Tries in Data Retrieval and Syntactic Pattern Recognition

By

Ghada Badr

B.Sc., M.Sc.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Doctor of Philosophy

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

May 2006

Your file Votre référence
ISBN: 978-0-494-16662-8
Our file Notre référence
ISBN: 978-0-494-16662-8

Abstract

String searching plays an important role in many problems, including text processing, information retrieval, speech and signal processing, pattern recognition, database operations, library systems, compilers, command interpreters, and Bioinformatics. This Thesis deals with problems related to exact and inexact string matching, and in particular, when these problems involve *tries.*

The main aim of this research is to enhance the search performance for strings when they are stored using the *trie* data structure, and to develop methods that work well, in practice, especially for dictionary-based techniques. The enhancing of the search will be done for both domains, namely the *exact* and *approximate* search for strings. The Thesis presents contributions in two main fields, namely *Information Retrieval* and *Syntactic Pattern Recognition.* The following summarize the problems addressed in each of the two fields.

- **Information retrieval**: Exact search. In this part of the Thesis, we consider the problem of performing a sequence of access operations on a set of strings $S = \{s_1, s_2, ..., s_N\}$. We assume that the strings are accessed based on a set of access probabilities $P = \{p_1, p_2, ..., p_N\}$. We also assume that P is not known *a priori*, and that it is time-invariant.

  The problems studied involve searching for "exact" patterns. This will be achieved by applying self-adjusting techniques for the trie data structure when the nodes of the trie are implemented as binary search trees, and by incorporating the concept of "direction" by proposing a new representation for the trie, namely the Dual-Trie (DT).

- **Syntactic pattern recognition**: Approximate string matching. In this part of the Thesis, we consider the traditional problem involved in the syntactic Pattern Recognition (PR) of strings, namely that of recognizing garbled words (sequences). Let Y be a misspelled (noisy) string obtained from an unknown word X^*, which is an element of a finite (but possibly, large) dictionary H stored as a trie, T. Y is assumed to contain Substitution, Insertion and Deletion (SID) errors, and we attempt to obtain an appropriate estimate X^+ of X^*, by processing the information contained in Y.

  We propose to use various Artificial Intelligence (AI) search techniques within a trie, and to optimize the dynamic programming calculations for the edit distances.

Dedicated to my wonderful husband, Osama,
our beautiful children, Omar and Mennah,
my great Mother, Mofida,
and
my twin sister, Mona.

Acknowledgements

First of all, I would like to thank GOD for giving me the health and patience to be able to continue over the past years, these extensive studies and research endeavors. I would like to thank him for the wonderful life, family, friends and the bounties he gave me in life.

I would like to thank my supervisor Prof. John Oommen, whom I consider the best supervisor I have ever met. I am actually proud to be one of his students. Even before coming to Canada, his support was exceptional, and he encouraged me to come here. I learned so much from him, both in research and life. He deals with all his students as a friend, as one who can give encouragement and support whenever he can. He gave me the freedom to conduct my research in any pertinent areas of the field, and in the best suitable manner. He encouraged and taught me how to prepare publications, and also how to present my work in famous journals and conference proceedings. His insight helped me to improve my skills in both the empirical and theoretical avenues. I was also one of his research and teaching assistants. Thank you for everything!

I would also like to thank Dr. Pat Morin for supporting me during my research. I also have had the pleasure of working as a teaching assistant for Dr. Michiel Smid and Dr. Anil Maheshwari.

I have many thanks for Linda Pfeiffer, our supportive School Administrator. She was so helpful to me, and I will miss her friendly manner. Thank you, Linda, for resolving the many issues we encountered in the School. I am also grateful to Claire, Sharmila, and Joanne for the remarkable work they do for all of us. Thanks also to Dr. Jean-Pierre Corriveau for opening his door for advice whenever we needed.

With all my heart and feelings, I would like to thank my loving and caring husband Dr. Osama Nasr. Osama is not just a helpful husband to me, but he is all that I have in life. He has been such a great support to me and to our children, and without him, I don't think I would have been able to achieve a single piece of work included in this Thesis. Thank you, Osama, thank you my dear.

I would like to thank my great children, Mennah and Omar, for giving me all the happiness and pleasure of life. Without their fun and noise, I don't think I would have achieved anything important in life. They are my goal, my success and my great happiness. GOD bless you both.

Thanks to my great mother Mofida for her continuous support and prayers for me. Without her I would not have been alive, and would neither have been able to achieve any success. She was the first to teach me how to do research and how to be a good person in life. She has always been there to help me with the children, and with all the experience she has had in her life. I have so many great feelings for her, and I hope that life will give me the chance to be able to make her happy. I also miss my twin sister Mona. She is always in my heart and I wish her success in her Ph.D. in Pharmaceutical studies as well. I would like to thank her for helping my husband and me in whatever we needed in our home country. Without her, my life would be just half of what it is, as she is my second half.

Thanks to my friends in the university, and specially Ebaa and Dragos. They are such good friends who are so helpful to every one. I also like their jokes and their cheerful smiles. I wish them the best in their studies and in life. I would like to also thank my friends Soha, Hala, Rehab, Amal, and Rasha for the great and wonderful times we have spent together.

Finally, I would like to thank my supportive friend Eisha. She was one of the reasons for the success of this work. She helped me with my children, and was always there to give me the advice and encouragement whenever I needed it, even without it appearing to be a bother to her. Thank you, Eisha, and GOD bless you and your children.

