Simple KMP Pattern-Matching on Indeterminate Strings*

Neerja Mhaskar[1] and W. F. Smyth[1,2]

[1] Algorithms Research Group, Department of Computing & Software
McMaster University, Canada
pophlin@mcmaster.ca, smyth@mcmaster.ca
[2] School of Engineering & Information Technology
Murdoch University, Western Australia

Abstract. In this paper we describe a simple, fast, space-efficient approach to finding all matches of an indeterminate pattern $\boldsymbol{p} = \boldsymbol{p}[1..m]$ in an indeterminate string $\boldsymbol{x} = \boldsymbol{x}[1..n]$, where both $\boldsymbol{p}$ and $\boldsymbol{x}$ are defined on a "small" ordered alphabet Σ — say, $\sigma = |\Sigma| \leq 9$. A preprocessing phase replaces Σ by an integer alphabet Σ_I of size $\sigma_I = \sigma$ that (reversibly, in time linear in string length) maps both $\boldsymbol{x}$ and $\boldsymbol{p}$ into equivalent regular strings $\boldsymbol{y}$ and $\boldsymbol{q}$, respectively, on Σ_I, whose maximum (indeterminate) letter can be expressed in a 32-bit word (for $\sigma \leq 4$, thus for DNA sequences, an 8-bit representation suffices). We then describe an efficient version KMP_INDET of the venerable Knuth-Morris-Pratt algorithm to find all occurrences of $\boldsymbol{q}$ in $\boldsymbol{y}$ (that is, of $\boldsymbol{p}$ in $\boldsymbol{x}$), but, whenever necessary, using the prefix array, rather than the border array, to control shifts of the transformed pattern $\boldsymbol{q}$ along the transformed string $\boldsymbol{y}$. Although requiring $\mathcal{O}(m^2 n)$ time in the theoretical worst case, in cases of practical interest KMP_INDET executes in $\mathcal{O}(n)$ time. A noteworthy feature is the very small additional space requirement: $\Theta(m)$ words in all cases. We conjecture that a similar approach may yield practical and efficient indeterminate equivalents to other well-known pattern-matching algorithms, especially Boyer-Moore and its variants.

Keywords: indeterminate, degenerate, conservative degenerate, pattern-matching, KMP, indeterminate encoding

1 Introduction

Given a fixed finite alphabet $\Sigma = \{\lambda_1, \lambda_2, \ldots, \lambda_\sigma\}$, a ***regular letter***, also called a ***character***, is any single element of Σ, while an ***indeterminate letter*** is any subset of Σ of cardinality greater than one. A ***regular string*** $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on Σ is an array of regular letters drawn from Σ. An ***indeterminate string*** $\boldsymbol{x}[1..n]$ on Σ is an array of letters drawn from Σ, of which at least one is indeterminate. Whenever entries $\boldsymbol{x}[i]$ and $\boldsymbol{x}[j]$, $1 \leq i, j \leq n$, both contain the same character (possibly other characters as well), we say that $\boldsymbol{x}[i]$ ***matches*** $\boldsymbol{x}[j]$ and write $\boldsymbol{x}[i] \approx \boldsymbol{x}[j]$.

In this paper we describe a simple transformation of Σ that permits all subsets of Σ to be replaced by single integer values, while maintaining matches and non-matches between all transformed entries $\boldsymbol{x}[i_1]$ and $\boldsymbol{x}[i_2], 1 \leq i_1, i_2 \leq n$, in $\boldsymbol{x}$. The method is effective on small alphabets (say $|\Sigma| = \sigma \leq 9$), including in particular the important case of DNA sequences ($\Sigma_{DNA} = \{a, c, g, t\}$). Thus, in many cases, cumbersome and time-consuming matches of indeterminate letters can be efficiently handled. For background on pattern-matching in indeterminate strings, see [6,1,8,11,9,2,16,17,4,5].

* Supported by Grant No. 105–36797 from the Natural Sciences & Engineering Research Council of Canada (NSERC). Also the authors thank anonymous reviewers for several valuable suggestions.

