ALGORITHM ENGINEERING: STRING PROCESSING

Thomas Berry

A DISSERTATION

in

COMPUTING AND MATHEMATICAL SCIENCES

Presented to the Faculties of Liverpool John Moores University in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2002

Dr. S. Ravindran

Supervisor of Dissertation

Prof. M. Merabti

Graduate Group Chairperson

DEDICATION

To Mum and Dad

ACKNOWLEDGEMENTS

I would like to express thanks to my principal supervisor Dr. S. Ravindran for his help and advice throughout my research. I would also like to thank my second supervisor Dr. C. Bamford for his help and support. I would also like to thank my third supervisor Prof. M. Merabti for his encouragement and support throughout my research. Nick Huxley and Marc Siegar for talking me into doing a Ph.D. in the first place.

I would like to thank my colleagues in the school of Computing and Mathematical Sciences for their continued support, especially those of my fellow researchers who have made it more fun than it would otherwise been, Bob Askwith, Mike Baskett, Chris Bewick, Henry Forsyth, Damian Gregory, David Gresty, John Haggerty, Ste Keller, Toni Reyes Moro and Ewan Smith.

Away from the office I would like to thank my friends and family who have suffered my rants about my research. Special thanks must be made to the 'MI7 quiz team' Allen Dunn, Chris Gill, Paul Lynam, Matt Pickles and Marc Siegar. The cinema buffs Dave Barnes, Kathryn Holmes, Nick Huxley, Jo Redmond and Neil Stafford.

I would also like to thank the 2000/2001 Liverpool Squad for winning the treble and giving me many nights of footballing ecstasy.

Abstract

The string matching problem has attracted a lot of interest throughout the history of computer science, and is crucial to the computing industry. The theoretical community in Computer Science has a developed a rich literature in the design and analysis of string matching algorithms. To date, most of this work has been based on the asymptotic analysis of the algorithms. This analysis rarely tell us how the algorithm will perform in practice and considerable experimentation and fine-tuning is typically required to get the most out of a theoretical idea.

In this thesis, promising string matching algorithms discovered by the theoretical community are implemented, tested and refined to the point where they can be usefully applied in practice. In the course of this work we have presented the following new algorithms. We prove that the time complexity of the new algorithms, for the average case is linear. We also compared the new algorithms with the existing algorithms by experimentation.

- We implemented the existing one dimensional string matching algorithms for English texts. From the findings of the experimental results we identified the best two algorithms. We combined these two algorithms and introduce a new algorithm.
- We developed a new two dimensional string matching algorithm. This algorithm uses the structure of the pattern to reduce the number of comparisons required to search for the pattern.
- We described a method for efficiently storing text. Although this reduces the size of the storage space, it is not a compression method as in the literature. Our aim is to improve both space and time taken by a string matching algorithm. Our new algorithm searches for patterns in the efficiently stored text without decompressing the text.
- We illustrated that by pre-processing the text we can improve the speed of the string matching algorithm when we search for a large number of patterns in a given text.
- We proposed a hardware solution for searching in an efficiently stored DNA text.

Contents

1 Introduction **1**

1.1 Novel aspects of the thesis 8

2 Analysis of Algorithms **11**

2.1 Introduction 11

2.2 Asymptotic Analysis of algorithms 12

2.3 Big-Oh notation 14

2.4 Theoretical Versus Practical evaluation of algorithms 16

3 String matching algorithms **21**

3.1 Introduction 21

3.1.1 Knuth, Morris and Pratt algorithm and its derivatives 22

3.1.2 Boyer-Moore algorithm and its derivatives 24

3.1.3 Other one dimensional string matching algorithms 30

3.1.4 Approximate string matching algorithms 32

4 A new string matching algorithm 34

4.1 Introduction 34

4.2 Experimental results of the existing algorithms 34

4.3 The new algorithm (BR) 38

4.4 Average case analysis of the BR algorithm 43

4.5 Experimental results and comparisons with the BR algorithm 45

4.6 Conclusions 55

5 Two dimensional string matching algorithms 56

5.1 Introduction 56

5.2 Existing two-dimensional string matching algorithms 56

5.3 The New Algorithm (2D-BR) 58

5.4 Average case analysis of the 2D-BR algorithm 61

5.5 Practical evaluation of the algorithms 66

5.6 Conclusions 70

6 Compression algorithms 71

6.1 Introduction 71

6.2 Huffman Encoding 73

6.3 Lempel-Ziv encoding and its derivatives 81

6.4 Byte pair encoding 90

7 String matching in an efficiently stored DNA text 91

7.1 Introduction 91

7.2 Efficient storage of a DNA text 92

7.3 Comparison with existing compression algorithms 93

7.4 Searching in the efficiently stored file - the DS algorithm 94

7.5 The average running time of the DS algorithm 99

7.6 Comparison with existing string matching algorithms 108

7.7 Conclusions 114

8 A linear time string matching algorithm on average with efficient text storage 115

8.1 Introduction 115

8.2 Efficient storage of a text 116

8.3 Comparison with existing compression algorithms 117

8.4 Searching in a text with efficient storage 118

8.5 The average running time 123

8.6 Comparison with existing string matching algorithms 131

8.7 Conclusions 134

9 String matching via pre-processing the text 136

9.1 Introduction 136

9.2 The algorithm 137

9.3 Average case analysis 141

9.4 Recording the positions of more than one character 142

9.4.1 Comparing the pre-processing algorithm with $x = 1$ and $x = 2$ 143

9.5 Comparing the new algorithm with the existing algorithms 146

9.6 Pre-processing DNA using a mapping function 149

9.7 Conclusions 154

10 Searching in an Efficiently Stored DNA Text Using a Hardware Solution 156

10.1 Introduction 156

10.2 Investigation into a hardware only solution to the string matching problem 156

10.3 Searching for multiple strings 161

10.4 Hardware acceleration 164

10.5 Hardware Implementation of string Matching 166

10.6 Conclusions 169

11 Conclusions and Further Work 171

11.1 Applications of Algorithm Engineering 171

11.2 Algorithm Engineering and String processing 173

11.3 Memory Management 179

11.4 Hardware implementation of string processing algorithms 182

List of Figures

2.1 *Algorithm to sum n numbers* 13

2.2 *Algorithm to find an entry in a list* 14

3.1 *Shift in the Knuth-Morris-Pratt algorithm (v border of u and* $d \neq c$*)* . 23

3.2 *Boyer-Moore good-suffix shift, u reappears preceded by a character different from 'b'* 26

3.3 *Boyer-Moore good-suffix shift, only a prefix of u reappears in P* 26

3.4 *Boyer-Moore bad-character shift, a appears in P* 27

3.5 *Boyer-Moore bad-character shift, a does not appear in P* 27

4.1 *Values for a shift using the BR algorithm* 38

4.2 *The shift values for the pattern 'onion' using the BR algorithm* 39

4.3 *The structure of the CON table for the pattern 'onion'* 40

5.1 *Each X denotes a sample point* $(T[(i+1)m][(j+1)m]$ *in the matrix that is compared with the Frequency array where T is a* 6×6 *matrix and P is a* 2×2 *matrix.* 59

5.2 *Shaded area is compared with the first row of the pattern using the BR algorithm* 60

6.1 *END and F added to the Huffman tree* 75

6.2 *E added to the Huffman tree* 75

6.3 *D added to the Huffman tree* 76

6.4 *C and B are added but not connected to the main tree* 76

6.5 *C and B subtree connected to the Huffman tree* 77

6.6 *A added to the Huffman tree and the tree is complete* 78

7.1 *The expressions for the pattern* **CGGTAGA** 96

7.2 *The expressions generated for a general pattern* 96

7.3 *The expressions generated for the pattern* **ACAC** 98

7.4 *The pattern blocks that would be chosen for the pattern* **ACAC** 98

7.5 *The expressions generated for the pattern* **ACGTAT** 100

7.6 *The number of wildcards in pattern-blocks, for* $m = 6$ 100

7.7 *The number of wildcards in pattern-blocks, for* $m = 16$ 101

7.8 *The number of wildcards in pattern-blocks, for* $m = 15$ 102

7.9 *The number of wildcards in pattern-blocks, for* $m' = 4$ 103

7.10 *The probability of the algorithm making at least 1 or 2 comparisons at an attempt.* 103

7.11 *The number of wildcards in pattern blocks, for* $m' = 4$ 105

7.12 The 62 patterns that don't contain a wildcard 107

7.13 *The 42 patterns that contain one or more wildcards* 107

8.1 *Expressions for a pattern* $P_1P_2..\ P_m$ *when* m *mod* $8 = 0$ 119

8.2 *The number of wildcards in pattern-blocks for* $m = 34$ 124

8.3 *The number of wildcards in pattern-blocks for* $m = 23$ 125

8.4 *The number of wildcards in pattern-blocks for* $m = 10$ 126

8.5 *The number of wildcards in pattern-blocks, for* $m' = 10$ 128

9.1 *The initial values for Next, Freq and First* 138

9.2 *The values for Next, Freq and First after considering the A at position 4*138

9.3 *The values for Next, Freq and First after considering the B at position 3*139

9.4 *The values for Next, Freq and First after considering the A at position 2*139

9.5 *The values for Next, Freq and First after considering the A at position 1*139

9.6 *The values for Next, Freq and First after considering the B at position 0*140

10.1 *The C code for searching for occurrences of a single pattern in a given text* 157

10.2 *Comparison of input stream against target* 158

10.3 *The use of the mask to reduce the number of bits compared* 160

10.4 *The steps required to determine whether the target matches the current data* 161

10.5 *An algorithm to search for multiple patterns in a single text* 162

10.6 *Illustration of Figure 10.5* 163

10.7 *Simplified version of the components to be implemented in hardware* . 168

List of Tables

2.1 *The size of input for a number of problems* 13

2.2 *Common time complexities for algorithms and their informal names* . 16

2.3 *Time in seconds to sort a randomised list* 19

2.4 *Time in seconds for sorting ordered lists for n = 2500* 19

4.1 *The number of comparisons in 1000's for searching a text of 200,000 words (1670005 characters).* 36

4.2 *The Shift Table for the pattern 'onion'* 41

4.3 *mismatch shift on ST(CON[n], CON[t]) = ST(1, 0) = 1* 42

4.4 *mismatch shift on ST(Con [t]], CON[ ]) = ST(0, 0) = 7* 42

4.5 *mismatch shift on ST(CON[s], CON[t]) = ST(0, 0) = 7* 42

4.6 *mismatch shift on ST(CON[ ], CON[o]) = ST(0, 2) = 6* 42

4.7 *So the pattern 'onion' has been matched and the text is exhausted* . . 43

4.8 *The number of comparisons in 1000's for searching Text A of 10,000 words (83360 characters)* 46

4.9 *The number of comparisons in 1000's for searching Text B of 10,000 words (83425 characters)* 47

4.10 *The number of comparisons in 1000's for searching a text of 50,000 words (417923 characters)* 48

4.11 *The number of comparisons in 1000's for searching a text of 100,000 words (834381 characters)* 49

4.12 *The average (of Tables 4.8- 4.11) percentage difference in the number of comparisons between existing algorithms and the BR algorithm* . . 51

4.13 *The average shift and the user time in seconds* 53

4.14 *User times in seconds for the eight chosen texts* 54

4.15 *The number of words and characters of the texts used in Table 4.14* . 54

5.1 *Estimated number of comparisons taken* 65

5.2 *Actual number of comparisons taken* 66

5.3 *Time in seconds to search for 50 matrices when* $\sigma = 256$ 67

5.4 *Time in seconds to search for 50 matrices when* $\sigma = 128$ 67

5.5 *Time in seconds to search for 50 matrices when* $\sigma = 64$ 68

5.6 *Time in seconds to search for 50 matrices when* $\sigma = 32$ 68

5.7 *Time in seconds to search for 50 matrices when* $\sigma = 16$ 68

5.8 *Time in seconds to search for 50 matrices when* $\sigma = 8$ 69

5.9 *Time in seconds to search for 50 matrices when* $\sigma = 4$ 69

5.10 *Time in seconds to search for 50 matrices when* $\sigma = 2$ 69

6.1 *The frequency of each of the characters in the text.* 74

6.2 *The bit patterns and their lengths for an Huffman encoding of the frequencies in Table 6.1* 78

6.3 *Position of the characters in the string 'AAGTCTGTCA'* 82

6.4 *LZ77 encoding of the string 'AAGTCTGTCA'* 83

6.5 *The positions of the characters in the string 'AAGTCTGTCTCA'* . . 84

6.6 *The LZ78 encoding of the string 'AAGTCTGTCTCA'* 85

6.7 *The LZW encoding of the string 'AAGTCTGTCTCA'* 87

7.1 *The size of the compressed generated when using compress and gzip* . 94

7.2 *The number of comparisons performed by the DS algorithm for each of the 10 efficiently stored DNA texts* 108

7.3 *Time in seconds to search for all the patterns without wildcards in the given texts* 109

7.4 *Time in seconds to search for all the patterns without wildcards in the given texts* 110

7.5 *Time in seconds to search for all the patterns with wildcards using bit masking in the given texts* 110

7.6 *Time in seconds to search for all the patterns with wildcards using bit masking in the given texts* 111

7.7 *Time in seconds to search for all the patterns with wildcards in the given texts* 113

7.8 *Time in seconds to search for all the patterns with wildcards in the given texts* 113

8.1 *Compressed text sizes for a random text of 500,000 bytes* 118

8.2 *The expressions considered at each comparison* 122

8.3 *The associated probabilities for α and β for each base case* 128

8.4 *Estimated versus actual number of comparisons of our BRS algorithm* 131

8.5 *Estimated versus actual number of comparisons of our BRS algorithm* 131

8.6 *Times in seconds to search for 100 random patterns in each given text with $\sigma = 2$* 132

8.7 *Times in seconds to search for 100 random patterns in each given text with $\sigma = 4$* 133

8.8 *Times in seconds to search for 100 random patterns in each given text with $\sigma = 8$* 133

8.9 *Times in seconds to search for 100 random patterns in each given text with $\sigma = 16$* 133

8.10 *Times in seconds to search for 100 random patterns in each given text with $\sigma = 32$* 134

9.1 *Time taken (in seconds) to search for the UNIX dictionary in the given texts* 144

9.2 *The number of attempts and comparisons taken when searching for the UNIX dictionary in the given texts* 145

9.3 *Time taken (in seconds) to search for the UNIX dictionary in the given texts using the HOR and BR algorithms* 146

9.4 *Time taken to build Next, First and Freq* 147

9.5 *Number of patterns that are required to be searched for, for the* $x = 2$ *method to be the fastest* 149

9.6 *Time taken in seconds to search for the 256 possible DNA patterns of length 4 including any pre-processing time.* 152

9.7 *Time taken in seconds to search for the 4096 possible DNA patterns of length 6 including any pre-processing time* 152

9.8 *Time taken to search for 5000 DNA patterns of length 10 including any pre-processing time* 153

9.9 *Time taken in seconds for pre-processing* 154

Table of acronyms

2D-BR	Two Dimensional Berry-Ravindran algorithm
AG	Apostolico-Giancarlo algorithm
BF	Brute Force algorithm
BM	Boyer-Moore algorithm
BR	Berry-Ravindran algorithm
BRS	Berry-Ravindran Search algorithm
BY	Baeza Yates algorithm
COL	Colussi algorithm
DFA	Deterministic Finite Automata
DS	DNA Search algorithm
GG	Galil-Giancarlo algorithm
HOR	Horspool algorithm
KMP	Knuth-Morris-Pratt algorithm
LDI	Liu, Du and Ishi algorithm
MS	Maximal Shift algorithm
NR	Navarro Raffinot algorithm
NSN	Not So Naive algorithm
QS	Quicksearch algorithm
RAI	Raita algorithm
RF	Reverse Factor algorithm
SMI	Smith algorithm
TBM	Turbo Boyer-Moore algorithm
ZT	Zhu-Takaoka algorithm

Chapter 1

Introduction

Theory and theoreticians have played a major role in the development of the field of computer science. Theoreticians have developed a set of basic concepts and methodologies that transcend application domains. These contributions include automata and natural language theoretic models, data structures and algorithms, methodologies for evaluating algorithm performance, the theory of NP completeness [64], logics of programs and correctness proofs and methods of public-key cryptography [56, 114].

Within theoretical computer science algorithms are usually evaluated by metrics such as their asymptotic worst case running time or average case running time or their competitive ratio. Many groundbreaking results have been proved according to the worst and/or average case running time of the algorithm. These metrics rarely tell us how well the algorithm will perform in practice. This is because the metrics are not sufficiently accurate to predict actual performance. The situation can be improved using models that take into account more details of the system architecture

and factors such as data movement and interprocessor communication, but even then considerable experimentation and fine-tuning is typically required to get the most out of a theoretical idea. An effort must be made to ensure that promising algorithms discovered by the theory community are implemented, tested and refined to the point where they can be usefully applied in practice [69, 77, 131]. During the practical evaluation of the algorithm it is possible to learn what affects the speed of the algorithm. It is possible to then modify the algorithm and remove any features that are slowing the algorithm down. This experimental testing and tuning of algorithms is known as *Algorithm Engineering.*

If an algorithm has a better theoretical evaluation than another algorithm we would expect that the algorithm would be faster in practice. Until we compare both algorithms in practice we can claim nothing. Although practical evaluation would seem dependant upon the environment, comparing algorithms in one environment is a good indication of how the algorithm may perform in other practical environments.

The method used for the theoretical evaluation of algorithms is known as the asymptotic analysis of algorithms using Big-Oh notation and is described in Chapter 2. When we theoretically analyse algorithms we consider the amount of time taken for the algorithm to complete its task and the amount of space or memory required for the task. Big-Oh notation gives a guideline of how the algorithm should perform. There are three main types of analysis used in the evaluation of algorithms. They are

the best case, worst case and average case analysis of the algorithm.

The best case is the optimal performance of an algorithm for a given data set. The best case is rarely used and does not show how the algorithm would perform on other input. We would expect the average case analysis of an algorithm to show the average performance of an algorithm over a range of input. A worst-case analysis of the algorithm gives the upper bound of the number of steps taken by the algorithm.

For example consider the sorting algorithms *Quicksort* and *Mergesort*, the worst-case analysis shows that the *Mergesort* algorithm is better than the *Quicksort* algorithm. When they are evaluated practically the *Quicksort* algorithm is faster than the *Mergesort* algorithm. The time taken for the algorithms to sort randomly ordered lists of different sizes are taken from [120] and from these results we would chose the *Quicksort* algorithm as it is up to eight times faster than the *Mergesort* algorithm. For sorted lists the *Quicksort* algorithm suffers a drop in performance. This is due to the worst-case for the *Quicksort* algorithm being used. For the sorted lists the *Mergesort* algorithm is up to five times faster than the *Quicksort* algorithm. If we were to choose an algorithm to sort a list we would want to use the *Quicksort* algorithm for unordered lists, and the *Mergesort* algorithm for ordered lists. This example clearly shows that an evaluation of algorithms in both theory and practice is required before choosing an algorithm for a specific task. These results are described in Chapter 2.

The worst-case for an algorithm may never occur or occur rarely for an algorithm in practice. By implementing the algorithms and evaluating them it is possible to see what features affect the time taken by the algorithm. It is then possible to fine tune the algorithms to produce faster algorithms using the knowledge gained from a practical evaluation.

String matching is the searching for a pattern P of length m in a Text T of length n. The pattern is aligned with the beginning of the text and the pattern characters are compared with the corresponding text characters. This is called an *attempt*. After a mismatch or match the pattern is shifted to the right and comparisons are again made between the pattern and text characters. A number of string matching algorithms have been developed [36, 84, 47]. The main difference among the algorithms is in the order the comparisons are made and how far the pattern is shifted to right after a mismatch or match. When searching for a pattern in a text we have to search through all of the text. Full descriptions of one-dimensional string matching algorithms used in this thesis are given in Chapter 3.

The string matching algorithms are practically evaluated in Chapter 4 using two different methods. We count the number of comparisons taken by each algorithm record the real time taken by the algorithms on the chosen data sets. The texts are written in English and the patterns are English words. From these tests we choose the best two algorithms. Using the features of these algorithms we designed a new

algorithm, the BR algorithm.

The average case analysis of the BR algorithm is given and is proven to be linear. The new algorithm is then compared to the nine fastest existing algorithms and the KMP algorithm by experimentation. Both the number of comparisons taken and the time taken by the BR algorithm are compared with the existing algorithms.

String matching can be two-dimensional as well as one-dimensional. Two-dimensional string matching usually involves searching for a pattern matrix P of dimensions $m_1 \times m_2$ in a text matrix T of dimensions $n_1 \times n_2$. In Chapter 5 we discuss two-dimensional string matching algorithms and describe a new algorithm for the task. We prove that the new algorithm has a linear average case time complexity. The algorithms are both theoretically and practically evaluated. The practical evaluation again records the number of comparisons taken and the time taken for the algorithms to perform specific searches on chosen data sets. We include results for alphabets and pattern of different sizes. The new algorithm is the fastest algorithm when the alphabet set is large (≥ 64).

We can decrease the length of the text by compressing it. Text compression is the re-representation of the characters in a text so that they take less space. There are many different text compressions algorithms available and we describe the most widely used algorithms in Chapter 6. They are Huffman encoding [73], Lempel-Ziv 77 [140], Lempel-Ziv 78 [141] and Lempel-Ziv-Welch [134]. The time taken to compress

the text can be a factor when choosing a text compression algorithm but generally the algorithm that offers the greatest amount of compression is used. Searching in the compressed file is called *Compressed string matching.* By performing compressed string matching we would expect to decrease the amount time taken to search for a pattern in a file.

We consider searching in the 'compressed' DNA text in Chapter 7. The DNA alphabet consists of four characters A, C, G and T. A common search using string matching in DNA texts is for boundary or cutting locations. These boundaries or cutting locations are strings of DNA characters and strings may contain wildcard characters, which can represent two or more of the DNA characters. Most DNA texts use eight bits to store each of the DNA characters. As there are only four characters we can represent each of the characters using only two bits. This efficient storage method would guarantee to reduce the size of the original DNA text by 75%.

We compare our new efficient storage method with existing compression algorithms and find that our method is competitive with the existing methods. We describe a new algorithm for searching in the efficiently stored DNA text. We prove that the average case time complexity of our new algorithm is linear. The new algorithm is compared practically with the existing string matching algorithms and the results show that our new algorithm is roughly more than three times faster than the existing string matching algorithms. This is mainly due to the use of the efficient

storage method.

The new algorithm can search for patterns with or without wildcards. The existing algorithms have to be modified to handle wildcards. The modified algorithms are compared to the new algorithm and new algorithm is still roughly three times faster than the existing algorithms.

We expand our idea of efficiently storing DNA texts to include storing texts with an alphabet of any size in Chapter 8. We give a method for efficiently storing texts with an alphabet of $\leq$ 128 characters. This new method will reduce the text to $\frac{\lceil \log_2 \sigma \rceil}{8}$ of its original size, where σ is the size of the alphabet set. We describe a new string matching algorithm that will search for patterns in the efficiently stored text. We prove that the new algorithm has a linear average case time complexity. We compare our new string matching algorithm with the existing string matching algorithms and find that as the alphabet increases the performance of the new string matching algorithm degrades. This is due to the amount of space that is saved by using the efficient storage method being reduced as the size of the alphabet increases.

The algorithms described in Chapters 3 and 4 require pre-processing of the pattern that they are searching. More over, these algorithms require the reading of the text into an array before searching for the pattern. In Chapter 9 we describe a new string matching algorithm which requires the pre-processing of the text. This means that we record the positions of characters or strings of characters in the text. Using this

method we are able to reduce the number of attempts required to search a text. This results in fewer comparisons and faster searches. Although this algorithm spends time on pre-processing the text, the running time of this algorithm is comparable to the other existing algorithms as they require the reading of the text into a text array before searching for the pattern. We prove the new algorithm has a linear average case time complexity. The new algorithm is compared to the existing string matching algorithms by recording the time taken to search for patterns in texts using the same data sets as in Chapter 4.

In Chapter 10 we develop a hardware solution for searching the efficiently stored DNA text. We outline the new algorithm, the BK algorithm, we show how we can build a new hardware solution for this algorithm. We give a modification to the basic BK algorithm, which will search a stream of DNA text for multiple sub-strings in a single pass of the text. Attention is paid to the inadequacies of modern micro-processors and the advantages which so-called 'hardware compilation techniques' can offer as a means of accelerating the execution of algorithms. We compare our BK algorithm with five of the fastest existing algorithms by experimentation.

1.1 Novel aspects of the thesis

We have devised a number of new algorithms for solving different string matching problems. The new algorithms have been compared with the existing algorithms to

show that the new algorithms are competitive with the existing algorithms.

- The BR algorithm, a one dimensional string matching algorithm described in Chapter 4, is the first algorithm to consider using the next two characters outside of the pattern and the text alignment window to calculate how far to shift the pattern to the right after a match or mismatch. This algorithm is also described in [31].

- We have developed a two-dimensional algorithm (2D-BR), described in Chapter 5, that uses the structure of the pattern to reduce the number of comparisons required to search for the pattern. We found that the algorithm is best when the size of the alphabet set being used is large.

- The DS algorithm is a one dimensional string matching algorithm that searches in an efficiently stored DNA text and is described in Chapter 7. The efficient storage method is not novel and has been documented by [91]. Using the efficient storage method a DNA text can be stored in 25% of the space required for the original text. The DS algorithm searches in the efficiently stored text and can search for patterns with and without wildcard characters. The DS algorithm was found to be the fastest algorithm for the task of searching for DNA patterns. The algorithm is also described in [32].

- We have extended our work from Chapter 7 to form a new algorithm that

searches in an efficiently stored text for any alphabet set of size ≤ 128. This new method will reduce the text to $\frac{\lceil \log_2 \sigma \rceil}{8}$ of its original size, where σ is the size of the alphabet set. The new algorithm will search for patterns in the efficiently stored text. The algorithm is also described in [34].

- In Chapter 9 we describe a new algorithm that searches for a pattern by pre-processing the text. This new algorithm reduces the number of comparisons required to search for the pattern. The string matching via pre-processing algorithm is many times faster than the existing algorithms when searching for a large number of patterns in the same text.

- In Chapter 10 we describe a hardware solution to searching in the efficiently stored DNA text. Although the algorithm has not been fully implemented in hardware we expect the algorithm to be faster than the DS algorithm. The hardware solution is also described in [33].

Chapter 2

Analysis of Algorithms

2.1 Introduction

Algorithms can be evaluated in a number of different ways. We need to be able to demonstrate that one algorithm is superior to another algorithm. For a given application we have to be able to decide which algorithm is the superior without relying on formal arguments, without being misled by special cases and without being influenced by the efficiency of the programming language used or the hardware used to run the algorithm.

To solve this problem we use a theoretical evaluation of the algorithms, which is independent of the environment used to implement the algorithm. To evaluate each algorithm we use an asymptotic analysis of the algorithm.

2.2 Asymptotic Analysis of algorithms

The most obvious way to evaluate the efficiency of an algorithm would be to measure the amount of processor time and memory space required to run the algorithm to obtain a correct solution using a specific data set. This process is called *benchmarking*. However this only gives a measure of efficiency for one data set. If we changed the data set then the algorithm may no longer be superior to the other available algorithms. For example, searching a telephone directory for a name in sequential order from A to Z may be acceptable if the directory only contains 40 entries but if we increase the number of entries to 400,000 then this algorithm would be unacceptable. Benchmarking is a good way of seeing if a finished program runs to the timing specifications desired for this algorithm. We need an appropriate method to evaluate algorithms before we start coding them.

A single number cannot describe the amount of work done because the number of steps performed is not the same for all inputs. We observe first that the amount of work done usually depends on the size of the input. For example, computing the sum of ten numbers usually requires less operations than computing the sum of 100 numbers. Suppose we have an array A of n integers, then the following algorithm computes the sum of the n integers.

```
for j=1 to n do
    sum=sum+A[j]
```

Figure 2.1: *Algorithm to sum n numbers*

It is convenient to use T(n) to represent the time complexity of an algorithm on any input size n. For example, T(n) = cn for Figure 2.1, where c is a constant. The first observation indicates that we need a measure of the size of the input for a problem. It is usually easy to choose a reasonable measure of size. In Table 2.1 we give some examples:

Problem	Size of input
Find x in a list of names	The number of names in the list
Sort a list of numbers	The number of entries in the list
Multiply two matrices	The dimensions of the matrices

Table 2.1: *The size of input for a number of problems*

Even if the input is fixed at, say n, the number of operations performed may depend on the particular input. Most often we describe the behaviour of an algorithm by stating its worst-case time complexity, which is the maximum number of basic operations performed by the algorithm on any input of size n.

If we have an array L of n distinct entries and wish to find the location of x in L. The following algorithm described in Figure 2.2 compares x to each entry in turn

until a match is found or the list is exhausted. If x is in the list, the algorithm returns the index of the array entry containing x, and index equal to 0 otherwise.

```
index=1
while (index ≤ n) and (L[index] ≠ x) do
        begin
           index = index + 1
        end
if index > n then index = 0
print(index)
```

Figure 2.2: *Algorithm to find an entry in a list*

Clearly the worst-case time complexity T(n) is equal to cn, where c is a constant. The worst cases occur when x appears only in the last position in the list or when x is not in the list at all. In both cases x is compared to all n entries.

2.3 Big-Oh notation

Suppose one algorithm for a problem does $2n$ basic operations, hence roughly $2c_1n$ operations in total, for some constant c_1 and another algorithm does $3n$ basic operations, or $3c_2n$ in total. We don't know which algorithm will run faster. The first algorithm may do many more overhead operations, i.e. its constant of proportionality may be a lot higher. Thus if functions describing the behaviour of two algorithms differ by a constant factor, it may be pointless to try to distinguish between them. We consider such algorithms to be in the same complexity class.

Suppose one algorithm for a problem does $\frac{1}{2}n^2$ basic operations and another algorithm does $5n$. For small values of $n \leq 10$ the first does fewer basic operations but for large values of $n > 10$ the second is better. In fact whatever had been the coefficients of n^2 and of n in these expressions, the second would be faster than the first for all n greater than some value, n_0 say. The rate of growth of a quadratic function is so much greater than a linear function that the coefficients don't matter when n is large.

For these reasons we usually express the time complexity of an algorithm using Big-Oh notation, which is designed to let us hide constant factors. For example, instead of saying the time complexity $\mathrm{T}(n)$ of the algorithm described in Figure 2.1 is cn we would say $\mathrm{T}(n) = \mathrm{O}(n)$, which is read "big-oh of n" which informally means "some constant times n".

Now let $f(n)$ be some function defined on the non negative integers n. We say that $\mathrm{T}(n)$ is $\mathrm{O}(f(n))$ if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n \geq n_0$ we have $\mathrm{T}(n) \leq cf(n)$. For example if we have an algorithm whose time complexity $\mathrm{T}(n) = 6n + 3$. We can say that $\mathrm{T}(n) = \mathrm{O}(n)$ because:

$$\mathrm{T}(n) = 6n + 3$$

$$\mathrm{T}(n) \leq 6n + 3n$$

$$\mathrm{T}(n) \leq 9n$$

We can let $c = 9$ and $n_0 = 1$ in the definition above.

Suppose $\mathrm{T}(n)$ is a polynomial of the form $a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n$,

where leading coefficient a_k is positive. Then we can remove all the terms but the first and the constant a_k replacing it by 1. That is we can conclude $T(n) = O(n^k)$.

Table 2.2 shows some of the more common time complexities for algorithms and their informal names.

Big-Oh	Informal Name
$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	$n \log n$
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$	exponential

Table 2.2: *Common time complexities for algorithms and their informal names*

2.4 Theoretical Versus Practical evaluation of algorithms

In 1984, Narendra Karmarkar [79] took the key step (not common among theoreticians) of implementing his new linear programming algorithm. In doing so, he discovered that it typically ran much more quickly than its worst-case guarantee indicated. His initial claims proved controversial, as other researchers could not at first duplicate

his results. The reason for this was that Karmarkar, coming from a computer science background, had implemented the algorithm using modern data structure techniques, something that was not yet common in the mathematical programming community. The ferment caused by Karmarkar's results and claims has been immensely beneficial for the field of mathematical programming. The packages have improved dramatically by adapting modern data structures and programming techniques.

Many algorithms have been proven to be efficient according to their worst-case or average case evaluations. The worst-case and average case results rarely tell us how an algorithm will work in practice. A practical evaluation of an algorithm requires information about the application area of the algorithm. Using a practical evaluation we can focus on the typical problems that will be solved by the algorithm.

The worst-case complexity describes an upper bound on the worst-case time we would see when running an algorithm. The average case complexity presents an upper bound on the average time taken when running the algorithm many times on various inputs. If the algorithm is to be run a hundred or a thousand times it is pessimistic that the worst-case time will occur each time. In this situation the cumulative time of thousands of different runs should show some averaging out of the worst case behaviour. An average case analysis may give a more realistic picture of the time taken by the algorithm. A practical analysis is still required to guarantee the speed of an algorithm.

A practical evaluation may involve no more than running a few tests to see what happens. After a while one develops an informed opinion about what is likely to affect performance. If we know what factors speed-up or slow-down an algorithm, we can try to apply the speed-ups to other algorithms and remove the slow-downs to improve the algorithm.

Two algorithms may have the same worst-case time complexity or one may have a better worst case time complexity than another. This does not mean that the algorithm with the lower order worst-case time complexity will be the fastest algorithm for all sizes of input. For example, in parallel computing the fastest algorithm known for permutation routing on the hypercube is by Cypher and Plaxton [42] and runs in $O(\log n(\log\log n)^2)$ time with a substantial amount of offline computation. Although asymptotically this is an improvement over the $O(\log^2 n)$ algorithm, because of the large constants hidden by the Big-Oh notation Cypher and Plaxton's algorithm only becomes competitive for hypercubes of dimension greater than 20.

Algorithms may have similar time complexities. For example, *Mergesort* and *Quicksort* have a worst-case time complexities of $O(n\log n)$ and $O(n^2)$ respectively and both algorithms have an average case time complexity of $O(n\log n)$. To determine which algorithm is the best for the task of sorting a list of n numbers we must code both algorithms and time them to see which actually does run faster. (The values in Tables 2.3 and 2.4 were taken from [120]). In Table 2.3, we show the time taken in

seconds to sort a randomised list of numbers with n = 500, 2500 and 10000.

