Fast Pattern-Matching on Indeterminate Strings*

Jan Holub[1], W. F. Smyth[2,3], and Shu Wang[2]

[1] Department of Computer Science & Engineering, Czech Technical University
Karlovo náměstí 13, Praha 2, CZ–121 35, Czech Republic
holub@fel.cvut.cz

[2] Algorithms Research Group, Department of Computing & Software
McMaster University, Hamilton ON L8S 4K1, Canada
www.cas.mcmaster.ca/cas/research/algorithms.htm
{smyth,shuw}@mcmaster.ca

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

September 27, 2006

Abstract. In a string $\boldsymbol{x}$ on an alphabet Σ, a position i is said to be ***indeterminate*** iff $\boldsymbol{x}[i]$ may be any one of a specified subset $\{\lambda_1, \lambda_2, \ldots, \lambda_j\}$ of Σ, $2 \le j \le |\Sigma|$. A string $\boldsymbol{x}$ containing indeterminate positions is therefore also said to be ***indeterminate***. Indeterminate strings can arise in DNA and amino acid sequences as well as in cryptological applications and the analysis of musical texts. In this paper we describe fast algorithms for finding all occurrences of a pattern $\boldsymbol{p} = \boldsymbol{p}[1..m]$ in a given text $\boldsymbol{x} = \boldsymbol{x}[1..n]$, where either or both of $\boldsymbol{p}$ and $\boldsymbol{x}$ can be indeterminate. Our algorithms are based on the Sunday variant of the Boyer-Moore pattern-matching algorithm, one of the fastest exact pattern-matching algorithms known. The methodology we describe applies more generally to all variants of Boyer-Moore (such as Horspool's, for example) that depend only on calculation of the δ ("rightmost shift") array: our method therefore assumes that Σ is ***indexed*** (essentially, an integer alphabet), a requirement normally satisfied in practice.

1 Introduction

Driven by applications to computational biology, cryptanalysis, musicology, and other areas, there has been recent interest in strings that contain letters that are not uniquely defined. In computational biology, DNA

* Supported in part by grants from the Natural Sciences & Engineering Research Council of Canada, the Ministry of Education, Youth and Sports of Czech Republic and Czech Science Foundation. The authors express their gratitude to three anonymous referees, whose comments have materially improved the quality of this paper.

sequences may still be considered to match each other if letter A (respectively, C) is juxtaposed with letter T (respectively, G); analogous juxtapositions may count as matches in protein sequences, and in fact the FASTA format [6] specifically includes indeterminate letters. In cryptanalysis, so far undecoded symbols may be known to match one of a specific set of letters in the alphabet. In music, single notes may match chords, or notes separated by an octave may match.

We refer to a letter $\boldsymbol{x}[i]$ in $\boldsymbol{x}$ that is not uniquely defined as ***indeterminate***, and we use the same term to refer to the position i at which it occurs. A string $\boldsymbol{x}$ that may possibly contain indeterminate letters is also called ***indeterminate***. The simplest form of indeterminate string is one in which indeterminate positions can contain only a ***don't-care*** letter — that is, a letter $*$ that matches any letter in the alphabet Σ on which $\boldsymbol{x}$ is defined. In 1974 an algorithm was described [7] for computing all occurrences of a pattern $\boldsymbol{p}$ in a text string $\boldsymbol{x}$, where both $\boldsymbol{p}$ and $\boldsymbol{x}$ are defined on the alphabet $\Sigma \cup \{*\}$, but although efficient in theory, the algorithm was not useful in practice. In 1987 [1] for the first time considered pattern-matching on indeterminate strings in our sense ("generalized pattern-matching"), but the algorithms again were not efficient in practice. In 1992 the bit-mapping technique for pattern-matching (often called the **ShiftOr** method) was reinvented [5, 2, 26] and applied (among several other applications) to finding matches for an indeterminate pattern $\boldsymbol{p}$ in a string $\boldsymbol{x}$. Since that time, the resulting `agrep` utility [25] has, with its variants [20–22], been virtually the only practical algorithm available for indeterminate pattern-matching.

Recently, an easily-implemented average-case $O(n)$ time algorithm was proposed [15] for computing all the periods of every prefix of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on $\Sigma \cup \{*\}$. In [11] this work was extended in two ways: first, by distinguishing two distinct forms of indeterminate match ("quantum" and "deterministic") on $\Sigma \cup \{*\}$; second, by refining the definition of indeterminate letters so that they can be restricted to matching only with specified subsets of Σ rather than with every letter of Σ (a case already considered in `agrep`). (Roughly speaking, a "quantum" match allows an indeterminate letter to match two or more distinct letters during a single matching process; a "determinate" match restricts each indeterminate letter to a single match.)

In this paper we present efficient practical algorithms for pattern-matching on indeterminate strings, where indeterminacy may be interpreted in any of the ways described in [11]. Since difficulties arise in efficiently implementing pattern-matching algorithms on indeterminate

strings that are based on some form of period [19, 16], we do not therefore pursue the ideas of [15, 11], but turn rather to the variants of the Boyer-Moore algorithm [3], such as [12, 24], that require only knowledge of the rightmost occurrence of each letter in the pattern $\boldsymbol{p}$ (the δ array). As documented in [14, 17] and more recently in [18, 8], these algorithms are in fact among the fastest available for exact pattern-matching on determinate strings, and as we shall see, these benefits extend also to indeterminate strings.

We begin with an alphabet $\Sigma = \{\lambda_1, \lambda_2, \ldots, \lambda_k\}$ that is finite and moreover, as required by all Boyer-Moore-like algorithms, ***indexed*** — that is, with the property that each λ_j, $1 \leq j \leq k$, can be used as an index (or position) in an array, say $\delta = \delta[\lambda_1..\lambda_k]$, allowing $\delta[\lambda_j]$ to be accessed in constant time. Thus without loss of generality we suppose that $\lambda_j = j$, so that $\Sigma = \{1, 2, \ldots, k\}$ is an ***integer*** alphabet.

In order to model indeterminate letters, we define an ***extended alphabet***

$$\Sigma' = \{1, 2, \ldots, k, k{+}1, \ldots, K\},$$

where to every $j \in k{+}1..K$ we associate a unique nonempty subset Σ_j of Σ, $|\Sigma_j| \geq 2$; for $j \in 1..k$ it is convenient to define $\Sigma_j = \{j\}$. (Throughout, for integers i, $j \geq i$, we use the notation $i..j$ to denote the ***range*** of integers $i, i{+}1, \ldots, j$.) Thus, in general, $K{-}k \in 0..2^k{-}k{-}1$. Note that if all letters are determinate, then $K = k$ and $\Sigma' = \Sigma$; while if only the don't-care letter occurs, then $K = k{+}1$ and $\Sigma_K = \Sigma$. It should normally be true in practice that $K{-}k$ is "not too large" — perhaps

$$K{-}k \leq d \tag{1}$$

for some small fixed integer d. Note that (1) will always be satisfied for $k = 4$ (the DNA alphabet) if we merely choose $d = 11$.

