An Adaptive Hybrid Pattern-Matching Algorithm on Indeterminate Strings*

W. F. Smyth[1,2] and Shu Wang[1]

[1] Algorithms Research Group, Department of Computing & Software
McMaster University, Hamilton ON L8S 4K1, Canada
{smyth,shuw}@mcmaster.ca

[2] Digital Ecosystems & Business Intelligence Institute
Curtin University, GPO Box U1987, Perth WA 6845, Australia
smyth@computing.edu.au

Abstract. We describe a hybrid pattern-matching algorithm that works on both regular and indeterminate strings. This algorithm is inspired by the recently proposed hybrid algorithm FJS [11] and its indeterminate successor [15]. However, as discussed in this paper, because of the special properties of indeterminate strings, it is not straightforward to directly migrate FJS to an indeterminate version. Our new algorithm combines two fast pattern-matching algorithms, ShiftAnd and BMS (the Sunday variant of the Boyer-Moore algorithm), and is highly adaptive to the nature of the text being processed. It avoids using the border array, therefore avoids some of the cases that are awkward for indeterminate strings. Although not always the fastest in individual test cases, our new algorithm is superior in overall performance to its two component algorithms — perhaps a general advantage of hybrid algorithms.

1 Introduction

String pattern-matching has been studied extensively for many years because of the fundamental role it plays in many areas: the operation of a text editor or compiler, bioinformatics, data compression, firewall interception, and so on. Two main approaches have been proposed for computing all the occurrences of a given nonempty pattern $p = p[1..m]$ in a given nonempty text $x = x[1..n]$. One is the use of window-shifting techniques to skip over sections of text [17, 8], the other the use of the bit-parallel processing capability of computers to achieve fast processing [10, 23, 7, 18]. For more complete descriptions of various string matching algorithms, see [19, 9, 20].

Driven by applications in DNA sequence analysis and search engine techniques, indeterminate pattern-matching (IPM) is becoming more and more widely used. But for this modifications have to be made. An intuitive approach to IPM is to make use of exact pattern-matching algorithms and make necessary changes. Some pattern-matching algorithms that use bit-array methods such as ShiftAnd[23] and BNDM [18] can be adapted to IPM. On the other hand, efforts have also been made to develop indeterminate pattern-matching algorithms that are based on fast window-shifting algorithms such as BMS (the Sunday variant of the Boyer-Moore algorithm) [14] and FJS [15]. In this paper, we present a new fast algorithm that not only works on regular strings but also on indeterminate strings — it inherits from the BMS and ShiftAnd algorithms, while exceeding both of them in overall performance.

* Supported in part by grants from the Natural Sciences & Engineering Research Council of Canada. The authors express their gratitude to three anonymous referees, whose comments have materially improved the quality of this paper.

We believe that this paradigm will lead to the design of other very efficient IPM algorithms with the ability to flip-flop seamlessly between two or more methods, in response to the changing nature of local segments of the text.

2 Preliminaries

A ***string*** x is a finite sequence of ***letters*** drawn from a set Σ called an ***alphabet***. Let λ_i, $|\lambda_i| \geq 2$, $1 \leq i \leq m$, be pairwise distinct subsets of the alphabet Σ. We form a new alphabet $\Sigma' = \Sigma \cup \{\lambda_1, \lambda_2, .., \lambda_m\}$ and define a new relation ***match*** ($\approx$) on Σ' as follows:

- for every $\mu_1, \mu_2 \in \Sigma$, $\mu_1 \approx \mu_2$ if and only if $\mu_1 = \mu_2$;
- for every $\mu \in \Sigma$ and every $\lambda \in \Sigma' - \Sigma$, $\mu \approx \lambda$ and $\lambda \approx \mu$ if and only if $\mu \in \lambda$;
- for every $\lambda_i, \lambda_j \in \Sigma' - \Sigma$, $\lambda_i \approx \lambda_j$ if and only if $\lambda_i \cap \lambda_j \neq \emptyset$.

In a string x on an alphabet Σ', a position i is said to be ***indeterminate*** iff $x[i] \in \Sigma' - \Sigma$, and $x[i]$ itself is said to be an ***indeterminate letter***. A string that may contain indeterminate letters is said to be ***indeterminate*** (or ***generalized*** [5]). Two indeterminate strings x and y are said to ***match*** iff they are of the same length and the letters in corresponding positions match.

Indeterminate strings can arise in DNA and amino acid sequences as well as in cryptoanalysis applications and the analysis of musical texts. A simple example of an indeterminate letter is the don't-care letter $*$ which matches any other letter in the alphabet.

We identify three models of IPM in increasing order of sophistication:

(M1) The only indeterminate letter is the don't-care $*$, whose occurrences may be in either pattern or text, or both.
(M2) Arbitrary indeterminate letters can occur in either pattern or text, but not both.
(M3) Indeterminate letters can occur in both pattern and text.