Contents

I Thesis Overview 1

1 Introduction 2

1.1 Motivations and Objectives 3

1.2 Trie data structure and its Applications 5

1.3 Thesis Contributions and Organization 7

II Tries for Information Retrieval 12

2 Tries in Data Retrieval: State of the Art 13

2.1 Introduction 13

2.2 Variations on Tries 14

2.2.1 Non-compact Tries 17

2.2.2 Compact Tries and Patricia tries 24

2.2.3 Two-Trie 29

2.2.4 Compact-Balanced Tries (CB-tries) 30

2.2.5 Compact Binary Tries 32

2.2.6 Level-Compressed Trie (LC-trie) and Level-Path-Compressed Trie (LPC-trie) 36

2.2.7 Order-Containing Trie (O-trie) 39

2.2.8 Variable-Depth Trie 40

2.2.9 b-Trie 41

2.2.10 Burst Trie 42

2.2.11 Cache-Efficient Tries 44

2.3 Conclusion 45

3 Self-Adjusting Ternary Search Tries **46**

3.1 Introduction 46

3.2 Ternary Search Tries (TSTs) 48

3.2.1 Rotation in TST 49

3.3 Self-Adjusting Heuristics 50

3.4 Proposed Restructuring Methods for Ternary Search Tries (TST) 56

3.4.1 Splaying for TSTs 56

3.4.2 Randomized TSTs 60

3.4.3 Conditional Rotations for TSTs 62

3.5 Experimental Results 67

3.5.1 Data Sets 68

3.5.2 Methodology 71

3.5.3 Analysis of Results 72

3.6 More Related Work 75

3.7 Conclusion 79

4 Dual-Tries **81**

4.1 Introduction 81

4.2 Two-trie 82

4.3 The Proposed Dual-Trie (DT) Sructure 84

4.3.1 Main idea 85

4.3.2 The DT Structure 85

4.3.3 Example 86

4.4 Operations on DT 87

4.4.1 Insertion 88

4.4.2 Retrieval 89

4.4.3 Deletion 90

4.4.4 Analysis 92

4.5 Experimental Results 93

4.5.1 Analysis of Results 95

4.6 Self-Adjusting Dual-Tries 98

4.7 Conclusion 101

III Tries for Syntactic Pattern Recognition **104**

5 Notations and Basic-Concepts **105**

5.1 Introduction 105

5.2 Notation and The Problem Definition 105

5.2.1 Notations 106

5.2.2 The Problem 106

5.2.3 Applications 107

5.3 String Distance 108

5.3.1 String-to-String Distances 108

5.3.2 Computing String Distances 111

5.4 Artificial Intelligence Search Techniques 113

5.4.1 Blind or Uninformed Techniques 113

5.4.2 Heuristic search 115

5.4.3 Branch and Bound (BB) search 117

5.5 Assumptions 118

5.6 Conclusion 119

6 State of the Art: Trie-based Syntactic PR **120**

6.1 Introduction 120

6.2 Classifications 121

6.3 String-to-String-Based Approaches 126

6.4 Text-Based Approaches 127

6.4.1 On-line text-based approaches 127

6.4.2 Off-line text-based approaches 128

6.5 Dictionary-Based Approaches 130

6.6 Trie-Based Approaches 133

6.6.1 Breadth-First Trie-based Scheme 135

6.6.2 Depth-First Trie-based Scheme 137

6.7 Conclusion 139

7 Breadth-First-Trie Based Scheme **140**

7.1 Introduction 140

7.2 Procedure and Data Structure for Obtaining X^+ 141

7.2.1 Tries and the Linked Lists of Prefixes (LLP) 141

7.2.2 The Procedure for Obtaining X^+ 144

7.3 Optimizing Computations When K is Known 148

7.4 Experimental Results 148

7.5 Conclusion 151

8 A Look-Ahead Branch and Bound Pruning Scheme **157**

8.1 Introduction 157

8.2 Tries and Cutoffs 158

8.3 Look-Ahead Branch and Bound Scheme 160

8.3.1 The look-ahead component 162
8.3.2 The dynamic component 162
8.3.3 The static component 163
8.3.4 The overall heuristic 164
8.3.5 Algorithm for Obtaining X^+ Using LHBB 165
8.4 A Look-Ahead BB Scheme for General Costs 165
8.5 Experimental Results 168
8.5.1 Experimental Setup I: 0/1 Costs 169
8.5.2 Experimental Setup II: General Costs 170
8.6 Conclusion 172

9 Clustered-Beam-Search **174**

9.1 Introduction 174
9.2 Proposed Clustered Beam Search (CBS) 175
9.2.1 The Proposed CBS Algorithm and its Complexity 176
9.2.2 Example 177
9.3 The CBS for Approximate String Matching 178
9.3.1 The Heuristic measure 178
9.3.2 Characteristics of the Heuristic Functions 180
9.3.3 Data Structures Used 181
9.3.4 Applying the CBS 182

9.4 Experimental Results 182
9.5 Optimizing Computations when Changing the Error Model 187
9.6 Conclusion 189

10 Trie-Based Dynamic Matrix Optimization **192**

10.1 Introduction 192
10.2 Optimizing the DP Matrix Computations 194
10.2.1 Main idea 195
10.2.2 Data-structures 196
10.2.3 Algorithm and correctness 198
10.3 Optimization when maximum number of errors K is known 203
10.4 An example 208
10.5 Experimental Results 208
10.6 Conclusion 211

IV Contributions and Future Directions **214**

11 Conclusions **215**

11.1 Tries in Information Retrieval 216
11.1.1 Self-Adjusting Ternary Search Tries 216
11.1.2 Dual-Tries 217
11.1.3 Future Work in Information Retrieval 218

11.2 Tries in Syntactic Pattern Recognition 218

11.2.1 Breadth-First-Trie Based Scheme 219

11.2.2 A Look-Ahead Branch and Bound Pruning Scheme 220

11.2.3 Clustered-Beam-Search 220

11.2.4 Trie-Based Dynamic Matrix Optimization 221

11.2.5 Future Work in Syntactic Pattern Recognition 222

A Pseudo-code for the Self-Adjusting TSTs **225**

Bibliography **231**

List of Tables

3.1 Statistics of the data set used in the experiments. 70

3.2 Snapshots of the strings in Dict, Genome and NASA data sets. 70

3.3 Space required, in KB, by the different data structures for the different dictionaries used. 75

4.1 Space and time performance for the different data structures when the **Dict** dictionary of 25,481 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the f portion ranging from 1 to 9, when r is set to unity. The symbol ">>" means that the time taken is relatively much more than the other corresponding times reported in the table. 95

4.2 Space and time performance for the different data structures when the **Dict** dictionary of 25,481 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the r portion ranging from 1 to 9, when f is set to unity. . 96

4.3 Space and time performance for the different data structures when the **Webster** dictionary of 91,479 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the f portion ranging from 1 to 9, when r is set to unity. The symbol ">>" means that the time taken is relatively much more than the other corresponding times reported in the table. 97

4.4 Space and time performance for the different data structures when the **Webster** of 91,479 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the r portion ranging from 1 to 9, when f is set to unity. . . . 97

4.5 Space and time performance for the different data structures when the **Genome** dictionary of 262,084 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the f portion ranging from 1 to 9, when r is set to unity. The symbol ">>" means that the time taken is relatively much more than the other corresponding times reported in the table. 98

4.6 Space and time performance for the different data structures when the **Genome** dictionary of 262,084 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the r portion ranging from 1 to 9, when f is set to unity. . 98

4.7 Space and time performance for the different data structures when the **NASA** dictionary of 15,699 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the f portion ranging from 1 to 9, when r is set to unity. . 99

4.8 Space and time performance for the different data structures when the **NASA** dictionary of 15,699 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the r portion ranging from 1 to 9, when f is set to unity. . 99

4.9 Time performance for the dual-trie when applying different adjusting techniques and when the **Dict** dictionary of 25,481 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the f portion ranging from 1 to 9, when r is set to unity. 101

4.10 Time performance for the dual-trie when applying different adjusting techniques and when the **Webster** dictionary of 91,479 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the f portion ranging from 1 to 9, when r is set to unity. The values in brackets show the time taken for the learning phase of the Conditional rotation heuristic. 102

4.11 Time performance for the dual-trie when applying different adjusting techniques and when the **Genome** dictionary of 262,084 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the f portion ranging from 1 to 9, when r is set to unity. The values in brackets show the time taken for the learning phase of the Conditional rotation heuristic. 102

4.12 Time performance for the dual-trie when applying different adjusting techniques and when the **NASA** dictionary of 15,699 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the f portion ranging from 1 to 9, when r is set to unity. 103

5.1 Primary variables and terms used in this part of the Thesis 107

7.1 Statistics of the data sets used in the experiments. 149

7.2 A subset of the dictionary, some noisy strings, from the set SE of the dictionary called Eng, and their characteristics. 153

7.3 Noise statistics of the set SA, SB, SC, SD, and SE that correspond to the **Eng** dictionary. 154

7.4 Noise statistics of the set SA, SB, SC, SD, and SE that correspond to the **Dict** dictionary. 154

7.5 Noise statistics of the set SA, SB, SC, SD, and SE that correspond to the **Webster** dictionary. 154

7.6 The experimental results obtained from each of the five sets for the **Eng** dictionary. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the BFS-trie-based method obtains over the DFS-trie-based method. 155

7.7 The experimental results obtained from each of the five sets for the **Dict** dictionary. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the BFS-trie-based method obtains over the DFS-trie-based method. 155

7.8 The experimental results obtained from each of the five sets for the **Webster** dictionary. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the BFS-trie-based method obtains over the DFS-trie-based method. 155

7.9 The experimental results obtained from each of the five sets for the **Dict** dictionary for the optimized case when the percentage of error is considered known *a priori*. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the BFS-trie-based method obtains over the DFS-trie-based method. 156

7.10 The experimental results obtained for each of the three different errors for the **Dict** dictionary for the optimized case when the number of error is not known. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the BFS-trie-based method obtains over the DFS-trie-based method. 156

7.11 The experimental results obtained for each of the three different errors for the **Dict** dictionary for the optimized case when the number of error is considered known *a priori*. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the BFS-trie-based method obtains over the DFS-trie-based method. 156

8.1 The results obtained in terms of the number of operations (additions and minimizations) needed when the maximum number of errors $K = 1$, and the costs are of a 0/1 form. The figures given are in Millions. The time shown is in seconds, and the *total* improvement obtained is in "bold" face. 169

8.2 The results obtained in terms of the number of operations (additions and minimizations) needed when the maximum number of errors $K = 2$, and the costs are of a 0/1 form. The figures given are in Millions. The time shown is in seconds, and the *total* improvement obtained is in "bold" face. 170

8.3 The results obtained in terms of the number of operations (additions and minimizations) needed when the maximum number of errors $K = 3$, and the costs are of a 0/1 form. The figures given are in Millions. The time shown is in seconds, and the *total* improvement obtained is in "bold" face. 171

8.4 The results obtained in terms of the number of operations (additions and minimizations) needed when maximum number of errors, K is 1, 2, *and* 3, the costs are of 0/1 from, the best match optimization is applied, and the *Dict* dictionary is used. The figures given are in Millions. The *total* improvement obtained is in "bold" face. 171

