbol{x}[i']$, then the Sunday shift provides an efficient mechanism for sliding $\boldsymbol{p}$ across $\boldsymbol{x}$. Viewed from one perspective, FJS just performs KMP matching in a slightly different order: position m of $\boldsymbol{p}$ is compared first, followed by $1, 2, \ldots, m-1$. Indeed, since the KMP shift is also employed, the number of worst-case letter comparisons required for FJS is bounded above, as we show below, by $3n-2m$, compared with KMP's $2n-m$.

Preprocessing

The algorithm uses two arrays: Sunday's array $\Delta = \Delta[1..k]$ computable in $\Theta(m+k)$ time, and the KMP array $\beta' = \beta'[1..m+1]$, computable in $\Theta(m)$ time.

Algorithm 1 (Hybrid Matching)

```
Find all occurrences of $\boldsymbol{p} = \boldsymbol{p}[1..m]$ in $\boldsymbol{x} = \boldsymbol{x}[1..n]$

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

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

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

  — Restore invariant $i' = i+m-j$ for next shift
  if $j = m$ then
    $i \leftarrow i+1;\ j \leftarrow j+1;$ output $i-m$
  $j \leftarrow \beta'[j];\ i' \leftarrow i+m-j$
```

The h^{th} position in the Δ array is accessed directly by the letter h of the alphabet, and so we need to make the assumption required by all BM-type algorithms, that the alphabet is ***indexed*** [20] – essentially, that $\Sigma = \{1, 2, \cdots, k\}$ for some integer k fixed in advance. Then for every $h \in 1..k$, $\Delta[h] = m - j' + 1$, where j' is the position of rightmost occurrence of the letter h in $\boldsymbol{p}$, if it exists; zero otherwise. Thus in case of a mismatch with $\boldsymbol{x}[i']$, $\Delta\big[\boldsymbol{x}[i'+1]\big]$ computes a shift that places the rightmost occurrence j' of letter $\boldsymbol{x}[i'+1]$ in $\boldsymbol{p}$ opposite position $i'+1$ of $\boldsymbol{x}$, whenever j' exists in $\boldsymbol{p}$, and otherwise shifts $\boldsymbol{p}$ right past position $i'+1$.

To define the β' array, consider first an array $\beta[1..m+1]$ in which $\beta[1] = 0$ and for every $j \in 2..m+1$, $\beta[j]$ is one more than the length of the longest border of $\boldsymbol{p}[1..j-1]$. 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 $\boldsymbol{p}[1..j-1]$ such that $\boldsymbol{p}[j'] \neq \boldsymbol{p}[j]$; if no such border exists, $\beta'[j] = 0$.

See [2] for C code and [9, 20] for further discussion of the preprocessing arrays.

The Algorithm

The KMP part of the algorithm needs to compare position i of $\boldsymbol{x}$ with position j of $\boldsymbol{p}$, where j may be the result of a partial match with $\boldsymbol{p}[1..j-1]$ or of an assignment $j \leftarrow \beta'[j]$. On the other hand, the BMS part needs to compare position

$$i' = i+m-j \tag{1}$$

of $\boldsymbol{x}$ with position m of $\boldsymbol{p}$. Thus (1) is maintained as an invariant by FJS. Since the assignment $j \leftarrow \beta'[j]$ may actually set j to zero, (1) may not hold if $j = 0$ and $\boldsymbol{p}[m] = \boldsymbol{x}[i']$ at the next match; it is for this reason that the assignment $j \leftarrow 1$ is made for $j \leq 1$ at the beginning of the KMP section.

Correctness

The correctness of Algorithm FJS is a consequence of the correctness of BMS and KMP. Note that a sentinel letter needs to be added at position $n+1$ of $\boldsymbol{x}$ to ensure that $\Delta\big[\boldsymbol{x}[i'+1]\big]$ is well defined for $i' = n$. If this is undesirable, the final alignment at $\boldsymbol{x}[n-m+1..n]$ can be tested as a special case outside of the main loop.

