



University of **HUDDERSFIELD**

University of Huddersfield Repository

Klaib, Ahmad

Exact string matching algorithms for searching DNA and protein sequences and searching chemical databases

Original Citation

Klaib, Ahmad (2014) Exact string matching algorithms for searching DNA and protein sequences and searching chemical databases. Doctoral thesis, University of Huddersfield.

This version is available at <http://eprints.hud.ac.uk/id/eprint/24266/>

The University Repository is a digital collection of the research output of the University, available on Open Access. Copyright and Moral Rights for the items on this site are retained by the individual author and/or other copyright owners. Users may access full items free of charge; copies of full text items generally can be reproduced, displayed or performed and given to third parties in any format or medium for personal research or study, educational or not-for-profit purposes without prior permission or charge, provided:

- The authors, title and full bibliographic details is credited in any copy;
- A hyperlink and/or URL is included for the original metadata page; and
- The content is not changed in any way.

For more information, including our policy and submission procedure, please contact the Repository Team at: E.mailbox@hud.ac.uk.

<http://eprints.hud.ac.uk/>

**EXACT STRING MATCHING ALGORITHMS FOR
SEARCHING DNA AND PROTEIN SEQUENCES AND
SEARCHING CHEMICAL DATABASES**

AHMAD F. KLAIB

A thesis submitted to the University of Huddersfield in partial fulfilment of the requirements for the degree of
Doctor of Philosophy

The University of Huddersfield

September 2014

COPYRIGHT STATEMENT

- i. The author of this thesis (including any appendices and/or schedules to this thesis) owns any copyright in it (the “Copyright”) and s/he has given The University of Huddersfield the right to use such copyright for any administrative, promotional, educational and/or teaching purposes.
- ii. Copies of this thesis, either in full or in extracts, may be made only in accordance with the regulations of the University Library. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.
- iii. The ownership of any patents, designs, trademarks and any and all other intellectual property rights except for the Copyright (the “Intellectual Property Rights”) and any reproductions of copyright works, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions

LIST OF PUBLICATIONS AND CONFERENCES

- 1 - Klaib, A. and Osborne, H. Searching Protein Sequence Databases Using BRBMH Matching Algorithm. International Journal of Computer Science and Network Security (IJCSNS). ISSN: 1738-7906, Vol. 8, No. 12, pp. 410-414, December, 2008.
- 2 - Klaib, A. and Osborne, H. BRQS Matching Algorithm for searching Protein Sequence Databases. International Conference of Future Computer and Communication (ICFCC 2009), Kuala Lumpur, Malaysia, 3-5 April 2009, pp. 223-227.
- 3 - Klaib, A. and Osborne, H. Exact String Matching Algorithms for Searching Biological Sequence Databases. Poster in the 3rd Saudi International Conference (SIC-2009), Surrey University, Guildford, United Kingdom, 4-6 June 2009.
- 4 - Klaib, A. and Osborne, H. OE Matching Algorithm for Searching Biological Sequences. International Conference on Bioinformatics, Computational Biology, Genomics and Chemoinformatics (BCBGC-09), Orlando, Florida, USA, 13-16 July 2009, pp. 36-42.
- 5 - Attended the 20th Annual Symposium on Combinatorial Pattern Matching Published by springer, in Lille, France, on June 22 - 24, 2009.

- 6 - Klaib, A. and Osborne, H. A New String Matching Algorithm for Searching Biological Sequences. In 2009 Conference Proc. International Conference on Information and Communication Systems (ICICS 2009), Amman, Jordan.
- 7 - Klaib, A. and Osborne, H. RSMA Matching Algorithm for Searching Biological Sequences. In 2009 IEEE Proc. 6th International Conference on Innovations in Information Technology (IIT '09), pp.195-199, 15-17 Dec. 2009, Alain, UAE.
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5413769&isnumber=54133>
54.

DEDICATIONS

To almighty Allah:

who gave me strength and good health while doing this research

To my main supervisor, Dr. Hugh Osborne:

who guided me throughout different stages of this research

To my co-supervisor, Dr. Andrew Crampton:

who helped in writing up this thesis

To my great parents, brothers and sisters:

who supported me until this point

To my parents-in-law, my lovely wife, daughter and new born son:

who stood next to me

To my relatives and friends:

who always encouraging me

May Allah keep all of you safe, happy and grant you a good health

With my love

ACKNOWLEDGMENT

First of all, I am grateful for the almighty Allah for his blessing and giving me the strength to write up this thesis. There are many people without whom this thesis might not have been written, and to whom I am greatly indebted.

I wish to express my sincere thanks to my supervisor, Dr. Hugh Osborne, for his guide, countless hours of reflecting, reading, his continuous support and most of all patience throughout different stages of this research.

Also I would like to thank my co-supervisor, Dr. Andrew Crampton for his cooperation and guidance while doing my research.

I would like to acknowledge and thank Dr. Christopher Newman for his support solving different problems I faced at the University of Huddersfield.

Special Thanks go to University of Huddersfield, who gave me this opportunity to complete my degree. Continuous thanks for every member of this university especially staff at the research and enterprise office, Computing and Engineering School and technician team for their continued support and cooperation.

Uncountable thanks to my parents, brothers, sisters, who spent their lives supporting and encouraging me, praying and dreaming of seeing this work completed. I wish I can return a very small part of their favors.

My warm appreciations go to my parents-in-law, my lovely wife (Alaa), my gorgeous daughter (Rama) and my newborn baby (Yamen) for being so patient with me and being very understandable for my situation.

Last, but not the least, great thanks to my relatives, friends especially Arshad who helped in implementing the parallel experiments, Ala', Abulrahman, Dr. Mahmoud, Dr. Baydaa, Dr. Ra'ad and all who had been helped and encouraged me in the past few years. Thank you very much for all of you.

Abstract

The enormous quantities of biological and chemical files and databases are likely to grow year on year, consequently giving rise to the need to develop string-matching algorithms capable of minimizing the searching response time. Being aware of this need, this thesis aims to develop string matching algorithms to search biological sequences and chemical structures by studying exact string matching algorithms in detail. As a result, this research developed a new classification of string matching algorithms containing eight categories according to the pre-processing function of algorithms and proposed five new string matching algorithms; BRBMH, BRQS, Odd and Even algorithm (OE), Random String Matching algorithm (RSMA) and Skip Shift New algorithm (SSN).

The main purpose behind the proposed algorithms is to reduce the searching response time and the total number of comparisons. They are tested by comparing them with four well-known standard algorithms, Boyer Moore Horspool (BMH), Quick Search (QS), TVSBS and BRFS.

This research applied all of the algorithms to sample data files by implementing three types of tests. The number of comparison tests showed a substantial difference in the number of comparisons our algorithms use compared to the non-hybrid algorithms such as QS and BMH. In addition, the tests showed considerable difference between our algorithms and other hybrid algorithm such as TVSBS and BRFS. For instance, the average elapsed search time tests showed that our algorithms presented better average elapsed search time than the BRFS, TVSBS, QS and BMH algorithms, while the average number of tests showed better number of attempts compared to BMH, QS, TVSBS and BRFS algorithms.

A new contribution has been added by this research by using the fastest proposed algorithm, the SSN algorithm, to develop a chemical structure searching toolkit to search chemical structures in our local database. The new algorithms were paralleled using OpenMP and MPI parallel models and tested at the University of Science Malaysia (USM) on a Stealth Cluster with different number of threads and processors to improve the speed of searching pattern in the given text which, as we believe, is another contribution.

Table of Contents

COPYRIGHT STATEMENT	2
LIST OF PUBLICATIONS AND CONFERENCES	3
DEDICATIONS.....	5
ACKNOWLEDGMENT.....	6
Abstract.....	7
Table of Contents	8
List of Figures	13
List of Tables	15
List of Abbreviations	18
CHAPTER 1: INTRODUCTION	21
1.1 Introduction	21
1.2 Background	22
1.2.1 Biological Data	22
1.2.1.1 DNA	23
1.2.1.2 Proteins	25
1.2.2 Chemical Data Representation	26
1.2.2.1 Antimicrobial Structures.....	27
1.2.3 Sequence Databases	27
1.2.3.1 Biological Databases	28
1.2.3.2 Chemical Databases.....	29
1.2.4 String Matching Algorithms.....	29
1.3 Research Motivation.....	30
1.4 Research Hypothesis and General Research Methodology	30
1.5 Research Questions	32
1.6 Research Objectives	33
1.7 Main Contribution	33
1.8 Overview of the Thesis.....	34
CHAPTER 2: A CURRENT STATE OF THE ART.....	37
2.1 Conventions.....	37
2.2 The First Category: Shift the Pattern a Single Position	37
2.2.1 The Brute Force Algorithm (BF)	38
2.3 The Second Category: Using Two Preprocessing Functions.....	41
2.3.1 The Boyer Moore Algorithm (BM).....	42
2.3.2 The Zhu Takaoka Algorithm (ZT)	46
2.3.3 The Fast Search Algorithm (FS)	48
2.4 The Third Category: Depending on the Rightmost Character	50
2.4.1 The Boyer Moore Horspool Algorithm (BMH)	50

2.5	The Fourth Category: Depending on the Next Character to the Rightmost Character	54
2.5.1	The Quick-Search Algorithm (QS)	54
2.6	The Fifth Category: Depending on Two Characters Next to the Rightmost Character	56
2.6.1	The Berry–Ravindran Algorithm (BR)	56
2.7	The Sixth Category: Using a Hashing Function	59
2.7.1	The Karp-Rabin Algorithm (KR).....	59
2.8	The Seventh Category: Computing Buckets for All Characters of the Alphabet	64
2.8.1	The Skip Shift Algorithm (SS).....	64
2.8.2	The Alpha Skip Shift Algorithm (ASS)	67
2.9	The Eighth Category: Using Hybrid Algorithms.....	70
2.9.1	The SSABS Algorithm.....	70
2.9.2	The FJS Algorithm.....	72
2.9.3	The TVSBS Algorithm	73
2.9.4	The ZTBMH Algorithm.....	75
2.9.5	The BRFS Algorithm	76
2.9.6	The BM-KMB Algorithm	78
2.9.7	The BRSS Algorithm	79
2.9.8	The ASSBR Algorithm	81
2.9.9	The MRCA Algorithm	84
2.9.10	The KRBMH Algorithm	84
2.9.11	The Quick-Skip Search Algorithm (QSS).....	84
2.9.12	The AKRAM Algorithm.....	85
2.10	Summary of String Matching Algorithms	85
2.11	SMILES Format	92
2.12	Parallel Computing.....	98
2.12.1	Flynn’s Taxonomy	99
2.12.2	Parallel Computing Speedup:.....	101
2.12.3	Parallel Programing Models:.....	102
2.12.3.1	The Shared Memory Model:.....	102
2.12.3.2	The Distributed Memory Model:	104
2.12.3.3	The Hybrid Memory Model:.....	106
2.13	Summary	107
CHAPTER 3: METHODOLOGY AND DESIGN		108
3.1	Framework of Research Methodology	108
3.2	Chemical Structures Toolkit Design.....	110
3.2.1	The First Stage: Downloading and Mining Structures	112
3.2.2	The Second Stage: Building the Local Database	112
3.2.3	The Third Stage: Using the JME Editor, SMILES and the SSN Algorithm	114
3.2.4	The Fourth Stage: Measuring Similarity Using the Proportion of Matching Characters	115

3.3	Parallel Algorithm Design	116
3.3.1	Parallel Algorithm Design for Shared Memory Model	116
3.3.2	Parallel Algorithm Design for Distributed Memory Model	117
3.4	Summary	118
CHAPTER 4: DEVELOPING NEW ALGORITHMS		119
4.1	The BRBMH Algorithm.....	119
4.1.1	The Preprocessing Phase of the BRBMH Algorithm:.....	119
4.1.2	The Searching Phase of the BRBMH Algorithm:	123
4.1.3	The BRBMH: Working Example.....	124
4.1.3.1	Input Sample	124
4.1.3.2	The BRBMH Example's Preprocessing Phase	124
4.1.3.3	The BRBMH Example's Searching Phase.....	124
4.2	The BRQS Algorithm.....	128
4.2.1	The Preprocessing Phase of the BRQS Algorithm:.....	128
4.2.2	The Searching Phase of the BRQS Algorithm:	128
4.2.3	The BRQS: Working Example.....	129
4.2.3.1	Input Sample	130
4.2.3.2	The BRQS Example's Preprocessing Phase	130
4.2.3.3	The BRQS Example's Searching Phase	130
4.3	The Odd and Even Algorithm (OE)	134
4.3.1	The Preprocessing Phase of the OE Algorithm:.....	134
4.3.2	The Searching Phase of the OE Algorithm:	134
4.3.3	The OE: Working Example.....	135
4.3.3.1	Input Sample	136
4.3.3.2	The OE Example's Preprocessing Phase	136
4.3.3.3	The OE Example's Searching Phase.....	136
4.4	The Random String Matching Algorithm (RSMA)	140
4.4.1	The Pre-processing Phase of the RSMA Algorithm:	140
4.4.2	The Searching Phase of the RSMA Algorithm	140
4.4.3	The RSMA: Working Example.....	143
4.4.3.1	Input Sample	143
4.4.3.2	The RSMA Example's Preprocessing Phase	143
4.4.3.3	The RSMA Example's Searching Phase.....	144
4.5	The Skip Shift New (SSN) Algorithm.....	147
4.5.1	The Preprocessing Phase of the SSN Algorithm:.....	148
4.5.2	The Searching Phase of the SSN Algorithm:	148
4.5.3	The SSN: Working Example.....	150
4.5.3.1	Input Sample	150
4.5.3.2	The SSN Example's Preprocessing Phase	150

4.5.3.3	The SSN Example's Searching Phase.....	151
4.6	Summary	154
CHAPTER 5: IMPLEMENTATION.....		155
5.1	System Specification	155
5.2	Chemical Structures Toolkit Implementation.....	155
5.2.1	Downloading and Mining Structures	155
5.2.2	Building the Local Database	156
5.2.2.1	Local Database Design	156
5.2.2.2	Table Format and Description	156
5.2.3	Using JME Editor, SMILES and the SSN Algorithm	158
5.2.4	Implementation of Similarity Measuring	158
5.3	Parallel Algorithm Implementation	159
5.3.1	OpenMP Model Implementation.....	159
5.3.2	MPI Model Implementation	161
5.4	Summary	162
CHAPTER 6: RESULTS AND DISCUSSION		163
6.1	Testing Algorithms Using a Short DNA Pattern	164
6.1.1	The Number of Comparisons Using a Short DNA Pattern	164
6.1.2	The Number of Attempts Using a Short DNA Pattern.....	165
6.1.3	The Average Elapsed Search Time Using a Short DNA Pattern.....	167
6.2	Testing Algorithms Using a Long DNA Pattern	169
6.2.1	The Number of Comparisons Using a Long DNA Pattern.....	169
6.2.2	The Number of Attempts Using a Long DNA Pattern	170
6.2.3	The Average Searching Elapsed Time Using a Long DNA Pattern.....	171
6.3	Testing Algorithms Using a Short Protein Pattern	173
6.3.1	The Number of Comparisons Using a Short Protein Pattern.....	173
6.3.2	The Number of Attempts Using a Short Protein Pattern	174
6.3.3	The Average Elapsed Search Time Using a Short Protein Pattern.....	175
6.4	Testing Algorithms Using a Long Protein Pattern.....	176
6.4.1	The Number of Comparisons Using a Long Protein Pattern.....	177
6.4.2	The Number of Attempts Using a Long Protein Pattern	178
6.4.3	The Average Elapsed Search Time Using a Long Protein Pattern.....	178
6.5	Testing Parallel Algorithms.....	181
6.5.1	Testing the OpenMP Model on DNA Sequences File.....	181
6.5.2	Testing the OpenMP Model on Protein Sequences File.....	182
6.5.3	Testing the MPI Model on DNA Sequences File	182
6.5.4	Testing the MPI Model on Protein Sequences File	183
6.6	Testing the Chemical Searching Toolkit Using the SSN Algorithm	184
6.7	Discussion	187

6.7.1	The Number of Comparisons Test Discussion	189
6.7.2	The Average Elapsed Search Time Test Discussion	190
6.7.3	The Number of Attempts Test Discussion	192
6.7.4	The Parallel Algorithm Tests Discussion	194
CHAPTER 7: CONCLUSION AND FUTURE WORK		198
7.1	Conclusion	198
7.2	Future Work	201
REFERENCES		202
APPENDIX A: STRING MATCHING ALGORITHMS CODE		215
APPENDIX B: SMILES EBNF		239
APPENDIX C: TOOLKIT AND PARALLEL MODELS IMLEMNTATION		241
APPENDIX D: LOCAL DATABASE TABLES FORMAT AND DESCRIPTION		249

List of Figures

FIGURE 1- 1: PRIMARY, SECONDARY, TERTIARY AND QUATERNARY STRUCTURES (BAILEY, 2006)	23
FIGURE 1- 2: DNA BASE PAIRS	24
FIGURE 1- 3: DNA DOUBLE HELIX (SETUBAL ET AL., 1997; AGUSTINA, 2012)	25
FIGURE 1- 4: GROWTH OF PROTEIN SEQUENCE ENTRIES IN THE SWISS-PROT DATABASE SINCE.....	28
FIGURE 1- 5: SYSTEM DEVELOPMENT METHODOLOGY (MORRISON & GEORGE, 1995)	32
FIGURE 2- 1: THE BRUTE FORCE ALGORITHM EXAMPLE	41
FIGURE 2- 2: THE BOYER-MOORE ALGORITHM EXAMPLE.....	45
FIGURE 2- 3: ZHU TAKAOKA ZTBC EQUATION	46
FIGURE 2- 4: THE ZHU-TAKAOKA ALGORITHM EXAMPLE	48
FIGURE 2- 5: THE FAST SEARCH ALGORITHM EXAMPLE.....	49
FIGURE 2- 6: THE HORSPOOL ALGORITHM EXAMPLE	53
FIGURE 2- 7: THE QUICK SEARCH ALGORITHM EXAMPLE	55
FIGURE 2- 8: THE BERRY-RAVINDRAN ALGORITHM EXAMPLE.....	58
FIGURE 2- 9: FIRST EXAMPLE OF THE KARP-RABIN ALGORITHM	60
FIGURE 2- 10: SECOND EXAMPLE OF THE KARP-RABIN ALGORITHM	63
FIGURE 2- 11: THE SKIP SEARCH ALGORITHM EXAMPLE.....	66
FIGURE 2- 12: AN EXAMPLE FOR TREE T(X) OF ALL SUBSTRINGS WITH L=3	67
FIGURE 2- 13: THE ALPHA SKIP SEARCH ALGORITHM EXAMPLE.....	69
FIGURE 2- 14: THE SSABS ALGORITHM EXAMPLE	72
FIGURE 2- 15: THE TVSBS ALGORITHM EXAMPLE	74
FIGURE 2- 16: THE ZTBMH ALGORITHM EXAMPLE.....	76
FIGURE 2- 17: THE BRFS ALGORITHM EXAMPLE.....	78
FIGURE 2- 18: THE BRSS ALGORITHM EXAMPLE	81
FIGURE 2- 19: THE ASSBR ALGORITHM EXAMPLE	83
FIGURE 2- 20: AN EXAMPLE FOR THE SEQUENTIAL PROGRAM TASKS EXECUTION	98
FIGURE 2-21: AN EXAMPLE FOR THE PARALLEL PROGRAM TASKS EXECUTION	99
FIGURE 2- 22: FLYNN’S TAXONOMY (FLYNN, 1972)	100
FIGURE 2- 23: PARALLEL COMPUTING SPEEDUP EQUATION.....	101
FIGURE 2- 24: DIFFERENT TYPE OF SPEEDUP USING DIFFERENT NUMBER OF PROCESSORS.....	101
FIGURE 2-25: THE RELATIONSHIP BETWEEN THE COMMON SPEEDUP AND NUMBER OF PROCESSORS.....	102
FIGURE 2- 26: THE SHARED MEMORY MODEL	103
FIGURE 2-27: THE THREADS MODEL	103
FIGURE 2- 28: THE FORK AND JOIN MODEL.....	104
FIGURE 2-29: THE DISTRIBUTED MEMORY MODEL	105
FIGURE 2- 30: THE MPI MODEL STRUCTURE	106
FIGURE 2- 31: THE HYBRID MEMORY MODEL	107

FIGURE 3- 1: RESEARCH OBJECTIVE FRAMEWORK	109
FIGURE 3- 2: CHEMICAL TOOLKIT DESIGN.....	111
FIGURE 3- 3: MINING AND DOWNLOADING STRUCTURES FROM NMRSHIFTDB	112
FIGURE 3- 4: THE LOCAL DATABASE ER DIAGRAM	113
FIGURE 3- 5: DRAWING A STRUCTURE USING JME TOOL	114
FIGURE 3- 6: RESULTS OF APPLYING SMILES RULES ON FIGURE 3-5 EXAMPLE	115
FIGURE 3- 7: SEARCH STRUCTURE IN THE LOCAL DATABASE USING THE SSN ALGORITHM.....	115
FIGURE 3- 8: AN EXAMPLE OF PARALLELIZING THE “FOR LOOP” USING THE OPENMP MODEL.....	117
FIGURE 3- 9: PARALLEL ALGORITHM DESIGN USING THE MPI MODEL.....	118
FIGURE 4- 1: THE FIRST SHIFT CASE OF THE BRBMH ALGORITHM	121
FIGURE 4- 2: THE SECOND SHIFT CASE OF THE BRBMH ALGORITHM	121
FIGURE 4- 3: THE THIRD SHIFT CASE OF THE BRBMH ALGORITHM	122
FIGURE 4- 4: THE FOURTH SHIFT CASE OF THE BRBMH ALGORITHM	122
FIGURE 4- 5: THE BRBMH ALGORITHM CODE.....	123
FIGURE 4- 6: THE BRQS ALGORITHM CODE.....	129
FIGURE 4-7: THE OE ALGORITHM CODE	135
FIGURE 4- 8: THE RSMA ALGORITHM SEARCHING PHASE EQUATIONS	141
FIGURE 4- 9: THE RSMA ALGORITHM CODE.....	142
FIGURE 4- 10: THE SSN ALGORITHM PREPROCESSING PHASE EQUATION	148
FIGURE 4- 11: THE SSN ALGORITHM CODE.....	149
FIGURE 6- 1: BRQS AND BRBMH SEARCHING TIME USING A SHORT DNA PATTERN.....	167
FIGURE 6- 2: SSN, RSMA AND OE SEARCHING TIME USING A SHORT DNA PATTERN.....	168
FIGURE 6- 3: THE AVERAGE SEARCHING ELAPSED TIME FOR A LONG DNA (32-256)	171
FIGURE 6- 4: THE BRQS, BRBMH, QS AND BMH SEARCHING TIME FOR A LONG DNA (384-1024)	172
FIGURE 6- 5: THE SSN, RSMA AND OE SEARCHING TIME FOR A LONG DNA (384-1024)	172
FIGURE 6- 6: THE AVERAGE SEARCHING ELAPSED TIME FOR A SHORT PROTEIN PATTERN (4 - 31)	176
FIGURE 6- 7: BRBMH AND BRQS SEARCHING TIME USING A LONG PROTEIN PATTERN.....	179
FIGURE 6- 8: RSMA AND OE SEARCHING TIME USING A LONG PROTEIN PATTERN.....	179
FIGURE 6- 9: THE SSN SEARCHING TIME USING A LONG PROTEIN PATTERN	180
FIGURE 6- 10: OPENMP MODEL: THE AVERAGE ELAPSED SEARCH TIME ON DNA SEQUENCES FILE	181
FIGURE 6- 11: OPENMP MODEL: THE AVERAGE ELAPSED SEARCH TIME ON PROTEIN SEQUENCES FILE	182
FIGURE 6- 12: MPI MODEL: THE AVERAGE ELAPSED SEARCH TIME ON DNA SEQUENCES FILE.....	183
FIGURE 6- 13: MPI MODEL: THE AVERAGE ELAPSED SEARCH TIME ON PROTEIN SEQUENCES FILE.....	183
FIGURE 6- 14: INPUT A PATTERN CHEMICAL STRUCTURE	184
FIGURE 6- 15: VERIFY THE SMILES INPUT STRUCTURE	185
FIGURE 6- 16: SEARCH AND LIST SIMILAR STRUCTURES WITH SIMILARITY PERCENTAGE	186
FIGURE 6- 17: DETAILS OF SELECTED CHEMICAL STRUCTURE.....	186

List of Tables

TABLE 1- 1: TWENTY AMINO ACID ABBREVIATIONS (WATERMAN, 1995).....	26
TABLE 2- 1: BAD-CHARACTER SHIFT IN BOYER-MOORE ALGORITHM EXAMPLE	42
TABLE 2- 2: THE BEST CASE EXAMPLE OF THE BOYER-MOORE ALGORITHM.....	43
TABLE 2- 3: THE BOYER-MOORE BMBC TABLE	44
TABLE 2- 4: BOYER-MOORE <i>BMGS</i> TABLE	44
TABLE 2- 5: THE ZHU-TAKAOKA ZTBC TABLE	47
TABLE 2- 6: AN EXAMPLE OF THE BMBC FUNCTION.....	50
TABLE 2- 7: AN EXAMPLE OF HRBC FUNCTION	51
TABLE 2- 8: HORSPOOL HSBC TABLE	51
TABLE 2- 9: QUICK SEARCH QBC TABLE.....	54
TABLE 2- 10: THE BERRY-RAVINDRAN BRBC TABLE.....	56
TABLE 2- 11: HASH VALUE FOR THE SEARCHED PATTERN.....	60
TABLE 2- 12: SKIP SEARCH TABLE USED BY SS ALGORITHM	64
TABLE 2- 13: ALPHA SKIP SEARCH TABLE USED BY ASS ALGORITHM.....	68
TABLE 2- 14: THE QBC TABLE USED BY SSABS ALGORITHM.....	71
TABLE 2- 15: THE BRBC TABLE USED BY TVSBS	73
TABLE 2- 16: THE ZTBC TABLE USED BY ZTBMH.....	75
TABLE 2- 17: THE BRBC TABLE USED BY THE BRFS ALGORITHM.....	77
TABLE 2- 18: THE BRBC TABLE USED BY THE BRSS ALGORITHM.....	80
TABLE 2- 19: SKIP SEARCH TABLE USED BY BRSS ALGORITHM	80
TABLE 2- 20: THE BRBC TABLE USED BY THE ASSBR ALGORITHM.....	82
TABLE 2- 21: ALPHA SKIP SEARCH TABLE USED BY ASSBR ALGORITHM.	82
TABLE 2- 22: SUMMARY OF ALGORITHMS BEEN USED IN THIS RESEARCH.....	91
TABLE 2-23: STRUCTURES IN SMILES FORMAT EXAMPLES	93
TABLE 2-24: THE SMILES ATOM RULE OF STRUCTURES WITH ORGANIC SUBSET ELEMENTS.....	94
TABLE 2-25: THE SMILES ATOM RULE OF NON-ORGANIC ELEMENTS	94
TABLE 2- 26: EXAMPLES OF POSITIVE AND NEGATIVE CHARGES REPRESENTED BY NUMBERS.....	94
TABLE 2-27: EXAMPLES OF POSITIVE AND NEGATIVE CHARGES REPRESENTED BY SIGNS.....	95
TABLE 2-28: EXAMPLES OF SMILES BONDS	95
TABLE 2-29: EXAMPLES OF SMILES BRANCHES RULE	96
TABLE 2-30: EXAMPLE OF SMILES CYCLIC STRUCTURE RULE.....	96
TABLE 2-31: EXAMPLE OF SMILES DISCONNECTED RULE	97
TABLE 2-32: EXAMPLE OF SMILES AROMATICITY RULE	97
TABLE 4- 1: THE BRBMH INPUT SAMPLE.....	124
TABLE 4-2: THE ENHANCED BRBC TABLE OF THE BRBMH ALGORITHM	124
TABLE 4-3: THE BRBMH ALGORITHM SEARCHING ORDER.....	125

TABLE 4-4: THE FIRST ATTEMPT OF THE BRBMH ALGORITHM.....	125
TABLE 4-5: THE SECOND ATTEMPT OF BRBMH	126
TABLE 4-6: THE THIRD ATTEMPT OF BRBMH	126
TABLE 4-7: THE FOURTH ATTEMPT OF BRBMH	127
TABLE 4-8: BRQS INPUT SAMPLE	130
TABLE 4- 9: THE BRQS ENHANCED BRBC PREPROCESSING TABLE	130
TABLE 4- 10: THE BRQS ALGORITHM SEARCHING ORDER.....	130
TABLE 4-11: THE BRQS ALGORITHM: THE FIRST ATTEMPT IN THE SEARCHING PHASE	131
TABLE 4-12: THE BRQS ALGORITHM: THE SECOND ATTEMPT IN THE SEARCHING PHASE	131
TABLE 4-13: THE BRQS ALGORITHM: THE THIRD ATTEMPT IN THE SEARCHING PHASE	132
TABLE 4-14: THE BRQS ALGORITHM: THE FOURTH ATTEMPT IN THE SEARCHING PHASE	133
TABLE 4- 15: OE INPUT SAMPLE	136
TABLE 4- 16: THE OE ALGORITHM SEARCHING ORDER.....	137
TABLE 4- 17: THE OE ALGORITHM: THE FIRST ATTEMPT IN THE SEARCHING PHASE	137
TABLE 4-18: THE OE ALGORITHM: THE SECOND ATTEMPT IN THE SEARCHING PHASE	138
TABLE 4-19: THE OE ALGORITHM: THE THIRD ATTEMPT IN THE SEARCHING PHASE	138
TABLE 4-20: THE RSMA ALGORITHM: THE SECOND ATTEMPT IN THE SEARCHING PHASE	145
TABLE 4-21: THE RSMA ALGORITHM: THE THIRD ATTEMPT IN THE SEARCHING PHASE	145
TABLE 4-22: THE RSMA ALGORITHM: THE FOURTH ATTEMPT IN THE SEARCHING PHASE	146
TABLE 4-23: THE SSN ALGORITHM: THE FIRST ATTEMPT IN THE SEARCHING PHASE	152
TABLE 4-24 (A): THE SSN ALGORITHM: THE SECOND ATTEMPT IN THE SEARCHING PHASE	152
TABLE 4-25(B): THE ASS_NEW ALGORITHM: THE SECOND ATTEMPT IN THE SEARCHING PHASE	153
TABLE 4-26: THE ASS_NEW ALGORITHM: THE THIRD ATTEMPT IN THE SEARCHING PHASE	153
TABLE 5- 1: MOLECULE TABLE DESIGN AND EXAMPLE	156
TABLE 5- 2: KEYWORD TABLE DESIGN AND EXAMPLE	157
TABLE 5- 3: MOLECULE_KEYWORD TABLE DESIGN AND EXAMPLE	158
TABLE 5- 4: FINDING THE TEXT AND PATTERN LENGTH FOR SIMILARITY MEASURING.....	159
TABLE 5- 5: THE MAIN OPENMP FUNCTIONS WHICH USED TO PARALLELIZE THE SSN ALGORITHM	160
TABLE 5- 6: THE MAIN SEVEN FUNCTIONS OF MPI WHICH USED TO PARALLELIZE THE SSN ALGORITHM	161
TABLE 6- 1: THE NUMBER OF COMPARISONS FOR A SHORT DNA PATTERN.....	164
TABLE 6- 2: THE NUMBER OF ATTEMPTS FOR A SHORT DNA PATTERN	166
TABLE 6- 3: THE NUMBER OF COMPARISONS FOR A LONG DNA PATTERN	169
TABLE 6- 4: THE NUMBER OF ATTEMPTS FOR A LONG DNA PATTERN.....	170
TABLE 6- 5: THE NUMBER OF COMPARISONS FOR A SHORT PROTEIN PATTERN	174
TABLE 6- 6: THE NUMBER OF ATTEMPTS FOR A SHORT PROTEIN PATTERN	175
TABLE 6- 7: THE NUMBER OF COMPARISONS FOR A LONG PROTEIN PATTERN.....	177
TABLE 6- 8: NUMBER OF ATTEMPTS FOR LONG PROTEIN PATTERN	178
TABLE 6- 9: MPI VS. OPENMP: AVERAGE ELAPSED SEARCH TIME FOR SEARCHING DNA.....	195
TABLE 6- 10: MPI VS. OPENMP: AVERAGE ELAPSED SEARCH TIME FOR SEARCHING PROTEIN.....	196

TABLE 6- 11: THE SPEEDUP OF MPI AND OPENMP MODELS FOR DNA PATTERNS	197
TABLE 6- 12: THE SPEEDUP OF MPI AND OPENMP MODELS FOR PROTEIN PATTERNS	197

List of Abbreviations

AKRAM	Akram Algorithm
APD	Antimicrobial Peptide Database
ASS	Alpha Skip Shift Algorithm
ASSBR	Alpha Skip Shift and Berry–Ravindran Algorithm
ATT	Algorithms Searching Attempt
BF	Brute Force Algorithm
BM	Boyer Moore Algorithm
bmBc	Boyer Moore Bad Character Function
bmGs	Boyer Moore Good Suffix
BMH	Boyer Moore Horspool Algorithm
BM-KMB	Boyer Moore and Knuth Morris Pratt Algorithm
BR	Berry–Ravindran Algorithm
brBc	Berry–Ravindran Bad Character Function
BRBMH	Berry–Ravindran and Boyer Moore Horspool Algorithm
BRFS	Berry–Ravindran and Fast Search Algorithm
BRQS	Berry–Ravindran and Quick-Search Algorithm
BRSS	Berry–Ravindran and Skip Shift Algorithm
DNA	Deoxyribonucleic Acids
E-R	Entity–relationship model
FJS	Frank, Jennings, and Smyth Algorithm
FS	Fast Search Algorithm
fsBc	Fast Search Bad Character Function
fsGs	Fast Search Good Suffix
HPC	High Performance Computing
hsBc	Boyer Moore Horspool Bad Character Function
JME	Java Molecular Editor
KMP	Knuth Morris Pratt Algorithm

KR	Karp-Rabin Algorithm
KRBMH	Karp-Rabin and Boyer Moore Horspool Algorithm
MIMD	Multiple Instructions, Multiple Data System
MISD	Multiple Instruction, Single Data System
MOD	Modulus Operation in Maths
MPI	Message Passing Interface Model
MRCA	Multiple Reference Character Algorithm
NCBI	National Centre of Biotechnology Information
ncRNAs	non-coding RNAs
NMRShiftDB	Nuclear Magnetic Resonance Shift Database
OE	Odd and Even Algorithm
OpenMP	Open Multi Processing Model
PDB	Protein Data Bank
Protein	Amino Acids
QS	Quick-Search Algorithm
qsBc	Quick-Search Bad Character Function
QSS	Quick-Search and Skip Shift Algorithm
RNA	Ribonucleic Acids
RSMA	Random String Matching Algorithm
SDM	System Development Methodology
SIMD	Single Instruction, Multiple Data System
SISD	Single Instruction, Single Data System
SMILES	Simplified Molecular Input Line Entry System
SS	Skip Shift Algorithm
SSABS	Sheik, Sumit, Anindya, Balakrishnan and Sekar Algorithm
SSN	Shift Skip New Algorithm
TVSBS	Thathoo, Virmani, Sai, Balakrishnan, and Sekar Algorithm
ZT	Zhu Takaoka Algorithm
ztBc	Zhu Takaoka Bad Character Function

CHAPTER 1: INTRODUCTION

1.1 Introduction

Bioinformatics is an important research area for scientific research, involving enormous amounts of data (Degraeve et al., 2002). It covers several fields including biology applications and management of bioinformatics infrastructure using computer science's software (Searls & Hogeweg, 2011). It involves computer science as a discipline giving tools for storing, manipulating, analysis, searching and integration data and for developing applications.

Chemoinformatics is the application of computer software and technology in the field of Chemistry and Pharmacy research to deal with chemical structures and drugs (Xu, 2002). Moreover, string matching algorithms play a vital role in different applications such as bioinformatics and chemoinformatics (SaiKrishna et al., 2012).

Files contain enormous quantities of biological and chemical data presented in a linear string format such as amino acids (proteins), Deoxyribonucleic acids (DNA) and the chemical structures (Horton, 2004).

Great numbers of biological and chemical files are likely to be produced every year, and that is why effective string-matching algorithms are used to reduce the searching response time and the total number of comparisons.

Chapter number one is divided as following: section 1.2 includes the research background, section 1.3 demonstrates research motivation, section 1.4 includes the research hypothesis and the general research methodology, section 1.5 comprises research questions, section 1.6 lists research objectives, section 1.7 includes the main contribution of this research and finally section

1.8 gives an overview of the thesis.

1.2 Background

In this section, we provide a brief explanation of the biological data, DNA, proteins, chemical data representation, antimicrobial structures, sequence databases, biological databases, chemical databases, and string-matching algorithms.

1.2.1 Biological Data

Biological experiments produce various types of data are divided into three main sequences such as DNA, Ribonucleic acid (RNA) and protein. Protein has four different types of structures; primary, secondary, tertiary and quaternary as shown in Figure 1-1 (Bailey, 2006). In this research, DNA and primary protein structures are being focused because they are presented as sequences of string and the quantities of their data files are likely to increase year on year. The next sub-section 1.2.1.1 presents the first and basic biological data DNA followed by sub-section 1.2.1.2 which represents protein sequences.

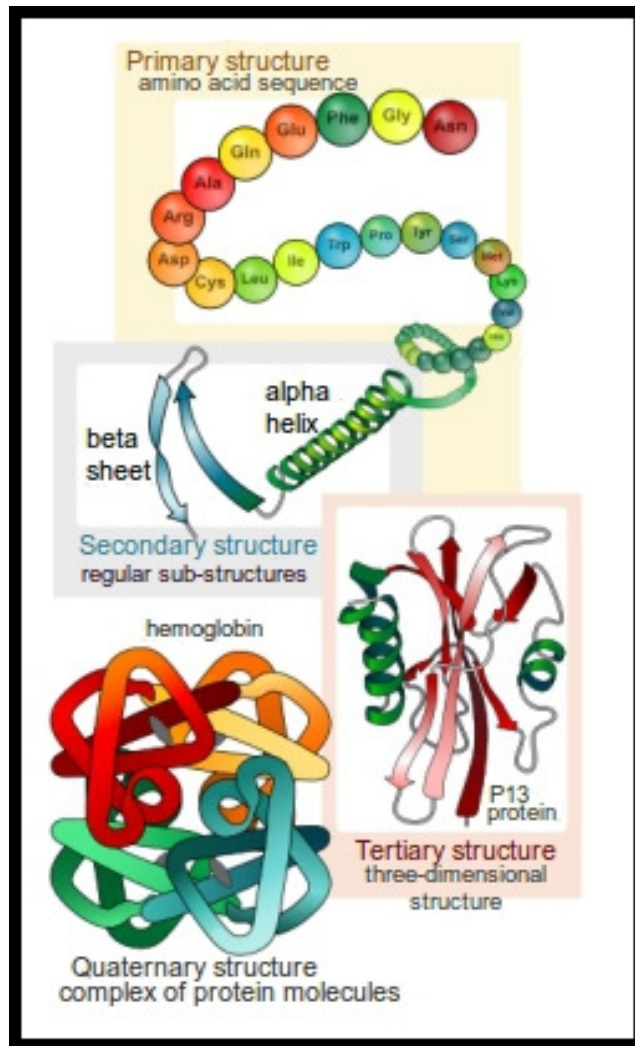


Figure 1- 1: Primary, secondary, tertiary and quaternary structures (Bailey, 2006)

1.2.1.1 DNA

Deoxyribonucleic acid (DNA) is the molecule that stores genetic information. Moreover, James Watson and Francis Crick were the first scientists who proposed the basic structure of DNA in 1953 (Crick, 1974).

DNA is a nucleic acid, made up of a double chain of small molecules called nucleotides. Four different kinds of nucleotides make up a DNA and four bases distinguish these nucleotides. DNA sequences are strings over the alphabet $\Sigma_{\text{DNA}} = \{\text{Adenine (A)}, \text{Cytosine (C)}, \text{Guanine (G)}, \text{Thymine (T)}\}$.

(G), and Thymine(T)} (Kim et al., 2007). According to the U.S. National Library of Medicine, 2013 “There are two DNA bases chained to a phosphate group and a sugar molecule together, and make a base pair like C binds with G and A binds with T” as shown in Figure 1-2.

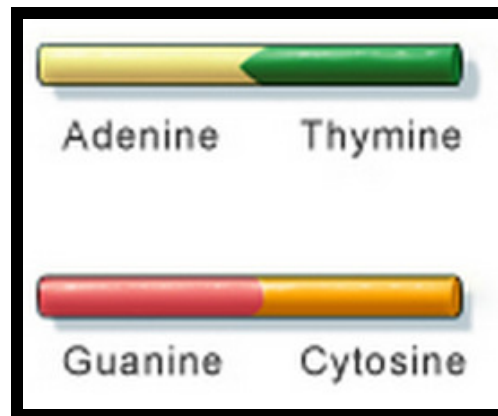


Figure 1- 2: DNA base pairs

Nucleotide is a combination of phosphate group and ribose sugar and they can be arranged in two long strands shaping the double helix where they look like a twisted ladder as shown in Figure 1-3 (Setubal et al., 1997; Agustina, 2012)

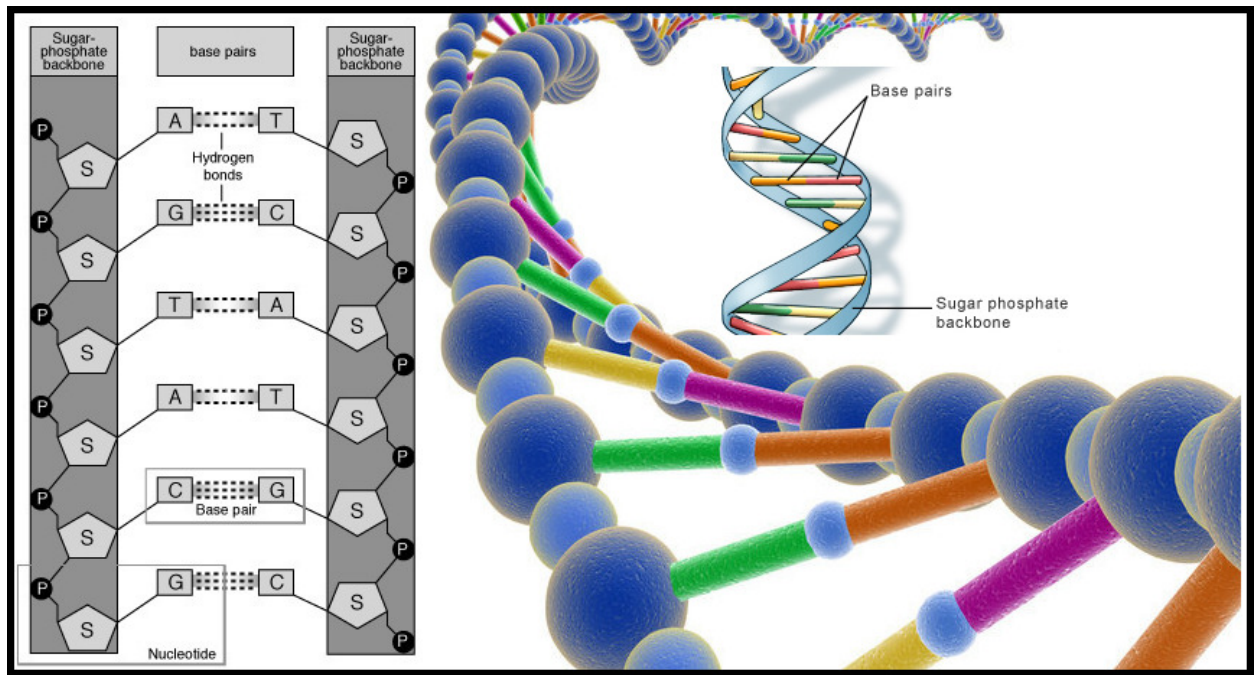


Figure 1- 3: DNA double helix (Setubal et al., 1997; Agustina, 2012)

1.2.1.2 Proteins

The most important contents of the biological databases are proteins. In addition, they are considered as a compound comprising twenty amino acids in string chain linked by peptide bonds (Berg JM, 2002). However, the secondary structure of protein presents the shape of hydrogen bonding built primarily from the primary structure of protein. The tertiary and quaternary structures are globular in shape. The tertiary shows the atomic position in three dimensional space and the quaternary structures are made up of more than one polypeptide chains (Horton, 2004). Furthermore, there are three letter codes and a single letter code of protein as presented in Table 1-1 (Waterman, 1995).

ID	Amino Acid	Three Letter Code	One Letter Code
1	Alanine	Ala	A
2	Arginine	Arg	R
3	Asparagine	Asn	N
4	Aspartate	Asp	D
5	Cysteine	Cys	C
6	Glutamine	Gln	Q
7	Glutamate	Glu	E
8	Glycine	Gly	G
9	Histidine	His	H
10	Isoleucine	Ile	I
11	Leucine	Leu	L
12	Lysine	Lys	K
13	Methionine	Met	M
14	Phenylalanine	Phe	F
15	Proline	Pro	P
16	Serine	Ser	S
17	Threonine	Thr	T
18	Tryptophan	Trp	W
19	Tyrosine	Tyr	Y
20	Valine	Val	V

Table 1- 1: Twenty amino acid abbreviations (Waterman, 1995)

1.2.2 Chemical Data Representation

A fundamental issue in chemoinformatics is the representation of chemical structures on computer systems. The Simplified Molecular Input Line Entry System (SMILES) presents chemical structures in digital databases as a linear string notation (Weininger et al., 1989; Weininger, 1988), rather than the traditional two dimensional structure formula.

1.2.2.1 Antimicrobial Structures

Antimicrobial structures are important substances functioning as self-defense against infection by various harmful pathogens (Fujimura et al., 2003; G Wang, 2010). They exist in all life categories and they found to kill viruses, bacteria and fungi (DeGray et al., 2001; Frece et al., 2004). They can be searched in databases using the antimicrobial structure keywords such as the antibiotics, anticancer, antiviral activity, antifungal activity and antibacterial activity (Wang & Wang, 2009).

1.2.3 Sequence Databases

Biologists and chemists have produced a large number of protein sequences, DNA, RNA, and chemical structures. In addition, the number of sequences uploaded to these databases increases every year. For example, as shown in Figure 1-4 in UniprotKB/Swiss-Prot database (Expasy Bioinformatics Resource Portal, 2013), in 1997 there were fifty thousand entries and in 2005 there were more than one hundred and fifty thousand entries, the size of the database grew three times in ten years. In 2007, more than fifty thousand entries were added in the database within a year. In 2013 there were more than five hundred and thirty thousand entries in the database.

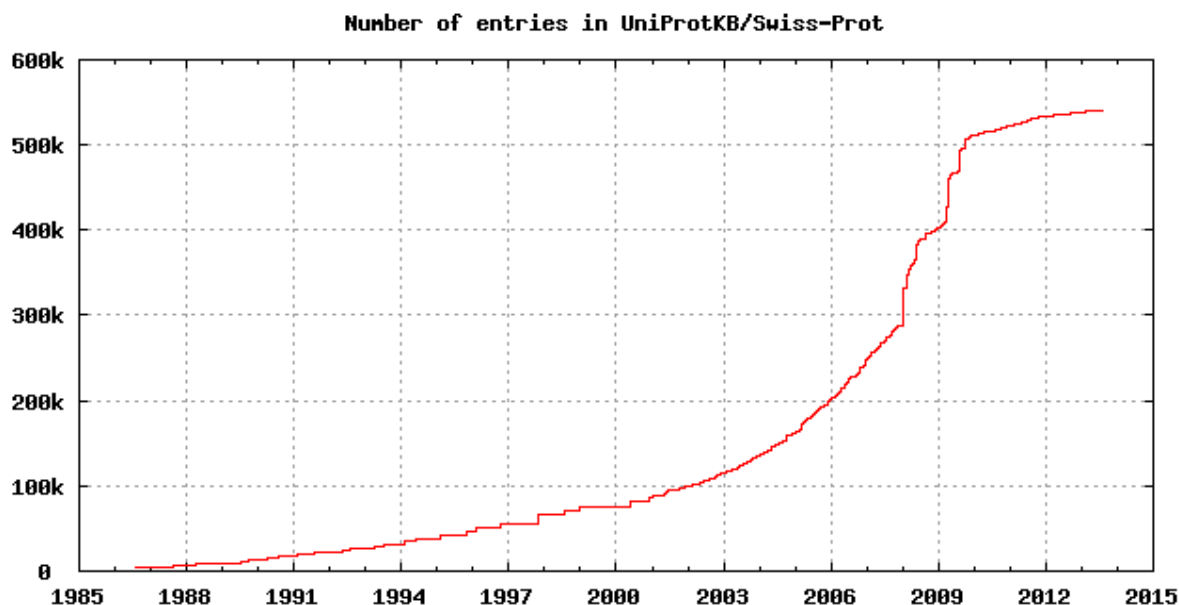


Figure 1- 4: Growth of protein sequence entries in the Swiss-Prot database since 1985 to July 2013 (Expasy Bioinformatics Resource Portal, 2013)

The following sub-sections 1.2.3.1 and 1.2.3.2 lists some of the databases that are used to store the biological and chemical data.

1.2.3.1 Biological Databases

There are a lot of distributed public databases with different aims and contents which are designed to integrate data. The GenBank database is endorsed by the help of the U.S. National Center of Biotechnology Information (NCBI). It comprises millions of DNA sequences (NCBI, 2013). In addition, the SWISS-PROT database contains protein sequences which provide a high percentage of integration, annotation, and the slightest level of repetition comparing to other databases (Bairoch A, 2000) . The Antimicrobial Peptide Database (APD) which maintained by the Department of Pathology and Microbiology at the University of Nebraska USA, contains detailed information for 2426 peptides. In addition, this database combines peptide inquiry,

forecasts, structure and data for a specific group of the peptides (Wang & Wang, 2009). The Protein Data Bank (PDB) database is a globular warehouse of proteins tertiary structures (Berman et al., 2000; Nakamura, 2003; Bourne et al., 2004). The RNAdb is an inclusive non-coding RNAs (ncRNAs) database of warm-blooded animals. Furthermore, it contains more than 800 unique different practically studied non-coding RNAs (Pang et al., 2007).

1.2.3.2 Chemical Databases

A chemical database is developed to store chemical structures' data such as the NMRShiftDB which stores organic structures and their core details. (Kuhn, 2010).

1.2.4 String Matching Algorithms

String matching algorithms take part in solving many computer problems and research issues such as text processing, image processing, signal processing, network security, information retrieval, and speech recognition (Baeza-Yates, 1992; Navarro & Raffinot, 2002; Y. Wang & Kobayashi, 2006; Raju & Babu, 2007; Wang & Li, 2011; SaiKrishna et al., 2012; Bhandari 2014). In addition, they were used widely in computational biology and computational chemistry such as proteins, DNA and RNA searching (Thathoo et al., 2006, Huang et al., 2008; Almazroi, 2011; Naser et al., 2012; Bhandari & Kumar, 2014).

String-matching algorithms target to find a pattern “sequence of characters” with length m $p[p_1 p_2 \dots p_m]$ in a given text with length n $T[t_1 t_2 \dots t_n]$ by matching the text window characters with the pattern characters (Lecroq, 1998; Deusdado & Carvalho, 2009; Sleit et al., 2009) and if

a whole match or a mismatch encountered, the pattern is shifted to the right. An example is to search a pattern “matching” in a text “string matching algorithms”. Text (T) is depicted as $T_1 \dots T_n$ and pattern (P) as $P_1 \dots P_m$ (Navarro & Raffinot, 2000; Fredriksson & Grabowski, 2005; Lecroq, 2007; Lokman & Zain, 2010).

Most algorithms consist of preprocessing and searching phases. The process which controls the pattern shift is called preprocessing phase and it analyses pattern characters to determine the shift value. The process of comparing pattern and text characters is called the searching phase (Chai et al, 2009; Radhakrishna et al., 2010).

Efficient string algorithms aim to maximize the pattern shifting value and therefore enhance the searching time (Chen, 2007). This research proposes new string matching algorithms to search protein sequences, DNA sequences and chemical structures.

1.3 Research Motivation

One of the main research issues in biology and chemistry is searching proteins, DNA, RNA and chemical structures from public databases such as UniProt, SWISS-PROT, APD, PDB, RNAdb and NMRShiftDB databases (EMBL-EBI, 2002). This gives the computer researcher a new research field to help researchers in biological and chemical sciences to use computer technologies, methodologies and capabilities for searching sequences and structures.

1.4 Research Hypothesis and General Research Methodology

Our research is based on the following hypothesis:

- 1- The existing string matching algorithms can be classified in a new way.

- 2- The new classification can be used to understand the mechanism of each searching algorithm.
- 3- The understanding of algorithm mechanism and features can help to enhance current algorithms or developing a new algorithm.
- 4- The enhanced algorithms or developed one(s) can be used to search biological sequences and chemical structures.

These hypotheses are implemented by using the System Development Methodology (SDM) (Nunamaker Jr & Chen, 1990; Morrison & George, 1995; Hevner, 2004). The SDM is used by information system analysts and software developers in order to implement their hypothesis (Abdelaziz et al., 2008; Baydaa, 2011). Figure 1-5 below shows the three main levels of SDM:

- 1- Identifying research problems: problems involve difficulties, conditions or questions where researchers wish to solve these problems, improve conditions or seek to answer questions. In our thesis, the research questions built up depends on classifying string matching algorithms, enhancing one or more of searching algorithms and then applying the one(s) developed to searching biological sequences and chemical structures.
- 2- Prototype and evaluation: in this level the System Development Methodology aims to prototype and implement the suggested work. It starts with designing the model, then identifying model requirements, implementing the model and finally evaluating the model by testing and analyzing results. In this thesis, the designing process and methodology framework are explained in CHAPTER 3. The model developing is presented in CHAPTER 4, the model implementation is presented in CHAPTER 5 and the testing evaluation and analysis part is presented in CHAPTER 6.

- 3- Conceptual and practical contributions: This level defines the main contribution to the knowledge. According to the results gained from the evaluation, the discussion is presented in section 6.7 and finally, the conclusion and the future work are presented in CHAPTER 7.

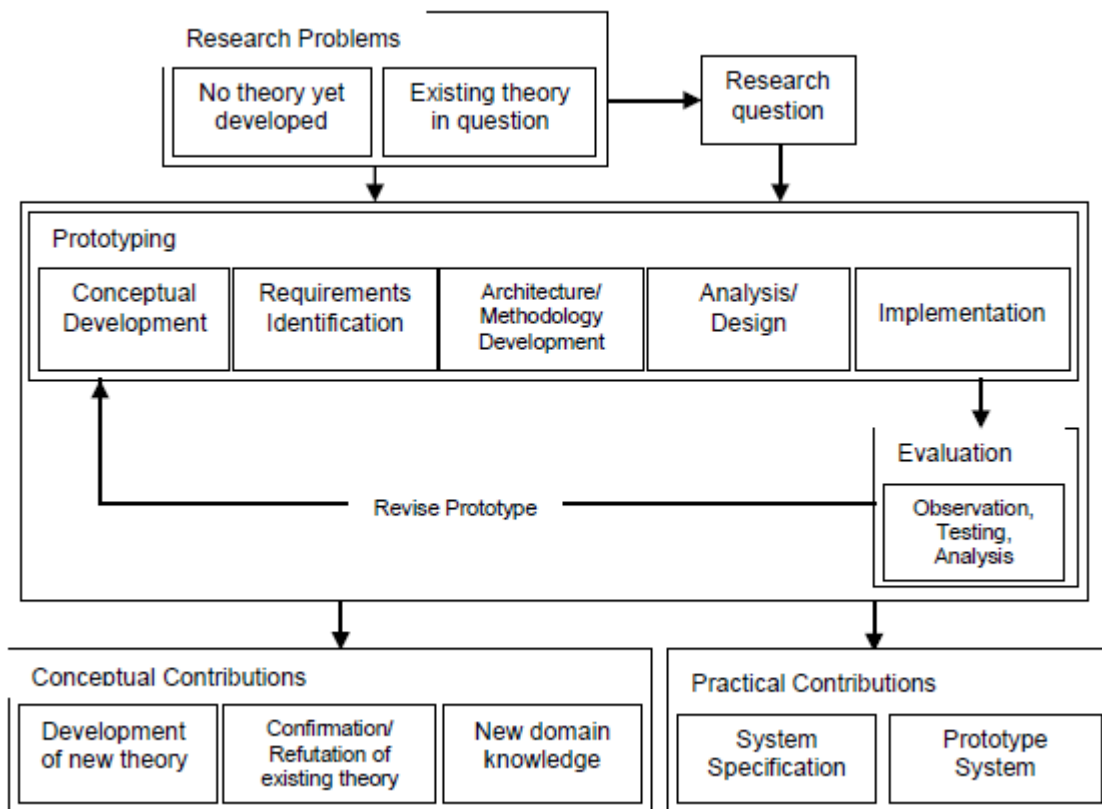


Figure 1- 5: System development methodology (Morrison & George, 1995)

1.5 Research Questions

- 1 - Which of the existing string pattern matching algorithm(s) is/are the most suitable for searching biological sequences and chemical structures?
- 2 - Can we enhance one or more of the proposed algorithms in 1, or develop (a) new algorithm(s) for string-matching?

- 3 - How we can measure the success of the newly developed algorithm(s) compared to the best algorithm in 1?
- 4 - Can we develop a classification of string matching algorithms which will help with achieving our aims?

1.6 Research Objectives

In this section, we explain the main parameters and factors that we intend to include in our study. Thus our study includes the following objectives:

- To study the existing string-matching algorithms in order to develop a taxonomy of such algorithms.
- To apply insights gained in the previous phase to enhance one or more of the existing algorithms, or to develop (a) new algorithm(s) to search biological sequences and chemical structures.
- To measure the success of the newly developed algorithm(s) compared to currently existing algorithms.

1.7 Main Contribution

The main contributions for this research are:

1. Classifying the main string matching algorithms into a new classification containing eight categories according to the preprocessing function in the algorithm.
2. Enhancing and developing five new string matching algorithms which improve the

searching response time by decreasing the number of comparisons in the searching stage and enhancing the preprocessing stage by maximizing the shifting value.

3. Applying enhanced and developed algorithms to search DNA and protein sequences.
4. Implementing a local database containing relational tables to store downloaded chemical structures from NMRShiftDB.
5. Using the fastest algorithm to develop a searching toolkit aims to search chemical structures in a local database.
6. Speeding up the fastest algorithm using parallel models.

1.8 Overview of the Thesis

This chapter (CHAPTER 1: INTRODUCTION) gives an overview of the background to the study which includes biological data such as DNA and proteins, chemical data representation such as antimicrobial structures, biological sequence databases, chemical databases and string matching algorithms. Then it deals with the research motivation, the research hypothesis, general research methodology, the research questions and ends with the research objectives and expected the outcomes of this study.

CHAPTER 2: A CURRENT STATE OF THE ART gives a survey, a new classification and summary of string matching algorithms. In addition, chapter 2 describes Simplified Molecular Input Line Entry System format (SMILES) with the syntax rules and finally it presents a discussion of parallel computing.

CHAPTER 3: METHODOLOGY AND DESIGN. This chapter gives an overview of the proposed work in this study which includes the research methodology framework. In addition, it includes the chemical structure toolkit design and the parallel algorithm design. The research methodology framework includes six main methodology stages which aim to achieve our research objectives. The chemical toolkit design includes four stages which aim to develop the toolkit in order to search chemical structures in our local database. Finally the parallel algorithm design includes two phases of parallelizing the SSN algorithm; the first phase parallelizes the SSN algorithm using the OpenMP model and the second phase uses the MPI model.

CHAPTER 4: DEVELOPING NEW ALGORITHMS. In this chapter, after classifying the main string matching algorithms in CHAPTER 2 into eight categories according to the preprocessing function in each algorithm, five new algorithms are developed which aim to maximize the pattern shifting value and therefore enhance the searching time.

CHAPTER 5: IMPLEMENTATION. In this chapter, the chemical structure toolkit has been implemented. This includes four stages. The first stage includes downloading and mining structures from NMRShiftDB. The second stage builds a local database to host structures. The third stage connects the toolkit to the local database and searches structures using Java Molecule Editor (JME), SMILES and the SSN algorithm. Finally, the proportion of matching characters is used in the fourth stage to measure the similarity between structures.

CHAPTER 6: RESULTS AND DISCUSSION. In this chapter all developed algorithms and some of the standard algorithms are implemented and tested. The chemical searching toolkit has

been tested in this chapter as well. A parallel version of the SSN algorithm is implemented and tested using OpenMP and MPI models. Finally, the discussion section analyses the results of three types of tests are implemented on the developed algorithms and other standard algorithms.

CHAPTER 7: CONCLUSION AND FUTURE WORK. This chapter gives a summary of research contribution and results. It also suggests some future work that can be used to expand the current research.

CHAPTER 2: A CURRENT STATE OF THE ART

In this chapter the main string matching algorithms are classified into eight categories according to the preprocessing function in the algorithm (Klaib & Osborne, 2009a). The first category shifts the pattern only one position at each attempt (section 2.2). The second category uses two preprocessing functions to shift the pattern (section 2.3). The third category uses one preprocessing function based on the rightmost character in the current window (section 2.4). The fourth category uses one preprocessing function based on the next character to the current window (section 2.5). The fifth category uses one preprocessing function based on the two characters next to the current window (section 2.6). The sixth category uses a preprocessing hashing function (section 2.7). The seventh category uses a single preprocessing function depends on computing buckets for all characters of the alphabet (section 2.8). The final category uses hybrid algorithms (section 2.9).

A summary describing and comparing all previous classification of algorithms is presented in section 2.10. Section 2.11 describes the SMILES format for chemical structures and finally section 2.12 contains a brief discussion of parallel computing concept.

2.1 Conventions

In this discussion, (T) is the text, (P) is the pattern, (m) is the pattern's length, (n) is the text's length, and the size of the alphabet used in T is σ .

2.2 The First Category: Shift the Pattern a Single Position

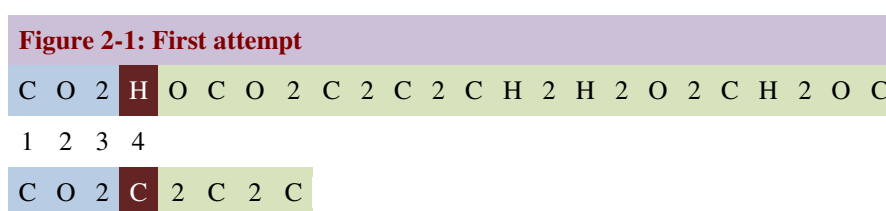
In this group, the pattern is shifted a single position whether there is a whole match or a

mismatch. The Brute Force algorithm (BF) is an example of this classification (Navarro & Raffinot, 2002).

