

Efficient pattern matching in degenerate strings with the Burrows–Wheeler transform

Jacqueline W. Daykin^{1,2,3}, Richard Groult^{4,3}, Yannick Guesnet³, Thierry Lecroq³, Arnaud Lefebvre³, Martine Léonard³, Laurent Mouchard³, Élise Prieur-Gaston³, and Bruce Watson^{5,6}

¹ Department of Computer Science, Aberystwyth Univ. (Mauritius Branch Campus), Quartier Militaire, Mauritius

² Department of Informatics, King’s College London, UK

³ Normandie Univ., UNIROUEN, LITIS, 76000 Rouen, France

⁴ Modélisation, Information et Systèmes (MIS), Univ. de Picardie Jules Verne, Amiens, France

⁵ Department of Information Science, Stellenbosch Univ., South Africa

⁶ CAIR, CSIR Meraka, Pretoria, South Africa

Abstract. A *degenerate* or *indeterminate* string on an alphabet Σ is a sequence of non-empty subsets of Σ . Given a degenerate string \mathbf{t} of length n , we present a new method based on the Burrows–Wheeler transform for searching for a degenerate pattern of length m in \mathbf{t} running in $O(mn)$ time on a constant size alphabet Σ . Furthermore, it is a *hybrid* pattern-matching technique that works on both regular and degenerate strings. A degenerate string is said to be *conservative* if its number of non-solid letters is upper-bounded by a fixed positive constant q ; in this case we show that the search complexity time is $O(qm^2)$. Experimental results show that our method performs well in practice.

Keywords: algorithm, Burrows–Wheeler transform, degenerate, pattern matching, string

1 Introduction

An *indeterminate* or *degenerate* string $\mathbf{x} = \mathbf{x}[1..n]$ on an alphabet Σ is a sequence of non-empty subsets of Σ . Degenerate strings date back to the groundbreaking paper of Fischer & Paterson [5]. This simple generalization of a regular string, from letters to subsets of letters, arises naturally in diverse applications: in musicology, for instance the problem of finding chords that match with single notes; search tasks allowing for occurrence of errors such as with web interfaces and search engines; bioinformatics activities including DNA sequence analysis and coding amino acids; and cryptanalysis applications.

For regular or solid strings, the main approaches for computing all the occurrences of a given nonempty pattern $\mathbf{p} = \mathbf{p}[1..m]$ in a given nonempty text $\mathbf{t} = \mathbf{t}[1..n]$ have been window-shifting techniques, and applying the bit-parallel

processing to achieve fast processing – for expositions of classic string matching algorithms see [2]. More recently the Burrows–Wheeler transform (BWT) has been tuned to this search task, where all the occurrences of the pattern \mathbf{p} can be found as a prefix of consecutive rows of the BWT matrix, and these rows are determined using a backward search process.

The degenerate pattern matching problem for degenerate strings \mathbf{p} and \mathbf{t} over Σ of length m and n respectively is the task of finding all the positions of all the occurrences of \mathbf{p} in \mathbf{t} , that is, computing every j such that $\forall 1 \leq i \leq |\mathbf{p}|$ it holds that $\mathbf{p}[i] \cap \mathbf{t}[i + j] \neq \emptyset$.

Variants of degenerate pattern matching have recently been proposed. A degenerate string is said to be *conservative* if its number of non-solid letters is upper-bounded by a fixed positive constant q . Crochemore *et al.* [3] considered the matching problem of conservative degenerate strings and presented an efficient algorithm that can find, for given degenerate strings \mathbf{p} and \mathbf{t} of total length n containing q non-solid letters in total, the occurrences of \mathbf{p} in \mathbf{t} in $O(nq)$ time, i.e. linear in the size of the input.

Our novel contribution is to implement degenerate pattern matching by modifying the existing Burrows–Wheeler pattern matching technique. Given a degenerate string \mathbf{t} of length n , searching for either a degenerate or solid pattern of length m in \mathbf{t} is achieved in $O(mn)$ time; in the conservative scenario with at most q degenerate letters, the search complexity is $O(qm^2)$ – competitive for short patterns. This formalizes and extends the work implemented in BWB-BLE [7].