Letter Comparisons

In the worst case there may be as many as $n-m+1$ executions of the BM-type match in Algorithm FJS. Then KMP-type letter comparisons could take place on the remaining $m-1$ positions of the pattern over a text string of length $n-1$ (the final letter of $\boldsymbol{x}$ would never be subject to KMP-type matching). It is well known (see, for example, [20]) that Algorithm KMP performs at most $2n'-m'$ letter comparisons to find matches for a pattern of length m' in a string of length n'. Substituting $n' = n-1$, $m' = m-1$, we find the largest possible number of letter comparisons by Algorithm FJS is

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

It is straightforward to verify that this bound is in fact attained by $\boldsymbol{p} = a^{m-2}ba$, $\boldsymbol{x} = a^n$. Thus

Theorem 1 *Algorithm FJS requires at most $3n-2m$ letter comparisons in the worst case, a bound attained by $\boldsymbol{p} = a^{m-2}ba$, $\boldsymbol{x} = a^n$.* $\square$

We see that, in terms of worst-case letter comparisons, there is a slight price to be paid for the improved average-case efficiency of FJS.

3 Experimental Results

As mentioned in Section 1, we conducted a number of experiments to compare Algorithm FJS with four competitors (BMH, BMS, RC, TBM) known to be among the fastest in practice[1].

[1] Colussi presents two versions of RC. We used the first version, which is faster on average but requires $\Theta(mn)$ time in the worst case; a second version matches in worst-case $\Theta(n)$ time but is typically slower. Tests of the second version did not produce significantly different results relative to FJS.

Our implementations were adapted from [2], which provides well-tested C code for many pattern-matching algorithms. This gave us a standard to mimic for FJS, ensuring fairness and consistency. A significant change was the replacement of calls to the C library functions *memcmp()* and *memset()*, made by a few of the implementations, with equivalent loops. The library calls, which are optimized for large memory regions, were four to five times slower for our tests.

Timing made use of the high-frequency hardware clocks available on modern computers. Each test was repeated 20 times and the fastest time was kept. This is closest to a true minimum independent of external effects such as cache misses. All times include both preprocessing and matching.

The results presented here were taken from the test system with the most precise timing – an AMD Athlon 2500+ PC with 512MB RAM, running Windows 2000 (SP4) and using Microsoft C/C++ 12.00.8168. However, the resulting trends were highly stable across eight environments using a variety of hardware, operating system, and compiler vendor combinations.

Test Data

The primary corpus was drawn from a set of 1000 randomly selected texts from Project Gutenberg [12], an online source of public domain documents. The corpus contained a total of 446 504 073 letters, with individual texts ranging from 10 115 to 4 823 268 letters in length.

A second corpus was constructed using the Human Genome Project's map of the first human chromosome. A string of 210 992 564 nucleotides was transformed into a bit string by mapping A, T, C, and G to 00, 01, 10, and 11, respectively. Random substrings were then used to make texts of varying alphabet size.

Frequency of Occurrence

We constructed high- and moderate-frequency pattern sets to observe the effect of match frequency on performance. To obtain comparable results, we fixed the number and length of the patterns in both sets. As both values must be kept fairly small to construct a meaningful high-frequency pattern set, we settled on sets of seven patterns of length six.

The high-frequency set was selected by finding the seven most frequent length six substrings in a random sample of 200 texts from our primary corpus. The resulting set, with ␣ representing the space symbol, is as follows:

␣of␣th *of␣the* *f␣the␣* *␣that␣* *,␣and␣* *␣this␣* *n␣the␣*

These patterns occur 3 366 899 times in the primary corpus. Figure 1 graphs performance on this set. The execution time for each text is the total time taken for all patterns, where the time for each pattern is determined as described above.

For the moderate frequency set, we constructed a set of 7 words of at least length six sharing an expected frequency of 4 instances per 10 000 words (long words were truncated to six letters). The actual match frequency may vary since patterns may occur often within other words, but this tends to occur in practice as well. After restricting all members to at most six letters the resulting set is:

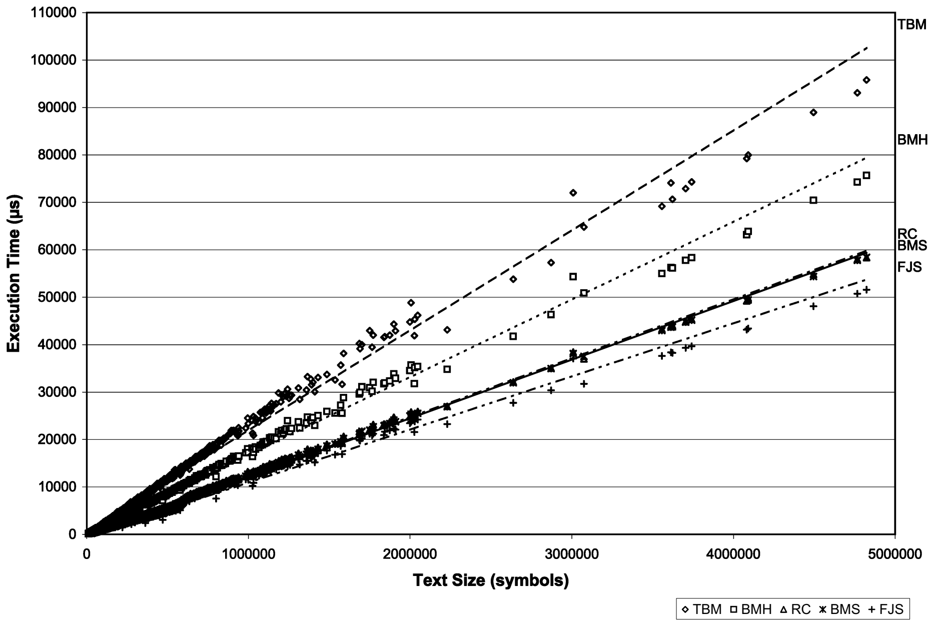

Fig. 1. Execution Time versus Text Length for High-frequency Pattern Set

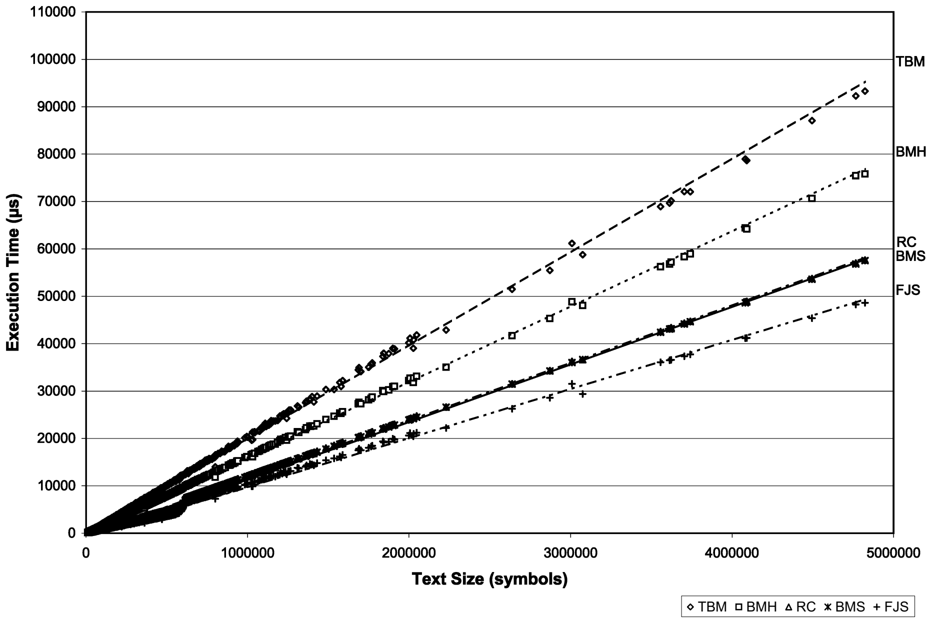

Fig. 2. Execution Time versus Text Length for Moderate-frequency Pattern Set

better enough govern public someth system though

These patterns occur 266 792 times in our corpus. Figure 2 graphs the results.

Pattern Length

Pattern sets were constructed consisting of nine patterns for each pattern length from three to nine. These were selected using a similar process to the one described above, but with an expected frequency of 2–3 times per 10 000 words. The reduced frequency was needed to allow sufficient candidates of each length.