2.2.1 The Brute Force Algorithm (BF)

The BF is the basic algorithm of searching algorithms and there is no preprocessing phase is used to shift the pattern. The searching order starts from the leftmost character moving forward to the rightmost character of both the text window and pattern characters, and if there is a whole match or a mismatch it shifts the pattern only a single position.

The disadvantage of this algorithm is the low efficiency by going through all characters of the string (Charras & Lecroq, 2004). This algorithm can work well with small strings, but not with large strings such as those in biological and chemical data (Stephen, 1994; Levitin, 2008). Figure 2-1 below shows an example which illustrates the main principles of the Brute Force algorithm where in each attempt the first line presents the text characters, the second line presents the search order and the third line shows the search pattern characters. This convention will be used for all examples.



In the first attempt, the first three characters have found a match. The fourth comparison produces a mismatch which causes a shift to the right by one position and starts the search again from position two as shown in the second attempt and so on.

Figure 2-1: Second attempt

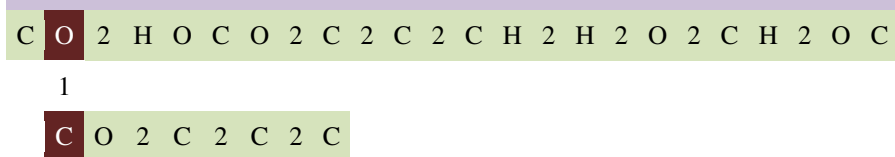


Figure 2-1: Third attempt

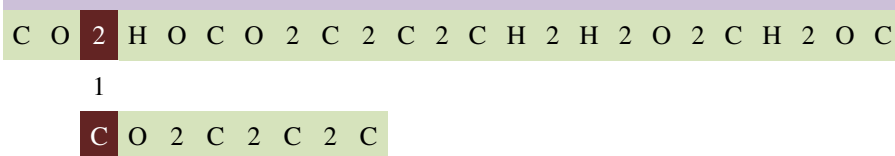


Figure 2-1: Fourth attempt

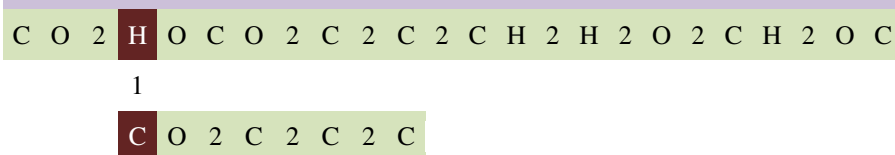


Figure 2-1: Fifth attempt

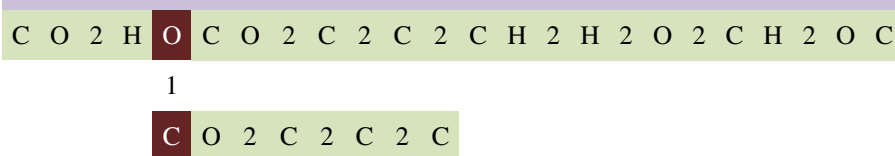


Figure 2-1: Sixth attempt

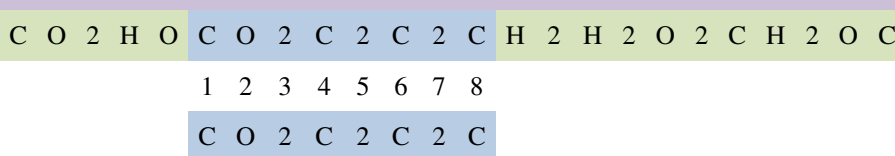


Figure 2-1: Seventh attempt

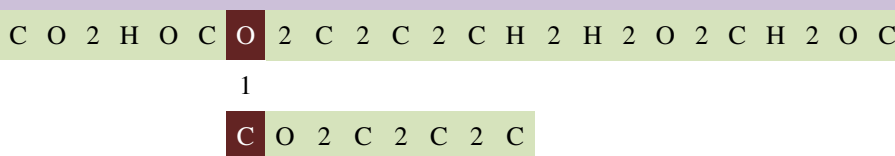


Figure 2-1: Eighth attempt

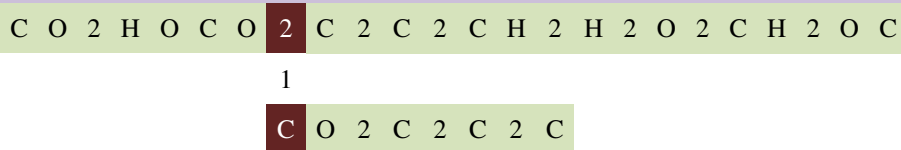


Figure 2-1: Ninth attempt

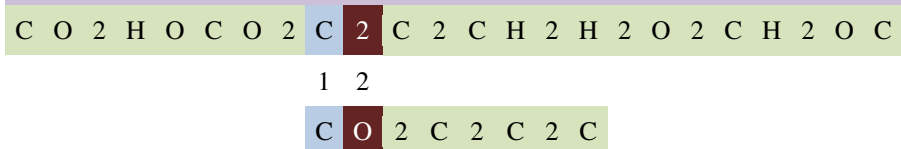


Figure 2-1: Tenth attempt

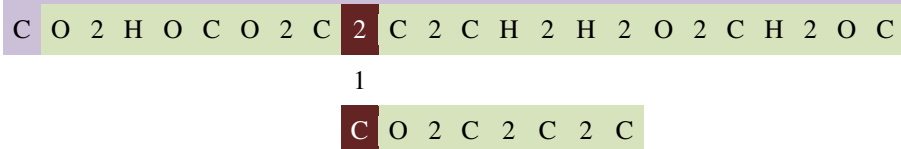


Figure 2-1: Eleventh attempt



Figure 2-1: Twelfth attempt

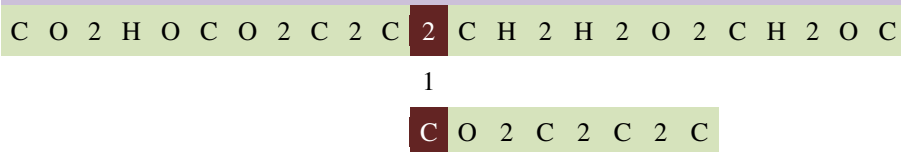
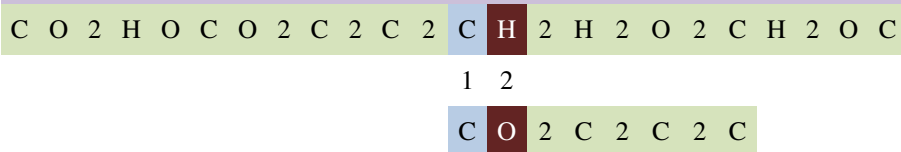


Figure 2-1: Thirteenth attempt



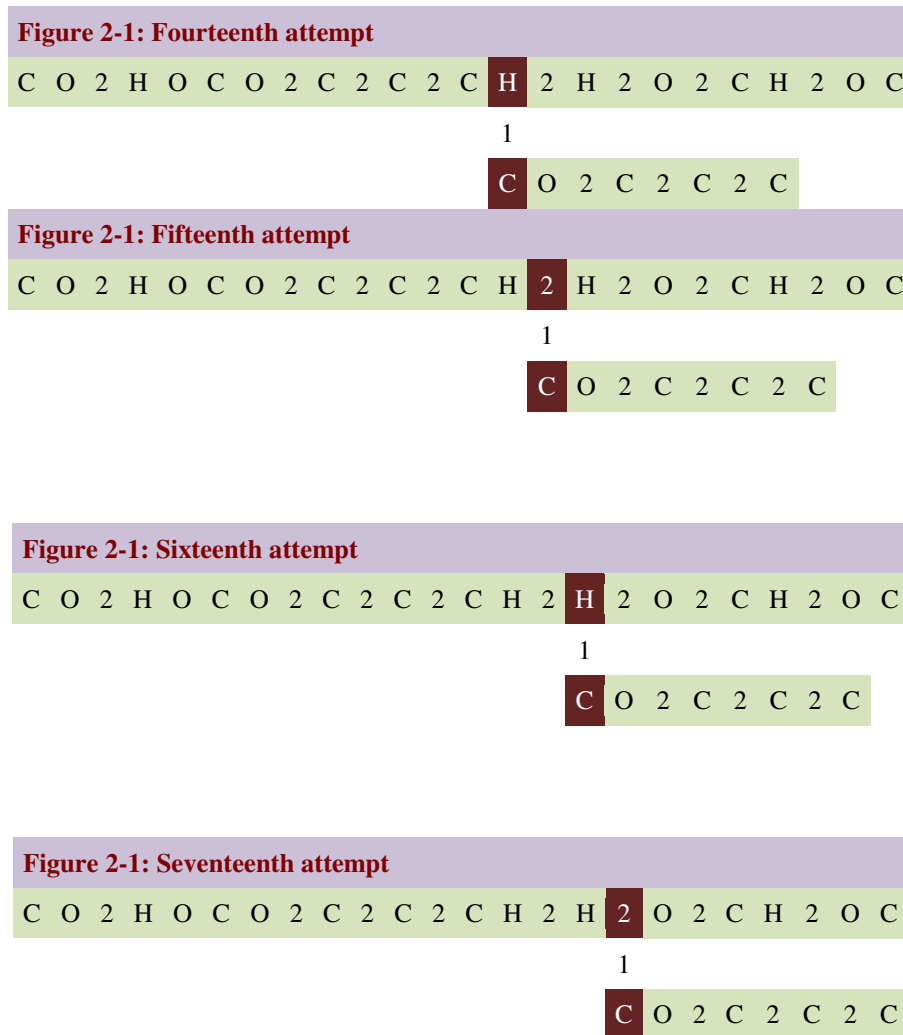


Figure 2- 1: The Brute Force algorithm example

In the previous example, the Brute Force algorithm performs seventeen attempts and thirty character comparisons to find the pattern in the text. Figure A-1 shows the Brute Force algorithm code (Charras & Lecroq, 1997).

2.3 The Second Category: Using Two Preprocessing Functions

In this group, whether there is a whole match or a mismatch, the pattern is shifted using two preprocessing functions. Examples of this group are the Boyer-Moore algorithm (BM) (Boyer & Moore, 1977), the Zhu Takaoka algorithm (ZT) (Feng & Takaoka, 1988) and the Fast Search

Algorithm (FS) (Cantone & Faro, 2003).

2.3.1 The Boyer Moore Algorithm (BM)

The BM algorithm was developed by R.S. Boyer and J.S. Moore in 1977(Boyer & Moore, 1977; J. Lee, 2004). It searches the pattern from the rightmost character to the leftmost character in each attempt. If there is a whole match or a mismatch, two preprocessing functions are used to shift the pattern by n positions. The first preprocessing function is named the bad-character function (bmBc) and the second one the good-suffix function (bmGs) (Wu & Manber, 1994; Fredriksson & Grabowski, 2005; Danvy & Rohde, 2006).

A bmBc is applied when the mismatch is caused by a text character that exists in different position in the pattern. And in this case, it shifts the pattern to align similar characters and start a new attempt. Table 2-1 shows an example for the bad-character case (Charras & Lecroq, 2004):

0	1	2	3	4	5	6	7	8	9	10	11
H	U	D	D	E	R	S	F	I	E	L	D
F	I	E	L	D							
		F	I	E	L	D					

Table 2- 1: Bad-character shift in Boyer-Moore algorithm example

A mismatch found between characters E and D at position 4. The text character E can be found in pattern characters at position 2. The bad-character function shifts the pattern to the right to align the text character E with the same character in the pattern which exists at position 4.

The bmGs function is used if the mismatch text character is not the first character and the

matched substring “suffix” exists in pattern characters, it shifts the pattern to align similar characters between the suffix and pattern characters (Charras & Lecroq, 2004).

The best case for the BM algorithm occurs if the first compared text character does not exist in the pattern characters, so in this case the algorithm needs only $O(n/m)$ comparisons. Table 2-2 below shows an example for this situation(Charras & Lecroq, 2004):

0	1	2	3	4	5	6	7	8	9	10	11
H	U	D	D	X	R	S	F	I	E	L	D
F	I	E	L	D							
					F	I	E	L	D		

Table 2- 2: The best case example of the Boyer-Moore algorithm

A mismatch is found between characters X and D at position 4 and X does not exist in the pattern. Therefore, it shifts the pattern to start from the next position to X at position number 5.

The worst case for the BM searching algorithm is $O(mn)$ and this happens if the text consists solely of a number of repetitions of the search pattern (Tsai, 2006). If the text alphabet is small, then the BM bad-character shift is not very efficient (Crochemore et al., 1994; Lecroq, 1995).

The time complexity of the bmBc function is $O(m+\sigma)$, the bmGs is $O(m^2)$ and of the average searching phase is $O(mn)$ (Charras & Lecroq, 2004). Figure 2-2 below shows an example illustrating the main principles of the Boyer-Moore algorithm, Table 2-3 shows the bmBc table and Table 2-4 shows the bmGs table:

Character	C	2	O	H
bmBc[character]	2	1	6	8

Table 2- 3: The Boyer-Moore bmBc table

The bmBc table in Table 2-3 shows the rightmost occurrence of each character in the pattern, while the bmGs table in Table 2-4 shows the maximum shift distance from the structure of the pattern that can be used in good suffix cases to shift the pattern to the right. In Table 2-3, the rightmost occurrence of character 'C' is 2, character '2' is 1, character 'O' is 6 and character 'H' is 8. The occurrence of character 'H' is the same length of the pattern which should be shifted 8 positions to the right if character 'H' is causing the mismatch. In Table 2-4, the suffix "2C" starts at positions 2, 4, and 6. The suffix "C2C" starts at positions 3 and 5. If the mismatch occurs at position 5, then pattern is shifted 4 positions. If the mismatch occurs at position 3, then the pattern is shifted 2 positions. If the mismatch occurs at position 4, then the pattern is shifted 3 positions, otherwise the suffix does not exist in the text window and the shift value is 7.

Pattern Array Index	0	1	2	3	4	5	6	7
Pattern Array [Index]	C	O	2	C	2	C	2	C
bmGs[i]	7	7	7	2	3	4	7	1

Table 2- 4: Boyer-Moore *bmGs* table

Figure 2-2: First attempt																							
C	O	2	H	O	C	O	2	C	2	C	2	C	H	2	H	2	O	2	C	H	2	O	C
							1																
C	O	2	C	2	C	2	C																

Total Shift Value = 1 (*bmBc*[2])

In the first attempt, the first comparison produces a mismatch which causes a shift to the

The second attempt, the first two characters have found a match. The third comparison produces a mismatch which causes a shift to the right using the rightmost occurrence of character 'O' in the text window. It aligns similar characters and starts new comparisons.

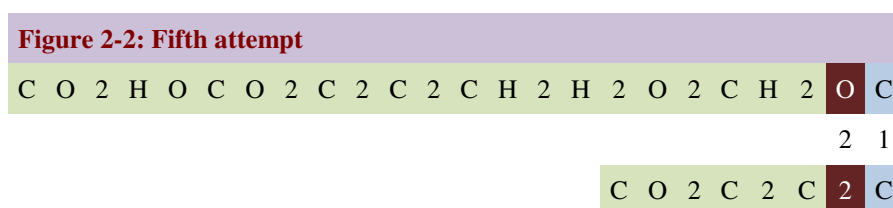
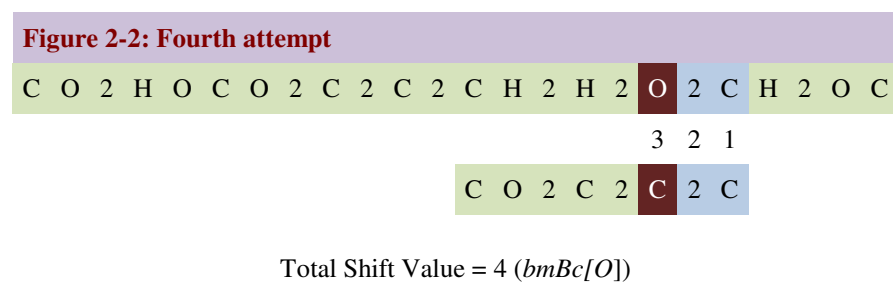
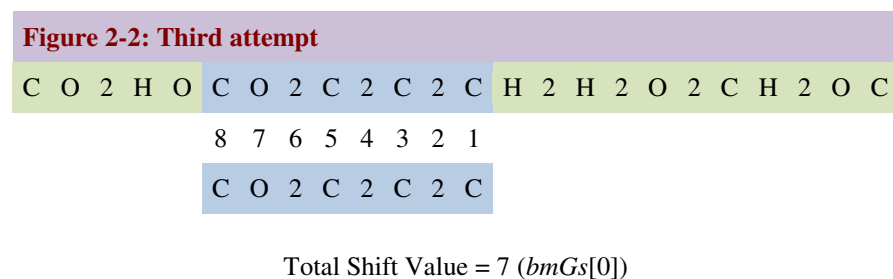


Figure 2- 2: The Boyer-Moore algorithm example

In this example the Boyer Moore algorithm performs five attempts and seventeen character comparisons to find the pattern in the text. Figure A-2 shows the Boyer-Moore algorithm code (Charras & Lecroq, 1997).

2.3.2 The Zhu Takaoka Algorithm (ZT)

Another example of this classification is the ZT algorithm which was developed by R. F. Zhu and T. Takaoka in 1988 ((Zhu & Takaoka, 1987). It uses the same rules as the bmGs preprocessing function and improves only the bmBc function by shifting the pattern using the last two characters of each text window (a, b) rather than using a single character same as the bmBc function (Kalsi, et al., 2008). This is done by constructing the Zhu-Takaoka bad character table (ztBc) table.

The ztBc table counts the shifting value of each pair of characters (a, b) as following Figure 2-3 (Zhu & Takaoka, 1987).

$$\text{For } a, b \in \Sigma$$

$$ztBc[a, b] = k \Leftrightarrow \begin{cases} k \leq m-2 & \text{and } x[m-k..m-k+1] = ab \\ & \text{and } ab \text{ does not occur} \\ & \text{in } x[m-k+2..m-2], \\ k = m-1 & x[0] = b \text{ and } ab \text{ does not occur} \\ & \text{in } x[0..m-2], \\ k = m & x[0] \neq b \text{ and } ab \text{ does not occur} \\ & \text{in } x[0..m-2] \end{cases} \dots\dots\dots(1)$$

Figure 2- 3: Zhu Takaoka ztBc equation

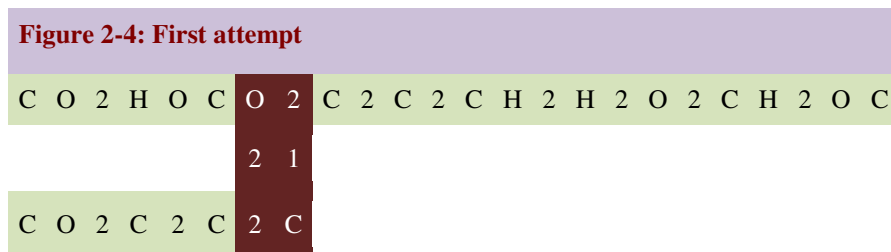
The ZT searching phase searches the pattern from the rightmost to the leftmost character in each attempt. The time complexity of the ztBc is $O(m+\sigma^2)$, the bmGs is $O(m^2)$ and of the searching phase is $O(mn)$ (Zhu & Takaoka, 1987). Figure 2-4 below shows an example illustrating the main principles of the Zhu Takaoka algorithm, Table 2-5 shows the ztBc and

Table 2-6 shows the bmGs table:

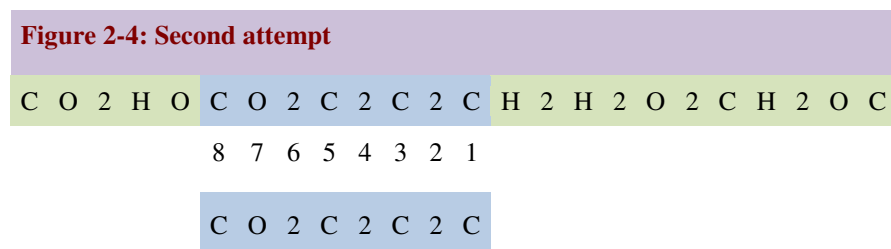
Character	2	O	C	H
2	8	8	2	8
O	5	8	7	8
C	1	6	7	8
H	8	8	7	8

Table 2- 5: The Zhu-Takaoka ztBc table

The ZT algorithm uses the bmGs table of the BM algorithm as shown in Table 2-4 and enhanced the bmBc table by using the rightmost occurrence of each pair of characters as shown in Table 2-5. In this example the rightmost occurrence of each pair of characters [C2] is 1, [2C] is 2, [O2] is 5 and [CO] is 6. The shift value for any pair ends with character ‘C’ is 7 because it is the first character in the pattern while other pairs will be shifted 8 positions because they do not exist in the pattern.



Total Shift Value = 5 (*ztBc*[O][2])



Total Shift Value = 7 (*bmGs*[0])

$\text{C O 2 H O C O 2 C 2 C 2 C H 2 H 2 O 2 C H 2 O C}$
 3 2 1
 C O 2 C 2 C 2 C

Figure 2- 4: The Zhu-Takaoka algorithm example

2.3.3 The Fast Search Algorithm (FS)

[illegible]

48

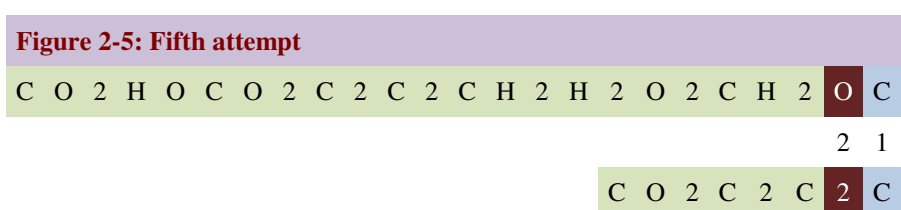
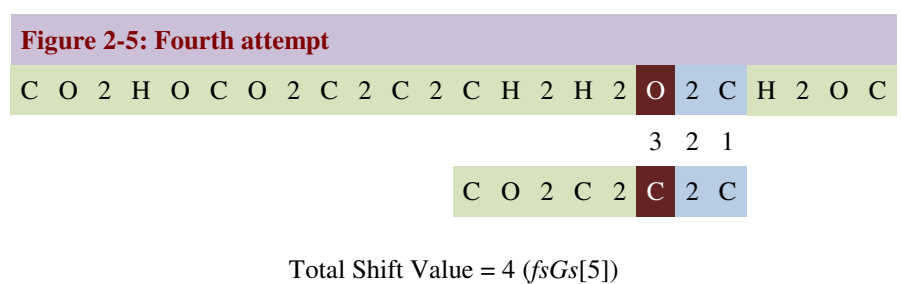
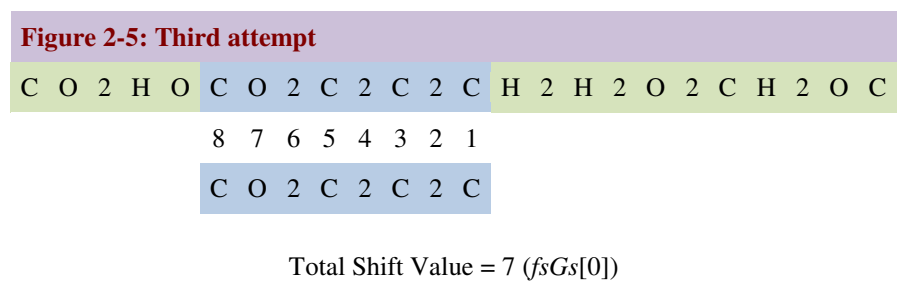
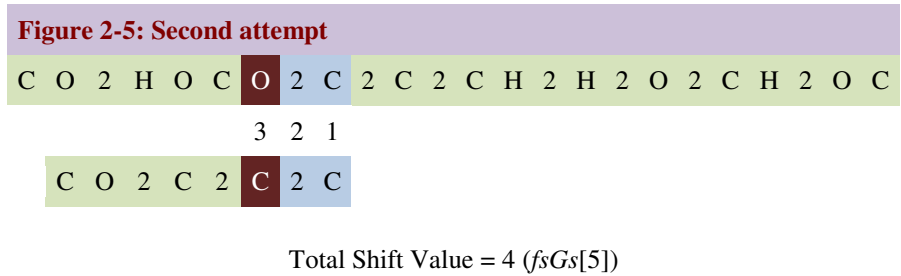


Figure 2- 5: The Fast Search algorithm example

In this example the Fast Search algorithm performs five attempts and seventeen character comparisons to find the pattern in the text. Figure A-4 shows the Fast Search algorithm code (Cantone & Faro, 2003).

2.4 The Third Category: Depending on the Rightmost Character

In this group, the pattern is shifted using a single preprocessing function depending on the last character. The Boyer Moore Horspool algorithm (BMH) is an example (Horspool, 1980).

2.4.1 The Boyer Moore Horspool Algorithm (BMH)

The BMH algorithm was developed by N. Horspool in 1980. It is a modification of the BM algorithm (Tarhio & Peltola, 1997). It is faster than the BM algorithm and the preprocessing function computes the shifts using only one heuristic function depending on the last character comparing to the BM algorithm (Crochemore & Rytter, 1994). The searching phase of the BMH algorithm starts from the rightmost character, then starts from the leftmost character and then moves forward up to the penultimate character (Raita, 1992).

Table 2-6 below shows an example for the Boyer-More algorithm and compares it with Horspool algorithm in Table 2-7 (Charras & Lecroq, 2004):

0	1	2	3	4	5	6	7	8	9	10	11
H	U	D	D	E	R	S	F	I	E	L	D
H	D	D	D	E							
	H	D	D	D	E						

Table 2- 6: An example of the bmBc function

0	1	2	3	4	5	6	7	8	9	10	11
H	U	D	D	E	R	S	F	I	E	L	D
H	D	D	D	E							
					H	U	D	D	E		

Table 2- 7: An example of hrBc function

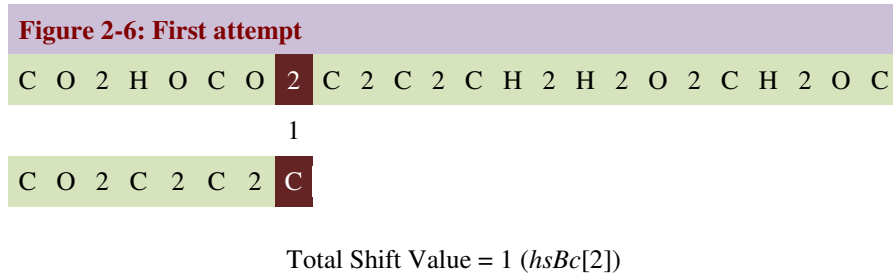
In this example, the current text window starts from position 0 to position 4. The BM algorithm in Table 2-6 starts the comparison from the rightmost character to the leftmost character. The fourth comparison produces a mismatch at position 1. The bmBc function of the BM algorithm uses the letter D at position 1 to shift the pattern to the right and align character D with the same character which exists at position 2. The BMH algorithm in Table 2-7 starts from the rightmost character, then starts from the leftmost character and then moves forward up to the penultimate character. The third comparison produces a mismatch at position 1. The hrBc function of the BMH algorithm shifts the pattern to position number 5 depending on the last character of the current text window ‘E’ which does not exist in the pattern in this example.

“The average time complexity of the preprocessing phase is $O(m+\sigma)$ and of the searching phase is $O(mn)$ ” (Regnier & Szpankowski, 1998). Figure 2-6 below shows an example illustrating the main principles of the Horspool algorithm and Table 2-8 shows the Horspool algorithm bad character table (hsBc):

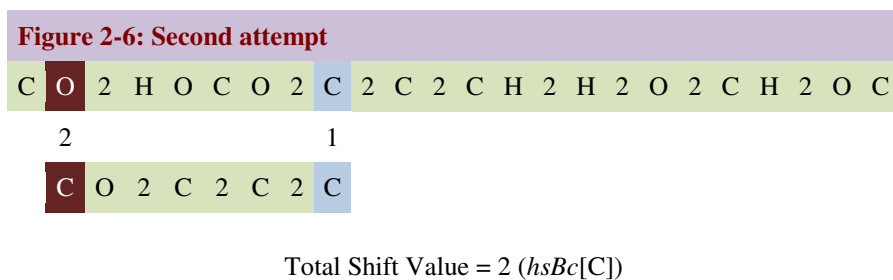
Character	C	2	O	H
hsBc[character]	2	1	6	8

Table 2- 8: Horspool hsBc table

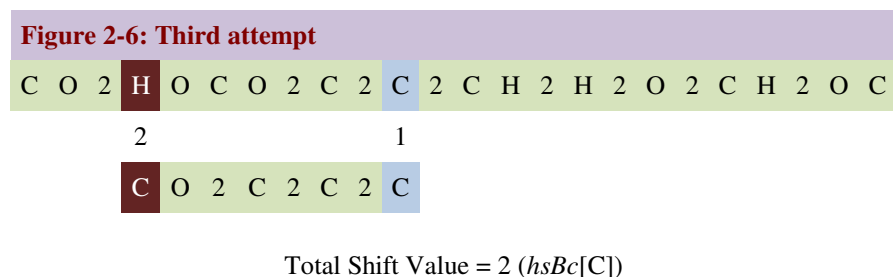
The *hsBc* table in Table 2-8 follows the same way as the *bmBc* table which shows the rightmost occurrence of each character in the pattern.



In the first attempt, the first character comparison produces a mismatch which causes a shift to the right by one position depends on the last character of current text window '2'.



In the second attempt, the first comparison (rightmost character) has found a match. The second comparison (leftmost character) produces a mismatch which causes a shift to the right by two positions depends on the last character of current text window 'C'. It starts the search again in other attempts following the same procedure.



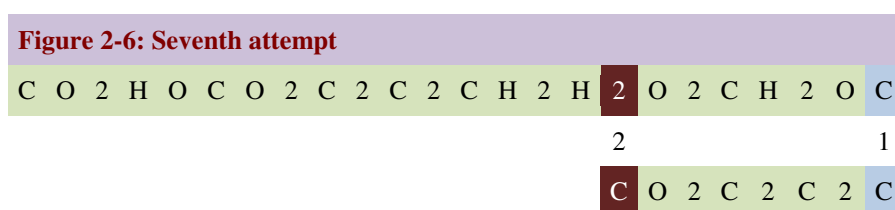
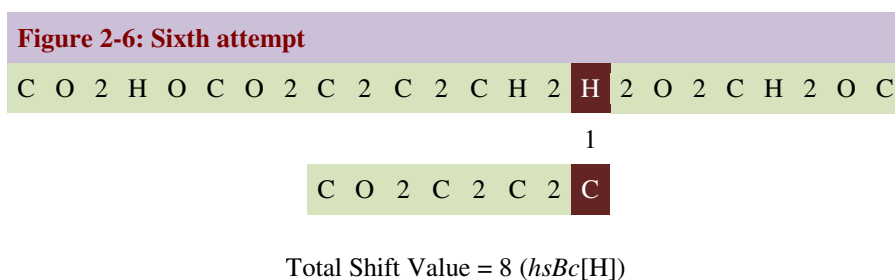
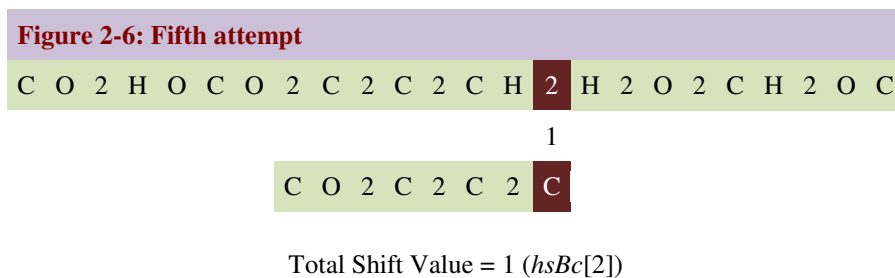
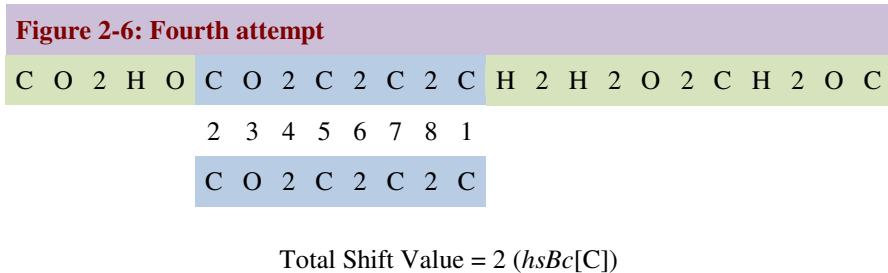


Figure 2- 6: The Horspool algorithm example

In this example the Horspool algorithm performs seven attempts and seventeen character comparisons to find the pattern in the text. Figure A-5 shows the Horspool algorithm code (Charras & Lecroq, 1997).

2.5 The Fourth Category: Depending on the Next Character to the Rightmost Character

In this group, the pattern is shifted using a single preprocessing function depending on the next character to the rightmost character. An example is the Quick Search algorithm (QS) (Sunday, 1990).

2.5.1 The Quick-Search Algorithm (QS)

The QS algorithm was developed by D. M. Sunday in 1990. The preprocessing phase of QS (qsBc) shifts the pattern by $m+1$ if the next character to the rightmost character does not exist in the pattern. The searching phase searches the pattern from leftmost character to the rightmost character (Sunday, 1990; Lecroq, 1998). “The time complexity of the preprocessing phase is $O(m+\sigma)$ and of the searching phase is $O(mn)$ ” (Charras & Lecroq, 2004). Figure 2-7 below shows an example illustrating the main principles of Quick-Search algorithm and Table 2-9 shows the preprocessing qsBc table:

Character	C	2	O	H
qsBc[character]	1	2	7	9

Table 2- 9: Quick Search qsBc table

The qsBc table in Table 2-8 follows the same way as the bmBc table which shows the rightmost occurrence of each character in the pattern.

Figure 2-7: First attempt

C	O	2	H	O	C	O	2	C	2	C	2	C	H	2	H	2	O	2	C	H	2	O	C
1	2	3	4																				
C	O	2	C	2	C	2	C																

Total Shift Value = 1 ($qsBc[C]$)

In the first attempt, the first three comparisons (leftmost characters) have found a match. The fourth comparison produces a mismatch which causes a shift to the right by one position depends on the next character to the current text window 'C'. It starts the search again in other attempts following the same procedure.

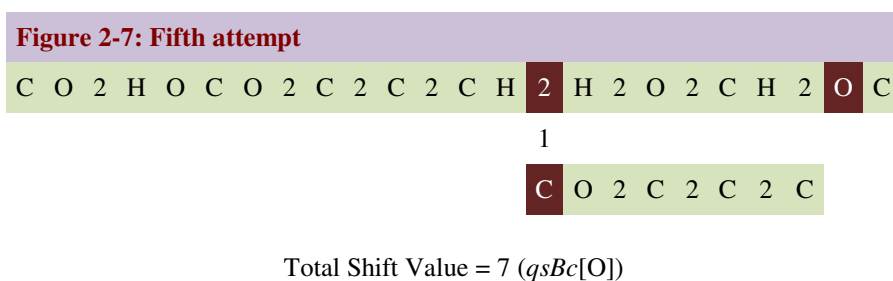
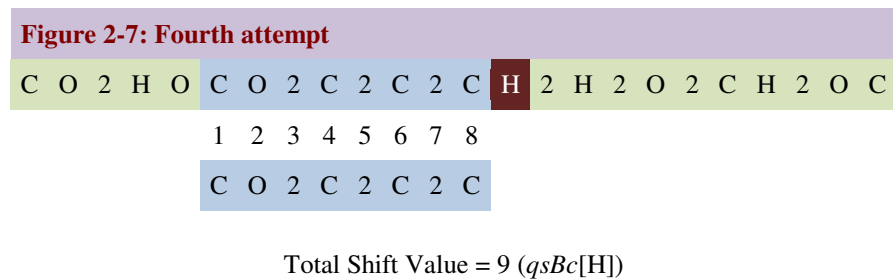
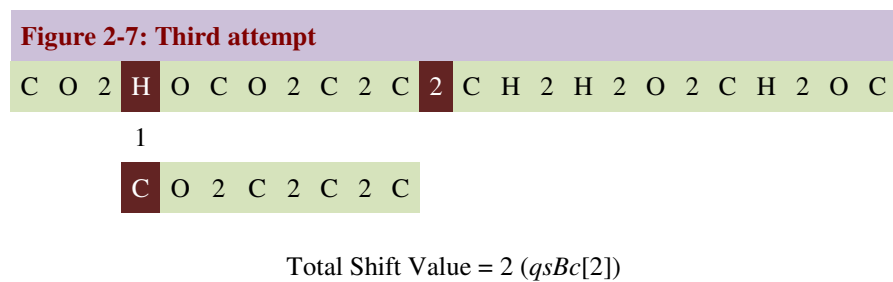
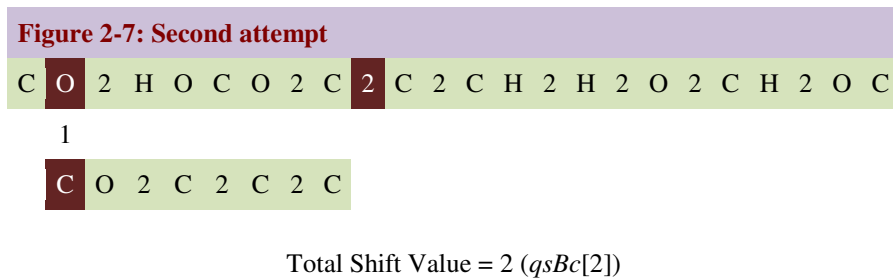


Figure 2- 7: The Quick Search algorithm example

Applying the Quick Search algorithm on the above example performs five attempts and fifteen character comparisons to find the pattern in the text. Figure A-6 shows the Quick-Search algorithm code (Charras & Lecroq, 1997).

2.6 The Fifth Category: Depending on Two Characters Next to the Rightmost Character

In this group, the pattern is shifted using a single preprocessing function depending on two characters next to the rightmost character. The Berry–Ravindran algorithm (BR) is an example (Berry & Ravindran, 1999).

2.6.1 The Berry–Ravindran Algorithm (BR)

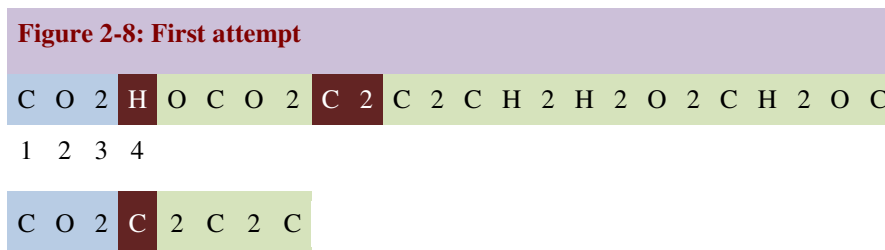
The BR algorithm was developed by T. Berry and S. Ravindran in 1999. The searching phase of the BR algorithm searches the pattern from the leftmost to the rightmost character. In addition, the preprocessing phase uses a two-dimensional array to shift the pattern by $m+2$ if the next two characters to the rightmost character do not exist in the pattern (Thathoo et al., 2006).

“The time complexity is $O(m+\sigma^2)$ for the preprocessing phase and $O(mn)$ for the searching phase” (Charras & Lecroq, 2004). Figure 2-8 below shows an example that illustrates the main principles of Berry-Ravindran algorithm and Table 2-10 shows the Berry Ravindran algorithm bad character table (brBc):

Character	2	O	C	H
2	10	10	2	10
O	7	10	9	10
C	1	1	1	1
H	10	10	9	10

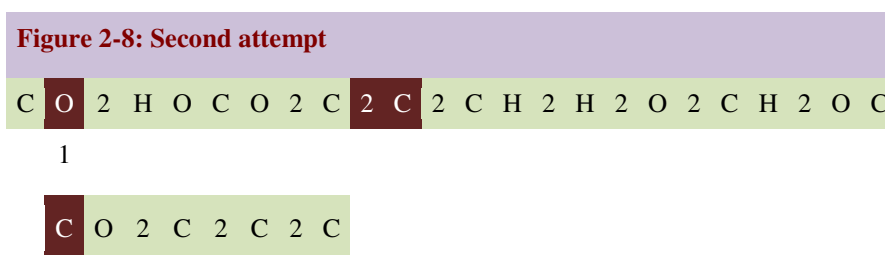
Table 2- 10: The Berry-Ravindran brBc table

The brBc table in Table 2-10 calculates the shift value using the next two characters to the current text window. If the next two characters do not exist in the pattern, then the shift value is $m+2$ and in this example is 10 positions. The rightmost occurrence of characters [2C] is 2 and of characters [O2] is 7. In the following example the pattern starts and ends with character 'C', so if the next character to the current text window is 'C', the shift value is 1. Otherwise if the second character next to the current text window is 'C' then the shift value is 9.



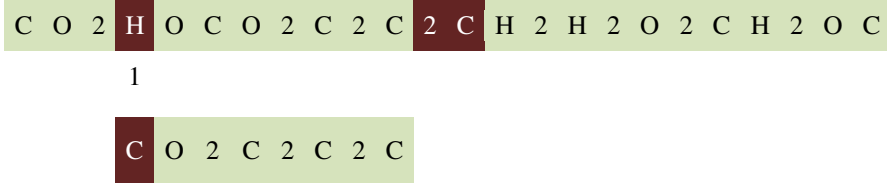
Total Shift Value = 1 ($brBc[C][2]$)

In the first attempt, the first three comparisons (leftmost characters) have found a match. The fourth comparison produces a mismatch which causes a shift to the right by one position depends on the next two characters to the current text window [C2]. It starts the search again in other attempts following the same procedure.



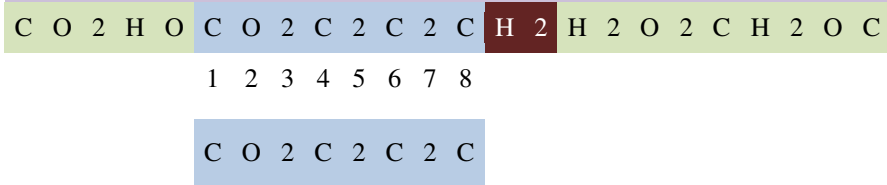
Total Shift Value = 2 ($brBc[2][C]$)

Figure 2-8: Third attempt



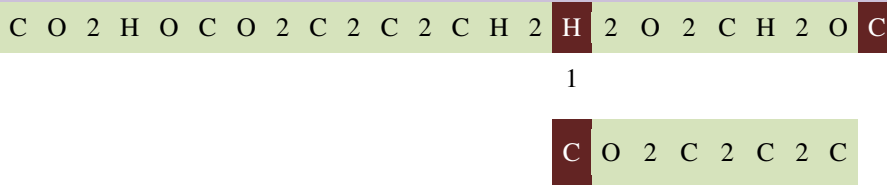
Total Shift Value = 2 ($brBc[2][C]$)

Figure 2-8: Fourth attempt



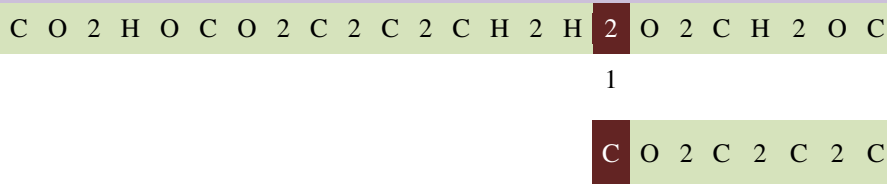
Total Shift Value = 10 ($brBc[H][2]$)

Figure 2-8: Fifth attempt



Total Shift Value = 1 ($brBc[C][0]$)

Figure 2-8: Sixth attempt



Total Shift Value = 10 ($brBc[0][0]$)

Figure 2- 8: The Berry-Ravindran algorithm example

Applying the Berry-Ravindran algorithm on the above example performs six attempts and sixteen character comparisons to find the pattern in the text. Figure A-7 shows the Berry-Ravindran algorithm code(Charras & Lecroq, 1997).

2.7 The Sixth Category: Using a Hashing Function

The preprocessing phase here is shifting the pattern based on a single preprocessing hashing function. The Karp-Rabin (KR) algorithm is an example (Karp & Rabin, 1987).

2.7.1 The Karp-Rabin Algorithm (KR)

The KR algorithm was developed by R. M. Karp and M. O. Rabin in 1987 (Karp & Rabin, 1987).. The KR algorithm uses a hashing value to find patterns inside the text. The hashing function counts a numeric value for the text window and the pattern, if it is a different value then a definite mismatch is encountered, therefore, it moves the pattern to the right one position each time (Stephen, 1994). The *MOD* (modulus) operation is used to reduce the hash value of substring. There is a big problem using the hashing method called “spurious hits” where different substring can have the same hashing value. This is why the searching phase of KR searches the pattern again from leftmost to rightmost character to check if the search pattern and text window characters are similar. In most cases in a good hashing function, this will not happen, which keeps the average search time good (Karp & Rabin, 1987; Cormen, et al. 1990).

The time complexity of the preprocessing phase is $O(m)$ and of the searching phase is $O(mn)$ (Cantone, et al., 2004) . Figure 2-9, Figure 2-10 below show examples which illustrate the main principles of Karp-Rabin algorithm and Table 2-11 shows the Hash value for the searched pattern:

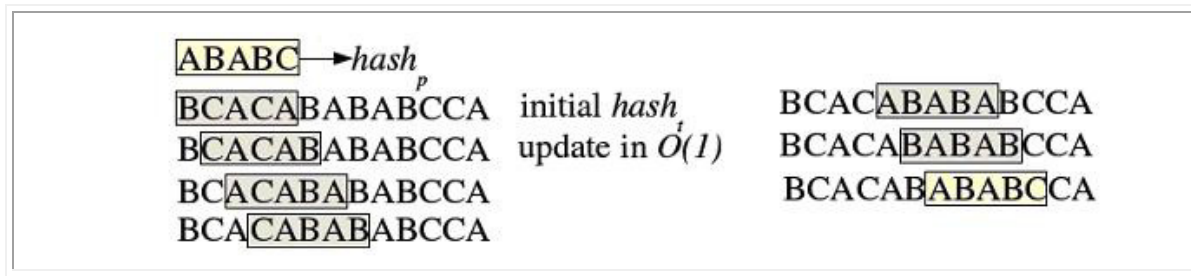


Figure 2- 9: First Example of the Karp-Rabin algorithm

Figure 2-9 illustrates the KR algorithm preprocessing and searching phases. It starts with calculating the hash value of the pattern “ABABC”. It searches the text by calculating the hash value of m length of the text “BCACA”. In this attempt the hash values are different, so it moves one position to the right and calculates the hash value for each text window until a match is found in the last attempt.

Character	C	O	2	C	2	C	2	C
Hash value[substring]	15246							

Table 2- 11: Hash value for the searched pattern

Table 2-11 calculates the hashing value of the pattern “CO2C2C2C” using the basic hashing function of the KR algorithm in Figure A-8. The hashing value of the pattern is 15246. The searching phase of the KR algorithm in Figure 2-16 starts with calculating the hashing value of each text window with length m . The first five attempts produce a different hashing value from the pattern hashing value. The sixth attempt produces the same hashing value of the pattern. The KR searching phase in the same attempt compares characters from the leftmost character to the rightmost character to check if the search pattern and text window characters are similar and to avoid the “spurious hits” problem which explained in section 2.7.1. After the whole match it moves again one position to the right and performs seventeen attempts to reach to the end of given text.

Figure 2-10: First attempt	Hash [0 .. 7]	15468
C O 2 H O C O 2 C 2 C 2 C H 2 H 2 O 2 C H 2 O C		

C O 2 C 2 C 2 C

Figure 2-10: Second attempt	Hash [1 .. 8]	15182
C O 2 H O C O 2 C 2 C 2 C H 2 H 2 O 2 C H 2 O C		

C O 2 C 2 C 2 C

Figure 2-10: Third attempt	Hash [2 .. 9]	15628
C O 2 H O C O 2 C 2 C 2 C H 2 H 2 O 2 C H 2 O C		

C O 2 C 2 C 2 C

Figure 2-10: Fourth attempt	Hash [3 .. 10]	17038
C O 2 H O C O 2 C 2 C 2 C H 2 H 2 O 2 C H 2 O C		

C O 2 C 2 C 2 C

Figure 2-10: Fifth attempt	Hash [4 .. 11]	14988
C O 2 H O C O 2 C 2 C 2 C H 2 H 2 O 2 C H 2 O C		

C O 2 C 2 C 2 C

Figure 2-10: Sixth attempt	Hash [5 .. 12]	15246
C O 2 H O C O 2 C 2 C 2 C H 2 H 2 O 2 C H 2 O C		

1 2 3 4 5 6 7 8

C O 2 C 2 C 2 C

Figure 2-10: Seventh attempt	Hash [6 .. 13]	14751
C O 2 H O C O 2 C 2 C 2 C H 2 H 2 O 2 C H 2 O C		

C O 2 C 2 C 2 C

Figure 2-10: Eighth attempt	Hash [7 .. 14]	14766
C O 2 H O C O 2 C 2 C 2 C H 2 H 2 O 2 C H 2 O C		

C O 2 C 2 C 2 C

Figure 2-10: Ninth attempt	Hash [8 .. 15]	15327
C O 2 H O C O 2 C 2 C 2 C H 2 H 2 O 2 C H 2 O C		

C O 2 C 2 C 2 C

Figure 2-10: Tenth attempt	Hash [9 .. 16]	14894
C O 2 H O C O 2 C 2 C 2 C H 2 H 2 O 2 C H 2 O C		

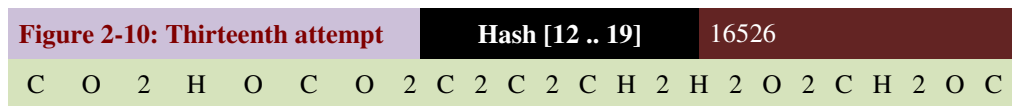
C O 2 C 2 C 2 C

Figure 2-10: Eleventh attempt	Hash [10 .. 17]	15566
C O 2 H O C O 2 C 2 C 2 C H 2 H 2 O 2 C H 2 O C		

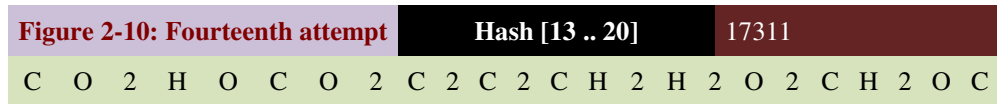
C O 2 C 2 C 2 C

Figure 2-10: Twelfth attempt	Hash [11 .. 18]	15372
C O 2 H O C O 2 C 2 C 2 C H 2 H 2 O 2 C H 2 O C		

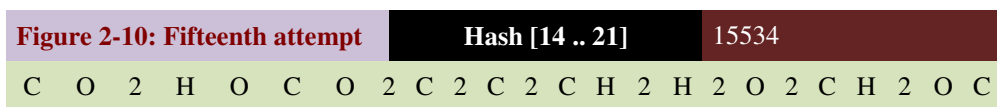
C O 2 C 2 C 2 C



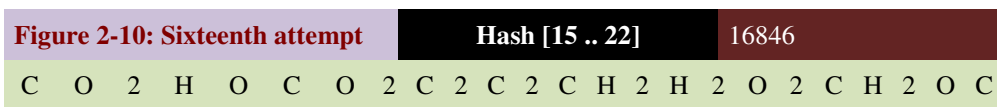
C O 2 C 2 C 2 C



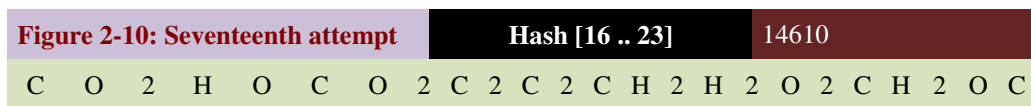
C O 2 C 2 C 2 C



C O 2 C 2 C 2 C



C O 2 C 2 C 2 C



C O 2 C 2 C 2 C

Figure 2- 10: Second Example of the Karp-Rabin algorithm

Applying the Karp-Rabin algorithm on the above example performs seventeenth attempts and twenty five character comparisons to find the pattern in the text. Figure A-8 shows the Karp-Rabin algorithm code (Charras & Lecroq, 1997).

2.8 The Seventh Category: Computing Buckets for All Characters of the Alphabet

In this group, the pattern shift depends on a single preprocessing function depending on computing buckets for all characters of the alphabet. The computing buckets process presents all the locations of each character in the pattern where if a character occurs k times in the pattern, there are k corresponding positions in the bucket of the character as explained in sections 2.8.1 and 2.8.2. Examples are the Skip Shift algorithm (SS) (Charras et al., 1998) and the Alpha Skip Shift algorithm (ASS) (Charras et al., 1998).

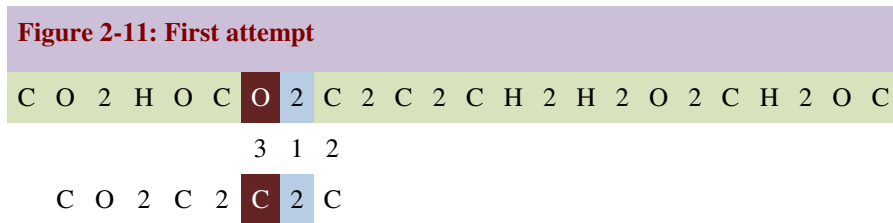
2.8.1 The Skip Shift Algorithm (SS)

The SS algorithm was developed by C. Charras, T. Lecroq and J. D. Pehoushek in 1998. The preprocessing phase of the SS algorithm preprocesses the pattern by computing buckets for all characters that exist in the text and the pattern. The search phase scans the m -th symbol to define a possible starting search point and to align identical symbols in the pattern and executes matching starting from the rightmost character to the leftmost character of remaining characters. When a whole match or a mismatch is encountered, the pattern is moved to align the next identical character in the pattern to the one in the text and start matching the other characters in the same previous order. (Charras & Lecroq, 2004). The time complexity of the preprocessing phase is $O(m+\sigma)$ and of the searching phase is $O(mn)$. Figure 2-11 illustrates the main principles of the SS algorithm and Table 2-12 shows the SS table of the Skip Search algorithm:

Character	Skip table[character]
2	{6,4,2}
O	{1}
C	{7,5,3,0}
H	$\Phi = 8$

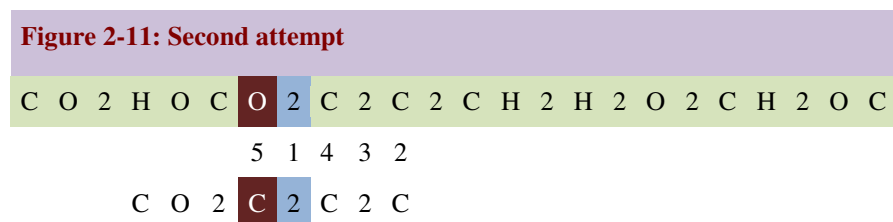
Table 2- 12: Skip Search table used by SS algorithm

Table 2-12 computes a bucket for each alphabet by presenting the k location(s) for each character. Character ‘2’ is located in locations 6,4 and 2, character ‘O’ in location 1, character ‘C’ in locations 7, 5, 3 and 0 and character ‘H’ is not in the pattern and will cause a shift to the right by 8 positions.



Total Shift Value = 3 (*Alpha Skip Search [2]*)

The searching phase of the SS algorithm defines a possible starting point by locating the m^{th} character of the text window with the same character of pattern using the bucket. In this attempt the m^{th} character is ‘2’ and it is rightmost location at position 2. It aligns similar characters and starts comparison from the right to the left. The third comparison produces a mismatch and in this case it aligns the next location of character ‘2’ from the bucket and in the second attempt is location 4. It follows the same procedure until a whole match is occurred in the third attempt and six attempts performed to reach to the end of the given text sample.



Total Shift Value = 2 (*Alpha Skip Search [2]*)

[illegible]

Figure 2-11: Fourth attempt

[illegible]

Figure 2-11: Fifth attempt

$\text{C O}_2 \text{H O C O}_2 \text{C}_2 \text{C}_2 \text{C H}_2 \text{H}_2 \text{O}_2 \text{C H}_2$ **O** C
1
 $\text{C O}_2 \text{C}_2 \text{C}_2 \text{C}$

Figure 2-11: Sixth attempt

$\text{C O}_2 \text{H O C O}_2 \text{C}_2 \text{C}_2 \text{C}_2 \text{C H}_2 \text{H}_2 \text{O}_2 \text{C H}_2 \text{C}$ **C**
 1
 $\text{C O}_2 \text{C}_2 \text{C}_2$

Applying the SS algorithm on the above example performs sixth attempts and nineteen character comparisons to find the pattern in the text. Figure A-9 shows the SS algorithm code (Charras & Lecroq, 1997).

2.8.2 The Alpha Skip Shift Algorithm (ASS)

The ASS algorithm was developed by C. Charras, T. Lecroq and J. D. Pehoushek in 1998. It is as an improvement of the SS algorithm (Charras, et al., 1998). The SS bucket list uses single identical characters to move the pattern, while the ASS algorithm uses substrings whose length may be longer than one character (Cantone et al, 2004). For any sub-text (b) in the text, find a nearest (b) in the pattern. If such (b) in the pattern exists, then move the pattern to align the two portions. If does not exist, then maybe consider a new text window. The preprocessing phase of the ASS algorithm depends on constructing a tree $T(x)$ of all sub-texts of length L . There is then one bucket for each leaf of $T(x)$ which stores the list of positions of all substrings with length $L = \log_{\sigma}(m)$ assuming that the size of the alphabet Σ of the text and the pattern is σ . The searching phase uses the information stored in the bucket to compare text T with pattern P (Cantone et al., 2005). Figure 2-12 below shows an example for a tree $T(x)$ of all substrings of the pattern “ababbaba” with length 8. The $T(x)$ length is $L = \log_2(8) = 3$.

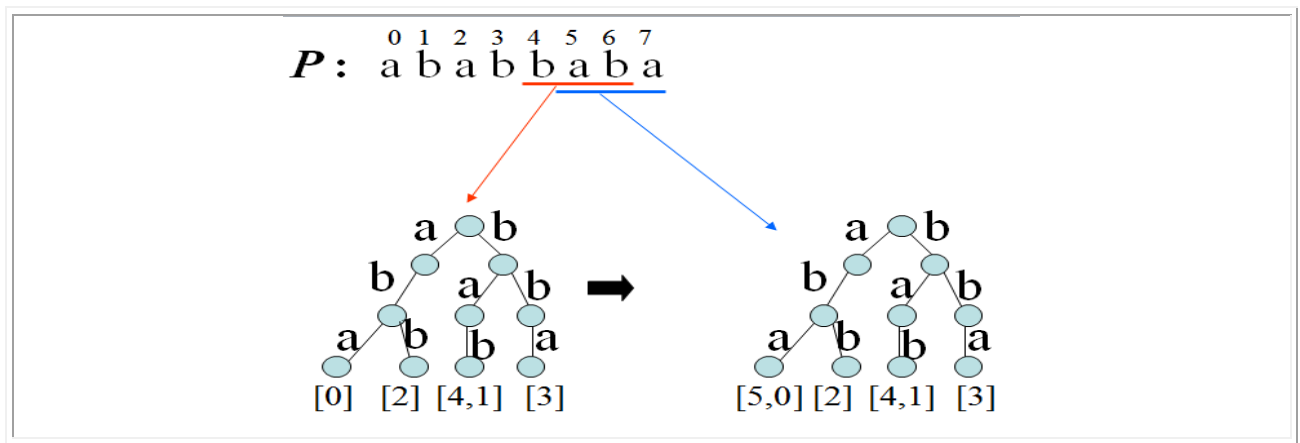


Figure 2- 12: An example for tree $T(x)$ of all substrings with $L=3$

Figure 2-12 structures a tree $T(x)$ for the given pattern “ababbaba” with substrings of length 3 and the location of each substring. For example substring “aba” is located at position [0], substring “bab” is located at positions [1] and [4], substring “abb” is located at position [2],

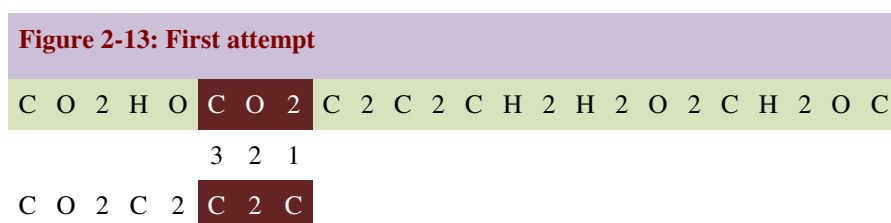
substring “bba” is located at position [3], and substring “aba” is located at positions [0] and [5].

The time complexity of the preprocessing phase is $O(m+\sigma)$ and of the searching phase is $O(mn)$ (Charras, et al., 1998). Figure 2-13 below shows an example that illustrates the main principles of the ASS algorithm and Table 2-13 shows the Alpha Skip Search algorithm skip table:

Character	Alpha Skip table[character] = m-position-length
2C2	{4,2} = 1
O2C	{1} = 4
C2C	{5,3} = 2
CO2	{0} = 5
H	Φ = 8

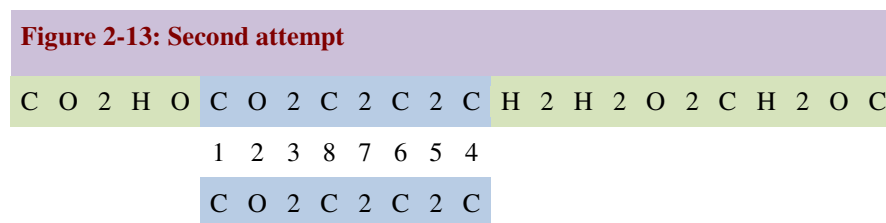
Table 2- 13: Alpha Skip Search table used by ASS algorithm

Table 2-13 computes a bucket in the same way as the preprocessing function of the SS algorithm. But the ASS algorithm uses substrings whose length may be longer than one character comparing to the SS algorithm. It presents the k location(s) for each substring. The substring length in this example is 3. So the substring “2C2” is located in locations 4 and 2, substring “O2C” in location 1, substring “C2C” in locations 5 and 3, substring “CO2” in location 0 and character ‘H’ is not in the pattern and will cause a shift to the right by 8 positions.

