A Comparison of BWT Approaches to Compressed-Domain Pattern Matching

Andrew Firth

Honours Project Report, 2002
Supervisor: Tim Bell

Abstract

A number of algorithms have recently been developed to search files compressed with the Burrows-Wheeler Transform (BWT) without the need for full decompression first. This allows the storage requirement of data to be reduced through the exceptionally good compression offered by BWT, while still allowing fast access to the information for searching. We provide a detailed description of five of these algorithms: Compressed-Domain Boyer-Moore (Bell et al. 2002), Binary Search (Bell et al. 2002), Suffix Arrays (Sadakane & Imai 1999), *q*-grams (Adjeroh et al. 2002) and the FM-index (Ferragina & Manzini 2001), and also present results from a set of extensive experiments that were performed to evaluate and compare the algorithms. Furthermore, we introduce a technique to improve the search times of Binary Search, Suffix Arrays and *q*-grams by around 20%, as well as reduce the memory requirement of the latter two by 40% and 31%, respectively.

Our results indicate that, while the compressed files of the FM-index are larger than those of the other approaches, it is able to perform searches with considerably less memory. Additionally, when only counting the occurrences of a pattern, or when locating the positions of a small number of matches, it is the fastest algorithm. For larger searches, *q*-grams provides the fastest results.

Contents

Chapter 1

Introduction

The amount of electronic data available is rapidly increasing, partly due to the phenomenal growth of the Internet, but also due to increases in other data sources such as digital libraries. This volume of data is quickly surpassing storage capabilities, so finding efficient methods to store, organise and manage data is becoming an important field of research.

Employing compression algorithms to reduce the amount of space that data occupies is one approach for making it more manageable. The process of compression removes redundancies in a file by using smaller bit patterns to represent the symbols most likely to be seen, so the file takes less space to store. Unfortunately, it also removes much of the structure of the data, so that it can be harder to search and retrieve information. The simple solution is a *decompress-then-search* approach that involves decompressing the data before a search is performed with a traditional pattern matching algorithm. The decompression process, however, can be very time consuming. A better solution would allow a direct search of the compressed data with minimal or no decompression. Searching without any decompression is called *fully-compressed pattern matching*, or sometimes just *compressed pattern matching*. This process involves compressing the pattern and matching it to the compressed text representation, but is often impossible, particularly with compression algorithms that use different representations for a substring depending on the substring's context. This is the case with adaptive compression algorithms, as well as some other coders, including arithmetic coders. An alternative technique is *compressed-domain pattern matching*, which allows partial decompression of the text to remove some of the obstacles of a fully-compressed algorithm, while still providing the advantages of avoiding complete decompression.

A survey by Bell et al. (2001) showed that the majority of research in the area of fully-compressed and compressed-domain pattern matching is based on the LZ (Lempel-Ziv) family of compression algorithms (Amir et al. 1996, Farach & Thorup 1998, Navarro & Raffinot 1999), Huffman code (Ziviani et al. 2000, Moura et al. 2000), and run-length encoding (Bunke & Csirik 1993, Bunke & Csirik 1995). Other researchers have devised methods to search text that has been compressed using antidictionaries (Shibata et al. 1999) or byte pair encoding (Shibata et al. 2001). In recent years, attention has also turned toward the Burrows-Wheeler Transform (BWT) (Burrows & Wheeler 1994), which provides a useful output containing every suffix of the text being compressed sorted into lexicographical order. This structure is closely related to suffix arrays (Manber & Myers 1993) and suffix trees (Weiner 1973), which both supply an efficient index for searching text.

Currently, BWT (described in detail in Chapter 2) is considered second only to PPM (Prediction by Partial Match) (Cleary & Witten 1984) for compression ratio, but has a decided advantage in terms of speed. LZ-based methods, though fast, perform poorly in size reduction, leaving BWT as an ideal compromise. Coupled with the promising ability to search its structure, it is an ideal tool in compressed-domain pattern matching. While there have been a number of competitive algorithms developed to search text after it has been compressed with the BWT algorithm, few comparisons have been provided. Thus, the main goal of this project is to evaluate and compare the approaches that are available. The search algorithms are described in Chapter 3, with Section 3.6 introducing modifications to four of the algorithms to improve search time and, in some cases, reduce memory requirement. Chapter 4 provides the results from a set of experiments that evaluate the performance of the search algorithms.

1.1 Offline and Online Algorithms

Pattern matching algorithms have traditionally been separated into two classes: *offline* and *online*. An offline approach constructs an index that is stored with the text and is subsequently used to process queries. This method requires additional storage space but generally increases search performance. Online pattern matching approaches, on the other hand, only store the text; thus, all work must be performed at query-time.

When discussing pattern matching in the compressed-domain, particularly with BWT algorithms, the boundary between these two classes is not as clear. Some papers have defined Binary Search (Section 3.2) and q-grams (Section 3.4) as online approaches and Suffix Arrays (Section 3.3) as an offline approach. Through the similarities outlined in Chapter 3, however, it is clear that all three should have the same classification.

Further evidence of the uncertainty in the classifications come from the creators of the FM-index (Section 3.5) who have argued that their approach is online and the rest are offline. Although the FM-index must store additional indexing information with the BWT compressed file, the index construction may be considered part of the compression process. At search time, even with the indexes available, further work beyond a simple index lookup must be performed for each query, possibly leading to an online classification. Furthermore, Binary Search, Suffix Arrays and q-grams also require indexes, and even though they are constructed at query-time and stored temporarily in memory, they must be created before searching begins. Once created, they may be used many times to locate patterns until the search program ends and the indexes are removed from memory. In this sense, they could potentially be considered offline because searching is separated from index construction.

For the purposes of this report, we consider an offline algorithm to be any algorithm that requires information beyond the compressed representation of the text to be stored on disk. Thus, we consider the FM-index an offline algorithm, with the remaining algorithms (Compress-Domain Boyer-Moore, Binary Search, Suffix Arrays and q-grams), classified as online.

1.2 Notation

In this report, pattern matching will be referenced in terms of searching for a pattern P of length m in a text T of length n. The number of times P occurs in the text is denoted by occ. The alphabet of the text is $\sum$, with $|\sum|$ representing the size of the alphabet. Other symbols will be defined as they are used, with Chapter 2, in particular, defining many of the arrays used to perform the Burrows-Wheeler Transform and to perform searches.

Chapter 2

The Burrows-Wheeler Transform

The Burrows-Wheeler Transform is a process whereby the symbols in a text are permuted to an order that can be efficiently compressed by some other means. In this permutation, symbols occurring in a similar context in the original text are located near each other. This is achieved by sorting the characters in the text according to the context in which they appear, where the context of a character is the characters that immediately follow it. Figure 2.1 illustrates this using a fragment of `alice29.txt` from the Canterbury Corpus (Arnold & Bell 1997, Bell & Powell 2002), and shows contexts starting with `ice` followed by a space. The characters in this context are shown in the left column and when read from top to bottom (`lololllmlllllllotll`) make up a small part of the BWT permutation for `alice29.txt`. Only a few characters (`l`, `o`, `m` and `t`) occur in this range of contexts making it particularly suitable for compression using move-to-front coding (Bentley et al. 1986), which represents recently occurring characters using a smaller number of bits than other characters. This compression is discussed further in Section 2.3.

Construction of the BWT permutation can be visualized using Figure 2.2, which demonstrates the steps using the text '`mississippi`'. The first step is to produce the matrix in Figure 2.2(a), which contains all cyclic rotations of the text. Next, this matrix is sorted to produce the matrix in Figure 2.2(b). Because each row is a cyclic rotation of the original text, the character at the end of a row cyclically proceeds the characters at the beginning, and thus, a row is the context of its last character. As a result, the last column of the sorted matrix (reproduced in Figure 2.2(c)) is the BWT permutation for the text. This column is often referred to as L, with the first column, which contains all characters of the text in sorted order, referred to as F. In practice, it is not necessary to construct these large n by n matrices. Instead, an array of length n is used with each entry pointing to a separate symbol of the text and sorted according to the context of the symbol that is referenced.

To aid the decompression process, it is common to transmit the position in the sorted column of the first character of the original text so the decoder knows where to begin. For the `mississippi` example in Figure 2.2, it is the fifth position, because that is the location of `m` in the first column of the sorted matrix. Thus, the BWT output for the text `mississippi` is the pair $\{$`pssmipissii`, $5\}$.

From just this pair it is possible to reconstruct the original text. This process relies on the following relationship between the first and last columns of the sorted matrix: The order in which corresponding characters appear in the two columns are the same. For example, the first occurrence of `s` in F (line 8) and the first occurrence in L (line 2) correspond to the same `s` in the input text. Likewise, the second occurrences (lines 9 and 3) also correspond to the same `s` of the text. Additionally, because each row is a cyclic rotation of the text, we know that a character in the last column will be followed in the text by the corresponding character in the first column of the same row.

Although the decoder only receives L, it is able to reproduce F by simply sorting the characters in L. After this is done, it can begin decoding the text, starting with the character in the position that was transmitted as part of the output pair. In our example, the first character is `m` in $F[5]$. Because this is the only occurrence of `m` in the text, it must correspond to the `m` in $L[4]$. As described previously, the next character will be in the same row of F, thus $F[4]$ reveals the second character is `i`. This is the fourth appearance of `i` in F, which corresponds to the fourth appearance in L ($L[11]$), allowing decoding to continue by locating the next character, which is stored in $F[11]$. The remaining characters are decoded using contexts 9, 3, 10,

```
l   ice again. ‘No, I give it up,’ Alice
o   ice all talking together: she made o
l   ice alone with the Gryphon. Alice di
o   ice along--‘Catch him, you by the he
l   ice aloud, addressing nobody in part
l   ice an excellent opportunity for cro
l   ice and all her wonderful Adventures
m   ice and rabbits. I almost wish I had
l   ice appeared, she was appealed to by
l   ice as he said do. Alice looked at t
l   ice as he spoke. ‘A cat may look at
l   ice as it spoke. ‘As wet as ever,’ s
l   ice as she picked her way through th
l   ice asked in a tone of great curiosi
l   ice asked. ‘We called him Tortoise b
l   ice asked. The Hatter shook his head
o   ice at her side. She was walking by
t   ice before, but she had read about t
l   ice began in a loud, indignant voice
l   ice began telling them her adventure
```

Figure 2.1: Sorted contexts for the Burrows-Wheeler Transform.

8, 2, 7, 6 and 1. An efficient implementation of this inverse BWT operation is described in Section 2.1.

2.1 Decoding Implementation

The seminal BWT paper (Burrows & Wheeler 1994) provides an algorithm to perform the inverse BWT operation in linear time by making an initial pass through the encoded string counting characters. A second pass, in an order dictated by the counts, results in the original text. This is shown in Algorithm 2.1, where the second parameter of BWT-DECODE, *index*, is the position in F of the first character of the text. Note that some variable names have been altered for consistency with other algorithms in this report. Figure 2.3 shows the values, using the `mississippi` example, for the arrays in this algorithm, as well as other arrays used to search BWT.

After the second `for` loop (starting on line 5), $C[i]$ contains the number of instances of the character $L[i]$ in $L[1 \ldots i-1]$ and $K[ch]$ contains the number of the times the character *ch* occurs in the entire text. The following `for` loop iterates through all characters in the alphabet and populates M so that it has a cumulative count of the values in K; that is, $M[i]$ contains the sum of $K[0 \ldots i-1]$. In effect, M stores the positions of the start of all groups of characters in F. As a result, we do not need to explicitly store F, and have constructed it in linear time rather than $O(n \log n)$, which would be required to actually sort the characters. Additionally, this saves memory and also has important implications in some of the search algorithms, as described in Chapter 3. Finally, the last `for` loop reconstructs the original text in the array T.

Burrows & Wheeler (1994) also introduce a *transform array* that provides an alternative mechanism for decoding the text. The array is constructed such that, for any character $L[i]$, the preceding character in the text is given by $L[V[i]]$, that is:

$$\forall i : 1 \leq i \leq n, T[n-i+1] = L[V^i[index]]$$

where $V^0[x] = x$, $V^{i+1}[x] = V[V^i[x]]$, and *index* is the position in F of the first character of the text. Thus, V simply stores the result of line 19 of Algorithm 2.1 in an array so that it can be accessed later, possibly in a random order. For decompression, using the transform array has no advantages over the technique already

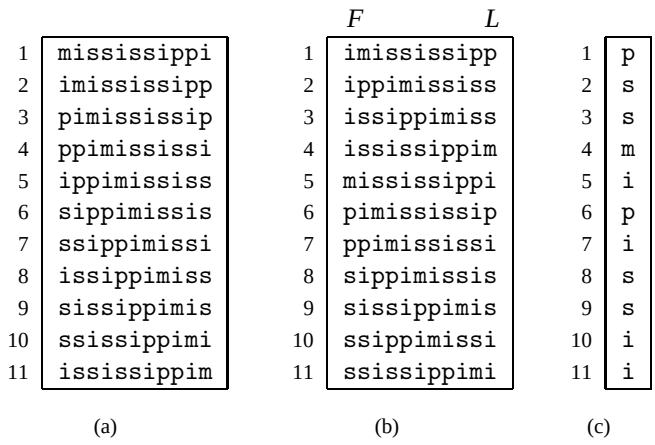


Figure 2.2: The Burrows-Wheeler Transform for the text `mississippi`: (a) cyclic rotations of the text; (b) sorted matrix; (c) resulting BWT permutation (last column of the sorted matrix).

i	T	F	L	C	V	W	I	Hr
1	m	i	p	0	6	5	5	11
2	i	i	s	0	8	7	4	8
3	s	i	s	1	9	10	11	5
4	s	i	m	0	5	11	9	2
5	i	m	i	0	1	4	3	1
6	s	p	p	1	7	1	10	10
7	s	p	i	1	2	6	8	9
8	i	s	s	2	10	2	2	7
9	p	s	s	3	11	3	7	4
10	p	s	i	2	3	8	6	6
11	i	s	i	3	4	9	1	3

Figure 2.3: Array values to perform and search the Burrows-Wheeler Transform of the text `mississippi`.

described, but the concept is an important part of many search algorithms because it provides a mechanism for decoding arbitrary length substrings of the text at random locations. The V transform array, however, reconstructs the text in reverse order. While this is acceptable when decoding the entire text (by populating the resulting text array in reverse order), it is useless for decoding random substrings during a search. With this in mind, Bell et al. (2002) have defined the *forwards transform array*, W, as follows:

$$\forall i : 1 \leq i \leq n, T[i] = L[W^i[index]]$$

where $W^0[x] = x$, $W^{i+1}[x] = W[W^i[x]]$, and *index* is the position in F of the first character of the text. Construction of both V and W is shown in Algorithm 2.2 using the M array as previously defined. Algorithm 2.3 illustrates how W can be used to decode the text.