List size	500	2,500	10,000
Mergesort	0.8	8.1	39.8
Quicksort	0.3	1.3	5.3

Table 2.3: *Time in seconds to sort a randomised list*

From Table 2.3 we would choose the *Quicksort* algorithm to sort lists. However, the *Quicksort* algorithm suffers a rapid degrade in performance if the list is ordered as can be seen in Table 2.4.

List size	Random	In Order	Reverse Order
Mergesort	8.1	7.8	7.9
Quicksort	1.3	35.1	37.1

Table 2.4: *Time in seconds for sorting ordered lists for n = 2500*

When the list is ordered the *Quicksort* algorithm performs in its worst-case of $O(n^2)$. If we were to sort a number of lists we would choose to use the *Quicksort* algorithm as long as the lists were in a random order. If the lists were partially sorted or fully sorted we would want to use the *Mergesort* algorithm. We wouldn't expect the worst-case of the list being sorted already to occur very often in real world applications of sorting algorithms. We wouldn't know which algorithm was best for sorting until we implement them and test them over a number of lists. The average-case and worst-case analysis of an algorithm are only indicative of how fast

the algorithm should be and do not guarantee any speed over other algorithms in practice. This example clearly shows that an evaluation of algorithms both in theory and practice is required before selecting an algorithm for a specific task.

In Chapters 4, 5, 7, 8, 9 and 10 we present and discuss a new algorithm and analyse the speed of the new algorithm. We perform a theoretical analysis of the new algorithm and calculate the worst-case and average case analyses of the algorithms. The worst-case analysis gives the upper bound number of steps taken by the algorithm. The average case analysis presents an upper bound on the average time we would expect when running the algorithm many times on various inputs. This shows that theoretically the new algorithms are competitive with the existing algorithms. To show the actual performance of the algorithms we implement them and compare them with the existing algorithms. The time taken by the new algorithms is shown to be competitive with the existing algorithms.

Chapter 3

String matching algorithms

3.1 Introduction

A number of different tasks are performed on strings [47, 63, 127]. String matching is finding an occurrence of a pattern string in a larger string of text. String matching can be one dimensional, for example the comparison of a word with a text or can be two dimensional, the comparison of one matrix with another matrix. In this chapter we will consider one dimensional string matching and its applications.

The string matching problem has attracted a lot of interest throughout the history of computer science, and is crucial to the computing industry. This problem arises in many computer packages in the form of spell checkers, search engines on the Internet, find utilities on various machines, matching of DNA strands and so on.

String matching algorithms [47, 127] work as follows. First the pattern of length m, P[1..m], is aligned with the extreme left of the text of length n, T[1..n]. Then the pattern characters are compared with the text characters. The algorithms can vary in

the order in which the comparisons are made. After a mismatch is found the pattern is shifted to the right and the distance the pattern can be shifted is determined by the algorithm that is being used. This shifting procedure and the speed at which a mismatch is found are the main difference between the string matching algorithms.

In the Naive or Brute Force (BF) algorithm, the pattern is aligned with the extreme left of the text characters and corresponding pairs of characters are compared from left to right. This process continues until either the pattern is exhausted or a mismatch is found. Then the pattern is shifted one place to the right and the pattern characters are again compared with the corresponding text characters from left to right until either the text is exhausted or a full match is obtained. This algorithm can be very slow. Consider the worst case when both pattern and text are all a's followed by a b. The total number of comparisons in the worst case is $O(nm)$. However, this worst case example is not one that occurs often in natural language text.

3.1.1 Knuth, Morris and Pratt algorithm and its derivatives

The number of comparisons performed by the BF algorithm can be reduced by moving the pattern to the right by more than one position when a mismatch is found. This is the idea behind the Knuth-Morris-Pratt (KMP) algorithm [84]. The KMP algorithm starts and compares the characters from left to right the same as the BF algorithm. When a mismatch occurs the KMP algorithm moves the pattern to the right by

maintaining the longest border of a prefix (the beginning) of the pattern with a suffix (the end) of the part of the text that has matched the pattern so far (See Figure 3.1).

A border is a repeated substring in the pattern with the repeated substring starting with the first character of the pattern. The border of '*babdcbabcadb*' is '*bab*'. If a border does exist then we shift the pattern so as to preserve any characters that have already matched. So for the example '*babdcbabcadb*' u = '*babdcbab*', v = '*bab*' in Figure 3.1.

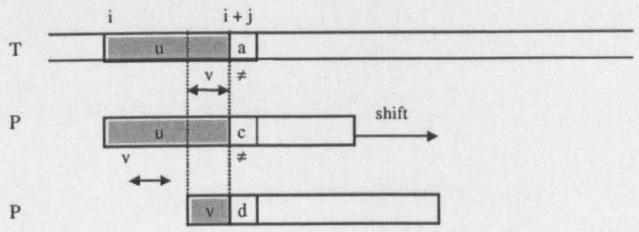

Figure 3.1: *Shift in the Knuth-Morris-Pratt algorithm (v border of u and $d \neq c$)*

As can be seen v is not compared again and the comparisons start at the character after the string v. In general the shift is calculated so that each character in the text is compared at most twice. Then the comparisons begin again at the character that mismatched in the text and the corresponding pattern character. If a border does not exist then the comparisons continue from the first character in the pattern. There are not many English words, on average, with a border bigger than 1. The KMP algorithm takes at most $2n$ character comparisons. Although when using English text it behaves very closely to the BF algorithm. The KMP algorithm does $O(m+n)$ operations in the worst case. The reason for this is that if we consider the worst

case again of a text of all a's and a pattern of all a's followed by a b. Then we have an overlap of $m-1$ (the length of the longest repeated substring). So the pattern is shifted by one position after a mismatch and begins the comparisons at the last character in the pattern as it already knows the previous $m-1$ characters will match. So the algorithm will take $n+m$ operations. A coding of the KMP algorithm in the programming language JAVA is shown in [110].

The Colussi (COL) algorithm [44, 45] is an improvement of the KMP algorithm. The number of character comparisons is $1.5n$ in the worst case. The set of pattern positions is divided into two disjoint subsets. First the comparisons are performed from left to right for the characters at positions in the first set. If there is no mismatch, the characters at positions in the second set are compared from right to left. This strategy reduces the number of comparisons.

Galil and Giancarlo (GG) [60] improved the COL algorithm by reducing the number of character comparisons in the worst case to $\frac{4}{3}n$. In these algorithms the preprocessing takes $O(m)$ time.

3.1.2 Boyer-Moore algorithm and its derivatives

The Boyer-Moore (BM) algorithm [36] differs in one main feature from the algorithms already discussed. Instead of the characters being compared from left to right, in the BM algorithm the characters are compared from right to left, starting with the

rightmost character of the pattern. In a case of mismatch it uses two functions, good suffix function (see Figures 3.2 and 3.3) and last occurrence function (see Figures 3.4 and 3.5) and shifts the pattern by the maximum number of positions computed by these functions. The good suffix function returns the number of positions for moving the pattern to the right by the least amount, so as to align the already matched characters with any other substring in the pattern that are identical. The last occurrence function moves the pattern to the last occurrence of the text character in the pattern (from the left) that mismatches at the current pattern position. If the character is not in the pattern then the pattern is shifted by m places to the right. It is the last occurrence shift that gives the BM its speed and is used in many of its derivatives. The worst case running time of the BM algorithm is $O(nm)$. This is because as in the BF algorithm characters can be compared m times to give a worst case run time of $O(nm)$.

In Figure 3.2 a portion of the text and pattern has matched up to the character 'a' in the text and 'b' in the pattern. The already matched portion of the text occurs again in the pattern at a position to the right of the current match. This other occurrence of u in the pattern is aligned with the characters in the text that matched u. This is not done if the character in the pattern that precedes the second occurrence of u is the same as the first occurrence. Comparisons resume at the rightmost position and continue from right to left.

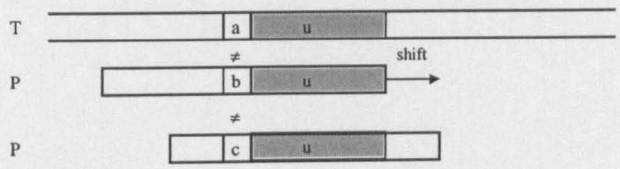

Figure 3.2: *Boyer-Moore good-suffix shift, u reappears preceded by a character different from 'b'*

In Figure 3.3 the portion of the pattern that matched the text u has a portion v that is a prefix (leftmost portion) of u. The pattern is shifted so that v is aligned with the text characters that matched with it. Comparisons resume at the rightmost position and continue from right to left.

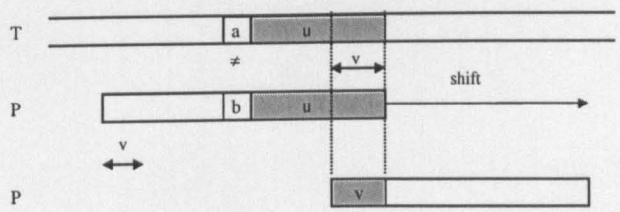

Figure 3.3: *Boyer-Moore good-suffix shift, only a prefix of u reappears in P*

In Figure 3.4 the pattern is shifted so that the character in the text that mismatched is aligned with the rightmost occurrence of that character in the pattern.

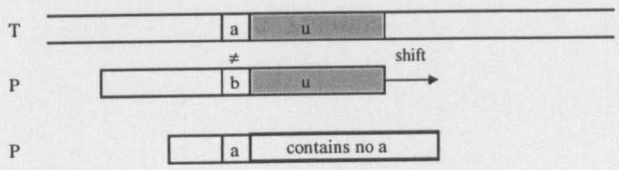

Figure 3.4: *Boyer-Moore bad-character shift, a appears in P*

In Figure 3.5 the pattern is shifted past the character that mismatch as it does not occur in the pattern.

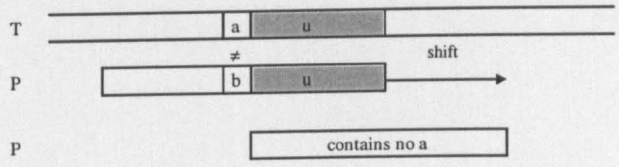

Figure 3.5: *Boyer-Moore bad-character shift, a does not appear in P*

The Turbo Boyer-Moore (TBM) algorithm [48] and the Apostolico-Giancarlo (AG) algorithm [8] are amelioration's of the BM algorithm. When a partial match is made between the pattern and the text these algorithms remember the characters that matched and do not compare them again with the text. The TBM algorithm and the Apostolico-Giancarlo algorithm perform in the worst case at most $2n$ and $1.5n$ [52] character comparisons respectively.

The Horspool (HOR) algorithm [70] is a simplification of the BM algorithm. It does not use the good suffix function, but uses a modified version of the last occurrence function. The modified last occurrence function determines the right most occurrence of the $k + m^{th}$ text character, T$[k + m]$ in the pattern, if a mismatch occurs when a pattern is aligned with T$[k..k + m]$. The comparison order is not described in [70].

We assumed that the order is from right to left as in the BM algorithm. The average case running time of the HOR algorithm is proven to be linear in [23]. The HOR heuristic is analysed in [92].

The Raita (RAI) algorithm [109] uses variables to represent the first, middle and last characters of the pattern. The process used is to compare the rightmost character of the pattern, then the leftmost character, then the middle character and then the rest of the characters from the second to the $m-1^{th}$ position. Using variables is more efficient than looking up the characters in the pattern array. The use of variables to represent characters in the array is known as 'Raita's trick'. This optimisation trick is only used in the RAI algorithm. If at any time during the procedure a mismatch occurs then it performs the shift as in the HOR algorithm.

The Quicksearch (QS) algorithm [129, 74] is similar to the HOR algorithm and the RAI algorithm. It does not use the good suffix function to compute the shifts. It uses a modified version of the last occurrence function. Assume that a pattern is aligned with the text characters T[$k..k+m$]. After a mismatch the length of the shift is at least one. So, the character at the next position in the text after the alignment (T[$k+m+1$]) is necessarily involved in the next attempt. The last occurrence function determines the right most occurrence of T[$k+m+1$] in the pattern. If T[$k+m+1$] is not in the pattern the pattern can be shifted by $m+1$ positions. The comparisons between text and pattern characters during each attempt can be done in any order.

The Maximal Shift (MS) algorithm [129] is another variant of the QS algorithm. The algorithm is designed in such a way that the pattern characters are compared in the order which will give the maximum shift if a mismatch occurs.

The Liu, Du and Ishi (LDI) algorithm [90] is a variant of the QS algorithm. The algorithm uses the same shifting function as the QS but changes the order in which the pattern characters are compared to the text. The characters are compared in a circular method starting at the first character in the pattern and finishing at the last. If a mismatch occurs then the pattern is shifted and searching restarts with the pattern character that mismatched. For example, if the pattern was '*string*' and the pattern mismatched the text at the '*r*' then we would search in the order '*ringst*'.

The Smith (SMI) algorithm [125] uses HOR and Quick Search last occurrence functions. When a mismatch occurs, it takes the maximum values between these functions. The characters are compared from left to right.

The Zhu and Takaoka (ZT) algorithm [138] is another variant of the BM algorithm. The comparisons are done in the same way as BM (i.e. from right to left) and it uses the good suffix function. If a mismatch occurs at T[i], the last occurrence function determines the right most occurrence of T[$i-1..i$] in the pattern. If the substring is in the pattern, the pattern and text are aligned at these two characters for the next attempt. If the two character substring is not in the pattern then we shift by m positions. The shift table is a two dimensional array of size alphabet size by alphabet

size.

The Baeza-Yates (BY) algorithm [12] is similar to the ZT algorithm. It calculates the shift according to the last k characters of the pattern aligned with the text. When k=2 the shifts are the same as ZT but without the good suffix function. The main differences are constructing and storing the shift table. The shift table is a one dimensional array of length σ^2, where σ is the size of the alphabet. The table is constructed by bit shifting the two characters to form a 16 bit number and storing the value of the shift at this location in the array.

The HOR algorithm can be improved by using a transformation that increases the size of the alphabet being used [22]. As the size of the alphabet is increased the probability of a larger shift increases. The transformation concatenates a substring to form a new character. For example, the i^{th} character is composed by the concatenation of the i^{th}, .. ,$i + k + 1^{th}$ characters of the original string for any k $\geq$ 1. This reduces the length of the string to $m - (k + 1)$ and the size of the alphabet is increased to c^k where c is the size of the original alphabet.

3.1.3 Other one dimensional string matching algorithms

The Karp-Rabin algorithm [80] uses a hashing function instead of comparing each character individually. The algorithm checks to see if a portion of the text aligned with the pattern is similar to the pattern. The pattern is hashed and then compared with

hashed portions until either a match is found or until the end of the text is reached. Upon a match of the hashed text portion and the hashed pattern the characters that are aligned are compared character by character to see if a true match is present at this position. The algorithm has a worst case time complexity of $O(nm)$ and an expected running time of $O(n+m)$

The Not So Naive algorithm (NSN) is based on the BF algorithm. The NSN is the BF with a constant time and space pre-processing phase added. The pattern is aligned with the text$[i, i+m]$. The pattern can be shifted by two positions to the right if one of two conditions are true: $pattern[0] \neq pattern[1]$ and pattern[1] = text$[i+1]$ or pattern[0] = pattern[1] and pattern[1] $\neq$ text$[i+1]$. The order in which the comparisons are performed is modified. The characters are compared in order from pattern[1] to pattern$[m-1]$ and then pattern[0] is compared to text$[i]$. The NSN algorithm has a worst case time complexity of $O(nm)$.

Searching can be done in $O(n)$ time using a minimal Deterministic Finite Automaton (DFA) [20, 68, 46, 97]. This algorithm uses $O(\sigma m)$ space and $O(m+\sigma)$ pre-processing time. Where σ is the size of the alphabet being used.

The Simon algorithm [122, 123] gives a more economical implementation of a DFA. Simon noticed that there are only a few significant edges in the a DFA. There are at most $2m$ significant edges in a DFA. Removing the least significant edges we can improve the preprocessing time to $O(m)$. Upon a search at most $2n$ - 1 text

comparisons are performed during a search for a pattern. A Boyer-Moore automaton is constructed and analysed in [19].

A pre-processing function is needed for all the algorithms to calculate the relevant shifts upon a mismatch or match except for the BF algorithm, which has no pre-processing. The pre-processing cost of the algorithms is an important factor in the speed of the algorithm with regard to the number of operations required and the amount of memory required. This will be most noticeable when we are searching in smaller texts.

Animations of string matching algorithms can be found at [39] and more information about the above string matching algorithms can be found in [38, 40].

3.1.4 Approximate string matching algorithms

Approximate string matching allows the erroneous matching of a user defined number of erroneous matches, k, between the pattern and text. An error or difference can be one of the following three types:

Substitution: A character of the pattern corresponds to a different character of the text.

Deletion: A character of the pattern corresponds to no character in the text.

Insertion: A character of the text corresponds to no character in the pattern.

A number of solutions exist to solve this problem [3, 9, 15, 16, 17, 55, 135]. Approxi-

mate string matching is used for many applications including aiding in the security of passwords [94], spell checkers and bibliographic search. Approximate string matching has recently been applied to approximate searching on hypertext [104] and compressed texts [99].

Chapter 4

A new string matching algorithm

4.1 Introduction

The algorithms described in chapter 3 are implemented and the results are given in this chapter. From the findings of the experimental results we identify the best two algorithms. We combine these two algorithms and introduce a new algorithm. We compare the new algorithm with the existing algorithms by experimentation.

4.2 Experimental results of the existing algorithms

Monitoring the number of comparisons performed by each algorithm was chosen as a way to compare the algorithms. All the algorithms were coded in C, which are taken from [38], animations of the algorithms can be found in [39]. This collection of string matching algorithms were easy to implement as functions into our main control program. The algorithms were coded as their authors had devised them in their papers. The main control program was the same for each algorithm and so did

not affect the performance of the algorithms. Each algorithm had an integer counter inserted into it, to count the number of comparisons made between the pattern and the text. The counter was incremented by one each time a comparison was made.

A random text of 200,000 words from the UNIX English dictionary was used for the first set of experiments. We decided to number each of the words in UNIX dictionary from 1 to 25,000. Then we used a pseudo random number generator to pick words from the UNIX dictionary and place them in the random text. Each word was separated by a space character. Punctuation was also removed as we were concerned with finding words and the punctuation would not affect the results obtained. We selected a word (pattern) from the UNIX dictionary and searched the text for the first occurrence of the word.

The text was searched for each word in the UNIX dictionary and the results are given in Table 4.1. The first column in Table 4.1 is the length of the pattern. The second column is the number of words of that length in the UNIX English dictionary. The abbreviations at the top of the remaining columns related to abbreviations defined in Chapter 3. For example, for a pattern length of 7, 4042 test cases were carried out and the average number of character comparisons made by the KMP algorithm was 197,000 (to the nearest 1000). The average was calculated by taking the total number of comparisons performed to find all 4042 cases and dividing this number by 4042. The figure given in the table is the total number of comparisons taken di-

vided by the number of words for the pattern length and then divided by 1000.These columns are arranged in descending order of the average of the total number of comparisons of the algorithms. An interesting observation is that for (almost) each row the values are in descending order except for the last two columns.

p. len	num.	BF	KMP	DFA	BY	BM	AG	HOR	RAI	TBM	MS	LDI	QS	ZT	SMI
2	133	7	7	7	7	3	3	3	3	3	2	2	2	3	2
3	765	38	38	37	19	13	13	13	13	13	11	10	10	13	10
4	2178	82	82	80	28	23	23	23	23	22	19	19	19	22	18
5	3146	151	150	145	39	34	34	34	34	34	30	30	30	32	28
6	3852	186	185	179	38	36	36	36	36	36	33	33	32	33	30
7	4042	198	197	191	34	34	34	34	34	34	32	31	31	30	28
8	3607	205	204	197	30	32	32	31	32	31	30	29	29	27	26
9	3088	212	211	204	28	30	30	30	30	30	29	28	28	25	24
10	1971	220	219	212	26	29	29	29	29	29	28	27	27	24	23
11	1120	209	207	201	22	26	26	26	26	25	25	24	24	21	21
12	593	218	217	210	21	25	25	25	25	25	24	24	24	21	20
13	279	224	222	215	20	24	24	24	24	24	23	24	23	19	19
14	116	228	227	220	19	23	23	23	23	23	23	23	23	19	19
15	44	151	150	144	11	15	15	15	15	14	14	14	14	11	12
16	17	227	225	217	16	20	21	21	21	20	20	21	20	18	16
17	7	233	231	222	16	20	20	20	20	19	19	20	20	15	16
18	4	236	234	225	15	19	20	20	20	19	19	20	20	14	16
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	1	132	131	122	7	10	10	10	10	10	10	10	10	7	8
21	2	311	309	295	16	24	24	25	25	23	23	24	24	15	18
22	1	491	486	455	23	33	33	33	33	33	31	34	34	22	27
Total	24966	180	179	174	31	31	31	30	30	30	28	28	28	27	25

Table 4.1: *The number of comparisons in 1000's for searching a text of 200,000 words (1670005 characters).*

The algorithm with the largest number of comparisons is the BF algorithm. This is because the algorithm shifts the pattern by one place to the right when a mismatch

occurs, no matter how much of a partial or full match has been made. This algorithm has a quadratic worst case time complexity. However, the KMP algorithm, which has a linear worst case time complexity, does roughly the same number of comparisons as the BF algorithm. The reason for this is that in a natural language a multiple occurrence of a substring in a word is not common. Other linear time algorithms, DFA, also have roughly the same number of comparisons as the BF algorithm. We will see below that the other quadratic worst case time complexity algorithms perform much better than these linear worst case time algorithms. This is a good example showing that asymptotic worst case running time analysis can be indicative of how algorithms are likely to perform in practice, but they are not sufficiently accurate to predict actual performance.

The BM algorithm uses the good suffix function to calculate the shift which depends on a reoccurrence of a substring in a word. But, it also uses the last occurrence function. It is this last occurrence function that reduces the number of comparisons significantly. In practice, on an English text, the BM algorithm is three or more times faster than the KMP algorithm [124]. From Table 4.1, one can see that the KMP algorithm takes six times as many comparisons than the BM algorithm on average. The other algorithms, BY, TBM, AG, HOR, RAI, LDI, QS, MS, SMI and ZT, are variants of the BM algorithm. The number of comparisons for these algorithms is roughly the same number as in the BM algorithm.

The SMI algorithm and the ZT algorithm do the least number of comparisons for pattern lengths less than or equal to twelve and greater than twelve respectively.

4.3 The new algorithm (BR)

From the findings of the experimental results discussed in section 3, it is clear that the SMI and ZT algorithms have the lowest number of comparisons among the others. We combined the calculations of a valid shift in QS and ZT algorithms to produce a more efficient algorithm, the BR algorithm [31]. If a mismatch occurs when the pattern P[1 .. m] is aligned with the text T[$k..k+m-1$], the shift is calculated by the rightmost occurrence of the substring T[$k+m+1..k+m+2$] in the pattern. If the substring is in the pattern then the pattern and text are aligned at this substring for the next attempt. This can be done shifting the pattern as shown in the table below.

$T[k+m+1]$	$T[k+m+2]$	*Shift*
*	$P[1]$	$m+1$
$P[i]$	$P[i+1]$	$m-i+1, 1 \leq i \leq m-1$
$P[m]$	*	1
Otherwise		$m+2$

Figure 4.1: *Values for a shift using the BR algorithm*

Let * be a wildcard character that is any character in the ASCII set. Note that if T[$k+m+1..k+m+2$] is not in the pattern, the pattern is shifted by m+2 positions.

For example, the following shifts would be associated with the pattern, 'onion'.

$T[k+m+1]$	$T[k+m+2]$	*Shift*
*	*o*	6
o	*n*	5
n	*i*	4
i	*o*	3
o	*n*	2
n	*	1
Otherwise		7

Figure 4.2: *The shift values for the pattern 'onion' using the BR algorithm*

After a mismatch the calculation of a shift in our new algorithm takes O(1) time. Note that for the substrings 'ni' and 'n*' have a value of 4 and 1 respectively. This ambiguity can be solved by the higher shift value being overwritten with the lower value. We will explain this later in this section. For a given pattern P[1 .. m] the preprocessing is done as follows, and it takes O($m^2+\sigma$) time, where σ is the size of the alphabet. The two dimensional array, ST (Shift Table), of size at most $m+1 \times m+1$ will store the shift values for all pairs of characters. The ST will be initialised as $m+2$. As the index of the ST is of type integer, we need to convert the pairs of characters into pairs of integers. This is done by defining an array of ASCII character set size called CON with each entry initialised to 0. For each character in the pattern the right most position (numbering from the right, starting with 1) is entered in the corresponding location in CON. For example, the relative position of the character

'a' in the ASCII set is 97. Assume that the character 'a' is in the pattern. The right most position of 'a' in the pattern is entered in CON[97].

If the pattern was the word 'onion' then the rightmost positions of n, o and i are 1, 2 and 3 respectively. The CON for 'onion' would look like this:

Character:	...	a	b	...	h	i	j	...	n	o	p	...
ASCII value:	...	97	98	...	104	105	106	...	110	111	112	...
CON:	...	0	0	...	0	3	0	...	1	2	0	...

Figure 4.3: *The structure of the CON table for the pattern 'onion'*

The value of a shift for the pair T[$k+m+1$] and T[$k+m+2$] is ST(CON[T[$k+m+1$]], CON[T[$k+m+2$]]).

All the entries in the ST will be initialised as 7, and the above shift values will be entered as follows:

$[wildcard][o] = 6$

$[o][n] = 5$

$[n][i] = 4$

$[i][o] = 3$

$[o][n] = 2$

$[n][wildcard] = 1$

The ST for the pattern 'onion' would look like this after the complete insertion of

all the values. The rows represent the T$[k+m+1]^{th}$ character and the columns are the T$[k+m+2]^{th}$ character.

	0	1	2	3
0	7	7	6	7
1	1	1	1	1
2	7	2	6	7
3	7	7	3	7

Table 4.2: *The Shift Table for the pattern 'onion'*

The order of performing the steps is important in ensuring the correct values appear in ST. Note that the higher values have been over written by the lower values. We search for the pattern starting at P[m] and searching from right to left and finish at P[1]. To find a shift value we look up in the CON table the first two characters after the pattern and the text alignment window. We use these values to the find the correct shift value in the ST.

We now give an example of our new algorithm in action to find the pattern 'onion'. The tables above, ST and CON for the pattern 'onion' were used to calculate the shift after a mismatch. In Tables 4.3 to 4.7, the first row shows the text and the third row shows the position of the pattern. The second row shows whether the aligned pattern and text characters match (=) or mismatch ($\neq$) as the comparisons are made.

w	e		w	a	n	t		t	o		t	e	s	t		w	i	t	h		o	n	i	o	n
				≠																					
o	n	i	o	n																					

Table 4.3: *mismatch shift on ST(CON[n], CON[t]) = ST(1, 0) = 1*

w	e		w	a	n	t		t	o		t	e	s	t		w	i	t	h		o	n	i	o	n
				≠	=																				
	o	n	i	o	n																				

Table 4.4: *mismatch shift on ST(Con [t]], CON[ ]) = ST(0, 0) = 7*

w	e		w	a	n	t		t	o		t	e	s	t		w	i	t	h		o	n	i	o	n
												≠													
								o	n	i	o	n													

Table 4.5: *mismatch shift on ST(CON[s], CON[t]) = ST(0, 0) = 7*

w	e		w	a	n	t		t	o		t	e	s	t		w	i	t	h		o	n	i	o	n
																			≠						
															o	n	i	o	n						

Table 4.6: *mismatch shift on ST(CON[ ], CON[o]) = ST(0, 2) = 6*

w	e		w	a	n	t		t	o		t	e	s	t		w	i	t	h		o	n	i	o	n
																					=	=	=	=	=
																					o	n	i	o	n

Table 4.7: *So the pattern 'onion' has been matched and the text is exhausted*

So the word onion is found in 10 comparisons in a text of length 26.

4.4 Average case analysis of the BR algorithm

The average case time complexity of the BR algorithm is the upper bound number of comparisons taken by the BR algorithm given the average case input. Let σ be the size of the alphabet, Σ, where Σ is the set of alphabets used by the text. We assume that the characters in the text all have an equal frequency and so the probability of a match between a character of the pattern and text is $\frac{1}{\sigma}$. The number of comparisons taken at an attempt can be 1 to m inclusive.

Lemma 4.1: The upper bound number of comparisons at an attempt is $\sum_{i=0}^{m-1}(\frac{1}{\sigma^i})$ for an average case text.

Proof: The probability of i pattern and text characters matching is $\frac{1}{\sigma^i}$. After a match we must make at least one more comparison unless we have a full match. Therefore the probability of at least i comparisons being made is $\frac{1}{\sigma^{i-1}}$. The probability that exactly i comparisons are made is (the probability of at least i comparisons are made) - (probability of at least i+1 comparisons are made). The probability of exactly

i comparisons being made is $\frac{1}{\sigma^{i-1}}$ - $\frac{1}{\sigma^i}$ for all $i < m$. When $i = m$ (the probability of the m^{th} comparison), the equation becomes $\frac{1}{\sigma^{m-1}}$. To find the total number of comparisons made we multiply each of the probabilities by i and sum the results. So the total number of comparisons made at an attempt is $(\sum_{i=1}^{m-1}((\frac{1}{\sigma^{i-1}} - \frac{1}{\sigma^i}) \times i) + \frac{1}{\sigma^{m-1}}$. If we expand the equation we get:

$(\frac{1}{\sigma^0} - \frac{1}{\sigma^1}) \times 1 + (\frac{1}{\sigma^1} - \frac{1}{\sigma^2}) \times 2 + \cdots + (\frac{1}{\sigma^{m-3}} - \frac{1}{\sigma^{m-2}}) \times m-2 + (\frac{1}{\sigma^{m-2}} - \frac{1}{\sigma^{m-1}}) \times m-1 + \frac{1}{\sigma^{m-1}} \times m)$

$= \frac{1}{\sigma^0} - \frac{1}{\sigma^1} + \frac{2}{\sigma^1} - \frac{2}{\sigma^2} + \cdots + \frac{m-2}{\sigma^{m-3}} - \frac{m-2}{\sigma^{m-2}} + \frac{m-1}{\sigma^{m-2}} - \frac{m-1}{\sigma^{m-1}} + \frac{m}{\sigma^{m-1}}$

$= \frac{1}{\sigma^0} + \frac{1}{\sigma^1} + \cdots + \frac{1}{\sigma^{m-3}} + \frac{1}{\sigma^{m-2}} + \frac{1}{\sigma^{m-1}}$

$= \sum_{i=0}^{m-1} \frac{1}{\sigma^i}$ □

We assume that $\sigma > 1$ and therefore the above is maximised when $\sigma = 2$. As m increases the equation approaches the limit for this equation which is 2. So we expect to make 2 comparisons at each attempt for a text where all the characters have equal frequency.

Lemma 4.2: The lower bound for a shift in an average case text is $\frac{7}{4}$.

Proof: To find the average shift we have to consider what values are in the shift table. There are σ^2 entries in the table. The entries are entered in order of size of shift from the smallest to the largest until the table is complete.

The size of the shift is minimised when $\sigma = 2$ and there are $\sigma^2 = 2^2 = 4$ entries in the shift table. Using Figure 4.1 to calculate the shifts and assuming that the first two pairs of characters in the pattern don't match. Then there will be two entries

with a value of one and one entry for each of the values two and three. We multiply each shift value by its frequency in the shift table and total the results. This result is divided by σ^2 to give the average shift. The average shift is $(2 \times 1) + (1 \times 2) + (1 \times 3)$ divided by $2^2 = \frac{7}{4}$. □

Theorem: The BR algorithm has a linear average case running time of $O(n+m)$.