Our problem in its most general form then becomes the following:

> Compute all occurrences of a pattern $\boldsymbol{p} = \boldsymbol{p}[1..m]$ in a text string $\boldsymbol{x} = \boldsymbol{x}[1..n]$, where both $\boldsymbol{p}$ and $\boldsymbol{x}$ are defined on Σ', and where every $j \in 1..K$ matches any element of Σ_j.

In fact we consider three pattern-matching models in increasing order of sophistication (and processing time requirements), of which only the third is the general form:

(M1) The only indeterminate letter is the don't-care $*$, whose occurrences may be in either $\boldsymbol{p}$ or $\boldsymbol{x}$, or both.

(M2) Arbitrary indeterminate letters can occur, but only in $\boldsymbol{p}$ (like `agrep`).
(M3) Indeterminate letters can occur in both $\boldsymbol{p}$ and $\boldsymbol{x}$.

In Section 2 we describe the basic algorithms to solve these three problems, all based on Sunday's variant [24] of Boyer-Moore. Then in Section 3 we discuss refinements of these algorithms, based on the ideas of quantum and deterministic matching mentioned above. Section 4 gives experimental results, and we conclude in Section 5 with an outline of future work.

2 The Basic Algorithm

The original BM algorithm [3] computes an array $\delta = \delta[1..k]$, where $k = |\Sigma|$, as follows:

> For every $j \in 1..k$, $\delta[j] = m-\ell$, where ℓ is the rightmost position in $\boldsymbol{p}$ at which j occurs; if no such ℓ exists, $\delta[j] = m$.

Then for every partial match

$$\boldsymbol{x}[i+1..i+h] = \boldsymbol{p}[m-h+1..m] \tag{2}$$

with a mismatch $\boldsymbol{x}[i] \neq \boldsymbol{p}[m-h]$ for some $h \in 0..m$, a shift of $\boldsymbol{p}$ along $\boldsymbol{x}$ is implemented by next comparing $\boldsymbol{p}[m]$ with

$$\boldsymbol{x}\Big[i+\delta\big[\boldsymbol{x}[i]\big]\Big].$$

In order to ensure that $\boldsymbol{p}$ is always shifted right along $\boldsymbol{x}$, BM needs to use another array. But Horspool [12] recognized that after a partial match (2) followed by a mismatch, $\boldsymbol{p}[m]$ could be compared instead with

$$\boldsymbol{x}\Big[i_0+\delta\big[\boldsymbol{x}[i_0]\big]\Big],$$

provided a slight change were made to the δ array:

> For every $j \in 1..k$, compute δ as for BM except when $j = \boldsymbol{p}[m]$: in that case set $\delta[j] = m$ if j occurs *only* at m; otherwise, let ℓ be the rightmost position *left* of m at which j occurs, and set $\delta[j] = m-\ell$.

Finally Sunday [24] proposed use of a modified array $\Delta[1..m]$, where $\Delta[j] = \delta[j]+1$ (the original BM δ) for every $j \in 1..m$, combined with a strategy that, after a partial match (2) and then a mismatch, compares $\boldsymbol{p}[m]$ with

$$\boldsymbol{x}\Big[i_0+\Delta\big[\boldsymbol{x}[i_0+1]\big]\Big],$$

where i_0 is defined by the requirement that the previous iteration of the algorithm began with a comparison of $\boldsymbol{p}[m]$ against $\boldsymbol{x}[i_0]$.

Sunday's algorithm (BMS) is probably the fastest in practice among this subfamily of algorithms [14, 18, 8]. In this section, we describe variants of BMS adapted to the three models of indeterminate strings. For further discussion of the underlying exact pattern-matching algorithms, see [23].

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

if $m < 1$ **then return**
$i_0 \leftarrow m$; $m' \leftarrow m-1$
while $i_0 \leq n$ **do**
  $\ell \leftarrow hctam(i_0, m)$
  if $\ell = 0$ **then output** $i_0 - m'$
  $i_0 \leftarrow i_0 + \Delta[\boldsymbol{x}[i_0+1]]$

Fig. 1. Algorithm BMS

BMS is shown in its entirety in Figure 1, where of course the function *hctam* (right-to-left matching of $\boldsymbol{p}$ with $\boldsymbol{x}[i_0 - m'..i_0]$, hence right-to-left spelling of *match*) and the preprocessing of the array Δ remain to be specified. In fact, the *only* differences between BMS on determinate strings and BMS on indeterminate strings reside in the details of *hctam* and the preprocessing of Δ. Thus the algorithm of Figure 1 is formally correct in both cases.

hctam

The function *hctam* returns either the rightmost position ℓ in $\boldsymbol{p}$ of mismatch between $\boldsymbol{p}$ and $\boldsymbol{x}[i_0 - m'..i_0]$ or, if $\boldsymbol{p} = \boldsymbol{x}[i_0 - m'..i_0]$, then $\ell = 0$. For indeterminate strings, two kinds of match between individual letters $\boldsymbol{p}[\ell]$ and $\boldsymbol{x}[i]$ are possible:

* $\boldsymbol{p}[\ell] \leq k$, $\boldsymbol{x}[i] \leq k$: the regular determinate case, compare $\boldsymbol{p}[\ell] = \boldsymbol{x}[i]$.
* At least one of $\boldsymbol{p}[\ell]$, $\boldsymbol{x}[i]$ exceeds k: we need to determine whether or not $\Sigma_{\boldsymbol{p}[\ell]} \cap \Sigma_{\boldsymbol{x}[i]} = \emptyset$.

Clearly an appropriate implementation of *hctam* will depend on circumstances. For example, if as in Model M1 only don't-care letters can occur ($K = k+1$), then code such as the following will suffice:

if $\boldsymbol{p}[\ell] = \boldsymbol{x}[i]$ **or** $\boldsymbol{p}[\ell] > k$ **or** $\boldsymbol{x}[i] > k$ **then**
  `LETTERS MATCH`

else
`NO MATCH`

In more complex cases (Models M2 & M3), it is generally convenient to implement each subset Σ_j as a linked list of integers, but as we shall see, it may also be efficient to implement a bit array to deal with indeterminate letters $j > k$. The following approach will handle all cases in a time-efficient manner at an additional cost of $K\lceil k/w \rceil$ words of storage, where w is the computer word length:

Precompute for each $j \in 1..K$ a bit array $b_j[1..k]$ where
– for $j \le k$, $b_j[j'] = 1 \iff j' = j$;
– for $j > k$, $b_j[j'] = 1 \iff j' \in \Sigma_j$.
Then $\boldsymbol{p}[\ell]$ and $\boldsymbol{x}[i]$ match if and only if $b_{\boldsymbol{p}[\ell]} \wedge b_{\boldsymbol{x}[i]} \neq 0$.

For $j \le k$, all the b_j can be computed in total time $O\big(k\lceil k/w \rceil + k\big)$, while for $j > k$, computation of each b_j requires time $O\big(\lceil k/w \rceil + |\Sigma_j|\big)$. Thus in the worst case the total time is $O\big(K\lceil k/w \rceil + (K-k+1)k\big)$. For example, if $K-k \le d$, $k \le w$ (for protein sequences, $k = 20$), this total reduces to $O(dk)$.

We discuss *hctam* further in Section 3.