2.2 Auxiliary Arrays

Many of the search algorithms evaluated in this report also require the use of other arrays, known as *auxiliary arrays*. They were defined by Bell et al. (2002) and Adjeroh et al. (2002) to provide a mapping between the text and the sorted array, F. Figure 2.3 shows the values for these arrays using the text `mississippi`.

Algorithm 2.1 Reconstruct the original text

```
BWT-DECODE(L, index)
 1 for i ← 0 to 255 do
 2     K[i] ← 0
 3 end for
 4
 5 for i ← 1 to n do
 6     C[i] ← K[L[i]]
 7     K[L[i]] ← K[L[i]] + 1
 8 end for
 9
10 sum ← 1
11 for ch ← 0 to 255 do
12     M[ch] ← sum
13     sum ← sum + K[ch]
14 end for
15
16 i ← index
17 for j ← n downto 1 do
18     T[j] ← L[i]
19     i ← C[i] + M[L[i]]
20 end for
```

Algorithm 2.2 Construct the BWT transform arrays

```
BUILD-TRANSFORM-ARRAYS(L, M)
1 for i ← 1 to n do
2     V[i] ← M[L[i]]
3     W[M[L[i]]] ← i
4     M[L[i]] ← M[L[i]] + 1
5 end for
```

Hr maps characters of the original text to their location in the sorted string F. It is defined as:

$$\forall i : 1 \leq i \leq n, T[i] = F[Hr[i]]$$

I is the inverse of Hr and is defined as:

$$\forall i : 1 \leq i \leq n, T[I[i]] = F[i]$$

Both arrays can be constructed in $O(n)$ time, as shown in Algorithm 2.4.

2.3 Compressing the BWT Output

The Burrows-Wheeler Transform does not actually produce any compression — the resulting permutation has the same length as the input string. It does however, provide a string that can be efficiently compressed by some other means. As revealed earlier in this chapter, the output of the transform usually contains clusterings of a small ranges of characters. Although there are many possibilities for compressing this kind of structure, only the two approaches we used for evaluating the search algorithms will be considered in this report.

The search algorithms described in Chapter 3, excluding the FM-index, will work with any compression scheme suitable for BWT because they do not take the compression technique into consideration and

Algorithm 2.3 Reconstruct the original text using the W array

BWT-DECODE$'(L, W, index)$
1 $i \leftarrow index$
2 **for** $j \leftarrow 1$ **to** n **do**
3 $\quad i \leftarrow W[i]$
4 $\quad T[j] \leftarrow L[i]$
5 **end for**

Algorithm 2.4 Construct the Hr and I auxiliary arrays

BUILD-AUXILIARY-ARRAYS$(W, index)$
1 $i \leftarrow index$
2 **for** $j \leftarrow 1$ **to** n **do**
3 $\quad Hr[j] \leftarrow i$
4 $\quad I[i] \leftarrow j$
5 $\quad i \leftarrow W[i]$
6 **end for**

must reverse the compression to retrieve the permuted string before searching can begin. For consistency with other evaluations of Binary Search and Compressed-Domain Boyer-Moore, the implementation used to evaluate these algorithms will employ the technique used by `bsmp` (Bell et al. 2002). This involves three stages: The first passes the BWT output through a move-to-front coder (Bentley et al. 1986) to take advantage of the clustering of characters. The resulting output is then piped into a run-length coder to remove long sequences of zeros. Finally, an order-0 arithmetic coder compresses the run lengths.

The compression for the FM-index is provided by a move-to-front coder, followed by a Multiple Table Huffman coder (Wheeler 1997). Although this results in a lower compression ratio than `bsmp`, it is faster and allows random access into the compressed file, which permits searching without reversing the compression of the entire file. As well as the compressed text, auxiliary indexing information is also stored to improve search performance at the cost of the size of the resulting file. Further details of the indexes are given in Section 3.5.2.

Chapter 3

BWT Search Algorithms

The following sections provide a brief description of the methods available to search text that has been compressed using the Burrows-Wheeler Transform. Excluding the FM-index (Section 3.5), they all operate on the BWT-encoded permutation of the text, which means partial decompression is required to return a compressed file to the appropriate structure before searching begins.

Section 3.6 introduces a technique to reduce the search times of Binary Search, Suffix Arrays and q-grams, as well as reducing the memory requirement of the latter two algorithms. A modification to the FM-index is also described with the aim of improving search time at the cost of a higher memory requirement.

Compressed-Domain Boyer-Moore (Section 3.1) uses a technique that allows the text to be accessed in the correct order without fully decompressing it with the inverse BWT operation, making the use of an ordinary search algorithm possible. The remaining algorithms use the sorted contexts provided by BWT to increase their search performance through a binary search technique.

Two more search algorithms (Sadakane 2000*a*, Sadakane 2000*b*) have often been referenced in recent literature discussing compressed-domain pattern matching with BWT. They will not be evaluated as part of this research, however, for the reasons discussed below.

Sadakane (2000*a*) introduced a data structure based on the compressed suffix array of Grossi & Vitter (2000). The compression provided is not related to BWT, and instead refers to the ability to store the array in $O(n)$ space rather than $O(n \log n)$ as required by traditional suffix arrays. This compression is achieved using a hierarchical structure where upper levels can be reconstructed from lower levels (that have fewer entries) and therefore do not need to be explicitly stored. Because it does not involve BWT, the algorithm will not be considered in this report.

Sadakane (2000*b*) provides an algorithm for case insensitive searches of a BWT compressed text. This algorithm is similar to Suffix Arrays (Section 3.3), and is trivial to implement by altering the function for comparing symbols in both the encoder and search programs. When case sensitive comparisons are necessary, the results from a case insensitive search need to be filtered to get the exact matches, increasing the search time. Excluding the difference in symbol comparisons, Suffix Arrays and the case insensitive search algorithm are identical, so the latter will not be considered further in this report.

3.1 Compressed-Domain Boyer-Moore

The Boyer-Moore algorithm (Boyer & Moore 1977) is currently considered to be one of the most efficient pattern matching algorithms for searching an ordinary text file (Gusfield 1997). Using shift heuristics, it is able to avoid making comparisons with some parts of the text and and can therefore produce a sub-linear performance of $O(m/n)$ in the best case, but in the worst case, deteriorates to $O(mn)$ time complexity. This requires access to the text in the correct order, so that after a file has undergone the Burrows-Wheeler Transform, an ordinary Boyer-Moore search is no longer possible. Section 3.1.1 provides details of the standard Boyer-Moore algorithm, with Section 3.1.2 describing the necessary changes, introduced by Bell et al. (2002), for use with BWT.

3.1.1 Boyer-Moore

The Boyer-Moore algorithm scans the query pattern from right to left, making comparisons with characters in the text. When a mismatch is found, the maximum of two precomputed functions, called the *good-suffix rule* and *bad-character rule*, is used to determine how far to shift the pattern before beginning the next set of comparisons. This shifts the pattern along the text from left to right, without missing possible matches, until the required patterns have been located or the end of the text is reached.

The good-suffix rule is used when a suffix of P has already been matched to a substring of T, but the next comparison results in a mismatch. Let the matched substring be t, and t' be the next rightmost occurrence of t in P, such that the characters in the pattern to the left of t and t' are not the same. The pattern is then shifted so that t' is aligned with the occurrence of t in the text. If t' does not exist, the pattern is shifted to the right until a prefix of P is aligned with a suffix of t in T, or completely past t if no such match exist. Figure 3.1(a) shows an example where a mismatch has been detected at P[5]. The good-suffix is `te`, which occurs again in the pattern at $P[2]$. The shift produced by the good-suffix rule, shown in Figure 3.1(b), aligns this occurrence with the previously matched substring in the text. Comparing then continues from the rightmost character of the pattern. A table of shift distances for the good-suffix rule can be computed before searching begins in $O(m)$ amortised time and requires $O(m)$ space to store.

The bad-character rule proposed by Boyer & Moore (1977) has been improved, resulting in the *extended bad-character rule* (Gusfield 1997), which usually provides larger shift distances. For the extended rule, when a mismatch occurs at position i in P, let the mismatched character in T be c. The patten is shifted so that c is aligned with the rightmost occurrence of c in $P[1 \ldots i-1]$, that is, to the rightmost occurrence that is to the left of the mismatch. If c does not occur in $P[1 \ldots i-1]$, the pattern is shifted completely past c. Figure 3.1(c) shows the shift proposed by the extended bad-character rule, where the mismatched character in T is `e`, which occurs in the pattern to the left of the mismatch at $P[3]$. This results in a shift of two characters so that these occurrences of `e` now align. A table for the extended bad-character rule can be calculated before searching begins in $O(m+|\sum|)$ time and requires $O(m+|\sum|)$ space.

3.1.2 Modifications for the Compressed-Domain

To be used in the compressed-domain, the Boyer-Moore algorithm must be able to access the text in the correct order. For BWT compression, this is achieved by decoding parts of the text, as needed, through the F array and Hr arrays as shown in Algorithm 3.1.

Algorithm 3.1 Boyer-Moore for BWT compressed text

```
COMPRESSED-DOMAIN-BOYER-MOORE-SEARCH(P,F,Hr)
 1 COMPUTE-GOOD-SUFFIX(P)
 2 COMPUTE-BAD-CHARACTER(P)
 3 k ← 1
 4 while k ≤ n − m + 2 do
 5    i ← m
 6    while i > 0 and P[i] = F[Hr[k + i − 2]] do
 7       i ← i − 1
 8    end while
 9    if i = 0 then
10       # Report a match beginning at position k − 1
11       k ← k + <shift proposed by the good-suffix rule>
12    else
13       s_G ← <shift proposed by the good-suffix rule>
14       s_B ← <shift proposed by the extended bad-character rule>
15       k ← k + MAX(s_G, s_B)
16    end if
17 end while
```

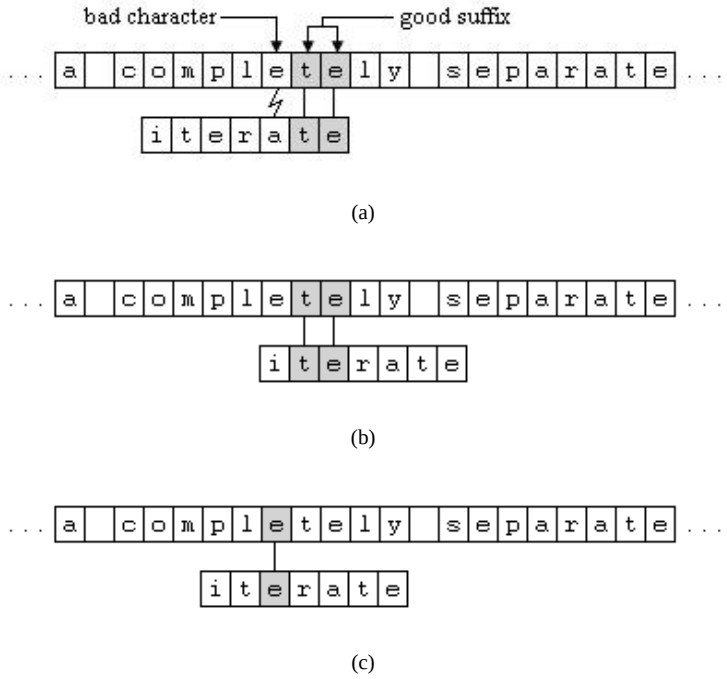


Figure 3.1: Examples of shifts proposed by the Boyer-Moore heuristics: (a) Matching the pattern `iterate` against the text results in a mismatch at position 5 of the pattern; (b) Shift proposed by the good-suffix rule; (c) Shift proposed by the extended bad-character rule.

3.2 Binary Search

The output of the Burrows-Wheeler Transform is remarkable in that it provides access to a list of all substrings of the text in sorted order. This makes it possible to use a binary search approach that operates in $O(m \log n)$ time. The sorted list of substrings for the text `mississippi` is shown in Figure 3.2. This is taken from the sorted matrix in Figure 2.2(b), but has the characters removed that occur cyclically after the last character of the text. If a search pattern appears in the text, it will be located at the beginning of one or more of these lines. Additionally, because the list is sorted, all occurrences of a search pattern will be located next to each other; for instance, `si` appears at the start of lines 8 and 9.