8.5 The results obtained in terms of the number of operations (additions and minimizations) needed when maximum number of errors, K is 1, 2, *and* 3, the costs are general, and the *Dict* dictionary is used. The figures given are in Millions. The *total* improvement obtained is in "bold" face. 172

8.6 The results obtained in terms of the number of operations (additions and minimizations) needed when maximum number of errors K is 1, 2, *and* 3, the costs are general, the best match optimization is applied, and the *Dict* dictionary is used. The figures given are in Millions. The *total* improvement obtained is in "bold" face. 172

9.1 The experimental results obtained from each of the three sets for the **Eng** dictionary. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the **CBS-LLP**-based method obtains over the **DFS-trie**-based method. 186

9.2 The experimental results obtained from each of the three sets for the **Dict** dictionary. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the **CBS-LLP**-based method obtains over the **DFS-trie**-based method. 187

9.3 The experimental results obtained from each of the three sets for the **Webster** dictionary. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the **CBS-LLP**-based method obtains over the **DFS-trie**-based method. 188

9.4 The experimental results obtained from each of the three sets for the **Dict** dictionary. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the **CBS-LLP**-based method obtains over the **BS-LLP**-based method when applied to set **SA**. 189

9.5 The experimental results obtained from each of the three sets for the **dict** dictionary. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the **CBS-LLP**-based method obtains over the **DFS-trie**-based method when the optimized **error model** is used and $q = 5$. 190

10.1 The results obtained in terms of the number of operations (additions and minimizations) needed when the maximum number of errors $K = 1$ and known *a priori*, the costs are of a 0/1 form, and the new OPT-DFS-trie is applied over the **LHBB** scheme. The figures given are in Millions, and the *total* improvement obtained is in "bold" face. 211

10.2 The results obtained in terms of the number of operations (additions and minimizations) needed when the maximum number of errors $K = 2$ and known *a priori*, the costs are of a 0/1 form, and the new OPT-DFS-trie is applied over the **LHBB** scheme. The figures given are in Millions, and the *total* improvement obtained is in "bold" face. 211

10.3 The results obtained in terms of the number of operations (additions and minimizations) needed when the maximum number of errors $K = 3$ and known *a priori*, the costs are of a 0/1 form, and the new OPT-DFS-trie is applied over the **LHBB** scheme. The figures given are in Millions, and the *total* improvement obtained is in "bold" face. 212

10.4 The results obtained in terms of the number of operations (additions and minimizations) needed when the maximum number of errors $K = 1$ and is **not** known *a priori*, the costs are of a 0/1 form, and the new OPT-DFS-trie is applied over the **DFS-trie** scheme. The figures given are in Millions. The time shown is in seconds, and the *total* improvement obtained is in "bold" face. 212

List of Figures

2.1 An example of the standard trie for the set {bbbc, ccabbba, ccaca, ccbaaca, ccbababc, cccacc, ccccac}. 14

2.2 A trie and a digital search tree built from the strings $X^1 = 11100\ldots$, $X^2 = 10111\ldots$, $X^3 = 00110\ldots$, *and* $X^4 = 00001\ldots$. 16

2.3 The array-trie for the set {bbbc, ccabbba, ccaca, ccbaaca, ccbababc, cccacc, ccccac} corresponding to the trie given in Figure 2.1. 18

2.4 The list-trie for the set {bbbc, ccabbba, ccaca, ccbaaca, ccbababc, cccacc, ccccac} corresponding to the trie given in Figure 2.1. 19

2.5 The BST-trie for the set {bbbc, ccabbba, ccaca, ccbaaca, ccbababc, cccacc, ccccac} corresponding to the trie given in Figure 2.1. 20

2.6 The compact trie and the corresponding double-array trie for the words: {bachelor, baby, badge, jar} taken from [12]. 23

2.7 The compact trie for the set {bbbc, ccabbba, ccaca, ccbaaca, ccbababc, cccacc, ccccac} corresponding to the trie given in Figure 2.1. 24

2.8 a suffix tree constructed from the string "ccacba". 25

2.9 The Patricia trie for the set {bbbc, ccabbba, ccaca, ccbaaca, ccbababc, cccacc, ccccac} corresponding to the trie given in Figure 2.1. 26

2.10 An example (adapted from [140]) for the trie and the corresponding two-trie for the words {precession, prevision, procession, provision, inspiration, instant, instrument, in}. 29

2.11 An example of a Compact trie and the corresponding CB-trie and the corresponding nodes. 31

2.12 The BDS-tree for a set of keys and the corresponding HBDS-tree. This figure is taken from [141]. 33

2.13 The pre-order bit stream for the BDS-tree and the HBDS-tree corresponding to Figure 2.12. The figure is taken from [141]. 33

2.14 An example of the Patricia BDS-tree. (Left) The ordinary trie. (Right) The Patricia BDS-tree. The figure is taken from [143]. 34

2.15 The pre-order bit stream for the ordinary trie and the Patricia BDS-tree corresponding to Figure 2.14. The figure is taken from [143]. 34

2.16 A binary trie and the corresponding LC-trie. Compressed levels are marked by rectangles. This figure was adapted from [9]. 37

2.17 (a) A binary trie; (b) A path compressed trie (Patricia tre); (c) The corresponding LPC-trie. This figure has been adapted from [111]. 38

2.18 The variable-depth trie for the words; {band, bank, bead, beam, bed, bell}. (Left) the standard trie. (Right) the corresponding variable-depth trie. This figure is adapted from [130]. 40

2.19 A 3-trie from [81] built from the strings $X^1 = 11000\ldots$, $X^2 = 11100\ldots$, $X^3 = 11111\ldots$, *and* $X^4 = 1000\ldots$, $X^5 = 10111\ldots$, $X^6 = 10101\ldots$, $X^7 = 00000\ldots$, $X^8 = 00111\ldots$, $X^9 = 00101\ldots$, $X^10 = 00100\ldots$ 42

2.20 An example of a burst trie (adapted from [69]) with BSTs used to represent containers. 43

3.1 Rotation operations as applicable to the TST. 50

3.2 Different cases for the splaying operation for the TST. Node x is the current node involved in the splaying. The cases are: (a) Zig. (b) Zig-Zig. (c) Zig-Zag. 57

3.3 The accumulated time in seconds needed for all the data structures studied for various numbers of words, when the *Dict* dictionary was used, and the document was skewed by the Zipf's distribution. 73

3.4 The accumulated time in seconds needed for all the data structures studied for various numbers of words, when the *Dict* dictionary was used, and the document was skewed by the Exponential distribution. 74

3.5 The accumulated time in seconds needed for all the data structures studied for various numbers of words, when the *Dict* dictionary was used, and the document was skewed by the wedge distribution. 75

3.6 The accumulated time in seconds needed for all the data structures studied for various numbers of words, when the *Dict* dictionary was used, and the document was skewed by the Uniform distribution. 76

3.7 A comparison between the different data structures when various *Dict* documents were accessed against the *Dict* dictionary for 150 million accesses, when the ensemble average of the time required for the last 50 million accesses was measured. 77

3.8 The accumulated time in seconds needed for all the data structures studied for various numbers of words in *Moby Dick* was searched against the *Webster's Unabridged Dictionary* in the order of occurrence in the novel. 78

3.9 The accumulated time in seconds needed for all the data structures studied for various numbers of words in the *Genome* document when searched against the *Genome* dictionary in the order of occurrence in the document. 78

3.10 The accumulated time in seconds needed for all the data structures studied for various numbers of words in the *NASA* document when searched against the *NASA* dictionary in the order of occurrence in the document. 79

4.1 An example (adapted from [140]) for the trie and the corresponding two-trie for the words {precession, prevision, procession, provision, inspiration, instant, instrument, in}. 83

4.2 An example for a standard trie (left) and the corresponding Dual-Trie (right) for 24 n-grams of length 6 for Genomic data. 87

5.1 An example of the calculation flow in the Dynamic Programming Matrix DPM when it is used to compute the edit distance $D(snip, fight)$ where a row-wise left-to-right traversal is used. 112

5.2 The depth-first search path on a trie example. 114

5.3 The breadth-first search path on a trie example. 114

6.1 A broad classification of approximate string matching approaches. 121

6.2 An example of a dictionary stored as a trie with the words {for, form, fort, fortran, formula, format, forward, forget}. 134