Proof: The lowest average shift value for a text with equally frequent characters is $\frac{7}{4}$. We would expect to make $\frac{n}{\frac{7}{4}} = \frac{4n}{7}$ attempts and expect to make 2 comparisons at each attempt. We expect to make $2 \times \frac{4n}{7} = \frac{8n}{7}$ comparisons. Therefore the BR algorithm has a linear average case running time of $O(n+m)$. Note that the m term in $O(n+m)$ comes from the time taken for the pre-processing. □

4.5 Experimental results and comparisons with the BR algorithm

We select the best eight algorithms from the results in Table 4.1, the BM algorithm and the KMP algorithm, and compare with our BR algorithm. Experiments were carried out for different random texts as described in Section 4.3. There were 2 different texts of 10,000 words (Texts A and B), a text of 50,000 words and a text of 100,000 words. The results are described in Tables 4.8 to 4.11 respectively. Tables 4.8 to 4.11 show the average number of comparisons required for a search for the given pattern length. They are based on taking the total number of comparisons for the search for all the patterns of a length and dividing the number by the number of

patterns of that size to give the average and then they are divided by 1000. So for example, in Table 4.8 the BM algorithm takes 12,000 comparisons (to the nearest thousand) on average if the pattern length is seven.

p len	num	KMP	BM	HOR	RAI	TBM	MS	LDI	QS	ZT	SMI	BR
2	133	6	3	3	3	3	2	2	2	3	2	2
3	765	20	7	7	7	7	6	6	6	7	5	4
4	2178	41	11	11	11	11	10	10	10	11	9	7
5	3146	60	14	13	13	13	12	12	12	12	11	9
6	3852	67	13	13	13	13	12	12	12	12	11	9
7	4042	68	12	12	12	12	11	11	11	10	10	8
8	3607	69	11	11	11	11	10	10	10	9	9	7
9	3088	70	10	10	10	10	9	9	9	8	8	7
10	1971	71	9	9	9	9	9	9	9	8	8	6
11	1120	70	9	9	9	9	8	8	8	7	7	6
12	593	70	8	8	8	8	8	8	8	6	7	5
13	279	72	8	8	8	8	8	8	8	6	6	5
14	116	69	7	7	7	7	7	7	7	5	6	5
15	44	72	7	7	7	7	7	7	7	5	6	5
16	17	70	6	6	6	6	6	6	6	5	5	4
17	7	75	7	7	7	6	6	6	6	5	5	4
18	4	87	7	7	7	7	7	7	7	5	6	5
19	0	0	0	0	0	0	0	0	0	0	0	0
20	1	89	7	7	7	7	7	7	7	4	5	4
21	2	88	7	7	7	7	6	7	7	4	5	4
22	1	89	6	6	6	6	6	6	6	4	5	4
Total	24966	64	11	11	11	11	10	10	10	10	9	7

Table 4.8: *The number of comparisons in 1000's for searching Text A of 10,000 words (83360 characters)*

p len	Num	KMP	BM	HOR	RAI	TBM	MS	LDI	QS	ZT	SMI	BR
2	133	6	3	3	3	3	2	2	2	3	2	2
3	765	21	7	7	7	7	6	6	6	7	6	4
4	2178	42	12	12	12	12	10	10	10	11	9	7
5	3146	59	13	13	13	13	12	12	12	12	11	9
6	3852	66	13	13	13	13	12	12	12	11	11	9
7	4042	68	12	12	12	12	11	11	11	10	10	8
8	3607	69	11	11	11	11	10	10	10	9	9	7
9	3088	70	10	10	10	10	9	9	9	8	8	7
10	1971	71	9	9	9	9	9	9	9	8	8	6
11	1120	70	9	9	9	9	8	8	8	7	7	6
12	593	71	8	8	8	8	8	8	8	6	7	5
13	279	71	8	8	8	8	8	8	7	6	6	5
14	116	70	7	7	7	7	7	7	7	6	6	5
15	44	64	6	6	6	6	6	6	6	5	5	4
16	17	74	7	7	7	7	7	7	7	5	5	4
17	7	64	6	6	6	6	5	6	6	4	4	4
18	4	87	7	7	7	7	7	7	7	5	6	5
19	0	0	0	0	0	0	0	0	0	0	0	0
20	1	41	3	3	3	3	3	3	3	2	3	2
21	2	72	5	6	6	5	5	6	5	4	4	3
22	1	89	6	6	6	6	6	6	6	4	5	4
Total	24966	63	11	11	11	11	10	10	10	10	9	7

Table 4.9: *The number of comparisons in 1000's for searching Text B of 10,000 words (83425 characters)*

p len	num	KMP	BM	HOR	RAI	TBM	MS	LDI	QS	ZT	SMI	BR
2	133	9	6	6	6	6	4	4	4	6	4	3
3	765	37	13	13	13	13	10	10	10	13	10	8
4	2178	77	21	21	21	21	18	18	18	20	17	14
5	3146	133	30	30	30	30	27	26	26	28	25	21
6	3852	159	31	31	31	31	29	28	28	28	26	22
7	4042	170	29	29	29	29	27	27	27	26	24	21
8	3607	176	27	27	27	27	26	25	25	24	22	19
9	3088	181	26	26	26	26	25	24	24	22	21	18
10	1971	185	24	24	24	24	23	23	23	20	20	17
11	1120	184	23	23	23	23	22	22	22	18	18	16
12	593	186	21	21	21	21	21	21	20	17	17	15
13	279	183	20	20	20	20	19	19	19	15	16	14
14	116	194	20	20	20	20	19	20	19	15	16	14
15	44	164	16	16	16	16	16	16	16	12	13	11
16	17	217	20	20	20	20	20	20	20	17	16	13
17	7	172	15	15	15	14	14	15	15	11	12	10
18	4	147	12	13	13	12	12	13	13	9	10	8
19	0	0	0	0	0	0	0	0	0	0	0	0
20	1	41	3	3	3	3	3	3	3	2	3	2
21	2	221	17	18	18	17	17	17	17	11	13	10
22	1	397	27	27	27	27	26	28	28	18	22	18
Total	24966	155	27	26	26	26	24	24	24	23	22	18

Table 4.10: *The number of comparisons in 1000's for searching a text of 50,000 words (417923 characters)*

P len	num	KMP	BM	HOR	RAI	TBM	MS	LDI	QS	ZT	SMI	BR
2	133	13	7	7	7	7	5	5	5	7	5	3
3	765	37	13	13	13	13	10	10	10	13	10	8
4	2178	80	22	22	22	22	19	18	18	21	17	14
5	3146	149	34	34	34	34	30	30	29	31	28	22
6	3852	182	36	36	36	36	33	32	32	33	29	24
7	4042	193	33	33	33	33	31	30	30	29	27	23
8	3607	201	31	31	31	31	29	29	29	27	26	21
9	3088	198	28	28	28	28	27	26	26	24	23	19
10	1971	198	26	26	26	26	25	25	25	22	21	18
11	1120	199	25	25	24	24	24	23	23	20	20	17
712	593	217	25	25	25	25	24	24	24	20	20	17
13	279	207	23	23	23	22	22	22	22	18	18	15
14	116	180	19	19	19	19	18	18	18	14	15	12
15	44	218	22	22	22	21	21	21	21	17	17	14
16	17	162	15	15	15	15	15	15	15	12	12	10
17	7	220	20	20	20	19	19	19	19	14	15	13
18	4	208	17	17	17	17	17	18	18	12	14	11
19	0	0	0	0	0	0	0	0	0	0	0	0
20	1	157	12	12	12	12	12	13	13	8	10	7
21	2	89	7	7	7	7	7	7	7	11	5	4
22	1	315	21	21	21	21	20	22	22	14	18	13
Total	24966	173	30	30	30	29	27	27	27	26	24	20

Table 4.11: *The number of comparisons in 1000's for searching a text of 100,000 words (834381 characters)*

From these tables one can observe that the relative order of their performance is the same as in Table 4.1. The main observation is that the BR algorithm performs better than the other algorithms for all pattern lengths and for all texts used in the experiments.

Table 4.12 summarises the results of Tables 4.8 to 4.11. The entries in Table 4.12 are in percentage form and describe how many more comparisons existing algorithms did than our BR algorithm. The figures are an average of the four different texts used. To calculate the difference as a percentage between our BR algorithm and the existing algorithms we used the following formula. The average number of comparisons was taken from the relevant cell in Tables 4.8 to 4.11 and divided by the value for that pattern length for our BR algorithm. This value was then deducted by 1 and multiplied by 100 to give the percentage difference between the two algorithms. An interesting observation of the existing algorithms when compared with the BR algorithm, is that for each individual text the percentages were within 1% for each specific algorithm. Each value in Table 4.12 is calculated by taking the difference as a percentage between each algorithm and our BR algorithm for each pattern length, adding them together and dividing by 4. For example, for a pattern length of 4 the BM algorithm takes on average 51.11% more comparisons than our BR algorithm. Note that the figures only include direct comparisons between the text and the pattern and not any text comparisons made during the calculation of a shift.

The result of a full search for the dictionary over all four texts is given in the last row of Table 4.12. From this we can see that the BM algorithm took on average 43.54% more comparisons than our BR algorithm (see 5th column, last row) for a complete search for all the words in the dictionary.

pat. len.	num.	KMP	BM	HOR	RAI	TBM	MS	LDI	QS	ZT	SMI
2	133	199.98	93.96	94.00	93.96	93.89	35.94	37.23	32.92	93.96	31.48
3	765	366.02	64.18	64.20	64.19	63.70	28.78	32.90	28.21	60.03	24.93
4	2178	449.02	51.11	50.86	50.90	50.77	28.25	31.09	25.77	43.19	19.73
5	3146	540.11	45.02	44.58	44.46	44.72	28.33	31.57	26.47	33.91	18.13
6	3852	626.30	42.42	41.83	41.68	41.91	30.02	32.38	27.32	27.71	16.42
7	4042	719.01	41.38	40.92	41.00	40.72	31.49	33.58	28.83	24.94	16.08
8	3607	807.61	40.58	40.28	40.35	39.95	32.27	34.99	30.10	21.67	15.49
9	3088	896.18	41.52	40.92	40.84	40.69	34.75	37.13	32.19	19.29	15.45
10	1971	982.63	42.19	41.69	41.79	41.16	36.62	39.31	34.37	17.75	15.64
11	1120	1067.87	44.14	43.67	43.79	42.97	38.57	42.14	37.18	17.06	16.32
12	593	1164.14	45.28	44.58	44.68	44.20	40.06	44.38	39.28	16.14	17.34
13	279	1245.53	47.88	47.22	47.32	46.36	42.26	46.69	41.61	12.65	17.54
14	116	1322.70	46.74	46.46	46.60	45.16	42.62	48.68	42.26	11.32	17.03
15	44	1426.02	51.20	51.51	51.59	49.23	44.73	52.89	45.29	8.72	19.00
16	17	1527.28	49.34	50.44	50.60	47.37	46.60	52.95	49.06	24.80	20.02
17	7	1598.50	45.29	44.51	44.58	43.42	40.22	50.20	45.01	6.72	16.95
18	4	1700.81	50.58	53.96	54.06	48.54	50.12	59.06	53.59	6.09	22.21
19	0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
20	1	1948.74	58.37	58.12	58.07	58.37	52.25	72.62	63.51	3.01	29.43
21	2	1947.96	57.38	63.98	63.99	56.32	57.59	64.09	57.50	2.22	21.84
22	1	2129.14	50.97	49.87	49.89	50.97	45.07	66.54	55.43	1.04	25.09
Total	24992	737.56	43.29	42.83	42.82	42.65	32.00	34.59	29.72	26.09	16.66

Table 4.12: *The average (of Tables 4.8- 4.11) percentage difference in the number of comparisons between existing algorithms and the BR algorithm*

We also measure the user time for these algorithms as the saving in the number of comparisons may be paid for by some extra overhead operations. We timed the search

of book1 of for all occurrences of 500 words from the UNIX dictionary. The words are of length 2 to 11 and there are 50 words of each length. The words were chosen at random from the UNIX dictionary. We show the average length of a shift performed by each algorithm in the second column. The percentage difference between the existing algorithms and the BR algorithm is shown in the third column. We used a 486-DX66 with 32 megabytes of RAM and a 100 megabyte hard drive running SUSE 5.2. The user time includes the time taken for any pre-processing and the reading of the text into memory. Each algorithm was evaluated ten times and the average time taken is given in Table 4.13. The timing was accurate to $\frac{1}{100}$ of a second but was rounded to the nearest second. The difference between the slowest and fastest time for each test for an algorithm was less than 0.2 of a second. The last column shows the percentage difference of the user time between existing algorithms and the BR algorithm.

If we list the algorithms in order of the average shift that they take from the highest to the lowest starting at the BM, we will get: BM, LDI, ZT, QS, MS, SMI and the BR. But, if we do the same for the timings we get BM, MS, SMI, LDI, QS, HOR, BY, RAI and the BR. The reason for the difference in the lists is due to overheads in traversing the data structures which are present in the algorithms for the calculation of the correct shift value. Therefore, we can not assume that because an algorithm has a higher average shift that it will be more efficient than another.

Algorithm	average shift	% difference	time in secs.	% difference
BF	1.00	708.00	3402	315.89
KMP	1.00	708.00	4727	477.87
DFA	1.00	708.00	3057	273.72
BY	5.61	44.03	987	20.66
BM	5.76	40.28	1518	85.57
AG	5.65	43.01	4396	437.41
HOR	5.72	41.26	1042	27.38
RAI	5.72	41.26	865	5.75
MS	6.40	26.25	1237	51.22
LDI	6.34	27.44	1115	36.31
QS	6.49	24.50	1094	33.74
ZT	6.38	26.65	1874	129.10
TBM	5.57	45.06	2240	173.84
SMI	7.11	13.64	1186	44.99
BR	8.08	N/A	818	N/A

Table 4.13: *The average shift and the user time in seconds*

We then considered eight other texts, 'Book2', 'news' and the six papers from the Calgary corpus [37]. The number of words and the number of characters of these texts are shown in Table 4.15. We searched for the same 500 random words from the UNIX dictionary for the BM, BR, BY, HOR, LDI, QS, RAI, and SMI algorithms. The reason for using different texts of different sizes was to check that the pre-processing of the BR didn't become too expensive as the text became smaller in size. We also needed to check that the distribution of the characters in the text didn't affect the speed of the BR algorithm.

	BM	BR	BY	HOR	LDI	QS	RAI	SMI	ZHU
Paper 1	103.7	56.0	68.2	71.3	76.6	74.7	59.3	81.3	169.9
Paper 2	161.8	86.8	106.2	111.2	120.1	116.9	92.4	126.5	247.1
Paper 3	93.2	50.1	61.2	64.0	69.2	67.4	53.3	72.8	164.9
Paper 4	26.7	15.5	17.6	18.2	19.8	19.2	15.1	20.9	85.5
Paper 5	23.3	13.9	15.7	16.2	17.8	17.1	13.5	18.7	82.2
Paper 6	74.2	40.2	48.7	51.0	54.5	53.2	42.4	58.2	143.3
Book 2	1195.0	639.0	784.0	820.0	884.0	862.0	681.0	934.0	1485.0
News	727.0	391.0	476.0	498.0	533.0	520.0	414.0	570.5	862.0

Table 4.14: *User times in seconds for the eight chosen texts*

	number of words	number of characters
Paper1	8512	53162
Paper2	13830	82205
Paper3	7220	47139
Paper4	2167	13292
Paper5	2100	11960
Paper6	6754	38111
Book1	139994	773635
Book2	101221	610856
News	53940	37711

Table 4.15: *The number of words and characters of the texts used in Table 4.14*

The results documented in Table 4.14 show that the BR algorithm is faster than the existing algorithms when the text is large. The RAI algorithm is the fastest algorithm for texts 'paper 4' and 'paper 5'. This is due to the time for the pre-processing in BR which is not as dominant in the RAI algorithm. The tests were carried out for a wide range of text sizes as shown in Table 4.15. The main reason for the speed of our BR algorithm is the improved maximum shift of $m+2$.

4.6 Conclusions

The experimental results show that the BR algorithm is more efficient than the existing algorithms in practice for most of the data sets from the Calgary Corpus [37]. Over our 4 random texts and 9 real texts where the BR algorithm is compared to the existing algorithms, our algorithm is more efficient for all but two of the texts. With the addition of punctuation and capital letters it does not affect the BR algorithm. So in the real world we would expect our savings to remain and make our BR algorithm competitive with the existing algorithms. It is also possible to apply some of our findings to what makes a fast algorithm to the existing algorithms. This may make them faster but we were concerned with the original algorithms that were devised by their authors.

Chapter 5

Two dimensional string matching algorithms

5.1 Introduction

Two dimensional string matching algorithms [5, 6, 49, 50, 13, 14] perform as follows. Given a text T $[1..n_1][1..n_2]$ find all occurrences of a pattern P $[1..m_1][1..m_2]$ in T. Note that in this chapter we use square matrices for our tests and so $n = n_1 = n_2$ and $m = m_1 = m_2$. We describe the existing two dimensional string matching algorithms and describe a new two dimensional algorithm (2D-BR). We prove that the new algorithm has a linear average-case time complexity. We compare the new algorithm with the existing algorithms by experimentation.

5.2 Existing two-dimensional string matching algorithms

The most basic of the two dimensional string matching algorithms is the naive or brute force (BF) algorithm. The BF algorithm for two dimensional pattern matching works in a similar way to the BF algorithm for one dimensional string matching. The pattern

and text are aligned so that P[1][1] is aligned with T[1][1]. The comparisons are done from left to right. If the first row of the pattern P[1][1..m] matches T[a][$k+1..k+m$] then the second row of the pattern is compared to the text starting at T[$a+1$][$k+1$]. This continues until a complete match of the pattern and the text or a mismatch occurs then as before in the one dimensional case the pattern matrix is moved one position to the right. The BF algorithm has a worst-case time complexity of $O(n^2m^2)$.

This worst-case was improved by Bird [35] and Baker [27] to $O(n^2 + m^2)$. The algorithm combines the Aho-Corasick multiple pattern matching algorithm [2] and the KMP algorithm [84] to form a two dimensional algorithm. The algorithm processes the text and identifies all occurrences of all pattern rows. Each row in the pattern is represented by a new symbol and the symbol replaces each occurrence of the pattern row in the text. The problem is now finding all occurrences of the string composed of new symbols in the text in the correct order.

The KMP algorithm itself can be adapted to two dimensional string matching. In one dimensional matching the algorithm compares the text and pattern character by character until either a mismatch or complete match is found. Upon a mismatch the pattern is shifted to the right by the greatest overlap with the old pattern position. In the two dimensional case this can be adapted by starting in the leftmost column and comparing a pattern row with a text subrow of length m. Proceeding in this fashion until a mismatch or match occurs. Upon a mismatch or match the pattern is shifted

to the right for the greatest overlap with the old pattern position and comparisons are resumed from this new location. The worst-case running time of the modified KMP algorithm is $O(n^2m)$.

The Zhu and Takaoka (ZT) algorithm [139] uses hashing to search for a pattern. The text is hashed into numeric values to form a one dimensional text. The pattern is hashed in the same way and a one dimensional pattern matching algorithm is used to search for the pattern. The worst-case running time is $O(n^2m^2)$.

5.3 The New Algorithm (2D-BR)

The new algorithm (2D-BR) reduces the number of comparisons required to search for a pattern. In one comparison we can check whether the entire pattern matrix isn't present in an area of the text [61]. We create an array of length σ, where σ is the size of the alphabet set called the Frequency array. In the Frequency array we record the frequency of each of the characters in the pattern. We look up the character at positions $T[(i+1)m][(j+1)m]$ for all $0 \leq i, j < \frac{n}{m}$ called the sample point in the Frequency array.

Note that when one entry in the matrix is examined we are only considering a $2m-1 \times 2m-1$ square centred around the sample point X.

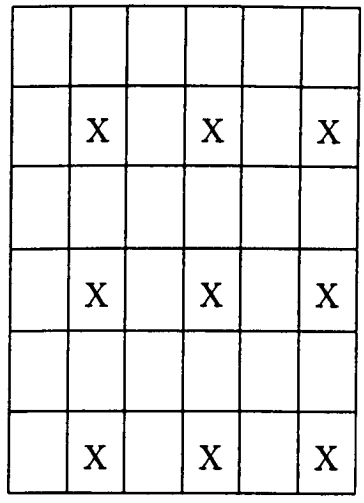

Figure 5.1: *Each X denotes a sample point (T[$(i+1)m$][$(j+1)m$] in the matrix that is compared with the Frequency array where T is a 6×6 matrix and P is a 2×2 matrix.*

There are three cases that arise from a comparison of the sample point and the Frequency array (T[$(i+1)m$][$(j+1)m$]) with the pattern:

- **Case 0:** The character at the sample point has a frequency of 0 in the Frequency array and therefore doesn't occur in the pattern. We then move to the next sample point in the text to be considered.

- **Case 1:** The character at the sample point has a frequency of 1 in the Frequency array. This means that there is only one occurrence of the character in the pattern. The pattern and text are aligned so that the occurrence of the sample point in the pattern is aligned with the sample point. The pattern and text are examined with the BF algorithm and after a partial or full match we move to the next sample point to be considered.

- **Case 2:** The character at the sample point has a frequency greater than one in the Frequency array. If case 2 occurred at $T[(i+1)m][(j+1)m]$ then we continue checking the sample points until either case 0 or case 1 occurs, while $j < \frac{n}{m}$. For the distance from the first occurrence of case 2 to the last occurrence of case 2 we use a one dimensional algorithm, the BR algorithm, to search for the first row of the pattern in the text. We begin the search at $T[(i+1)m-(m-1)][(j+1)m-(m-1)]$ for the i, j that give the location of the first occurrence of case 2. The search ends at $T[(i+1)m][(j+1)m]$ for the i, j that give the location of the last occurrence of case 2. For example, we consider a pattern of size 3×3 and a text of size 5×30 in Figure 5.2. For each of the sample points we give which case that has occurred. The shaded area represents the area that would need to be searched by the BR algorithm for the first row of the pattern. Upon a full match of the first row of the pattern with the text the remaining pattern characters are compared with the corresponding text characters. Upon a full or partial match of the pattern and the text, the pattern is moved according to the shift value calculated by the BR algorithm.

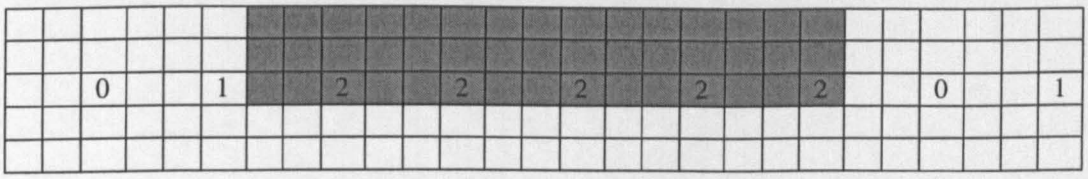

Figure 5.2: *Shaded area is compared with the first row of the pattern using the BR algorithm*

The new algorithm is called the two dimensional BR (2D-BR) algorithm. The worst case running time of the 2D-BR algorithm is $O(n^2m^2)$.

5.4 Average case analysis of the 2D-BR algorithm

Let σ be the size of the alphabet set, Σ. We assume that the characters in the text all have an equal frequency and so the probability of a match between a character of the pattern and text is $\frac{1}{\sigma}$.

The frequency of the character at the sample point in the pattern will tell us which case is used. For each case we need to know

- The number of patterns with no occurrences of the character at the sample point (case 0).
- The number of patterns with one occurrence of the character at the sample point (case 1).
- The number of patterns with one or more occurrences of the character at the sample point (case 2).

Note that we only need to calculate the number of matrices for a character of the alphabet as each character has the same number of matrices for each of the cases.

Lemma 5.1: The number of matrices that don't contain a specific character of the alphabet is $(\sigma - 1)^{m^2}$.

Proof: The number of possible matrices for an alphabet of size σ is σ^{m^2}. We want to know the number of matrices where a character in the alphabet doesn't occur in the pattern matrix. This is the same as how many matrices are possible with one character of the alphabet set removed $(\sigma - 1)$ to give $(\sigma - 1)^{m^2}$ □.

Lemma 5.2: The number of matrices that contain one occurrence of a specific character of the alphabet is $m^2 \times (\sigma - 1)^{m^2-1}$.

Proof: If the matrix contains only one of a specific character then the other $m^2 - 1$ positions in the matrix contain any number of the remaining characters in the alphabet. The number of combinations that the remaining characters can take is $(\sigma - 1)^{m^2-1}$. The character that must occur only once in the matrix can be placed in any of the m^2 positions in the matrix. Therefore there are $m^2 \times (\sigma - 1)^{m^2-1}$ possible matrices.

The number of matrices that contain more than one occurrence of a specific character is equal to the total number of possible matrices minus the number of matrices that contain one or no occurrences of the specific character. To give σ^{m^2} - $(\sigma - 1)^{m^2}$ - $m^2 \times (\sigma - 1)^{m^2-1}$ □

- The probability of case 0 is the number of matrices that don't contain a specific character divided by the total number of possible matrices, $\frac{(\sigma-1)^{m^2}}{\sigma^{m^2}}$.

- The probability of case 1 is the number of matrices that contain one occurrence of the specific character divided by the total number of possible matrices,

$\frac{m^2 \times (\sigma-1)^{m^2-1}}{\sigma^{m^2}}$.

- The probability of case 2 is the number of matrices that contain one or more of the specific character divided by the total number of possible matrices, $\frac{\sigma^{m^2} - (\sigma-1)^{m^2} - m^2 \times (\sigma-1)^{m^2-1}}{\sigma^{m^2}}$.

Case 2 is dominant when σ is small i.e. $\sigma \leq 32$. When case 2 is dominant the algorithm then behaves as the BR plus $([n_1/m_1]) \times ([n_2/m_2])$ extra comparisons. As the pattern size increases $([n_1/m_1]) \times ([n_2/m_2])$ decreases.

The average case time complexity of the 2D-BR algorithm is the upper bound total number of comparisons taken for the average case text.

Lemma 5.3: The upper bound total number of comparisons taken by the 2D-BR algorithm is $\frac{n^2}{m^2} + \frac{8n^2}{7}$.

Proof: The upper bound total number of comparisons taken is the upper bound number of comparisons taken at an attempt multiplied by the upper bound number of attempts made plus the number of sample points considered. The highest amount of comparisons are made when case 2 is found at all of the sample points. When this happens we use the BR one dimensional string matching algorithm to search the entire text for the first row of the pattern. Upon a match of the first row of the pattern with the text we compare the remaining characters in the pattern with the corresponding text characters until we have a mismatch or a full match. The distance the pattern is shifted is calculated by the BR algorithm. □

From Lemma 4.1 in Chapter 4 we know the upper bound number of comparisons at an attempt for a one dimensional pattern is $\sum_{i=0}^{m-1}(\frac{1}{\sigma^i})$. The number of entries in a two dimensional pattern is m^2 and therefore we replace m with m^2. The upper bound number of comparisons taken at an attempt for a two dimensional text is $\sum_{i=0}^{m^2-1}(\frac{1}{\sigma^i})$. If we assume that $\sigma > 1$, then the above equation is maximised when $\sigma = 2$. To give, $\sum_{i=0}^{m^2-1}\frac{1}{2^i}$. As m tends to infinity the equation tends to its limit of 2. The upper bound number of comparisons taken at an attempt is 2 comparisons.

The upper bound number of attempts made is calculated by dividing the size of the text by the lower bound shift. From Lemma 4.2 in Chapter 4 the lower bound for a shift is $\frac{7}{4}$ for the BR algorithm. This is also the lower bound shift for the two dimensional text as we are using the BR algorithm to shift the pattern. The upper bound number of attempts made is $\frac{n^2}{\frac{7}{4}} = \frac{4n^2}{7}$ attempts.

The upper bound total number of comparisons is the upper bound number of comparisons made at an attempt multiplied by the upper bound number of attempts made plus the number of sample points considered. This gives $2 \times \frac{4n^2}{7} + \frac{n^2}{m^2} = \frac{8n^2}{7} + \frac{n^2}{m^2}$.

Theorem: The 2D-BR has a linear average case running time of $O(N+M)$ where N and M are the sizes of the text and pattern respectively.

Proof: For the pattern to be two dimensional m has to be greater than or equal to 2. The equation $\frac{8n^2}{7} + \frac{n^2}{m^2}$ is maximised when $m = 2$. To give $\frac{8n^2}{7} + \frac{n^2}{4} =$

$\frac{39n^2}{28}$. Therefore the 2D-BR algorithm has a linear average case time complexity of $O(N+M)$ where N and M are the sizes of the text and pattern respectively. □

alphabet	2x2	3x3	4x4	5x5	6x6	7x7	8x8	9x9	10x10
2	68987	75849	75300	73928	73165	72704	72405	72200	72054
4	31775	25687	23696	21168	19046	17535	16480	15753	15277
8	23221	15193	13642	12967	11941	10744	9652	8748	8010
16	19441	10928	8637	8185	8160	8017	7651	7141	6662
32	17552	8913	6134	5187	4993	5094	5248	5331	5302
64	16594	7924	4962	3712	3183	3019	3048	3174	3329
128	16111	7434	4416	3055	2366	2011	1846	1798	1826
256	15869	7189	4156	2762	2019	1590	1333	1182	1100

Table 5.1: *Estimated number of comparisons taken*

For the stated σ, m and a text of 62500 characters the estimated number of comparisons are given in Table 5.1. We then tested 2D-BR to see how many comparisons the algorithm takes in practice. The results are given in Table 5.2.

So for larger alphabets when case 2 becomes dominant $([n1/m1]) \times ([n2/m2])$ is quite small. So for larger alphabets the difference between the 2D-BR algorithm and the BR algorithm would be very small. As σ increases the number of actual comparisons is less than the estimated number of comparisons.

alphabet	2x2	3x3	4x4	5x5	6x6	7x7	8x8	9x9	10x10
2	89168	82389	75782	75008	72949	70455	67217	63843	65892
4	31905	33265	27961	24635	21491	19267	17588	16039	15173
8	19082	15657	15890	14824	13117	11580	10357	9267	8489
16	16510	9223	8605	8941	9163	8642	8010	7251	6684
32	15865	7581	5391	4821	4997	5278	5542	5508	5404
64	15686	7101	4296	3266	2879	2845	2935	3156	3333
128	15640	6945	3975	2704	2052	1807	1657	1721	1695
256	15628	6904	3875	2554	1783	1394	1181	1074	976

Table 5.2: *Actual number of comparisons taken*

5.5 Practical evaluation of the algorithms

From the theoretical evaluation of the algorithms we can see that the best algorithm is the Bird and Baker algorithm. We wanted to know how the algorithms performed in practice. We already know that the Bird-Baker and Zhu-Takaoka algorithms required a lot of space and preprocessing [27, 35, 139] which would take more time than the other algorithms and so these algorithms were omitted.

For our experiments we used alphabets of size 2, 4, 8, 16, 32, 64, 128 and 256. For each algorithm we randomly generated 100 matrices for each pattern size from 2×2 to 10×10 and generated a text of size 500×500.

We used a 486-DX66 with 32 megabytes of RAM and a 100 megabyte hard drive running SUSE 5.2. Each algorithm was evaluated ten times and the average user time taken is given in Tables 5.3 to 5.10. The timing was accurate to 1/100 of a second.

The difference between the slowest and fastest time for each test for an algorithm was less than 0.2 of a second.

	2x2	3x3	4x4	5x5	6x6	7x7	8x8	9x9	10x10
BF	10.36	9.77	9.35	8.49	10.82	10.66	10.40	9.76	9.06
BM	13.66	9.45	8.52	8.82	7.56	4.91	4.27	4.39	6.16
BR	8.21	6.88	6.63	7.49	6.49	4.53	3.78	4.34	3.72
KMP	14.11	14.38	14.54	14.34	13.90	14.01	14.22	16.28	16.82
2D-BR	4.24	3.53	1.15	2.55	0.77	0.43	1.89	0.52	1.67
SMI	12.90	8.03	8.15	6.60	5.15	5.64	4.91	5.02	4.47

Table 5.3: *Time in seconds to search for 50 matrices when* $\sigma = 256$

	2x2	3x3	4x4	5x5	6x6	7x7	8x8	9x9	10x10
BF	11.28	11.54	12.34	11.24	11.49	11.47	12.17	11.48	11.98
BM	13.51	13.51	10.12	8.19	9.80	6.83	6.89	7.00	6.42
BR	10.93	8.35	8.40	7.80	8.61	6.59	6.14	6.89	6.18
KMP	16.35	16.12	15.81	15.39	16.32	17.28	16.49	16.63	16.09
2D-BR	6.65	7.41	4.06	3.70	3.95	4.23	4.35	6.40	5.81
SMI	11.72	10.46	9.35	9.00	8.52	7.91	8.20	7.41	7.16

Table 5.4: *Time in seconds to search for 50 matrices when* $\sigma = 128$

From Tables 5.3 and 5.4 we can see that the 2D-BR algorithm is the fastest algorithm for the tests conducted. In these tests the alphabet is large and so the probability of case 2 is low. As cases 0 and 1 occur more frequently we get large shifts of m^2 more often. In Table 5.5 the performance of the 2D-BR algorithm starts to suffer as the pattern size increases. This is due to case 2 becoming dominant. This observation is also true in Tables 5.6 to 5.10.

	2x2	3x3	4x4	5x5	6x6	7x7	8x8	9x9	10x10
BF	8.99	8.75	10	10.19	9.75	9.21	9.56	9.73	10.24
BM	12.26	9.63	8.21	8.45	6.25	5.33	4.98	5.27	7.79
BR	9.37	6.94	7.96	6.02	4.19	4.53	4.95	4.93	4.41
KMP	13.79	14.08	14.2	14.26	14.1	16.14	16.36	16.2	16.64
2D-BR	4.49	3.89	3.38	4.11	5.23	5.43	6.78	7.5	7.17
SMI	10.26	8.4	7.5	6.99	8.12	5.45	5.84	4.86	4.59

Table 5.5: *Time in seconds to search for 50 matrices when* $\sigma = 64$

	2x2	3x3	4x4	5x5	6x6	7x7	8x8	9x9	10x10
BF	9.14	9.05	8.80	9.19	8.81	9.67	9.52	12.49	9.43
BM	11.29	12.07	8.19	6.19	5.45	5.34	4.77	5.96	3.03
BR	7.03	6.89	5.37	5.34	4.48	4.62	4.40	3.62	3.63
KMP	12.28	13.53	14.27	13.47	14.87	14.33	13.94	14.39	14.16
2D-BR	5.26	4.22	6.05	11.96	10.40	9.68	9.14	6.51	7.10
SMI	9.55	9.40	6.85	6.60	7.20	6.84	4.58	4.97	6.14

Table 5.6: *Time in seconds to search for 50 matrices when* $\sigma = 32$

	2x2	3x3	4x4	5x5	6x6	7x7	8x8	9x9	10x10
BF	9.88	10.76	9.96	10.06	9.75	9.47	8.92	10.80	10.04
BM	11.96	10.06	8.96	8.52	7.72	6.28	5.48	7.20	5.80
BR	8.72	7.83	7.59	5.90	5.99	5.88	3.76	5.66	4.59
KMP	14.34	14.76	14.64	16.24	14.86	14.90	13.37	14.55	14.70
2D-BR	9.59	13.02	18.68	21.78	14.77	12.02	6.93	6.21	5.28
SMI	12.61	10.58	7.96	9.13	7.56	6.43	7.98	5.20	6.01

Table 5.7: *Time in seconds to search for 50 matrices when* $\sigma = 16$

	2x2	3x3	4x4	5x5	6x6	7x7	8x8	9x9	10x10
BF	10.23	9.62	9.30	8.87	10.46	10.70	11.49	10.38	9.62
BM	12.11	10.28	7.95	10.59	7.52	7.43	6.54	6.36	6.28
BR	8.73	8.72	7.27	6.03	6.05	6.30	5.65	5.27	5.44
KMP	14.64	13.55	14.50	14.87	13.86	14.24	14.42	14.42	14.94
2D-BR	23.84	34.71	27.90	15.13	7.42	6.58	6.64	4.89	4.96
SMI	11.62	8.86	9.05	8.51	6.28	7.08	8.03	5.99	6.56

Table 5.8: *Time in seconds to search for 50 matrices when* $\sigma = 8$

	2x2	3x3	4x4	5x5	6x6	7x7	8x8	9x9	10x10
BF	14.03	11.62	11.95	11.76	11.18	12.67	11.51	12.28	12.81
BM	15.54	12.08	10.51	10.70	10.26	9.04	8.77	8.32	9.43
BR	11.72	10.96	10.45	8.41	8.31	8.51	8.41	7.45	7.91
KMP	16.24	16.31	15.40	16.12	17.19	16.91	15.82	17.77	17.01
2D-BR	8.21	44.16	15.91	9.92	7.80	8.40	7.69	7.65	8.17
SMI	14.97	12.43	11.26	10.28	10.38	8.93	10.55	9.48	9.59

Table 5.9: *Time in seconds to search for 50 matrices when* $\sigma = 4$

	2x2	3x3	4x4	5x5	6x6	7x7	8x8	9x9	10x10
BF	15.58	15.41	15.40	14.52	14.24	13.79	13.90	14.93	13.80
BM	17.18	16.54	15.21	13.50	11.20	10.89	8.48	8.14	7.81
BR	20.35	19.56	16.83	16.52	15.02	14.73	13.70	15.18	15.45
KMP	19.14	21.49	14.03	15.42	15.79	15.53	15.07	15.44	18.00
2D-BR	80.58	22.86	18.48	17.22	15.67	15.80	14.57	15.49	15.91
SMI	20.88	18.41	16.81	16.86	17.22	14.12	13.54	14.60	14.87

Table 5.10: *Time in seconds to search for 50 matrices when* $\sigma = 2$

The probability of case 2 occurring is approximately 1 when $\sigma = 2$. This is when the 2D-BR became one of the slowest algorithms. As the pattern size increases the algorithm performs as the BR algorithm but with some extra time taken checking the extra $\frac{n^2}{m^2}$ comparisons.