In the case of (M2), observe that it does not matter whether the indeterminate letters occur in p or in x: the processing required to determine a match will be essentially the same. The techniques used for matching in the various cases are discussed in some detail below in Subsection 4.1.

In addition, two different constraints can be imposed on IPM:

- Quantum (**q**). Allow an indeterminate letter to match two or more distinct letters during a single matching process.
- Determinate (**d**). Restrict each indeterminate letter to be assigned to only one regular letter during a single matching process.

For example, given two strings $u = 551, v = 121$ including one indeterminate letter $5 = \{1, 2\}$, does $u \approx v$? The answer is yes in quantum pattern-matching and no in determinate pattern-matching, because we require that 5 first match 1 and then match 2 in a single match between 551 and 121.

Combining the three models and the two constraints q and d, we identify six interesting versions of IPM:

$$\text{M1q, M1d, M2q, M2d, M3q, M3d.} \tag{1}$$

3 Nontransitivity of Indeterminate Matching

In this section we briefly discuss a central problem that arises in IPM due to the possible nontransitivity of the match relation: in the example considered above, $1 \approx 5$ and $5 \approx 2$ does not imply $1 \approx 2$.

To describe the consequences of nontransitivity, recall that a *border* of x is any proper prefix of x that equals a suffix of x. For a string $x[1..n]$, an array $\beta[1..n]$ is called the *border array* of x iff for $i = 1, 2, ..., n$, $\beta[i]$ gives the length of the longest border of $x[1..i]$. The classic border array algorithm is given in [6], variants for indeterminate strings can be found in [13].

A great many of the exact pattern-matching algorithms (for example, Knuth-Morris-Pratt [17], Boyer-Moore [8], and their numerous variants) make use of the border array of the pattern or some version of it. The trouble is that for indeterminate strings, the nontransitivity of matching causes essential properties of the border array to fail [22], as we now demonstrate by example.

Index	1 2 3 4 5 6 7
x	$\cdots\ a\ a\ b\ b\ a\ b\ b \cdots$
p	$a\ *\ *\ b\ a\ *\ a$
$1st$ Shift	$a\ *\ *\ b\ a \cdots$
$2nd$ Shift	$a\ *\ * \cdots$
$3rd$ Shift	$a\ * \cdots$

Table 1. First example of the nontransitivity effect

Table 1 shows KMP pattern-matching of p against x. The first six positions of p match x, but there is a mismatch in position 7. According to the traditional definition of border, the longest border of $p[1..6]$ is $a**b \approx *ba*$, the second-longest border is $a*$ and the third is $a \approx *$. KMP performs shifts according to the borders of p in decreasing order of length, as shown by the shifts in the table. Observe however that if we perform a shift according to the longest border, aligning $p[1..4]$ with $x[3..6]$, we will then have letter a aligned with b in position 3. So indeterminate strings have the following property as opposed to traditional strings:

Proposition 1 *Shifting the indeterminate pattern p to the right in x according to the longest border does not guarantee a prefix match.*

Moreover, we see that between the first and second shifts lies a border $a** = **b$ of length 3 that is the longest border of substring $a**b$. This reveals another property of indeterminate strings:

Proposition 2 *A border of a border of an indeterminate string x is not necessarily a border of x.*

Index	1 2 3 4 5 6 7
x	$\cdots\ a\ b\ a\ *\ a\ *\ a \cdots$
p	$a\ b\ a\ a\ a\ b\ b$
Wrong Shift	$a\ b\ a \cdots$
Correct Shift	$a\ b\ a\ a\ a \cdots$

Table 2. Second example of the nontransitivity effect

In Table 2 we see that the length of the longest border of substring $p[1..6]$ is 2. But if we shift the pattern p to the right according to its longest border by $6-2=4$, we miss a prefix match in position 3, again due to nontransitivity. Thus:

Proposition 3 *Shifting the indeterminate pattern p to the right in x according to the longest border can miss occurrences of p.*

4 The New Hybrid Algorithm

The results of Section 3 warn us that a variant of any exact pattern-matching algorithm adapted for IPM is problematic if it depends on any form of border array calculation. In fact, one such variant has been proposed: Algorithm iFJS [15] describes an IPM adaptation of the FJS exact pattern-matching algorithm [11], that combines the border-independent Sunday version BMS [21] of the Boyer-Moore algorithm with the border-dependent KMP algorithm. This variant uses the border array only up to the longest prefix of p that does not contain any indeterminate letters. The problem is that if an indeterminate letter appears close to the left end of the pattern, then only a very small shift can occur each time, slowing the algorithm's speed significantly.