Computing Δ

In all cases the Δ array needs to be initialized to zero. There will of course be only k entries in Δ in the determinate case, but it is important to observe that this will also be true using model M2, since Δ is accessed in BMS using letters from $\boldsymbol{x}$ only:

for $j \leftarrow 1$ **to** (k or K) **do**
  $\Delta[j] \leftarrow 0$

For determinate matching $K = k$, the calculation of the rightmost position of each letter in $\boldsymbol{p}$ is straightforward:

for $\ell \leftarrow 1$ **to** m **do**
  $\Delta\big[\boldsymbol{p}[\ell]\big] \leftarrow \ell$

In all cases a final loop computes the skip:

for $j \leftarrow 1$ **to** (k or K) **do**
  $\Delta[j] \leftarrow m - \Delta[j] + 1$

The preprocessing of Δ is complicated in the indeterminate models M1–M3 essentially by the requirement to determine for each letter in $\boldsymbol{p}$, not

only its rightmost position, but also the rightmost position with which it may match. Thus, for example, if the rightmost don't-care occurs at position ℓ' in $\boldsymbol{p}$, then every letter $j \in \Sigma'$ has a rightmost match in $\boldsymbol{p}$ of at least ℓ'; if in particular $\boldsymbol{p}[m] = *$ ($\ell' = m$), we must set $\Delta[j] \leftarrow m$ for every j, and only very small skips will be possible for indeterminate versions of BMS. (We can alleviate this problem to some extent by removing all trailing don't-cares from $\boldsymbol{p}$.)

We consider below the calculation of the rightmost matching position for each of the three models:

(M1) As remarked above, we may suppose without loss of generality that $\boldsymbol{p}[m] \neq *$. Since both $\boldsymbol{p}$ and $\boldsymbol{x}$ can contain don't-cares, we need to compute $\Delta[1..K] = \Delta[1..k{+}1]$; the calculation of rightmost match can then be implemented as follows:

$right* \leftarrow 0;\ \Delta[K] \leftarrow m$
for $\ell \leftarrow 1$ **to** m **do**
  if $\boldsymbol{p}[\ell] \neq K$ **then**
    $\Delta\big[\boldsymbol{p}[\ell]\big] \leftarrow \ell$
  else
    $right* \leftarrow \ell$
if $right* \neq 0$ **then**
  for $j \leftarrow 1$ **to** k **do**
    if $right* > \Delta[j]$ **then**
      $\Delta[j] \leftarrow right*$

If $*$ occurs in $\boldsymbol{p}$, $\Theta(m{+}k)$ time is required; if not, then $\Theta(m)$ time.

(M2) As noted earlier, here only k entries in Δ need to be computed. A natural approach is as follows:

for $\ell \leftarrow 1$ **to** m **do**
  $\forall\ j \in \Sigma_{\boldsymbol{p}[\ell]}$ **do**
    $\Delta[j] \leftarrow \ell$

In order to implement this processing efficiently, we assume as noted above that each $\Sigma_{j'}$ is specified as a linked list of letters $j \in \Sigma_{j'}$, thus allowing all such j to be identified in $O(|\Sigma_{j'}|)$ time. Therefore this preprocessing will require $\Theta(S_1)$ time, where

$$S_1 = \sum_{\ell=1}^{m} |\Sigma_{\boldsymbol{p}[\ell]}|, \quad m \leq S_1 \leq km.$$

If m is large with respect to $K-k$, the following routine can be substituted (now making temporary use of entries $\Delta(k{+}1..K)$):

for $\ell \leftarrow 1$ **to** m **do**
  $\Delta\big[\boldsymbol{p}[\ell]\big] \leftarrow \ell$
for $j' \leftarrow k{+}1$ **to** K **do**
  if $\Delta[j'] \neq 0$ **then**
    $\forall\ j \in \Sigma_{j'}$ **do**
      if $\Delta[j] < \Delta[j']$ **then**
        $\Delta[j] \leftarrow \Delta[j']$

The modified preprocessing requires $O(m{+}S_2)$ time, where

$$S_2 = \sum_{j'=k+1}^{K} |\Sigma_{j'}|, \quad K-k \leq S_2 \leq k(K-k).$$

(M3) In this model $\Delta[1..K]$ needs to be computed, and two options analogous to those for M2 can be used, now depending on the relative sizes of m and K:

for $\ell \leftarrow 1$ **to** m **do**
  $j' \leftarrow \boldsymbol{p}[\ell]$
  $\forall\ j \in \Sigma_{j'}$ **do**
    $\Delta[j] \leftarrow \ell$
  for $j \leftarrow k{+}1$ **to** K **do**
    if $b_j \wedge b_{j'} \neq 0$ **then**
      $\Delta[j] \leftarrow \ell$

Note that in this preprocessing, it will be convenient as in case M2 to suppose that $\Sigma_{j'}$ is available as a linked list, but also, for $j > k$, to make use of the bit arrays b_j. Using, for all j, the encoding b_j of Σ_j, the time requirement is $O\big(S_1{+}mk(K-k)/w\big)$. For m large with respect to K (as may arise, for example, in applications to computational biology), the following, more complicated, preprocessing may be preferable:

for $\ell \leftarrow 1$ **to** m **do**
  $\Delta\big[\boldsymbol{p}[\ell]\big] \leftarrow \ell$
for $j' \leftarrow 1$ **to** K **do**
  $\Delta_0[j'] \leftarrow \Delta[j']$
for $j' \leftarrow k{+}1$ **to** K **do**
  for $j \leftarrow 1$ **to** K **do**
    if $b_j \wedge b_{j'} \neq 0$ **then**

if $\Delta[j] < \Delta_0[j']$ **then**
  $\Delta[j] \leftarrow \Delta_0[j']$
elsif $\Delta_0[j] > \Delta[j']$ **then**
  $\Delta[j'] \leftarrow \Delta_0[j]$

Here an auxiliary array $\Delta_0[1..K]$ has been introduced to store the initial values of Δ computed in the first loop; the time requirement is now $O\big(m+Kk(K-k)/w\big)$.

3 Constrained & Unconstrained Matching

In [11] a distinction was drawn between "quantum" and "deterministic" indeterminate letters in a string. An indeterminate letter $j > k$ was said to be ***quantum*** if it could simultaneously match more than one letter of Σ_j. For example, if $\Sigma_5 = \{1, 2\}$, the string 152 would have ***border*** (both proper prefix and suffix) 12, even though this would require 5 to simultaneously match both 1 and 2. In the ***deterministic*** case, however, either prefix 15 or suffix 52 could match 12, but not both at the same time: thus 152 would in fact have only the empty border.

In Section 2 we have implicitly assumed that indeterminate letters are quantum; for example, the algorithm described there would find two occurrences of $\boldsymbol{p} = 152$ in $\boldsymbol{x} = 1122$, even though this would require 5 to match both 1 and 2. This situation appears to be acceptable, since we are effectively allowing 5 to match 1 in one interpretation and 2 in another. More problematic however would be allowing a match between 551 and 121, where we let the first 5 match 1, while at the same time the second 5 matches 2.

