Volume 5, issue 4 December 2007 ISSN 1570-8667

ELSEVIER

JOURNAL OF DISCRETE ALGORITHMS

Selected papers from Combinatorial Pattern Matching 2005
Editors
Kumsoo Park
Laurent Mouchard

AVAILABLE AT
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT

Available online at www.sciencedirect.com

ScienceDirect

Journal of Discrete Algorithms 5 (2007) 682–695

JOURNAL OF DISCRETE ALGORITHMS

www.elsevier.com/locate/jda

A simple fast hybrid pattern-matching algorithm [☆]

Frantisek Franek [a], Christopher G. Jennings [b], W.F. Smyth [a,c,*]

[a] *Algorithms Research Group, Department of Computing & Software, McMaster University, Hamilton ON L8S 4K1, Canada*
[b] *School of Computing Science, Simon Fraser University, 8888 University Drive, Burnaby BC V5A 1S6, Canada*
[c] *School of Computing, Curtin University, GPO Box U1987, Perth WA 6845, Australia*

Received 5 December 2005; received in revised form 31 October 2006; accepted 23 November 2006

Available online 16 January 2007

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, apparently dominant for alphabet size above 15–20.

MSC: 68U15; 68R15

Keywords: String; Word; Pattern-matching; Algorithm

1. Introduction

Since 1977, with the publication of both the Boyer–Moore [2] and Knuth–Morris–Pratt [24] 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 pattern-matching literature has had two main foci:

1. reducing the number of letter comparisons required in the worst/average case (for example, [4–6,8,13–15], see also [29, pp. 219–225]);
2. reducing the time requirement in the worst/average case (for example, [7,9,20,21,30]).

[☆] A preliminary version of this paper appeared in Proc. 16th Annual Symp. Combinatorial Pattern Matching, LNCS 3537, Springer-Verlag, 2005. The authors were supported in part by grants from the Natural Sciences & Engineering Research Council of Canada.

[*] Corresponding author.
E-mail addresses: franek@mcmaster.ca (F. Franek), cjennings@acm.org (C.G. Jennings), smyth@computing.edu.au (W.F. Smyth).

1570-8667/$ – see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.jda.2006.11.004

This contribution resides in the second of these categories: in an effort to reduce processing time, we propose a mixture of Sunday's variant [30] of BM [2] with KMP [24,27]. Our goal is to combine the best/average case advantages of Sunday's algorithm (BMS) with the worst case guarantees of KMP. According to the experiments we have conducted, our new algorithm (FJS) is among the fastest in practice for the computation of 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. Moreover, based on $\Theta(m+k)$-time preprocessing, it guarantees that matching requires at most $3n - 2m$ letter comparisons and $\mathrm{O}(n)$ matching time.

Several comparative surveys of pattern-matching algorithms have been published over the years [10,21,25,26,29] and a useful website [3] is maintained that gives executable C code for the main algorithms (including preprocessing) 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. Next, extensions to the algorithm for large alphabets and patterns with don't-care letters are discussed in Section 3. Then in Section 4 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) [20];
- Sunday's algorithm (BMS) [30];
- Reverse Colussi (RC) [7];
- Turbo-BM (TBM) [9].

Finally, in Section 5, we draw conclusions from our experiments.

2. The new algorithm

Algorithm FJS searches for matches to $\boldsymbol{p} = \boldsymbol{p}[1..m]$ in $\boldsymbol{x} = \boldsymbol{x}[1..n]$ by shifting $\boldsymbol{p}$ from left to right along $\boldsymbol{x}$. At the start of each step, position $j = 1$ of $\boldsymbol{p}$ is aligned with a position $i \in 1..n-m+1$ in $\boldsymbol{x}$, so that position m of $\boldsymbol{p}$ is therefore aligned with position $i' = i + m - j$ in $\boldsymbol{x}$.

In KMP matching, $\boldsymbol{p}$ is matched left-to-right with $\boldsymbol{x}[i], \boldsymbol{x}[i+1]$, and so on until either a match with all of $\boldsymbol{p}$ is found or else a mismatch is located at some position $j \in 1..m$. In the case of a mismatch at $j > 1$, we say that a *partial match* has been determined with $\boldsymbol{p}[1..j-1]$. As matching proceeds, both i and j are incremented, thus maintaining the relationship $i' = i + m - j$, the basic invariant of Algorithm FJS.

The strategy of FJS is as follows:

1. Whenever no partial match of $\boldsymbol{p}$ with $\boldsymbol{x}[i..i+m-1]$ has been found, Sunday shifts (described below) are performed to determine the next position i' at which $\boldsymbol{x}[i'] = \boldsymbol{p}[m]$. When such an i' has been found, KMP matching is then performed on $\boldsymbol{p}[1..m-1]$ and $\boldsymbol{x}[i'-m+1..i'-1]$.
2. If a partial match of $\boldsymbol{p}$ with $\boldsymbol{x}$ has been found, KMP matching is continued on $\boldsymbol{p}[1..m]$.

The pseudocode shown below illustrates this overall strategy.

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
$j \leftarrow 1; i \leftarrow 1; i' \leftarrow m; m' \leftarrow m - 1$
while $i' \leqslant n$ do
    if $j \leqslant 1$ then

        – If no partial match with $\boldsymbol{p}$, perform Sunday shifts,
        – returning next position $i'$ such that $\boldsymbol{x}[i'] = \boldsymbol{p}[m]$
        SUNDAY-SHIFT($i'$)
```

```
– Reset invariant for KMP matching of $\boldsymbol{p}[1..m-1]$
$j \leftarrow 1; i \leftarrow i' - m'$
KMP-MATCH($m'$; $j, i$)

else

– Continue KMP matching of $\boldsymbol{p}[1..m]$
KMP-MATCH($m$; $j, i$)

