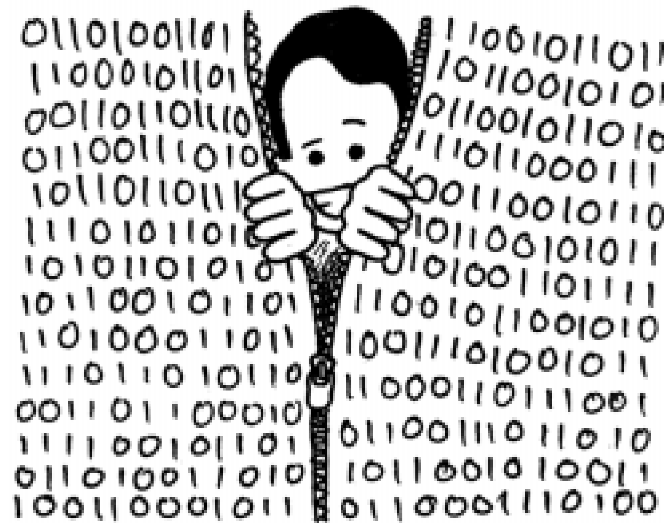
Compressed-Domain Pattern Matching with the Burrows-Wheeler Transform



Matt Powell

Honours Project Report, 2001

Abstract

This report investigates two approaches for online pattern-matching in files compressed with the Burrows-Wheeler transform (Burrows & Wheeler 1994). The first is based on the Boyer-Moore pattern matching algorithm (Boyer & Moore 1977), and the second is based on binary search. The new methods use the special structure of the Burrows-Wheeler transform to achieve efficient, robust pattern matching algorithms that can be used on files that have been only partly decompressed. Experimental results show that both new methods perform considerably faster than a decompress-and-search approach for most applications, with binary search being faster than Boyer-Moore at the expense of increased memory usage. The binary search in particular is strongly related to efficient indexing strategies such as binary trees, and suggests a number of new applications of the Burrows-Wheeler transform in data storage and retrieval.

Contents

1 Introduction **7**

2 The Burrows-Wheeler Transform **11**
2.1 Implementing the Burrows-Wheeler Transform 13
2.2 Auxiliary arrays 13
2.3 Concluding remarks on the BWT 14

3 A Boyer-Moore-based approach **15**
3.1 Shift heuristics 15
3.2 Extensions to the basic shift heuristics 16
3.3 The compressed-domain Boyer-Moore algorithm 18

4 Binary Searching **19**

5 Experimental Results **25**
5.1 Search performance 26
5.2 Memory usage 27
5.3 Number of comparisons 28
5.4 Unsuccessful searches 29

6 Conclusion **31**
6.1 Further work 31
6.1.1 Moving further into the compressed domain 32
6.1.2 Dealing with blocked files 32
6.1.3 Approximate matching 32
6.1.4 Relationship to other data structures 32
6.2 Concluding remarks 33

A Searching with q-grams **35**

B Evaluating Lossless Compression **43**

Chapter 1

Introduction

“The greatest masterpiece in literature
is only a dictionary out of order.”
—*Jean Cocteau*

Computer science is about information. This information must somehow be stored—the more efficiently the better—and then presumably retrieved at some later date.

Information storage and retrieval are not only integral to computer science, but to society in general. The amount of information available in today’s world surpasses ‘overwhelming’, and the task of finding a desired piece of data is becoming more and more like finding a needle in a whole field of haystacks. For instance, the internet search engine Google now indexes 1,610,476,000 pages[1]. It has also been estimated that the amount of information in the world doubles every twenty months (Piatetsky & Frawley 1991). Information is getting out of control.

The pattern matching problem

The problem of pattern matching, or finding occurrences of a pattern in a text or collection of texts, is one of the oldest and most thoroughly-researched problems in Computer Science. Many techniques exist for building efficient indexes to facilitate fast searching. These include binary search trees, which allow logarithmic-time access and retrieval, and hash tables, which give almost constant time performance.

There are several problems with an indexed approach to pattern matching. Perhaps the major disadvantage is that an index places restrictions on the types of queries that can be performed. For example, if the index is built using only whole words, then partial word queries or queries that cross word boundaries (such as a search for the word *pattern* immediately followed by the word *matching*) may not be possible. The less restrictive the index, the more information it has to contain—and this information must itself be stored somewhere. In the extreme case, the index could even be larger than the text.

The other main problem with indexing is that it takes much more work to construct the index than to look for a single piece of information. This makes static indexing (constructing an index that cannot be modified without discarding the old index and re-indexing the entire text again) essentially worthless for situations where data is constantly being updated, since the index must be kept current, and even worse for ‘one-off’ queries, since the computational effort required to construct the index far outweighs the cost of performing a single search for a given pattern.

Most databases or collections that allow their contents to be updated use dynamic indexing systems, where the index can be updated without starting from scratch. This approach offers all the retrieval benefits of a static index at a fraction of the maintenance costs. However, there is still a trade-off between the size of the index and its potential power for searching.

[1] `http://www.google.com`, 25 October, 2001

Search strategies that involve an index are called ‘offline’ approaches, since the work of constructing and maintaining the index is kept separate from the user’s interaction with the system via a query interface. Often, offline systems need not even access the original text, since all that needs to be done is to return some information about *where* the pattern occurs (for example, a list of page numbers), and this information will be stored in the index.

The opposite of offline pattern matching is, naturally, ‘online’ pattern matching, where no other information than the text itself is stored, and all the work of pattern matching is done at query time. Naturally, online algorithms will not perform as quickly as, say, a static index built on a hash table, since almost no documents are so nicely structured, but what they lack in speed they make up for in power. Online algorithms are capable of approximate matching (with wildcards or regular expressions), and require no additional storage space for an index.

With no available information about the structure of the file, an online algorithm must traverse the entire text in order to search for all occurrences of a particular pattern. For this reason, online approaches are heavily dependent on the size of the text being searched. The simplest ‘brute force’ algorithm, a linear search which aligns the pattern at each position in the text and checks for a match, has time complexity $\mathcal{O}(mn)$, where m is the length of the pattern and n the length of the text. Other, more sophisticated algorithms such as the Knuth-Morris-Pratt algorithm (Knuth, Morris & Pratt 1977) and the Boyer-Moore algorithm (Boyer & Moore 1977), are able to identify portions of the text that cannot possibly contain a match, and can consequently be ‘skipped’. In this way, the Knuth-Morris-Pratt algorithm achieves a worst case time complexity of $\mathcal{O}(m+n)$, and the Boyer-Moore algorithm, although in the worst case it degenerates to the brute force algorithm, has a best case time complexity of $\mathcal{O}(\frac{n}{m})$. In general, the Boyer-Moore algorithm manages to achieve so-called ‘sub-linear’ complexity; that is, for most files, it solves the pattern matching problem in fewer than n comparisons. This makes the Boyer-Moore algorithm the most efficient online pattern matching algorithm in general use.

Searching compressed files

Compression is the art of reducing the size of a file by removing redundancy in its structure. In general, compression works by assigning shorter bit patterns to symbols that are more likely to occur.

With compressed files becoming more commonplace, the problem of how to search them is becoming increasingly important. There are two options to consider when deciding how to approach compressed pattern matching. The first is a ‘decompress-then-search’ approach, where the compressed file is first decompressed, and then a traditional pattern matching algorithm applied. This approach has the advantage of simplicity, but brings with it tremendous overheads, in terms of both computation time and storage requirements. Firstly, the entire file must be decompressed—often a lengthy process, especially when considering files several megabytes in size. Additionally, the decompressed file must be stored somewhere once decompressed, so that pattern matching may occur.

The second alternative is to search the compressed file without decompressing it, or at least only partially decompress it. This approach is known as *compressed-domain* pattern matching, and offers several enticing advantages. The file is smaller, so a pattern matching algorithm should take less time to search the full text. It also avoids the work that would be needed to completely decompress the file.

The main difficulty in compressed-domain pattern matching is that the compression process may have removed a great deal of the structure of the file. The more structure removed, the better the compression likely to be achieved. There is therefore a subtly-balanced tension between obtaining good compression and leaving enough ‘hints’ to allow pattern-matching to proceed. It would appear that these two goals are in constant opposition, but in fact compression is very closely related to pattern matching, in that many compression systems use some sort of pattern matching technique to find repetitions in the input, which can be exploited to give better compression. The effect of this is that these patterns are coded in a special manner, which, if suitably represented,

may actually aid in pattern matching.

Many techniques exist for compressed-domain pattern matching. So-called 'fully-compressed pattern matching'—compressing the pattern, then searching the compressed text for the compressed pattern—is one popular method, but will not work in situations where a given substring could have a number of different representations depending on context. This happens when boundaries in the compressed text do not correspond to those in the original text, such as in arithmetic coding, or where the input file is coded adaptively.

A recent survey by Bell, Adjeroh & Mukherjee (2001) outlines several techniques for online compressed-domain pattern matching in both text and images. Many of these techniques are based on the LZ family of compression systems (Ziv & Lempel 1977, Ziv & Lempel 1978), but others include methods for Huffman-coded text and run-length encoding. The authors of this survey noted the potential of a relatively new method called the Burrows-Wheeler transform (Burrows & Wheeler 1994), which is used in some compression systems as a preprocessing step to achieve some of the best compression available. The Burrows-Wheeler transform is explained in detail in Chapter 2.

The authors of the survey noted that very little work had been done with the Burrows-Wheeler transform, although some research has been undertaken in the area of offline pattern matching (Ferragina & Manzini 2000, Ferragina & Manzini 2001, Sadakane & Imai 1999, Sadakane 2000*b*). This is hardly surprising, given that the BWT itself is a recent development in the field of text compression. Additionally, the idea of using the BWT for pattern matching seems somewhat counterintuitive, since it works by permuting the text into a seemingly random order. However, the ouput from the Burrows-Wheeler transform actually contains a lexically sorted version of the text—the perfect index.

Chapter 3 describes the adaptation of the Boyer-Moore algorithm to handle BWT-encoded text efficiently, while Chapter 4 describes a new technique that exploits the sort process used in the BWT to give an extremely fast pattern matching algorithm with all the speed benefits of a static index, but very little extra cost in terms of space, and the convenience of an online system.

Approximate pattern matching with the BWT

The structure contained in the Burrows-Wheeler transform can also be used to aid in approximate pattern matching—that is, finding substrings of the text that are *similar* to the pattern, but may not match exactly. One method, 'k-approximate' matching, involves computing the *edit distance* between two strings as the number of basic operations (insertions, deletions and substitutions) needed to transform one string into the other. k-approximate matching involves finding all the substrings of the text for which the edit distance from the pattern is less than some value of k. Approximate pattern matching is beyond the scope of this project, but the author has been involved in some preliminary work in this area. Appendix A contains the current draft of a paper outlining an approximate pattern matching method based on the Burrows-Wheeler transform and the idea of q-grams—substrings of length q of both the pattern and the text. By taking the intersection of both sets of q-grams, areas of possible approximate matches can be identified and examined.

Evaluating lossless compression methods

The question "what is the best compression method?" can only be answered with another question: "why do you ask?" There is no single 'best' compression method, simply because there are so many ways of measuring their performance. Some are better in terms of speed, while others may take slightly longer, but achieve better compression. Some methods compress and decompress at approximately the same speed, while others take a long time to compress to allow faster decompression.

The Canterbury Corpus (Arnold & Bell 1997) is a set of eleven 'typical' files, collected for the purpose of evaluating lossless compression methods. The original focus of this report was the exploration of suitable performance metrics for lossless compression methods, using the Canterbury Corpus as a benchmark. This research was sidelined and eventually abandoned in order to explore the burgeoning area of compressed-domain pattern matching. Some initial results are discussed in Appendix B.

Notation

Throughout this paper we will refer to the pattern matching problem in terms of searching for a pattern P of length m in a text T of length n. The input alphabet will be referred to as Σ; similarly, $|\Sigma|$ will denote the length of the alphabet. Other variables will be defined as they are used.

In general, arrays will be represented by uppercase letters, while lowercase letters will denote variables of scalar type. All arrays will begin at one unless otherwise noted, and will be indexed with square brackets.

Acknowledgements

The author is grateful to Dr Tim Bell for inspiration, advice, support, encouragement, and, above all, patience. Professor Amar Mukherjee of the University of Central Florida and Assistant Professor Don Adjeroh of West Virginia University have made many valuable contributions to this research, and resolved a lot of very confusing issues. Thanks are also due to Associate Professor Krzysztof Pawlikowski for his help with the performance measurement experiments undertaken in Appendix B, and to Jane McKenzie for proofreading initial drafts of this report.

Chapter 2

The Burrows-Wheeler Transform

The Burrows-Wheeler transform (BWT), also called 'block sorting' (Burrows & Wheeler 1994), is a technique for making text files more amenable to compression by permuting the input file so that characters that appear in similar contexts are clustered together in the output.

Figure 2.1 shows a portion of the `book1` text (from the Calgary Corpus) that has been permuted with the Burrows-Wheeler transform. The context is characters followed by the letters '`aking`'; notice how only a few characters (`e`, `m` and `t`) appear in this context. This makes the output of the Burrows-Wheeler transform particularly suitable for move-to-front coding (Bentley, Sleator, Tarjan & Wei 1986), which assigns higher probablilities to more recently-seen symbols.

The transformation is performed by sorting all the characters in the input text, using their context as the sort key. Here the 'context' can be either the characters immediately before or immediately after the character being sorted; although Witten, Moffatt & Bell (1999) use the preceding characters as the sort key to illustrate the parallel between the BWT and other context-based compression models, most other descriptions of the BWT seem to use the substring *after* the character in question. This latter method has several advantages for the applications in this report (not the least of which is perspicuity), and so for the purposes of this discussion, the term 'context' refers to as many characters after the character being sorted as are needed to resolve any question of ordering. In fact, the two approaches are practically equivalent, and result in very similar compression performance in most situations (Fenwick 1996*a*).