We can exclude such possibilities by requiring that when two strings are compared, matches of letters $j > k$ cannot be inconsistent. We call such a match ***constrained***. For example, a constrained match of $\boldsymbol{u} = 512,\ \boldsymbol{v} = 115$ would fail (5 cannot match both 1 and 2), while one of $\boldsymbol{u} = 512,\ \boldsymbol{v} = 515$ would yield $\boldsymbol{u} = \boldsymbol{v}$. If we take instead $\Sigma_5 = \{1, 2, 3\}$, $\Sigma_6 = \{2, 3, 4\}$, then a constrained match of $\boldsymbol{u} = 515,\ \boldsymbol{v} = 611$ would fail ($\Sigma_5 \wedge \Sigma_6 = \{2, 3\}$ is not consistent with $\Sigma_5 \wedge \{1\} = \{1\}$), while on the other hand $\boldsymbol{u} = 515,\ \boldsymbol{v} = 613$ yields a match.

We identify three constraint types: ***no constraint*** which is equivalent to the above-mentioned quantum case, ***local constraint*** where matches of letters $j > k$ must be consistent within each match of the pattern in the text, and ***global constraint*** where matches of letters $j > k$ must be consistent for all pattern matches within the entire text.

Note that a set of occurrences of locally constrained matches is a subset of unconstrained matches and a set of globally constrained matches is a subset of locally constrained matches.

The distinction between locally constrained (deterministic) and unconstrained (quantum) matching is a fundamental one in the design of *hctam*. Also, as discussed earlier, *hctam* is also determined by the pattern-matching model M1, M2, or M3. There are thus six variants of *hctam* that need to be specified:

$$\text{M1q, M1d, M2q, M2d, M3q, M3d.} \tag{3}$$

$j \leftarrow \boldsymbol{p}[\ell]$; $j' \leftarrow \boldsymbol{x}[i]$
if $j > k$ **and** $mark[j] \neq 0$ **then**
  $B_1 \leftarrow current[j]$
else
  $B_1 \leftarrow b_j$
if $j' > k$ **and** $mark[j'] \neq 0$ **then**
  $B_2 \leftarrow current[j']$
else
  $B_2 \leftarrow b_{j'}$
$vector \leftarrow B_1 \wedge B_2$
if $vector \neq 0$ **then**
  if $j > k$ **then**
    $mark[j] \leftarrow 1$; $current[j] \leftarrow vector$
  if $j' > k$ **then**
    $mark[j'] \leftarrow 1$; $current[j'] \leftarrow vector$
  `MATCH`$(\boldsymbol{p}[\ell], \boldsymbol{x}[i])$
else
  `NO MATCH`$(\boldsymbol{p}[\ell], \boldsymbol{x}[i])$

Fig. 2. Handling constrained matches

We describe first a general strategy for deterministic matching that will handle all three models M1d, M2d and M3d. We maintain in *hctam* a bit vector $mark[k+1..K]$ in which $mark[j] = 1$ if and only if the indeterminate letter j has been encountered (either as $\boldsymbol{p}[\ell]$ or as $\boldsymbol{x}[i]$ or both) during the current match of $\boldsymbol{p}$ against $\boldsymbol{x}[i_0-m+1..i_0]$. Thus *mark* needs to be cleared at each invocation of *hctam*. Also required is an array $current[k+1..K]$ of bit vectors $[1..k]$: if $mark[j] = 1$, $current[j]$ stores the bit vector most recently computed as a match for b_j. Figure 2 shows the processing required to compare two letters $\boldsymbol{p}[\ell]$ and $\boldsymbol{x}[i]$, as well as to update *mark* and *current* in case of a match.

The storage required for *mark* and *current* is $\lceil (K-k)/w \rceil + (K-k)\lceil k/w \rceil$ words. The time required to initialize *mark* at each invocation of *hctam* is $\Theta\big((K–k)/w\big)$, and the total time requirement of the processing shown in Figure 2 is $\Theta(k/w)$ to compare two letters. Thus if an invocation of *hctam* performs $m' \leq m$ matches before exit from the routine, the total time requirement will be $\Theta\big((K+m'k)/w\big)$. As in Section 2, we remark that special cases can be handled more simply and efficiently; for example, if only don't-care letters occur (model M1d), or if indeterminate letters occur in $\boldsymbol{p}$ but not in $\boldsymbol{x}$ (model M2d).

Another option is to replace the bit array $mark[k+1..K]$ with an integer array $M[k+1..K]$ — working storage is increased but processing time is decreased, as we now explain. The idea is to replace the tests

$$mark[j] \neq 0 \quad \text{and} \quad mark[j'] \neq 0$$

by

$$M[j] = i_0 \quad \text{and} \quad M[j'] = i_0,$$

respectively, in Figure 2, while also replacing the assignments

$$mark[j] \leftarrow 1 \quad \text{and} \quad mark[j'] \leftarrow 1$$

by

$$M[j] \leftarrow i_0 \quad \text{and} \quad M[j'] \leftarrow i_0,$$

respectively. Thus the current position i_0 in $\boldsymbol{x}$ is used to indicate whether or not an indeterminate letter has been referenced during the current invocation of *hctam*; time requirement is reduced to $O(m'k/w)$, while additional storage space increases slightly to $(K-k)(\lceil k/w \rceil + 1)$ words.

Globally constrained matching is an optimization problem that has to be handled in a different way, as yet apparently unstudied. For example, suppose that in the string

$$\boldsymbol{x} = abcabadbcabcaba$$

we look for a pattern $\boldsymbol{p} = *b[a, c]$, where the indeterminate letters $*$ and $[a, c]$ are constrained to match in only one way across all occurrences in $\boldsymbol{x}$. We would find two matches with *abc*, two with *aba*, and one with *dbc* — possibly all of these would be interesting to the user, possible only the most frequent matches. This is a topic for future research.

4 Experimental Results

The bit-mapping algorithm ShiftOr (the underlying algorithm of `agrep`) and its improvement BNDM [20, 21] (the underlying algorithm of `Nrgrep`) are both extendible to indeterminate pattern-matching. In [21] variants of ShiftOr and BNDM that handle "classes in the text/pattern" are described. Viewed in this context, they correspond to our M2q and M3q models.

A strong point of ShiftOr and BNDM is that for large values of n (long text strings), they depend only slightly on pattern length and alphabet size: they both preprocess a two-dimensional bit array of size $K \times m$, but normally $Km \ll n$. On the other hand, ShiftOr needs to process every position of $\boldsymbol{x}$, something that in most cases indeterminate versions of BMS (and BNDM) would be able to avoid.

Some factors likely to be of interest in experiments on the comparative running times of algorithms are as follows: text and pattern length; frequency of occurrence of pattern in text; nature of the text, especially alphabet size (for example, DNA, natural language, random string); frequency of indeterminate letters in $\boldsymbol{p}$ and $\boldsymbol{x}$, and in the alphabet.

In our tests we try to determine the effects of specific factors on the behaviour of the algorithms, recognizing that in some cases there may be unexpected interactions among them. Above all, we try to devise tests that are "realistic" — that correspond to pattern-matching problems likely to arise in practice.