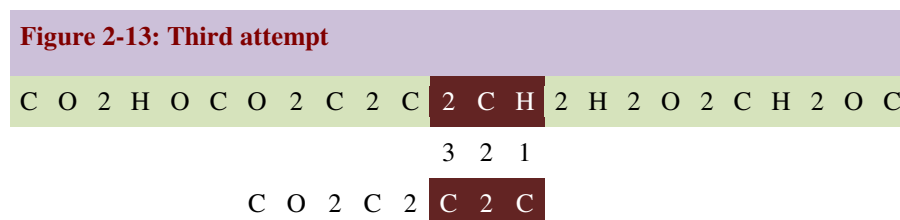


Total Shift Value = 5 (*Alpha Skip Search [CO2]*)

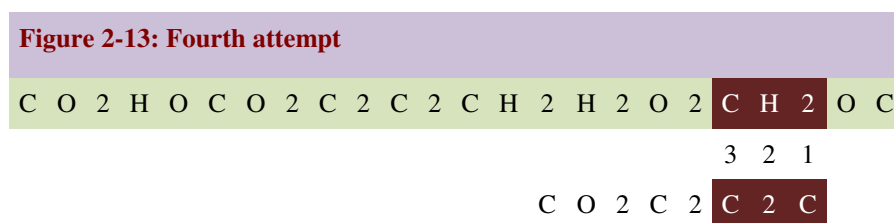
The first attempt using the last three characters of current text window produces a mismatch. The SS algorithm defines a possible starting point by locating the text window's substring with the same substring of pattern using the bucket. In this attempt the text window's substring is "CO2" and it is rightmost location at position 0 of the pattern. It aligns similar characters and starts comparison from the right to the left as shown in the second attempt. It follows the same procedure until a whole match is occurred in the second attempt and four attempts performed to reach to the end of the given text sample.



Total Shift Value = 1 whole match occur



Total Shift Value = 8 (*Alpha Skip Search [H]*)



Total Shift Value = 8 (*Alpha Skip Search [H]*)

Figure 2- 13: The Alpha Skip Search algorithm example

Applying the ASS algorithm on the above example performs four attempts and nineteen character comparisons to find the pattern in the text. Figure A-10 shows the ASS algorithm code (Charras & Lecroq, 1997).

2.9 The Eighth Category: Using Hybrid Algorithms

In this classification, to enhance the efficiency of searching algorithms, a combination of two or more algorithms is used. Examples are the SSABS (Sheik et al., 2004), FJS (Franek, et al., 2005), TVSBS (Thathoo, et al., 2006), ZTMBH (Huang et. al, 2008), BRFS (Huang et Al., 2008), BM-KMB (Xian-feng et al., 2010), BRSS (Almazroi & Rashid, 2011), ASSBR (Almazroi, 2011), MRCA (Mhashi, 2012), KRBMH (Hasan & Rashid, 2012), QSS (Naser, et al., 2012) and AKRAM (AbdulRazzaq, et al., 2013) algorithms.

2.9.1 The SSABS Algorithm

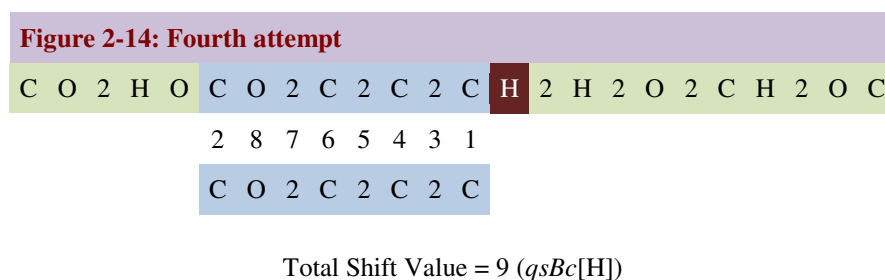
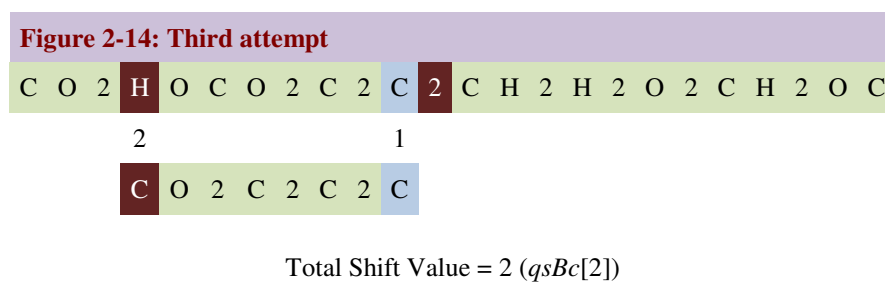
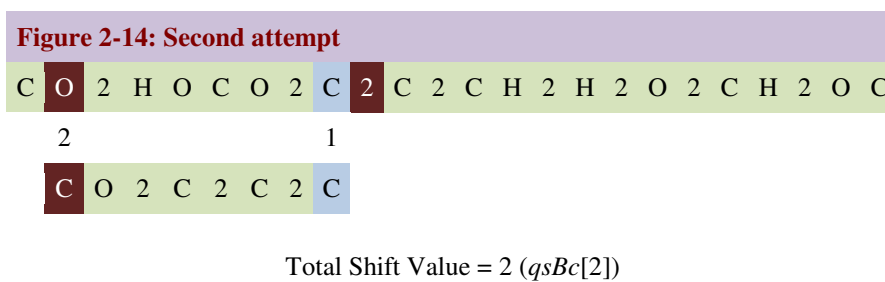
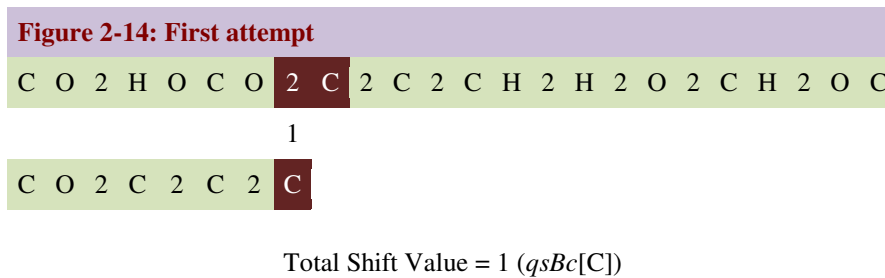
The SSABS algorithm was developed by S. S. Sheik, S. K. Aggarwal, A. Poddar, N. Balakrishnan and K. Sekar in 2004 (Sheik, et al., 2004). The SSABS searching phase firstly compares the rightmost character, then the leftmost character, and finally it starts from position $m-1$ moving backward to the second position of the pattern (Kalsi, et al., 2008). The preprocessing phase of SSABS algorithm uses the same qsBc function of the QS algorithm (Sheik, et al., 2004).

The time complexity of the preprocessing phase is $O(m+\sigma)$ and of the searching phase is $O(mn)$ (Sheik, et al., 2004). Figure 2-14 illustrates the main principles of the SSABS algorithm

and Table 2-14 shows the Quick Search algorithm bad character table (qsBc) which is used by the SSABS algorithm:

Character	C	2	O	H
qsBc[Character]	1	2	7	9

Table 2- 14: The qsBc table used by SSABS algorithm



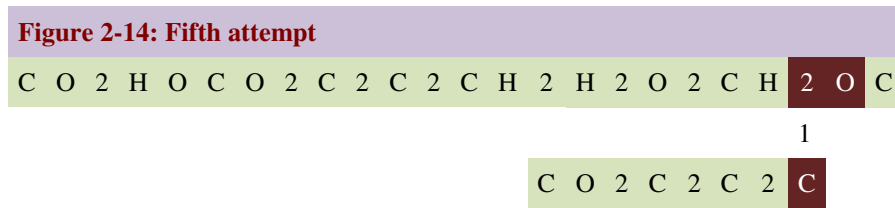


Figure 2- 14: The SSABS algorithm example

Applying the SSABS algorithm on the above example performs five attempts and fourteen character comparisons to find the pattern in the text. Figure A-11 shows the SSABS code (Sheik, et al., 2004).

2.9.2 The FJS Algorithm

The FJS algorithm was developed by F. Franek, G. J. Christopher, and F. S. William in 2005. It uses both the Knuth Morris Pratt (KMP) and the Quick Search algorithms (Franek, et al., 2005).

The FJS searching phase searches the pattern in the same way as the KMP searching phase where it starts from the leftmost character and moves forward to the rightmost character (Franek, et al., 2005). In each window if the mismatch occurs in the first position or if a whole match is encountered it shifts the pattern using the qsBc table of the Quick Search algorithm which depends on the rightmost character of the current window. Otherwise it uses the KMP preprocessing phase which uses the matched characters in each window as a sub-pattern (prefix) for shifting the pattern (Knuth, et al., 1977). The time complexity of the qsBc is $O(m+\sigma)$, the KMPBc is $O(m^2)$ and of the searching phase is $O(mn)$. Figure A-12 below shows the code for the FJS algorithm.

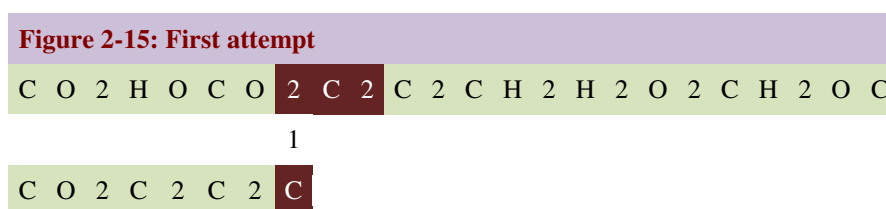
2.9.3 The TVSBS Algorithm

The TVSBS algorithm was developed by R. Thathoo, A. Virmani, S. S. Lakshmi, N. Balakrishnan and K. Sekar in 2006. In addition, the searching phase of the TVSBS algorithm searches the pattern in the same way as the SSABS searching phase which compares the rightmost character, then the leftmost character, and finally it starts from position $m-1$ moving backward to the second position of the pattern. If a whole match or a mismatch is encountered, the brBc function of the BR algorithm is used.

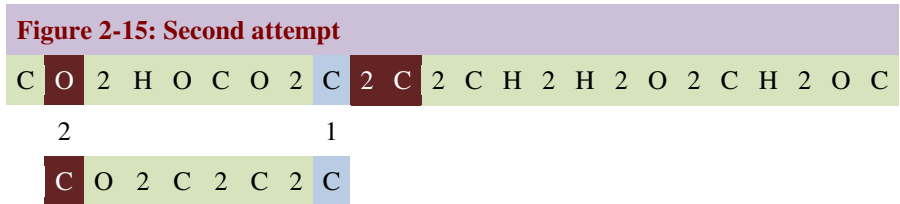
The time complexity of the preprocessing phase is $O(m+\sigma^2)$ and of the searching phase is $O(mn)$ (Thathoo, et al., 2006). Figure 2-15 illustrates the main principles of the TVSBS algorithm and Table 2-15 shows the brBc table which used by the TVSBS algorithm:

Character	2	O	C	H
2	10	10	2	10
O	7	10	9	10
C	1	1	1	1
H	10	10	9	10

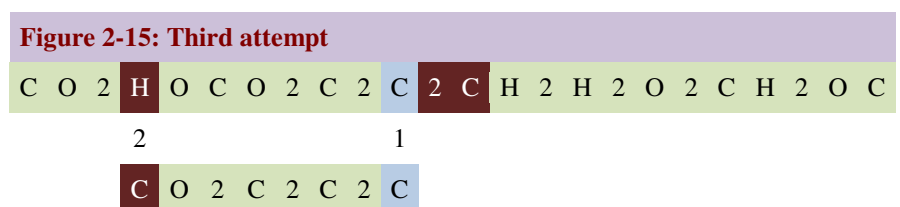
Table 2- 15: The brBc table used by TVSBS



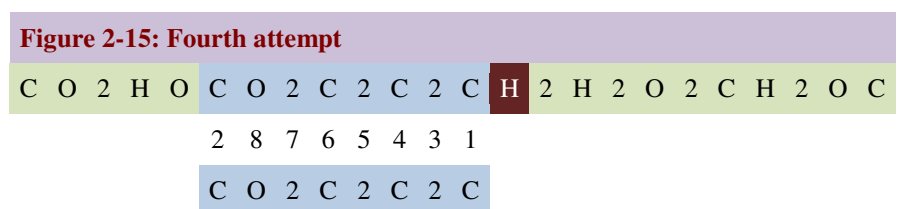
Total Shift Value = 1 (*brBc*[C][2])



Total Shift Value = 2 ($bmBc[2][C]$)



Total Shift Value = 2 ($bmBc[2][C]$)



Total Shift Value = 10 ($qsBc[H][2]$)

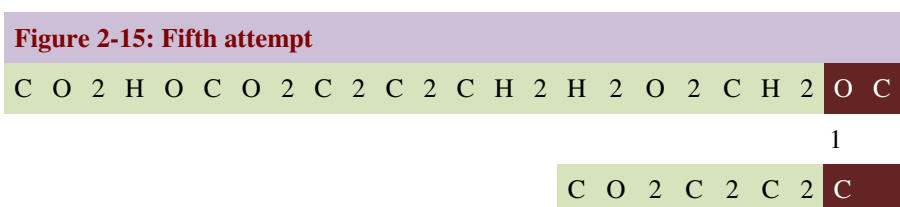


Figure 2- 15: The TVSBS algorithm example

Applying the TVSBS algorithm on the above example performs five attempts and fourteen character comparisons to find the pattern in the text. Figure A-13 shows the TVSBS code (Thathoo, et al., 2006; Kalsi, et al., 2008)).

2.9.4 The ZTBMH Algorithm

The ZTBMH algorithm combines the ZT and BMH algorithms. The algorithm was developed in 2008 by Y. Huang, X. Pan, Y. Gao, and G. Cai (Huang et al., 2008a). It searches the pattern in the same way as the BMH searching phase and it shifts the pattern if there is any mismatch or a whole match using the *ztBc* of the ZT algorithm.

The time complexity of the preprocessing phase is $O(m+\sigma^2)$ and of the searching phase is $O(mn)$ (Huang et al., 2008a). Figure 2-16 illustrates the main principles of ZTBMH algorithm and Table 2-16 shows the Zhu-Takaoka algorithm bad character table (*ztBc*) which is used by ZTBMH algorithm:

Character	2	O	C	H
2	8	8	2	8
O	5	8	7	8
C	1	6	7	8
H	8	8	7	8

Table 2- 16: The *ztBc* table used by ZTBMH

Figure 2-16: First attempt																			
C	O	2	H	O	C	O	2	C	2	C	2	C	H	2	H	2	O	2	C
						2	1												
C	O	2	C	2	C	2	C												

Total Shift Value = 5 (*ztBc*[O][2])

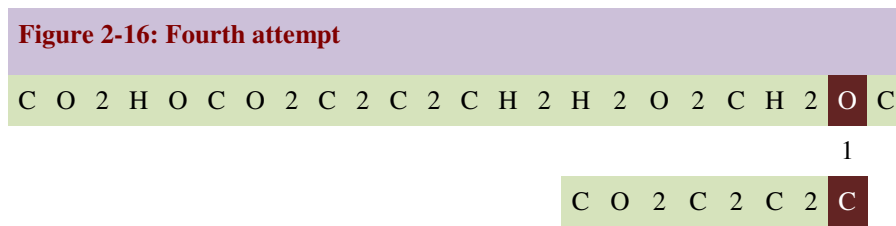
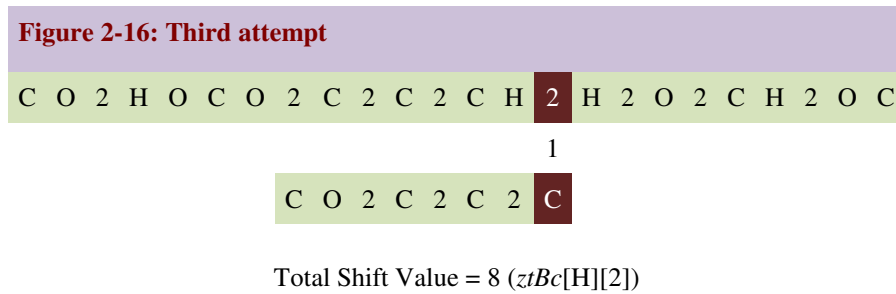
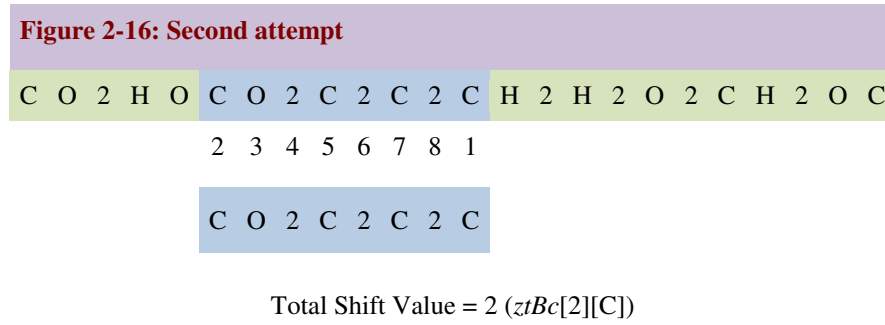


Figure 2- 16: The ZTBMH algorithm example

Applying the ZTBMH algorithm to the above example performs four attempts and eleven character comparisons to find the pattern in the text. Figure A-14 shows the ZTBMH code (Huang et al., 2008a).

2.9.5 The BRFS Algorithm

The BRFS algorithm combines the BR and the FS algorithms. The algorithm was developed in 2008 by Y. Huang, L. Ping, X. Pan, and G. Cai (Huang, Ping et al., 2008). The preprocessing

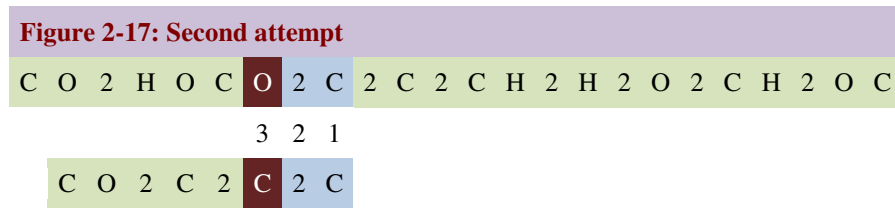
phase of the BRFS algorithm uses two preprocessing functions of the BR and the FS algorithms. In addition, the searching phase searches the pattern in the same way as the searching phase of the FS algorithm (Huang et al., 2008b). The preprocessing function of the BRFS always uses the good-suffix preprocessing function of the FS algorithm to shift the pattern, but if there is a whole match or a mismatch at the last text character, it uses the BR preprocessing function (Huang, Ping et al., 2008). The time complexity of the brBc function is $O(m+\sigma^2)$, the fsGs is $O(m^2)$ and of the searching phase is $O(mn)$ (Huang et al., 2008b). Figure 2-17 illustrates the main principles of the BRFS algorithm, Table 2-20 shows the Berry-Ravindran algorithm, the bad character table (brBc) and Table 2-17 shows the Fast Search good suffix (fsGs) table which is used by the BRFS algorithm:

Character	2	O	C	H
2	10	10	2	10
O	7	10	9	10
C	1	1	1	1
H	10	10	9	10

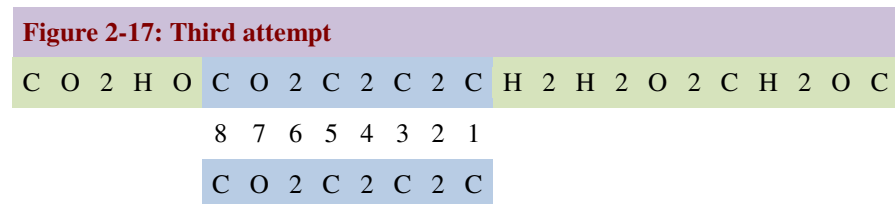
Table 2- 17: The brBc table used by the BRFS algorithm

Figure 2-17: First attempt																								
C	O	2	H	O	C	O	2	C	2	C	2	C	H	2	H	2	O	2	C	H	2	O	C	
							1																	
C	O	2	C	2	C	2	C																	

Total Shift Value = 1 ($brBc[C][2]$)



Total Shift Value = 4 (*fsGs*[O])



Total Shift Value = 10 (*brBc*[H][2])

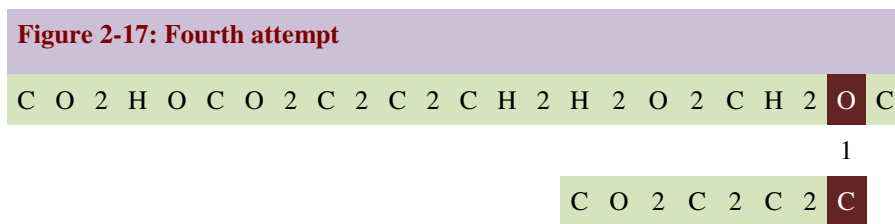


Figure 2- 17: The BRFS algorithm example

In this example the BRFS searching algorithm performs four attempts and thirteen character comparisons to find the pattern in the text. Figure A-15 shows the BRFS algorithm code (Huang et al., 2008b).

2.9.6 The BM-KMB Algorithm

The BM-KMB algorithm was developed by H. Xian-feng, Y. Yu-bao and X. Lu in 2010. It combines two searching phases of the KMP and the BM algorithms (Xian-feng et al., 2010). Firstly, it starts from the rightmost character in the same way as the BM algorithm. If there is a

mismatch, the BM preprocessing functions are used to shift the pattern. If there is a match it starts from the leftmost character in the same way as the KMP algorithm and moves forward to the right. If there is a mismatch while using the KMP searching phase it uses the KMP preprocessing function to shift the pattern (Xian-feng et al., 2010). The time complexity of the bmGs function is $O(m^2)$, the KMP table is $O(m^2)$ and of the searching phase is $O(mn)$. Figure A-16 shows the code for the BM-KMB algorithm.

2.9.7 The BRSS Algorithm

The BRSS algorithm was developed by A. Almazroi and N. Rashid in 2011. It combines the BR and the SS algorithms (Almazroi & Rashid, 2011). It uses a hybrid preprocessing phase by building two tables: the first one is the bucket list table of the SS algorithm, and the second table is the brBc(a,b) of the BR bad character function. The bucket list table contains all the location of the pattern and the text alphabets which will be used to align the next similar character if there is a mismatch (Almazroi & Rashid, 2011). The searching phase of the BRSS algorithm uses the SS searching phase to scan text window characters for a possible start point and if a whole match or a mismatch occurs, the pattern is shifted using the bigger shift value between the brBc and the bucket list (Almazroi & Rashid, 2011).

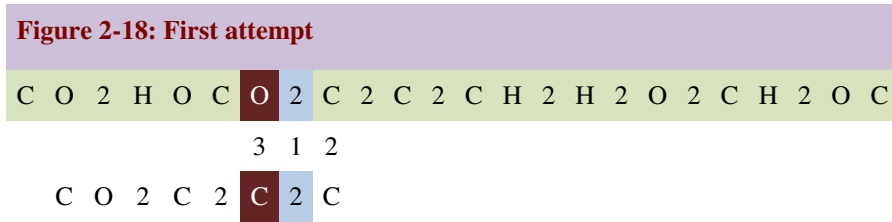
The time complexity of the brBc is $O(m+\sigma^2)$, the SS table is $O(m+\sigma)$ and of the searching phase is $O(mn)$ (Almazroi & Rashid, 2011). Figure 2-18 below shows an example which illustrates the main principles of the BRSS algorithm, Table 2-18 shows the Berry-Ravindran bad character table (brBc) and Table 2-19 shows the Skip Search table which are used by the BRSS algorithm:

Character	2	O	C	H
2	10	10	2	10
O	7	10	9	10
C	1	1	1	1
H	10	10	9	10

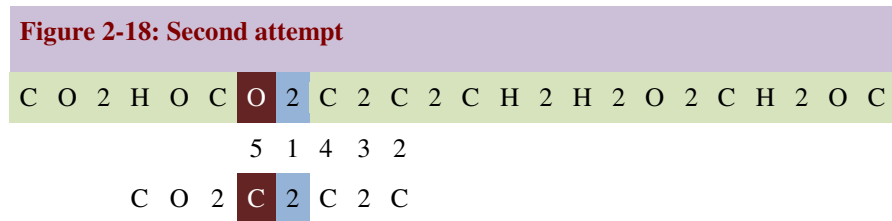
Table 2- 18: The brBc table used by the BRSS algorithm

Character	Skip table[character]
2	{6,4,2} = 2
O	{1} = 7
C	{7,5,3,0} = 1
H	Φ = 8

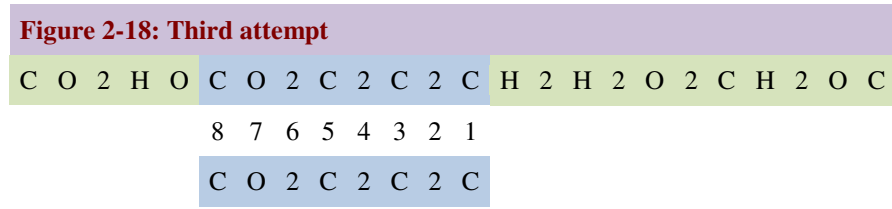
Table 2- 19: Skip Search table used by BRSS algorithm



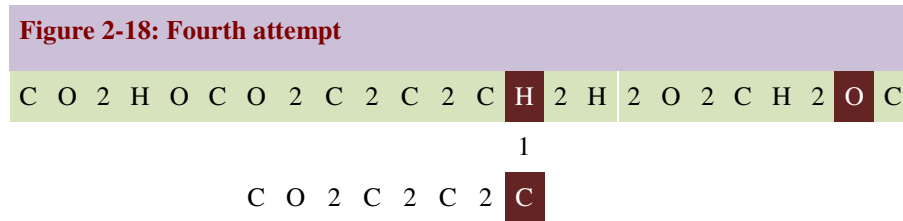
Total Shift Value = 3 ($Max(brBC [C][2], Skip Search [2])$)



Total Shift Value = 2 ($Max(brBC [2][C], Skip Search [2])$)



Total Shift Value = 1 whole match occur



Total Shift Value = 10 ($Max(brBC[2][H], Skip Search[H])$)

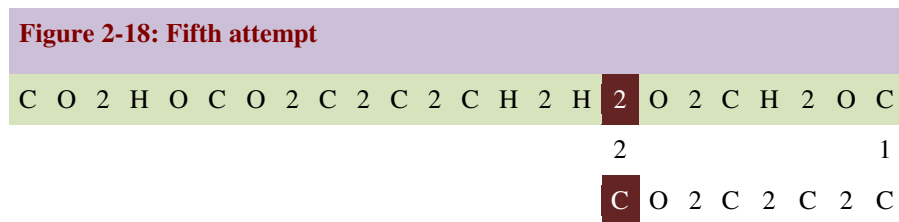


Figure 2- 18: The BRSS algorithm example

In this example the BRSS searching algorithm performs five attempts and nineteen character comparisons to find the pattern in the text. Figure A-17 shows the BRSS algorithm code (Charras & Lecroq, 1997).

2.9.8 The ASSBR Algorithm

The ASSBR algorithm was developed by A. Almazroi in 2011. It combines the ASS and the BR algorithms (Almazroi, 2011). The searching phase of the ASS algorithm searches the pattern using the ASS searching phase and if a whole match or a mismatch occurs it shifts the pattern using the $brBc(a,b)$ function of Berry-Ravindran algorithm (Almazroi, 2011).

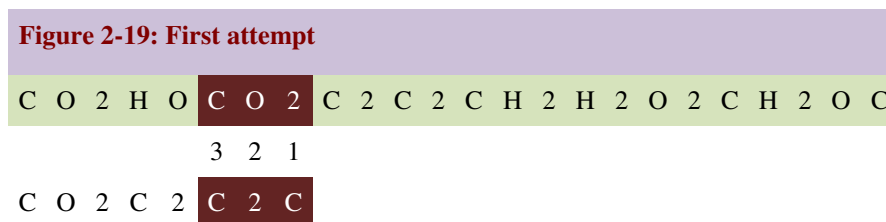
The time complexity of the brBc is $O(m+\sigma^2)$, the ASS table is $O(m+\sigma)$ and of the searching phase is $O(mn)$ (Almazroi, 2011). Figure 2-19 illustrates the main principles of the BRSS algorithm, Table 2-20 shows the Berry-Ravindran algorithm bad character table (brBc) and the Table 2-21 shows the ASS table which used by the ASSBR algorithm:

Character	2	O	C	H
2	10	10	2	10
O	7	10	9	10
C	1	1	1	1
H	10	10	9	10

Table 2- 20: The brBc table used by the ASSBR algorithm

Character	Alpha Skip table[character] = m-position-length
2C2	{4,2} = 1
O2C	{1} = 4
C2C	{5,3} = 2
CO2	{0} = 5
H	Φ = 8

Table 2- 21: Alpha Skip Search table used by ASSBR algorithm.



Total Shift Value =1 (*brBC[C][2]*)

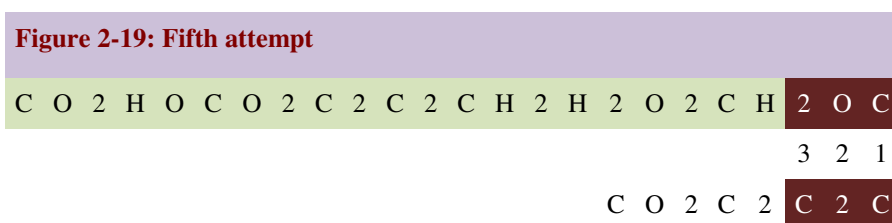
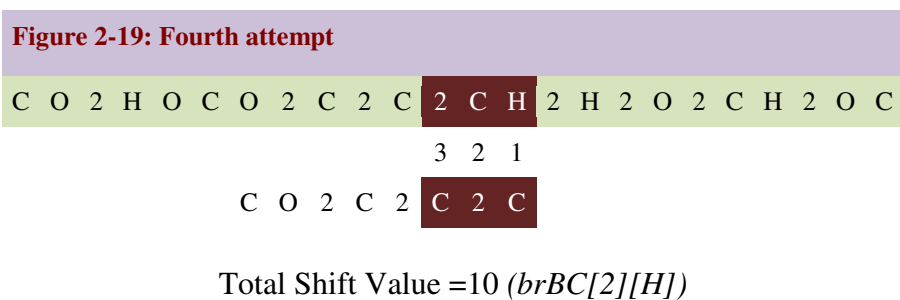
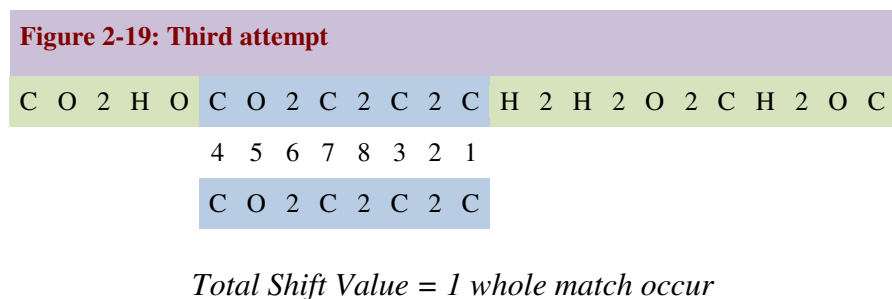
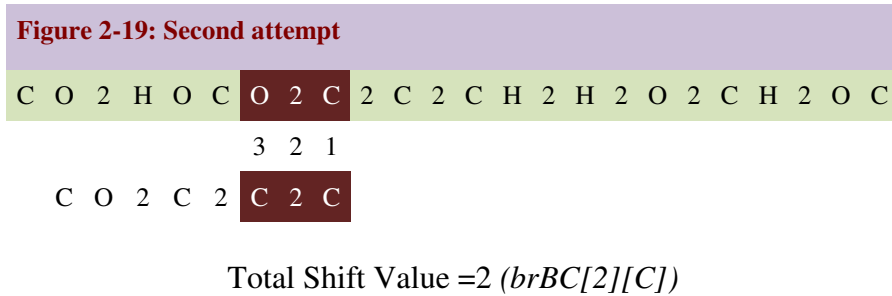


Figure 2- 19: The ASSBR Algorithm Example

In this example the ASSBR searching algorithm performs five attempts and twenty character comparisons to find the pattern in the text. Figure A-18 shows the ASSBR algorithm code (Charras & Lecroq, 1997).

2.9.9 The MRCA Algorithm

The Multiple Reference Character algorithm (MRCA) was developed by M. Mhashi in 2012. It combines the KMP and QS algorithms (Mhashi, 2012).

The preprocessing phase of the MRCA algorithm depends on a “multiple references” role to shift the pattern. It stores the last character of the text window in (ref_1). Furthermore, it stores the last character of the next text window ($2 * \text{pattern length}$) in (ref_2) and whenever the letter does exist in the pattern it allocates the pattern and starts the comparisons from the leftmost character to the rightmost character (Mhashi, 2012). The time complexity of the preprocessing function is $O(m+\sigma)$ and of the searching phase is $O(mn)$. Figure A-19 shows the MRCA algorithm code.

2.9.10 The KRBMH Algorithm

The KRBMH algorithm was developed by A. Hasan and N. Abdul Rashid in 2012. It combines the KR and BMH algorithms (Hasan & Rashid, 2012).

The searching phase of the KRBMH algorithm searches the pattern using the hashing value of the Karp Rabin algorithm, if a whole match or a mismatch is founded, the hrBc table of the BMH algorithm is used to shift the pattern (Hasan & Rashid, 2012). The time complexity of the hrBc is $O(m+\sigma)$, the KR hashing table is $O(m)$ and of the searching phase is $O(mn)$. Figure A-20 shows the KRBMH algorithm code.

2.9.11 The Quick-Skip Search Algorithm (QSS)

The QSS algorithm was developed by M. Naser, N. Abdul Rashid and M. Aboalmaaly in 2012. It combines the Skip Search and Quick Search algorithms (Naser, et al., 2012).

The searching phase of the QSS algorithm uses the SS searching phase. In addition, the preprocessing phases of the SS and QS algorithms are used where if a whole match or a mismatch is found, the pattern is shifted by choosing the bigger shifting value from either the qsBc or the Skip Search tables. The time complexity of the qsBc is $O(m+\sigma)$, the SS table is $O(m+\sigma)$ and of the searching phase is $O(mn)$. Figure A-21 shows the QSS algorithm code.

2.9.12 The AKRAM Algorithm

The AKRAM algorithm was developed by A. A. AbdulRazzaq, N. Abdul Rashid and M. F. Aboalmaaly in 2013. It combines the Two-way, QS and KR algorithms (AbdulRazzaq, et al., 2013).

The preprocessing phase of the AKRAM algorithm divides the pattern into two blocks (prefix and suffix). In addition, the preprocessing phase of KR which depends on hashing value is used for both blocks (AbdulRazzaq, et al., 2013).

The searching phase of the AKRAM algorithm starts with the prefix block. If the hashing value of the prefix does match, then it starts comparing all prefix characters from the leftmost character to the rightmost character. If a whole match in the prefix part is found, it follows the same procedure with the suffix part. If there is a whole match in both blocks and a mismatch either in the prefix or suffix, the pattern is shifted using the qsBc function of the QS algorithm (AbdulRazzaq, et al., 2013). The time complexity of the qsBc is $O(m+\sigma)$, the KR hashing table is $O(m)$ and of the searching phase is $O(mn)$. Figure A-22 shows the AKRAM algorithm code.

2.10 Summary of String Matching Algorithms

This chapter gives a survey and a new classification of main string matching algorithms

which includes the Brute Force algorithm (BF), the Boyer-Moore algorithm (BM), the Zhu Takaoka algorithm (ZT), the Fast Search algorithm (FS), the Boyer Moore Horspool algorithm (BMH), the Quick Search algorithm (QS), the Berry–Ravindran algorithm (BR), the Karp-Rabin algorithm (KR), the Skip Shift (SS), the Alpha Skip Shift (ASS), the SSABS, FJS, TVSBS, ZTBMH, BRFS, BM-KMP, BRSS, ASSBR, MRCA, KRBMH, QSS and AKRAM algorithms.

Table 2-22 summarizes and compares the algorithms that used in this research:

Algorithm Name	Year	Comparison Order	Preprocessing Time		Searching		Main Characteristics
			First Function	Second Function	Searching Time	Shifting Value	
The First Category: Shift the Pattern a Single Position							
Brute Force Algorithm (BF)	Very Old	From left to right	N/A	N/A	O(mn)	1	Shifts the pattern only a single position each attempt. It does not use the information that could be gained from the last comparison made
The Second Category: Using Two Preprocessing Functions							
Boyer-Moore Algorithm (BM)	1977	Form right to left	O(m+σ) bmBc	O(m ²) bmGs	O(mn)	m	Uses two pre-processing functions; the bad-character shift and the good-suffix shift

Zhu-Takaoka Algorithm (ZT)	1987	From right to left	$O(m+\sigma^2)$ ztBc	$O(m^2)$ bmGs	$O(mn)$	m	It is a variant of the BM algorithm by improving only the bmBc function. It uses the last two text characters to compute the bad character shift instead of only one character being used in the BM algorithm. The good suffix rules are still used to compute shifts
Fast Search Algorithm (FS)	2003	From right to left	$O(m+\sigma)$ fsBc	$O(m^2)$ fsGs	$O(mn)$	m	Uses the bad-character function only if the character is causing the mismatch is the last character of the pattern or a whole match occurs, otherwise the good-suffix function is to be used.
The Third Category: Depending on the Rightmost Character							
Boyer-Moore Horspool Algorithm (BMH)	1980	Right-most character then left-most character then moves forward	$O(m+\sigma)$ hrBc	N/A	$O(mn)$	m	Uses the Horspool bad-character pre-processing function based on the rightmost character in the current window.

The Fourth Category: Depending on the Next Character to the Rightmost Character

Quick-Search Algorithm (QS)	1990	From left to right	$O(m+\sigma)$ qsBc	N/A	$O(mn)$	$m+1$	Uses the Quick-Search bad-character preprocessing function based on the next character to the current window.
------------------------------------	------	--------------------	-----------------------	-----	---------	-------	---

The Fifth Category: Depending on Two Characters Next to the Rightmost Character

Berry-Ravindran Algorithm (BR)	1999	From left to right	$O(m+\sigma^2)$ brBc	N/A	$O(mn)$	$m+2$	Uses the Berry-Ravindran pre-processing function based on the next two characters after the current window in order to increase the shifting value of the pattern
---------------------------------------	------	--------------------	-------------------------	-----	---------	-------	---

The Sixth Category: Using a Hashing Function

Karp-Rabin Algorithm (KR)	1987	From left to right	$O(m)$ KMP Hashing Function	N/A	$O(mn)$	1	Uses the Karp-Rabin pre-processing hashing function.
----------------------------------	------	--------------------	--------------------------------	-----	---------	---	--

The Seventh Category: Computing Buckets for All Characters of the Alphabet

Skip Shift (SS)	1998	Form right to left	$O(m+\sigma)$ SS Table	N/A	$O(mn)$	1	Computes a bucket for pattern and text alphabets, with start positions of each alphabet in the pattern to be used for a possible shift
Alpha Skip Shift (ASS)	1998	Form right to left	$O(m+\sigma)$ ASS Table	N/A	$O(mn)$	1	Computes a bucket for substrings with length $L = \log_{\sigma}(m)$, with start positions of each substring in the pattern to be used for a possible shift

The Eighth Category: Using Hybrid Algorithms

SSABS Algorithm	2004	Right-most character then left-most character then starts from $m-2$ moving backward	$O(m+\sigma)$ qsBc	N/A	$O(mn)$	$m+1$	A Combination of the Quick-Search bad-character pre-processing functions with a new searching order.
FJS Algorithm	2005	From left to right	$O(m+\sigma)$ qsBc	$O(m^2)$ KMP Table	$O(mn)$	$m+1$	Uses the Karp-Rabin pre-processing hashing function and shifts the pattern using the preprocessing function of the QS algorithm

TVSBS Algorithm	2006	Same way as the SSABS	$O(m+\sigma^2)$ brBc	N/A	$O(mn)$	$m+2$	A combination of the Berry Ravindran pre-processing function and the searching phase of the SSABS algorithm.
ZTBMH Algorithm	2008	Same way as the BMH	$O(m+\sigma^2)$ ztBc	N/A	$O(mn)$	M	A combination of the Zhu Takaoka preprocessing function and the searching phase of the Boyer Moore Horspool algorithm.
BRFS Algorithm	2008	Using the searching phase of the Fast Search algorithm	$O(m+\sigma^2)$ brBc	$O(m+\sigma^2)$ fsGs	$O(mn)$	$m+2$	A combination of the Berry Ravindran preprocessing function and the searching phase of the Fast Search algorithm.
BM-KMP Algorithm	2010	Same way as the BMH	$O(m+\sigma^2)$ bmGs	$O(m^2)$ KMP Table	$O(mn)$	M	If last character is causing the mismatch, the bmBc is used. Otherwise the KMP table is used to shift the pattern
BRSS Algorithm	2011	Same way as the SSABS	$O(m+\sigma^2)$ brBc	$O(m+\sigma)$ SS Table	$O(mn)$	$m+2$	It shifts the pattern using the bigger shift value between the SS Table and the brBc function

ASSBR Algorithm	2011	Form right to left	$O(m+\sigma^2)$ brBc	$O(m+\sigma)$ ASS Table	$O(mn)$	$m+2$	Uses the same searching phase as the ASS table and shifts the pattern using the brBc function
MRCA Algorithm	2012	From left to right	$O(m+\sigma)$ MRCA Table	N/A	$O(mn)$	m	Uses the “multiple references” role to shift the pattern. It stores the position of the last character of current window and next windows in references and move pattern accordingly.
KRBMH Algorithm	2012	From left to right	$O(m+\sigma)$ hrBc	$O(m)$ KR Hashing Function	$O(mn)$	m	Uses the KR pre-processing hashing function. If same value then searches from left to right. It and shifts the pattern using the qsBc function
QSS Algorithm	2012	From left to right	$O(m+\sigma)$ qsBc	$O(m+\sigma)$ SS Table	$O(mn)$	$m+1$	Uses the same searching phase as the SS algorithm. It shifts the pattern using the SS table and the qsBc function
AKRAM Algorithm	2013	From left to right	$O(m+\sigma)$ qsBc	$O(m)$ KR Hashing Function	$O(mn)$	$m+1$	Divide the text window and the pattern to prefix and suffix. Use the same searching phase as the KRBMH for prefix part first then suffix part. If there is a whole match, the qsBc is used

Table 2- 22: Summary of algorithms been used in this research

2.11 SMILES Format

SMILES is a chemical language that is commonly used amongst chemists which presents chemical structures in digital databases as a linear string notation (Weininger et al., 1989; Weininger, 1988). In addition, the linear string notation can be easily used and implemented using computer programs rather than using graphical structures (Rowley et al., 2001).

It was originally developed in the 1980's by David Weininger, and has since been modified by Daylight Chemical Information Systems Incooperation.

The SMILES system was designed in order to achieve three main objectives – (1) the representation of the chemical structure can be uniquely designed to include structure components such as atoms and bonds. (2) Unique notations are to be interpreted and generated through a machine friendly and machine independent system, (3) A structure specification should be provided in order to provide ease for the user (Weininger, 1988).

SMILES can be used as a text to represent a chemical structure. It is a language paradigm rather than a data structure, and this is why it is more valuable and important. “It takes 50% to 70% less space than an equivalent connection table. For example, a database of 23,137 structures, with an average of 20 atoms per structure, uses only 1.6 bytes per atom when they represented with SMILES format” (Daylight Chemical Information Systems, 2008).

SMILES is a formal language with a well-defined grammar over an alphabet of symbols, atoms and bonds with certain grammar rules. SMILES's format strength lies in the unique generated format for each molecule which makes it easy to search the molecule structure (Weininger et al., 1989; Neglur et al., 2005). Table 2-23 below shows some examples of structures in SMILES format:

SMILES	Structure Name
CC	Ethane
O=C=O	Carbon Dioxide
C#N	Hydrogen cyanide
CCN(CC)CC	Triethylamine
CC(=O)O	Acetic acid
C1CCCCC1	Cyclohexane
[OH ₃ ⁺]	Hydronium ion
[2H]O[2H]	Deuterium oxide

Table 2-23: Structures in SMILES format examples

SMILES consists of 6 syntax rules that can be applied to any chemical structure which allow two-dimensional chemical structures to be represented in SMILES as follows (Rowley et al., 2001; De Raedt & Kramer, 2003 ; Daylight Chemical Information Systems, 2008; U.S. Environmental Protection Agency EPA, 2009):

1) SMILES Atoms: there are two types of SMILES atoms rule. The first one presents compounds of elements not in the organic subset “S, O, C, I, B, F, N, P, Cl and Br” which are represented in SMILES using their atomic symbol enfolded between square brackets. The second type presents compounds of the “organic subset” elements which are represented in SMILES using their atomic symbol (a letter) without hydrogen atom symbols and without the square brackets. Some of the organic subset needs to be enfolded between square brackets if they have atoms with valences “unusual number of bonds an atom forms”. For both types of atoms the second letter of a two character symbol is represented in a lower case. Below Table 2-24 and Table 2-25 are examples of how atoms are represented in SMILES for both types.

Structure	Structure Name	SMILES Atoms Rule
(CH ₄)	Methane	C
(PH ₃)	Phosphine	P
(NH ₃)	Ammonia	N
(H ₂ S)	Hydrogen sulphide	S
(H ₂ O)	Water	O
(HCl)	Hydrochloric acid	Cl

Table 2-24: The SMILES atom rule of structures with organic subset elements

Element	Structure Name	SMILES Non-Hydrogen Atoms Rule
S	Sulfur	[S]
Au	Gold	[Au]

Table 2-25: The SMILES atom rule of non-organic elements

Hydrogens and charges attached to elements in brackets must always be stated. Ions that have one or more electrical charges are represented in SMILES by either a + for a positively charged or – for a negative charge then followed by the number which indicates the number of charges, all of which is enclosed in brackets, below Table 2-26 is an example. An alternative method in which a charge can be represented in SMILES is by having the sign for the number of ions that are to be represented, below Table 2-27 is an example.

Structure Name	SMILES
Iron (II) Cation	[Fe+2]
Sulphides	[S-2]

Table 2- 26: Examples of positive and negative charges represented by numbers

Structure Name	SMILES
Iron (II) Cation	[Fe++]
Sulphides	[S--]

Table 2-27: Examples of positive and negative charges represented by signs

2) Bonds: a single bond can either be ignored or be represented by the symbol “-“, double bond are presented by the symbol “=”, a triple bond is represented by the symbol “#” and an aromatic bond is represented by “:” (Daylight Chemical Information Systems, 2008). In the instance of SMILES, adjacent atoms are to consider to be either connected via single or aromatic bonds. A typical example of how SMILES represent bonds is as follows in Table 2-28:

Structure	Structure Name	Bond	SMILES
(CH ₃ CH ₃)	Ethane	Single	C-C
(CH ₃ CH ₃)	Ethane	Single	CC
(CH ₂ O)	Formaldehyde	Double	C=O
(CH ₂ =CH ₂)	Ethene	Double	C=C
(HCN)	Hydrogen cyanide	Triple	C#N
c1ccccc1	Benzene	Aromatic	c1:c:c:c:c:c1

Table 2-28: Examples of SMILES bonds

3) Branches: A branch is represented in SMILES by placing the symbol between round brackets. The string in bracket is always placed after the symbol of the atom from which branches. If there is a double or triple bond then bond symbol follows the left hand side of the bracket (U.S. Environmental Protection Agency EPA, 2009) as shown in Table 2-29.

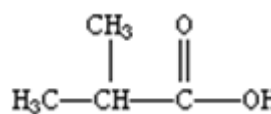
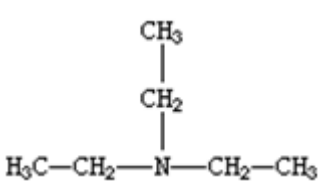
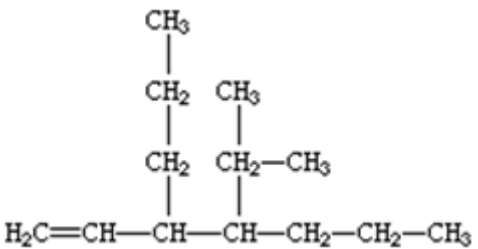
Isobutyric acid	Triethylamine	3-propyl-4-isopropyl-1-heptene
		
<chem>CC(C)C(=O)O</chem>	<chem>CCN(CC)CC</chem>	<chem>C=CC(CCC)C(C(C)C)CCC</chem>

Table 2-29: Examples of SMILES branches rule

4) Cyclic Structures: or cyclic bonds can be defined by referencing the carbon atoms with numbers. For example, C1CCCCC1 (cyclohexane) is a string of six carbon atoms where the first and sixth atoms are bonded together as defined with the number 1. In the instance where there may be a multiple bond more numbers can be used to denote where multiple bonds exist. Below Table 2-30 is an example:

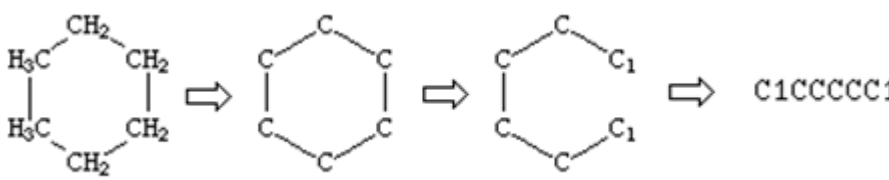
Cyclohexane Structure


Table 2-30: Example of SMILES cyclic structure rule

5) Disconnected Structures – In order to represent disconnected compounds, which do not have a covalent bond to join the two structure together, these are constructed by writing individual structures which are separated by a “.” as below in Table 2-31.

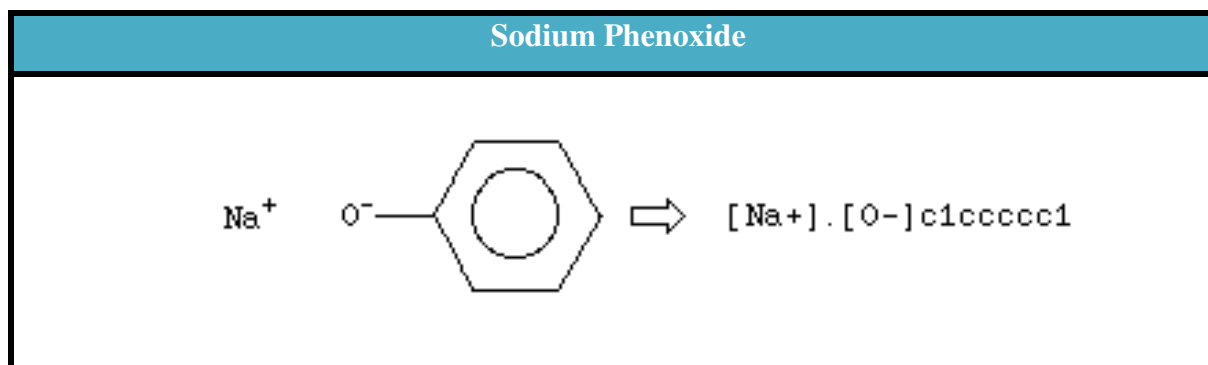


Table 2-31: Example of SMILES disconnected rule

6) Aromaticity – Aromaticity is a chemical property that defines unsaturated bonds, empty orbitals or lone pairs of a conjugated ring (Daylight Chemical Information Systems, 2008). Aromaticity rule presents S, O, N and C atoms in SMILES as lower-case (s, o, n and c). Table 2-32 below shows some examples:

Structure Name	SMILES
Benzene	<chem>c1ccccc1</chem>
Pyridine	<chem>n1ccccc1</chem>
Furan	<chem>o1ccccc1</chem>

Table 2-32: Example of SMILES aromaticity rule

The complete EBNF (Extended Backus-Naur Form) of SMILES language is listed in Appendix B.

2.12 Parallel Computing

In this section, a brief explanation of the parallel computing concept, Flynn's Taxonomy, and parallel programming models are provided.

In a sequential program, tasks ($t_1, t_2 \dots t_n$) will run on a single CPU and one task will be executed at any moment as shown in Figure 2-20 (Barney, 2010).

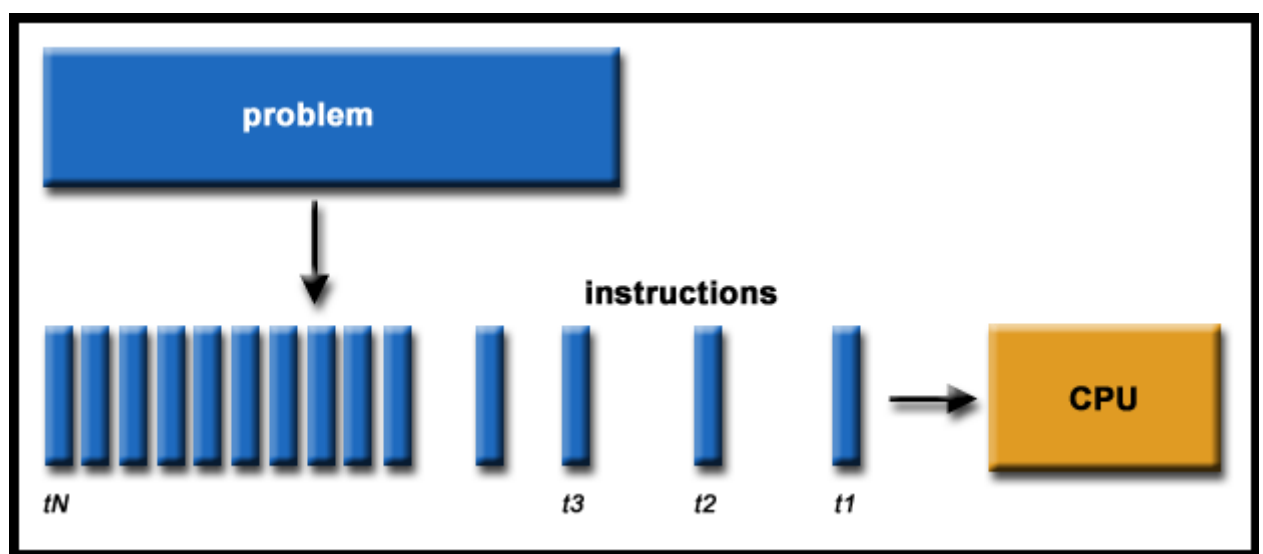


Figure 2- 20: An example for the sequential program tasks execution

The simplest definition of parallel computing is a simultaneous use of more than one CPU to run a computational problem as shown in Figure 2-21 (Barney, 2010). The multi CPU computing resources could be a single computer with multiple CPUs or a number of computers connected by a network, or a combination of both. The computational problem should be able to be broken down into tasks that can be solved simultaneously, executing multiple tasks at any time and be solved in less time with multiple CPUs.

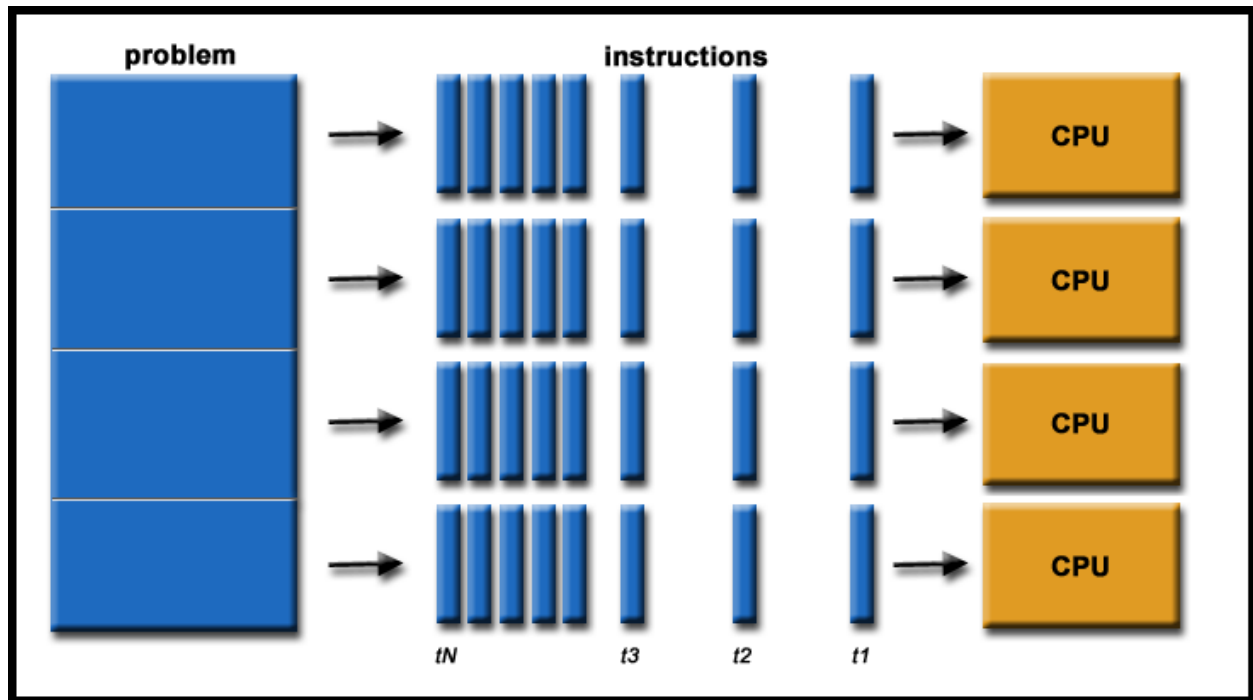


Figure 2-21: An example for the parallel program tasks execution

Parallel computing has been used in many areas of engineering and sciences such as molecular sciences, biotechnology, medical imaging and diagnosis, bioscience, pharmaceutical design, computer science, networked video and multi-media technologies, mathematics, atmosphere, etc. (Barney, 2010; Rajasekaran & Reif, 2007).

2.12.1 Flynn's Taxonomy

The best known way of classifying parallel computing is called Flynn's Taxonomy, introduced in 1966, and depends on two independent types the instruction stream and data stream (Flynn, 1966). These types can have only one state, either single or multiple (Tucker, 2004). The following Figure 2-22 shows the classifications of Flynn's taxonomy (Flynn, 1972).

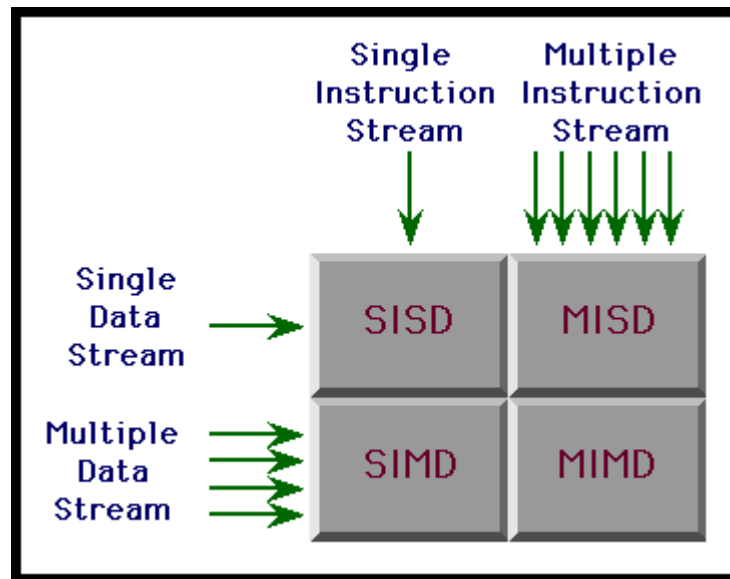


Figure 2- 22: Flynn's taxonomy (Flynn, 1972)

1. **Single Instruction, Single Data (SISD):** There is no parallelism in data or instruction streams.
2. **Single Instruction, Multiple Data (SIMD):** Running same instruction stream on parallelized data sets such as running the same algorithm on different blocks of text.
3. **Multiple Instruction, Single Data (MISD):** Operating multiple instruction streams on the same data such as running different algorithms on the same block of text.
4. **Multiple Instructions, Multiple Data (MIMD):** Operating multiple instruction streams independently on multiple data streams.

In this research, the SSN algorithm is paralleled using the MISD type to implement the OpenMP model and the SIMD type to implement the MPI model as explained in section 3.3.

2.12.2 Parallel Computing Speedup:

The parallel computing speedup calculates the increase of running time after finding the sequential and parallel execution time of an algorithm as Equation(2) in Figure 2-23 (Akl, 1997; Wilkinson & Allen, 2005):

$$S_p = \frac{T_s}{T_p} \dots\dots\dots(2)$$

Figure 2- 23: Parallel computing speedup equation

where S_p is the speed up, T_s is the sequential execution time on a single processor and T_p is the parallel execution time with p processors. The following Figure 2-24 (Willmore, 2012) presents the different types of speedup using different number of processors and Figure 2-25 (Barney, 2010) presents the relationship between the common speedup (sub-linear) and the number of processors used based on the fraction of code that can be parallelized.

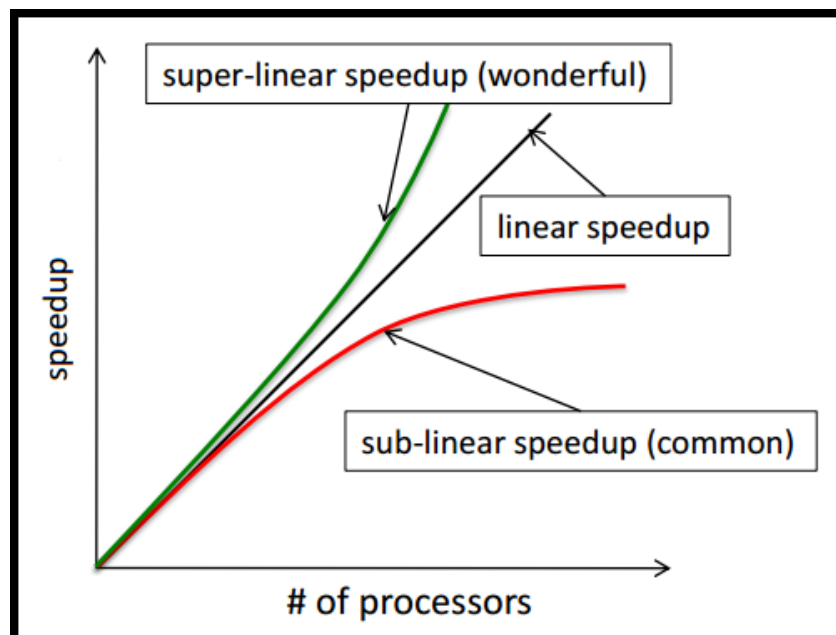


Figure 2- 24: Different type of speedup using different number of processors

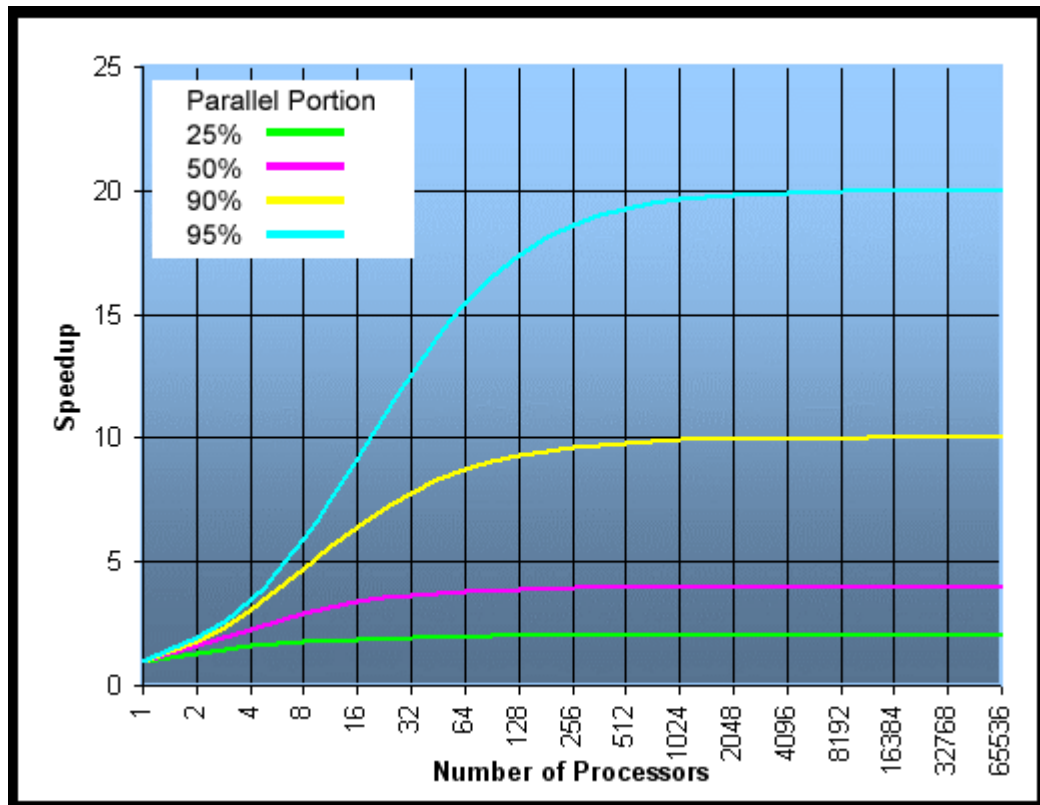


Figure 2-25: The relationship between the common speedup and number of processors

2.12.3 Parallel Programming Models:

There are several parallel programming models can be used to program data and instructions on processors such as the shared memory model, distributed memory model or a hybrid model combining more than one model (Kontoghiorghe, 2010). Note that there is no “best” model. It depends on the machines available and on the nature of the problem being addressed.