The transformed text is simply the characters in order of their sorted contexts. Figure 2.2 shows how the Burrows-Wheeler transform would be performed on the shorter text '`mississippi`'. First, all the cyclic rotations of the text are produced. Next, the characters are sorted into the order of their contexts. Finally, the permuted characters are transmitted. For example, the permuted text of the `book1` text would contain the sequence `ememttemmtmmeeeteeee` (see Figure 2.1), and the complete permuted text for the string `mississippi` is `pssmipissii` (Figure 2.2(c)).

It is also necessary to transmit the position in the permuted text of the first character of the original text. In the case of the example in Figure 2.2, we transmit the number four, since the letter `m`, which is the first character of the input, appears at position four in the permuted string. Therefore, the output of the Burrows-Wheeler transform for the string `mississippi` is the pair $\{\texttt{pssmipissii}, 4\}$.

Remarkably, it is possible, given only this output, to reconstruct the original text. This is possible because the matrix in Figure 2.2 is constructed using *cyclic* rotations; that is, the characters in the last column of Figure 2.2(b) cyclically precede those in the first column. Figure 2.2(d) shows this relationship more clearly: the characters in the left-hand column (which are taken from the last column of the sorted matrix) immediately precede those in the right-hand column (the first character of the sorted matrix).

If we refer to the first and last columns of the sorted matrix as F and L respectively, then L is the output from the Burrows-Wheeler transform, and F can be obtained by the decoder simply by sorting the characters of L. Then we see that each character in L is followed by the corresponding

```
e  aking it so decisively,
m  aking it stick to his he
e  aking it, and turning th
m  aking itself visible low
t  aking just now. You are
t  aking just the least thi
e  aking loneliness of the
m  aking me a woman, and de
m  aking movements associat
t  aking my own measure so
m  aking no attempt to cont
m  aking no reply. 'I fanci
e  aking of a daring attemp
e  aking of Bathsheba. Ther
e  aking of Boldwood, " He'
t  aking of her hand by the
e  aking of machinery behin
e  aking of one. He was at
e  aking of? ' she asked. T
e  aking off my engagement
```

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

(a)

1	mississippi
2	ississippim
3	ssissippimi
4	sissippimis
5	issippimiss
6	ssippimissi
7	sippimissis
8	ippimississ
9	ppimississi
10	pimississip
11	imississipp

(b)

	F		L
1	i	mississip	p
2	i	ppimissis	s
3	i	ssippimis	s
4	i	ssissippi	m
5	m	ississipp	i
6	p	imississi	p
7	p	pimississ	i
8	s	ippimissi	s
9	s	issippimi	s
10	s	sippimiss	i
11	s	sissippim	i

(c)

1	p
2	s
3	s
4	m*
5	i
6	p
7	i
8	s
9	s
10	i
11	i

(d)

	L	F
1	p	i
2	s	i
3	s	i
4	m*	i
5	i	m
6	p	p
7	i	p
8	s	s
9	s	s
10	i	s
11	i	s

Figure 2.2: Burrows-Wheeler transform of the string `mississippi`: (a) rotations of the string; (b) sorted matrix; (c) permuted string (last character of sorted matrix); (d) permuted string and sorted string.

character in F. In particular, we know that in this case $L[4]$ is the first character of the text; therefore, $F[4]$ (the letter `i`) must be the second character.

We must now consider the problem of locating the next character to decode. We know that each character in F must correspond to a character in L, since the two strings are merely permutations of one another. However, the letter `i` occurs four times in L. Which one corresponds to the `i` that has just been decoded (the fourth `i` in F)? To answer this question, we must observe that each group of characters in F occurs in the same order in L. For example, the third `s` in F corresponds to the third `s` in L, and so on. Since we are dealing in this instance with the fourth `i` in F, this corresponds to the fourth `i` in L ($L[11]$), which, we then discover, is followed by an `s`, and so forth. Decoding the rest of the string in a similar manner gives the order 4, 11, 9, 3, 10, 8, 2, 7, 6, 1, 5.

2.1 Implementing the Burrows-Wheeler Transform

Although it may seem that the BWT requires substantial overheads in terms of arrays, in practice, it can be implemented quite efficiently. For example, the matrix shown in Figure 2.2(a) is never constructed; instead, a list of indices into the text is created, and when two indices are being compared during the sorting process, the corresponding substrings are compared.

A further improvement comes from the fact that it is not necessary to construct the F array. Since the order of the alphabet is fixed, and we know from L how many occurrences of each character there are, we can compute the F array implicitly. Each symbol in the alphabet will occur in one contiguous group in F (whose size may be zero if the symbol does not occur in the text); let the array K store the lengths of these groups. Then from K we can compute M, which gives the starting index of each group in the (virtual) F array. (This approach has other merits than space efficiency, as we shall see in Chapter 4.)

As presented above, the inverse BWT takes $\mathcal{O}(n^2)$ time, because of the need to count the previous occurrences of each character. Naturally, there is a way to improve this as well, and in fact, the inverse transform can be made to run in $\mathcal{O}(n)$ time by means of a 'transform array', V, which can be created in linear time from L, K and M.

In the original paper by Burrows & Wheeler (1994), the transform array is computed 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 index of the first character in the text. Of course, using V in this way to generate the text results in its being constructed *backwards*. In fact, Burrows and Wheeler noted this in their paper: "We could have defined [*the transform array*] so that the string... would be generated from front to back, rather than the other way around." Exactly why they did not do so remains a mystery, and may be one of the reasons why the paper is so notoriously misunderstood.

Fortunately, it is just as easy to generate a 'forwards transform array', which we shall call W. A method for generating both the V and W arrays from the M array in linear time is given as Algorithm 2.1. This algorithm uses the key property outlined above, namely that the occurrences of each symbol in the alphabet appear in the same order in L as they do in F. For each character in L, BUILD-TRANSFORM-ARRAYS uses the M array to locate the corresponding character in F, and this relationship is recorded in the transform array.

Algorithm 2.2 performs the actual reconstruction of the string in 'front-to-back' fashion, using the W array, but could be trivially changed to use V instead. For most of the other applications in this report, W will certainly prove more useful.

2.2 Auxiliary arrays

Depending on the application, a number of auxiliary arrays may prove useful with the Burrows-Wheeler transform. Adjeroh, Mukherjee, Bell, Zhang & Powell (2001) describe a number of these

Algorithm 2.1 Build the transform arrays for the inverse BWT

```
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[i] ← M[i] + 1
5  end for
```

Algorithm 2.2 Reconstruct the original text using the V array

```
BWT-DECODE(L, W, index)
1  i ← index
2  for j ← 1 to n do
3      T[j] ← L[i]
4      i ← W[i]
5  end for
```

arrays; here, we will focus only on those auxiliary arrays relevant to the applications in this report.

Let Hr be defined as an array relating the characters in the original text T to their position in the sorted string F; that is,

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

Then let I be the inverse of this array; that is,

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

These two arrays are particularly useful in pattern matching, as they can be used to report the position of a match in the original text. The Hr array is used by the compressed-domain Boyer-Moore algorithm in Chapter 3, and the I array by the binary search algorithm in Chapter 4.

Algorithm 2.3 Compute the Hr and I arrays

```
COMPUTE-AUXILIARY-ARRAYS(V, index)
1  i ← index
2  for j ← 1 to n do
3      i ← V[i]
4      Hr[n − j + 1] ← i
5      I[i] ← n − j + 1
6  end for
```

2.3 Concluding remarks on the BWT

Because of the context method used in the BWT, it behaves very similarly to other context-based compression models such as PPM and DMC. In fact, block-sorting is closely related to the PPM* method (Cleary, Teahan & Witten 1995), which allows arbitrary-length contexts, and is among the best known models for achieving good compression. Block-sorting forms the basis of the popular bzip family of compression systems (Seward 2000), whose performance is widely regarded to be among the best publicly-available systems.

Chapter 3

A Boyer-Moore-based approach

The Boyer-Moore algorithm (Boyer & Moore 1977) is considered one of the most efficient algorithms for general pattern matching applications, and can achieve 'sub-linear' performance in most situations.

In principle, the Boyer-Moore algorithm is very similar to brute force pattern matching, in that searching is performed by 'sliding' the pattern across the text, and comparing the characters from the pattern with the corresponding text characters. The main differences are that the pattern is checked from right to left (rather than left to right, as in the brute force algorithm), and that the Boyer-Moore algorithm is able to recognise and skip certain areas in the text where no match would be possible.

3.1 Shift heuristics

Underlying Boyer-Moore's ability to skip portions of the text are two key heuristics: the **bad character rule** and the **good suffix rule**.

> **Definition 3.1.1 (Bad character rule)**
> Suppose the beginning of the pattern is aligned with the text at position k, and that a mismatch has occurred at position i in the pattern, that is, $P[i] \neq T[k+i-1]$. Let $c = T[k+i-1]$, the mismatched character from the text. The bad character rule proposes that the pattern be shifted to the right so that the mismatched character c is aligned with the rightmost position of c in P. If this would yield a negative shift (that is, the rightmost occurrence of c in P is *after* the mismatch), then the pattern is simply shifted to the right by one position.

In Figure 3.1(a), a mismatch has been detected between the `c` of `reduced`, and the letter `d` in the text. Since the rightmost `d` in `reduced` is to the right of the mismatch, the bad character heuristic simply proposes a shift by one character to the right.

It is trivial to generate a lookup array for the bad character heuristic in $\mathcal{O}(m + |\Sigma|)$ time and $\mathcal{O}(|\Sigma|)$ space; one such algorithm is given in Cormen, Leiserson & Rivest (1989).

> **Definition 3.1.2 (Good suffix rule)**
> Suppose that a substring t of T matches a suffix of P, but that a mismatch occurs at the next character to the left. Let t' be the rightmost occurrence of t in P such that t' is not a suffix of P, and the character to the left of t' differs from the character to the left of t in P. If t' exists, then shift P to the right so that the substring t' in P is aligned with the substring t in the text. Otherwise, shift P to the right until a prefix of P matches a suffix of t in T. If no such shift is possible, then shift P completely past t; that is, shift the pattern m places to the right.

Figure 3.1(c) shows a shift proposed by the good suffix heuristic, after the suffix `ed` has been matched. In this case, there is a further occurrence of `ed` in the pattern, and so this is aligned with the matched `ed` in the text, giving a shift of four characters.

The lookup array for the good suffix rule requires $\mathcal{O}(m)$ space, and can be computed in $\mathcal{O}(m)$ amortised time, although doing so is much more difficult than computing the bad character array. Gusfield (1997) details some of the history of the good suffix rule, and laments at some length the lack of adequate explanation in the literature. Ironically enough, it now appears that his own algorithm may contain errors. Fortunately, Cormen et al. (1989) gives a clear, concise explanation of the preprocessing algorithm, and an amortised analysis of its $\mathcal{O}(m)$ time complexity.

We have, as a result of the two rules, two possible candidates for shift distances at each mismatch during the search process. To resolve this, the maximum of the two shifts is chosen, so that the heuristic that yields a greater shift is the one that is taken. It is also easy to prove that the algorithm will make progress at each step, since the heuristics always give a positive shift, and that no occurrences of the pattern are missed due to over-zealous shifting.

3.2 Extensions to the basic shift heuristics

The bad character rule is useful when a mismatch occurs near the right end of P, but has no effect when the mismatching character from T occurs in P to the right of the mismatch position, since this would give a negative shift. To overcome this problem, the bad character rule is extended as follows:

Definition 3.2.1 (Extended bad character rule)
When a mismatch occurs at position i of P, and the mismatching character in T is c, then shift P to the right so that the closest c to the left of i in P is aligned with the mismatched c in T. If no such shift is possible (that is, there is no c to the left of position i in P), then shift P completely past the mismatched c in T.

Figure 3.1(b) shows a shift proposed by the extended bad character heuristic. Although the rightmost `d` in `reduced` is to the left of the mismatch position (and therefore of no use to the standard bad character heuristic), there is another `d` to the left of the mismatch. This is aligned with the mismatched `d` in the text, giving a shift of two places, instead of the single-place shift proposed by the original bad character rule.

The extended rule can yield much better shifts than the original bad character heuristic, with only a very slight trade-off in performance—at most one extra step per character comparison (Gusfield 1997).

There are several algorithms which claim to compute the extended bad character rule in $\mathcal{O}(m)$ space and time (Gusfield 1997, Mukherjee, Bell, Powell, Adjeroh & Zhang 2001). However, these use a linked list of pattern indices for each symbol in the alphabet (see Figure 3.2, so, strictly speaking, their space requirement is actually $\mathcal{O}(m + |\Sigma|)$. To retrieve the extended bad character shift for a text character c and a position i in P, traverse the list of indices for c until a value $k < i$ is obtained. If no such value is found, then there is no c to the left of i in P, and P should be shifted completely past the mismatched character; otherwise, the correct shift is given by $i - k$.

One way to avoid the overheads associated with linked lists is to use a two-pass technique similar to that used to compute the F array in the BWT decoding process. During the first pass, the number of occurrences of each character are counted, and an array of size m is partitioned accordingly. During the second pass, this array is populated with the indices in the pattern of each character. For example, the extended bad character array for the pattern `reduced` is shown in Figure 3.3. Note that this technique does not avoid the $\mathcal{O}(m + |\Sigma|)$ space requirement of the linked list technique described above, since one index for each symbol in the alphabet is required to keep track of the partitioning. However, it seems generally more efficient to implement this method than to use linked lists, especially when the pattern size is likely to be small.