4.1 Testing Details

Although we have implemented all six models (3) of BMS, there do not as yet exist determinate models (1d, 2d, 3d) of other algorithms to test against them. The implementation of such models for BNDM should be possible, and is left for future work. Thus in this paper, most of our testing is confined to quantum models (q-variants) of both BMS and BNDM (1q, 2q, 3q). In many cases we have found that q-variants of BNDM behave almost identically so for clarity sometimes we use one of them to represent them collectively. We do not include the original ShiftOr, because in our test cases BNDM is almost always faster.

We performed tests on an application server machine with an Intel Xenon 2.4GHz PU running GNU/Linux. We also performed tests on other platforms such as Microsoft Windows: the results across these platforms are consistent. We use the C++ standard library function *Clock()* to time the consumption of processor time of different algorithms. Each

test was repeated 20 times and the minimum was taken as the final result. All preprocessing time was included. In order to eliminate the effect of function call overhead, all function calls were moved inline or declared as inline functions. The main corpus of this project was taken from Project Gutenberg [10], which has a collection of more than 15,000 eBooks produced by hundreds of volunteers. Ten of them in different lengths were selected at random as the testing text. Another corpus of data was taken from The Human Genome Project as an auxiliary corpus. The DNA files downloaded from [13] contain header information and some extra alphabet letters and these were filtered out so that only characters from the nucleotide alphabet $\{A, C, G, T\}$ remained. Protein files are downloaded from [4].

To properly generate indeterminate letters in our tests we use a handy "Matching Table" `MT`. This $K \times k$ table contains entries of boolean value 0 and 1. $\texttt{MT}[i][j] = 1$ if and only if $j \in \Sigma_i$. At the beginning each algorithm reads from table `MT` to conduct preprocessing properly (e.g. to generate Δ and b array for BMS-derived algorithms and B array (see [21]) for BNDM-derived algorithms). An affiliating $K \times 1$ table `IT` with boolean value is also used to indicate whether a letter is an indeterminate letter or not. Namely $\texttt{IT}[i] = 1$ indicate that $i > k$. Therefore by modifying table `MT` and `IT` we can easily control the indeterminacy of any ASCII symbols. For example, we can define the symbol '$*$' so that it can match either all or only half the letters of the alphabet.

4.2 Test Results

There are many factors that could affect the execution time of the algorithms. In this subsection we present our test results and analyze these factors separately. We plot graphs to demonstrate the execution time against different factors such as text/pattern length, alphabet size etc. We label names of algorithms being tested on the right of every graph. For clarity reason they are listed in the same order as their order of performance in the last data points.

Text Length As discussed above, for most of our testing we consider only the q-variants of BMS. For this first test only, however, in order to establish a rough pecking order among the algorithms, as well as to demonstrate the linearity of all of them in text length n, we include in addition the original BMS together with BMS_2d and BMS_3d. We use only normal texts and patterns without indeterminate letters. We

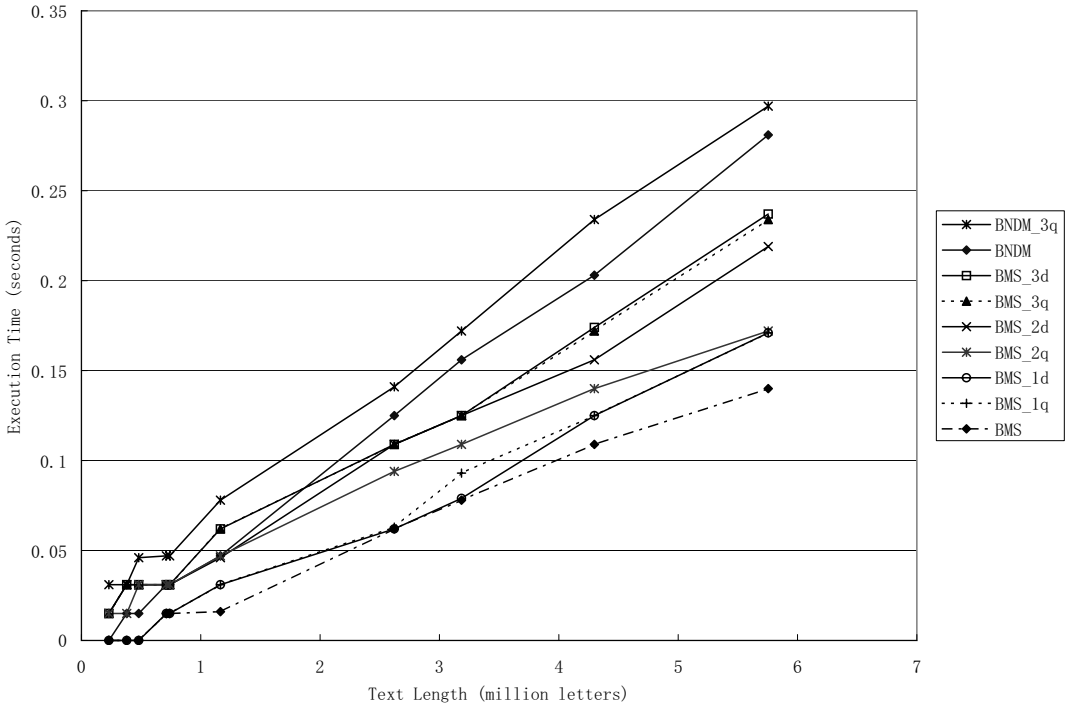

Fig. 3. Execution time against text length on high-frequency pattern set

used English texts (ASCII alphabet with about 85 symbols actually used) obtained from [10] and a high-frequency pattern set from [8]. The pattern set consisted of 7 patterns of length 6:

*_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. Further discussion of these choices can be found in [8].

The results are shown in Figure 3. All algorithms are linear in text length. BMS has the best performance, followed by BMS_1q, BMS_1d, BMS_2q, BMS_2d, BMS_3q, BMS_3d, BNDM(original, 1q and 2q) and finally BNDM_3q.

In these tests BNDM and its q-variants (BNDM_1q and BNDM_2q) have almost identical performance so they are represented collectively here by BNDM. We can see that in this test BNDM and its q-variants are all slower than their corresponding BMS variants. In particular, BNDM_3q is about 25% slower than BMS_3d. Observe, however, that the time range for BMS variants is larger than that for BNDM variants.

Frequency of Occurrence In this test we replace the high-frequency pattern set with a moderate-frequency pattern set (also from [8]), and run it on the same texts as in the previous test. The pattern set used in this test, still of length 7, is as follows:

better enough govern public someth system though

The frequency of occurrence (Occurrences/Text Length) of the moderate frequency pattern set (0.05% – 0.09%) is about 1/10 of the occurrence rate of the high-frequency pattern set (0.62% – 0.92%).

The relative performances of the algorithms in this test are virtually identical to those in the previous test (Figure 3), and so are not shown here. BMS-derived algorithms actually perform slightly better on the moderate-frequency pattern set, because the letters in these patterns are usually rarer than those in high-frequency patterns; thus longer shifts tend to occur. In general we conclude that, at least for fairly short non-indeterminate patterns on a fairly large alphabet, BMS-derived algorithms are preferable over a wide range of pattern frequency in the text.

