MULTIPLE STRING PATTERN MATCHING ALGORITHMS

MAJOR PROJECT SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE AWARD OF DEGREE OF

Master of Technology

In

Information Systems

Submitted By:

Punit Kanuga

(2K12/ISY/21)

Under the Guidance of

Ms. Anamika Chauhan

(Asst. Prof., Department of IT)

DEPARTMENT OF INFORMATION TECHNOLOGY
DELHI TECHNOLOGICAL UNIVERSITY
(2013-2014)

ABSTRACT

Multiple string pattern matching is an approach to find all occurrences of a set of patterns in given text. With evolution of computation capacity along with storing capacity, we are in quest of analysing large data sets for search of information in terms of patterns. Efficiency of searching algorithms depends on development of an accurate & precise shift table. In case of a mismatch between text and pattern, shift table determines maximum length of part of text which can be skipped without missing any pattern match. However, with increase in size of data to be searched, there is a constant urge to reduce search time. Thus, we need a faster searching algorithm.

This study extends Boyer Moore concept to cultivate a new shift table algorithm which works on multiple variable length patterns and can cohesively be used with various searching techniques. Run-time complexity of the presented algorithm is O(N) where N denotes sum of lengths of all variable length patterns. This study also presents a new hashing based algorithm for fast search of multiple variable length patterns in large data sets. It can accommodate patterns which come up during search time. Furthermore, its speed enhances as the minimum pattern length P increases for data set of length n taking O(n/P) time during search.

Further, idea of clustering the pattern set prior to searching phase is proposed which experimentally speed up search time by factor of 4 in concerned case study. This case study consists of search set with more than 420,000 characters with number of patterns ranging from 50 to 100 and pattern length varying from 4 to 26. It also identifies various factors which effects searching time and establishes mathematical relationship among them.

ACKNOWLEDGEMENT

I take the opportunity to express my sincere gratitude to my project mentor Ms. Anamika Chauhan, Assistant Professor, Department of Information Technology, Delhi Technological University, Delhi for providing valuable guidance and constant encouragement throughout the project. It is my pleasure to record my sincere thanks towards her for her constructive criticism and insight without which the project would not have shaped as it has. I would like to thank her especially for constant confidence she had in me throughout the time.

I humbly extend my words of gratitude to other faculty members of this department for providing their valuable help and time whenever it was required.

Punit Kanuga

Roll No. 2K12/ISY/21

M.Tech (Information Systems)

E-mail: punitkanuga@gmail.com

CERTIFICATE

This is to certify that **Punit Kanuga (2K12/ISY/21)** has carried out the major project titled **"Multiple String Pattern Matching Algorithms"** in partial fulfillment of the requirements for the award of Master of Technology degree in Information Systems by **Delhi Technological University**.

The major project is a bona fide piece of work carried out and completed under my supervision and guidance during the academic session **2012-2014**. To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other University / Institute for the award of any Degree or Diploma.

Ms. Anamika Chauhan
Assistant Professor
Department of Information Technology
Delhi Technological University
Delhi-110042

Table of Contents

Figures

Tables

Chapter 1

INTRODUCTION

1.1 Overview

Multiple string pattern matching is one of the key areas of research in field of data structures and algorithms. It focuses on finding existences of string set known as Patterns in a larger string or data set. Patterns which have to be searched may be of variable lengths and are expected to be searched simultaneously.

As technology advances, both data storing capacity and computation capability have evolved. Storing capability in case of string pattern matching takes into account two types. First is main memory or memory available to processor during computation. Second is storing capacity which can scale up to huge data saved in warehouses. This affects the length of string which needs to be searched as well as length and number of patterns which are to be searched. Computation capability increases as quality of processors have evolved. It effects response time, throughput and ability to implement complex data structures.

Advancement in both of these have now enabled us to analyze data like never before. For quick analysis of data to provide real time results there is a constant quest for development of algorithms which can search several patterns in large data sets in one go.

1.2 Motivation

Multiple string pattern algorithms can be used in variety of fields. Search is a tool which is inherently used in every sphere of life. With number of options and opportunities increasing like never before, traditional way of single specific keyword based search has evolved into complex search systems involving variety of attributes or features. Multiple keyword use in evident from

searching product's location from a large super market to matching complex DNA sequences [13], [14].

Development of a fast algorithm enables us to find patterns in large data sets (like data warehouses) which we couldn't imagine earlier [12]. Similarly, network intrusion detection system offers consistent monitoring and tracking of networks and computer systems to ensure security [8]. Patterns have to be matched for various intrusions and virus or malware program signature. Novel approaches to analyse & search data patterns are used in various other fields such as multi keyword ranked search over encrypted cloud data [11], peer to peer networks [10], search engines [9], palm print matching [22] and many more.

1.3 Present Work

Present work can be chiefly divided into four broad categories:

i. New shift table algorithm for multiple variable length string pattern matching
ii. New adaptive hashing based multiple variable length pattern search algorithm
iii. Cohesive use of clustering with adaptive hashing based algorithm
iv. Mathematical relations to determine theoretical search time speed up and comparisons with practical results

1.3.1 New Shift Table Algorithm

A shift table is a table which reflects maximum number of characters which can be skipped in text when mismatch happens while searching pattern in it ensuring that no possible match is missed. Overall, shift table reduces search time for searching patterns in text. Presented algorithm for shift table generation determines correlations among patterns depending on characters constituting them. Empirical results show it has better accuracy in comparison to existing algorithms in generating shift table for each character.

1.3.2 New Adaptive Hashing Based Search Algorithm

A new hashing based multiple string pattern matching algorithm is presented. . It rules out traditional way of generation of shift table for each character present in pattern. Instead of focusing on each character to generate shift table, it focuses on characteristics inherited in pattern by taking two consecutive characters. This enables it to concentrate on character as well as its neighborhood. Additionally, it is capable of adding new patterns as search proceeds and from point of insertion of new patterns, it can adapt to search them too.

1.3.3 Cohesive Clustering

Idea of clustering the pattern set on basis on length of patterns prior to performing new adaptive hashing based search in presented. It is observed that it significantly condenses the search time. Thus, this combination proves to be of substantial use when length of patterns within the pattern set varies by large scale.

1.3.4 Mathematical Relationships

Intense analysis of results obtained by various experiments has led to identification of various parameters that effect search time of algorithm. This inspired us to relate proportional dependency among parameters and successfully formulate mathematical relationships.

It helps to theoretically determine the speed up in search time in comparison to that taken by using non-clustering approach. Experimental results show that practical results are synchronous with respect to theoretical predictions hence proves accuracy of the mathematical relationship.

1.4 Thesis Organization

The remainder of the thesis is organized as follows: chapter two is Literature Review. It consists of summary of work related and used as background in presented work. Without efforts of those researchers, this work would not be possible. This section specially emphasizes on two concepts. First is Boyer Moore concept as it is used in carving new shift table algorithm. Second is hashing concept and its usage in earlier work. Chapter three defines the problem statement. In other words, in defines what exactly multiple string matching problem is. It also gives brief description

of terms common to this problem. Chapter four explains development of new shift table generation algorithm. It covers the concept behind it, the algorithm and the process by which it generates shift table. Process is explained clearly with use of an example. Chapter five explains development of new adaptive hashing based search algorithm. It covers its concept, various parameters, definitions and algorithms used. Searching mechanism is explained using an example. Chapter six describes how clustering is merged with newly developed search algorithm. It covers concepts involved along with new definitions and parameters to measure effectiveness of algorithm. It also determines relationships of various parameters with search time and help to deduce mathematical relationships. Chapter seven explains experimental setup used. It describes conditions under which results are measured so that recreation of same can be easily done. It covers technical as well as logical aspects of implementations. Useful of any experimental result depend on two things. Firstly, how clearly the result is recorded. Secondly, how well it is understood. Chapter eight records experimental results and further performs analysis on them. It determines correctness of developed mathematical relationships with practical results and help to understand nature of searching process and effect of clustering process. Additionally, it provides runtime analysis of various implementations. Chapter nine presents final conclusion of presented work. It covers advantages, bottlenecks and scope of future work.

Chapter 2

LITERATURE REVIEW

Work done in this field is immense and enormous both in terms of concepts and implementations. It is not practically possible to sum it all up here in few pages. However, best effort has been put in to categorize it highlighting the path breaking concepts.

2.1 Overview

Pattern matching has been one of the evolving fields in algorithmic arena. Initially oriented towards matching single pattern in less time and memory has now progressed to multiple pattern matching. As memory of systems increase, memory constraint for this algorithm has reduced opening window for faster algorithms using fast data structures.

Initial solution was recommended by Rabin Karp algorithm [2] along with its generalizations. Further KMP algorithm reduced the time taken for searching [3]. It did so by skipping the already matched characters. On the contrary, it pro-processed each character beforehand to determine degree of skipping it could provide during the actual searching phases. Boyer Moore provided a major breakthrough is this field when they presented their idea to match patterns from end in order to save time [1]. Two major concepts introduced by them were bad character and good suffix heuristics. All three combined enabled searching pattern with rapid pace due to large jumps in could take when a mismatch occurred. These scientists laid the foundation which enabled future work in this field.

Aho-Corasick proposed an algorithm in which they extended earlier work with automata theory and carved a process in which search time was independent of number of pattern to be searched [4]. This process reduced the runtime logarithmically. Comments Walter progressed the work by using Aho-Corasick algorithm [4] and Boyer Moore concepts [1]. Solution were proposed by Commentz Walter in 1979 [5] and Navarro, Raffinot in 2002 [6]. Both offered sub linear time solutions for single pattern matching. Shift table were majorly used in many of the above

mentioned works to identify amount of shift that could be taken in case of mismatch. Years later, Khancome and Boonjing proposed algorithms which used shift table algorithms are core to their searching techniques for multiple pattern matching [7],[15].

A variety of data structures have been used by different researchers in this development process either to store data in significantly sorted manner or to enable fast access to data. Ordered tree data structure or trie was used in Aho-Corasick algorithm for accommodating pattern in linear time [4]. Crochemore and Czumaj used this data structure for multiple pattern matching in 1993 and 1999 [16],[17]. Navarro and Raffinot used trie for flexible multiple string pattern matching in 2002 [6]. However, memory requirement for ordered tree data structure was large. It was improved to reduce search time but then search process became difficult [6],[18].

Patterns were also accommodated in form of bits. These algorithms were known as Bit-parallel based algorithms. However, it was restricted by computer word and additionally bit conversion took significant time [19]. Use of hash table is gaining popularity as it provides match result in order of O(1). Details of hashing concept are provided later.

2.2 Boyer Moore Concept

It is one of the initial path breaking concepts which changed traditional perspective towards the string pattern matching problem. The algorithm of Boyer and Moore compares the pattern with the text from right to left. If the text symbol that is compared with the rightmost pattern symbol does not occur in the pattern at all, then the pattern can be shifted by *m* positions behind this text symbol. The following example illustrates this situation.

Example:

```
0 1 2 3 4 5 6 7 8 9 ...
```

a B b a d a b a C B a

b A b a c

b a b A C

The first comparison d-c at position 4 produces a mismatch. The text symbol d does not occur in the pattern. Therefore, the pattern cannot match at any of the positions 0,…, 4, since all corresponding windows contain a d. The pattern can be shifted to position 5.

The best case for the Boyer-Moore algorithm is attained if at each attempt the first compared text symbol does not occur in the pattern. Then the algorithm requires only $O(n/m)$ comparisons.

2.2.1 Bad character heuristics

If the bad character, i.e. the text symbol that causes a mismatch, occurs somewhere else in the pattern, then this method can be applied. Then the pattern can be shifted so that it is aligned to this text symbol. The next example illustrates this situation.

Example:

```
0 1 2 3 4 5 6 7 8 9 ...
```

a b b a b a b a C B a

b a b a c

b a b a c

Comparison *b*-*c* causes a mismatch. Text symbol *b* occurs in the pattern at positions 0 and 2. The pattern can be shifted so that the rightmost *b* in the pattern is aligned to text symbol *b*.