The 2D-BR is the best algorithm to use when σ is large. This is the case when searching in image files.

5.6 Conclusions

The reason for the speed of the 2D-BR algorithm is due to the fact that the algorithm exploits the characteristics of the matrices. The only disadvantage of our 2D-BR algorithm is that it requires a large alphabet due to its complex searching procedure. The large alphabet ensures that cases 0 and 1 are used for the searching phase of the algorithm more frequently. The 2D-BR algorithm performs best with larger alphabets due to the probability of case 2 being reduced.

In our tests we used square matrices. If the pattern was a rectangle or an irregular shape as in [59, 75] we could extend the 2D-BR algorithm to search in these texts.

Chapter 6

Compression algorithms

6.1 Introduction

The dictionary definition of compression is: "to squeeze together or compact into less space" [43]. Text compression [29] is exactly that, taking a file and compacting it so that it takes less space. How the compacting is performed and how much compacting is done is the difference between the various algorithms available. Although the times to encode and decode texts are considered, normally we are interested in maximizing the factor (ratio) by which the text has been compressed. The aim of a text compression algorithm is to decrease the number of bits required to represent a piece of text.

The compression ratio of a text is the amount of space saved by compressing the text. The compression ratio is calculated using the following equation:

$$compression\ ratio = \frac{originalsize - encodedsize}{originalsize}$$

When comparing compression algorithms [89] to see which gives the greater amount of compression we use the compression ratio. Although an algorithm may have a higher compression ratio than another it may take a long time to compress the text. Therefore compression algorithms are evaluated on their compression ratio and the time taken to compress a text.

There are two types of compression algorithm, lossless compression and lossy compression. Lossless meaning that none of the information in the source file is lost when the file is decompressed. The file is compressed and when it is decompressed it is identical to the file that was compressed. In lossy compression some of the information may be lost during the compression. Lossy compression attempts to eliminate redundant or unnecessary information. Most music compression technologies, such as MP3, use a lossy technique. For text compression we use lossless compression algorithms as we need to compress the source file and decompress it to recover the original source file. There are many compression algorithms for text compression. Text compression algorithms use the statistical data and structure of the text file to compress the file. There are two famous compression algorithms that are used in text compression, Huffman Coding [73] and Lempel-Ziv encoding [134, 140, 141].

6.2 Huffman Encoding

Huffman encoding [73] can be used to compress data such as text or images [72]. Text files contains characters and formatting commands. Most text files use the 128 characters in the ASCII character set. They are numbered from 0 to 127. Numbers 0 to 31 and 127 are used to represent control or formatting characters and 32 to 126 represents the alphanumeric characters in the text. Each of the characters is represented in the file by eight bits or one byte. Although we only need 7 bits to represent the numbers from 0 to 127. Numbers 128 to 255 are used for some characters in the text that may be from the extended ASCII character set such as Greek letters, math symbols and various geometric patterns.

In Morse code [98] the more frequently used letters have shorter patterns associated with them. This is so that a message can be passed quickly and accurately. Although both the sender and recipient must know Morse code to use this form of signalling. This is the idea behind Huffman encoding. More frequent characters can be assigned shorter bit patterns and less frequent characters can be assigned longer bit patterns. Huffman encoding can be dynamic or static. First, we will consider static Huffman encoding and give an example of how the algorithm compresses a text. A visualisation of Huffman encoding can be found at [121].

To use static Huffman encoding we must first evaluate the frequencies of the characters in the text. To do this we must read the text character by character

updating the relevant frequencies from the start to the end. Consider the following frequency table:

Character	Frequency
A	70
B	33
C	27
D	21
E	12
F	7
END	1

Table 6.1: *The frequency of each of the characters in the text.*

To construct the bit pattern for each character we build a Huffman tree (binary tree) and then reading the Huffman tree gives the relevant bit pattern. Let each of the characters in the text be a leaf in the Huffman tree. To build the tree we perform the following steps.

Step 1: Pick the nodes n_1 and n_2 that have the smallest weights.

Step 2: Replace them with a new node whose children are n_1 and n_2 and whose weight is the sum of the weights of n_1 and n_2.

Each time we perform these steps we will replace two nodes in the alphabet with one. Until only one single node remains and this node is the root of the Huffman tree.

The two nodes with the smallest frequencies are END and F in Table 6.1. We make these the children of a parent node and replace them in the frequency table

with END,F = 8.

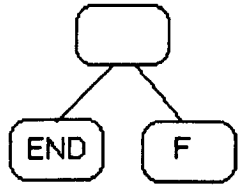

Figure 6.1: *END and F added to the Huffman tree*

Listing the frequencies in order we now get A=70, B=33, C=27, D=21, E=12 and END,F=8. The two nodes with the smallest frequencies are END,F and E. We make these the children of a root node and replace them in the frequency table with END,F,E = 20. Note that the E and the root of END,F are on the same level of the tree.

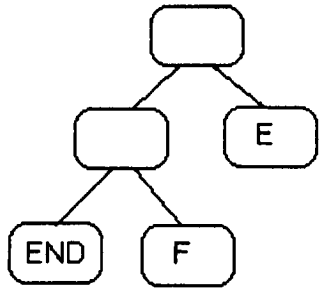

Figure 6.2: *E added to the Huffman tree*

Listing the frequencies in order we now get A=70, B=33, C=27, D=21 and END,F,E=20. The two nodes with the smallest frequencies are END,F,E and D. We make these the children of a root node and replace them in the frequency table with END,F,E,D = 41. Note that the D and the root of E are on the same level of the tree.

Listing the frequencies in order we now get A=70, END,F,E,D=41, B=33 and

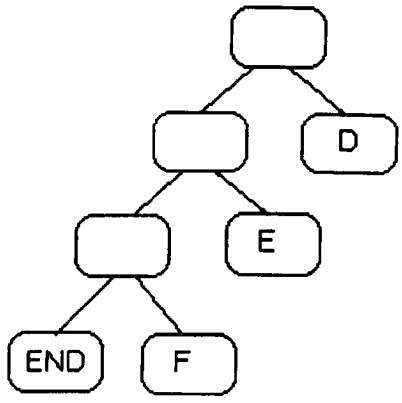

Figure 6.3: *D added to the Huffman tree*

C=27. The two nodes with the smallest frequencies are C and B. We make these the children of a root node and replace them in the frequency table with C,B = 60. Note that the subtree containing C,B is not connected to the main tree.

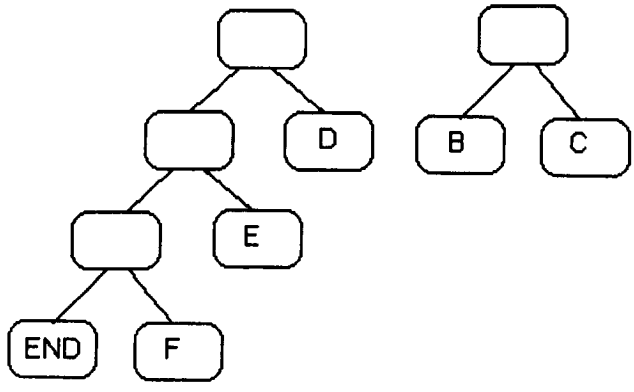

Figure 6.4: *C and B are added but not connected to the main tree*

Listing the frequencies in order we now get A=70, C,B=60 and END,F,E,D=41. The two nodes with the smallest frequencies are C,B and END,F,E,D. We make these the children of a root node and replace them in the frequency table with END,F,E,D,C,B = 101. Note that the C,B and D are on the same level of the tree.

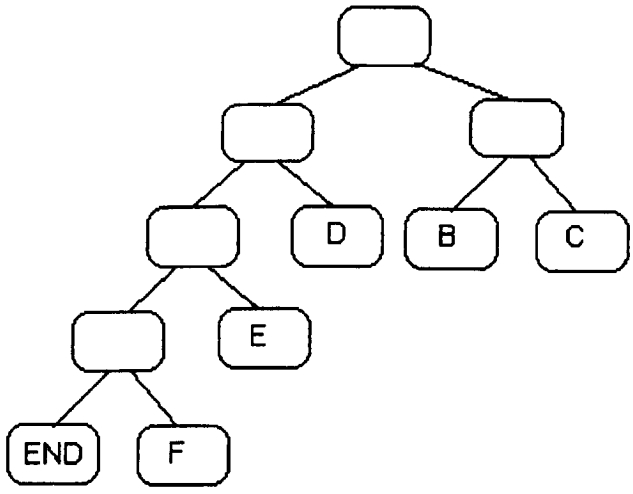

Figure 6.5: *C and B subtree connected to the Huffman tree*

Listing the frequencies in order we now get A=70 and END,F,E,D,C,B=101. The two nodes with the smallest frequencies are A and END,F,E,D,C,B. We make these the children of a root node and replace them in the frequency table with END,F,E,D,C,B = 101.

The tree is now complete and we can now read the bit patterns related to each character. The level which each character is on is the length of the bit pattern associated with it. For example E will have a bit pattern of length 4. Note that the root of the tree is level 0. To get the bit pattern we traverse the tree to each of the characters. If we traverse a left path we record a 0 and if we traverse a right path we record a 1. If we traverse left, right and then left and we are at B. So B has a bit pattern of 010. Using the same method for each character we get the following bit patterns as shown in Table 6.2.

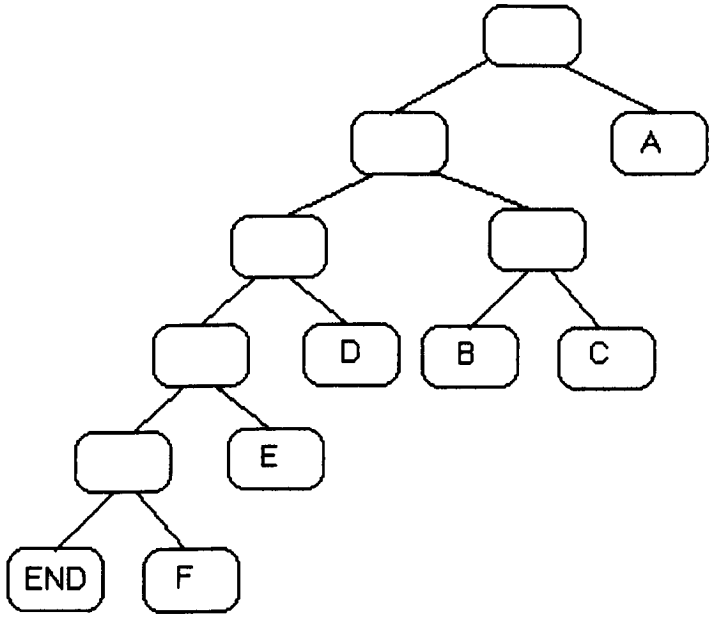

Figure 6.6: *A added to the Huffman tree and the tree is complete*

Character	Frequency	Bit Pattern	Bit Pattern Length	Freq × BPL
A	70	1	1	70
B	33	011	3	99
C	27	010	3	81
D	21	001	3	63
E	12	0001	4	48
F	7	00001	5	35
END	1	00000	5	5

Table 6.2: *The bit patterns and their lengths for an Huffman encoding of the frequencies in Table 6.1*

In Table 6.2 the fourth column shows the length of each bit pattern. Multiplying this figure with the frequency of a character gives the number of bits required to represent that character using Huffman encoding. So the total number of bits required is 401 bits. The original file took $171 \times 8 = 1368$.

The set of distinct characters in the text is known as the dictionary. The dictionary is stored as a binary sequence that allows us to reconstructed our encoding tree. If we let 0 represent an internal node and 1 represent a leaf then we can use a binary string to represent the tree. We start with a fixed 6 bit string which indicates how many bits each character in the original text requires to represent it. In the above example we are storing ASCII characters which have a value between 0 and 127. So we will need 7 bits to represent each character. So we will output 7 which is 000111 to our encoded file.

There are 7 leaves and 6 nodes in our tree in Figure 6.6. We traverse the tree in preorder: visit the root, traverse the left subtree, traverse the right subtree. So for the tree in Figure 6.6 we would get 0000011110111. Note that this does not include the data required to represent the characters at each of the leaves. At each of the leaves we have the value of the character at that position in the tree. When encoding the tree each time we output a 1 to indicate a leaf we record the character at that leaf. In this example each character in the text requires 7 bits to represent it. So the 7 bits of data after a 1 indicate the character at that leaf. The binary representations of each of the

ASCII characters in the dictionary are A = 1000001, B = 1000010, C = 1000011, D = 1000100, E = 1000101, F = 1000110 and END = 0000100. So the full encoding of the dictionary is 00000100001001100011011000101110001000110000101100001111000001. To reconstruct the tree we would use a preorder traversal mapping the nodes and leaves, and decoding the character at the leaf each time we add a new leaf.

So we represent the above tree using 13 bits and we use 49 bits to show the information stored at each of the leaves. So we used 6 + 13 + 49 = 68 bits to represent the dictionary. So the total number of bits required is 469 bits. The original file took 1368 bits. So the compression ration is $\frac{1368-469}{1368} = 0.65716$. So we have a compression ratio of 65.72%.

Adaptive or dynamic Huffman encoding [85, 132, 133] works in a similar way to static Huffman encoding only the Huffman tree is constantly updated. The tree is first populated with only the first character of the text. As the characters are read from the text then the tree is built and modified so that the characters that appear with the highest frequency have the smallest bit patterns.

Although Huffman encoding has a good compression ratio there are a few problems associated with it. For example if one of the bits in the text is changed then this can cause the entire text to be changed or cause the wrong character to be printed. These errors that can occur mean that the decompressed file may not be an exact copy of the original. Standish [126] shows that Huffman codes should recover from

errors with only a partial loss of data. This recovery from errors is known as self synchronisation. A difficulty of self-synchronisation is that it provides no indication that an error has occurred. In [128] it is proven that a code is never self-synchronising if and only if none of the proper suffixes of the codewords are themselves codewords.

An algorithm is introduced for constructing an optimal Huffman code for a weighted alphabet in [86]. An optimal Huffman code yields the best possible code for a collection of symbols and frequencies. The algorithm has a worst case time complexity of $O(nL)$ where n is the size of the weighted alphabet and each code string must have a length no greater than L.

6.3 Lempel-Ziv encoding and its derivatives

Lempel and Ziv decided to use the structure of the text to compress it. Most texts contain repeated patterns, be they repeated phrases, words, suffixes, prefixs or characters. Lempel and Ziv devised three algorithms all based around a similar idea. The first of which was LZ77 [140] which was first documented in 1977. The algorithm replaced reoccurring strings of characters with pointers to earlier occurences of that string.

In the LZ77 compression of a text the coding position is the position of the character that is currently being coded. A window of size w that contains w characters from the coding position backwards, where the characters in the window are the last

w characters processed. A look ahead buffer which is the character sequence from the coding position to the end of the input. The algorithm searches for the longest match with the beginning of the look ahead buffer. Upon a match between the window and the look ahead buffer, a pointer is output giving the position and length of the match. The position is given as the distance away from the current coding position. It is possible that not even one character of the window will match with the character at the coding position. If there is no match then a null pointer and the character at the coding position is the output. Otherwise, after each pointer it outputs the first character in the look ahead buffer after the match.

In Table 6.3 we show the positions of the characters in '*AAGTCTGTCA*' and we show the full encoding in Table 6.4

Pos.	1	2	3	4	5	6	7	8	9	10
Char.	A	A	G	T	C	T	G	T	C	A

Table 6.3: *Position of the characters in the string 'AAGTCTGTCA'*

In Table 6.4, step is the number of times the algorithm has iterated. The position indicates the current coding position. Match shows the longest match with the characters in the window. Char shows the first character after the match. If there is no match then Char is the character at the coding position. Output shows the output in the form (D,L)C where D is the distance to the matching characters in the window, L is the length of the match and C is the character after the match or the character

Step	Pos.	Match	Char	Output
1	1	NONE	A	(0,0)A
2	2	A	G	(1,1)G
3	4	NONE	T	(0,0)T
4	5	NONE	C	(0,0)C
5	6	T	G	(2,1)G
6	8	TC	A	(4,2)A

Table 6.4: *LZ77 encoding of the string 'AAGTCTGTCA'*

at the coding position if no match exists.

We firstly set the coding position to the beginning of the text string. We then search for the longest match in the window to left. As the window is empty there is no match and the null pointer (0,0) is output and the character at the coding position is '*A*'. We then move to the next coding position which is the next character in the text string to the right that hasn't been output. In this case the second '*A*'. The '*A*' matches with the '*A*' in the window and so we output a pointer and the character that is after the character that matched, namely '*G*'. The pointer would be (1,1) as the distance from the coding position to the matching character is 1 and the length of the match is 1. At the sixth and final step we output (4,2)A. This means that there is a match in the window of length 2 that is 4 positions away from the current coding position. The current coding position is 8 and TC is repeated 4 positions to the left starting at 4. The '*A*' is the first character in the text string after the match.

The window usually contains between 4,000 and 64,000 characters. Once the

window is full it is shifted to the right to keep it one position to the left of the coding position. The characters that are not in the window are not considered when searching for a match. This process of searching for matches can be time consuming, but decoding the compressed file is simple and fast. The pointers can easily be converted back to the characters that are represented by them. The LZ77 algorithm offers a very good compression ratio but the amount of time taken for the compression is a major drawback.

The second Lempel-Ziv algorithm is the LZ78 [141] which uses a dictionary to create the pointers rather using a window. As the text string is compressed a dictionary is built containing already scanned characters and strings. We output the codes or pointers to the compressed file in the following format (I,C), where I is the index of the dictionary string that has matched with the characters at the coding position and C is the first character after the matching characters in the text string. If there is no match between the dictionary and the character at the coding position then we output (0,C), where C is the character at the coding position. In Table 6.6 we show the full encoding of the characters in '*AAGTCTGTCTCA*' using LZ78. The positions of the characters are shown in Table 6.5

Pos.	1	2	3	4	5	6	7	8	9	10	11	12
Char.	A	A	G	T	C	T	G	T	C	T	C	A

Table 6.5: *The positions of the characters in the string 'AAGTCTGTCTCA'*

Step	Pos.	Dictionary	Index	Output
1	1	A	1	(0,A)
2	2	AG	2	(1,G)
3	4	T	3	(0,T)
4	5	C	4	(0,C)
5	6	TG	5	(3,G)
6	8	TC	6	(3,C)
7	10	TCA	7	(6,A)

Table 6.6: *The LZ78 encoding of the string 'AAGTCTGTCTCA'*

In Table 6.6, step is the current iteration of the algorithm. Position is the current coding position. Dictionary shows what string has been added to the dictionary. The index for each entry in the dictionary is the step number.

To compress '*AAGTCTGTCTCA*' we start with an empty dictionary and start at position 1 in the text string. We search the dictionary for the character at the coding position, '*A*', but the dictionary is empty and so we don't find a match. We put '*A*' in the dictionary and it has an index of 1 which is the step number. We output (0,A) to the compressed file. We then move to position 2 and search for '*A*' in the dictionary. We find a match and try to extend the match. The string '*AG*' is not in the dictionary and so the longest possible match is '*A*'. We output (1,G) as the index of the dictionary entry '*A*' is 1 and the first character after the match is '*G*'. We also enter '*G*' into the dictionary and it has an index of 2. The next two coding positions are not in the dictionary and both single characters are entered into the

dictionary with 'T' and 'C' having indexes of 3 and 4 respectively. We output (0,T) and (0,C) to the compressed file. When we check for a match with the next coding position 6 equal to 'T' and the dictionary we get a match. We try to extend the match but 'TG' is not in the dictionary and so we enter 'TG' into the dictionary and give it an index of 5. We output (3,G) to the compressed file as 3 is the dictionary index for 'T'. We continue this process on to the end of the string. Note that for step 7 we search the dictionary for 'T' and get a match and then try to extend the match to 'TC' which also matches. We then search for 'TCA' which is not in the dictionary. We enter 'TCA' into the dictionary and give it an index of 7 and output (6,A) to the compressed file as 6 is the index for 'TC'.

The decoding process is simple and the dictionary is rebuilt in a similar way to that used to encode the text string. The compression ratio is very good.

The third Lempel-Ziv algorithm is a modified version of the LZ78 algorithm. The LZW algorithm [105, 134] was devised by Lempel, Ziv and Welch. The algorithm removed the need for characters in the compressed file. The compressed file is a string of numbers related to the entries in the dictionary and their indexes. As there are no characters in the compressed file then the dictionary cannot be empty at the beginning of the compression process. The dictionary contains each character in the alphabet being used in the text string and each entry is indexed in the dictionary from 1 to the alphabet size. As in LZ78 we build a dictionary of strings that have already been

scanned. As before we try to find the longest possible match from the coding position in the dictionary. When we get a mismatch we output the index of the last string to match the characters in the dictionary. The string that didn't match is entered into the dictionary and is indexed with (the alphabet size + current iteration/step number). Unlike in LZ78 the next coding position is the character that caused the mismatch between the characters in the text string and the dictionary. This process continues until there are no more characters to be compressed. In Table 6.7 we show how '*AAGTCTGTCTCA*' would be compressed using LZW. The positions of the characters is the same as in Table 6.5. Note that there are 4 characters in the alphabet and so they would be assigned the numbers 1 to 4 as follows: A = 1, C = 2, G = 3 and T = 4.

Step	Pos.	Dictionary	Index	Output
1	1	AA	5	1
2	2	AG	6	1
3	3	GT	7	3
4	4	TC	8	4
5	5	CT	9	2
6	6	TG	10	4
7	7	GTC	11	7
8	9	CTC	12	9
9	11	CA	13	2
10	12	n/a	n/a	1

Table 6.7: *The LZW encoding of the string 'AAGTCTGTCTCA'*

In Table 6.7 step is the number of iterations that the algorithm has performed. Position is the current coding position. Dictionary is the strings that have been entered into the dictionary and index is the number that has been assigned to them. Output is what is written to the compressed file.

When compressing '*AAGTCTGTCTCA*' we start with a dictionary containing the four characters A, C, G and T numbered as above. We then scan in '*AA*' which is not in the dictionary and so we enter it in the dictionary with index equal to 5. We output the value of '*A*' (1) to the compressed file. We then move to position 2 and search for '*AG*' in the dictionary. It isn't there so we enter it into the dictionary with an index of 6. We output the value of '*A*' (1) to the compressed file. We then move to position 3 and search for '*GT*' which is not in the dictionary. We enter it into the dictionary with an index of 7. This process is repeated to the end of the text string. Note that for step 7 we search for '*GT*' and we find a match. We try to extend the match by searching for '*GTC*' but that is not in the dictionary. We enter '*GTC*' in the dictionary with an index of 11. We output the index of '*GT*' to the compressed file which is equal to 7.

So we have compressed '*AAGTCTGTCTCA*' to the numbers 1, 1, 3, 4, 2, 4, 7, 9, 2 and 1. Each of the numbers are represent by a 12 bit binary string in the compressed file. So our compressed file would take $10 \times 12 = 120$ bits to store the compressed file. The original file took $12 \times 8 = 96$ bits to store it. Although the

compressed file from this example is larger than the original file for larger texts we achieve much improved compression ratios.

An improvement to LZW encoding is shown in [71] which improves the compression ratio without a significant loss in speed. In [30] a method is described that decreases the amount of time required to build the encoding dictionary. This is done by comparing the available data structures and introducing two new data structures designed specifically for the task of Lempel-Ziv compression.

Once a text has been compressed we still need to be able to access the data contained in the text. This has led to the development of compressed string matching algorithms. These algorithms allow the user to search for a pattern in a text without the need to decompress the text [7, 100, 96, 118]. As the compressed text is smaller than the original text this can increase the speed of the search. Compressed string matching has been performed in a number of compressed texts and is not limited to one type of compression.

A theoretical analysis of searching in the Lempel-Ziv compressed files is given in [58, 65, 66, 67, 78]. A practical evaluation of string matching in the Lempel-Ziv file is given in [83, 102, 116]. The problem of multiple matching in the LZW compressed text is discussed in [81]. Multiple pattern matching [47, 101, 136] is searching for multiple patterns in one pass of the text. Existing string matching algorithms have also been adapted to search in the compressed Lempel-Ziv compressed file. The Shift-

AND algorithm is adapted and used in [82]. The Boyer-Moore algorithm is adapted and used in [103, 119].

6.4 Byte pair encoding

Byte pair encoding compresses two characters in the text in one byte. In English texts the characters in the special ASCII set (numbers 128-255) are normally not used. This means that for each character in the text 8 bits are being used to store information that can be stored using 7 bits. This means we have 128 characters that are unused. The frequency of pairs of characters or digrams are taken and the most frequent pairs or digrams are assigned to use ASCII values (128 to 255). Optimal compression would produce a compressed file that would be 50% of the size of the original. This method is used in [93, 117] and a method is explained for searching in the compressed file.

Chapter 7

String matching in an efficiently stored DNA text

7.1 Introduction

String matching and Compression are two widely studied areas in computer science [47]. String matching is detecting a pattern P of length m in a larger text T of length n. Compression involves transforming a string into a new string which contains the same information but whose length is as small as possible. These two areas naturally lead to Compressed String Matching, i.e. searching for a pattern in a compressed text. This method will save both space and time.

In this chapter we describe a String Matching algorithm to search for a pattern in an efficiently stored text. A DNA text (or molecule) encodes information, which by convention is represented as a string over the DNA alphabet, {A, C, G, T}. String Matching in an efficiently stored DNA text is useful for the following reasons. Although the cost of memory is reducing, the sizes of DNA databases are growing exponentially. A typical question in molecular biology is whether a pattern (boundary)

occurs in a DNA text, i.e. we don't need to view the text.

Optimal efficient storage will devote two bits to represent each DNA character, if each character is drawn uniformly at random from the DNA alphabet and that all positions in the text are independent [91].

7.2 Efficient storage of a DNA text

In the DNA alphabet, Σ, there are four characters, namely A, C, G and T. As there are only 4 possible characters in a DNA text we can represent the characters with the function, f: $\Sigma \rightarrow [0 .. 3]$, such that f(A) = 0, f (C) = 1, f (G) = 2, and f (T) = 3.

After we replace the characters in a text with 2 bits per character, it is possible to replace eight consecutive bits in the binary text with its corresponding ASCII character. These eight consecutive bits are called a *block*. The decimal value of a block is the *code* of the block given by the following function g.

g: $\Sigma \times \Sigma \times \Sigma \times \Sigma \rightarrow [0 .. 255]$,

$g(\alpha\beta\gamma\delta) = (f(\alpha) \times 4^3) + (f(\beta) \times 4^2) + (f(\gamma) \times 4^1) + (f(\delta) \times 4^0)$

A DNA text-block will be represented by 32-bits in the original DNA text, as each character needs 8-bits. Using the function g we can represent a text-block with 8-bits. As the function g is a bijective function, we can efficiently store any text block into 8-bits and it is possible to reconstruct the original DNA text exactly.

This efficient storage method will guarantee to efficiently store the DNA text in

25% of the space required for the original text.

7.3 Comparison with existing compression algorithms

In this section we compare the well known text compression methods, Lempel-Ziv encoding [134] and Huffman encoding [73] with our efficient storage method. Note that our method requires 2 bits per DNA character.

In Lempel-Ziv (LZ) encoding [140, 141] the file may be compressed to less than 2 bits per character but requires re-occurring strings of length at least 6. This is because each of the strings that are output to the compressed file are of 12 bits in length. Also the strings length less than 6 that have been written to the compressed file have to be offset. The experiments in [91] show that LZ encoding compresses a DNA text to 2.14 bits per character for their chosen text, which is worse than our method.

The LZ encoding and its derivative LZW encoding [134] are used in UNIX utilities, compress and gzip. We selected DNA texts of different sizes from a database in [57] and compressed the texts using these utilities. Table 7.1 shows that our compression method is comparable to these methods. The third and fourth columns show the size of the compressed file as a percentage of its original size.

DNA text	Original size (bytes)	gzip %	compress %
Text 1	100000	28.523	27.264
Text 2	100000	28.758	27.347
Text 3	100000	28.886	27.348
Text 4	100000	28.659	27.283
Text 5	172000	29.167	27.335
Text 6	217000	28.992	27.143
Text 7	253505	29.098	27.193
Text 8	287000	29.065	27.217
Text 9	319000	29.141	27.183
Text 10	995000	28.994	26.911

Table 7.1: *The size of the compressed generated when using compress and gzip*

The Huffman encoding determines the length of the bit representation of the characters according to their frequency. The Huffman encoding compressed the texts used in Table 7.1 to 25% of their original size. Although Huffman encoding gives a figure that is the same as ours, the Huffman encoding requires the dictionary from the encoding for the decoding process. Our method is also simpler than the Huffman encoding, as our method does not require any pre-computation to compress a DNA text.

7.4 Searching in the efficiently stored file - the DS algorithm

In this section we describe an algorithm to find all exact occurrences of a pattern in an efficiently stored DNA text. The original DNA text contains four DNA characters,

namely A, C, G and T. The pattern may be composed of more than these four characters. These extra characters are wildcards, which can represent two or more DNA characters. For example [4],

B = C, G or T	D = A, G or T
H = A, C or T	K = G or T
M = A or C	N = A, C, G or T
R = A or G	S = C or G
V = A, C or G	W = A or T
Y = C or T	

A substring of the pattern may overlap between consecutive text-blocks and a pattern may start in a text-block at any one of four positions. For example, efficient storage of a DNA text ACCGGTAGAGGC will divide the text into blocks, ACCG, GTAG and AGGC. The pattern CGGTAGA occurs in the consecutive blocks as shown in bold fonts and the pattern starts at the third position in the first text-block.

During the search we need to look whether a substring of the pattern matches a text-block, and whether a prefix (or suffix) of a pattern is a suffix (or prefix) of a text-block. Due to this problem we have to look for four different expressions in the efficiently stored text. For example, the pattern CGGTAGA will have the following expressions, where N can be any DNA character, A, C, G or T.

Expression 0:	NNNC (0)	GGTA (4)	GANN (8)
Expression 1:	NNCG (1)	GTAG (5)	ANNN (9)
Expression 2:	NCGG (2)	TAGA (6)	
Expression 3:	CGGT (3)	AGAN (7)	

Figure 7.1: *The expressions for the pattern* **CGGTAGA**

Each expression is made up of pattern-blocks of length four. There will be m+3 pattern-blocks (see Figure 7.2), where m is the length of a pattern. We number the pattern blocks as above (shown in brackets) starting from 0 at the top left to 9 in the bottom right.

For a pattern $P_1P_2..\ P_m$ we can construct the expressions as follows, where m mod 4 = 0. The pattern-block numbers are shown in brackets.

Expression 0:	$NNNP_1$	(0)	.....	$P_{m-6}P_{m-5}P_{m-4}P_{m-3}$	$(m-4)$	$P_{m-2}P_{m-1}P_mN$	(m)
Expression 1:	NNP_1P_2	(1)	.....	$P_{m-5}P_{m-4}P_{m-3}P_{m-2}$	$(m-3)$	$P_{m-1}P_mNN$	$(m+1)$
Expression 2:	$NP_1P_2P_3$	(2)	.....	$P_{m-4}P_{m-3}P_{m-2}P_{m-1}$	$(m-2)$	P_mNNN	$(m+2)$
Expression 3:	$P_1P_2P_3P_4$	(3)	.....	$P_{m-3}P_{m-2}P_{m-1}P_m$	$(m-1)$		

Figure 7.2: *The expressions generated for a general pattern*

The naive algorithm will compare a text-block with the first pattern-blocks in each expression. If any of these pattern-blocks matched with the text-block, we need to compare the consecutive text-blocks with the rest of the pattern-blocks in the expression. If a pattern-block contains a wildcard, we need to compare a text-block with all the possible pattern-blocks by considering the DNA characters represented by the wildcard.

The DNA Search (DS) algorithm first constructs a table called the Block Table. The Block Table has 256 columns and m+3 rows as there are 256 possible blocks in a DNA text and m+3 is the number of pattern-blocks. The table is initialised to 0. The $(i,j)^{th}$ entry in the table is defined as follows, where i, $0 \leq i \leq m+2$, is the pattern-block number and j, $0 \leq j \leq 255$, is the code for a block of DNA characters. Block Table(i,j) = 1 if j matches the value for pattern-block i, otherwise Block Table(i,j) = 0. Suppose that the pattern-block does not have a wildcard character, the $(i,j)^{th}$ entry is 1, if the code for pattern-block i is equal to j. If there is one or more wild cards in the pattern-block, we consider all the possible blocks by considering the DNA characters represented by the wildcard. For example, if the i^{th} pattern-block is NAWT, the $(i,j)^{th}$ entry is equal to 1 for all j, where j is the code for AAAT, AATT, CAAT, CATT, GAAT, GATT, TAAT or TATT.

For each expression we only have to compare one pattern-block with a text block, and if these two match then we compare the rest of the pattern-blocks in the expression with the corresponding text-blocks. We choose a pattern block (from each expression) which has the minimum number of possibilities of matching with a text-block. For each pattern-block the number of possibilities of matching a text-block can be found by adding the values in the row of the pattern-block in the Block Table. For example, the pattern ACAC will have the following expressions.