2.12.3.1 The Shared Memory Model:

In the shared memory model, jobs share a common memory address to write and read from as shown in Figure 2-26 (Barney, 2010).

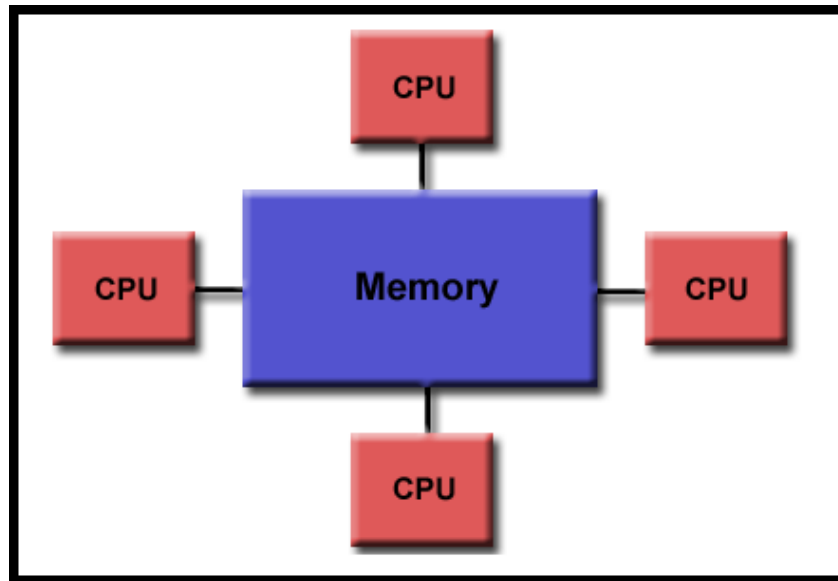


Figure 2- 26: The shared memory model

The threads model is a type of shared memory model, but a single big job can be divided into small ones and implemented with simultaneous execution paths as shown in Figure 2-27:

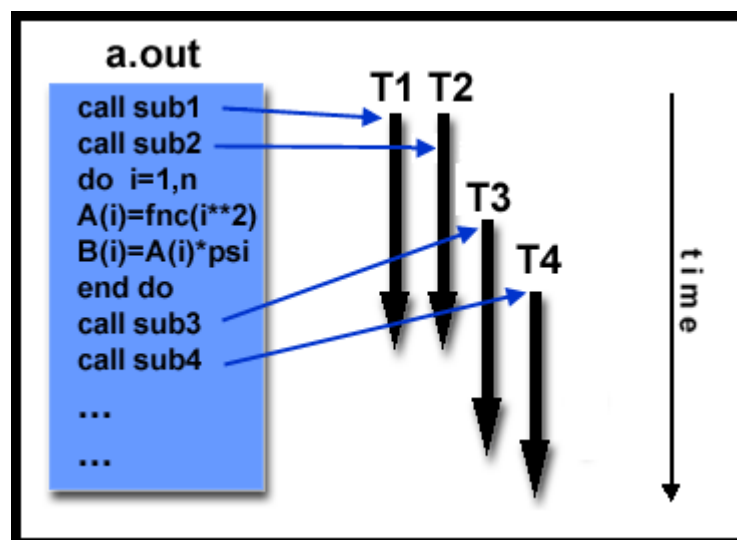


Figure 2-27: The threads model

“The Open Multi Processing (OpenMP) model was released in 1997 as a standard

Application Programming Interface (API) for writing shared memory parallel applications in C, C++ and FORTRAN” (Kiessling, 2009). It is easy to implement and widely used with multicore architecture to parallelize serial code.

OpenMP model use the Fork and Join Model. They start sequentially as a single thread, called the “master thread”, until they reach a parallel section where they fork into multiple “worker threads” as shown in Figure 2-28 (Kiessling, 2009; Barney, 2010). At the end of a parallel section, the threads re-join to become a master thread again. It is possible to run more than one thread on a single processor but it is common and safer to run a single thread per processor (Kiessling, 2009).

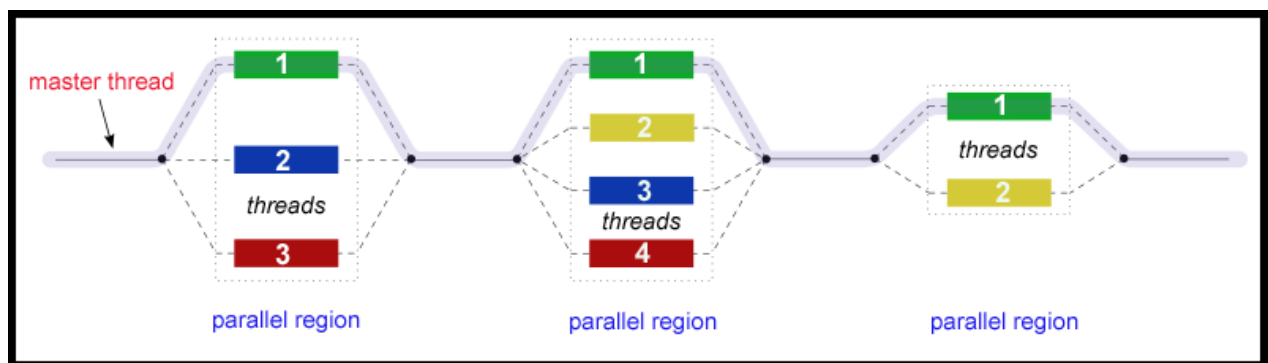


Figure 2- 28: The Fork and Join model

2.12.3.2 The Distributed Memory Model:

In the distributed memory model, jobs use their own memory and can exist either on their own physical machine or they can be transferred between a number of machines over a network using sending and receiving procedures as shown in Figure 2-29 (Barney, 2010).

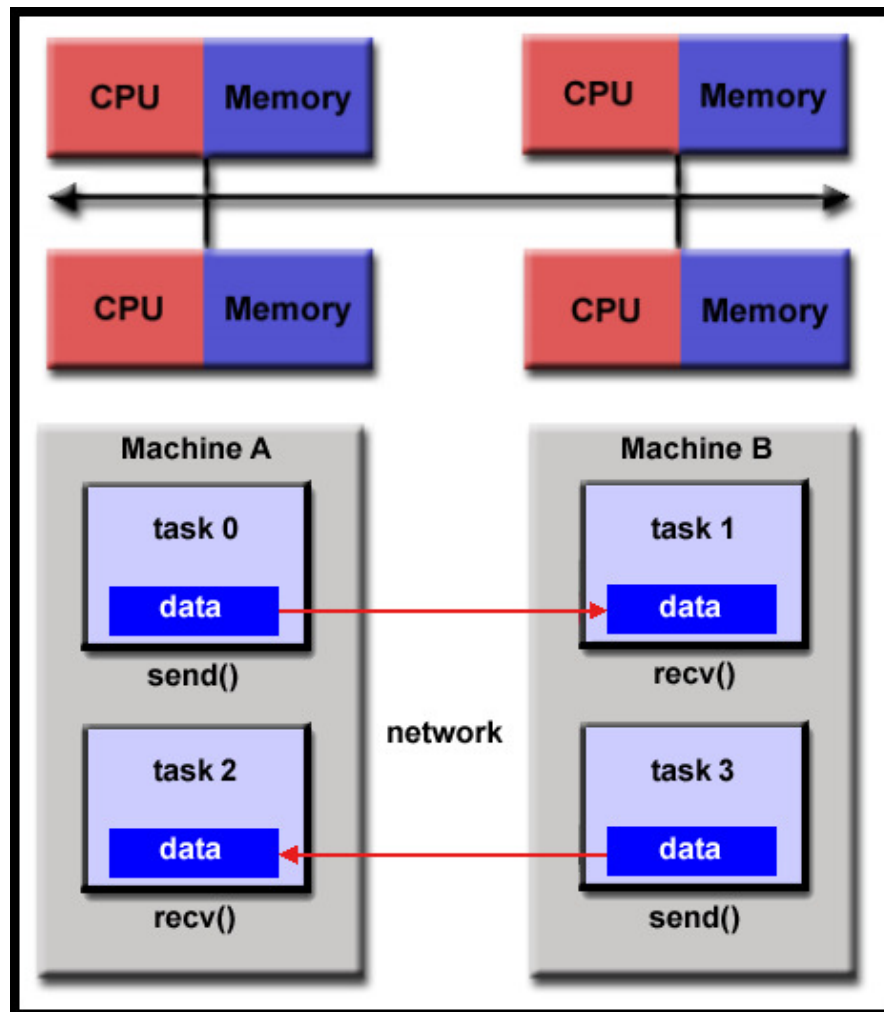


Figure 2-29: The distributed memory model

The Message Passing Interface (MPI) was released in 1991 as a programming interface for writing distributed memory applications, which include one or more communicators to send and receive messages through available nodes by calling library procedures to other processors. It is widely used on High Performance Computing (HPC) platforms. Figure 2-30 shows the MPI model structure (Barney, 2013).

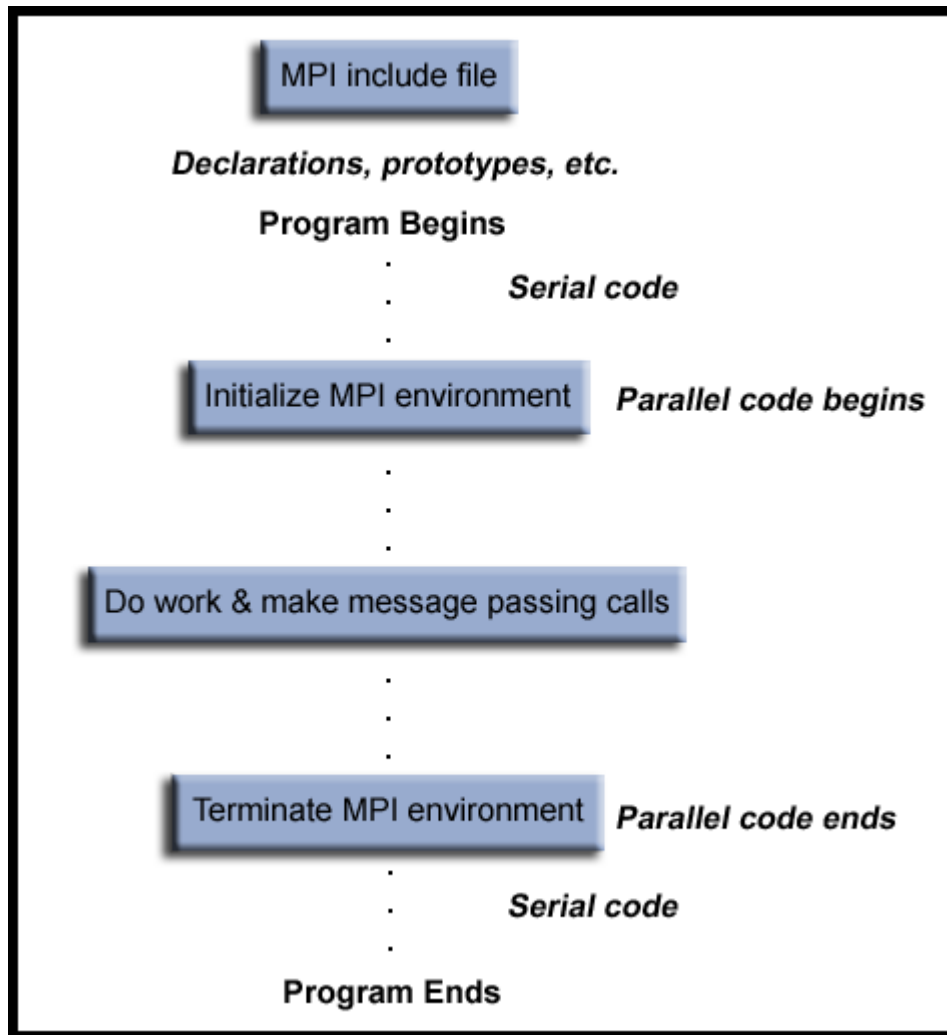


Figure 2- 30: The MPI Model structure

2.12.3.3 The Hybrid Memory Model:

The shared and distributed memory models are combined in the hybrid memory model as shown in Figure 2-31. In this model both the OpenMP and MPI models can be used. In this research, the hybrid model is not used and it is suggested as a future work.

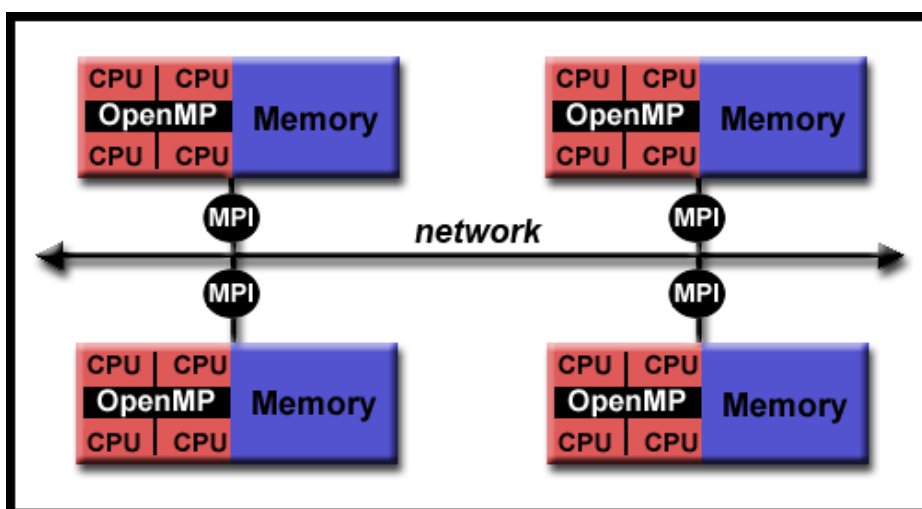


Figure 2- 31: The hybrid memory model

2.13 Summary

This chapter provided a survey of string matching algorithms where a new classification containing eight categories was developed depending on the preprocessing phase of searching algorithms.

The first category shifts the pattern only one position at each attempt. The second category uses two preprocessing functions. The third category uses one preprocessing function based on the rightmost character in the current window. The fourth category uses one preprocessing function based on the next character to the current window. The fifth category uses one preprocessing function based on the two characters next to the current window. The sixth category uses a preprocessing hashing function. The final category uses hybrid algorithms. A summary of all algorithms used in this research was listed in Table 2-26.

The SMILES chemical language with the syntax rules which can be used to convert two-dimensional chemical structure to a sequence was presented in section 2.11 and finally the parallel computing, Flynn's taxonomy and parallel programming models were presented in section 2.12.

CHAPTER 3: METHODOLOGY AND DESIGN

In this chapter, we present the research methodology framework, chemical toolkit design and parallel algorithm design. The research methodology framework achieves our initial research objectives. The new toolkit design shows the stages of developing a chemical structure searching toolkit using the developed SSN searching algorithm. Furthermore, the parallel algorithm design shows the phases of parallelizing the developed SSN algorithm using the OpenMP and the MPI models.

3.1 Framework of Research Methodology

The framework of the research methodology achieves the research objectives and consists of six stages as shown in Figure 3-1. The first stage studies the current existing algorithms. In addition, the second stage implements the algorithms studied in the first stage. Moreover, the third stage identifies (the) suitable algorithm(s) to be applied for searching biological sequence and chemical structure databases. The fourth stage is to enhance one or more of the existing algorithms or to develop (a) new algorithm(s). The fifth stage is to apply the new algorithm(s) to search biological sequence and chemical structure databases. The final stage is to measure the success of the new developed algorithm(s) compared to the currently existing algorithms.

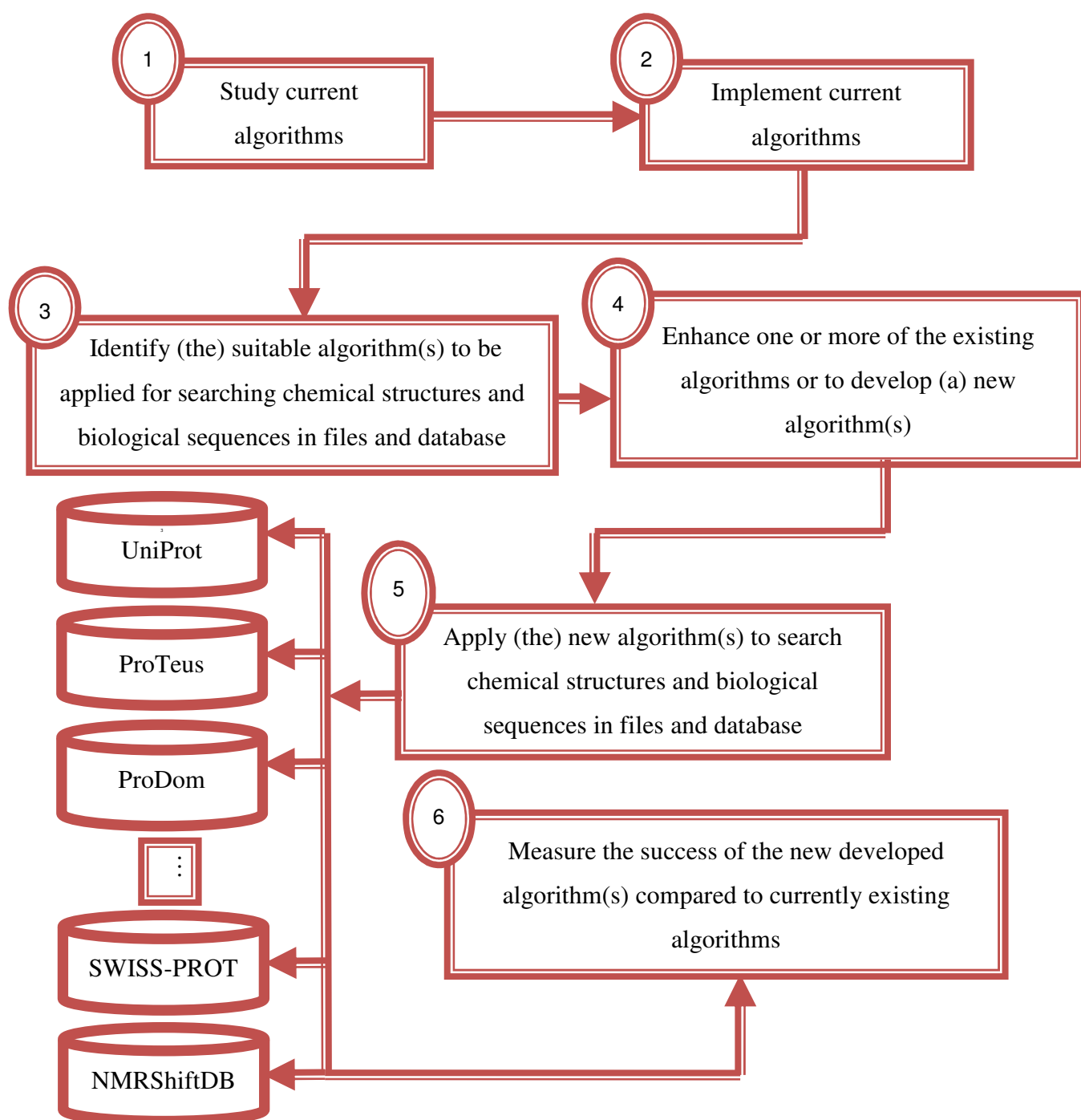


Figure 3- 1: Research objective framework

3.2 Chemical Structures Toolkit Design

The chemical structures toolkit design consists of four stages as shown in figure 3-2. The first stage downloads a sample data set from the NMRShiftDB (Kuhn, 2010). The second stage stores the downloaded data in a local database. The third stage uses Java Molecular Editor (JME), which was developed by Peter Ertl in 2000, to convert structures to SMILES format (Ertl, 2006, 2010 in press). Then our SSN matching algorithm searches a structure query in the local database as shown in section 5.2.3. The final stage uses the proportion of matching characters to measure the similarity between matched structures. Each stage will be discussed in detail in the following sections.

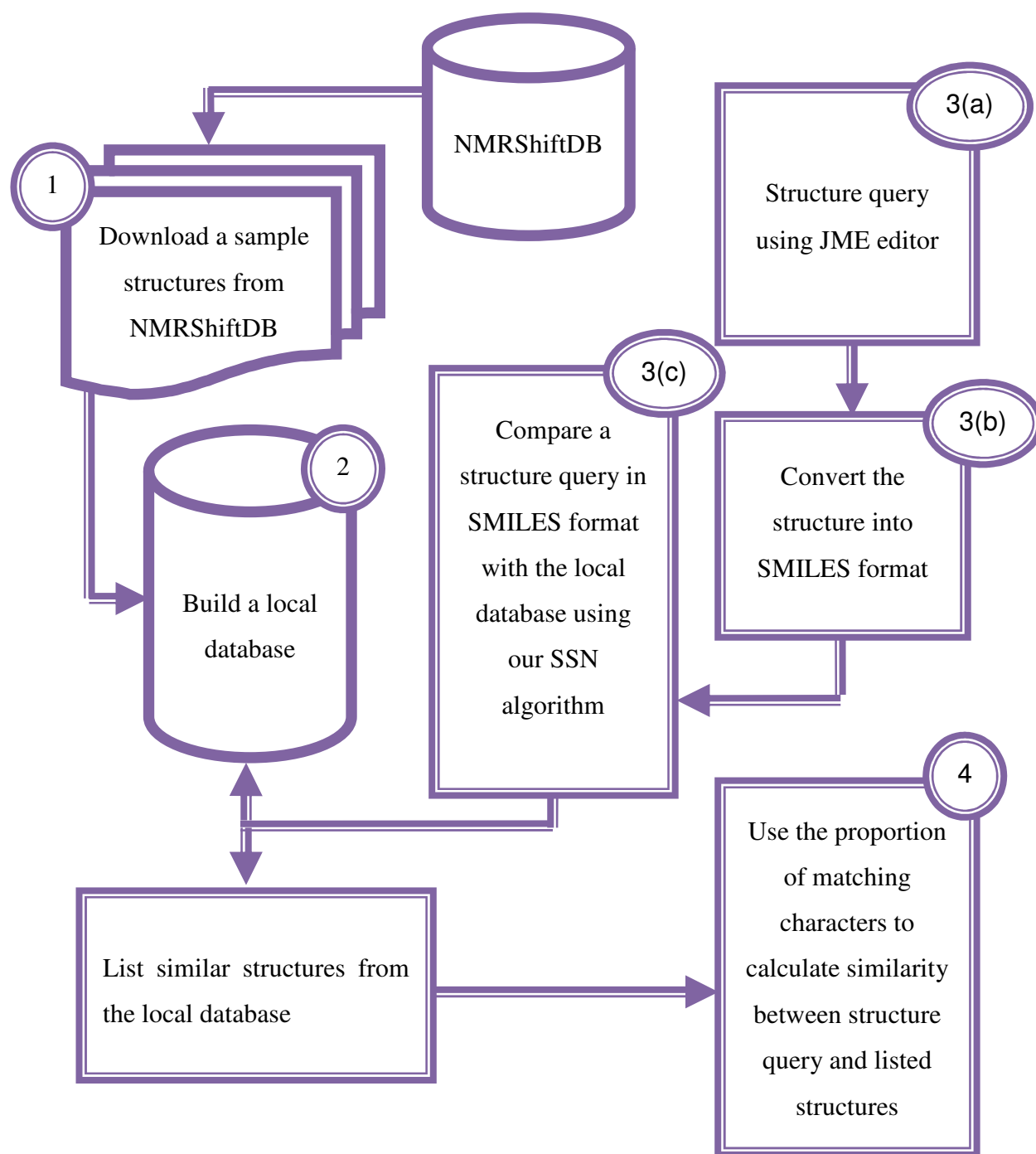


Figure 3- 2: Chemical toolkit design

3.2.1 The First Stage: Downloading and Mining Structures

We used the NMRShiftDB (Kuhn, 2010) database to download and mine chemical structures using keyword/category search such as “Antimicrobial”, “Antibacterial”, “Antifungal” and “Antiviral” from NMRShiftDB. Figure 3-3 below shows the mining process flowchart:

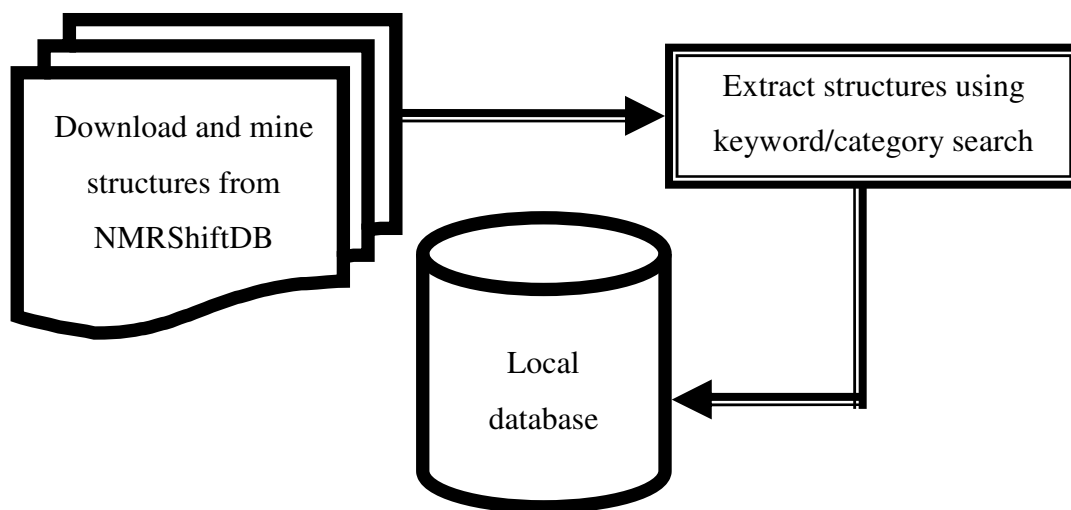


Figure 3- 3: Mining and downloading structures from NMRShiftDB

3.2.2 The Second Stage: Building the Local Database

Downloaded structures from the previous stage are stored in the local database. The local database includes relational tables to connect chemical molecules and their corresponding information. Figure 3-4 below shows the schema for the local database (Kuhn, 2010):

3.2.3 The Third Stage: Using the JME Editor, SMILES and the SSN Algorithm

The JME allows users to draw chemical structure and then converts the drawn structure into SMILES format (Ertl, 2006, 2010 in press). Figures 3-5 and 3-6 illustrate the structure drawing and conversion using the JME tool. The SSN algorithm is used to search the local database as described in section 4.5. Figure 3-7 below shows the structure matching flowchart using the SSN algorithm.

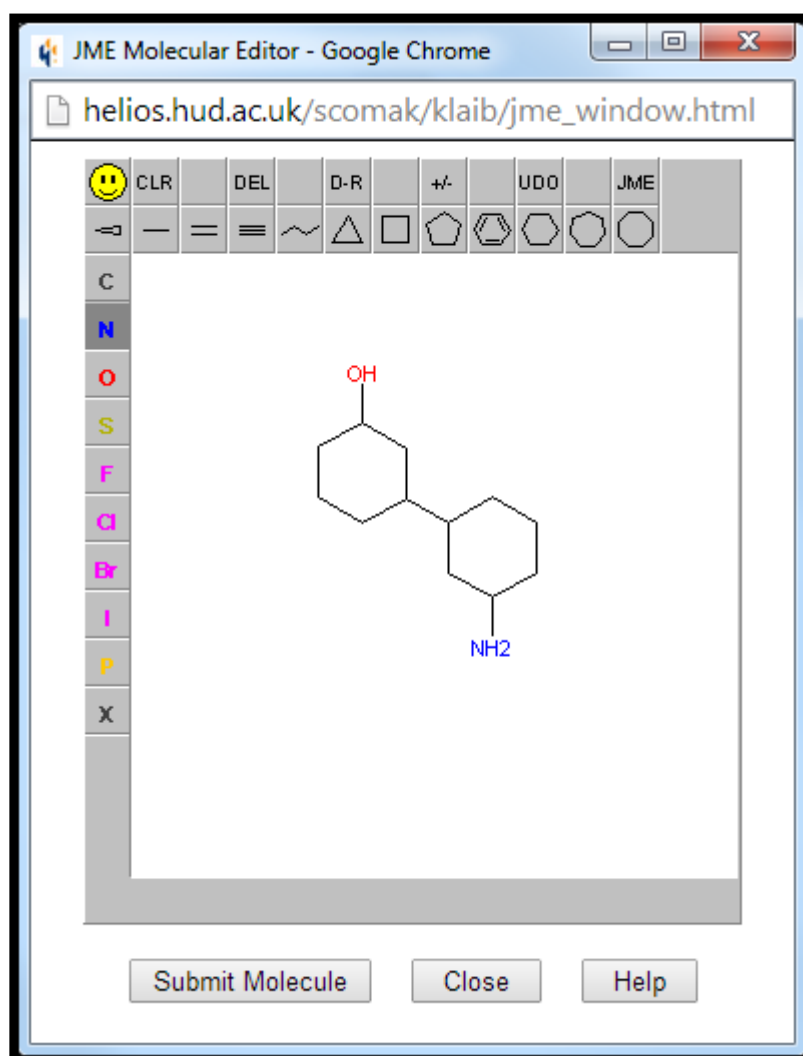


Figure 3- 5: Drawing a structure using JME tool

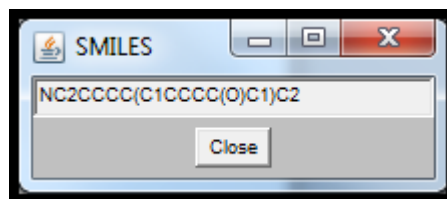


Figure 3- 6: Results of applying SMILES rules on Figure 3-5 example

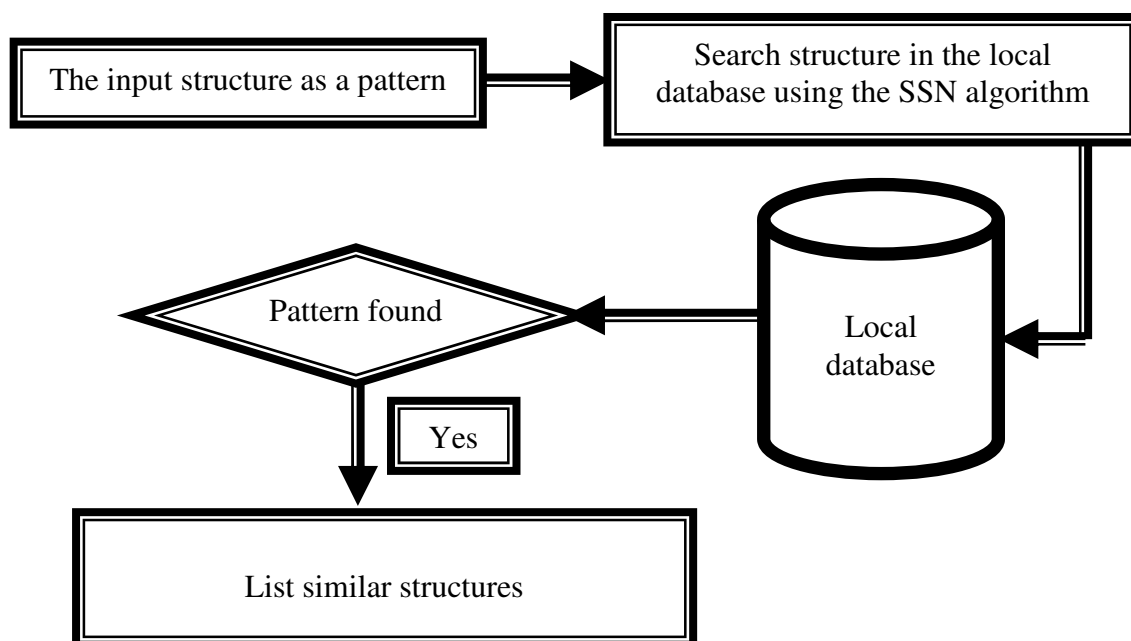


Figure 3- 7: Search structure in the local database using the SSN algorithm

3.2.4 The Fourth Stage: Measuring Similarity Using the Proportion of Matching Characters

Similarity is a quantity function that reflects the strength of relationship between two objects or two features with the idea that a higher value indicates greater similarity. In this stage the similarity is presented using the proportion of matching characters as the similarity measure, and that this is acceptable as a Jaccard's coefficient (Teknomo, 2006; Schulz, 2008).

3.3 Parallel Algorithm Design

Parallel computing can be used where tasks, calculation and problems can be divided into smaller ones that can be worked on simultaneously. Our parallel algorithm design includes a new contribution where the SSN algorithm will be parallelized using the OpenMP and the MPI models to improve the speed of searching a pattern in the given text. In this framework we are dividing the parallel algorithms on two levels.

3.3.1 Parallel Algorithm Design for Shared Memory Model

In this level the DNA text file is stored in a shared memory address and the SSN algorithm is parallelized using the OpenMP model. In the SSN algorithm, independent “for loops” and “if conditions” are defined in parallel regions. This gives a MISD system where, in the parallel regions, separate threads run on the same data. Figure 3-8 shows an example of the “for loop” parallelized using the OpenMP model.

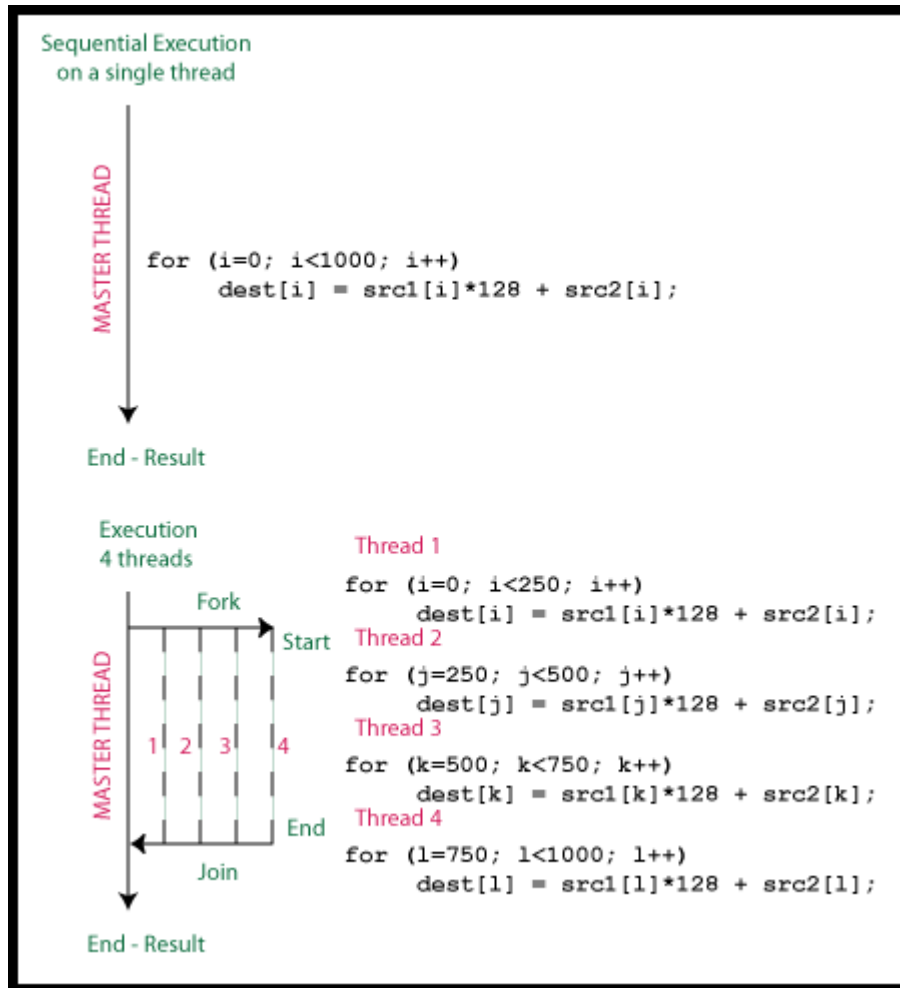


Figure 3- 8: An example of parallelizing the “for loop” using the OpenMP model

3.3.2 Parallel Algorithm Design for Distributed Memory Model

Here a SIMD system is created using the MPI model. The DNA text file is divided by the master node (first processor) into subtexts. The number of subtexts is based on the number of available nodes.

The master sends the subtext and query pattern to each available node. Each node starts comparing the pattern with the text using the SSN algorithm and sends the result back to the master. Finally, the master combines the results together and prints out the final result. Figure 3- 9 shows the parallel algorithm design using the MPI model.

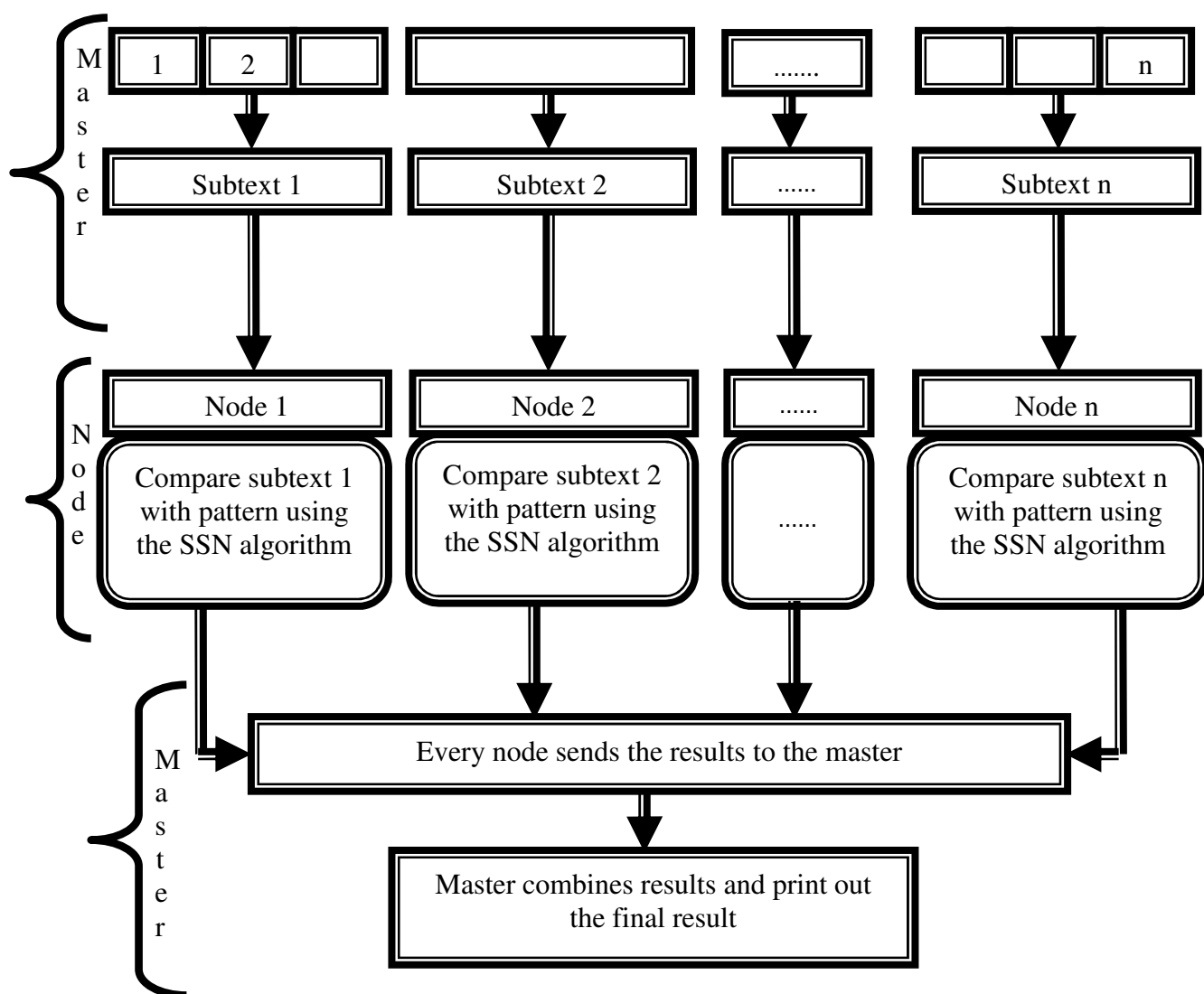


Figure 3- 9: Parallel algorithm design using the MPI Model

3.4 Summary

This chapter provided an overview of the proposed work in this thesis. It also includes the research methodology framework, chemical toolkit design and parallel algorithm design. The research methodology framework achieved our research objectives through six stages as explained in section 3.1. The new toolkit design developed a chemical structure searching toolkit using the SSN algorithm through four stages as explained in section 3.2. The parallel algorithm design presented the OpenMP model of the SSN algorithm in sub-section 3.3.1 and the MPI model in sub-section 3.3.2.

CHAPTER 4: DEVELOPING NEW ALGORITHMS

As mentioned earlier, efficient string algorithms aim to maximize the pattern shifting value and therefore enhance searching time. In this research the main string matching algorithms were therefore classified in chapter two into eight categories according to the preprocessing function of the algorithm (Klaib & Osborne, 2009a). In addition, we propose five new algorithms which aim to maximize the pattern shifting value and therefore enhance searching time.

The BRBMH (Klaib & Osborne, 2008) algorithm is explained in section 4.1, the BRQS (Klaib & Osborne, 2009a) is explained in section 4.2, the OE algorithm (Klaib & Osborne, 2009b) is explained in section 4.3, the RSMA (Klaib & Osborne, 2009c) algorithm is explained in section 4.4 and the SSN algorithm is explained in section 4.5.

4.1 The BRBMH Algorithm

The BRBMH algorithm uses the same searching process as the BMH algorithm and if there is a whole match or a mismatch, the enhanced brBc table is used as described in section 4.1.1 (Klaib & Osborne, 2008).

4.1.1 The Preprocessing Phase of the BRBMH Algorithm:

The preprocessing function of the BMH algorithm computes the shifts using only one heuristic function based on the last character in the current text window (Crochemore, et al., 1994). In addition, the preprocessing phase of the BR algorithm scans all text characters and uses

a two-dimensional array to shift the pattern to $m+2$ if the two characters located after the current window do not exist in the pattern.

In order to maximize the pattern shifting value and therefore enhancing the searching time, firstly the BR preprocessing function (brBc) was enhanced and then it was used instead of the BMH preprocessing function (hsBc).

The enhancement of the brBc preprocessing function includes creating a one dimensional array to store the pattern characters rather than using a two dimensional array to store the text characters. Therefore, using the enhanced brBc over the hsBc results in two benefits: the first one is the brBc table shifts the pattern to the right by $m+2$ compared to the hsBc which shifts pattern only m positions if there is a whole match or a mismatch encountered. Furthermore, the second benefit is reducing the preprocessing time by scanning only the pattern characters rather than scanning the text characters as in the original brBc.

There are four shift cases in the BRBMH preprocessing phase as shown in Figure 4-1, Figure 4-2, Figure 4-3 and Figure 4-4 where:

- (a) and (b) are the first and the second characters next to the current text window and any of them or both of them can be exist in pattern characters at location (i).
- (g) and (t) are the first and the second characters next to the current text window and they do not exist at all in pattern characters.
- (k) is the size of the previous compared text portion which starts from t_0 to the beginning of current text window.

Case 1: when the pattern $[m-1] = \text{text}[m+k]$, then the pattern will be shifted only one position.

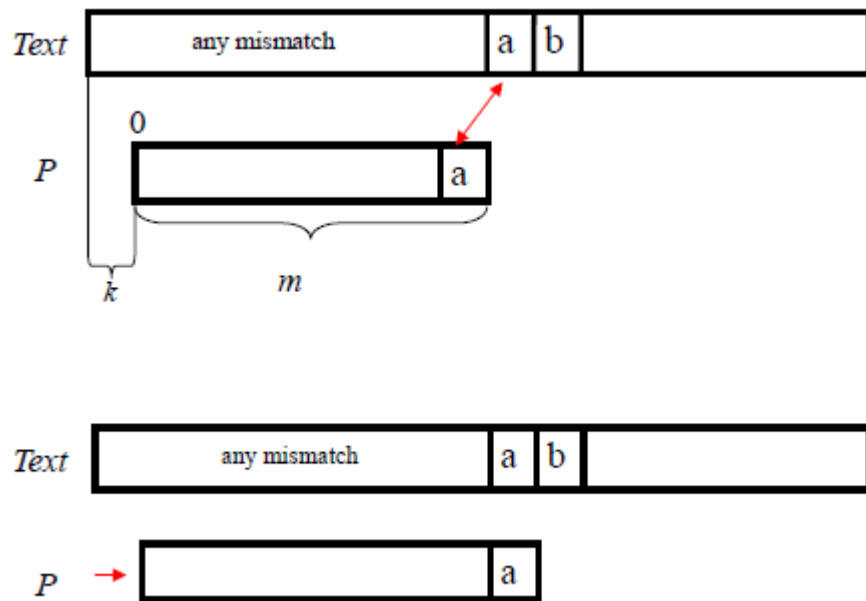


Figure 4- 1: The first shift case of the BRBMH algorithm

Case 2: when the pattern $[i] = \text{text}[m+k]$ and pattern $[i+1] = \text{text}[m+k+1]$, then the pattern will be shifted $m-i+1$.

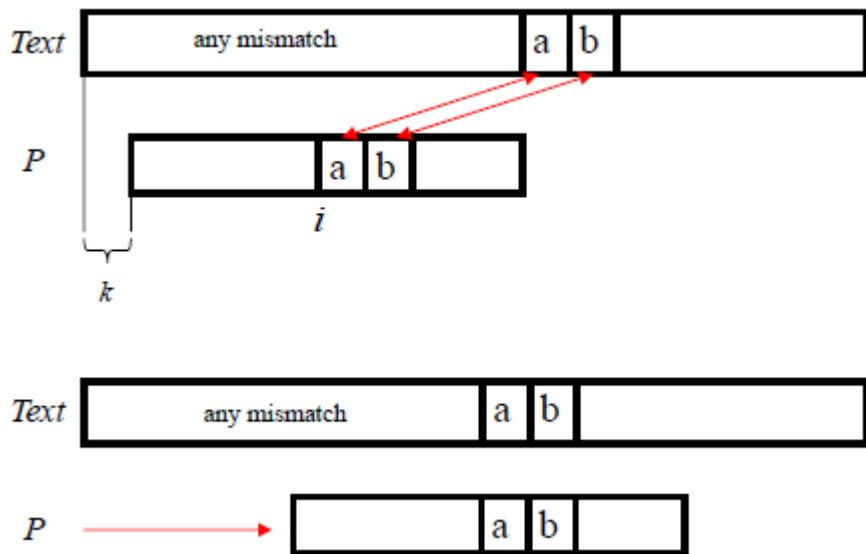


Figure 4- 2: The second shift case of the BRBMH algorithm

Case 3: when the pattern $[0] = \text{text}[m+k+1]$, then the pattern will be shifted $m+1$.

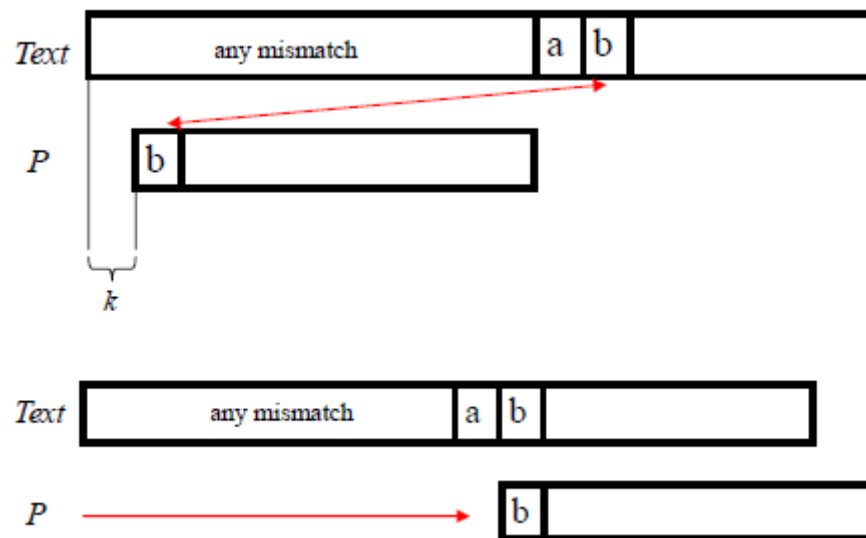
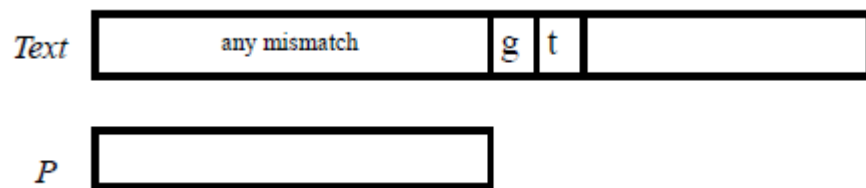


Figure 4- 3: The third shift case of the BRBMH algorithm

Case 4: when the pattern $[i]$ and pattern $[i+1]$ do not exist in the text window, then the pattern will be shifted $m+2$.



If characters (g) and (t) are not in the pattern, then $\text{shift} = m+2$

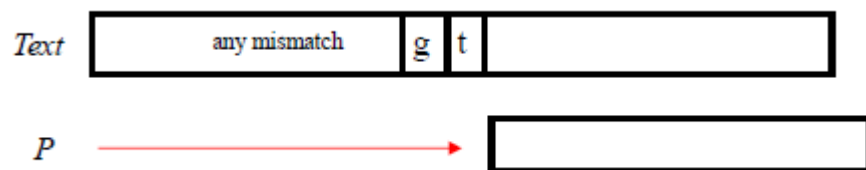


Figure 4- 4: The fourth shift case of the BRBMH algorithm

4.1.2 The Searching Phase of the BRBMH Algorithm:

The searching phase of the BRBMH algorithm starts from the rightmost character in the current window, and then starts from the leftmost character and moves to the right until the penultimate character. Figure 4-5 shows the BRBMH algorithm code for the preprocessing and searching phases.

```
void preprocessing_enhancedBrBc(char *pattern, int patternLength, int enhancedBrBc[patternLength - 1]) {
    int iCounter, jCounter;
    char *enhancedBrBcCharacters;
    for (iCounter = 0; iCounter < patternLength; ++iCounter){
        enhancedBrBcCharacters[iCounter]=pattern.substr(iCounter,2);
    }
    for (jCounter = 0; jCounter <= patternLength-2; ++jCounter){
        enhancedBrBc[jCounter] = patternLength - jCounter;
    }
}

void BRBMH(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, jCounter, patternCounter, lastCharacter, enhancedBrBc[patternLength-1];
    char nextTwoCharacters;
    /* Preprocessing */
    preprocessing_enhancedBrBc(pattern, patternLength, enhancedBrBc);

    /* Searching */
    iCounter = 0;
    while (iCounter <= textLength - patternLength) {
        patternCounter = patternLength - 1;
        jCounter = iCounter + patternLength - 1;
        lastCharacter = iCounter + patternLength - 1;
        if (pattern[jCounter] == text[lastCharacter]){
            patternCounter = 0;
            while (jCounter >= iCounter && pattern[patternCounter] == text[iCounter]) {
                --jCounter; ++patternCounter;
            }
            if (jCounter < iCounter) {
                OUTPUT(iCounter);
            }
        }
        nextTwoCharacters=text[patternLength+iCounter] + text[patternLength+1+iCounter];
        iCounter += enhancedBrBc[nextTwoCharacters];
    }
}
```

Figure 4- 5: The BRBMH algorithm code

4.1.3 The BRBMH: Working Example

The input sample, the preprocessing phase and searching phase in sub-sections 4.1.3.1, 4.1.3.2 and 4.1.3.3 respectively, provide an example of the BRBMH algorithm.

4.1.3.1 Input Sample

Text Length (n) = 64	LRFDSLYKQILAMGLAVKANQHIVLAVKLATA IVLATHTSPVVPVTTTPGTKPDLNASFVSANAE
Pattern Length (m) = 12	LAVKLATAIVLA

Table 4- 1: The BRBMH input sample

4.1.3.2 The BRBMH Example's Preprocessing Phase

Figure 4-1 calculates the enhanced brBc table for input sample, as shown in the Table 4-2:

LA	AV	VK	KL	LA	AT	TA	AI	IV	VL	LA
12	11	10	9	8	7	6	5	4	3	2

Table 4-2: The enhanced brBc table of the BRBMH algorithm

4.1.3.3 The BRBMH Example's Searching Phase

As explained in sub-section 4.1.2, the searching order of the BRBMH algorithm is shown in Table 4-3.

P ₀	P ₁	P ₂	P ₃	P ₄	P ₅
2	3	4	5	6	7
P ₆	P ₇	P ₈	P ₉	P ₁₀	P ₁₁
8	9	10	11	12	1

Table 4-3: The BRBMH algorithm searching order

1. **In the first attempt:** the current text window starts from position 0 to position 11 as shown in Table 4-4.

1 st ATT	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16
Text	L	R	F	D	S	L	Y	K	Q	I	L	A	M	G	L	A	...
BRBMH	2	3										1					
Pattern	L	A	V	K	L	A	T	A	I	V	L	A					
1 st ATT	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11					
Shift by 14 (brBc[MG])																	

Table 4-4: The first attempt of the BRBMH algorithm

- The BRBMH Algorithm: the first comparison between t_{11} and p_{11} causes a match. The searching phase in sub-section 4.1.3.3 starts from the leftmost character t_0 with p_0 which matched. Furthermore, the algorithm moves forward to the next character t_1 with p_1 which produces a mismatch.
- The preprocessing phase in sub-section 4.1.3.2 uses the brBc table for t_{12} and t_{13} (MG). However, (MG) does not exist in Table 4-2, so the shifting value of (MG) is $m+2$ and in this case is 14 positions.

2. **In the second attempt:** the current text window starts from position 14 to position 25 as shown in Table 4-5.

2 nd ATT	T14	T15	T16	T17	T18	T19	T20	T21	T22	T23	T24	T25	T26	T27	T28	T29	T30
Text	L	A	V	K	A	N	Q	H	I	V	L	A	V	K	L	A	...
BRBMH	2	3	4	5	6							1					
Pattern	L	A	V	K	L	A	T	A	I	V	L	A					
2 nd ATT	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11					
Shift by 10 (brBc[VK])																	

Table 4-5: The second attempt of BRBMH

- The BRBMH Algorithm: the first comparison in the second attempt is between t_{25} and p_{11} which produces a match. The searching phase in sub-section 4.1.3.3 starts from leftmost character t_{14} with p_0 which matched. The algorithm moves forward to the next character and follows the same procedure until the comparison between t_{18} with p_4 produces a mismatch.
- The preprocessing phase in sub-section 4.1.3.2 uses the brBc table for t_{26} and t_{27} (VK). The shifting value of (VK) from Table 4-2 is 10 positions.

3. In the third attempt: the current text window starts from position 24 to position 35 as shown in Table 4-6.

3 rd ATT	T24	T25	T26	T27	T28	T29	T30	T31	T32	T33	T34	T35	T36	T37	T38	T39	T40
Text	L	A	V	K	L	A	T	A	I	V	L	A	T	H	T	S	...
BRBMH	2	3	4	5	6	7	8	9	10	11	12	1					
Pattern	L	A	V	K	L	A	T	A	I	V	L	A					
3 rd ATT	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11					
Shift by 14 (brBc[TH])																	

Table 4-6: The third attempt of BRBMH

- The BRBMH Algorithm: the first comparison in the third attempt is between t_{35} and p_{11} which creates a match. The searching phase in sub-section 4.1.3.3 starts from leftmost

character t_{24} with p_0 which matched. The algorithm moves forward to the next character and follows the same procedure until a whole match of the pattern is found in the text.

- After a whole match is found in the current attempt, the preprocessing phase in sub-section 4.1.3.2 uses the brBc table for t_{36} and t_{37} (TH). However, (TH) does not exist in Table 4-2, so the shifting value of (TH) is $m+2$ and in this case is 14 positions.

4. In the fourth attempt: the current text window starts from position 38 to position 49 as shown in Table 4-7.

4 th ATT	T38	T39	T40	T41	T42	T43	T44	T45	T46	T47	T48	T49	T50	T51	T52	T53	T54
Text	T	S	P	V	V	P	V	T	T	P	G	T	K	P	D	L	...
BRBMH	1																
Pattern	L	A	V	K	L	A	T	A	I	V	L	A					
4 th ATT	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11					
Shift by 14 (brBc[TH])																	

Table 4-7: The fourth attempt of BRBMH

- In BRBMH, the first comparison in the fourth attempt is between t_{49} and p_{11} which generates a mismatch.
- The preprocessing phase in sub-section 4.1.3.2 uses the brBc table for t_{50} and t_{51} (KP). However, (KP) does not exist in Table 4-2, so the shifting value of (KP) is $m+2$ and in this case is 14 positions.
- The BRBMH algorithm ignores the pattern shifting since the pattern is longer than the remaining text.

The results of the BRBMH algorithm are presented and discussed in chapter 6.

4.2 The BRQS Algorithm

The BRQS algorithm (Klaib & Osborne, 2009a) uses the same searching process as the QS algorithm and if there is a whole match or a mismatch, the enhanced brBc table is used as described in sub-section 4.1.1.

4.2.1 The Preprocessing Phase of the BRQS Algorithm:

The BRQS algorithm uses the same enhanced pre-processing phase as the BRBMH algorithm as shown in sub-section 4.1.1.

4.2.2 The Searching Phase of the BRQS Algorithm:

The searching phase of BRQS algorithm starts from the leftmost character then it moves forward single position each occasion up to the rightmost character. Figure 4-6 shows the BRQS algorithm code for the preprocessing and searching phases.

```

void preprocessing_enhancedBrBc(char *pattern, int patternLength, int enhancedBrBc[patternLength - 1]) {
    int iCounter, jCounter;
    char *enhancedBrBcCharacters;
    for (iCounter = 0; iCounter < patternLength; ++iCounter){
        enhancedBrBcCharacters[iCounter]=pattern.substr(iCounter,2);
    }
    for (jCounter = 0; jCounter <= patternLength-2; ++jCounter){
        enhancedBrBc[jCounter] = patternLength - jCounter;
    }
}

void BRQS(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, jCounter, patternCounter, lastCharacter, enhancedBrBc[patternLength-1];
    char nextTwoCharacters;
    /* Preprocessing */
    preprocessing_enhancedBrBc(pattern, patternLength, enhancedBrBc);

    /* Searching */
    iCounter = 0;
    while (iCounter <= textLength - patternLength) {
        patternCounter = 0;
        jCounter = iCounter + patternLength - 1;
        while (jCounter >= iCounter && pattern[patternCounter] == text[iCounter]) {
            --jCounter;
            ++patternCounter;
        }
        if (jCounter < iCounter){
            OUTPUT(iCounter);
        }
        nextTwoCharacters=text[patternLength+iCounter] + text[patternLength+1+iCounter];
        iCounter += enhancedBrBc[nextTwoCharacters];
    }
}

```

Figure 4- 6: The BRQS algorithm code

4.2.3 The BRQS: Working Example

The input sample, the preprocessing phase and searching phase in sub-sections 4.2.3.1, 4.2.3.2 and 4.2.3.3 respectively, provide an example of the BRQS algorithm.

4.2.3.1 Input Sample

Text Length (n) = 64	LRFDSLYKQILAMGLAVKANQHIVLAVKLATA IVLATHTSPVVPVTTTPGTKPDLNASFVSANAE
Pattern Length (m) = 12	LAVKLATAIVLA

Table 4-8: BRQS input sample

4.2.3.2 The BRQS Example's Preprocessing Phase

Figure 4-6 calculates the enhanced brBc table for the input sample, as shown in Table 4-9:

LA	AV	VK	KL	LA	AT	TA	AI	IV	VL	LA
12	11	10	9	8	7	6	5	4	3	2

Table 4- 9: The BRQS enhanced brBc preprocessing table

4.2.3.3 The BRQS Example's Searching Phase

As explained in sub-section 4.2.2, the searching order of the BRQS algorithm is shown in the Table 4-10.

P ₀	P ₁	P ₂	P ₃	P ₄	P ₅
1	2	3	4	5	6
P ₆	P ₇	P ₈	P ₉	P ₁₀	P ₁₁
7	8	9	10	11	12

Table 4- 10: The BRQS algorithm searching order

1. **In the first attempt:** the current text window starts from position 0 to position 11 as shown in Table 4-11.

1 st ATT	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16
Text	L	R	F	D	S	L	Y	K	Q	I	L	A	M	G	L	A	...
BRQS	1	2															
Pattern	L	A	V	K	L	A	T	A	I	V	L	A					
1 st ATT	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11					
Shift by 14 (brBc[MG])																	

Table 4-11: The BRQS algorithm: the first attempt in the searching phase

- The BRQS Algorithm: the first comparison between t_0 and p_0 causes a match. The searching phase in sub-section 4.2.3.3 moves to the next position t_1 with p_1 which produces a mismatch.
- The preprocessing phase in sub-section 4.2.3.2 uses the brBc table for t_{12} and t_{13} (MG). However, (MG) does not exist in Table 4-9, so the shifting value of (MG) is $m+2$ and in this case is 14 positions.

