DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS C
REPORT C-2004-20

# New Search Algorithms and Time/Space Tradeoffs for Succinct Suffix Arrays

Veli Mäkinen and Gonzalo Navarro

UNIVERSITY OF HELSINKI
FINLAND

**New Search Algorithms and Time/Space Tradeoffs for Succinct Suffix Arrays**

Veli Mäkinen and Gonzalo Navarro

Department of Computer Science
P.O. Box 26, FIN-00014 University of Helsinki, Finland
vmakinen@cs.Helsinki.FI, gnavarro@dcc.uchile.cl

Technical report, Series of Publications C, Report C-2004-20
Helsinki, April 2004, 36 pages

**Abstract**

This paper is about *compressed full-text indexes*. That is, our goal is to represent full-text indexes in as small space as possible and, at the same time, retain the functionality of the index. The most important functionality for a full-text index is the ability to find out whether a given pattern string occurs inside the text string on which the index is built. In addition to supporting this *existence query*, full-text indexes usually support *counting queries* and *reporting queries*; the former is for counting the number of times the pattern occurs in the text, and the latter is for reporting the exact locations of the occurrences.

Suffix trees and arrays are well-known full-text indexes that support the above queries nearly optimally. This optimality refers only to the time complexity of the queries, since in *space requirement* neither are optimal; both structures occupy $O(n \log n)$ bits, where $n$ is the length of the text. Notice that the text itself can be represented in $n \log \sigma$ bits, where $\sigma$ is the alphabet size. Since the text (in some form) is crucial for the full-text index, it is convenient to express the size of an index as the total size of the structure plus the text. Then obviously $O(n \log \sigma)$ space for a full-text index would be optimal. For compressible texts it is still possible to achieve space requirement that is proportional to the *entropy* of the text.

In recent years, many compressed full-text indexes have been proposed. Most of them are based on representing the suffix array in succinct form. The most succinct index achieves both the information theoretic lower bound (size proportional to the entropy) and, at the same time, optimal query time for counting queries. However, this solution assumes a constant alphabet, and hides exponential alphabet-size factors, making the approach impractical but only for small alphabets. Other almost optimal solutions exist that work well also on large alphabets.

In this paper, we focus on two recent solutions for implementing succinct suffix arrays. We start with a comprehensive and self-contained exposition of the techniques that yield efficient solutions for the compressed full-text index problem. Then, we propose many alternative implementations that are in several aspects better than previously known. In particular, we show how to avoid the exponential alphabet-size factor in the most succinct solution. Also, we show how to improve both the space and time requirent of an almost optimal solution. Most of our results are very practical and can be — and some are already — implemented as such.

**Computing Reviews (1998) Categories and Subject Descriptors:**

F.2.2 Analysis of Algorithms and Problem Complexity: Nonnumerical Algorithms and Problems—pattern matching, sorting and searching, geometrical problems and computations

**General Terms:**

Algorithms, Theory, Compression

**Additional Key Words and Phrases:**

Suffix tree, suffix array, full-text index, data-structure compression

# Contents


**1 Introduction** **1**

**2 Preliminaries** **3**
- 2.1 Definitions . . . . . 3
- 2.2 The FM-index and the Compressed Suffix Array . . . . . 3
  - 2.2.1 The FM-index . . . . . 3
  - 2.2.2 The Compressed Suffix Array (CSA) Structure . . . . . 5
- 2.3 Reporting Occurrences and Displaying the Text . . . . . 5

**3 Backward Searching** **7**
- 3.1 Summary of Results and Related Work . . . . . 7
- 3.2 Replacing *Occ* Structure by Individual Bit Arrays . . . . . 8
- 3.3 Balanced Hierarchy of Bit Arrays . . . . . 10
  - 3.3.1 Basic Structure . . . . . 10
  - 3.3.2 Improving the Average Case . . . . . 11
  - 3.3.3 Ensuring $O(m \log \sigma)$ Worst Case Search Time . . . . . 12
- 3.4 Reporting Occurrences and Displaying the Text . . . . . 13
- 3.5 Conclusions . . . . . 13

**4 Backward Searching and RL-encoding** **15**
- 4.1 Summary of Results . . . . . 15
- 4.2 The RLFM-Index . . . . . 15
- 4.3 Space Analysis . . . . . 18
- 4.4 Conclusions . . . . . 19

**5 Backward Searching in CSA** **21**
- 5.1 Summary of Results and Related Work . . . . . 21
- 5.2 Backward Search on CSA . . . . . 22
- 5.3 Improved Search Complexity . . . . . 23
- 5.4 A Simpler and Smaller CSA Structure . . . . . 25
- 5.5 Reporting Occurrences . . . . . 29
- 5.6 A Secondary Memory Implementation . . . . . 29
- 5.7 A Distributed Implementation . . . . . 30
- 5.8 Conclusions . . . . . 31

**6 Discussion** **33**

**A Entropy Bound on RLE of BWT** **37**

# Chapter 1

# Introduction

The classical problem in string matching is to determine the *occ* occurrences of a short pattern $P = p_1p_2 \dots p_m$ in a large text $T = t_1t_2 \dots t_n$. Text and pattern are sequences of characters over an alphabet $\Sigma$ of size $\sigma$. In practice one wants to know the text positions of those *occ* occurrences, and usually also a text context around them. Usually the same text is queried several times with different patterns, and therefore it is worthwhile to preprocess the text in order to speed up the searches. Preprocessing builds an index structure for the text.

To allow fast searches for patterns of any size, the index must allow access to all suffixes of the text (the $i$th suffix of $T$ is $t_it_{i+1} \dots t_n$). These kind of indexes are called *full-text indexes.* Optimal query time, which is $O(m + occ)$ as every character of $P$ must be examined and the *occ* occurrences must be reported, can be achieved by using the *suffix tree* [27, 15, 25] as the index.

The suffix tree takes much more memory than the text. In general, it takes $O(n \log n)$ bits, as the text takes $n \log \sigma$ bits[1]. A smaller constant factor is achieved by the *suffix array* (SA) [12]. Still, the space complexity does not change. Moreover, the searches take $O(m \log n + occ)$ time with the SA (this can be improved to $O(m + \log n + occ)$ using twice the original amount of space [12]).

The large space requirement of full-text indexes has raised a natural question: Is there an index that occupies the same amount of space as the text itself? This question has been recently answered affirmatively. Nowadays, there are several so-called *succinct* full-text indexes that achieve good tradeoffs between search time and space complexity [10, 3, 7, 21, 24, 5, 17, 19, 18]. Most of these are *opportunistic* as they take less space than the text itself, and also *self-indexes* as they contain enough information to reproduce the text: A self-index does not need the text to operate.

For example, the FM-index of Ferragina and Manzini [3] is a self-index that takes in practice the same amount of space as the *compressed* text. The size of the index is in fact close to the best compression ratios achievable, and the text is included in the index. Existence and counting queries on this index take the optimal $O(m)$ time. There is, however, an exponential dependence on the alphabet size. A practical implementation [4] avoids this by using heuristics

[1]By log we mean $\log_2$ in this paper.

with the side effect of not achieving the optimal search time anymore.

In this paper we mostly concentrate on improving the FM-index, in particular its large alphabet dependence. This dependence shows up not only in the space usage, but also in the time to show an occurrence position and display text substrings. The compressed suffix array (CSA) of Sadakane [21] can be seen as a tradeoff with larger search time but much milder dependence on the alphabet size in all the other aspects. We will show in this paper that the CSA can be searched using almost the same algorithm as the FM-index. This gives one way of avoiding the large alphabet factors. We will also show some other ways to achieve the same result by directly modifying the way FM-index is implemented. Our best solutions take $\log \sigma$ times more space than FM-index on (very) small alphabets, but as the alphabet size is increased our solution soon becomes more space-efficient due to the exponential alphabet factor of the original FM-index.

The paper is organized as follows. Chapter 2 introduces the FM-index and the novel *backward search* algorithm it uses. Chapter 3 gives new implementations of an internal function used in the backward search. These new implementations avoid the exponential alphabet factor of the original FM-index. In Chapter 4 we show how run-length compression can be combined with the backward search. This improves the space requirement of the solutions given in Chapter 3, while the search times remain the same. Chapter 5 considers the implementation of the backward search on top of the CSA. Interestingly, the space and time complexities become almost identical to the solutions in the two former chapters that build on the FM-index. The representation of these structures is still completely different. Finally, we discuss the practicality (how much space they take and how fast they are on real strings) of the indexes and possible further improvements in Chapter 6.

# Chapter 2

# Preliminaries

## 2.1 Definitions

A *string* is a sequence of *characters* from an alphabet $\Sigma$. We sometimes use symbols or letters to denote characters. The size of the alphabet is $\sigma$, and for clarity of the exposition, we assume that $\Sigma = \{0, 1, \ldots, \sigma - 1\}$. We denote by $T = t_1 t_2 \cdots t_n$ a *text* string. A substring of the text is denoted by $T[i, j] = t_i t_{i+1} \cdots t_j$. An empty string is denoted $\epsilon$. If $i > j$, then $T[i, j] = \epsilon$. A *suffix* of $T$ is any substring $T[i, n]$. A *prefix* of $T$ is any substring $T[1, i]$. A *cyclic shift* of $T$ is any string $t_i t_{i+1} \cdots t_n t_1 t_2 \cdots t_{i-1}$. The *lexicographic order* of two strings is the natural order induced by the alphabet order: if two string have the same first $k$ letters, then their order depends on the order of their $(k+1)$th letter. We assume that a special *endmarker* "\$" has been appended to $T$, such that the endmarker is smaller than any other text character.

## 2.2 The FM-index and the Compressed Suffix Array

### 2.2.1 The FM-index

The FM-index [3] is based on the *Burrows-Wheeler transform (BWT)* [1], which produces a permutation of the original text, denoted by $T^{bwt} = bwt(T)$. String $T^{bwt}$ is a result of the following *forward* transformation: (1) Append to the end of $T$ a special end marker $\#$, which is lexicographically smaller than any other character; (2) form a *conceptual* matrix $\mathcal{M}$ whose rows are the cyclic shifts of the string $T\#$, sorted in lexicographic order; (3) construct the transformed text $L$ by taking the last column of $\mathcal{M}$. The first column is denoted by $F$.

The *suffix array (SA)* $\mathcal{A}$ of text $T\#$ is essentially the matrix $\mathcal{M}$: $\mathcal{A}[i] = j$ iff the $i$th row of $\mathcal{M}$ contains string $t_j t_{j+1} \cdots t_n \# t_1 \cdots t_{j-1}$. Given the suffix array, the search for the occurrences of the pattern $P = p_1 p_2 \cdots p_m$ is now trivial. The occurrences form an interval $[sp, ep]$ in $\mathcal{A}$ such that suffixes $t_{\mathcal{A}[i]} t_{\mathcal{A}[i]+1} \cdots t_n$, $sp \le i \le ep$, contain the pattern as a prefix. This interval can be searched for using two binary searches in time $O(m \log n)$.

The suffix array of text $T$ is represented implicitly by $T^{bwt}$. The novel idea of the FM-index is to store $T^{bwt}$ in compressed form, and to simulate the search

```
Algorithm FM_Search(P[1,m],T^bwt[1,n])
c = P[m]; i = m;
sp = C[c] + 1; ep = C[c + 1];
while ((sp ≤ ep) and (i ≥ 2) do
      c = P[i − 1];
      sp = C[c] + Occ(T^bwt, c, sp − 1)+1;
      ep = C[c] + Occ(T^bwt, c, ep);
      i = i − 1;
if (ep < sp) then return "not found" else return "found (ep − sp + 1) occs".
```

Figure 2.1: Algorithm for counting the number of occurrences of $P[1, m]$ in $T[1, n]$.

in the suffix array. To describe the search algorithm, we need to introduce the *backward* BWT that produces $T$ given $T^{bwt}$:

1. Compute the array $C[1 \ldots \sigma]$ storing in $C[c]$ the number of occurrences of characters $\{\#, 1, \ldots, c-1\}$ in the text $T$. Notice that $C[c] + 1$ is the position of the first occurrence of $c$ in $F$ (if any).

2. Define the *LF-mapping* $LF[1 \ldots n+1]$ as follows: $LF[i] = C[L[i]] + Occ(L, L[i], i)$, where $Occ(X, c, i)$ equals the number of occurrences of character $c$ in the prefix $X[1, i]$.