Expression 0:	NNNA (0) - 64	CACN (4) - 4
Expression 1:	NNAC (1) - 16	ACNN (5) - 16
Expression 2:	NACA (2) - 4	CNNN (6) - 64
Expression 3:	ACAC (3) - 1	

Figure 7.3: *The expressions generated for the pattern* **ACAC**

The pattern-block numbers and the number of possibilities are stated with the pattern-block. The pattern-block numbers are in brackets. The pattern-blocks will be chosen as follows,

Expression 0:	CACN (4) - 4
Expression 1:	NNAC (1) - 16
Expression 2:	NACA (2) - 4
Expression 3:	ACAC (3) - 1

Figure 7.4: *The pattern blocks that would be chosen for the pattern* **ACAC**

From these we construct a Search Table of dimensions 4×256, and it is initialised to -1. In the first row of the Search Table, we enter the chosen pattern-block numbers at the j^{th} column, for all j, $0 \leq j \leq 255$, if j is the code for these pattern-blocks. A column number may be the code for more than one of the chosen pattern-blocks. In this situation we enter only one pattern block number in each row of that column. As there are only four expressions we need a maximum of four rows. In the above example, the chosen pattern-blocks from Expression 0 and 2, CACN and NACA, will both match the block CACA. We enter the pattern-blocks (CACN and NACA) numbers 4 and 2 in the first and second rows respectively of the column k, where k

is the code for CACA.

We begin the search at the beginning of the efficiently stored DNA text and compare the text-blocks with chosen pattern-blocks in the Search Table. We check the j^{th} column in the Search Table, where j is the code of the text block. If the entry is -1 then we check the next text-block. Otherwise we know that the text-block is in the pattern. We compare the rest of the pattern-blocks in the expression with the corresponding text-blocks until either full match or mismatch is found using the Block Table. Adding or subtracting 4 from the pattern-block numbers can easily identify the rest of the pattern-blocks of that expression. Before we move to the next text-block, we check if the entry in the next row of the Search Table is -1. We repeat this process if the entry is not -1, otherwise we check the next text-block.

7.5 The average running time of the DS algorithm

The pre-processing of the DS algorithm takes $O(m)$ time, as the Block table and the Search Table can be constructed in $O(m)$ time and $O(1)$ time respectively. The worst case for the search will take $O(mn)$ time. In this section we will show that the algorithm does on average $O(n)$ comparisons. From this we can say that the average running time of the algorithm is $O(n + m)$. We also justify this with experiments.

A block is a string of four characters. If the size of the alphabet set is 4, then we have only 256 different blocks. If we assume that each of the 256 blocks occurs in the

text with equal frequency, then we have Lemma 7.1.

Let $\Gamma_{PB}(j)$ be the probability of a pattern-block j matches a text-block.

Lemma 7.1: If a character in pattern-block i is either A, C, G, T or N (the wildcard character N represents A, C, G and T), then $\Gamma_{PB}(j) = \frac{1}{4^{4-w}}$, where w is the number of wildcard character N in the pattern-block.

Recall that when we compare a text-block with a pattern-block, we choose a pattern-block (from each expression) which has the minimum number of possibilities of matching with a text-block (i.e. the pattern-block with minimum number of wildcard character N). For example, consider the expressions for the pattern ACGTAT (shown below with pattern-block numbers in brackets).

Expression 0:	NNNA (0)	CGTA (4)	TNNN (8)
Expression 1:	NNAC (1)	GTAT (5)	
Expression 2:	NACG (2)	TATN (6)	
Expression 3:	ACGT (3)	ATNN (7)	

Figure 7.5: *The expressions generated for the pattern* **ACGTAT**

The following shows the value of w in the above pattern-blocks.

Expression 0:	3 (0)	0 (4)	3 (8)
Expression 1:	2 (1)	0 (5)	
Expression 2:	1 (2)	1 (6)	
Expression 3:	0 (3)	2 (7)	

Figure 7.6: *The number of wildcards in pattern-blocks, for* $m = 6$

The pattern-blocks 4, 5, 2 and 3 have minimum values of w in expressions 0, 1, 2 and 3 respectively. We would choose these pattern-blocks for the first comparison. If any of these pattern-blocks matches with the text-block, then we choose the pattern-block with the minimum number of wild cards among the remaining pattern-blocks in the expression. In an attempt, for each expression we repeat this step until either a full match or mismatch is found.

Suppose $m = 16$, the following shows the values of w in a pattern-block for each expression (pattern-block numbers are in brackets).

Expression 0:	3 (0)	0 (4)	0 (8)	0 (12)	1 (16)
Expression 1:	2 (1)	0 (5)	0 (9)	0 (13)	2 (17)
Expression 2:	1 (2)	0 (6)	0 (10)	0 (14)	3 (18)
Expression 3:	0 (3)	0 (7)	0 (11)	0 (15)	

Figure 7.7: *The number of wildcards in pattern-blocks, for* $m = 16$

There are three columns having all zeros. In general, for all m, if $m \bmod 4 \neq 3$, there are $\lambda = \lfloor \frac{m-3}{4} \rfloor$ number of columns will have all zeros. If $m \bmod 4 = 3$, we will have $\lambda - 1$ columns with all zeros, and the last one with three zeros in a column and the fourth zero in another column. For example, if $m = 15$ (i.e. $m \bmod 4 = 3$) we will have 2 (i.e. $\lambda - 1$) columns with all zeros, and the last one with three zeros in a column and the fourth zero in another column (shown in bold font):

Expression 0:	3 (0)	0 (4)	0 (8)	**0** (12)	2 (16)
Expression 1:	2 (1)	0 (5)	0 (9)	**0** (13)	3 (17)
Expression 2:	1 (2)	0 (6)	0 (10)	**0** (14)	
Expression 3:	**0** (3)	0 (7)	0 (11)	1 (15)	

Figure 7.8: *The number of wildcards in pattern-blocks, for* $m = 15$

From this observation we have Lemma 7.2.

Let Φ_i be the probability of i number of pattern-blocks matching with the text-blocks in an expression at an attempt. In other words Φ_i is the probability of the algorithm making *at least* $i + 1$ comparisons at an attempt.

Lemma 7.2: For all m and for all i, $1 \leq i \leq \lambda$, $\Phi_i = 4 \times \frac{1}{256^i}$, where $\lambda = \lfloor \frac{m-3}{4} \rfloor$.

Proof: For all m, each expression has λ number of pattern-blocks with $w = 0$. At an attempt, we can choose pattern-blocks with $w = 0$ from each of the four expressions for the first λ comparisons. From Lemma 7.1 we have $\Gamma_{PB}(j) = 1/256$ if $w = 0$. In a comparison we compare a text-block with the four pattern-blocks (one from each expression). Probability of any of these pattern-blocks (i.e. with $w = 0$) matching in a comparison is $4/256$ which is Φ_1. In an attempt we will have the i^{th} comparison only if i number of pattern-blocks matches the corresponding text-blocks. The probability of i matches for an expression is $\frac{1}{256^i}$ and there are four expressions and so Φ_i is $\frac{4}{256^i}$, for all i. □

In an attempt, for all $m \geq 8$, after λ comparisons the pattern-blocks which have

not yet been compared will be similar to the expressions for patterns of length m', $4 \leq m' \leq 7$, where $m' = (m \bmod 4) + 4$. In other words, if we remove all the λ columns with all zeros from the expressions of pattern length $m \geq 8$, the number of wildcards in pattern-blocks will be the same as in the expressions of pattern length m'. For example, if we remove 3 (i.e. λ) columns of all zeros from the number of wildcards in pattern-blocks, for $m = 16$ (see above), we will get the number of wildcards in pattern-blocks, for $m' = 4$ as below.

Expression 0:	3 (0)	1 (16)
Expression 1:	2 (1)	2 (17)
Expression 2:	1 (2)	3 (18)
Expression 3:	0 (3)	

Figure 7.9: *The number of wildcards in pattern-blocks, for $m' = 4$*

Lemma 7.3: For $2 \leq m \leq 7$, Φ_1 and Φ_2 are as follows:

pattern length	Φ_1	Φ_2
2	1/4	
3	1/8	
4	3/32	
5	5/128	
6	7/256	1/1024
7	1/64	1/2048

Figure 7.10: *The probability of the algorithm making at least 1 or 2 comparisons at an attempt.*

Proof: In an attempt, for $2 \leq m \leq 5$ and $6 \leq m \leq 7$ we have at most 2 and 3

comparisons respectively. Hence we only need to know the values of Φ_1 for $2 \leq m \leq 5$, and Φ_1 and Φ_2 for $6 \leq m \leq 7$. □

We show how the Φ_1 and Φ_2 are calculated with an example for m = 6. First we will select the pattern-blocks 4, 5, 2, and 3 (see above for the expressions for the number of wildcards in pattern-blocks, for $m = 6$).

$$\begin{aligned}
\Phi_1 &= \Gamma_{PB}(4) + \Gamma_{PB}(5) + \Gamma_{PB}(2) + \Gamma_{PB}(3) \\
&= \frac{1}{4^{4-0}} + \frac{1}{4^{4-0}} + \frac{1}{4^{4-1}} + \frac{1}{4^{4-0}} \text{ (Lemma 7.1)} \\
&= 1/256 + 1/256 + 1/64 + 1/256 \\
&= 7/256
\end{aligned}$$

For Φ_2 we only need to consider the first expression. We can have at least 3 comparisons, iff pattern-blocks 4 and (assume we select) 0 match with the corresponding text-blocks.

$$\begin{aligned}
\Phi_2 &= \Gamma_{PB}(4) \times \Gamma_{PB}(0) \\
&= \frac{1}{4^{4-0}} \times \frac{1}{4^{4-3}} \text{ (Lemma 7.1)} \\
&= 1/256 \times 1/4 \\
&= 1/1024
\end{aligned}$$

Note that in any attempt for all m, we can have at most $\lambda + 2$ matches if m mod $4 = 2$ and $m \geq 3$, otherwise $\lambda + 1$ matches. For $m \geq 8$, to calculate $\Phi_{\lambda+1}$ and $\Phi_{\lambda+2}$, we only need to know the values of Φ_1 and Φ_2 for m, $4 \leq m \leq 7$. From these values we can have the following Lemma.

Lemma 7.4: For $m \geq 4$,

$$\Phi_{\lambda+1} = (1/256)^{\lambda} \times \alpha_b \text{ and}$$

$$\Phi_{\lambda+2} = (1/256)^{\lambda} \times \beta_b,$$

where α_b and β_b are the values of b^{th} base case in the first and second columns in the table below respectively and $b = m \bmod 4$.

base case	α	β
0	3/32	
1	5/128	
2	7/256	1/1024
3	1/2048	

Figure 7.11: *The number of wildcards in pattern blocks, for* $m' = 4$

Let Ψ_i be the probability of making *exactly* i comparisons at an attempt. Using Φ_i we can have an equation for Ψ_i:

$$\Phi_i = \Psi_{i+1} + \Psi_{i+2} + \cdots$$

From this equation we have

$$\Psi_i = \Phi_{i-1} - \Phi_i$$

We know that we will make at least one comparison in every attempt. So Φ_0 is 1.

For all m, the maximum number of comparisons in any attempt is $\mu = \lceil \frac{m+3}{4} \rceil$, which is equal to $\lambda + 3$ if $m \bmod 4 = 2$ and $m \neq 2$, otherwise $\lambda + 2$. So Φ_i is 0 for all $i \geq \mu$. This gives:

$$\Psi_1 = 1 - \Phi_1$$

$$\Psi_i = \Phi_{i-1} - \Phi_i, 2 \leq i \leq \mu - 1$$

$$\Psi_\mu = \Phi_{\mu-1}$$

Lemma 7.5: The total number of comparisons, Ψ_{Total}, is less than or equal to $2n'$ on average, where n' is the number of text-blocks in the efficiently stored file.

Proof:

$$\begin{aligned}
\Psi_{Total} &= n' \times \sum_{i=1}^{\mu} i \times \Psi_i \\
&= n' \times ((1 - \Phi_1) + 2(\Phi_1 - \Phi_2) + 3(\Phi_2 - \Phi_3) + \cdots + \mu - 1(\Phi_{\mu-2} - \Phi_{\mu-1}) + \mu\Phi_{\mu-1}) \\
&= n' \times (1 + \Phi_1 + \Phi_2 + \cdots + \Phi_{\mu-1}) \\
&= n' \times (1 + \sum_{i=1}^{\lambda} \frac{4}{256^i} + \Phi_{\lambda+1} + \Phi_{\lambda+2}) \text{ (Lemma 7.2)} \\
&\leq 2n'
\end{aligned}$$

From Lemmas 7.2, 7.3 and 7.4 we can see that $\sum_{i=1}^{\lambda} \frac{4}{256^i} + \Phi_{\lambda+1} + \Phi_{\lambda+2} \leq 1$ □

From these Lemmas we have the following Theorem.

Theorem: The average running time of the DS algorithm is $O(n + m)$.

To show this is also true in practice we counted the number of comparisons by running the DS algorithm when searching for the patterns without wildcards. Table 7.2 shows the estimated number of comparisons (Ψ_{Total}) and the actual number of comparisons. We used the same texts as in Table 7.1. The patterns are the cutting locations or boundaries for enzymes and are taken from [4]. There are 104 cutting

locations or boundaries for enzymes given in [4], 62 of them don't contain a wildcard and 42 of them contain at least one wildcard and are shown in Figures 7.12 and 7.13 respectively.

AAGCTT	AATATT	ACGCGT	ACTAGT
AGATCT	AGCGCT	AGCT	AGGCCT
AGTACT	ATCGAT	ATGCAT	ATTAAT
CACGTG	CAGCTG	CATATG	CATG
CCATGG	CCCGGG	CCGG	CCGCGG
CCTAGG	CCTGCAGG	CGATCG	CGCG
CGGCCG	CGTACG	CTCGAG	CTGCAG
CTTAGG	GAATTC	GACGTC	GAGCTC
GATATC	GATC	GCATGC	GCCGGC
GCGCGC	GCGC	GCGGCCGC	GCTAGC
GGATCC	GGCC	GGCCGGCC	GGCGCC
GGGCCC	GGTACC	GTAC	GTCGAC
GTGCAC	GTTAAC	TACGTA	TCATGA
TCCGAA	TCCGGA	TCGA	TCGCGA
TCTAGA	TGATCA	TGCGCA	TGGCCA
TTCGAA	TTTAAA		

Figure 7.12: The 62 patterns that don't contain a wildcard

CACNNNGTG	CAGNNNCTG	CCANNNNNTGG
CCANNNNNNTGG	CCANNNNNNNNNTGG	CCTCNNNNNNN
CCTNNNNNAGG	CCTNAGG	CCSGG
CCWGG	CCWWGG	CGGWCCG
CMGCKG	CTNAG	CYCGRG
GAATGCN	GAAGANNNNNNN	GACGCNNNNN
GACNNNGTC	GACNNNNGTC	GANTC
GCCNNNNNGGC	GCTNAGC	GDGCHC
GGATGNNNNNNNNN	GGCCNNNNNGGCC	GGNCC
GGTGANNNNNNNN	GGTNACC	GGWCC
GGYRCC	GRGCYC	GTMKAC
GTYRAC	RCATGY	RCCGGY
RGATCY	RGCGCY	RGGNCCY
RGGWCCY	YGGCCR	

Figure 7.13: *The 42 patterns that contain one or more wildcards*

We use the 62 patterns that don't contain a wildcard shown in Figure 7.12 to calculate the number of comparisons taken by the DS algorithm. For each pattern length, the actual number of comparisons in the table is the total number of comparisons divided by the number of patterns of that length. There are 9 patterns of length 4, 50 of length 6 and 3 of length 8. The text length is the length of the efficiently

stored file.

		pattern lengths of 4		pattern lengths of 6		pattern lengths of 8	
text no.	n'	Ψ_{Total}	Actual	Ψ_{Total}	Actual	Ψ_{Total}	Actual
1	25000	27344	26874	25706	25588	25398	25289
2	25000	27344	26952	25706	25592	25398	25312
3	25000	27344	26857	25706	25592	25398	25274
4	25000	27344	26829	25706	25581	25398	25256
5	43000	47031	46269	44218	44064	43688	43349
6	54250	59326	58446	55786	55600	55118	54696
7	63377	69319	68193	65172	64946	64390	63936
8	71750	78477	77444	73782	73549	72897	72425
9	79750	87227	85926	82003	81720	81025	80411
10	248750	272070	268106	255774	254942	252728	250885

Table 7.2: *The number of comparisons performed by the DS algorithm for each of the 10 efficiently stored DNA texts*

From Table 7.2 we can see that the number of comparisons performed by the DS algorithm is slightly more than the efficiently stored text size. Note that the algorithm takes fewer comparisons than the estimate suggests.

7.6 Comparison with existing string matching algorithms

In this section we compare the existing string matching algorithms with our DS algorithm. All searches with the DS algorithm were conducted on efficiently stored DNA texts. The texts used for these experiments are the same texts used in Table 7.1 and the patterns are taken from Figures 7.12 and 7.13. For each of the 10 texts we

measure the total (user) time (including pre-computation time) in seconds to search for all 104 patterns.

For each text we give the total time taken in seconds and the time taken for each algorithm divided by the time taken by the DS algorithm (Ratio). We used an Intel 486-DX2-66 processor based machine with 8 megabytes of RAM and a 100 megabyte hard drive running S.u.S.E. Linux 5.2 to conduct the experiments. All the algorithms were coded in C.

Tables 7.3 and 7.4 show the time taken to search for the 62 patterns that don't contain a wildcard (Figure 7.12). The existing algorithms searched in the original DNA text and the DS algorithms searched in the efficiently stored DNA text.

	text1		text2		text3		text4		text5	
	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio
BR	19.09	2.34	19.14	2.37	19.11	2.41	20.00	2.49	32.56	2.65
BM	30.48	3.74	30.48	3.78	34.44	4.35	30.51	3.80	52.38	4.26
HOR	25.51	3.13	26.51	3.29	25.46	3.21	25.59	3.19	43.37	3.53
QS	26.45	3.24	26.38	3.27	26.45	3.34	26.52	3.31	45.13	3.67
RAI	24.32	2.98	24.28	3.01	24.43	3.08	24.31	3.03	41.42	3.37
DS	8.15	1.00	8.06	1.00	7.92	1.00	8.02	1.00	12.28	1.00
SMI	26.15	3.21	26.15	3.24	26.15	3.30	26.19	3.26	44.76	3.64

Table 7.3: *Time in seconds to search for all the patterns without wildcards in the given texts*

	text6		text7		text8		text9		text10	
	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio
BR	41.14	2.75	48.05	2.77	54.39	2.84	60.30	2.93	191.7	3.17
BM	65.66	4.39	76.70	4.43	86.96	4.54	98.88	4.80	301.1	4.98
HOR	55.41	3.70	64.10	3.70	72.97	3.81	80.35	3.90	254.1	4.20
QS	56.98	3.81	66.40	3.83	76.39	3.99	83.49	4.05	264.7	4.38
RAI	52.23	3.49	60.93	3.52	69.22	3.62	76.65	3.72	243.7	4.03
DS	14.96	1.00	17.33	1.00	19.14	1.00	20.60	1.00	60.5	1.00
SMI	56.42	3.77	66.69	3.85	74.67	3.90	82.86	4.02	262.0	4.33

Table 7.4: *Time in seconds to search for all the patterns without wildcards in the given texts*

	text1		text2		text3		text4		text5	
	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio
BR	319	39.13	319	39.58	320	40.40	317	39.52	804	65.46
BM	131	16.07	131	16.26	131	16.54	131	16.33	354	28.82
HOR	233	28.58	226	28.04	228	28.79	225	28.05	566	46.08
QS	286	35.08	277	34.37	312	39.39	280	34.91	694	56.51
RAI	222	27.23	236	29.28	222	28.03	238	29.67	558	45.43
DS	8	1.00	8	1.00	8	1.00	8	1.00	12	1.00
SMI	354	43.42	355	44.05	354	44.70	353	44.01	898	73.12

Table 7.5: *Time in seconds to search for all the patterns with wildcards using bit masking in the given texts*

	text6		text7		text8		text9		text10	
	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio
BR	911	60.90	1008	58.18	544	28.43	698	33.89	3155	52.18
BM	376	25.13	417	24.07	233	12.18	294	14.27	1354	22.40
HOR	673	44.99	716	41.32	386	20.17	488	23.69	2241	37.07
QS	791	52.87	990	57.14	471	24.61	595	28.89	2730	45.16
RAI	636	42.51	704	40.63	380	19.86	478	23.21	2207	36.50
DS	15	1.00	17	1.00	19	1.00	21	1.00	61	1.00
SMI	1017	67.98	1131	65.27	612	31.98	767	37.24	3522	58.26

Table 7.6: *Time in seconds to search for all the patterns with wildcards using bit masking in the given texts*

The algorithms were modified so that they would search in the efficiently stored text using bit masking. By bit masking we mean that we read in an 8 bit ASCII character and masked out the bits that we didn't want. We did this by manipulating the ASCII character so as to reveal the character that we were comparing to the pattern. We used the same method when calculating the value of a shift.

As can be seen from the results in Tables 7.5 and 7.6, we can see that searching in the efficiently stored file with the existing algorithms for a pattern is not as efficient as searching in the original DNA text. This is due to the cost of having to use bit masking to return the character that we are interested in comparing.

Due to the massive difference in time between searching in the efficiently stored DNA text and the original DNA text we will compare in the original DNA text for the tests conducted using patterns that contain wildcards.

The existing string matching algorithms (except the BM algorithm) that are considered in this section could be adapted to search for patterns with wildcards.

We assign prime numbers to the DNA characters, A = 2, C = 3, G = 5 and T = 7. The wildcard characters can be expressed as a product of its possible prime numbers. For example, the value for K is 35 (5 × 7) as K can be G or T. Using prime numbers ensures that any value given to a wildcard character is unique. We compare a text character and a pattern character by dividing the value of the pattern character by the value of the text character. If the remainder is zero then these two characters match.

After a mismatch we need to consider the wildcard characters when calculating the position of the rightmost occurrence of the mismatched text character. For example, in the pattern ACTWG, where the wildcard character W is A or T, the rightmost occurrences of A and T are at W.

From Tables 7.3 to 7.8 we can see that our DS algorithm is faster than the existing exact string matching algorithms for the chosen data. As the size of the original file increases, the Ratio increases. This is because the factor of four difference between the original and efficiently stored DNA texts becomes more significant.

	text1		text2		text3		text4		text5	
	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio
BR	27.44	4.09	27.23	4.12	28.61	4.30	29.00	4.31	48.42	4.78
HOR	30.92	4.61	30.37	4.59	32.66	4.91	32.82	4.88	54.90	5.43
QS	22.79	3.40	21.41	3.24	26.19	3.93	26.21	3.90	44.28	4.38
RAI	30.46	4.54	30.69	4.64	28.37	4.26	28.42	4.23	49.08	4.85
DS	6.71	1.00	6.61	1.00	6.66	1.00	6.72	1.00	10.12	1.00
SMI	23.10	3.44	22.24	3.36	25.41	3.82	25.54	3.80	44.18	4.37

Table 7.7: *Time in seconds to search for all the patterns with wildcards in the given texts*

	text6		text7		text8		text9		text10	
	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio
BR	58.55	4.87	68.69	4.98	81.59	5.25	86.32	5.26	279.7	5.67
HOR	65.92	5.49	73.75	5.34	92.10	5.93	97.04	5.92	317.0	6.43
QS	48.68	4.05	60.22	4.36	74.27	4.78	71.51	4.36	255.8	5.18
RAI	64.52	5.37	74.27	5.38	81.84	5.27	95.56	5.83	283.3	5.74
DS	12.02	1.00	13.80	1.00	15.53	1.00	16.40	1.00	49.3	1.00
SMI	50.53	4.21	56.35	4.08	72.45	4.66	78.25	4.77	248.8	5.04

Table 7.8: *Time in seconds to search for all the patterns with wildcards in the given texts*

7.7 Conclusions

Using the DS algorithm one can keep texts (with an alphabet of four characters) efficiently stored indefinitely and perform the search for a pattern. These methods will save both time and space. The experimental results show that our algorithm is more efficient than the existing algorithms for the chosen data sets.

Even though the DS algorithm takes $O(nm)$ time for the worst case, we prove that the average time taken by the algorithm is $O(n + m)$. We also justified our average running time by experiments.

Chapter 8

A linear time string matching algorithm on average with efficient text storage

8.1 Introduction

In this Chapter we extend our efficient storage method from Chapter 7 to include the storage of texts with alphabets of size, $\sigma \leq 128$. The storage method will efficiently store the text in $\frac{\lceil \log_2 \sigma \rceil}{8}$ of the space required for the original text. We describe a new string matching algorithm to search for a pattern in the efficiently stored text. We prove that on average this string matching algorithm takes $O(n + m)$ time. We compare our new string matching algorithm with other well known existing string matching algorithms by experimentation.

8.2 Efficient storage of a text

We assume that the size of the alphabet set, σ, is in the range $1 \leq \sigma \leq 128$ and that we are representing each character in the alphabet with one byte. There are redundant bits in each byte as we only need $\lceil \log_2 \sigma \rceil$ bits to represent a character.

After we replace the characters in a text with $\lceil \log_2 \sigma \rceil$ bits, it is possible to replace eight consecutive bits in the binary text with its corresponding ASCII character. These eight consecutive bits are called a *block*. The decimal value of a block is the *code* of the block. This representation will reduce the storage space to $\frac{\lceil \log_2 \sigma \rceil}{8} n$, where n is the size of the original text.

For example, consider the text $T = CACDABEB$ with the alphabet set $\Sigma = \{A, B, C, D, E\}$. This text T of eight characters can be represented with three characters $T' = A0a$. First we represent the characters with $A = 000, B = 001, C = 010, D = 011$ and $E = 100$. This will give the binary representation of the text T:

010000010011000001100001

The first bit in each block are shown in bold font. The codes for the text blocks are 65, 48 and 97 and their corresponding ASCII characters are 'A', '0', and 'a' respectively.

8.3 Comparison with existing compression algorithms

The method described in section 8.2 is not compression as in the literature but does reduce the size of the original text. In this section we compare the well known text compression methods, Huffman encoding [73] and Lempel-Ziv encoding [102, 134, 141] with our method.

The Huffman encoding determines the length of the bit representation of the characters according to their frequency. It assigns smaller codes to high frequency characters and larger codes to low frequency characters.

In Lempel-Ziv (LZ) encoding [141] the file may be compressed to less than $\lceil \log_2 \sigma \rceil$ bits per character but requires re-occurring strings. Each of the repeated strings and each of the characters in the alphabet are represented by 12 bits. The gains from this method are reliant on there being enough repeated strings to counter the 12 bits which are used to represent each of the compressed strings.

The LZ encoding and its derivative LZW encoding [134] are used in UNIX utilities, *compress* and *gzip*. Another variation of LZ encoding (NR) is described in [102].

Table 8.1 shows that our efficient storage method is comparable to these methods. Although our method is not very good for text files with large alphabets, the method is competitive for DNA, RDNA and hexadecimal files. Note that the main purpose of this method is not compression, but for the searching of a pattern in a efficiently stored file.

σ	Our method	Huffman	Compress	Gzip	NR
2	62500	62500	71579	79644	121110
3	125000	104107	110629	118776	178706
4	125000	125000	136945	146402	215764
5	187500	149935	161641	168813	244192
8	187500	187500	209053	211543	297634
9	250000	201313	223571	226617	310964
16	250000	250000	288546	285834	373658
17	312500	257293	294476	290854	377491
32	312500	312500	367527	330150	449265
33	375000	316232	370975	332592	451570
64	375000	375000	461069	378224	493981

Table 8.1: *Compressed text sizes for a random text of 500,000 bytes*

8.4 Searching in a text with efficient storage

In this section we describe an algorithm to find all exact occurrences of a pattern in a text. Here we assume that the text is stored as described in Section 8.2 and $\sigma \leq 128$. We describe the algorithm for $\sigma = 2$, we will see later that the algorithm can be easily adapted for $\sigma > 2$.

A substring of the pattern may overlap between consecutive text-blocks and a pattern may start in a text-block at any one of eight positions. During the search we need to look whether a prefix (or suffix) of a pattern is a suffix (or prefix) of a text-block. Due to this problem we have to consider eight different expressions. Each

expression is made up of pattern-blocks of length eight bits. There will be $m + 7$ pattern-blocks in total (see Figure 8.1), where m is the length of a pattern.

For a pattern $P_1P_2..\ P_m$ we can construct the expressions as shown in Figure 8.1. Here we consider the case for m mod $8 = 0$. We number the pattern-blocks starting from 0 at the top left corner to $m+6$ in the bottom right corner as shown in brackets. The wildcard character N represents either 0 or 1, and $P_{i..j}$ represents $P_i..P_{j-1}P_j$, for $1 \leq i < j \leq m$.

Exp0:	NNNNNNNP_1	(0)	.....	$P_{m-14..m-7}$	(m-8)	$P_{m-6..m}$N	(m)
Exp1:	NNNNNN$P_{1..2}$	(1)	.....	$P_{m-13..m-6}$	(m-7)	$P_{m-5..m}$NN	$(m+1)$
Exp2:	NNNNN$P_{1..3}$	(2)	.....	$P_{m-12..m-5}$	(m-6)	$P_{m-4..m}$NNN	$(m+2)$
Exp3:	NNNN$P_{1..4}$	(3)	.....	$P_{m-11..m-4}$	(m-5)	$P_{m-3..m}$NNNN	$(m+3)$
Exp4:	NNN$P_{1..5}$	(4)	.....	$P_{m-10..m-3}$	(m-4)	$P_{m-2..m}$NNNNN	$(m+4)$
Exp5:	NN$P_{1..6}$	(5)	.....	$P_{m-9..m-2}$	(m-3)	$P_{m-1..m}$NNNNNN	$(m+5)$
Exp6:	N$P_{1..7}$	(6)	.....	$P_{m-8..m-1}$	(m-2)	P_mNNNNNNN	$(m+6)$
Exp7:	$P_{1..8}$	(7)	.....	$P_{m-7..m}$	(m-1)		

Figure 8.1: *Expressions for a pattern $P_1P_2..\ P_m$ when m mod $8 = 0$*

The naive algorithm will compare a text-block with the first pattern-blocks in each expression. If any of these pattern-blocks matched with the text-block, we need to compare the consecutive text-blocks with the rest of the pattern-blocks in the expression.

Our algorithm first constructs a table called the *Block-Table*. The Block-Table has 256 columns and $m + 7$ rows as there are 256 possible blocks in a text and $m+7$

is the number of pattern-blocks we need to consider. The table is initialised to 0. The $(i,j)^{th}$ entry in the table is defined as follows, where i, $0 \leq i \leq m+6$, is the pattern-block number and j, $0 \leq j \leq 255$, is the code for a block. Suppose that the pattern-block does not have a wildcard character, the $(i,j)^{th}$ entry is 1, if the code for pattern-block i is equal to j. If there is one or more wild cards in the pattern-block, we consider all the possible blocks. For example, if the i^{th} pattern-block is NN111000, the $(i,j)^{th}$ entry is equal to 1 for all j, where j is the code for 00111000, 01111000, 10111000 or 11111000.

For each expression we only have to compare one pattern-block with a text-block, and if these two match then we compare the rest of the pattern-blocks in the expression with the corresponding text-blocks. We choose a pattern-block (from each expression) which has the minimum number of possibilities of matching with a text-block. We build the *Order-Table* of dimensions 8 by $\lceil \frac{m+7}{8} \rceil$ which contains the order in which to examine the pattern-blocks for each expression. For each pattern-block the number of possibilities of matching a text-block can be found by adding the values in the row of the pattern-block in the Block-Table.

From these we construct a *Search-Table* of dimensions 8×256, and it is initialised to -1. In the first row of the Search-Table, we enter pattern-block numbers from the first column of the Order-Table. If j is the code for these pattern-blocks, we enter the pattern-block numbers at the j^{th} column, for all j, $0 \leq j \leq 255$. A column number

may be the code for more than one of the chosen pattern-blocks. This is because a text-block can match pattern-blocks from more than one expression. As there are only eight expressions we need a maximum of eight rows. For example, the chosen pattern-blocks, 110011NN and NN001100, will both match the block 11001100. We enter the pattern-blocks (110011NN and NN001100) numbers in the first and second rows respectively of the column k, where k is the code for 11001100.

We begin the search at the beginning of the text and compare the text-blocks with chosen pattern-blocks in the Search-Table. We check the j^{th} column in the Search-Table, where j is the code of the text-block. If the entry is -1 then we check the next text-block. Otherwise we know that the text-block is in the pattern. We compare the rest of the pattern-blocks in the expression with the corresponding text-blocks until either full match or mismatch is found using the Block-Table and Order-Table. Before we move to the next text-block, we check if the entry in the next row of the Search-Table is -1. We repeat this process if the entry is not -1, otherwise we check the next text-block.

If $\sigma > 2$, we have to convert the pattern into a binary string by mapping the characters into $\lceil \log_2 \sigma \rceil$ bits as we did in Section 8.2. Here we don't have to consider all the expressions. This is because in the pattern-blocks 0, 1, .. , 7 (from expressions 0 to 7 respectively) the pattern starts at positions 7, 6, .. , 0 respectively (see Figure 8.1). The positions are numbered from left to right in a pattern-block.

σ	Bit Length	1	2	3	4	5	6	7
1-2	1	1 - 8						
3-4	2	1, 3, 5, 7						
5-8	3	1, 4, 7	0, 3, 6	2, 5				
9-16	4	3, 7						
17-32	5	2, 7	0, 5	3	1, 6	4		
33-64	6	1,7	3	5				
65-128	7	7, 0	1	2	3	4	5	6

Table 8.2: *The expressions considered at each comparison*

We can show that for all σ, in a comparison we need at most $\lceil \frac{8}{\lceil \log_2 \sigma \rceil} \rceil$ expressions. The number of expressions that are considered at a comparison are determined by the length of the pattern being searched for. If $\sigma = 8$ then we know that we need 3 bits to efficiently store each character of the alphabet. When performing the search, occurrences of the pattern are limited to beginning every 3 bits. In other words at positions 0, 3, 6, 9, 12, etc. in the text. So at the first comparison we need only to consider expressions 7, 4 and 1, at the second comparison, expressions 6, 3 and 0 and at the third comparison, expressions 5 and 2 (see Figure 8.1).