2.2.2 Good suffix heuristics

Sometimes the bad character heuristics fails. In the following situation the comparison *a*-*b* causes a mismatch. An alignment of the rightmost occurrence of the pattern symbol *a* with the text symbol *a* would produce a negative shift. Instead, a shift by 1 would be possible. However, in this case it is better to derive the maximum possible shift distance from the structure of the pattern. This method is called good suffix heuristics.

Example:

```
0 1 2 3 4 5 6 7 8 9 ...
```

a b a a b a b a c B A

c a b a b

  c a b a b

The suffix *ab* has matched. The pattern can be shifted until the next occurrence of *ab* in the pattern is aligned to the text symbols *ab*, i.e. to position 2.

In the following situation the suffix *ab* has matched. There is no other occurrence of *ab* in the pattern. Therefore, the pattern can be shifted behind *ab*, i.e. to position 5.

Example:

```
0 1 2 3 4 5 6 7 8 9 ...
```

a b c a b a b a c B A

c b a a b

    c b a a B

In the following situation the suffix *bab* has matched. There is no other occurrence of *bab* in the pattern. But in this case the pattern can't be shifted to position 5 as before, but only to position 3, since a prefix of the pattern (*ab*) matches the end of *bab*. We refer to this situation as case 2 of the good suffix heuristics.

Example:

```
0 1 2 3 4 5 6 7 8 9 ...
```

a a b a b a b a c B A

a b b a b

a b b a b

The pattern is shifted by the longer of the two distances that are given by the bad character and the good suffix heuristics.

2.2.3 Searching Algorithm

The searching algorithm compares the symbols of the pattern from right to left with the text. After a complete match the pattern is shifted according to how much its widest border allows. After a mismatch the pattern is shifted by the maximum of the values given by the good-suffix and the bad-character heuristics.

2.2.4 Proposed Extension of Boyer Moore Concept

Boyer Moore algorithm has been used as a guide to carve shift table algorithm for multiple string pattern matching. Proposed idea to generate shift table for multiple string pattern matching is given in Figure 2.1. Atomically, it combines shift distances for each pattern calculated individually to construct shift table for more than one patterns.

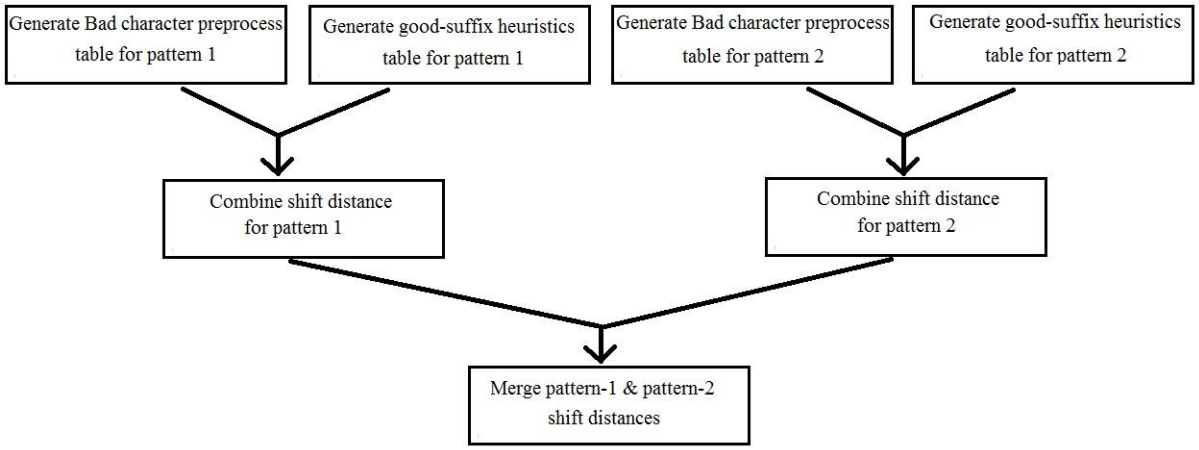

Figure 2.1: Extended Boyer Moore Algorithm

2.3 Hashing

Admiration of hashing expanded as it provides match result in order of O(1). It becomes more of use as size of string to be matched increases. It use in pattern matching was initiated when Rabin and Karp used it in searching phase [2]. Wu and Manber then formulated an algorithm for same purpose which was much quicker than other solutions present at that time [20]. Their idea was evolved and used in Network Intrusion Detection System [8]. Recently, Khancome and Boonjing have used hashing extensively to improve runtime efficiency of their algorithms in subsequent versions of their work [7],[15]. Three important hashing search algorithms proposed are given below to give idea of what it is capable of doing.

2.3.1 Suffix search

This algorithm works as following:

1. A Prefix-pattern (PPT) table in made. It holds prefix all the patterns.
2. Window is set from the first character of the text.
3. Last character of window is hashed into Shifting Table (ST).
4. If hash match is found, rest of pattern is hashed into PPT, else go to step 6.
5. If hash match if found, a hash match is recorded, else go to step 6.

6. Shift the window by shifting value of the current last character of the window.
7. Repeat the process till full text is exhausted.

Flowchart for Suffix search algorithm is shown in Figure 2.2.

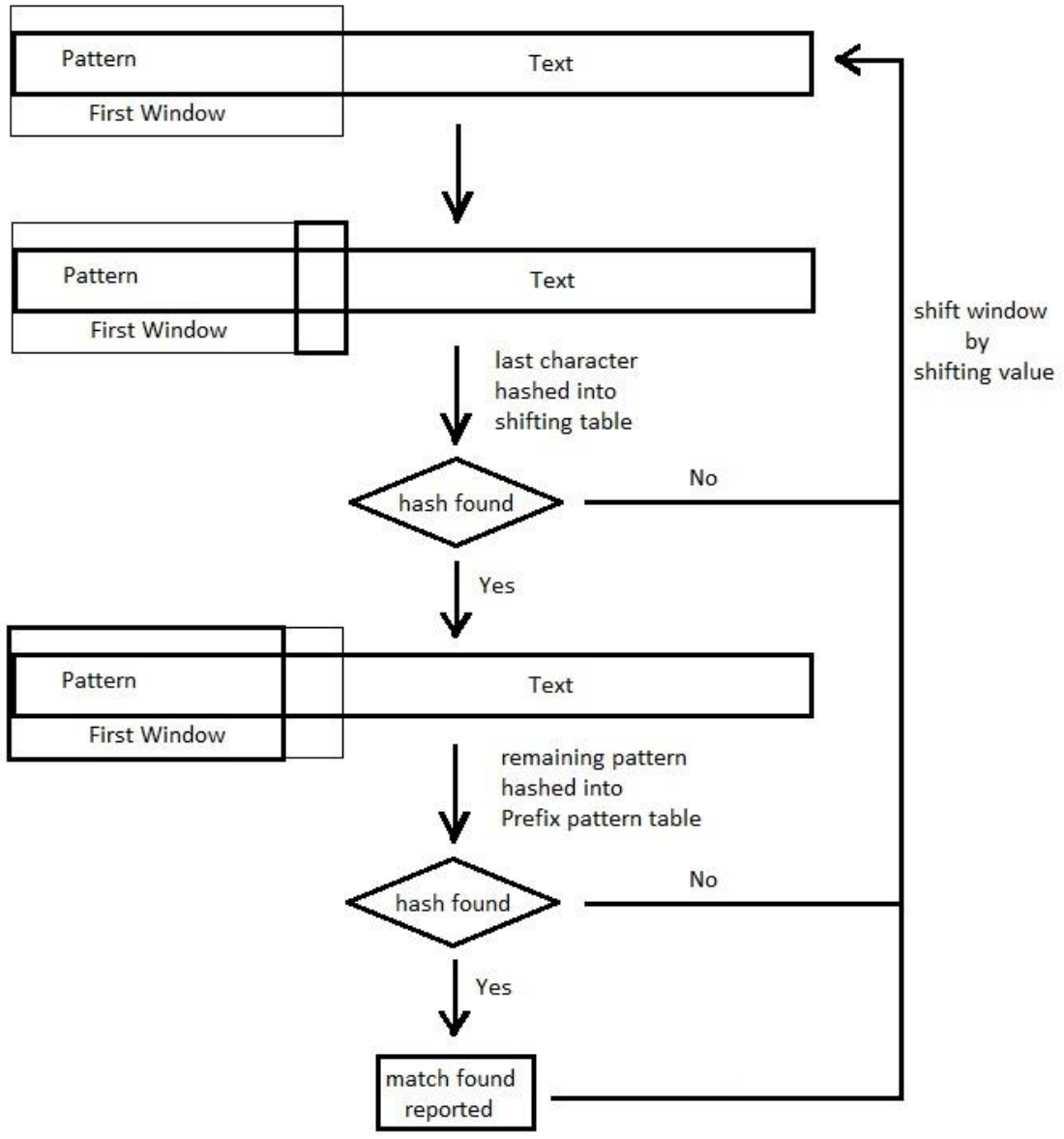

Figure 2.2: Flowchart for Suffix Search Algorithm

2.3.2 Suffix-prefix search

This algorithm works as following:

1. Suffix-prefix pattern (S-PPT) & Middle of pattern (MPT) table in made. They hold suffix-prefix & remaining part respectively of all the patterns.
2. Window is set from the first character of the text.
3. First & last character of window is hashed into S-PPT.
4. If hash match is found, rest of pattern is hashed into MPT, else go to step 6.
5. If hash match if found, a hash match is recorded, else go to step 6.
6. Shift the window by shifting value of the current last character of the window.
7. Repeat the process till full text is exhausted.

Flowchart for Suffix-prefix search algorithm is given in Figure 2.3.

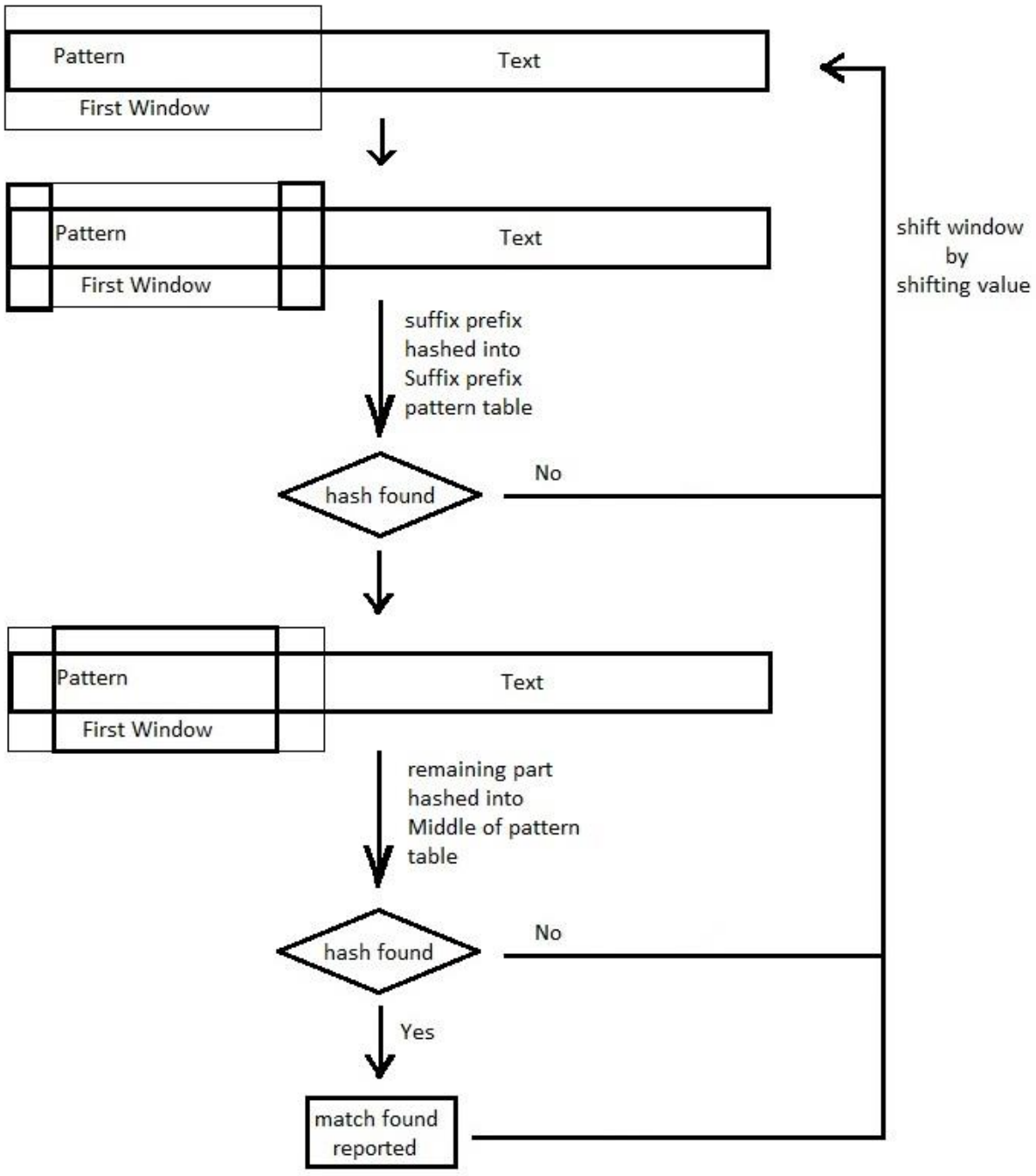

Figure 2.3: Flowchart for Suffix Prefix Search Algorithm

2.3.3 Suffix-middle-prefix search

This algorithm works as following:

1. Suffix middle prefix pattern (S-M-PPT) & before and after middle of pattern (BM-AMPT) table in made. They hold suffix, middle, prefix and part before & after middle respectively of all the patterns.
2. Window is set from the first character of the text.
3. First, middle & last character of window is hashed into S-M-PPT.
4. If hash match is found, before & after of middle pattern is hashed into BM-AMPT, else go to step 6.
5. If hash match if found, a hash match is recorded, else go to step 6.
6. Shift the window by shifting value of the current last character of the window.
7. Repeat the process till full text is exhausted.

Flowchart for Suffix-middle-prefix search algorithm is given in Figure 2.4.

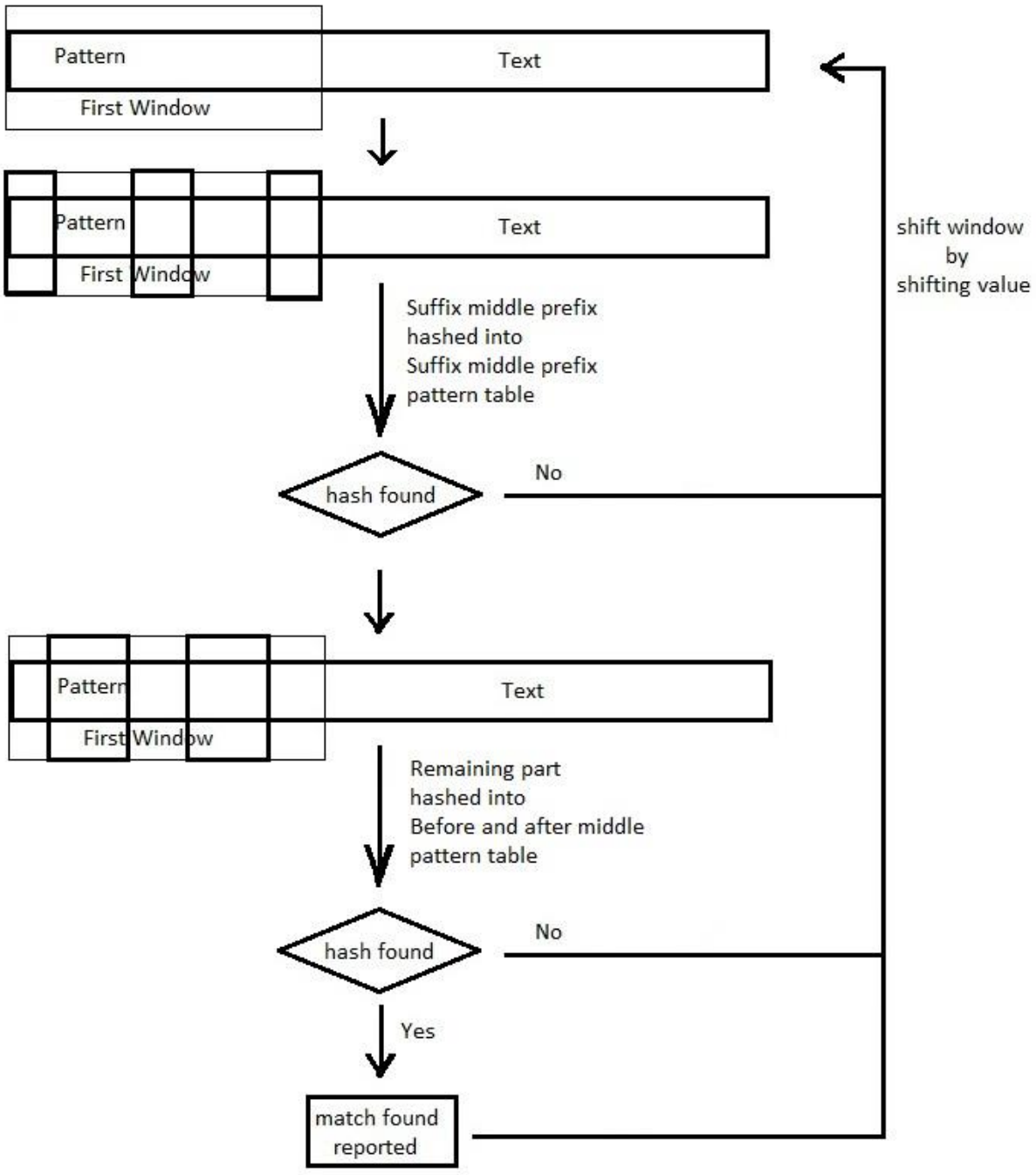

Figure 2.4: Flowchart for Suffix Middle Prefix Search Algorithm

Chapter 3

PROBLEM STATEMENT

Multiple strings pattern matching simultaneously searches for all occurrences of patterns $P = \{p1, p2, p3,..,pr\}$ which appeared in a given text $T=\{t1,t2,t3…tn\}$ over a finite alphabet set. This primarily requires the following:

i) *Set of patterns*: More than one data pattern which is required to be searched. A pattern, from the French word *patron*, is a type of theme of recurring events or objects, sometimes referred to as elements of a set of objects.

The elements of a pattern repeat in a predictable manner. Patterns can be based on a template or model which generates pattern elements, especially if the elements have enough in common for the underlying pattern to be inferred, in which case the things are said to *exhibit* the unique pattern. There are many different patterns in the world.

ii) *Text*: the data domain in which set of patterns have to be searched.

iii) *Window*: It is the part of text under consideration. In other words, it is the sub-set of text in which currently pattern is being searched. Size of window plays an important role in performance of searching techniques. It generally is more than the longest pattern which is expected to be searched.

iv) *Simultaneous searching capability*: It is the property of a searching procedure enabling it to search more than more data patterns at the same time. In other words, at every step in searching, it is open to find any of the data patterns which can be detected.

v) *Powerful search*: Powerful search is a search takes minimal attempting time. Its further speed primarily depends on rate at which data can be accessed & performance of algorithm being used.

vi) *Shifting mechanism*: It is the extent to which the window is to be shifted to further continue the searching procedure. It could be a shift value which has to precisely chosen depending on the characteristics of data patterns being searched so as to optimize both the searching time & number of matches. A shifting table is maintained which holds shifting value of each character. However, concept of shifting mechanism is not restricted to consideration of a single character at a time.

Chapter 4

PROPOSED METHODOLOGY: SHIFT TABLE ALGORITHM

This section contains thought and theory behind development of new shift table algorithm. It works well on patterns of variable lengths. It takes into account each character in every pattern and overall presents a shift table which could be used for searching pattern in given pattern set.

4.1 Concept

Let string which has to be searched be called *pattern*. Let string in which searching has to be done be called *text*. Character in *pattern* where matching fails be called *mismatch*. Part of *pattern* matched before *mismatch* be called *join* and part of pattern which remained to match after *mismatch* be called *leftover*. Thus, overall pattern will be sum of *leftover*, *mismatch* and *join*.

Bad character heuristics in Boyer Moore algorithm [1] decides the shifting of *text* with respect to *pattern* in case of mismatch. This is done by finding the rightmost occurrence of text's character which causes mismatch in pattern and aligning both of them. However, in case of dynamic text generation text is not available during the generation of shift table. Objective here is to develop a shift table which works well in searching phase whether text is available statically or it appears dynamically as in case of network intrusion detection systems.

Good suffix heuristics aims to search *join* in *leftover*. Best case would be finding a complete match of *join* in *leftover*. Alternatively, a sub-part of *join* can be searched in *leftover* starting from right end of *join*. Worst case is seen when even first character of *join* doesn't appear in *leftover*.

For multiple patterns, consider window of *len-1* length, where *len* is the length of smallest pattern to be searched. We process window from right to left direction to ensure minimum comparisons. We process each pattern one by one and calculate shift distance for each of the character. Maximum shift value for any character appearing in pattern would be *len*-1. We maintain the minimum shift value among all values for each character as we process each pattern. This shift value for each character guarantees the maximum possible shift with which we can skip *text* when a mismatch occurs due to that character.

We will consider every possible situation when a character at any position of window may cause a mismatch and we will calculate shift distance for each case. If mismatch occurs at last character of window with text, we will check possibility of match with the first character which is not same as last character of the window moving from right to left within window. In order to align text with that character of window, we consider shift value for mismatch at last character to be equal to distance between those two characters. In order words, we exhaustively calculate shift value for each character of *pattern* considering that it will cause mismatch with *text* during actual searching phase. Then, we will take minimum of possible set of shift values for each character. Here we assume all mismatching cases because we do not have *text* while generating shift table.

We take window of *len*-1 length instead of minimum length of pattern. Doing so ensures that in the worst case either beginning or end part of smallest *pattern* is considered as part of window. This guarantees finding of smallest *pattern* in the *text*. If we take window of length *len* and if we suppose that the last character of pattern creates mismatch and further never appears in leftover again then we would take shift value as *len*. This will rule out any matching of patterns which would start from that particular last character in next *len* number of characters. Specifically, it may lead to missing possible match of pattern of shortest length only. Thus, to avoid this problem, we take window length of *len*-1 and maximum shift distance for all character appearing in any pattern to be *len*-1. This is further explained in Example 1. Shift value for all characters which do

not appear in any pattern will be taken as *len* because there will be no possible match for such character and thus all combination of windows with this character can be solely avoided.

4.2 Shift Table Algorithm

Consider a shift table *shift* with two columns. First is the character column and second is the value column corresponding to each character. Let value column be initialized by length of smallest pattern (*l*).

Consider *Insert_shift(char, n)* function to enter value n corresponding to character *char* in shift table.

Consider *value(char)* function to give existing shift value of character *char* from shift table.

Consider *window* being the characters being considered in pattern.

Consider *windowlength* = length of smallest pattern-1

Consider matching starts from right to left position inside window, ie, character at last position of window is matched first, followed by last two characters combined and so on.

Consider *join* to be the part of pattern which is assumed to be matched with text till a mismatch occurs.

Consider *leftover* be the part of pattern which is still to be matched after a mismatch occurs.

Consider L to be length of pattern being processed.

Algorithm:

1. For each pattern
2. For 1 to (L - *windowlength* +1)
3. Consider *window* of *windowlength* characters starting from left of *pattern*
4. If (let) mismatch occurs at last character *char* of window
5. Insert_shift(char, distance with first different character)
6. Else if (let) mismatch occurs at any other character *char*
7. X = length of join
8. pos = position of mismatch + 1
9. while (X > 0)
10. find match of string[pos,window end] with any part of leftover moving from right to left
11. if match found
12. calculate Y
13. Where Y is the shift needed to align string with leftover
14. if(Y < value(char))
15. Insert_shift (char, Y)
16. Endif
17. Break out of while loop
18. else
19. X = X – 1
20. pos = pos + 1
21. End if
22. End while
23. end if
24. if(X = 0)
25. Insert_shift (char, *windowlength*)
26. End if
27. Move window by one position towards right till end of pattern is reached
28. End for
29. End for

4.3 Shift Table Construction

Consider set of patterns P {*aaba, aabab, aababc, aababcd, aababcde, abcb, zmnd, qope,jmqfm*}. Minimum length of pattern here is 4. Thus, size of window would be 3. Let us observe how processing of two of the above patterns is done.

