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

Daykin, Jacqueline; Groult, Richard; Guesnet, Yannick; Lecroq, Thierry; Lefebvre, Arnaud; Léonard, Martine; Mouchard, Laurent; Prieur-Gaston, Élise; Watson, Bruce

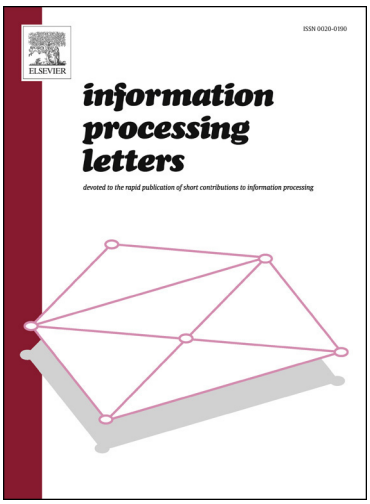
Published in:
Information Processing Letters

DOI:
10.1016/j.ipl.2019.03.003

Publication date:
2019

Citation for published version (APA):
Daykin, J., Groult, R., Guesnet, Y., Lecroq, T., Lefebvre, A., Léonard, M., Mouchard, L., Prieur-Gaston, É., & Watson, B. (2019). Efficient pattern matching in degenerate strings with the Burrows–Wheeler transform. *Information Processing Letters*. https://doi.org/10.1016/j.ipl.2019.03.003

Accepted Manuscript



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

J.W. Daykin, R. Groult, Y. Guesnet, T. Lecroq, A. Lefebvre et al.

PII: S0020-0190(19)30053-5
DOI: https://doi.org/10.1016/j.ipl.2019.03.003
Reference: IPL 5809

To appear in: *Information Processing Letters*

Received date: 18 December 2017
Revised date: 11 January 2019
Accepted date: 8 March 2019

Please cite this article in press as: J.W. Daykin et al., Efficient pattern matching in degenerate strings with the Burrows–Wheeler transform, *Inf. Process. Lett.* (2019), https://doi.org/10.1016/j.ipl.2019.03.003

Highlights

- Formalization of the search in degenerate strings with the Burrows-Wheeler transform
- Proofs that consecutive intervals generated during the search can be merged
- Complexity proof for the conservative case
- Experimental results

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

J. W. Daykin[a,b,c,e], R. Groult[d,c], Y. Guesnet[c], T. Lecroq[c], A. Lefebvre[c], M. Léonard[c], L. Mouchard[c], É. Prieur-Gaston[c], B. Watson[e,f]

[a] *Department of Computer Science, Aberystwyth Univ., Wales & Mauritius*
[b] *Department of Informatics, King's College London, UK*
[c] *Normandie Univ., UNIROUEN, LITIS, 76000 Rouen, France*
[d] *Modélisation, Information et Systèmes (MIS), Univ. Picardie Jules Verne, Amiens, France*
[e] *Department of Information Science, Stellenbosch Univ., South Africa*
[f] *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 $\boldsymbol{t}$ of length n and its Burrows–Wheeler transform we present a new method for searching for a degenerate pattern of length m in $\boldsymbol{t}$ 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 time complexity is $O(qm^2)$ for counting the number of occurrences and $O(qm^2 + occ)$ for reporting the found occurrences where occ is the number of occurrences of the pattern in $\boldsymbol{t}$. Experimental results show that our method performs well in practice.

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

1. Introduction

An *indeterminate* or *degenerate* string $\boldsymbol{x}$ 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]. Then a solid letter is a singleton. Non-solid letters are called degenerate letters. This generalization of a regular (or solid) 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 inter-

[☆]The first author was part-funded by the European Regional Development Fund through the Welsh Government, Grant Number 80761-AU-137 (West)

faces and search engines; bioinformatics activities (DNA sequences and proteins analysis); and cryptanalysis applications.

For solid strings, the main approaches for computing all the occurrences of a given non-empty pattern $\boldsymbol{p}$ in a given non-empty text $\boldsymbol{t}$ have been window-shifting techniques, and applying 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 $\boldsymbol{p}$ 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 $\boldsymbol{p}$ and $\boldsymbol{t}$ over Σ of length m and n respectively is the task of finding all the positions of all the occurrences of $\boldsymbol{p}$ in $\boldsymbol{t}$, that is, computing every j such that $\forall\, 1 \leq i \leq |\boldsymbol{p}|$ it holds that $\boldsymbol{p}[i] \cap \boldsymbol{t}[i+j] \neq \emptyset$. Following the first significant contribution to this problem by Fischer and Paterson [5], interest over the years has produced a faster algorithm by Kalai [9], and practical methods by Smyth *et al.* [12].