3. Reconstruct $T$ backwards as follows: set $s = 1$ and $T[n] = L[1]$ (because $\mathcal{M}[1] = \#T$); then, for each $n-1, \ldots, 1$ do $s \leftarrow LF[s]$ and $T[i] \leftarrow L[s]$.

We are now ready to describe the novel *backward search algorithm* given in [3] (Fig. 2.1). It finds the interval of $\mathcal{A}$ containing the occurrences of the pattern $P$. It uses the array $C$ and function $Occ(X, c, i)$ defined above. Using the properties of the backward BWT, it is easy to see that the algorithm maintains the following invariant [3]: *At the ith phase, the variable sp points to the first row of $\mathcal{M}$ prefixed by $P[i, m]$ and the variable ep points to the last row of $\mathcal{M}$ prefixed by $P[i, m]$.* The correctness of the algorithm follows from this observation.

Ferragina and Manzini [3] go on to describe an implementation of $Occ(T^{bwt}, c, i)$ that uses a compressed form of $T^{bwt}$; they show how to compute $Occ(T^{bwt}, c, i)$ for any $c$ and $i$ in constant time. However, to achieve this they need exponential space (in the size of the alphabet). In a practical implementation [4] this was avoided, but the constant time guarantee for answering $Occ(T^{bwt}, c, i)$ was no longer valid.

The FM-index can also show the text positions where $P$ occurs, and display any text substring. The details are deferred to next chapter.

In the next chapter, we give alternative implementations for the $Occ()$ function, whose dependence on the alphabet size is null or very mild.

### 2.2.2 The Compressed Suffix Array (CSA) Structure

Recall that the suffix array $\mathcal{A}$ of $T$ is the set of suffixes $1 \dots n$, arranged in lexicographic order. That is, the $\mathcal{A}[i]$-th suffix is lexicographically smaller than the $\mathcal{A}[i+1]$-th suffix of $T$ for all $1 \leq i < n$.

Given the suffix array, the search for the occurrences of the pattern $P = p_1 p_2 \cdots p_m$ is now trivial. The occurrences form an interval $[sp, ep]$ in $\mathcal{A}$ such that suffixes $t_{\mathcal{A}[i]} t_{\mathcal{A}[i]+1} \cdots t_n$, $sp \leq i \leq ep$, contain the pattern as a prefix. This interval can be searched for using two binary searches in time $O(m \log n)$.

The *compressed suffix array (CSA)* structure of Sadakane [21] is based on that of Grossi and Vitter [7]. In the CSA, the suffix array $\mathcal{A}[1 \dots n]$ is represented by a sequence of numbers $\Psi(i)$, such that $\mathcal{A}[\Psi(i)] = \mathcal{A}[i] + 1$. Furthermore, the sequence is differentially encoded, $\Psi(i) - \Psi(i-1)$. If there is a *self-repetition*, that is $\mathcal{A}[j \dots j+\ell] = \mathcal{A}[i \dots i+\ell] + 1$, then $\Psi(i \dots i+\ell) = j \dots j+\ell$, and $\Psi(i) - \Psi(i-1) = 1$ in all that area. Hence the differential array is encoded with a method that favors small numbers and permits constant time access to $\Psi$. Note in particular that $\Psi$ values are increasing in the areas of $\mathcal{A}$ where the suffixes start with the same character $a$, because $ax < ay$ iff $x < y$.

Additionally, the CSA stores an array $C[1 \dots \sigma]$, such that $C[c]$ is the number of occurrences of characters $\{\$, 1, \dots, c-1\}$ in the text $T$. Notice that all the suffixes $\mathcal{A}[C[c]+1] \dots \mathcal{A}[C[c+1]]$ start with character $c$. The text is discarded.

A binary search over $\mathcal{A}$ is simulated by extracting from the CSA strings of the form $t_{\mathcal{A}[i]} t_{\mathcal{A}[i]+1} t_{\mathcal{A}[i]+2} \dots$ for any index $i$ required by the binary search. The first character $t_{\mathcal{A}[i]}$ is easy to obtain because all the first characters of suffixes appear in order when pointed from $\mathcal{A}$, so $t_{\mathcal{A}[i]}$ is the character $c$ such that $C[c] < i \leq C[c+1]$. This is found in constant time by using small additional structures based on the four-Russians technique. Once the first character is obtained, we move to $i' \leftarrow \Psi(i)$ and go on with $t_{\mathcal{A}[i']} = t_{\mathcal{A}[i]+1}$. We continue until the result of the lexicographical comparison against the pattern $P$ is clear. The overall search complexity is the same as with the original suffix array, $O(m \log n)$.

It is shown [21] that the CSA takes $n(H_0 + O(\log \log \sigma))$ bits of space, while supporting direct access to $\Psi$.

We will show in Chapter 5 that the backward search algorithm of FM-index can also be carried out using CSA. In fact, it turns out that backward searching does not require direct access to $\Psi$, and we get a simpler and more space-efficient representation of the CSA.

## 2.3 Reporting Occurrences and Displaying the Text

Up to now we have focused on the search time, that is, the time to determine the suffix array interval containing all the occurrences. In practice, one needs also the text positions where they appear, as well as a text context. Since self-indexes replace the text, in general one needs to extract any text substring from the index.

Given the suffix array interval that contains the *occ* occurrences found, the FM-index reports their positions in $O(occ\ \sigma \log^{1+\varepsilon} n)$ time, for any $\varepsilon > 0$

(which appears in the multiplicative constant of the sublinear space component). The CSA can report them in $O(occ \log^{\varepsilon} n)$ time, where $\varepsilon$ is paid in the $nH_0/\varepsilon$ space. Similarly, a text substring of length $L$ can be displayed in time $O(\sigma(L + \log^{1+\varepsilon} n))$ by the FM-index and $O(L + \log^{\varepsilon} n)$ by the CSA.

Let us review how the FM-index reports occurrences and displays text. We modify the exposition slightly for our purposes; in the next chapter we will give variations of this scheme that improve the time bounds.

We sample one out of $\frac{2}{\varepsilon} \log n$ text positions, and store an array $TS$ with the $\frac{\varepsilon n}{2 \log n}$ suffix array entries pointing to them, in increasing text position order. This requires $\frac{\varepsilon n}{2}$ bits. The same suffix array positions are now sorted and the text positions pointed to by them are stored in array $ST$, for other $\frac{\varepsilon n}{2}$ bits. Finally, we store an array $S$ of $n$ bits telling, at position $i$, whether the $i$th suffix array entry is in the sampled set or not. Again, according to [20], a bit array where there are $\ell$ 1's can be represented using $\log \binom{n}{\ell} + o(n)$ bits, while still supporting constant time access and $rank$. Hence array $S$ requires $\log \binom{n}{n/\log n} + o(n) = O(n \log \log n / \log n) + o(n) = o(n)$ bits. Overall, we have spent $\varepsilon n(1 + o(1))$ bits for these three arrays.

Let us focus first in how to determine the text position corresponding to a suffix array entry $i$. Use bit array $S[i]$ to determine whether it is sampled or not. If it is, then find the corresponding text position in $ST[rank(S,i)]$ and we are done. Then, use the LF-mapping to determine position $i'$ whose value is $\mathcal{A}[i'] = \mathcal{A}[i] - 1$. Use $S[i']$ to determine whether suffix array position $i'$ is present in $ST$. If it is, report text position $1 + ST[rank(S,i')]$ and finish. Otherwise, continue with $i''$ such that $\mathcal{A}[i''] = \mathcal{A}[i'] - 1$, reporting $2 + ST[rank(S,i'')]$, and so on. The process must finish after $\frac{2}{\varepsilon} \log n = O\left(\frac{1}{\varepsilon} \log n\right)$ steps because we are considering consecutive text positions and these are sampled at regular intervals.

The LF-mapping is defined as $i' = C[T^{bwt}[i]] + Occ(T^{bwt}, T^{bwt}[i], i)$. Now, observe [3] that $T^{bwt}[i] = c$ iff $Occ(T^{bwt}, c, i) = Occ(T^{bwt}, c, i-1) + 1$ (while for $c' \neq c$, $Occ(T^{bwt}, c', i) = Occ(T^{bwt}, c', i-1)$). Hence one can try $Occ$ for every character until the good one is found. This takes $O(\sigma)$ time.

In order to show a text substring of length $L$, the first step is to find the smallest sampled text position that follows the substring in the text. This can be directly found in array $TS$ and the corresponding suffix array position can be obtained. From there on, we perform at most $L + \frac{2}{\varepsilon} \log n$ steps going backward in the text position by position until traversing the whole text we have to display. For every text position that has to be displayed we determine the current character (which is precisely $T^{bwt}[i]$) and use it for the LF-mapping so as to find the previous text position. Just as before, the time complexity is $O(\sigma(L + \frac{2}{\varepsilon} \log n))$.

Exactly the same search mechanisms can be carried out with CSA. The constant time access to $\Psi$ saves the $\sigma$ factor.

# Chapter 3

# Efficient Implementations of Backward Searching

In this chapter we present two structures that contribute to the space/time tradeoff of FM-index by modifying an internal function in its backward search method. The space requirement of our structures is independent of the alphabet size, and is slightly lower than that of the compressed suffix array (CSA) of Sadakane. The first structure retains all the FM-index times, including the good one for searching and the bad ones for reporting occurrences and displaying text. Hence its advantage over the FM-index is its independence on the alphabet, and its advantage over the CSA is its lower space usage and search time. The second structure searches in $O(m(H_0+1))$ average time and $O(m \log \sigma)$ worst case time, being superior to the CSA both in space and search time. Its interest compared to the FM-index and to our first structure lies in the better complexities to report occurrences and display text substrings.

## 3.1 Summary of Results and Related Work

Table 3.1 shows the complexities obtained. We recall that $0 < \gamma < 1$ and $\varepsilon > 0$ are constants that can be made smaller by increasing the sublinear terms hidden in the $O()$ space complexities, $H_k$ is the $k$th order empirical entropy of $T$, $occ$ is the number of occurrences to report, and $L$ is the length of the text to display.

Very recently Grossi, Gupta, and Vitter [5, 6] have also proposed improvements to compressed suffix arrays. Unlike our work, they do not build directly on the FM-index. They obtain a structure that takes $nH_k + O(n \log\log n / \log_\sigma n)$ bits of space and searches in $O(m \log \sigma + \text{polylog}(n))$ time. Their space usage is better than any other compressed index, including ours. In search time, they pay an $O(\log \sigma)$ multiplicative factor and an $O(\text{polylog}(n))$ additive factor that are not present in the FM-index nor in our first structure. Also, they need at least $o(\log^2 n)$ time to report each occurrence, while most other indexes need only $O(\log n)$ time.

| Structure | Space |
|---|---|
| FM-index | $5H_kn + O\left((\sigma\log\sigma + \log\log n)\frac{n}{\log n} + n^\gamma\sigma^{\sigma+1}\right)$ |
| CSA | $(H_0/\varepsilon + O(\log\log\sigma))n$ |
| **Ours1** | $(H_0 + 1.44 + \varepsilon)(1 + o(1))n$ |
| **Ours2** | $(H_0 + 1 + \varepsilon)(1 + o(1))n$ ($+n$ to obtain the worst cases) |

| Structure | Search | Report | Display |
|---|---|---|---|
| FM-index | $O(m)$ | $O\left(occ\ \sigma\ \log^{1+\varepsilon} n\right)$ | $O\left(\sigma\ (L + \log^{1+\varepsilon} n)\right)$ |
| CSA | $O(m\log n)$ | $O\left(occ\ \log^{\varepsilon} n\right)$ | $O\left((L + \log^{\varepsilon} n)\right)$ |
| **Ours1** | $O(m)$ | $O\left(\frac{1}{\varepsilon}occ\ \sigma\ \log n\right)$ | $O\left(\frac{1}{\varepsilon}\sigma\ (L + \log n)\right)$ |
| **Ours2**(avg) | $O(m(H_0+1))$ | $O\left(\frac{1}{\varepsilon}occ\ H_0\ \log n\right)$ | $O\left(\frac{1}{\varepsilon}H_0(L + \log n)\right)$ |
| **Ours2**(worst) | $O(m\log\sigma)$ | $O\left(\frac{1}{\varepsilon}occ\ \log\sigma\ \log n\right)$ | $O\left(\frac{1}{\varepsilon}\log\sigma(L + \log n)\right)$ |

Table 3.1: Comparison of space and time complexities of different succinct suffix arrays.