– Restore invariant for next try (SUNDAY or KMP)
$j \leftarrow \beta'[j]; i' \leftarrow i + m - j$
```

As discussed in [22], the real rationale for the algorithm resides in the avoidance of Markov effects combined with efficient shifting. A match of $\boldsymbol{p}[m]$ with $\boldsymbol{x}[i']$ can in most circumstances reasonably be regarded as independent of a match (or mismatch) with $\boldsymbol{p}[1]$, where KMP matching begins. That is, we suppose that when $\boldsymbol{p}[1] = \boldsymbol{x}[i]$ but $\boldsymbol{p}[1..m] \neq \boldsymbol{x}[i..i']$, it is more likely that $\boldsymbol{p}[m] \neq \boldsymbol{x}[i']$ than $\boldsymbol{p}[2] \neq \boldsymbol{x}[i+1]$ because $\boldsymbol{p}[1]$ is better able to predict $\boldsymbol{x}[i+1]$ than $\boldsymbol{x}[i']$. When this supposition holds, we can detect partial matches more quickly by testing positions at opposite ends of the pattern than by testing positions in sequence. In addition, testing first at position m of $\boldsymbol{p}$ provides a very efficient mechanism for sliding $\boldsymbol{p}$ across $\boldsymbol{x}$ using the Sunday shift when $\boldsymbol{p}[m] \neq \boldsymbol{x}[i']$, which is expected to be the most common case.

Thus, in a sense, once a suitable i' has been found, the first half of FJS (executed when $j \leqslant 1$) just performs KMP matching in a 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 $2n - m$.

2.1. *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.

The hth 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* [29]—essentially, that $\Sigma = \{1, 2, \ldots, k\}$ 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 $\boldsymbol{x}[i']$, $\Delta[\boldsymbol{x}[i'+1]]$ 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 $i'+1$. The computation of Δ is straightforward (see Appendix A for C code and [10,29] for further discussion).

To describe the β' array, we introduce the *border* of a string $\boldsymbol{u}$: a proper prefix of $\boldsymbol{u}$ that is also a suffix of $\boldsymbol{u}$. The empty string is a border of every nonempty $\boldsymbol{u}$. Next define an array $\beta[1..m+1]$ in which $\beta[1] = 0$ and for every $j \in 2..m+1$, $\beta[j]$ is one plus 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 Appendix A for C code and [10,29] for further discussion.

2.2. *The algorithm*

The KMP parts of the algorithm need to compare position i of $\boldsymbol{x}$ with position j of $\boldsymbol{p}$, where j may be the extension of a partial match with $\boldsymbol{p}[1..j-1]$ or the result of an assignment $j \leftarrow \beta'[j]$. We show below the pseudocode for the KMP version that does matching of $\boldsymbol{p}[1..m-1]$:

```
KMP-MATCH($m'$; $j, i$)

while $j < m$ and $x[i] = p[j]$ do
    $i \leftarrow i + 1$; $j \leftarrow j + 1$
if $j = m$ then
    $i \leftarrow i + 1$; $j \leftarrow j + 1$; output $i - m$
```

The full $\boldsymbol{p}[m]$ version is as follows:

```
KMP-MATCH($m$; $j, i$)

while $j \leqslant m$ and $x[i] = p[j]$ do
    $i \leftarrow i + 1$; $j \leftarrow j + 1$
if $j > m$ then output $i - m$
```

Observe that after each execution of a KMP version, the last line of Hybrid Matching is executed, performing a KMP shift ($j \leftarrow \beta'[j]$) and reestablishing the invariant $i' = i + m - j$ required subsequently by SUNDAY-SHIFT:

```
SUNDAY-SHIFT($i'$)