Variants of degenerate pattern matching have recently been proposed. A degenerate string is said to be *conservative* if its number of degenerate 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 $\boldsymbol{p}$ and $\boldsymbol{t}$ of total length n containing q degenerate letters in total, the occurrences of $\boldsymbol{p}$ in $\boldsymbol{t}$ in $O(nq)$ time, i.e. linear in the size of the input.

Our contribution is to implement degenerate pattern matching by modifying the existing Burrows–Wheeler pattern matching technique using the standard RAM model of computation. Given a degenerate string $\boldsymbol{t}$ of length n, searching for either a degenerate or solid pattern of length m in $\boldsymbol{t}$ is achieved in $O(mn)$ time; in the conservative scenario with at most q degenerate letters in the pattern and in $\boldsymbol{t}$, the search complexity is $O(qm^2)$ for counting the number of occurrences and $O(qm^2 + occ)$ for reporting the found occurrences where occ is the number of occurrences of the pattern in $\boldsymbol{t}$ – competitive for short patterns. This formalizes and extends the work implemented in BWBBLE [8]. The rest of the paper is organized as follows. In Section 2 we give notation and recall basic definitions. The following Section 3 presents the previous work on the problem. Then in Section 4 we provide proofs for pattern matching in degenerate strings with the Burrows–Wheeler transform. In Section 5 we consider the case of pattern matching in conservative degenerate strings. We discuss our experimental results in Section 6.

2. Notation and definitions

Consider a finite totally ordered alphabet Σ of constant size σ 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 $\boldsymbol{x}$, $\boldsymbol{y}$. The lexicographic order (*lexorder*) on strings is also denoted by the symbol $<$.

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

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

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

The BWT is easily invertible via a linear LF last first mapping [1] using an array C indexed by all the letters c of the alphabet Σ and defined by: $C[c] = \sharp\{i \mid \boldsymbol{x}[i] < c\}$ and $rank_c(\boldsymbol{x}, i)$ which gives the number of occurrences of the letter c in the prefix $\boldsymbol{x}[1\,..\,i]$. A property of the LF mapping is that the i-th occurrence of a letter c in the last column L has the same rank as the i-th occurrence of c in the first column F which can be calculated using the array C and the function *rank*. Traversing the letters repeatedly between L and F recovers the input.

Given the BWT of $\boldsymbol{x}$ it is easy to find the number of occurrences of a pattern $\boldsymbol{p}$ of length m in $\boldsymbol{x}$ by performing a right to left, that is a backwards, scan of $\boldsymbol{p}$ as computed by the pseudocode in Figure 1. Note that although the BWT is defined as a pair, Definition 1, for the backwards search technique, the convention for the argument list is to describe the last column L in the BWT matrix as the string BWT. The procedure returns an interval (i, j) such that $\boldsymbol{p}$ is a prefix of $M_{\boldsymbol{x}}[k]$ for $i \leq k \leq j$, or, it returns $\perp$ if $\boldsymbol{p}$ is not a prefix of any rows of $M_{\boldsymbol{x}}$. Hence the number of occurrences of the pattern is given by the size of the interval. The positions of the occurrences can be computed with the help of a full or sampled suffix array of $\boldsymbol{x}$: a suffix array SA gives the starting positions of the suffixes of $\boldsymbol{x}$ in lexicographical order, so that SA[i] is the starting position of the i-th smallest suffix of $\boldsymbol{x}$; a sampled SA has been sampled at a subset of its indices thus providing succinctness.

In [4], Daykin and Watson present a simple modification of the classic BWT, the *degenerate Burrows–Wheeler transform* which, analogously to the classic case, exhibits clustering of letters in degenerate strings – the focus here is applications of the transforms to pattern matching.