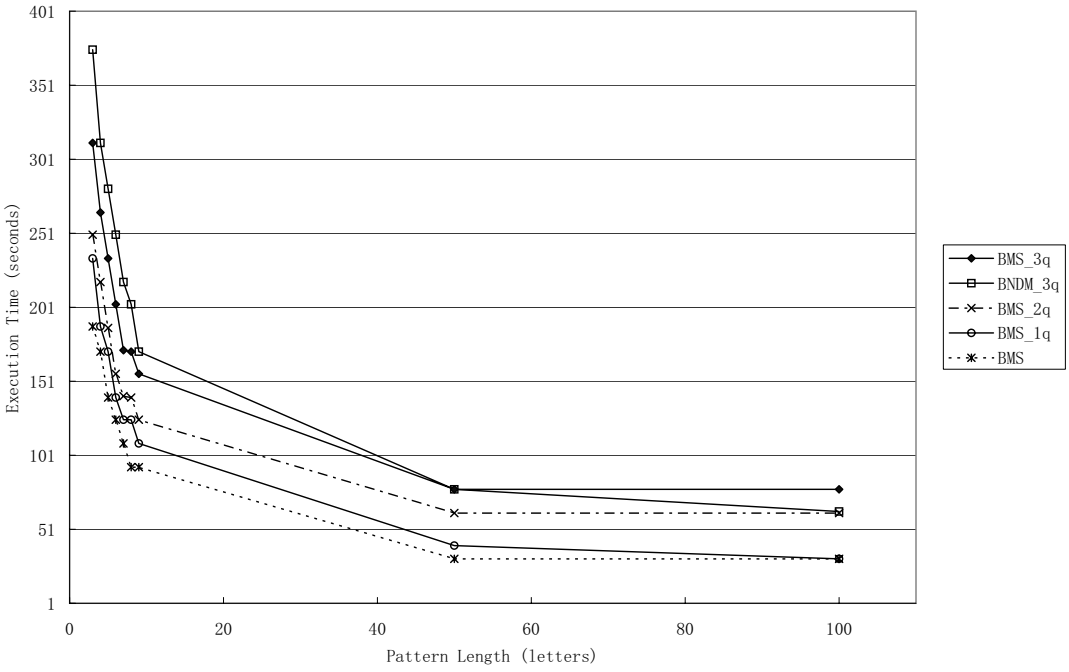

Fig. 4. Execution time against pattern length

Pattern Length Here we attempt to assess the impact of variation in pattern length on the relative efficiency of the algorithms studied. In this test all BNDM variants have almost identical performance therefore we represent them here all by BNDM_3q. We again make use of English texts and patterns with no indeterminate letters. Since in practice there is an inverse relationship between pattern length and frequency of pattern occurrence in the text, this experiment is a generalization of those performed above, making use of variable pattern lengths: ten groups of 9 equal-length patterns, ranging from 3 to 100 letters.

We have implemented the BNDM-derived algorithms so that, when handling patterns of length greater than the system word size (32 in this case), they use just the 32-character-long prefix of the pattern, and check for a full match only when a match of the prefix is found. This method was described in [20] and appears to speed up BNDM significantly.

The results in Figure 4 demonstrate that along with the increase of pattern size, execution time of both BMS-derived algorithms and BNDM-derived dropped steadily. This is because the average shift when a mismatch occurs for BMS-derived algorithms is greater when the pattern length is large. So the longer the pattern, the longer the shift when a mismatch occurs, thus the faster the algorithm. Same for BNDM-derived algorithms. It also demonstrates that at least within a realistic pattern size (less than 50 letters) BMS-derived algorithms seem to have an advantage.

Alphabet Size (Type of Texts) Here we attempt to assess the effect of alphabet size on the four main algorithms being tested. We test against files whose effective alphabet size ranges from 4 (DNA sequences), through 20 (protein sequences) and 85 (English texts), to 250 or so (object files). We use pattern sets of length six, randomly selected from text (DNA) files. Since in practice a larger alphabet necessarily implies fewer occurrences of patterns of whatever length m, the frequencies of patterns chosen in these experiments satisfy this inverse relationship.

Again BNDM_1q and BNDM_2q here are represented by BNDM. We can see from the results that except for a few cases (very small alphabet) BMS-derived algorithms remain advantageous over their corresponding BNDM variants.

Frequency of Indeterminate Letters To test how frequency of indeterminate letters in input affects the speed of different algorithms, we

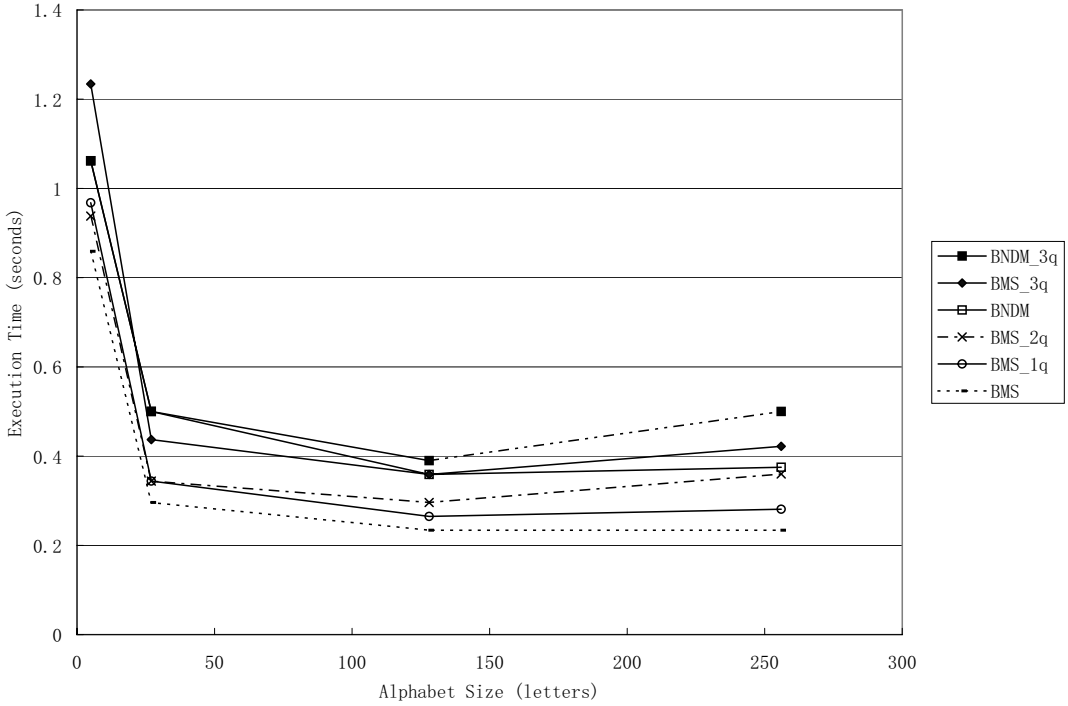

Fig. 5. Execution time against alphabet size

conducted tests with increasing frequencies of indeterminate letters in pattern, text and alphabet respectively.

First we tested algorithms able to handle M2 matching on patterns with variable frequency of indeterminate letters. We used the high-frequency pattern sets described earlier, modified by randomly substituting indeterminate letters for determinate ones in various positions. In this test we define the letter '∧' to be the only kind of indeterminate letter, matching half of the lower case English letters. As shown in Figure 6, in this test if up to about 12% of the letters in the pattern are indeterminate, BMS-derived algorithms are faster than BNDM-derived ones. As noted earlier, the execution time of BMS-derived can also be affected by the size of Σ_j and the locations (left or right) at which the indeterminate letter occurs in the pattern.