In practice, this structure is accessed through the M array, which stores the starting locations of each group of characters in F, and thus provides a 'virtual index' to the first character of each row in the sorted substring list. The remaining characters in a row are decoded as needed using the W transform array. A row need only be decoded to perform a string comparison as part of the binary search, and even then, only enough is decoded to determine whether the line provides a match. This comparison is illustrated in Algorithm 3.2, where i is the number of the row being compared to the pattern, P. If t is a string representing that row, the return value of the function is 0 if P is a prefix of t, negative if $p < t$ and positive if $p > t$.

The use of the M array to index the substrings also allows an improvement on the $O(m \log n)$ performance of binary search by narrowing the initial range of the search. If c is the first character of the pattern, the initial lower and upper bounds for a binary search are given by $M[c]$ and $M[c+1]-1$. For instance, in the example in Figure 3.2, if the search pattern begins with the letter `s`, M tells us that it can only occur between lines 8 and 11. This range contains $\frac{1}{|\Sigma|}$ of the total number of rows on average and therefore reduces the search time to $O(m \log \frac{n}{|\Sigma|})$ on average.

Binary Search on a BWT compressed file is illustrated in Algorithm 3.3 (extracted from Powell (2001))

```
 1 i
 2 ippi
 3 issippi
 4 ississippi
 5 mississippi
 6 pi
 7 ppi
 8 sippi
 9 sissippi
10 ssippi
11 ssissippi
```

Figure 3.2: Sorted substrings for the text `mississippi`.

and operates as follows: A standard binary search on the range $M[c] \dots M[c+1]-1$ results in a match with one occurrence of the pattern if any exists. It is also necessary, however, to locate other occurrences. This could be done by a simple linear search backward through the sorted substrings until the first mismatch is found, as well as forward to find the first mismatch in that direction (thus, identifying the first and last occurrence of the pattern). This would take $O(occ)$ time, however, and would be rather time consuming if there are many occurrences. Instead, it is more efficient to apply two further binary searches. The first search locates the first substring that has P as a prefix and operates on the range $M[c] \dots p-1$, where p is the location of the initial match. Like a standard binary search, each step compares the midpoint of the range to the pattern, however, if the comparison function returns a negative value or zero, it continues searching the range $low \dots mid$; otherwise, it searches the range $mid+1 \dots high$. The second search locates the last occurrence of P and is performed in the range $p+1 \dots M[c+1]-1$, but this time choosing the range $low \dots mid-1$ for a negative comparison result and $mid \dots high$ for a positive or zero result. Although it was not noted by Bell et al. (2002), a further improvement can be made by basing the ranges for the two subsequent searches on mismatches of the initial search. The first operates in the range $q \dots p-1$ where q is the largest known mismatched row in the range $M[c] \dots p-1$. A similar range can be identified for the second search.

Finally, after all occurrences have been found in the sorted matrix, the corresponding matches in the text must be located. This is achieved using the I array. If the pattern matches lines $i \dots j$ of the sorted matrix, which corresponds to $F[i \dots j]$, then the indices for the matches in the text are identified by $I[i \dots j]$, because I maps between F and T.

Algorithm 3.2 String comparison function for Binary Search

```
BINARY-SEARCH-STRCMP(P, W, L, i)
 1 m ← LENGTH(P)
 2 j ← 1
 3 i ← W[i]
 4 while m > 0 and L[i] = P[j] do
 5     i ← W[i]
 6     m ← m − 1
 7     j ← j + 1
 8 end while
 9 if m = 0 then
10     return 0
11 else
12     return P[j] − L[i]
13 end if
```

Algorithm 3.3 Binary Search algorithm

```
BINARY-SEARCH(P,W,L,I)
 1  c ← P[1]
 2  P' ← P[2...m]
 3  low ← M[c]
 4  high ← M[c+1]−1
 5
 6  while low < high do
 7     mid ← (low+high)/2
 8     cmp ← BINARY-SEARCH-STRCMP(P',W,L,W[mid])
 9     switch cmp
10        case = 0 : break
11        case > 0 : low ← mid+1
12        case < 0 : high ← mid
13     end switch
14  end while
15
16  if cmp = 0 then
17     p ← mid
18     h ← p−1
19     while low < h do
20        m ← (low+h)/2
21        if BINARY-SEARCH-STRCMP(P',W,L,W[m]) > 0 then
22           low ← m+1
23        else
24           h ← m
25        end if
26     end while
27     if BINARY-SEARCH-STRCMP(P',W,L,W[low]) ≠ 0 then
28        low ← mid          # No matches in low...mid−1
29     end if
30
31     l ← p+1
32     while l < high do
33        m ← (l+high+1)/2          # Round up
34        if BINARY-SEARCH-STRCMP(P',W,L,W[m]) ≥ 0 then
35           l ← m
36        else
37           high ← m−1
38        end if
39     end while
40     if BINARY-SEARCH-STRCMP(P',W,L,W[high]) ≠ 0 then
41        high ← mid          # No matches in mid+1...high
42     end if
43
44     return {I[low...high]}
45  else
46     return {}          # No matches found
47  end if
```

3.3 Suffix Arrays

Sadakane & Imai (1999) provide an algorithm for efficiently creating a suffix array (Manber & Myers 1993) for a text from the BWT permutation of that text. A suffix array is an index to all substrings of a text sorted in the lexicographical order of the substrings, and therefore allows patterns to be located in the text through a binary search of the index. This array is very similar to the sorted context structure used by Binary Search (Section 3.2) but Suffix Arrays indexes the decoded text, whereas Binary Search uses W to index the corresponding encoded substrings in L. Additionally, Binary Search uses W to decode the substrings as needed, but Suffix Arrays must decode the entire text before searching begins. For this reason, Suffix Arrays cannot actually be considered a compressed-domain pattern matching algorithm and may be better classified as an *indexed-decompress-then-search* approach.

The suffix array is simply the I array defined in Section 2.2. Sadakane & Imai (1999), however, describe an implementation where I is constructed at the same time as the text is decoded. This is shown in Algorithm 3.4 as a modification to Algorithm 2.1, which only decodes the text.

Algorithm 3.4 Modification to Algorithm 2.1 to construct a suffix array as the text is decoded

```
⋮
12 i ← index
13 for j ← n downto 1 do
14    I[i] ← j + 1
15    if I[i] = n + 1 then
16       I[i] ← 1
17    T[j] ← L[i]
18    i ← C[i] + M[L[i]]
19 end for
```

Pattern matching with this structure can be performed in a manner similar to that of the Binary Search approach. In fact, the steps described in Algorithm 3.3 can be reused, with only alterations to the calls to BINARY-SEARCH-STRCMP. These calls are replaced with:

$$\text{SUFFIX-ARRAY-STRCMP}(P', L, I[x])$$

where x is the same as that of $W[x]$ in the corresponding line of the original algorithm. This string comparison function for Suffix Arrays is much simpler than that of Binary Search because the text has already been decoded and is referenced directly. It differs from an ordinary string comparison that might be found in a standard programming language library in that it also reports that a match exists if the first string (the pattern) is a prefix of the second — they are not required to have the same length.

3.4 q-grams

Adjeroh et al. (2002) describe their q-gram approach in terms of sets and set intersections. For exact pattern matching, however, the most efficient implementation of these operations is very similar to the Binary Search approach (Section 3.2).

A q-gram is a substring of a text, where the length of the substring is q. For example, the set of 3-grams for the text `abraca` is {`abr`,`bra`,`rac`,`aca`}. For exact pattern matching, we construct all m length q-grams (the m-grams) of the pattern and the text. Intersecting these two sets produces the set of all matches. If instead we wish to perform approximate matching, the size of the q-grams depends on the allowable distance between the pattern and a matching string. Approximate pattern matching, however, will not be considered further in this report.

There is just one m-gram of a pattern, which is simply the pattern itself. Construction of the required m-grams of the text is also straightforward and can be performed in $O(n)$ time. This involves the use of the F and Hr arrays, which are used to be generate the q-grams for any given q as follows:

$$\forall i : 1 \leq i \leq n-q+1, Q_q^T[i] = F[Hr[i]] \ldots F[Hr[i+q-1]]$$

Although this definition does not list the q-grams in sorted order, sorting can be performed efficiently by reordering them according to the values in the I auxiliary array. For example, the text `abraca` has $I = \{6,1,4,2,5,3\}$. Thus, for $q = 3$, the sorted q-grams are $\{Q_3^T[1], Q_3^T[4], Q_3^T[2], Q_3^T[3]\}$, with 5 and 6 being ignored because they are greater than $n - q + 1$.

Because the set of q-grams for the pattern contains only one item and the q-grams for the text can be obtained in sorted order, the intersection of these two sets can be performed using binary search with the single string from the pattern's set used as the search pattern. The implementation of this search is almost identical to that of Binary Search, and Algorithm 3.3 may be reused with modifications to only the BINARY-SEARCH-STRCMP calls. These calls are replaced with:

$$\text{QGRAM-STRCMP}(P', Hr, F, I[x])$$

where x is the same as that of $W[x]$ in the corresponding line of the original algorithm. In this respect, it is more closely related to Suffix Arrays (Section 3.3) because both use the I array in place of W to determine the position for a comparison. Like Binary Search, however, it is the job of the string comparison function to decode the required text, whereas Suffix Arrays need only provide a basic comparison of two strings because the text is decoded before searching begins. The q-gram approach to string comparison is shown in Algorithm 3.5 and decodes the text using Hr and F following the q-gram definition given previously.

Algorithm 3.5 String comparison function for q-gram search

```
QGRAM-STRCMP(P, Hr, F, i)
 1  m ← LENGTH(P)
 2  j ← 1
 3  i ← i + 1
 4  while m > 0 and F[Hr[i]] = P[j] do
 5     i ← i + 1
 6     m ← m − 1
 7     j ← j + 1
 8  end while
 9  if m = 0 then
10     return 0
11  else
12     return P[j] − F[Hr[i]]
13  end if
```

3.5 FM-index

Ferragina & Manzini (2000) proposed an Opportunistic Data Structure, so named because it reduces the storage requirements of the text without lowering the query performance. It uses a combination of the BWT compression algorithm and a suffix array data structure to obtain a compressed suffix array. Indexing is added to the resulting structure to allow random access into the compressed data without the need to decompress completely at query-time. The discussion in that paper, however, is purely theoretical and major problems prevent its implementation. In particular, it must be run on a machine with a RAM of word size $\log n$. A more practical implementation that does not have the same asymptotic worst case behaviour, but works well in general, has been described by Ferragina & Manzini (2001). This implementation, referred to as the FM-index by the authors because it provides a Full-text index and requires only Minute storage space, is described here and evaluated in Section 4.

3.5.1 Searching

Searching with the FM-index is performed through two key functions: COUNT and LOCATE. Both use the OCC function, which for $\text{OCC}(c,k)$ returns the number of occurrences of the character c in $L[1 \ldots k]$.

This can be calculated in $O(1)$ time using the auxiliary information stored within the compressed file, as described in Section 3.5.2. The OCC function is an important feature of the FM-index because it allows random entries of the *LF* array (which is identical to the *V* array described in Section 2.1 and will be referred to as *V* from now) to be calculated as needed. Thus, unlike the other algorithms in this chapter, the transform arrays need not be constructed in their entirety before searching begins. When required, an entry $V[i]$ is calculated as $M[c] + \text{OCC}(c,i) - 1$, where $c = L[i]$. This is equivalent to line 19 of Algorithm 2.1. Note that the formula given in Ferragina & Manzini (2001) uses an array defined as *C*. For clarity and consistency with other algorithms, we refer to it as *M* (Section 2.1), where $C[i] = M[i] - 1$. Access to *M* is described in Section 3.5.2

COUNT identifies the starting position *sp* and ending position *ep* of the pattern in the rows of the sorted matrix. The number of times the pattern appears in the text is then $ep - sp + 1$. This takes $O(m)$ time and is illustrated in Algorithm 3.6. The algorithm has *m* phases, where, at the *i-th* phase, *sp* points to the first row of the sorted matrix that has $P[i \ldots m]$ as a prefix and *ep* points to the last row that has $P[i \ldots m]$ as a prefix. Thus, after the *m* phases, the first and last occurrences of the pattern are referenced.

Algorithm 3.6 Counting pattern occurrences with the FM-index

```
COUNT(P,M)
 1  i ← m
 2  c ← P[m]
 3  sp ← M[c]
 4  ep ← M[c+1] − 1
 5
 6  while sp ≤ ep and i ≥ 2 do
 7     c ← P[i−1]
 8     sp ← M[c] + OCC(c, sp−1)
 9     ep ← M[c] + OCC(c, ep) − 1
10     i ← i−1
11  end while
12  if ep < sp then
13     return ep − sp + 1
14  else
15     return 0
16  end if
```

LOCATE takes the index of a row in the sorted matrix and returns the starting position of the corresponding substring in the text. Thus, an iteration over the range $sp \ldots ep$ identified by COUNT, calling LOCATE for each position, will result in a list of all occurrences of the pattern in the text. The locations are also calculated using the auxiliary information, as shown in Algorithm 3.7. For a subset of the rows in the sorted matrix, known as *marked rows*, their location in the text is stored explicitly. The technique for determining which rows are marked and how they are represented is discussed in Section 3.5.2. The location of row *i* is denoted by $pos(i)$, and if it is a marked row, the value is available directly. If *i* is not marked, however, *V* is used to locate the previous character, $T[pos(i) - 1]$, in the text. This is repeated *v* times until a marked row, i_v, is found, and therefore $pos(i) = pos(i_v) + v$. In fact, $pos(i)$ will have the same value as $I[i]$, so we are simply storing a subset of the *I* array.