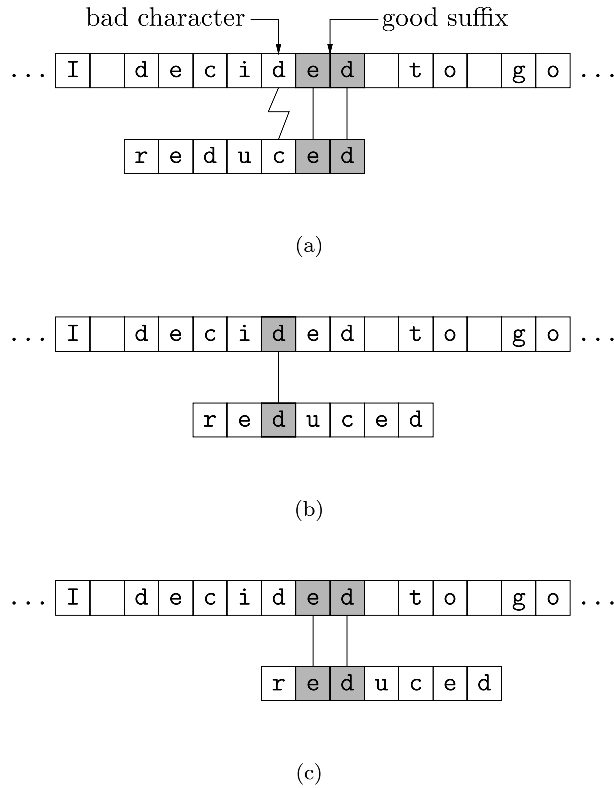


Figure 3.1: An illustration of the Boyer-Moore heuristics. (a) Matching the pattern `reduced` against a text by comparing characters from right to left gives a mismatch at position 5. (b) Shift proposed by the extended bad character rule. (c) Shift proposed by the good suffix rule.

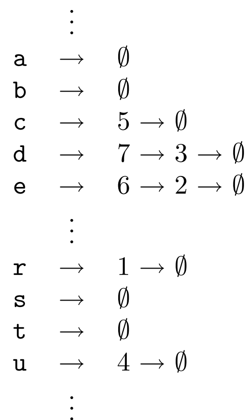


Figure 3.2: An array of linked lists showing the extended bad character shifts for the pattern `reduced`

5	c	← c
7	d	← d
3	d	
6	e	← e
2	e	
1	r	← r
4	u	← u

Figure 3.3: Extended bad character shift array for the pattern `reduced`

3.3 The compressed-domain Boyer-Moore algorithm

The compressed-domain application of the Boyer-Moore algorithm given by Mukherjee et al. (2001) relies on the fact that, given the F and Hr vectors, it is possible to decode portions of the text in a random-access fashion, without decoding the entire block. Pseudocode for this algorithm is given as Algorithm 3.1.

Algorithm 3.1 The compressed-domain Boyer-Moore algorithm.

```
BM-MATCH(P, F, Hr)
 1  COMPUTE-GOOD-SUFFIX(P)
 2  COMPUTE-BAD-CHARACTER(P)
 3  k ← 1
 4  while k ≤ n − m + 1 do
 5      i ← m
 6      while i > 0 and P[i] = F[Hr[k + i − 1]] do
 7          i ← i − 1
 8      end while
 9      if i = 0 then
10          # Report a match beginning at position k − m + 1
11          k ← k + G[0]
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
```

The exact details of how the algorithm computes the proposed shifts (lines 13–14) have been omitted here for clarity, but are not difficult to understand. In the case of the good suffix rule, the proposed shift can be obtained with a single array lookup; for the bad character rule, it is a matter of traversing the list of positions of the mismatched character in P until either a viable shift is found, or the end of the list is reached.

This BWT-Boyer-Moore algorithm allows searching in BWT-encoded files that is comparable to the fastest known algorithms for uncompressed text.

Chapter 4

Binary Searching

One interesting side effect of the Burrows-Wheeler transform is that it produces, as an artifact of the inverse transform, a list of all the substrings of the text in sorted order, making it possible to perform a binary search on any file encoded with the Burrows-Wheeler transform—an $\mathcal{O}(m \log n)$ pattern matching algorithm!

Figure 4.1 shows the sorted list of substrings for the text `mississippi`, taken from the sorted matrix in Figure 2.2(b). Note that a 'match' for the string `miss` actually involves finding a substring of the text that has `miss` as a *prefix*. (This is really no different from other pattern matching algorithms (such as Boyer-Moore), where the characters following a substring have no effect on its suitability as a candidate for matching.) Also, notice that all the occurrences of the substring `is` are together in the list (positions 3–4). This is an important observation in the application of binary search to BWT-compressed files.

In fact, it is possible to improve even on the above $\mathcal{O}(m \log n)$ figure, and obtain a pattern matching algorithm that runs in $\mathcal{O}(m \log \frac{n}{|\Sigma|})$ time, as a direct result of the efficiency improvements made in Chapter 2. Recall that the F array of the Burrows-Wheeler transform is not stored explicitly, but that the starting position and length of each run of characters are stored in the M and K arrays. Therefore, if the first character of the pattern is c, then the lower and upper bounds for a binary search are given, respectively, by $M[c]$ and $M[c+1]-1$. Thus, if the pattern is only one character long, or c occurs at most once in the text, then any results are immediately accessible from two array lookups; in all other cases, this observation effectively decreases the length of the pattern by one character and the search space by a factor of, on average, $\frac{1}{|\Sigma|}$.

To take advantage of this structure, it is necessary to make some minor modifications to the standard binary search algorithm. Firstly, we must devise a string-comparison function that works with the structure of the BWT. Such an algorithm is given in Algorithm 4.1. It uses the W array

```
 1  i
 2  i  ppi
 3  i  ssippi
 4  i  ssissippi
 5  m  ississippi
 6  p  i
 7  p  pi
 8  s  ippim
 9  s  issippim
10  s  sippi
11  s  sissippi
```

Figure 4.1: Sorted list of substrings of `mississippi`

to decode as much of the text as is needed to determine a match[1]. The parameter i signifies a row in the sorted matrix (Figure 2.2(b)), which corresponds to a substring t of the text. The return value is the same as C's STRCMP; that is, negative if $P < t$, positive if $P > t$, and zero if there is an exact match. Note that, in this case, an 'exact match' means that P matches a *substring* of the text.

Algorithm 4.1 String-comparison routine for BWT binary search

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

The binary search algorithm itself must be modified, because in this case we are looking for a range of results, rather than just one. The modified algorithm works as follows:

Step 1. [standard binary search]
Perform a standard binary search on the range $M[c] \ldots M[c+1] - 1$, as described above. Either this algorithm will terminate without a match (in which case the pattern is not present in the text), or a matching substring will be found, in which case we continue with the next step.

Step 2. [binary search on first half of list]
Suppose the match found in Step 1 occurs at position p of F. Then we have two cases to consider:

(a) There is some index p' such that $M[c] \leq p' < p$, and each index in $[p' \ldots p-1]$ points to a match. (This occurs because the list is in sorted order, so all matches will be contiguous in the list.)

(b) There are no matches with indices in $M[c] \ldots p-1$; let $p' = p$.

In either case, perform a binary search on $M[c] \ldots p-1$. At each step, compare the pattern to the substring at the midpoint of the list. If the result is negative or zero, then choose the first sublist (up to and including the midpoint); otherwise, choose the second sublist. In this way, the first match in $M[c] \ldots p-1$ (if it exists) will be located; if no such match is present, then p is the first occurrence of the pattern in the complete list.

Step 3. [binary search on second half of list]
A further two cases are presented by the match found in Step 1. Either:

(a) There is some index p'' such that $p < p'' \leq M[c+1] - 1$, and each index in $[p \ldots p'']$ points to a match.

(b) There are no matches with indices in $p+1 \ldots M[c+1] - 1$; let $p'' = p$.

[1] Or, more correctly, a mismatch, since ascertaining a match will always involve m comparisons.

Text entered	Results	Number of candidates
c	c...czarina	2110
co	coach...cozy	890
com	coma...comrade	155
comp	compact...compute	82
compr	comprehend...compromise	11

Figure 4.2: Incremental search for the word *compression*.

Again, in either case, perform a binary search on $p + 1 \ldots M[c + 1] - 1$, this time choosing the latter half of the list if the comparison value is positive or zero. This will ensure that the last occurrence of the pattern is found.

Step 4. [search results]
If a list of results is generated by the above steps, then $p' \ldots p''$ are the indices of the matches in the sorted matrix; that is, $F[p' \ldots p'']$ contains the first characters of the matches, and their indices in the text T are given by $I[p' \ldots p'']$. (Recall that the I array relates F and T; see Section 2.2 on page 13 for details.)

This process is summarised in Algorithm 4.2.

The enhanced binary search algorithm obtains all the matches with a fraction of the comparisons needed by linear-based algorithms such as the Boyer-Moore method presented in Chapter 3. In fact, in situations where there are a large number of matches present in the text, some matches will be found without any comparisons at all. This is possible because once two matches have been found, it is known that everything between them must also be a match, due to the sorted nature of the F array.

This extremely fast performance makes the binary search approach ideal for situations where speed is of the essence. One example is the 'incremental search' found in multimedia encyclopædias and online application help. As the user enters a word character by character, a list of possible results is displayed. This list becomes shorter as more of the word is typed and the query made more specific. For example, a search for the word *compression* in a dictionary might begin with the results shown in Figure 4.2[2]. By the time the prefix *compr* has been typed, the number of possible results is probably small enough to fit on one screen.

Saving the start and end indices of the previous searches allows an improvement in the performance of the binary search, since these indices can be used as the bounds for the next search.

Figure 4.3 shows an example of a full-text retrieval system that uses the binary search approach to provide a responsive incremental search. As the user enters a query in the text box in the top-left corner of the window, the list of results is updated with each keystroke. The system is so fast, in fact, that the only noticeable lag comes from the time taken to populate the list of results.

[2]Results taken from `/usr/dict/words`

Algorithm 4.2 BWT Binary search algorithm

```
BWT-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 ← BWT-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 BWT-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 BWT-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 BWT-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 BWT-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
```

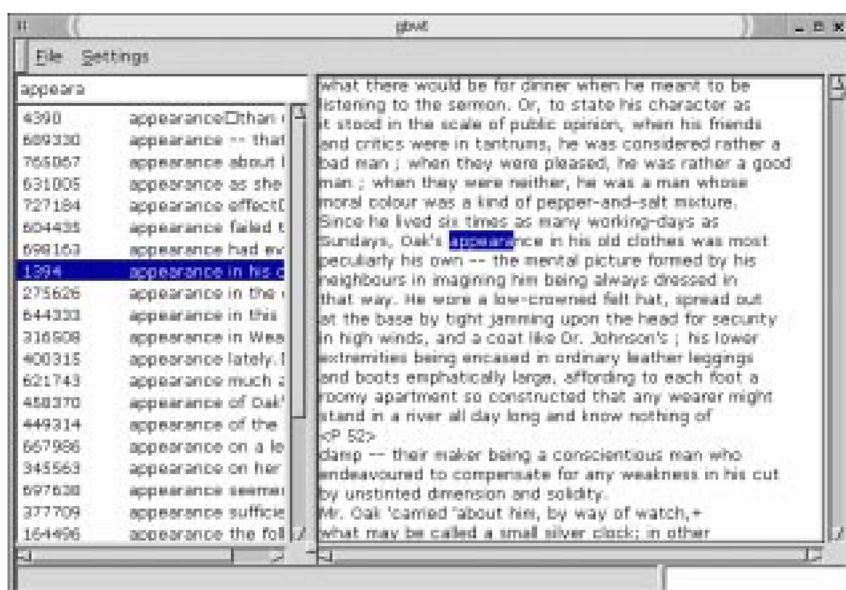


Figure 4.3: gbwt: a full-text retrieval system using binary search.

Chapter 5

Experimental Results

For the purposes of experimentation, a simple Burrows-Wheeler-based compression program, bsmp, was developed. bsmp uses a four-stage compression system:

1. a Burrows-Wheeler transform, with the block size set to the size of the entire file,
2. a move-to-front coder (Bentley et al. 1986), which takes advantage of the high level of local repetition in the BWT output,
3. a run-length coder, to remove the long sequences of zeroes in the MTF output, and
4. an order-0 arithmetic coder.

Table 5.1 compares the performance of bsmp with bzip2, a production-quality block-sorting compressor (Seward 2000). Compression and decompression times are given in seconds, and compression ratios in output bits per input byte. The files shown are all text files from the Canterbury Corpus (Arnold & Bell 1997, Powell 2001); the three larger files are from the 'Large' collection, and demonstrate the performance of block sorting on large files. Naturally, bsmp does not compare particularly favourably in terms of speed, since no real effort was spent in optimising this simple system, but the compression ratios achieved by bsmp are at least comparable to bzip2's. For the remainder of the experiments in this chapter, bsmp will be used as the compression system, since this allows us to work with partially decompressed files, and to treat the entire file as one block. It is intended to examine in more detail the relationship between the BWT and the compression process, and eventually to adapt the techniques in this report for use with bzip2-compressed files.