## 3.2 Replacing *Occ* Structure by Individual Bit Arrays

Recall the backward search algorithm from previous chapter. We will now give an efficient implementation of the *Occ* function it uses.

Our first succinct suffix array variant takes $n(H_0 + 1.44)(1 + o(1))$ bits of space, with essentially no dependence on the alphabet size $\sigma$. All the time complexities of the FM-index are preserved, including the optimal $O(m)$ search time.

The idea is to replace the *Occ* implementation of the FM-index so as to have one bitmap $B_c$ per text character $c$, so that $B_c[i] = 1$ iff $T^{bwt}[i] = c$. Hence $Occ(T^{bwt}, c, i) = rank(B_c, i)$, where $rank(B, i)$ is the number of 1's in $B[1 \ldots i]$ and can be computed in constant time using $o(n)$ extra bits [9, 16, 2]. Note that we can completely remove $T^{bwt}$, together with all its access structures, since this was only required to compute *Occ*. This is interesting because the index uses the BWT concept without representing the BWT transformed text at all. The huge additive term $O(n^\gamma\sigma^{\sigma+1})$ of the FM-index comes from the structure for fast access to $T^{bwt}$, while the multiplicative constant $O((\sigma\log\sigma)/\log n)$ appears in the representation of *Occ*. Both terms vanish once we remove *Occ* and $T^{bwt}$.

Let us consider which is our space complexity using the new structures. According to [20], a bit array where there are $\ell$ 1's can be represented using $\log\binom{n}{\ell} + o(\ell) + O(\log\log n)$ bits, while still supporting constant time access and constant time *rank* function. [1] Therefore, given the alphabet of text characters

[1] ERRATUM 9th December 2004: We noticed later that we use incorrectly the result in [20]. The result with the mentioned space complexity only gives *rank* for positions containing 1 in the bitmap. The structure with full *rank* functionality has an $o(n)$ term instead of the $o(\ell) + O(\log\log n)$. This affects the analysis so that we get an additional $o(\sigma n)$ term, which is too much. However, there is a way to use *rank* structures so that one obtains the same space complexity as we claim here, but restricted to alphabets $\sigma = O(polylog(n))$. For details, see [22] for an almost identical (but correct) proposal that builds on the compressed suffix arrays, or our subsequent work on *rank* structures for sequences that directly gives the same

$c_1 \ldots c_\sigma$ with text frequencies $\ell_1 \ldots \ell_\sigma$, so that $\sum_{i=1\ldots\sigma} \ell_i = n$, we call $p_i = \ell_i/n$ the empirical probability of character $c_i$. The representation as sparse bitmaps needs

$$\sum_{i=1\ldots\sigma} \left( \log \binom{n}{\ell_i} + o(\ell_i) + O(\log\log n) \right)$$

bits. By Stirling's approximation, calling $\ell = \ell_i$ and $p = \ell/n$, we have

$$\begin{aligned}\binom{n}{\ell} &= \frac{n^n}{\ell^\ell (n-\ell)^{n-\ell}} \sqrt{\frac{n}{2\pi\ell(n-\ell)}} \left(1 + O\left(\frac{1}{\min(\ell, n-\ell)}\right)\right) \\ &= \frac{1}{(p^p(1-p)^{1-p})^n \sqrt{2\pi np(1-p)}} \left(1 + O\left(\frac{1}{\min(\ell, n-\ell)}\right)\right)\end{aligned}$$

and therefore

$$\begin{aligned}\log \binom{n}{\ell} &= n\left(p \log \frac{1}{p} + (1-p)\log\frac{1}{1-p}\right) - \frac{\log n}{2} + O(\log p + \log(1-p)) + O(1) \\ &= n\left(p \log \frac{1}{p} + (1-p)\log\frac{1}{1-p}\right) + O(\log n)\end{aligned}$$

where we have used the fact that, except for a trivial text, $1/n \leq p \leq 1 - 1/n$, and therefore $\log p$ and $\log(1-p)$ are both $O(\log n)$. Now, summing up over our $\ell_i$ values we get

$$\begin{aligned}&\sum_{i=1\ldots\sigma} \left( \log \binom{n}{\ell_i} + o(\ell_i) + O(\log\log n) \right) \\ &= n \sum_{i=1\ldots\sigma} \left( p_i \log \frac{1}{p_i} + (1-p_i)\log\frac{1}{1-p_i} \right) \ + \ o(n) \ + \ O(\sigma\log n) \\ &= nH_0 \ + \ n \sum_{i=1\ldots\sigma} (1-p_i)\log\frac{1}{1-p_i} \ + \ o(n) \ + \ O(\sigma \log n)\end{aligned}$$

Since $\log \frac{1}{1-x} = \log\left(1 + \frac{x}{1-x}\right) \leq \frac{x}{1-x}\frac{1}{\ln 2}$, the total space requirement for the sparse bit arrays is upper bounded by

$$\begin{aligned}& nH_0 \ + \ n/\ln 2 \sum_{i=1\ldots\sigma} (1-p_i)\frac{p_i}{1-p_i} \ + \ o(n) \ + \ O(\sigma\log n) \\ &= nH_0 + n/\ln 2 + o(n) + O(\sigma \log n) \\ &\leq n(H_0 + 1.44) + o(n)\end{aligned}$$

if we neglect the $O(\sigma \log n)$ additive term. This is rather reasonable, not only because it holds as long as $\sigma = o(n/\log n)$, but also because the $C$ array, present also in the FM-index and the CSA, has the same size.

Although the constant term that multiplies $n$ is larger than that of the FM-index, we have the benefit of having removed the huge additive term $O(n^\gamma \sigma^{\sigma+1})$

result: Ferragina, Manzini, Mäkinen, and Navarro: Compressed Representations of Sequences and Full-Text Indexes. Technical Report 2004-5, Technische Fakultät, Universität Bielefeld, Germany, December 2004.

and the multiplicative term $O(\sigma \log \sigma / \log n)$. This makes our index preferable when the alphabet size is not very small.

The search time stays optimal, $O(m)$. The other time complexities of the FM-index can be obtained with similar mechanisms. These are $O(occ\ \sigma\ \log n)$ for reporting the *occ* occurrence positions of $P$ in $T$ and $O(\sigma\ (L + \log n))$ to extract a text substring of length $L$ (more details in Section 2.3). These complexities retain their dependence on $\sigma$. In the sequel we present a second structure that improves this situation.

## 3.3 Balanced Hierarchy of Bit Arrays

Our second structure builds on a hierarchical scheme to code the bitmaps of the alphabet characters. This structure has been used recently for another index [5], under the name *wavelet tree*. In this paper we use them for a different purpose and improve it in some aspects. As explained, the main advantage of this structure lies in its efficiency to report occurrence positions and to display text substrings. This will be considered in Section 2.3.

### 3.3.1 Basic Structure

We form a perfectly balanced binary tree, where each node corresponds to a subset of the alphabet. The children of the node partition the node subset into two subsets. A bitmap at the node indicates to which children does each text position belong.

The first partition will put characters $c_1 \ldots c_{\lfloor \sigma/2 \rfloor}$ on the left child, and characters $c_{\lfloor \sigma/2 \rfloor + 1} \ldots c_\sigma$ on the right child. A bitmap $B$ at the root will contain $n$ bits, so that $B[i] = 1$ iff $T^{bwt}[i] \in \{c_{\lfloor \sigma/2 \rfloor + 1} \ldots c_\sigma\}$, that is, if the $i$th character of $T^{bwt}$ belongs to the right child.

The two children are processed recursively. However, for each of them, we will only consider the text positions whose character belongs to their subset. That is, the bitmap of the left child of the root will have only $\ell_1 + \ldots + \ell_{\lfloor \sigma/2 \rfloor}$ bits and that of the right child only $\ell_{\lfloor \sigma/2 \rfloor + 1} + \ldots + \ell_\sigma$. Fig. 3.1 gives an example over a 4-letter alphabet.

It is clear that the tree has height $\lceil \log \sigma \rceil$, and that every text position is considered exactly once at each level, so there are overall at most $n \log \sigma$ bits (the last level does not need bit arrays). Each bit array is enriched with the sublinear data structures that permit performing *rank* over them.

Value $Occ(T^{bwt}, c, i)$ can be computed by traversing the subtree until reaching the leaf for $c$. The *rank* values obtained at each level permit us repositioning at the right place in the child. For example, in Fig. 3.1, to compute $Occ(T^{bwt}, \mathtt{C}, 7)$ we first compute $7 - rank(B_{\mathtt{ACGT}}, 7) = 4$, which tells us the number of occurrences of `A`'s and `C`'s up to position 7. Hence the occurrences of `A` and `C` up to position 7 are all packed in positions 1 to 4 in node $B_{\mathtt{AC}}$. To find out how many of those correspond to `C`, we compute $rank(B_{\mathtt{AC}}, 4) = 3$. Overall, $Occ(T^{bwt}, \mathtt{C}, 7) = rank(B_{\mathtt{AC}}, 7 - rank(B_{\mathtt{ACGT}}, 7)) = 3$.

This structure needs $n \log \sigma (1 + o(1))$ bits to represent $Occ$, and can access it

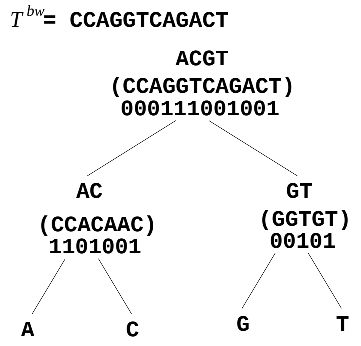


Figure 3.1: The hierarchical structure for a short string over alphabet {A, C, G, T}. Only the bit arrays are represented. The short texts are the character set represented by each node, and the longer texts in parentheses just illustrate the string considered at that node.

in $O(\log \sigma)$ time, so we can search in $O(m \log \sigma)$ time. Note also that the index is not opportunistic, as it needs at least the size of the original text, $n \log \sigma$ bits. In the next section we show how this can be improved on the average.

### 3.3.2 Improving the Average Case

Let us replace the balanced tree of the previous section by a Huffman tree. That is, collect the frequencies $\ell_i$ of the text characters, build their Huffman tree (with left and right branches labeled 0 and 1 according to the Huffman code) and use the tree to partition the alphabet (instead of balanced partitioning it as in the previous section).

To compute $Occ(T^{bwt}, c, i)$ we proceed as with the perfect binary tree. This time, in order to know the child corresponding to character $c$, we examine the next bit of the Huffman code assigned to $c$. If we must go to the left, we descend with value $i \leftarrow i - rank(B, i)$, on the other hand we descend to the right with value $i \leftarrow rank(B, i)$, being $B$ the bitmask of the current node.

We show now that the number of bits used by the structure is $n(H_0 + 1)$. Consider a particular character $c_i$, stored at some leaf of the tree. The $\ell_i$ text positions where character $c_i$ appears are represented as $\ell_i$ bits at every ancestor of $c_i$. Hence, if the Huffman code assigned to $c_i$ has length $l_i$, then the total number of bits allocated for the positions of $c_i$ in the text is $\ell_i \; l_i = n \; p_i \; l_i$. Adding up over all the characters we get $\sum_{i=1\ldots\sigma} n \; p_i \; l_i = n \sum_{i=1\ldots\sigma} p_i \; l_i$, where the last summation is exactly the average length of the Huffman code. Since this average length cannot exceed $H_0+1$, we have overall $n(H_0+1)$ bits of space in our structure. Adding their $rank$ data structures gives $n(H_0 + 1)(1 + o(1))$. This is strictly better than the CSA space requirement.

Search time is not immediately $O(m \log \sigma)$ anymore, however. If we assume that search patterns share the same text distribution, that is, that the frequency of character $c_i$ in the pattern is $mp_i$, then we will search $mp_i$ times the leaf $c_i$, which is at depth $l_i$. Therefore, adding over all the pattern characters we get $\sum_{i=1\ldots\sigma} m \; p_i \; l_i = O(m(H_0 + 1))$ time, which is at most our previous time

$O(m \log \sigma)$.

The above argument, however, holds on the average and under the assumption that patterns and text distribute similarly. The worst case, however, is not that bad. Any character that appears in the text has frequency at least 1 and hence $p \geq 1/n$. Huffman cannot assign it a code larger than $1 + \log(1/p) \leq 1 + \log n$. Therefore, the deepest leaf in the tree has depth $O(\log n)$ and hence our search time is, in the worst case, $O(m \log n)$. This is the worst case search cost of the original CSA.