while $x[i'] \neq p[m]$ do
    $i' \leftarrow i' + \Delta[x[i' + 1]]$
    if $i' > n$ then return
```

2.3. 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 $\Delta[\boldsymbol{x}[i' + 1]]$ is well defined for $i' = n$. There are no restrictions on the choice of sentinel except that it should be in Σ: it may occur anywhere in $\boldsymbol{x}$ and $\boldsymbol{p}$. If using a sentinel is undesirable, the algorithm can be modified without sacrificing efficiency by testing the final alignment at $\boldsymbol{x}[n - m + 1..n]$ as a special case outside of the main loop.

2.4. Letter comparisons

If, in Algorithm FJS, it is always true that $j > 1$, then the maximum number of letter comparisons is $2n - m$, identical to that of Algorithm KMP (see, for example, [29]). Suppose then that it is always true that $j \leqslant 1$. This means that in the worst case there may be as many as $n - m + 1$ executions of the match in SUNDAY-SHIFT. Then KMP-type letter comparisons could take place on the remaining $m - 1$ positions of the pattern over a text string of length $n - 1$ (under this hypothesis, the final letter of $\boldsymbol{x}$ would never be subject to KMP-type matching). Since Algorithm KMP performs at most $2n' - m'$ letter comparisons to find matches for a pattern of length m' in a string of length n', we find, substituting $n' = n - 1$, $m' = m - 1$, that 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 $\boldsymbol{p} = aba$, $\boldsymbol{x} = a^n$. Thus

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

It appears that the matching problem $\boldsymbol{p} = aba$, $\boldsymbol{x} = a^n$ is essentially the *only* one that yields worst case behaviour. Thus there is not a large price to be paid for the improved average-case efficiency of FJS. We remark that a preliminary version of this algorithm published in conference proceedings [12], though faster in practice than the one described here, did not in fact achieve the $3n - 2m$ bound given here.

3. Extensions

3.1. "Don't-care" approximate patterns

There has recently been revived interest in using extensions of exact pattern-matching algorithms to handle "don't-care" letters $*$ or letters $\lambda = [a, e, i, o, u]$ that match any one of a specified subset $\{a, e, i, o, u\}$ of "regular" letters [17–19].

[17] distinguishes, apparently for the first time, two forms of matching between strings that contain *indeterminate* letters—that is, letters (such as $*$ or λ) that match specified subsets of letters of a given alphabet Σ. One form of matching of indeterminate letters, called *quantum*, permits an indeterminate letter to match distinct letters in the same match—for example, $a*$ is allowed to match $*b$ even though this requires that $*$ match *both* a and b. On the other hand, *determinate* matching requires that an indeterminate letter should always match consistent subsets of letters in the same match: $[a, b]b$ cannot match $a[a, b]$ because it requires that $[a, b]$ match both a and b.

In [18] it is shown that, for both the quantum and indeterminate cases, variants of the BMS algorithm generally outperform the agrep-style algorithms [1,11,28,31]; in [19] it is shown that these benefits extend, rather more clearly, to corresponding variants of FJS. These results on BMS/FJS extensions thus to some degree confirm the results of this paper.

3.2. Preprocessing large alphabets

The alphabet-based preprocessing arrays of BM-type algorithms are their most useful feature, but they can be a source of trouble as well. We have already discussed one such caveat: that the alphabet must be indexed, but that is not the only pitfall. When the size of the alphabet is a significant fraction of the size of the text, preprocessing time can easily overwhelm any skip benefits gained while searching.