Neerja Mhaskar, W. F. Smyth: Simple KMP Pattern-Matching on Indeterminate Strings, pp. 125–133.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9 © Czech Technical University in Prague, Czech Republic

We will make use of a mapping $\mu_j \leftarrow \lambda^{(j)},\ j = 1, 2, \ldots, \sigma$, of the letters $\lambda^{(j)}$ of Σ chosen in some order, where μ_j is the j^{th} prime number ($\mu_1 = 2, \mu_2 = 3$, and so on). Then, given $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on Σ (the ***source string***), we can apply the mapping to compute $\boldsymbol{y} = \boldsymbol{y}[1..n]$ (the ***mapped string***) according to the following rule:

(R) For every $\boldsymbol{x}[i] = \{\lambda_1, \lambda_2, \ldots, \lambda_k\}$, $1 \leq k \leq \sigma$, $1 \leq i \leq n$, where $\lambda_h \in \Sigma, 1 \leq h \leq k$, set

$$\boldsymbol{y}[i] \leftarrow \prod_{h=1}^{k} \mu_{\lambda_h}.$$

When $k = \sigma$, $\boldsymbol{y}$ achieves the maximum value, which we denote by $P_\sigma = \prod_{j=1}^{\sigma} \mu_j$ (often called a ***hole*** [2]). More generally, since the mapping π yields all possible products of the first σ prime numbers, it imposes an order on indeterminate letters drawn from Σ: $\boldsymbol{x}[i_1] < \boldsymbol{x}[i_2] \Leftrightarrow \boldsymbol{y}[i_1] < \boldsymbol{y}[i_2]$.

For example, consider a DNA source string $\boldsymbol{x} = a\{a, c\}g\{a, t\}t\{c, g\}$, over Σ_{DNA}. Then $\sigma = 4$, and applying (R) for $1 \leq k \leq 4$ (based on the mapping $\mu : 2 \leftarrow a, 3 \leftarrow c, g \leftarrow 5, 7 \leftarrow t$), we compute a mapped string $\boldsymbol{y} = 2/6/5/14/7/15$, so that

$$a < g < \{a, c\} < t < \{a, t\} < \{c, g\}.$$

On the other hand, a different mapping (say, $\mu : 2 \leftarrow t, 3 \leftarrow c, 5 \leftarrow a, 7 \leftarrow g$) would yield $\boldsymbol{y} = 5/15/7/10/2/21$ and a quite different ordering

$$t < a < g < \{a, t\} < \{a.c\} < \{c, g\}.$$

Lemma 1 *Let k_i denote the number of letters in $\boldsymbol{x}[i]$. Then Rule (R) computes $\boldsymbol{y}$ in time $\Theta(K_{\boldsymbol{x}})$, where $K_{\boldsymbol{x}} = \sum_{i=1}^{n} k_i$.*

Note that when the letters in $\boldsymbol{x}$ are strongly indeterminate — that is, $K_{\boldsymbol{x}} \in \Theta(\sigma n)$ —, then the approach proposed here (replacing $\boldsymbol{x}$ by $\boldsymbol{y}$) has the advantage that subsequent processing of $\boldsymbol{y}$ requires access only to a single integer at each position.

Lemma 2 *If $\boldsymbol{y}$ is computed from $\boldsymbol{x}$ by Rule (R), then for every $i_1, i_2 \in 1..n$, $\boldsymbol{x}[i_1] \approx \boldsymbol{x}[i_2]$ if and only if $\gcd(\boldsymbol{y}[i_1], \boldsymbol{y}[i_2]) > 1$.*

Proof.
($\Rightarrow$) By contradiction. Suppose $\boldsymbol{x}[i_1] \approx \boldsymbol{x}[i_2]$, $1 \leq i_1, i_2 \leq n$, but $\gcd(\boldsymbol{y}[i_1], \boldsymbol{y}[i_2]) = 1$; that is, $\boldsymbol{y}[i_1]$ and $\boldsymbol{y}[i_2]$ have no common divisor. Since for every i, the letter $\boldsymbol{y}[i]$ is a product of the prime numbers assigned to the characters in $\boldsymbol{x}[i]$, we see that therefore $\boldsymbol{x}[i_1]$ and $\boldsymbol{x}[i_2]$ can have no character in common; that is, $\boldsymbol{x}[i_1] \not\approx \boldsymbol{x}[i_2]$, a contradiction.
($\Leftarrow$) By the reverse argument. □

Two strings $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$ of equal length n are said to be ***isomorphic*** if and only if for every $i, j \in \{1, \ldots, n\}$,

$$\boldsymbol{x_1}[i] \approx \boldsymbol{x_1}[j] \Longleftrightarrow \boldsymbol{x_2}[i] \approx \boldsymbol{x_2}[j]. \tag{1}$$