2. **In the second attempt:** the current text window starts from position 14 to position 25 as shown in Table 4-12.

2 nd ATT	T14	T15	T16	T17	T18	T19	T20	T21	T22	T23	T24	T25	T26	T27	T28	T29	T30
Text	L	A	V	K	A	N	Q	H	I	V	L	A	V	K	L	A	...
BRQS	1	2	3	4	5												
Pattern	L	A	V	K	L	A	T	A	I	V	L	A					
2 nd ATT	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11					
Shift by 10 (brBc[VK])																	

Table 4-12: The BRQS algorithm: the second attempt in the searching phase

- The BRQS Algorithm: the first comparison in the second attempt is between t_{14} and p_0 which creates a match. The searching phase in sub-section 4.2.3.3 moves forward and compares the next positions until the comparison between t_{18} with p_4 produces a mismatch.
- The preprocessing phase in sub-section 4.2.3.2 uses the brBc table for t_{26} and t_{27} (VK). The shifting value of (VK) from Table 4-9 is 10 positions.

3. In the third attempt: the current text window starts from position 24 to position 35 as shown in Table 4-13.

3 rd ATT	T24	T25	T26	T27	T28	T29	T30	T31	T32	T33	T34	T35	T36	T37	T38	T39	T40
Text	L	A	V	K	L	A	T	A	I	V	L	A	T	H	T	S	...
BRQS	1	2	3	4	5	6	7	8	9	10	11	12					
Pattern	L	A	V	K	L	A	T	A	I	V	L	A					
3 rd ATT	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11					
Shift by 14 (brBc[TH])																	

Table 4-13: The BRQS algorithm: the third attempt in the searching phase

- The BRQS Algorithm: the first comparison in the third attempt is between t_{24} and p_0 which creates a match. The searching phase in sub-section 4.2.3.3 starts from next position t_{25} with p_1 which matched. The algorithm moves forward to the next characters and follows the same procedure until a whole match of the pattern is found in the text.
- After a whole match is found in the current attempt, the preprocessing phase in sub-section 4.2.3.2 uses the brBc table for t_{36} and t_{37} (TH). However, (TH) does not exist in Table 4-9, so the shifting value of (TH) is $m+2$ and in this case is 14 positions.

4. In the fourth attempt: the current text window starts from position 38 to position 49 as shown in Table 4-14.

4 th ATT	T38	T39	T40	T41	T42	T43	T44	T45	T46	T47	T48	T49	T50	T51	T52	T53	T54
Text	T	S	P	V	V	P	V	T	T	P	G	T	K	P	D	L	...
BRQS	1																
Pattern	L	A	V	K	L	A	T	A	I	V	L	A					
4 th ATT	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11					
Shift by 14 (brBc[TH])																	

Table 4-14: The BRQS algorithm: the fourth attempt in the searching phase

- In BRQS, the first comparison in the fourth attempt is between t_{38} and p_0 which generates a mismatch.
- The preprocessing phase in sub-section 4.2.3.2 uses the brBc table for t_{50} and t_{51} (KP). The shifting value of (KP) from Table 4-2 is 10 positions. However, (KP) does not exist in Table 4-9, so the shifting value of (KP) is $m+2$ and in this case is 14 positions.
- The BRQS algorithm ignores the pattern shifting since the pattern is longer than the remaining text.

The results of the BRQS algorithm are presented and discussed in chapter 6.

4.3 The Odd and Even Algorithm (OE)

The OE algorithm searches the pattern in the text using a new searching order and if there is a whole match or a mismatch, the enhanced brBc table is used as described in sub-section 4.1.1 ([Klaib & Osborne, 2009b](#)).

4.3.1 The Preprocessing Phase of the OE Algorithm:

The OE algorithm uses the same enhanced pre-processing phase as the BRBMH algorithm as shown in sub-section 4.1.1.

4.3.2 The Searching Phase of the OE Algorithm:

The OE algorithm searches the pattern in the text using a new searching order. It starts with the rightmost position, and then it moves toward the rear to compare the odd index positions of pattern and text window characters. If all these characters match, it starts from right and compares the whole even index pattern and text window characters. Figure 4-7 shows the OE algorithm code for the preprocessing and searching phases.

```

void preprocessing_enhancedBrBc(char *pattern, int patternLength, int enhancedBrBc[patternLength - 1]) {
    int iCounter, jCounter;
    char *enhancedBrBcCharacters;
    for (iCounter = 0; iCounter < patternLength; ++iCounter){
        enhancedBrBcCharacters[iCounter]=pattern.substr(iCounter,2);
    }
    for (jCounter = 0; jCounter <= patternLength-2; ++jCounter){
        enhancedBrBc[jCounter] = patternLength - jCounter;
    }
}

void OddAndEven(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, jCounter, patternCounter, lastCharacter, enhancedBrBc[patternLength-1];
    char nextTwoCharacters;
    bool Odd=false;
    /* Preprocessing */
    preprocessing_enhancedBrBc(pattern, patternLength, enhancedBrBc);

    /* Searching */
    iCounter = 0;
    while (iCounter <= textLength - patternLength) {
        if(patternLength % 2 != 0){
            jCounter = patternLength-2;
            Odd=true;
        }
        else{
            jCounter = patternLength - 1;
        }
        while (jCounter >= 0 && pattern[jCounter] == text[iCounter + jCounter]){
            jCounter -= 2;
            if(jCounter == 1 && Odd == true)
            {
                jCounter = patternLength-1;
            }
            else{
                jCounter = patternLength-2;
            }
        }
        if (jCounter < 0){
            OUTPUT(iCounter);
        }
        nextTwoCharacters=text[patternLength+iCounter] + text[patternLength+1+iCounter];
        iCounter += enhancedBrBc[nextTwoCharacters];
    }
}

```

Figure 4-7: The OE algorithm code

4.3.3 The OE: Working Example

The input sample, the preprocessing phase and searching phase in sub-sections 4.3.3.1,

4.3.3.2 and 4.3.3.3 respectively, provide an example of the BRBMH algorithm.

4.3.3.1 Input Sample

Text Length (n) = 64	LRFDSLYKQILAMGLAVKANQHIVLAVKLATA IVLATHTSPVVPVTTTPGTPDLNASFVSANAE
Pattern Length (m) = 12	LAVKLATAIVLA

Table 4- 15: OE input sample

4.3.3.2 The OE Example's Preprocessing Phase

Figure 4-7 calculates the enhanced brBc table for input sample, as shown in Table 4-15:

LA	AV	VK	KL	LA	AT	TA	AI	IV	VL	LA
12	11	10	9	8	7	6	5	4	3	2

Table 4-15: The OE enhanced brBc preprocessing table

4.3.3.3 The OE Example's Searching Phase

As explained in sub-section 4.3.2, the searching order of the OE algorithm is shown in Table 4-16.

P ₀	P ₁	P ₂	P ₃	P ₄	P ₅
12	6	11	5	10	4
P ₆	P ₇	P ₈	P ₉	P ₁₀	P ₁₁
9	3	8	2	7	1

Table 4- 16: The OE algorithm searching order

- 1. In the first attempt:** the current text window starts from position 0 to position 11 as shown in Table 4-17.

1 st ATT	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16
Text	L	R	F	D	S	L	Y	K	Q	I	L	A	M	G	L	A	...
OE	2										1						
Pattern	L	A	V	K	L	A	T	A	I	V	L	A					
1 st ATT	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11					
Shift by 14 (brBc[MG])																	

Table 4- 17: The OE algorithm: the first attempt in the searching phase

- The OE Algorithm: the first comparison between t_{11} and p_{11} causes a match. The searching phase in sub-section 4.3.3.3 compares the next odd position t_9 with p_9 which produces a mismatch.
- The preprocessing phase in sub-section 4.3.3.2 uses the brBc table for t_{12} and t_{13} (MG). However, (MG) does not exist in Table 4-15, so the shifting value of (MG) is $m+2$ and in this case is 14 positions.

- 2. In the second attempt:** the current text window starts from position 14 to position 25 as shown in Table 4-18.

2 nd ATT	T14	T15	T16	T17	T18	T19	T20	T21	T22	T23	T24	T25	T26	T27	T28	T29	T30	
Text	L	A	V	K	A	N	Q	H	I	V	L	A	V	K	L	A	...	
OE								3				2						1
Pattern	L	A	V	K	L	A	T	A	I	V	L	A						
2 nd ATT	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11						
Shift by 10 (brBc[VK])																		

Table 4-18: The OE algorithm: the second attempt in the searching phase

- The OE Algorithm: the first comparison in the second attempt is between t_{25} and p_{11} which creates a match. The searching phase in sub-section 4.3.3.3 moves forward and compares the next odd index positions until the comparison between t_{21} with p_7 produces a mismatch.
- The preprocessing phase in sub-section 4.3.3.2 uses the brBc table for t_{26} and t_{27} (VK). The shifting value of (VK) from Table 4-15 is 10 positions.

3. In the third attempt: the current text window starts from position 24 to position 35 as shown in Table 4-19.

3 rd ATT	T24	T25	T26	T27	T28	T29	T30	T31	T32	T33	T34	T35	T36	T37	T38	T39	T40
Text	L	A	V	K	L	A	T	A	I	V	L	A	T	H	T	S	...
OE	12	6	11	5	10	4	9	3	8	2	7	1					
Pattern	L	A	V	K	L	A	T	A	I	V	L	A					
3 rd ATT	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11					
Shift by 14 (brBc[TH])																	

Table 4-19: The OE algorithm: the third attempt in the searching phase

- The OE Algorithm: the first comparison in the third attempt is between t_{35} and p_{11} which creates a match. The searching phase in sub-section 4.3.3.3 moves forward and compares

the next odd index positions until the comparison between t_{25} with p_1 produces a match.

It then returns and compares t_{34} with p_{10} and follows the same procedure for all even indices until a whole match of the pattern is found in the text.

- After a whole match is found in the current attempt, the preprocessing phase in sub-section 4.3.3.2 uses the brBc table for t_{36} and t_{37} (TH). However, (TH) does not exist in Table 4-15, so the shifting value of (TH) is $m+2$ and in this case is 14 positions.

- 4. In the fourth attempt:** the current text window starts from position 38 to position 49 as shown in Table 4-20.

4 th ATT	T38	T39	T40	T41	T42	T43	T44	T45	T46	T47	T48	T49	T50	T51	T52	T53	T54
Text	T	S	P	V	V	P	V	T	T	P	G	T	K	P	D	L	...
OE	1																
Pattern	L	A	V	K	L	A	T	A	I	V	L	A					
4 th ATT	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11					
Shift by 14 (brBc[TH])																	

Table 4-20: The OE algorithm: the fourth attempt in the searching phase

- In OE, the first comparison in the fourth attempt is between t_{49} and p_{11} which generates a mismatch.
- The preprocessing phase in sub-section 4.3.3.2 uses the brBc table for t_{50} and t_{51} (KP). However, (KP) does not exist in Table 4-15, so the shifting value of (KP) is $m+2$ and in this case is 14 positions.
- The OE algorithm ignores the pattern shifting since the pattern is longer than the remaining text.

The results of OE algorithm are presented and discussed in chapter 6.

4.4 The Random String Matching Algorithm (RSMA)

The RSMA searches the pattern in the text using a new searching order and if there is a whole match or a mismatch, the enhanced brBc table is used as described in sub-section 4.1.1 ([Klaib & Osborne, 2009c](#)).

4.4.1 The Pre-processing Phase of the RSMA Algorithm:

This RSMA algorithm uses the same enhanced pre-processing phase as the BRBMH algorithm as shown in section 4.1.1.

4.4.2 The Searching Phase of the RSMA Algorithm

The RSMA algorithm starts with the rightmost character, and then it searches the pattern in the text using a new searching order using a random value ([Klaib & Osborne, 2009c](#)).

A random step size S is generated, with $3 \leq S < m$, where m is the pattern length. The search will then move along the pattern visiting every i^{th} character until reaching or passing the end of the pattern, to return to the start, and repeat the process one character further down the pattern. This is shown in Equation (2) and Equation (3) where $P_{i,j}$ is the position visited on the j^{th} step of the i^{th} pass. Figure 4-8 shows the RSMA searching phase equations and Figure 4-9 shows the RSMA algorithm code for the preprocessing and searching phases.

$$P_{i,0} = m - i - 1 \dots\dots\dots(3)$$

$$P_{i,j+1} = \begin{cases} P_{i,j} - S & \text{if } P_{i,j} \geq S, \\ P_{i+1,0} & \text{Otherwise} \end{cases} \dots\dots\dots(4)$$

Figure 4- 8: The RSMA algorithm searching phase equations

```

void preprocessing_enhancedBrBc(char *pattern, int patternLength, int enhancedBrBc[patternLength - 1]) {
    int iCounter, jCounter;
    char *enhancedBrBcCharacters;
    for (iCounter = 0; iCounter < patternLength; ++iCounter){
        enhancedBrBcCharacters[iCounter]=pattern.substr(iCounter,2);
    }
    for (jCounter = 0; jCounter <= patternLength-2; ++jCounter){
        enhancedBrBc[jCounter] = patternLength - jCounter;
    }
}

void RSMA(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, jCounter, nCounter, patternCounter, lastCharacter, enhancedBrBc[patternLength-1];
    char nextTwoCharacters;
    /* Preprocessing */
    preprocessing_enhancedBrBc(pattern, patternLength, enhancedBrBc);

    /* Searching */
    int aCounter = 1, bCounter = 0, randomValue, *newOrder;
    randomValue = rand() % patternLength + 1;
    newOrder[0]=patternLength-1;
    for (iCounter=1; iCounter<= randomValue; i++){
        while(newOrder[aCounter]- randomValue >=0 && bCounter<patternLength)
        {
            newOrder[bCounter]=newOrder[aCounter]- randomValue;
            ++aCounter;
            ++bCounter;
        }
        if (bCounter < patternLength)
        {
            newOrder[bCounter]=newOrder[0]-randomValue;
            aCounter=bCounter;
            ++bCounter;
        }
    }
    iCounter = 0, jCounter=0, nCounter=0;
    while (iCounter <= textLength - patternLength) {
        jCounter = patternLength-1;
        while (jCounter >= 0 && pattern[newOrder[nCounter]] == text[iCounter + newOrder[nCounter]]){
            --jCounter;
            ++nCounter;
        }
        if (jCounter < 0){
            OUTPUT(iCounter);
        }
        nextTwoCharacters=text[patternLength+iCounter] + text[patternLength+1+iCounter];
        iCounter += enhancedBrBc[nextTwoCharacters];
    }
}

```

Figure 4- 9: The RSMA algorithm code

4.4.3 The RSMA: Working Example

The input sample in sub-section 4.4.3.1, the preprocessing phase in sub-section 4.4.3.2 and the searching phase in sub-section 4.4.3.3 provide an example of the RSMA algorithm.

4.4.3.1 Input Sample

Text Length (n) = 64	LRFDSLYKQILAMGLAVKANQHIVLAVKLATA IVLATHTSPVVPVTTTPGTKPDLNASFVSANAE
Pattern Length (m) = 12	LAVKLATAIVLA
Random Division Value (RD) =	3 (Random Value)

Table 4-21: RSMA Input Sample

4.4.3.2 The RSMA Example's Preprocessing Phase

Figure 4-1 calculates the enhanced brBc table for input sample, as shown in Table 4-22:

LA	AV	VK	KL	LA	AT	TA	AI	IV	VL	LA
12	11	10	9	8	7	6	5	4	3	2

Table 4-22: The RSMA enhanced brBc preprocessing table

4.4.3.3 The RSMA Example's Searching Phase

As explained in section 4.4.2, in this example the random value size (S) is equal to three. The RSMA algorithm uses Equations (2) and (3) in Figure 4-8 to calculate the new searching order as shown in Table 4-23.

P ₀	P ₁	P ₂	P ₃	P ₄	P ₅
12	8	4	11	7	3
P ₆	P ₇	P ₈	P ₉	P ₁₀	P ₁₁
10	6	2	9	5	1

Table 4-23: The RSMA algorithm searching order

1. **In the first attempt:** the current text window starts from position 0 to position 11 as shown in Table 4-24.

1 st ATT	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16
Text	L	R	F	D	S	L	Y	K	Q	I	L	A	M	G	L	A	...
RSMA	2										1						
Pattern	L	A	V	K	L	A	T	A	I	V	L	A					
1 st ATT	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11					
Shift by 14 (brBc[MG])																	

Table 4-24: The RSMA algorithm: the first attempt in the searching phase

- The RSMA Algorithm: the first comparison between t_{11} and p_{11} causes a match. The searching phase in sub-section 4.4.3.3 compares the next position t_8 with p_8 which produces a mismatch.
- The preprocessing phase in sub-section 4.4.3.2 uses the brBc table for t_{12} and t_{13} (MG). However, (MG) does not exist in Table 4-22, so the shifting value of (MG) is $m+2$ and in this case is 14 positions.

2. In the second attempt: the current text window starts from position 14 to position 25 as shown in Table 4-25.

2 nd ATT	T14	T15	T16	T17	T18	T19	T20	T21	T22	T23	T24	T25	T26	T27	T28	T29	T30
Text	L	A	V	K	A	N	Q	H	I	V	L	A	V	K	L	A	...
RSMA	3						2			1							
Pattern	L	A	V	K	L	A	T	A	I	V	L	A					
2 nd ATT	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11					
Shift by 10 (brBc[VK])																	

Table 4-20: The RSMA algorithm: the second attempt in the searching phase

- The RSMA Algorithm: the first comparison in the second attempt is between t_{25} and p_{11} which creates a match. The searching phase in sub-section 4.4.3.3 moves forward and compares the next positions until the comparison between t_{19} with p_5 produces a mismatch.
- The preprocessing phase in sub-section 4.4.3.2 uses the brBc table for t_{26} and t_{27} (VK). The shifting value of (VK) from Table 4-22 is 10 positions.

3. In the third attempt: the current text window starts from position 24 to position 35 as shown in Table 4-26.

3 rd ATT	T24	T25	T26	T27	T28	T29	T30	T31	T32	T33	T34	T35	T36	T37	T38	T39	T40
Text	L	A	V	K	L	A	T	A	I	V	L	A	T	H	T	S	...
RSMA	12	8	4	11	7	3	10	6	2	9	5	1					
Pattern	L	A	V	K	L	A	T	A	I	V	L	A					
3 rd ATT	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11					
Shift by 14 (brBc[TH])																	

Table 4-21: The RSMA algorithm: the third attempt in the searching phase

- The RSMA Algorithm: the first comparison in the third attempt is between t_{35} and p_{11} which creates a match. The searching phase in sub-section 4.4.3.3 moves forward and follows the same procedure for all indices until a whole match of the pattern is found in the text.
- After a whole match is found in the current attempt, the preprocessing phase in sub-section 4.4.3.2 uses the brBc table for t_{36} and t_{37} (TH). However, (TH) does not exist in Table 4-22, so the shifting value of (TH) is $m+2$ and in this case is 14 positions.

4. In the fourth attempt: the current text window starts from position 38 to position 49 as shown in Table 4-27.

4 th ATT	T38	T39	T40	T41	T42	T43	T44	T45	T46	T47	T48	T49	T50	T51	T52	T53	T54
Text	T	S	P	V	V	P	V	T	T	P	G	T	K	P	D	L	...
RSMA	1																
Pattern	L	A	V	K	L	A	T	A	I	V	L	A					
4 th ATT	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11					
Shift by 14 (brBc[KP])																	

Table 4-22: The RSMA algorithm: the fourth attempt in the searching phase

- In RSMA, the first comparison in the fourth attempt is between t_{49} and p_{11} which generates a mismatch.
- The preprocessing phase in sub-section 4.4.3.2 uses the brBc table for t_{50} and t_{51} (KP). However, (KP) does not exist in Table 4-22, so the shifting value of (KP) is $m+2$ and in this case is 14 positions.
- The RSMA algorithm ignores the pattern shifting since the pattern is longer than the remaining text.

The results of RSMA algorithm are presented and discussed in chapter 6.

4.5 The Skip Shift New (SSN) Algorithm

The searching phase of our algorithms RSMA, OE, BRQS and BMH starts from the first m positions to compare pattern and text characters and if a whole match or a mismatch it uses the enhanced brBc table to shift the pattern depending on the next two characters to the rightmost character as described in section 4.1.1. Starting from the first m positions may cause m comparisons before the pattern is shifted to the right.

The preprocessing function of the SS and the ASS algorithms aim to find a possible starting point before comparing the pattern with the text which reduces the number of comparisons. However, the disadvantages of using the searching phase of the SS algorithm is the use of all text and pattern alphabets to create the bucket for shifting the pattern as explained in sub-section 2.8.1 (Almazroi & Rashid, 2011) and then moves the pattern only a single position if the last character does not exist in the pattern.. The disadvantages of using the searching phase of the ASS algorithm is the use of all substrings positions with length $L=\log_{\sigma}(m)$ for each leaf of $T(x)$ to shift the pattern as explained in sub-section 2.8.2 and then moves the pattern only a single position if the $T(x)$ does not exist in the pattern (Charras, et al., 1998)

An enhancement of the ASS algorithm was presented in 2011 through the ASSBR algorithm (section 2.27) which uses the brBc table to shift the pattern using the next two characters to the rightmost character which causes a bigger shifting value comparing to the original ASS algorithm. However, the disadvantage of the ASSBR algorithm is the use of two preprocessing function to determine the starting point and then to shift the pattern (Almazroi, 2011).

In our SSN algorithm, if there is a whole match or a mismatch the ASS table is used to define a possible starting point to compare the text and the pattern characters. In addition, the ASS table is used as a new preprocessing phase to shift the pattern as described in section 4.5.1.

4.5.1 The Preprocessing Phase of the SSN Algorithm:

The preprocessing phase of the SSN algorithm uses the ASS table only in contrast to the ASSBR algorithm. It determines the starting position of each three character sequence in the pattern as presented in the given example in sub-section 4.5.3.2. The shifting value of the three letters depends on the following Equation (5) as shown in Figure 4-10:

$$\text{ShiftingValue} = m - \text{position} \dots\dots\dots(5)$$

Figure 4- 10: The SSN algorithm preprocessing phase equation

4.5.2 The Searching Phase of the SSN Algorithm:

In the searching phase, the algorithm first checks for a possible starting point by checking the last three characters in the text window, if they exist in the pattern it aligns the pattern with the text and then compares the remaining characters from the leftmost character to the rightmost character. If a whole match or a mismatch is found it compares the next three characters to the rightmost character with the ASS table instead of two characters in the same way as the enhanced brBc table. If the next three characters exist in the pattern, then they are aligned again, otherwise the pattern is shifted $m+3$ positions. An example is presented in sub-section 4.5.3.3.

Figure 4-11 shows the SSN algorithm code for the preprocessing and searching phases.

```

void preprocessing_enhancedASS(char *pattern, int patternLength, int enhancedBrBc[patternLength - 1]) {
    int iCounter, jCounter;
    char *enhancedBrBcCharacters;
    for (iCounter = 0; iCounter < patternLength; ++iCounter){
        enhancedBrBcCharacters[iCounter]=pattern.substr(iCounter,3);
    }
    for (jCounter = 0; jCounter <= patternLength-3; ++jCounter){
        enhancedBrBc[jCounter] = patternLength - jCounter;
    }
}

void SSN(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, patternCounter, enhancedASS[patternLength-1];

    /* Preprocessing */
    preprocessing_enhancedASS(pattern, patternLength, enhancedASS);

    /* Searching */
    int wStart = 0, wEnd = patternLength - 1;
    char currentThree, nextThree;
    while (wEnd <= textLength) {
        currentThree = text[wEnd-2] + text[wEnd-1] + text[wEnd];
        if ((wEnd + 3) < textLength){
            nextThree = text[wEnd+1] + text[wEnd+2] + text[wEnd+3];
        }
        for(iCounter=0; iCounter<= patternLength-3; iCounter++) {
            if(currentThree == enhancedASS[iCounter])
            {
                wStart = wStart+enhancedASS[iCounter]-3;
                wEnd = wStart + (patternLength - 1);
            }
            else if(nextThree == enhancedASS[iCounter])
            {
                wStart = wStart+enhancedASS[iCounter];
                wEnd = wStart + (patternLength - 1);
            }
            else
            {
                wStart = wStart+ patternLength + 3;
                wEnd = wStart + (patternLength - 1);
            }
        }
        patternCounter = 0;
        while (wEnd >= wStart && pattern[patternCounter] == text[wStart]) {
            --wEnd;
            ++patternCounter;
        }
        if (wEnd < wStart){
            OUTPUT(iCounter);
        }
        wStart = wEnd+1;
        wEnd = wStart + (patternLength - 1);
    }
}

```

Figure 4- 11: The SSN algorithm code

4.5.3 The SSN: Working Example

The input sample, the preprocessing phase and searching phase in sub-sections 4.5.3.1, 4.5.3.2 and 4.5.3.3 respectively, provide an example of the SSN algorithm.

4.5.3.1 Input Sample

Text Length (n) = 64	LRFDSLYKQILAMGLAVKANQHIVLAVKLATA IVLATHTSPVVPVTTTPGTKPDLNASFVSANAE
Pattern Length (m) = 12	LAVKLATAIVLA

Table 4- 21: The SSN input sample

4.5.3.2 The SSN Example's Preprocessing Phase

The preprocessing phase calculates the ASS table for input sample using three characters, as shown in Table 4-22:

Character	Alpha Skip table[character] = m-position
LAV	{0} = 12
AVK	{1} = 11
VKL	{2} = 10
KLA	{3} = 9
LAT	{4} = 8
ATA	{5} = 7
TAI	{6} = 6
AIV	{7} = 5
IVL	{8} = 4
VLA	{9} = 3

Table 4-22: The ASS table of the SSN algorithm

4.5.3.3 The SSN Example's Searching Phase

As explained in sub-section 4.5.2, the searching phase checks for a possible starting point by checking the last three characters in the text window. If it exists in the pattern it aligns the pattern with the text and then compares the remaining characters from the leftmost character to the rightmost character, otherwise it shifts pattern to the right depending on the next three characters to the rightmost character.

1. **In the first attempt:** the last three characters in the current text window starts from position 9 to position 11 as shown in Table 4-23.

1 st ATT	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16
Text	L	R	F	D	S	L	Y	K	Q	I	L	A	M	G	L	A	...
										Does Not Exist							
Does Not Match: MGL (m+3) = 15																	

Table 4-23: The SSN algorithm: the first attempt in the searching phase

- The SSN Algorithm: the (ILA) portion in positions t_9 , t_{10} and t_{11} does not exist in the ASS table in Table 4-22.
- The new preprocessing phase uses the ASS table for t_{12} , t_{13} and t_{14} (MGL). However, (MGL) does not exist in Table 4-22, so the shifting value of (MGL) is $m+3$ and in this case is 15 positions.

2. In the second attempt: the last three characters in the current text window starts from position 24 to position 26 as shown in Table 4-24 (a).

2 nd ATT	T15	T16	T17	T18	T19	T20	T21	T22	T23	T24	T25	T26	T27	T28	T29	T30	T31
Text	A	V	K	A	N	Q	H	I	V	L	A	V	K	L	A	T	A
										Exist							
Does Match: LAV (m) = 12																	

Table 4-24 (a): The SSN algorithm: the second attempt in the searching phase

- The SSN Algorithm: the (LAV) portion in positions t_9 , t_{10} and t_{11} exists in the ASS table.
- The searching phase in this attempt aligns the pattern to match the similar portions as shown in Table 4-24 (b).

2 nd ATT	T24	T25	T26	T27	T28	T29	T30	T31	T32	T33	T34	T35	T36	T37	T38	T39	T40
Text	L	A	V	K	L	A	T	A	I	V	L	A	T	H	T	S	P
				1	2	3	4	5	6	7	8	9					
Pattern	L	A	V	K	L	A	T	A	I	V	L	A					
2 nd ATT	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11					
A whole match and (THT) Does Not Match: $(m+3) = 15$																	

Table 4-25(b): The ASS_NEW algorithm: the second attempt in the searching phase

- The searching phase compares the remaining characters from the leftmost character to the rightmost character starting from a comparison between t_{27} with P_3 and moving forward until a comparison between t_{35} with P_{11} which causes a whole match.
- After a whole match is found in the current attempt, the new preprocessing phase uses the ASS table for t_{36} , t_{37} and t_{38} (THT). However, (THT) does not exist in Table 4-22, so the shifting value of (THT) is $m+3$ and in this case is 15 positions.

3. In the third attempt: the last three characters in the current text window starts from position 48 to position 50 as shown in Table 4-25.

3 rd ATT	T39	T40	T41	T42	T43	T44	T45	T46	T47	T48	T49	T50	T51	T52	T53	T54	T55
Text	S	P	V	V	P	V	T	T	P	G	T	K	P	D	L	N	A
										Does Not Exist							
Does Not Match: (m+3) = 15 T54 – T65																	

Table 4-26: The ASS_NEW algorithm: the third attempt in the searching phase

- The SSN Algorithm: the (GTK) portion in positions t_{48} , t_{49} and t_{50} does not exist in the ASS table.
- The new preprocessing phase uses the ASS table for t_{51} , t_{52} and t_{53} (PDL). However,

(PDL) does not exist in Table 4-22, so the shifting value of (PDL) is $m+3$ and in this case is 15 positions.

- The SSN algorithm ignores the pattern shifting since the pattern is longer than the remaining text.

The results of SSN algorithm are presented and discussed in chapter 6.

4.6 Summary

This chapter illustrated five new string matching algorithms: The BRBMH, BRQS, OE, RSMA and the SSN algorithms.

The BRBMH algorithm uses a searching process in the same way as the BMH searching phase and if there is a whole match or a mismatch it uses the enhanced brBc table of the Berry-Ravindran algorithm. The BRQS algorithm uses a searching process in the same way as the QS searching phase and if there is a whole match or a mismatch it uses the pre-processing phase of the BRBMH algorithm. The OE algorithm searches the pattern in the text using a new searching order by comparing the odd indices first and then even indices and if there is a whole match or a mismatch it uses the pre-processing phase of the BRBMH algorithm. The RSMA algorithm starts with the rightmost character, and then it searches the pattern in the text using a new searching order using a random value with size S to visit all pattern character positions and if there is a whole match or a mismatch it uses the pre-processing phase of the BRBMH algorithm. Finally, the SSN algorithm uses the ASS table to define a possible starting point to compare the text and the pattern characters. If the last three characters in the current text window or the next three characters exists in the ASS table, the pattern is aligned and compared otherwise the ASS table is used again to shift the pattern.

CHAPTER 5: IMPLEMENTATION

5.1 System Specification

In this project, sequential algorithms have been implemented and tested on our PC (Windows 7 64-bit, RAM is 4.00GB and CPU is Intel Core i3 2.40 GHz). The chemical toolkit is developed using Dreamweaver CS6 using HTML/JavaScript, MySQL and PHP code.

The parallel algorithm part was implemented and tested at the University of Science Malaysia (USM) on two units of the Stealth Cluster. Each unit is Sun Fire V210 which contains 2 x UltraSPARC IIIi 1002 Mhz processors, 1MB L2 Cache 2GB of RAM. They use the following software: Solaris9 (SunOS 2.9) as an operating system.

5.2 Chemical Structures Toolkit Implementation

The toolkit implementation is presented in four stages as explained in sub-sections 5.2.1, 5.2.2, 5.2.3 and 5.2.4.

5.2.1 Downloading and Mining Structures

The NMRShiftDB database contains more than forty thousand records representing different molecular structures (Kuhn, 2010).

The extraction process of structures, as mentioned in chapter 3 (section 3.3), is done via search by Keyword/Category such as antiviral activity, antifungal activity and antibacterial activity. Figure C-1 shows a sample of extracted structures.

5.2.2 Building the Local Database

Downloaded structures from the previous stage are stored in the local database. The local database includes 78 relational tables to connect chemical molecules and their corresponding information.

5.2.2.1 Local Database Design

Our Local Database schema is shown in Figures C-2, C-3, C-4 and C-5.

5.2.2.2 Table Format and Description

In this section we provide descriptions for three tables as examples while the remaining tables are shown in Appendix D. These main tables are related to each other as they store the details of the chemical structure such as keyword_id, keyword, molecule_id, molecule_weight, SMILES_string and so on.

1- Molecule Table

This table contains seven fields storing the main information about a particular molecule. An example is given in Table 5-1:

Field	Type	Null	Key	Example
MOLECULE_ID	int(11)		PRI	22207
DATE	datetime			2006-06-14 14:53:57
MOLECULAR_WEIGHT	float		MUL	542.572
SMILES_STRING	mediumblob		MUL	<chem>O=C1OC4C(=C1C)CC=C(COC3OC(COC2OCC(O)(CO)C2(O))C(O)C(O)C3(O))CCC=C(C)C4</chem>
USER_ID	int(11)		MUL	30003481
SAR	int(11)			5
COMMENT	longblob	YES		NULL

Table 5- 1: Molecule table design and example

The following is a brief explanation for each field:

- **MOLECULE_ID:** is a unique number assigned to each molecule in the database.
- **Date:** is the date and time when the molecule was inserted in the NMRShiftDB.
- **MOLECULAR_WEIGHT:** is the total weight of all molecular atoms and can be presented by mass units (u) where u is the mass of 1/12 of Carbon atom (C^{12}) (Steven, 2011; NIST, 2013).
- **SMILES_STRING:** is the chemical structure presentation in sequence format (Daylight Chemical Information Systems, 2008).
- **USER_ID:** is a unique number assigned for each user using the NMRShiftDB database.
- **SAR:** is the Structure Activity Relationships (SAR) which presents relations between the molecular structure and biological or physicochemical activity of chemicals (Hulzebos et al., 2001).
- **COMMENT:** any additional information can be added about the molecule.

2- Keyword Table

This table stores the keyword information about a particular molecule. It contains 3 fields as shown in Table 5-2. Note that the entire keyword field in our database should be one of the antimicrobial keywords mined in section 5.2.

Field	Type	Null	Key	Default	Example
KEYWORD_ID	int(11)		PRI	0	20010639
KEYWORD	varchar(255)				antibacterial and antitumor effects
KEYWORD_SOUNDEX	mediumtext				53123645353561232

Table 5- 2: Keyword table design and example

3- Molecule-Keyword Table

This table presents the relationship between the Keyword and Molecule tables which includes two primary key fields as shown in Table 5-3:

Field	Type	Null	Key	Default	Example
MOLECULE_ID	int(11)		PRI	0	22207
KEYWORD_ID	int(11)		PRI	0	20010639

Table 5- 3: Molecule_Keyword table design and example

5.2.3 Using JME Editor, SMILES and the SSN Algorithm

The implementation of this stage includes the development of the chemical searching toolkit, which allows users to draw the molecular structure and then converts it into a structure in SMILES format. Figure C-6 shows the toolkit login page and Figure C-7 shows the searching toolkit webpage. The SMILES's format for any chemical structure can either be typed straight into the specified text field or can be drawn using the JME editor after clicking the "Draw molecule" button. After the drawn chemical structure is submitted, it is converted to SMILES format as shown in Figure C-8. The "Check SMILES" button in Figure C-9 allows users to check whether the entered SMILES sequence is correct or not. Finally, the "Search structure using the Skip Shift New Algorithm" button searches the structure pattern in the local database using the SSN algorithm.

5.2.4 Implementation of Similarity Measuring

In this phase the proportion of matching characters is used as a measure of the similarity percentage between two structures.

Table 5-4 shows an example where the pattern structure is “C3CCC4” and the text structure is “OCCCC1CC(OC)C2OC(CC2(C1))C3CCC4OCOC4(C3)”, then the implementation process will be done according to the following steps:

- 1) Find the length of the searched text and the searched pattern:

OCCCC1CC(OC)C2OC(CC2(C1))C3CCC4OCOC4(C3)➔	Text Length = 40
C3CCC4➔	Pattern Length = 5

Table 5- 4: Finding the text and pattern length for similarity measuring

- 2) Calculate the proportion of matching (S_{ab}) characters by dividing the pattern length on the text length as following:

$$S_{ab} = \frac{\text{Pattern Length}}{\text{Text Length}} = \frac{5}{40} = 0.125$$

The above example shows that the searched pattern is only similar to the compared structure by 12.5% and different by 87.5%.

5.3 Parallel Algorithm Implementation

The parallel algorithm is designed using the OpenMP model (shared memory model) and the MPI model (distributed memory model). Both models were implemented and tested at the University of Science Malaysia (USM) on a Stealth Cluster which consists of four Sun Fire V210 containing 2 x UltraSPARC IIIi 1002 Mhz processors, 1MB L2 Cache 2GB of RAM. They use the SunOS 2.9 operating system and Sun Studio 12 as a Compiler.

5.3.1 OpenMP Model Implementation

The OpenMP model is implemented on a single unit of the Sun Fire V210 using four threads. Microsoft Visual Studio 2013 Express is used to write the C++ code of the OpenMP model. The

OpenMP model should be activated on Microsoft Visual Studio before writing the program as shown in Figure C-10.

The results of the OpenMP model of the SSN algorithm are presented and discussed in chapter 6. Table 5-5 lists the main OpenMP functions which are used to parallelize the SSN algorithm:

Function	The function's job
<code>#pragma omp parallel</code>	To define an OpenMP parallel region
<code>#pragma omp parallel for</code>	The <code>for</code> refers that we are parallelizing the for loop.
<code>#pragma omp parallel for</code> <code>shared(patternLength, shiftArray, iCounter)</code> <code>num_threads(NUM_THREADS)</code>	<code>Shared</code> will specify the number of shared variables used by the threads and <code>num_threads</code> specifies the number of threads to use for the parallel portion.
<code>#pragma omp critical</code>	Specifies a region of code that must be executed by only one thread at a time. If a thread is currently executing inside a CRITICAL region and another thread reaches that CRITICAL region and attempts to execute it, it will block until the first thread exits that CRITICAL region.
<code>#pragma omp for schedule (static, chunk)</code>	Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent. In our program we have used the STATIC keyword where loop iterations are divided into pieces of size chunk and then statically assigned to threads.
<code>#pragma omp parallel</code> <code>num_threads(NUM_THREADS)</code>	Specifies the number of threads that should execute the following block. It is used in the mean searching phase where if conditions have been applied.

Table 5- 5: The main OpenMP functions which used to parallelize the SSN algorithm

5.3.2 MPI Model Implementation

The MPI model is implemented on two units of the Sun Fire V210 using four processors. Microsoft Visual Studio 2013 Express is used to write the C++ code of the MPI model. The HPC package should be installed from the Microsoft website to add the MPI model library to Microsoft Visual Studio before writing the program as shown in Figure C-11. Table 5-6 shows the main seven functions used by the MPI model to parallel the SSN algorithm:

Function	The function's job
MPI_Init();	Initializes the MPI program
MPI_Comm_size(MPI_COMM_WORLD, &nproc);	Uses objects called communicators and groups to define which collection of processes may communicate with each other. It determines the number of processes
MPI_Comm_rank(MPI_COMM_WORLD, &rank);	Within a communicator, every process has its own unique, id (label) assigned by the system when the process initializes.
MPI_Scatter (buffer, count, MPI_CHAR, recvBuf, scount, MPI_CHAR, 0, MPI_COMM_WORLD);	Involves data decomposition and division among the master and slave processes e.g. spreading an array to all processors.
MPI_SEND(buffer, count, MPI_CHAR, recvBuf, rank, MPI_COMM_WORLD);	Sends a message to a different process within a communicator
MPI_RECV(buffer, count, MPI_CHAR, sendBuf, rank, MPI_COMM_WORLD, status);	Receives a message from different process within a communicator
MPI_Finalize ();	Terminates the MPI execution environment

Table 5- 6: The main seven functions of MPI which used to parallelize the SSN algorithm

5.4 Summary

This chapter showed the implementation of the chemical toolkit and the parallel algorithm. Four stages were used to implement the toolkit. The first stage included downloading and mining structures from the NMRShiftDB. The second stage included the local database building process. The third stage included the implementation of connecting the toolkit to the local database and searching structures using JME Editor, SMILES and the SSN algorithm. The fourth stage included the similarity measurement between structures using the proportion of matching characters. The parallel algorithm was implemented using the OpenMP model (shared memory model) and the MPI model (distributed memory model).

CHAPTER 6: RESULTS AND DISCUSSION

To evaluate our new algorithms, and to compare them to standard algorithms, we implemented our algorithms and some of the standard algorithms and tested them on DNA and protein sample files by taking an average of 10 executions for each pattern length.

This is a well-established approach, heavily used by:

- (Sheik et al.) testing the SSABS algorithm in 2004;
- (Thathoo et al.) testing the TVSBS algorithm in 2006;
- (Huang et al.) testing the ZTBMH algorithm in 2008;
- (Huang et al.) testing the BRFS algorithm in 2008;
- (Almazroi and Rashid) testing the BRSS algorithm in 2011;
- (Almazroi) testing the ASSBR algorithm in 2011.

These were all tests of simple sequential implementations. We also implemented and tested the best algorithm using the OpenMP model on a single Sun Fire unit and using the MPI model on two Sun Fire units at the University of Science Malaysia (USM). Finally, we used our best algorithm the SSN algorithm to develop the chemical structure searching toolkit.

This Chapter consists of the following sections: section 6.1 shows the results of testing sequential algorithms on short DNA pattern sequences; section 6.2 shows the results of testing sequential algorithms on long DNA pattern sequences; section 6.3 shows the results of testing sequential algorithms on short protein pattern sequences; section 6.4 shows the results of testing sequential algorithms on long protein pattern sequences, section 6.5 shows the results of testing parallel algorithms on short and long DNA and protein pattern sequences on different numbers

of processors; section 6.6 shows the results of testing the chemical toolkit using the SSN algorithm; and finally section 6.7 discusses our research results.

6.1 Testing Algorithms Using a Short DNA Pattern

A sample file of FASTA format DNA sequences was downloaded from the U.S. National Centre of Biotechnology Information ([NCBI, 2012](#)). The total number of characters of all sequences in the downloaded sample file is 661080 characters. Our algorithms, and some standard algorithms were applied to this file and three types of tests implemented. The first test determines the number of comparisons, the second one calculates the number of attempts and the final one finds the elapsed search time. Sub-sections 6.1.1, 6.1.2 and 6.1.3 show the results and analysis of these tests on the short DNA pattern.

6.1.1 The Number of Comparisons Using a Short DNA Pattern

The total numbers of comparisons when searching for a DNA pattern using a short pattern length is shown in Table 6-1:

Pattern Length	SSN	RSMA	OE	BRQS	BRBMH	BRFS	TVSBS	QS	BMH
4	195816	274575	263175	240175	275439	276175	278244	305023	341347
7	86538	195102	180369	171790	194777	197790	198852	204529	269826
10	80367	139168	140069	134835	138618	137131	147481	193313	247906
13	72641	104696	101580	103342	106636	105320	115409	179328	237386
16	62120	101014	97257	94281	102481	99815	111263	140112	204237
19	51914	79199	63472	76972	79405	83018	90771	114326	173499
22	48795	67201	65586	66230	68346	73538	80807	109708	146909
25	45444	63881	56913	59509	64315	70794	78363	101637	127133
28	45418	50898	50914	51164	52701	55291	60140	83130	100152
31	41269	50158	48581	50142	50569	52377	59177	76001	93895

Table 6- 1: The number of comparisons for a short DNA pattern

From Table 6.1, we can see that our hybrid algorithms show a significant improvement over the original single algorithms in the number of comparisons. For example, the BMH algorithm processes the comparison for a DNA pattern with length 4 in the sample DNA file 341347 times, while the hybrid BRBMH algorithm processes the same pattern 275439 times. Similarity, the QS algorithm processes the same sample 305023 times while the hybrid BRQS algorithm processes it 240175 times.

Our algorithms are also more efficient than the TVSBS hybrid algorithm, for example, the odd and even algorithm processes a pattern with length 19 in the DNA sample file with 63472 comparisons while the TVSBS algorithm requires 90771.

Our algorithm also uses fewer comparisons than the BRFS algorithm. For example, searching a pattern with length 28 using the BRFS algorithm requires 55291 comparisons while the RSMA algorithm only requires 50898 times.

The best algorithm in this test is the SSN algorithm. It shows a significant improvement over all algorithms, for example it searches a pattern with length 31 using 41269 comparisons while the RSMA, OE, BRQS, BRBMH, BRFS, TVSBS, QS and BMH algorithms require 50158, 48581, 50142, 52377, 50569, 59177, 76001 and 93895 comparisons respectively.

6.1.2 The Number of Attempts Using a Short DNA Pattern

Table 6-2 shows the numbers of attempts in the same DNA sample file using a short pattern length:

Pattern Length	SSN	RSMA	OE	BRQS	BRBMH	BRFS	TVSBS	QS	BMH
4	94623	161219	161219	161219	161219	161219	161003	231464	243417
7	66368	115993	115993	115993	115993	115993	115795	171707	202675
10	49942	88242	88242	88242	88242	89871	90077	159765	171911
13	39396	67406	67406	67406	67406	71476	71351	139545	155845
16	32257	60969	60969	60969	60969	66861	67164	109967	138584
19	27488	49718	49718	49718	49718	55752	55472	95645	121018
22	23833	42538	42538	42538	42538	49199	49318	91843	107653
25	20958	41291	41291	41291	41291	49136	49134	86276	99130
28	18529	33212	33212	33212	33212	37264	37270	65285	92937
31	16612	29249	29249	29249	29249	35100	35268	59555	84363

Table 6- 2: The number of attempts for a short DNA pattern

Numbers of attempts for all algorithms are counted when there is a whole match or a mismatch encountered which shifts the pattern and starts a new attempt.

The pre-processing phase makes a significant contribution to the total number of attempts. Our algorithms: RSMA, OE, BRQS and BRBMH algorithms use the enhanced Berry-Ravindran pre-processing phase which means they have the same number of attempts for all algorithms on the same pattern length.

The different number of attempts in our algorithms compared to the BRFS, TVSBS, QS and BMH algorithm is due to the preprocessing phase used in our algorithms. The SSN algorithm is the best one in this test as well. As an example our SSN algorithm requires 32257 attempts when searching a pattern with length 16 while the RSMA, OE, BRQS and BRBMH algorithms require 60969 attempts. The BRFS, TVSBS, QS and BMH algorithms require 60969, 66861, 67164, 109967, 90685 and 138584 attempts respectively using the same pattern length.

6.1.3 The Average Elapsed Search Time Using a Short DNA Pattern

Figure 6-1 and Figure 6-2 show the elapsed search time when searching for the search pattern in the same DNA sample file using a short pattern length:

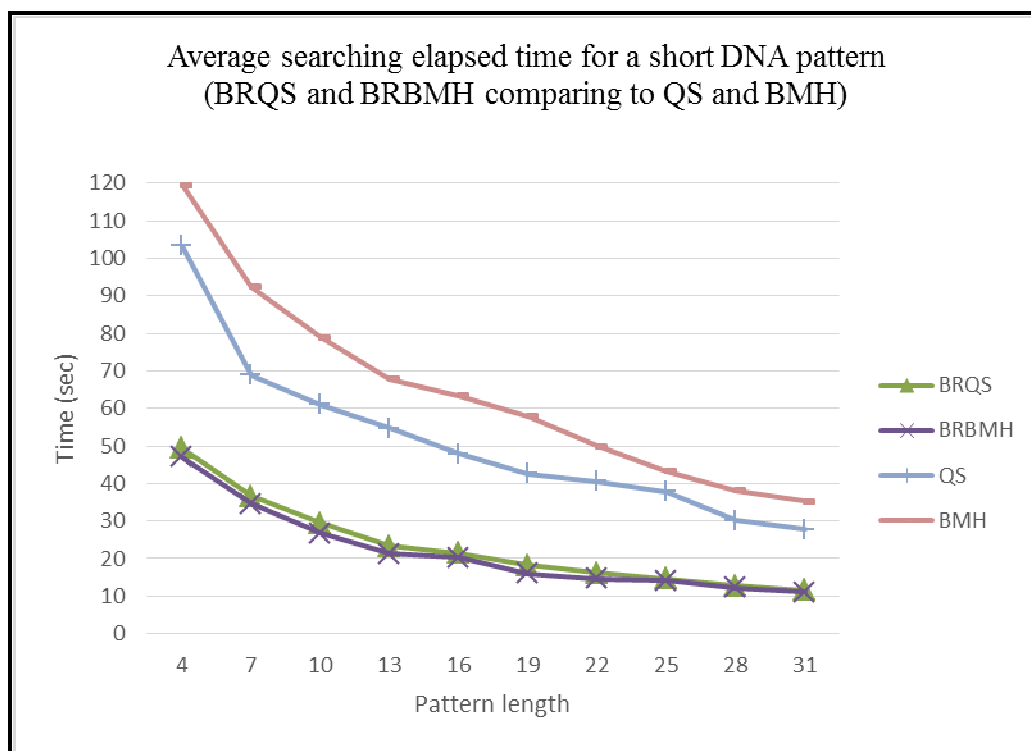


Figure 6- 1: BRQS and BRBMH searching time using a short DNA pattern

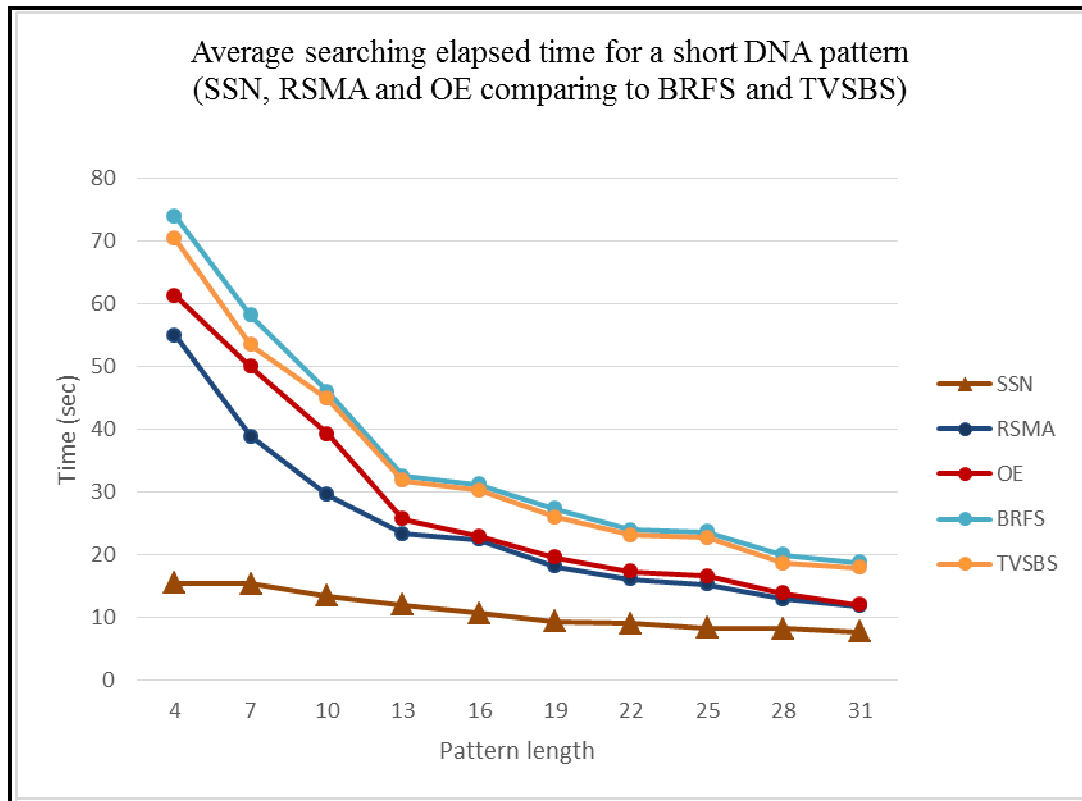


Figure 6- 2: SSN, RSMA and OE searching time using a short DNA pattern

Our hybrid algorithms register a lower time. As an example in Figure 6-1, the BMH algorithm searches the chosen DNA pattern with length 7 in the DNA sample file in 92.283 seconds while the hybrid BRBMH searches the same pattern in 36.617 seconds. The QS algorithm finds the same sample in 68.907 seconds while the hybrid BRQS algorithm finds it in 34.593 seconds. Our algorithms such as the SSN, RSMA and OE algorithms find the pattern in less time from the TVSBS and BRFS algorithms. As an example in Figure 6-2, the SSN algorithm searches pattern with length 4 in the same sample data file in 7.731 seconds while the RSMA, OE, BRFS and TVSBS algorithms search the same pattern length in 11.843, 12.018, 18.741 and 17.928 seconds respectively.

6.2 Testing Algorithms Using a Long DNA Pattern

The same sample file of DNA sequences in FASTA format is used to test the algorithms on a long DNA pattern length. Sub-sections 6.2.1, 6.2.2 and 6.2.3 show the results and analysis of tests on a long DNA pattern.

6.2.1 The Number of Comparisons Using a Long DNA Pattern

The following Table 6-3 shows the total number of comparisons when searching for a DNA pattern using a long pattern length:

Pattern Length	SSN	RSMA	OE	BRQS	BRBMH	BRFS	TVSBS	QS	BMH
32	40545	48953	46682	48712	48569	51982	58842	70688	89585
48	28723	30057	29479	28290	30345	46930	57556	63650	76379
64	24096	27982	28936	25618	28964	43329	55898	45989	65854
96	17892	19307	19150	17483	19906	42078	51578	32621	49769
128	13437	14450	14410	12803	14714	32060	38821	24002	38208
192	7024	7500	7568	6998	7607	21774	24114	18839	29614
256	5254	5688	5703	5778	5794	16304	17553	15035	21147
384	3672	3873	3909	3906	3942	11983	13321	10516	18335
512	3412	3670	3468	3305	3588	12103	14957	6774	11063
768	2487	2547	2549	2580	2620	6761	7445	5885	9168
1024	2224	2268	2268	2347	2313	5460	6384	5234	8081

Table 6- 3: The number of comparisons for a long DNA pattern

Our hybrid algorithms show a significant important on the long DNA pattern as well. The OE algorithm searches the DNA pattern with length 32 in the sample file with 46682 comparisons while the BMH requires 89585 comparisons. The TVSBS algorithm searches the

same pattern length using 48305 comparisons. A longer pattern such as the length 512 requires 3305 comparisons by the BRQS while the QS algorithm needs 6774 comparisons.

The SSN algorithm searches a pattern with length 768 using 2487 comparisons while the RSMA, OE, BRQS, BRBMH, BRFS, TVSBS, QS and BMH algorithms require 2547, 2549, 2580, 2620, 6761, 7445, 5885 and 9168 comparisons respectively.

6.2.2 The Number of Attempts Using a Long DNA Pattern

Table 6-4 shows the numbers of attempts in the same sample file using a long pattern length:

Pattern Length	SSN	RSMA	OE	BRQS	BRBMH	BRFS	TVSBS	QS	BMH
32	16191	28939	28939	28939	28939	34765	34726	48044	82118
48	10926	19509	19509	19509	19509	32687	33428	46192	73852
64	7906	17298	17298	17298	17298	31511	31675	34956	58458
96	5177	11718	11718	11718	11718	30239	30028	32259	49882
128	3829	8523	8523	8525	8523	22683	22057	28063	37540
192	2784	4858	4858	4858	4858	15206	15548	24839	26458
256	2086	3541	3541	3541	3541	11002	10948	16810	18917
384	1430	2287	2287	2287	2287	8006	8020	9076	10625
512	1136	1846	1846	1846	1846	8316	8314	8641	9130
768	779	1153	1153	1153	1153	4069	4154	6095	7268
1024	599	837	837	837	837	2977	3297	4304	5361
Table 6- 4: The number of attempts for a long DNA pattern									

Again RSMA, OE, BRQS, BRBMH and BRFS shift the pattern using the same pre-processing phase, and therefore have the same number of attempts for each pattern length. They also use fewer attempts than the BRQS, TVSBS, QS and BMH algorithms. For example, the RSMA algorithm searches a pattern with length 48 in 19509 attempts while the BMH algorithm searches the same pattern in 73852 attempts. The SSN algorithm searches the same pattern in 10926 attempts.

6.2.3 The Average Searching Elapsed Time Using a Long DNA Pattern

Figures 6-3, 6-4 and 6-5 show the elapsed search used to search the enquired pattern in the same DNA sample file using a long pattern length:

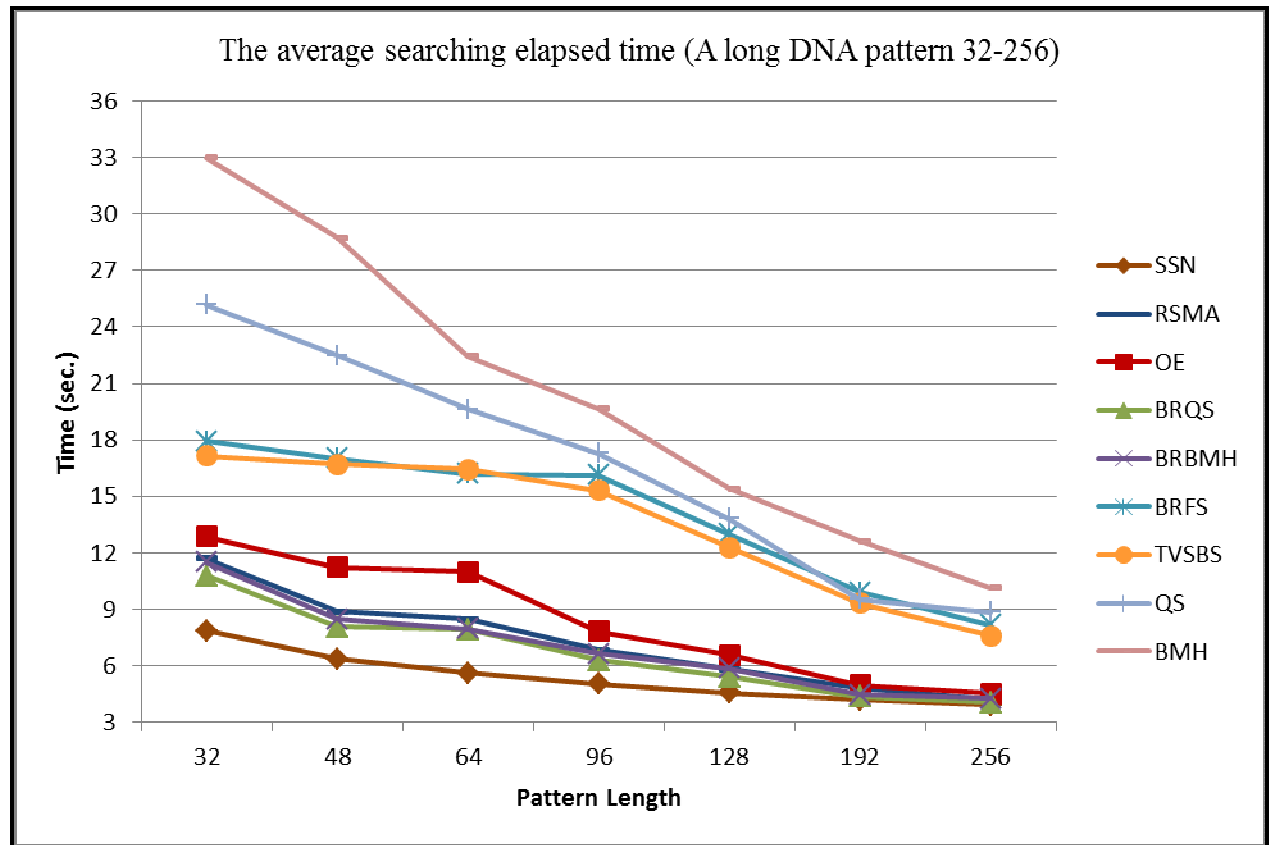


Figure 6- 3: The average searching elapsed time for a long DNA (32-256)

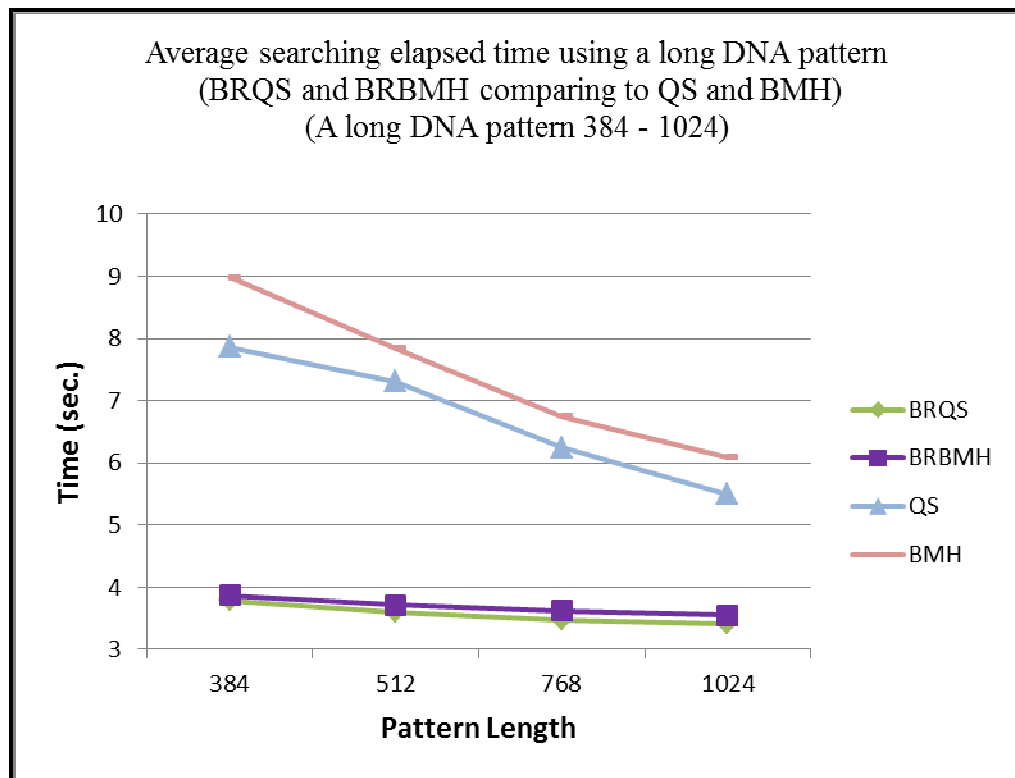


Figure 6- 4: The BRQS, BRBMH, QS and BMH searching time for a long DNA (384-1024)

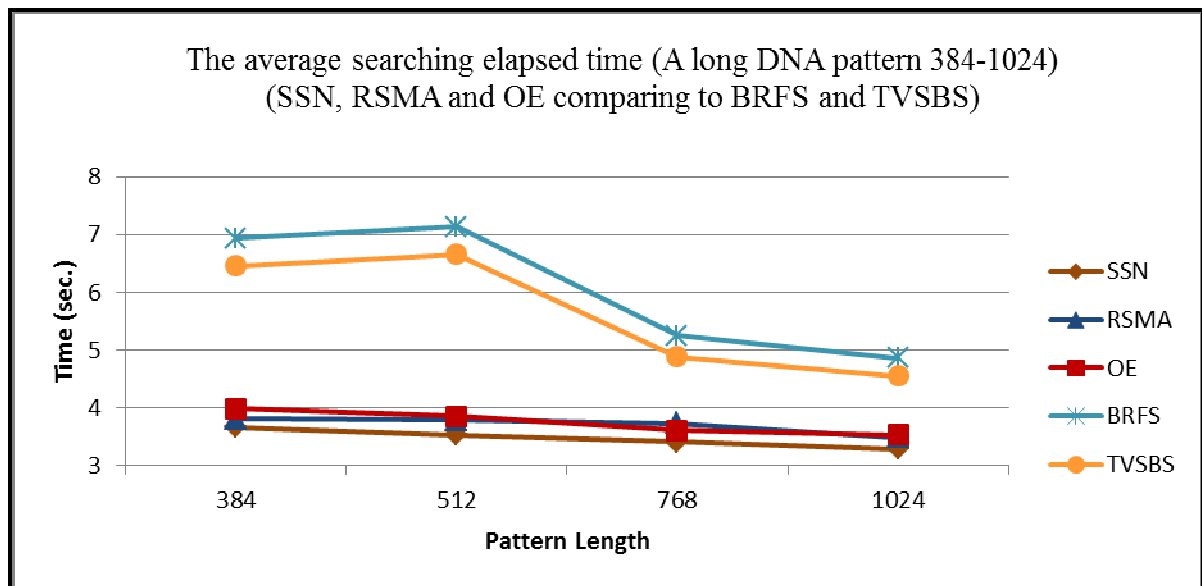


Figure 6- 5: The SSN, RSMA and OE searching time for a long DNA (384-1024)

The BRQS and the BRBMH search a pattern with length 32 in 10.7666 and 11.495 seconds respectively while the QS and the BMH search the same pattern in 25.164 and 33.004 seconds respectively.