Consider pattern *aababc*.

Window 1: aab

If mismatch happens at last character b, then we can shift this window by 1 towards right with respect to text to get nearest different character. Thus we consider shift table as {(b,1)}, ie, shift value of character b is 1.

If mismatch happen at middle character a, join = b and leftover is a. Here we are sure that last character that was matched in text was b. There is no match of any part of join in leftover. So now shift table would be {(a,3), (b,1)}.

If mismatch happens at first character a, join = ab and leftover is NULL. There is no match of any part of join in

leftover, neither ab or b alone. Now shift table remains as {(a,3), (b,1)}.

Window 2: aba

If mismatch happens at last character a, shift value for a will be 1 which is smaller than existing value for a in table. Thus, shift table will update as {(a,1), (b,1)}.

If mismatch happen at middle character b, join = a and leftover is a. Last matched character in text was a. Thus, we can shift pattern by 2 position to get align a (of join) with a (of text). But existing shift value for a is 1 which is less than 2. Thus, shift table remains as {(a,1), (b,1)}.

If mismatch happens at first character a, join = ba and leftover = NULL. There is no match of any part of join in leftover. So shift table remains as {(a,1), (b,1)}.

Window 3: bab

If mismatch occurs at last character b, shift will be 1. If mismatch occurs at a, shift will be 2 to align leftover b with matched text character b.

If mismatch occurs at first character b, join ab cannot be found. Further, join b cannot be found. So shift will be 3. Thus, shift table will be {(a,1), (b,1)}.

Window 4: abc

If mismatch occurs at last character c, shift will be 1. If mismatch occurs at b, join will be c and leftover will be a. We cannot find join in leftover. So shift value will be 3.

If mismatch occurs at a, join will be bc and leftover will be NULL. So shift value will be 3. Thus, shift table will be {(a,1), (b,1),(c,1)}.

Consider pattern *jmqfm*.

Window 1: jmq

If mismatch occurs at last character q, shift will be 1. If mismatch occurs at m, join will be q and leftover will be j. We cannot find join in leftover. So shift value will be 3.

If mismatch occurs at j, join will be mq and leftover will be NULL. So shift value will be 3. Thus, shift table will be {(q,1), (m,3),(j,3)}.

Window 2: mqf

If mismatch occurs at last character f, shift will be 1. If mismatch occurs at q, join will be f and leftover will be m. We cannot find join in leftover. So shift value will be 3. But this is greater than existing (q,1). So no change will be made in shift table.

If mismatch occurs at m, join will be qf and leftover will be NULL. So shift value will be 3. Thus, shift table will be {(q,1), (m,3),(j,3),(f,1)}.

Window 3: qfm

If mismatch occurs at last character m, shift will be 1. This is smaller than (m,3). So it will change. If mismatch occurs at f, join will be m and leftover will be q. We cannot find join in leftover. So shift value will be 3 but already lesser shift value is assigned.
If mismatch occurs at q, join will be fm and leftover will be NULL. So shift value will be 3. Thus, shift table will be {(q,1), (m,1),(j,3),(f,1)}.
This illustrates the calculation procedure for various conditions that appear during the process. Shift table for complete pattern set P can be calculated in similar manner.

Example 1: This example demonstrates need of taking window length equal to one short of smallest pattern length.
Let there be a pattern *zmnd* and z appears only here in this pattern together will all other patterns. Let shift value of z be 4 (assuming we take maximum shift distance equal to *l* instead of *l-1*). Let there be a text (…aabab*cdez*mndjmq…) with current window consisting of *cdez*. Shift value for z is 4. Thus, next window will be *mndj*. Thus, we can see that pattern *zmnd* is missed. Thus, we take maximum shift value for any character appearing across all patterns to be *l-1*.

Chapter 5

PROPOSED METHODOLOGY: ADAPTIVE HASHING BASED SEARCH ALGORITHM

This section contains thought and theory behind development of new adaptive hashing based search algorithm. It works well on variable length patterns and also works well if new patterns appear to be added into search at runtime.

5.1 Concept

Let the string in which patterns are to be search be called *BaseString*. This process first creates a Match Table containing hash value of various unique consecutive character pairs across all patterns. Let this consecutive character pair be called *ConPair*. Along with the hash values, Match Table will contain one or more set of following four values depending on number of matches of *ConPair* found in patterns:

i) Offset of pattern in which *ConPair* is found. Each pattern will be uniquely identified by its offset number.

ii) Position of last character of *ConPair* (Pos).

iii) Distance of last character of *ConPair* from left extremity of pattern (LExt).

iv) Distance of last character of *ConPair* from right extremity of pattern (RExt).

It must be noted that length of pattern at Offset is equal to Pos+RExt+1 in that set value. Here we are considering first character of pattern to be at zero position.

After creation of Match Table we will consider a window of p-1 characters of Base String starting from first character where p is the length of smallest pattern. We will match hash of last two characters of each window with the Match Table. If a match (or multiple matches) is found we will process corresponding sets of values present in table for each match one by one before

proceeding further. The processing is explained later in this section. After processing of all possible matches, we will move to next window of p-1 characters till Base String terminates.

Processing of corresponding set value for each match will be as follows:

i) Check validity of LPos of set value. LPos is start position of possible match. This is calculated using current position of considered character in Base String (CurPos) and LExt. It is equal to CurPos - LExt. Each LPos must be larger than or equal to the minimum possible value which is 0. We call this minimum value LPosMin. Otherwise processing for this set value is over.

ii) Check validity of RPos of set value. RPos is the end position of possible match. This is calculated as CurPos + RExt. Each RPos must be smaller than or equal to the maximum value which is length L of Base String. We call this maximum value RPosMax. Otherwise processing for this set value is over.

iii) Check to remove redundant entry of same match. This is explained in Redundancy Check algorithm. If a match already exists processing for this set value is over.

iv) Match hash value of Base String from LPos to RPos with hash value of pattern associated with the offset value. If they match a pattern is found. Otherwise processing of this set value is over.

v) If match is found its entry has to be made in Master Record structure. This will keep track of all found matches. This helps in step iii.

Master Record is a tabular structure where each entry contains a pair of values. These will be the start and end position in Base String where a match is found. For instance, entry (12, 16) signifies that a match is found from position 12 to 16 in Base String of length 5 (both values inclusive).

5.1.1 Window length

It is kept one short of minimum pattern length and not equal to minimum pattern length. This is because keeping it equal to later may lead to exclusion of possible matches for smallest pattern. This will happen when CurPos will point to start of minimum length pattern in Base String. Here we are considering only last two characters. And next window will start just after termination of smallest pattern. Both these ConPair will not consider any part of pattern. Thus, this pattern would get excluded. This is further explained in Example 2.

The above process explains how this algorithm works when we have complete pattern set before searching initiates. This is known as Offline or Static Search as pattern set is statically available. Suppose we have already searched a part of *BaseString* and we wish to include more patterns in search from that point. Here earlier generated Match Table needs to be only updated with *ConPairs* as per newly added patterns. Earlier Match Table constructed will still be of use. However, window length may decrease if any of the newly added patterns is smaller than all other existing pattern. This enables us to search added patterns from that point and provides adaptive nature to algorithm as pattern set updates at run time. Thus, it is called Online Search or Dynamic Search.

5.1.2 Hashing

Hashing is dominantly used in both pre-processing as well as search phase of proposed work. This is because match result for m length string can be calculated in O(1) time using hashing. It doesn't depend on length of m after hash value is calculated once. Hashing is used here in two areas. First is using hash values of *ConPairs* to find match of terminal *ConPair* present in window to those present in *Match Table*. Implementation is done in such a way that time taken to match hash value of *ConPair* will include time taken to search presence of that *ConPair* in *Match Table* as well. No additional time will be required. This has helped in reducing search time.

Second use is made in matching hash values of expected match in *Base String* with those present in *PHash* table. Here values of all patterns in pattern sets (or clusters) are calculated beforehand and saved in *PHash* table. Implementation of *PHash* table is ordered. Thus, here also we do not have to waste time in searching which entry of *PHash* table needs to be matched as each quadruplet already stores information regarding that. This also reduces search time further. We can conclude that use of hashing combined with way of implementation has led to significant reduction in search time.

5.2 Algorithms

This section contains the searching algorithm and its subsidiary algorithms.

Algorithm 1: Create PHash

This algorithm will create a table containing a pair of values. First is Hvalue which is hash value of each pattern. Second is Offset which maps pattern to its corresponding hash value. This means each pair (X,i) will signify that hash value of pattern number i is equal to X.

Input: Pattern set P {P1, P2, …., Pn}
Output: Pattern Hash Table (PHash)
1. Initiate the empty PHash
2. For i = 1 to n
3. PHash[i].Hvalue = Hash(P[i])
4. PHash[i].Offset = i
5. End For
6. Return PHash

Algorithm 2: Create MTable

This algorithm will create a table which will contain hash value of various uniquely possible ConPair across all patterns along with their Offset, Pos, LExt and RExt. We are using hash values to reduce the time taken to match ConPair during searching.

Input: Pattern set P {P1, P2, …., Pn}
Output: Match Table (MTable)
1. Initiate empty MTable
2. For i = 1 to n
3. For j = 0 to l-2 // where l is length of current pattern

4. Calculate val = Hash(j,j+1)

5. If val does not exist in MTable

6. Add new column in MTable

7. Add val in first column

8. Else go to column with val

9. Add set (Offset, Pos, LExt, RExt) in second column

10. End If

11. End For

12. End For

13. Return MTable

Algorithm 3: Redundancy Check

This algorithm search presence of value pair (LPos, RPos) in Master Record. It returns 1 when no match of value pair is found in Master Record indicating that first entry of this value should be made in Master Record. If value pair exists in Master Record it means that this match is already found in Base String thus it should not be further processed. This case comes up when a pattern is found in Base String which is substantially larger than the minimum length pattern.

Input: ValuePair(LPos, RPos)

Output: 0/1

1. Initiate flag to 1

2. While (MasterRecord exhauts)

3. If(MasterRecord.entry = ValuePair)

4. Return flag-1

5. End If

6. MasterRecord.get_Next_Entry

7. End While

8. Return flag

Algorithm 4: Search

This algorithm will search BaseString for all patterns. It will return MasterRecord.

Input: Base String, PHash, MTable

Output: MasterRecord

Set variables:

Lmin = length of smallest pattern

CurPos = Lmin -2

LPosMin = 0

RPosMax = L-1

Assuming first character to be at position 0

1. While (CurPos <= RPosMax)
2. If (Hash(CurPos, CurPos+1) found in MTable)
3. For all set values of Hash(CurPos, CurPos+1)
4. LPos = CurPos - LExt
5. If (LPos < LPosMin)
6. Goto next set value
7. End If
8. RPos = CurPos + RExt
9. If (RPos > RPosMax)
10. Goto next set value
11. End If
12. If (Redundancy Check(LPos, RPos))
13. If (Hash(BaseString, LPos, RPos) = PHash[Offset])
14. Add (LPos, RPos) in MasterRecord
15. End If
16. End If

17. Goto next set value
18. End For
19. End If
20. CurPos = CurPos + Lmin – 1
21. End While
22. Return MasterRecord

5.3 Searching

Example 1: This example shows searching of pattern set in given BaseString.

Consider set of pattern P {scare, care,arch}. Here P[0] is {scare}, P[1] is {care}, P[2] is {arch}.

Consider Base String (BS) of length 24 shown in Table 5.1.

Table 5.1: Base String for Example 1

0	1	2	3	4	5	6	7	8	9	10	11
a	r	e	s	c	a	r	e	h	s	t	a
12	13	14	15	16	17	18	19	20	21	22	23
r	c	h	s	r	a	r	c	h	s	c	a

We get Table 5.2 for Example 1 hash table PHash as per the algorithm 1.