In the past, this has rarely been a problem. The typical application is searching natural language text, and the alphabets used to represent that text were usually of 8 bits or less. This is small enough that the preprocessing time is unlikely to be noticed regardless of text length. However, circumstances are changing: larger alphabets are increasingly common when dealing with natural language strings. Coding schemes such as Unicode make use of "wide characters" that may be up to 32 bits wide, with 16 bits being more common. But even a 16-bit alphabet is large enough to make it counterproductive to use FJS in many applications. Typed consecutively, the letters of a 16-bit alphabet would cover about 20 sheets of paper. To see any gain, the text being searched would need to be longer still. Using FJS with a 32-bit alphabet is not practical—nor even possible on the majority of present-day computers.

On the other hand Algorithm FJS—or nearly any BM-derived algorithm— can easily be extended to allow effective use of indexed alphabets of any practical size. The best- and worst-case number of letter comparisons remains unchanged by this extension. The key to this is the observation that the values stored in Δ are effectively upper bounds on the number of positions that can be safely skipped. If fewer positions are skipped, then no harm is done other than to force the algorithm to do a bit more work than needed—but even if the skip amount is fixed at 1 and the inequality $\boldsymbol{p}[m] \neq \boldsymbol{x}[i']$ always holds, the limit of $n - m + 1$ executions of the BM-type match is unchanged.

The extension works by using a smaller preprocessing array as a kind of hash table. First, choose a new size $t < k$ for the modified array $\Delta'[1..t]$, and define a hash function $f : \Sigma \to [1..t]$ with constant time and space complexity. Then for every $g \in 1..t$ and every $h \in 1..k$, the value stored in $\Delta'[g]$ is the smallest r such that $f(h) = g$ and $\Delta[h] = r$. The Δ' array can be computed in $\Theta(t + m)$ time and space with simple modifications to the standard algorithm for Δ. Finally, the completed extension replaces the lookup $\Delta[h]$ in Algorithm FJS with $\Delta'[f(h)]$.

As a practical example of this technique, we suppose that Σ is a wide character encoding with $k = 2^{16}$ or $k = 2^{32}$. We let $t = 2^8$, and define f as the masking function that keeps the lowest 8 bits of its input. Although the upper and lower bounds on time and space complexity are unchanged, the effect on average-case behaviour is harder to predict and will be highly sensitive to the patterns processed and the way the alphabet is indexed. For example, the subset of Unicode corresponding to the ASCII character set occurs in a single 256 character block. Searching a 16-bit Unicode string that consisted only of ASCII characters using the example just described would require exactly the same number of letter comparisons as unmodified FJS since no hash collisions would occur in Δ'.

4. Experimental results

We conducted a number of experiments to compare Algorithm FJS with other exact pattern matching algorithms known to be among the fastest in practice. As mentioned in Section 1, we selected the four algorithms BMH, BMS, RC, and TBM for this purpose.

Colussi presents two versions of the RC algorithm in [7]; we use the first version, which matches in $O(mn)$ time in certain pathological cases. The alternative version matches in $\Theta(n)$ time but requires additional computation after each match, and so is typically slower in practice. Tests with the alternative version did not produce significantly different results relative to FJS.

We note that, of the four algorithms tested against FJS, only TBM shares with FJS the property of having an order n upper bound in the worst case; in fact, TBM requires at most $2n$ letter comparisons as opposed to $3n - 2n$ for FJS, but in our tests it consistently runs slower than FJS.

Implementations of the four competing algorithms were adapted from [3], which provides well-tested C code for a large number of exact pattern matching algorithms. This gave a standard which we mimicked closely in implementing FJS, yielding code of consistent quality and thereby helping to ensure fairness in testing.