In many respects, the search algorithm of the FM-index is very similar to that of Binary Search (Section 3.2), but where Binary Search first locates one instance of the pattern in the sorted matrix and then uses another two binary searches to locate the first and last instances, the FM-index uses an incremental approach, identifying the first and last occurrences of the suffixes of the pattern, increasing the size of the suffix until the locations have been found for the entire pattern. Additionally, lines 8 and 9 of Algorithm 3.6 effectively perform mappings using the *V* array rather than *W* as used by Binary Search. Because the pattern is processed backwards, it is necessary to construct the text in reverse, which can be achieved using *V*. Also, Binary Search is able to report the location in the text of a match with one array lookup to the *I* auxiliary array, instead of the more complex operations employed by the LOCATE function, which

Algorithm 3.7 Locating the position of a match in the original text using the FM-index

```
LOCATE(i)
1 i' ← i
2 v ← 0
3 while row i' is not marked do
4     c ← L[i']
5     m ← Occ(c, i')
6     i' ← M[c] + m − 1
7     v ← v + 1
8 end while
9 return pos(i') + v
```

effectively reconstructs parts of I as needed.

3.5.2 Compression and Auxiliary Information

The compression process used by the FM-index is different from the other algorithms in this chapter. This is to allow random access into the compressed file. Additional indexing information is also stored with the compressed file, so that the search algorithm may perform the OCC function efficiently and report the location of matches.

To compress the text, the BWT permuted text, L, is created and partitioned into segments of size ℓ_{sb} known as *superbuckets*, with each superbucket being partitioned into smaller segments of size ℓ_b known as *buckets*. The buckets are then compressed individually using Multiple Tables Huffman coding (Wheeler 1997). Ferragina & Manzini (2001) performed extensive experiments with the FM-index and found that 16 kilobyte superbuckets and 1 kilobyte buckets provide a good compromise between compression and search performance in general, so these are the values used for the evaluation in this report.

For each superbucket, a header is created that stores a table of the number of occurrences of all characters in the previous superbuckets. That is, the header for superbucket S_i contains the number of occurrences for each character $c \in \sum$ in $S_1 \ldots S_{i-1}$. Each bucket has a similar header, but contains character counts for the buckets from the beginning of its superbucket. Thus, $\text{OCC}(c,k)$ can be calculated in $O(1)$ time by decompressing the bucket containing $L[k]$ and counting the occurrences in that bucket up to $L[k]$, then adding the values stored for c in the corresponding superbucket and bucket headers. To increase search performance, a *bucket directory* has also been proposed. This directory records the starting positions in the compressed file of each bucket, so that any bucket may be located with a single directory lookup.

This auxiliary information can also be compressed because, as described in Section 2, the L array often has clusterings of characters, which means that the range of characters in each superbucket will usually be small. A bitmap is stored to identify the characters appearing in each superbucket. Thus, a header only needs to contain counts for characters that are recorded in the corresponding superbucket's bitmap. Furthermore, variable integer coding may be used to reduce the space required for the entries that are stored.

One further structure that must be considered contains the information about the marked rows that identify the location in the text of some of the rows in the sorted matrix. Empirical results have shown that marking 2% of the rows provides a suitable compromise between storage requirements and search speed when using a superbucket size of 16 kilobytes and a bucket size of 1 kilobyte (Ferragina & Manzini 2001). Ferragina & Manzini (2001) have also outlined a number of marking schemes that decide which of the rows should be marked. One possibility marks rows at evenly spaced intervals, where the interval is determined by the percentage of rows that are marked. However, they chose to implement an alternative scheme, which was also used for the evaluation in this report, to make the search algorithm simpler even though it performs poorly in some circumstances. It takes advantage of the fact that each character in the alphabet appears roughly evenly spaced throughout an ordinary English text. The character, c, that appears in the text with the frequency closest to 2% is selected, and any row ending with c is marked by storing its corresponding location using $\log n$ bits. This simplifies the searching because, if i is a marked row, $pos(i)$ is stored in entry

$\text{OCC}(c,i)$ of the marked rows, whereas the former strategy requires extra information to be calculated or stored to relate a marked row to the position where its value is stored. The latter strategy, however, relies heavily on the structure of the text and performance deteriorates significantly if characters are not evenly spaced.

Finally, we note that the search algorithm also requires access to the M array. Although the original paper does not define how M is accessed, because it only contains $|\sum|$ entries, it is possible to store M as part of the auxiliary information. Alternatively, it could be constructed with a single pass over the auxiliary information before searching begins.

3.6 Algorithm Improvements

This section describes possible improvements to the search algorithms, with the goal of reducing search time or memory requirement. Section 3.6.1 introduces overwritten arrays to achieve both of these goals and Section 3.6.2 proposes a modification to the FM-index to reduce search time at the cost of memory usage. The effect of these modifications is investigated in Section 4.5.

3.6.1 Binary Search, Suffix Arrays and q-grams

Through a simple modification to the Binary Search, Suffix Arrays and q-grams algorithms, it is possible to reduce search time, and for the latter two, reduce memory usage. This modification uses a concept called *overwritten arrays* to increase efficiency in the construction of the I array.

The original code, used by q-grams, for creating I from W is shown in Algorithm 2.4. During one iteration of the `for` loop, the i-th element of W is read and a value is stored in the i-th element of I. Those elements are not required by subsequent iterations, and in fact for q-grams, after completing the loop, W is not needed at all. Thus, it is possible to write the entry for $I[i]$ in $W[i]$, avoiding the need to allocate a separate area of memory for a second array. Furthermore, as we shall see in Section 4.5.1, due to a reduction in the number of cache misses during the creation of I, this modification also increases the speed at which the array is created.

In a similar manner, Suffix Arrays is able to create I by writing over C. Binary Search, which also uses Algorithm 2.4 to create I, requires W as part of the searching process, and therefore cannot overwrite it. In Section 4.5.1 however, we find that it is still more efficient than the original approach to recreate W after the initial copy is overwritten by I. This provides the faster performance of the optimisation, but unlike the other algorithms, does not reduce memory usage.

3.6.2 Modified FM-index

To locate the position of a match in the text, the FM-index uses a linear search backwards through the text until it finds a row of the sorted matrix for which the position is stored (see Section 3.5.1). With 2% of the text marked, this will require $0.01n$ steps on average, and because each step requires multiple disk accesses, it is a particularly inefficient approach. Data that is read from disk for each step includes: entries in the bucket directory, bitmaps and possibly a bucket and superbucket header, as well as an entire bucket that must also be partially decompressed.

A possible speed increase could result from caching, in memory, the data that is read from disk to avoid reading some data multiple times. For large searches, however, a more substantial improvement is likely to result from copying all data into memory before searching begins. Although this technique will undoubtedly copy data that is never used, it will be read from disk in a sequential manner, which is considerably faster than the random access used if the information was retrieved individually when needed. The implementation of the Modified FM-index that is used in the experiment in Section 4.5.2 takes this approach by reading all the data and storing it in memory in an uncompressed format (without performing the reverse BWT transform on L). This is an attempt to compromise between the efficiency of the online algorithms, which access all data from memory, and the efficiency of the FM-index, which does not need to create any indexes at search time.

As well as a potential speed increase, the modification has the added advantage of reducing the size of the compressed file. Because there is no need to provide random access into the compressed file, it is unnecessary to store the bucket directory. Additionally, the L array does not need to be compressed in a manner that allows random access. Thus, the Huffman coder used by the FM-index may be replaced by the technique used by the online algorithms (see Section 2.3). This technique employs an arithmetic coder, which provides better compression than a Huffman coder (Witten et al. 1999). Furthermore, without the random access to the headers, we are able to store a value in a header as the difference between it and the corresponding value in the previous header. The differences are compressed using the delta code, much like the compression of an inverted file (Witten et al. 1999). Like the original FM-index, however, a bitmap is used to avoid storing an entry in a header for a character that does not occur in the corresponding bucket.

Chapter 4

Experimental Results

Extensive experiments were conducted to compare the compression performance and search performance of the algorithms in Chapter 3, with the results outlined in the following sections. Results for a decompress-then-search approach (using the standard Boyer-Moore algorithm described in Section 3.1.1) have also been included to provide a reference point. Boyer-Moore was selected as the reference because it is currently considered to be one of the most efficient pattern matching algorithms for searching an ordinary text file (Gusfield 1997).

The implementations of Binary Search, Suffix Arrays and q-grams used in the experiments employ the optimisation technique introduced in Section 3.6.1, with Section 4.5 illustrating the improvement provided by the optimisation. Section 4.5 also explores the performance of the Modified FM-index, which was described in Section 3.6.2.

All experiments were conducted on a 1.4GHz AMD Athlon with 512 megabytes of memory, running Red Hat Linux 7.2. The CPU had a 64 kilobyte first level cache and a 256 kilobyte second level cache.

Unless stated otherwise, searching was performed on `bible.txt`, a 3.86 megabyte English text file from the Canterbury Corpus (Arnold & Bell 1997, Bell & Powell 2002). For most experiments, patterns were randomly selected from the set of words that appear in the text being searched. It is important to note, however, that the selected words may have been substrings of other words. These substrings were also located by the search algorithms. For the experiment on pattern length (Section 4.2.3), the search patterns were not restricted to English words and could be any string that appeared in the text and had the required length.

Each experiment was run 50 times. Graphs show the mean of the 50 samples and, where appropriate, error bars have been included to indicate one standard error above and below the mean. Search experiments used a different set of patterns for each sample unless it was impossible to obtain enough patterns; for instance, when testing the effect of the number of occurrences (Section 4.2.1), large occurrence values did not have more than one pattern.

4.1 Compression Performance

Table 4.1 compares the compression ratio of `bzip2` (Seward 2002), a production-quality compression program that uses BWT, with that of the FM-index and `bsmp` (the compression approach used by all search algorithms in this report, excluding the FM-index). Results are shown for the text files in the Canterbury Corpus.

In most cases, `bzip2` provided the best compression, closely followed by `bsmp`. The exception was `E.coli` where `bsmp` was marginally better. This file contains genetic data, which has little structure, and thus is only compressible due to the ability to store the characters in two bits (because the alphabet has a size of four) instead of the eight bits used in the uncompressed file. In this situation, the technique used by `bsmp` of compressing the entire file in one block has a lower overhead than that of `bzip2`, which segments the file into 900 kilobyte blocks and compresses each block independently of the others.

In all cases, the FM-index produced the largest files. Their size, on average, was more than one bit

File	Size	Compression Ratio bzip2	bsmp	FM-i
alice29.txt	152,089	2.27	2.56	3.52
asyoulik.txt	125,179	2.53	2.85	3.79
bible.txt	4,047,392	1.67	1.79	2.58
cp.html	24,603	2.48	2.72	4.26
E.coli	4,638,690	2.16	2.12	2.69
fields.c	11,150	2.18	2.43	3.88
grammar.lsp	3,721	2.76	2.92	4.65
lcet10.txt	426,754	2.02	2.30	3.30
plrabn12.txt	481,861	2.42	2.74	3.57
world192.txt	2,473,400	1.58	1.60	2.66
xargs.1	4,227	3.33	3.54	5.24
mean		2.31	2.51	3.65

Table 4.1: Compression achieved by algorithms based on the Burrows-Wheeler Transform. Size is in bytes and compression ratio is in bits per character.

per character larger, which is due to the additional indexing information that is stored (see Section 3.5.2). This compares favourably, however, to `mg` (Witten et al. 1999), another offline system for compressing and indexing text. `mg` uses an inverted file for indexing, which, for `bible.txt`, occupies 14.4% of the space of the original file. In contrast, the index structure of the FM-index occupies less than 10%. The FM-index also saves a small amount of space by compressing the text with BWT, as opposed to the word-based Huffman coder used by `mg`. Overall, the FM-index uses 0.68 bits per character less than `mg` when the auxiliary files of `mg` are ignored, and 1.56 less, when they are included.

Table 4.2 shows the time taken by the three compression approaches to compress and decompress the files in the Large Collection of the Canterbury Corpus. Results for the smaller files of the Canterbury Collection were also examined and revealed the same trends. Due to their small sizes, however, the recorded times were often negligible, and thus, the results from these files have been omitted here.

Little effort has been spent in optimising `bsmp`, so it performs poorly when considering compression time. This is not a major concern because the main goal of the project is to examine search performance, which is not affected by the one-off cost of compression. Additionally, a high quality implementation could improve the compression time significantly without affecting the capability of the search algorithms. Furthermore, sorting the suffixes of the text is the slowest part of `bsmp`. For files less than 900 kilobytes, the sorted order of the suffixes is identical to that of `bzip2` (because `bzip2` also compresses the entire file in one block for files of this size), so that if the same sorting implementation was used, compression times would be comparable.

In all cases, `bzip2` recorded the best compression time. The FM-index was slightly slower, partly because it is not as highly optimised as `bzip2`, but also because of the additional time required to create the necessary indexing information.

For this project, decompression time is the more important measurement, because all of the search methods require as least partial decompression for searching. When decompressing, the performance of `bsmp` was substantially closer to that of the FM-index, with most of the difference caused by the slower nature of an arithmetic coder (used by `bsmp`) over a Huffman coder (used by the FM-index). Again, the highly tuned `bzip2` significantly outperformed the other two approaches.