Table 5.2: Hash Table for Example 1

Hvalue	Offset
Hash(scare)	0
Hash(scar)	1
Hash(arch)	2

Consider first pattern scare. Here first ConPair will be sc with c being the last character of ConPair. It belongs to P[0]. Thus, Offset will be 0. Position of c in P[0] is 1. Thus Pos is 1. Distance of c from left extremity and right extremity of P[0] is 1 and 3 respectively. Thus, LExt is 1 and RExt is 3. Thus, set value for this ConPair will be (0,1,1,3). Same ConPair is again found in P[1] with set value (1,1,1,2). Similarly generating set values for all ConPair will generate Table 5.3 MTable from Algorithm 2.

Table 5.3: Match Table for Example 1

ConPair	Set Value (Offset, Pos, LExt, RExt)
sc	(0,1,1,3), (1,1,1,2)
ca	(0,2,2,2), (1,2,2,1)
ar	(0,3,3,1),(1,3,3,0),(2,1,1,2)
re	(0,4,4,0)
rc	(2,2,21)
ch	(2,3,3,0)

Actual hash value should be stored in ConPair column of MTable. Here we are using character pair for ease of understanding. However, other implementations are also possible.

After creation of PHash and MTable, we start the search phase. MasterRecord is built as search phase proceeds. Algorithm 3 will be used for Redundancy Check in search phase.

Initiate the following variable as:

Lmin: 4, length of minimum pattern

CurPos: 2, current position of consideration in BS. It is one short of Lmin but here we are taking first character position as 0. So, effectively it equals 2.

LPosMin: 0

RPosMax: 23

L: 24, length of string

Consider the following notations:

P: Present in MTable(X)

X: Number of entries

NP: Not present in MTable

RCV: Redundancy Check Value

Step 1: CurPos = 2

ConPair (re): P (1)

(0,4,4,0)

LPos: 2-4 = -2

LPos < LPosMin

CurPos = 2 + 4 – 1

CurPos = 5

Step 2: CurPos = 5

ConPair (ca): P(2)

i) (0,2,2,2)

LPos: 3

RPos: 7

RCV(3,7) = 1

Hash(BS,3,7),PHash(0)

Hash(scare) = Hash(scare)

MasterRecord={(3,7)}

ii) (1,2,2,1)

LPos: 3

RPos: 6

RCV(3,6) = 1

Hash(scar)=Hash(scar)

MasterRecord={(3,7), (3,6)}

CurPos = 5 + 3

Step 3: CurPos = 8

ConPair (eh): NP

CurPos = 8 + 3

Step 4: CurPos = 11

ConPair (ta): NP

CurPos = 11 + 3

Step 5: CurPos = 14

ConPair (ch): P(1)

(2,3,3,0)

LPos: 11

RPos: 14

RCV(11,14) = 1

Hash(arch) = Hash(arch)

MasterRecord={(3,7), (3,6),(11,14)}

CurPos = 14 + 3

Step 6: CurPos = 17

ConPair (ra): NP

CurPos = 17 + 3

Step 7: CurPos = 20

ConPair (ch) : P(1)

LPos: 17

RPos: 20

RCV(17,20) = 1

Hash(arch) = Hash(arch)

MasterRecord={(3,7), (3,6),(11,14),(17,20)}

CurPos = 20 + 3

Step 8: CurPos = 23

ConPair (ca): P(2)

i) (0,2,2,2)

LPos: 21

RPos: 25

ii) (1,2,2,1)

LPos: 21

RPos: 24

CurPos = 23 + 3

CurPos = 26 > RPosMax

Searching Over

Example 2: This example shows need of taking window size equal to one short of minimum pattern length.

Consider ear to be one of the pattern and minimum pattern length be 3. Let window size should be equal to minimum pattern length, 3. Consider Table 5.4. Part of BaseString is under consideration during of the intermediate steps with CurPos at 17. Suppose ConPairs {ea,ar} appears only in pattern ear.

Table 5.4: Table for Example 2

16	17	18	19	20
i	e	a	r	t

At position 17, ConPair under consideration will be (ie). After processing, next value of CurPos would be 17+3=20. At postion 20, ConPair (rt) will be considered. Thus, patter ear is missed during search phase and this happens because it is the smallest pattern and CurPos happened to be at its initial position. This lead to two ConPairs (ie,rt) none of which were part of pattern ear. Thus, to make sure pattern gets identified in such cases, we reduce window size to one short of smallest pattern. If this were the case, second ConPair under consideration would be (ar). Thus, pattern ear would get searched.

Chapter 6

PROPOSED APPROACH: COHESIVE CLUSTERING

This section contains thought and theory behind merging concept of clustering along with newly developed adaptive hashing based search algorithm. Basic definitions used are same as defined in previous chapter.

6.1 Clustering

Irrespective to the frequency of length of patterns appearing in pattern set P, window of searching phase is solely determined by length of smallest pattern only. Searching phase completes in order O(n/k) where k is the length of smallest pattern. Thus, we can deduce that smallest pattern determines the searching time of the entire process. Problem with this approach is even if there is only one pattern of comparatively smaller length in comparison to all other patterns then this small pattern alone determines the speed of search phase. To solve this problem, we propose that clusters should be made of patterns of near about same length prior to pre-processing phase itself. Then entire process should be carried out separately for each cluster. Finally, results can be merged. Here emphasis is more on acknowledging the need of generating clusters rather than the process followed to generate clusters. Several clustering algorithm are present which can be used for this task. Number of generated clusters will depend on frequency of lengths of various patterns of different as well as same length. Intention is to make sure range of pattern lengths within a cluster is not very drastic. Thus it enables us to achieve overall minimized searching time with respect to searching process that does not use clustering. Following gives the above proposed process step by step.

i. Generate clusters of patterns with nearly same length.

ii. Make *Match Table* for all clusters separately.

iii. Perform search process for each cluster separately.

iv. Combine result for several clusters to get final result.

Here we generate separate *Match Table* for each cluster because *Match Table* is generated by *ConPairs* and *ConPairs* are intrinsic property of pattern. Thus, each cluster of pattern may result in separate set of *ConPairs* which may or may not be present in other clusters.

6.2 Redundancy Function

Redundancy functions come across cases which can be categorised under three sections as shown in Figure 6.1. Section A are those cases when a new pattern is found at that place for the first time. These are the ordered values which are later added into *Master Record*. Section C consists of cases where hash value of expected pattern doesn't match with that of expected pattern. Thus, these are the failed expectations of patterns. Cases in Section B are the tricky ones. These are the patterns which are already found at the same place due to identification of pattern by a *ConPair* earlier in search. Thus, these patterns now need to be rejected to avoid duplicate entry in *Master Record* which would return in duplication of search result. This is case which motivated formulation of Redundancy Function. Thus, Redundancy function is calculated prior to hash value check so that time taken in calculating hash value for redundant entries can be saved.

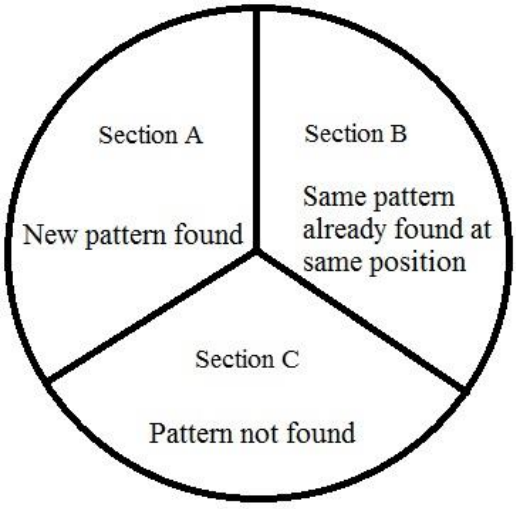

Figure 6.1: Sections of Redundancy Functions

6.3 Parameters Affecting Search Time

Advantage of performing clustering can be seen when length of patterns vary at high degree. Let suppose the pattern set P be divided into two clusters c1 and c2. Let smallest pattern length in P, c1 and c2 be l1, l1 and l2 respectively (because length of smallest pattern in P is same as that of c1). Let time taken to search *Base String* is T1, T2, T3 in case of pattern set P, c1 and c2 respectively. Then, total search time T2+T3 turns out to be drastically lesser than T1 alone. This is shown in Experimental Analysis section.

One may argue that time taken by combining both searches should be larger because entire *Base String* is traversed separately in both cases. However, we have to also consider the fact that different *Match Table* is created for both clusters individually. Thus, number of *ConPairs* generated as well as number of entries per *ConPair* will be drastically reduced. Further, search for a major section of patterns which belong to cluster with larger smallest pattern length will finish very fast as compared to other. Thus, overall time will reduce drastically.

There could be other parameters as well which depend on implementation of the algorithm. However, those are not considered because we consider factors which are intrinsic to algorithm and are expected to be equally weighted across various implementations.

6.3.1 Window Size (W)

Search time largely depends on size of window which is one short of length of smallest pattern. As length of window increases, number of characters skipped in *Base String* after each step increases. Let search time be called ST and size of window be called W. Then we can assume that ST is inversely proportional to W.

$$ST \propto 1/W \qquad (1)$$

6.3.2 ConPair Repeatition Ratio (CRR)

For every *ConPair* that is found at the terminal of window, one of three cases is probable. Firstly, *ConPair* may not present in *Match Table*. Secondly, there may be only one entry in list corresponding to that *ConPair*. Thirdly, there may be multiple entries corresponding to that

ConPair each of which needs to be checked. Third condition is of concern to us. During the search phase, time taken to identify whether a *ConPair* is present in *Match Table* or not is O(1) because we are using hashing. However, we have to process every element of list individually to find whether a pattern match is there or not. Thus, time taken to traverse that particular *ConPair* list grows linearly. Thus, search time increases as the number of repeated *ConPairs* increases. We need to focus on number of *ConPairs* repeated per unique *ConPair* on average. Thus, we define ConPair Repetation Ratio (CRR) as ratio of Repeated *ConPairs* in *Match Table* to Unique *ConPairs* present in *Match Table*. As this ratio increases, search time will also increase. Thus, ST is directly proportional to CRR.

$$ST \propto CRR \qquad (2)$$

6.4 Mathematical Relationships

Combining equation 1 and 2, we can derive following mathematical relation.

$$ST = K * CRR / W \qquad (3)$$

where K is constant of proportionality.

However, in case of clustering based search we will have separate search time for each cluster. Let there be n clusters with window size w1,w2,…,wn respectively with ConPair Repetation Ratio values CRR1, CRR2,…,CRRn respectively. Let their individual search times be ST1, ST2,…,STn repectively. Thus, we derive total search time relation for entire pattern set P as follows:

$$ST_{Combined} = \sum_{i=1}^{n} ST_i \qquad (4)$$

where $ST_{Combined}$ is combined search time after summation of search time of different clusters.

$$ST_i = K * CRR_i / W_i \qquad (5)$$

where ST_i is search time, CRR_i is ConPair Repeatition Ratio and W_i is window length for ith cluster. Here we assume constant of proportionality (K) remains constant for different clusters or pattern sets.

In order to compare search time taken for single patterns set with that of combined search time of various clusters, we can take ratio of search time calculated by equation (3) and equation (4) to give theoretical Speed Up Ratio (s_{Th}) as follows:

$$s_{Th} = \frac{K * (CRR_{set} / W_{set})}{\sum_{i=1}^{n} ST_i} \qquad (6)$$

where set represents pattern set P. Interestingly, we will never need value of K as it will cancel out in ratio calculations.

Chapter 7

EXPERIMENTAL SETUP

The following system configuration has been used while conducting the experiments:

Processor: Intel Core i3

Clock Speed: 2.53 GHz

Main Memory: 4 GB (3.67 Usable)

Hard Disk Capacity: 512 GB

System Type: 64-bit Operating System

Software: Code::Blocks version 12.11

7.1 Language and Implementation

Implementation of proposed algorithm is done in C language on Code::Blocks version 12.11. C is chosen as implementation language to efficiently design data structures and hashing functions as desired. Execution time is measured as mean of several runs and is the amount of CPU time taken in searching phase only. CPU time is different from time measured by clock for same duration. It is time spent by CPU specifically on this program itself. It excludes time taken by CPU to execute other applications running on system simultaneously. This helps us to have a closer look at performance in terms of time taken in milliseconds in entire experimental setup. In C language, we have inbuilt libraries which enable us to track time taken up to sectional level granularity. In other words, we can precisely calculate CPU time taken by particular section of program. This is one of the many reasons C was preferred as language of implementation.