Our implementations vary from [3] in three significant ways. First, pattern preprocessing was moved inline to remove the function call overhead from our execution times. Second, the arrays used for preprocessing, which were allocated on the stack, were replaced with statically allocated arrays. This was done to prevent the stack from being dynamically extended by the operating system and distorting our results. Finally, calls to the C library functions *memcmp*() and *memset*(), which were made by some implementations, were replaced with equivalent simple loops. The library calls are optimized for large arrays, while most of our patterns are relatively short. Changing the call made the relevant competitors four to five times faster in our tests.

These C functions are optimized for operation on large memory regions. Our tests feature relatively small pattern lengths, so these functions perform poorly in our tests. Eliminating them increased execution speed by four to five times, maintaining our consistent quality imperative.

To time the algorithms, we made use of high-frequency hardware clocks that are available on the majority of modern computer systems. A run of each implementation for a given text and pattern was repeated 20 times, and the fastest time was taken as the result. The fastest time is closest to the minimum time that the run would have taken independent of interrupts, cache misses, or other external effects. All timings include preprocessing as well as the string search algorithm itself.

The results presented here were taken from the system that, of those available, provided the most precise and accurate timing hardware—namely an AMD Athlon 2500+ PC with 512 MB RAM, running Windows 2000 (SP4), and using software compiled with version 12.00.8168 of Microsoft's C/C++ compiler. However, the resulting trends were highly stable across seven environments using a variety of hardware, operating system, and compiler vendor combinations (see Table 1).

Table 1

CPU	OS	Compiler
AMD Athlon 2500+	Windows 2000 (SP4)	MS C/C++ 12.00
AMD Athlon 1000 MHz	Windows 98	MS C/C++ 12.00
AMD Athlon 1000 MHz	Windows 98	DJGPP 2.03
AMD Athlon 1000 MHz	Red Hat Linux 7.1	GNU C++ 2.96
Hitachi SuperH SH-3 133 MHz	Windows CE H/PC 2.11	MS Embedded C++ 3.0
Motorola MC68EC030 25 MHz	AmigaOS 3.1	SAS C 6.0
Motorola MC68EC030 25 MHz	AmigaOS 3.1	GNU C++ 2.96

4.1. Test data

The primary source of text strings for these experiments was Project Gutenberg [16], an online source of copyright-expired or otherwise freely distributable texts. A set of 2434 texts was obtained, consisting of texts published by Project Gutenberg on or before August 16, 2001. Of these, some texts were present multiple times owing to corrected editions being published at a later date. In these cases, all but the latest editions were removed from the set. Of the remaining texts, 1000 were selected at random to make up the final corpus. The corpus contained a total of 446 504 073 letters, with individual texts ranging from 10 115 to 4 823 268 letters in length. The distribution of lengths is left-skewed as there are many more short files than long ones, a similar distribution to many practical applications.

A second corpus was constructed from a version of the Human Genome Project's map of the first human chromosome. Filtering the original file to remove non-DNA data left a string of 210 992 564 letters drawn from the nucleotide alphabet $\{A, T, C, G\}$. This was transformed into a bit string by mapping each A, T, C, or G to 00, 01, 10, or 11, respectively. Random substrings of the resulting binary string were then used to construct strings drawn from alphabets of size varying by powers of 2 from 2 to 64.

4.2. 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 pattern set consisted of the following (where ␣ represents the space symbol):

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