2 Notation and definitions

Consider a finite totally ordered alphabet Σ of constant size which consists of a set of *letters*. The order on letters is denoted by the usual symbol $<$. A *string* is a sequence of zero or more letters over Σ . The set of all strings over Σ is denoted by Σ^* and the set of all non-empty strings over Σ is denoted by Σ^+ . Note we write strings in mathbold such as \mathbf{x} , \mathbf{y} . The lexicographic order (*lexorder*) on strings is also denoted by the symbol $<$.

A string \mathbf{x} over Σ^+ of length $|\mathbf{x}| = n$ is represented by $\mathbf{x}[1..n]$, where $\mathbf{x}[i] \in \Sigma$ for $1 \leq i \leq n$ is the i -th letter of \mathbf{x} . The symbol \sharp gives the number of elements in a specified set.

The concatenation of two strings \mathbf{x} and \mathbf{y} is defined as the sequence of letters of \mathbf{x} followed by the sequence of letters of \mathbf{y} and is denoted by $\mathbf{x} \cdot \mathbf{y}$ or simply \mathbf{xy} when no confusion is possible. A string \mathbf{y} is a *substring* of \mathbf{x} if $\mathbf{x} = \mathbf{u}\mathbf{y}\mathbf{v}$, where $\mathbf{u}, \mathbf{v} \in \Sigma^*$; specifically a string $\mathbf{y} = \mathbf{y}[1..m]$ is a substring of \mathbf{x} if $\mathbf{y}[1..m] = \mathbf{x}[i..i + m - 1]$ for some i . Strings $\mathbf{u} = \mathbf{x}[1..i]$ are called *prefixes* of \mathbf{x} , and strings $\mathbf{v} = \mathbf{x}[i..n]$ are called *suffixes* of \mathbf{x} of length n for $1 \leq i \leq n$. The prefix \mathbf{u} (respectively suffix \mathbf{v}) is a proper prefix (suffix) of a string \mathbf{x} if $\mathbf{x} \neq \mathbf{u}, \mathbf{v}$. A string $\mathbf{y} = \mathbf{y}[1..n]$ is a *cyclic rotation* of $\mathbf{x} = \mathbf{x}[1..n]$ if $\mathbf{y}[1..n] = \mathbf{x}[i..n]\mathbf{x}[1..i - 1]$ for some $1 \leq i \leq n$.

Definition 1 (Burrows–Wheeler transform). *The BWT of \mathbf{x} is defined as the pair (L, h) where L is the last column of the matrix $M_{\mathbf{x}}$ formed by all the lexicorder sorted cyclic rotations of \mathbf{x} and h is the index of \mathbf{x} in this matrix.*

The BWT is easily invertible via a linear last first mapping [1] using an array C indexed by all the letters c of the alphabet Σ and defined by: $C[c] = \#\{i \mid \mathbf{x}[i] < c\}$ and $rank_c(\mathbf{x}, i)$ which gives the number of occurrences of the letter c in the prefix $\mathbf{x}[1..i]$.

In [4], Daykin and Watson present a simple modification of the classic BWT, the *degenerate Burrows–Wheeler transform*, which is suitable for clustering degenerate strings.

Given an alphabet Σ we define a new alphabet Δ_{Σ} as the non-empty subsets of Σ : $\Delta_{\Sigma} = \mathcal{P}(\Sigma) \setminus \{\emptyset\}$.

Formally a non-empty *indeterminate* or *degenerate* string \mathbf{x} is an element of Δ_{Σ}^+ . We extend the notion of prefix on degenerate strings as follows. A degenerate string \mathbf{u} is called a *degenerate prefix* of \mathbf{x} if $|\mathbf{u}| \leq |\mathbf{x}|$ and $\mathbf{u}[i] \cap \mathbf{x}[i] \neq \emptyset \forall 1 \leq i \leq |\mathbf{u}|$.

A degenerate string is said to be *conservative* if its number of non-solid letters is upper-bounded by a fixed positive constant q .