File	**Size**	**Compress Time**		**Decompress Time**		**Ratio**	
	(bytes)	**bzip2**	**bsmp**	**bzip2**	**bsmp**	**bzip2**	**bsmp**
text	152,089	0.12	2.55	0.04	0.16	2.272	2.630
play	125,179	0.11	1.95	0.04	0.15	2.529	2.915
Csrc	11,150	0.03	0.14	0.01	0.02	2.180	2.470
list	3,721	0.02	0.05	0.00	0.00	2.758	2.969
tech	426,754	0.33	8.85	0.10	0.42	2.019	2.351
poem	481,861	0.40	8.98	0.12	0.53	2.417	2.797
man	4,227	0.02	0.04	0.01	0.01	3.335	3.577
E.coli	4,638,690	3.94	151.82	1.13	7.77	2.158	2.337
world	2,473,400	2.10	79.34	0.46	2.59	1.584	1.633
bible	4,047,392	2.10	113.36	0.77	4.96	1.671	1.836

Table 5.1: Performance of the bzip2 and bsmp compression systems

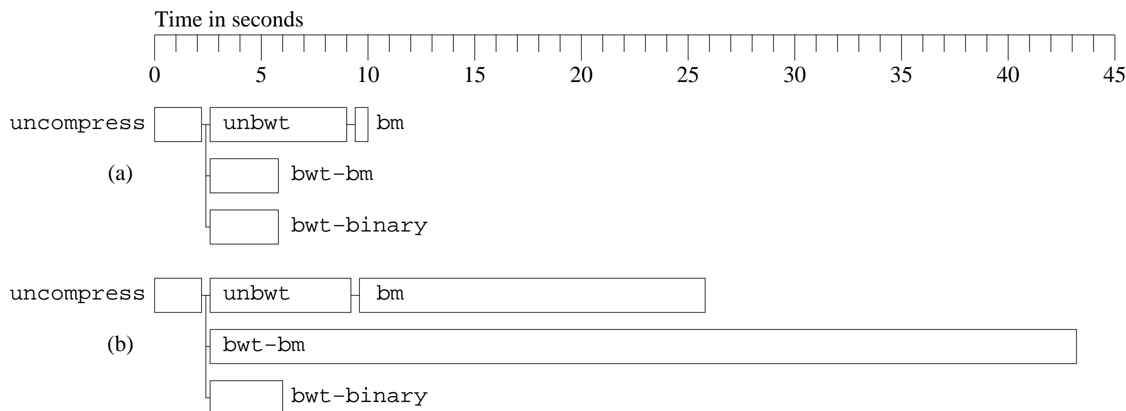


Figure 5.1: Search times for (a) a single pattern, and (b) a hundred randomly-selected words from the file `bible.txt`

There are a number of interesting choices to be made when implementing a block-sorting compressor such as bsmp. Fenwick (1996*a*) examines these choices in some detail, but his compression system does not use run-length coding (step 3 above). In implementing bsmp, we have chosen to use two models for the arithmetic coder. The first models the lengths of runs in the MTF output, and the second the MTF values themselves. By always outputting a run length, we effectively compose a flag denoting whether this is a run or not with the actual run length, since there will be approximately a 90% chance that the run length will be one[1]. In a system based on Huffman coding, this would be wasteful, but an arithmetic coder allows a representation that is arbitrarily close to the entropy (Witten et al. 1999), and so a probablility of 90% can be coded in $-\log_2 0.9 = 0.152$ bits.

There is some room for fine-tuning, but we are mainly concerned here with relative speed, not compression performance.

5.1 Search performance

The main result we are interested in is, of course, whether or not searching is actually faster with the Burrows-Wheeler transform. Assuming that we have a file that has been compressed as described above, we could either completely decompress the file and then use a tool like grep to do the pattern matching, or only partially decompress the file (to the BWT-permuted output) and use one of the BWT search algorithms described in this report.

In considering which method to use, we must take into account the overheads associated with each stage of the decompression process. For example, if the inverse BWT phase is particularly costly, it may be better to search on the BWT-encoded file. However, if the time taken to decompress the arithmetic coding, run-length coding and move-to-front coding together far outweighs the cost of the inverse BWT and the search on the decoded text, or the search on the BWT file, then the choice of whether or not to perform the inverse BWT in full is largely arbitrary.

To investigate the individual parts of the decompression process, the bsmp decompressor was modified to produce as output a BWT file, which could then be fed into a BWT decoder if required. This allowed us to compare the relative performance of the BWT search algorithms with a decode-and-search approach.

Figure 5.1 shows a timeline view of the three search algorithms on the 3.86 megabyte `bible.txt` file from the Canterbury Corpus (the actual results are shown in Table 5.2). The '`uncompress`' phase represents the time taken to decode the arithmetic coding, run-length coding and move-to-front coding, producing a BWT file. This file is then used for each of the three search methods:

[1] Based on empirical data.

	Component steps					Total time taken		
Patterns	`uncomp`	`unbwt`	`bm`	`bwt-bm`	`bwt-binary`	`unbwt-bm`	`bwt-bm`	`bwt-binary`
1	2.3	6.7	0.7	3.3	3.6	9.7	5.6	5.9
10	2.3	6.7	2.0	6.7	3.6	11.0	9.0	5.9
20	2.3	6.7	3.7	10.7	3.6	12.7	13.0	5.9
30	2.3	6.7	5.6	15.4	3.6	14.7	17.7	5.9
40	2.3	6.7	6.7	17.2	3.6	15.7	19.5	5.9
50	2.3	6.7	8.6	21.1	3.5	17.6	23.4	5.8
60	2.3	6.7	10.4	25.2	3.6	19.4	27.5	5.9
70	2.3	6.7	11.4	28.5	3.6	20.5	30.8	5.9
80	2.3	6.7	12.9	32.2	3.6	21.9	34.5	5.9
90	2.3	6.7	14.8	36.1	3.5	23.8	38.4	5.8
100	2.3	6.7	16.7	40.8	3.6	25.7	43.1	5.9

Table 5.2: Time in seconds to search a BWT-compressed file for one or more patterns

- `unbwt` → `bm` applies the inverse BWT and uses a standard Boyer-Moore algorithm to search for the pattern. Rather than use a pre-built utility like `grep` for the pattern-matching stage, it was decided to build a pattern-matching program from scratch, for consistency of implementation (`grep` is heavily optimised, and this may have resulted in biased results). The pattern matching algorithm used here is the same Boyer-Moore code used in Chapter 3, but without the modifications for searching BWT files.
- `bwt-bm` is the compressed-domain Boyer-Moore algorithm, as described in Chapter 3.
- `bwt-binary` is the binary search algorithm defined in Chapter 4.

Figure 5.1(a) shows the time taken to search for a single pattern. In this case, the `unbwt` phase takes approximately twice as long as the `bwt-bm` and `bwt-binary` methods, which take approximately equal time. In this case, either of the two compressed-domain algorithms would be appropriate. The main overhead for these algorithms is the time taken to construct the various transform arrays from the BWT output, as the search time for a single pattern is trivial.

Figure 5.1(b) illustrates the behaviour of the different search algorithms when searching for more than one pattern. In this case, a hundred words were randomly selected from the file as candidate patterns, and the pattern matching algorithms searched for each one in turn. In this case, the compressed-domain Boyer Moore algorithm was dramatically slower than even the plain-text search. One possible reason for this is that although the two programs share almost identical code, the permuted nature of the BWT output means that a compressed-domain Boyer-Moore algorithm must perform several array lookups per comparison, whereas in the plain-text algorithm, these can be replaced with pointer arithmetic. It is interesting to note that there is almost no difference between searching for one pattern with the binary algorithm and searching for a hundred. This underscores the fact that most of the time taken for the binary search is in loading and processing the BWT file.

As can be seen from Figure 5.2, above about twenty patterns it becomes more efficient to decode the BWT file and search on the plain text than to use the compressed-domain Boyer-Moore algorithm. However, both these methods are dramatically slower than the compressed-domain binary search, which is almost constant at around 5.9 seconds for any number of patterns.

This general pattern holds true independent of the size of the input file. For single pattern searches, there is no significant difference between the two compressed-domain methods, while for more than one search, binary search is the clear winner.

5.2 Memory usage

The main disadvantage of the binary search approach is its memory requirements. All the search methods use a number of arrays whose size is proportional to the length of the input file. Com-

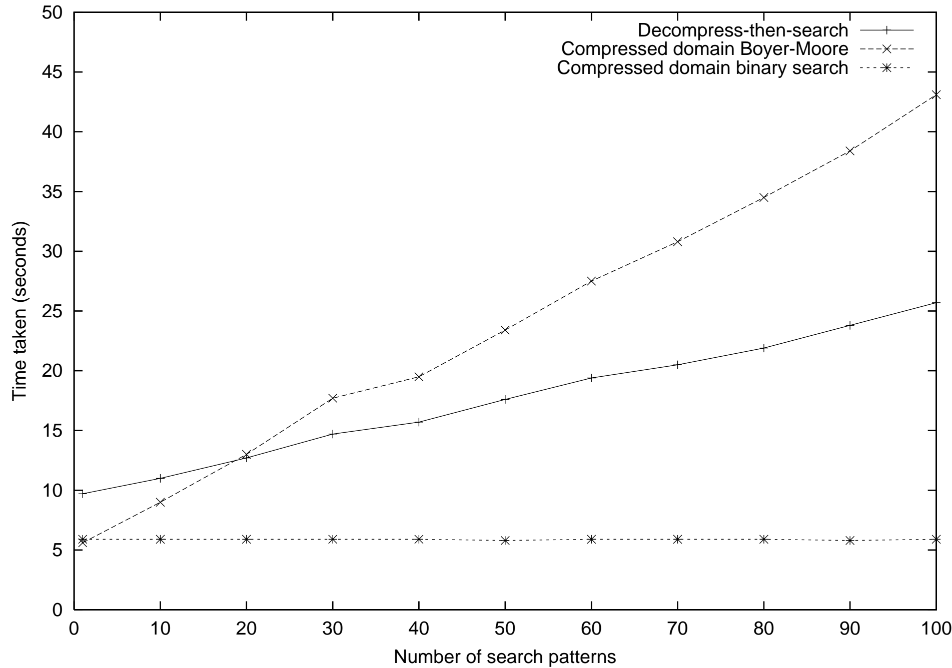


Figure 5.2: Search times for one or more patterns

monly, characters are stored in a single byte and integers take four bytes. Since integers in this context are used as indices into the file, this representation allows for files up to $2^{32} = 4$ GB in size.

Assuming single-byte characters and four-byte integers, an array of n characters will take n bytes, and an array of integers will take $4n$ bytes. As can be seen from Table 5.3, both the decompress-then-search and compressed-domain Boyer-Moore methods use one array of characters and one of integers, for a total of $5n$ bytes of memory (disregarding other, smaller arrays whose size is $\mathcal{O}(|\Sigma|)$), while the binary search method uses one character array and two integer arrays, for a total of $9n$ bytes. For a typical file one megabyte in size, this is an additional memory requirement of four megabytes. However, we believe that this cost is justified by the performance benefits of binary search, especially in light of the increasing amount of memory available on personal computers.

5.3 Number of comparisons

The other figure of interest is the number of comparisons performed during the search phase, since this represents the amount of work done by the algorithm. Figure 5.3 shows the mean number of comparisons to find all occurrences of all the words in `bible.txt`, by word length. Somewhat incredibly, the binary search algorithm is able to find every instance of a particular pattern in under a hundred comparisons on average, compared to Boyer-Moore (hundreds of thousands of

Method	**Arrays used**	**Memory required**	**Example for 4MB file**
Decompress-then-search	L, K	$5n$ bytes	20MB
Compressed-domain Boyer-Moore	F, Hr	$5n$ bytes	20MB
Compressed-domain binary	L, W, I	$9n$ bytes	36MB

Table 5.3: Memory requirements for searching

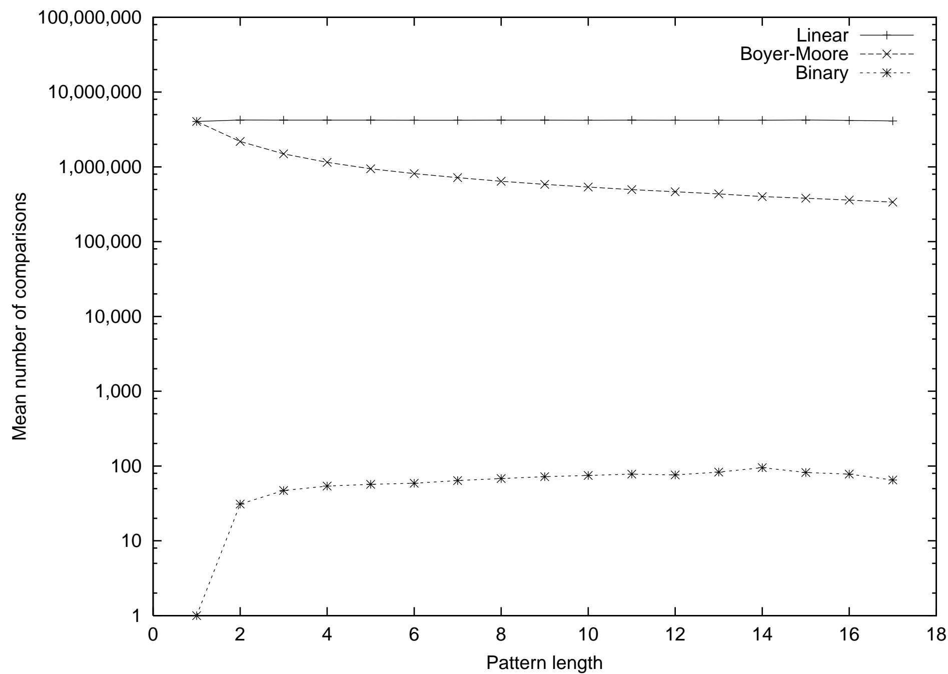


Figure 5.3: Mean number of comparisons by pattern length for the file `bible.txt`

comparisons) and linear search (millions of comparisons).

5.4 Unsuccessful searches

In the above experiments we have used patterns that are known to appear in the input file. However, in practice, there is no noticeable difference when searching for patterns that do *not* appear in the file. Binary search may have to perform one or two extra iterations, because it has direct access to patterns starting with a given character, but terminates almost as quickly as if an occurrence of the pattern had been found, while Boyer-Moore still traverses the entire file.