Length	Matches	Pattern Set
3	586 198	*air, age, ago, boy, car, I'm, job, run, six*
4	346 355	*body, half, held, past, seem, seen, tell, week, word*
5	250 237	*death, field, money, quite, seems, shall, taken, whose, words*
6	182 353	*became, behind, cannot, having, making, moment, period, really, result*
7	99 109	*already, brought, college, control, federal, further, provide, society, special*
8	122 854	*anything, evidence, military, position, probably, problems, question, students, together*
9	71 371	*available, community, education, following, necessary, political, situation, sometimes, therefore*

Figure 3 graphs the results from this test. The time at each length is the total time for all nine patterns in the relevant set over the entire primary corpus. As the trends of FJS and BMS suggest that they are asymptotically the same, an additional experiment was performed testing 90 patterns with lengths from 25 to 175, randomly selected from the corpus. FJS and BMS became indistinguishable from about $m = 100$ onward. RC was burdened by its $O(m^2)$ preprocessing time, placing last from $m = 30$ on. At $m = 175$, it was more than an order of magnitude slower than its nearest competitor.

Alphabet Size

These tests used the secondary corpus of DNA-derived texts. This corpus contained 6 texts of 500 000 letters, each over an alphabet of a different size. Each text was searched with 20 random substrings of length 6. The results, seen in Figure 4, suggest that Algorithm FJS may be a poor choice for $k < 8$ – thus, perhaps, not optimal for DNA sequences ($|\Sigma| = 4$), but suitable for proteins composed of amino acids ($|\Sigma| = 20$).

Pathological Cases

Periodic strings can induce theoretical worst-case behaviour in pattern-matching algorithms. For FJS in particular, $\boldsymbol{x} = a^n, \boldsymbol{p} = aba$ maximizes the comparison