We thus have:

Observation 3 *If $\boldsymbol{x}$ is an indeterminate string on Σ, and $\boldsymbol{y}$ is the numerical string constructed by applying Rule (R) to $\boldsymbol{x}$, then $\boldsymbol{x}$ and $\boldsymbol{y}$ are isomorphic.*

Observation 4 *By virtue of Lemma 2 and (1),* $\boldsymbol{y}$ *can overwrite the space required for* $\boldsymbol{x}$ *(and vice versa) with no loss of information.*

Observation 5 *Suppose* ℓ_1 *and* ℓ_2 *are integers representable in at most* B *bits. Then* $\gcd(\ell_1, \ell_2)$ *can be computed in time bounded by* $\mathcal{O}(M_B \log B)$[1]*, where* M_B *denotes the maximum time required to compute* $\ell_1 \ell_2$ *over all such integers.*

Observation 6 *For* $\sigma = 9$ *corresponding to the first nine prime numbers*

$$2, 3, 5, 7, 11, 13, 17, 19, 23$$

$P_\sigma = 223,092,870$*, a number representable in less than* $B = 32$ *bits, a single computer word. Thus by Observation 5, the time required to match any two indeterminate letters is bounded by* $\mathcal{O}(5M_{32})$*. When* $\sigma = 4$*, corresponding to* Σ_{DNA}*,* $2 \times 3 \times 5 \times 7 = 210 < 256$*, and so* $B = 8$ *and the matching time reduces to* $\mathcal{O}(3M_8)$*.*

Observation 7 *We assume therefore that, for* $\sigma \leq 9$*, computing a match between* $\boldsymbol{x}[i_1]$ *and* $\boldsymbol{x}[i_2]$ *on* Σ *(that is, between* $\boldsymbol{y}[i_1]$ *and* $\boldsymbol{y}[i_2]$ *computed using Rule (R)) requires time bounded above by a (small) constant.*

Other models to represent indeterminate strings have been proposed [14,10]. For example, the model proposed in [14] maps all the non-empty letters (both regular and indeterminate) over the DNA alphabet $\Sigma_{DNA} = \{A, C, G, T\}$ to the IUPAC symbols $\Sigma_{IUPAC} = \{A, C, G, T, R, Y, S, W, K, M, B, D, H, V, N\}$. Then given an indeterminate string over Σ_{DNA}, it constructs an isomorphic regular string over Σ_{IUPAC}. In [10], the model proposed maps each symbol in the DNA alphabet to a 4-bit integer power of 2; that is, $\{A, C, G, T\}$ is mapped to $\{2^0, 2^1, 2^2, 2^3\}$. Then a non-empty indeterminate letter over Σ_{DNA} is represented as $\Sigma_{\{s \in \mathcal{P}(\Sigma)\}} s$ of maximum size $15 = 1111_2$. Also, instead of using the natural order on integers, [10] uses a Gray code [7] to order indeterminate letters over Σ_{DNA}. Note that with the Gray code two successive values differ by only one bit, such as 1100 and 1101, which enables minimizing the number of separate intervals associated with each of the four symbols of Σ_{DNA}.

2 Pattern Matching Algorithm for Indeterminate Strings

In this section we describe a simple, fast, space-efficient algorithm KMP_Indet that, in order to compute all occurrences of a source pattern $\boldsymbol{p} = \boldsymbol{p}[1..m]$ in a source string $\boldsymbol{x} = \boldsymbol{x}[1..n]$, computes all the positions at which the corresponding mapped pattern $\boldsymbol{q} = \boldsymbol{q}[1..m]$ occurs in the mapped string $\boldsymbol{y} = \boldsymbol{y}[1..n]$. We begin with the following result:

Lemma 8 *For alphabet* Σ *of size* $\sigma \leq 9$*, the positions of occurrence of* $\boldsymbol{p}$ *in* $\boldsymbol{x}$ *can be computed in* $O(mn)$ *time.*

Proof. By Lemma 2 and Observation 5, the positions of occurrence of $\boldsymbol{q}$ in $\boldsymbol{y}$ can be trivially computed in $O(mnM_B \log B) = O(mn)$ time, with constant of proportionality $M_B \log B$. □