As a result, we propose replacing the KMP algorithm in iFJS by the ShiftAnd algorithm [10, 7, 23] that not only makes no use of the border array, but that furthermore has already been suggested [23] as a paradigm for IPM. We note that this strategy could be extended in a straightforward manner to use more sophisticated versions of ShiftAnd, such as the BNDM algorithm described in [18]. Our experiments suggest that the judicious combination of algorithms flipflopping from one to another based on the nature of local segments of text is more efficient than a single algorithm on its own.

Our algorithm adopts the following simple strategy:

(1) Perform a Sunday shift along the text.
(2) When a match is found at the end of the pattern, switch to ShiftAnd matching.
(3) Continue ShiftAnd matching until no match is found at the current position, then skip to the next possible position and switch back to Sunday shift.

Figure 1 shows the pseudocode for finding all the matches of pattern $p = p[1..m]$ in text $x = x[1..n]$:

```
i' ← m; m' ← m − 1;
while i' ≤ n do
  Sunday-Shift;
  — After Sunday-Shift exits, perform ShiftAnd-Match
  i ← i' − m';
  ShiftAnd-Match;
  — After ShiftAnd-Match exits, shift pattern right
  i' ← i + m';
```

Figure 1. Algorithm ShiftAnd-Sunday

For completeness we provide sketches of the Sunday and ShiftAnd algorithms:

The Sunday (BMS) Algorithm [21]

BMS has a $O(mn)$ worst-case running time but in practice is one of the fastest exact pattern-matching algorithms. To control shifts, it computes a Δ array in a preprocessing phase as follows:

> For every $\lambda \in \Sigma$, $\Delta[\lambda] = m - l + 1$, where l is the rightmost position in p where λ occurs; if λ does not occur in p, then $\Delta[\lambda] = m + 1$.

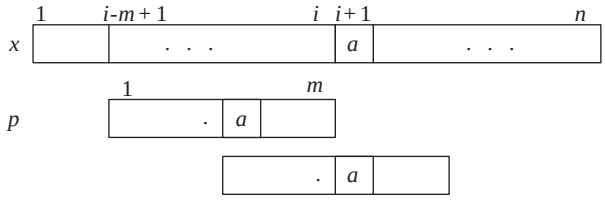

Figure 2. Sunday Shift of BMS

Figure 2 demonstrates the basic shift strategy of BMS: positions in pattern and text are compared until a mismatch occurs, say at position i in x, at which point the pattern is shifted to the next position at which an occurrence is possible, using Δ to align $x[i+1] = a$ with the rightmost occurrence of a in p. Since there can be no occurrence in between (otherwise Δ does not record the rightmost occurrence of a, a contradiction), we are safe to do so.

The ShiftAnd Algorithm [10, 7, 23]

ShiftAnd makes use of the bit-parallel capability inherent in a computer word. It has time complexity $O(mn/w)$, where w is the computer word length in bits, and is widely used in pattern-matching programs such as Unix agrep [1]. In a preprocessing phase, for each $\lambda \in \Sigma$ and every $i \in 1..m$, the algorithm computes a bit-array $S = S[1..m, 1..\alpha]$ such that $S[i, \lambda] = 1$ iff $p[i] = \lambda$, otherwise 0. This table controls the state of the calculation at each of w preceding positions in x as the pattern is shifted from position i to $i+1$. For example, for a DNA alphabet $\Sigma = \{A, C, G, T\}$ and a pattern $p = AATCG$, ShiftAnd preprocesses S as shown in Table 3.

$m\backslash\Sigma$	A	C	G	T
A	1	0	0	0
A	1	0	0	0
T	0	0	0	1
C	0	1	0	0
G	0	0	1	0

Table 3. Bit-array S after Preprocessing

The New Algorithm: ShiftAnd-Sunday

Pseudocode for the Sunday and ShiftAnd preprocessing is shown in Figures 3–4.

It is formally identical to the pseudocode used for exact pattern matching when indeterminate letters are not involved — the difference resides in the implementation

```
for i = 1 to |Δ|
  Δ[i] = m + 1
for i = 1 to m
    for j = 1 to |Σ|
      if MATCH(p[i], Σ[j]) then Δ[p[i]] = i
```

Figure 3. `Sunday-Preprocessing`

```
for i = 1 to m
  for j = 1 to |Σ|
    if MATCH(p[i], Σ[j]) then S[i, j] = 1
    else S[i, j] = 0
```

Figure 4. `ShiftAnd-Preprocessing`

of the `MATCH` function that determines whether or not two letters of the possibly extended alphabet Σ' match. The various implementations of MATCH corresponding to each of the six indeterminate processing models (1) are discussed in detail in [15] and also next section.

```
while not MATCH(p[m], x[i']) do
  i' ← i'+Δ x[i'+1]
  if i' > n then return
```

Figure 5. `Sunday-Shift`

The procedures `Sunday-Shift` and `ShiftAnd-Match` are also formally identical to their exact matching equivalents, again depending only on an implementation of `MATCH`. They are shown in Figures 5–6. Note that in practice the ShiftAnd algorithm needs to be implemented in a more sophisticated way in order to allow pattern length longer than the system word size. An example of pattern matching using this new algorithm is shown in Figures 12–18 in the Appendix.