4.2 Search Performance

Search performance is often reported in terms of the number of comparisons required for a search. As shown in Chapter 3, the algorithms based on binary search (Binary Search, Suffix Arrays, q-grams and the FM-index), require $O(m \log \frac{n}{|\Sigma|})$ comparisons. The remaining two algorithms — Compressed-Domain

File	Size	Compression Time			Decompression Time		
		bzip2	bsmp	FM-i	bzip2	bsmp	FM-i
bible.txt	4,047,392	3.29	48.62	6.98	0.98	4.05	1.68
E.coli	4,638,690	4.01	64.45	6.96	1.39	5.53	2.17
world192.txt	2,473,400	2.06	33.14	4.24	0.66	2.39	0.95

Table 4.2: Speed of the compression and decompression algorithms. Size is in bytes and times are in seconds.

Boyer-Moore and the decompress-then-search approach evaluated here (both based on Boyer-Moore), use $O(\frac{n}{m})$ comparisons. These two formulae only consider the actual searching process, however, and ignore the requirements of some algorithms to create indexes or decompress the text before searching begins. A better measure of search time would be $O(n+sm\log\frac{n}{|\Sigma|})$ and $O(n+\frac{sn}{m})$, respectively, where s is the number of searches performed, and the additional $O(n)$ term covers the decompression and indexing steps, which operate in linear time. An exception is the FM-index, which creates the necessary indexing information at compression time; thus, its entire search process operates in $O(sm\log\frac{n}{|\Sigma|})$ time.

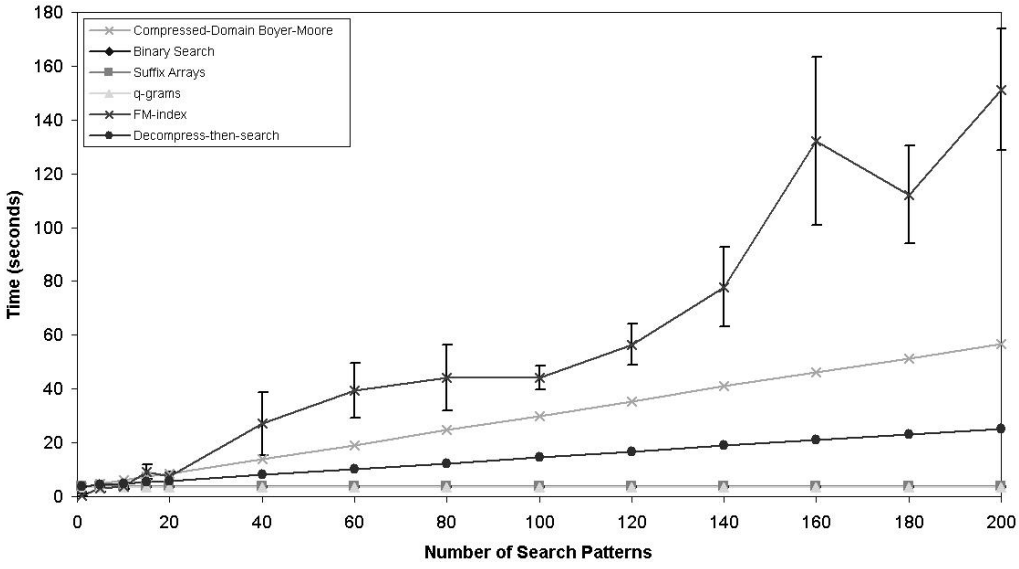
The actual performances of the algorithms vary greatly depending on which arrays are used for indexing and how they are constructed. These differences are hidden by the O-notation, so that an empirical evaluation is important for a meaningful comparison. In Section 4.2.1 we evaluate the performances when locating patterns and explore reasons for the differences between algorithms. Section 4.2.2 discusses the situation where it is only necessary to count the number of times a pattern occurs in the text, without needing to identify the locations of the occurrences. Finally, in Section 4.2.3, we explore additional factors that affect search times, that is, file size, pattern length and file content.

4.2.1 Locating Patterns

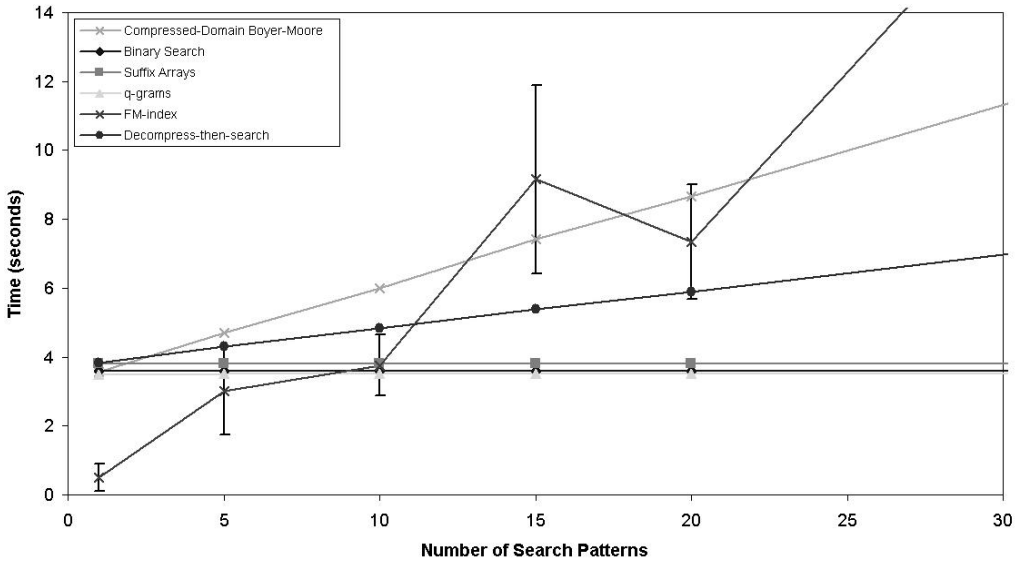
Excluding the FM-index, the search algorithms require the compression of the move-to-front coder, run length coder and arithmetic coder to be reversed, as well as temporary arrays to be constructed in memory before searching begins. Once created, however, the arrays may be used to execute many searches. Thus, multiple searches during one run of a search program will not take the same amount of time as the equivalent number of searches on separate occasions. Situations where multiple searches may be useful include boolean queries with many terms, or interactive applications where users refine or change their queries. Figure 4.1(a) shows the results from an experiment where the number of searches executed during a run of the search programs were varied. Figure 4.1(b) shows the same data, but focuses on a smaller range of the results.

Figure 4.1(a) indicates that Binary Search, Suffix Arrays and q-grams had virtually constant performances regardless of the number of patterns involved. In fact, further experiments with larger numbers of patterns revealed that times increased by just 1.2 milliseconds per 100 patterns. This is because of the small number of comparisons required for a search and means that almost all of the time recorded was used to construct the required arrays before searching began. From Figure 4.1(b), we see that q-grams was consistently the faster of these three algorithms, closely followed by Binary Search. The differences exist because of the time taken to construct the various indexing arrays that each algorithm requires, and this is discussed further in Section 4.4.

The search times for the decompress-then-search and Compressed-Domain Boyer-Moore algorithms increased linearly as the number of patterns increased. For a small number of patterns, the decompress-then-search approach was slower than Compressed-Domain Boyer-Moore because of the overhead of completely decompressing the text before searching begins. It was the more efficient algorithm, however, when there are a larger number of searches performed. This is because it has direct access to the text to make comparisons, whereas the compressed-domain version must decompress the required substrings before a comparison can be made, and with more searches, more comparisons are required. Figure 4.1(b) shows that the overhead of the comparisons outweighed the initial savings of Compressed-Domain Boyer-Moore when more than three searches were performed. It also shows that for a small number of patterns, Compressed-



(a)



(b)

Figure 4.1: (a) Search times for multiple patterns; (b) Magnified view of the search times.

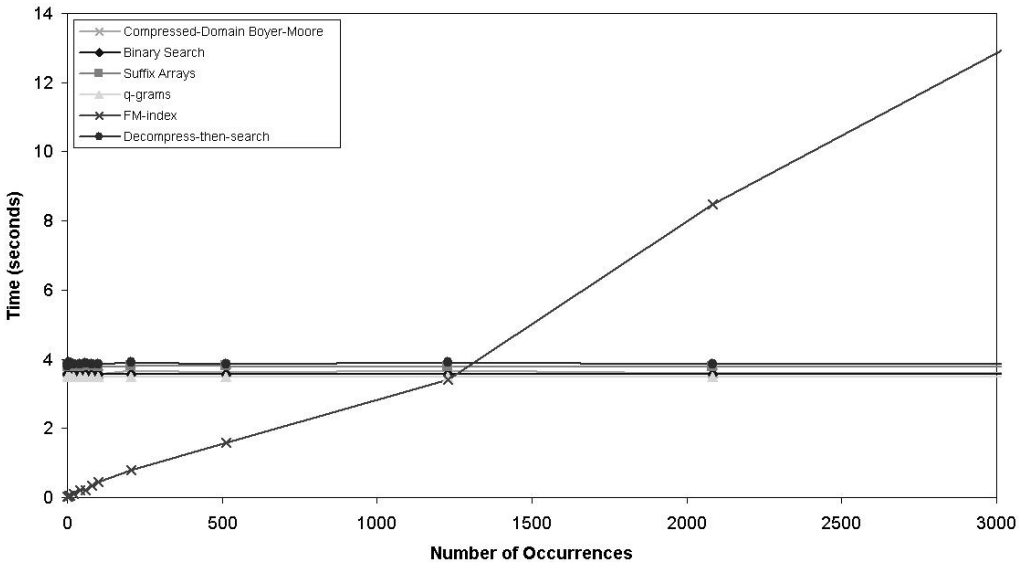


Figure 4.2: Search times for patterns with various numbers of occurrences in the text.

Domain Boyer-Moore was more efficient than Suffix Array and Binary Search, and for a single pattern, provided almost the same performance as *q*-grams. At best, decompress-then-search provided a similar performance to Suffix Arrays.

These results differ from those of Bell et al. (2002) which reported that, at best (for one pattern), decompress-then-search took almost twice as long as Binary Search. They also found that it was not until approximately 20 patterns that decompress-then-search became more efficient than Compressed-Domain Boyer-Moore. After consulting the authors of that paper, it was determined that an error in the decompression part of their decompress-then-search program was reducing its performance significantly, and that the results given here are more accurate.

Finally, we note that the FM-index had the best performance on average until 10 patterns were involved. For a single search, it took only 0.5 seconds on average because, unlike the other algorithms, there is no need to construct any indexes before searching begins. Without the indexing information in memory, however, performance deteriorated significantly as the number of patterns increased, and for more than 25 patterns, it had the worst performance on average.

From the error bars in Figure 4.1(a), we can see that the performance of the FM-index was highly variable. Variations in the other algorithms were insignificant, so error bars, which in most cases were not even visible, have been omitted from the results. The inconsistency of the FM-index is caused by the technique used to locate the positions of matches. If the matching row of the sorted matrix is not marked, the FM-index must iterate backwards through the text until a marked row is found (see Section 3). When the search pattern appears in the text many times, this inefficient location process is executed often, resulting in a poor performance overall. Thus, when a set of randomly selected words contained a pattern that occurred in the text often, its results were significantly slower than the mean. For example, the outlier at 160 patterns occurred because one of the samples of 160 words contained the word ‘an’ which appears in the text (including matching substrings of other words) 61,509 times. Locating all occurrences of this string took 269 seconds. Searches using smaller numbers of patterns exhibit less variation because there is a lower probability of selecting a frequently occurring pattern when only a few patterns are required.

This relationship between the number of occurrences of a pattern and search time is illustrated clearly in Figure 4.2. It shows that search times for the FM-index increased rapidly as the number of occurrences increased. It also shows that the other algorithms had constant performances.

The dramatic effect on the FM-index caused by the number of occurrences of a pattern is illustrated further in Figure 4.3. Here, the search patterns were restricted to those that occurred only once in the

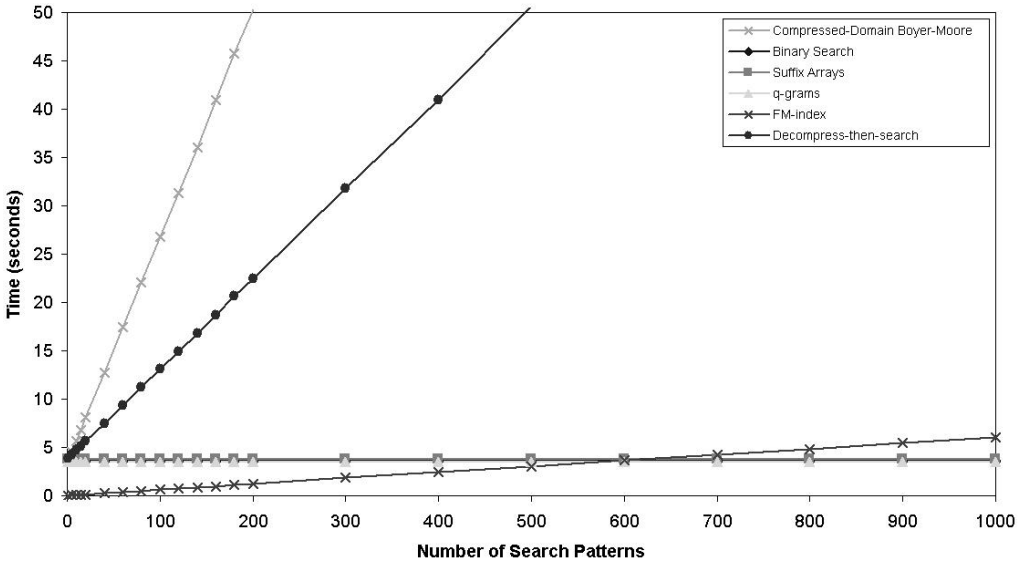


Figure 4.3: Search times for multiple single-occurrence patterns.

text. The FM-index increased slowly at a constant rate, with all other algorithms exhibiting the same performance that was shown in Figure 4.1(a). In this situation, the FM-index was consistently the fastest algorithm when searching for fewer than 600 patterns, at which point *q*-grams became the better option.