Definition 2. *A degenerate string $\mathbf{y} = \mathbf{y}[1..n]$ is a degenerate cyclic rotation of a degenerate string $\mathbf{x} = \mathbf{x}[1..n]$ if $\mathbf{y}[1..n] = \mathbf{x}[i..n]\mathbf{x}[1..i-1]$ for some $1 \leq i \leq n$ (for $i = 1, \mathbf{y} = \mathbf{x}$).*

Given an order on Δ_{Σ} denoted by the usual symbol $<$, we can compute the BWT of a degenerate string \mathbf{x} in the same way as for a regular string; here we apply lexicorder.

3 Searching for a degenerate pattern in a degenerate string

Let \mathbf{p} and \mathbf{t} be two degenerate strings over Δ_{Σ} of length m and n respectively. We want to find the positions of all the occurrences or matches of \mathbf{p} in \mathbf{t} i.e. we want to compute every j such that $\forall 1 \leq i \leq |\mathbf{p}|$ it holds that $\mathbf{p}[i] \cap \mathbf{t}[i+j] \neq \emptyset$. For determining the matching we will apply the usual backward search but at each step we may generate several different intervals which will be stored in a set H . Then step k (processing $\mathbf{p}[k]$ with $1 \leq k \leq m$) of the backward search can be formalized as follows:

$$\begin{aligned} \text{OneStep}(H, k, C, \text{BWT} = (L, h), \mathbf{p}) = & ((r, s) \mid r = C[c] + rank_c(L, i-1) + 1, \\ & s = C[c] + rank_c(L, j), \\ & r \leq s, (i, j) \in H, c \in \Delta_{\Sigma} \text{ and } c \cap \mathbf{p}[k] \neq \emptyset). \end{aligned}$$

Let $\text{Step}(m, C, \text{BWT}, \mathbf{p}) = \text{OneStep}(\{(1, n)\}, m, C, \text{BWT}, \mathbf{p})$ and $\text{Step}(i, C, \text{BWT}, \mathbf{p}) = \text{OneStep}(\text{Step}(i+1, C, \text{BWT}, \mathbf{p}), i, C, \text{BWT}, \mathbf{p})$ for $1 \leq i \leq m-1$. In words, $\text{Step}(i, C, \text{BWT}, \mathbf{p})$ applies step m through to i of the backward search.

```

DEGENERATEBACKWARDSEARCH( $\mathbf{p}$ ,  $m$ ,  $BWT = (L, h)$ ,  $n$ ,  $C$ )
1  $H \leftarrow \{(1, n)\}$ 
2  $k \leftarrow m$ 
3 while  $H \neq \emptyset$  and  $k \geq 1$  do
4    $H' \leftarrow \emptyset$ 
5   for  $(i, j) \in H$  do
6     for  $c \in \Delta_\Sigma$  such that  $c \cap \mathbf{p}[k] \neq \emptyset$  do
7        $H' \leftarrow H' \cup \{(C[c] + \text{rank}_c(L, i - 1) + 1, C[c] + \text{rank}_c(L, j))\}$ 
8    $H \leftarrow H'$ 
9    $k \leftarrow k - 1$ 
10 return  $H$ 

```

Fig. 1. Backward search for a degenerate pattern in the BWT of a degenerate string.

Lemma 3. *The interval $(i, j) \in \text{Step}(k, C, BWT, \mathbf{p})$ if and only if $\mathbf{p}[k..m]$ is a degenerate prefix of $M_{\mathbf{t}}[h]$ for $i \leq h \leq j$.*

Proof. \implies : By induction. By definition of the array C when $(i, j) \in \text{Step}(m, C, BWT, \mathbf{p})$ then $\mathbf{p}[m]$ is a degenerate prefix of $M_{\mathbf{t}}[h]$, for $i \leq h \leq j$. So assume that the property is true for all the integers k' such that $k < k' \leq m$. If $(r, s) \in \text{Step}(k, C, BWT, \mathbf{p})$ then $r = C[a] + \text{rank}_a(BWT, i - 1) + 1$ and $s = C[a] + \text{rank}_a(BWT, j)$ with $r \leq s$, $(i, j) \in \text{Step}(k+1, C, BWT, \mathbf{p})$, $a \in \Delta_\Sigma$ and $a \cap \mathbf{p}[k] \neq \emptyset$. Thus by the definition of the BWT, $\mathbf{p}[k..m]$ is a degenerate prefix of rows of $M_{\mathbf{t}}[h]$ for $r \leq h \leq s$.