6.3 An example of the calculation done on the same trie shown in Figure 6.2, when the calculation is done for the path corresponding to the word "formula" where the noisy word is "farm", and $K = 1$. One additional branch is shown to demonstrate that only *one* column is to be calculated per node. The shaded area represents the entries that will not be calculated if the Ukkonen cutoff and the observations in [137] are used. 138

7.1 The corresponding LLP (right) for the trie (left) with words {for, form, fort, fortran, formula, format, forward, forget}. 144

7.2 An example for the calculations done to obtain X^+ for the noisy word $Y = port$ which corresponds to the original word $U = fort$. The computation uses the LLP which corresponds to the trie with words {for, form, fort, fortran, formula, format, forward, forget}. The distances are obtained as described in the text. . 147

8.1 An example of a dictionary stored as a trie with the words {For, Form, Fort, Fortran, Forma, Forget, Format, Formula, Forward}. 160

8.2 The cutoff done for the trie example when applying Ukkonen's cutoff, for $Y =$ "fwt" and $K = 2$. 161

8.3 The cutoff done for the trie example when applying only the LHBB technique, for $Y =$ "fwt" and $K = 2$. 165

8.4 The cutoff done for the trie example when applying both Ukkonen's cutoff and the LHBB technique, for $Y =$ "fwt" and $K = 2$. 167

9.1 An example showing the benefits of the CBS over the BS. 178

9.2 The results for comparing the **CBS-LLP**-based method with the **DFS-trie**-based method for (top) the **Eng** dictionary, (middle) the **Dict** dictionary, and (bottom) the **Webster** dictionary. The time is represented by total number of operations in millions. 185

9.3 The results for comparing the **CBS-LLP**-based method with the **BS-LLP**-based method for the **Dict** dictionary when applied to set **SA**. The time is represented by total number of operations in millions. 189

9.4 The results for comparing the **CBS-LLP**-based method with the **DFS-trie**-based method for the **Dict** dictionary when the optimized **error model** is used and $q = 5$. The time is represented by total number of operations in millions. 190

10.1 An example for the tuple representation, V, of Y when the alphabet, A, is the English alphabet and Y equals "farm". 197

10.2 An example which shows a common column between the symbols t and g. 209

10.3 An example for edit distance calculations for the noisy word "farm" against the whole dictionary in Figure 8.1. 213

List of Algorithms

2.1 Algorithm Generic-Trie-Search 15
4.1 Algorithm DT-Insert 89
4.2 Algorithm DT-Search 90
4.3 Algorithm DT-Delete 91
7.1 Algorithm Build-LLP 143
7.2 Recognize-Using-LLP 145
8.1 Algorithm LHBB 166
9.1 CBS 179
9.2 CBS-LLP-based Scheme 183
10.1 Optimized-DFS-Trie 204
10.2 edit_distance(i, *common*, *commonp*) 205
10.3 Optimized-LHBB-Trie 207
A.1 Algorithm Splay-TST-Access 226
A.2 Splay 227
A.3 Rand-TST-Insert 228
A.4 Rebalance 229
A.5 Optimized-Cond-TST 230

Part I

Thesis Overview

Chapter 1

Introduction

The main trend since the advent of computers, when it concerns man-machine interaction, is to place more of the communication burden on the machine and less on the human being. This trend is evident by the development in computer communication languages, from the early, very primitive machine languages, to the current, highly-sophisticated higher-level languages.

String searching plays an important role in many problems, including text processing, information retrieval, database operations, library systems, compilers, command interpreters, DNA processing, signal processing, error correction, speech and pattern recognition, and in several other fields. This Thesis deals with the problems related to exact and inexact string matching, and, in particular, when these problems involve *tries*.

This Chapter provides an introduction to the Thesis, and illustrates its main objectives and contributions. The Chapter is organized as follows: Section 1.1 illustrates the main motivations and the objectives of the work. Section 1.2 presents the *trie* data structure and the different applications for which it is mainly used. Section 1.3 gives the organization of the Thesis and the main contributions of each Chapter.

1.1 Motivations and Objectives

Finding the occurrences of a given query string (pattern) from a (possibly) very large text is an old and fundamental problem in computer science. This task, in its entirety, known as string matching, has several different variations. The most natural and simple of these involves exact string matching, in which case the user wishes to find only occurrences that are exactly identical to the pattern string.

As opposed to this, in many situations, the pattern and/or the text are not exact. This could be due to inaccuracies in the optical character recognition, typing or misspelling errors, or because the user is looking for the approximate pattern. For example, a name we are looking for may be misspelled in the text, or may be entered wrongly because the user fails to remember its exact spelling. The approximate text searching problem is to find all substrings in a text database that are at a given distance K or less, from a pattern P. The distance between two strings (which will be formalized later) is the minimum cost associated with the insertions, deletions, and substitutions of single characters in the strings that are needed to make them equal. The case $K = 0$ corresponds to the classical exact string matching problem.

When the text collection is large, it demands specialized indexing techniques for efficient text retrieval. A simple and popular indexing technique is the inverted list. It is especially adequate when the pattern to be searched for consists of simple words. This is a common type of query encountered, for instance, when searching the World Wide Web, and consequently, inverted lists have been widely used in that context.

Data structures and algorithms on strings are used in a variety of applications [149] ranging from information theory, telecommunications, wireless communications, approximate pattern matching, molecular biology, game theory, coding theory, and source coding, to stock market analysis. These structures and their associated schemes have experienced a new wave of interest due to a growing number of novel applications [149] in computer science, communications, and biology. These applications include dynamic hashing, the partial match retrieval of multidimensional data, conflict resolution, algorithms for broadcast communications, data compression, coding, security, gene searching, DNA sequencing, and genone mapping etc.

Although various data structures have been proposed for each of these problems, undoubtedly

the most popular data structures used in algorithms involving strings and words are digital trees, or *tries*.

The trie supports the search for any string w in a set S by following an access path dictated by the successive letters of w [45]. In a similar manner, the trie can be implemented to deal with insertions and deletions, so that it is a fully "dynamic dictionary" data type. In addition, tries efficiently support set-theoretic operations like union and intersection, as well as partial match queries or interval search, and suitable adaptations make them a method of choice for complex text processing tasks.

The main aim of this Thesis is to enhance the search performance for strings when they are stored using the *trie* data structure, and to develop methods that work well in practice, especially for dictionary-based techniques. The enhancing of the search will be done for two domains, namely the *exact* and the *approximate* string-searching domains. This will be achieved in the "exact" domain in two ways. In the first method, we will apply self-adjusting techniques for the trie data structure when nodes of the trie are implemented as binary search trees. In the second method, we will incorporate the concept of "direction" by proposing a new representation for the trie, namely the Dual-Trie (DT). With regard to the "approximate" domain, we will use Artificial Intelligence (AI) search techniques when the strings are stored in a trie, and we will also optimize the dynamic programming calculations for edit distances.

This Thesis is structured in four parts. The first part presents an overview of the Thesis, and a description of the problems addressed in this research work. The second part involves a complete survey of the trie data structure and the different possible variations for it. It includes a study of solutions that we enhanced with respect to the *exact* domain. The third part starts with the basic notations that will be used and with an overview of the background needed. It also includes the different contributions done in the *approximate* domain. The last part of the Thesis discusses the conclusions of this research work, and the possible future extensions.

The following are the general definitions of the two problems considered in the second and third parts of the thesis.

- **Part II**: Tries for Information Retrieval: Exact domain.

  In this part, we consider the problem of performing a sequence of access operations on a

set of strings $S = \{s_1, s_2, ..., s_N\}$. We assume that the strings are accessed based on a set of access probabilities $P = \{p_1, p_2, ..., p_N\}$. We also assume that P is not known *a priori*, and that it is time-invariant.

We attempt to solve this problem by representing the set of strings using the Ternary Search Trie (TST) [45] data structure as will be described in Chapters 2 and 3, and by using a new proposed Dual-Trie (DT) data structure as will be presented in Chapter 4.

- **Part III**: Tries for Syntactic Pattern Recognition: Approximate domain.

  In this part, we consider the traditional problem involved in the syntactic Pattern Recognition (PR) of strings, namely that of recognizing garbled words (sequences). Let Y be a misspelled (noisy) string obtained from an unknown word X^*, which is an element of a finite (but possibly, large) dictionary H; the dictionary is stored as a trie, T. Y is assumed to contain Substitution, Insertion and Deletion (SID) errors and we attempt to obtain an appropriate estimate X^+ of X^*, by processing the information contained in Y. The advantages of using a finite dictionary in text recognition applications are many: The accuracy of the recognition is very high, and a noisy string is never recognized as a word which is not in the language, thus avoiding "meaningless" decisions. Furthermore, the time complexity of the computations involved in the recognition is, in the worst case, quadratic per word and linear in the size of the dictionary. The quadratic complexity can always be decreased if the dictionary is modelled using a trie, which is the main objective of the work done in the Thesis.