The time taken may vary as per implementation of algorithm, architecture of machine used for implementation and nature of patterns being searched (as it decides uniqueness of *ConPairs*). However, intent of presenting these experimental results is to show the relative order of variation

among different parameters which are established under various running conditions and to prove validity of the mathematical relation established.

7.2 Language of Base String and Patterns

Here language chosen is English. Entire alphabet set is considered for implementation and analysis processes. Case sensitivity is not taken into account. No special characters have been included. However, successful implementation of this prototype guarantee that proposed work can be extended to any language with any number of special characters included.

7.3 Base String

Length of Base String for analysis of adaptive hashing based search is varied from 10,000 to 200,000 characters of English language. Length of Base String for clustering analysis is kept constant at 420,138 of same language.

7.4 Patterns

In case of adaptive hashing based search, number of patterns varies from 3 to 21 keeping length of the smallest pattern same across all variations.

For cohesive clustering, number of patterns is increased from 50 to 100 in steps of 10 to generate data for analysis. Length of patterns varies from 4 to 26. For clustering analysis, pattern set is divided into two clusters. It is necessary to ensure that as number of patterns increase, characteristic features displayed by patterns in terms on *ConPairs* (both total number and unique number), average length of patterns increase accordingly. Drastic increment, drastic decrement or stagnancy in these features affect environment generated for analysis of algorithm. In affects all parameters as *ConPairs* present in *Match Table*, search time taken by algorithm, generation of clusters and search time taken when clustering is done beforehand.

Thus, efforts have been put in to make sure average length of patterns in all three scenarios of search, i.e., single pattern set and two clusters remains constant across varying number of patterns. This is done to ensure no drastic change in range of length in a cluster or in complete pattern set.

It ensures that variation of distance from centroid of cluster remains same even though number of elements in cluster increase.

Since search algorithm proceeds in steps proportional to length of smallest pattern, doing this guarantees that there is not much variation in length of patterns disturbing effect of window shifts in search process. Additionally, at least one of the smallest patterns is added in smallest pattern set from the beginning to make sure length of window remains same as number of patterns increases.

Details of pattern distribution for cluster 1 (cluster with small length patterns), cluster 2 (cluster with comparatively larger length patterns) and complete pattern set P are shown in Table 7.1, Table 7.2 and Table 7.3 respectively. Column 1 denotes the length of pattern, Column 2 and onwards denotes frequency of patterns of length mentioned in Column 1.

Table 7.1: Pattern distribution for Cluster 1

Length of pattern	25 patterns	30 patterns	35 patterns	40 patterns	45 patterns	50 patterns
4	3	3	3	3	3	3
5	3	4	5	6	7	8
6	3	4	5	6	7	8
7	5	6	7	8	10	12
8	3	4	5	7	8	9
9	3	4	5	5	5	5
10	5	5	5	5	5	5
Average Length	7.24	7.2	7.17	7.13	7.07	7.02

Table 7.2: Pattern distribution for Cluster 2

Length of pattern	25 patterns	30 patterns	35 patterns	40 patterns	45 patterns	50 patterns
20	3	4	5	6	6	6
21	3	4	5	6	6	8
22	6	7	8	9	10	10
23	2	3	3	3	4	5
24	3	4	4	5	6	7
25	3	3	4	4	5	6
26	5	5	6	7	8	8
Average Length	23.12	22.93	22.91	22.88	23	22.98

Table 7.3: Pattern distribution for complete pattern set

	50 patterns	60 patterns	70 patterns	80 patterns	90 patterns	100 patterns
Average Length	15.18	15.07	15.04	15	15.03	15

Above tables show that average length of patterns for Cluster 1 is 7, for Cluster 2 is 23 and for complete pattern set P is 15. This will help in controlling characteristics of search space as its size grows.

Chapter 8

EXPERIMENTAL RESULTS AND ANALYSIS

Experimental results are divided into various sections.

8.1 Shift Table Result

Few multiple pattern shift table algorithms have been proposed. However, their implementation results are in exception with their theoretical results. A shift table algorithm was proposed taking patterns of equal lengths [7]. Comparison of theoretical result of this algorithm with respect to experimental one is mentioned in Table 8.1.

There are many reasons of these variations. Algorithm used in this paper decreases shift value of any particular character only when it is same as the last character of the pattern. Additionally, it excludes last character from processing. Thus, value of characters which appear only as last characters are not calculated.

Table 8.1: Result Comparison

Character	Theoretical shift value	Experimental shift value
a	2	3
b	1	3
c	1	4
d	1	4
e	1	-
f	1	-
m	2	4
n	4	4
o	3	4
p	1	4

q	3	4
j	3	4
z	3	4
*	4	4

Another shift table algorithm was proposed considering patterns of variable lengths [15]. Comparison of theoretical result of this algorithm with respect to experimental one is mentioned in Table 8.2.

Reason for negative values in implementation of this algorithm is because there is no restriction on decreasing shift value. Further, here only first n characters of any pattern are processed where n is the length of smallest pattern. Thus, it does not take into account of any character and its order of appearance after n characters. Here also shift value decreases only when any character matches with nth character of pattern. Also, last character is excluded from processing. This algorithm deals with patterns of variable length but it provides no consideration for different lengths of various patterns.

Table 8.2: Result Comparison

Character	Theoretical shift value	Experimental shift value
a	2	-5
b	1	3
c	1	4
d	1	-
e	1	-
f	1	-
m	2	4
n	4	4
o	3	4
p	1	4

q	3	4
j	3	4
z	3	4
*	4	4

In order to see accuracy of proposed algorithm let us consider set of patterns P = {*aaba, aabab, aababc, aababcd, aababcde, abcb, zmnd, qope, jmqfm*}.

Using shift table algorithm mentioned in this paper, shift values for various characters were obtained which are mentioned in Table 8.3.

Table 8.3: Proposed Algorithm's Shift Values

Character	Experimental shift value
a	1
b	1
c	1
d	1
e	1
f	1
m	1
n	1
o	3
p	1
q	1
j	3
z	3
*	4

The advantage of this algorithm is that it takes into account number of occurrence as well as relative order of appearance of characters. Thus, it successfully differentiates between patterns *abba* and *abab*. It takes into account the length of individual patterns and thus processing of each character is done depending on its position and neighbourhood. Additionally, it aims at finding biggest length match between *join* and *leftover* to ensure maximum shift distance possible. This enables it to reduce searching time which is crucial is situations where text is of unknown length or is generated dynamically at run time.

Using these values of shift table, we search pattern set P in text T = *aababcdezmndjmqfmaababcd* using two hashing table technique [11]. We are able to search all patterns successfully. Time taken by this algorithm can be further reduced by use of fast access data structures such as hash table. It will reduce order of time taken because matching of *join* with *leftover* will take O(1) time. This algorithm works well for variable size of patterns as illustrated in the example above.

8.2 Adaptive Hashing Based Search Result

8.2.1 Performance Analysis: Varying Base String Length

Number of patterns is kept fixed and length of *BaseString* is varied from 10,000 to 200,000.

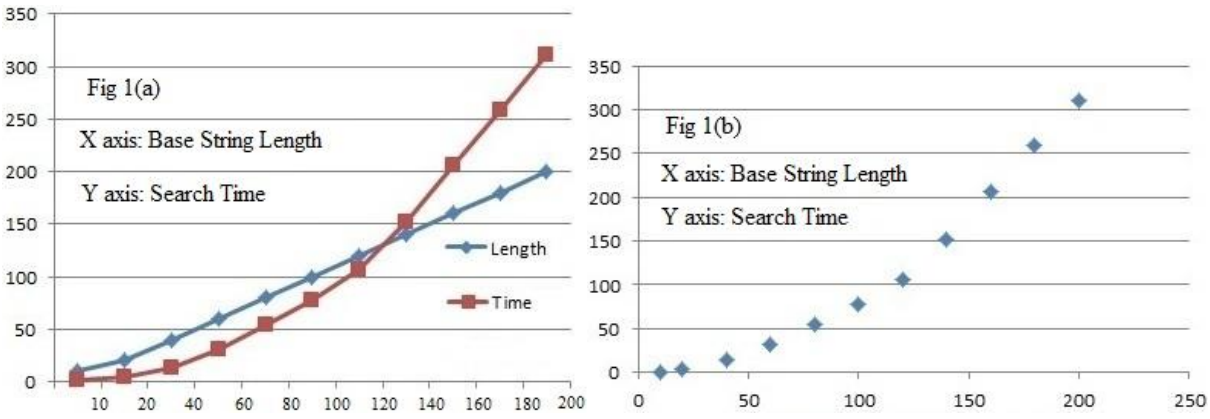

Figure 8.1: Performance Analysis on Variable *BaseString* Length

Figure 8.1(a) shows variation of *BaseString* length (in thousands) with searching time measured in milliseconds. Figure 8.1(b) shows dependent variation of the same. Both figures show that searching time taken increases linearly as *BaseString* length increases thus proving that this algorithm takes searching time of O(n/P) for *BaseString* of length n.

8.2.2 Performance Analysis: Varying Number of Patterns

BaseString length is kept fixed (100,000) and number of patterns are increased from 3 to 21 keeping length of smallest pattern same among all sets.

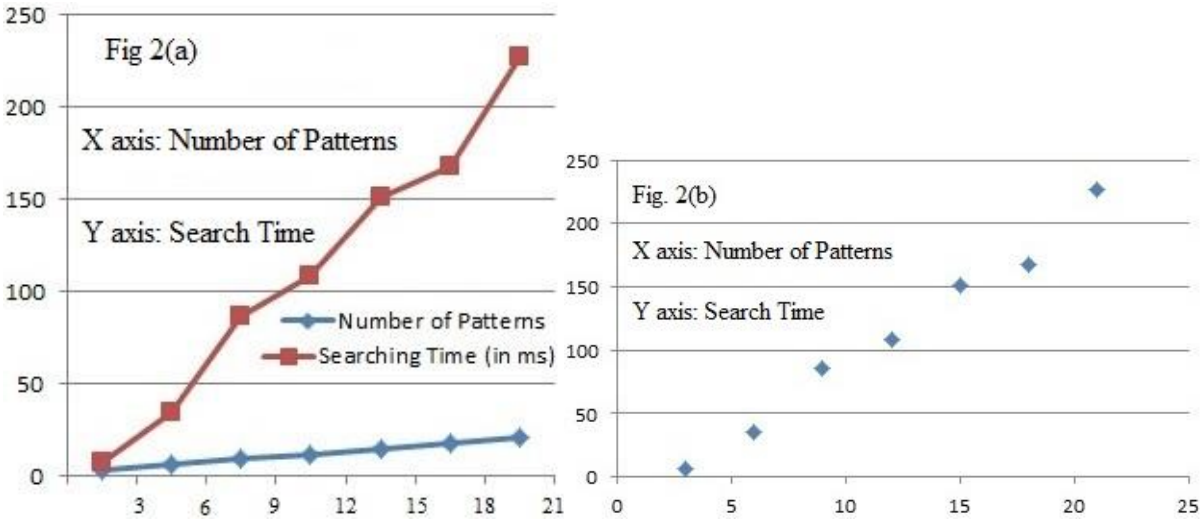

Figure 8.2: Performance Analysis on Varying Number of Patterns

Figure 8.2(a) shows variation of number of patterns to be searched in *BaseString* with searching time (in milliseconds). Figure 8.2(b) shows dependent variation of the same. It can be seen that time is increasing with number of patterns. This is so because instead of number of patterns, searching time atomically depends on nature of *ConPairs* found. They can be unique as well as repetitive.