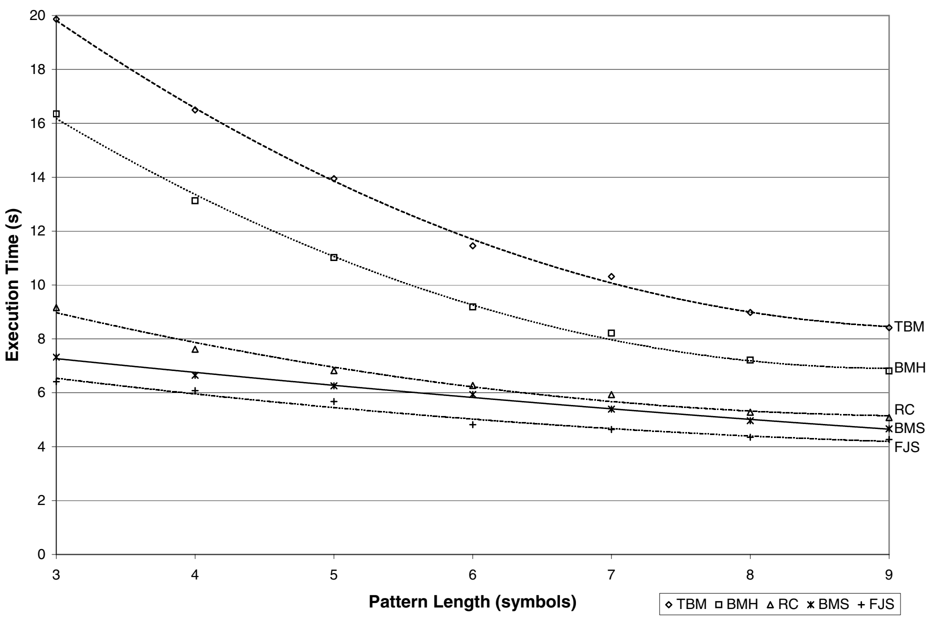

Fig. 3. Execution Time versus Pattern Length

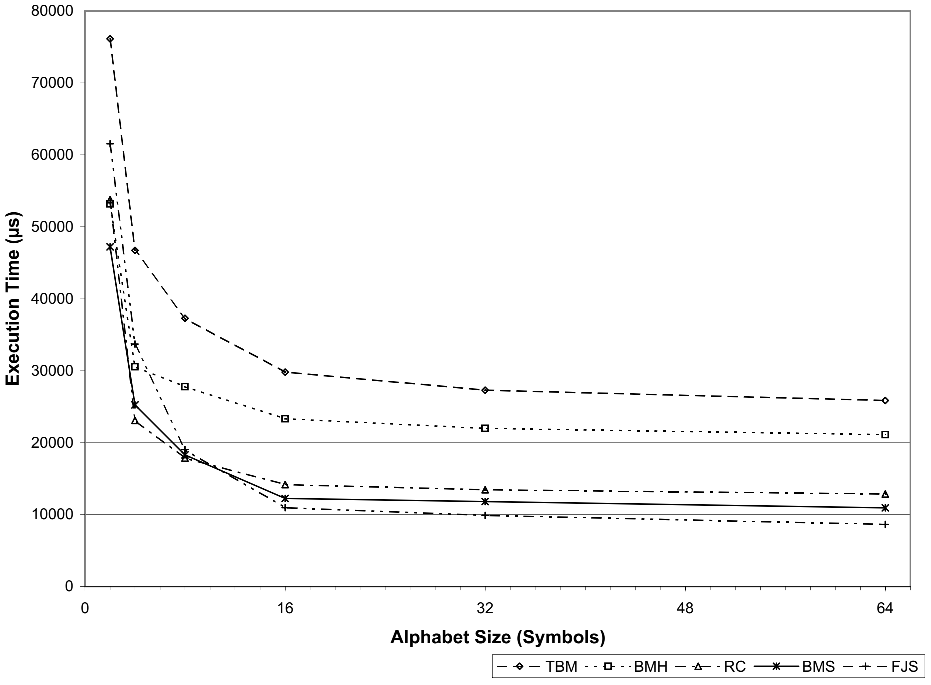

Fig. 4. Execution Time versus Alphabet Size

count. We compared performance on these strings for values of n from 10 000 to 100 000 at 10 000 letter intervals. The rank order, from fastest to slowest, was RC, BMH, BMS, FJS, TBM; RC timed 41% faster than FJS on average.

On the other hand, patterns of the form $\boldsymbol{x} = a^n, \boldsymbol{p} = a^m$ (and in particular, $m = \lceil n/2 \rceil$) maximize comparisons for many non-linear BM variants, including BMH and BMS. Fixing the text as $\boldsymbol{x} = a^{100000}$, we timed the five implementations for $\boldsymbol{p} = a^3$ through $\boldsymbol{p} = a^9$. The results showed FJS to be an average of 36% faster than second-place TBM, the only $\Theta(n)$ competitor.

The worst case for the original BM algorithm can be triggered with $\boldsymbol{x} = (a^k b)^r, \boldsymbol{p} = a^{k-1} b a^{k-1}$. As in the previous case, both the pattern and the text are highly repetitive. We performed two tests on string pairs of this kind. In the first test, we held r at 100 000 and varied k from 2 through 20. In the second test, we held k at 10 and varied r from 100 000 through 500 000. Both experiments indicated an advantage to FJS by 15–20% over BMS and RC and by a large margin over all others.

4 Discussion and Conclusion

We have tested FJS against four high-profile competitors (BMH, BMS, RC, TBM) over a range of contexts: pattern frequency (C1), pattern length (C2), alphabet size (C3), and pathological cases (C4).

Over contexts (C1) and (C2) for English text, FJS was uniformly 10% or so faster than BMS and RC, its closest adversaries. For very long patterns, the Sunday skip loop dominates and FJS and BMS are interchangeable.

In terms of (C3), for small alphabets the expected incidence of $\boldsymbol{p}[m] = \boldsymbol{x}[i']$ is high: KMP will be invoked often, slowing FJS with respect to BMS and RC. For $k \geq 8$, FJS regains its 10% or so advantage.

The overall speed advantage of FJS probably relates to its efficient average-case processing of the most common case (initial mismatch) resulting from the use of the skip loop to avoid repeated settings of $j = 1$ and Markov independence of position m and position j in the pattern $\boldsymbol{p}$. As discussed in [15], the first factor appears to be the more important.

For FJS the pathological cases (C4) are those in which the KMP part is forced to execute on prefixes of patterns where KMP has no advantage (no nonempty borders). On the other hand, FJS performs well on periodic patterns, precisely because of its KMP component.