### 3.3.3 Ensuring $O(m \log \sigma)$ Worst Case Search Time

Still, we can limit our worst case to $O(m \log \sigma)$ and retain our $O(m(H_0 + 1))$ average complexity. Pick a constant $t$, so that the Huffman tree shape is followed up to depth $t \log \sigma$. All the subtrees at that depth are converted to perfectly balanced subtrees. It follows that the depth of the tree is at most $t \log \sigma + \log \sigma = (t + 1) \log \sigma$, and hence the search cost is worst case $O(m \log \sigma)$.

The question is which is the space and the average search time now. Consider first the space, the average search time will be similar. Take a leaf that in the original Huffman tree has depth $l \geq t \log \sigma$. This means that its empirical probability $p$ must satisfy $\log(1/p) + 1 \geq l \geq t \log \sigma$, and therefore $p \leq 2/\sigma^t$. So the character can appear in the text at most $2n/\sigma^t$ times.

Adding these upper bounds on the probabilities over all the possible characters that might be so deep, we have that the overall number of occurrences of such characters in the text cannot exceed $2n\sigma/\sigma^t = 2n/\sigma^{t-1}$. Those characters can be at depth at most $(t + 1) \log \sigma$, and therefore the total number of bits allocated to them cannot exceed $(2n/\sigma^{t-1})(t + 1) \log \sigma$. On the other hand, all the other characters maintain their original depths, and we pessimistically assume that they add up $n(H_0 + 1)$ bits. Hence the point is to find out how large must $t$ be so that $(2/\sigma^{t-1})(t + 1) \log \sigma = O(H_0 + 1)$. Solving for $(2/\sigma^{t-1})(t + 1) \log \sigma \leq x(H_0 + 1)$, one obtains that for any $x \geq 1/(H_0 + 1)$ it is enough that

$$t \geq 1 + \log_\sigma 2 + \log_\sigma \log \sigma + \log_\sigma (t + 1)$$

for which it is sufficient that

$$t \geq 1 + \log_\sigma 2 + \log_\sigma \log \sigma + t/\ln \sigma$$

that is,

$$t \geq (1 + \log_\sigma 2 + \log_\sigma \log \sigma) \frac{\ln \sigma}{\ln \sigma - 1}$$

which tends to 1 as $\sigma$ grows. Using this $t$ value, we have that $t \log \sigma = (\log \sigma + \log \log \sigma + 1) \ln \sigma / (\ln \sigma - 1) = \log \sigma (1 + o(1))$.

Overall, the number of bits needed is $n(H_0 + 1 + x(H_0 + 1)) = n(H_0 + 2)$, and thus the overall space requirement is $n(H_0 + 2)(1 + o(1))$ bits, without large constants hidden in the sublinear parts. Hence the space price of ensuring $O(m \log \sigma)$ worst case has been $n$ extra bits.

The analysis of the modified average search time is similar. We have that there are at most $2m\sigma/\sigma^t$ times where our search cost will be $O(\log \sigma)$, while the others will cost $O(H_0+1)$ and we do not need to consider them further. Exactly as for the analysis of the space, we seek a constant $t$ such that $2m\sigma/\sigma^t \le xm(H_0+1)$ and find the same solution as for guaranteeing $O(n(H_0+1))$ extra space. This is fortunate, as we must use the same $t$ for both purposes.

## 3.4 Reporting Occurrences and Displaying the Text

We can use exactly the same search mechanism for reporting as in the original FM-index described in Section 2.3. In fact, we can implement some internal parts more efficiently by exloiting the hierarchy.

Recall from Section 2.3 the part where time $O(\sigma)$ is used for finding character $c$ for which holds $T^{bwt}[i]=c$, i.e. $Occ(T^{bwt},c,i)=Occ(T^{bwt},c,i-1)+1$.

We can do the same in $O(\log\sigma)$ time thanks to our hierarchical structure. Let us consider $B[i]$, for the bitmap at the root of the tree. If $B[i]=0$, then $T^{bwt}[i] \in \{c_1 \ldots c_{\lfloor \sigma/2 \rfloor}\}$, otherwise it belongs to the upper half. In the first case, we go to the left child, else we go to the right child (using $rank(B,i)$ to reposition). We repeat the process until we find the leaf that corresponds to $T^{bwt}[i]$. If we use a Huffman instead of a balanced tree, then the time is $O(H_0+1)$ if random text positions are sought, and we can still limit the worst case to $O(\log\sigma)$. Same improvement can be applied to showing text substrings.

To summarize, using $\varepsilon n$ additional bits, we can report the occurrence positions in $O(\frac{1}{\varepsilon} occ \log\sigma \log n)$ time and display a text context in time $O(\log\sigma(L+\frac{1}{\varepsilon}\log n))$. On average, assuming that random text positions are involved, the complexities become $O(\frac{1}{\varepsilon} occ H_0 \log n)$ and $O(H_0(L+\frac{1}{\varepsilon}\log n))$. If we use $\varepsilon = 1/\log^{\varepsilon'} n$, we get only $o(n)$ extra space and the worst case complexities become $O(occ \log\sigma \log^{1+\varepsilon'} n)$ and $O(\log\sigma(L+\log^{1+\varepsilon'} n))$, better than the FM-index.

## 3.5 Conclusions

Our motivation in this chapter has been to improve the time/space tradeoffs that appear in the implementation of the FM-index [3] search strategy. We have shown how to reduce the dependence on the alphabet size that appears both in the space usage and time complexities of the FM-index.

We have obtained two structures. A first one retains all the time complexities of the FM-index, while completely removing the dependence on the alphabet size. The price is that its size is proportional to the zero-order entropy of the text rather than the $k$-th order achieved by the FM-index. However, the dependence of the FM-index space is so sharp that the alphabet size can be neglected only for very small alphabets. A second structure has somewhat larger search times but better complexities to show occurrence positions and display text substrings. Compared to the CSA [21] our index takes slightly less space and has better worst case complexities for searching.

We implemented the structure described in Section 3.3. The implementation can be found at `http://www.cs.helsinki.fi/u/vmakinen/software/`. Experiments have been deferred until we have a complete implementation.

# Chapter 4

# Combining Run-length Encoding with Backward Searching

In this chapter, we introduce a structure called the RLFM-index (for Run-Length FM-index) that requires $2n(H_k \log \sigma + 1 + o(1))$ bits of space. It counts occurrences in the same optimal $O(m)$ time achieved by the FM-index. Our space complexity, on the other hand, turns out to be better as soon as $\sigma \log \sigma > \log n$, which means in practice that our index is smaller for all but very small alphabet sizes, even if the text is huge.

## 4.1 Summary of Results

Table 4.1 shows the complexities obtained. We recall that $0 < \gamma < 1$ and $\varepsilon > 0$ are constants that can be made smaller by increasing the sublinear terms hidden in the $O()$ space complexities, $H_k$ is the $k$th order empirical entropy of $T$, *occ* is the number of occurrences to report, and $L$ is the length of the text to display. Value $H \leq \log \sigma$.

## 4.2 The RLFM-Index

The essential property of the BWT transform, which makes it appealing for compression, is that few different characters appear in local contexts of the transformed text. In particular, *runs* of consecutive occurrences of the same character tend to appear.

Our idea is to exploit run-length compression to represent $T^{bwt}$. An array $S$ contains one character per run in $T^{bwt}$, while an array $B$ contains $n$ bits and marks the beginnings of the runs.

**Definition 4.1** *Let string* $T^{bwt} = c_1^{\ell_1} c_2^{\ell_2} \ldots c_{n'}^{\ell_{n'}}$ *consist of* $n'$ *runs, so that the* $i$*-th run consists of* $\ell_i$ *repetitions of character* $c_i$*. Our representation of* $T^{bwt}$ *consists of string* $S = c_1 c_2 \ldots c_{n'}$ *of length* $n'$*, and bit array* $B = 10^{\ell_1 - 1} 10^{\ell_2 - 1} \ldots 10^{\ell_{n'} - 1}$.

| Structure | Space |
|---|---|
| FM-index | $5H_kn + O\left((\sigma \log \sigma + \log\log n)\frac{n}{\log n} + n^{\gamma}\sigma^{\sigma+1}\right)$ |
| CSA | $(H_0/\varepsilon + O(\log\log\sigma))n$ |
| **RLFM1** | $(2H_kH + 1.44 + \varepsilon)(1 + o(1))n$ |
| **RLFM2** | $(2H_kH + 1 + \varepsilon)(1 + o(1))n$ ($+n$ to obtain the worst cases) |

| Structure | Search | Report | Display |
|---|---|---|---|
| FM-index | $O(m)$ | $O\left(occ\ \sigma\ \log^{1+\varepsilon} n\right)$ | $O\left(\sigma\ (L + \log^{1+\varepsilon} n)\right)$ |
| CSA | $O(m \log n)$ | $O\left(occ\ \log^{\varepsilon} n\right)$ | $O\left((L + \log^{\varepsilon} n)\right)$ |
| **RLFM1** | $O(m)$ | $O\left(\frac{1}{\varepsilon} occ\ \sigma\ \log n\right)$ | $O\left(\frac{1}{\varepsilon}\sigma\ (L + \log n)\right)$ |
| **RLFM2**(worst) | $O(m \log \sigma)$ | $O\left(\frac{1}{\varepsilon} occ\ \log\sigma\ \log n\right)$ | $O\left(\frac{1}{\varepsilon} \log\sigma (L + \log n)\right)$ |

Table 4.1: Comparison of space and time complexities of different succinct suffix arrays.

It is clear that $S$ and $B$ contain enough information to reconstruct $T^{bwt}$: $T^{bwt}[i] = S[rank(B,i)]$, where $rank(B,i)$ is the number of 1's in $B[1 \ldots i]$ (so $rank(B,0) = 0$). Function $rank$ can be computed in constant time using $o(n)$ extra bits [9, 16, 2]. Hence, $S$ and $B$ give us a representation of $T^{bwt}$ that permits us accessing any character in constant time and requires at most $n' \log\sigma + n + o(n)$ bits.

The problem, however, is not only how to access $T^{bwt}$, but also how to compute $C_T[c] + Occ(T^{bwt}, c, i)$ for any $c$ and $i$. This not immediate, because we want to add up all the run lengths corresponding to character $c$ up to position $i$.

In the following we show that the above can be computed by means of a bit array $B'$, obtained by reordering the runs of $B$ in lexicographic order of the characters of each run. Runs of the same character are left in their original order. The use of $B'$ will add $n + o(n)$ bits to our scheme. We also use $C_S$, which plays the same role of $C_T$, but it refers to string $S$.

**Definition 4.2** *Let* $S = c_1c_2 \ldots c_{n'}$ *of length* $n'$, *and* $B = 10^{\ell_1 - 1}10^{\ell_2 - 1} \ldots 10^{\ell_{n'} - 1}$. *Let* $p_1p_2 \ldots p_{n'}$ *be a permutation of* $1 \ldots n'$ *such that, for all* $1 \le i < n'$, *either* $c_{p_i} < c_{p_{i+1}}$ *or* $c_{p_i} = c_{p_{i+1}}$ *and* $p_i < p_{i+1}$. *Then, bit array* $B'$ *is defined as* $B' = 10^{\ell_{p_1} - 1}10^{\ell_{p_2} - 1} \ldots 10^{\ell_{p_{n'}} - 1}$.

We now prove our fundamental theorems. They make use of $select$, which is the inverse of $rank$: $select(B', j)$ is the position of the $j$th 1 in $B'$ (and $select(B', 0) = 0$). Function $select$ can be computed in constant time using $o(n)$ extra bits [9, 16, 2]. We start with a lemma and then prove two theorems that cover different cases in the computation of $C_T[c] + Occ(T^{bwt}, c, i)$.