The only case where this will be relevant is if the pattern contains a character that is not found in the input. In this case, the binary search algorithm will terminate immediately if the character is at the start of the pattern, but will behave normally otherwise. The Boyer-Moore algorithm will obtain something close to its best-case performance of $\mathcal{O}(\frac{n}{m})$ comparisons, depending on the position of the 'bad character' in the pattern.

Chapter 6

Conclusion

In this report we have demonstrated how to take advantage of the structure placed in files compressed with the Burrows-Wheeler transform, and in particular how to use this structure for pattern-matching applications, introducing two new methods for solving the pattern matching problem in the context of BWT-compressed files. The first, the Boyer-Moore-based approach discussed in Chapter 3, applies an existing algorithm to the special structure of the Burrows-Wheeler transform, while the second method, the binary search approach of Chapter 4, uses the unique properties of the Burrows-Wheeler transform to produce a pattern-matching strategy that would be impossible with any other compression method.

As noted in Section 5.1, there is almost no difference in speed between the two compressed-domain algorithms for straightforward, single-pattern searches. In this case, the Boyer-Moore algorithm has the advantage of lower memory requirements, and should prove useful for applications where memory is at a premium. On the other hand, the slightly higher memory footprint of the binary search algorithm is offset by dramatic performance gains for multiple pattern searches, and by a considerable increase in the power of possible searches, including boolean and 'pseudo-boolean' queries (AND, OR, NEAR) and approximate matching.

In fact, the Burrows-Wheeler transform has many more uses than simple pattern-matching. There are many other uses of a sorted list of substrings, especially in the area of computational biology. For example, it is possible to determine, with only one pass through the encoded E.coli file, that the longest repeated substring in the liverwort DNA sequence is 2,815 characters long, and begins with "AAAGAAACATCTTCGGGTTG...". Similar searches have revealed a 551-character repetition in the King James Bible[1]. It is also conceivable that similar strategies could be used to efficiently detect plagiarism in collections of text or source code.

In general, the Burrows-Wheeler transform is useful in any situation where a sorted index is useful. Using the V and W transform arrays, it is possible to generate infinite forwards and backwards contexts for a keyword-in-context (KWIC) index. It is also possible to use the Burrows-Wheeler transform for simple data mining. For example, in the sorted list, all substrings containing the letters '`www.`' or '`Mon`' will be adjacent, allowing fast retrieval.

6.1 Further work

Although initial results with the techniques described in this report are very pleasing, there is still considerable scope for future investigation and research.

[1] Both occurrences are contained in Numbers 7:60–71, and relate to the dedication of the tabernacle, where the leaders of the twelve tribes of Israel each bring an identical sacrifice. Interestingly, some versions of the Bible use a form of run-length coding, where verses 12–83 are represented with a list of the tribal leaders and a single description of the sacrifice.

6.1.1 Moving further into the compressed domain

In Chapter 5, we showed that it is possible to search directly on the permuted text, and in this way to achieve significant reductions in search time. However, if a file has been compressed with the 'BWT → MTF → RLE → VLC' method, it is still necessary to perform three stages of decoding to obtain the permuted text. This represents a significant amount of work—up to a third of the total search time in some cases.

Mukherjee et al. (2001) describes a method for obtaining the F array from the move-to-front output in $\mathcal{O}(n \log |\Sigma|)$ time. It may be possible to extend this to other arrays required for pattern-matching, such as Hr, to gain a performance increase by skipping the reverse move-to-front step altogether. It may even be possible to skip the run-length decoding process in a similar way, leaving only the arithmetic decoding stage.

6.1.2 Dealing with blocked files

The major disadvantage of the compressed-domain binary search algorithm is that it requires that the entire input file be treated as a single block. This requires enough memory to load the entire file at once, which is may be infeasible when decoding files that are many megabytes or even gigabytes in size, although when encoding it may be possible to perform the sorting in several phases, similar to mergesort.

Most block-sorting compressors use much smaller blocks. In the original paper by (Burrows & Wheeler 1994), the authors tested block sizes from one kilobyte to 103 megabytes before apparently settling on a block size of 16 kilobytes[2]. bzip2 is capable of using block sizes between 100 and 900 kilobytes, with a default of 900k. As the block size increases, more context becomes available, and better compression is likely to be achieved, although the man page for bzip2 notes that "larger block sizes give rapidly diminishing marginal returns".

It is therefore necessary to investigate the effect of 'blocked' files on the performance of the binary search algorithm. This will involve developing a method to deal with the case when an instance of a pattern overlaps a block boundary. It is expected that the modified routine will be approximately $\mathcal{O}(\frac{mn}{b} \log b)$, where b is the block size.

6.1.3 Approximate matching

So far we have only discussed exact pattern matching with the Burrows-Wheeler transform. The paper by Adjeroh et al. (2001) (draft enclosed as Appendix A) discusses a new method for approximate pattern-matching using q-grams, but it should be possible to adapt the binary search techniques outlined here to the task of approximate pattern matching. This might involve rotating the pattern so that the longest substring is at the beginning and can then be used to filter the text for possible matches, which can then be verified (Witten et al. 1999). This is similar to the apporach of Bratley & Choueka (1982), but without the memory overhead of storing the rotated lexicon, since this is captured by the BWT.

6.1.4 Relationship to other data structures

There is a strong relationship between the Burrows-Wheeler transform and the binary search tree. In fact, a binary search tree is implicit in the inverse BWT process (Bell 1986). This has applications in 'longest match' searching, since the longest match for a particular string will be on the path from the root to that string in a binary search tree. It is worth investigating whether strategies and algorithms that apply to search trees in general can be applied to the Burrows-Wheeler transform, and whether the BWT is similarly related to any other search structures.

[2] This figure was mysteriously mentioned in the 'Performance of implementation' section of the paper with no apparent justification.

6.2 Concluding remarks

In conclusion, the Burrows-Wheeler transform not only yields excellent compression, but provides an excellent means of performing many different kinds of search tasks, because it provides easy access to a list of all sorted substrings. This is a very promising and rewarding area, and one that certainly deserves further attention.

Appendix A

Searching with q-grams

This appendix contains the draft of a paper to be submitted to DCC 2002, of which the author of this project is a co-author. It is included here to further illustrate the usefulness of the Burrows-Wheeler transform and its usefulness in pattern-matching applications.

Pattern Matching in BWT-Compressed Text

Don Adjeroh
Lane Department of Computer Science
and Electrical Enginerring
West Virginia University
Morgantown, WV 26506-6109, USA

Amar Mukherjee, Nan Zhang
School of Electrical Engineering
and Computer Science
University of Central Florida
Orlando, FL 32816, USA

Tim Bell, Matt Powell
Department of Computer Science
University of Canterbury
Christchurch, New Zealand

Draft of 1 November, 2001

Abstract

The compressed pattern matching problem is to locate the occurrence(s) of a pattern P in a text string T using a compressed representation of T, with minimal (or no) decompression. In this paper, we provide **on-line** algorithms for solving both the exact pattern matching problem and the k-approximate matching problem directly on BWT compressed text. The BWT provides a lexicographic ordering of the input text as part of its inverse transformation process. Based on this observation, pattren matching is performed by text pre-filtering, based on a fast q-gram intersection of segements from the pattern P and the text T. Algorithms are proposed that perform exact pattern matching $O(u+m\log\frac{u}{|\Sigma|})$ time on average, and k-approximate matching in $O(u+|\Sigma|\log|\Sigma|+\frac{m^2}{k}\log\frac{u}{|\Sigma|}+\alpha k)$ time on average, $(\alpha \leq u)$, where $u=|T|$ is the size of the text, $m=|P|$ is the size of the pattern, and Σ is the symbol alphabet.

1 Introduction

The pattern matching problem is to find the occurrence of a given pattern in a given text string. This is an old problem, which has been approached from different fronts, motivated by both its practical significance, and its algorithmic importance. Matches between strings are determined based on the *string edit distance*. Given two strings $A : a_1...a_u$, and $B : b_1...b_m$, over an alphabet Σ, the *exact string matching problem* is to check for the existence of a substring of the text that is an exact replica of the pattern string. That is, the edit distance between the substring of A and the pattern should be zero. In the *k-approximate string matching*, the task is to check if there exists a substring A_s of A, such that the edit distance between A_s and B is less than or equal to k. Various algorithms have been proposed for both exact and approximate pattern matching. See (Navarro 2001) for a recent survey.

With the sheer volume of data easily available to an ordinary user and the fact that most of this data is increasingly in compressed format, efforts have been made to address the *compressed pattern matching problem*. Given T a text string, P a search pattern, and Z the compressed representation of T, the problem is to locate the occurrences of P in T with minimal (or no) decompression of Z. Initial attempts on compressed pattern matching were directed on compresson schemes based on the LZ family (Amir, Benson & Farach 1996, Farach & Thorup 1998), where algorithms have been proposed that can search for a pattern in an LZ77-compressed text string in $O(n \log^2(\frac{u}{n}) + m)$ time, where $m = |P|, u = |T|, andn = |Z|$. Bunke and Csirik (Bunke & Csirik 1995) proposed methods that can search for patterns in run-length encoded files in $O(um_c)$, when m_c is the lenght of the pattern when it is compressed. In (Moura, Navarro, Ziviani & Baeza-Yates 2000), $O(n + m\sqrt{u})$ algorithms were proposed for searching Huffman-encoded files. Special compression schemes that facilitate later pattern matching directly on the compressed file have also been proposed (Manber 1997, Shibata, Kida, Fukamachi, Takeda, Shinohara, Shinohara & Arikawa 1999).

Although there has been a substantiatial work in compressed pattern matching, a recent survey (Bell et al. 2001) shows that little has been done on searching directly on text compressed with the Burrows-Wheeler Transform (BWT). Yet, in terms of data compression, empirical evidence (Burrows & Wheeler 1994, Fenwick 1996*b*, Balkenhol, Kurtz & Shtarkov 1999) shows that the BWT is significantly superior to the more popular LZ-based methods (such as GZIP and COMPRESS), and is only second to the PPM* algorithm (Cleary & Witten 1984). In terms of running time, the BWT is much faster than the PPM*, but comparable with LZ-based algorithms. So far, the major reported work on searching on BWT-compressed text are those of Sadakane (Sadakane 2000*a*) and Ferragina and Manzini (Ferragina & Manzini 2000), who proposed $O(m \log u + \eta_{occ} \log^\epsilon u)$ and $O(m + \eta_{occ} \log^\epsilon u)$ time algorithms respectively, to locate all η_{occ} occurrences of P in T, where $0 < \epsilon \leq 1$. However, the methods are for **off-line** pattern matching, and they considered only exact pattern matching.

In this paper, we provide **on-line** algorithms for solving both the exact pattern matching problem and the k-approximate matching problem directly on BWT compressed text.

2 The Burrows-Wheeler Transform

The BWT performs a permutation of the characters in the text, such that characters in lexically similar contexts will be near to each other. Important procedures in BWT-based compression/decompression are the forward and inverse BWT, and the subsequent encoding of the permuted text.

The forward transform

Given an input text $T = t_1 t_2 \dots t_u$, the forward BWT is composed of three steps: 1) Form u permutations of T by cyclic rotations of the characters in T. The permutations form a $u \times u$ matrix M', with each row in M' representing one permutation of T; 2) Sort the rows of M'

lexicographically to form another matrix M. M includes T as one of its rows; 3) Record L, the last column of the sorted permutation matrix M, and id, the row number for the row in M that corresponds to the original text string T.

The output of the BWT is the pair, (L, id). Generally, the effect is that the contexts that are similar in T are made to be closer together in L. This similarity in nearby contexts can be exploited to achieve compression. As an example, suppose $T = mississippi$. Let F and L denote the array of $first$ and $last$ characters respectively. Then, $F = iiiimppssss$ and $L = pssmipissii$. The output of the tranformation will be the pair: $(pssmipissii, 5)$ (indices are from 1 to u).

The inverse transform

The BWT is reversible. It is quite striking that given only the (L, id) pair, the original text can be recovered exactly. The inverse transformation can be performed using the following steps (Burrows & Wheeler 1994): 1) Sort L to produce F, the array of first characters; 2) Compute V, the *transformation vector* that provides a one-to-one mapping between the elements of L and F, such that $F[V[j]] = L[j]$. That is, for a given symbol $\sigma \in \Sigma$, if $L[j]$ is the c-th occurrence of σ in L, then $V[j] = i$, where $F[i]$ is the c-th occurrence of σ in F; 3) Generate the original text T, since the rows in M are cyclic rotations of reach other, the symbol $L[i]$ cyclically precedes the symbol $F[i]$ in T. That is, $L[V[j]]$ cyclically precedes $L[j]$ in T.

For the example with $mississippi$, we will have $V = [6\ 8\ 9\ 5\ 1\ 7\ 2\ 10\ 11\ 3\ 4]$. Given V and L, we can generate the original text by iterating with V. This is captured by a simple algorithm: $T[u+1-i] = L[V^i[id]], \forall i = 1, 2, \ldots u$, where $V^1[s] = s$; and $V^{i+1}[s] = V[V^i[s]]], 1 \leq s \leq u$. In practical implementations, the transformation vector V is computed by use of two arrays of character counts $C = c_1, c_2, \ldots c_{|\Sigma|}$, and $R = r_1, r_2, \ldots r_u$: $V[i] = R[i] + C[L[i]], \forall i = 1, 2, \ldots u$

where, for a given index, c, $C[c]$ stores the number of occurrences in L of all the characters preceding σ_c, the c-th symbol in Σ. $R[j]$ keeps count of the number of occurrences of character $L[j]$ in the prefix $L[1, 2, \ldots, j]$ of L. With V, we can use the relation between L, F, and V to avoid the sorting required to obtain F. Thus, we can compute F in $O(u)$ time.