  The work reported in Chapters 7, 8, 9, and 10 assumes the use of Levenshtein distance assignments for inter-symbol costs, as detailed in Chapter 5.

1.2 Trie data structure and its Applications

The concept of tries [12, 28, 69] was originally conceived by Brandais [38] and later given that name by Fredkin [63] which he derived from the word *re***trie***val* in *information retrieval systems.* Tries are one of the most general-purpose data structures used in computer science today [28] because of the wide range of applications that they support improve most performance criteria.

A *trie* is an alternative to a Binary Search Tree (BST) for storing strings in a sorted order

[69]. Tries are both an abstract structure and a data structure that can be superimposed on a set of strings over some fixed alphabet [45]. As an abstract structure, they are based on a splitting scheme, which, in turn, is based on the letters encountered in the strings. This rule can be described as follows:

If S is a set of strings, and $A = \{a_j\}_{j=1}^{r}$ is the alphabet, then the trie associated with S is defined recursively by the rule [45]:

$$trie(S) = (trie(S\backslash a_1), ..., trie(S\backslash a_r)).$$

where $S\backslash a_i$ means the subset of S consisting of strings that start with a_i, stripped of their initial letter a_i. The above recursion is halted as soon as S contains less than two elements. The advantage of the trie is that it only maintains the minimal prefix set of characters that is necessary to distinguish all the elements of S.

Tries are widely used for the efficient storage, matching and retrieval of strings over a given alphabet. They are also extensively used to represent a *set* of strings [69], that is, for dictionary management. Thus, the range of applications encompasses natural language processing, and searching for reserved words in a compiler, for IP routing tables, and text compression. Balanced tries (represented in a single array) are used as a data structure for dynamic hashing. Clearly, this broad range of applications justifies the perception of tries as a general-purpose data structure, whose properties are fairly well-understood.

One of the main applications encountered is approximate string matching [26, 32, 92, 105, 107, 112]. This is one of the main domains that we will consider in the third part of the Thesis, for which we will use the trie as the main data structure in our underlying implementations. A complete survey on the different variations for the trie data structure and for using it in information retrieval and exact matching will be given in Chapter 2. A complete survey of the field of approximate string matching, and in particular, about dictionary-based approaches involving tries will be given in Chapter 6.

1.3 Thesis Contributions and Organization

In this Section we present the overall organization of the Thesis, and the contributions made in each Chapter.

- **Chapter 2: Tries in Data Retrieval: State of the Art**

  This Chapter embarks on a fairly comprehensive survey of the *trie* data structure. Although we will not be utilizing many of its variants, we feel that it is still educative to submit such a survey because it will serve as an overview of the field, and a foundation for the rest of the Thesis.

- **Chapter 3: Self-Adjusting Ternary Search Tries**

  The Ternary Search Trie (TST) is another representation for the trie that helps to save space by changing the representation of the trie node. In this Chapter, we demonstrate how we enhance the search time for TSTs by applying two self-adjusting heuristics, that have been previously used for Binary Search Trees (BSTs), to TSTs. These heuristics are namely, the **splaying** [3, 145] and the **Conditional Rotation** [3, 44] heuristics. We have also applied another balancing strategy used for BSTs, to TSTs, which is the basis for the **Randomized** search trees, or Treaps [13, 53, 154]. In an earlier paper [145], Oommen and his co-authors showed that the conditional rotation heuristic is the best strategy for the BST when the access distribution is skewed.

  In this Chapter, the results demonstrate the superiority of the conditional rotation heuristic when compared to other heuristics considered in this part of the Thesis. The heuristic has the ability to learn when to stop "self-adjusting" whenever the trie does not need further adjustments. A brief report of the new work presented here has been published in the *Proceedings of ACMSE'2005, the 2005 ACM South Eastern Conference*, in Kennesaw, Georgia, in March 2005 [20] and a more extensive journal version has been published in the *Computer Journal* [22].

- **Chapter 4: Dual-Tries (DT)**

  This Chapter generalizes the whole concept of using tries by incorporating the concept of "direction". The Chapter introduces a new data structure, namely the Dual-Tries (DT),

that uses the idea of using two tries, one for the front of string and the other for the rear of the strings. The difference between the DT and the Two-Trie described in [12, 140] will be outlined in this Chapter. However, both structures are common in the goal of decreasing the storage space of the trie. This is achieved by trying to decrease the number of nodes to build the trie, while trying to keep the same search time for the standard trie. This decrease in the number of nodes is a result of making simultaneous use of the common suffixes and the common prefixes of the words stored in the dictionary.

The structure shows great benefits for storing data involving Bio-informatics. For this kind of data, the results demonstrate a superior improvement of 99% saving in the number of trie nodes with respect to the two-trie data structure. The DT structure is also recommended for storing a large number of strings. The results also show an improvement for data containing longer strings, and will possess the same advantages as the two-trie data structure. No results from this Chapter have been published yet.

- **Chapter 5: Notations and Basic-Concepts**

  The main objective of this Chapter is to gather all the important concepts and notations needed in this part of the Thesis. This Chapter does not contain any new results. The Chapter assumes a basic understanding of the design and analysis of algorithms, data structures, basic AI search techniques, and elementary text-based algorithms.

- **Chapter 6: State of the Art: Trie-based Syntactic Pattern Recognition**

  The main objective of this Chapter is to give an overview of the state-of-the-art for the previous work (that we could trace) on approximate string matching. The Chapter starts with a classification of the field, and then gives an overview of each area within this classification. In particular, we will concentrate mainly on dictionary-based approximate string matching techniques, and more specifically, on situations when the dictionary is stored in a trie.

- **Chapter 7: Breadth-First-Trie Based Scheme**

  In this Chapter, we show how we can optimize[1] non-sequential PR computations by incorporating a Breadth-First Search (BFS) scheme on the underlying graph structure. Although the new scheme marginally restricts the types of errors found in Y (like most researchers do); in practice, there is no loss of PR accuracy. The Chapter shows how these searches can be effectively implemented using a new data structure called the Linked List of Prefixes (LLP), applicable to arbitrary inter-symbol distances. The latter permits *level-by-level* traversal of the trie (as opposed to traversal along the "branches"). The BFS-LLP-based algorithm for the syntactic PR of strings has been rigorously tested on three benchmarks dictionaries, and the results have been compared with the acclaimed standard [131], the Depth-First Search (DFS) trie-based technique [137]. The algorithm was tested by recognizing noisy strings generated using the model discussed in [121] on each of these dictionaries. The main contribution is that our new approach can be used for Generalized Levenshtein distances and not for just 0/1 costs, and when the maximum number of errors, K, is not known *a priori*, which is the case where the so called "cutoffs" cannot be used in the DFS-trie-based technique [131, 137]. Additionally, our method is applicable when all possible correct candidates need to be known, and not just the best match. We also show that further improvements can be gained by introducing the knowledge of the maximum number or percentage of errors in Y. All our claims have been verified experimentally and will be explained presently. The work done in this Chapter was published in the *Proceedings of the Joint IARR International Workshops SSPR 2004 and SPR 2004*, in Lisbon, Portugal, in August 2004 [118], and a more detailed journal version is currently being reviewed [117].