Given an alphabet Σ we define a new alphabet Δ_Σ as the non-empty subsets of Σ: $\Delta_\Sigma = \mathcal{P}(\Sigma) \setminus \{\emptyset\}$, where $\mathcal{P}$ is the usual power set. Formally a non-empty

```
BackwardSearch(p, m, BWT, n, C)
1  (i, j, k) ← (1, n, m − 1)
2  while i ≤ j and k ≥ 1 do
3     c ← p[k]
4     (i, j, k) ← (C[c] + rank_c(BWT, i − 1) + 1, C[c] + rank_c(BWT, j), k − 1)
5  if i ≤ j then
6     return (i, j)
7  else return ⊥
```

Figure 1: Backward search for a pattern $\boldsymbol{p}$ in the BWT of a string $\boldsymbol{x}$.

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

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

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

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

3. Previous work

In [8] the authors present a bioinformatics software tool called BWBBLE that enables performing pattern matching on a pan-genome (collection of genomes of individuals of the same species) that they called a reference multi-genome. BWBBLE can take into account various types of differences between the different genomes. For substitutions it basically aligns the different genomes and the symbol at each position of the reference multi-genome is composed of the union of the symbols of the different genomes at this position. More specifically, it considers strings of Δ_Σ^+ where $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Each element of Σ is represented as a 4-bit integer power of 2 (2^i with $i \in \{0, 1, 2, 3\}$), where an element $S \in \Delta_\Sigma$ is represented by $\sum_{\{s \in S\}} s$. Then instead of using the natural order on integers it uses a Gray code [7] (also known as the reflected binary code) to order the elements of Δ_Σ. With the Gray code two successive values differ only by one bit, such as 1100 and 1101, which enables minimizing the number of separate intervals associated with each of the four symbols of Σ. Then the authors generalize the usual backward search technique, shown in Figure 1, for searching in a reference multi-genome but they do not provide any proofs of correctness. In the next section we provide a proof of correctness of the generalization of the

backward search for the degenerate Burrows-Wheeler transform. We also show that adjacent intervals generated during the backward search can be merged.

4. Searching for a degenerate pattern in a degenerate string

Let $\boldsymbol{p}$ and $\boldsymbol{t}$ 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 $\boldsymbol{p}$ in $\boldsymbol{t}$ i.e. we want to compute every j such that $\forall\, 1 \leq i \leq |\boldsymbol{p}|$ it holds that $\boldsymbol{p}[i] \cap \boldsymbol{t}[i+j] \neq \emptyset$. 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 $\boldsymbol{p}[k]$ with $1 \leq k \leq m$) of the backward search can be formalized as follows: $\mathit{1Step}(H, k, C, BWT = (L, h), \boldsymbol{p}) = (((r, s)) \mid r = C[c] + \mathit{rank}_c(L, i-1) + 1, s = C[c] + \mathit{rank}_c(L, j), r \leq s, (i, j) \in H, c \in \Delta_\Sigma$ and $c \cap \boldsymbol{p}[k] \neq \emptyset)$.

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

Lemma 1. *The interval $(i, j) \in \mathit{Step}(k, C, BWT, \boldsymbol{p})$ if and only if $\boldsymbol{p}[k\,..\,m]$ is a degenerate prefix of $M_{\boldsymbol{t}}[h]$ for $i \leq h \leq j$.*

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

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

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

Corollary 2. *The interval $(i, j) \in \mathit{Step}(1, C, BWT, \boldsymbol{p})$ if and only if $\boldsymbol{p}$ is a degenerate prefix of $M_{\boldsymbol{t}}[h]$ for $i \leq h \leq j$.*

The proposed algorithm, see Figure 2, computes $\mathit{Step}(1, C, BWT, \boldsymbol{p})$ 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 3. *The intervals in $\mathit{1Step}(\{(i, j)\}, k, C, BWT, \boldsymbol{p})$ do not overlap.*

```
DegenerateBackwardSearch(p, m, BWT = (L, h), n, C)
1  (H, k) ← ({(1, n)}, m)
2  while H ≠ ∅ and k ≥ 1 do
3     H' ← ∅
4     for (i, j) ∈ H do
5        for c ∈ Δ_Σ such that c ∩ p[k] ≠ ∅ do
6           H' ← H' ∪ {(C[c] + rank_c(L, i − 1) + 1, C[c] + rank_c(L, j))}
7     (H, k) ← (H', k − 1)
8  return H
```

Figure 2: Backward search for a degenerate pattern in the BWT of a degenerate string.