The SSN algorithm searches the same pattern length in 7.864 seconds while the OE, BRFS and TVSBS algorithms search the same pattern in 12.863, 17.923 and 17.15 seconds respectively.

6.3 Testing Algorithms Using a Short Protein Pattern

A sample file of FASTA format protein sequences was downloaded from the SwissProt Database ([UniProt Consortium, 2013](#)). The total number of characters of all sequences in the downloaded sample file is 1006778 characters. The same algorithms used to search DNA sequences in section 6.1 and 6.2 were used again to search for amino acids sequences in the downloaded sample file. Sub-sections 6.3.1, 6.3.2 and 6.3.3 show the results and analysis of tests on a short protein pattern.

6.3.1 The Number of Comparisons Using a Short Protein Pattern

The following Table 6-5 shows the numbers of comparisons between the chosen patterns with whole sequences in the protein sample file using a short pattern length:

Pattern Length	SSN	RSMA	OE	BRQS	BRBMH	BRFS	TVSBS	QS	BMH
4	143877	191073	191078	185328	191053	185328	191071	270395	237848
7	100810	122636	121243	122530	122608	122530	122599	160645	154818
10	77593	99493	99621	94415	99602	94474	99742	116987	117756
13	63555	92390	86677	80828	92101	80862	92207	107765	111769
16	54165	78730	78827	67664	77825	67779	77893	91054	97101
19	47083	59886	59733	56609	59432	56691	59534	83225	87702
22	41724	57629	58245	50904	57440	51294	58464	76229	75425
25	37328	46270	42193	43988	46049	44315	46137	71640	72248
28	33673	43890	44054	40330	43861	40658	44049	59379	65277
31	30811	37931	36807	36030	37819	36294	38319	53115	58126

Table 6- 5: The number of comparisons for a short protein pattern

Our hybrid algorithms achieve better results than the original algorithms as shown in Table 6-5. The BMH algorithm requires 46795 more comparisons than the BRBMH algorithm when searching for a pattern of length 4. The QS when searching for a pattern of length 13 requires 26937 more comparisons than the BRQS algorithm. When searching for a pattern of length 22 the TVSBS requires 835 more comparisons than the RSMA algorithm. The BRFS algorithm requires 6985 more comparisons than the RSMA algorithm when searching a pattern with length 28.

6.3.2 The Number of Attempts Using a Short Protein Pattern

Table 6-6 shows the numbers of attempts where attempts are counted when the pattern is shifted using a short protein pattern length:

Pattern Length	SSN	RSMA	OE	BRQS	BRBMH	BRFS	TVSBS	QS	BMH
4	143834	175258	175258	175258	175258	175258	175256	258512	213554
7	100672	115674	115674	115674	115674	115674	115674	156107	139026
10	77432	89193	89193	89193	89193	89277	89281	110505	105585
13	62925	76481	76481	76481	76481	76533	76511	97997	100402
16	52941	64029	64029	64029	64029	64114	64095	82208	87126
19	45662	53390	53390	53390	53390	53460	53464	78922	78708
22	40168	48164	48164	48164	48164	48624	48619	73918	67674
25	35805	41558	41558	41558	41558	41791	41791	68052	63437
28	32316	38098	38098	38098	38098	38429	38429	55112	58642
31	29425	34005	34005	34005	34005	34226	34219	50118	52246

Table 6- 6: The number of attempts for a short protein pattern

Table 6-6 shows that the pre-processing phase of RSMA, OE, BRQS and BRBMH algorithms all have the same number of attempts for each length of short protein pattern lengths, but they still provide better results than hybrid algorithms in most cases. The SSN algorithm shows better results than all algorithms. As an example, there is a difference of 331 attempts between RSMA and TVSBS with pattern length 28 and a difference of 6113 attempts between the SSN and the BRFS algorithms. Additionally, they perform better than original algorithms such as QS and BMH. For example, the OE algorithm searches a pattern with length 7 by 38296 attempts less than the BMH algorithm.

6.3.3 The Average Elapsed Search Time Using a Short Protein Pattern

Figures 6-6 shows the average elapsed search time using a short pattern length on protein sample file:

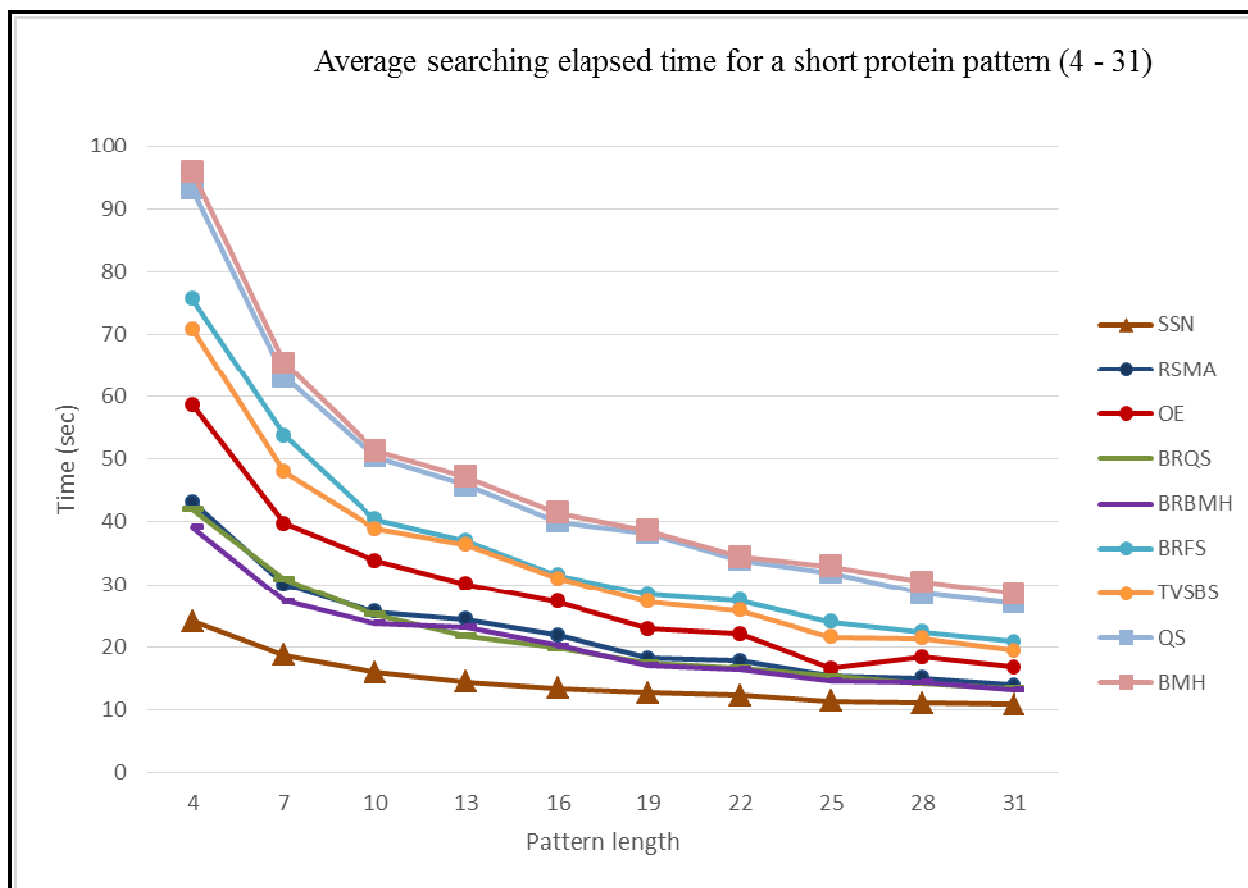


Figure 6- 6: The average searching elapsed time for a short protein pattern (4 - 31)

Figure 6-6 shows the shorter elapsed search time for our SSN algorithm compared to other well-known ones. The SSN is faster than the RSMA, OE, BRQS, BRBMH, BRFS, TVSBS, QS and BMH algorithms on patterns with length 7 by 37.7%, 52.9%, 39.6%, 31.5%, 65.2%, 61.1%, 70.4% and 71.4% respectively, and with length 25 by 26.6%, 31.4%, 26.3%, 22.7%, 52.8%, 47.5%, 64.5% and 65.6% respectively.

6.4 Testing Algorithms Using a Long Protein Pattern

The same sample file of protein sequences in FASTA format is used to test the algorithms on a long protein pattern length. Sub-sections 6.4.1, 6.4.2 and 6.4.3 show the results and analysis of tests on a long protein pattern.

6.4.1 The Number of Comparisons Using a Long Protein Pattern

Table 6-7 shows the numbers of comparisons between the chosen patterns within the protein sample file using a long pattern length:

Pattern Length	SSN	RSMA	OE	BRQS	BRBMH	BRFS	TVSBS	QS	BMH
32	29907	36967	35878	34952	36867	35215	37137	50704	55238
48	20808	27003	26822	24803	26798	25283	27459	44850	51557
64	15940	18733	18702	18104	18677	18318	19047	32501	46737
96	10843	15076	15293	13151	15111	13342	15408	31982	42174
128	8419	10670	10773	9857	10695	10422	11385	30727	40867
192	5999	7442	7493	7296	7453	7683	7847	29458	37073
256	4754	5873	5893	5756	5876	6549	6703	29298	34201
384	3528	4601	4592	4381	4590	5615	6021	24463	32571
512	3048	3931	3928	3568	3955	4728	5364	21846	30772
768	2535	2720	2717	2708	2718	4305	4377	19044	28162
1024	2294	2400	2400	2403	2399	4025	4034	17027	26190

Table 6- 7: The number of comparisons for a long protein pattern

The BMH algorithm requires 24554 more comparisons than the RSMA algorithm to search a pattern with length 48 in the sample protein file. When searching for a protein pattern with length 512 in the sample file, the BRQS algorithm requires 3931 comparisons while TVSBS requires 5364 comparisons. In longer patterns the QS algorithm searches a pattern with a length 768 using 16327 more comparisons than the Odd and Even algorithm.

The BRFS algorithm searches a pattern with a length 1024 using 1731 more comparisons than the SSN algorithm. Our hybrid algorithms show a significant improvement on the long protein pattern as well.

6.4.2 The Number of Attempts Using a Long Protein Pattern

Table 6-8 shows the numbers of attempts in the sample file using a long pattern length:

Pattern Length	SSN	RSMA	OE	BRQS	BRBMH	BRFS	TVSBS	QS	BMH
32	28586	33024	33024	33024	33024	33247	33247	47844	49751
48	19484	23371	23371	23371	23371	23829	23829	41635	46257
64	14770	17073	17073	17073	17073	17283	17283	31078	43008
96	9944	12330	12330	12330	12330	12570	12570	30665	41111
128	7480	9238	9238	9238	9238	9725	9728	29293	39100
192	4974	6726	6726	6726	6726	7105	7105	28211	37267
256	3699	5214	5214	5214	5214	5966	5966	27984	35966
384	2430	3741	3741	3741	3741	4981	5003	19905	33413
512	1778	2866	2866	2866	2866	4015	4012	17788	31958
768	1167	1815	1815	1815	1815	3334	3334	16536	29962
1024	850	1313	1313	1313	1313	2859	2859	15474	26800
Table 6- 8: Number of Attempts for Long Protein Pattern									

The BMH algorithm searches the pattern of length 48 in the sample file in 46257 attempts while the BMBMH algorithm needs 23371. The BRQS algorithm searches a pattern of length 192 in 30541 fewer attempts than the QS algorithm. The SSN algorithm searches a pattern of length 1024 in 1835 fewer attempts than the BRFS algorithm. Our algorithms require fewer attempts than the BRFS, TVSBS, BRQS and BMH algorithms due to their pre-processing phase.

6.4.3 The Average Elapsed Search Time Using a Long Protein Pattern

Figure 6-7, Figure 6-8 and Figure 6-9 show the elapsed search time in the protein sample file using a long pattern length:

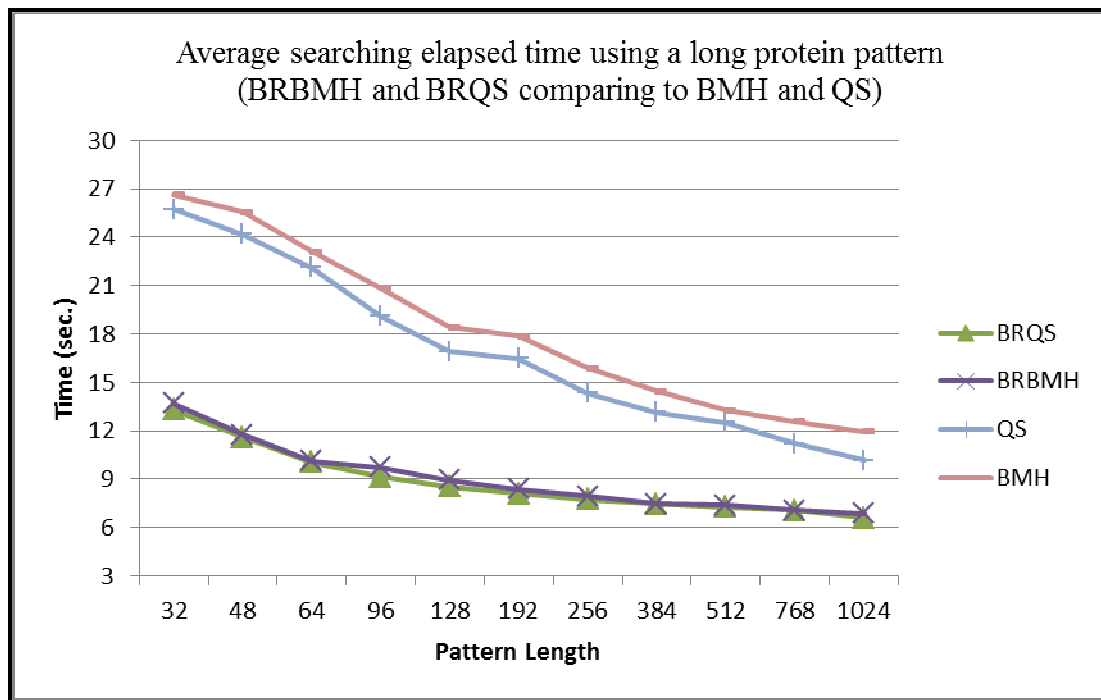


Figure 6- 7: BRBMH and BRQS searching time using a long protein pattern

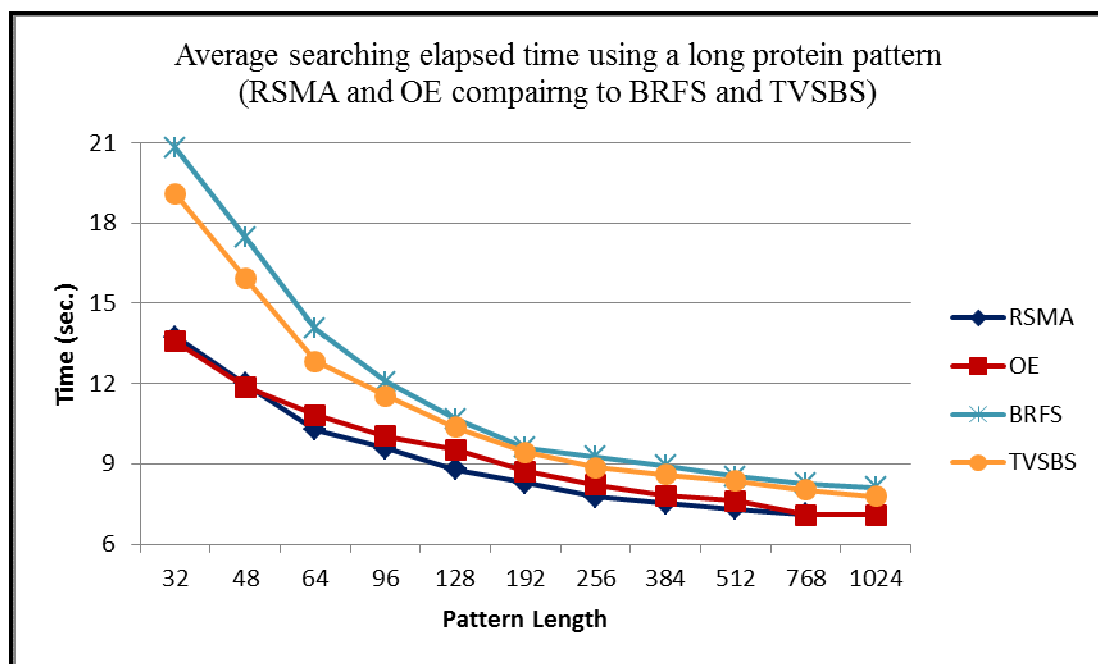


Figure 6- 8: RSMA and OE searching time using a long protein pattern

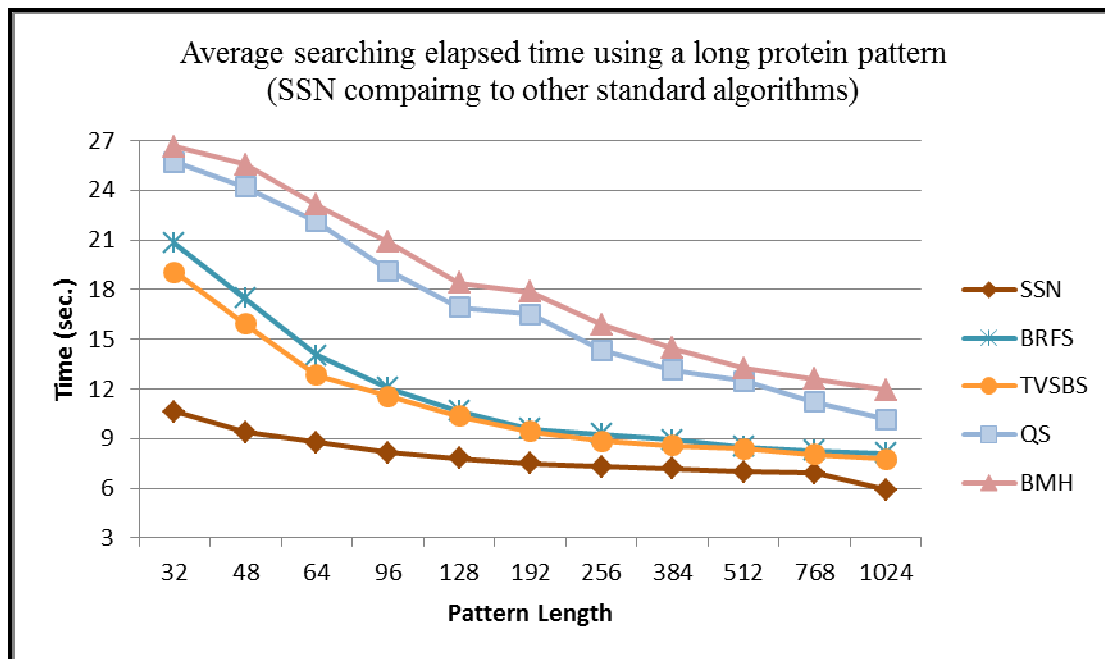


Figure 6- 9: The SSN searching time using a long protein pattern

The SSN algorithm shows better average elapsed search time than the RSMA, OE, BRQS, BRBMH, BRFs, TVSBS, QS and BMH algorithms. The difference is obvious in short patterns, although the difference in searching time is very small between searching algorithms in long patterns. As an example, the BMH searches a pattern with length 32 in 26.641 seconds while the hybrid BRBMH is searching the same pattern in 13.685 seconds. Another example, the QS algorithm searches the same sample in 25.715 seconds while the hybrid BRQS algorithm searches in 13.304 seconds. The SSN algorithm is faster than the RSMA, OE, BRQS, BRBMH, BRFs, TVSBS, QS and BMH algorithms on patterns with length 48 by 21.7%, 20.9%, 19.1%, 20.3%, 46.2%, 41.1%, 61.2% and 63.25% respectively, and with length 1024 by 16.5%, 16.5%, 10.3%, 13.9%, 26.9%, 23.9%, 41.8% and 50.4% respectively. Our algorithms such as the RSMA algorithm search the pattern in less time from the TVSBS and the BRFs algorithms.

6.5 Testing Parallel Algorithms

Four parallel experiments were implemented and tested. Each one represents the average of ten execution times for parallel pattern searching. The first test in sub-section 6.5.1 shows the OpenMP model searching DNA sequences file. The second test in sub-section 6.5.2 shows the OpenMP model searching protein sequences file. The third test in sub-section 6.5.3 shows the MPI model searching DNA sequences file and finally the fourth test in sub-section 6.5.4 shows the MPI model searching protein sequences file.

6.5.1 Testing the OpenMP Model on DNA Sequences File

Figure 6-15 shows the average elapsed search time in the DNA sample file using the OpenMP parallel model:

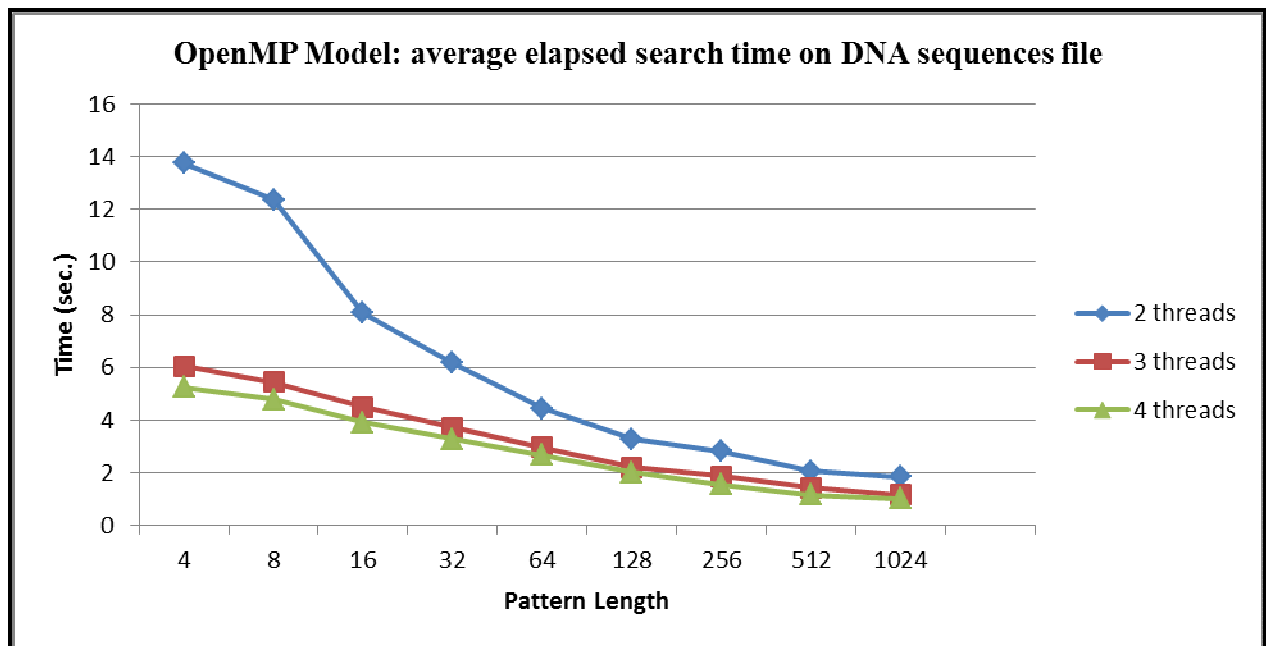


Figure 6- 10: OpenMP model: the average elapsed search time on DNA sequences file

6.5.2 Testing the OpenMP Model on Protein Sequences File

Figure 6-16 shows the average elapsed search time in the protein sample file using the OpenMP parallel model:

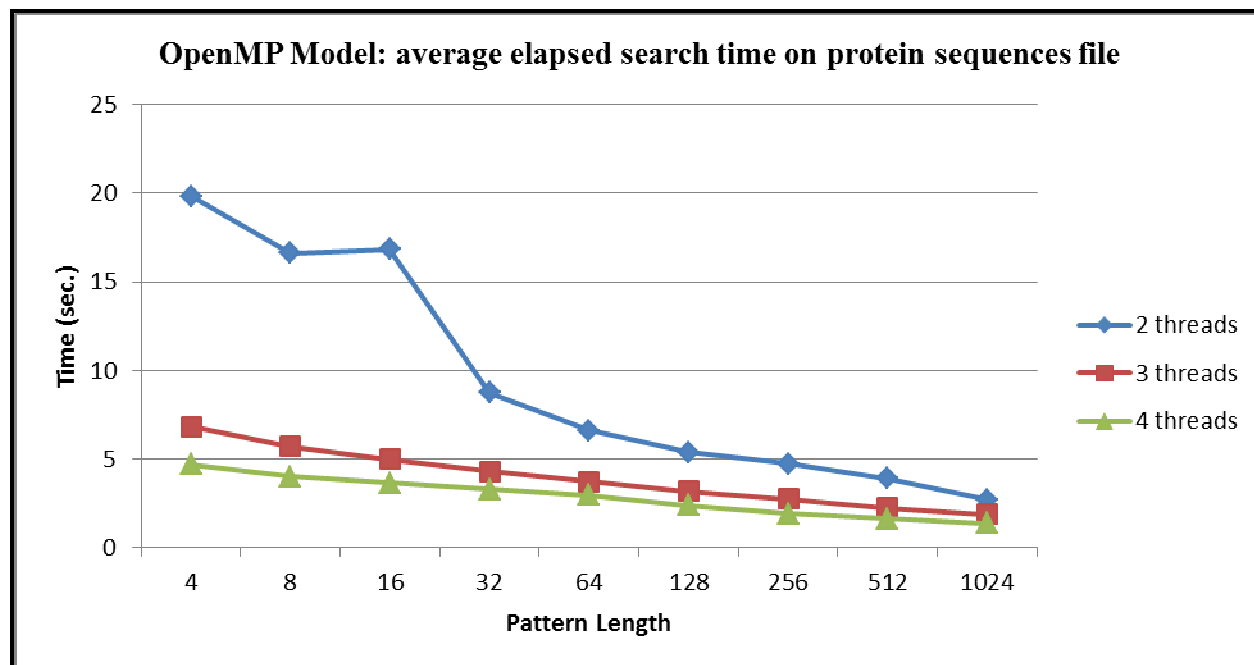


Figure 6- 11: OpenMP model: the average elapsed search time on protein sequences file

6.5.3 Testing the MPI Model on DNA Sequences File

Figure 6-17 shows the average elapsed search time in the DNA sample file using the MPI parallel model:

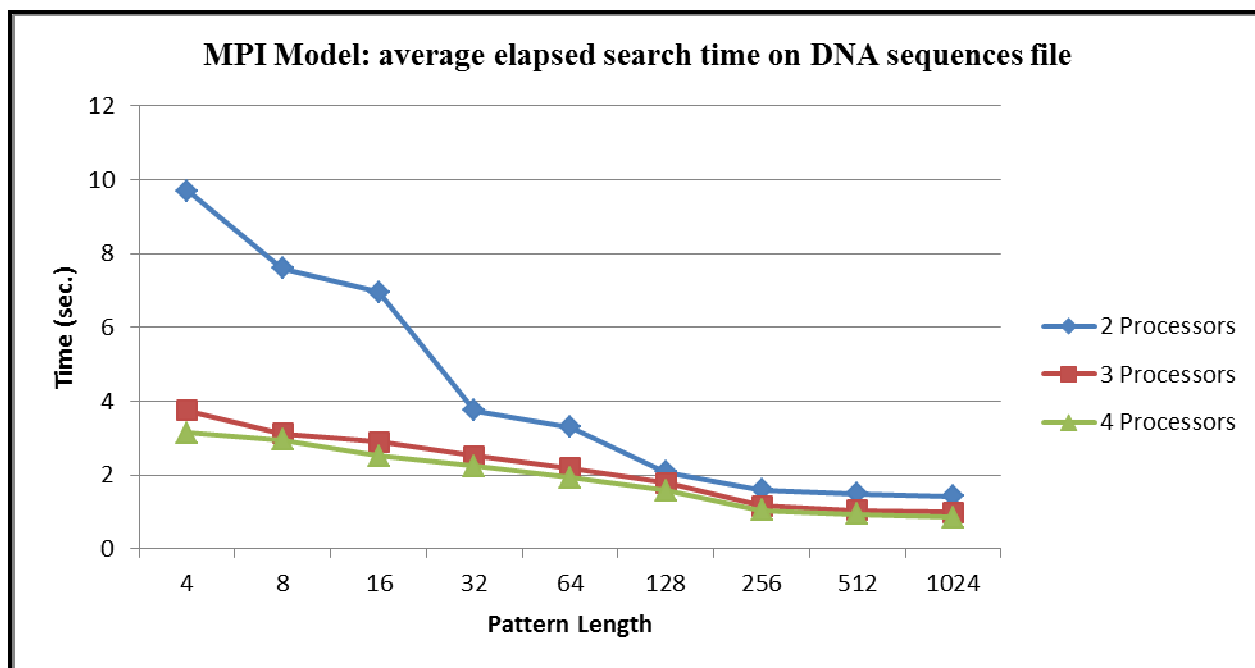


Figure 6- 12: MPI model: the average elapsed search time on DNA sequences file

6.5.4 Testing the MPI Model on Protein Sequences File

Figure 6-18 shows the average elapsed search time in the protein sample file using the MPI parallel model:

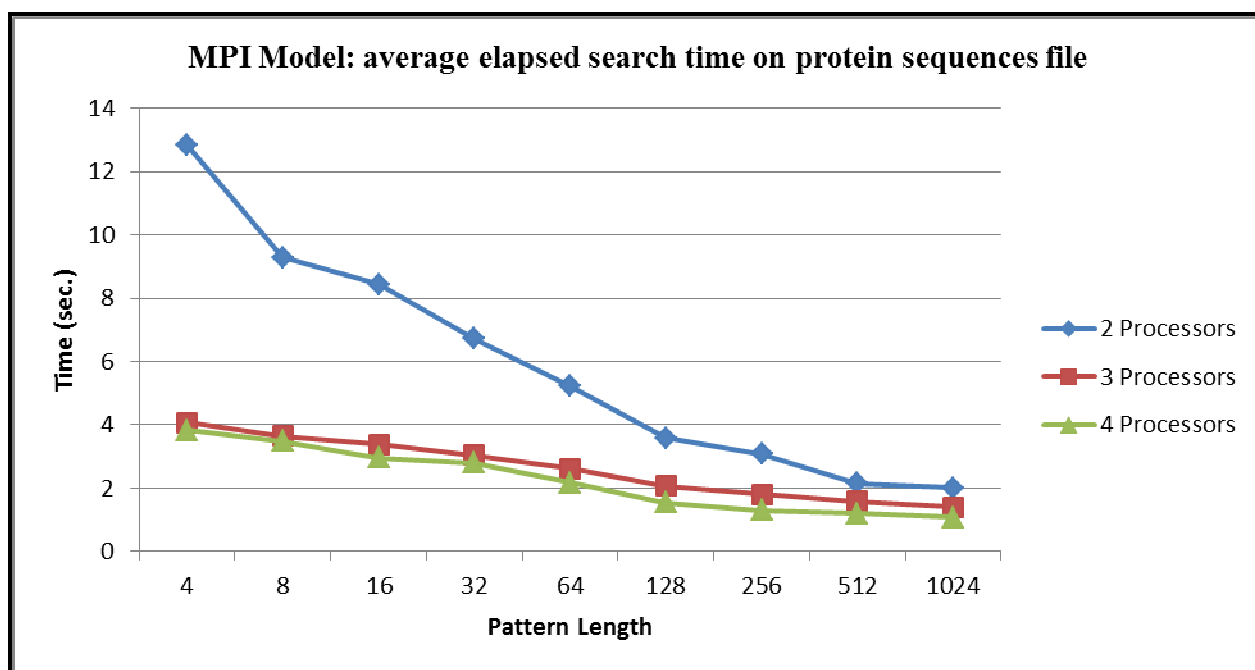


Figure 6- 13: MPI model: the average elapsed search time on protein sequences file

6.6 Testing the Chemical Searching Toolkit Using the SSN Algorithm

The chemical searching toolkit as described in section 5.4 allows user to either input the chemical structure in SMILES format or draw the structure using the JME editor. Our SMILES checking tool checks if the entered SMILES is correct or not. The searching button uses the SSN string matching algorithm to search structures in the local database and list all similar structures with the similarity percentage using the proportion of matching characters. Finally if one of the found structures is chosen it shows the structure details. Figures 6-19, 6-20, 6-21 and 6-22 show an example of searching a chemical structure pattern using the chemical toolkit.

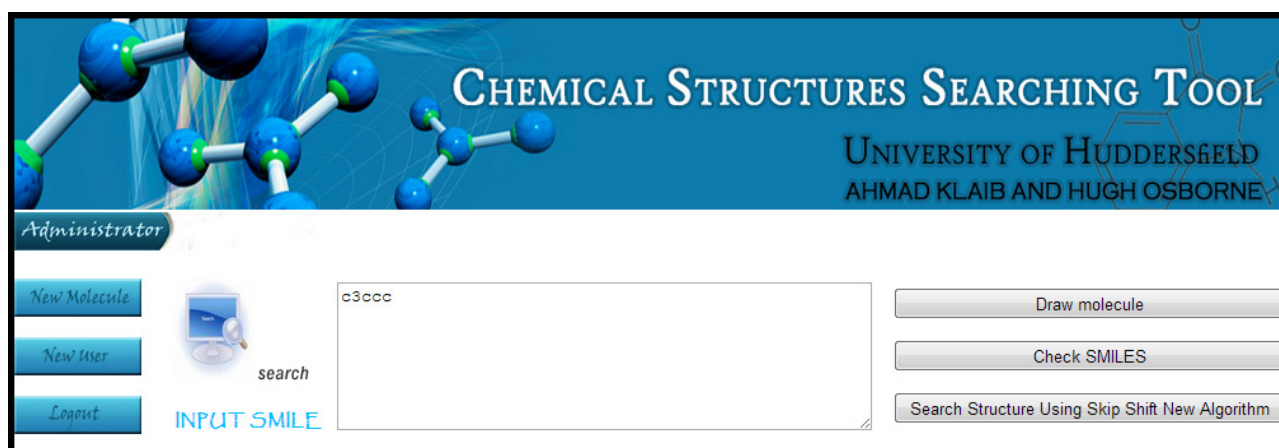


Figure 6- 14: Input a pattern chemical structure

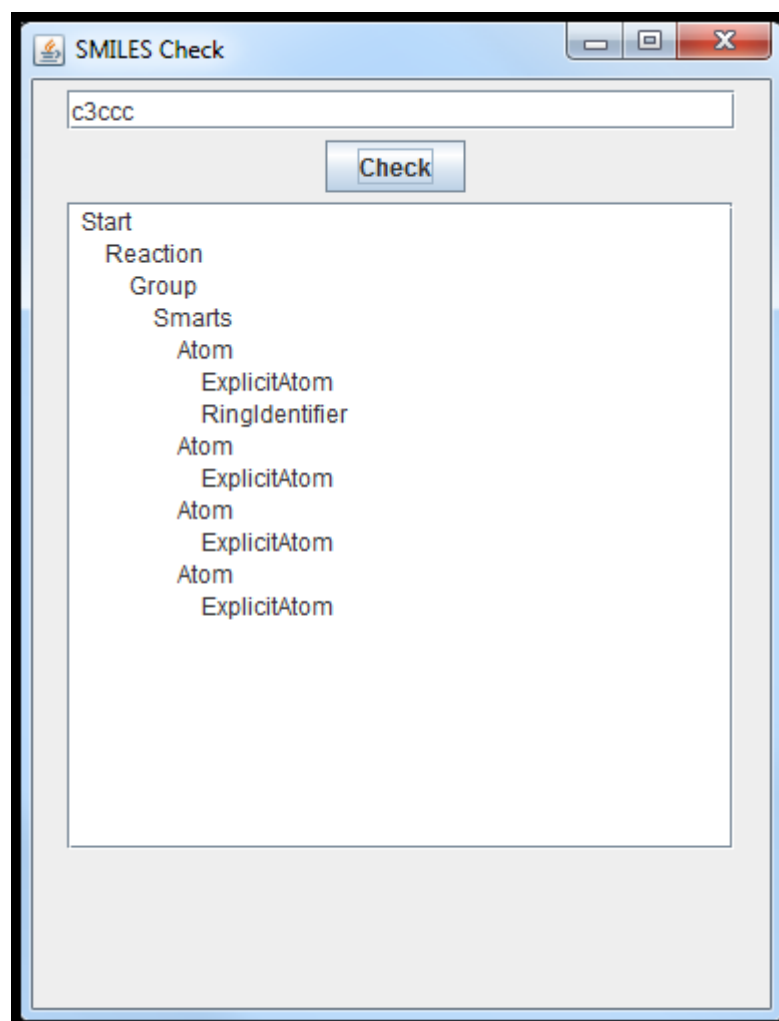
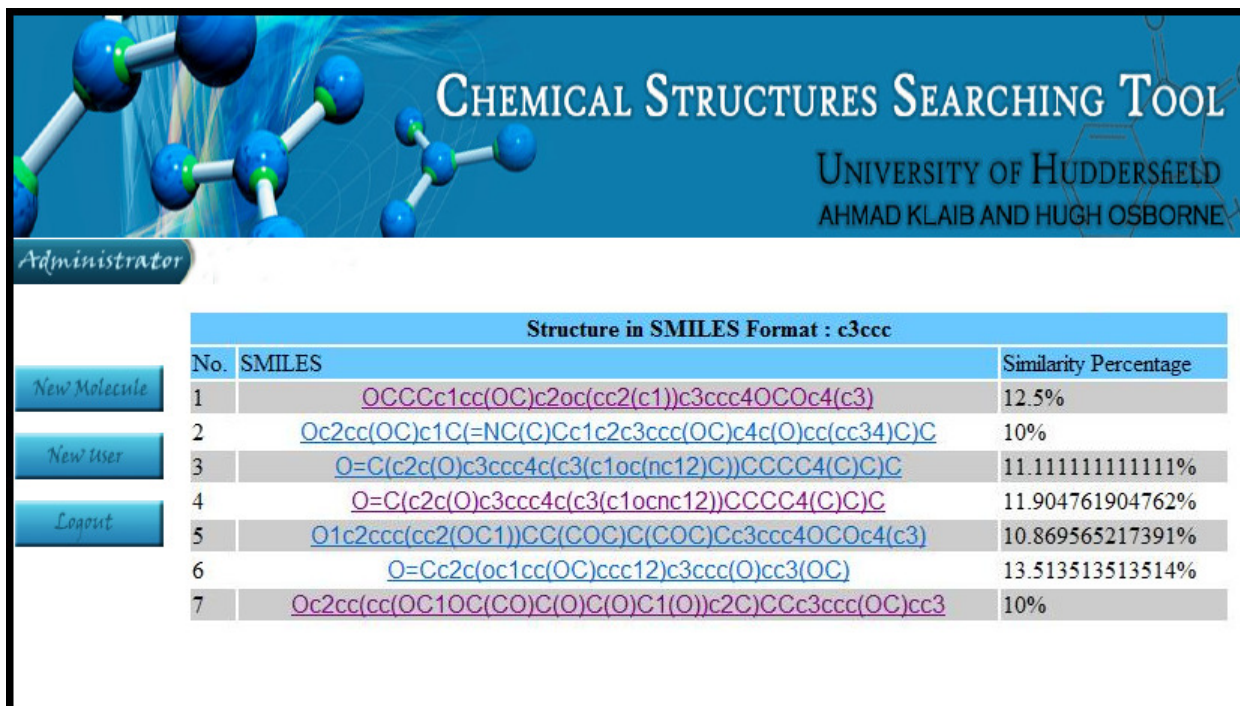


Figure 6- 15: Verify the SMILES input structure



CHEMICAL STRUCTURES SEARCHING TOOL
UNIVERSITY OF HUDDERSFIELD
AHMAD KLAIB AND HUGH OSBORNE

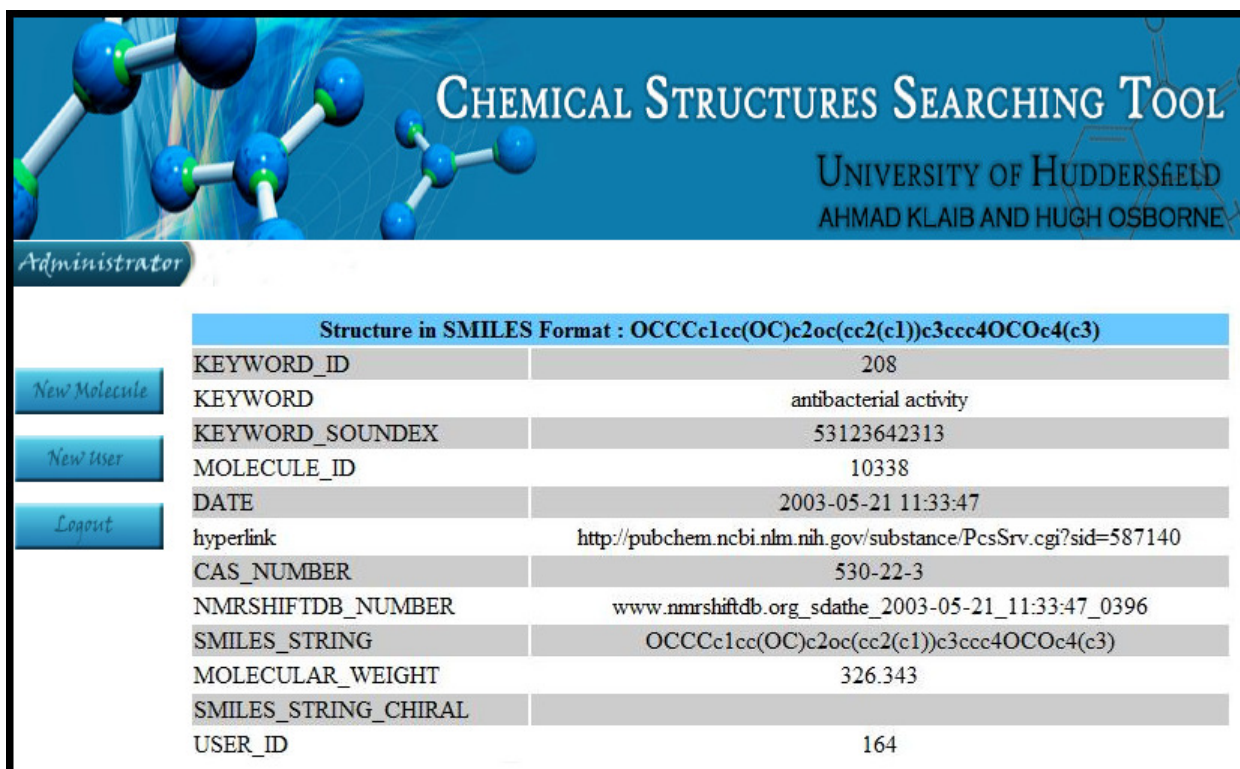
Administrator

Structure in SMILES Format : c3ccc

No.	SMILES	Similarity Percentage
1	<chem>OCCc1cc(OC)c2oc(cc2(c1))c3ccc4OCOc4(c3)</chem>	12.5%
2	<chem>Oc2cc(OC)c1C(=NC(C)Cc1c2c3ccc(OC)c4c(O)cc(cc34)C)C</chem>	10%
3	<chem>O=C(c2c(O)c3ccc4c(c3(c1oc(nc12)C))CCCC4(C)C)C</chem>	11.11111111111111%
4	<chem>O=C(c2c(O)c3ccc4c(c3(c1ocnc12))CCCC4(C)C)C</chem>	11.904761904762%
5	<chem>O1c2ccc(cc2(OC1))CC(COC)C(COC)Cc3ccc4OCOc4(c3)</chem>	10.869565217391%
6	<chem>O=Cc2c(oc1cc(OC)ccc12)c3ccc(O)cc3(OC)</chem>	13.513513513514%
7	<chem>Oc2cc(cc(OC1OC(CO)C(O)C(O)C1(O))c2C)CCc3ccc(OC)cc3</chem>	10%

New Molecule
New User
Logout

Figure 6- 16: Search and list similar structures with similarity percentage



CHEMICAL STRUCTURES SEARCHING TOOL
UNIVERSITY OF HUDDERSFIELD
AHMAD KLAIB AND HUGH OSBORNE

Administrator

Structure in SMILES Format : OCCc1cc(OC)c2oc(cc2(c1))c3ccc4OCOc4(c3)

KEYWORD_ID	208
KEYWORD	antibacterial activity
KEYWORD_SOUNDEX	53123642313
MOLECULE_ID	10338
DATE	2003-05-21 11:33:47
hyperlink	http://pubchem.ncbi.nlm.nih.gov/substance/PcsSrv.cgi?sid=587140
CAS_NUMBER	530-22-3
NMRSHIFTDB_NUMBER	www.nmrshiftdb.org_sdathe_2003-05-21_11:33:47_0396
SMILES_STRING	<chem>OCCc1cc(OC)c2oc(cc2(c1))c3ccc4OCOc4(c3)</chem>
MOLECULAR_WEIGHT	326.343
SMILES_STRING_CHIRAL	
USER_ID	164

New Molecule
New User
Logout

Figure 6- 17: Details of selected chemical structure

6.7 Discussion

All algorithms except the Brute Force algorithm have a searching phase and a pre-processing phase ([Stephen, 1994](#); [Levitin, 2008](#)).

In this research, exact string matching algorithms were studied in detail and a new classification based on the pre-processing phase of algorithms has been presented. The new classification contains eight categories according to the pre-processing function in the algorithm.

After classifying string matching algorithms in this new taxonomy, the aim was to develop or enhance (a) new string matching algorithm(s) in order to decrease the searching time by increasing the shifting value of the pattern. This research therefore proposes some new string matching algorithms for searching protein sequences, DNA sequences and chemical structures.

The first research methodology aimed to study string matching algorithms, classify them, enhance them or develop new algorithms and then apply them to Protein, DNA sequences and chemical structures.

The result of this methodology proposed five new string matching algorithms; BRBMH, BRQS, OE, RSMA and SSN algorithms. These algorithms aimed to maximize the pattern shifting value, decrease the number of comparisons and therefore enhance searching time.

We chose four well known standard algorithms, BMH, QS, TVSBS and BRFS for comparisons with our algorithms. The BMH algorithm ([Horspool, 1980](#)) was chosen because it

is an enhancement of the original BM algorithm ([Boyer & Moore, 1977](#)) and is the base algorithm of our BRBMH algorithm ([Klaib & Osborne, 2008](#)). The pre-processing function of BMH algorithm depends on the rightmost character. Furthermore, the QS algorithm ([Sunday, 1990](#)) is the basis of our second algorithm, the BRQS algorithm ([Klaib & Osborne, 2009a](#)) and the pre-processing phase depends on a single character next to the rightmost character. The TVSBS algorithm was developed in 2006 ([Thathoo, et al., 2006](#)) as an enhancement of the SSABS algorithm ([Sheik et al., 2004](#)) and it uses the original pre-processing of the BR algorithm which depends on two characters next to the rightmost character as well. The BRFS algorithm was developed in 2008 ([Huang et Al., 2008](#)) as an enhancement of the ZTBMH algorithm ([Huang et. al, 2008](#)) and it uses the original brBc function of the BR algorithm which based on two characters next to the rightmost character.

After implementing the standard algorithms and our enhancements, a sample file of FASTA format DNA sequences was downloaded from the U.S. National Centre of Biotechnology Information ([NCBI, 2012](#)) and another sample file of FASTA format protein sequences was downloaded from the SwissProt Database (UniProt Consortium, 2013).

All of the algorithms were applied to these files and three types of tests were implemented. The first test determines the number of comparisons. The second one calculates the number of attempts and the final one finds the elapsed search time.

Short DNA and protein patterns with lengths 4 – 31 and long patterns with lengths 32-1024 were searched in sample DNA and protein files by taking an average of 10 executions for each pattern length.

For short and long protein and DNA experiments in (section 6.1 – section 6.4) our algorithms showed a lower elapsed search time and required fewer pattern comparisons than all the standard

algorithms. Our algorithms RSMA, OE, BRQS and BRBMH need the same number of attempts because they use the same pre-processing function which used in all of them. Our algorithms require fewer attempts than BMH, QS, TVSBS and BRFS algorithms.

6.7.1 The Number of Comparisons Test Discussion

This test was implemented as described in sub-sections 6.1.1, 6.2.1, 6.3.1 and 6.4.1 and applied to both short and long DNA and protein sequences, and shows a big difference in the number of comparisons our algorithms use, compared to the non-hybrid algorithms and a good difference between our algorithms and other hybrid algorithm such as the TVSBS and the BRFS algorithms.

For example, the BRBMH algorithm searches for short DNA patterns with 41.7% fewer comparisons than the BMH algorithm as shown in Table 6-1, 59.6% fewer when searching for long DNA patterns as shown in Table 6-3, 23.2 % fewer when searching for short protein patterns as shown in Table 6-5 and 68.2% fewer when searching for long protein patterns as shown in Table 6-7.

Table 6-1 shows that the SSN algorithm searches for short DNA patterns with 40.2% fewer comparisons than the TVSBS algorithm, Table 6-3 shows that the SSN algorithm searches for long DNA patterns with 57.1% fewer comparisons than the TVSBS algorithm, Table 6-5 shows that the SSN algorithm searches for short protein patterns with 24 % fewer comparisons than the TVSBS algorithm and Table 6-7 shows that the SSN algorithm searches for long protein patterns with 25.3% fewer comparisons than the TVSBS algorithm.

To compare the RSMA and BRFS algorithms: searching for a DNA pattern with length 28 using the RSMA algorithm compares 45418 times while the BRFS algorithm compares the same pattern for 55291 times.

The reason for this difference, as discussed in literature review and implementation chapters, is that the BMH algorithm depends only on the last character and the QS algorithm depends on a single character next to the rightmost character, while the pre-processing phase in BRBMH and BRQS algorithms depends on two characters next to the rightmost character. This difference in the pre-processing phase in both BRBMH and BRQS algorithms allow an extra shifting value for the pattern which results in fewer comparisons between the pattern and the DNA and protein files.

The TVSBS algorithm uses the BR preprocessing function which shifts the pattern depends on two characters are next to the rightmost character, by at least one character but not more than $m+2$ characters. The SSN algorithm finds a possible start point which reduces the number of comparisons as well as it depends on three characters are next to the rightmost character, which can shift the pattern by at least one character but not more than $m+3$.

This difference in shifting the pattern causes a bigger shift value which results a less comparison time for our algorithms.

6.7.2 The Average Elapsed Search Time Test Discussion

The average elapsed search time tests, as described in sub-sections 6.1.3, 6.2.3, 6.3.3 and 6.4.3 which were applied to both short and long DNA and protein sequences, showed our

algorithms presenting better average elapsed search time than the BRFS, TVSBS, QS and BMH algorithms.

The BRQS algorithm searches for short DNA patterns in 57.7% less time than the QS algorithm as shown in Figure 6-1, 57.4% less time when searching for long DNA patterns as shown in Figure 6-3 and Figure 6-4, 52.1% less time when searching for short protein patterns and 47.8% less time when searching for long protein patterns as shown in Figure 6-6 and Figure 6-7 respectively.

Figure 6-2 shows that the SSN algorithm searches short DNA patterns with an average of 69.2% less time than the BRFS algorithm, Figure 6-3 and Figure 6-5 show that the SSN algorithm searches for long DNA patterns with an average of 57.9% less time than the BRFS algorithm, Figure 6-6 shows that the SSN algorithm searches short protein patterns with 59.9% less time comparing to the BRFS algorithm and Figure 6-9 shows that the SSN algorithm searches for long protein patterns with average 32.3% less time than the BRFS algorithm.

The reason that our algorithms RSMA, OE, BRQS, and BRBMH performed the search in less time comparing to other standard algorithm, that they use the enhanced brBc table which scans only the pattern characters and depends on the next two characters to the rightmost character. Additionally, the enhanced pre-processing phase in the RSMA, OE, BRQS, BRBMH shifts the pattern to the right by $m+2$ comparing to the TVSBS, BMH and QS algorithms.

The variety of searching order in the searching phase in our algorithms such as the RSMA and OE algorithm in certain cases will reduce the searching phase by comparing the odd positions first then the even positions in the OE algorithm. The RSMA algorithm uses a random

value with size S to visit all pattern character positions in a new searching order as explained in sub-section 4.4.2. This gives a chance in certain cases to discover mismatched characters earlier than other algorithms, which leads to fewer comparison numbers and ends after less search time.

The reason that our SSN algorithm performed the search in less time comparing to other algorithms, that it searches for a possible starting point which will reduce the comparisons time and therefore reduces the elapsed search time. In addition, it uses only one preprocessing phase comparing to the BRFS algorithm. The preprocessing phase of the SSN algorithm shifts the pattern to the right by $m+3$ comparing to the RSMA, OE, BRQS, BRFS, TVSBS, BMH and QS algorithms.

6.7.3 The Number of Attempts Test Discussion

The total number of attempts is counted if there is a whole match or a mismatch is encountered which shifts the pattern and a new attempt is started. The pre-processing phase plays a significant role in determining the pattern shifts which affects the total number of attempts.

The average number of attempts tests, in sections 6.1.2, 6.2.2, 6.3.2 and 6.4.2 on both short and long DNA and protein sequences, showed our algorithms, the SSN, RSMA, OE, BRQS and BRBMH algorithms, achieve a better number of attempts compared to the BMH, QS, TVSBS and BRFS algorithms due to the pre-processing phase used in our algorithms. The BRFS algorithm uses two preprocessing functions, QS depends on a single character next to the rightmost character and in the BMH algorithm depends on the rightmost character of the current window.

As mentioned earlier, the RSMA, OE, BRQS and BRBMH algorithms use the enhanced Berry-Ravindran pre-processing phase which results that all algorithms have the same number of attempts on the same pattern length.

Table 6-2 shows that the SSN algorithm searches for short DNA patterns with 43.5% fewer attempts than the RSMA, OE, BRQS and BRBMH algorithms, 46.71% fewer attempts than the BRFS algorithm, 46.7% fewer attempts than the TVSBS algorithm, 67.8% fewer attempts than the QS algorithm and 72.5% fewer attempts than the BMH algorithm.

Table 6-4 shows that the SSN algorithm searches for long DNA patterns with 47.4% fewer attempts than the RSMA, OE, BRQS and BRBMH algorithms, 73.8% fewer attempts than the BRFS algorithm, 73.9% fewer attempts than the TVSBS algorithm, 79.6% fewer attempts than the QS algorithm and 86.1% fewer attempts than the BMH algorithm.

Table 6-6 shows that the SSN algorithm searches for short protein patterns with 15.6% fewer attempts than the RSMA, OE, BRQS and BRBMH algorithms, 15.76% fewer attempts than the BRFS algorithm, 15.75% fewer attempts than the TVSBS algorithm, 39.8% fewer attempts than the QS algorithm and 35.7% fewer attempts than the BMH algorithm.

Table 6-8 shows that the SSN algorithm searches for long protein patterns with 18.5% fewer attempts than the RSMA, OE, BRQS and BRBMH algorithms, 23.81% fewer attempts than the BRFS algorithm, 23.83% fewer attempts than the TVSBS algorithm, 68.9% fewer attempts than the QS algorithm and 77.1% fewer attempts than the BMH algorithm.

6.7.4 The Parallel Algorithm Tests Discussion

Our first algorithm, the BRBMH was parallelized by Prasad and Panicker (2010), and their results show that our algorithm is the best algorithm compared to other ten well-known searching algorithms.

Their result compares eleven searching algorithms on a file with size 12 MB running both sequentially and in parallel on “two Beowulf cluster configurations (Dakshina I & Dakshina II)”. They ran their experiments ten times for different pattern lengths and the results give the average of these ten executions for each length. They had five experiments; sequential execution time, parallel execution time with 5 Nodes on Beowulf cluster Dakshina I, parallel execution time with 5 Nodes on Beowulf cluster Dakshina II, parallel execution time with 10 Nodes on Beowulf cluster Dakshina I, and finally parallel execution time with 10 Nodes on Beowulf cluster Dakshina II (Prasad & Panicker, 2010).

In order to speed up the search time of sequential algorithms, the best algorithm SSN algorithm was implemented and tested using the OpenMP model (shared memory model) and the MPI model (distributed memory model) using the same sample files of DNA and Protein sequences that used in the sequential tests. Both models were implemented and tested at the University of Science Malaysia (USM) on a Stealth Cluster.

The OpenMP model was implemented on a single unit of the Sun Fire V210 using four threads while the MPI model was implemented on two units of the Sun Fire V210 using four processors. The average searching elapsed time result of the OpenMP model using the DNA and protein sequences files was presented in Table 6-9 and Table 6-10 while the result of the MPI

models was presented in Table 6-11 and Table 6-12. The following Table 6-9 and Table 6-10 summarize and compare the average searching elapse time for searching the DNA and protein sequences using the OpenMP and the MPI models:

Pattern Length	MPI	OpenMP	MPI	OpenMP	MPI	OpenMP
	Two Processors	Two Threads	Three Processors	Three Threads	Four Processors	Four Threads
4	9.693	13.752	3.752	6.038	3.146	5.25
8	7.592	12.375	3.128	5.437	2.947	4.775
16	6.954	8.057	2.905	4.489	2.508	3.918
32	3.755	6.172	2.514	3.713	2.239	3.273
64	3.301	4.435	2.181	2.947	1.924	2.663
128	2.086	3.28	1.793	2.205	1.574	2.008
256	1.599	2.819	1.158	1.869	1.043	1.547
512	1.488	2.067	1.047	1.416	0.947	1.159
1024	1.419	1.849	0.992	1.154	0.86	1.001
Table 6- 9: MPI vs. OpenMP: average elapsed search time for searching DNA						

Table 6-9 shows that the MPI model performed the search of DNA patterns in less time compared to the OpenMP model. As an example, the MPI model searches a pattern with length 4 in 9.693 seconds using two processors, 3.752 seconds using three processors and 3.146 seconds using four processors while the OpenMP model searched the same pattern length in 13.752 seconds using two threads, 6.038 seconds using three threads and 5.25 seconds using four threads.

In addition, Table 6-9 shows that the MPI model searched for DNA patterns with an average of 37.6% less time using two processors compared to the OpenMP using two threads, 33.8% less time using three processors compared to the OpenMP using three threads and 21.6% less time using four processors compared to the OpenMP using four threads.

Pattern Length	MPI	OpenMP	MPI	OpenMP	MPI	OpenMP
	Two Processors	Two Threads	Three Processors	Three Threads	Four Processors	Four Threads
4	12.824	19.823	4.072	6.827	3.829	4.691
8	9.282	16.607	3.636	5.73	3.482	4.017
16	8.439	16.832	3.386	4.992	2.954	3.672
32	6.708	8.753	3.041	4.305	2.792	3.304
64	5.211	6.615	2.605	3.741	2.176	2.98
128	3.573	5.385	2.075	3.195	1.544	2.375
256	3.073	4.725	1.801	2.743	1.307	1.927
512	2.153	3.914	1.572	2.248	1.201	1.628
1024	1.997	2.708	1.404	1.871	1.067	1.376
Table 6- 10: MPI vs. OpenMP: average elapsed search time for searching protein						

Table 6-10 shows that the MPI model performed the search of protein patterns in less time compared to the OpenMP model. As an example, the MPI model searched a pattern with length 16 in 8.439 seconds using two processors, 3.386 seconds using three processors and 2.954 seconds using four processors while the OpenMP model searched the same pattern length in 16.832 seconds using two threads, 4.992 seconds using three threads and 3.672 seconds using four threads.

In addition, Table 6-10 shows that the MPI model searched for protein patterns with an average of 30.8% less time using two processors compared to the OpenMP model using two threads, 33.4% less time using three processors compared to the OpenMP model using three threads and 32.8% less time using four processors compared to the OpenMP model using four threads.

To evaluate the speedup of using the MPI and OpenMP models over the original SSN algorithm we used the equation that mentioned in sub-section 2.12.2. Table 6-11 and Table 6-12 shows the speedup of using the MPI and OpenMP models

Pattern Length	MPI	OpenMP	MPI	OpenMP	MPI	OpenMP
	Two Processors	Two Threads	Three Processors	Three Threads	Four Processors	Four Threads
4	1.596	1.125	4.123	2.562	4.917	2.946
8	1.537	1.327	3.680	2.381	4.262	2.728
16	2.094	1.274	3.128	2.118	3.512	2.403
32	1.699	1.265	2.572	1.904	2.916	2.107
64	2.192	1.394	2.550	2.073	2.905	2.277
128	2.445	1.387	3.376	2.091	3.748	2.527
256	2.375	1.710	3.375	2.496	3.732	3.049
512	2.385	1.831	3.412	2.933	3.936	3.382
1024	1.596	1.125	4.123	2.562	4.917	2.946
Table 6- 11: The speedup of MPI and OpenMP models for DNA patterns						

Pattern Length	MPI	OpenMP	MPI	OpenMP	MPI	OpenMP
	Two Processors	Two Threads	Three Processors	Three Threads	Four Processors	Four Threads
4	1.874	1.213	5.903	3.521	6.278	5.124
8	1.583	0.793	3.944	2.675	4.521	3.637
16	1.582	1.213	3.491	2.466	3.802	3.213
32	1.682	1.325	3.365	2.343	4.029	2.942
64	2.188	1.452	3.768	2.447	5.063	3.292
128	2.377	1.546	4.056	2.663	5.589	3.791
256	3.241	1.783	4.438	3.104	5.809	4.286
512	2.967	2.188	4.220	3.167	5.553	4.306
1024	1.874	1.213	5.903	3.521	6.278	5.124
Table 6- 12: The speedup of MPI and OpenMP models for protein patterns						

Table 6-11 and Table 6-12 show that the average speedup of searching DNA and protein sequences using the OpenMP and MPI models is a sub-linear speedup.