- **Chapter 8: A Look-Ahead Branch and Bound Pruning Scheme**

  In this Chapter, we attempt to use the same data structure, the trie, for storing the strings in the dictionary so as to take advantage of the compact calculations for the distance matrix, by utilizing the common paths for the common prefixes [76, 137]. We then introduce a new Branch-and-Bound (BB) pruning strategy that makes use of the fact that the length of the strings to be compared and the maximum number of errors, K, are

[1]Throughout this Thesis we shall use the terms "optimize" and "minimize" to effectively imply the task of reducing the cost function involved. Strictly speaking, although we do not attain the true optimal/minimal state of nature, we submit that we are still "optimizing" (as per the Websters' Dictionary definition of the verb) the underlying cost function.

known *a priori*. We thus propose to apply this new pruning strategy to the trie-based approximate search algorithm, which we call the Look-Ahead Branch and Bound (LHBB) scheme. By using these four features (the trie, BB, look-ahead, and dictionary-based dynamic programming), we can demonstrate a marked improvement, because this pruning can be done before we even start the edit distance calculations. LHBB helps us to search in portions of the dictionary where the word lengths are available, without actually having to partition the dictionary, and at the same time make use of the effective properties of tries. The experimental results presented here show improvements (for small and large benchmark dictionaries) of up to 30% when the costs are of a 0/1 form, and up to 47% when the costs are general. This high improvement is at the expense of storing just two extra memory locations for each node in the trie. Also, if the length of the noisy word is large compared to that of all the acceptable words in the dictionary, i.e., those which can give an edit error smaller than K; the edit distance computations for this noisy word can be totally pruned with only a single comparative test. All of these concepts will be presented in this Chapter, and some of them have been published in the *Proceedings of the 4th International Conference on Computer Recognition Systems CORES'05*, in Rydzyna Castle, Poland, in May, 2005 [19]. The corresponding journal paper is currently being reviewed [17].

- **Chapter 9: Clustered-Beam Search**

  This Chapter[2] shows how we can optimize non-sequential PR computations by incorporating heuristic search schemes used in AI into the approximate string matching problem. First, we present a new technique enhancing the Beam Search (BS), which we call the Clustered-Beam Search (CBS), and which can be applied to any tree searching problem[3]. We then apply the new scheme to approximate string matching when the dictionary is stored as a trie. The trie is implemented as a Linked List of Prefixes (LLP) as shown in Chapter 7. The latter permits *level-by-level* traversal of the trie (as opposed to traversal along the "branches"). The newly-proposed scheme can be used for Generalized Levenshtein distances and also when the maximum number of errors is not given *a priori*. It has been rigorously tested on the three benchmarks dictionaries, and the results have been

[2]Patent applications have been filed to protect the intellectual property and the results contained in this Chapter.

[3]The new scheme can also be applied to a general graph structure, but we apply it to the trie due to the dominance of the latter in our application domain, approximate string matching.

compared with the acclaimed standard [131], the Depth-First Search (DFS) trie-based technique [137]. The new scheme yields a marked improvement (of up to 75%) with respect to the number of operations needed, and at the same time maintains almost the same accuracy. The improvement in the number of operations increases with the size of the dictionary. The CBS heuristic is also compared with the performance of the original BS heuristic when applied to the trie structure, and the experiments again show an improvement of more than 91%. Furthermore, by marginally sacrificing a small accuracy in the general error model, or by permitting an error model that increases the errors as the length of the word increases, an improvement of more than 95% in the number of operations can be obtained. The work done in this Chapter was presented as a plenary talk at *ICAPR2005, the 2005 International Conference on the Advances of Pattern Recognition*, in Bath, UK, in August 2005 [18]. The extended journal version of this work is to be published in the *IEEE Transactions on Systems, Man, and Cybernetics* [23].

- **Chapter 10:Trie-Based Dynamic Matrix Optimization**

  This Chapter introduces a new technique with its associated operations, namely the so-called *Opt-DFS-trie*, for doing the dynamic matrix calculations for edit distances. The new technique utilizes two DP matrices, and is based on the concept of extracting the maximum information of the characters in the noisy string, Y. It thus examines the specific character of the corresponding column to be calculated with respect to the characters of Y, and optimizes on the fact that we are doing calculations against the whole dictionary, H, stored in a trie, T.

  The experimental results presented in this context show improvements obtained by these methods to be up to 48% for the number of operations (minimizations and additions) with small and large benchmark dictionaries. This high improvement is at the expense of just storing a "secondary" dynamic programming matrix. No results from this Chapter have yet been published[4].

- **Chapter 11: Conclusions**

  This Chapter summarizes the work done in the Thesis and gives the final conclusions. The possibilities for future work will also be proposed in this Chapter.

[4]Patent applications will be filed to protect the intellectual property and the results contained in this Chapter.

Part II

Tries for Information Retrieval

Chapter 2

Tries in Data Retrieval: State of the Art

2.1 Introduction

One of the major scopes of this Thesis is to fathom how we utilize the "Trie" data structure in various real-life applications. To enable us to perceive the full potential of tries, in this Chapter, we shall embark on a fairly comprehensive survey of this structure. Although we will not be utilizing many of the variants, we feel that it is still educative to submit such a survey because it will serve as an overview of the field, and as a foundation for the rest of the Thesis.

This Chapter briefly surveys the available results in the design and analysis of the trie data structure and its different variations and applications. The Chapter is organized as follows: Section 2.2 presents the different variations of tries that are found in the literature and includes subsections for each of the variants. Each subsection describes the new variant, the objective for which it is designed, and provides an example for this structure. It also shows how the search is accomplished, mentions its complexity and illustrates the applications for which the structure is designed. Section 2.3 concludes the Chapter.

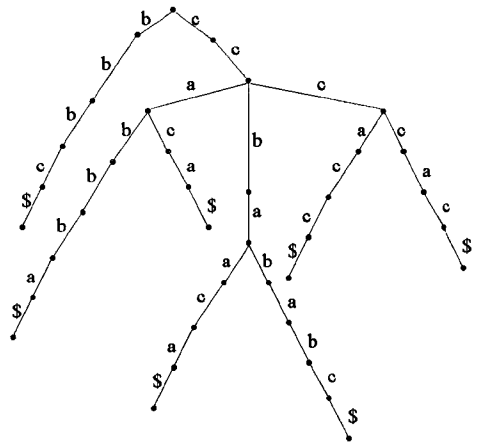


Figure 2.1: An example of the standard trie for the set {bbbc, ccabbba, ccaca, ccbaaca, ccbababc, cccacc, ccccac}.

2.2 Variations on Tries

As mentioned earlier, tries are one of the most general-purpose data structures used in computer science today [28], and can lead to significant enhancements.

The primary challenge in implementing tries is to avoid using excessive memory for the trie nodes that are nearly empty [31]. When it comes to implementation [45], several options are possible depending on the decision structure chosen to guide the descent in the subtrees.

The trie is typically constructed from input strings, where the input is a set of n strings, say $S_1, S_2, \ldots S_n$, where the symbols of each S_i consists of symbols from a finite alphabet, A, and which has a unique terminal symbol which we call \$. Some examples of alphabets are: $\{0, 1\}$ for binary files, the 256-set ASCII set and the 26/52-symbol English alphabet. Tries are not only used to search for strings in normal text, but can also be used to search for a pattern in a picture.

Figure 2.1 shows an example of a trie constructed from the following seven strings for the alphabet $\{a, b, c\}$: $\{bbbc, ccabbba, ccaca, ccbaaca, ccbababc, cccacc, ccccac\}$. The labels on the branches show the character that leads from the previous node to the next node, independent of how the node is implemented.

Algorithm 2.1 Algorithm Generic-Trie-Search

Input: A trie T and a string s to be searched for, whose characters are s_i. The string s is terminated with \$.

Output: A Boolean variable "found" which has a value "True" if s is stored in the trie, T. If it is, it also returns the data A_i stored for string s.

Method:

```
1: t = root
2: i = 1
3: if t = NULL then
4:    Return found = False
5: end if
6: while t ≠ NULL and i ≤ |s| do
7:    t = next(t, s_i)
8:    i++
9: end while
10: {if s has been processed}
11: if i > |s| and t ≠ NULL then
12:    Return {found = True; record A_i}
13: else
14:    found = False
15: end if
16: End Algorithm Generic-Trie-Search
```

The pseudo code for searching for a string in a generic trie is given in Algorithm 2.1.

The general problem of considering m-way branches in a trie can be encapsulated in the function *next* which has two arguments, namely, the node t and the symbol s_i indicating which sub-trie has to be followed. This function returns the sub-trie node or the value NULL, if there is no valid sub-trie node. The way in which the trie node is implemented demonstrates how the *next* function could be implemented.

The trie structure makes use of the representation of keys as a sequence of digits or characters from the alphabet [141, 142]. Searching for a key in the trie can be achieved by examining the key, character by character, and hence this variant of the trie is called the *Digital Search-tree* [141, 142].