[1] https://en.wikipedia.org/wiki/Greatest_common_divisor#Complexity

As noted earlier, many pattern matching algorithms have been proposed for indeterminate strings. In [12] Iliopoulos and Radoszewski propose an $\mathcal{O}(n \log m)$ algorithm for a constant alphabet. This is the best theoretical bound known so far for pattern matching algorithms on indeterminate strings over a constant alphabet. Recently, pattern matching algorithms for ***conservative*** indeterminate strings, where the number of indeterminate letters in text $\boldsymbol{x}$ and pattern $\boldsymbol{p}$ is bounded above by a constant k, have been proposed [4,5]. In [4], Crochemore et. al present an $\mathcal{O}(nk)$ algorithm which uses suffix trees and other auxiliary data structures to search for $\boldsymbol{p}$ in $\boldsymbol{x}$. In [5], Daykin et. al propose a pattern matching algorithm by first constructing the Burrows Wheeler Transform (BWT) of $\boldsymbol{x}$ in $\mathcal{O}(mn)$ time, and use it to find all occurrences of $\boldsymbol{p}$ in $\boldsymbol{x}$ in $\mathcal{O}(km^2 + q)$ time, where q is the number of occurrences of the pattern in $\boldsymbol{x}$, and $\mathcal{O}(km^2)$ is the time required to compute it.

2.1 Definitions

We give here a few essential definitions.

Given $\boldsymbol{x}[1..n]$, then for $1 \leq i \leq n$ and $1 \leq j \leq n$, $\boldsymbol{u} = \boldsymbol{x}[i..j]$ is called a ***substring*** of $\boldsymbol{x}$, an ***empty*** substring $\boldsymbol{\varepsilon}$ if $j < i$. If $i = 1$, $\boldsymbol{u}$ is a ***prefix*** of $\boldsymbol{x}$, a ***suffix*** if $j = n$. A string $\boldsymbol{x}$ has a ***border*** $\boldsymbol{u}$ if $|\boldsymbol{u}| < |\boldsymbol{x}|$ and $\boldsymbol{x}$ has both prefix and suffix equal to $\boldsymbol{u}$. Note that a border of $\boldsymbol{x}$ may be empty.

A ***border array*** $\beta_{\boldsymbol{x}} = \beta_{\boldsymbol{x}}[1..n]$ of $\boldsymbol{x}$ is an integer array where for every $i \in [1..n]$, $\beta_{\boldsymbol{x}}[i]$ is the length of the longest border of $\boldsymbol{x}[1..i]$. A ***prefix array*** $\pi_{\boldsymbol{x}} = \pi_{\boldsymbol{x}}[1..n]$ of $\boldsymbol{x}$ is an integer array where for every $i \in [1..n]$, $\pi_{\boldsymbol{x}}[i]$ is the length of the longest substring starting at position i that matches a prefix of $\boldsymbol{x}$. See Figure 1 for an example.

	1	2	3	4	5	6	7	8	9	10	11	12	13
$\boldsymbol{x}$	a	a	b	a	a	b	a	a	{a, b}	b	a	a	{a, c}
$\beta_{\boldsymbol{x}}$	0	1	0	1	2	3	4	5	6	3	4	5	2
$\pi_{\boldsymbol{x}}$	13	1	0	6	1	0	3	5	1	0	2	2	1

Figure 1. Border array $\beta_{\boldsymbol{x}}$, and Prefix array $\pi_{\boldsymbol{x}}$ computed for the string $\boldsymbol{x} = aabaabaa\{a, b\}baa\{a, c\}$.

In Lemmas 9 and 10, we rephrase earlier results on running times for computing the border array and prefix array of a string of length n.

Lemma 9 ([15,16]) *The border array and prefix array of a regular string of length m can be computed in $\mathcal{O}(m)$ time.*

Lemma 10 ([15,16]) *The border array and prefix array of an indeterminate string of length m can be computed in $\mathcal{O}(m^2)$ time in the worst-case, $\mathcal{O}(m)$ in the average case.*