PROOF. *1Step*$(\{(i,j)\}, k, C, BWT, \boldsymbol{p})$ will generate one interval for every distinct letter $c \in \Delta_\Sigma$ such that $c \cap \boldsymbol{p}[k] \neq \emptyset$. Thus these intervals cannot overlap. □

Lemma 4. *The intervals in 1Step*$(\{(i,j),(i',j')\}, k, C, BWT, \boldsymbol{p})$ *with* $i \leq j < i' \leq j'$ *do not overlap.*

PROOF. From Lemma 3, the intervals generated from (i,j) do not overlap, and similarly 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 $r = C[c] + rank_c(BWT, i-1) + 1$, $s = C[c] + rank_c(BWT, j)$, $c \in \Delta_\Sigma$ and $c \cap \boldsymbol{p}[k] \neq \emptyset$. Let r', s', c' be such that $r' = C[c'] + rank_{c'}(BWT, i'-1) + 1$, $s' = C[c'] + rank_{c'}(BWT, j')$, $c' \in \Delta_\Sigma$ and $c' \cap \boldsymbol{p}[k] \neq \emptyset$.

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

Corollary 5. *Let* H *be a set of non-overlapping intervals. The intervals in 1Step*$(H, k, C, BWT, \boldsymbol{p})$ *do not overlap.*

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

Theorem 6. *The algorithm* DegenerateBackwardSearch(p, m, BWT, n, C) *computes a set of intervals* H*, where* $(i,j) \in H$ *if and only if* $\boldsymbol{p}$ *is a degenerate prefix of consecutive rows of* $M_{\boldsymbol{t}}[k]$ *for* $i \leq k \leq j$*, in time* $O(mn)$ *for a constant size alphabet.*

PROOF. The correctness comes from Corollary 2. The time complexity mainly comes from Lemma 3 and the fact that the alphabet size is constant. □

From Corollary 5, 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 *Mrg* that consists in merging two adjacent intervals: $Mrg(\emptyset) = \emptyset$ and $Mrg((i,j)) = ((i,j))$, $Mrg(((i,j),(j+1,j')) \cdot I) = Mrg(((i,j')) \cdot I)$, $Mrg(((i,j),(i',j')) \cdot I) = ((i,j)) \cdot Mrg(((i',j')) \cdot I)$ for $i' > j+1$. The next lemma justifies the merging of adjacent intervals in H.

Lemma 7. $Mrg(1Step(((i,j),(j{+}1,j')),k,C,BWT,\boldsymbol{p})) = Mrg(1Step(((i,j')),k,C,BWT,\boldsymbol{p}))$.

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

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

Complete example

Let $\boldsymbol{t} = \{\mathtt{c},\mathtt{e}\} \cdot \{\mathtt{c},\mathtt{d}\} \cdot \{\mathtt{a},\mathtt{b},\mathtt{c}\} \cdot \{\mathtt{a},\mathtt{e}\} \cdot \{\mathtt{a},\mathtt{b},\mathtt{c}\}$. Then by renaming $\{\mathtt{a},\mathtt{b},\mathtt{c}\}$ as A, $\{\mathtt{a},\mathtt{e}\}$ as B, $\{\mathtt{c},\mathtt{d}\}$ as C and $\{\mathtt{c},\mathtt{e}\}$ as D, $\boldsymbol{t}$ = DCABA and with the order A < B < C < D we have $BWT(\boldsymbol{t})$ = CBADA see [4] for the ordering technique.

i					
1	D	C	A	B	A
2	C	A	B	A	D
3	A	B	A	D	C
4	B	A	D	C	A
5	A	D	C	A	B

cyclic rotations of $\boldsymbol{t}$

i	F				L
1	A	B	A	D	C
2	A	D	C	A	B
3	B	A	D	C	A
4	C	A	B	A	D
5	D	C	A	B	A

$M_{\boldsymbol{t}}$

Thus the array C is as follows:

	A	B	C	D
C	0	2	3	4

Let $\boldsymbol{p} = \{\mathtt{c}\} \cdot \{\mathtt{a},\mathtt{b}\} \cdot \{\mathtt{a}\}$ and let us search for $\boldsymbol{p}$ in $\boldsymbol{t}$ with the algorithm DegenerateBackwardSearch.

$\boldsymbol{p}[3] = \{\mathtt{a}\} \cap A = \{\mathtt{a},\mathtt{b},\mathtt{c}\} \neq \emptyset$ and $\boldsymbol{p}[3] = \{\mathtt{a}\} \cap B = \{\mathtt{a},\mathtt{e}\} \neq \emptyset$.