In a binary trie, each character is translated into its binary code and the binary code is used as the value of the key. For example, the left arc is labelled with value '0' and the right

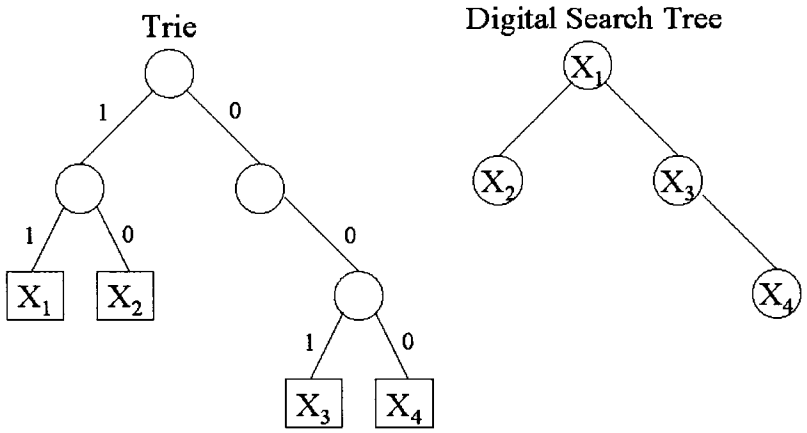


Figure 2.2: A trie and a digital search tree built from the strings $X^1 = 11100\ldots$, $X^2 = 10111\ldots$, $X^3 = 00110\ldots$, *and* $X^4 = 00001\ldots$.

arc with the value '1', which is why the binary trie is called the Binary Digital Search-tree (BDS-tree) [141]. In trie structures the strings themselves are stored in external nodes or leaves while the internal nodes are branching nodes used to direct the search from the substrings to their respective destinations [149]. In a Digital Search Tree (DST) strings are directly stored in nodes and hence external nodes are eliminated. The branching policy used is the same as the one used for tries. Figure 2.2 presents an example (adapted from [149]) which shows the difference between a trie and a digital search tree built from the strings $X^1 = 11100\ldots$, $X^2 = 10111\ldots$, $X^3 = 00110\ldots$, *and* $X^4 = 00001\ldots$.

In [74] Jacquet and Szpankowski studied the complete characteristics of a digital tree from the viewpoint of the depth of the trie when the source that generates the symbols is Markovian. They proved that, asymptotically, as the number of strings n tends to infinity, the average depth becomes $1/h_1 \cdot \log n + c'$, where h_1 is the entropy of the Markovian alphabet, and c' is a constant.

Tries are fast but space-intensive [69]. The high memory usage of tries has long been recognized as a serious problem, and many techniques have been proposed to reduce their memory requirement. Proposals for modified tries that address the issue of high memory usage can be broadly placed in two groups [69]: reduction in trie node *size*, and reduction in the *number* of trie nodes. The different proposals for modified tries, as reported in the literature, will be discussed in the following subsections.

2.2.1 Non-compact Tries

The primary challenge in implementing tries is to avoid using excessive memory for the trie nodes that are nearly empty [31]. When it comes to implementation [45], several options are possible, depending on the decision structure chosen to guide the descent in the subtrees. The literature reports three major choices namely: the "array-trie", the "list-trie" and the "BST-trie".

The "array-trie" (the basic structure of tries) uses an array of pointers to access subtrees directly. Although direct array indexing represents the fastest way to find a sub-trie [28], all the sub-tries must be created as branch arrays with a length equivalent to the cardinality of the alphabet. This solution is adequate only when the cardinality of the alphabet is small (typically for binary strings), otherwise, it creates a large number of null pointers. The space required for arrays is proportional to $|A|^p$, where $|A|$ is the cardinality of the symbol alphabet, and p is the average length of strings. For example, when $|A| = 26$ and if we are dealing with 5-letter keys, it could require 26^5 or about 12 million nodes.

Search in an array-trie is fast, requiring only a single pointer traversal for each letter in the query string. In this case, the *next* function will use the character s_i as a direct index in the array of the current node to access the next node. In other words, the search cost is bounded by the length of the query string. The trie supports the search for any string w by following an access path dictated by the successive letters of w [45]. Figure 2.3 shows the corresponding "array-trie" for Figure 2.1. The dark boxes correspond to the sub-tries for the string stored in the boxes which are depicted in this way just to render the drawings simple. By way of example, only one string "ccaca" is completely shown for all its nodes. Also, none of the null pointers are shown. If we search for the string "ccaca$", we will use the characters as an index for the array nodes as we go along the path until we finish the string including the $. This means that the string exists. If we search for "ccb$", we will reach null pointer before we finish the string, which means that the string is not found in the trie.

The "list-trie" structure remedies the high storage cost of array-tries by linking sister subtrees at the expense of replacing direct array access by a linked list traversal [38].

A list implementation offers space savings if a trie node has only a few children, whenever an array of fixed size would consist largely of null pointers. However, the space savings come at

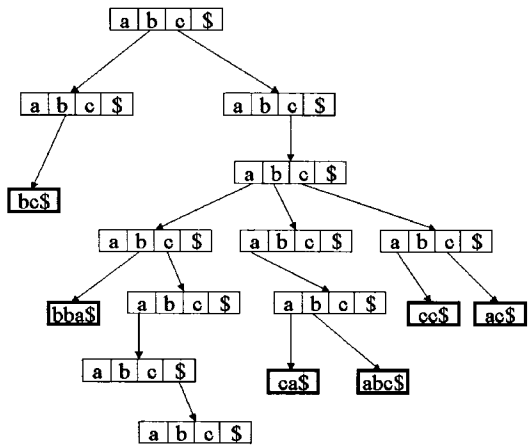


Figure 2.3: The array-trie for the set {bbbc, ccabbba, ccaca, ccbaaca, ccbababc, cccacc, ccccac} corresponding to the trie given in Figure 2.1.

the expense of a linked list traversal in each node instead of a simple array lookup [69]. In this case, the *next* function will use the character s_i to do a sequential search in the list associated with the current node, in order to access the next node. Figure 2.4 shows a typical "list-trie" implementation of the trie. If we search for the string "ccaca$", as in Figure 2.3, we will use the characters to search each list that represents the nodes as we go along the path till we terminate the string including the $. Such a finding shows that the string exists. If we search for "ccb$", we will reach the null pointer before we finish the string, meaning that the string does not exist.

The list implementation was referred to as a "doubly chained tree" by Sussenguth [148]. The node is represented by one computer word divided into three fields. The first field indicates the key element value of the node, the second contains a pointer to the next sibling, and the third contains the address of the first node in the list of its children. Based on theoretical analysis of a list trie, Sussenguth suggested that the expected search time can be minimized when the list trie nodes stop branching when there are less than six keys. Instead of a further branching, the keys should be kept directly in the node in a linked list where they can be sequentially accessed [148]. However, the analysis concerns a slow variant of tries and cannot be applied to array tries.

The "BST-trie" uses binary search trees (BST) as the subtree access method, with the goal of combining advantages of array-tries in terms of the time costs, and list-tries in terms of their

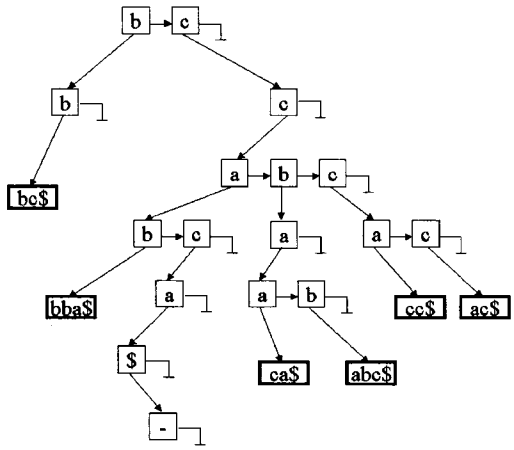


Figure 2.4: The list-trie for the set {bbbc, ccabbba, ccaca, ccbaaca, ccbababc, cccacc, ccccac} corresponding to the trie given in Figure 2.1.

storage costs.

Bentley and Sedgewick [30, 31] proposed the Ternary Search Trie (TST) as an implementation for the BST-trie, where the trie nodes are binary search trees. Consequently, in this case, the *next* function will use the character s_i to search the BST representation of the current node to access the next node. Figure 2.5 shows the corresponding "BST-trie" for the same set of words $\{bbbc, ccabbba, ccaca, ccbaaca, ccbababc, cccacc, ccccac\}$. If we search for the string "ccaca$", we will use the characters to search each BST that represents the nodes as we go along the path till we terminate the string with the $, which means that the string exists. The figure shows the implementation of the "BST-trie" as a TST. In the TST, whenever we find the current character in the string equal to the character stored in the node, we follow the middle pointer to the next BST searching for the next character in the string. If we search for "ccb$" as before, we will reach the null pointer before the string terminates, which means that the string does not exist.