BWT-based compression

Compression with the BWT is usually accomplished in four phases, viz: $bwt \rightarrow mtf \rightarrow rle \rightarrow vlc$, where *bwt* is the forward BWT transform; *mtf*—move-to-front encoding (Bentley et al. 1986) to further transform L for better compression (this usually produces runs of the same symbol); *rle*—run length encoding of the runs produced by the *mtf*; and *vlc*—variable length coding of the *rle* output using entropy encoding methods, such as Huffman or arithmetic coding.

3 Overview of Our Approach

The motivation for our approach is the observation that the BWT provides a lexicographic ordering of the input text as part of its inverse transformation process. The decoder only has limited information about the sorted context, but it may be possible to exploit this to perform an initial match on two symbols (a character and its context), and then decode only that part of the text to see if the pattern match continues. We give our description in terms of the F, L and V arrays, (i.e. the output of the *bwt* transformation—before the *mtf* and further encoding). The methods can be modified to search directly on the encoded output.

Given F and L, we can obtain a set of *bi-grams* for the original text sequence T. Let $\mathcal{Q}_2^T$ and $\mathcal{Q}_2^P$ be the set of bi-grams for the text string T and the pattern P respectively. We can use these bi-grams for at least two purposes:

Pre-filtering

To search for potential matches, we consider only the bi-grams that are in the set $\mathcal{Q}_2^T \cap \mathcal{Q}_2^P$. If the intersection is empty, it means that the pattern does not occur in the text, and we don't need

to do any further decompression.

Approximate pattern matching

We can generalize the bi-grams to the more usual q-grams, and perform q-gram intersection on $\mathcal{Q}_q^T$ and $\mathcal{Q}_q^P$ — the set of q-grams from T and P. At a second stage we verify if the q-grams in the intersection are part of a true k-approximate match to the pattern.

Example. Suppose $T = abraca$, and $P = rac$. We will have $L = caraab$, and $F = aaabcr$. Using F and L, we can obtain the bi-grams: $\mathcal{Q}_2^T = \{ac, ab, br, ca, ra\}$ and $\mathcal{Q}_2^P = \{ra, ac\}$. Intersecting the two, we see that only $\{ra, ac\}$ are in the intersection. For exact pattern matching, ac will be eliminated, and thus we will only need to check in the area in T that contains ra, since any match must contain ra. Suppose we had $P = abr$ as the pattern, the intersection will produce $\{ab, br\}$, eliminating the other potential starting points in F that also started with a. $\diamondsuit$

3.1 Generating q-grams from the BWT output

With only F and L, we can easily generate the bi-grams, and all the other q-grams in T, $(q \leq u)$. A simple rocedure that generates sorted q-grams is given below. We denote the sorted x-grams as $F(x\text{-}gram)$ which is a vector of length $u = |T|$ of x-tuples of characters. Obviously, $F = F(1\text{-}gram)$ and the lexicographically sorted matrix of all cyclic rotations of T is $F(u\text{-}gram)$. We assume $x \leq u$. The symbol '*' denotes concatenation of character strings.

grams(F, L, V, q)

```
F(1-gram)=F;
for x=2 to q do
  for i=1 to u do
    F(x-gram)[T(i)]:= L[i]*F((x-1)-gram)[i];
  end;
end;
```

If $q = 2$, the result will be the sorted bi-grams. Results from the precedure for the first few values of x (with $T = abraca$) are shown in the table below. The M matrix is shown for convenience. The procedure $O(u^2)$ in the worst case.

F....	L	V	FS	FST ...	M
a	c	5	aa	aab	aabrac
a	a	1	ab	abr	abraca
a	r	6	ac	aca	acaabr
b	a	2	br	bra	bracaa
c	a	3	ca	caa	caabra
r	b	4	ra	rac	racaab

4 Fast q-gram generation and intersection

4.1 Permissible q-grams

Given q, the q-gram generateion procedure produces **all** x-grams, where $x = 1, 2, \ldots q$. However, for a given pattern, we do not need every one of the $O(n^2)$ possible q-grams that is generated. The key to a faster approach is to generate *only* the q-grams that are necessary, using the F and Hr arrays. We call these q-grams that are necessary the *permissible q-grams* — they are the only q-grams that are permissible given u, m, and the fact that matching can not progress beyond the last characters in T and P.

Thus, the bi-gram aa and the 3-gram aab produced in the previous example, are not permissible. Further, if we wish to perform exact-pattern matching for a pattern P, where $|P| = m$, all we need will be the m-length q-grams (i.e. the m-grams) in the text T. The m-length q-grams (and excluding the q-gram from the rotations of the text) are the *permissible q-grams.* In general, we have a total of $u - q + 1$ permissible q-grams for a u-length text. The major problems are to find cheap ways to generate the q-grams from T, and then how to perform the intersection quickly.

4.2 Fast q-gram generation

The inverse BWT transformation is defined as: $\forall_{i=1,2,\ldots,u}, T[u+1-i] = L[V^i[id]]$, where $V^i[s] = V[V[\ldots V[s]]]$ (s times) and $V^1[s] = s$. Since $V^i[z]$ is just one more indirection on $V^{i-1}[z]$, we can reduce the time required by storing the intermediate results, to be used at the next iteration of the loop. Since F is already sorted, and $F[z] = L[V[z]]$, we can use a mapping between T and F, (rather than L), so that we can use binary search on F. We can use an auxiliary array H (or its reverse Hr to hold the intermediate steps of the indexing using V: $\forall i, i = 1, 2, \ldots, u$ $H[i] = V[V^i[id]]$, and $T[i] = F[H[u+1-i]]$; or $Hr = \textbf{reverse}(H)$, and $T[i] = F[Hr[i]]$

Example. The mapping vectors are shown below for $T = abraca$, $u = |T| = 6$. I is a index vector to the elements of Hr in sorted order.

idx	T	L	F	V	|	Hr	I
1	a	c	a	5	|	2	6
2	b	a	a	1	|	4	1
3	r	r	a	6	|	6	4
4	a	a	b	2	|	3	2
5	c	a	c	3	|	5	5
6	a	b	r	4	|	1	3

Hr (also H) represents a one-to-one mapping between F and T. By simply using F and Hr, we can access any character in the text string, without using T itself — which is not available without complete decompression. Therefore, there is a simple algorithm to generate the q-grams, for any given q: $\forall_{x=1,2,\ldots u-q+1}$, $\mathcal{Q}_q^T[x] = F[Hr[x]] \ldots F[Hr[x+q-1]]$;

These q-grams are not sorted. However, we can obtain the sorted q-grams directly by picking out the x's according to their *order* in Hr, and then use F to locate them in T. We make the following claim:

Lemma 1: *q-grams* **generation in constant time.** *Given $T = t_1 t_2, \ldots t_u$ transformed with the BWT, F, the array of first characters, and I, the index vector to sorted form of the indices in Hr, for any $q, 1 \le q \le u$, $\mathcal{Q}_q^T$, the set of permissible q-grams can be generated in constant time and constant space.*

Sketch of Proof. The availability of F and I (or Hr) implies constant-time access to any area in the text string T. We notice that the x used in the previous description is simply an index on the elements of T. $\diamondsuit$

4.3 Fast q-gram intersection

We present different fast q-gram intersection algorithms as a series of refinements on a basic algorithm. The refinements are based on the nature of the different arrays used in the BWT process, and the new transformation vectors previously described.

Algorithm 1

Let $\mathcal{MQ}_q = \mathcal{Q}_q^P \cap \mathcal{Q}_q^T$. We call $\mathcal{MQ}_q$, the *set of matching q-grams.* For each q-gram, we use indexing on F and Hr to pick up the required areas in T, and then match the patterns. To compute $\mathcal{MQ}_q$, we need to search for the occurrence of each member of $\mathcal{Q}_q^P$ in $\mathcal{Q}_q^T$. This will

require a running time proportional to $q(u-q+1)(m-q+1))$. This will be $O(mu)$ on average, and $O(u^3)$ worst case.

We can improve the search time by using the fact that F is already sorted. Hence, we can use binary search on F to determine the location (if any) of each q-gram from $\mathcal{Q}_q^P$. This will reduce the time to search for each q-gram to $q\log(u-q+1)$ time units, giving an $O(q(m-q)\log(u-q))$ time for the intersection. Average time will be in $O(m\log u)$, while the worst case will be in $O(\frac{u^2}{4}\log\frac{u}{2}) = O(u^2\log u)$.

Algorithm 2

With the sorted characters in F, we can view the F array as being divided into $|\Sigma|$ disjoint partitions, some of which may be empty: $F = \bigcup_{i=1,2,\ldots,|\Sigma|}\mathcal{PF}_i$, where the $\cup$ operation maintains the ordering in F. The size of $\mathcal{PF}_i$, the i-th partition, is simply the number of occurrences of the i-th symbol in the text string, T. This number can be pre-computed by using C, the count array used in constructing V from L (see 2). Let $\mathcal{Z}_i^{\mathcal{F}} = |\mathcal{PF}_i|$. Then, $\mathcal{Z}_i^{\mathcal{F}} = C[i+1]-C[i], \forall_{i=1,2,\ldots|\Sigma|-1}$, and $\mathcal{Z}_i^{\mathcal{F}} = u - C[i]$ **if** $i = |\Sigma|$. Similarly, for P, we have equivalent $|\Sigma|$ disjoint partitions. $\mathcal{PQ}_i^P$, the i-th partition contains the q-grams in $\mathcal{Q}_q^P$ that start with the i-th symbol. Let $\mathcal{Z}_i^{\mathcal{P}} = |\mathcal{PQ}_i^P|$. Let $\mathcal{Z}_{d_P}$ be the number of q-grams in $\mathcal{Q}_q^P$ that started with *distinct* characters — simply, the number of non-empty partitions in $\mathcal{Q}_q^P$. Thus, $\mathcal{Z}_{d_P} \leq m-q+1$ and $\mathcal{Z}_{d_P} \leq |\Sigma|$.

We note that a q-gram in one partition in $\mathcal{Q}_q^P$, say $(\mathcal{PQ}_i^P)$, can only match a q-gram in the corresponding partition in F, (i.e. a q-gram in $\mathcal{PF}_i$). Thus, we can limit the search to within only the relevant partition in F. Also, we only need to do the search for the first character **just once** for each distict symbol in $\mathcal{Q}_q^P$. The running time will be in: $O(\mathcal{Z}_{d_P}\log(u) + q\sum_{i\in\Sigma}\mathcal{Z}_i^{\mathcal{F}}\mathcal{Z}_i^{\mathcal{P}})$, where $\sum_{i\in\Sigma}\mathcal{Z}_i^{\mathcal{P}} = m-q+1$, and$\sum_{i\in\Sigma}\mathcal{Z}_i^{\mathcal{F}} = u$. Since we already know $\mathcal{Z}_i^{\mathcal{F}}$, the size of each partition in F, instead of doing a sequential search until the end of the current partition in F, we do a binary search. With this, the time for q-gram intersection can be reduced to: $O(\mathcal{Z}_{d_P}\log(u) + q\sum_{i\in\Sigma}\mathcal{Z}_i^{\mathcal{P}}\log\mathcal{Z}_i^{\mathcal{F}})$.

Algorithm 3

The final modification is based on the observation that we can obtain not only $\mathcal{Z}_i^{\mathcal{F}}$, but also the starting position of each distinct character in F using the count array, C. Since both C and F are sorted in the same order, we can determine the start position (sp_c) and end position (ep_c) of each character partition in F by using C. We could compute sp_c and ep_c as needed, or we can pre-compute them and store in $2|\Sigma|$ space. This reduces the first term in the complexity figure for **Algorithm 2**, leading to a running time of $O(\mathcal{Z}_{d_P}\log|\Sigma| + q\sum_{i\in\Sigma}\mathcal{Z}_i^{\mathcal{P}}\log\mathcal{Z}_i^{\mathcal{F}})$. We sumarize the foregoing with the following lemma:

Lemma 2: *q-gram* **intersection**. *Given $T = t_1t_2\ldots t_u$, transformed with the BWT, $P = p_1p_2\ldots p_m$, an alphabet with equi-probable symbols $\Sigma = \{\sigma_1,\sigma_2,\ldots,\sigma_{|\Sigma|}\}$ and the arrays F, Hr, C,* **Algorithm 3** *performs q-gram intersection in $O(|\Sigma|\log|\Sigma| + q(m-q)\log\frac{u}{|\Sigma|})$ time on average, and $O(|\Sigma|\log|\Sigma| + m^2\log\frac{u}{|\Sigma|})$ worst case.* $\diamondsuit$.

We omit the proof. Comapared to **Algorithm 2**, the improvement in speed produced by Algorithm 3 can be quite significant, since typically $|\Sigma| \ll u$ (e.g. for DNA sequences, or binary strings with **1's** and **0's**). We conclude the discussion on exact pattern matching with the following theorem:

Theorem 1: Exact pattern matching in BWT-compressed text. *Given a text string $T = t_1t_2\ldots t_u$, a pattern $P = p_1p_2,\ldots,p_m$, and an alphabet with equi-probable symbols $\Sigma = \{\sigma_1,\sigma_2,\ldots,\sigma_{|\Sigma|}\}$. Let Z be the* **bwt** *output when T is the input. There is an* **on-line** *algorithm that can locate* **all** *occurrences of P in T, using only Z (i.e. without full decompression)) in $O(u + m\log\frac{u}{|\Sigma|})$ time on average, and in $O(u + \log|\Sigma| + m\log u)$ worst case.*

Sketch of Proof. The proof is based on two facts: 1) For exact pattern matching, $q = m$ and $\mathcal{Z}_{d_P} = 1$; 2) We need to consider **just one** partition in F, since we have a single non-empty

partition in $\mathcal{Q}_q^P$. That is, the partition in F that starts with symbol $P[1]$. The O(u) time in the theorem is needed to compute the V, F, Hr arrays. $\diamondsuit$

The worst case extra space for the algorithms above is in $O(u + m + |\Sigma|)$.