Next we tested BMS_3q and BNDM_3q against frequency of indeterminate letters in the text. These are algorithms that have the capability of handling indeterminate letters in both text and pattern. Indeterminate letters in this test appear in the text only. We define these indeterminate letters to match 1/4 of the lower case English letters and increase their frequency in the text. The results are shown in Figure 7. As expected,

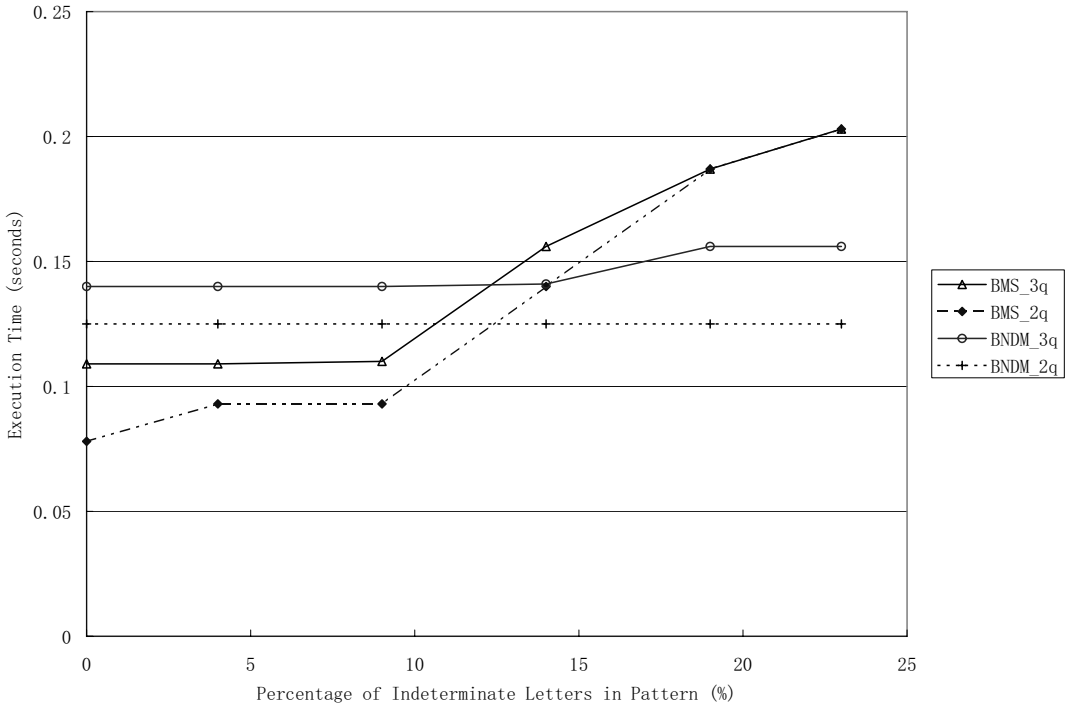

Fig. 6. Execution time against percentage of indeterminate letters in pattern

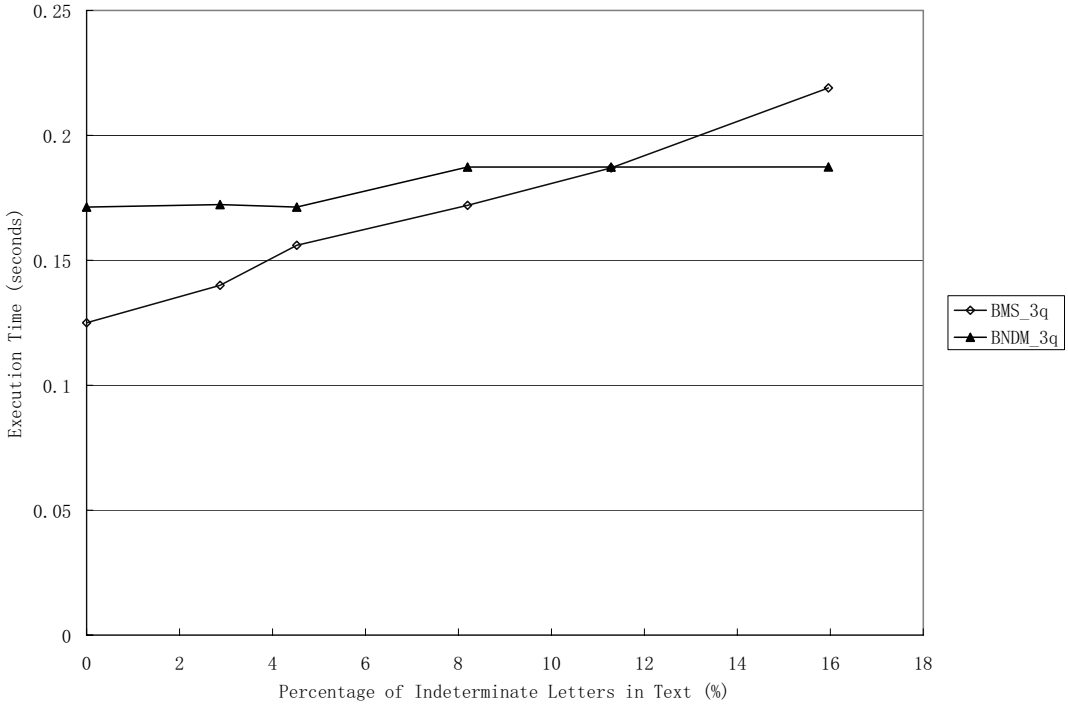

Fig. 7. Execution time against percentage of indeterminate letters in text

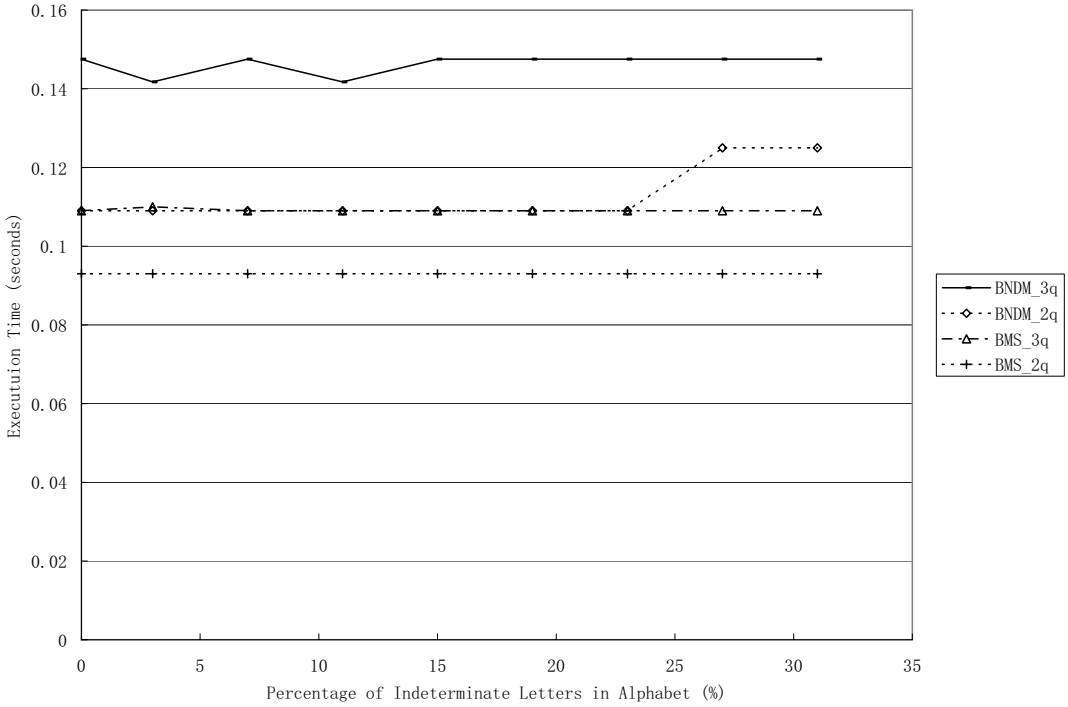

Fig. 8. Execution time against percentage of indeterminate letters in alphabet

execution time of BMS-derived algorithms increases with increasing frequency, because more computation is required, both in preprocessing and in *hctam*.

Finally we test BMS_2q, BMS_3q BNDM_2q and BNDM_3q based on various frequencies of indeterminate letters in the alphabet. Because in general increasing the frequency of indeterminate letters in the alphabet will also cause an increase in the indeterminate letters in both text and pattern, which will have compound effects on the running time, we therefore conduct the tests in such a way that newly introduced indeterminate letters will neither appear in the text nor in the pattern. In other words, the frequency of indeterminate letters in both text and pattern is constant, only the size of the alphabet is changed. The results are shown in Figure 8. We see none of the algorithms are affected by increased frequency in the alphabet, an expected result, since the ratio $(K - k)/K$ only affects preprocessing time slightly.

4.3 Conclusions from Experiments

We have tested indeterminate pattern-matching algorithms based on BMS, together with BMS itself, against equivalent BNDM-derived algorithms.

Our test results agree with the result from [21] that BMS, one of the fastest exact pattern-matching algorithms, performs well on large-alphabet texts such as English text. BMS-derived indeterminate pattern-matching algorithms however are affected somewhat more by the percentage of indeterminate letters in both text and pattern than BNDM-derived algorithms. With a relatively small level of indeterminacy (about 12% of indeterminate letters in pattern/text in our tests), the BMS variants seem to have an advantage.

5 Future Work

In this paper we have described efficient algorithms for pattern-matching on indeterminate strings, including the case of constrained matching, derived from Sunday's adaptation of the Boyer-Moore algorithm. Our implementations are not the only ones possible, and we wonder whether faster approaches can be found. In particular, we would like to investigate the following possible improvements to the BMS-derived algorithms:

* a modified matching strategy for d-variants that avoids some of the heavy matching overheads by first performing quantum matching, then checking only in case of a match to see whether indeed a determinate match has been achieved;
* a strategy similar to the BNDM strategy in the case that pattern length exceeds computer word length.