In Table 8.2, σ is the size of the alphabet being used and Bit Length is the number of bits used in the efficiently stored text to represent each of the characters of the alphabet. The numbers 1 to 7 are the first to seventh comparison of the pattern and text. The values given for each value of σ are the expressions considered for each comparison. From this table we can see that if $\sigma = 4$ then we consider expressions 1,

3, 5 and 7 at each comparison.

8.5 The average running time

The pre-processing of our algorithm takes $O(m)$ time, as the Block-Table, Order-Table and the Search-Table can be constructed in $O(m)$ time. The worst case for the search will take $O(mn)$ time. In this section we will show that the algorithm performs on average at most $2n$ comparisons. From this we can say that the average running time of the algorithm is $O(n + m)$. We also justify this with experiments at the end of this section.

At the end of section 8.3 we showed that we need to consider all eight expressions only when $\sigma = 2$. First we prove that the average number of comparisons for this worst case.

There are only 256 possible different blocks. If we assume that each of the 256 blocks occurs in a text with equal frequency, then we have the following lemma. Let $\Gamma_{PB}(j)$ be the probability of a pattern-block j matches a text-block.

Lemma 8.1: $\Gamma_{PB}(j) = \frac{1}{2^{8-w}}$, where w is the number of wildcard character N in the pattern-block.

Recall that when we compare a text-block with a pattern-block, we choose a pattern-block (from each expression) which has the minimum number of possibilities of matching with a text-block (i.e. the pattern-block with minimum number of

wildcard character N). If any of these pattern-blocks matches with the text-block, then we choose the pattern-block with the minimum number of wild cards among the remaining pattern-blocks in the expression. In an attempt, for each expression we repeat this step until either a full match or mismatch is found.

For example, consider the expressions for $m = 34$. Figure 8.2 shows the values of w in a pattern-block for each expression (pattern-block numbers are in brackets).

Exp0:	7 (0)	0 (8)	0 (16)	0 (24)	0 (32)	7 (40)
Exp1:	6 (1)	0 (9)	0 (17)	0 (25)	0 (33)	
Exp2:	5 (2)	0 (10)	0 (18)	0 (26)	1 (34)	
Exp3:	4 (3)	0 (11)	0 (19)	0 (27)	2 (35)	
Exp4:	3 (4)	0 (12)	0 (20)	0 (28)	3 (36)	
Exp5:	2 (5)	0 (13)	0 (21)	0 (29)	4 (37)	
Exp6:	1 (6)	0 (14)	0 (22)	0 (30)	5 (38)	
Exp7:	0 (7)	0 (15)	0 (23)	0 (31)	6 (39)	

Figure 8.2: *The number of wildcards in pattern-blocks for* $m = 34$

There are three columns with all zeros which are the first three columns in the Order-Table. In general, for all m, if $m \bmod 8 \neq 7$, there are $\lambda = \lfloor \frac{m-7}{8} \rfloor$ number of columns will have all zeros. If $m \bmod 8 = 7$, and $m \geq 15$ we will have $\lambda - 1$ columns with all zeros, and the remaining one with seven zeros in a column and the eighth zero in another column. For example, Figure 8.3 shows the number of wildcards in pattern-blocks for $m = 23$ (i.e. $m \bmod 8 = 7$.). We can see that there is one (i.e. $\lambda - 1$) column which is the second column with all zeros. The remaining column of all zeros is the fourth column with seven zeros and the eighth zero is in the first column

(shown in bold font).

Exp0:	7 (0)	0 (8)	0 (16)	2 (24)
Exp1:	6 (1)	0 (9)	0 (17)	3 (25)
Exp2:	5 (2)	0 (10)	0 (18)	4 (26)
Exp3:	4 (3)	0 (11)	0 (19)	5 (27)
Exp4:	3 (4)	0 (12)	0 (20)	6 (28)
Exp5:	2 (5)	0 (13)	0 (21)	7 (29)
Exp6:	1 (6)	0 (14)	0 (22)	
Exp7:	**0 (7)**	0 (15)	1 (23)	

Figure 8.3: *The number of wildcards in pattern-blocks for* $m = 23$

From this observation we have Lemma 8.2. Let Φ_i be the probability of i number of pattern-blocks matching with the text-blocks in an expression at an attempt. In other words Φ_i is the probability of the algorithm making *at least* $i + 1$ comparisons at an attempt.

Lemma 8.2: For all m and $\sigma = 2$, $1 \leq i \leq \lambda$, $\Phi_i = 8 \times \frac{1}{256^i}$, where $\lambda = \lfloor \frac{m-7}{8} \rfloor$.

Proof: For all m, each expression has λ number of pattern-blocks with $w = 0$. At an attempt, we can choose pattern-blocks with $w = 0$ from each of the eight expressions for the first λ comparisons. From Lemma 8.1 we have $\Gamma_{PB}(j) = 1/256$ if $w = 0$. In an attempt we will have the $i + 1^{th}$ comparison only if i number of pattern-blocks in an expression matches the corresponding text-blocks. The probability of i matches for an expression is $\frac{1}{256^i}$ and there are eight expressions and so Φ_i is $\frac{8}{256^i}$, $1 \leq i \leq \lambda$.

□

In an attempt, for $2 \leq m \leq 9$ and $10 \leq m \leq 14$ we have at most 2 and 3 comparisons respectively. Hence we only need to know the values of Φ_1 for $2 \leq m \leq 9$, and Φ_1 and Φ_2 for $10 \leq m \leq 14$. We can calculate these values easily. For example, the following shows the values of w in a pattern-block for each expression (pattern-block numbers are in brackets) for $m = 10$. First we will select the pattern-blocks 8 to 11 and 4 to 7.

Exp0:	7	(0)	0	(8)	7	(16)
Exp1:	6	(1)	0	(9)		
Exp2:	5	(2)	1	(10)		
Exp3:	4	(3)	2	(11)		
Exp4:	3	(4)	3	(12)		
Exp5:	2	(5)	4	(13)		
Exp6:	1	(6)	5	(14)		
Exp7:	0	(7)	6	(15)		

Figure 8.4: *The number of wildcards in pattern-blocks for* $m = 10$

$$\begin{aligned}
\Phi_1 &= \Gamma_{PB}(8) + \Gamma_{PB}(9) + \Gamma_{PB}(10) + \Gamma_{PB}(11) + \Gamma_{PB}(4) + \Gamma_{PB}(5) + \Gamma_{PB}(6) + \Gamma_{PB}(7) \\
&= \frac{1}{8^{8-0}} + \frac{1}{8^{8-0}} + \frac{1}{8^{8-1}} + \frac{1}{8^{8-2}} + \frac{1}{8^{8-3}} + \frac{1}{8^{8-2}} + \frac{1}{8^{8-1}} + \frac{1}{8^{8-0}} \text{ (Lemma 8.1)} \\
&= 1/256 + 1/256 + 1/128 + 1/64 + 1/32 + 1/64 + 1/128 + 1/256 \\
&= 23/256
\end{aligned}$$

For Φ_2 we only need to consider the first expression. We can have at least 3 comparisons, iff pattern-blocks 8 and (assume we select) 0 match with the corresponding text-blocks.

$$\begin{aligned}
\Phi_2 &= \Gamma_{PB}(8) \times \Gamma_{PB}(0) \\
&= \frac{1}{8^{8-0}} \times \frac{1}{8^{8-7}} \text{ (Lemma 8.1)} \\
&= 1/256 \times 1/2 \\
&= 1/512
\end{aligned}$$

In an attempt, for all $m \geq 15$, after λ comparisons the pattern-blocks which have not yet been compared will be similar to the expressions for patterns of length m', $7 \leq m' \leq 14$, where $m' = (m \bmod 8) + 8$ if $m \bmod 8 \neq 7$. Otherwise $m' = 7$. In other words, if we remove all the λ columns with all zeros from the expressions of pattern length $m \geq 15$, the number of wildcards in pattern-blocks will be the same as in the expressions of pattern length m'. For example, if we remove (i.e. λ) columns of all zeros from the number of wildcards in pattern-blocks, for $m = 34$ (see Figure 8.2), we will get the number of wildcards in pattern-blocks, for $m' = 10$ (see Figure 8.4) as in Figure 8.5.

Note that in any attempt for all m, we can have at most $\lambda + 1$ matches before we make the last comparison, if $m \bmod 8 = 0$, 1 or 7, otherwise $\lambda + 2$. For $m > 15$, we need to know $\Phi_{\lambda+1}$ and $\Phi_{\lambda+2}$. From the above observation we can calculate these values from the values of Φ_1 and Φ_2 for m, $7 \leq m \leq 14$. From these base values we can have Lemma 8.3. Note that $\lambda = 0$ for all $m \leq 14$.

Exp0:	7 (0)	0 (32)	7 (40)
Exp1:	6 (1)	0 (33)	
Exp2:	5 (2)	1 (34)	
Exp3:	4 (3)	2 (35)	
Exp4:	3 (4)	3 (36)	
Exp5:	2 (5)	4 (37)	
Exp6:	1 (6)	5 (38)	
Exp7:	0 (7)	6 (39)	

Figure 8.5: *The number of wildcards in pattern-blocks, for* $m' = 10$

Lemma 8.3: For $m \geq 7$,

$$\Phi_{\lambda+1} = (1/256)^{\lambda} \times \alpha_b \text{ and}$$

$$\Phi_{\lambda+2} = (1/256)^{\lambda} \times \beta_b,$$

where α_b and β_b are the values of b^{th} base case in the first and second columns in Table 8.3 respectively and $b = m \bmod 8$.

base case	α	β
0	11/64	
1	15/128	
2	23/256	1/512
3	1/16	1/512
4	13/256	1/512
5	5/128	3/2048
6	9/256	5/4096
7	7/32	

Table 8.3: *The associated probabilities for* α *and* β *for each base case*

Let Ψ_i be the probability of making *exactly* i comparisons at an attempt. Using Φ_i we can have an equation for Ψ_i:

$$\Phi_i = \Psi_{i+1} + \Psi_{i+2} + \cdots$$

This gives

$$\Psi_i = \Phi_{i-1} - \Phi_i$$

We know that we will make at least one comparison in every attempt. So Φ_0 is 1.

For all m and $\sigma = 2$, the maximum number of comparisons in any attempt is $\mu = \lceil \frac{m+7}{8} \rceil$, which is equal to $\lambda + 2$ if $m \bmod 8 = 0$, 1 or 7, otherwise $\lambda + 3$. So Φ_i is 0 for all $i \geq \mu$. This gives:

$$\Psi_1 = 1 - \Phi_1$$

$$\Psi_i = \Phi_{i-1} - \Phi_i, 2 \leq i \leq \mu - 1$$

$$\Psi_\mu = \Phi_{\mu-1}$$

Lemma 8.4: For $\sigma = 2$, the total number of comparisons, Ψ_{Total}, is less than or equal to $2n'$ on average, where n' is the number of text-blocks in the text.

Proof:
$$\begin{aligned}
\Psi_{Total} &= n' \times \sum_{i=1}^{\mu} i \times \Psi_i \\
&= n' \times ((1 - \Phi_1) + 2(\Phi_1 - \Phi_2) + \cdots + \mu - 1(\Phi_{\mu-2} - \Phi_{\mu-1}) + \mu\Phi_{\mu-1}) \\
&= n' \times (1 + \Phi_1 + \cdots + \Phi_{\mu-2} + \Phi_{\mu-1}) \\
&= n' \times (1 + \sum_{i=1}^{\lambda} \frac{8}{256^i} + \Phi_{\lambda+1} + \Phi_{\lambda+2}) \text{ (Lemma 8.2)} \\
&\leq 2n'
\end{aligned}$$

This is because $\sum_{i=1}^{\lambda} \frac{8}{256^i} + \Phi_{\lambda+1} + \Phi_{\lambda+2} \leq 1$ (Lemmas 8.2 and 8.3) □

Lemma 8.5: For $\sigma > 2$, the total number of comparisons, Ψ_{Total}, is $O(n)$, where n is the size of the original text.

Proof: The probability of more than one comparison in an attempt is $\Phi_1 + \cdots + \Phi_{\mu-2} + \Phi_{\mu-1}$ (see Lemma 8.4), where $\mu = \lceil \frac{m\lceil \log_2 \sigma \rceil + 7}{8} \rceil$. Note that $m\lceil \log_2 \sigma \rceil$ is the length of the pattern when we convert it into a binary string. We show in section 8.4 that in an attempt we only need to consider a maximum of $\lceil \frac{8}{\lceil \log_2 \sigma \rceil} \rceil$ expressions when $\sigma > 2$. Hence, for $\sigma > 2$, $\Phi_1 + \cdots + \Phi_{\mu-2} + \Phi_{\mu-1}$ is less than the value given for $\sigma = 2$.

□

From these Lemmas we have the following Theorem.

Theorem: The average running time of our algorithm is $O(n + m)$.

To show this is also true in practice we counted the number of comparisons by running our algorithm. Tables 8.4 and 8.5 shows the estimated number of comparisons (Ψ_{Total}) and the actual number of comparisons. We used the same texts for each σ as in Table 8.1. For each pattern length we use 100 random patterns. The actual number of comparisons in Tables 8.4 and 8.5 is the total number of comparisons divided by the number of patterns of that length. The pattern length given in Tables 8.4 and 8.5 is the length of the original pattern.

alphabet of 2			alphabet of 4			alphabet of 8		
Pat Len.	Ψ_{Total}	Actual	Pat Len.	Ψ_{Total}	Actual	Pat Len.	Ψ_{Total}	Actual
5	85938	85413	2	156250	156258	2	207031	255959
10	68237	68276	4	135742	136513	4	190795	191710
20	64556	64446	8	127288	126999	6	189632	189931
30	64460	64460	12	126962	126962	8	189462	189537
40	64460	64467	18	126960	126962	12	189460	189898
50	64460	64473	24	126960	126962	16	189460	189551

Table 8.4: *Estimated versus actual number of comparisons of our BRS algorithm*

alphabet of 16			alphabet of 32			alphabet of 64		
Pat. Len.	Ψ_{Total}	Actual	Pat Len.	Ψ_{Total}	Actual	Pat Len.	Ψ_{Total}	Actual
2	260742	265331	2	318237	322155	2	378296	378678
4	252288	252013	3	314507	314567	3	377132	376990
6	251960	251956	4	314556	314581	4	376962	376980
8	251960	251962	6	314460	314509	5	376962	376965
10	251960	251957	8	314460	314297	7	376960	376482
12	251960	251959	10	314460	314348	9	376960	376503

Table 8.5: *Estimated versus actual number of comparisons of our BRS algorithm*

8.6 Comparison with existing string matching algorithms

In this section we compare the existing string matching algorithms with our algorithm, the BRS algorithm. There are a number of string matching algorithms available in the literature. We have chosen seven of them, BR, BM, HOR, QS, RAI, SMI, RF and

NR algorithms which can be found in [31, 36, 70, 129, 109, 125, 48, 102] respectively. The first six algorithms were found to be fast in [31]. Animations of these algorithms can be found at [39] and more information about the algorithms can be found in [38].

The experiments were carried out for all the algorithms on an un-compressed text, except for our BRS algorithm and the NR algorithm [102]. The text used for these experiments was the same text as in Table 8.1. The patterns used in these experiments are generated randomly. For each σ and m, we tested 100 patterns and we measured the total (user) time (including pre-computation time) in seconds to search for all 100 patterns. We repeat each test 10 times and take the average. We used an Intel 486-DX2-66 processor based machine with 8 megabytes of RAM and a 100 megabyte hard drive running S.u.S.E. Linux 5.2 to conduct the experiments. All the algorithms were coded in C. The results of the experiments are in Tables 8.6 to 8.10 .

Pat. Length	BRS	BR	BM	HOR	QS	RAI	SMI	RF	NR
5	14.2	29.1	31.3	31.5	31.1	28.7	31.8	32.6	30.7
10	5.3	27.0	24.7	30.9	31.0	27.7	31.4	22.0	30.5
20	4.4	27.3	20.4	28.8	32.4	26.6	31.0	18.2	29.5
30	4.2	27.3	18.3	31.2	31.2	28.0	31.4	16.0	27.5
40	4.2	28.3	17.3	29.7	31.3	27.9	30.7	13.5	28.5
50	5.2	26.5	16.4	30.5	30.0	27.7	31.1	15.0	28.4

Table 8.6: *Times in seconds to search for 100 random patterns in each given text with* $\sigma = 2$

Pat. Length	BRS	BR	BM	HOR	QS	RAI	SMI	RF	NR
4	8.6	15.3	20.5	20.9	20.3	20.2	23.8	21.3	21.6
8	5.3	13.1	17.6	18.1	19.3	18.7	19.5	17.3	20.7
12	5.7	12.5	19.3	18.8	18.7	18.0	18.3	15.3	17.6
16	5.7	12.9	17.4	15.8	17.4	17.3	17.7	13.6	18.4
20	5.7	12.0	17.2	18.5	17.6	17.9	18.5	14.1	20.5
24	5.7	12.5	16.7	17.7	18.6	16.6	18.1	12.7	20.2

Table 8.7: *Times in seconds to search for 100 random patterns in each given text with* $\sigma = 4$

Pat. Length	BRS	BR	BM	HOR	QS	RAI	SMI	RF	NR
3	9.9	15.7	22.6	18.6	17.6	16.9	19.3	18.0	28.9
4	14.0	17.0	25.8	21.2	18.7	21.1	21.3	18.9	27.6
6	8.6	13.7	19.7	16.9	16.0	16.5	16.0	15.8	23.5
10	8.5	12.7	15.7	14.1	15.1	14.9	15.1	14.1	25.5
14	8.7	12.0	15.7	12.4	14.2	13.3	13.8	13.0	25.2
18	8.4	11.1	15.0	13.6	14.0	12.7	13.4	13.2	25.5

Table 8.8: *Times in seconds to search for 100 random patterns in each given text with* $\sigma = 8$

Pat. Length	BRS	BR	BM	HOR	QS	RAI	SMI	RF	NR
2	14.1	19.4	33.0	25.8	21.0	24.4	24.4	19.6	33.4
4	9.8	15.2	21.7	17.8	17.0	17.9	17.7	16.2	31.5
6	9.8	13.4	16.6	14.0	14.6	13.6	15.0	13.4	31.5
8	9.7	12.3	16.2	14.4	13.9	13.2	13.8	13.2	27.7
10	9.7	12.1	14.2	13.2	13.6	12.3	13.0	13.6	29.6
12	9.9	11.1	14.3	13.0	12.3	13.0	13.4	12.9	31.1

Table 8.9: *Times in seconds to search for 100 random patterns in each given text with* $\sigma = 16$

Pat. Length	BRS	BR	BM	HOR	QS	RAI	SMI	RF	NR
2	66.5	18.6	33.0	23.6	19.5	23.8	22.3	19.0	39.1
3	42.1	16.3	24.0	20.2	17.7	19.8	20.3	16.6	40.1
4	31.9	14.8	21.2	17.1	15.5	15.4	17.5	15.0	37.2
6	39.7	12.3	17.5	13.2	14.7	14.6	15.5	14.1	36.2
8	37.9	12.3	15.5	13.4	13.4	13.2	13.9	13.5	38.8
10	48.2	11.5	15.0	12.4	11.8	12.5	13.7	13.0	34.2

Table 8.10: *Times in seconds to search for 100 random patterns in each given text with* $\sigma = 32$

8.7 Conclusions

The method described in Section 8.2 to store a text will reduce the original text size to $\frac{\lceil \log_2 \sigma \rceil}{8} n$. Although this method is not compression as in the literature, it reduces the space and it is comparable with the existing methods.

The main aim of this Chapter is string matching in an efficiently stored text. Our string matching algorithm compares two blocks, checks whether a prefix (or suffix) of a block is a suffix (or prefix) of the other block. This takes constant time and uses byte processing. In practice, byte processing is much faster than bit processing because bit shifting and masking operations are not necessary at search time. We prove that the average time taken by our algorithm is $O(n + m)$. We also justified our average running time by experiments.

Using our algorithm one can keep texts (with an alphabet of $2 \leq \sigma \leq 128$ characters) efficiently stored indefinitely and perform the search for a pattern. These

methods will save both time and space. The experimental results show that our algorithm is more efficient than the existing algorithms for $\sigma \leq 16$. Texts with such a small alphabet are DNA, RDNA and hexadecimal files. One can improve our algorithm so that it performs well for large alphabet sets.

Chapter 9

String matching via pre-processing the text

9.1 Introduction

In the previous chapters we have pre-processed the pattern before searching for it in the text. Using this method we make $\frac{n}{m}$ and n attempts for the best case and worst-case respectively, for finding a pattern in a text where n is the length of the text and m is the length of the pattern. In this Chapter we show that the number of attempts can be reduced by pre-processing the text. This method takes $2n + 2\sigma$ space to store all the information required compared to a suffix tree that requires $4n$ to store it.

When comparing a pattern and text at an attempt we get partial matches. The number of partial matches depends on the frequency of the pattern characters in the text. The more frequent the pattern characters the more partial matches we achieve. The number of attempts can be decreased by considering positions in the text where the least frequent pattern character occurs. To calculate the positions of the least

frequent pattern character we pre-process the text and record the positions of each character in the text.

9.2 The algorithm

The following algorithm records the frequency, the first position and next position of the characters of the text. This can be done in one pass of the text.

The position of the next occurrence of a character is stored in an array of size n called *Next*. We use an array of length σ called *Freq* to record the frequency of the characters of the alphabet in the text where σ is the size of the alphabet set. We use another array of length σ called *First* to record the first occurrence of each character, starting from the beginning of the text. *Next[i]* is the position in the text of the next occurrence of the character at position i in the text. *First[α]* is the first position in the text of the character, α and *Freq[α]* is the frequency of the character, α.

The text is stored in an array. The start of the text being at the leftmost end of the array. We start from the end of the text or the rightmost end of the text array. The *Next* array is aligned with the text array and the array *First* is set to -1 and *Freq* is set to zero.

We consider the last character in the text array first and consider character from right to left until we are at the first character in the text. When we are at position i in the text and the character at this position is α, we do the following steps:

Step 1: Increase *Freq[α]* by one

Step 2: Copy *First[α]* into *Next[i]*

Step 3: *First[α]* = *i*

For example, consider the string BAABA, there are two characters and so *Freq* and *First* are of length two. Figure 9.1 shows the initial state of the arrays *Freq*, *First* and *Next*. The *Next* array is empty and is aligned with text as shown.

Pos.	0	1	2	3	4
Text	B	A	A	B	A
Next					

Frequency		First Pos.	
A	B	A	B
0	0	-1	-1

Figure 9.1: *The initial values for Next, Freq and First*

We start at the last position (position 4) of the text. The last character in the text is an A and *Freq[A]* is incremented by one. *First[A]* is copied into *Next[4]*. Therefore *Next[4]* = -1 and we update *First* and thus *First[A]* = 4. This is shown in Figure 9.2.

Pos.	0	1	2	3	4
Text	B	A	A	B	A
Next					-1

Frequency		First Pos.	
A	B	A	B
1	0	4	-1

Figure 9.2: *The values for Next, Freq and First after considering the A at position 4*

The next character in the text is a B and so the frequency of B is incremented by one. The value for B in the *First* array is entered into the *Next* array. Therefore *Next[3]* = -1 and we update *First* and thus *First[B]* = 3. This is shown in Figure 9.3.

Pos.	0	1	2	3	4
Text	B	A	A	B	A
Next				-1	-1

Frequency		First Pos.	
A	B	A	B
1	1	4	3

Figure 9.3: *The values for Next, Freq and First after considering the B at position 3*

The next character in the text is an A and so the frequency of A is incremented by one. The value for A in the *First* array is entered into the *Next* array. Therefore $Next[2] = 4$ and we update *First* and thus $First[A] = 2$. This is shown in Figure 9.4.

Pos.	0	1	2	3	4
Text	B	A	A	B	A
Next			4	-1	-1

Frequency		First Pos.	
A	B	A	B
2	1	2	3

Figure 9.4: *The values for Next, Freq and First after considering the A at position 2*

The next character in the text is an A and so the frequency of A is incremented by one. The value for A in the *First* array is entered into the *Next* array. Therefore $Next[1] = 2$ and we update *First* and thus $First[A] = 1$. This is shown in Figure 9.5.

Pos.	0	1	2	3	4
Text	B	A	A	B	A
Next		2	4	-1	-1

Frequency		First Pos.	
A	B	A	B
3	1	1	3

Figure 9.5: *The values for Next, Freq and First after considering the A at position 1*

The next character in the text is a B and so the frequency of B is incremented by one. The value for B in the *First* array is entered into the *Next* array. Therefore $Next[0] = 3$ and we update *First* and thus $First[B] = 0$. This is shown in Figure 9.6. The arrays are now complete.

Pos.	0	1	2	3	4
Text	B	A	A	B	A
Next	3	2	4	-1	-1

Frequency		First Pos.	
A	B	A	B
3	2	1	0

Figure 9.6: *The values for Next, Freq and First after considering the B at position 0*

Once we have the list of the positions of the characters in the text we can search for a pattern. For a given pattern we chose the character in the pattern that appears in the text the least number of times. If there are two characters with equal lowest frequency then we choose the character that is leftmost in the pattern. The frequency of this character is the number of attempts required to search for the pattern. The pattern and text are aligned so that the least frequent character in the pattern is aligned with the first occurrence of that character in the text. We can find this position from the array, *First*. Then we compare the text and pattern characters from left to right.

Upon a mismatch or match we shift the pattern so as to align the next occurrence of the least frequent character in the pattern with its matching text character. This position can be found from the array, *Next*.

For example, if we were to search for the pattern, BA, using the above arrays then we first check which character has the lowest frequency. In this case B has the lowest frequency. As the character B only occurs twice then we will only make two attempts at matching the pattern and text characters. Using *first* we know that the first occurrence of B is at position 0. So we align the B in the pattern with the

corresponding B in the text and begin our attempt:

Text	B	A	A	B	A
Comparison	=	=			
Pattern	B	A			

Both of the pattern characters match the text and we have a full match. The pattern now has to be shifted. The value of *Next* at position 0 is 3, we shift the pattern as shown:

Text	B	A	A	B	A
Comparison				=	=
Pattern				B	A

The value of *Next* at position 3 is -1 and so we know that there are no more occurences of B in the text.

The pre-processing can be done using an extra $n + 2\sigma$ space. Where σ is the size of the alphabet set. This pre-processing can be done in $O(n)$ time.

The worst-case is that we have to make n attempts and compare m characters at each attempt. Therefore the new algorithm has a worst-case run time of $O(nm)$.

9.3 Average case analysis

If we assume that each character in the text occurs with equal frequency. Then the probability of a match between a text character and a pattern character is $\frac{1}{\sigma}$. When we construct the list of the positions of the characters in the text there are going to be σ lists each of length $\frac{n}{\sigma}$.

Lemma 9.1: The upper bound number of attempts is $\frac{n}{2}$ for an average case text.

Proof: When we search for a pattern, all of the characters in the pattern will have the same frequency. We will choose the first character in the pattern as it is the leftmost character in the pattern. The least likely character in the pattern occurs $\frac{n}{\sigma}$ times and so we have to make $\frac{n}{\sigma}$ attempts. If we assume that $\sigma \geq 2$. This means that at most we will make $\frac{n}{2}$ attempts. □

From Lemma 4.1 we know that the upper bound number of comparisons at an attempt is $\sum_{i=0}^{m-1}(\frac{1}{\sigma^i})$ for an average case text. The equation is maximised when $\sigma = 2$. As m increases the equation approaches the limit for this equation which is 2. So we expect to make at most 2 comparisons on average at each attempt.

So upon an attempt we expect to make at most 2 comparisons and to make at most $\frac{n}{2}$ attempts. If we multiply these two values we get n. This gives the following Theorem.

Theorem: The new algorithm has an average case running time of $O(n)$.

9.4 Recording the positions of more than one character

The algorithm described in Section 9.2 can be expanded to record the positions of strings of characters rather than just single characters. The array of size n containing the positions of the characters or strings remains the same length as we record the positions of the start of the strings. From these positions we can calculate the starting

position of a possible match between the text and the pattern. When we increase the size of the strings to be recorded, the *Freq* and *First* arrays increase in size quadratically. So to store strings of length x we require $n + 2\sigma^x$ space. When σ is small we can store the positions of longer strings. When σ is large the amount of space required grows very quickly as x increases.

If the characters in the text are approximately equal then we would expect an even distribution of the possible σ^x strings throughout the text. When searching for a pattern we expect to make approximately $\frac{n}{\sigma^x}$ attempts. So to decrease the amount of attempts required to search for a pattern we increase the size of the strings that we record.

9.4.1 Comparing the pre-processing algorithm with $x = 1$ and $x = 2$

Using the pre-processing algorithm we compare the time taken for searching for patterns recording strings of lengths 1 ($x = 1$) and 2 ($x = 2$). The searches were conducted on different texts with different sizes of σ. From these times we will be able to see if the length of the string to be recorded in the *Next* array affects the amount of time taken by the algorithm. The main application of searching is in text files. So we will use eight English texts from [37]. The texts range in size and are the same as those used in Chapter 4.1. We searched the English texts for the 24,966 words in the UNIX dictionary.

We used a 486-DX66 with 32 megabytes of RAM and a 100 megabyte hard drive running SUSE 5.2. The user time includes the time taken for any pre-processing and the reading of the text into memory. Each algorithm was evaluated ten times and the average time taken is given in Table 9.5. The difference between the slowest and fastest time for each test for an algorithm was less than 0.2 of a second.

Text (size)	Total		Preprocessing		search		search / no. of pats.	
	x = 1	x = 2	x = 1	x = 2	x = 1	x = 2	x = 1	x = 2
book 1 (773635)	667.5	107.8	74.3	78.8	593.2	29.0	0.02359	0.00124
book 2 (610856)	540.8	85.3	58.4	59.7	482.4	25.6	0.01919	0.00091
paper 1 (53162)	33.9	11.2	5.2	5.2	28.7	6.0	0.00114	0.00030
paper 2 (82205)	64.1	15.2	7.7	8.3	56.4	6.9	0.00224	0.00029
paper 3 (47139)	30.9	11.0	4.4	4.7	26.5	6.3	0.00105	0.00031
paper 4 (13292)	9.2	6.6	1.3	1.4	7.9	5.2	0.00031	0.00027
paper 5 (11960)	8.5	6.4	1.2	1.3	7.3	5.1	0.00029	0.00029
paper 6 (38111)	22.0	9.1	3.6	3.7	18.4	5.4	0.00073	0.00025

Table 9.1: *Time taken (in seconds) to search for the UNIX dictionary in the given texts*

In Table 9.1, the second row shows the length of the strings that were recorded. Column 1 shows the text used from [37]. The number of characters in the text is given in brackets. Columns 2 and 3 show the total time taken when searching for the UNIX dictionary (24966 words) in the given texts. Columns 4 and 5 show the time taken for preprocessing such as reading the text into the text array and building the *Next*, *First* and *Freq* arrays. Columns 6 and 7 show the amount of time taken to search for

the dictionary. This is calculated by deducting the time for the preprocessing from the total time taken. Columns 8 and 9 show the average time taken to search for a pattern in the text.

From Table 9.1 we can see that recording strings of length 2 are the best method, especially when the text is large. When the time taken for the pre-processing is removed from the total time, we can see that the actual search for the patterns in the text is quicker when we record the positions of longer strings. This is due to the number of attempts and comparisons being reduced as the length of the strings we record increases.

Text (size)	Number of comparisons		Number of attempts	
	x = 1	x = 2	x = 1	x = 2
book 1 (773635)	323282898	20215522	226448219	11106768
book 2 (610856)	264843964	15997981	187032763	9044580
paper 1 (53162)	22889598	1209387	16303290	664394
paper 2 (82205)	39086347	2125583	27979108	1186094
paper 3 (47139)	20544541	1322385	14709200	829796
paper 4 (13292)	5804608	330685	4089852	168079
paper 5 (11960)	4821030	264370	3425096	126769
paper 6 (38111)	14426883	758209	10201079	413144

Table 9.2: *The number of attempts and comparisons taken when searching for the UNIX dictionary in the given texts*

From Table 9.2 we can see that if we increase the size of the strings that we record then the number of comparisons and the number of attempts are reduced.

9.5 Comparing the new algorithm with the existing algorithms

From Chapter 4, we found that the 2 fastest string matching algorithms were the HOR [70] and the BR algorithm [31]. Both algorithms read the text into an array of length n and all searches are performed upon the text array. We searched the texts used in Table 9.1 for the 24996 words in the UNIX dictionary using the HOR and BR algorithms. The results are given in Table 9.3.

Text (size)		Total time taken		search		search / no. of pats.	
	read in	BR	HOR	BR	HOR	BR	HOR
book 1 (773635)	74.05	2812.00	3514.00	2737.95	3439.95	0.10890	0.13682
book 2 (610856)	57.87	2215.00	2780.00	2157.13	2722.13	0.08579	0.10827
paper 1 (53162)	5.12	147.76	228.00	142.64	222.88	0.00567	0.00886
paper 2 (82205)	8.12	259.06	307.00	250.94	298.88	0.00998	0.01189
paper 3 (47139)	4.46	131.95	129.00	127.49	124.54	0.00507	0.00495
paper 4 (13292)	1.26	40.87	38.41	39.61	37.15	0.00158	0.00148
paper 5 (11960)	1.14	35.28	33.81	34.14	32.67	0.00136	0.00130
paper 6 (38111)	3.17	108.08	105.17	104.91	102.00	0.00417	0.00406

Table 9.3: *Time taken (in seconds) to search for the UNIX dictionary in the given texts using the HOR and BR algorithms*

In Table 9.3, the second column shows the amount of time required to read the text into the text array ready for searching. Columns 3 and 4 show the total time taken for the given texts. Columns 5 and 6 show the time taken to search for the dictionary in the given texts. This is calculated by deducting the time taken to read the text into the text array from the total time. Columns 7 and 8 show the average

time required to search for a pattern. Note that the time taken is only fractions of a second. The scanning of the text takes the bulk of the time. For the preprocessing method to be better than the HOR and BR algorithms the time taken to build the *Next*, *First* and *Freq* arrays has to be offset.

The time taken to build the *Next*, *First* and *Freq* arrays is the time taken for the preprocessing in Table 9.1 minus the time taken for reading the text into the text array. In Table 9.4 we shows the time taken to build *Next*, *First* and *Freq* for the eight texts used.