For completeness we give in Figure 2 the KMP algorithm for regular strings $\boldsymbol{x} = \boldsymbol{x}[1..n]$. In case of a mismatch or after a full match, KMP computes the shift of the pattern $\boldsymbol{p} = \boldsymbol{p}[1..m]$ along $\boldsymbol{x}$ by using the border array of $\boldsymbol{p}$, which as we have seen is computable in $\mathcal{O}(m)$ time. Thus KMP runs in $\mathcal{O}(n)$ time.

```
function KMP(x, n, p, m) : Integer List
i ← 0; j ← 0
indexlist ← ∅ ▷ List of indices where p occurs in x
β_p ← Border array of pattern p
while i < n do
    if p[j + 1] = x[i + 1] then
        j ← j + 1; i ← i + 1
        if j = m then
            indexlist ← indexlist ∪ {i−j+1}
            j ← β_p[j]
    else
        if j = 0 then i ← i+1
        else
            j ← β_p[j]
return indexlist
```

Figure 2. KMP checks whether the regular pattern $\boldsymbol{p}$ occurs in the regular text $\boldsymbol{x}$. If it does, then it outputs the set of indices at which p occurs in $\boldsymbol{x}$; otherwise returns an empty set.

2.2 KMP Algorithm for Indeterminate Strings

We now describe KMP_INDET (see Figure 3), which searches for pattern $\boldsymbol{q} = \boldsymbol{q}[1..m]$ in text $\boldsymbol{y} = \boldsymbol{y}[1..n]$, outputting the indices at which $\boldsymbol{q}$ occurs in $\boldsymbol{y}$ (thus, at which $\boldsymbol{p}$ occurs in $\boldsymbol{x}$). Thus our algorithm implements the KMP algorithm [13] on indeterminate strings that have been transformed using Rule (R). However, note that this transformation is not necessary for the algorithm to work: we use it to improve space and time efficiency. The algorithm also works with other indeterminate string encoding/transformations mentioned in the previous section. While scanning $\boldsymbol{y}$ from left to right and performing letter comparisons, KMP_INDET checks whether the prefix of $\boldsymbol{q}$ and the substring of $\boldsymbol{y}$ currently being matched are both regular. If so, then it uses the border array $\beta_{\boldsymbol{q}_\ell}$ of the longest regular prefix $\boldsymbol{q}_\ell$ of $\boldsymbol{q}$ of length ℓ, to compute the shift; if not, it constructs a new string $\boldsymbol{q'}$, which is a concatenation of the longest proper prefix of the matched prefix of $\boldsymbol{q}$ and the longest proper suffix of the matched substring of $\boldsymbol{y}$, using the prefix array $\pi_{\boldsymbol{q'}}$ of $\boldsymbol{q'}$ to compute the shift. The COMPUTE_SHIFT function given in Figure 4 implements this computation.

In order to determine whether or not indeterminate letters are included in any segment $\boldsymbol{q'} = \boldsymbol{q}[1..j-1]\boldsymbol{y}[i-j+2..i]$, two variables are employed: $indet_y$ and the length ℓ of the longest regular prefix $\boldsymbol{q}_{\boldsymbol{\ell}}$ of $\boldsymbol{q}$. $indet_y$ is a Boolean variable that is true if and only if the current segment $\boldsymbol{y}[i-j+2..i]$ contains an indeterminate letter; ℓ is pre-computed in $\mathcal{O}(m)$ time as a byproduct of the one-time calculation of $\boldsymbol{q}_{\boldsymbol{\ell}}$.

If $\boldsymbol{y}$ and $\boldsymbol{q}$ are both regular, then KMP_INDET reduces to the KMP algorithm [13]. Otherwise, it checks whether indeterminate letters exist in the matched prefix of $\boldsymbol{q} = \boldsymbol{q}[1..j-1]$, or the matched substring of $\boldsymbol{y} = \boldsymbol{y}[i-j+2..i]$. If they do, then the shift in $\boldsymbol{q}$ is equal to the maximum length of the prefix of $\boldsymbol{q}[1..j-1]$ that matches with a suffix of $\boldsymbol{y}[i-j+2..i]$. To compute this length, the algorithm first builds a new string $\boldsymbol{q'} = \boldsymbol{q}[1..j-1]\boldsymbol{y}[i-j+2..i]$ and, based on an insight given in [16], computes its *prefix* array $\pi_{\boldsymbol{q'}}$ rather than its border array. To compute the shift only the last j entries of $\pi_{\boldsymbol{q'}}$ are examined; that is, entries $k = j+1$ to $2(j-1)$. Note that we need to consider only those entries k in $\pi_{\boldsymbol{q'}}[j+1..2(j-1)]$, where a prefix of $\boldsymbol{q'}$ matches the suffix at

```
function KMP_Indet(y, n, q, m) : Integer List
i ← 0; j ← 0; indet_y ← false
indexlist ← ∅ ▷ List of indices where q occurs in y
q_ℓ ← longest regular prefix of q of length ℓ
β_q ← Compute_β(q_ℓ) ▷ Border Array of q_ℓ
while i < n do
    if q[j + 1] ≈ y[i + 1] then
        if INDET(y[i + 1]) then indet_y ← true
        j ← j + 1; i ← i + 1
        if j = m then
            indexlist ← indexlist ∪ {i−j+1}
            j ← Compute_Shift(indet_y, y, q, i, j, β_q, ℓ)
            indet_y ← false
    else
        if j = 0 then i ← i+1
        else
            j ← Compute_Shift(indet_y, y, q, i, j, β_q, ℓ)
            indet_y ← false
return indexlist
```