**Lemma 4.3** *Let* $S$ *and* $B'$ *be defined for a string* $T^{bwt}$. *Then, for any* $c \in \Sigma$ *it holds*

$$C_T[c] + 1 \quad = \quad select(B', C_S[c] + 1)$$

*Proof.* $C_S[c]$ is the number of runs in $T^{bwt}$ that represent characters smaller than $c$. Since in $B'$ the runs of $T^{bwt}$ are sorted in lexicographic order, $select(B', C_S[c]+1)$ indicates the position in $B'$ of the first run belonging to character $c$, if any. Therefore, $select(B', C_S[c]+1)-1$ is the sum of the run lengths for all characters smaller than $c$. This is, in turn, the number of occurrences of characters smaller than $c$ in $T^{bwt}$, $C_T[c]$. Hence $select(B', C_S[c]+1)-1 = C_T[c]$. □

**Lemma 4.4** *Let $S$, $B$, and $B'$ be defined for a string $T^{bwt}$. Then, for any $c \in \Sigma$ and $1 \le i \le n$, such that $i$ is the final position of a run in $B$, it holds*

$$C_T[c] + Occ(T^{bwt}, c, i) \quad = \quad select(B', C_S[c] + 1 + Occ(S, c, rank(B, i))) - 1$$

*Proof.* Note that $rank(B, i)$ gives the position in $S$ of the run that finishes at $i$. Therefore, $Occ(S, c, rank(B, i))$ is the number of runs in $T^{bwt}[1 \ldots i]$ that represent repetitions of character $c$. Hence it is clear that $C_S[c] < C_S[c] + 1 + Occ(S, c, rank(B, i)) \le C_S[c+1] + 1$, from which follows that $select(B', C_S[c] + 1 + Occ(S, c, rank(B, i)))$ points to an area in $B'$ belonging to character $c$, or to the character just following that area. Inside this area, the runs are ordered as in $B$ because the reordering in $B'$ is stable. Hence $select(B', C_S[c]+1+Occ(S, c, rank(B, i)))$ is $select(B', C_S[c]+1)$ plus the sum of the run lengths representing character $c$ in $T^{bwt}[1 \ldots i]$. That sum of run lengths is $Occ(T^{bwt}, c, i)$. The argument holds even if $T^{bwt}[i] = c$, because $i$ is the last position of its run and therefore counting the whole run $T^{bwt}[i]$ belongs to is correct. Hence $select(B', C_S[c] + 1 + Occ(S, c, rank(B, i))) = select(B', C_S[c]+1) + Occ(T^{bwt}, c, i)$, and then, by Lemma 4.3, $select(B', C_S[c] + 1 + Occ(S, c, rank(B, i))) - 1 = C_T[c] + Occ(T^{bwt}, c, i)$. □

**Theorem 4.5** *Let $S$, $B$, and $B'$ be defined for a string $T^{bwt}$. Then, for any $c \in \Sigma$ and $1 \le i \le n$, such that $T^{bwt}[i] \ne c$, it holds*

$$C_T[c] + Occ(T^{bwt}, c, i) \quad = \quad select(B', C_S[c] + 1 + Occ(S, c, rank(B, i))) - 1$$

*Proof.* Let $i'$ be the last position of the run that precedes that of $i$. Since $T^{bwt}[i] \ne c$ in all the run of $i$, we have $Occ(T^{bwt}, c, i) = Occ(T^{bwt}, c, i')$ and also $Occ(S, c, rank(B, i)) = Occ(S, c, rank(B, i'))$. Then the theorem follows trivially by applying Lemma 4.4 to $i'$. □

**Theorem 4.6** *Let $S$, $B$, and $B'$ be defined for a string $T^{bwt}$. Then, for any $c \in \Sigma$ and $1 \le i \le n$, such that $T^{bwt}[i] = c$, it holds*

$$\begin{aligned} C_T[c] + Occ(T^{bwt}, c, i) \quad &= \quad select(B', C_S[c] + Occ(S, c, rank(B, i))) \\ &\quad\quad +i - select(B, rank(B, i)). \end{aligned}$$

*Proof.* Let $i'$ be the last position of the run that precedes that of $i$. Then, by Lemma 4.4 we have $C_T[c] + Occ(T^{bwt}, c, i') = select(B', C_S[c] + 1 + Occ(S, c, rank(B, i'))) - 1$. Now, $rank(B, i') = rank(B, i) - 1$, and since $T^{bwt}[i] = c$, it follows that $S[rank(B,i)] = c$. Therefore, $Occ(S,c,rank(B,i')) = Occ(S,c,rank(B,i)-1) = Occ(S,c,rank(B,i))-1$. On the other hand, since $T^{bwt}[i''] = c$ for $i' < i'' \leq i$, we have $Occ(T^{bwt},c,i) = Occ(T^{bwt},c,i')+(i-i')$. Thus, the outcome of Lemma 4.4 can be now rewritten as $C_T[c]+Occ(T^{bwt},c,i)-(i-i') = select(B', C_S[c]+Occ(S,c,rank(B,i)))-1$. The only remaining piece to prove the theorem is that $i-i'-1 = i-select(B,rank(B,i))$, that is, $select(B,rank(B,i)) = i'+1$. But this is clear, since the left term is the position of the first run $i$ belongs to and $i'$ is the last position of the run preceding that of $i$. $\square$

Since functions *rank* and *select* can be computed in constant time, the only obstacle to use the theorem is the computation of *Occ* over string $S$. We have already given solutions for this problem in the previous chapter. For example, we can redo the analysis of Section 3.2 replacing $n$ with $n' = |S|$. We get constant time query time on *Occ* in $n'(H+1.44)(1+o(1))$ bits of space, where $H \leq \log\sigma$ is the zeroth order entropy of $S$.

## 4.3 Space Analysis

The representation of our index needs the bit arrays $S_c$ (see Section 3.2), $B$, and $B'$, plus the sublinear structures to perform *rank* and/or *select* over them. We also store the small array $C_S$. We have shown that all the arrays $S_c$ need together $n'(\log\sigma+1.44)+o(n')+O(\sigma\log n')$ bits, where $n'$ is the number of runs in $T^{bwt}$. Array $C_S$ needs $\sigma\log n'$ bits too. Arrays $B$ and $B'$ can be represented using $n$ bits each.

We need now a bound to $n'$. We have shown in [18] that the number of runs in $T^{bwt}$ is limited by $2H_k n + \sigma^k$ (the proof is also in Appendix A). Although a finer analysis in [13] considers move-to-front coding in addition to run-length compression, the latter alone ensures attaining the $k$th order entropy of the text.

By adding up all our space complexities we obtain $2n(H_k(\log\sigma+1.44+o(1))+1+o(1))+O(\sigma\log n) = 2n(1+H_k\log\sigma)(1+o(1))$ bits of space if $\sigma = o(n/\log n)$.

It is possible to represent $B$ and $B'$ in a more compact way, since only $n'$ bits are set in either array. Another result in [20] shows that they could be represented in $\log\binom{n}{n'}+o(n)$ bits while still supporting constant time *rank* and *select* operations. Redoing the analysis of Section 3.2 for this case we obtain $n'(1+\log(n/n'))$ bits for each. The total space complexity becomes $2H_k n(\log\sigma+3.44+O(\log(1/H_k))+o(1))+O(\sigma\log n)$ bits of space.

Although the constant term that multiplies $H_k n$ is larger than that of the FM-index, we have the benefit of having removed the huge additive term $O(n^\gamma\sigma^{\sigma+1})$ and the multiplicative term $O(\sigma\log\sigma/\log n)$. This makes our index preferable for all but small alphabets, that is, whenever $(\sigma+1)\log\sigma > \log n$. This means, for example, $\sigma \geq 12$ even for 1 terabyte of text.

We observe that we do not have direct access to $S[i]$ anymore. While this is not essential to count the number of pattern occurrences, it is necessary to report occurrence positions and display text contexts. Just as done with the FM-index [3], we can still find $S[i]$ in $O(\sigma)$ time by searching for which $c$ is

$S_c[i] = 1$.

The complexities obtained by this structure are summarized in Table 4.1 (RLFM1).

## 4.4 Conclusions

We have presented an alternative implementation of the FM-index, the RLFM-index, that retains the $O(H_k n)$ FM-index space complexity while removing its exponential dependence on the alphabet size. In exchange we pay a logarithmic factor on the alphabet size in the space requirement. The time complexities stay the same.

When comparing the resulting space complexities, it turns out that our index is smaller whenever $\sigma \log \sigma > \log n$. Even for huge texts, this means that the original FM-index structure is smaller only for very small alphabet sizes. This usually includes DNA sequences, for example, but hardly proteins, music, or natural language texts.

Just as done in previous chapter, it is possible to implement the *Occ* function using the balanced tree structure that retains the same space complexity and changes the search time from $O(m)$ to $O(m \log \sigma)$, with the benefit of obtaining $S[i]$ in $O(\log \sigma)$ time instead of $O(\sigma)$. The complexities obtained by this structure are summarized in Table 4.1 (RLFM2).

We implemented part of this structure using the balanced hierarchy. The implementation can be found at `http://www.cs.helsinki.fi/u/vmakinen/software/`. Experiments have been deferred until we have a complete implementation.

# Chapter 5

# Backward Searching in Compressed Suffix Arrays

The compressed suffix array (CSA) of Sadakane [21] is an alternative to FM-index. It takes $n(H_0 + O(\log\log\sigma))$ bits of space and supports counting queries in time $O(m \log n)$. The search simulates the binary search in suffix array, which results the $\log n$ factor in the search time.

Although the access pattern of a binary search is rather regular, the search string must be compared against some text substring at each step of the binary search. Since the text is not stored, the CSA must be traversed in irregular access patterns to extract each necessary text substring. This makes it unappealing for a secondary memory implementation.

In this chapter we propose new ways to implement the search on CSA using backward searching.

## 5.1 Summary of Results and Related Work

An immediate consequence of backward searching is that a simpler implementation of the CSA is possible, while still retaining the the original search time and at the same time slightly improving the space requirement. Furthermore, the CSA becomes amenable of a secondary memory implementation. If we call $B$ the disk block size, then we can search the CSA in secondary memory using $O(m \log_B n)$ disk accesses. Finally, we show that the structure permits a distributed implementation with optimal speedup and negligible communication effort. Table 5.1 compares the original and new CSAs.

The idea of applying backward searching on CSA is not new. Sadakane [22] has shown how to implement CSA so that it requires $(1/\varepsilon + o(1))nH_0 + O(n)$ bits of space and supports backward searching in $O(m)$ time. This structure assumes that $\sigma = O(\log^{O(1)} n)$.

We use the same backward search routine as in [22]. Our proposal takes slightly less space, but on the other hand, we do not achieve the optimal search time. However, our restriction on $\sigma$ is milder; we only assume that $\sigma = o(n/\log n)$.

A distinguishing feature of our proposal is that it is the only succinct text

| | Original CSA | Our CSA, version 1, or<br>Our CSA, version 2 |
|---|---|---|
| Space (bits) | $n(H_0 + O(\log\log\sigma))$ | $n(\sum_{i=0}^{\ell-1} H_i + \varepsilon)$ or<br>$2n(H_k(\log\sigma + \log\log n) + \varepsilon)$ |
| Search time | $O(m \log n)$ | $O(\lceil m/\ell \rceil \log n)$ or<br>$O(m \log n)$ |
| Disk search time | $O(m \log n)$ | $O(\lceil m/\ell \rceil \log_B n)$ or<br>$O(m \log_B n)$ |
| Remote messages | $m \log n$ | $\lceil m/\ell \rceil$ or<br>$m$ |

Table 5.1: Space and time complexities of the original and new CSA implementations.

index that does not require heavy use of four-Russians tricks. These consist of precomputing all the answers for small arguments and storing the answers in a table, and then expressing computations over larger arguments in terms of those for smaller ones. An example is the constant time *rank* and *select* implementations [9, 16, 2], which appear several times in all the alternative succinct indexes. We use this trick only on vanishing small arguments, and can avoid using the trick at all with a cost of $\log^* n$ factor in search times. The absence of this trick makes our index usable on weaker machine models where individual bits cannot be accessed, and also makes it easier to implement.

## 5.2 Backward Search on CSA

Recall the definition of CSA and the binary search algorithm from Chapter 2. Fig. 5.1 gives an example of the search process.

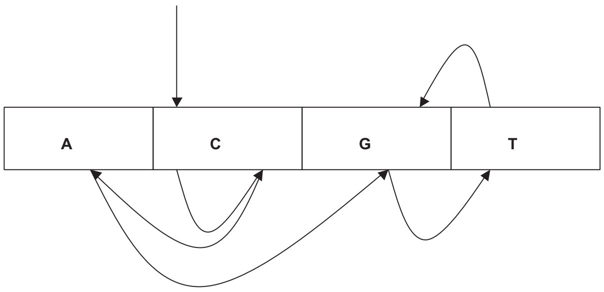


Figure 5.1: One step of binary search for pattern $P = CCAGTA$. The blocks correspond to the areas of suffix array whose suffixes start with the corresponding letter. The straight arrow on top indicates the suffix the pattern is compared against. Other arrows indicate the extraction of the prefix of the compared suffix. The extraction ends at letter $G$, and hence the suffix does not correspond to an occurrence, and the search is continued to the left of the current point.