Text (size)		Preprocessing		preproc. - scan in	
	read in	x = 1	x = 2	x = 1	x = 2
book 1 (773635)	73.52	74.30	78.8	0.78	5.25
book 2 (610856)	57.87	58.40	59.7	0.53	1.83
paper 1 (53162)	5.11	5.36	5.2	0.25	0.09
paper 2 (82205)	7.22	7.70	8.3	0.48	1.08
paper 3 (47139)	4.40	4.51	4.7	0.11	0.30
paper 4 (13292)	1.26	1.40	1.4	0.14	0.14
paper 5 (11960)	1.12	1.22	1.3	0.10	0.18
paper 6 (38111)	3.17	3.60	3.7	0.43	0.53

Table 9.4: *Time taken to build Next, First and Freq*

In Table 9.4, column 2 shows the amount of time required to read into the text array each of the given texts. Columns 3 and 4 show the amount of time taken to pre-process the given texts as shown in Table 9.1. By taking the time taken for reading the text into the text array from the total time for pre-processing we get the time

taken for building the arrays *Next*, *First* and *Freq*, which are given in columns 5 and 6.

Due to the time taken to build the arrays *Next*, *First* and *Freq* the new algorithm is not always better than using the HOR and BR string matching algorithms. If we are searching for a low number of patterns then the HOR and BR algorithms will be better. We can calculate how many patterns need to be searched for the pre-processing method to become the fastest algorithm for the task.

The pre-processing algorithm is the fastest algorithm once the time taken to build the arrays *Next*, *First* and *Freq* is offset. The method using $x = 2$ is faster for all searches that search for more than

$$\frac{\textit{time taken to build the Next, First and Freq arrays}}{(\textit{HOR or BR average search}) - (\textit{pre-processing } (x{=}2) \textit{ average search})}.$$

All answers must be rounded up as we cannot search for a fraction of a pattern.

In Table 9.5, column 2 shows the amount of time required to build the arrays *Next*, *First* and *Freq*. Columns 3 and 4 show the average amount of time taken to search for a pattern using $x = 2$ and the BR one dimensional string matching algorithm. Column 5 shows the difference between the BR and $x = 2$ average search. Column 6 shows the number of patterns that need to be searched for the $x = 2$ method to be faster than the BR string matching algorithm. The figure is found by dividing column 2 by column 5. This is the same as the equation above.

Text (size)	construct arrays	x = 2 search	BR search	BR - x = 2	number of patterns
book 1 (773635)	5.25	0.001240	0.108895	0.107655	49
book 2 (610856)	1.83	0.000914	0.085794	0.084881	22
paper 1 (53162)	0.09	0.000299	0.005673	0.005374	17
paper 2 (82205)	1.08	0.000293	0.009981	0.009687	112
paper 3 (47139)	0.3	0.000312	0.005071	0.004759	64
paper 4 (13292)	0.14	0.000271	0.001575	0.001305	108
paper 5 (11960)	0.18	0.000287	0.001358	0.001071	169
paper 6 (38111)	0.53	0.000254	0.004173	0.003918	136

Table 9.5: *Number of patterns that are required to be searched for, for the* $x = 2$ *method to be the fastest*

When the text is long we only need to search for a small number of patterns for the $x = 2$ method to become the best. When the text is shorter we need to search for more patterns to get faster times with the $x = 2$ method. We can choose which algorithm to use depending on the number of patterns and the length of text being searched.

9.6 Pre-processing DNA using a mapping function

When we pre-processed the English texts in the previous section we used a one dimensional array to record the *Next* array. If we are to record longer strings we have to use a mapping function to map each of the possible strings to a unique location in a linear array. Using the mapping function we can then update the *Next*, *First* and *Freq* arrays.

The easiest mapping function to use is based on the number of characters in the text and the length of the strings being recorded. We can assign a value to each string in a similar way to that which is used to assign values to each of the ASCII characters. We firstly number each of the characters in the alphabet from 0 to $\sigma - 1$. If we let each of characters in the string to be recorded to be numbered from 1 to m numbering from left to right. Then the value of a string is $\sum_{i=1}^{x} string[i] \times \sigma^{x-i}$, where $string[i]$ is the numeric value of the characters at the i^{th} position in the *string*. We can then enter the position of the string into the relevant arrays. There are σ^x possible strings to be recorded.

For example if x = 5 and $\sigma = 4$ (DNA alphabet) and consider the string GATCTAGACAC, the first string we would record is GACAC. As we are recording the strings from right to left the last string to be recorded is GATCT. Calculating this string we firstly replace the characters with their numeric equivalents, A = 0, C = 1, G = 2 and T = 3 as in Chapter 7. To give GACAC = 20101. We now need to find the value of the string filling in the values in the equation we have: $\sum_{i=1}^{5} string[i] \times 4^{5-i}$. To give :

$i = 1 : string[1] \times 4^{5-1} = 2 \times 4^4 = 512$

$i = 2 : string[2] \times 4^{5-2} = 0 \times 4^3 = 0$

$i = 3 : string[3] \times 4^{5-3} = 1 \times 4^2 = 16$

$i = 4 : string[4] \times 4^{5-4} = 0 \times 4^1 = 0$

$i = 5 : string[5] \times 4^{5-5} = 1 \times 4^0 = 1$

The total for GACAC = 512 + 0 + 16 + 0 + 1 = 529.

The next string to be calculated is AGACA. We can calculate the value represented by this string by using the above method or we can modify the above total to give us the next total. We need to remove the value of the last character in the string away from the total. In the above case this would be 1 which is the value of the second C in GACAC. This makes the total 528. We now move each of characters to the right. This makes each of the characters decrease in value by a factor of 4. So we divide the total by 4 to get 132. We now add the value of the new character multiplied by $4^4 = 0 * 256 = 0$. So the value of AGACA is 132. We can check this using the above method which gives 0 + 128 + 0 + 4 + 0 = 132. Using this method we reduce the number of mathematical operations from 5 to 3. No matter how long the strings that we record are we always make 3 mathematical operations.

Using this mapping function we were able to search in texts recording longer strings in the text. Using 5 of the DNA texts used in Chapter 7 we searched for patterns of different lengths and used different values of x for the building of the arrays *Next*, *First* and *Freq*. The sizes of the texts can be found in Table 7.1 in Chapter 7. In our first experiment using DNA we searched for the unique 256 patterns of length 4.

In Table 9.6, the numbers in the first row show the length of the strings that we are recording. In Column 1 we give the number of the text and the number of

characters in that given text is given in brackets. The texts used are the same as those used in Chapter 7. We can see that recording strings of length 4 is the fastest method for all 5 texts searched. This is due to the amount of text that is not searched when searching for a pattern. As we are searching for strings of length 4 and we are recording strings of length 4 we have recorded the positions of all possible matches. Searching for the patterns is simply the printing of the list of recorded positions for that pattern.

Text (size)	x = 1	x = 2	x = 3	x = 4
1 (100000)	20.6	13.5	11.7	11.0
3 (100000)	20.5	13.5	11.8	11.1
5 (172000)	49.9	34.0	27.9	26.4
7 (253505)	62.3	42.9	35.4	34.0
9 (319000)	42.8	28.8	25.8	23.0

Table 9.6: *Time taken in seconds to search for the 256 possible DNA patterns of length 4 including any pre-processing time.*

Text (size)	x = 1	x = 2	x = 3	x = 4	x = 5	x = 6
1 (100000)	163.49	46.21	19.84	13.95	12.09	11.90
3 (100000)	160.49	45.74	19.98	13.97	12.23	11.66
5 (172000)	392.25	135.40	53.89	34.42	28.82	27.47
7 (253505)	511.93	176.63	68.90	42.40	36.06	35.57
9 (319000)	344.96	117.79	46.49	29.28	25.28	24.27

Table 9.7: *Time taken in seconds to search for the 4096 possible DNA patterns of length 6 including any pre-processing time*

In Table 9.7, we show the time taken to search for all 4096 possible strings of length 6. We can see that recording strings of length 6 are the best method for searching for the patterns. This method ensures that the least amount of text possible is searched for the patterns. Again the length of the strings recorded is the same as the length of the strings we are searching for. This means that we have a recorded list of all possible matches and to search for all occurrences of a pattern is just a case of printing out the list of recorded positions for that pattern.

Text (size)	x = 1	x = 2	x = 3	x = 4	x = 5	x = 6	x = 7	x = 8	x = 9
1 (100000)	162.7	53.5	22.6	14.5	12.6	12.1	12.1	11.8	11.9
3 (100000)	159.0	52.8	22.4	14.7	25.4	11.8	11.7	11.6	12.0
5 (172000)	389.9	162.0	54.5	34.2	28.4	27.5	27.3	26.9	26.8
7 (253505)	507.7	184.0	77.6	43.8	37.0	34.5	33.5	33.9	33.3
9 (319000)	418.0	122.9	56.5	29.2	21.4	23.4	23.7	23.6	24.0

Table 9.8: *Time taken to search for 5000 DNA patterns of length 10 including any pre-processing time*

In Table 9.8, we can see that recording strings of length 9 is not the best method for all the texts that we used. Using strings of lengths 8, 8, 9, 9 and 5 are the best methods for each of the texts 1 to 9 respectively. This may be due to the chosen patterns not occurring very frequently in some of the texts and it could also be possible that prefixs of the patterns don't occur in the text. For patterns of length 10 it is difficult to choose a single length of string to record for all the texts. Recording patterns from length 6 to 10 for each of the texts, the times for a search differ by

a very small amount. As the length of the strings being recorded is increased it is possible that the time taken for the pre-processing is degrading the savings made in the search phase of the algorithm.

text	1	2	3	4	5	6	7	8	9
1 (100000)	10.4	10.6	11.0	10.5	10.8	10.5	10.8	10.6	11.0
3 (100000)	10.9	10.8	11.0	10.8	10.9	10.9	10.8	10.8	10.9
5 (172000)	26.6	26.4	25.5	25.7	25.8	26.0	26.0	26.7	26.1
7 (253505)	31.8	32.2	32.2	32.5	32.6	31.6	32.6	32.6	32.5
9 (319000)	22.1	22.2	23.1	22.2	22.5	22.2	22.9	22.5	23.2

Table 9.9: *Time taken in seconds for pre-processing*

From Table 9.9, we can see that as we increase the value of x the time taken are all within a second of each other. This due to the way in which we calculate the mapping function for recording the positions of the strings in the text.

9.7 Conclusions

From the experiments performed, we have found that indexing the positions of strings of varying lengths can be faster than traditional string matching. Although this new method is only faster when the number of patterns to be searched for is quite large. If we are searching for only a few patterns then the traditional algorithms are the best algorithms to use. This is due to the amount of time that is required to build the required arrays in the new method. The preprocessing in the new method requires

more memory and time to record the positions of the chosen length strings in the text. We have to search for enough patterns to offset the time taken building the arrays *Next*, *First* and *Freq*. The length of the text is linked with the number of patterns that need to be searched. From Table 9.5 we can see that for the larger texts (books 1 and 2) the number of patterns that need to be searched to offset the time taken to build the arrays *Next*, *First* and *Freq* is less than those required for the smaller files (papers 1-6). This is due to the larger percentage of text not compared with the pattern when the larger texts are searched.

Chapter 10

Searching in an Efficiently Stored DNA Text Using a Hardware Solution

10.1 Introduction

In this Chapter we describe a hardware solution that searches in the efficiently stored DNA text used in Chapter 7. We also describe an algorithm coded in the programming language C that will be synthesized into hardware. A DNA text (or molecule) encodes information which by convention is represented as a string over the DNA alphabet A, C, G and T.

10.2 Investigation into a hardware only solution to the string matching problem

The string matching algorithm illustrated in Figure 10.1 was devised to investigate the feasibility of performing computational algorithms in hardware. String matching

was chosen as one of the areas to be tested as such algorithms typically involve many hardware manipulations of words of binary data. These manipulations are invoked by the machine code instructions, which constitute the program and performed by the general-purpose hardware within the microprocessor itself. So called software to hardware synthesis techniques aim to accelerate algorithm execution by first of all removing the need for machine instructions and by also performing computational and logical operations on bespoke hardware.

```
while (match != 0 && word_count != 0) {
      result = current & mask;
      match  = result - target;
      if (match != 0) {
         current = current >> 2;
         temp = buffer << 14;
         current = current | temp;
         if (shifted == 7) {
            word_count--;
            shifted = 0;
            buffer = *ptr;
            ptr++;
         }
         else {
            buffer = buffer >> 2;
            shifted++;
         }
      }
   }
```

Figure 10.1: *The C code for searching for occurrences of a single pattern in a given text*

The example code shown works on a word size of 16 bits and can detect a pattern of up to 8 DNA characters in length. However, the algorithm is by no means limited to this word size.

The algorithm works by shifting the input stream through the variable *current*. When the data is shifted, it is shifted two bits at a time to the right. It is shifted two

bits at a time because this is more efficient as the algorithm are searching for DNA features which are encoded into two bit patterns. Each time *current* is shifted to the right it is checked for a match with the target pattern. This concept is illustrated in Figure 10.2.

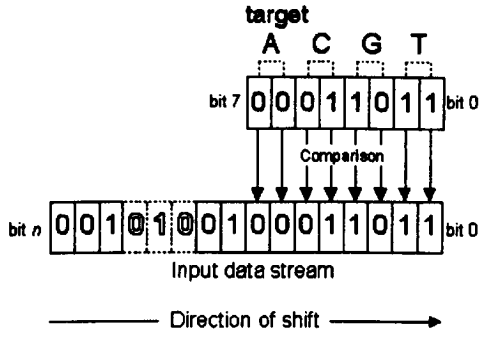

Figure 10.2: *Comparison of input stream against target*

When shifted, the two least significant bits (LSBs), which are bits 1 and 0, are lost and the two most significant bits (MSBs), which are bits 15 and 14, become empty. These two null MSBs are filled with the two LSBs of *buffer*. The variable *buffer* is a pre-fetch word, which will contain word i+1 with *current* containing word i. This is necessary if *current* is to kept full at all times. During initialisation, the first word of data is copied to *current* from the input buffer and *buffer* is filled with the second word of data.

In order to copy the two LSBs of *buffer* to the two MSBs of *current*, *buffer* is first copied to a variable *temp*, which is then shifted 14 bits to the left. This shift operation results in the two least significant bits of buffer (1 and 0) being moved to the two most significant bits (15 and 14), with the remainder of the word (bits 13 to

0) being filled with 0's. The contents of *temp* is then ORed with *current* resulting in the two most significant bits of *current* being replaced with the two least significant bits of *buffer*.

In order to make sure that *buffer* always has at least two bits available for *current*, a count is kept of how many times *current* has been shifted to the right. This count is stored in the variable *shifted*, which is initialised to 0 and then incremented each time *shifted* is shifted to the right and the two MSBs replaced with the two LSBs of *buffer*. If after a comparison *shifted* is less than 7, then *buffer* is shifted two bits to the right in order to replace the two LSBs which have been moved to *current* and the variable *shifted* is incremented. If shifted reaches 7, then the last two bits of data have moved from *buffer* to *current* and *buffer* requires re-filling. When this occurs, *shifted* is set back to 0 and *buffer* is loaded with a complete new word from the input stream.

The next byte to be fetched from the input stream is pointed to by the pointer variable *ptr*, which is incremented once *buffer* has been refilled with a word from data buffer named *data_buffer*.

To ascertain whether *current* contains a match for the bit pattern being searched for, *current* is first ANDed with a variable named *mask*. The purpose of *mask* is to mask out those bits of *current* which are not required for the comparison. To ignore a bit during the comparison between *target* and *current*, then the associated bit of

mask should be 0. Likewise, to include a bit in the comparison, then that bit of the mask should be set to 1. As illustrated in Figure 10.3 below, the pattern 'ACGT' is being searched for, which is only an 8 bit pattern. Hence the remaining upper eight bits can be ignored during the comparison and are thus set to 0.

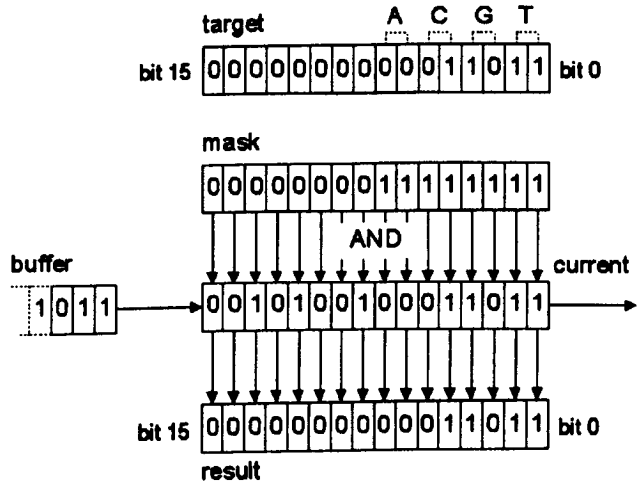

Figure 10.3: *The use of the mask to reduce the number of bits compared*

When *current* is ANDed with the mask, the result of the logical AND is stored in *result.* A bit of *result* will only be set to 1 if both the corresponding bits of *mask* and *current* are 1, otherwise the bit will be set to 0. A match with the target can now be determined by subtracting *target* from *result.* If the result of this subtraction is all 0's, then both result and the target must have contained the same values and hence a match has been found. This process is illustrated in Figure 10.4.

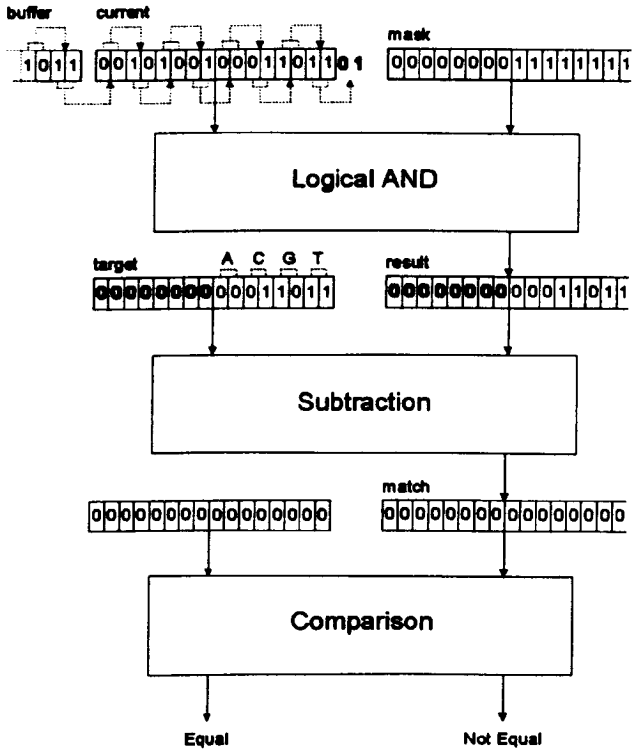

Figure 10.4: *The steps required to determine whether the target matches the current data*

The program has been written to locate patterns of DNA up to and including eight two bit codes. Hence, all words are 16bits in length and are declared as being of type *unsigned short*. However, the program could easily be amended to locate longer patterns by simply changing the variable types and program constants.

10.3 Searching for multiple strings

The example algorithm illustrated in Figure 10.1, simply searches an input stream for all occurrences of a single string. The program can be easily modified to search an input stream for all occurrences of many strings by reading in many targets from

a file and storing them in an array. This way, each time *current* is shifted, it may be compared with many targets before it is once more shifted. In order to do this, a second array must be created to store the masks for each of the targets. These masks may be automatically generated from the targets as they are read in.

```
while (shifts>0) {
      for (i=0; i<no_of_targets; i++) {
         result = current & mask_array[i];
         match  = result - target_array[i];
         if (match == 0) {
          .. match found
         }
      }

      current = current >> 2;
      shifts--;
      temp = buffer << 14;
      current = current | temp;

      if (shifted == 7) {
         shifted = 0;
         buffer = *ptr;
         ptr++;
      }
      else {
         buff = buff >> 2;
         shifted++;
      }
  }
```

Figure 10.5: *An algorithm to search for multiple patterns in a single text*

Apart from this simple modification, the program remains relatively unchanged. This is the version of the program, which will be the subject of the investigation into hardware acceleration of string matching.

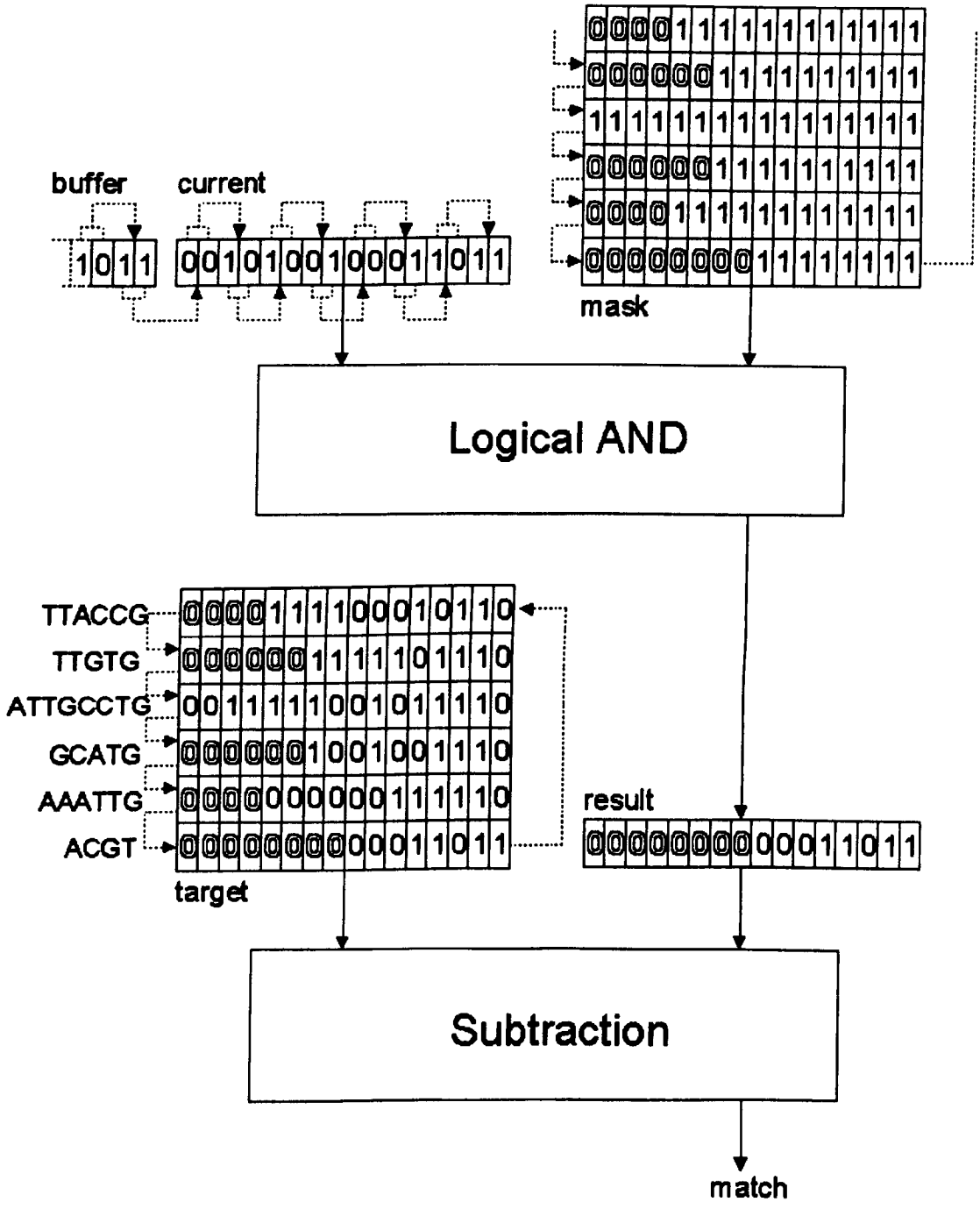

Figure 10.6: *Illustration of Figure 10.5*

10.4 Hardware acceleration

Over the past decade hardware synthesis has been explored as a method of accelerating computing tasks at which conventional general-purpose microprocessors are inefficient. The problem is that current microprocessors, although being suitable for many tasks, are not particularly efficient at performing any one task. This is because they are designed to be applicable to as many problem areas or tasks as possible. Therefore, through necessity they possess many features which although utilised by one application may never be used by another application. Another problem with conventional machines is the stored program concept whereby and algorithm is executed by the microprocessor obeying a series of commands, which are stored in memory. These commands are the machine code instructions, which the microprocessor fetches, decodes and then executes one at time. This fetching and decoding takes comparatively vast amount of time due to the slow speed of memory and the numerous instructions within the instruction set of the processor. Even the execution phase is by no means efficient. The execution circuits of a processor are finite and although some resources are replicated, many must be shared. This resource contention slows execution times. Additionally, the execution circuits of microprocessors are designed to perform many tasks, making them less efficient.

Hardware and software co-design or hardware to software synthesis is a process whereby computing algorithms expressed in high-level languages, are compiled to pro-

duce either an executable program and a hardware circuit design or just a hardware circuit. In the case of hardware and software co-design [107, 108], the majority of the program is turned into an executable binary for execution on a microprocessor, whilst the remainder of the algorithm is synthesised to hardware. The portion synthesised to hardware would be the section of the algorithm at which the microprocessor would be least efficient. The hardware portion is usually programmed into a Field Programmable Gate Array (FPGA) [137], which then acts as a co-processor to the host microprocessor. Producing programs for such architectures is usually performed using a hybrid programming language and appropriate compilers and synthesis tools [106]. Such programming languages tend to be based on C, with extensions being added to express the hardware-only components for the FPGA.

With pure software to hardware synthesis [10, 11], an attempt is made to map the entire algorithm into an FPGA, resulting in a digital circuit, which is functionally identical to and directly derived from an algorithm, which was originally expressed in a programming language. Such approaches tend to use hardware description languages such as VHDL [76], which are exclusively used for expressing the function of hardware circuits.

Synthesis to a hardware only solution offers the greatest potential increase in speed, removing the need for instructions and a conventional fetch-decode-execute cycle. However, it is also the most difficult to achieve. The difficulty arises from the

design features of current FPGAs, which were originally intended for implementing digital circuits. Although suitable for the prototyping and implementation of general circuits comprising of digital logic, they are not well suited for implementing algorithms. This is because algorithms require data storage for variables, buses for register to register and register to execution unit transfers. Data storage and buses are not available within an FPGA and must be created using the FPGAs resources, such as macro-cells and signal lines. What makes the situation worse is that both registers and buses are expensive in terms of FPGA resources, which ultimately limits the size of the algorithm that may be implemented in hardware.

As part of the research into implementing string matching algorithms in hardware-solutions, recommendations will be made regarding the development of a new FPGA architecture, which will be more suited to the purpose of implementing software in hardware.

10.5 Hardware Implementation of string Matching

The research currently being undertaken aims to overcome the limitations of current FPGAs, with regards to configurable computing. First of all, it aims to do this by recommending a new configurable device architecture, which lends itself more to the mapping of software to hardware. The device will feature the bussing systems, areas of storage and synchronization circuits required to facilitate both effective and

efficient hardware generation. Secondly, software tools are being developed which will process standard C programs and as their output, will produce configuration files for the programmable device.

Because of the low-level nature of the task of string matching, it is an ideal candidate for such acceleration techniques. At the hardware level, the most efficient method of searching a string for a sub-string is as illustrated in Figure 10.2. The stream to be searched is passed through a register, shifting one bit at a time. Each time the register is shifted, the register is compared with the sub-string being searched for. This is the same method as employed in the C algorithm discussed previously. The number of register bits to be compared need only be equal in length to the number of bits in the sub-string, with any additional bits simply being masked out or ignored in the same way as the C algorithm. Additionally, the register being searched need not only be shifted one bit at a time. In the case of searching for occurrences of bit patterns consisting of two bit sub-patterns, it is more efficient to shift the register two bits before a comparison with the target is made.

Figure 10.7, illustrates a simplified diagram of the components to be implemented in hardware. Missing are the hardware components responsible for shifting both *current* and *buffer* to the right. Also missing are the circuits required for synchronizing the activities of the components in order to perform the operations of the algorithm in the correct order.

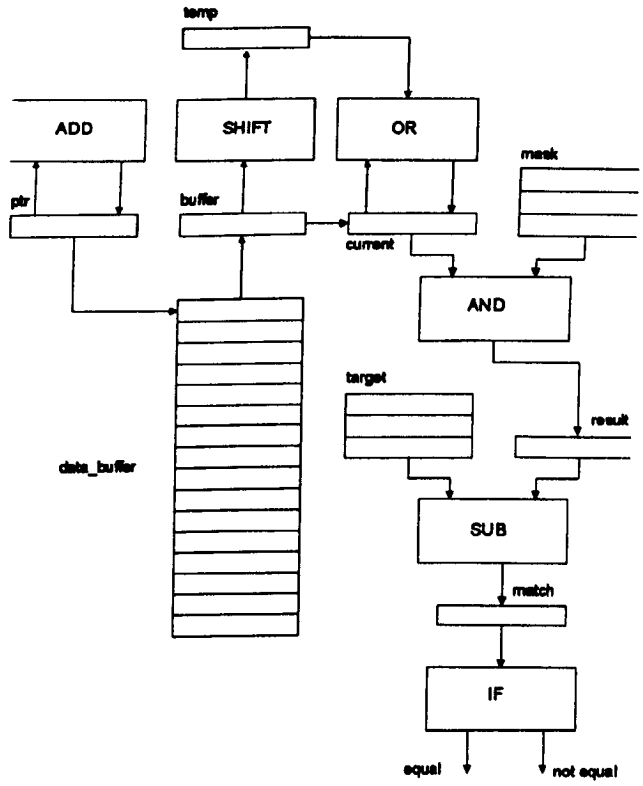

Figure 10.7: *Simplified version of the components to be implemented in hardware*

The memory labelled *data_buffer* holds the data to be searched for a sub-string. The width of the words contained in *data_buffer* is immaterial and may be of any width.

The registers labelled *ptr* and *buffer* are associated with the fetching of the words from memory. The register *buffer* is the same width as the words contained in *data_buffer*. This register is used to contain a pre-fetch word. The register *ptr* is used as a pointer to reference the words contained in *data_buffer*. As such, its width need only be sufficient to reference all of the words in *data_buffer*.

The register *current* contains the current bit pattern to be matched against a

sub-string bit pattern. It is a shift register, with the data contained in the register being shifted right two bits at a time, with the two least significant bits being lost and the two most significant bits being replaced with the two least significant bits of buffer. This is the purpose of buffer, to keep *current* full of bits. Only once all the bits contained in *buffer* have been shifted into current, will new data be loaded into *buffer* from *data_buffer*.

As with the C algorithm, the mask register is used to contain a bit pattern to mask off the bits of *current*, which are not to be compared. When ANDed with the contents of *current*, then the resulting word is stored in the register *result*. It is the contents of *result*, which will be compared with the target to determine whether or not a matching bit pattern has been located. To ascertain whether the contents of *result* and *target* do match, *result* is subtracted from *target*. Again, if the result of the subtraction is zero, then a match has been located.

10.6 Conclusions

Using the storage method described in Section 10.2 we can store DNA text files in 25% of space required for the original DNA text file. Using algorithms such as the DS and BK algorithm we can keep DNA texts efficiently stored and perform searches on them. Thus saving both time and space.

The hardware synthesis of the BK algorithm into a hardware only implementation

is expected to produce a solution that we estimate to be significantly faster than even the DS algorithm.

Chapter 11

Conclusions and Further Work

11.1 Applications of Algorithm Engineering

The topic of algorithms is a topic that is central to computer science. Measuring an algorithm's efficiency is important because your choice of algorithm for a given application often has a great impact. Although the efficient use of both time and space is important, inexpensive memory has reduced the significance of space efficiency. The main problem with memory has become the speed of the transfer of data from memory to the CPU. Thus we generally focus on time efficiency (see Chapter 2).

The efficiency of an algorithm can be measured both theoretically and practically. The theoretical evaluation of algorithms is usually performed using asymptotic analysis, which doesn't consider the constant factors that are hiding behind the Big-Oh notation. Asymptotic analysis doesn't show the breakpoint where an asymptotically slow algorithm with a small constant factor is faster than an asymptotically fast algorithm with a large constant factor. Also the asymptotic analysis focuses primarily

on worst-case inputs that may not be representative of the typical input for a certain problem. The algorithm may also be too complicated to allow us to effectively bound its performance. In such cases experimentation can often help us perform our algorithm analysis.

In performing an experiment we must decide what to measure. We can measure the actual time taken by the algorithm. This can be difficult as the time taken can be affected by other factors such as other programs running on the same computer, how the computer makes use of memory cache and whether or not there is enough memory available. In our experiments we used a standalone computer, which was only running the desired algorithm that is being evaluated.

An alternative approach is to count the number of times a basic operation is executed. This can be the number of comparisons taken as in our string matching algorithms. As found in our experiments this may give an indication of speed but an algorithm may take less basic operations but take more time to perform the same task.

The necessary step of coding up our algorithm correctly and efficiently involves a certain amount of programming skill. If we are comparing two algorithms against each other then we must be sure to code up the competing algorithm using the same level of skill as is used for our algorithm. The degree of code optimisation between the algorithms must be as close as possible, achieving a level playing field for the

algorithms being compared. Ultimately we should strive to make sure that our results are reproducible by other programmers with similar skills. We must also include the computer architecture of our chosen computer.

We have used Algorithm Engineering to modify and improve algorithms mainly in the field of string matching. Within this field there are many more string related problems and other fields, which would benefit from practical as well as theoretical evaluation. From the practical evaluation it may be possible to improve the algorithms and identify the features that determine the speed of the algorithms. Although the new algorithms may not improve their theoretical evaluations the practical gains should not be dismissed. The trend of improving the worst-case time complexity has delivered a number of groundbreaking algorithms. But no claims were made about their practical performance.

11.2 Algorithm Engineering and String processing

We have described the existing one-dimensional string matching algorithms and evaluated their performance both in theory and in practice. We found that the theoretical evaluation of each of the algorithms could be divided into two categories. Algorithms with a linear worst-case time complexity of $(O(n + m))$ which were mainly based on the Knuth-Morris-Pratt (KMP) algorithm. The remaining algorithms mainly had a quadratic worst-case time complexity of $O(nm)$ and were mainly based on the Boyer-

Moore algorithm.