CHAPTER 7: CONCLUSION AND FUTURE WORK

This thesis introduced a new classification of string matching algorithms, developed five new string matching algorithms, developed a chemical searching tool kit using the best string matching algorithm and ends by parallelizing the best algorithm to speed up the searching time. This chapter includes the main conclusion of the research and suggests some possible future work.

7.1 Conclusion

The massive amount of biological and chemical data which is used daily and stored in files and databases requires an efficient string matching algorithm to speed up the searching processes for required sequences and structures. Here is a recap of the research questions for chapter 1. The answers developed in this thesis are discussed below:

1. Which of the existing string pattern matching algorithm(s) is/are the most suitable for searching biological sequences and chemical structures?
2. Can we enhance one or more of the proposed algorithms in 1, or develop (a) new algorithm(s) for string-matching?
3. How we can measure the success of the new developed algorithm(s) compared to the best algorithm in 1?
4. Can we develop a classification of string matching algorithms which will help with achieving our aims?

The answers to questions one and four are presented by studying the standard string matching algorithms. Question number four was answered by developing a new classification containing eight categories based on the pre-processing function of each algorithm. Question number one was answered by analysing the standard algorithms and choosing the appropriate ones before we tested them with our algorithms.

The above helped us to answer question number two by studying, in depth, most of standard algorithms and applying changes to enhance them. Firstly we enhanced the pre-processing phase of Berry Ravindran algorithm by creating a one dimensional array to store the pattern characters rather than using a two dimensional array to store the text characters. In addition, the use of the enhanced brBc over the hsBc and qsBc tables provided two benefits: the first one is the enhanced brBc table shifts the pattern to the right by $m+2$ positions comparing to the hsBc which shifts pattern only m positions as well as the qsBc which shifts pattern only $m+1$ positions if there is a whole match or a mismatch encountered. The second benefit is reducing the preprocessing time by scanning only the pattern characters.

Secondly we developed five new algorithms, the BRBMH, BRQS, OE, RSMA and SSN algorithms. The BRBMH algorithm used the enhanced preprocessing phase instead of the preprocessing phase of Horspool algorithm and combined it with the searching phase of Horspool algorithm. The BRQS algorithm used the enhanced preprocessing phase instead of the preprocessing phase of Quick Search algorithm and combined it with the searching phase of the Quick Search algorithm. The OE algorithm combined the enhanced pre-processing phase and searches the pattern in the text using a new searching order. The RSMA algorithm combined the enhanced pre-processing phase and reduced the searching phase by comparing the pattern with text window in a new order depending on a generated random value of size S . The SSN

algorithm uses the ASS table to define a possible starting point to compare the text and the pattern characters. If the last three characters in the current text window or the next three characters exists in the ASS table, the pattern is aligned and compared otherwise the pattern is shifted to the right by $m+3$ positions.

Question three was answered by downloading sample DNA and proteins sequence FASTA files, applying our algorithms and four well known standard algorithms and then measuring the success through three types of tests. The number of comparisons tests, showed a big difference in the number of comparisons our algorithms use, compared to the non-hybrid algorithms and a good difference between our algorithms and other hybrid algorithm such as TVSBS and BRFS. The average elapsed search time tests showed our algorithms presenting better average searching elapsed time than the BRFS, TVSBS, QS and BMH algorithms. The average number of attempts tests showed our algorithms achieve better number of attempts comparing to the BMH, QS, TVSBS and BRFS algorithms.

A chemical toolkit was developed to draw chemical structures and convert them to SMILES format and then use the SSN algorithm to search for structures in the local database.

And finally the parallel algorithm design included a new contribution where the SSN algorithm was parallelized using the OpenMP and the MPI models.

7.2 Future Work

Several research issues can be explored in the future:

- The toolkit can be developed by applying approximate string matching algorithms. This enhancement will help biologists and chemists in their advanced search purposes where they can predict missing sequences and structures.
- The toolkit can be expanded by adding more sequences and structures from different available sources.
- Cooperating with biologists and chemists can lead to the addition of more features to the toolkit.
- The toolkit can be presented as a portal or searching engine for different users.
- Our SSN algorithm can be implemented alongside up to date string matching algorithms to test the efficiency of the new algorithm.
- The parallel algorithm design in section 3.3 can be implemented to improve the searching time using the hybrid memory model which combines both the Open Multi-Processing (OpenMP) and Message Passing Interface (MPI) parallel models.
- To run the parallel algorithm on all processors of the Stealth Cluster at the University of Science Malaysia which contains eight processors running on four Sun Fire machines.
- The parallel algorithm tests and results in section 6.5 can be expanded to test large files size which will show the efficiency of parallel algorithms and the used cluster.

REFERENCES

- Abdelaziz, T., Elammari, M., & Branki, C. (2008). MASD: towards a comprehensive multi-agent system development methodology. Paper presented at the On the Move to Meaningful Internet Systems: OTM 2008 Workshops.
- AbdulRazzaq, A., Rashid, N., & Ali, A. (2013). Fast Hybrid String Matching Algorithm. *International Journal of Digital Content Technology and its Applications*, 7(10), 62-71.
- Agustina, T. P. (2012). The Double Helix Retrieved July, 2013, from <http://alangaesia.blogspot.co.uk/2012/12/the-double-helix.html>
- Akl, S. G. (1997). *Parallel Computation: Models and Methods*. New Jersey: Prentice Hall, Inc.
- Almazroi, A. (2011). A Fast Hybrid Algorithm Approach for the Exact String Matching Problem Via Berry Ravindran and Alpha Skip Search Algorithms. *Journal of Computer Science*, 7(5), 644-650.
- Almazroi, A., & Rashid, N. (2011). A Fast Hybrid Algorithm for the Exact String Matching Problem. *American J. of Engineering and Applied Sciences*, 4(1), 102-107.
- Atallah, M. J. (2002). *Algorithms and theory of computation handbook*: CRC press.
- Baeza-Yates, R. (1992). String searching algorithms. In *Information Retrieval: Algorithms and Data Structures*: Prentice Hall, Englewood Cliffs, N.J.
- Bailey, R. (2006). Protein Structure Retrieved July, 2013, from <http://biology.about.com/od/molecularbiology/ss/protein-structure.htm>
- Bairoch A, A. R. (2000). The SWISS-PROT protein sequence database and its supplement TrEMBL in 2000. *Nucleic acids research*, 28(1), 45-48. doi: 10.1093/nar/28.1.45

- Barney, B. (2010). Introduction to parallel computing. Lawrence Livermore National Laboratory, 6(13), 10.
- Barney, B. (2013). Message Passing Interface (MPI). Retrieved August, 2013, from <https://computing.llnl.gov/tutorials/mpi/>
- Baydaa, A. (2011). Retrieving Information from Compressed XML Documents According to Vague Queries. Doctor of Philosophy, University of Huddersfield, Huddersfield, UK.
- Berg JM, T. J., Stryer L. (2002). Biochemistry (5 ed.): W. H. Freeman.
- Berman, H., Westbrook J., Feng Z., Gilliland G., (2000). The Protein Data Bank. Nucleic Acids Research. (2000) 28 (1):235-242.doi: 10.1093/nar/28.1.235
- Berry, T., & Ravindran, S. (1999). A fast string matching algorithm and experimental results. Proceedings of the Prague Stringology Club Workshop'99 (Prague: Czech Republic), 16-26.
- Bhandari, J. (2014). Techniques Used in String Matching for Network. International Journal of Computer, Information, Systems and Control Engineering, 8(5), 805-808
- Bhandari, J. & Kumar, A. (2014). A Survey of Fast Hybrids String matching Algorithms. International Journal of Emerging Sciences, 4(1), 24-37.
- Bourne, P. E., Westbrook, J., & Berman, H. M. (2004). The Protein Data Bank and lessons in data management. Briefings in bioinformatics, 5(1), 23-30. doi: 10.1093/bib/5.1.23
- Boyer, R. S., & Moore, J. S. (1977). A fast string searching algorithm. Communications of the ACM, 20(10), 762-772.
- Cantone, D., Cristofaro, S., & Faro, S. (2004). Efficient Algorithms for the delta-Approximate String Matching Problem in Musical Sequences. Paper presented at the Stringology.

- Cantone, D., Cristofaro, S., & Faro, S. (2005). An efficient algorithm for δ -approximate matching with α -bounded gaps in musical sequences *Experimental and Efficient Algorithms* (pp. 428-439): Springer.
- Cantone, D., & Faro, S. (2003). Fast-Search: A new efficient variant of the Boyer-Moore string matching algorithm *Experimental and Efficient Algorithms* (pp. 47-58): Springer.
- Chai, T. Y., Juhari, M., Woo, S. S., & Tan, C. S. (2009). Facial features for template matching based face recognition.
- Charras, C., Lecrog, T., & Pehoushek, J. D. (1998). A very fast string matching algorithm for small alphabets and long patterns. Paper presented at the Combinatorial Pattern Matching.
- Charras, C., & Lecroq, T. (1997). Exact string matching algorithms *Animation in Java*
Retrieved from <http://www-igm.univ-mlv.fr/~lecroq/string/>
- Charras, C., & Lecroq, T. (2004). *Handbook of exact string matching algorithms*: King's College Publications.
- Chen, Y. (2007). A new algorithm for subset matching problem. *Journal of Computer Science*, 3(12), 924.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (1990). *Introduction to Algorithms*: MIT Press and McGraw Hill.
- Crick, F. (1974). The double helix: a personal view. *Nature*, 248(5451), 766-769.
- Crochemore, M., Czumaj, A., Gasieniec, L., Jarominek, S., Lecroq, T., Plandowski, W., & Rytter, W. (1994). Speeding up two string-matching algorithms. *Algorithmica*, 12(4-5), 247-267.
- Crochemore, M., & Rytter, W. (1994). *Text algorithms* (Vol. 698): World Scientific.

- Danvy, O., & Rohde, H. K. (2006). On obtaining the boyer–moore string-matching algorithm by partial evaluation. *Information Processing Letters*, 99(4), 158-162.
- Daylight Chemical Information Systems, I. (2008). SMILES - A Simplified Chemical Language. Retrieved July, 2012, from <http://www.daylight.com/dayhtml/doc/theory/theory.smiles.html>
- De Raedt, L., & Kramer, S. (2003). Inductive databases for bio-and chemo-informatics. *NATO SCIENCE SERIES SUB SERIES III COMPUTER AND SYSTEMS SCIENCES*, 183, 193-207.
- Degrave, W., Huynh, C., Roos, D., Oduola, A., & Morel, C. M. (2002). Bioinformatics for disease endemic countries: opportunities and challenges in science and technology development for health. *IUPAC Journal*, 1, 1-7.
- DeGray, G., Rajasekaran, K., Smith, F., Sanford, J., & Daniell, H. (2001). Expression of an antimicrobial peptide via the chloroplast genome to control phytopathogenic bacteria and fungi. *Plant physiology*, 127(3), 852-862. doi: 10.1104/pp.010233
- Deusdado, S., & Carvalho, P. (2009). GRASPM: an efficient algorithm for exact pattern-matching in genomic sequences. *International Journal of Bioinformatics Research and Applications*, 5(4), 385-401.
- Ertl, P. (2006) JME Molecular Editor. Retrieved October, 2013, from <http://www.molinspiration.com/jme/index.html>
- Ertl, P. (2010). Molecular structure input on the web. *Journal of Cheminformatics*, 2(1), 1-9.
- EMBL-EBI, European Bioinformatics Institute UniProt. (2002). UniProt keywords. Retrieved January, 2012, from <http://www.uniprot.org/keywords/>
- European Molecular Biology Laboratory EMBL. (2006). EMBL Nucleotide Sequence Database. Retrieved December, 2008

- Expasy Bioinformatics Resource Portal. (2013). UniProtKB/Swiss-Prot protein knowledgebase release 2013_08 statistics. Retrieved July, 2013, from <http://web.expasy.org/docs/relnotes/relstat.html>
- Flynn, M. J. (1966). Very high-speed computing systems. *Proceedings of the IEEE*, 54(12), 1901-1909.
- Flynn, M. J. (1972). Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9), 948-960.
- Foster, I. (1995). *Designing and building parallel programs* (Vol. 95): Addison-Wesley Reading.
- Franek, F., Jennings, C., & Smyth, W. (2005). A Simple Fast Hybrid Pattern-Matching Algorithm. *Springer-Verlag CPM2005* (pp. 288-297). Berlin: Springer-Verlag.
- Freder, V., Ho, B., & Ding, J. L. (2004). De Novo design of potent antimicrobial peptides. *ANTIMICROBIAL AGENTS AND CHEMOTHERAPY*, 48(9), 3349-3357. doi: 10.1128/aac.48.9.3349-3357.2004
- Fredriksson, K., & Grabowski, S. (2005). Practical and optimal string matching. Paper presented at the String Processing and Information Retrieval.
- Fredriksson, K., & Mozgovoy, M. (2006). Efficient parameterized string matching. *Information Processing Letters*, 100(3), 91-96.
- Fujimura, M., Minami, Y., Watanabe, K., & Tadera, K. (2003). Purification, characterization, and sequencing of a novel type of antimicrobial peptides, Fa-AMP1 and Fa-AMP2, from seeds of buckwheat (*Fagopyrum esculentum* Moench.). *Bioscience, biotechnology, and biochemistry*, 67(8), 1636-1642. doi: 10.1271/bbb.67.1636
- Hasan, A., & Rashid, N. (2012). Hybrid Exact String Matching Algorithm for Intrusion Detection System. *Taibah University International Conference on Computing and Information Technology (ICCIT2012)*, (pp. 181-185). Madinah.

- Hevner, A., March, S., Park, J. & Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly*, 28, 75-105.
- Horspool, R. N. (1980). Practical fast searching in strings. *Software: Practice and Experience*, 10(6), 501-506.
- Horton, R. M. (2004). *Bioinformatics Algorithm Demonstrations in Microsoft Excel*. Master, California State University, Sacramento.
- Huang, Y., Pan, X., Gao, Y., & Cai, G. (2008a). [ZTBMH]A fast pattern matching algorithm for biological sequences. Paper presented at the Bioinformatics and Biomedical Engineering, 2008. ICBBE 2008. The 2nd International Conference on.
- Huang, Y., Ping, L., Pan, X., & Cai, G. (2008b). [BRFS]A fast exact pattern matching algorithm for biological sequences. Paper presented at the BioMedical Engineering and Informatics, 2008. BMEI 2008. International Conference on.
- Hume, A., & Sunday, D. (1991). Fast string searching. *Software: Practice and Experience*, 21(11), 1221-1248.
- Hulzebos, E., Janssen, P., Maslankiewicz, L., Meijerink, M., Muller, J., Pelgrom, S., . . . Vermeire, T. (2001). The application of structure-activity relationships in human hazard assessment: a first approach.
- Jaccard P. (1912). The distribution of the flora in the alpine zone. *New Phytologist*, 11(2), 37-50.
- Kalsi, P., Peltola, H., & Tarhio, J. (2008). Comparison of exact string matching algorithms for biological sequences *Bioinformatics Research and Development* (pp. 417-426): Springer.
- Karp, R. M., & Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2), 249-260.
- Kiessling, A. (2009, April). The Royal Observatory, Edinburgh. Retrieved September 2014, from An Introduction to Parallel Programming with OPenMP: http://www.roe.ac.uk/ifa/postgrad/pedagogy/2009_kiessling.pdf

- Kim, J. W., Kim, E., & Park, K. (2007). Fast matching method for DNA sequences. *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies* (pp. 271-281): Springer.
- Klaib, A., & Osborne, H. (2008). Searching protein sequence databases using BRBMH matching algorithm. *International Journal of Computer Science and Network Security*, 8(12), 410-414.
- Klaib, A., & Osborne, H. (2009a). BRQS matching algorithm for searching protein sequence databases. Paper presented at the Future Computer and Communication, 2009. ICFCC 2009. IEEE Conference
- Klaib, A., & Osborne, H. (2009b). OE Matching Algorithm for Searching Biological Sequences: ISRST.
- Klaib, A., & Osborne, H. (2009c). RSMA matching algorithm for searching biological sequences. Paper presented at the Innovations in Information Technology, 2009. IIT'09. IEEE International Conference.
- Knuth, D., Morris, J., & Pratt, V. (1977). Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2), 323-350.
- Kontoghiorghes, E. J. (2010). *Handbook of parallel computing and statistics*: CRC Press.
- Kuhn, S. (2010). NMRShiftDB. Retrieved January, 2013, from <http://nmrshiftdb.nmr.uni-koeln.de/>
- Lecroq, T. (1995). Experimental results on string matching algorithms. *Software: Practice and Experience*, 25(7), 727-765.
- Lecroq, T. (1998). Experiments on string matching in memory structures. *Software-Practice and Experience*, 28(5), 561-568.

- Lecroq, T. (2007). Fast exact string matching algorithms. *Information Processing Letters*, 102(6), 229-235.
- Lee, J. (2004). Analysis of Fundamental Exact and Inexact Pattern Matching Algorithms.
- Levitin, A. (2008). *Introduction To Design And Analysis Of Algorithms*, 2/E: Pearson Education India.
- Lokman, A. S., & Zain, J. M. (2010). One-Match and All-Match Categories for Keywords Matching in Chatbot. *American Journal of Applied Sciences*, 7(10), 1406.
- Medicine, U. S. N. L. o. (2013). What is DNA? Retrieved July, 2013, from <http://ghr.nlm.nih.gov/handbook/basics/dna>
- Mhashi, M. (2012). An Intelligent and Efficient Matching Algorithm to Finding a DNA Pattern. *International Magazine on Advances in Computer Science and Telecommunications*, 3(1), 13-25.
- Morley, P. S., Apley, M. D., Besser, T. E., Burney, D. P., Fedorka-Cray, P. J., Papich, M. G., . . . American College of Veterinary Internal, M. (2005). Antimicrobial drug use in veterinary medicine. *Journal of veterinary internal medicine / American College of Veterinary Internal Medicine*, 19(4), 617-629. doi: 10.1111/j.1939-1676.2005.tb02739.x
- Morrison, J., & George, J. F. (1995). Exploring the software engineering component in MIS research. *Communications of the ACM*, 38(7), 80-91.
- Nakamura, H. (2003). Announcing the worldwide Protein Data Bank. *Nature Structural Biology*, 10(12), 980.
- Naser, M., Rashid, N., & Aboalmaaly, M. (2012). Quick-Skip Search Hybrid Algorithm for the Exact String Matching Problem. *International Journal of Computer Theory and Engineering*, 4(2), 259-265.

- National Center for Biotechnology Information NCBI. (2012). FTP access to GenBank data. Retrieved July, 2012, from <ftp://ftp.ncbi.nih.gov/genbank>
- NCBI, National Center for Biotechnology Information (2013). GeneBank Database Overview. Retrieved July, 2013, from <http://www.ncbi.nlm.nih.gov/genbank/>
- NIST, National Institute of Standards and Technology. (2013). Computing molar mass (molar weight). Retrieved August, 2013, from <http://www.webqc.org/mmcalc.php>
- Navarro, G., & Raffinot, M. (2000). Fast and flexible string matching by combining bit-parallelism and suffix automata. *Journal of Experimental Algorithmics (JEA)*, 5, 4.
- Navarro, G., & Raffinot, M. (2002). Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences: Cambridge University Press.
- Neglur, G., Grossman, R. L., & Liu, B. (2005). Assigning unique keys to chemical compounds for data integration: Some interesting counter examples. Paper presented at the Data Integration in the Life Sciences.
- Nunamaker Jr, J. F., & Chen, M. (1990). Systems development in information systems research. Paper presented at the System Sciences, 1990., Proceedings of the Twenty-Third Annual Hawaii International Conference on.
- Pang, K. C., Stephen, S., Dinger, M. E., Engstrom, P. G., Lenhard, B., & Mattick, J. S. (2007). RNADB 2.0--an expanded database of mammalian non-coding RNAs. *Nucleic acids research*, 35(Database), D178-D182. doi: 10.1093/nar/gkl926
- Prasad, J., & Panicker, K. (2010). String Searching Algorithm Implementation-Performance Study with Two Cluster Configuration. *International Journal of Computer Science and Communication*, 551-555.
- Radhakrishna, V., Phaneendra, B., & Sangeeth Kumar, V. (2010). A two way pattern matching algorithm using sliding patterns. Paper presented at the Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on.

- Raita, T. (1992). Tuning the boyer moore horspool string searching algorithm. *Software: Practice and Experience*, 22(10), 879-884.
- Rajasekaran, S., & Reif, J. (2007). *Handbook of parallel computing: models, algorithms and applications*: CRC Press.
- Raju, S. V., & Babu, A. V. (2007). Parallel algorithms for string matching problem on single and two dimensional reconfigurable pipelined bus systems. *Journal of Computer Science*, 3(9), 754.
- Regnier, M., & Szpankowski, W. (1998). *Complexity of sequential pattern matching algorithms*: Springer.
- Rivals, E., Salmela, L., & Tarhio, J. (2011). EXACT SEARCH ALGORITHMS FOR BIOLOGICAL SEQUENCES. *Algorithms in Computational Molecular Biology: Techniques, Approaches and Applications*, 1, 1-22.
- Roosta, S. H. (2000). *Parallel processing and parallel algorithms: theory and computation*: Springer.
- Rowley, R. J., Oscarson, J. L., Rowley, R. L., & Wilding, W. V. (2001). Development of an automated SMILES pattern matching program to facilitate the prediction of thermophysical properties by group contribution methods. *Journal of Chemical & Engineering Data*, 46(5), 1110-1113.
- SaiKrishna, V., Rasool, A., & Khare, N. (2012). String Matching and its Applications in Diversified Fields. *IJCSI International Journal of Computer Science*, 9(1), 219-226.
- Schulz, J. (2008). Jaccard Similarity. *Algorithms - Similarity*. Retrieved August 2014, from http://www.code10.info/index.php%3Foption%3Dcom_content%26view%3Darticle%26id%3D60:article_jaccard-similarity%26catid%3D38:cat_coding_algorithms_data-similarity%26Itemid%3D57

- Searls, D. B., & Hogeweg, P. (2011). The Roots of Bioinformatics in Theoretical Biology. *PLoS computational biology*, 7(3), 1-5. doi: 10.1371/journal.pcbi.1002021
- Sedgewick, R. (1988). *Algorithms in Java, Parts 1-4 (Vol. 1)*: Addison-Wesley Publishing Company.
- Setubal, J. C., Meidanis, J., & Setubal-Meidanis. (1997). *Introduction to computational molecular biology*: PWS Pub.
- Sheik, S., Aggarwal, S. K., Poddar, A., Balakrishnan, N., & Sekar, K. (2004). A FAST pattern matching algorithm. *Journal of chemical information and computer sciences*, 44(4), 1251-1256.
- Sleit, A., AlMobaideen, W., Qatawneh, M., & Saadeh, H. (2009). Efficient processing for binary submatrix matching. *American Journal of Applied Sciences*, 6(1), 78.
- Srikantha, A., Bopardikar, A. S., Kaipa, K. K., Venkataraman, P., Lee, K., Ahn, T., & Narayanan, R. (2010). A fast algorithm for exact sequence search in biological sequences using polyphase decomposition. *bioinformatics*, 26(18), i414-i419.
- Stephen, G. A. (1994). *String searching algorithms*: World Scientific.
- Steven, H. (2011). *Chemistry 14C: Structure of Organic Molecule: Course Thinkbook : Concept Focus Questions, OWLS Problems, Practice Problems*: Plymouth, MI: Hayden-McNeil.
- Sunday, D. M. (1990). A very fast substring search algorithm. *Communications of the ACM*, 33(8), 132-142.
- Tarhio, J., & Peltola, H. (1997). String matching in the DNA alphabet. *Software-Practice and Experience*, 27(7), 851-861.
- Teknomo, K. (2006). Similarity Measurement, from <http://people.revoledu.com/kardi/tutorial/Similarity/Jaccard.html>

- Thathoo, R., Virmani, A., Sai Lakshmi, S., Balakrishnan, N., & Sekar, K. (2006). TVSBS: A fast exact pattern matching algorithm for biological sequences. *Current Science*, 91(1), 47-53.
- Tsai, T. H. (2006). Average case analysis of the Boyer-Moore algorithm. *Random Structures & Algorithms*, 28(4), 481-498.
- Tucker, A. B. (2004). *Computer science handbook*: CRC press.
- U.S. Environmental Protection Agency EPA. (2009). SMILES Tutorial. Retrieved July, from http://www.epa.gov/med/Prods_Pubs/smiles.htm
- U.S. National Library of Medicine. (2013). What is DNA? Retrieved July, 2013, from <http://ghr.nlm.nih.gov/handbook/basics/dna>
- UniProt Consortium. (2013). Downloads Protien Fasta Format. Retrieved July, 2013, from <http://www.uniprot.org/downloads>
- Wang, G. (2010). *Antimicrobial Peptides: Discovery, Design and Novel Therapeutic Strategies*: CAB International.
- Wang, G., Li, X., & Wang, Z. (2009). APD2: the updated antimicrobial peptide database and its application in peptide design. *Nucleic acids research*, 37(Database), D933-D937. doi: 10.1093/nar/gkn823
- Wang, P. & Li, Y. (2011). Automaton Based String Matching Algorithm and its application in Intrusion Detection. *International Journal of Advancements in Computing Technology (IJACT)*, 3(9), 278-285.
- Wang, Y., & Kobayashi, H. (2006). High performance pattern matching algorithm for network security. *IJCSNS*, 6(10), 83.
- Waterman, M. S. (1995). *Introduction to computational biology: maps, sequences and genomes*: Chapman & Hall Ltd.

- Weininger, D. (1988). SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules. *Journal of chemical information and computer sciences*, 28(1), 31-36.
- Weininger, D., Weininger, A., & Weininger, J. L. (1989). SMILES. 2. Algorithm for generation of unique SMILES notation. *Journal of chemical information and computer sciences*, 29(2), 97-101.
- Wilkinson B. & Allen M. (2005). *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. New Jersey: Prentice Hall Inc.
- Willmore, F. (2012, February). Texas Advances Computing Center. Retrieved September 2014, from Introduction to parallel Computing: https://www.tacc.utexas.edu/c/document_library/get_file?uuid=e05d457a-0fbf-424b-87ce-c96fc0077099
- Wu, S., & Manber, U. (1994). A fast algorithm for multi-pattern searching: Technical Report TR-94-17, University of Arizona.
- Xian-feng, H., Yu-bao, Y., & Lu, X. (2010). Hybrid pattern-matching algorithm based on BM-KMP algorithm. 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE) , (pp. 310-313).
- Xu, J. H., A. (2002). Chemoinformatics and Drug Discovery. *Molecules Journal*, 7(8), 566-600. doi: 10.3390/70800566
- ZHU, R.F. and TAKAOKA, T. (1987). On improving the average case of the Boyer-Moore string matching algorithm. *Journal of Information Processing* 10(3), 173-177.

APPENDIX A: STRING MATCHING ALGORITHMS CODE

```
void BruteForce(char *text, int textLength, char *pattern, int patternLength) {
    int iCount = 0, jCount = 0;

    /* Preprocessing */
    // There is no preprocessing phase in the BF algorithm

    /* Searching */
    for (iCount = 0; iCount <= textLength - patternLength; ++iCount) {
        jCount = 0
        for (; jCount < patternLength && pattern[jCount] == text[iCount + jCount]; ++jCount);
        if (jCount == patternLength){
            OUTPUT(iCount);
        }
    }
}
```

Figure A- 1: The Brute Force algorithm code


```

void preprocessing_bmBc(char *pattern, int patternLength, int bmBc[ASIZE]) {
    int iCounter, jCounter;
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        bmBc[iCounter] = -1;
    }
    for (jCounter = 0; jCounter < patternLength; ++jCounter){
        iCounter=pattern[jCounter];
        bmBc[iCounter] = jCounter;
    }
}

void preprocessing_bmGs(char *pattern, int patternLength, int bmGs[XSIZE]) {
    int iCounter= patternLength, jCounter= patternLength + 1;
    bmGs[iCounter] = jCounter;
    while (iCounter > 0) {
        while (jCounter <= patternLength && pattern[iCounter - 1] != pattern[jCounter - 1]){
            if (bmGs [jCounter] == 0) {
                bmGs [jCounter] = jCounter - iCounter;
            }
            jCounter = bmGs[jCounter];
        }
        iCounter--, jCounter--;
        bmGs[iCounter] = jCounter;
    }
}

void BoyerMoore(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, jCounter, bmBc[ASIZE], bmGs[XSIZE];

    /* Preprocessing */
    preprocessing_bmBc(pattern, patternLength, bmBc);
    preprocessing_bmGs(pattern, patternLength, bmGs);

    /* Searching */
    iCounter = 0;
    while (iCounter <= textLength - patternLength) {
        jCounter = patternLength - 1;
        while (jCounter >= 0 && pattern[jCounter] == text[iCounter + jCounter]){
            --jCounter;
        }
        if (jCounter < 0){
            OUTPUT(iCounter);
            iCounter += bmGs[0];
        }
        else{
            iCounter += MAX(bmGs[jCounter+1], jCounter - bmBc[text[iCounter + jCounter]]);
        }
    }
}

```

Figure A- 2: The Boyer-Moore algorithm code

```

void preprocessing_ztBc(char *pattern, int patternLength, int ztBc[ASIZE][ASIZE]) {
    int iCounter, jCounter;
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        for (jCounter = 0; jCounter < ASIZE; ++jCounter){
            ztBc[iCounter][jCounter] = patternLength;
        }
    }
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        ztBc[iCounter][pattern[0]] = patternLength - 1;
    }
    for (iCounter = 1; iCounter < patternLength - 1; ++iCounter){
        ztBc[pattern[iCounter - 1]][pattern[iCounter]] = patternLength - 1 - iCounter;
    }
}

void preprocessing_ztGs(char *pattern, int patternLength, int ztGs[XSIZE]) {
    // Same preprocessing code as the bmGs function
}

void ZhuTakaoka(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, jCounter, ztBc[ASIZE][ASIZE], ztGs[XSIZE];

    /* Preprocessing */
    preprocessing_ztBc(pattern, patternLength, ztBc);
    preprocessing_ztGs(pattern, patternLength, ztGs);

    /* Searching */
    iCounter = 0;
    while (iCounter <= textLength - patternLength) {
        jCounter = patternLength - 1;
        while (jCounter < patternLength && pattern[jCounter] == text[iCounter + jCounter]){
            --jCounter;
        }
        if (jCounter < 0){
            OUTPUT(iCounter);
            iCounter += ztGs[0];
        }
        else{
            iCounter += MAX(ztGs[jCounter], ztBc[text[iCounter + patternLength - 2]][text[iCounter +
patternLength - 1]]);
        }
    }
}

```

Figure A- 3: The Zhu-Takaoka algorithm code

```

void preprocessing_fsBc(char *pattern, int patternLength, int fsBc[ASIZE]) {
    // Same preprocessing code as the bmBc function
}

void preprocessing_fsGs(char *pattern, int patternLength, int fsGs[XSIZE]) {
    // Same preprocessing code as the bmGs function
}

void FastSearch(unsigned char *pattern, int patternLength, unsigned char *text, int textLength) {
    int iCounter, jCounter, fsBc[ASIZE], fsGs[XSIZE];

    /* Preprocessing */
    preprocessing_fsBc(pattern, patternLength, fsBc);
    preprocessing_fsGs(pattern, patternLength, fsGs);

    /* Searching */
    iCounter = 0;
    while (fsBc[text[iCounter + patternLength - 1]] > 0){
        iCounter = iCounter + fsBc[text[iCounter + patternLength - 1]];
    }
    while (iCounter <= textLength - patternLength){
        jCounter = patternLength - 2;
        while (jCounter >= 0 && pattern[jCounter] == text[iCounter + jCounter]){
            jCounter = jCounter - 1;
        }
        if (jCounter < 0){
            OUTPUT(iCounter);
        }
        iCounter += fsGs[jCounter + 1];
        while (fsBc[text[iCounter + patternLength - 1]] > 0){
            iCounter = iCounter + fsBc[text[iCounter + patternLength - 1]];
        }
    }
}

```

Figure A- 4: The Fast Search algorithm code

```

void preprocessing_hrBc(char *pattern, int patternLength, int hrBc[ASIZE]) {
    // Same preprocessing code as the bmBc function
}

void Horspool(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, jCounter, patternCounter, lastCharacter, hrBc[ASIZE];

    /* Preprocessing */
    preprocessing_hrBc(pattern, patternLength, hrBc);

    /* Searching */
    iCounter = 0;
    while (iCounter <= textLength - patternLength) {
        patternCounter = patternLength - 1;
        jCounter = iCounter + patternLength - 1;
        lastCharacter = text[jCounter];
        if (pattern[patternCounter] == lastCharacter){
            patternCounter = 0;
            while (jCounter > iCounter && pattern[patternCounter] == text[iCounter]) {
                --jCounter; ++patternCounter;
            }
            if (jCounter == iCounter) {
                OUTPUT(iCounter);
            }
        }
        iCounter += hrBc[patternLength-1];
    }
}

```

Figure A- 5: The Horspool algorithm code

```

void preprocessing_qsBc(char *pattern, int patternLength, int qsBc[ASIZE]) {
    int iCounter, jCounter;
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        qsBc[iCounter] = -1;
    }
    for (jCounter = 0; jCounter < patternLength-1; ++jCounter){
        qsBc[jCounter] = patternLength - jCounter;
    }
}

void QuickSearch(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, jCounter, patternCounter, lastCharacter, qsBc[ASIZE];

    /* Preprocessing */
    preprocessing_qsBc(pattern, patternLength, qsBc);

    /* Searching */
    iCounter = 0;
    while (iCounter <= textLength - patternLength) {
        patternCounter = 0;
        jCounter = iCounter + patternLength - 1;
        while (jCounter >= iCounter && pattern[patternCounter] == text[iCounter]) {
            --jCounter;
            ++patternCounter;
        }
        if (jCounter < iCounter){
            OUTPUT(iCounter);
        }
        iCounter += qsBc [patternLength];
    }
}

```

Figure A- 6: The Quick-Search algorithm code

```

void preprocessing_brBc(char *pattern, int patternLength, int brBc[ASIZE][ASIZE]) {
    int iCounter, jCounter;
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        for (jCounter = 0; jCounter < ASIZE; ++jCounter){
            brBc[iCounter][jCounter] = patternLength+2;
        }
    }
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        brBc[iCounter][pattern[0]] = patternLength + 1;
    }
    for (iCounter = 0; iCounter < patternLength - 1; ++iCounter){
        brBc[pattern[iCounter]][pattern[iCounter+1]] = patternLength - iCounter;
    }
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        brBc[pattern[iCounter-1]][iCounter] = 1;
    }
}

void BerryRavindran(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, jCounter, brBc[ASIZE][ASIZE];

    /* Preprocessing */
    preprocessing_brBc(pattern, patternLength, brBc);

    /* Searching */
    iCounter = patternLength-1;
    while (iCounter <= textLength){
        jCounter= iCounter - patternLength + 1;
        patternCounter = patternLength-1;
        while (jCounter <= iCounter && pattern[patternCounter] == text[iCounter]) {
            --iCounter;
            --patternCounter;
        }
        if (iCounter < jCounter) {
            OUTPUT(iCounter);
        }
        iCounter = iCounter+brBc[patternLength][patternLength+1];
    }
}

```

Figure A- 7: The Berry-Ravindran algorithm code

```

int preprocessing_krHashing(char *pattern_OR_textWindow, int patternLength) {
    // This function compute the hashing value for either the pattern characters or the current text from
    text[start] ... text[end] and then return the hashing value
    int theHashingValue;
    return theHashingValue;
}

void KarpRabin(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, iCounter2, jCounter, jCounter2, patternCounter,
    int patternHashingValue = 0, textHashingValue = 0;
    char *textWindow;

    /* Preprocessing */
    patternHashingValue = preprocessing_krHashing(pattern, patternLength);

    /* Searching */
    iCounter = 0;
    while (iCounter <= textLength- patternLength) {
        patternCounter = 0;
        jCounter = iCounter + patternLength - 1;
        iCounter2=iCounter;
        jCounter2=jCounter;
        while (jCounter >= iCounter){
            textWindow += text[jCounter];
            ++iCounter;
        }
        textHashingValue= preprocessing_krHashing(textWindow, patternLength);
        if (patternHashingValue == textHashingValue){
            while (jCounter2 >= iCounter2 && pattern[patternCounter] == text[iCounter2]){
                --jCounter2;
                ++patternCounter;
            }
        }
        if (jCounter2 < iCounter2){
            OUTPUT(iCounter2);
        }
        iCounter ++;
    }
}

```

Figure A- 8: The Karp-Rabin algorithm code.

```

void preprocessing_SSTable(char *pattern, int patternLength, int ssTable[ASIZE]) {
    int iCounter, jCounter;
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        ssTable[iCounter] = pattern[iCounter];
    }
    for (jCounter = 0; jCounter < patternLength-1; ++jCounter){
        ssTable[jCounter] = patternLength - jCounter;
    }
}

void SkipShift(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, jCounter, startCounter, patternCounter, lastCharacter, ssTable[ASIZE];
    char currentChar;

    /* Preprocessing */
    preprocessing_SSTable(pattern, patternLength, ssTable);

    /* Searching */
    iCounter = 0;
    while (iCounter <= textLength - patternLength) {
        patternCounter = patternLength-1;
        startCounter = patternLength-1;
        jCounter = iCounter + patternLength - 1;
        currentChar = pattern[patternCounter];
        for (;patternCounter >= 0;--patternCounter){
            if(currentChar == ssTable[startCounter]){
                iCounter = ssTable[startCounter]+startCounter;
                jCounter = iCounter + patternLength-1;
            }
        }
        patternCounter = 0;
        while (jCounter >= iCounter && pattern[patternCounter] == text[iCounter]) {
            --jCounter;
            ++patternCounter;
        }
        if (jCounter < iCounter){
            OUTPUT(iCounter);
        }
        iCounter ++;
    }
}

```

Figure A- 9: The Skip Search algorithm code


```

void preprocessing_ASSTable(char *pattern, int patternLength, int ASSTable[ASIZE]) {
    int bCounter, iCounter, jCounter, tCounter, logPattern, trieLength;
    char *trieCharacters;
    logPattern = 0;
    bCounter = 0;
    tCounter = patternLength;
    while (tCounter > ASIZE) {
        ++logPattern;
        tCounter /= ASIZE;
    }
    if (logPattern == 0){
        logPattern = 1;
    }
    trieLength = 2 + (2*patternLength - logPattern + 1)*logPattern;
    trieCharacters = newTrie(trieLength, trieLength*ASIZE);
    for (iCounter = logPattern; iCounter < ASIZE; ++iCounter){
        ASSTable[iCounter] = trieCharacters[iCounter];
    }
    for (jCounter = logPattern; jCounter < patternLength-1; ++jCounter){
        ASSTable[jCounter] = patternLength - bCounter;
        ++bCounter;
    }
}

void AlphaSkipShift(char *pattern, int patternLength, char *text, int textLength, char *trieCharacters, int
trieLength) {
    int iCounter, jCounter, startCounter, patternCounter, lastCharacter, ASSTable[ASIZE];
    char *patternTrieCharacters;

    /* Preprocessing */
    preprocessing_ASSTable(pattern, patternLength, ASSTable);
    /* Searching */
    iCounter = 0;
    while (iCounter <= textLength - patternLength) {
        patternCounter = patternLength-1;
        startCounter = patternLength-1;
        jCounter = iCounter + patternLength - 1;
        patternTrieCharacters = newTrie(trieLength, trieLength*pattern);
        for (;patternCounter >= logPattern;--patternCounter){
            if(patternTrieCharacters == ASSTable[startCounter]){
                iCounter = ASSTable[startCounter]+startCounter;
                jCounter = iCounter + patternLength-1;
            }
        }
        patternCounter = 0;
        while (jCounter >= iCounter && pattern[patternCounter] == text[iCounter]) {
            --jCounter;
            ++patternCounter;
        }
        if (jCounter < iCounter){
            OUTPUT(iCounter);
        }
        iCounter ++;
    }
}

```

Figure A- 10: The Alpha Skip search algorithm code

```

void preprocessing_qsBc(char *pattern, int patternLength, int qsBc[ASIZE]) {
    int iCounter, jCounter;
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        qsBc[iCounter] = -1;
    }
    for (jCounter = 0; jCounter < patternLength-1; ++jCounter){
        qsBc[jCounter] = patternLength - jCounter;
    }
}

void SSABS(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, jCounter, patternCounter, lastCharacter, qsBc[ASIZE];

    /* Preprocessing */
    preprocessing_qsBc(pattern, patternLength, qsBc);

    /* Searching */
    iCounter = 0;
    while (iCounter <= textLength - patternLength) {
        patternCounter = patternLength - 1;
        jCounter = iCounter + patternLength - 1;
        lastCharacter = iCounter + patternLength - 1;
        if (pattern[jCounter] == text[lastCharacter]){
            patternCounter = 0;
            while (jCounter > iCounter && pattern[patternCounter] == text[iCounter]) {
                --jCounter; ++patternCounter;
            }
            if (jCounter == iCounter) {
                OUTPUT(iCounter);
            }
        }
        iCounter += qsBc [patternLength-1];
    }
}

```

Figure A- 11: The SSABS algorithm code

```

void preprocessing_KMPBc(char *pattern, int patternLength, int KMPBc[ASIZE]){
    int iCounter,jCounter;
    KMPBc[0]=-1;
    for (iCounter = 1; iCounter < patternLength; iCounter++){
        jCounter = KMPBc[iCounter-1];
        while (jCounter>=0){
            if(pattern[jCounter] ==pattern[iCounter-1]){
                break;
            }
            else{
                jCounter=KMPBc[jCounter];
            }
            KMPBc[iCounter] = jCounter + 1;
        }
    }
}

void preprocessing_qsBc(char *pattern, int patternLength, int qsBc[ASIZE]) {
    int iCounter, jCounter;
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        qsBc[iCounter] = -1;
    }
    for (jCounter = 0; jCounter < patternLength-1; ++jCounter){
        qsBc[jCounter] = patternLength - jCounter;
    }
}

void FJS(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, jCounter, patternCounter, lastCharacter, qsBc[ASIZE], KMPBc[ASIZE];

    /* Preprocessing */
    preprocessing_qsBc(pattern, patternLength, qsBc);
    preprocessing_KMPBc(pattern, patternLength, KMPBc);

    /* Searching */
    iCounter = 0;
    while (iCounter <= textLength - patternLength) {
        patternCounter = 0;
        jCounter = iCounter + patternLength - 1;
        while (jCounter >= iCounter && pattern[patternCounter] == text[iCounter]) {
            --jCounter;
            ++patternCounter;
        }
        if (jCounter < iCounter){
            OUTPUT(iCounter);
        }
        if(pattern[patternCounter] != text[iCounter]){
            iCounter += qsBc [patternLength];
        }
        else{
            iCounter += KMPBc [patternCounter];
        }
    }
}

```

Figure A- 12: The FJS algorithm code

```

void preprocessing_brBc(char *pattern, int patternLength, int brBc[ASIZE][ASIZE]) {
    int iCounter, jCounter;
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        for (jCounter = 0; jCounter < ASIZE; ++jCounter){
            brBc[iCounter][jCounter] = patternLength+2;
        }
    }
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        brBc[iCounter][pattern[0]] = patternLength + 1;
    }
    for (iCounter = 0; iCounter < patternLength - 1; ++iCounter){
        brBc[pattern[iCounter]][pattern[iCounter+1]] = patternLength - iCounter;
    }
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        brBc[pattern[iCounter-1]][iCounter] = 1;
    }
}

void TVSBS(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, jCounter, patternCounter, lastCharacter, brBc[ASIZE][ASIZE];

    /* Preprocessing */
    preprocessing_brBc(pattern, patternLength, brBc);

    /* Searching */
    iCounter = 0;
    while (iCounter <= textLength - patternLength) {
        patternCounter = patternLength - 1;
        jCounter = iCounter + patternLength - 1;
        lastCharacter = iCounter + patternLength - 1;
        if (pattern[jCounter] == text[lastCharacter]){
            patternCounter = 0;
            while (jCounter >= iCounter && pattern[patternCounter] == text[iCounter]) {
                --jCounter; ++patternCounter;
            }
            if (jCounter < iCounter) {
                OUTPUT(iCounter);
            }
        }
        iCounter += brBc [patternLength][ patternLength+1];
    }
}

```

Figure A- 13: The TVSBS algorithm code

```

void preprocessing_ztBc(char *pattern, int patternLength, int ztBc[ASIZE][ASIZE]) {
    int iCounter, jCounter;
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        for (jCounter = 0; jCounter < ASIZE; ++jCounter){
            ztBc[iCounter][jCounter] = patternLength;
        }
    }
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        ztBc[iCounter][pattern[0]] = patternLength - 1;
    }
    for (iCounter = 1; iCounter < patternLength - 1; ++iCounter){
        ztBc[pattern[iCounter - 1]][pattern[iCounter]] = patternLength - 1 - iCounter;
    }
}

void ZTBMH(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, jCounter, patternCounter, lastCharacter, ztBc[ASIZE][ASIZE];

    /* Preprocessing */
    Preprocessing_ztBc(pattern, patternLength, ztBc);

    /* Searching */
    iCounter = 0;
    while (iCounter <= textLength - patternLength) {
        patternCounter = patternLength - 1;
        jCounter = iCounter + patternLength - 1;
        lastCharacter = iCounter + patternLength - 1;
        if (pattern[jCounter] == text[lastCharacter]){
            patternCounter = 0;
            while (jCounter > iCounter && pattern[patternCounter] == text[iCounter]) {
                --jCounter; ++patternCounter;
            }
            if (jCounter == iCounter) {
                OUTPUT(iCounter);
            }
        }
        iCounter += ztBc [patternLength-2][ patternLength-1];
    }
}

```

Figure A- 14: The ZTBMH algorithm code

```

void preprocessing_fsGs(char *pattern, int patternLength, int fsGs[XSIZE]) {
    // Same preprocessing code as the bmGs function
}

void preprocessing_brBc(char *pattern, int patternLength, int brBc[ASIZE][ASIZE]) {
    int iCounter, jCounter;
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        for (jCounter = 0; jCounter < ASIZE; ++jCounter){
            brBc[iCounter][jCounter] = patternLength+2;
        }
    }
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        brBc[iCounter][pattern[0]] = patternLength + 1;
    }
    for (iCounter = 0; iCounter < patternLength - 1; ++iCounter){
        brBc[pattern[iCounter]][pattern[iCounter+1]] = patternLength - iCounter;
    }
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        brBc[pattern[iCounter-1]][iCounter] = 1;
    }
}

void BRFS(unsigned char *pattern, int patternLength, unsigned char *text, int textLength) {
    int iCounter, jCounter, fsGs[XSIZE], brBc[ASIZE][ASIZE];
    char *lastTextCharacter;
    char *lastPatternCharacter = pattern[patternLength - 1];

    /* Preprocessing */
    preprocessing_fsGs(pattern, patternLength, fsGs);
    preprocessing_brBc(pattern, patternLength, brBc);

    /* Searching */
    iCounter = 0;
    while (iCounter <= textLength - patternLength){
        if (lastPatternCharacter != lastTextCharacter){
            iCounter = iCounter + brBc[pattern[iCounter + patternLength]] [pattern[iCounter + patternLength+1]];
        }
        jCounter = patternLength - 2;
        while (jCounter >= 0 && pattern[jCounter] == text[iCounter + jCounter]){
            jCounter = jCounter - 1;
        }
        if (jCounter < 0){
            OUTPUT(iCounter);
            iCounter = iCounter + brBc[pattern[iCounter + patternLength]] [pattern[iCounter + patternLength+1]];
        }
        else{
            iCounter += fsGs[jCounter + 1];
        }
    }
}

```

Figure A- 15: The BRFS algorithm code

```

void preprocessing_KMPBc(char *pattern, int patternLength, int KMPBc[ASIZE]){
    int iCounter,jCounter;
    KMPBc[0]=-1;
    for (iCounter = 1; iCounter < patternLength; iCounter++){
        jCounter = KMPBc[iCounter-1];
        while (jCounter>=0)
        {
            if(pattern[jCounter] ==pattern[iCounter-1]){
                break;
            }
            else{
                jCounter=KMPBc[jCounter];
            }
            KMPBc[iCounter] = jCounter + 1;
        }
    }
}

void preprocessing_bmGs(char *pattern, int patternLength, int bmGs[XSIZE]) {
    int iCounter= patternLength, jCounter= patternLength + 1;
    bmGs[iCounter] = jCounter;
    while (iCounter > 0) {
        while (jCounter <= patternLength && pattern[iCounter - 1] != pattern[jCounter - 1]){
            if (bmGs [jCounter] == 0) {
                bmGs [jCounter] = jCounter - iCounter;
            }
            jCounter = bmGs[jCounter];
        }
        iCounter--, jCounter--;
        bmGs[iCounter] = jCounter;
    }
}

void BM-KMB(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, jCounter, patternCounter, lastCharacter, bmGs[ASIZE], KMPBc[ASIZE];
    /* Preprocessing */
    preprocessing_bmGs(pattern, patternLength, bmGs);
    preprocessing_KMPBc(pattern, patternLength, KMPBc);
    /* Searching */
    iCounter = 0;
    while (iCounter <= textLength - patternLength) {
        patternCounter = 0;
        jCounter = iCounter + patternLength - 1;
        if(pattern[patternCounter + patternLength-1 ] != text[jCounter]){
            iCounter += bmGs[jCounter];
        }
        else{
            while (jCounter-1 >= iCounter && pattern[patternCounter] == text[iCounter]) {
                --jCounter;
                ++patternCounter;
            }
            if (jCounter-1 < iCounter){
                OUTPUT(iCounter);
            }
            iCounter += KMPBc [patternCounter];
        }
    }
}

```

Figure A- 16: The BM-KMB algorithm code

```

void preprocessing_SSTable(char *pattern, int patternLength, int ssTable[ASIZE]) {
    int iCounter, jCounter;
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        ssTable[iCounter] = pattern[iCounter];
    }
    for (jCounter = 0; jCounter < patternLength-1; ++jCounter){
        ssTable[jCounter] = patternLength - jCounter;
    }
}

void preprocessing_brBc(char *pattern, int patternLength, int brBc[ASIZE][ASIZE]) {
    int iCounter, jCounter;
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        for (jCounter = 0; jCounter < ASIZE; ++jCounter){
            brBc[iCounter][jCounter] = patternLength+2;
        }
    }
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        brBc[iCounter][pattern[0]] = patternLength + 1;
    }
    for (iCounter = 0; iCounter < patternLength - 1; ++iCounter){
        brBc[pattern[iCounter]][pattern[iCounter+1]] = patternLength - iCounter;
    }
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        brBc[pattern[iCounter-1]][iCounter] = 1;
    }
}

void BRSS(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, jCounter, startCounter, patternCounter, lastCharacter, ssTable[ASIZE], brBc[ASIZE][ASIZE];
    char currentChar;
    /* Preprocessing */
    preprocessing_SSTable(pattern, patternLength, ssTable);
    preprocessing_brBc(pattern, patternLength, brBc);

    /* Searching */
    iCounter = 0;
    iCounter = 0;
    while (iCounter <= textLength - patternLength) {
        patternCounter = patternLength-1;
        startCounter = patternLength-1;
        jCounter = iCounter + patternLength - 1;
        currentChar = pattern[patternCounter];
        for (; patternCounter >= 0; --patternCounter){
            if(currentChar == ssTable[startCounter]){
                iCounter = ssTable[startCounter]+startCounter;
                jCounter = iCounter + patternLength-1;
            }
        }
        patternCounter = 0;
        while (jCounter >= iCounter && pattern[patternCounter] == text[iCounter]) {
            --jCounter;
            ++patternCounter;
        }
        if (jCounter < iCounter){
            OUTPUT(iCounter);
        }
        iCounter += MAX(ssTable[text[iCounter + patternLength]], brBc[text[iCounter + patternLength]] [text[iCounter
+ patternLength + 1]]);
    }
}

```

Figure A- 17: The BRSS algorithm code


```

void preprocessing_ASSTable(char *pattern, int patternLength, int ASSTable[ASIZE]) {
    int bCounter, iCounter, jCounter, tCounter, logPattern, trieLength;
    char *trieCharacters;
    logPattern = 0;
    bCounter = 0;
    tCounter = patternLength;
    while (tCounter > ASIZE) {
        ++logPattern;
        tCounter /= ASIZE;
    }
    if (logPattern == 0) {
        logPattern = 1;
    }
    trieLength = 2 + (2*patternLength - logPattern + 1)*logPattern;
    trieCharacters = newTrie(trieLength, trieLength*ASIZE);
    for (iCounter = logPattern; iCounter < ASIZE; ++iCounter){
        ASSTable[iCounter] = trieCharacters[iCounter];
    }
    for (jCounter = logPattern; jCounter < patternLength-1; ++jCounter){
        ASSTable[jCounter] = patternLength - bCounter;
        ++bCounter;
    }
}

void preprocessing_brBc(char *pattern, int patternLength, int brBc[ASIZE][ASIZE]) {
    int iCounter, jCounter;
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        for (jCounter = 0; jCounter < ASIZE; ++jCounter){
            brBc[iCounter][jCounter] = patternLength+2;
        }
    }
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        brBc[iCounter][pattern[0]] = patternLength + 1;
    }
    for (iCounter = 0; iCounter < patternLength - 1; ++iCounter){
        brBc[pattern[iCounter]][pattern[iCounter+1]] = patternLength - iCounter;
    }
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        brBc[pattern[iCounter-1]][iCounter] = 1;
    }
}

void ASSBR(char *pattern, int patternLength, char *text, int textLength, char *trieCharacters, int trieLength) {
    int iCounter, jCounter, startCounter, patternCounter, lastCharacter, ASSTable[ASIZE], brBc[ASIZE][ASIZE];
    char *patternTrieCharacters;

    /* Preprocessing */
    preprocessing_ASSTable(pattern, patternLength, ASSTable);
    preprocessing_brBc(pattern, patternLength, brBc);

    /* Searching */
    iCounter = 0;
    while (iCounter <= textLength - patternLength) {
        patternCounter = patternLength-1;
        startCounter = patternLength-1;
        jCounter = iCounter + patternLength - 1;
        patternTrieCharacters = newTrie(trieLength, trieLength*pattern);
        for (; patternCounter >= logPattern; --patternCounter){
            if(patternTrieCharacters == ASSTable[startCounter]){

```

```

        iCounter = ASSTable[startCounter]+startCounter;
        jCounter = iCounter + patternLength-1;
    }

}
patternCounter = 0;
while (jCounter >= iCounter && pattern[patternCounter] == text[iCounter]) {
    --jCounter;
    ++patternCounter;
}
if (jCounter < iCounter){
    OUTPUT(iCounter);
}
iCounter += brBc[text[iCounter + patternLength]] [text[iCounter + patternLength + 1]]);
}
}

```

Figure A- 18: The ASSBR algorithm code

```

void preprocessing_mrcaTable(char *pattern, int patternLength, int *ref1, int *ref2) {
    int iCounter, jCounter;
    char currentcharacter;
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        ref1[iCounter] = 0;
        ref2[iCounter] = 2 * patternLength;
    }
    for (jCounter = 0; jCounter < patternLength; ++jCounter){
        currentcharacter = pattern[jCounter];
        ref1[currentcharacter] = jCounter + 1;
        ref2[currentcharacter] = 2 * patternLength - jCounter - 1;
    }
}

void MRCA(char *pattern, int patternLength, char *text, int textLength, int *ref1, int *ref2) {
    int iCounter, jCounter, startCounter, patternCounter, patternCounterNew, lastCharacter, mrcaTable[ASIZE];
    int ref1_CharPos, ref2_CharPos;

    /* Preprocessing */
    preprocessing_mrcaTable(pattern, patternLength, ref1, ref2);

    /* Searching */
    iCounter = 0;
    mrcaTable[pattern[0]]=1;
    lastCharacter =patternLength;
    patternCounter = 0;

    while (lastCharacter <= textLength - patternLength) {
        if(mrcaTable[lastCharacter - patternLength + patternCounterNew] == pattern[patternCounter];
        {
            for (iCounter = 0; patternCounter = patternLength - 1; --patternCounter){
                if(text[lastCharacter - ++iCounter] != pattern[patternCounter])
                {
                    patternCounterNew = patternCounter;
                    goto next;
                }
            }
        }

        next:
        ref1_CharPos = ref1[text[lastCharacter]];

        if(!ref1_CharPos){
            lastCharacter = ref1 + patternLength;
            ref2_CharPos = ref1[text[lastCharacter]];
        }
        else
        {
            ref1 = ref1 + patternLength - ref1_CharPos;
            lastCharacter += ref2[text[ref1]]- ref1_CharPos + 1;
        }
    }
}

```

Figure A- 19: The MRCA algorithm code

```

int preprocessing_krHashing(char *pattern_OR_textWindow, int patternLength) {
    // This function compute the hashing value for either the pattern characters or the current text from
    text[start] ... text[end] and then return the hashing value
    int theHashingValue;
    return theHashingValue;
}

void preprocessing_hrBc(char *pattern, int patternLength, int hrBc[ASIZE]) {
    // Same preprocessing code as the bmBc function
    int iCounter, jCounter;
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        hrBc[iCounter] = -1;
    }
    for (jCounter = 0; jCounter < patternLength; ++jCounter){
        iCounter=pattern[jCounter];
        hrBc[iCounter] = jCounter;
    }
}

void KRBMH(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, iCounter2, jCounter, jCounter2, patternCounter, hrBc[ASIZE];
    int patternHashingValue = 0, textHashingValue = 0;
    char *textWindow;
    /* Preprocessing */
    patternHashingValue = preprocessing_krHashing(pattern, patternLength);
    preprocessing_hrBc(pattern, patternLength, hrBc);

    /* Searching */
    iCounter = 0;
    while (iCounter <= textLength- patternLength) {
        patternCounter = 0;
        jCounter = iCounter + patternLength - 1;
        iCounter2=iCounter;
        jCounter2=jCounter;
        while (jCounter >= iCounter){
            textWindow += text[jCounter];
            ++iCounter;
        }
        textHashingValue= preprocessing_krHashing(textWindow, patternLength);
        if (patternHashingValue == textHashingValue){
            while (jCounter2 >= iCounter2 && pattern[patternCounter] == text[iCounter2]){
                --jCounter2;
                ++patternCounter;
            }
        }
        if (jCounter2 < iCounter2){
            OUTPUT(iCounter2);
        }
        iCounter += hrBc [patternLength-1];
    }
}

```

Figure A- 20: The KRMBH algorithm code

```

void preprocessing_SSTable(char *pattern, int patternLength, int ssTable[ASIZE]) {
    int iCounter, jCounter;
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        ssTable[iCounter] = pattern[iCounter];
    }
    for (jCounter = 0; jCounter < patternLength-1; ++jCounter){
        ssTable[jCounter] = patternLength - jCounter;
    }
}

void preprocessing_qsBc(char *pattern, int patternLength, int qsBc[ASIZE]) {
    int iCounter, jCounter;
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        qsBc[iCounter] = -1;
    }
    for (jCounter = 0; jCounter < patternLength-1; ++jCounter){
        qsBc[jCounter] = patternLength - jCounter;
    }
}

void QuickSkipSearch(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, jCounter, startCounter, patternCounter, lastCharacter, ssTable[ASIZE], qsBc[ASIZE];
    char currentChar;

    /* Preprocessing */
    preprocessing_SSTable(pattern, patternLength, ssTable);
    preprocessing_qsBc(pattern, patternLength, qsBc);

    /* Searching */
    iCounter = 0;
    while (iCounter <= textLength - patternLength) {
        patternCounter = patternLength-1;
        startCounter = patternLength-1;
        jCounter = iCounter + patternLength - 1;
        currentChar = pattern[patternCounter];
        for (;patternCounter >= 0;--patternCounter){
            if(currentChar == ssTable[startCounter]){
                iCounter = ssTable[startCounter]+startCounter;
                jCounter = iCounter + patternLength-1;
            }
        }
        patternCounter = 0;
        while (jCounter >= iCounter && pattern[patternCounter] == text[iCounter]) {
            --jCounter;
            ++patternCounter;
        }
        if (jCounter < iCounter){
            OUTPUT(iCounter);
        }
        iCounter += MAX(ssTable[text[iCounter + patternLength]], qsBc[text[iCounter + patternLength]]);
    }
}

```

Figure A- 21: The QSS algorithm code

```

int preprocessing_krHashing(char *pattern_OR_textWindow, int patternLength) {
    // This function compute the hashing value for either the pattern characters or the current text from text[start] ...
    text[end] and then return the hashing value
    int theHashingValue;
    return theHashingValue;
}

void preprocessing_qsBc(char *pattern, int patternLength, int qsBc[ASIZE]) {
    int iCounter, jCounter;
    for (iCounter = 0; iCounter < ASIZE; ++iCounter){
        qsBc[iCounter] = -1;
    }
    for (jCounter = 0; jCounter < patternLength-1; ++jCounter){
        qsBc[jCounter] = patternLength - jCounter;
    }
}

void AKRAM(char *pattern, int patternLength, char *text, int textLength) {
    int iCounter, iCounter2, jCounter, jCounter2, tCounter, patternCounter, qsBc[ASIZE];
    int patternPrefixHashingValue = 0, patternSuffixHashingValue = 0;
    int textPrefixHashingValue = 0, textSuffixHashingValue = 0;
    int prefixLength, suffixLength;
    char *patternPrefix, *patternSuffix, *textWindow, *textPrefix, *textSuffix;
    prefixLength = (int)patternLength/2;
    suffixLength = patternLength - prefixLength;
    for (iCounter = 0; iCounter < prefixLength; iCounter++){
        patternPrefix += pattern[iCounter];
    }
    for (iCounter = prefixLength; iCounter < patternLength; iCounter++){
        patternSuffix += pattern[iCounter];
    }

    /* Preprocessing */
    //pattern hashing here while the text hasing in searching phase
    patternPrefixHashingValue = preprocessing_krHashing(patternPrefix, prefixLength);
    patternSuffixHashingValue = preprocessing_krHashing(patternSuffix, suffixLength);
    preprocessing_qsBc(pattern, patternLength, qsBc);

    /* Searching */
    iCounter = 0;
    while (iCounter <= textLength- patternLength) {
        patternCounter = 0;
        jCounter = iCounter + patternLength - 1;
        iCounter2=iCounter;
        jCounter2=jCounter;
        while (jCounter >= iCounter){
            textWindow += text[jCounter];
            ++iCounter;
        }
        for (tCounter = 0; tCounter < prefixLength + iCounter2; tCounter++){
            textPrefix += text[tCounter];
        }
        for (tCounter = prefixLength + iCounter2; tCounter < jCounter2; tCounter++){
            textSuffix += text[tCounter];
        }
        textPrefixHashingValue= preprocessing_krHashing(textPrefix, prefixLength);
        textSuffixHashingValue= preprocessing_krHashing(textSuffix, suffixLength);
        if (patternPrefixHashingValue == textPrefixHashingValue && patternSuffixHashingValue ==
        textSuffixHashingValue){
            while (jCounter2 >= iCounter2 && pattern[patternCounter] == text[iCounter2]){