The advantage of FJS over TBM seems to be related to the extra time involved in TBM's shift logic; while the advantage over BMH seems to be a compound of an improved shift methodology together with the effect of the KMP component.

In summary: we presented the hybrid algorithm FJS which combines the benefits of KMP and BMS. It requires $\Theta(m+k)$ time and space for preprocessing, and finds all matches in at most $3n - 2m$ letter comparisons. We also compared FJS with some of the fastest algorithms available. The results suggest that FJS is competitive with these algorithms in general, and that for $k \geq 8$ it is the algorithm of choice.

References

1. Robert S. Boyer & J. Strother Moore, **A fast string searching algorithm**, *Commun. Assoc. Comput. Mach. 20–10* (1977) 762–772.
2. Christian Charras & Thierry Lecroq, *Exact String Matching Algorithms*, Laboratoire d'Informatique, Université de Rouen (1997): `http://www-igm.univ-mlv.fr/~lecroq/string/index.html`.
3. Richard Cole & Ramesh Hariharan, **Tighter bounds on the exact complexity of string matching**, *Proc.* 33rd *IEEE Symp. Found. Comp. Sci.* (1992) 600–609.
4. Richard Cole, Ramesh Hariharan, Michael S. Paterson & Uri Zwick, **Tighter lower bounds on the exact complexity of string matching**, *SIAM J. Comput. 24–1* (1995) 30–45.
5. Livio Colussi, **Correctness and efficiency of pattern matching algorithms** *Information & Computation 95* (1991) 225–251.
6. Livio Colussi, **Fastest pattern matching in strings**, *J. Algs. 16–2* (1994) 163–189.
7. Livio Colussi, Zvi Galil & Raffaele Giancarlo, **On the exact complexity of string matching**, *Proc.* 31st *IEEE Symp. Found. Comp. Sci.* (vol. I) (1990) 135–143.
8. Maxime Crochemore, Artur Czumaj, Leszek Gąsieniec, Stefan Jarominek, Thierry Lecroq, Wojciech Plandowski & Wojciech Rytter, **Speeding up two string-matching algorithms**, *Algorithmica 12* (1994) 247–267.
9. Maxime Crochemore, Christophe Hancart & Thierry Lecroq, *Algorithmique du Texte*, Vuibert, Paris (2001).
10. Zvi Galil & Raffaele Giancarlo, **On the exact complexity of string matching: lower bounds**, *SIAM J. Comput. 20–6* (1991) 1008–1020.
11. Zvi Galil & Raffaele Giancarlo, **On the exact complexity of string matching: upper bounds**, *SIAM J. Comput. 21–3* (1992) 407–437.
12. Michael Hart, *Project Gutenberg*, Project Gutenberg Literary Archive Foundation (2004): `http://www.gutenberg.net`.
13. R. Nigel Horspool, **Practical fast searching in strings**, *Software – Practice & Experience 10–6* (1980) 501–506.
14. Andrew Hume & Daniel Sunday, **Fast string searching**, *Software – Practice & Experience 21–11* (1991) 1221–1248.
15. Christopher G. Jennings, *A Linear-Time Algorithm for Fast Exact Pattern Matching in Strings*, M. Sc. thesis, McMaster University (2002) 97 pp.
16. Donald E. Knuth, James H. Morris & Vaughan R. Pratt, **Fast pattern matching in strings**, *SIAM J. Comput. 6–2* (1977) 323–350.
17. Thierry Lecroq, **Experimental results on string matching algorithms**, *Software – Practice & Experience 25–7* (1995) 727–765.
18. Thierry Lecroq, *New Experimental Results on Exact String-Matching*, Rapport LIFAR 2000.03, Université de Rouen (2000).
19. James H. Morris & Vaughan R. Pratt, *A Linear Pattern-Matching Algorithm*, Tech. Rep. 40, University of California, Berkeley (1970).
20. Bill Smyth, *Computing Patterns in Strings*, Pearson Addison-Wesley (2003) 423 pp.
21. Daniel M. Sunday, **A very fast substring search algorithm**, *Commun. Assoc. Comput. Mach. 33–8* (1990) 132–142.