4.1 Time & Space Complexity

Since the time complexity of our algorithm depends on the underlining `MATCH` routine, which in turn depends on the matching model, we discuss them in details here. Throughout we assume Σ' is an integer alphabet where $|\Sigma'| = K$. Letters $1, 2, .., k$ in Σ are regular letters and $k+1, .., K$ are indeterminate letters.

First we examine only the quantum model. As defined in section 2, in Model M1 only don't-care letters can occur, in other words $K = k + 1$. A simple `MATCH` routine as shown in Figures 7 will suffice and the time complexity for matching of a letter is constant.

In the M2q and M3q model, the `MATCH` routine are more complex. As shown in [14], one efficient way of implementing it is to construct a bit-array for each letter in the alphabet. In other words, we precompute a bit arrray $b_j[1..k]$ for each $j \in 1..K$ where

- for $j \leq k, b_j[j'] = 1 \Leftrightarrow j' = j$;
- for $j > k, b_j[j'] = 1 \Leftrightarrow j' \in \lambda_j$

Therefore the match routine will return true if $b_{p[\ell]} \wedge b_{x[i]} \neq 0$ and vice versa. Since the size of bit array b_j might exceed one system word, the time complexity for the `MATCH` routine in this case is therefore $O(k/w)$ where w is the system word size.

```
D ← 0
repeat
    — Here and throughout this paper operator ≪ means shifting D one position
    — towards the most significant bit and bring a 1 to the least significant bit
  D ← (D ≪ 1)&S_{x[i]}
  if D&10^m ≠ 0 then output i
  i ← i + 1
 — If D = 0, exit: no position in p has a current match.
until D = 0 or i > n
```

Figure 6. `ShiftAnd-Match`

```
if (p[ℓ] = x[i] or p[ℓ] = ∗ or x[i] = ∗) then
  return true
else
  return false
```

Figure 7. Routine `Match` for M1q model

Bit arrays of regular letters in the alphabet, namely $b_1, .., b_k$ can be computed in total time $O(k \cdot \lceil k/w \rceil)$. The computation of a bit array for an indeterminate letter j,requires $O(\lceil k/w \rceil + |\lambda_j|)$. Additional $(K-k)k/w$ space is required to store these bit arrays.

Since the subroutine `Sunday-Shift` increases the variable i' monotonically and subroutine `ShiftAnd-Match` increases the variable i monotonically, these two subroutines can be executed at most n times altogether. Each loop in `ShiftAnd-Match` runs in $O(m/w)$ time. Therefore the total worst case running time is $O(mn/w)$ for model M1q and $O(n \cdot max(m,k)/w)$ for model M2q and M3q.

The determined model (see definition on section 2), requires more complicated computing. In [14] the authors presented algorithms to handel local constrained matching (determined model). The general strategy is to maintain an array $current_{k+1}$,..,$current_K$ of bit vectors $[1..k]$ for each indeterminate letters in the alphabet. These bit vectors initially stores the same values as $b_{k+1}, .., b_K$ described previously. During matching, if $current_{p[\ell]} \wedge current_{x[i]} \neq 0$ then it indicates that $p[\ell]$ matches $x[i]$, both $current_{p[\ell]}$ and $current_{x[i]}$ are updated by the value of $current_{p[\ell]} \wedge current_{x[i]}$. We also need to use an array $mark[k+1..K]$ to indicate whether an indeterminate letter is used in current matching or not. For more details of the algorithm, see [14]. Using bit vectors $b_{k+1}, .., b_K$ requires additional $(K-k)k/w$ storage and each letter comparison requires $O(k/w)$ time.

In next section, we will see that the new algorithm has an additional nice feature of adapting well to the input, as shown in the test results.

5 Experiments

5.1 Experimental Details

Since the new algorithm is a hybrid of Sunday and ShiftAnd, we compare its running time with its components.

Factors that affect the performance of string pattern-matching are text length, pattern occurrence frequency, pattern length, alphabet size and frequency of indeterminate letters. We try to show the behaviour of the algorithms by changing only one factor at a time. However, there could be interactions between them. For example,

changing the alphabet size might cause the pattern occurrence frequency to change. We have tried to design our test cases to be both meaningful and realistic.

The main platform for our tests is a SUN X4600 server with four 2.6GHz dual core Opteron CPUs (total 8), 16GB RAM, four SAS disks, running GNU Linux 2.6.18-53.1.4.e15. We also ran tests, with consistent results, on other platforms such as a PC running Windows XP SP2.