Implementing the existing algorithms and counting the number of comparisons performed by the algorithms on a number of texts it was interesting to find that the Boyer-Moore based algorithms took less comparisons than the KMP based algorithm although their worst-case time complexities would indicate the opposite. This is due to the pattern being shifted to the right by a greater distance in the Boyer-Moore based algorithms. The KMP based algorithms reduce the number of times that a character in the text can be compared again to the text, although this feature doesn't reduce the number of comparisons much more than a Brute Force search of the text.

From the results of our evaluation we developed a new string matching algorithm, the BR algorithm, which is described in Chapter 4. The BR algorithm is the combination of two existing algorithms the Zhu-Takeda and the Quicksearch algorithm. We use the features of both algorithms that give them their speed. We have shown that the BR algorithm has a worst-case time complexity of $O(nm)$ and have proven that it has a linear average case time complexity. When the BR algorithm was compared to the existing algorithms in practice we found that the number of comparisons taken by the BR algorithm is fewer than those taken by any of the existing algorithms. Also the shift of the BR algorithm is greater than the existing algorithms. When timing each of the algorithms over a range of different English texts and using words from the UNIX English dictionary as patterns we found that the BR algorithm was faster

than the existing algorithms.

Two-dimensional string matching is considered in Chapter 5 and we describe a new algorithm, the 2D-BR algorithm. In one dimensional string matching we are only trying to match a single row of text. In two-dimensional string matching both the pattern and text are a matrix composed of rows and columns and for a full match we must match the entire pattern matrix with the text matrix.

When comparing the text matrix and pattern matrix, if we find that the current character to be compared is not in the pattern then no portion of the pattern matrix can overlap this mismatched character. If we compare a sample point of the text matrix with the elements in the pattern matrix then we can quickly tell whether the sample point is in the pattern matrix at any location. This will allow the use of sample points to search in a larger area than just the pattern matrix at an attempt. In fact we can check a $(2m_1 - 1) \times (2m_2 - 1)$ area to see if it possible for the pattern matrix to occur at that location (where m_1 and m_2 are the dimensions of the pattern matrix). This method could increase the size of the shift made although it requires an extra $\frac{n_1 \times n_2}{m_1 \times m_2}$ comparisons (where n_1 and n_2 are the dimensions of the text matrix). It was found that the lower the probability of a match between the sample point and the pattern the faster the algorithm ran. When the probability of a match between the pattern and sample point was high then the algorithm was slower than the existing algorithms. The 2D-BR algorithm is more efficient when the size of the alphabet is

large. We proved that the 2D-BR algorithm had a linear average case time complexity.

When performing the practical evaluation for our texts in Chapter 4, it was noticeable that the main portion of the time taken was used reading the text into memory. Reducing the size of the text would reduce the time taken to read the text into memory. The texts could not be changed and the data contained within the text must remain in the same order. The texts were reduced by efficiently storing the text. In Chapter 7 we consider DNA texts and show how they can be reduced to a quarter of their original size by assigning a 2 bit string to each of the 4 characters of the alphabet. This meant that 4 characters could be combined to form one byte (8 bits).

Once the text is efficiently stored an algorithm, the DS algorithm, was developed to search in the efficiently stored text. This was done so that a DNA text can be efficiently stored indefinitely and still search for a pattern. The DS algorithm has a worst-case time complexity of $O(nm)$ and was proven to have a linear average case time complexity of $O(n+m)$. For the practical evaluation we searched for enzyme cutting locations in DNA texts. The times recorded showed that the DS algorithm was roughly 3 times or more faster than the existing algorithms. The increase in speed was due to the reduction in the size of the original text. Others have also made savings by compressing the text and searching in the compressed text.

The DS algorithm was expanded to consider alphabets of any size ≤ 128 and is described in Chapter 8. Each character in an alphabet would take $\lceil \log_2 \sigma \rceil$ bits per

character where σ is the size of the alphabet. A new algorithm was developed that searched on the efficiently stored text. The algorithm also made use of the fact that a character that required x bits in a pattern then the pattern may only start every x bits in the text.

The algorithm has a worst-case time complexity of $O(nm)$ and was proven to have a linear average case time complexity of $O(n + m)$. In practice the algorithm was found to be fast when σ was small ≤ 32. This is due to the reduction in the size of the text not being as important a factor as the speed of the algorithm that is performing the search.

When performing a search if the number of comparisons and attempts made could be reduced then the time taken could also be reduced. In Chapter 9 we record the positions of characters or strings of characters in the text. This allows us to make attempts where one or more of the characters of the pattern have matched the text. This reduces the amount time taken to search for a pattern. The algorithm has a worst-case time complexity of $O(nm)$ and was proven to have a linear average case time complexity of $O(n + m)$.

In practice we found that the algorithm needed to perform a larger amount of pre-processing than the existing algorithms. The actual search for a pattern was many times faster than the existing algorithm due to the amount of text characters not considered during a search. Although to make this algorithm efficient we must

search for a number of patterns to offset the amount of pre-processing required by this algorithm. The number of patterns that are required for the algorithm to become efficient are also dependent on the size of the text being searched. We found that the larger the text the fewer the number of patterns required to be searched for the algorithm to become efficient. We adapted the algorithm to search in DNA files using a mapping function.

In Chapter 10 we investigated the feasibility of performing computational algorithms in hardware. String matching was chosen as one of the areas to be tested as such algorithms typically involve many hardware manipulations of words of binary data. These manipulations are invoked by the machine code instructions, which constitute the program and performed by the general-purpose hardware within the microprocessor itself. So called software to hardware synthesis techniques aim to accelerate algorithm execution by first of all removing the need for machine instructions. We expect to produce a solution that we estimate to be significantly faster than even the DS algorithm.

From the work undertaken in this thesis we have devised a number of novel new algorithms that solve a number of distinct string matching problems. The new algorithms have been theoretically and practically evaluated and compared to the existing algorithms. An average analysis has been conducted of the new algorithms and where performed they have all been found to have a linear average case time complexity.

Through the implementation of theoretical ideas and algorithms and practical evaluation of the ideas and algorithms it is possible to develop improved algorithms. Considerable experimentation and fine-tuning is typically required to get the most out of a theoretical idea. Algorithms and ideas discovered by the theoretical community should be implemented, tested and refined to the point where they can be usefully applied in practice.

We propose to create a software library of efficient string processing algorithms. These algorithms will enable many of the basic operations of string processing to be performed. The aim of the library is to provide a library of tools and data structures for managing sequences of symbols. The algorithms will have to be adaptable to different data types such as trees, arrays, multi-dimensional arrays and graphs. An architecture for such a library is described in [54].

11.3 Memory Management

A typical computer has several different levels of storage. Each level of storage has a different speed, cost, and size. These levels form a storage hierarchy, in which the topmost levels (those nearest the processor) are fastest, most expensive and smallest. The fastest type of storage is most typical main memory usually RAM (Random Access Memory) or ROM (Read Only Memory). Memory or storage is where data and instructions are stored ready for it to be transferred to the CPU (Computer

Processing Unit) for processing.

A processor also usually has its own memory in addition to any RAM connected to the computer and it is called cache memory. Cache memory is a small piece of fast, but more expensive memory, usually static memory used for copies of parts of main memory. It is automatically used by the processor for fast access to any data that currently contained in it. Access to the cache typically takes only a few processor clock cycles, whereas access to main memory may take tens or even hundreds of cycles.

Any cache uses a cache policy to decide which data to store. A cache policy is an attempt to predict the future, so that the cache will provide swift responses to future requests.

Cache policy may be implemented in hardware, software, or a combination of both. Some systems allow programs to influence cache policy, by giving hints or directions about future use of data. There are three main aspects of cache behaviour, which the cache policy can affect:

- Fetch policy - This determines which data is fetched into the cache, usually as a result of receiving a request for data that isn't cached.

- Eviction policy - This determines which data is discarded from the cache to provide space for newly fetched data.

- Write policy - This determines how and when modifications to cached data are synchronized with the underlying storage.

If the CPU requests the contents of a main memory location and the value of that location are held in some level of cache then the CPUs request is answered by the cache itself (a cache hit). Otherwise the request is answered by main memory (a cache miss). A cache hit has no penalty (1-3 cycles is typical) but a cache miss requires a main memory access and is therefore very expensive. To amortise the cost of a main memory access in the case of cache miss, an entire block of consecutive main memory locations containing the required request is brought into cache on a miss. A program that exhibits of accesses memory locations near those that it accessed previously will incur fewer cache misses and will consequently run faster.

In the experiments undertaken when calculating the time taken by an algorithm, cache memory was disabled so that it wouldn't affect the times recorded. By devising a new cache management policy it may be possible to improve the speed of string matching algorithms by storing more frequently requested data at the cache level. The design of algorithms could be modified so as to make more utilisation of cache memory.

11.4 Hardware implementation of string processing algorithms

We intend to fully implement the algorithm described in Chapter 10 into a hardware solution and practically evaluate the new solution. We expect the hardware implementation to be many times faster than the software implementation. From the results received from the experiments we expect to learn more about the features that are responsible for the increase in speed. We intend to use any information we learn from the experiments to improve the existing algorithms. The information learnt from this hardware implementation will allow us to improve other string matching algorithms.

Bibliography

[1] Abrahamson K., *"Generalized String Matching"*, SIAM Journal of Computing, 16(6), pp 1039 - 1051, 1989.

[2] Aho A. V., Corasick M. J., *"Efficient String Matching: An Aid to Bibliographic Search"*, Communications of the ACM, 18(6), pp 333-340, 1975.

[3] Akutsu T., *"Approximate String Matching with don't Care Characters"*, Information Processing Letters, 55(5), pp 235-239, 1995.

[4] Amersham life science products catalogue, pp 378-379, 1998.

[5] Amir A., *"Multidimensional Pattern Matching: A Survey"*, Georgia Tech, Technical Report TR 92/29, 1992.

[6] Amir A., Benson G., Farach M., *"An Alphabet Independent Approach to Two-Dimensional Pattern Matching"*, SIAM Journal of Computing, 23(2), pp 313-323, 1994.

[7] Amir A., Benson G., Farach M., *"Let Sleeping Files Lie: Pattern Matching*

in Z-Compressed Files", Journal of Computer and System Sciences, 52(2), pp 299-307, 1996.

[8] Apostolico A., Giancarlo R., *"The Boyer-Moore-Galil string searching strategies revisited"*, SIAM Journal of Computing, 15(1), pp 98-105, 1986.

[9] Arajo M., Navarro G., Ziviani N., *"Large Text Searching Allowing Errors"*, Proceedings of the 4^{th} South American Workshop on String Processing, (WSP'97), Chile, pp 2-20, 1997.

[10] Athanas P. M., O'Connor R. B., Peterson J. B., *"Scheduling and Partitioning ANSI-C Programs onto Multi-FPGA CCM Architectures"*, The Bradley Department of Electrical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, Virginia.

[11] Athanas P. M., Peterson J. B., *"High-Speed 2-D Convolution with a Custom Computing Machine"*, The Bradley Department of Electrical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, Virginia.

[12] Baeza-Yates R. A., *"Improved string searching"*, Software Practice and Experience, 19(3), pp 257-271, 1989.

[13] Baeza-Yates R. A., Rgnier M., *"Fast Algorithms for Two Dimensional and Multiple Pattern Matching (Preliminary Version)"*, Proceedings of the 2^{nd} Scan-

dinavian Workshop on Algorithm Theory, (SWAT '90), Norway, pp 332-347, 1990.

[14] Baeza-Yates R. A., *"Similarity in Two-Dimensional Strings"*, Proceedings of the Computing and Combinatorics Fourth International Conference, (COCOON 1998), pp 319-328, 1998.

[15] Baeza-Yates R. A., Navarro G., *"A Faster Algorithm for Approximate String Matching"*, Proceedings of Seventh Annual Symposium on Combinatorial Pattern Matching 1996, (CPM '96), pp 1-23, 1996.

[16] Baeza-Yates R. A., Navarro G., *"A Fast Heuristic for Approximate String Matching"*, Proceedings of 3^{rd} South American Workshop on String Processing (WSP'96), Brazil, (WSP '96). pp 47-63, 1996.

[17] Baeza-Yates R. A., Gonnet G. H., *"Fast String Matching with Mismatches"*, Information and Computation, 108(2), pp 187-199, 1994.

[18] Baeza-Yates R. A., Gonnet G. H., *"A New Approach to Text Searching"*, Communications of the ACM, 35(10), pp 74-82, 1992.

[19] Baeza-Yates R. A., Choffrut C., Gonnet G. H., *"On Boyer-Moore Automata"*, Algorithmica, 12(4/5), pp 268-292, 1994.

[20] Baeza-Yates R. A., *"A Unified View to String Matching Algorithms"*, Proceed-

ings of the 23^{rd} Seminar on Current Trends in Theory and Practice of Informatics (SOFSEM '96), Czech Republic, pp 1-15, 1996.

[21] Baeza-Yates R. A., *"Searching the Web: Challenges and Partial Solutions"*, Proceedings of String Processing and Information Retrieval, (SPIRE '98), Bolivia, pp 23-31, 1998.

[22] Baeza-Yates R. A., *"Improved String Searching"*, Software, Practice and Experience, 19(3), pp 257-271, 1989.

[23] Baeza-Yates R. A., Regnier M., *"Average Running Time of the Boyer-Moore-Horspool Algorithm"*, Theoretical Computer Science, (TCS), 92(1), pp 19-31, 1992.

[24] Baeza-Yates R. A., *"Text-Retrieval: Theory and Practice"*, Proceedings of the International Federation for Information Processing Congress, Spain, Vol. 1, pp 465-476, 1992.

[25] Baeza-Yates R. A., Gonnet G. H., *"Fast Text Searching for Regular Expressions or Automaton Searching on Tries"*, Journal of the ACM, 43(6), pp 915-936, 1996.

[26] Baeza-Yates R. A., *"Space-time trade-offs in text retrieval"*, Proceedings of 1^{st} South American Workshop on String Processing, (WSP '93), Brazil, pp 15-21, 1993.

[27] Baker T. P., *"A Technique for Extending Rapid Exact-Match String Matching to Arrays of More Than One Dimension"*, SIAM Journal of Computing, 7(4), pp 533-541, 1978.

[28] Barbosa E. F., Navarro G., Baeza-Yates R. A., Perleberg C. H., Ziviani N., *"Optimized Binary Search and Text Retrieval"*, Proceedings of the 3rd Annual European Symposium on Algorithms, (ESA '95), Greece, pp 311-326, 1995.

[29] Bell T. C., Cleary J. G., Witten I. H., *"Text compression"*, Prentice-Hall, 1990.

[30] Bell T. C., Kulp D., "Longest-match String Searching for Ziv-Lempel Compression", Software, Practice and Experience, 23(7), pp 757-771, 1993.

[31] Berry T., Ravindran S., *"A fast string matching algorithm and experimental results"*, Proceeding of the Prague Stringology Club Workshop, (PSCW '99), Czech Republic, pp 16-28, 1999.

[32] Berry T., Ravindran S., *"String matching in a compressed DNA text"*, Proceedings of the Australian Workshop on Combinatorial Algorithms, (AWOCA '99), Australia, pp 53-62, 1999.

[33] Berry T., Keller S., Ravindran S., *"Searching in an Efficiently Stored DNA Text using a Hardware Solution"*, Proceeding of the Prague Stringology Conference '01, (PSC '01), Czech Republic, pp 1-13, 2001.

[34] Berry T., Ravindran S., *"A Linear Time String Matching Algorithm on Average with Efficient Text Storage"*, Proceeding of the Prague Stringology Conference '01, (PSC '01), Czech Republic, pp 14-25, 2001.

[35] Bird R. S., *"Two Dimensional Pattern Matching"*, Information Processing Letters, 6(5), pp 168-170, 1977.

[36] Boyer R. S., Moore J. S., *"A fast string searching algorithm"*, Communications of the ACM, 23(5), pp 1075-1091, 1977.

[37] The Calgary Corpus available at:

`ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/`

[38] Charras C., Lecroq T., 1997, Exact string matching, available at:

`http://www-igm.univ-mlv.fr/~lecroq/string.ps`

[39] Charras C., Lecroq T., 1998, Exact string matching animation in JAVA available at:

`http://www-igm.univ-mlv.fr/~lecroq/string/`

[40] Lecroq T., Charras C., *"Exact string matching animation in JAVA"*, Proceedings of the Prague Stringology Club Workshop, (PSCW '98), Czech Republic, pp 36-43, 1998.

[41] Charras C., Lecroq T., Pehoushek J. D., *"A Very Fast String Matching Algo-*

rithm for Small Alphabeths and Long Patterns (Extended Abstract)", Proceeding of the Ninth Annual Conference on Combinatorial Pattern Matching, (CPM '98), USA, pp 55-64, 1998.

[42] Cypher R., Plaxton C. G., *"Deterministic sorting in nearly logarithmic time on the hypercube and related computers"*, Proceedings of the 22^{nd} ACM Symposium on the Theory of Computing, (STOC '99), pp 193-203, 1990.

[43] Collins English Dictionary, 3^{rd} edition.

[44] Colussi L., *"Correctness and efficiency of the pattern matching algorithms"*, Information Computing, 95(2), pp 225-251, 1991.

[45] Colussi L., Galil Z., Giancarlo R., *"On the Exact Complexity of String Matching"*, Extended Abstract, Proceedings of the 31^{st} IEEE Foundations of Computer Science, (FOCS '90), pp 135-144, 1990.

[46] Cormen T.H., Leiserson C.E., Rivest R.L., *"Introduction to Algorithms"*, MIT Press, 1990.

[47] Crochemore M., Rytter W., *"Text algorithms"*, Oxford University Press, 1994.

[48] Crochemore M., Czumaj, A., Gasieniec, L., Jarominek, S., LeCroq, T., Plandowki, W., Rytter, W., *"Speeding up two string matching algorithms"*, Algorithmica, 12(4/5), pp 247-267, 1994.

[49] Crochemore M., Iliopoulos C. S., Korda M., *"Two-Dimensional Prefix String Matching and Covering on Square Matrices"*, Algorithmica, 20(4), pp 353-373, 1998

[50] Crochemore M., Gasieniec L., Plandowski W., Rytter W., *"Two-Dimensional Pattern Matching in Linear Time and Small Space"*, Proceedings of the 12^{th} Symposium on Theoretical Aspects of Computer Science 1995, (STACS '95), Germany, pp 181-192, 1995.

[51] Crochemore M., Vrin R., *"On Compact Directed Acyclic Word Graphs"*, Structures in Logic and Computer Science, pp 192-211, 1997.

[52] Crochemore M., Lecroq T., *"Tight Bounds on the Complexity of the Apostolico-Giancarlo Algorithm"*, Information Processing Letters, 63(4), pp 195-203, 1997.

[53] Czumaj A., Galil Z., Gasieniec L., Park K., Plandowski W., *"Work-Time-Optimal Parallel Algorithms for String Problems"*, 27^{th} Annual ACM Symposium on Theory of Computing, (STOC '95), USA, pp 713-722, 1995.

[54] Czumaj A., Ferrgina P., Gasieniec L., Muthukrishnan S., Traff J. L., *"The Architecture of a Software Library for String Processing"*, Proceedings of the Workshop on Algorithm Engineering, (WAE '97), Italy, pp 166-176, 1997.

[55] Dermouche A., *"A Fast Algorithm for String Matching with Mismatches"*, Information Processing Letters, 55(2), pp 105-110, 1995.

[56] Diffie W., Hellman M. E., *"New directions in cryptography"*, IEEE transactions on information theory, 22(6), pp 644-653, 1976.

[57] Entrez database available at: `http://www.ncbi.nlm.nih.gov/Entrez/`

[58] Farach M., Thorup M., "String Matching in Lempel-Ziv Compressed Strings", Algorithmica, 20(4), pp 388-404, 1998.

[59] Friedrich R., Ottmann T., Schuierer S., *"Two-Dimensional String Matching for Non-Rectangular Patterns"*, Journal of Computing and Information, 1(1), Special Issue: Proceedings of the 6^{th} International Conference on Computing and Information, Trent University, pp. 247-262, 1994.

[60] Galil Z., Giancarlo R., *"On the Exact Complexity of String Matching: Upper Bounds"*, SIAM Journal of Computing, 21, pp 407-437, 1992.

[61] Galil Z., Park K., *"Truly Alphabet-Independent Two-Dimensional Pattern Matching"*, Proceedings of the 33^{rd} Symposium on Foundations of Computer Science 1992, (FOCS '92), pp 247-256, 1992.

[62] Galil Z., *"On Improving the Worse Case Running Time of the Boyer-Moore String Matching Algorithm"*, Communications of the ACM, 22(9), pp 505-508, 1979.

[63] Galil Z., *"Open problems in stringology"*, NATO ASI series - Combinatorial algorithms on words, Volume F12, 1985.

[64] Garey M. R., Johnson D. S., Freeman W. H., *"Computers and Intractability: A Guide to the Theory of NP-Completeness"*, 1979.

[65] Gasieniec L., Rytter W., "Almost Optimal Fully LZW-Compressed Pattern Matching", Proceedings of the Data Compression Conference, (DCC '99), USA, pp 316-325, 1999.

[66] Gasieniec L., Karpinski M., Plandowski W., Rytter W., *"Efficient Algorithms for Lempel-Zip Encoding (Extended Abstract)"*, Proceedings of the Fifth Scandinavian Workshop on Algorithm Theory, (SWAT '96), Iceland, pp 392-403, 1996

[67] Gasieniec L., Gibbons A., Rytter W., *"Efficiency of Fast Parallel Pattern Searching in Highly Compressed Texts"*, Proceedings of the The 24th International Symposium on Mathematical Foundations of Computer Science, (MFCS '99), Poland, pp 48-58, 1999.

[68] Hancart C., *"Analyse exacte et en moyenne d'algorithmes de recherche d'un motif dans un texte"*, These de doctorat de l'Universite de Paris 7, France, 1993.

[69] Hooker J. N., *"Needed: An empirical science of algorithms"*, Operations Research 42, pp 201-212, 1994.

[70] Horspool R. N., *"Practical fast searching in strings"*, Software Practice and Experience, 10(6), pp 501-506, 1980.

[71] Horspool R. N., *"Improving LZW"*, Proceedings of the Data Compression Conference, (DCC'91), USA, pp 332-341, 1991.

[72] Howard P. G., Vitter J. S., *"Parallel Lossless Image Compression Using Huffman and Arithmetic Coding"*, Information Processing Letters, 59, pp 65-73, 1996.

[73] Huffman D.A., *"A method for the construction of minimum redundancy codes"*, Proceedings of the Institute of Radio Engineers, 40, pp 1098-1101, 1951.

[74] Hume A., Sunday D., *"Fast String Searching"*, Software, Practice and Experience, 21(11), pp 1221-1248, 1991.

[75] Idury R. M., Schffer A. A., *"Multiple Matching of Rectangular Patterns"*, Information and Computation, 117(1), pp 78-90, 1995.

[76] *"IEEE Standard VHDL Language Reference Manual"*, The Institute of Electrical and Electronics Engineers, Inc. 1987.

[77] Johnson D. S., *"A Theoretician's Guide to the Experimental Analysis of Algorithms*, available at:

`http://www.research.att.com/~dsj/papers/exper.ps`

[78] Krkkinen J., Ukkonen E., *"Lempel-Ziv parsing and sublinear-size index structures for string matching (extended abstract)"*, Proceedings of the 3rd South American Workshop on String Processing, (WSP '96), Brazil, pp 141-155, 1996.

[79] Karmarkar N., *"A new polynomial time algorithm for linear programming"*, Combinatorica, Volume 4, pp 373-395, 1984.

[80] Karp R.M., Rabin M.O., *"Efficient randomized pattern-matching algorithms"*, IBM Journal of Research and Development, 31(2), pp 249-260, 1987.

[81] Kida T., Takeda M., Shinohara A., Miyazaki M., Arikawa S., *"Multiple Pattern Matching in LZW Compressed Text"*, Proceedings of the Data Compression Conference, (DCC '98), USA, pp 103-112, 1998.

[82] Kida T., Takeda M., Shinohara A., Arikawa S., *"Shift-And Approach to Pattern Matching in LZW Compressed Text"*, Proceedings of the 10th Annual Combinatorial Pattern Matching, (CPM '99), pp 1-13, UK, 1999.

[83] Kida T., Shibata Y., Takeda M., Shinohara A., Arikawa S., *"A Unifying Framework for Compressed Pattern Matching"*, Proceedings of String Processing and Information Retrieval, (SPIRE '99), Mexico, pp 89-96, 1999.

[84] Knuth D.E., Morris Jr. J.H., Pratt V.R., *"Fast Pattern Matching in Strings"*, SIAM Journal of Computing, 6(2), pp 323-350, 1977.

[85] Knuth D. E., *"Dynamic Huffman coding"*, Journal of Algorithms, 6(2), pp 163-180, 1985.

[86] Larmore L.L., Hirschberg D.S., *"A fast algorithm for optimal length-limited codes"*, Journal of the ACM, 37(3), pp 464-473, 1990.

[87] Lecroq T., *"Experimental Results on String Matching Algorithms"*, Software, Practice and Experience, 25(7), pp 727-765, 1995.

[88] Lecroq T., *"Experimental results on string matching over infinite alphabets"*, Report LIR 97.01, Universit de Rouen, 1997.

[89] Lelewer D.A., Hirschberg D.S., *"Data compression"*, Computing Surveys, 19(3), pp 261-297, 1987.

[90] Liu Z., Du X., Ishi N., *"An improved adaptive string searching algorithm"*, Software Practice and Experience, 28(2), pp 191-198, 1998.

[91] Loewenstern D., Yianilos P., *"Significantly lower entropy estimates for natural DNA sequences"*, Journal of Computational Biology, 6(1), 1999.

[92] Mahmoud H. M., Smythe R. T., Rgnier M., *"Analysis of Boyer-Moore-Horspool*

string-matching heuristic", Random Structures and Algorithms, 10(1-2), pp 169-186, 1997.

[93] Manber U.,*"A text compression scheme that allows fast searching directly in the compressed file"*, ACM Trans. on Information Systems, 15 (2), pp 124-136, 1997.

[94] Manber U., Wu S., *"An Algorithm for Approximate Membership checking with Application to Password Security"*, Information Processing Letters, 50(4), pp 191-197, 1994.

[95] Manber U., Smith M., Gopal B., *"WebGlimpse - Combining Browsing and Searching"*, Usenix '97 Annual Technical Conference, pp 195-206, 1997.

[96] Matias Y., Rajpoot N., Sahinalp S.C., *"Implementation and experimental evaluation of flexible parsing for dynamic dictionary based data compression"*, Proceedings of the 2^{nd} Workshop on Algorithm Engineering, (WAE '98), Germany, 1998.

[97] Melichar B., *"Approximate string matching by finite automata"*, Computer Analysis of Images and Patterns, Lecture Notes in Computer Science 970, pp 342-349, Springer-Verlag, Berlin, 1995.

[98] Morse S. F. B., 1844, available at:

`http://www.morsecode.com/`

[99] de Moura E. S., Navarro G., Ziviani N., Baeza-Yates R. A., *"Fast Searching on Compressed Text Allowing Errors"*, Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval 1998, (SIGIR '98), Australia, pp 298-306, 1998.

[100] de Moura E. S., Navarro G., Ziviani N., Baeza-Yates R. A., *"Fast and Flexible Word Searching on Compressed Text"*, ACM Transactions on Information Systems (TOIS), 18(2), pp 113-139, 2000.

[101] Muth R., Manber U., *"Approximate Multiple Strings Search"*, Proceedings of Seventh Annual Symposium on Combinatorial Pattern Matching, (CPM '96), pp 75-86, 1996.

[102] Navarro G., Raffinot M., *"A general practical approach to pattern matching over Ziv-Lempel compressed text"*, Proceedings of Combinatorial Pattern Matching, (CPM '99), Lecture Notes in Computer Science, 1645, pp 14-36, 1999.

[103] Navarro G., Tarhio J., "Boyer-Moore String Matching over Ziv-Lempel Compressed Text", Proceedings of the Combinatorial Pattern Matching, (CPM '00), Canada, pp 166-180, 2000.

[104] Navarro G., *"Improved approximate pattern matching on hypertext"*, Journal of Theoretical Computer Science, 237(1-2), pp 455-463, 2000.

[105] Nelson M., "LZW Data Compression", Dr. Dobb's Journal October, 1989 available at:

http://www.dogma.net/markn/articles/lzw/lzw.htm

[106] Page I., Luk W., *"Compiling occam into FPGAs"*, in FPGA, eds., Will Moore and Wayne Luk, 271-283, Abingdon EE&CS books, 1991.

[107] Page I., *"Constructing Hardware-Software Systems from a Single Description"*, Oxford University Computing Laboratory.

[108] Page I., Aubury M., Randall G., Saul J., Watts R., *"hcc: A Handel-C Compiler"*, Oxford University Computing Laboratory.

[109] Raita T., *"Tuning the Boyer-Moore-Horspool string searching algorithm"*, Software Practice and Experience, 22(10), pp 879-884, 1992.

[110] Reingold E. M., Urban K. J., Gries D., *"K-M-P String Matching Revisited"*, Information Processing Letters, 64(5), pp 217-223, 1997.

[111] Rivals ., Delgrange O., Delahaye J-P., Dauchet M., *"A First Step Towards Chromosome Analysis by Compression Algorithms"*, IEEE Symposium on Intelligence in Neural and Biological Systems (INBS), 1995.

[112] Rivals ., Dauchet M., Delahaye J-P., Delgrange O., *"Compression and genetic sequences analysis"*, Biochimie, vol. 78, pp 315-322, 1996.

[113] Rivals ., Delahaye J-P., Dauchet M., Delgrange O., *"A Guaranteed Compression Scheme for Repetitive DNA Sequences"*, abstract in the Proceedings of the 6^{th} Data Compression Conference IEEE Computer Science Press, (DCC '96), 1996.

[114] Rivest R. L., *"On the Worst-Case Behavior of String-Searching Algorithms"*, SIAM Journal of Computing, 6(4), pp 669-674, 1977.

[115] Rivest R. L., Shamir A., Adleman L., *"A method for obtaining digital signatures and public-key cryptosystems"*,Communications of the ACM, 21(2), pp 120-126, 1978.

[116] Sadakane K., *"An Improvement on Hash-Based Algorithms for Searching the Longest-Match String Used in LZ77-type Data Compression"*, IPSJ SIG Notes, 97-AL-56, 1997

[117] Shibata Y., Kida T., Fukamachi S., Takeda M., Shinohara A., Shinohara T., Arikawa S., *"Byte pair encoding: a text compression scheme that accelerates pattern matching"*, Technical Report DOI-TR-CS-161, 1999.

[118] Shibata Y., Takeda M., Shinohara A., Arikawa S., *"Pattern Matching in Text Compressed by Using Antidictionaries"*, Proceeding of the Combinatorial Pattern Matching, (CPM '99), UK, pp 37-49, 1999.

[119] Shibata Y., Matsumoto T., Takeda M., Shinohara A., Arikawa S., *"A Boyer-*

Moore Type Algorithm for Compressed Pattern Matching", Proceedings of the Combinatorial Pattern Matching, (CPM '00), Canada, pp 181-194, 2000.

[120] Schneider G. M., Bruell S. C., *"Concepts in data structures and software development"*, West publishing company,

[121] Practical Huffman coding by Michael Schindler at:
`http://www.compressconsult.com/huffman/`

[122] Simon I., *"String matching algorithms and automata"*, In First American Workshop on String Processing, Brazil, pp 151-157, 1993.

[123] Simon I., *"String matching algorithms and automata"*, In Results and Trends in Theoretical Computer Science, Austria, Lecture Notes in Computer Science 814, pp 386-395, Springer Verlag, 1994.

[124] Smit G. de V., *"A Comparison of Three String Matching Algorithms"*, Software Practice and Experience, 12(1), pp 57-66, 1982.

[125] Smith P. D., *"Experiments with a very fast substring search algorithm"*, Software Practice and Experience 21(10), pp 1065-1074,1991.

[126] Standish T. A., *"Data structure techniques"*, Addison Wesley, 1980.

[127] Stephen G. A., *"String Searching Algorithms"*, World Scientific, 1994.

[128] Stiffler J. J., *"Theory of synchronous communications"*, Prentice-Hall, 1970.

[129] Sunday D. M., *"A very fast substring search algorithm"*, Communications of the ACM, 33(8), pp 132-142, 1990.

[130] Takaoka T., *"An On-Line Pattern Matching Algorithm"*, Information Processing Letters, 22(6), pp 329-330, 1986.

[131] Tichy W. F., *"Should Computer Scientists Experiment More?"*, IEEE Computer, volume 31, pp 32-40, 1998.

[132] Vitter J. S., *"Design and Analysis of Dynamic Huffman Codes"*, Journal of the ACM, 34(4), pp 825-845, 1987.

[133] Vitter J. S., *"Algorithm 673 Dynamic Huffman Coding"*, ACM Transactions on Mathematical Software, 15(2), pp 158-167, 1989.

[134] Welch T. A., *"A technique for high-performance data compression"*, IEEE Computer, 17(6), pp 8-19, 1984.

[135] Wu S., Manber U., *"Fast Text Searching Allowing Errors"*, Communications of the ACM, 35(10), pp 83-91, 1992.

[136] Wu S., Manber U., *"A fast algorithm for multi-pattern searching"*, Technical Report TR-94-17, Department of Computer Science, University of Arizona, 1993.

[137] Xilinx Inc., *"Spartan and SpartanXL Families of Field Programmable Gate Arrays"*, Preliminary Product Specification, San Jose, CA, 1999.

[138] Zhu R.F., Takaoka T., *"On Improving the Average Case of the Boyer-Moore String Matching Algorithm"*, Journal of Information Processing, 10(3), pp 174 - 177, 1987.

[139] Zhu R.F., Takaoka T., *"A Technique for Two-Dimensional Pattern Matching"*, Communications of the ACM, 32(9), pp 1110-1120, 1989.

[140] Ziv J., Lempel A., *"A Universal Algorithm for Sequential Data Compression"*, IEEE Transactions on Information Theory, 23(3), pp 337-343, 1977.

[141] Ziv J., Lempel A., *"Compression of Individual Sequences via Variable-Rate Coding"*, IEEE Transactions on Information Theory, 24(5), pp 530-536, 1978.