\impliedby : By induction. By definition, if $\mathbf{p}[m]$ is a degenerate prefix of $M_{\mathbf{t}}[h]$ for $r \leq h \leq s$ then $(r, s) \in \text{Step}(m, C, BWT, \mathbf{p})$. So assume that the property is true for all integers $k'+1$ such that $k < k' \leq m$. If $\mathbf{p}[k+1..m]$ is a degenerate prefix of $M_{\mathbf{t}}[h]$ for $i \leq h \leq j$ then $(i, j) \in \text{Step}(k+1, C, BWT, \mathbf{p})$. When $\mathbf{p}[k..m]$ is a degenerate prefix of $M_{\mathbf{t}}[h]$ for $r \leq h \leq s$ then $(r, s) \in \text{OneStep}(\text{Step}(k+1, C, BWT, \mathbf{p}), i, C, BWT, \mathbf{p}) = \text{Step}(k, C, BWT, \mathbf{p})$ by definition of the array C and of the rank function.

We conclude that the property holds for $1 \leq k \leq m$.

Corollary 4. *The interval $(i, j) \in \text{Step}(1, C, BWT, \mathbf{p})$ if and only if \mathbf{p} is a degenerate prefix of $M_{\mathbf{t}}[h]$ for $i \leq h \leq j$.*

The proposed algorithm, see Fig. 1, computes $\text{Step}(1, C, BWT, \mathbf{p})$ by first initializing the variable H with $\{(1, n)\}$ and then performing steps m to 1, while exiting whenever H becomes empty.

The following two lemmas show that the number of intervals in H cannot grow exponentially.

Lemma 5. *The intervals in $\text{OneStep}(\{(i, j)\}, k, C, BWT, \mathbf{p})$ do not overlap.*

Proof. $\text{OneStep}(\{(i, j)\}, k, C, BWT, \mathbf{p})$ will generate one interval for every distinct letter $c \in \Delta_\Sigma$ such that $c \cap \mathbf{p}[k] \neq \emptyset$. Thus these intervals cannot overlap.

Lemma 6. *The intervals in $\text{OneStep}(\{(i, j), (i', j')\}, k, C, \text{BWT}, \mathbf{p})$ with $i \leq j < i' \leq j'$ do not overlap.*

Proof. From Lemma 5, the intervals generated from (i, j) do not overlap, and the intervals generated from (i', j') do not overlap.

Let (r, s) be an interval generated from (i, j) , and let (r', s') be an interval generated from (i', j') . Formally, let r, s, c be such that $r = C[c] + \text{rank}_c(\text{BWT}, i - 1) + 1$, $s = C[c] + \text{rank}_c(\text{BWT}, j)$, $c \in \Delta_\Sigma$ and $c \cap \mathbf{p}[k] \neq \emptyset$. Let r', s', c' be such that $r' = C[c'] + \text{rank}_{c'}(\text{BWT}, i' - 1) + 1$, $s' = C[c'] + \text{rank}_{c'}(\text{BWT}, j')$, $c' \in \Delta_\Sigma$ and $c' \cap \mathbf{p}[k] \neq \emptyset$.

If $c \neq c'$ then (r, s) and (r', s') cannot overlap since $C[c] \leq r \leq s < C[c] + \#\{i \mid \mathbf{t}[i] = c\}$ and $C[c'] \leq r' \leq s' < C[c'] + \#\{i \mid \mathbf{t}[i] = c'\}$. Otherwise if $c = c'$ then since $j < i'$, it follows that $\text{rank}_c(\text{BWT}, j) < \text{rank}_c(\text{BWT}, i' - 1) + 1$ and thus $(r, s) = (C[c] + \text{rank}_c(\text{BWT}, i - 1) + 1, C[c] + \text{rank}_c(\text{BWT}, j))$ and $(r', s') = (C[c] + \text{rank}_c(\text{BWT}, i' - 1) + 1, C[c] + \text{rank}_c(\text{BWT}, j'))$ do not overlap.