To time the CPU time consumed by different algorithms, we use the standard C library function clock() [2]. Since the running time can be affected by factors such as CPU and memory usage of the system, temperature etc, each test was repeated 20 times. From our past experience we take the minimum time as the most accurate result. All preprocessing time is included. Functions are declared inline to eliminate the effect of function call overhead. The results are very stable across different runs.

The main test file corpus was taken from [3], itself collected from sources such as [12] for English text, [4] for DNA and protein files.

5.2 Experimental Results

Since all three algorithms are capable of handling both regular and indeterminate strings, we first test their performance on regular pattern-matching without specifying any indeterminate letters.

Execution Time against Text Length in English Files Here we run the algorithms on ten English files from [12] of sizes ranging from 240KB to 5158KB (Table 4). We use a pattern set from [16] consisting of several words that occur with moderate frequency in regular English text:

better enough govern public someth system though

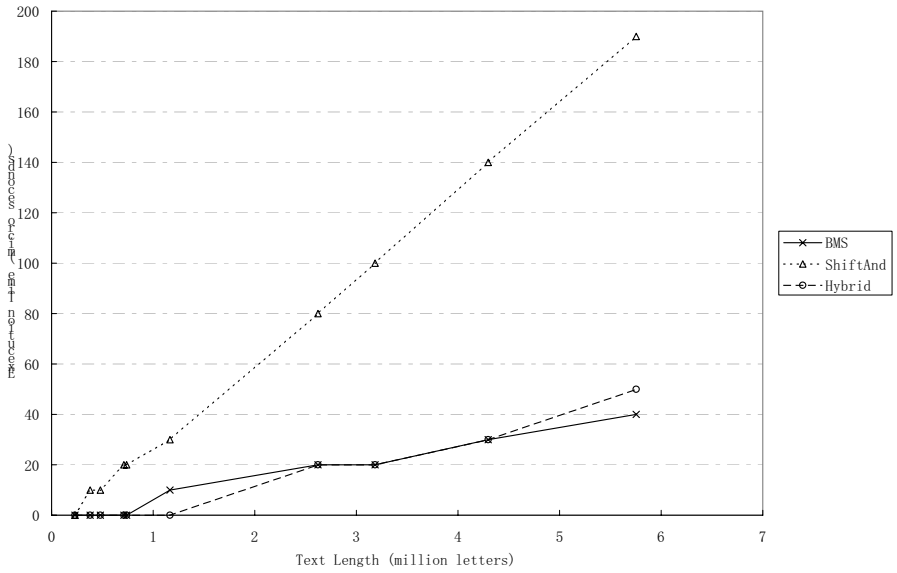

Figure 8. Execution time against text length in English files

From Figure 8 we see that the new algorithm has performance close to BMS. This is because it adapts to the nature of the text and chooses to use the BMS engine most

File Name	Length(bytes)	Description
English0.txt	237599	HAMLET, PRINCE OF DENMARK
English1.txt	389204	The Mysterious Affair at Styles
English2.txt	491905	Secret Adversary
English3.txt	699594	Pride and Prejudice (partial)
English4.txt	754019	Pride and Prejudice
English5.txt	1186876	The Adventures of Harry Richmond(partial)
English6.txt	2672650	The Adventures of Harry Richmond(partial)
English7.txt	3251887	War and Peace(partial)
English8.txt	4387156	War and Peace
English9.txt	5872902	The Adventures of Harry Richmond

Table 4. English text files

of the time. Table 5 gives the average speed of the three algorithms in microseconds per million letters (Minimum execution time divided by the length of text then take the average result of 10 files, the same for all following tables).