Without merging
$1Step(((1,5)),3,C,BWT,\boldsymbol{p}) = ((1,2),(3,3))$

		F	L
$\rightarrow_B$	1	A	C
$\rightarrow_E$	2	A	B
$\Rightarrow$	3	B	A
	4	C	D
	5	D	A

With merging
$1Step(((1,5),3,C,BWT,\boldsymbol{p}) = ((1,3))$

		F	L
$\rightarrow_B$	1	A	C
	2	A	B
$\rightarrow_E$	3	B	A
	4	C	D
	5	D	A

$\boldsymbol{p}[2] = \{\mathtt{a},\mathtt{b}\} \cap A = \{\mathtt{a},\mathtt{b},\mathtt{c}\} \neq \emptyset$ and $\boldsymbol{p}[2] = \{\mathtt{a},\mathtt{b}\} \cap B = \{\mathtt{a},\mathtt{e}\} \neq \emptyset$.

Without merging
$1Step(((1,2),(3,3)),2,C,BWT,\boldsymbol{p}) = ((1,1),(3,3))$

		F	L
$\Rightarrow$	1	A	C
	2	A	B
$\Rightarrow$	3	B	A
	4	C	D
	5	D	A

With merging
$1Step(((1,3)),2,C,BWT,\boldsymbol{p}) = ((1,1),(3,3))$

		F	L
$\Rightarrow$	1	A	C
	2	A	B
$\Rightarrow$	3	B	A
	4	C	D
	5	D	A

$\boldsymbol{p}[1] = \{\mathtt{c}\} \cap A = \{\mathtt{a},\mathtt{b},\mathtt{c}\} \neq \emptyset$, $\boldsymbol{p}[1] = \{\mathtt{c}\} \cap C = \{\mathtt{c},\mathtt{d}\} \neq \emptyset$ and $\boldsymbol{p}[1] = \{\mathtt{c}\} \cap D = \{\mathtt{c},\mathtt{e}\} \neq \emptyset$.

Without merging
$1Step(((1,1),(3,3)),1,C,BWT,\boldsymbol{p}) = ((1,1),(4,4))$

		F	L
$\Rightarrow$	1	A	C
	2	A	B
	3	B	A
$\Rightarrow$	4	C	D
	5	D	A

With merging
$1Step(((1,1),(3,3)),1,C,BWT,\boldsymbol{p}) = ((1,1),(4,4))$

		F	L
$\Rightarrow$	1	A	C
	2	A	B
	3	B	A
$\Rightarrow$	4	C	D
	5	D	A

$\rightarrow_B$ stands for the beginning of an interval, $\rightarrow_E$ for the end of an interval and $\Rightarrow$ for an interval of one element.
Thus $\boldsymbol{p}$ has two occurrences in $\boldsymbol{t}$.

5. Degenerate pattern in a conservative degenerate string

For conservative degenerate strings the search complexity can be reduced.

Theorem 8. *Let $\boldsymbol{t}$ and $\boldsymbol{p}$ be two conservative degenerate strings over a constant size alphabet such that their total number of degenerate letters is bounded by a constant q. Then given the BWT of $\boldsymbol{t}$, all the intervals in the BWT of occurrences of a pattern $\boldsymbol{p}$ of length m can be detected in time $O(qm^2)$.*

PROOF. The largest number of intervals at the first step of the backward search is $O(1)$ for solid letters and q for the degenerate letters. Then at each step the q intervals for the degenerate letters will generate q other intervals while each interval corresponding to a solid letter will generate $O(1)$ intervals for solid letters and q intervals for the degenerate letters. Since there are m steps the result follows. □

Corollary 9. *Let $\boldsymbol{t}$ and $\boldsymbol{p}$ be two conservative degenerate strings over a constant size alphabet such that their total number of degenerate letters is bounded by a constant q. Then given the BWT of $\boldsymbol{t}$, the occurrences of $\boldsymbol{p}$ of length m can be reported in time $O(qm^2 + occ)$ where occ is the number of occurrences of $\boldsymbol{p}$ in $\boldsymbol{t}$.*

6. 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 [8]. Then intersections between degenerate letters can be performed by a bitwise `and` operation. But contrary to [8] we used the natural order on integers. The patterns have also been randomly generated.