Corollary 7. *Let H be a set of non-overlapping intervals. The intervals in $\text{OneStep}(H, k, C, \text{BWT}, \mathbf{p})$ do not overlap.*

We can now state the complexity of the degenerate backward search.

Theorem 8. *The algorithm $\text{DEGENERATEBACKWARDSEARCH}(p, m, \text{BWT}, n, C)$ computes a set of intervals H , where $(i, j) \in H$ if and only if \mathbf{p} is a degenerate prefix of consecutive rows of $M_{\mathbf{t}}[k]$ for $i \leq k \leq j$, in time $O(mn)$ for a constant size alphabet.*

Proof. The correctness comes from Corollary 4. The time complexity mainly comes from Lemma 5 and the fact that the alphabet size is constant.

For conservative degenerate string the overall complexity of the search can be reduced.

Theorem 9. *Let \mathbf{t} be a conservative degenerate string over a constant size alphabet. Let the number of degenerate letters of \mathbf{t} be bounded by a constant q . Then given the BWT of \mathbf{t} , all the intervals in the BWT of occurrences of a pattern \mathbf{p} of length m can be detected in time $O(qm^2)$.*

Proof. The number of intervals of occurrences of \mathbf{p} in \mathbf{t} that do not overlap a degenerate letter is bounded by $k + 1$. The number of intervals of occurrences of \mathbf{p} in \mathbf{t} that overlap one degenerate letter is bounded by k times m . Since there are at most k degenerate letters in \mathbf{t} and since the backward search has m steps the result follows.

From Corollary 7, the number of intervals at each step of the backward search cannot exceed n . However, in practice, it may be worthwhile decreasing the number of intervals further: the next lemma shows that adjacent intervals can be merged. In order to easily identify adjacent intervals we will now store them in a sorted list-like data structure as follows. For two lists I and J the

concatenation of the elements of I followed by the elements of J is denoted by $I \cdot J$.

We proceed to define the operation *Merge* that consists in merging two adjacent intervals: $Merge(\emptyset) = \emptyset$ and $Merge((i, j)) = ((i, j))$, $Merge(((i, j), (j+1, j')) \cdot I) = Merge(((i, j')) \cdot I)$, $Merge(((i, j), (i', j')) \cdot I) = ((i, j)) \cdot Merge(((i', j')) \cdot I)$ for $i' > j + 1$. The next lemma justifies the merging of adjacent intervals in H .

Lemma 10. $Merge(OneStep(((i, j), (j + 1, j')), k, C, BWT, \mathbf{p})) = Merge(OneStep(((i, j')), k, C, BWT, \mathbf{p}))$.

Proof. For a letter $c \in \Delta_\Sigma$ such that $c \cap \mathbf{p}[k] \neq \emptyset$ the intervals generated from (i, j) and $(j + 1, j')$ are, by definition, necessarily adjacent which shows that if $(p, q) \in Merge(OneStep(((i, j), (j + 1, j')), k, C, BWT, \mathbf{p}))$ then $(p, q) \in Merge(OneStep(((i, j')), k, C, BWT, \mathbf{p}))$. The reciprocal can be shown similarly.

This means that H can be implemented with an efficient data structure such as red-black trees adapted for storing non-overlapping and non-adjacent intervals.

4 Experiments

We ran algorithm DEGENERATEBACKWARDSEARCH (DBS) for searching for the occurrences of a degenerate pattern in different random strings: solid strings, degenerate strings and conservative degenerate strings. The alphabet consists of subsets of the DNA alphabet encoded by integers from 1 to 15. Solid letters are encoded by powers of 2 (1, 2, 4 and 8) as in [7]. Then intersections between degenerate letters can be performed by a bitwise `and`. The conservative string contains 500,000 degenerate letters.