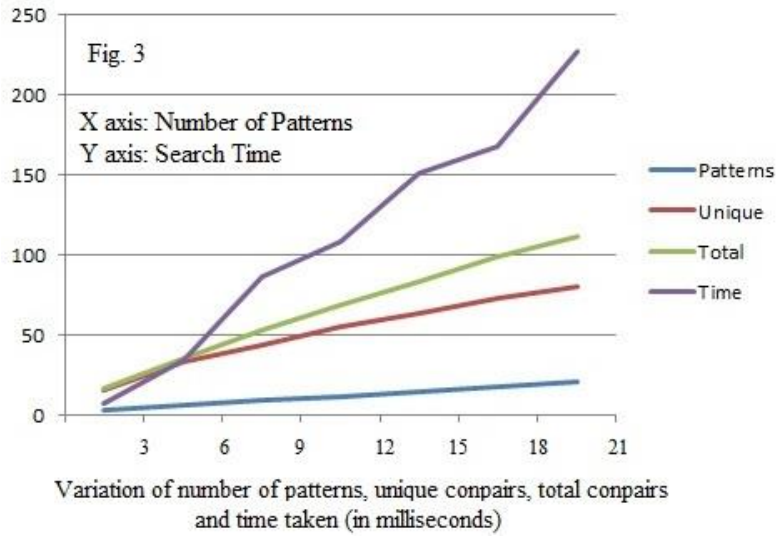

Figure 8.3: Relation among number of patterns, *ConPair* types and time taken (in ms)

Figure 8.3 shows increase in number of uniquely generating *ConPairs* and total *ConPairs* as number of patterns to be searched in *BaseString* increases. Overall, increase in searching time is also evident. This also signifies the fact that searching time depends on whether a *ConPair* exists for the given position. If it does then it further depends on number of patterns in which that *ConPair* is found.

8.3 Cohesive Clustering Result

This section provides the result obtained in various runs across various parameters. Three kinds of searches are performed. Firstly, search is done on pattern set P. Secondly, search is done on cluster 1 and lastly search is done on cluster 2. Each kind of search is repeated while changing total number of patterns from 50 to 100. Results across various parameters are mentioned below.

8.3.1 ConPairs

This considers both unique and total number of *ConPairs* generated across various runs. Figure 8.4 shows variation both as number of patterns increase.

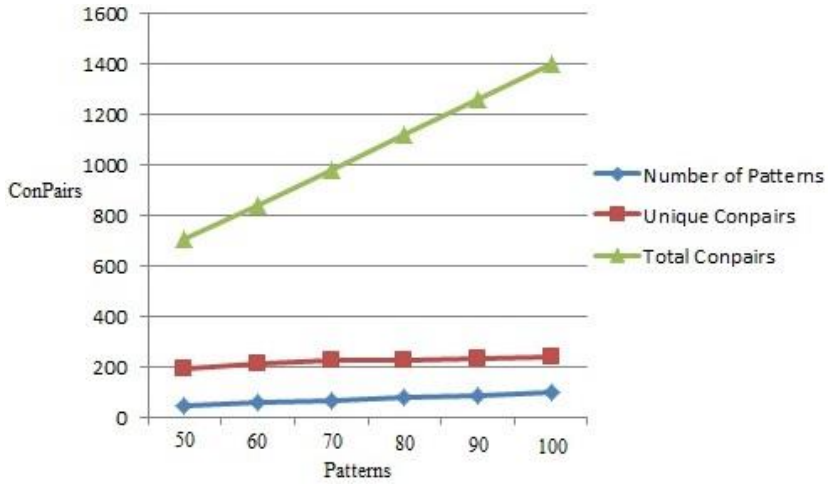

Figure 8.4: Variation of ConPairs with number of patterns

Linear increment in all the lines in Figure 8.4 proves that uniqueness of patterns is maintained as number of patterns is increased. This helps in ensuring that patterns are not anagrams of each other as there number is increased. This guarantees constant updating in search environment keeping good mix of repeating as well as new *ConPairs*. Thus, domain of search space increases along with proportional increase in characteristic which defines the search space. Figure 8.4 shows constant rate of growth in unique as well as total number of *ConPairs* as number of patterns increases.

We can observe that number of total *ConPairs* increase much rapidly with respect to number of unique *ConPairs*. We have to focus on fact that language we have chosen for implementation of algorithms is English. Number of *ConPairs* possible in English language is fixed to 676 (26 *ConPairs* starting with each of 26 characters, like aa, ab, ac to zx, zy, zz). Thus, we must see direct proportional growth in number of total *ConPairs* (as seen practically). This makes ConPair Repeatition Ration (CRR) a crucial factor in speed up of runtime because as number of patterns increase CRR will also increase. On the other hand, as number of patterns increase to high numbers, we should see constant number of unique *ConPairs* due to inherent upper limit for it.

8.3.2 Search Time

This section show variation in search time as number of patterns is increased keeping the *Base String* constant. Search time variation is shown for all three searches. Figure 3 shows variation of search time for pattern set P with elements in set increasing from 50 to 100. Time taken is in milliseconds.

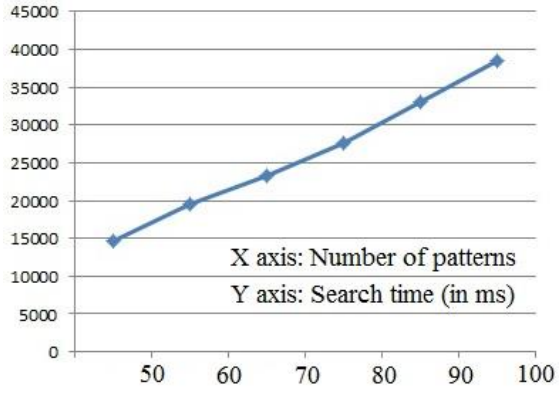

Figure 8.5: Search time vs. number of patterns

Figure 8.5 shows that search time in directly proportional to number of patterns. This can also be drawn from Figure 8.4 which depicts that uniqueness of patterns keep on increasing as there count increases. Since uniqueness increases, so does the search time.

Figure 8.6 shows the variation of search time as number of patterns increase from 25 to 50 in cluster 1 which consists of smaller lengths patters. Figure 8.7 shows the variation of same for cluster 2 which consists of comparatively larger patterns.

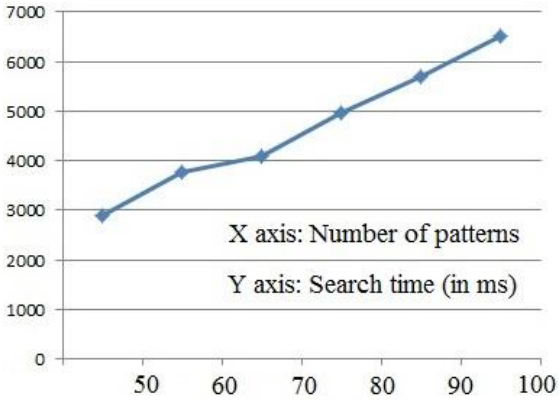

Figure 8.6: Search time for cluster 1 vs. number of patterns

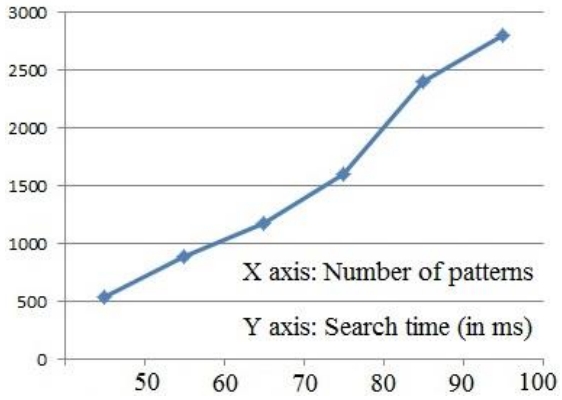

Figure 8.7: Search time for cluster 2 vs. number of patterns

Figure 8.6 and Figure 8.7 shows that search time in milliseconds is roughly in direct proportion to number of patterns. This is because as number of patterns increase, total number of *ConPairs* increases (as shown in Figure 8.4) which results in increase in CRR ratio too.

8.4 Post Experiment Clustering Analysis

Both Figure 8.6 and Figure 8.7 shows that search time are approximately proportional to the number of patterns. However, one thing to observe here is that time taken to search equal number of larger patterns is always much lesser than that of smaller patterns for same *Base String*. This proves that window length plays a significant role in reduction of search time. In order to analyse effect of clustering we add search time for cluster 1 and cluster 2 and compare it with search time of pattern set. Figure 8.8 shows comparison of the two.

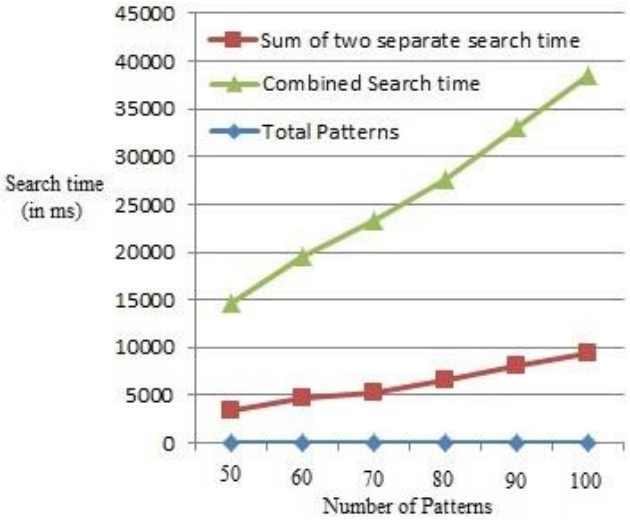

Figure 8.8: Search Time Comparisons

Figure 8.8 shows that clustering has drastically reduced the search time for every pattern set. It is mainly because difference in window size for clusters in considerable thus reducing search time for independent clusters individually which further reduces it collectively. It shows that clustering should be used prior to pre-processing phase. Let us further analyse degree to which clustering has speed up the search time in Figure 8.9.

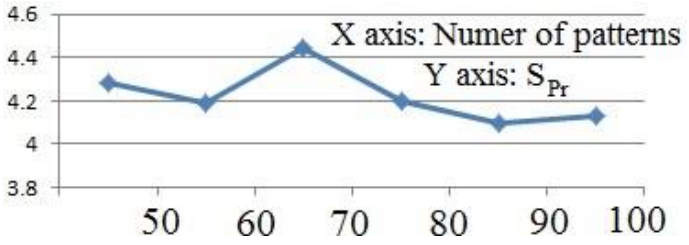

Figure 8.9: s_{Pr} with number of patterns

Figure 8.9 shows that practical Speed Up Ratio (s_{Pr}) in search time is almost constant for all pattern sets ranging from 4.1 to 4.5. This is the practical ratio that we are getting. Since this ratio is almost constant we can conclude that our strategy to configure a search domain for series of

practical such that characteristic parameters of search space grow in proportion to growth of its size has been successful.

8.5 Theoretical Clustering Analysis

We have established some equations earlier based on factors affecting search time. However, values fed in to these equations are not theoretical but they are practical parameters as measured among during run time. Motivation behind these calculations is to get an idea of speed up ratio prior to performing clustering process followed by searching process. Recall that there were two factors responsible for determining search time. We need to combine search time for cluster 1 and cluster 2 as per equation (4). Table 8.4 provides details of both the factors as observed.

Table 8.4: CRR and W for each pattern set

Nature	Number of Patterns	ConPair Repeatition Ratio (CRR)	Window Length (W)
Complete Pattern Set	50	2.593908629	3
Complete Pattern Set	60	2.957746479	3
Complete Pattern Set	70	3.345132743	3
Complete Pattern Set	80	3.899563319	3
Complete Pattern Set	90	4.360169492	3
Complete Pattern Set	100	4.733606557	3
Cluster 1	25	0.534653465	3
Cluster 1	30	0.697247706	3
Cluster 1	35	0.706349206	3
Cluster 1	40	0.848484848	3

Cluster 1	45	0.98540146	3
Cluster 1	50	1.068965517	3
Cluster 2	25	1.989189189	19
Cluster 2	30	2.340101523	19
Cluster 2	35	2.741463415	19
Cluster 2	40	3.180952381	19
Cluster 2	45	3.597222222	19
Cluster 2	50	3.95045045	19

Table 8.4 shows that ConPair Repeatition Ratio (CRR) increases as number of patterns increase. This shows there is tendency of having same *ConPairs* as pattern set increases. This holds true because number of possible ConPair is English language is 676 (26 starting for each of 26 characters). We will use the above data to verify the mathematical relationship we established in equation (4).