BMS	ShiftAnd	Hybrid
990	4550	1060

Table 5. Average microseconds/million letters in Figure 8

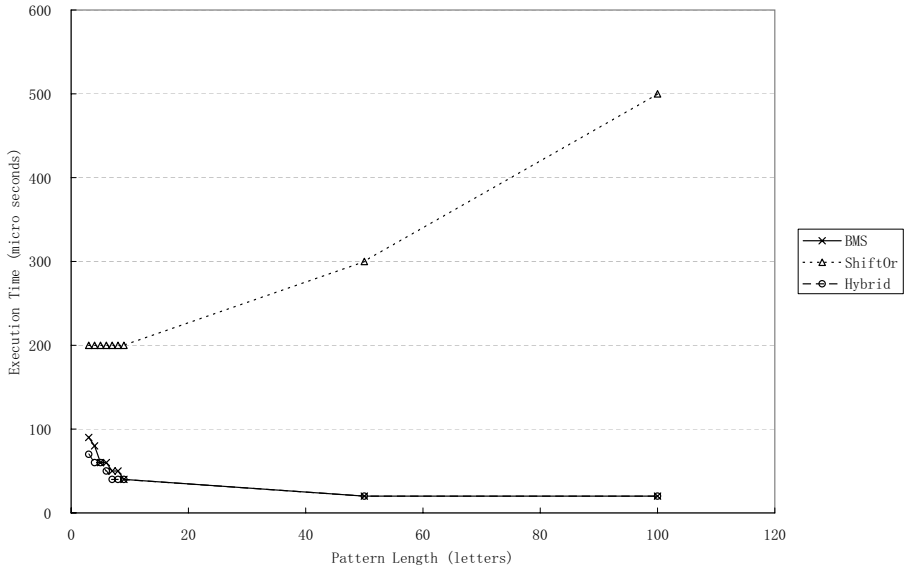

Figure 9. Execution time against pattern length in English files

Execution Time against Pattern Length in English Files Next we test the performance of the algorithms on varying pattern lengths. We use the file English8.txt, gradually increasing pattern length from 3 to 100 (see Table 6). Since longer patterns will as a rule occur less frequently, we insert the patterns randomly into the text with a frequency that decreases as pattern length increases.

From Figure 9 we see that the running times of both BMS and Hybrid decrease as pattern length increases. This is expected since the longer the pattern, the longer the skip that can be achieved by both BMS and Hybrid. As indicated by the increasing slope of the line from pattern lengths 9 to 50, when the pattern length passes the

File Name	Pattern length	Example	Total occurrences
p3.txt	3	air, age, ago	5563
p4.txt	4	body, half, held	4160
p5.txt	5	death,field, money	2665
p6.txt	6	became, behind, cannot	2426
p7.txt	7	already,brought, college	1038
p8.txt	8	anything, evidence	1685
p9.txt	9	available, community	612
p50.txt	50	Welcome To The World of ...	286
p100.txt	100	"If you have nothing better to do, ..."	275

Table 6. Details of the pattern sets used

BMS	ShiftAnd	Hybrid
1600	14390	1430

Table 7. Average microseconds/million letters in Figure 9

system word size (32), the running time of ShiftAnd begins to increase. By mainly using its BMS engine, Hybrid avoids this kind of performance slowdown.

Table 7 gives the average speed of the three algorithms over all the pattern sizes. Note that in this case the hybrid algorithm is slightly faster overall.

Execution Time against Number of Indeterminate Letters in the Alphabet Next we test the ability of our algorithm to handle indeterminate strings. In this test we again use English8.txt and the same pattern set as in our first test, but gradually increase the number of indeterminate letters in the alphabet, thus increasing their number in both text and pattern. We use the `MATCH` function corresponding to the M3q version of the hybrid algorithm, the most general (and therefore slowest) of the three quantum versions identified in Section 2. Run times are shown in Figure 10. We can see that BMS_3q runs fastest when indeterminate letters are few, but is overtaken by both ShiftAnd and the new algorithm as the number of indeterminate letters grows. Table 8 gives the average speed of the three algorithms.

BMS	ShiftAnd	Hybrid
5220	4651	4841

Table 8. Average microseconds/million letters in Figure 10

Execution Time against Text Length in DNA Files with Indeterminate Letters Finally we test the execution time against text length in DNA files with a 4-letter alphabet. We use FASTA files of increasing length as described in Table 9, with the following patterns:

CTGTAA, CAGACC, TATCCA, GGAGCC, TCCAGG, GCGGAT, AGAGAC

Letters A and C are defined as indeterminate letters. From Figure 11 we see that the three algorithms have very similar performance.

Table 10 gives the average speed of the three algorithms.

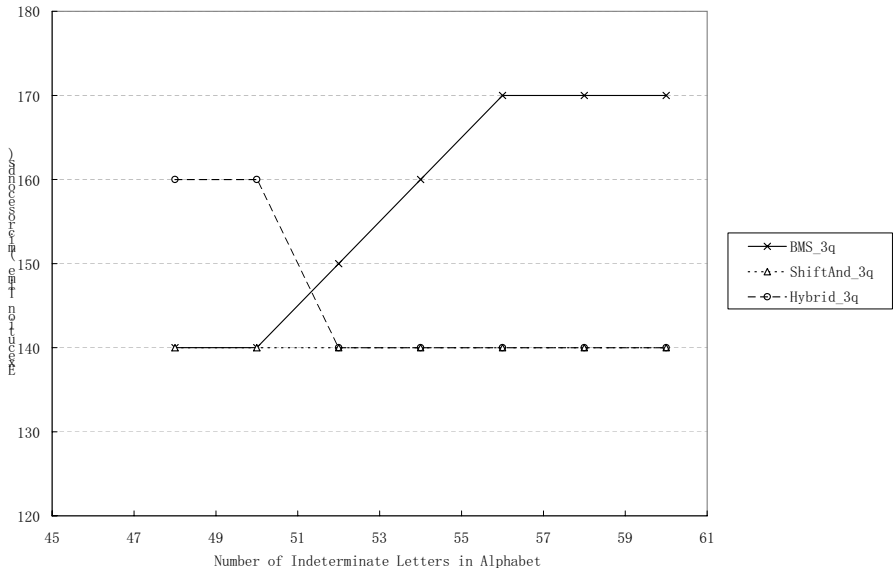

Figure 10. Execution time against number of indeterminate letters in the alphabet