In the binary search algorithm each string comparison may require accessing up to $m$ arbitrary cells in the $\Psi$ array. Hence using the CSA in secondary memory is not attractive because of the scattered access pattern. Also, a complex part in the implementation of the CSA is the compression of the $\Psi$ array, since it must support constant time direct access at any position.

We show in this section (following [22]) that the CSA can be searched in a completely different way. Let us use the notation $R(X)$, for a string $X$, to denote the range of suffix array positions corresponding to suffixes that start with $X$. The search goal is therefore to determine $R(P)$. We start by computing $R(p_m)$ simply as $R(p_m) = [C[p_m]+1, C[p_m+1]]$. Now, in the general case, given $R(P[i+1 \dots m])$, it turns out that $R(P[i \dots m])$ consists exactly of the suffix array positions in $R(p_i)$ containing values $j$ such that $j+1$ appears in suffix array positions in $R(P[i+1 \dots m])$. That is, the occurrences of $P[i \dots m]$ are the occurrences of $p_i$ followed by occurrences of $P[i+1 \dots m]$. Since $\mathcal{A}[\Psi(i)] = \mathcal{A}[i]+1$, it turns out that

$$x \in R(P[i \dots m]) \quad \Leftrightarrow \quad x \in R(p_i) \ \wedge \ \Psi(x) \in R(P[i+1 \dots m])$$

Now, $\Psi$ can be accessed in constant time, and its values are increasing inside $R(p_i)$. Hence, the set of suffix array positions $x$ such that $\Psi(x)$ is inside some range forms a continuous range of positions and can be binary searched inside $R(p_i)$, at $O(\log n)$ cost. Therefore, by repeating this process $m$ times we find $R(P)$ in $O(m \log n)$ time.

The pseudocode of the algorithm is very simple (min and max stand for binary searches):

---

**Algorithm** $BackwardCSA(P, C, \Psi)$:
  $left_{m+1} := 1$; $right_{m+1} := n$;
  **for** $i := m$ **downto** $1$ **do begin**
    $left_i = \min\{j \in [C[p_i]+1, C[p_i+1]],\ \Psi(j) \in [left_{i+1}, right_{i+1}]\}$;
    $right_i = \max\{j \in [C[p_i]+1, C[p_i+1]],\ \Psi(j) \in [left_{i+1}, right_{i+1}]\}$;
    **if** $left_i > right_i$ **return** "no occurrences found";
  **return** "$right_1 - left_1 + 1$ occurrences found"

---

Fig. 5.2 gives an example of the search algorithm.

Note that this backward search does not directly improve the original CSA search cost. However, it is interesting that the search does not use the text at all, while the original CSA search algorithm is based on the concept of extracting text strings to compare them against $P$. These string extractions make the access pattern to array $\Psi$ irregular and non-local.

In the backward search algorithm, the accesses to $\Psi$ always follow the same pattern: binary search inside $R(c)$, for some character $c$. In the next sections we study different ways to take advantage of this feature. See [22] another way to proceed (from this on, our approach differs from [22]).

## 5.3 Improved Search Complexity

A first improvement due to the regular access pattern is the possibility of reducing the search complexity from $O(m \log n)$ to $O(m \log \sigma + \log n)$. Albeit in [22]

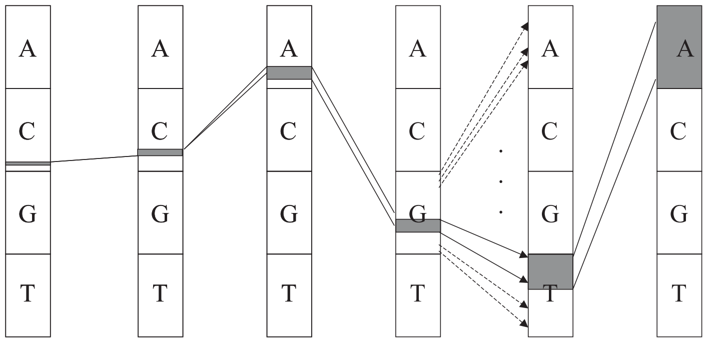


Figure 5.2: Searching for pattern $P = CCAGTA$ backwards (right-to-left). The situation after reading each character is plotted. The gray-shaded regions indicate the interval of the suffix array that contain the current pattern suffix. The computation of the new interval is illustrated in the second step (starting from right). The $\Psi$-values from the block of letter $G$ point to consecutive positions in the suffix array. Hence it is easy to binary search the top-most and bottom-most pointers that are included in the previous interval.

they obtain $O(m)$ search time, the more modest improvement we obtain here does not need any four-Russians technique.

The idea is that we can extend the $C$ array so as to work over strings of length $\ell$ ($\ell$-grams) rather than over single characters. Given $\ell$-gram $x$, $C[x]$ is the number of text $\ell$-grams that are lexicographically smaller than $x$. The final $\ell - 1$ suffixes of length less than $\ell$ are accounted as $\ell$-grams by appending them as many "\$" characters as necessary.

With this $C$ array, we can search for pattern $P$ of length $m$ in $O(\lceil m/\ell \rceil \log n)$ time as follows. We first assume that $m$ is a multiple of $\ell$. Let us write $P = G_1 G_2 \ldots G_{m/\ell}$, where all $G_i$ are all of length $\ell$. We start by directly obtaining $R(G_{m/\ell}) = [C[G_{m/\ell}] + 1, C[next(G_{m/\ell})]]$, where $next(x)$ is the string of length $|x|$ that lexicographically follows $x$ (if no such string exists, then assume $C[next(G_{m/\ell})] = n$). Once this is known, we binary search in $R(G_{m/\ell-1})$ the subinterval that points inside $R(G_{m/\ell})$. This becomes $R(G_{m/\ell-1}G_{m/\ell})$. The search continues until we obtain $R(P)$. The number of steps performed is $m/\ell$, each being a binary search of cost $O(\log n)$.

Let us consider now the case where $\ell$ does not divide $m$. We extend $P$ so that its length is a multiple of $\ell$. Let $e = m - (m \bmod \ell)$. Then we build two patterns out of $P$. The first is $P_l$, used to obtain the left limit of $R(P)$. $P_l$ is the lexicographically smallest $\ell$-gram that is not smaller than $P$, $P_l = P\$^e$, that is, $P$ followed by $e$ occurrences of character "\$". The second is $P_r$, used to obtain the right limit of $R(P)$. $P_r$ is the smallest $\ell$-gram that is lexicographically larger than any string prefixed by $P$, $P_r = next(P)\$^e$. Hence, we search for $P_l$ and $P_r$ to obtain $R(P_l) = [sp_l, ep_l]$ and $R(P_r) = [sp_r, ep_r]$. Then, $R(P) = [sp_l, sp_r - 1]$.

Note that $next(x)$ is not defined if $x$ is the largest string of its length. In this case we do not need to search for $P_r$, as we use $sp_r = n + 1$.

We have obtained $O(\lceil m/\ell \rceil \log n)$ search time, at the cost of a $C$ table with

$\sigma^\ell$ entries. If we require that $C$ can be stored in $n$ bits, then $\sigma^\ell \log n = n$, that is, $\ell = \log_\sigma n - \log_\sigma \log n$. The search complexity becomes $O(\lceil m/\log_\sigma n \rceil \log n) = O(m \log \sigma + \log n)$ as promised.

Moreover, we can reduce the $C$ space complexity to $O(n/\log^t n)$ for any constant $t$. The condition $\sigma^\ell \log n = n/\log^t n$ translates to $\ell = \log_\sigma n - (t + 1)\log_\sigma \log n$, and the search cost remains $O(m \log \sigma + \log n)$.

Notice that we cannot use the original $\Psi$ function anymore to find the subintervals, since we read $\ell$ characters at a time. Instead, we need to store values $\Psi^\ell[i] = \Psi[\Psi[\cdots \Psi[i]] \cdots]$, where $\Psi$ function is recursively applied $\ell$ times. Next section considers how to represent the $\Psi^\ell$ values succinctly.

## 5.4 A Simpler and Smaller CSA Structure

One of the difficulties in implementing the CSA is to provide constant time access to array $\Psi$ (and to $\Psi^\ell$). This is obtained by storing absolute samples every $O(\log n)$ entries and differential encoding for the others, and hence several complex four-Russians-based structures are needed to access between samples in constant time.

#### Binary Search on Absolute Samples.

Our binary searches inside $R(p_i)$, instead, could proceed over the absolute samples first. When the right interval between samples has been found, we decode the $O(\log n)$ entries sequentially until finding the appropriate entry. The complexity is still $O(\log n)$ per binary search (that is, $O(\log n)$ accesses for the initial binary search plus $O(\log n)$ for the final sequential search), and no extra structure is needed to access $\Psi$ (or $\Psi^\ell$). The search is illustrated in Fig. 5.3.

Let us consider the space usage now. As explained, we store $O(n/\log n) = \frac{\varepsilon n}{2 \log n}$ absolute samples of $\Psi$. For each such sample, we need to store value $\Psi$ in $\log n$ bits, as well as a pointer to its corresponding position in the differentially encoded $\Psi$ sequence, in other $\log n$ bits. Overall, the absolute samples require $\varepsilon n$ bits, for any $\varepsilon > 0$, and permit doing each binary search in $\log n + \frac{2}{\varepsilon} \log n = O(\log n)$ steps. On the other hand, array $C$ needs $o(n)$ bits by choosing any $t > 1$ for its $O(n/\log^t n)$ entries.

The most important issue is how to efficiently store the differences between consecutive $\Psi$ cells. The $n(H_0 + O(\log \log \sigma))$ space complexity of the original CSA is due to the need of constant time access inside absolute samples, which forces the use of a zero-order compressor. In our case, we could use any compression mechanism between absolute samples, because they will be decoded sequentially.

#### Compressing $\Psi$ using Elias-encoding.

We now give a simple encoding that slightly improves the space complexity of the original CSA.

The differences $\Psi(i) - \Psi(i-1)$ can be encoded efficiently using variable-length Elias coding [28]. Let $b(p)$ be the binary string representation of a number $p$.

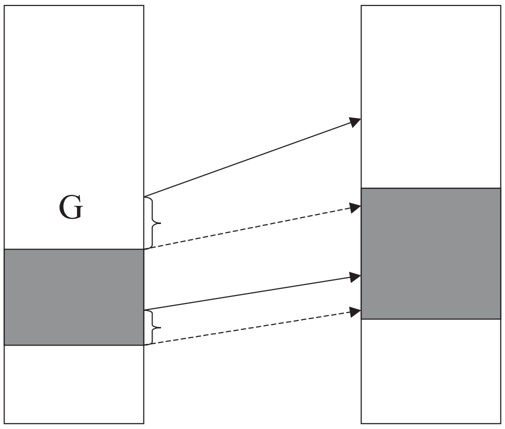


Figure 5.3: Binary search followed by sequential search. The top-most sampled value closest to the previous interval is found using binary search (indicated by the top-most solid arrow). Then the next $\Psi$ values are decoded until the first value inside the interval (if exists) is encountered (indicated by the top-most dashed arrow). The same is repeated to find the bottom-most sampled value and then the bottom-most encoded value.

We use $1^{|b(r)|}0b(r)b(p)$ to encode $p$, where $r = |b(p)|$. The length of the encoding is $\log(2\log p + 1) + 1 + \log p = (1 + o(1))\log p$. The overall length of the codes for all differences can now be bounded using the following observation: The $\Psi$-values are increasing inside a suffix array region corresponding to a character $c$. In other words,

$$\sum_{i,i-1\in R(c)} |\Psi(i) - \Psi(i-1)| = \sum_{i,i-1\in R(c)} \Psi(i) - \Psi(i-1) \leq n. \tag{5.1}$$

To encode the differences for character $c$, we thus need $\sum_{i,i-1\in R(c)}(1 + o(1))\log(\Psi(i) - \Psi(i-1))$ bits. This becomes $|R(c)|(1 + o(1))\log(n/|R(c)|)$ in the worst case, where $|R(c)| = r - \ell + 1$ is the length of the range $R(c) = [\ell, r]$.