5 Approximate pattern matching

There are two phases in our approach. In the first phase, we locate areas in the text that contain potential matches by performing some filtering operations using *appropriately sized* q-grams. In the second phase, we verify the results that are hypothesized by the filtering operations. The verification stage is generally slow, but usually, it will be performed on only a small proportion of the text. Thus, the performance of the algorithm depends critically on the number of hypothesis generated.

Locating potential matches. The first phase is based on a known fact in approximate pattern matching:

Lemma 3: k-approximate match (Baeza-Yates & Perleberg 1992) *Given a text T, a pattern P, ($m = |P|$), and k, for a k-approximate match of P to occur in T, there must exist at least one r-length block of symbols in P that form an* **exact match** *to some r-length substring in T, where $r = \lfloor \frac{m}{k+1} \rfloor$.* $\diamondsuit$

This is trivially the case for exact approximate matching, in which $k = 0$, and hence $r = m$. With the lemma, we can perform the filtering phase in three steps: 1) Compute r, the minimum block size for the q-grams. 2) Generate $\mathcal{Q}_r^T$ and $\mathcal{Q}_r^P$, the permissible r-grams from the text T, and the pattern P, respectively. 3) Perform q-gram intersection of $\mathcal{Q}_r^T$ and $\mathcal{Q}_r^P$.

Let $\mathcal{MQ}_k = \mathcal{Q}_r^P \cap \mathcal{Q}_r^T$, and $\eta_h = |\mathcal{MQ}_k|$. Let $\mathcal{MQ}_k^i$ be the i-th matching q-gram, Let $\mathcal{MQ}_k^i[j]$ be the j-th character in $\mathcal{MQ}_k^i$, $j = 1, 2, \ldots r$. Further, let $\overline{\mathcal{MQ}_k^{\mathcal{F}}}[i]$ be the index of the first character of $\mathcal{MQ}_k^i$ in the array of first characters, F. That is, $\overline{\mathcal{MQ}_k^{\mathcal{F}}}[i] = x$, if $F[x] = \mathcal{MQ}_k^i[1]$. We call $\mathcal{MQ}_k$ the *matching q-grams at k*. Its size is an important parameter for the next phase.

By **Lemma 1**, the indices j and $\overline{\mathcal{MQ}_k^{\mathcal{F}}}[i]$ can be generated in O(1) time. By the same lemma, step 2 above can be done in constant time and space. The cost of step 3, will grow slower than $\frac{m^2}{k+1} \log u$. The time for hypothesis generation is simply the time needed for q-gram intersection, where q is given by **Lemma 3**. Plugging these into the analysis for **Algorithm 3**, we have the following:

Lemma 4: Locating potential k-approximate matches. *Given $T = t_1 t_2 \ldots t_u$, transformed with the BWT, $P = p_1 p_2 \ldots p_m$, an alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$, and the arrays $F, andHr$. Let k be given. The hypothesis phase can be performed in time proportional to: $\mathcal{Z}_{d_P} \log(|\Sigma|) + \lfloor \frac{m}{k+1} \rfloor \sum_{i \in \Sigma} \mathcal{Z}_i^{\mathcal{P}} \log \mathcal{Z}_i^{\mathcal{F}}$* $\diamondsuit$

Verifying the matches. Here, we need to verify if the r-blocks that were hypothesized in the first phase are true matches. The time required will depend critically on η_h. We perform the verification in two steps: 1) Using Hr and F determine the matching neighborhood in T for each r-block in $\mathcal{MQ}_k$. The maximum size of the neighborhood will be $m + 2k$; 2) Verify if there is a k-approximate match within the neighborhood.

Let $\mathcal{N}_i$ be the neighborhood in T for $\mathcal{MQ}_k^i$, the i-th matching q-gram. Let t be the position in T where $\mathcal{MQ}_k^i$ starts. That is, $t = Hr[F[\overline{\mathcal{MQ}_k^{\mathcal{F}}}[i]]]$. The neighborhood is defined by the left and right limits: t_{left} and t_{right} viz: $t_{left} = t - k$ **if** $t - k \geq 1$; $t_{left} = 1$ otherwise; $t_{right} = t + m + k$ **if** $t + m + k \leq u$; $t_{right} = u$ otherwise. Hence, the i-th matching neighborhood in T is given by: $\mathcal{N}_i = T[t_{left} \ldots t \ldots t_{right}]$. Thus, $|\mathcal{N}_i| \leq m + 2k, \forall_i, i = 1, 2, \ldots \eta_h$. We then obtain a set of matching neighorhoods $\mathcal{S}_{\mathcal{MQ}} = \{\mathcal{N}_1, \mathcal{N}_2, \ldots \mathcal{N}_{\eta_h}\}$. Verifying a match within any given $\mathcal{N}_i$ can be done with any fast algorithm for k-approximate matching, for instance, Ukkonen's $O(ku)$ algorithm (Ukkonen 1985). The cost of the fist step in the verification will be in $O(\eta_h)$. The cost of the second step will thus be in $O(\eta_h k(m + 2k)) \leq O(\eta_h k(3m)) \approx O(\eta_h km)$.

Example. Let $T = abraca$ and $P = brace$, with $k = 1$. Then, $r = 2$, and permissible q-grams will be $\mathcal{Q}_2^P = \{ac, br, ce, ra\}$, $\mathcal{Q}_2^T = \{ab, ac, br, ca, ra\}$, yielding $\mathcal{MQ}_1 = \{ac, br, ra\}$, and

$\mathcal{N}_1 = [3 \ldots 6]; \mathcal{N}_2 = [1, \ldots, 6]; \mathcal{N}_3 = [2, \ldots, 6]$.

Matches will be found in $\mathcal{N}_1$ and $\mathcal{N}_2$ at positions 1 and 2 in T, respectively. $\diamondsuit$

We conclude this section with the following theorem:

Theorem 2: k-approximate matching in BWT-compressed text *Given a text string $T = t_1 t_2 \ldots t_u$, a pattern $P = p_1 p_2 \ldots p_m$, and an alphabet with equi-probable symbols $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$. Let Z be the* **bwt** *output when T is the input. There is an* **on-line** *algorithm that can locate* **all** *k-approximate matches of P in T, using only Z, (i.e. without full decompression nor with off-line index structures) in $O(u + |\Sigma| \log |\Sigma| + \frac{m^2}{k} \log \frac{u}{|\Sigma|} + k|\psi_{\mathcal{MQ}}|)$ time on average, ($|\psi_{\mathcal{MQ}}| \leq u$), and in $O(ku + |\Sigma| \log |\Sigma| + \frac{m^2}{k} \log \frac{u}{|\Sigma|})$ worst case.* $\diamondsuit$

Once again, we omit the proof for brevity. With the q-gram approach, we can treat exact pattern matching as no different from k-approximate pattern matching. We just have k=0, and hence no verification stage.

6 Results

7 Conclusion

Although the performance of the BWT has made it an important addition to the long list of text compression algorithms, very little has been reported in searching directly on text compressed with the BWT. The BWT with its sorted contexts however provides an ideal platform for compressed domain pattern matching. This paper has described algorithms for **on-line** exact and in-exact pattern matching directly on BWT-tranformed text.

The proposed algorithms could be further improved. For instance, the space requirement could be reduced by considering the compression blocks typically used in BWT-based compression schemes, while the time requirement could be further reduced by using faster pattern matching techniques for the q-gram intersection. We note that the methods as described basically operate on the output of the BWT transformation stage. One challenge is to extend the approach to operate beyond the BWT output, i.e. after the later encoding stages in the BWT compression process.

Appendix B

Evaluating Lossless Compression

This appendix describes work carried out towards the initial topic of this report, the evaluation of lossless compression methods with the Canterbury Corpus, and consists of a combination of the initial research proposal and the mid-year progress report.

1 The Calgary and Canterbury Corpora

The Canterbury Corpus (and its predecessor, the Calgary Corpus), are collections of "typical" files for use in the evaluation of lossless compression methods. The Canterbury Corpus consists of 11 files, shown in Table 1.1; an explanation of how the files were chosen, and why it is difficult to find "typical" files, can be found in (Arnold & Bell 1997). Previously, compression software was tested using a small subset of one or two "non-standard" files. This was a possible source of bias to experiments, as the data used may have caused the programs to exhibit anomalous behaviour.

Several criteria were identified for choosing the Canterbury Corpus, including that it should be *representative* of the files that are likely to be used by a compression system in the future; that it should be *widely available* and *contain only public domain material*; that it *not be larger than necessary* (to limit distribution costs); and that it should be *perceptibly and actually valid and useful*. With these criteria in mind, about 800 candidate files were identified as being acceptable and relevant for inclusion in a corpus, and compressed using a variety of compression techniques. For each group of files, a scatter plot of file size before and after compression was produced, and a line of best fit calculated using ordinary regression techniques. The "best" file in each category was identified as the file that was most consistently close to the regression line, and was then deemed suitable for inclusion.

These two corpora[1] have become *de facto* standards for evaluation of lossless compression methods; in fact, (Salomon 1998) goes so far as to state that the Calgary corpus is "traditionally used to test data compression programs". Certainly both corpora have gained a great deal of support in the compression community. The data in Table 1.2 was obtained by surveying the files used in papers and posters presented at the Data Compression Conference in 1998 and 1999, in a similar way to (Arnold & Bell 1997). The Canterbury and Calgary Corpora were by far the most common data sets used for lossless compression testing (the next most popular would be the `Jefferson` data set, which was used a total of only three times).

2 A system for maintaining the corpus results website

The web page at `http://corpus.canterbury.ac.nz/` provides information about the corpora, as well as links to download the files and a list of results for various compression algorithms and software. The pages include results, discussion, analysis, and links to other benchmarks.

[1]Yes, "corpora" is the plural of "corpus"

File	Category	Size
alice29.txt	English text	152089
asyoulik.txt	Shakespeare	125179
cp.html	HTML source	24603
fields.c	C source	11150
grammar.lsp	LISP source	3721
kennedy.xls	Excel spreadsheet	1029744
lcet10.txt	Technical writing	426754
plrabn12.txt	Poetry	481861
ptt5	CCITT test set	513216
sum	SPARC Executable	38240
xargs.1	GNU manual page	4227

Table 1.1: Files in the Canterbury Corpus

	1998		1999		Total	%
	papers	posters	papers	posters		
No test data used	26	20	20	27	93	32.6
Data not identified:						
—lossless	1	11			12	4.2
—lossy	5	13	2	3	23	8.1
Data not used by others	16	10	25	13	64	22.5
Lena	4	5	10	2	21	7.4
Calgary Corpus	5	4	2	2	13	4.6
Canterbury Corpus	2		3	3	8	2.8
Barbara	1	2	3	1	7	2.5
Goldhill	1	1	4	1	7	2.4
Peppers	1	2	2		5	1.8
Girl		1	2		3	1.1
Jefferson	2			1	3	1.1
Austen	2				2	0.7
Bike	1		1		2	0.7
Bridge			2		2	0.7
Brown	2				2	0.7
Coastguard	1		1		2	0.7
JPEG2000	1	1			2	0.7
King James Bible	2				2	0.7
LOB	2				2	0.7
Mandrill/Baboon		2			2	0.7
Palette Image Corpus	1		1		2	0.7
Shakespeare	2				2	0.7
Stefan		1	1		2	0.7
Wall Street Journal	2				2	0.7
Total	80	73	79	53	285	100.0

Table 1.2: Test data used in Data Compression Conference papers and posters

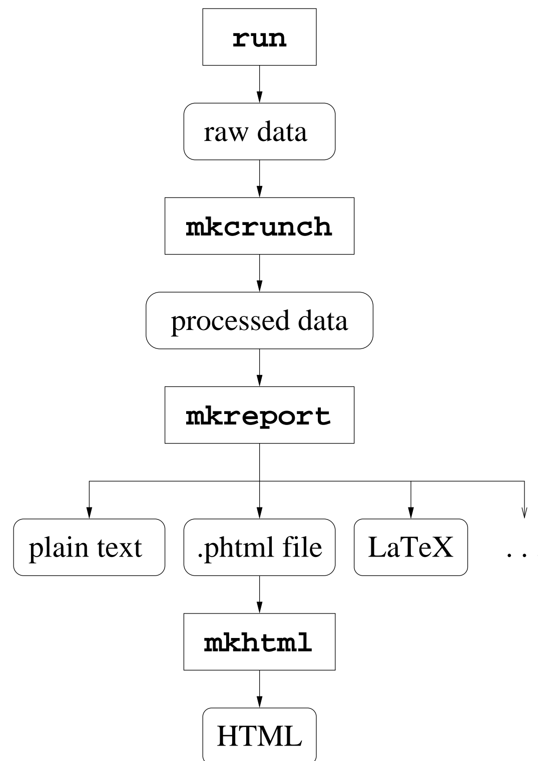


Figure 2.1: A typical maintenance run

Previously this results database was maintained by hand, but more recently it has been updated to allow automatic maintenance of the pages.

A series of scripts has been developed using Python and Unix shell scripts to provide a modular approach to maintaining the website, as shown in Figure 2.1. Each method/file combination is run several times by the `run` script, and the results are then "crunched" by the `mkcrunch` script, which calculates the mean results over all runs to avoid errors. The speed statistics are also "normalised" against a run of the standard UNIX `compress` utility, to avoid bias due to abnormal loads on system resources.