We also ran the adaptive Hybrid pattern-matching algorithm of [10], and, since the alphabet size is small we also ran a version of the Backward-Non-Deterministic-Matching (BNDM) adapted for degenerate pattern matching (see [8]). The Hybrid and BNDM are bit-parallel algorithms and have only been tested for pattern lengths up to 64. The patterns have also been randomly generated. For the computation of the BWT we used the SAIS library [9] and for its implementation we used the SDSL library [6]. All the experiments have been performed on a computer with a 1.3 GHz Intel Core i5 processor and 4 GB 1600 MHz DDR3 RAM.

We performed various experiments and present only two of them. For DBS the measured times exclude the construction of the BWT but include the reporting of the occurrences using a suffix array. This can be justified by the fact that, in most cases, strings are given in a compressed form through their BWTs. Fig. 2(a) shows the searching times for different numbers of degenerate patterns of length 8 in a solid string. Times are in centiseconds. It can be seen that when enough patterns have to be searched for in the same string then it is worth using the new DBS algorithm. The BNDM algorithm performs better than the Hybrid one due to the small size of the alphabet which favors shifts based on suffixes

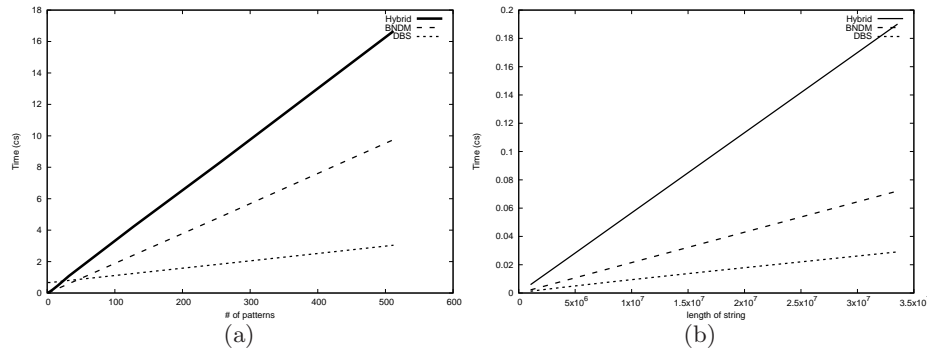


Fig. 2. (a): Running times for searching for several degenerate patterns of length 8 in a solid string of length 5MB. (b): Running times for searching for one degenerate pattern of length 8 in a conservative degenerate string of variable length.

of the pattern rather than shifts based on single letters. Fig. 2(b) shows the searching times for a degenerate pattern of length 8 in conservative degenerate strings of various lengths (for each length the strings contain 10% of degenerate letters). As expected when the length of the string increases the advantage of using DBS also increases.

References

1. Burrows, M., Wheeler, D.J.: A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation (1994)
2. Charras, C., Lecroq, T.: Handbook of exact string matching algorithms. King’s College Publications (2004)
3. Crochemore, M., Iliopoulos, C.S., Kundu, R., Mohamed, M., Vayani, F.: Linear algorithm for conservative degenerate pattern matching. *Eng. Appl. of AI* 51, 109–114 (2016)
4. Daykin, J.W., Watson, B.: Indeterminate string factorizations and degenerate text transformations. *Math. Comput. Sci.* in press (2017)
5. Fischer, M., Paterson, M.: String matching and other products. In: Karp, R. (ed.) *Proceedings of the 7th SIAM-AMS Complexity of Computation*. pp. 113–125 (1974)
6. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: Plug and play with succinct data structures. In: *SEA*. pp. 326–337 (2014)
7. Huang, L., Popic, V., Batzoglu, S.: Short read alignment with populations of genomes. *Bioinformatics* 29(13), i361–i370 (2013)
8. Navarro, G., Raffinot, M.: *Flexible pattern matching in strings - practical on-line search algorithms for texts and biological sequences*. CUP (2002)
9. Nong, G., Zhang, S., Chan, W.H.: Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers* 60(10), 1471–1484 (2011)
10. Smyth, W.F., Wang, S., Yu, M.: An adaptive hybrid pattern-matching algorithm on indeterminate strings. In: *Proc. Stringology*. pp. 95–107 (2008)