We also need to consider efficiency when searching for a single pattern. Figure 4.1(b) shows that, on average, the FM-index significantly outperformed the other algorithms. It is not guaranteed to be the best in all situations, however, because, as described previously, if the pattern appears many times, locating all occurrences will take a considerable amount of time. When considering algorithms that offer consistent performances, Compressed-Domain Boyer-Moore and *q*-grams jointly provided the fastest results. Again, differences among the algorithms are caused by time taken to construct the various indexing arrays that each algorithm requires.

4.2.2 Counting Occurrences

For some applications, it may only be necessary to determine the number of times that a pattern appears in a text, or perhaps, to determine whether it exists at all. An example of such an application is an Internet search engine that returns any page containing a specified pattern, possibly ranked by the number of times the pattern appears. Figure 4.4 shows the results of an experiment where the search programs were only required to count occurrences, with results plotted against the number of patterns that counts were obtained for.

Excluding the FM-index and Binary Search, the results were identical to those in Figure 4.1(a), where matches were also located. Even when match positions are not required, decompress-then-search and Compress-Domain Boyer-Moore must still pass through the entire file to count the appearances. Suffix Arrays and *q*-grams identify the positions of matches using just a single array lookup for each occurrence, so in this case where the positions are not required, they only avoid simple lookup operations and therefore showed no noticeable improvement in their performances.

As described previously in this chapter, the process that the FM-index uses to locate matches is inefficient due to the linear search required for each occurrence. Thus, the FM-index improved substantially when locating matches was unnecessary, and in fact, returned the counts almost instantly regardless of the number of patterns. Binary Search also experienced a significant improvement in this situation, so that in this experiment it was the second fastest algorithm, ahead of *q*-grams by approximately one second. Because the only function of the I array in Binary Search is to determine the positions of matches, it is unnecessary to construct I when the positions are not required, thereby saving a considerable amount of

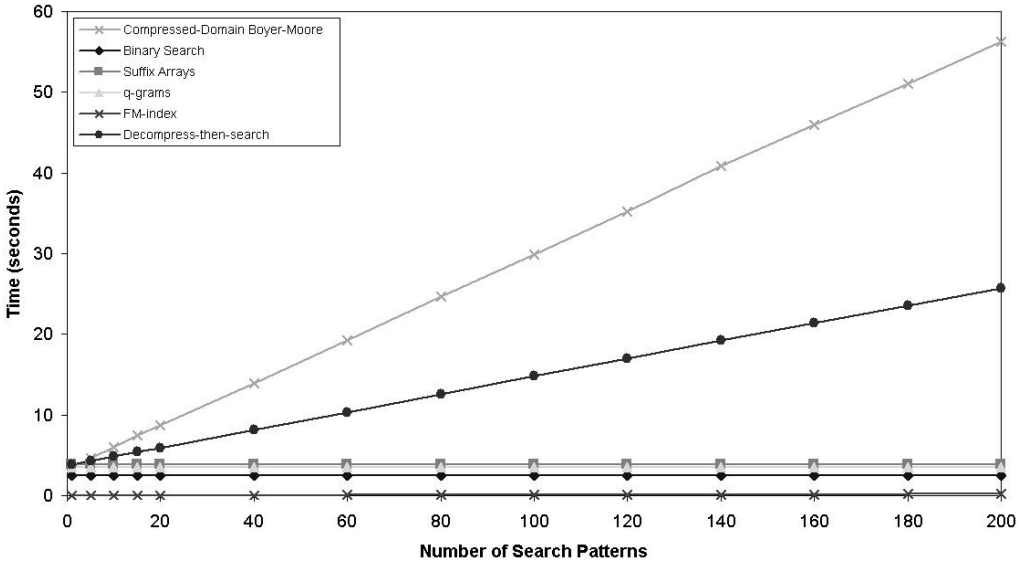


Figure 4.4: Times for counting occurrences of multiple patterns.

time.

4.2.3 Other Factors

Until now, the experiments in this section have only been reported for `bible.txt`, a 3.86 megabyte English text file. The performance of many algorithms can vary considerably, however, if a file of a different size is used, or if the file type is altered. Additionally, the length of the search pattern affects the performance of some algorithms. The following sections outline the results of experiments in which the effects of these factors were explored.

File Size

To determine the effect file size has on the performance of the algorithms, an experiment was run in which searches were executed on files of various sizes. The files were created by concatenating the 1990 'LA Times' files on Disk 5 of the TREC collection (TREC 2002) and truncating to the required sizes. Results from the experiment are shown in Figure 4.5 and reveal that regardless of file size, the FM-index completed a search, on average, almost instantly. Of course, these results are still dependent on the number of occurrences of the pattern and the FM-index will perform poorly if the pattern appears often.

The search times of the other algorithms increased linearly as the file size increased. For small sizes, the algorithms all had similar results. With larger files, however, two groups begin to form: decompress-then-search had a similar performance to Suffix Arrays regardless of file size, and Binary Search, Compressed-Domain Boyer-Moore and q-grams also had similar performances. In fact, even the performances within the groups diverged. With larger file sizes, however, the search times are larger, and the differences, which were all less than a second, were insignificant. The reason that the rate of increase varied among the algorithms is that each requires a distinct set of indexing arrays for searching and those arrays have different construction times (see Section 4.4 for further details).

Figure 4.5 shows only the time required for a single search. Regardless of file size, multiple searches had no noticeable effect on Binary Search, Suffix Arrays and q-grams because the number of comparisons they require is logarithmically proportional to the file size. As indicated in previous experiments, this small number of comparisons led to an efficient performance when searching for a large number of patterns. Likewise, the effect of multiple searches on the FM-index was consistent for any file size. Decompress-then-search and Compressed-Domain Boyer-Moore, however, took longer with larger files because the

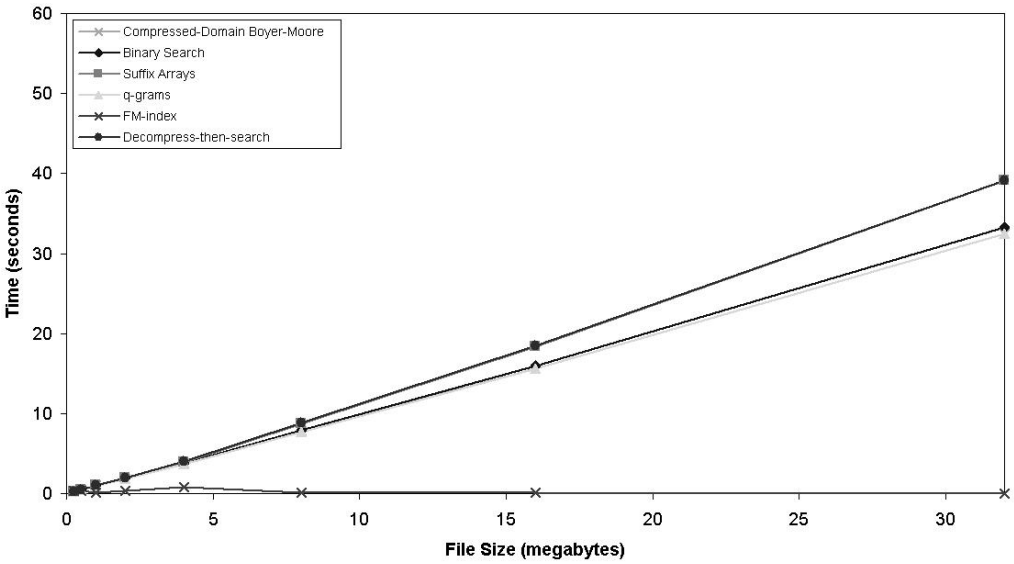


Figure 4.5: Search times for files of various sizes.

number of comparisons they perform is directly proportional to the file size.

Although it is not shown in Figure 4.5, search time increased dramatically when the memory requirement of the search program exceeded the resources of the computer because parts of the memory were continually swapped to the hard drive. For example, *q*-grams requires 576 megabytes of memory for a 64 megabyte file (see Section 4.3). This exceeded the available memory of the computer and required a phenomenal 457 minutes to locate a single pattern. As described in Section 4.3, the FM-index has a very low memory requirement, and is therefore able to avoid this problem.

File Type

The performances of the algorithms on alternatives file types were evaluated and compared to the plain English text file used in earlier experiments. The alternative files were:

- `E.Coli`, which is available from the Canterbury Corpus and contains genetic data that has an alphabet size of four.
- An HTML File, which was obtained by concatenating the HTML files in the `jdk1.4.0` documentation (Sun Microsystems 2002), then truncating the resulting file to an appropriate size.
- A Java Source File, which was obtained by concatenating the Java files, also in the `jdk1.4.0` documentation.

Experiments showed that, in most cases, the performances of the algorithms were insensitive to file type. The exception was the FM-index when searching genetic data. Due to the small alphabet, short patterns have higher frequencies in genetic data than other files. Thus, the inefficiency of the FM-index when locating patterns that appear often was accentuated with the genetic file.

Ferragina & Manzini (2001) performed an experiment with the FM-index using a file containing a concatenation of both HTML and Java files from the documentation of `jdk1.3.0`. They reported that the time required to locate a match in this type of file was significantly higher than for other files they tested. They suggested that this was caused by the marking technique (described in Section 3.5.2), which assumes that the occurrences in the text of a given character are evenly spaced, and speculated that this assumption is probably not valid for HTML and Java files. We ran further experiments using the documentation of other `jdk` versions and found that `jdk1.2.2` and `jdk1.3.1` had performances consistent with those of other

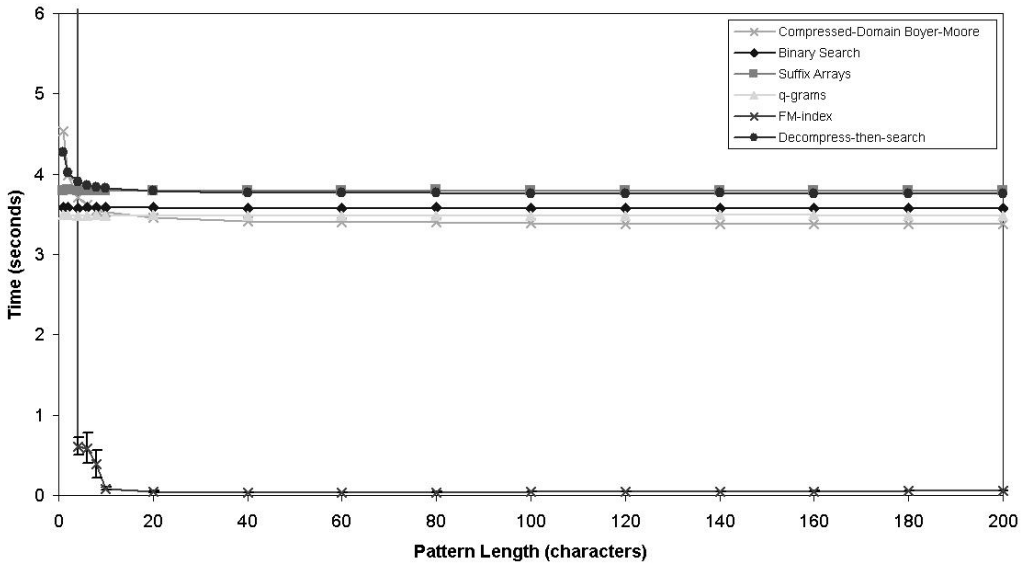


Figure 4.6: Search times for patterns of various lengths.

file types in our previous experiments, while `jdk1.4.0` was only marginally slower. It is likely that their anomalous results were caused by irregularities in the occurrences of the character chosen by the marking scheme, but the problem only exists in specific files and is not indicative of all files containing HTML and Java code.

Pattern Length

The length of the search pattern also had a considerable effect on the efficiency of some of the algorithms. The results of an experiment illustrating this effect are shown in Figure 4.6.

The experiment revealed that Binary Search, Suffix Arrays and q-grams were unaffected by pattern length. Interestingly, for patterns of length one, these three algorithms do not require any comparisons. This is due to the use of the M array, which can be used to identify the first and last positions in the sorted array of any character with only two array lookups (see Section 3.2) and thus, the location of any pattern containing just one character. This is not reflected in the experimental results because there are remarkably few comparisons required even for large patterns.

Search times for decompress-then-search and Compressed-Domain Boyer-Moore were reduced initially as the pattern size increased, but eventually reached an asymptote. Both algorithms are based on Boyer-Moore, which requires $O(\frac{n}{m})$ comparisons to execute a search. Thus, as m increased, the time was reduced. Before searching begins, however, the text must be decompressed, or arrays must be constructed. Therefore, no matter how large the pattern is, search time cannot drop below the time required to complete the initial setup. Furthermore, when searching for patterns of length one, decompress-then-search was more efficient then Compressed-Domain Boyer-Moore. In Section 4.2.1, it was shown that decompress-then-search has benefits when many comparisons are required, which is the case for small values of m.