Figure 3. KMP_Indet checks whether the pattern $\boldsymbol{q}$ occurs in the text $\boldsymbol{y}$ (both possibly indeterminate). If it does, then it outputs the set of indices at which q occurs in $\boldsymbol{y}$; otherwise returns an empty set.

k ($\boldsymbol{q'}[k..2(j-1)]$); that is, the entries where $\pi_{\boldsymbol{q'}}[k] = 2j-k-1$. The shift is simply the maximum over such entries in $\pi_{\boldsymbol{q'}}$. (Recall that computing the border array for an indeterminate string is not useful as the matching relation $\approx$ is not transitive [8].)

```
function Compute_Shift(indet_y, y, q, i, j, β_q, ℓ) : Integer
▷ ℓ is length of longest regular prefix of q.
if indet_y or j > ℓ then
    q' = q[1..j−1]y[i−j+2..i]
    π_q' ← Compute_π(q') ▷ Prefix Array of pattern q'
    max ← 0
    for k = j to 2(j − 1)
        if max < π_q'[k] and π_q'[k] = 2j−k−1 then
            max ← π_q'[k]
    j ← max
else ▷ prefix of q & substring of y are regular
    j ← β_q[j]
return j
```

Figure 4. Compute_Shift computes the shift in the pattern when a mismatch occurs or the end of pattern is reached.

Figure 5 represents the processing of the text $\boldsymbol{x} = aabaabaa\{a,b\}baa\{a,c\}$ and pattern $\boldsymbol{p} = aabaa$ corresponding to the processing of $\boldsymbol{y}$ and $\boldsymbol{q}$ by KMP_Indet. KMP_Indet first computes $\beta_{\boldsymbol{p}} = (0,1,0,1,2)$ and $\ell = 5$. Initially the pattern is aligned with $\boldsymbol{x}$ at position 1. Since it matches with the text ($j = 5$), and $indet_x = false$ and $5 \leq (\ell = 5)$, we compute the shift from $\beta_{\boldsymbol{p}}[5] = 2$. Therefore, the pattern

is then aligned with $\boldsymbol{x}$ at position $i = 4$. Analogously, the pattern is next aligned with $\boldsymbol{x}$ at position $i = 7$. Since a mismatch occurs at $i + 1 = 10, j + 1 = 4$, and because $indet_x = true$, we construct $\boldsymbol{p'} = \boldsymbol{p}[1..2]\boldsymbol{x}[8..9] = aba\{a, b\}$ and compute $\pi_{\boldsymbol{p'}} = (4, 0, 2, 1)$. Then shift is equal to 2. Therefore the pattern is aligned with $\boldsymbol{x}$ at 8. Since it matches (and because it is the last match), KMP_INDET returns the list $\{1, 4, 8\}$.

KMP_INDET contains a function INDET that determines whether or not the current position $\boldsymbol{y}[i+1]$ is indeterminate. To enable this query to be answered efficiently, we suppose that an array $P = P[1..9]$ has been created with $P[t]$ equal to the t^{th} prime number in the range 2..23 (for $\sigma = 9$). Then $\boldsymbol{y}[i+1]$ is indeterminate if and only if it exceeds 23 or else does not occur in P. The worst case time requirement for INDET is therefore $\log_2 9$ times a few microseconds, the time for a binary search.