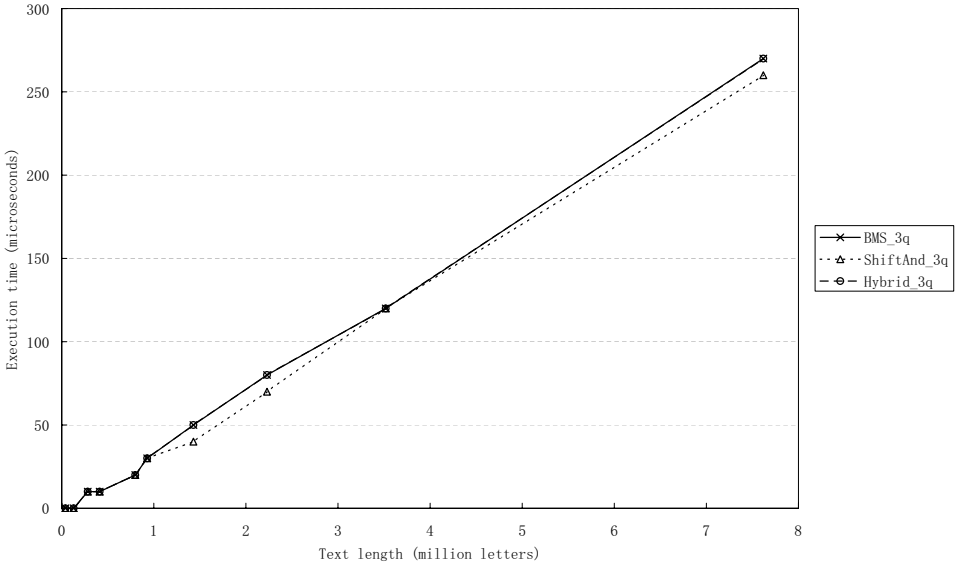

Figure 11. Execution time against text length in DNA files with indeterminate letters

File Name	Length(bytes)
DNA0.fasta	40302
DNA1.fasta	129145
DNA2.fasta	282348
DNA3.fasta	411493
DNA4.fasta	798564
DNA5.fasta	927709
DNA6.fasta	1430159
DNA7.fasta	2228723
DNA8.fasta	3518496
DNA9.fasta	7618319

Table 9. Lengths of DNA text files

BMS	ShiftAnd	Hybrid
4531	4297	4531

Table 10. Average microseconds/million letters in Figure 11

Tests\ Algorithms	BMS	ShiftAnd	Hybrid
Text Length	990	4550	1060
Pattern Length	1600	14390	1430
Number of Indeterminate Letters	5220	4651	4841
DNA file	4531	4297	4531
TOTAL	12341	27888	11862

Table 11. Summary of all test results in microseconds/million letters

We see from Table 11 that in all of these tests, the hybrid algorithm's behaviour is very close to that of the better of BMS and ShiftAnd. Moreover, due to its adaptiveness, its overall running time is actually the least over all of these rather diverse test cases. This dynamic adaptivity is useful when we do not know in advance the nature of the text or pattern: we don't need to make a decision ahead of time which algorithm to use.

6 Conclusion

We designed a new algorithm that performs fast pattern-matching on both regular and indeterminate strings. We showed in the experiments that although this new algorithm is not always the fastest, it has a strong ability to adapt to the nature of text/pattern and to achieve very good performance in all cases. In future we would like to see more competitive IPM algorithms, perhaps adapted from other exact pattern-matching algorithms such as BNDM or the convolution method.

References

[1] AGREP V3.37, Homepage V1.12, T. Gries, http://www.tgries.de/agrep/:

[2] GNU C Library, http://www.gnu.org/software/libc/manual:

[3] Simon's Collection of Test Strings, http://www.cas.mcmaster.ca/ bill/strings:

[4] National Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov/:

[5] K. Abrahamson: *Generalized string matching.* SIAM Journal on Computing, 16(6) 1987, pp. 1039–1051.

[6] A. V. Aho, J. E. Hopcroft, and J. D. Ullman: *The Design & Analysis of computer Algorithms*, Addison-Wesley, 1974.

[7] R. Baeza-Yates and G. Gonnet: *A new approach to text searching.* Communications of the ACM, 35(10) 1992, pp. 74–82.

[8] R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Communications of the ACM, 20(10) 1977, pp. 762–772.

[9] C. Charras and T. Lecroq: *Handbook of Exact String Matching Algorithms*, King's College Publications, 2004.