The efficiency of the FM-index also increased with pattern length, but for a different reason. Search times to locate all occurrences of patterns with lengths one and two are not displayed on the graph but were, on average, 218 seconds and 77 seconds, respectively. Section 4.2.1 showed that the FM-index is highly dependent on the number of occurrences of the search pattern. A small pattern is likely to appear in the text more often then a longer one, so short patterns usually produce a poor performance. For example, a pattern with a length of one is just a single character from the alphabet and is used often throughout the text. A pattern with a length of 100 will form a large part of a sentence and is unlikely to be repeated again in the text.

4.3 Memory Usage

The experiment on file size in Section 4.2.3 showed that the memory requirement of an algorithm is important to its performance because searching becomes incredibly inefficient when it exceeds the resources of the computer.

Excluding the FM-index, the search algorithms require the use of a number of indexing arrays that are created before searching begins and are stored temporarily in memory. The size of many of these arrays is proportional to the length of the text. The smaller arrays (K and M), of size $O(|\sum|)$, do not require much memory and will be ignored here.

The arrays that are actually used to perform searches will be referred to as *search arrays*. Others, which are only used to aid the construction of the search arrays, will be referred to as *construction arrays*. One further category is *overwritten arrays* (Section 3.6.1), which is actually a subset of construction arrays. When creating a search array from an overwritten array, the elements of the overwritten array are accessed in the same order as the search array is created. Thus, it is possible to write values for the search array in place of the entries in the overwritten array. As well as saving memory, this provides a significant improvement to the search time due to a reduction in cache misses, as described in Section 4.5.1. Furthermore, in Section 4.2.3 we found that the memory usage is proportional to file size, and that if the memory resources of the computer are exceeded, searching becomes impractically slow. Thus, with the reduced memory requirement, it is possible to efficiently search larger files that would have otherwise surpassed the memory limit.

The arrays used by each algorithm are listed in Table 4.3 and are separated into the categories that describe their purpose. The different purposes lead to two measurements of memory usage — the search requirement, which includes only the search arrays; and the the maximum requirement, which specifies the largest amount of memory used concurrently and usually involves most of the construction arrays and search arrays. Although the maximum usage dictates the overall search performance, as long as the search requirement is below the resource limit, multiple searches may be performed efficiently after the necessary search arrays are available.

The values for both types of memory requirements are also given in Table 4.3 and assume that arrays storing characters (F, L and T) use 1-byte entries and the remaining arrays store 4-byte integers. For decompress-then-search and Binary Search, the maximum requirement consists of all of arrays that they use. The other online algorithms can achieve an optimal maximum requirement through the use of overwritten arrays and by freeing memory as soon as it is no longer needed. For q-grams and Compressed-Domain Boyer-Moore, this process involves freeing L after K has been created. Compressed-Domain Boyer-Moore must also create Hr and then free W before beginning the construction of F. Furthermore, C and W must be implemented as overwritten arrays for Suffix Arrays and q-grams, respectively.

Thus, the online algorithms have a maximum memory requirement between $6n$ and $9n$ bytes and a search requirement between n and $9n$ bytes. These requirements are viable for small files; for example, 36 megabytes is needed at most, to search the 4 megabyte file used in many of the experiments in this chapter. A larger 128 megabyte file, however, would need more than a gigabyte of memory in the worst cases. In contrast, the offline approach of the FM-index uses just one kilobyte regardless of file size because it is able to search with only one bucket decompressed at a time. The remaining data is stored on disk until it is needed.

Section 4.2.2 described an application that does not need to locate the position of matches, but instead counts the number of occurrences of a pattern. In this situation, Binary Search was able to operate without the I array, reducing both the maximum memory usage and memory required for searching to $5n$ bytes.

4.4 Array Construction

Previous experiments in this chapter have shown that the total operation time of an algorithm is highly dependent on the time required to create the indexing arrays used by that algorithm. In fact, for Binary Search, Suffix Arrays and q-grams, the total operation time is almost entirely due to array construction. The FM-index is an exception because it constructs the necessary indexes during the compression stage to save time when searching, and will therefore be ignored in this section.

Algorithm	Search Arrays	Construction Arrays	Memory for Searching	Maximum Memory
Decompress-then-search	T	L, W	n	$6n$
Compressed-Domain BM	F, Hr	L, W	$5n$	$8n$
Binary Search	I, L, W		$9n$	$9n$
Suffix Arrays	I, T	C, L	$5n$	$6n$
q-grams	F, Hr, I	L, W	$9n$	$9n$
FM-index	1 Bucket		1 KB	1 KB

Table 4.3: Memory requirements of the search algorithms.

Figure 4.7 shows the arrays used by each algorithm and indicates the time required to construct them from the compressed `bible.txt` file. The average time to search for one pattern is indicated in grey, although for some algorithms this search time was insignificant and is not visible on the diagram. All indicated times increased linearly as file size increased, with the ratios between array construction times remaining constant for all sizes.

All algorithms require L, the BWT permutation of the file that was compressed. The construction of L involves reading the compressed file from disk, then reversing the arithmetic coding, run length coding and move-to-front coding that was originally used to compress L. Each algorithm also uses K and M, both of which can be created relatively quickly in comparison to other arrays. They are primarily used in the construction of W or C and have therefore been included in the cost of those arrays.

Usage of the remaining arrays varies, and is the cause of the difference in performances of the algorithms. In particular, decompress-then-search and Suffix Arrays both use T. While producing T, however, Suffix Arrays simultaneously creates I. This takes additional time but makes searching considerably more efficient (see Section 4.2.1) so that the first search, and subsequent searches, were performed almost instantly. In contrast, the first search by decompress-then-search took almost the same amount of time as the construction of I in Suffix Arrays, so that the time to search for a single pattern was similar for both algorithms. With the availability of I, however, multiple pattern searches were more efficient with Suffix Arrays. A similar situation occurs between Compressed-Domain Boyer-Moore and q-grams. Both use Hr, but q-grams constructs I at the same time to increase search performance later. For reasons discussed in Section 4.5.1, the cost of creating Hr is lower than that of T which means that, even though they also require F, Compressed-Domain Boyer-Moore and q-grams were more efficient for a single search than decompress-then-search and Suffix Arrays.

The last algorithm is Binary Search. It also constructs I to make searching more efficient, but performs comparisons while searching using W instead of the Hr or T arrays used by other algorithms. Because the most efficient approach for creating I involves overwriting W, it is is necessary for Binary Search to create a second copy of W (see Section 3.6.1). This additional time to construct the second copy made it slower than the algorithms that use Hr, but it was still more efficient than those that use T.

4.5 Evaluation of Algorithm Improvements

In Section 3.6, we proposed some modifications to improve the performance of four of the search algorithms. These modifications are evaluated in the following sections, with Section 4.5.1 exploring the effect of overwritten arrays on Binary Search, Suffix Arrays and q-grams, and Section 4.5.2 assessing the performance of the Modified FM-index.

4.5.1 Overwritten Arrays

Overwritten arrays were introduced in Section 3.6.1 as a mechanism for reducing the maximum memory requirement of Suffix Arrays and q-grams by writing the I array over C or W. Table 4.3 shows that, when using overwritten arrays, their maximum memory requirements are $6n$ and $9n$, respectively. Without overwriting however, the additional storage of I, which requires $4n$ bytes, produces a total requirement of

File	Binary Search		Suffix Arrays		q-grams	
Size	Original	Optimised	Original	Optimised	Original	Optimised
1	1.14	1.00	1.21	1.01	1.17	0.98
2	2.26	1.95	2.45	1.96	2.32	1.90
4	4.59	3.88	5.12	4.03	4.71	3.81
8	9.63	8.04	11.21	8.74	9.91	7.91
16	19.9	15.95	23.63	17.97	20.25	15.61
32	44.78	34.04	53.10	38.23	46.42	33.38

Table 4.4: Search times for the original and optimised versions of Binary Search, Suffix Arrays and q-grams. File size is in megabytes and time is in seconds.

$10n$ bytes for Suffix Arrays and $13n$ bytes for q-grams. Thus, overwritten arrays provide a saving of 40% and 31%, respectively.

It was also noted that overwritten arrays increased the search performance of the algorithms. Furthermore, it was shown how the concept can be included in Binary Search to improve its performance as well. Table 4.4 illustrates the effect of the optimisation on these three algorithms when searching the files used in Section 4.2.3 for the file size experiment. For the results shown, there was an average improvement of 17% for Binary Search, 22% for Suffix Arrays and 21% for q-grams, with the improvements increasing as file size increased.

To understand to reason for this improvement, it is first useful to consider the cause of the variation in construction times of the different arrays. Although all arrays are constructed in $O(n)$ time, the actual times differ considerably. This is largely due to the order in which the arrays are created. When constructing arrays in a sequential manner (C, F, Hr and T), that is, starting with the first element and progressing through the following elements in order (or, in the case of T, in the reverse order from the last element), blocks of the array are read into the CPU cache and can be accessed and modified from there. Creation by means of a non-sequential manner (I and W) results in many cache misses, affecting performance considerably because the memory must be accessed for almost every element produced. For instance, the code that produces Hr and I (shown combined in Algorithm 2.4) is almost identical, but, without overwriting optimisation, execution took 0.80 and 1.63 seconds respectively. There is also considerable variation among sequentially created arrays. For instance, the 0.80 second creation time of Hr was slower than that of F or C, which took 0.08 and 0.20 seconds respectively. Although Hr is created sequentially, its values are calculated from the entries in W, and those entries are accessed non-sequentially, again resulting in a significant number of cache misses. Even slower was the 1.37 second construction time of T which is created by non-sequentially accessing two arrays: L and C.

Thus, a substantial gain is available by avoiding non-sequential use of arrays. This leads to the concept of overwritten arrays for constructing I, which, using its original algorithm, is created non-sequentially by random access to the information in W. Section 3.6.1 showed that it is possible to write the elements of I over those of W, meaning that only one array is accessed during the construction of I. Even though that array is used non-sequentially, the cache misses will be reduced substantially, thus decreasing the amount of data that must be read from memory. For the text `bible.txt`, this reduced the construction time of I from 1.37 seconds to 0.87 seconds, and thus, provides a significant improvement to search performance when incorporated in the Binary Search, Suffix Arrays and q-grams algorithms.

4.5.2 Modified FM-index

The Modified FM-index (Section 3.6.2) was designed to increase the speed of searching through a reduction in the time taken to locate the position of matches. In Section 4.2.1, we reported that the FM-index took 269 seconds to locate all 61,509 appearances of ‘an’ in `bible.txt`. The Modified FM-index achieves a significant improvement and required just 118 seconds to perform the same task. For patterns with lower numbers of occurrences, however, the difference in performance was not as great. Further experiments revealed that when the pattern occurred less than one thousand times, the Modified FM-index was actually

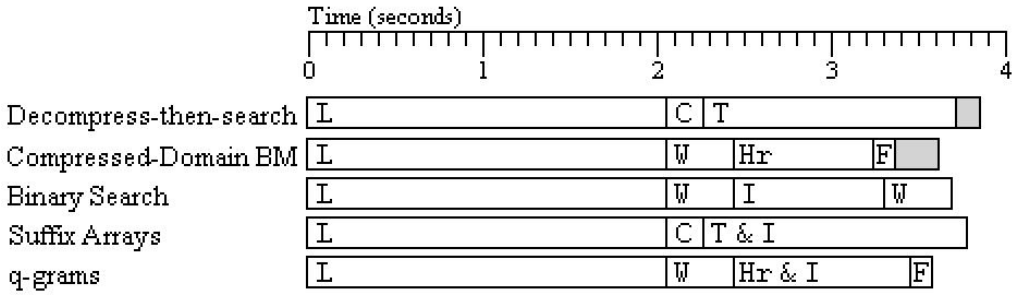


Figure 4.7: Time taken by each algorithm to construct the required indexing arrays. Regions shaded grey indicate the search time for a single pattern.

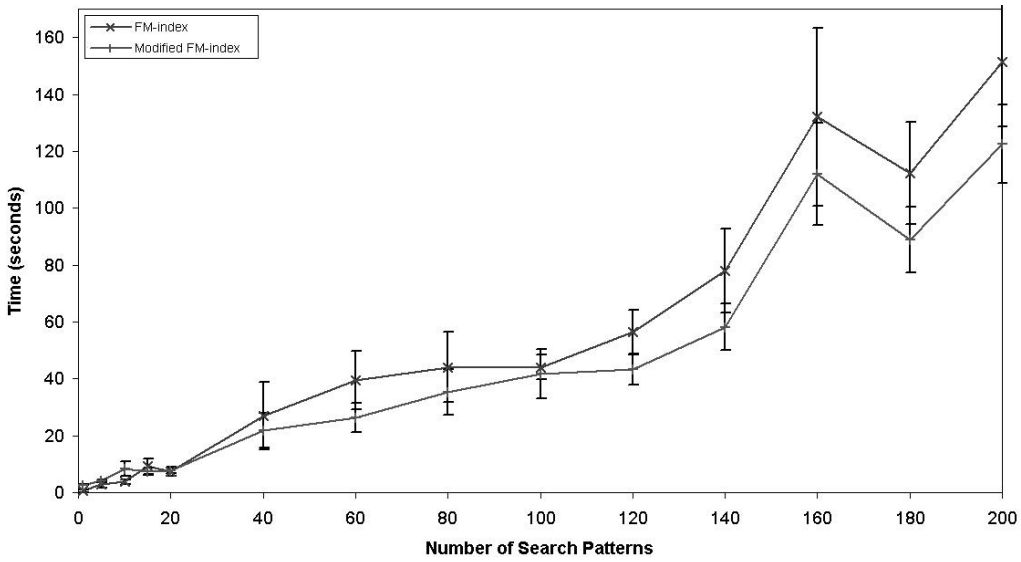


Figure 4.8: Search times with multiple patterns for the original FM-index and Modified FM-index.

slower. This is due to the overhead of the initial work performed by the Modified FM-index to read the data into memory and decompress it before searching begins. Furthermore, it is likely that most searches would require fewer than than one thousand matches to be located, and we therefore anticipate that the modification will decrease performance for general use.