8.6 Experimental Result vs. Theoretical Equations

We now need to calculate accuracy of our theoretical equations. s_{Th} denotes the theoretical factor by which search time has been reduced by clustering. s_{Th} is calculated using equations fed with data provided in table 8.4. $s_{\Pr}$ denotes the practical factor by which search time has been reduced by clustering. It is calculated using the search time that we recorded during experiments up to the order of milliseconds. Figure 8.10 shows variation of practical speed up ratio ($s_{\Pr}$) with respect to theoretical speed up ratio (s_{Th}).

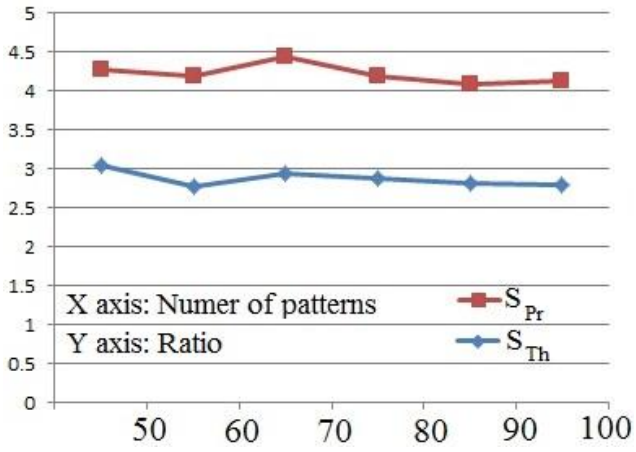

Figure 8.10: $s_{\Pr}$ and s_{Th} vs. number of patterns

Though values of practical runtime speed up ratio is not exactly equal to mathematically calculated runtime speed up ratio as shown in Figure 810. However, it is evident from the above figure that nature of increment or decrement in speed up ration is significantly equal. In other words, the rise and dip of both values are similar in nature. This proves the following:

i. The factors which we expect to be of significant importance actually hold expected degree of importance.

ii. The nature of relationship of factors with search time (like directly proportional or inversely proportional) are correct.

This closeness in nature of both plots proves that equations defined are correct. The ratio determining degree of correctness of our equations is called ***Accuracy Factor*** **(AF)** and is calculated by ratio of theoretical speed up ratio (s_{Th}) to practical speed up ratio($s_{\Pr}$).

$$AF = {s_{Th}}/{s_{\Pr}} \qquad (7)$$

The *Accuracy Factor* should give a better view of degree of correctness.

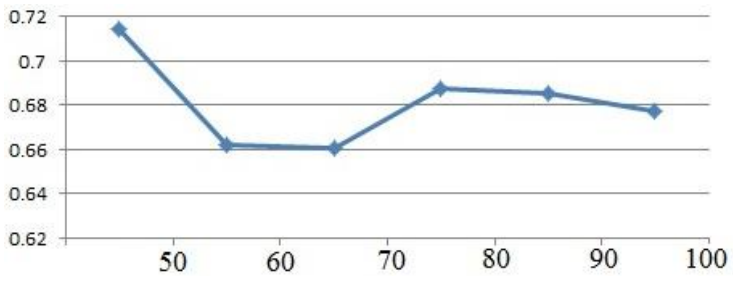

Figure 8.11: Accuracy Factor vs. Number of patterns

Figure 8.11 shows the ratio thus obtained are in the range 0.66 to 0.72, ideally it should be unity. Variation that this ratio has from unity shows that it our mathematical relationships need to be improved. However, closeness among values in all cases show that factors which we have considered are appropriate and mathematical relationships hold correct up to a large extent.

8.7 Runtime Analysis

8.7.1 New Shift Table Algorithm

For a single pattern with,

P: Length of single pattern

W: Size of window

Mathematically runtime of this algorithm on single pattern will be O(P-W+1). Effectively, runtime complexity will be O(P) considering pattern to be much larger than window.

For a set of variable length patterns with,

X: Number of pattern

N: Sum of lengths of all patterns

M: Average pattern length

L: Smallest pattern length, thus W=L-1

Mathematically runtime of generating shift table by presented algorithm is O(N/M*(M-W+1)). Considering, N is much larger in comparison to L, effectively runtime complexity for presented algorithm can be given as O(N).

8.7.2 New Adaptive Hashing Based Search Algorithm

Match Table generation time is linearly proportional to summation of lengths of every element in pattern set P. Let summation of all elements equate to L and number of elements be p then time taken to generate *Match Table* will be of order O(L-p).

Consider length of *Base String* to be n and length of smallest pattern in set P be k, then search time be of the order O(n/k). This has been proved in search time vs number of patterns plot shown above in Figure 8.5, Figure 8.6 and Figure 8.7.

8.7.3 Cohesive Clustering

Generation of clusters prior to entire process doesn't change runtime order of either *Match Table* generation or search time. However, it significantly condenses the search time as shown in Figure 8.8 and Figure 8.9.

Chapter 9

CONCLUSION

A shift table algorithm which is able to solve problem of variable length multiple string pattern matching is presented. This is inspired by Boyer Moore concept for single pattern matching. It further successfully enhances that concept to gain promising results. With use of hash table to search occurrence of join in leftover this algorithm takes O(|P|) time where |P| defines sum of all pattern lengths. As shown in results section, it produces better shift table than existing algorithms in terms of calculation of shift value for each character and its accurate value to gain maximum shift. This shift table algorithm ensures that we are able to pre-process pattern set for successful search in case of both static as well as dynamic text.

A hashing based search algorithm is presented in this paper which is capable of searching multiple variable length patterns. Also, it can adapt itself to new pattern made available to it during search. Match Table evolves as the number of pattern increase dynamically and need not to be rebuilt completely. It also avoids need of shift table and takes uniform jumps over the BaseString taking O(n/P) time exactly in all cases. Further this algorithm can be effectively used in two cases. First is where *BaseString* is available prior to search. Second case is when *BaseString* generates dynamically like in case of network analysis. However, RPosMax will not be available in this case. Then terminating condition of search algorithm will be till end of *BaseString*.

A concept of merging clustering with adaptive hashing based multiple pattern matching algorithm is presented in this paper. This works well keeping adaptive nature of algorithm alive. Results have shown speed up in search time by factor of 4. We have identified parameters which affect the speed up in search time. Mathematical relations have been successfully carved out from those parameters. This mathematical relation is significant for us to get an idea of speed up in search time of using clustering beforehand merging it with algorithm. Additionally, closeness in result as

well as in nature of increment or decrement in speed up of search time can now be predicted before the actual run which will save a lot of runtime cost. Thus, it will optimize both cost and time of our experiments.

However, major areas of improvement which appear as of now are also identified. Firstly, section B of redundancy check function is aimed to be further reduced. This will improve search time both in case of clustered and non-clustered algorithms. Secondly, there is scope of improvement in mathematically calculated speed up ratio.

There are various dimensions in which future work in this field can be directed. However, it majorly revolved round reducing section C of redundancy check function. Additionally, weightage of parameters affecting search time needs to be defined theoretically and verified practically. Further, there is also scope for finding new parameter affecting search time.

REFERENCES

[1] R.S. Boyer, and J.S. Moore, “A fast string searching algorithm”, Communications of the ACM, 20(10), 1977, pp.762-772.

[2] K. M. Karp, and M.O. Rabin, “Efficient randomized pattern matching algorithms”, IBM Journal of Research and Development, 31(2), 1987, pp.249-260.

[3] D.E. Knuth, J.H. Morris, V.R Pratt, “Fast pattern matching in strings”, SIAM Journal on Computing 6(1), 1997, pp.323-350.

[4] A. V. Aho, and M. J. Corasick, “Efficient string matching: An aid to bibliographic search”, Comm. ACM, 1975, pp.333-340.

[5] B. Commentz-Walter, “A string matching algorithm fast on the average”, In Proceedings of the Sixth International Collogium on Automata Languages and Programming, 1979, pp.118-132.

[6] G. Navarro, and M. Raffinot, “Flexible Pattern Matching in Strings”, The press Syndicate of The University of Cambridge. 2002.

[7] C. Khancome and V. Boonjing, “New Hashing-Based Multiple String Pattern Matching Algorithms”, 2012 Ninth International Conference on Information Technology- New Generations, (ITNG 2012), LasVegas, USA, 2-4 April 2012, pp.195-200.

[8] P.C. Bosnjak, and S. M. Cisar, “EWMA based threshold algorithm for intrusion detection”, Computing and Informatics, Vol. 29 No. 6+, 2010, pp. 1089-1101.

[9] Z. Wu, V. Raghavan, H. Qian, V. Rama, W. Meng, H. He and C. Yu, “Towards Automatic Incorporation of Search engines into a Large-Scale Metasearch Engine”, IEEE/WIC International Conference on Web Intelligence (WI’03), 13-17 Oct. 2003, pp. 658-661.

[10] C. Zhu, T. Liu, W. Zhang, D. Yang, “Greedy-search based service location in P2P networks”, Journal of Systems Engineering and Electronics, Volume 16, Issue 4, December 2005, pp. 886- 89.

[11] N. Cao, C. Wang, M. Li, K. Ren, W. Lou, “Privacy preserving multi keyword ranked search over encrypted cloud data”, IEEE transactions on Parallel and Distributed Systems, Vol. 25, No. 1, January 2014, pp. 222- 233.

[12] S.P. Bora, “Data mining and ware housing”, 3rd International Conference on Electronics Computer Technology (ICECT), 8-10 April 2011, Vol. 1, pp. 1-5.

[13] L. Chen, S. Lu and J. Ram, “Compressed Pattern Matching in DNA Sequences”, IEEE Computational and Systems Bioinformatics Conference (CBS 2004), 16-19 Aug 2004, pp. 62-68.

[14] S. Kuruppu, B. Beresford-Smith, T. Conway, and J. Zobel “Iterative dictionary construction for compression of large DNA data sets”, IEEE/ACM transactions on Computational Biology and Bioinformatics, Vol. 9, No. 1, Jan. - Feb. 2012, pp. 137- 149.

[15] C. Khancome, V. Boonjing and P. Chanvarasuth, “A Two-Hashing Table Multiple String Pattern Matching Algorithm”, 2013 Tenth International Conference on Information Technology-New Generations, (ITNG 2013), LasVegas, USA, 15-17 April 2013, pp.696-701.

[16] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter, “Fast practical multi-pattern matching”, Report 93-3, Institute Gaspard Monge, Universitĕ de Marne-la-Vallĕe, 1993.

[17] M. Crochemore, A. Czumaj, L. Gąsieniec, T. Lecroq, W. Plandowski, and W. Rytter, “Fast practical multi-pattern matching”, Information Processing Letters, 71(3/4), 1999, pp.107-113.

[18] L. Gongshen, L. Jianhua, and L. Shenghong, “New multi-pattern matching algorithm”, Journal of Systems Engineering and Electronics, Vol. 17, No. 2, 2006, pp.437-442.

[19] H. HYYRO, K. F. SSON, and G. Navarro, “Increased Bit-Parallelism for Approximate and Multiple String Matching”, ACM Journal of Experimental Algorithms, Vol.10, Article No. 2.6, 2005, pp.1-27.

[20] S. Wu, and U. Manber, “A fast algorithm for multi-pattern searching”, Report tr-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.

[21] P. Kanuga, A. Chauhan, “Adaptive Hashing Based Multiple Variable Length Pattern Search Algorithm for Large Data Sets”, International Conference on Data Science and Engineering, Cochin, in press.

[22] E. Liu, A. K. Jain, J. Tian, “A Coarse to Fine Minutiae-Based Latent Palmprint Matching”, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 35, Issue 10, 2013, pp. 2307-2322