[10] B. Dömölki: *A universal computer system based on production rules.* BIT, 8 1968, pp. 262–275.

[11] F. Franek, C. G. Jennings, and W. F. Smyth: *A simple fast hybrid pattern-matching algorithm (preliminary version)*, in Proc. 16th Annual Symposium on Combinatorial Pattern Matching, LNCS 3537, Springer-Verlag, 2005, pp. 288–297.

[12] M. Hart: *Project gutenberg, project gutenberg literary archive foundation (2004):.*

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

[14] J. Holub, W. F. Smyth, and S. Wang: *Fast pattern-matching on indeterminate strings.* Journal of Discrete Algorithms, 6 2006, pp. 37–50.
[15] ———: *Hybrid pattern-matching algorithms on indeterminate strings.* London Algorithmics and Stringology 2006, J. Daykin, M. Mohamed and K. Steinhoefel (eds.), King's College London Series Texts in Algorithmics, 2006, pp. 115–133.
[16] C. G. Jennings: *A linear-time algorithm for fast exact pattern matching in strings*, Master's thesis, McMaster University, 2002.
[17] D. E. Knuth, J. H. Morris, and V. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6(2) 1977, pp. 323–350.
[18] G. Navarro and M. Raffinot: *A bit-parallel approach to suffix automata: Fast extended string matching*, in Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching, M. Farach-Colton, ed., no. 1448, Piscataway, NJ, 1998, Springer-Verlag, Berlin, pp. 14–33.
[19] G. Navarro and M. Raffinot: *Flexible Pattern Matching In Strings : Practical on-line search algorithms for texts and biological sequences*, Cambridge University Press, 2002.
[20] B. Smyth: *Computing Patterns in Strings*, Addison Wesley, 2003.
[21] D. M. Sunday: *A very fast substring search algorithm.* Communications of the ACM, 8 1990, pp. 132–142.
[22] S. Wang: *Pattern-matching algorithms on indeterminate strings*, Master's thesis, McMaster University, Hamilton, Canada, 2006.
[23] S. Wu and U. Manber: *Fast text searching with errors.* Communications of the ACM, 35(10) 1992, pp. 83–91.

A An Example of ShiftAnd-Sunday Algorithm

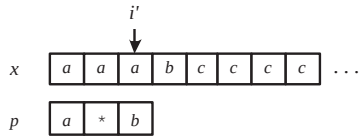

Figure 12. Starting position

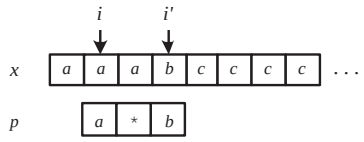

Figure 13. After one step in Sunday-Shift

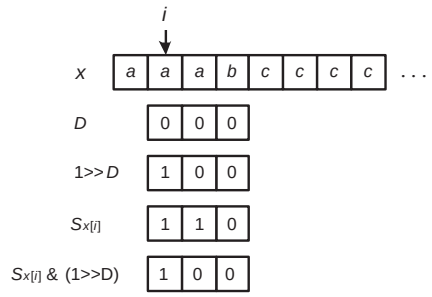

Figure 14. Switch to ShiftAnd-Matching

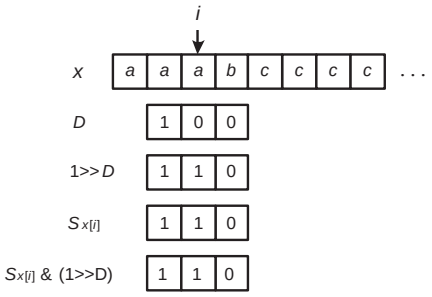

Figure 15. ShiftAnd-Matching Continues

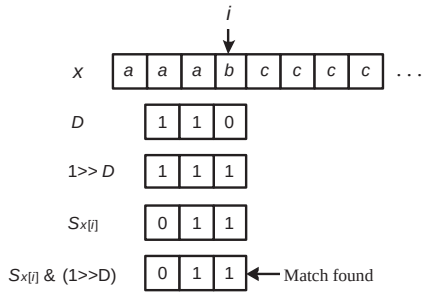

Figure 16. A match is found

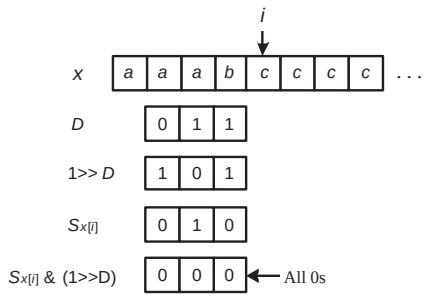

Figure 17. D' contains all zeros

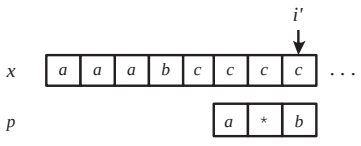

Figure 18. Switch back to Sunday-Shift