Also we intend to implement the d-variants of BNDM matching. We hope to be able to apply these results to similar extensions of Algorithm FJS as described in [8], also to the exact pattern-matching algorithms described in [9]. And we anticipate further study of the global constraint problem mentioned in Section 3.

References

1. K. Abrahamson, **Generalized string matching**, *SIAM J. Comput. 16–6* (1987) 1039–1051.
2. R. A. Baeza-Yates & G. H. Gonnet, **A new approach to text searching**, *Commun. Assoc. Comput. Mach. 35–10* (1992) 74–82.
3. Robert S. Boyer & J. Strother Moore, **A fast string searching algorithm**, *Commun. Assoc. Comput. Mach. 20–10* (1977) 762–772.
4. The Data Compression Resource on the Internet (2006): `http://www.data-compression.info/Corpora/ProteinCorpus/index.htm`
5. B. Dömölki, **An algorithm for syntactical analysis**, *Computational Linguistics 3*, Hungarian Academy of Science (1964) 29–46.

6. FASTA: http://www.ncbi.nlm.nih.gov/blast/fasta.shtml
7. Michael J. Fischer & Michael S. Paterson, **String matching and other products**, *Proc. Seventh SIAM-AMS Complexity of Computation*, Richard M. Karp (ed.) (1974) 113–125.
8. Frantisek Franek, Christopher G. Jennings & W. F. Smyth, **A simple fast hybrid pattern-matching algorithm**, *Proc. 16th Annual Symposium on Combinatorial Pattern Matching*, LNCS 3537, Springer-Verlag (2005) 288–297.
9. Kimmo Fredriksson & Szymon Grabowski, **Practical & optimal string matching**, *Proc. 12th Symp. String Processing & Information Retrieval*, LNCS 3772 (2005) 376–387.
10. Michael Hart, *Project Gutenberg*, Project Gutenberg Literary Archive Foundation (2004): http://www.gutenberg.net.
11. Jan Holub & W. F. Smyth, **Algorithms on indeterminate strings**, *Proc. 14th Australasian Workshop on Combinatorial Algorithms*, Mirka Miller & Kunsoo Park (eds.) (2003) 36–45.
12. R. Nigel Horspool, **Practical fast searching in strings**, *Software — Practice & Experience 10–6* (1980) 501–506.
13. *Human Genome Project* (2005): http://www.ornl.gov/sci/techresources/Human_Genome/home.shtml.
14. Andrew Hume & Daniel Sunday, **Fast string searching**, *Software — Practice & Experience 21–11* (1991) 1221–1248.
15. C. S. Iliopoulos, Manal Mohamed, Laurent Mouchard, W. F. Smyth, Katerina G. Perdikuri & Athanasios K. Tsakalidis, **String regularities with don't cares**, *Nordic J. Computing 10–1* (2003) 40–51.
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. Gonzalo Navarro & Mathieu Raffinot, **A bit-parallel approach to suffix automata: fast extended string matching**, *Proc. 9th Annual Symp. Combinatorial Pattern Matching*, LNCS 1448, Springer-Verlag (1998) 14–33.
21. Gonzalo Navarro & Mathieu Raffinot, *Flexible pattern matching in strings*, Cambridge University Press (2002).
22. Hannu Peltola & Jorma Tarhio, **Alternative algorithms for bit-parallel string matching**, *Proc. 10th Symp. String Processing & Information Retrieval*, LNCS 2857, Springer-Verlag (2003) 80–94.
23. Bill Smyth, *Computing Patterns in Strings*, Pearson Addison-Wesley (2003) 423 pp.
24. Daniel M. Sunday, **A very fast substring search algorithm**, *Commun. Assoc. Comput. Mach. 33–8* (1990) 132–142.
25. S. Wu & U. Manber, **Agrep – a fast approximate pattern-matching tool** *Proc. USENIX Winter 1992 Technical Conference*, San Francisco, CA (1992) 153–162.
26. S. Wu & U. Manber, **Fast text searching allowing errors**, *Commun. Assoc. Comput. Mach. 35–10* (1992) 83–91.