	1	2	3	4	5	6	7	8	9	10	11	12	13
$\boldsymbol{x}$	a	a	b	a	a	b	a	a	{a, b}	b	a	a	{a, c}
	a	a	b	a	a								
				a	a	b	a	a					
							a	a	b	x			
								a	a	b	a	a	

Figure 5. The figure simulates the execution of KMP_INDET on the text $\boldsymbol{x} = aabaabaa\{a, b\}baa\{a, c\}$ and pattern $\boldsymbol{p} = aabaa$. After execution, KMP_INDET returns the list of positions $\{1, 4, 8\}$ at which $\boldsymbol{p}$ occurs in $\boldsymbol{x}$. 'x' in the third alignment identifies a mismatch.

Now we discuss the running time of algorithm KMP_INDET. It is clear that the running time of KMP_INDET for a regular pattern and regular text is linear. Otherwise, when a matched prefix of $\boldsymbol{q}$ or a matched substring of $\boldsymbol{y}$ contains an indeterminate letter, then the algorithm constructs the prefix array of a new string $\boldsymbol{q'}$ which is a concatenation of the matched strings. In the worst case we might need to construct the prefix array of $\boldsymbol{q'}$ for each iteration of the **while** loop. By Lemma 10 and because $\boldsymbol{q'}$ can be of length at most $2(m-1)$, in the worst case the total time required for the execution of KMP_INDET is $\mathcal{O}(m^2 n)$. Theorem 11 states these conclusions:

Theorem 11 *Given text $\boldsymbol{y} = \boldsymbol{y}[1..n]$ and pattern $\boldsymbol{q} = \boldsymbol{q}[1..m]$ on an alphabet of constant size σ,* KMP_INDET *executes in $\mathcal{O}(n)$ time when $\boldsymbol{y}$ and $\boldsymbol{q}$ are both regular; otherwise, when both are indeterminate, the worst-case upper bound is $\mathcal{O}(m^2 n)$. The algorithm's additional space requirement is $\mathcal{O}(m)$, for the pattern $\boldsymbol{q'}$ and corresponding arrays $\beta_{\boldsymbol{q'}}$ and $\pi_{\boldsymbol{q'}}$.*

An improved theoretical bound to compute the prefix array for a string over a constant alphabet is given in [12], and is summarized in Lemma 12. Using Lemma 12 we restate Theorem 11 resulting in an improved run time complexity for KMP_INDET.

Lemma 12 ([12]) *The prefix array of an indeterminate string of length n over a constant-sized alphabet can be computed in $\mathcal{O}(n\sqrt{n})$ time and $\mathcal{O}(n)$ space.*

Theorem 13 *Given text $\boldsymbol{y} = \boldsymbol{y}[1..n]$ and pattern $\boldsymbol{q} = \boldsymbol{q}[1..m]$ on an alphabet of constant size σ,* KMP_INDET *executes in $\mathcal{O}(n)$ time when $\boldsymbol{y}$ and $\boldsymbol{q}$ are both regular; otherwise, when both are indeterminate, the worst-case upper bound is $\mathcal{O}(nm\sqrt{m})$.*

The algorithm's additional space requirement is $\mathcal{O}(m)$, for the pattern $\boldsymbol{q'}$ and corresponding arrays $\beta_{\boldsymbol{q'}}$ and $\pi_{\boldsymbol{q'}}$.

We provide context for the result given in Theorems 11 and 13 by the following:

Remark 14 *One of the features that makes this algorithm truly practical is that, apart from the $\mathcal{O}(n)$ time in-place mapping of $\boldsymbol{x}$ into $\boldsymbol{y}$ and $\boldsymbol{p}$ into $\boldsymbol{q}$, there is no preprocessing and no auxiliary data structure requirement. As a result, processing is direct and immediate, requiring negligible additional storage.*

Remark 15 *The worst case time requirement is predicated on a requirement for $\mathcal{O}(n)$ (short) shifts of $\boldsymbol{q}$ along $\boldsymbol{y}$, each requiring a worst-case $\mathcal{O}(m^2)$ prefix array calculation. For example, this circumstance could occur with $\boldsymbol{p} = \{a, b\}c^{m-1}$ and $\boldsymbol{x} = a^n$ or with $\boldsymbol{p} = ab$ and $\boldsymbol{x} = \{a, c\}^n$.*

Remark 16 *Indeed, given a regular pattern and a string $\boldsymbol{x}$ containing Q indeterminate letters (a case considered in both [4] and [5]),* KMP_Indet *may make as many as mQ shifts, each requiring $\mathcal{O}(m^2)$ processing, thus $\mathcal{O}(m^3Q)$ overall. Therefore, if m^3Q is small with respect to m^2n — Q small with respect to n/m — then* KMP_Indet *will execute in $\mathcal{O}(n)$ time.*