Figure 4.8 compares the search performance of the original and modified versions based on the number of searches performed. The data shown for the original FM-index is same as that in Figure 4.1. The graph reveals that, while there was some improvement when searching for a large number of patterns with the Modified FM-index, it was not substantial. Furthermore, like the original version, performance of the Modified FM-index varied greatly because of its dependence on the number of occurrences of the search pattern, and thus searching was still unacceptably slow for patterns that appeared often.

Additionally, for small numbers of patterns, the Modified FM-index was slower than the original version. Again, this is due to the overhead of reading all the data into memory when only a small number of occurrences are being located. It is worth noting however, that for a single pattern, a search took 2.60 seconds on average, which is still 25% faster than q-grams, the most effiecient online algorithm.

In Section 3.6.2, we also reported that the Modified FM-index achieves better compression than then the original FM-index because it is unnecessary to store a bucket directory, and because of the ability to compress L and the bucket headers with more effective techniques. For the files listed in Table 4.1, the Modified FM-index has an average compression ratio of 3.12 bits per character. While this is a 0.53 bits

per character improvement over the original approach, it is still substantially worse than that of `bzip2` and `bsmp` due to the indexing information that must be stored.

Finally, we consider the memory requirement of the Modified FM-index. The entire L array must be kept in memory, requiring n bytes. It is also necessary to store an array containing the information for the marked rows. With 2% of the rows marked, there are $0.02n$ entries in this array, each stored in a 4-byte integer, and thus, contributing $0.08n$ bytes to the memory usage. The other structures that are keep in memory are the bucket and superbucket headers. Each header has $|\sum|$ entries and, assuming the alphabet has 256 character, requires 1024 bytes. There will be $\frac{n}{1024}$ bucket headers and $\frac{n}{16384}$ superbucket headers, giving a total of $1.0625n$ bytes for all headers. Thus, the overall memory usage for the Modified FM-index is $2.1425n$ bytes for both the maximum and searching requirements. When compared to the requirements of other algorithms, shown in Table 4.3, we can see that it is considerably larger than the constant one kilobyte used by the original FM-index, but is almost one third of the maximum requirement of the best online algorithms. Furthermore, in situations where search time is reduced, if there is enough memory available, it is worthwhile exploiting the modification to efficiently utilise resources.

Chapter 5

Conclusion

We have provided an evaluation of five approaches to searching BWT compressed files: Compressed-Domain Boyer-Moore, Binary Search, Suffix Arrays, q-grams and the FM-index. A qualitative summary of their characteristics is given in Table 5.1. Ratings for the search performance of the FM-index are based on its average performance for the given situation.

The FM-index is an offline approach, which means that it creates indexing information that it stores with the compressed file at compression time. This information is used to increase search performance, and allowed the FM-index to achieve the fastest results, on average, when searching for a small number of patterns. Its use of the indexes to locate the positions in which the matches occur in the text is inefficient, however, and for a pattern that appeared in the text often, or for a large number of searches, which also involved locating a large number of matches, it was the slowest approach.

The remaining algorithms can be classified as online because they store only the compressed data. This approach requires less storage space than the offline approach, and in fact, provides a compression ratio similar to that of production-quality compression programs. To perform a search, however, it is necessary to create indexing structures, in the form of temporary arrays, in memory. After the arrays have been created, Binary Search, Suffix Arrays and q-grams are able to perform many searches almost instantly using a binary search technique that requires only $O(m \log n)$ comparisons per search. The slowest aspect of these algorithms is therefore the construction of the indexing arrays. In Section 3.6.1, we introduced a technique that reduced this construction time for the three algorithms by around 20%. With this improvement, q-grams was always the fastest online algorithm. For a single search, Compressed-Domain Boyer-Moore provided similar results; however, unlike the other three online algorithms, its performance deteriorated significantly as the number of searches increased.

The biggest disadvantage of the online algorithms is their memory usage. In Section 3.6.1, we provided an approach for creating the indexing arrays that reduces the memory requirements of Suffix Arrays and q-grams by 40% and 31%, respectively. However, q-grams, which consistently produced the fastest search times, still requires $9n$ bytes of memory, where n is the size of the uncompressed file. For a small decrease in speed, the Suffix Arrays algorithm, which requires only $6n$ bytes, provides a useful alternative. Even this amount of memory is excessive for large files, however, and if the memory requirement exceeds the available resources of the computer, the algorithms become impractically slow. In contrast, the FM-index accesses the necessary indexing information from disk only when it is needed, and therefore uses remarkably small amounts of memory, even for large files.

Finally, we note that the FM-index is particularly suited to applications that only require the appearances of a pattern to be counted, rather than also locating the positions in which they occur. Because it avoids the inefficient location process, the FM-index is able to return counts almost instantly, regardless of the number of patterns that the counts are obtained for. Binary Search also works better for this style of application, because it requires fewer indexing arrays to be constructed. Creating fewer arrays saves time, so its performance surpassed that of q-grams, but it was still significantly slower than the FM-index.

Algorithm	Compression	Memory Usage	Single Search	Multiple Searches	Single Count
Compressed-Domain BM	high	high	moderate	slow	moderate
Binary Search	high	high	moderate	fast	moderate
Suffix Arrays	high	high	moderate	fast	moderate
q-grams	high	high	moderate	fast	moderate
FM-index	moderate	low	fast	slow	fast
decompress-then-search	high	high	moderate	slow	moderate

Table 5.1: Summary of algorithm characteristics.

5.1 Future Work

Currently, the memory usage of the online algorithms is dependent on the size of the input file. If the file exceeds a particular size, the memory requirement of the search programs can exceed the resources of the computer and searching becomes extraordinarily inefficient. The problem could be avoided, however, by introducing a blocking technique similar to that of `bzip2` (Seward 2002), where the input file is segmented into blocks, and each block is permuted and compressed independent of the others. Thus, when searching, it would be necessary to bring only one block into memory at a time so that memory usage is dependent on the block size instead of file size. Furthermore, Seward (2002) has shown that there is little advantage, in terms of compression ratio, to using block sizes larger than 900 kilobytes. Searching a blocked file with Binary Search, Suffix Arrays or q-grams, however, requires individual searches to be applied to each block separately, and it also would be necessary to consider matches that cross block boundaries.

Our evaluations have only considered exact pattern matching approaches where a matching substring must be identical to the search pattern. A common variation is approximate pattern matching. A k-approximate match occurs when the edit distance between the search pattern and a substring in the text is less than k. The edit distance is calculated from the number of character insertions, deletions and substitutions required to change one string to the other (Navarro 2001). Adjeroh et al. (2002) describe a technique that allows the k-approximate problem to be solved with q-grams in $O(n+|\sum|\log|\sum|+\frac{m^2}{k}\log\frac{n}{|\sum|}+\alpha k)$ time on average, with $\alpha \leq n$. Due to the similarities between q-grams, Binary Search and Suffix Arrays that were identified in Chapter 3, it is likely that this approximate matching technique of q-grams could be adapted for the two additional algorithms. Once developed, an evaluation of these pattern matching variants would also be useful.

Acknowledgements

I would like to thank Tim Bell for his guidance throughout the year, Paolo Ferragina for his help in understanding the finer details of the FM-index and Stacey Mickelbart for providing technical writing assistance. Thanks also go to my family for the support they have given me, and to the 2002 COSC fourth year students, who made this year enjoyable.

Bibliography

Adjeroh, D., Mukherjee, A., Bell, T., Powell, M. & Zhang, N. (2002), 'Pattern matching in BWT-compressed text', *Proceedings, Data Compression Conference* p. 445.

Amir, A., Benson, G. & Farach, M. (1996), 'Let sleeping files lie: Pattern matching in Z-compressed files', *Journal of Computer and System Sciences* **52**, 299–307.

Arnold, R. & Bell, T. C. (1997), A corpus for the evaluation of lossless compression algorithms, *in* 'Designs, Codes and Cryptography', pp. 201–210.

Bell, T., Adjeroh, D. & Mukherjee, A. (2001), Pattern matching in compressed text and images, Technical report, Department of Computer Science, University of Canterbury.

Bell, T. & Powell, M. (2002), 'The Canterbury Corpus', `http://corpus.canterbury.ac.nz`.

Bell, T., Powell, M., Mukherjee, A. & Adjeroh, D. (2002), 'Searching BWT compressed text with the Boyer-Moore algorithm and binary search', *Proceedings, Data Compression Conference* pp. 112–121.

Bentley, J. L., Sleator, D. D., Tarjan, R. E. & Wei, V. (1986), 'A locally adaptive data compression scheme', *Communications of the ACM* **29**(4), 320–330.

Boyer, R. & Moore, J. (1977), 'A fast string searching algorithm', *Communications of the ACM* **20**(10), 762–772.

Bunke, H. & Csirik, J. (1993), 'An algorithm for matching run-length coded strings', *Computing* **50**, 297–314.

Bunke, H. & Csirik, J. (1995), 'An improved algorithm for computing the edit distance of run-length coded strings', *Information Processing Letters* **54**, 93–96.

Burrows, M. & Wheeler, D. (1994), A block-sorting lossless data compression algorithm, Technical report, Digital Equipment Corporation, Palo Alto, California.

Cleary, J. & Witten, I. (1984), 'Data compression using adaptive coding and partial string matching', *IEEE Transactions on Communications* **COM-32**, 396–402.

Farach, M. & Thorup, M. (1998), 'String matching in Lempel-Ziv compressed strings', *Algorithmica* **20**, 388–404.

Ferragina, P. & Manzini, G. (2000), 'Opportunistic data structures with applications', *Proceedings, 41st IEEE Symposium on Foundations of Computer Science, FOCS 2000* pp. 390–398.

Ferragina, P. & Manzini, G. (2001), 'An experimental study of an opportunistic index', *Proceedings, 12th ACM-SIAM Symposium on Discrete Algorithms, SODA 2001* pp. 269–278.

Grossi, R. & Vitter, J. (2000), 'Compressed suffix arrays and suffix trees with applications to text indexing and string matching', *Proceedings, 32nd ACM Symposium on Theory of Computing* pp. 397–406.

Gusfield, D. (1997), *Algorithms on strings, trees, and sequences: computer science and computational biology*, Cambridge University Press.

Manber, U. & Myers, G. (1993), 'Suffix arrays: A new method for on-line string searches', *SIAM Journal of Computing* **22**(5), 935–948.

Moura, E. S., Navarro, G., Ziviani, N. & Baeza-Yates, R. (2000), 'Fast and flexible word searching on compressed text', *ACM Transactions on Information Systems* **18**(2), 113–139.

Navarro, G. (2001), 'A guided tour of approximate string matching', *ACM Computing Surveys* **33**(1), 31–88.

Navarro, G. & Raffinot, M. (1999), 'A general practical approach to pattern matching over Ziv-Lempel compressed text', *Proceedings, Combinatorial Pattern Matching, LNCS 1645* pp. 14–36.

Powell, M. (2001), Compressed-Domain Pattern Matching with the Burrows-Wheeler Transform, Honours report, Department of Computer Science, University of Canterbury.

Sadakane, K. (2000*a*), 'Compressed text databases with efficient query algorithms based on the compressed suffix array', *Proceedings, ISAAC 2000* pp. 410–421.

Sadakane, K. (2000*b*), Unifying Text Search and Compression – Suffix Sorting, Block Sorting and Suffix Arrays, PhD thesis, Graduate School of Information Science, University of Tokyo.

Sadakane, K. & Imai, H. (1999), 'A cooperative distributed text database management method unifying search and compression based on the Burrows-Wheeler Transform', *Proceedings, Advances in Database Technology, LNCS 1552* pp. 434–445.

Seward, J. (2002), 'The bzip2 and libbzip2 official home page', `http://sources.redhat.com/bzip2/index.html`.

Shibata, Y., Kida, T., Fukamachi, S., Takeda, M., Shinohara, A., Shinohara, T. & Arikawa, S. (2001), 'Speeding up pattern matching by text compression', *Transactions of Information Processing Society of Japan* **42**(3), 370–384.

Shibata, Y., Takeda, M., Shinohara, A. & Arikawa, S. (1999), 'Pattern matching in text compressed by using antidictionaries', *Proceedings, Combinatorial Pattern Matching, LNCS 1645* pp. 37–49.

Sun Microsystems (2002), 'Java Development Kit', `http://java.sun.com/j2se/index.html`.

TREC (2002), 'Official webpage for TREC – Text REtrieval Conference series. `http://trec.nist.gov`'.

Weiner, P. (1973), 'Linear pattern matching algorithm', *Proceedings, 14th IEEE Symposium on Switching and Automata Theory* **21**, 1–11.

Wheeler, D. (1997), 'Upgrading bred with multiple tables', `ftp://ftp.cl.cam.ac.uk/users/djw3/bred3.ps`.

Witten, I. H., Moffat, A. & Bell, T. C. (1999), *Managing Gigabytes: Compressing and Indexing Documents and Images*, second edition edn, Morgan Kaufman.

Ziviani, N., Moura, E. S., Navarro, G. & Baeza-Yates, R. (2000), 'Compression: A key for next generation text retrieval systems', *IEEE Computer* **33**(11), 37–44.