We additionally ran the adaptive Hybrid pattern-matching algorithm of [12], 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 [10]). The Hybrid and BNDM are bit-parallel algorithms and have only been tested for pattern lengths up to 64. The source of our method is available at `https://github.com/YGuesnet/dbwt` and the inputs have been made compatible to those of [3]. However we excluded the algorithm in [3] from the comparison since it is more general and performs slower than the two previously mentioned algorithms. For the computation of the BWT we used the SAIS library [11] and the SDSL library [6]. All the experiments have been performed on a computer with a 3.5 GHz i7-4800MQ processor and 16 GB RAM.

We performed various experiments and select four of them for presentation here. 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. Figure 3(a) shows the searching times for a degenerate pattern of length 8 in solid strings of various lengths with an alphabet of size 4, where clearly when the length of the string increases the advantage of using DBS also increases. Figure 3(b) shows the searching times for various numbers of degenerate patterns of length 8 in a solid string with an alphabet of size 8. Running times include preprocessing times for all methods. 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. Figure 3(c) shows the searching times with DBS for various numbers of degenerate patterns of length 8 in degenerate

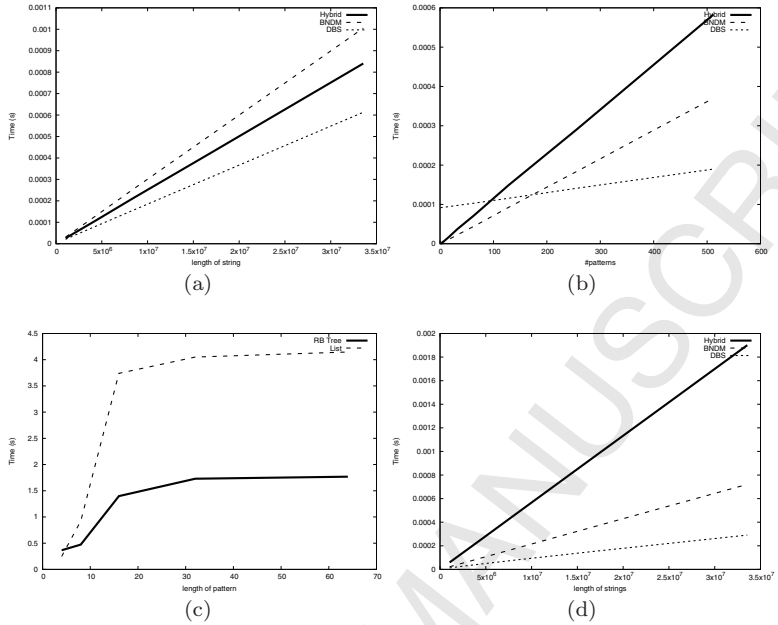


Figure 3: (a): Running times for searching for a degenerate pattern of length 8 in a solid string of various lengths with $\sigma = 4$. (b): Running times for searching for several degenerate patterns of length 8 in a solid string of length 250MB with $\sigma = 8$. (c): Running times with DBS for searching for degenerate patterns in a conservative degenerate string of length 250MB with $\sigma = 4$ and 25M degenerate letters when the list of intervals is implemented with a red-black tree or with a linked list. (d): Running times for searching for one degenerate pattern of length 8 in a conservative degenerate string of variable length with $500,000$ degenerate letters.

strings with an alphabet of size 4 when intervals are stored with red-black trees or with linked lists. As expected, for efficiency it is worth using an advanced data structure, such as red-black trees, for merging intervals. All times are in seconds. Figure 3(d) 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).

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. 11(2), 209–218 (2017)

[5] Fischer, M.J., Paterson, M.S.: 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] Gray, F.: Pulse code communication (1953), U.S. Patent No. 2,632,058

[8] Huang, L., Popic, V., Batzoglou, S.: Short read alignment with populations of genomes. Bioinformatics 29(13), i361–i370 (2013)

[9] Kalai, A.: Efficient pattern-matching with don't cares. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms. vol. 2, pp. 655–656 (2002)

[10] Navarro, G., Raffinot, M.: Flexible pattern matching in strings - practical on-line search algorithms for texts and biological sequences. CUP (2002)

[11] Nong, G., Zhang, S., Chan, W.H.: Two efficient algorithms for linear time suffix array construction. IEEE Trans. Computers 60(10), 1471–1484 (2011)

[12] Smyth, W.F., Wang, S.: An adaptive hybrid pattern-matching algorithm on indeterminate strings. Int. J. Found. Comput. Sci. 20(06), 985–1004 (2009)