These were selected by finding the seven most frequent substrings of length six in a random sample of 200 texts from our primary corpus. These patterns occur 3 366 899 times in the full corpus. Fig. 1 graphs the performance of the tested algorithms for the high-frequency 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. Strictly speaking, as the selection of patterns was not randomized, it is not statistically valid to infer that these results apply to all possible frequent patterns. However, considering all our results together, we are confident that the results given here are typical.

For the moderate frequency set, we randomly selected 7 words of at least length 6 from those words sharing an expected frequency of 4 instances per 10 000 words. Actual match frequency may vary since patterns may occur often

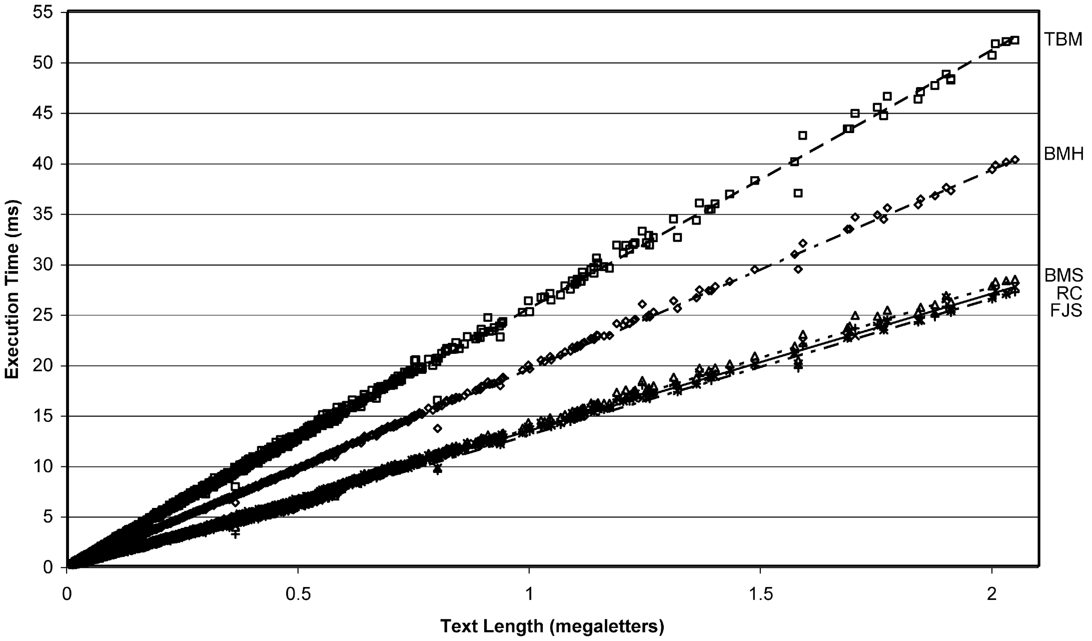

Fig. 1. Execution time versus text length for high-frequency pattern set.

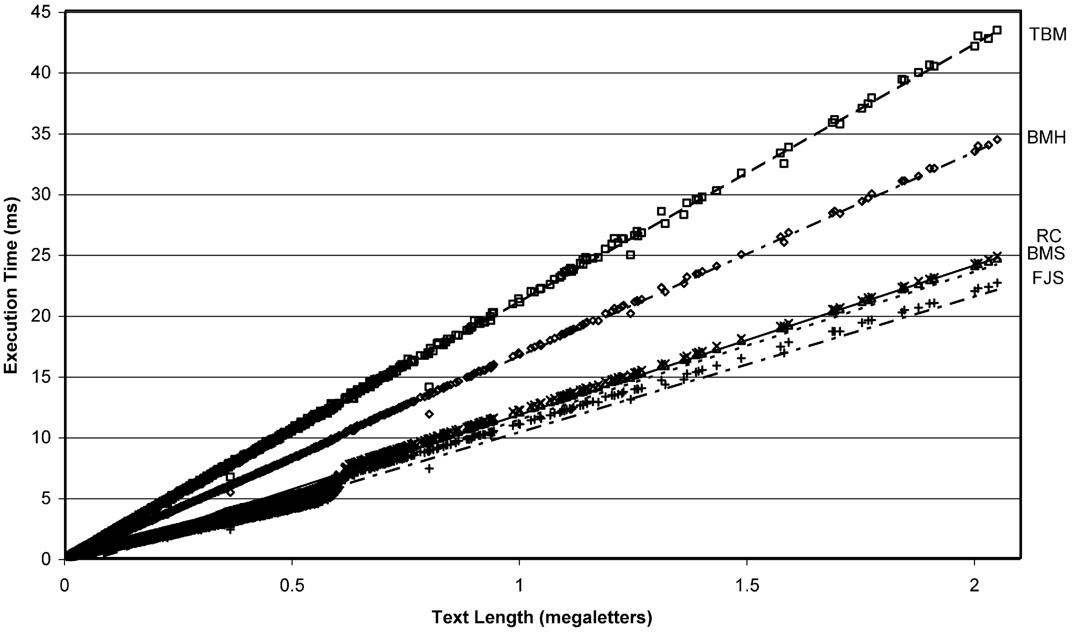

Fig. 2. Execution time versus text length for moderate-frequency pattern set.

within other words, but this tends to occur in practice as well. After truncating long words at six letters the final set was:

better enough govern public someth system though

These patterns occur 266 792 times in our corpus, or about one thirteenth as often as the patterns in the high-frequency set. Fig. 2 graphs the results of this test; as in the high-frequency test, the execution time is the total time for all patterns on a given text.

4.3. Pattern length

Pattern sets were constructed consisting of nine patterns for each possible length from three to nine. These were selected using the methods similar to the moderate frequency experiment 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. As above, the actual frequency of some patterns, shorter patterns in particular, will be higher due to their occurring more frequently within other words. The sets, and their respective match frequencies are shown in Table 2.

Fig. 3 graphs the results from this test. The time for each pattern length is the total time for searching for all nine patterns in the relevant set in each of the texts of the primary corpus.

The trends of FJS and BMS suggested that they are asymptotically the same. To investigate further, we tested 90 patterns with lengths from 25 to 175, randomly selected from our corpus. We found that FJS and BMS became indistinguishable from about $m = 100$ onward. RC's performance was burdened by its $\mathrm{O}(m^2)$ preprocessing time in

Table 2

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, political, situation, sometimes, necessary, therefore*

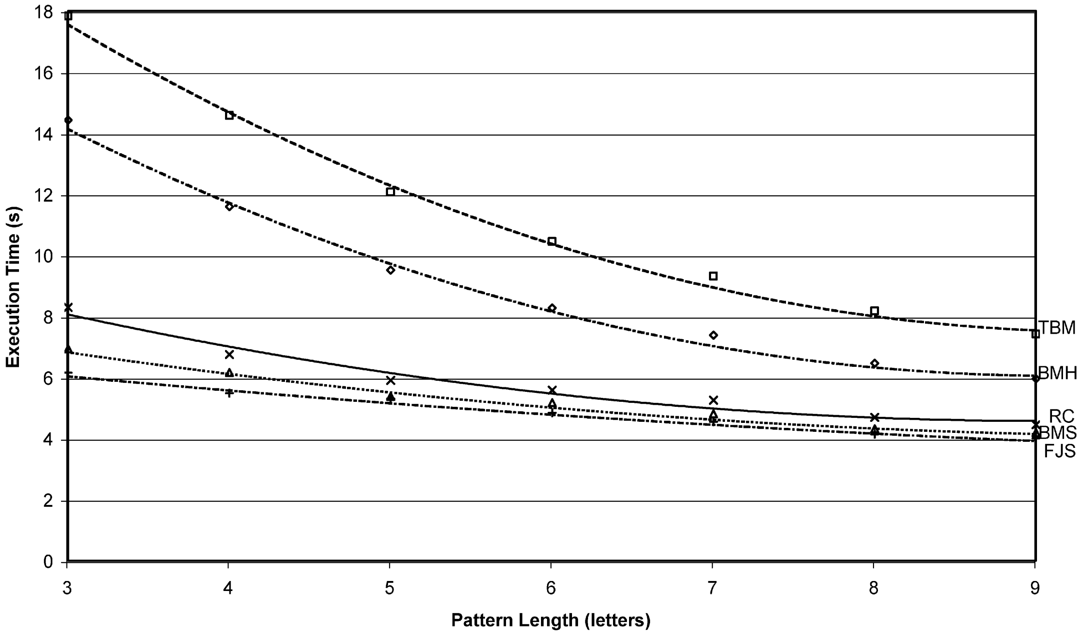

Fig. 3. Execution time versus pattern length.

this test. RC was second last at $m = 25$, and last from $m = 30$ on. At $m = 175$, it was already more than an order of magnitude slower than its nearest competitor.

4.4. Alphabet size

The secondary corpus of DNA-derived texts was used to test the effect of alphabet size on the algorithms. This corpus consists of six texts of 500 000 letters each, each text drawn from an alphabet of a different size from 2, 4, 8, 16, 32, and 64. Each text was searched with 20 patterns of length six, each pattern selected at random from the source text. The results, seen in Fig. 4, suggest that Algorithm FJS may be a poor choice for $k < 15$—thus not optimal for DNA sequences ($|\Sigma| = 4$), but suitable for proteins composed of amino acids ($|\Sigma| = 20$).

4.5. Pathological cases

It is well known that periodic strings can induce theoretical worst-case behaviour in pattern-matching algorithms. For FJS, $\boldsymbol{x} = a^n$, $\boldsymbol{p} = aba$ maximizes the comparison count for a given text size. We compared performance on this text and pattern for values of n from 10 000 to 100 000 at 10 000 letter intervals. The rank order, from fastest to slowest, was consistently RC, BMH, BMS, FJS, TBM, with RC timing 41% faster than FJS on average. These results are graphed in Fig. 5.