Summing over all characters gives

$$\sum_{c\in\Sigma} |R(c)|(1 + o(1))\log(n/|R(c)|) = nH_0(1 + o(1)). \tag{5.2}$$

Hence the overall structure occupies $n(H_0 + \varepsilon)(1 + o(1))$ bits.

Also, the "small additional structures" mentioned in Section 2.2.2, used to find in constant time the character $c$ such that $C[c] < i \leq C[c + 1]$, are not anymore necessary. These also made use of four-Russians techniques.

Let us consider how to decode a number $p$ coded with Elias. We can read $1^{|b(r)|}0$ bitwise in $O(|b(r)|) = O(\log r) = O(\log |b(p)|) = O(\log\log p) = O(\log\log n)$ time. Then we get $b(r)$ and $b(p)$ in constant time. The complexity can be lowered to $O(\log\log\log n)$ if we code $r$ as $1^{b(r')}0b(r')b(r)$, where $r' = |b(r)|$. Indeed, we can apply this until the number to code is zero, for a complexity of $O(\log^* n)$. Alternatively, we can decode Elias in constant time by only small abuse of four-Russians technique: Precompute the first zero of any sequence of length $\log\log n$ and search the table with the first bits of $1^{|b(r)|}0b(r)b(p)$. This table needs only $O(\log n \log\log n)$ space.

**Compressing $\Psi^\ell$ using Elias encoding.**

When using $\Psi^\ell$ instead of $\Psi$, the analysis above applies provided we consider the suffix array areas that start with the same prefix of length $\ell$. Therefore, the overall length of the $\Psi^\ell$ array with Elias encoding is

$$\sum_{x\in\Sigma^\ell} |R(x)|(1+o(1))\log(n/|R(x)|). \tag{5.3}$$

Decomposing $x = x_1 \ldots x_\ell$, we note that, for any $i$,

$$\begin{aligned}|R(x_1\ldots x_i)| &= (|R(x_1)|)\cdot(|R(x_1x_2)|/|R(x_1)|)\cdot(|R(x_1x_2x_3)|/|R(x_1x_2)|)\\ &\quad\cdots\ (|R(x_1x_2\ldots x_i)|/|R(x_1x_2\ldots x_{i-1}|)\\ &= n\cdot\prod_{j=1}^{i} r_j\end{aligned}$$

where $r_j = (|R(x_1x_2\ldots x_j)|/|R(x_1x_2\ldots x_{j-1}|)$ and $|R(\varepsilon)| = n$. Hence Eq. (5.3) can be rewritten as

$$n(1+o(1))\sum_{x_1\ldots x_\ell\in\Sigma}\prod_{j=1}^{\ell} r_j \sum_{i=1}^{\ell}\log(1/r_i)$$

where we can distribute summations and products to get

$$n(1+o(1))\sum_{i=1}^{\ell}\sum_{x_1\ldots x_i\in\Sigma^*}\prod_{j=1}^{i} r_j\log(1/r_i)\sum_{x_{i+1}\ldots x_\ell\in\Sigma^*}\prod_{j=i+1}^{\ell} r_j\ ,$$

by transforming back products of $r_j$'s into $R$'s we get

$$n(1+o(1))\sum_{i=1}^{\ell}\sum_{x_1\ldots x_i\in\Sigma^*}(|R(x_1\ldots x_i)|/n)\log(1/r_i)\sum_{x_{i+1}\ldots x_\ell\in\Sigma^*}\frac{|R(x_1\ldots x_\ell)|}{|R(x_1\ldots x_i)|}\ ,$$

and by factoring out the dividing $R$, we have

$$n(1+o(1))\sum_{i=1}^{\ell}\sum_{x_1\ldots x_i\in\Sigma^*}(|R(x_1\ldots x_i)|/n)\log(1/r_i)\frac{\sum_{x_{i+1}\ldots x_\ell\in\Sigma^*}|R(x_1\ldots x_\ell)|}{|R(x_1\ldots x_i)|}\ .$$

Note that the inner summation is equal to $|R(x_1\ldots x_i)|$, except for the few occurrences of $x_1\ldots x_i$ starting at text positions larger than $n-\ell+i$, as these are not followed by any $x_{i+1}\ldots x_\ell$. In any case, the summation cannot be larger than $|R(x_1\ldots x_i)|$, so a tight upper bound to the space usage is

$$n(1+o(1))\sum_{i=1}^{\ell}\sum_{x_1\ldots x_i\in\Sigma^*}(|R(x_1\ldots x_i)|/n)\log(1/r_i)$$

$$= n(1+o(1)) \sum_{i=1}^{\ell} \sum_{x_1 \ldots x_i \in \Sigma^*} (|R(x_1 \ldots x_i)|/n) \log(|R(x_1 \ldots x_i)|/|R(x_1 \ldots x_{i-1})|)$$

$$= n(1+o(1)) \sum_{i=0}^{\ell-1} H_i$$

This shows that, the larger $\ell$ we use, the higher space usage we pay. In fact, we should use $\ell = \log n$ to reach $O(m \log \sigma + \log n)$ search time, in which case the space usage is equivalent to that of a plain suffix array. For smaller $\ell$, however, the tradeoff of $O(\lceil m/\ell \rceil \log n)$ time and $n \sum_{0 \le i < \ell} H_i$ bits of space is useful in practice.

**Alternative encoding for $\Psi$.**

Let us now consider how the Elias encoding could be improved for $\Psi$ values (the following analysis does not apply for $\Psi^\ell$ values). We make use of the fact that many differences $\Psi(i) - \Psi(i-1)$ are small numbers. In particular, we use different encoding for differences that equal 1. That is, we encode the *runs* of values $\Psi(i) - \Psi(i-1) = 1$ using run-length encoding. As explained in Section 2.2.2, if there is a self-repetition $\mathcal{A}[j \ldots j+\ell] = \mathcal{A}[i \ldots i+\ell] + 1$, then $\Psi(i) - \Psi(i-1) = 1$ in all that area. In [18] we have proved that the number of such self-repetitions is upper bounded by $2H_k n + \sigma^k$ (the proof is also in Appendix A). Since we add $O(n/\log n)$ absolute samples, we can cut those self repetitions and create only $O(n/\log n)$ additional areas. Other $O(n/\log^t n)$ areas may be added to ensure that each entry of array $C$ points to an absolute sample.

Since we restrict the lengths of runs to be at most $O(\log n)$, we can encode all run lengths in $2H_k n \log\log n(1+o(1))$ bits. Alternatively, we could encode the runs as in [18], using a bit-vector of length $n$, but this would make use of four-Russians techniques.

In addition to the run lengths, we still need to encode the differences $\Psi(i) - \Psi(i-1)$ that are greater than one. These can be encoded using the same Elias coding as above. By redoing the analysis for this case (Eqs. (5.1) and (5.2)), one finds out that the overall length of the coding is bounded by $2nH_k H(1+o(1))$, where $H$ is the zero-order entropy of the multiset of characters corresponding to the start of the runs. This entropy is usually larger than $H_0$ of the original string, and always less than $\log \sigma$.

Hence the overall size of the structure using run-length encoding is $2n(H_k(H + \log\log n) + \varepsilon)(1+o(1))$. The alternative implementation using the same idea but encoding the runs in overall $n + o(n)$ bits (as mentioned above, based on four-Russians [18]) gives the bound $n(2H_k H + 1 + \varepsilon)(1+o(1))$.

Note that an implicit assumption in all the above discussion is that $\sigma = O(n/\log n)$, as otherwise it is unfeasible to store only $O(n/\log n)$ absolute samples. This assumption is rather mild. Moreover, if this was not true, then even the $C$ array stored by the original CSA and the FM-index would require $\omega(n)$ bits, making them useless.

## 5.5 Reporting Occurrences

In the previous sections, we have considered the implementation of counting queries. Reporting the occurrences can be done exactly as in the original CSA structure, as described in Section 2.3. The only exception is that we do not have direct access to $\Psi$ anymore. We need to spend $\log n$ time to compute each $\Psi$ value. Thus, using $\varepsilon n$ additional bits, we can report the occurrence positions in $O(\frac{1}{\varepsilon} occ \log^2 n)$ time and display a text context of length $L$ in time $O(\log n(L + \frac{1}{\varepsilon}\log n))$.

## 5.6 A Secondary Memory Implementation

We show now how the regular access pattern can be exploited to obtain an efficient implementation in secondary memory, where the disk blocks can accommodate $B \log n$ bits.

Let us consider again the absolute samples. We pack all the $O(n/\log n)$ absolute samples together, using $O(n/(B \log n))$ blocks. However, the samples are stored in a level-by-level order of the induced binary search tree: For each character $c$, we store the root of the binary search hierarchy corresponding to the area $C[c]+1 \ldots C[c+1]$, that is, $\Psi(\lfloor((C[c]+1)+C[c+1])/2\rfloor)$. Then we store the roots of the left and right children, and so on. When the disk block is filled, the subtrees are recursively packed into disk pages. Fig. 5.4 illustrates.

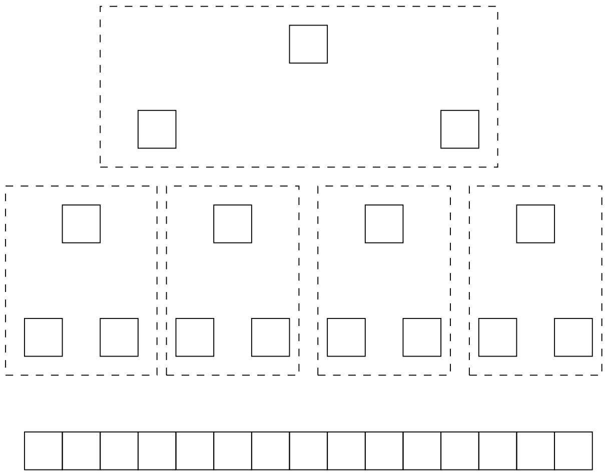

Figure 5.4: Packing of array cells to optimize binary search in secondary memory. The dashed boxes indicate cells mapped to a disk block. Any binary search on the array at the bottom is carried out with 2 disk accesses.

Using this arrangement, any binary search inside the area of a character $c$ can make the first $\log B$ accesses by reading only the first disk block. Each new disk block read permits making other $\log B$ accesses. Overall, we find the interval between two consecutive samples in $O(\log(n)/\log(B)) = O(\log_B n)$ disk accesses.

The compressed entries of $\Psi$ are stored in contiguous disk pages. Once we determine the interval between consecutive samples, we sequentially read all the necessary disk pages. This requires reading $O(\log(n)/B)$ additional disk pages, which contributes a lower order term to the cost.

Overall, we can maintain the CSA on disk and search it in $O(m \log_B n)$ disk accesses. The original structure requires $O(m \log n)$ disk accesses. If we can hold $O(n)$ bits in main memory, then we can cache all the absolute samples and pay only $O(m \log(n)/B)$ disk accesses.

This scheme can be extended to use a table $C$ of $\ell$-grams rather than of individual characters. Each individual binary search takes still $O(\log_B n)$ time, but we perform only $\lceil m/\log_\sigma n \rceil$ of them. One extra disk access per binary search may be necessary to fetch the appropriate $C$ cell, since table $C$ might have to be stored on disk now. Overall, we obtain $O(m \log_B \sigma + \log_B n)$ disk accesses.

One obstacle to a secondary memory CSA implementation might be in building such a large CSA. This issue has been addressed satisfactorily [23].

## 5.7 A Distributed Implementation

Distributed implementations of suffix arrays face the problem that not only the suffix array, but also the text, are distributed. Hence, even if we distribute suffix array $\mathcal{A}$ according to lexicographical intervals, the processor performing the local binary search will require access to remote text positions [14]. Although some heuristics have been proposed, $\log n$ remote requests for $m$ characters each are necessary in the worst case.

The original CSA does not help solve this. If array $\Psi$ is distributed, we will need to request cells of $\Psi$ to other processors for each character of each binary search step, for a total of $m \log n$ remote requests. Actually this is worse than $\log n$ requests for $m$ characters each.

The backward search mechanism permits us to do better. Say that one processor is responsible for the interval corresponding to each character. Then, we can process pattern characters $p_m$, $p_{m-1}$, $\ldots$, $p_1$ as follows: The processor responsible for $p_m$ sends $R(p_m)$ to the processor responsible for $p_{m-1}$. That processor binary searches in its local memory for the cells that point inside $R(p_m)$, without any communication need. Upon completing the search, it sends $R(p_{m-1}p_m)$ to the processor responsible for $p_{m-2}$ and so on. After $m$ communication steps exchanging $O(1)$ data, we have the answer.