```

```
        --jCounter2;  
        ++patternCounter;  
    }  
}  
if (jCounter2 < iCounter2){  
    OUTPUT(iCounter2);  
}  
iCounter += qsBc [patternLength];  
}  
}
```

Figure A- 22: The AKRAM algorithm code

APPENDIX B: SMILES EBNF

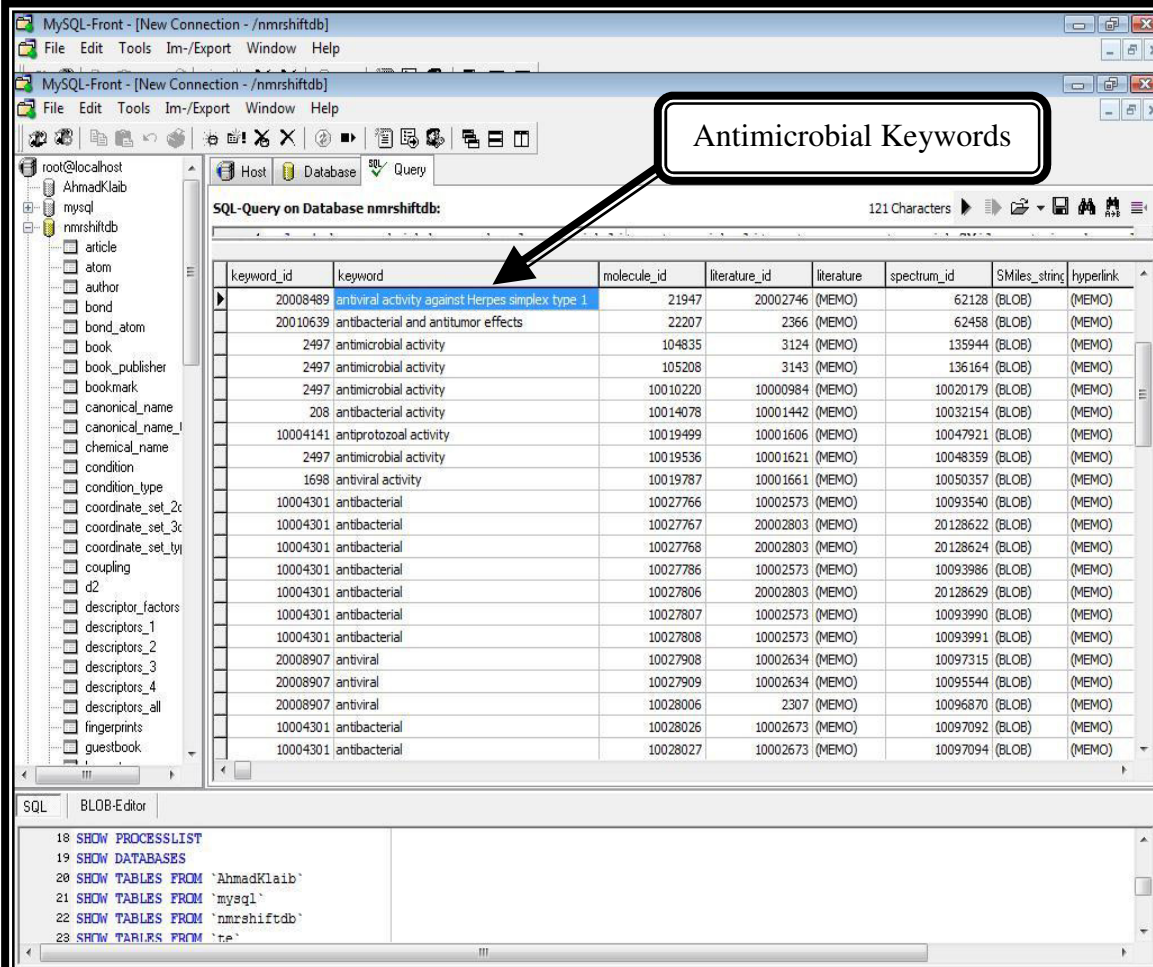
The complete EBNF of the SMILES is listed in this appendix.

```
smiles          ::= chain
chain           ::= branched_atom
                  | branched_atom chain
                  | branched_atom bond chain
                  | branched_atom DOT chain
bond            ::= BOND_1
                  | BOND_2
                  | BOND_3
                  | BOND_4
                  | BOND_ARO
branched_atom   ::= atom kleene_ringbond kleene_branch kleene_ringbond
kleene_ringbond ::= kleene_ringbond ringbond
                  | ringbond
                  | ""
kleene_branch   ::= kleene_branch branch
                  | branch
                  | ""
ringbond        ::= RINGBOND
                  | DIGIT
branch          ::= LPAREN chain RPAREN
                  | LPAREN bond chain RPAREN
                  | LPAREN DOT chain RPAREN
atom            ::= bracket_atom
                  | aliphatic_organic
                  | aromatic_organic
                  | WILDCARD
bracket_atom    ::= LBRACKET isotope symbol chiral hcount charge class RBRACKET
aliphatic_organic ::= ELEMENT
aromatic_organic ::= AROMATIC
isotope         ::= number
```


symbol	:: =	ELEMENT AROMATIC WILDCARD
chiral	:: =	CHIRAL CHIRAL CHIRAL CHIRAL CHIRAL_CODE ""
hcount	:: =	ELEMENT ELEMENT DIGIT ""
charge	:: =	CHARGE_MINUS CHARGE_MINUS DIGIT CHARGE_PLUS CHARGE_PLUS DIGIT DEPR_MIN DEPR_PLUS ""
class	:: =	CLASS_COLON NUMBER ""
number	:: =	number DIGIT DIGIT

APPENDIX C: TOOLKIT AND PARALLEL MODELS

IMPLEMENTATION



Antimicrobial Keywords

keyword_id	keyword	molecule_id	literature_id	literature	spectrum_id	SMiles_string	hyperlink
20008489	antiviral activity against Herpes simplex type 1	21947	20002746	(MEMO)	62128	(BLOB)	(MEMO)
20010639	antibacterial and antitumor effects	22207	2366	(MEMO)	62458	(BLOB)	(MEMO)
2497	antimicrobial activity	104835	3124	(MEMO)	135944	(BLOB)	(MEMO)
2497	antimicrobial activity	105208	3143	(MEMO)	136164	(BLOB)	(MEMO)
2497	antimicrobial activity	10010220	10000984	(MEMO)	10020179	(BLOB)	(MEMO)
208	antibacterial activity	10014078	10001442	(MEMO)	10032154	(BLOB)	(MEMO)
10004141	antiprotozoal activity	10019499	10001606	(MEMO)	10047921	(BLOB)	(MEMO)
2497	antimicrobial activity	10019536	10001621	(MEMO)	10048359	(BLOB)	(MEMO)
1698	antiviral activity	10019787	10001661	(MEMO)	10050357	(BLOB)	(MEMO)
10004301	antibacterial	10027766	10002573	(MEMO)	10093540	(BLOB)	(MEMO)
10004301	antibacterial	10027767	20002803	(MEMO)	20128622	(BLOB)	(MEMO)
10004301	antibacterial	10027768	20002803	(MEMO)	20128624	(BLOB)	(MEMO)
10004301	antibacterial	10027786	10002573	(MEMO)	10093986	(BLOB)	(MEMO)
10004301	antibacterial	10027806	20002803	(MEMO)	20128629	(BLOB)	(MEMO)
10004301	antibacterial	10027807	10002573	(MEMO)	10093990	(BLOB)	(MEMO)
10004301	antibacterial	10027808	10002573	(MEMO)	10093991	(BLOB)	(MEMO)
20008907	antiviral	10027908	10002634	(MEMO)	10097315	(BLOB)	(MEMO)
20008907	antiviral	10027909	10002634	(MEMO)	10095544	(BLOB)	(MEMO)
20008907	antiviral	10028006	2307	(MEMO)	10096870	(BLOB)	(MEMO)
10004301	antibacterial	10028026	10002673	(MEMO)	10097092	(BLOB)	(MEMO)
10004301	antibacterial	10028027	10002673	(MEMO)	10097094	(BLOB)	(MEMO)

```

18 SHOW PROCESSLIST
19 SHOW DATABASES
20 SHOW TABLES FROM 'AhmadKlaib'
21 SHOW TABLES FROM 'mysql'
22 SHOW TABLES FROM 'nmrshiftdb'
23 SHOW TABLES FROM 're'
  
```

Figure C- 1: A sample of extracted Antimicrobial structures

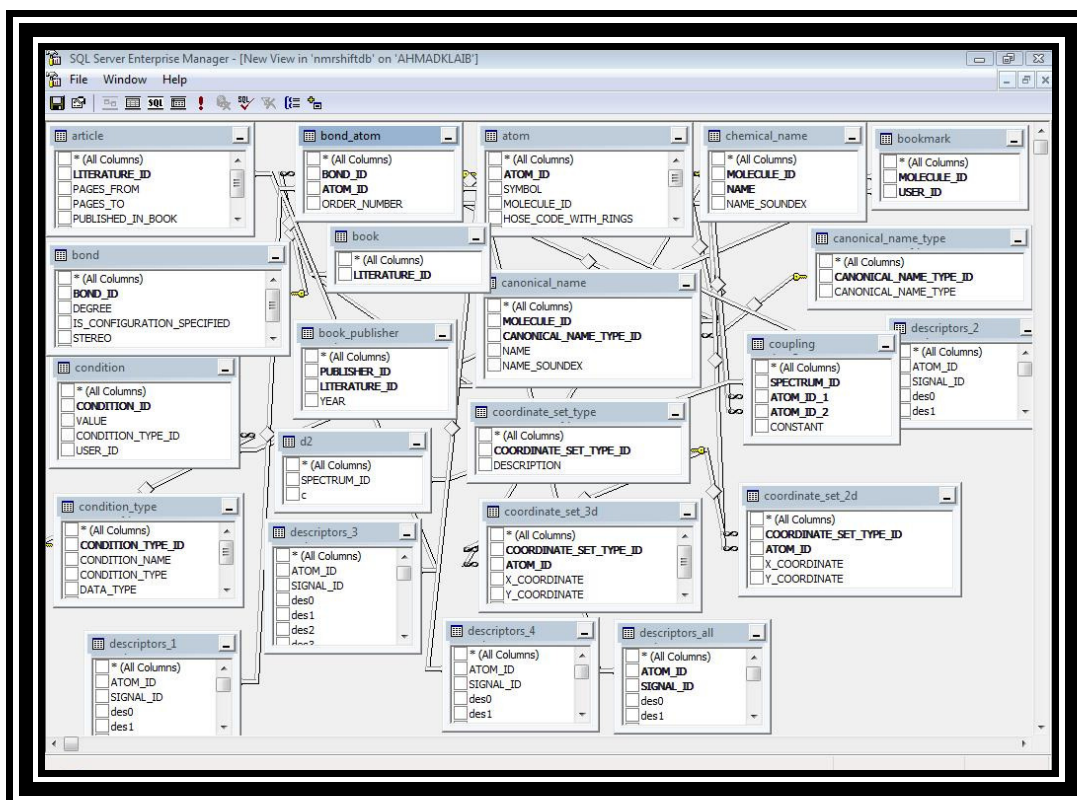


Figure C- 2: The local database schema (1)

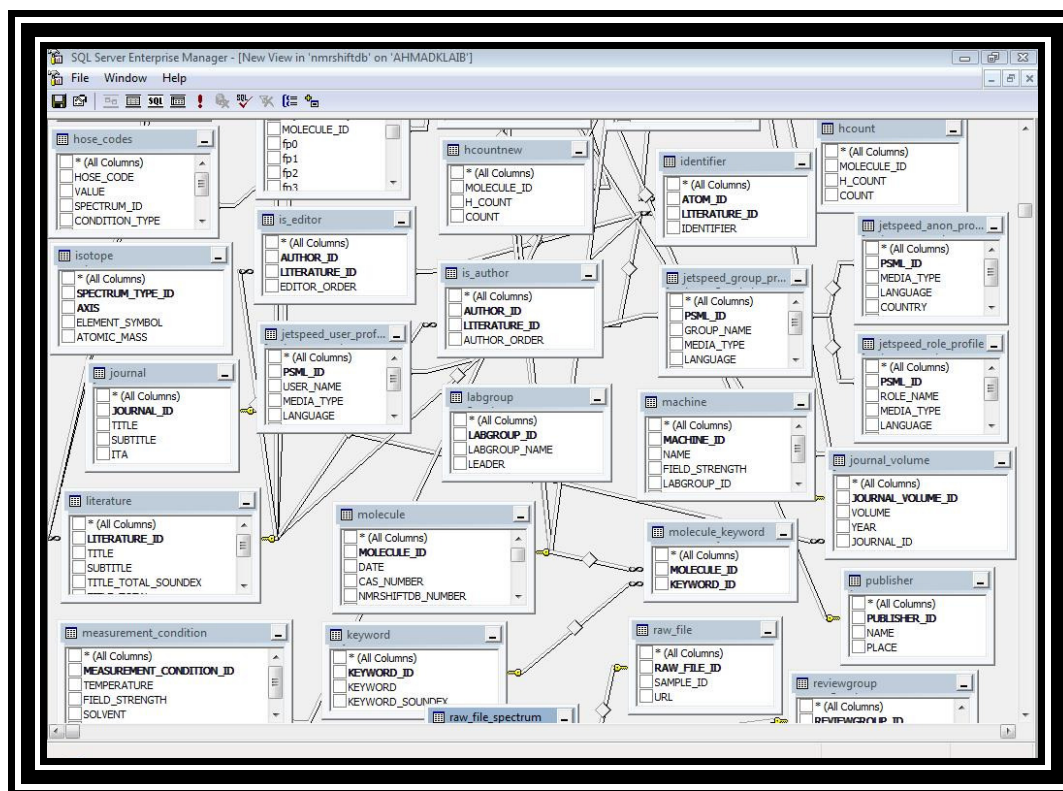


Figure C- 3: The local database schema (2)

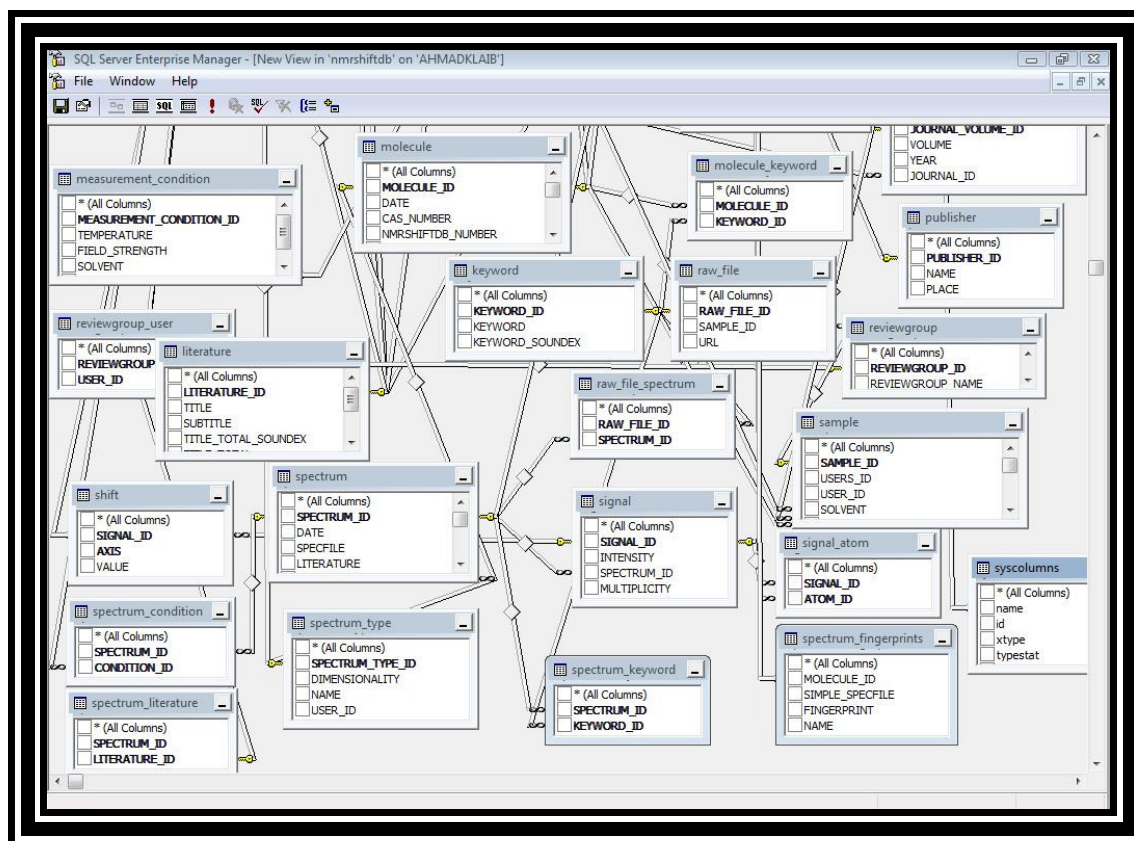


Figure C- 4: The local database schema (3)

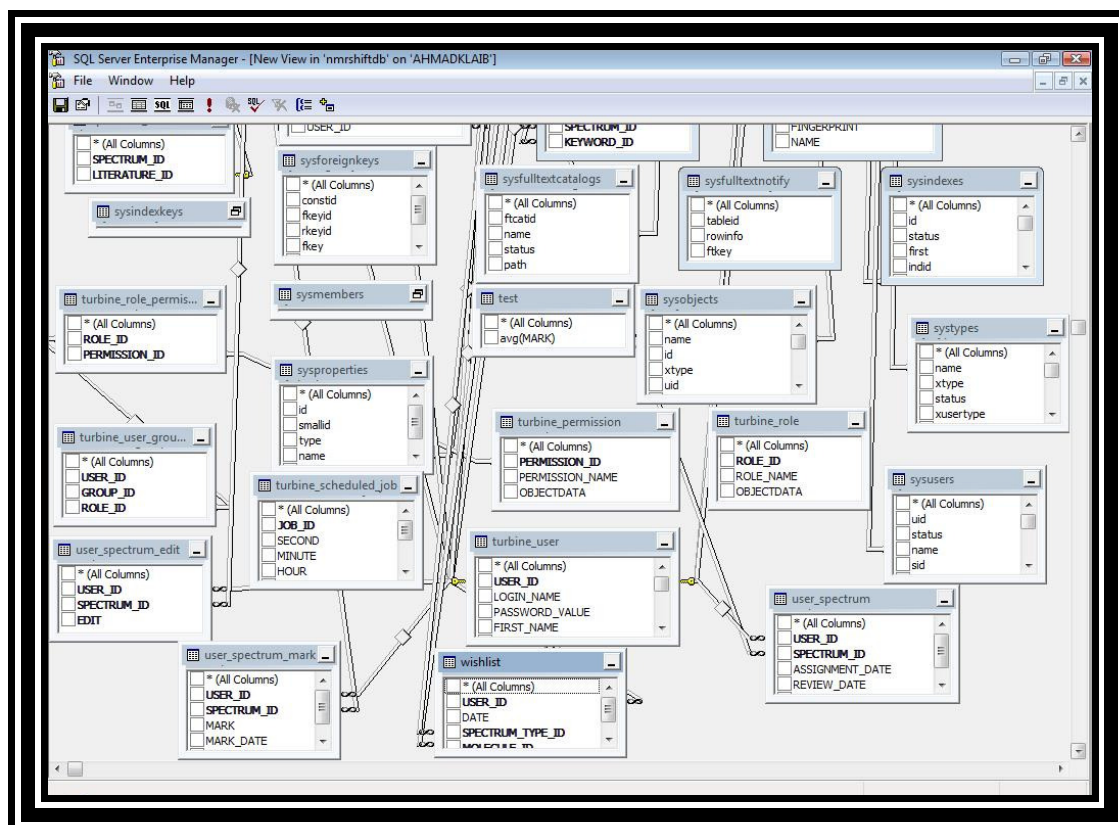


Figure C- 5: The local database schema (4)

The image shows the login page of the 'CHEMICAL STRUCTURES SEARCHING TOOL'. The header has a blue background with a molecular structure graphic on the left and the text 'CHEMICAL STRUCTURES SEARCHING TOOL' and 'UNIVERSITY OF HUDDERSFIELD AHMAD KLAIB AND HUGH OSBORNE' on the right. Below the header, there is a 'Welcome' message in a blue box. The main content area contains a login form with two input fields: 'User id' and 'Password'. Below these fields are two buttons: 'Submit' and 'Reset'. The footer of the page states 'Copyright 2014 @ University of Huddersfield'.

Figure C- 6: Developed toolkit login's page

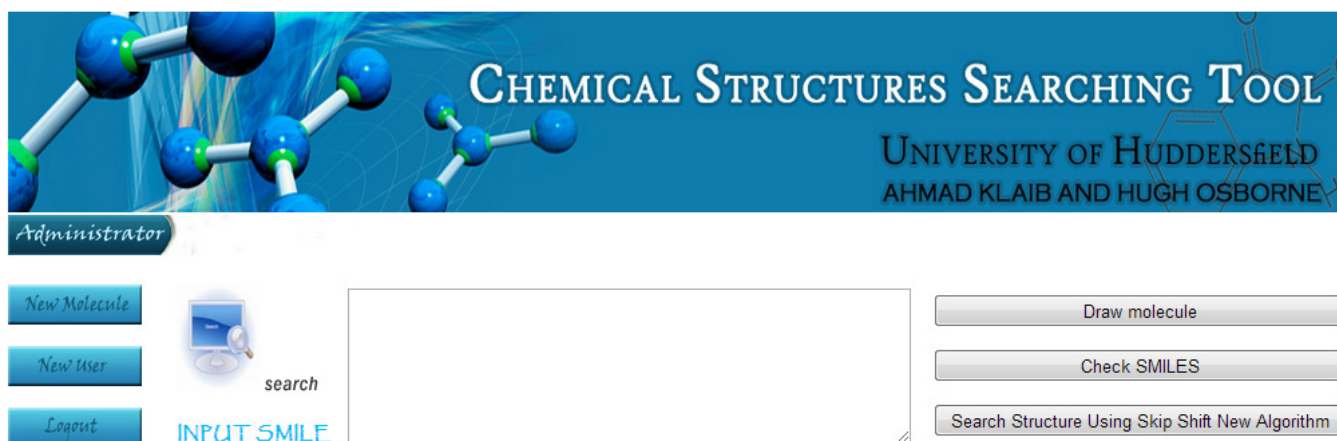


Figure C- 7: Developed chemical structure toolkit

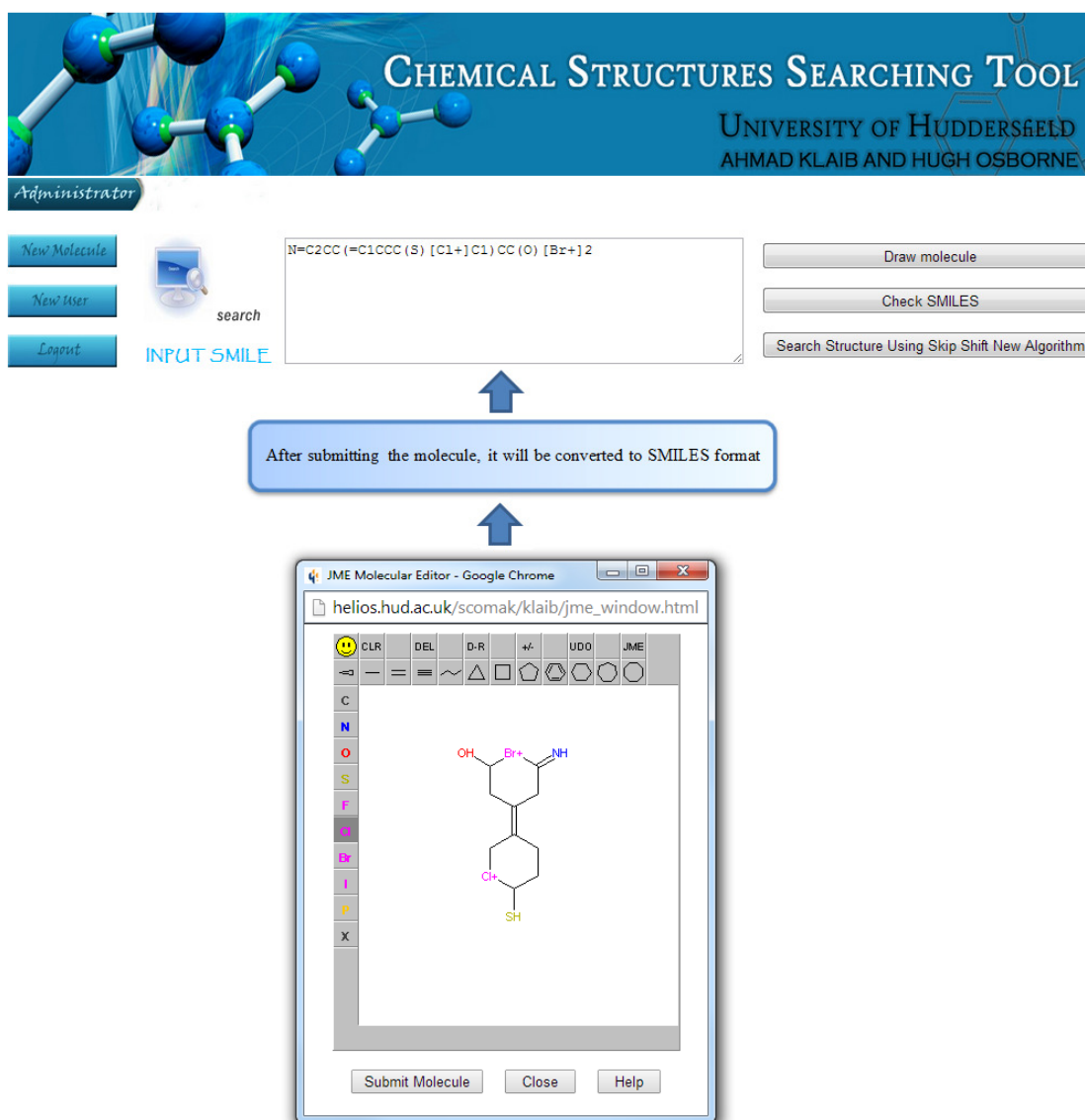


Figure C- 8: Converting chemical structure into SMILES

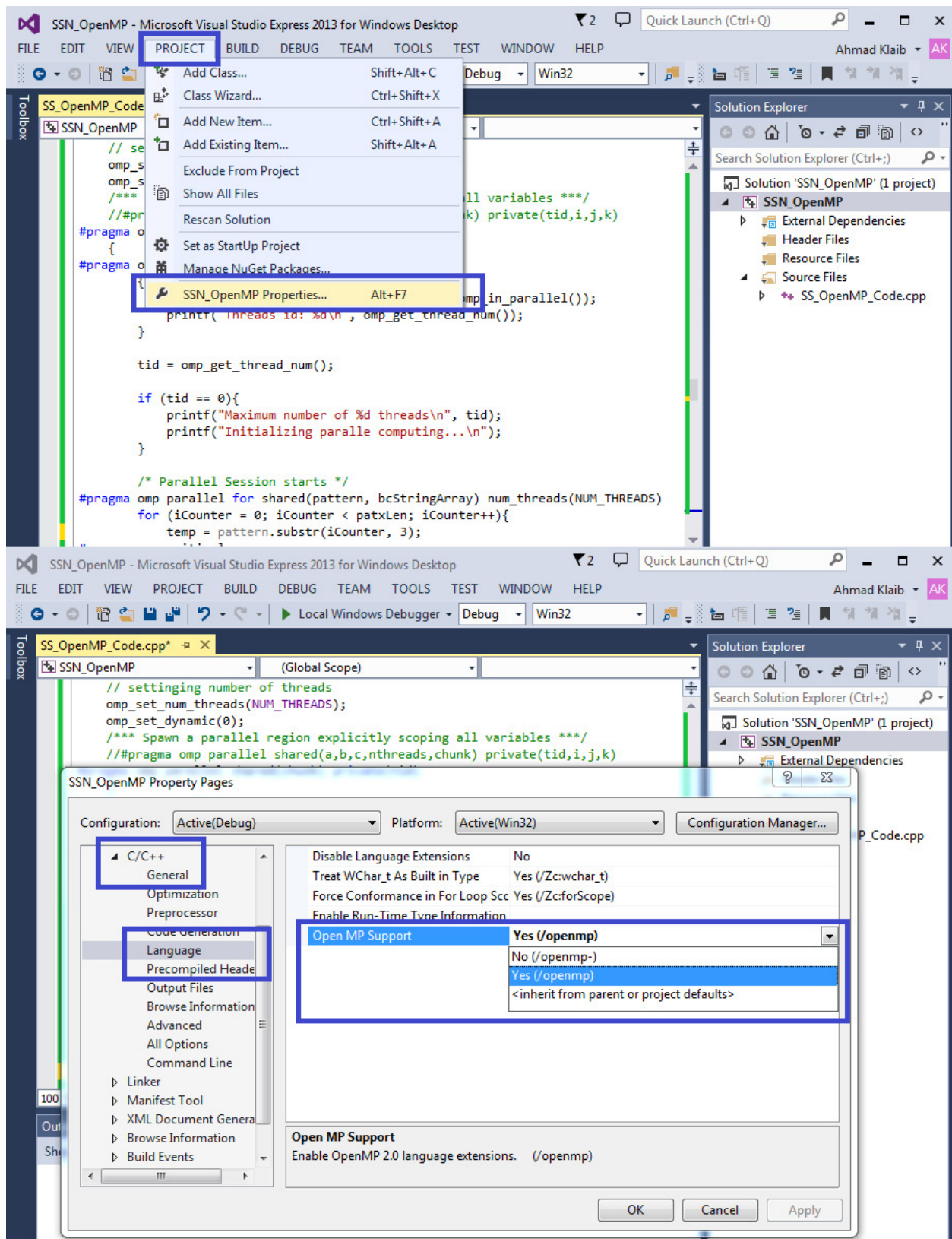


Figure C- 10: Enabling OpenMP Model in Microsoft Visual Studio

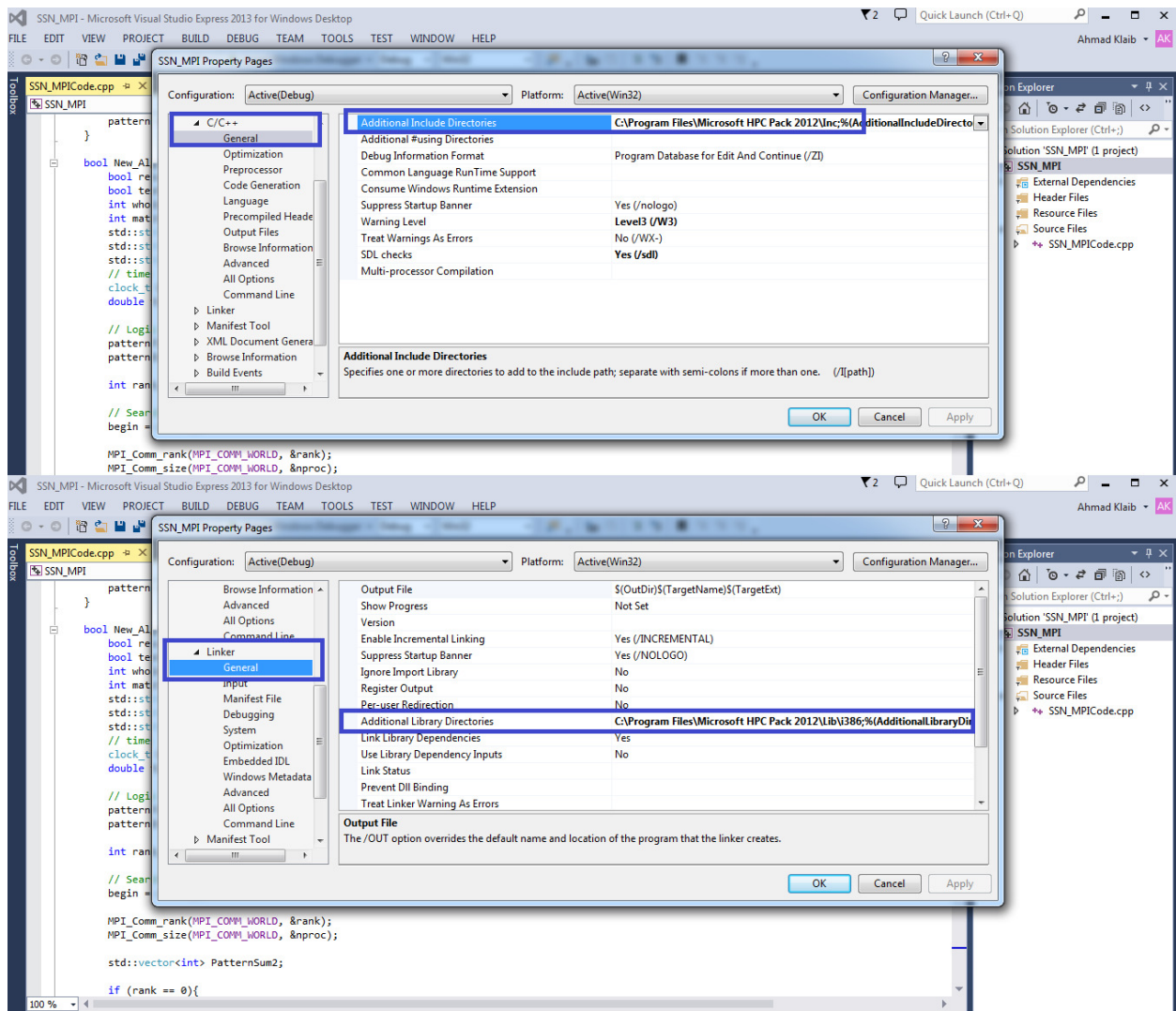


Figure C- 11: Enabling MPI Model in Microsoft Visual Studio

APPENDIX D: LOCAL DATABASE TABLES FORMAT AND DESCRIPTION

We mentioned in chapter 5 (section 5.5), three tables of our Local Database. The following tables show the remaining relational tables which connecting chemical molecules and their corresponding information:

Literature Table (8 Fields)

Field	Type	Null	Key	Default	Example
LITERATURE_ID	int(11)		PRI	0	2366
TITLE	varchar(255)				Hepatoprotective Sesquiterpene Glycosides from Sarcandra glabra
SUBTITLE	varchar(255)	YES		NULL	
TITLE_TOTAL_SOUNDEX	varchar(255)				131632312361524 2321652625362416
TITLE_TOTAL	varchar(255)				Hepatoprotective Sesquiterpene Glycosides from Sarcandra glabra
ET_AL	enum('false','true')			false	false
URL	varchar(255)	YES		NULL	http://pubs3.acs.org/acs/journals/toc.page?incoden=jnprdf&indecade=0&involume=69&inissue=4
DOI	varchar(255)	YES		NULL	10.1021/np050480d

Spectrum Table (15 Fields)

Field	Type	Null	Key	Default	Example
SPECTRUM_ID	int(11)		PRI	0	62458
DATE	datetime		MUL	0000-00-00 00:00:00	2006-06-14 14:53:57
SPECFILE	mediumtext		MUL		
LITERATURE	mediumtext				
LITERATURE_SOUNDEX	mediumtext				
REVIEW_FLAG	enum('false','true','rejected','change','edited','hidden')		MUL	false	false
REVIEW_KEY	varchar(6)				42370
NMRSHIFTDB_NUMBER	varchar(255)		MUL		nmrshiftdb.ice.mpg. de_patel_2006-06- 14_02:53:57_0652
FINGERPRINT	bigint(20)			0	0
USER_ID	int(11)		MUL	0	30003481
SPECTRUM_TYPE_ID	int(11)		MUL	0	1
COMMENT	longblob	YES		NULL	
COMMENT_SOUNDEX	mediumtext				
MOLECULE_ID	int(11)		MUL	0	22207
SIMPLE_SPECFILE	varchar(255)				8.8;0 16.7;0 27.1;0 27.8;0 35.9;0 48.1;0 65.6;0 68.2;0 68.6;0 71.8;0 75;0 75.1;0 77.1;0 78;0 78.2;0 80.5;0 84.5;0 104.2;0 111;0 126.8;0 129.6;0 131.3;0 134;0 165.5;0 176.1;0

Spectrum_ Literature Table (2 Fields)

Field	Type	Null	Key	Default	Example
SPECTRUM_ID	int(11)		PRI	0	62458
LITERATURE_ID	int(11)		PRI	0	2366

Spectrum_ Type Table (4 fields)

Field	Type	Null	Key	Default	Example
SPECTRUM_TYPE_ID	int(11)		PRI	0	1
DIMENSIONALITY	int(11)		MUL	0	1
NAME	varchar(255)				13C
USER_ID	int(11)		MUL	0	30003481

Chemical_Name Table (3 fields)

Field	Type	Null	Key	Default	Example
MOLECULE_ID	int(11)		PRI	0	22207
NAME	varchar(255)		PRI		Sarcaglaboside E
NAME_SOUNDEX	varchar(255)				2624123

Canonical_Name Table (4 fields)

Field	Type	Null	Key	Default	Example
MOLECULE_ID	int(11)		PRI	0	22207
CANONICAL_NAME_TYPE_ID	int(11)		PRI	0	4
NAME	varchar(255)		MUL		InChI=1/C26H38O12/c1-13-4-3-5-15(6-7-16-14(2)23(32)37-17(16)8-13)9-34-24-21(30)20(29)19(28)18(38-24)10-35-25-22(31)26(33,11-27)12-36-25/h4,6,17-22,24-25,27-31,33H,3,5,7-12H2,1-

					2H3/b13-4+,15-6-/t17- ,18?,19?,20?,21+,22?,24+,25?,26?/m0/s1
NAME_SOUNDEX	mediumtext				521263812134351567161 423237171681393424213 291928183824135252312 631271236254617242527 313571212313415631718 1921242526521

Molecule_Hyperlink Table (4 fields)

Field	Type	Null	Key	Default	Example
`	int(11)		PRI	0	22207
HYPERLINK	mediumtext		PRI		http://pubs3.acs.org/acs/journals/toc. page?incoden=jnprdf&indecade=0& involume=69&inissue=4
DESCRIPTION	mediumtext				
DESCRIPTION_SOUNDEX	mediumtext				

Article Table (6 Fields)

Field	Type	Null	Key	Default	Example
LITERATURE_ID	int(11)		PRI	0	2366
PAGES_FROM	int(11)			0	616
PAGES_TO	int(11)			0	620
PUBLISHED_IN_BOOK	int(11)	YES	MUL	NULL	NULL
PUBLISHED_IN_JOURNAL	int(11)	YES	MUL	NULL	10000681
ISSUE_NUMBER	int(11)	YES		NULL	4

Atom Table (10 Fields)

Field	Type	Null	Key	Default	Example
ATOM_ID	int(11)		PRI	0	458931
SYMBOL	char(3)				C
MOLECULE_ID	int(11)		MUL	0	22207
HOSE_CODE_WITH_RINGS	text	YES	MUL	NULL	C-3- 10;=CC(CC,C/C,,C/ CO,=CC)=CC,C,&,O/& C ,&,=O&,C/
ATOMIC_MASS	int(11)			0	6
FORMAL_CHARGE	int(11)			0	0
IS_AROMATIC	enum('false','true')			false	false
IS_VISIBLE	enum('false','true')			false	true
HETERO	varchar(5)			false	false
HOSE_CODE	varchar(120)				C-3;=CC(CC,C/C,,C/ CO,=CC)=CC,C,&,O/ &C,&,=O&,C/

Author Table (5 Fields)

Field	Type	Null	Key	Default	Example
AUTHOR_ID	int(11)		PRI	0	211
SURNAME	varchar(255)				A.
NAME	varchar(255)				Hisham
NAME_TOTAL_SOUNDEX	varchar(255)				25
NAME_TOTAL	varchar(255)				A.Hisham

Bond Table (5 Fields)

Field	Type	Null	Key	Default	Example
BOND_ID	int(11)		PRI	0	561393
DEGREE	int(11)			0	1
IS_CONFIGURATION_SPECIFIED	enum('false','true')			false	false
STEREO	int(11)			0	0
IS_AROMATIC	enum('false','true')			false	false

Bond_Atom Table (3 Fields)

Field	Type	Null	Key	Default	Example
BOND_ID	int(11)		PRI	0	561393
ATOM_ID	int(11)		PRI	0	10070728
ORDER_NUMBER	int(11)			0	1

Book Table (1 Field)

Field	Type	Null	Key	Default	Example
LITERATURE_ID	int(11)		PRI	0	10000651

Book_Publisher Table (3 Fields)

Field	Type	Null	Key	Default	Example
PUBLISHER_ID	int(11)		PRI	0	140
LITERATURE_ID	int(11)		PRI	0	10000651
YEAR	int(11)			0	1972

Bookmark Table (2 Fields)

Field	Type	Null	Key	Default	Example
MOLECULE_ID	int(11)		PRI	0	2217
USER_ID	int(11)		PRI	0	20003220

Canonical_Name_Type Table (2 Fields)

Field	Type	Null	Key	Default	Example
CANONICAL_NAME_TYPE_ID	int(11)		PRI	0	4
CANONICAL_NAME_TYPE	varchar(255)				INChI

Condition Table (4 Fields)

Field	Type	Null	Key	Default	Example
CONDITION_ID	int(11)		PRI	0	1
VALUE	varchar(255)				Unknown
CONDITION_TYPE_ID	int(11)		MUL	0	11
USER_ID	int(11)		MUL	0	1

Condition_Type (7 Fields)

Field	Type	Null	Key	Default	Example
CONDITION_TYPE_ID	int(11)		PRI	0	11
CONDITION_NAME	varchar(255)				Assignment Method
CONDITION_TYPE	char(1)		MUL		M
DATA_TYPE	varchar(255)				String
DICT_REF	varchar(255)				nmr:assignmentMethod
UNITS	varchar(255)				
CML_ENTRY_TYPE	varchar(255)				metadata

Coordinate_Set_2d Table (4 Fields)

Field	Type	Null	Key	Default	Example
COORDINATE_SET_TYPE_ID	int(11)		PRI	0	1
ATOM_ID	int(11)		PRI	0	10070728
X_COORDINATE	double			0	84
Y_COORDINATE	double			0	45

Coordinate_Set_3d (5 Fields)

Field	Type	Null	Key	Default	Example
COORDINATE_SET_TYPE_ID	int(11)		PRI	0	2
ATOM_ID	int(11)		PRI	0	10070728
X_COORDINATE	double			0	-1.6107
Y_COORDINATE	double			0	6.0441
Z_COORDINATE	double			0	-1.229

Coordinate_Set_Type (2 Fields)

Field	Type	Null	Key	Default	Example
COORDINATE_SET_TYPE_ID	int(11)		PRI	0	2
DESCRIPTION	longblob	YES		NULL	Corina generated

Coupling (4 Fields)

Field	Type	Null	Key	Default	Example
SPECTRUM_ID	int(11)		PRI	0	62151
ATOM_ID_1	int(11)		PRI	0	11353692
ATOM_ID_2	int(11)		PRI	0	11353701
CONSTANT	double			0	9.4

D2 (2 Fields)

Field	Type	Null	Key	Default	Example
SPECTRUM_ID	int(11)		MUL	0	
c	bigint(21)			0	

Descriptor_Factors Table (4 Fields)

Field	Type	Null	Key	Default	Example
PROTONCLASS	int(11)	YES		NULL	1
DESCRIPTOR	varchar(6)	YES		NULL	des0
MIN_VALUE	float	YES		NULL	-4.85783
DIVISOR	float	YES		NULL	15.6864

Descriptors_1 Table (12 Fields)

Field	Type	Null	Key	Default	Example
ATOM_ID	int(11)			0	10025644
SIGNAL_ID	int(11)			0	118780
des0	float			0	0.870193
des1	float			0	1
des2	float			0	1
des3	float			0	0
des5	float			0	0
des6	float			0	0.577553
des7	float			0	3.88273E-36
des8	float			0	0.631443
des9	float			0	0.261656
des10	float			0	0.787642

Descriptors_2 Table (13 Fields)

Field	Type	Null	Key	Default	Example
ATOM_ID	int(11)			0	10032017
SIGNAL_ID	int(11)			0	294112
des0	float			0	0.802921
des1	float			0	4.76838E-7
des2	float			0	1
des3	float			0	0
des4	float			0	0.5
des5	float			0	0
des6	float			0	0.0925837
des7	float			0	0.528582
des8	float			0	0.180124
des9	float			0	0.713736
des10	float			0	0.751165

Descriptors_3 Table (13 Fields)

Field	Type	Null	Key	Default	Example
ATOM_ID	int(11)			0	10032068
SIGNAL_ID	int(11)			0	294084
des0	float			0	0.929478
des1	float			0	1
des2	float			0	1
des3	float			0	0
des4	float			0	0
des5	float			0	0
des6	float			0	0.159696
des7	float			0	0.748386
des8	float			0	0.315996
des9	float			0	0.846154
des10	float			0	0.724766

Descriptors_4 Table (11 Fields)

Field	Type	Null	Key	Default	Example
ATOM_ID	int(11)			0	10025646
SIGNAL_ID	int(11)			0	118778
des0	float			0	0.789115
des1	float			0	0.310345
des4	float			0	0
des5	float			0	0
des6	float			0	0.547023
des7	float			0	1.32143E-36
des8	float			0	0.305556
des9	float			0	0.5
des10	float			0	0.960485

Descriptors_All Table (13 Fields)

Field	Type	Null	Key	Default	Example
ATOM_ID	int(11)			0	10025646
SIGNAL_ID	int(11)			0	118778
des0	float			0	8.37889
des1	float			0	1.7
des2	float			0	0
des3	float			0	0
des4	float			0	1
des5	float			0	4
des6	float			0	0.420249
des7	float			0	3.54736
des8	float			0	0.6875
des9	float			0	0.375
des10	float			0	7.49834

Fingerprints Table (17 Fields)

Field	Type	Null	Key	Default	Example
MOLECULE_ID	int(11)			0	
fp0	bigint(20) unsigned	YES		NULL	
fp1	bigint(20) unsigned	YES		NULL	
fp2	bigint(20) unsigned	YES		NULL	
fp3	bigint(20) unsigned	YES		NULL	
fp4	bigint(20) unsigned	YES		NULL	
fp5	bigint(20) unsigned	YES		NULL	
fp6	bigint(20) unsigned	YES		NULL	
fp7	bigint(20) unsigned	YES		NULL	
fp8	bigint(20) unsigned	YES		NULL	
fp9	bigint(20) unsigned	YES		NULL	
fp10	bigint(20) unsigned	YES		NULL	
fp11	bigint(20) unsigned	YES		NULL	
fp12	bigint(20) unsigned	YES		NULL	
fp13	bigint(20) unsigned	YES		NULL	
fp14	bigint(20) unsigned	YES		NULL	
fp15	bigint(20) unsigned	YES		NULL	

Guestbook Table (5 Fields)

Field	Type	Null	Key	Default	Example
GUESTBOOK_ID	int(11)		PRI	0	100
USER_ID	int(11)	YES	MUL	NULL	140
DATE	datetime			0000-00-00 00:00:00	2004-03-05 12:54:34
TEXT	mediumtext				We hope to get more feedback from our users via this guestbook. For example we would like to know why quite a few people register, but only

					some do submit spectra.
VALID	enum('false','true')			false	true

Hcount Table (3 Fields)

Field	Type	Null	Key	Default	Example
MOLECULE_ID	int(11)			0	
H_COUNT	bigint(21)			0	
COUNT	bigint(21)			0	

Hcountnew Table (3 Fields)

Field	Type	Null	Key	Default	Example
MOLECULE_ID	int(11)			0	
H_COUNT	bigint(21)			0	
COUNT	bigint(21)			0	

Hose_Codes Table (7 Fields)

Field	Type	Null	Key	Default	Example
HOSE_CODE	text	YES		NULL	H- ;C(HHC/,, CCC/HCC, H H&,HH.),H C &,HHC,,,,/ ,= CC,,HHC/
VALUE	float			0	0.96

SPECTRUM_ID	int(11)			0	62151
CONDITION_TY PE	char(1)				m
SYMBOL	char(3)				H
REVIEW_FLAG	enum('false','true','rejected','change','edited','hi dden')			false	True
WITH_RINGS	int(1)			0	1

Id_Table (4 Fields)

Field	Type	Null	Key	Default	Example
ID_TABLE_ID	int(11)		PRI	NULL	12
TABLE_NAME	varchar(255)		UNI		MOLECULE
NEXT_ID	int(11)	YES		NULL	124282
QUANTITY	int(11)	YES		NULL	187

Identifier Table (3 Fields)

Field	Type	Null	Key	Default	Example
ATOM_ID	int(11)		PRI	0	20707736
LITERATURE_ID	int(11)		PRI	0	20002226
IDENTIFIER	varchar(255)				C

Is_Author Table (3 Fields)

Field	Type	Null	Key	Default	Example
AUTHOR_ID	int(11)		PRI	0	20001883
LITERATURE_ID	int(11)		PRI	0	20002226
AUTHOR_ORDER	int(11)			0	3

Is_Editor Table (3 Fields)

Field	Type	Null	Key	Default	Example
AUTHOR_ID	int(11)		PRI	0	
LITERATURE_ID	int(11)		PRI	0	
EDITOR_ORDER	int(11)			0	

Isotope Table (4 Fields)

Field	Type	Null	Key	Default	Example
SPECTRUM_TYPE_ID	int(11)		PRI	0	1
AXIS	int(11)		PRI	0	1
ELEMENT_SYMBOL	varchar(255)		MUL		C
ATOMIC_MASS	int(11)			0	13

Jetspeed_Anon_Profile Table (6 Fields)

Field	Type	Null	Key	Default	Example
PSML_ID	int(11)		PRI	0	120
MEDIA_TYPE	varchar(99)	YES	MUL	NULL	wml
LANGUAGE	char(2)	YES		NULL	en
COUNTRY	char(2)	YES		NULL	
PAGE	varchar(99)	YES		NULL	default.psml
PROFILE	longblob	YES		NULL	<?xml version="1.0"?><portlets xmlns= http://www.apache.org/2000/02/C >

Jetspeed_Group_Profile Table (7 Fields)

Field	Type	Null	Key	Default	Example
PSML_ID	int(11)		PRI	0	120
GROUP_NAME	varchar(99)	YES	MUL	NULL	Jetspeed
MEDIA_TYPE	varchar(99)	YES		NULL	
LANGUAGE	char(2)	YES		NULL	
COUNTRY	char(2)	YES		NULL	
PAGE	varchar(99)	YES		NULL	news.psml

PROFILE	longblob	YES		NULL	<?xml version="1.0"?><portlets xmlns= http://www.apache.org/2000/02/C >
---------	----------	-----	--	------	--

Jetspeed_Role_Profile Table (7 Fields)

Field	Type	Null	Key	Default	Example
PSML_ID	int(11)		PRI	0	120
ROLE_NAME	varchar(99)	YES	MUL	NULL	admin
MEDIA_TYPE	varchar(99)	YES		NULL	
LANGUAGE	char(2)	YES		NULL	
COUNTRY	char(2)	YES		NULL	
PAGE	varchar(99)	YES		NULL	default.psml
PROFILE	longblob	YES		NULL	<?xml version="1.0"?><portlets xmlns= http://www.apache.org/2000/02/C >

Jetspeed_User_Profile Table (7 Fields)

Field	Type	Null	Key	Default	Example
PSML_ID	int(11)		PRI	0	120
USER_NAME	varchar(32)	YES	MUL	NULL	admin
MEDIA_TYPE	varchar(99)	YES		NULL	html
LANGUAGE	char(2)	YES		NULL	
COUNTRY	char(2)	YES		NULL	
PAGE	varchar(99)	YES		NULL	default.psml
PROFILE	longblob	YES		NULL	<?xml version="1.0"?><portlets xmlns= http://www.apache.org/2000/02/C >

Journal Table (4 Fields)

Field	Type	Null	Key	Default	Example
JOURNAL_ID	int(11)		PRI	0	283

TITLE	varchar(255)				Journal of Chemical Society
SUBTITLE	varchar(255)	YES		NULL	Perkin Transactions 2
ITA	varchar(255)	YES		NULL	J.Chem.Soc.,Perkin Trans.2

Journal_Volume Table (4 Fields)

Field	Type	Null	Key	Default	Example
JOURNAL_VOLUME_ID	int(11)		PRI	0	466
VOLUME	int(11)			0	0
YEAR	int(11)			0	2001
JOURNAL_ID	int(11)		MUL	0	283

Labgroup Table (3 Fields)

Field	Type	Null	Key	Default	Example
LABGROUP_ID	int(11)		PRI	0	
LABGROUP_NAME	varchar(20)		UNI		
LEADER	int(11)			0	

Machine Table (6 Fields)

Field	Type	Null	Key	Default	Example
MACHINE_ID	int(11)		PRI	0	
NAME	varchar(20)				
FIELD_STRENGTH	int(11)		MUL	0	
LABGROUP_ID	int(11)			0	
VALID	enum('false','true')			false	
DEFAULT_MACHINE	enum('false','true')			false	

Measurement_Condition Table (5 Fields)

Field	Type	Null	Key	Default	Example
MEASUREMENT_CONDITION_ID	int(11)		PRI	NULL	100
TEMPERATURE	float			0	0
FIELD_STRENGTH	int(11)			0	0
SOLVENT	mediumtext				
USER_ID	int(11)			0	100

Publisher Table (3 Fields)

Field	Type	Null	Key	Default	Example
PUBLISHER_ID	int(11)		PRI	0	30000100
NAME	varchar(255)				University of Florida
PLACE	varchar(255)				Florida

Raw_File Table (3 Fields)

Field	Type	Null	Key	Default	Example
RAW_FILE_ID	int(11)		PRI	0	
SAMPLE_ID	int(11)		MUL	0	
URL	varchar(255)				

Raw_File_Spectrum Table (2 Fields)

Field	Type	Null	Key	Default	Example
RAW_FILE_ID	int(11)		PRI	0	
SPECTRUM_ID	int(11)		PRI	0	

ReviewGroup Table (2 Fields)

Field	Type	Null	Key	Default	Example
REVIEWGROUP_ID	int(11)		PRI	0	100
REVIEWGROUP_NAME	varchar(20)		UNI		ice

Review_Group_User Table (2 Fields)

Field	Type	Null	Key	Default	Example
REVIEWGROUP_ID	int(11)		PRI	0	100
USER_ID	int(11)		PRI	0	163

Sample Table (16 Fields)

Field	Type	Null	Key	Default	Example
SAMPLE_ID	int(11)		PRI	0	
USERS_ID	varchar(20)				
USER_ID	int(11)		MUL	0	
SOLVENT	int(11)		MUL	0	
MACHINE	int(11)		MUL	0	
PROBABLE_STRUCTURE	int(11)	YES	MUL	NULL	
DATE	datetime			0000-00-00 00:00:00	
OTHER_NUCLEI	varchar(50)				
SPECIAL_CARE	varchar(50)				
WISHED_SPECTRUM	varchar(50)				
FINISHED	varchar(8)				
PROCESS	enum('self','worker','robot')			self	
ATTACHMENT_NAME	varchar(20)				
ATTACHMENT	longblob				
USERS_ID_COMMENT	varchar(40)				
OTHER_WISHED_SPECTRUM	varchar(50)				

Sessions Table (3 Fields)

Field	Type	Null	Key	Default	Example
SESSIONS	int(11)		PRI	0	2
LOWEST_LOAD_SERVER	varchar(255)	YES		NULL	nmrshiftdb.cubic.uni-koeln.de
SESSIONS_ALL	int(11)			0	176126

Shift Table (3 Fields)

Field	Type	Null	Key	Default	Example
SIGNAL_ID	int(11)		PRI	0	1281322
AXIS	int(11)		PRI	0	1
VALUE	float			0	124.9

Signal Table (4 Fields)

Field	Type	Null	Key	Default	Example
SIGNAL_ID	int(11)		PRI	0	1281322
INTENSITY	float			0	0
SPECTRUM_ID	int(11)		MUL	0	50367
MULTIPLICITY	varchar(255)	YES		NULL	S

Signal_Atom Table (2 Fields)

Field	Type	Null	Key	Default	Example
SIGNAL_ID	int(11)		PRI	0	1281322
ATOM_ID	int(11)		PRI	0	88

Spectrum_Condition Table (2 Fields)

Field	Type	Null	Key	Default	Example
SPECTRUM_ID	int(11)		PRI	0	58353
CONDITION_ID	int(11)		PRI	0	1

Spectrum_Fingerprints Table (4 Fields)

Field	Type	Null	Key	Default	Example
MOLECULE_ID	int(11)			0	
SIMPLE_SPECFILE	varchar(255)				
FINGERPRINT	bigint(20)			0	
NAME	varchar(255)				

Spectrum_Hyperlink Table (4 Fields)

Field	Type	Null	Key	Default	Example
SPECTRUM_ID	int(11)		PRI	0	6887
HYPERLINK	mediumtext		PRI		http://www.mdpi.org/molbank/molbank2002/m0287hnmr.jdx
DESCRIPTION	mediumtext				jdx-File
DESCRIPTION_SOUND EX	mediumtext				23214

Spectrum_Keyword Table (2 Fields)

Field	Type	Null	Key	Default	Example
SPECTRUM_ID	int(11)		PRI	0	17064
KEYWORD_ID	int(11)		PRI	0	3506

Statistics Table (10 Fields)

Field	Type	Null	Key	Default	Example
YEAR	int(11)	YES		NULL	2003
MONTH	int(11)	YES		NULL	8
PREDICTION	int(11)	YES		NULL	88
EXACT_SUBSTRUCTURE	int(11)	YES		NULL	45
LINE_SEARCH	int(11)	YES		NULL	50
SUBSTRUCTURE_FUZZY_SEARCH	int(11)	YES		NULL	47
WEIGHT_SEARCH	int(11)	YES		NULL	0
NAME	int(11)	YES		NULL	138
CAS	int(11)	YES		NULL	894
OTHER_SEARCH	int(11)	YES		NULL	209

Turbine_Group Table (3 Fields)

Field	Type	Null	Key	Default	Example
GROUP_ID	int(11)		PRI	0	1
GROUP_NAME	varchar(99)		UNI		Jetspeed
OBJECTDATA	mediumblob	YES		NULL	

Turbine_Permission Table (3 Fields)

Field	Type	Null	Key	Default	Example
PERMISSION_ID	int(11)		PRI	0	1
PERMISSION_NAME	varchar(99)		UNI		view
OBJECTDATA	mediumblob	YES		NULL	

Turbine_Role Table (3 Fields)

Field	Type	Null	Key	Default	Example
ROLE_ID	int(11)		PRI	0	1
ROLE_NAME	varchar(99)		UNI		User
OBJECTDATA	mediumblob	YES		NULL	

Turbine_Role_Permission Table (2 Fields)

Field	Type	Null	Key	Default	Example
ROLE_ID	int(11)		PRI	0	1
PERMISSION_ID	int(11)		PRI	0	1

Turbine_Scheduled_Job Table (9 Fields)

Field	Type	Null	Key	Default	Example
JOB_ID	int(11)		PRI	0	
SECOND	int(11)			-1	
MINUTE	int(11)			-1	
HOURL	int(11)			-1	
WEEK_DAY	int(11)			-1	
DAY_OF_MONTH	int(11)			-1	
TASK	varchar(99)				
EMAIL	varchar(99)	YES		NULL	
PROPERTY	mediumblob	YES		NULL	

Turbine_User Table (14 Fields)

Field	Type	Null	Key	Default	Example
USER_ID	int(11)		PRI	0	1
LOGIN_NAME	varchar(32)		UNI		admin

PASSWORD_VALUE	varchar(32)				
FIRST_NAME	varchar(99)				
LAST_NAME	varchar(99)				
TITLE	varchar(99)	YES		NULL	
ADDRESS	varchar(99)				
CITY	varchar(99)				
STATE	varchar(99)	YES		NULL	
ZIP_CODE	varchar(99)				
COUNTRY	varchar(99)				
WEB_PAGE	varchar(99)	YES		NULL	
AFFILIATION_1	varchar(99)	YES		NULL	
EMAIL	varchar(99)				

Turbine_User_Group_Role Table (3 Fields)

Field	Type	Null	Key	Default	Example
USER_ID	int(11)		PRI	0	1
GROUP_ID	int(11)		PRI	0	1
ROLE_ID	int(11)		PRI	0	1

User Table (3 Fields)

Field	Type	Null	Key	Default	Example
userid	varchar(20)			0	admin
password	varchar(16)			0	admin
level	varchar(5)	YES		NULL	admin

User_Spectrum Table (5 Fields)

Field	Type	Null	Key	Default	Example
USER_ID	int(11)		PRI	0	140
SPECTRUM_ID	int(11)		PRI	0	10040588
ASSIGNMENT_DATE	datetime	YES		NULL	
REVIEW_DATE	datetime	YES		NULL	2003-08-15 10:31:22
USER_SPECTRUM_ID	int(11)		PRI	0	0

User_Spectrum_Edit Table (3 Fields)

Field	Type	Null	Key	Default	Example
USER_ID	int(11)		PRI	0	163
SPECTRUM_ID	int(11)		PRI	0	3403
EDIT	datetime		PRI	0000-00-00 00:00:00	2002-11-08 00:00:00

User_Spectrum_Mark Table (5 Fields)

Field	Type	Null	Key	Default	Example
USER_ID	int(11)		PRI	0	140
SPECTRUM_ID	int(11)		PRI	0	10014699
MARK	int(11)			0	1
MARK_DATE	datetime			0000-00-00 00:00:00	2004-06-23 10:45:34
COMMENT	mediumtext				Assignments obviously nonsense (but correctly read from dkfz files)

Wishlist Table (4 Fields)

Field	Type	Null	Key	Default	Example
USER_ID	int(11)		PRI	0	1
DATE	datetime			0000-00-00 00:00:00	2005-04-07 15:33:32
SPECTRUM_TYPE_ID	int(11)		PRI	0	2
MOLECULE_ID	int(11)		PRI	0	20053596