Once the "crunched" data has been produced, the `mkreport` script is used to massaged into a more useful format (`mkcrunch` produces code readable by Python, but not by many humans). Again, a modular approach has been applied, and `mkreport` passes its input to several Formatter objects, each specialised to a particular output format. Current formatters include HTML (which produces HTML tables), LaTeX(`tabular` format), and comma-delimited text (suitable for importing into spreadsheets such as Microsoft Excel).

Finally, the "`.phtml`" ("Python HTML") files produced by `mkreport` are run through `mkhtml` to add formatting changes like heading styles and footer data. This is so that the formatting of the site may be kept separate from the data it contains, and the look and feel of the site can be updated without re-running the compression tests (a lengthy process!)

It is possible to add results to the database without their being automatically generated by `run`. In such a case, all that is needed is a set of results that complies with the format of the `.crunch` files produced by `mkcrunch`. Each `.crunch` file contains results for each file in a given collection, as well as details about the run, including when it was completed, by whom, and on what system.

3 Issues with the current system

The Canterbury Corpus was primarily intended to replace the Calgary Corpus, which was showing its age. Unfortunately, the new collection of files has aged even more quickly than its predecessor, and may no longer fully reflect the range of files that need to be compressed. In particular, large files (in the order of tens or even hundreds of megabytes) are no longer uncommon, and there are many new file formats which could be investigated. To this end, several new "collections" have been added to the project, including a collection of large files.

Additionally, the files in the Canterbury Corpus were carefully chosen to examine "typical" behaviour on the part of the compression software. However, it is often useful to investigate "worst case" behaviour. An "artificial" collection has been added, consisting of files for which the compression methods may exhibit pathological or worst-case behaviour—for example, files full of randomly-generated characters, a file consisting of a single letter 'a', or a file with many repetitions of a short sequence.[2]

Perhaps a more serious concern is the issue of compression speed. For the corpus to provide reliable results, the compression tests should ideally be run under identical test conditions. In practice, this is never possible. Issues such as processor speed and available system resources can have a dramatic effect on the running time of a program. To counter this, all speed measurements are given relative to the time taken by the standard UNIX `compress` utility, as mentioned above. This ensures that the times listed in the results remain consistent.

However, if an author of a new compression program wishes to add his or her results to the list, there may be genuine reasons why the software cannot be tested in this way. Perhaps the software was written for a different architecture, or the program is unable to be released for reasons of commercial sensitivity. In such cases it will be necessary to assess the speed of such software *absolutely*—that is, without respect to the system on which the tests were run. Possible solutions to this problem are discussed in Section 7.3.

It will also be necessary to be able to verify that the results submitted for inclusion on the website were actually generated by the compression software. Such verification may be possible by means of testing software which outputs an encrypted version of the results.

4 Statistical Analysis

4.1 Analysis of existing Corpus data

An additional statistical investigation has been performed on the Corpus results, including the addition of standard deviation reporting to the results website. Scripts have also been developed for the automatic generation of graphs of the results from the Corpus, which may be made available online.

4.2 Pseudo-random files

A considerable investigation has been undertaken into the behaviour of compression algorithms on *random* files (that is, files with no discernibly deliberate patterns), to approximate the behaviour of such compression programs over *all files*.

The first part of this investigation was a study of the distribution of the compressed sizes of random files, based on the observation that the average size of compressed text over all files of length l must be *at least* l. To this end, a collection of empirical data has been made over a large set of randomly-generated files, and initial graphs have been generated to ascertain areas of further study; however, detailed conclusions have yet to be drawn.

[2]Such artificial files proved very useful in the investigation of a compression scheme brought to the curators of the corpus for testing. Its author claimed that the program was capable of compressing any file to a nearly constant ratio of around 7%—which is clearly impossible. Suspiciously, it was even able to compress completely random files to this remarkable ratio. The only drawback to the system was a mysterious loss of free hard drive space, roughly equal to the size of the file being compressed. . .

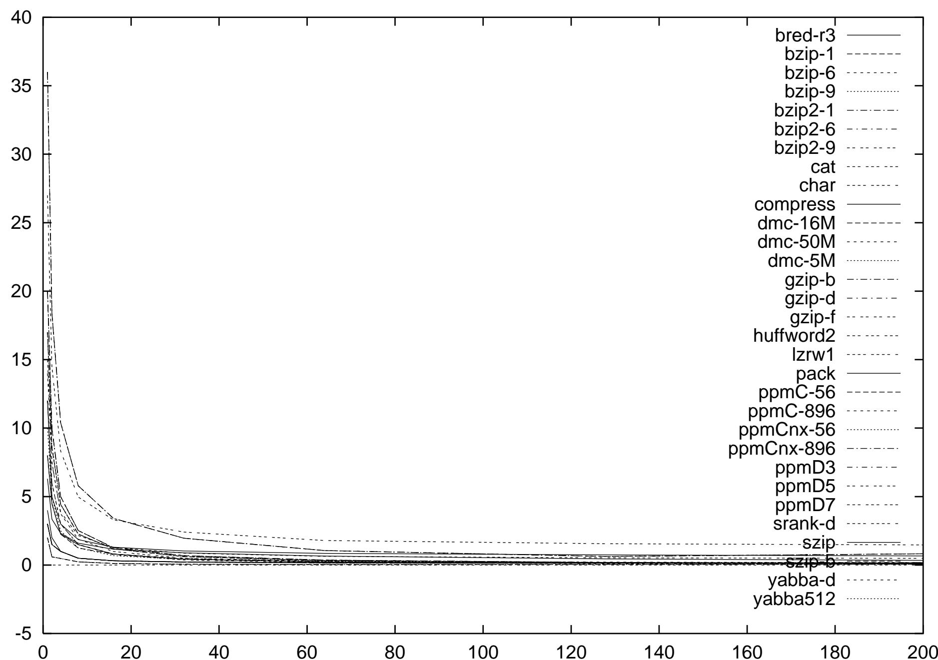


Figure 4.2: Graph of average 'bloat' on compressed files. The y-axis measures the per-byte file size difference between compressed files and the uncompressed files, on average

A second study of random files compared the sizes of files before and after compression. Figure 4.2 shows the results of this experiment. A series of sets of pseudo-random files of increasing sizes was generated, and compressed by each compressor in the Corpus compressor set. The difference between the average output size of these files and the input size was divided by the input size to obtain a 'bytes-per-byte' measure of the difference in file size. Results showed that this difference was very large for small files (the worst case was 36 bytes-per-byte for `bzip2-1` on a single-character file), but decreased almost exponentially, so that for files of more than about 80 characters, most of the compressors had stabilised at under two bytes-per-byte of 'bloat'.

It must be remembered that these tests were performed on *random* files, and as such, the idea of 'compression' is almost farcical, since compression systems, by nature, rely on the existence of patterns in the input. Nevertheless, by examining the behaviour of these compressors on random input, we can gain an insight into their behaviour over all inputs.

5 Performance Measurement

Some headway has been made on the issue of performance of compression software. A selection of programs designed to exhibit 'compression-like' behaviour has been developed, and it is hoped that further experimentation with these programs will reveal a suitable method for determining the relative speed of compression software.

6 Reliability of Results

Tentative research has been made into the issue of the verification of compression results from third parties, especially in light of a recent 'compression scam'. However, this initial research has proved largely fruitless, due to the inherently complex nature of such verification, and this topic has been postponed, at least for the moment.

7 Proposed research

7.1 User interface

To simplify the process of maintaining the results website, a graphical user interface has been developed which allows the user to perform various mundane or complex tasks with relative ease. For example, by selecting a menu item it is possible to generate the reports, process the HTML files, move them onto the server, set correct file permissions and remove “artifact” files created in the process. By automating such tasks, the capacity for inadvertent error is greatly reduced.

There is still substantial room for improvements to the interface, including adding the facilities to add new files or compression methods.

7.2 Data analysis

At present the data presented on the results website is relatively rudimentary—only the raw results and the statistical mean are given. However, given the automated nature of the maintenance process, it would be a simple matter to add other statistics (for example, the standard deviation of the compression ratios for a given algorithm) to the site.

7.3 Cross-platform benchmarking

As mentioned above, the question of performance measurement poses a definite problem. The current system, which uses `compress` to measure the relative speeds of compression software, is a good solution for many existing methods, but may lack wider applicability because current software tends to be geared more and more towards the Windows/Intel platform, and as such will be incompatible with the current UNIX-based benchmarking system.

Although `compress` is available for a number of different platforms, it is difficult to tell whether the UNIX version will be “the same” as the DOS version, and so on. In fact, even when compiled from the same source code, two versions of `compress` may differ in performance because of optimisation routines built into the source code by means of conditional `#define`s. Thus the end result is a program that runs as fast as possible on each target platform, but whose internal behaviour may vary to take advantage of each platform.

It is proposed that research be undertaken to investigate whether such optimisations make any real difference to the performance of `compress` (or any other compression software we may wish to test). One possible method of testing this would be to write a program that compiles on as many platforms as possible, and whose behaviour does not depend on idiosyncracies of the target platform. Such software need not necessarily *be* a compression program, as long as it performs the same *sort* of tasks, e.g. complex array processing, file input and output and sub-byte processing.

8 Conclusions

Although the Canterbury and Calgary Corpora are already widely used, there is significant opportunity to make results more accessible via the Canterbury Corpus Website. In particular, more accurate time measurements are called for, as well as methods for verifying results remotely.

Bibliography

Adjeroh, D., Mukherjee, A., Bell, T., Zhang, N. & Powell, M. (2001), Pattern matching in BWT compressed text. Expected to be submitted to DCC 2002.

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. (1997), A corpus for the evaluation of lossless compression algorithms, *in* 'Data Compression Conference', IEEE Computer Society Press, pp. 201–210.

Baeza-Yates, R. & Perleberg, C. (1992), 'Fast and practical approximate string matching', *Proceedings, Combinatorial Pattern Matching, LNCS 644* pp. 185–192.

Balkenhol, B., Kurtz, S. & Shtarkov, Y. (1999), 'Modifications of the Burrows and Wheeler data compression algorithm', *Proceedings, IEEE Data Compression Comference* .

Bell, T., Adjeroh, D. & Mukherjee, A. (2001), Pattern matching in compressed texts and images. Draft of May 23, 2001 (Submitted to ACM Computing Surveys).

Bell, T. C. (1986), 'Better OPM/L text compression', *IEEE transactions on communications* **COM-34**, 1176–1182.

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

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

Bratley, P. & Choueka, Y. (1982), 'Processing truncated terms in document retrieval systems', *Information Processing and Management* **18**(5), 257–266.

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.

Cleary, J. G., Teahan, W. J. & Witten, I. H. (1995), Unbounded length contexts for PPM, *in* 'Data Compression Conference', IEEE Computer Society Press, pp. 52–61.

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

Cormen, T. H., Leiserson, C. E. & Rivest, R. L. (1989), *Introduction to Algorithms*, MIT Press.

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

Fenwick, P. M. (1996*a*), Block sorting text compression—final report, Technical report, Department of Computer Science, The University of Auckland.

Fenwick, P. M. (1996*b*), 'The Burrows-Wheeler Transform for block sorting text compression', *The Computer Journal* **39**(9), 731–740.

Ferragina, P. & Manzini, G. (2000), Opportunistic data structures with applications, *in* 'IEEE Symposium on Foundations of Computer Science'.

Ferragina, P. & Manzini, G. (2001), An experimental study of a compressed index. Part of this work appeared in ACM-SIAM Symposium on Discrete Algorithms.

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

Knuth, D. E., Morris, Jr, J. H. & Pratt, V. R. (1977), 'Fast pattern matching in strings', *SIAM Journal of Computing* **6**(1), 323–350.

Manber, U. (1997), 'A text compression scheme that allows fast searching directly in the compressed file', *ACM Transactions on Information Systems* **15**(2), 124–136.

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.

Mukherjee, A., Bell, T., Powell, M., Adjeroh, D. & Zhang, N. (2001), Searching BWT compressed text with the Boyer-Moore algorithm and binary search. Expected to be submitted to DCC 2002.

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

Piatetsky, G. & Frawley, W. J. (1991), *Knowledge Discovery in Databases*, MIT Press.

Powell, M. (2001), 'The Canterbury corpus', www.corpus.canterbury.ac.nz.

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

Sadakane, K. (2000*b*), Unifying Text Search and Compression—Suffix Sorting, Block Sorting and Suffix Arrays, PhD thesis, University of Tokyo.

Sadakane, K. & Imai, H. (1999), A cooperative distributed text database management method unifying search and compression based on the Burrows-Wheeler transformation, *in* 'Advances in Database Technologies', number 1552 *in* 'Lecture Notes in Computer Science', pp. 434–445.

Salomon, D. (1998), *Data Compression: the complete reference*, Springer-Verlag.

Seward, J. (2000), 'The bzip2 and libbzip2 official home page', http://sources.redhat.com/bzip2/index.html.

Shibata, Y., Kida, T., Fukamachi, S., Takeda, T., Shinohara, A., Shinohara, S. & Arikawa, S. (1999), Byte-pair encoding: A text compression scheme that accelerates pattern matching, Technical report, Department of Informatics, Kyushu University, Japan.

Ukkonen, E. (1985), 'Finding approximate patterns in strings', *Journal of Algorithms* **6**, 132–137.

Witten, I. H., Moffatt, A. & Bell, T. C. (1999), *Managing Gigabytes*, second edn, Morgan Kaufmann.

Ziv, J. & Lempel, A. (1977), 'A universal algorithm for sequential data compression', *IEE Transactions on Information Theory* **IT-23**(3), 337–343.

Ziv, J. & Lempel, A. (1978), 'Compression of individual sequences via variable rate coding', *IEE Transactions on Information Theory* **IT-24**(5), 530–536.