In the BSP model [26], we need $m$ supersteps of $O(\log n)$ CPU work and $O(1)$ communication each. In comparison, the CSA needs $O(m \log n)$ supersteps of $O(1)$ CPU and communication each, and the basic suffix array needs $O(\log n)$ supersteps of $O(m)$ CPU and communication each.

More or less processors can be accommodated by coarsening or refining the lexicographical intervals. Although the real number of processors, $r$, is irrelevant to search for one pattern (note that the elapsed time is still $O(m \log n)$), it becomes important when performing a sequence of searches.

If $N$ search patterns, each of length $m$, are entered into the system, and we assume that the pattern characters distribute uniformly over $\Sigma$, then a pipelining effect occurs. That is, the processor responsible for $p_m$ becomes idle after the first superstep and it can work on subsequent patterns. On average the $N$ answers are obtained after $Nm/r$ supersteps and $O(Nm \log(n)/r)$ total CPU work, with $O(Nm/r)$ communication work.

Hence, we have obtained $O(m \log(n)/r)$ amortized time per pattern with $r$ processors, which is the optimal speedup over the sequential algorithm. The communication effort is $O(m/r)$, of lower order than the computation effort. We can apply again the technique of Section 5.3 to reduce the CPU time to $O((m \log \sigma + \log n)/r)$.

## 5.8 Conclusions

We have proposed a new implementation of backward search algorithm for the Compressed Suffix Array (CSA). The new method has the advantage of displaying a rather regular access pattern to the structure. This allows several improvements over the CSA: (*i*) simpler and more compact structure implementation, (*ii*) efficient secondary memory implementation, and (*iii*) efficient distributed implementation. In particular, ours is the only structure not relying on four-Russians techniques.

## Acknowledgements

This chapter was written joint with Kunihiko Sadakane from Kyushu University, Japan. The main part of the results of this chapter appear as a conference article in ISAAC 2004.

# Chapter 6

# Discussion

We have presented many alternative implementations of succinct suffix arrays supporting backward searching. Some of these implementations improve existing bounds. Not all bounds we achieve are comparable as such, since some entropy bounds we use are pessimistic. In particular, this applies to the the analysis in Appendix A that relates the number of runs, $n'$, in the Burrows-Wheeler transformed text (of length $n$) with the $k$th order entropy. For this reason, all bounds of the form $2H_kn$ should be written as $\min(2H_kn, n)$, since obviously $n' \leq n$; we omitted this for clarity.

We have implemented parts of the structures introduced in this paper. Some preliminary experiments show that the structure of Chapter 4 is very competitive in search times compared to other existing implementations of succinct suffix arrays. This structure took 74.5MB on a natural language text of size 83MB. That is, 0.89 times the text size. The value $n'$ was $n/3.01$ in this experiment.

It is reasonably straightforward to implement the proposed structures. For example, the main part of the solution in Chapter 4 takes about 700 code lines. This includes the implementations of rank and select queries. For rank queries we adopted the solution in [2]. For select queries we were not aware of any practical solution, and hence we used a tailored solution that uses optimized binary search over the rank-structure.

Another pragmatic issue is the construction time and space of the indices. All the proposed structures can be constructed in linear time on fixed alphabets. In practice, the construction took 2 minutes in the experiment above. Finally, our implementation still uses $O(n \log n)$ bits of memory in the construction time to construct the suffix array. This could be avoided using a space-efficient construction algorithm [11, 8].

We mainly concentrated on counting queries in this paper; reporting queries use the same mechanism as previous papers. The future challenge is to improve reporting capabilities of all the suggested compressed indexes.

# Bibliography


[1] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. *DEC SRC Research Report 124*, 1994.

[2] D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.

[3] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS'00*, pp. 390–398, 2000.

[4] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. SODA'01*, pp. 269–278, 2001.

[5] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA'03*, pp. 841–850, 2003.

[6] R. Grossi, A. Gupta, and J. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proc. SODA'04*, 2004.

[7] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. STOC'00*, pp. 397–406, 2000.

[8] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. In *Proc. FOCS 2003*, pp. 251–260, 2003.

[9] G. Jacobson. *Succinct Static Data Structures*. PhD thesis, CMU-CS-89-112, Carnegie Mellon University, 1989.

[10] J. Kärkkäinen. *Repetition-Based Text Indexes*, PhD Thesis, Report A-1999-4, Department of Computer Science, University of Helsinki, Finland, 1999.

[11] Tak Wah Lam, Kunihiko Sadakane, Wing-Kin Sung, and Siu-Ming Yiu. A Space and Time Efficient Algorithm for Constructing Compressed Suffix Arrays. In *Proc. COCOON 2002*, pp. 401–410, 2002.

[12] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22, pp. 935–948, 1993.

[13] G. Manzini. An Analysis of the Burrows-Wheeler Transform. *J. of the ACM* 48(3):407–430, 2001.

[14] M. Marín and G. Navarro. Suffix Arrays in Parallel. In *Proc. 9th International Conference on Parallel and Distributed Computing (EuroPar 2003)*, LNCS 2790, pp. 338–341, 2003.

[15] E. M. McCreight. A space economical suffix tree construction algorithm. *J. of the ACM*, 23, pp. 262–272, 1976.

[16] I. Munro. Tables. In *Proc. FSTTCS'96*, pp. 37–42, 1996.

[17] V. Mäkinen. Compact Suffix Array — A space-efficient full-text index. *Fundamenta Informaticae* 56(1-2), pp. 191–210, 2003.

[18] V. Mäkinen and G. Navarro. Compressed compact suffix arrays. To appear in *Proc. CPM'04*, LNCS, 2004.

[19] G. Navarro. Indexing text using the Ziv-Lempel trie. *J. of Discrete Algorithms* 2(1):87–114, 2004.

[20] R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding $k$-ary trees and multisets. In *Proc. SODA'02*, pp. 233–242, 2002.

[21] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. ISAAC'00*, LNCS 1969, pp. 410–421, 2000.

[22] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proc. SODA 2002*, ACM-SIAM, pp. 225–232, 2002.

[23] Wing-Kai Hon, Tak Wah Lam, Kunihiko Sadakane, Wing-Kin Sung. Constructing compressed suffix arrays with large alphabets. In *Proc. ISAAC'03*, LNCS 2906, pp. 240–249, 2003.

[24] S. Srinivasa Rao. Time-space trade-offs for compressed suffix arrays. *Inf. Proc. Lett.*, 82 (6), pp. 307-311, 2002.

[25] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14, pp. 249–260, 1995.

[26] L. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33:103–111, Aug. 1990.

[27] P. Weiner. Linear pattern matching algorithms. In *Proc. IEEE 14th Ann. Symp. on Switching and Automata Theory*, pp. 1–11, 1973.

[28] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes.* Morgan Kaufmann Publishers, New York, second edition, 1999.

# Appendix A

# An Entropy Bound on the Length of Run-Length Encoded Burrows-Wheeler-Transformed Text

We will now prove that the length $n'$ of run-length encoded Burrows-Wheeler transformed text is at most $\sigma^k + 2H_k n$, where $H_k$ is the *k-th order empirical entropy* of $T$ [13].

Let us first recall some basic facts and definitions from [13]. Let $n_i$ denote the number of occurrences in $T$ of the $i$-th symbol of $\Sigma$. The zero-order empirical entropy of the string $T$ is

$$H_0(T) = -\sum_{i=1}^{\sigma} \frac{n_i}{n} \log \frac{n_i}{n}, \tag{A.1}$$

where $0 \log 0 = 0$. If we use a fixed codeword for each symbol in the alphabet, then $H_0 n$ bits is the smallest encoding one can achieve for $T$ ($H_0 = H_0(T)$). If the codeword is not fixed, but it depends on the $k$ previous symbols that may precede it in $T$, then $H_k n$ bits is the smallest encoding one can achieve for $T$, where $H_k = H_k(T)$ is the $k$-th order empirical entropy of $T$. It is defined as

$$H_k(T) = \frac{1}{n} \sum_{W \in \Sigma^k} |W_T| H_0(W_T), \tag{A.2}$$

where $W_T$ is a concatenation of all symbols $t_j$ (in arbitrary order) such that $Wt_j$ is a substring of $T$. String $W$ is the *k-context* of each such $t_j$. Note that the order in which the symbols $t_j$ are permuted in $W_T$ does not affect $H_0(W_T)$, and hence we have not fixed any particular order for $W_T$.

Recall Burrows-Wheeler transform $bwt(T)$ from Chapter 2. We will now prove that the number of runs in $bwt(T)$ is at most $\sigma^k + 2H_k n$.

Let $rle(S)$ be the *run-length encoding* of string $S$, that is, a sequence of pairs $(s_i, \ell_i)$ such that $s_i s_{i+1} \cdots s_{i+\ell-1}$ is a *maximal run* of symbol $s_i$ (i.e., $s_{i-1} \neq s_i$ and $s_{i+\ell} \neq s_i$), and all such maximal runs are listed in $rle(S)$ in the order they appear in $S$. The length $|rle(S)|$ of $rle(S)$ is the number of pairs in it. Notice that $|rle(S)| \leq |rle(S_1)| + |rle(S_2)| + \cdots + |rle(S_p)|$, where $S_1 S_2 \cdots S_p = S$ is any partition of $S$.

Recall string $W_T$ as defined in Eq. (A.2) for a $k$-context $W$ of a string $T$. Note that we can apply any permutation to $W_T$ so that (A.2) still holds. Now, $bwt(T)$ can be given as a concatenation of strings $W_T$ for $W \in \Sigma^k$, if we fix the permutation of each $W_T$ and the relative order of all strings $W_T$ appropriately [13]. As a consequence, we have that

$$|rle(bwt(T))| \leq \sum_{W \in \Sigma^k} |rle(W_T)|, \tag{A.3}$$

where the permutation of each $W_T$ is now fixed by $bwt(T)$. In fact, Eq. (A.3) holds also if we fix the permutation of each $W_T$ so that $|rle(W_T)|$ is maximized. This observation gives a tool to upper bound $|rle(bwt(T))|$ by the sum of code lengths when zero-order entropy encoding is applied to each $W_T$ separately. We next show that $|rle(W_T)| \leq 1 + 2|W_T|H_0(W_T)$.

First notice that if $|\Sigma_{W_T}| = 1$ then $|rle(W_T)| = 1$ and $|W_T|H_0(W_T) = 0$, so our claim holds. Let us then assume that $|\Sigma_{W_T}| = 2$. Let $x$ and $y$ ($x \leq y$) be the number of occurrences of the two letters, say $a$ and $b$, in $W_T$, respectively. We have that

$$H_0(W_T) = -(x/(x+y))\log(x/(x+y)) - (y/(x+y))\log(y/(x+y)) \geq x/(x+y), \tag{A.4}$$

since $-\log(x/(x+y)) \geq 1$ (because $x/(x+y) \leq 1/2$) and $-(y/(x+y))\log(y/(x+y)) > 0$. The permutation of $W_T$ that maximizes $|rle(W_T)|$ is such that there is no run of symbol $a$ longer than 1. This makes the number of runs in $rle(W_T)$ to be $2x+1$. By using Eq. (A.4) we have that

$$|rle(W_T)| \leq 2x + 1 = 1 + 2|W_T|x/(x+y) \leq 1 + 2|W_T|H_0(W_T). \tag{A.5}$$

We are left with the case $|\Sigma_{W_T}| > 2$. This case splits into two sub-cases: (i) the most frequent symbol occurs at least $|W_T|/2$ times in $W_T$; (ii) all symbols occur less than $|W_T|/2$ times in $W_T$. Case (i) becomes analogous to case $|\Sigma_{W_T}| = 2$ once $x$ is redefined as the sum of occurrences of symbols other than the most frequent. In case (ii) $|rle(W_T)|$ can be $|W_T|$. On the other hand, $|W_T|H_0(W_T)$ must also be at least $|W_T|$, since it holds that $-\log(x/|W_T|) \geq 1$ for $x \leq |W_T|/2$, where $x$ is the number of occurrences of any symbol in $W_T$. Therefore we can conclude that Eq. (A.5) holds for any $W_T$.

Combining Eqs. (A.2) and (A.5) we get the following result:

**Theorem A.1** *The length of the run-length encoded Burrows-Wheeler transformed text of length $n$ is at most $\sigma^k + 2H_k n$, for any fixed $k \geq 1$.*