3 Conclusion

We have described a simple procedure, based on the KMP algorithm, to do pattern-matching on indeterminate strings that is very time-efficient in cases that arise in practice and moreover uses negligible $\Theta(m)$ space in all cases. We conjecture that a similar approach is feasible for the Boyer-Moore algorithm [3], together with its numerous variants (BM-Horspool, BM-Sunday, BM-Galil, Turbo-BM): see [15, Ch. 8] and

`https://www-igm.univ-mlv.fr/~lecroq/string/`

It would also be of interest to optimize KMP_Indet for the conservative indeterminate strings mentioned in Section 2. And we look forward to experimental comparison of the running times of existing indeterminate pattern-matching algorithms with those of KMP_Indet, assuming various frequencies of indeterminate letters.

References

1. K. Abrahamson: *Generalized string matching.* SIAM Journal of Computing, 16(6) 1987, pp. 1039–1051.
2. F. Blanchet-Sadri: *Algorithmic Combinatorics on Partial Words*, Chapman & Hall CRC, 2008.
3. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Communications of the ACM, 20(10) 1977, pp. 762–772.
4. M. Crochemore, C. S. Iliopoulis, R. Kundu, M. Mohamed, and F. Vayani: *Linear algorithm for conservative degenerate pattern matching.* Eng. Appls. of Artificial Intelligence, 51 2016, pp. 109–114.
5. J. W. Daykin, R. Groult, Y. Guesnet, T. Lecroq, A. Lefebvre, M. Léonard, L. Mouchard, E. Prieur-Gaston, and B. Watson: *Efficient pattern matching in degenerate strings with the Burrows-Wheeler transform.* Information Processing Letters, 147 2019, pp. 82–87.

6. M. Fischer and M. Paterson: *String matching and other products*, in Complexity of Computation,, R. Karp, ed., American Mathematical Society, 1974, pp. 113–125.
7. F. Gray: *Pulse code communication*. Hughes Aircraft Company, U.S. Patent no. 2632058, 1953.
8. J. Holub and W. F. Smyth: *Algorithms on indeterminate strings*. Proc. 14th Australasian Workshop on Combinatorial Algs. (AWOCA), 2003, pp. 36–45.
9. J. Holub, W. F. Smyth, and S. Wang: *Hybrid pattern-matching algorithms on indeterminate strings*, in London Algorithmics and Stringology, J. W. Daykin, M. Mohamed, and K. Steinhofel, eds., King's College Texts in Algorithmics, 2006, pp. 115–133.
10. L. Huang, V. Popic, and S. Batzoglou: *Short read alignment with populations of genomes*. Bioinformatics, 29(13) 06 2013, pp. i361–i370.
11. C. S. Iliopoulos, M. Mohamed, L. Mouchard, W. F. Smyth, K. G. Perdikuri, and A. K. Tsakalidis: *String regularities with don't cares*. Nordic J. Computing, 10(1) 2003, pp. 40–51.
12. C. S. Iliopoulos and J. Radoszewski: *Truly subquadratic-time extension queries and periodicity detection in strings with uncertainties*, in CPM, 2016.
13. D. E. Knuth, J. H. Morris, and V. R. Pratt: *Fast pattern matching in strings*. SIAM Journal of Computing, 6(2) 1977, pp. 323–350.
14. P. Procházka and J. Holub: *On-line searching in IUPAC nucleotide sequences*, in Proceedings of the 12th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC 2019) - Volume 3: BIOINFORMATICS, Prague, Czech Republic, February 22-24, 2019, E. D. Maria, A. L. N. Fred, and H. Gamboa, eds., SciTePress, 2019, pp. 66–77.
15. B. Smyth: *Computing Patterns in Strings*, Pearson/Addison–Wesley, 2003.
16. W. F. Smyth and S. Wang: *New perspectives on the prefix array*. Proc. 15th String Processing & Inform. Retrieval Symp. (SPIRE), 5280 2008, pp. 133–143.
17. W. F. Smyth and S. Wang: *An adaptive hybrid pattern-matching algorithm on indeterminate strings*. Internat. J. Foundations of Computer Science, 20(6) 2009, pp. 985–1004.