On the other hand, patterns of the form $\boldsymbol{x} = a^n$, $\boldsymbol{p} = a^m$ (and in particular, $m = [n/2]$) maximize comparisons for many non-linear BM variants, including BMH and BMS. Fixing the text as $\boldsymbol{x} = a^{100\,000}$, we timed the five algorithms for $\boldsymbol{p} = a^3$ through $\boldsymbol{p} = a^9$. The results, shown in Fig. 6, showcase the $O(n)$ matching time guarantees of FJS and TBM and indicate a clear preference for FJS.

The worst case for the original BM algorithm can be triggered with strings of the form $\boldsymbol{x} = (a^k b)^r$, $\boldsymbol{p} = a^{k-1}ba^{k-1}$. As in the previous case, both the pattern and the text are highly repetitive. We performed two final tests using string pairs of this kind. In the first test, shown in Fig. 7, we held r constant at 100 000 and varied k from 2 through 20. In the second test, shown in Fig. 8, we held k at 10 and varied r from 100 000 through 500 000. Both cases showed a 15–20% advantage for FJS over BMS and RC, which were indistinguishable, and by a large margin over all others.

5. 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).

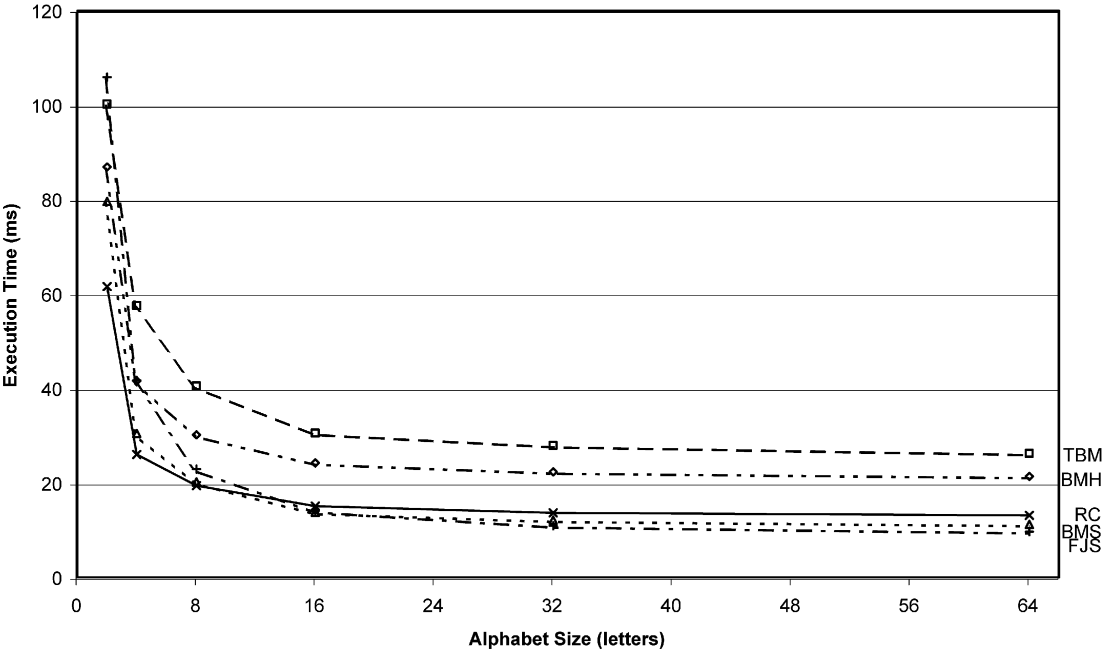

Fig. 4. Execution time versus alphabet size.

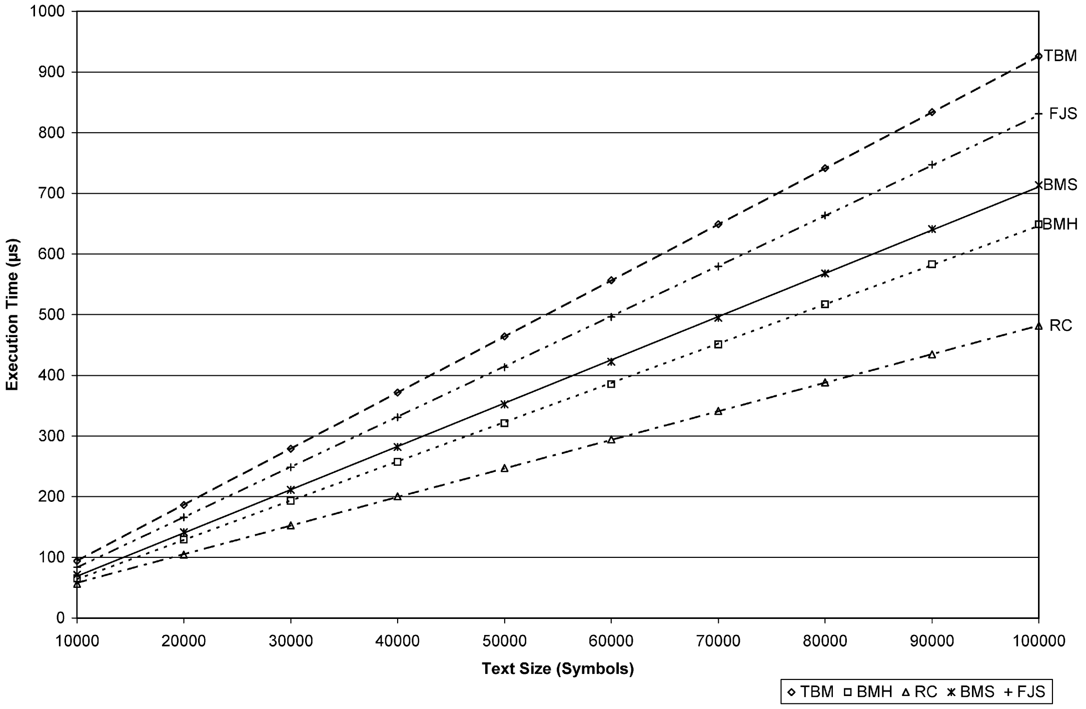

Fig. 5. Execution time versus text length for $\boldsymbol{x} = a^n$, $\boldsymbol{p} = aba$.

Over contexts (C1) and (C2), and for English texts and patterns, FJS was uniformly superior to its competitors, with an up to 10% advantage over BMS and RC, the closest adversaries. For very long patterns, it appears that FJS and BMS have approximately the same behaviour, a not unexpected result since the Sunday skip loop dominates.

In terms of (C3), for small alphabets the statistical incidence of $\boldsymbol{p}[m] = \boldsymbol{x}[i']$ will be high, and so KMP will be invoked often, thus slowing FJS with respect to BMS and RC. However, for $k \geqslant 15$ or so, FJS appears to again be the algorithm of choice, again with a time advantage of 5–10%. We remark again that an alternate version of FJS runs faster in practice than the one tested here [12], and in fact appears to be dominant for $k \geqslant 8$.

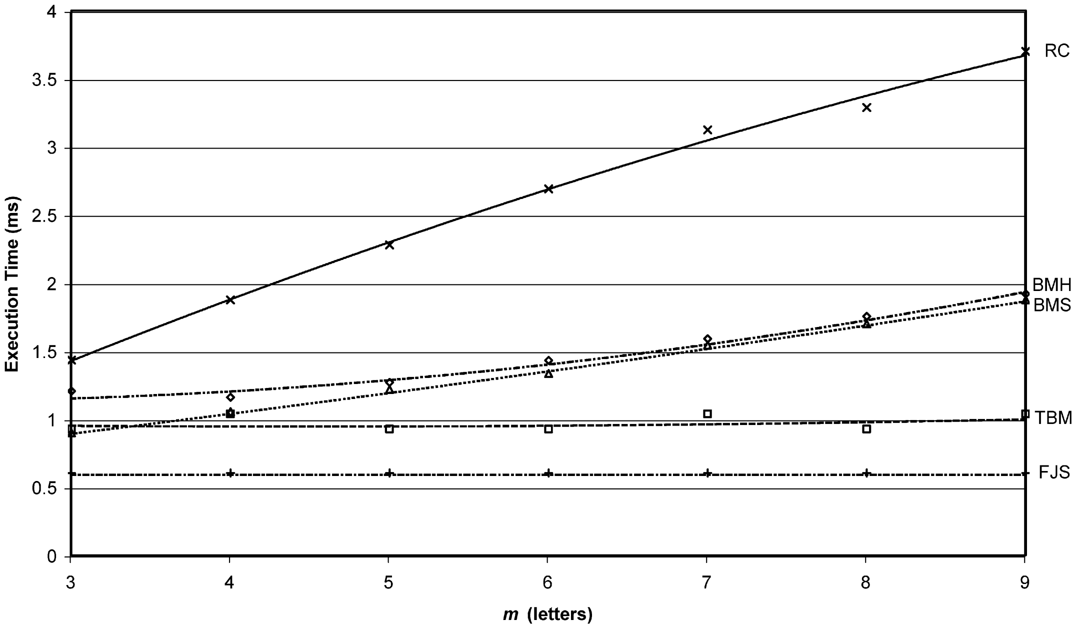

Fig. 6. Execution time versus pattern length for $\boldsymbol{x} = a^{100\,000}$, $\boldsymbol{p} = a^m$.

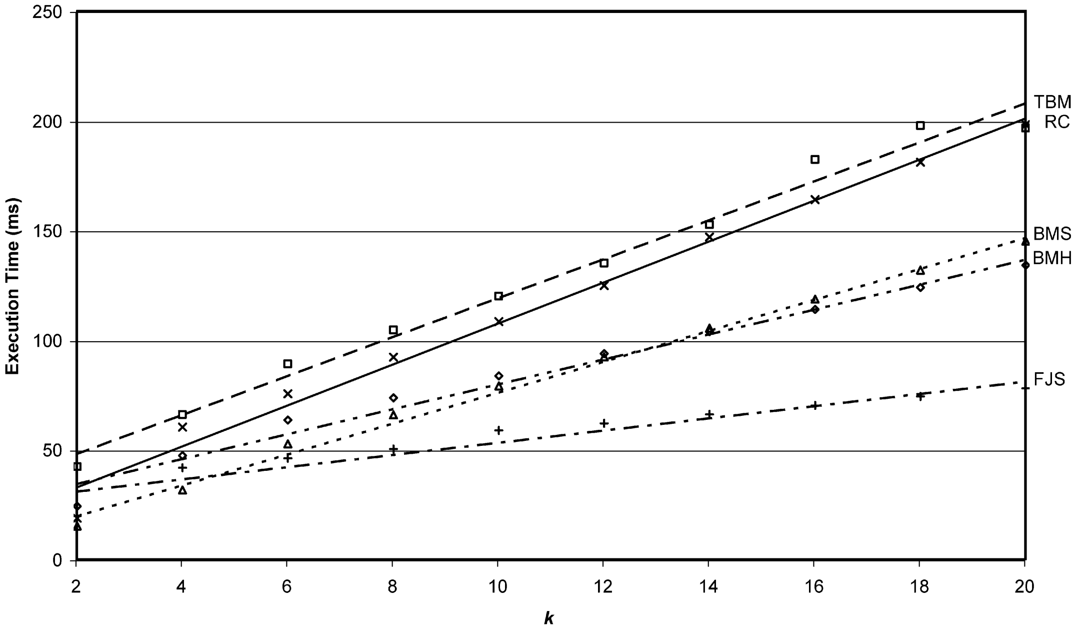

Fig. 7. Execution time versus k for $\boldsymbol{x} = (a^k b)^{100000}$, $\boldsymbol{p} = a^{k-1}ba^{k-1}$.

The overall speed advantage of FJS probably relates to its efficient average-case processing of the most common case (initial mismatch):

- use of a tight skip loop to find alignments with an initial match and avoid repeated settings of $j = 1$;
- Markov independence of position m and position j in the pattern $\boldsymbol{p}$.

As discussed in [22], the first of these factors, slight as it is, 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 provides no advantage (that is, the portion of the pattern $\boldsymbol{p}$ that partially matches the text always has only an empty border). On the other hand, FJS performs well on periodic patterns, precisely because of its KMP component.

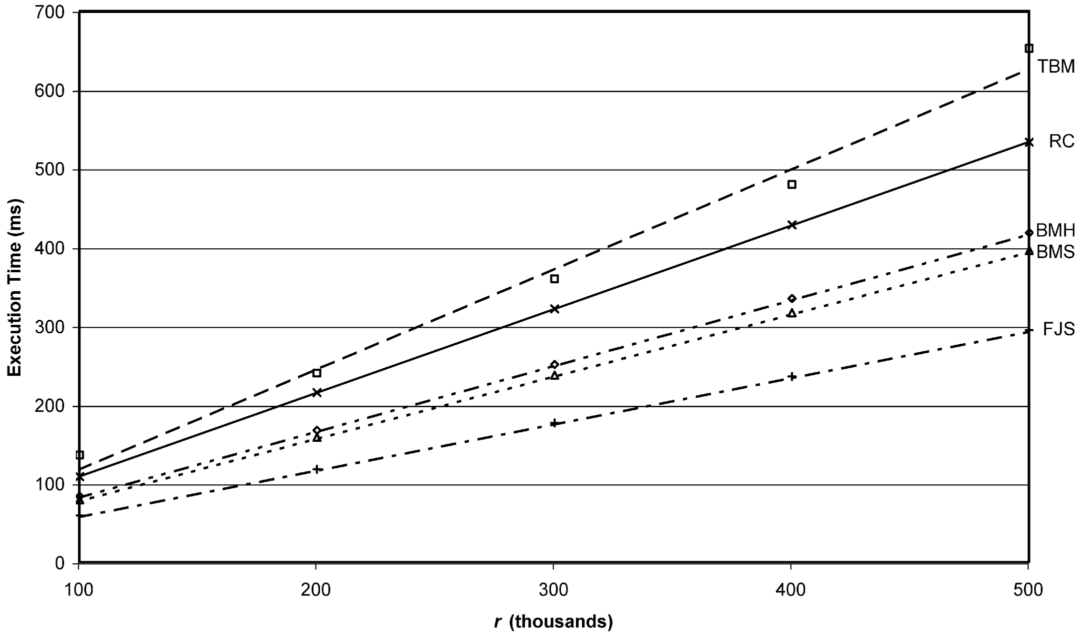

Fig. 8. Execution time versus r for $\boldsymbol{x} = (a^{10}b)^r$, $\boldsymbol{p} = a^9ba^9$.

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 a hybrid exact pattern-matching 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 presented extensions of FJS that encourage its use in a wider variety of practical contexts: approximate pattern matching with "don't-care" letters, and large alphabets. Finally, we compared FJS with some of the fastest algorithms available. The results suggest that FJS is competitive with these algorithms in general, and that for $k \geqslant 8$ it is the algorithm of choice.

Appendix A. An implementation

What follows is a straightforward implementation of FJS in the C programming language. The null character used to terminate strings in C provides the needed sentinel. This and related code can be downloaded from [23], which also provides an interactive animation of the algorithm.

```
#include <stdio.h>
#include <string.h>

typedef unsigned char CTYPE;   // type of alphabet letters

#define ALPHA      (256)       // alphabet size
#define MAX_PATLEN (100)       // maximum pattern length

int betap[ MAX_PATLEN+1 ];
int Delta[ ALPHA ];

void output( int pos ) {
    static int matches = 0;
    printf( "match %d found at position %d\n", ++matches, pos );
}

void makebetap( const CTYPE* p, int m ) {
    int i = 0, j = betap[0] = -1;
    while( i < m ) {
        while( (j > -1) && (p[i] != p[j]) )
            j = betap[j];
        if( p[++i] == p[++j] )
            betap[i] = betap[j];
        else
            betap[i] = j;
    }
}

void makeDelta( const CTYPE* p, int m ) {
    int i;

    for( i = 0; i < ALPHA; ++i )
        Delta[i] = m + 1;
    for( i = 0; i < m; ++i )
        Delta[ p[i] ] = m - i;
}

void FJS( const CTYPE* p, int m, const CTYPE* x, int n ) {
    if( m < 1 ) return;
    makebetap( p, m );
    makeDelta( p, m );

    int i = 0, j = 0, mp = m-1, ip = mp;
    while( ip < n ) {
        if( j <= 0 ) {
            while( p[ mp ] != x[ ip ] ) {
                ip += Delta[ x[ ip+1 ] ];
                if( ip >= n ) return;
            }
            j = 0;
            i = ip - mp;
            while( (j < mp) && (x[i] == p[j]) )
                { ++i; ++j; }
            if( j == mp ) {
                output( i-mp );
                ++i; ++j;
            }
            if( j <= 0 )
                ++i;
            else
                j = betap[j];
        } else {
            while( (j < m) && (x[i] == p[j]) )
                { ++i; ++j; }
            if( j == m )
                output( i-m );
            j = betap[j];
        }
        ip = i + mp - j;
    }
}

int main( int argc, char** argv ) {
    int m;
    if( argc == 3 ) {
        if( ( m = strlen( argv[2] )) <= MAX_PATLEN )
            FJS( (CTYPE*) argv[2], m,
                 (CTYPE*) argv[1], strlen( argv[1] ) );
        else
            printf( "Recompile with MAX_PATLEN >= %d\n", m );
    } else
        printf( "Usage: %s text pattern\n", argv[0] );
    return 0;
}
```

References

[1] R.A. Baeza-Yates, G. Gonnet, A new approach to text searching, Commun. Assoc. Comput. Mach. 35 (1992) 74–82.

[2] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Commun. Assoc. Comput. Mach. 20 (1977) 762–772.

[3] C. Charras, T. Lecroq, Exact string matching algorithms, Laboratoire d'Informatique, Université de Rouen, 1997, http://www-igm.univ-mlv.fr/lecroq/string/index.html.

[4] R. Cole, R. Hariharan, Tighter bounds on the exact complexity of string matching, in: 33rd Symp. Found. Comp. Sci., Pittsburgh Pennsylvania, 24–27 October 1992, IEEE, 1992, pp. 600–609.

[5] R. Cole, R. Hariharan, M.S. Paterson, U. Zwick, Tighter lower bounds on the exact complexity of string matching, SIAM J. Comput. 24 (1995) 30–45.

[6] L. Colussi, Correctness and efficiency of the pattern matching algorithms, Information and Computation 95 (1991) 225–251.

[7] L. Colussi, Fastest pattern matching in strings, J. Algorithms 16 (1994) 163–189.

[8] L. Colussi, Z. Galil, R. Giancarlo, On the exact complexity of string matching, in: 31st Symp. Found. Comp. Sci. vol. I, St. Louis, Missouri, 22–24 October 1990, IEEE, 1990, pp. 135–144.

[9] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, W. Rytter, Speeding up two string-matching algorithms, Algorithmica 12 (1994) 247–267.

[10] M. Crochemore, C. Hancart, T. Lecroq, Algorithmique du Texte, Vuibert, Paris, 2001.

[11] B. Dömölki, A universal computer system based on production rules, BIT 8 (1968) 262–275.

[12] F. Franek, C.G. Jennings, W.F. Smyth, A simple fast hybrid pattern-matching algorithm, in: A. Apostolico, M. Crochemore, K. Park (Eds.), 16th Annual Symp. Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 3537, Springer-Verlag, 2005, pp. 288–297.

[13] Z. Galil, On improving the worst case running time of the Boyer–Moore string matching algorithm, CACM 22–9 (1979) 505–508.

[14] Z. Galil, R. Giancarlo, On the exact complexity of string matching: Lower bounds, SIAM J. Comput. 20 (1991) 1008–1020.

[15] Z. Galil, R. Giancarlo, On the exact complexity of string matching: Upper bounds, SIAM J. Comput. 21 (1992) 407–437.

[16] M. Hart, Project Gutenberg, Project Gutenberg Literary Archive Foundation, 2004, http://www.gutenberg.net.

[17] J. Holub, W.F. Smyth, Algorithms on indeterminate strings, in: 14th Australasian Workshop on Combinatorial Algorithms, 2003, pp. 36–45.

[18] J. Holub, W.F. Smyth, S. Wang, Fast pattern-matching on indeterminate strings, JDA (2006), in press.

[19] J. Holub, W.F. Smyth, S. Wang, Hybrid pattern-matching algorithms on indeterminate strings, in: J. Daykin, M. Mohamed, K. Steinhöfel (Eds.), London Algorithmics and Stringology 2006, in: Texts in Algorithmics, King's College London, 2006, in press.

[20] R.N. Horspool, Practical fast searching in strings, Software— Practice & Experience 10 (1980) 501–506.

[21] A. Hume, D. Sunday, Fast string searching, Software—Practice & Experience 21 (1991) 1221–1248.

[22] C.G. Jennings, A linear-time algorithm for fast exact pattern matching in strings, Master's thesis, McMaster University, 2002.

[23] C.G. Jennings, Algorithm FJS, 2005, http://www.sfu.ca/~cjenning/fjs/index.html.

[24] D.E. Knuth, J.H. Morris, V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (1977) 323–350.

[25] T. Lecroq, Experimental results on string matching algorithms, Software—Practice & Experience 25 (1995) 727–765.

[26] T. Lecroq, New experimental results on exact string-matching, Rapport LIFAR 2000.03, Université de Rouen, 2000.

[27] J.H. Morris, V.R. Pratt, A linear pattern-matching algorithm, Tech. Rep. 40, University of California, Berkeley, 1970.

[28] G. Navarro, M. Raffinot, A bit-parallel approach to suffix automata: fast extended string matching, in: 9th Annual Symp. Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 1448, Springer-Verlag, 1998, pp. 14–33.

[29] B. Smyth, Computing Patterns in Strings, Addison Wesley Pearson, 2003.

[30] D.M. Sunday, A very fast substring search algorithm, Commun. Assoc. Comput. Mach. 33 (1990) 132–142.

[31] S. Wu, U. Manber, Fast text searching allowing errors, Commun. Assoc. Comput. Mach. 35 (1992) 83–91.