Bentley and Sedgewick [30] analyzed TSTs, and their results can be summarized in the following theorems [1].

Theorem 2.1. *A search in a perfectly balanced TST representing n k-vectors requires at most* $\lfloor \log n \rfloor + k$ *scalar comparisons, and this is optimal.*

Theorem 2.2. *The expected number of comparisons in a successful search in a TST built by*

[1]The proofs of the theorems in this Section are not included here. They can be found in their respective papers.

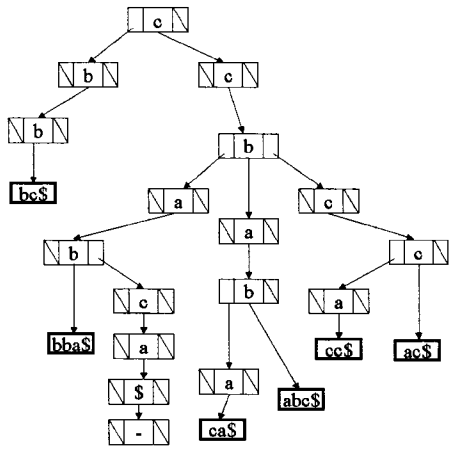


Figure 2.5: The BST-trie for the set {bbbc, ccabbba, ccaca, ccbaaca, ccbababc, cccacc, ccccac} corresponding to the trie given in Figure 2.1.

partitioning around the median of $2t+1$ *selected elements is* $2H_n/(H_{2t+2} - H_{t+1}) + k + O(1)$, *where* $H_n = \sum_{1 \le i \le n} \frac{1}{i}$ *is the harmonic numbers.*

Theorem 2.3. *The number of nodes in a TST is constant for a given input set, independent of the order in which the nodes are inserted.*

Clement *et al.* [45] analyzed and empirically tested these hybrid trie structures. Array tries, list tries, and BST tries are shown to have logarithmic access costs, require linear space, albeit with different constants. They used exact and asymptotic methods to analyze these three trie structures in terms of their time and space complexity for both the memoryless (sometimes called Bernoulli) and Markovian sources. They also conducted a simulation campaign on large real textual data. Their results can be summarized as follows:

- They proved two main theorems for TSTs, that gave an exact and asymptotic analysis of the path length (L) and the search cost (R). These results are cited below.

  Theorem 2.4. *The path length and the (comparison) cost of a random search in a TST made of* n *keys have expectations given by* ($P_{h.[i,j]} = \sum_{k=i}^{j} p_{h.k}$, $p_{h.\alpha} = p_h p_{\alpha/h}$), *and thus*

$$E_n[L] = 2 \sum_{h \in A^*} \sum_{i<j} \frac{p_{h.i} p_{h.j}}{P_{h.[i,j]^2}} [n P_{h.[i,j]} - 1 + (1 - P_{h.[i,j]})^n],$$

$$E_n[R] = 2 \sum_{h \in A^*} \sum_{i<j} \frac{p_{h.i} p_{h.j}}{P_{h.[i,j]}} [1 - (1 - P_{h.[i,j]})^n].$$

where the TST is assumed to be built on a random n-tuple $S = \{s_1, \ldots, s_n\}$ *of infinite strings from the universe of keys* $U = A^*$, A *is the alphabet,* p_h *is the source dependent probability for a random element of* A^* *to start with the string* h, *and* $p_{\alpha/h}$ *is the conditional probability to have a prefix* h *followed by the letter* a_α.

Theorem 2.5. *The (comparison) external path length and random search for a TST built on* n *keys produced by a source* S, *either memoryless* (m) *or Markovian* (M), *have averages that satify:*

$$\begin{aligned} E_n[L] &= \frac{1}{H_s} C_S \cdot n \log n + O(n) \\ E_n[R] &= \frac{1}{H_s} C_S \cdot \log n + O(1), \end{aligned}$$

where the entropy H_S *and the quantity* C_S *are source-dependent constants.*

- They showed that the path length (L^o) and the search cost (R^o) of standard array-tries are [82]:

$$\begin{aligned} E_n[L^o] &\sim \frac{1}{H_s} n \\ E_n[R^o] &\sim \frac{1}{H_s} n \log n. \end{aligned}$$

- The authors proved that the exact and asymptotic methods used for TSTs could be applied to list-tries, using a simplified analysis of linked lists which is also applicable for the analysis of BSTs. In this case, the path length (L^*) and the search cost (R^*) of list-tries obey:

$$\begin{aligned} E_n[L^*] &\sim \frac{C_S^*}{H_s} n \log n \\ E_n[R^*] &\sim \frac{C_S^*}{H_s} \log n. \end{aligned}$$

- Based on the simulation conducted on a large real textual data, the empirical results for the search costs in array-tries, list-tries, and BST-tries can be summarized as:

$$\begin{aligned} Array - tries: \quad E_n[R^o] &\approx 0.8 \log n \\ List - tries: \quad E_n[R^*] &\approx 3.0 \log n \end{aligned}$$

$$BST - tries: \quad E_n[R] \approx 1.0 \log n.$$

- Finally, on the basis of the analytical and experimental work, Clement *et al.* arrived at the conclusion that TSTs are an efficient data structure from an information theoretic point of view, since the search typically costs about $O(\log n)$ comparisons on real-life textual data. List-tries require about three times as many comparisons as a TST that implements the BST-trie. Furthermore, for an alphabet of cardinality 26, the storage cost of TSTs is about nine times smaller than that of standard array-tries. The array-trie implementation has the lowest access cost but the highest memory cost. Their results confirm that TSTs have lower memory requirements than array-tries. In conclusion, TSTs were described by Clement *et al.* [45] as "the method of choice for managing textual data" since, of the three hybrid trie structures, they offer the best time-space trade-off.

Another implementation of the trie is the *double-array-trie* which was proposed by Aoe *et al.* [12]. They achieved this by introducing a new internal array structure, called a *double-array*, thus implementing a trie structure. It utilizes a combination of the fast access of a matrix form, and the compaction of a list form. The paper introduced the retrieval, insertion and deletion algorithms. The structure is briefly described as follows: The trie is compressed into two uni-dimensional arrays BASE and CHECK, and hence its name, the "double-array". Non-empty locations of any node are mapped by the array BASE, into the array CHECK in such a way that no two non-empty locations in each node are mapped to the same position in CHECK. The double-array trie is built from the compact trie for large sets of keys, where the tail of the key that is not needed for further disambiguation is stored in a string array, denoted as TAIL. Figure 2.6 (taken from [12]) shows an example for a double-array trie for the words {bachelor, baby, badge, jar}.

Aoe *et al.* showed that the worst-case time complexity for retrieving a string s becomes $O(|s|)$, where $|s|$ is the length of s. They also empirically tested the double-array structure with respect to the storage and time efficiency for seven data sets. Their results can be summarized as follows:

- The storage of the double-array form, for most of the data sets, was between 7 to 8 per cent less than the storage of the list form.

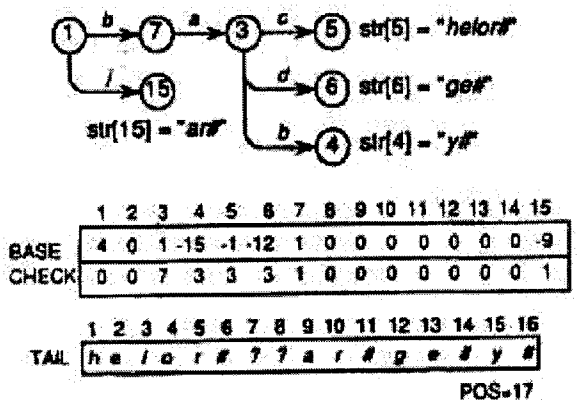


Figure 2.6: The compact trie and the corresponding double-array trie for the words: {bachelor, baby, badge, jar} taken from [12].

- The storage of double-array was just 1.1 to 1.2 times larger than the storage required by the source file of all keys, which included a delimiter between the keys.
- The insertion of the double-array needed from 5 to 9 times more time than the insertion of the list-tries.
- The deletion of the double-array was from 1.2 to 1.5 times in the average case, and from 1.5 to 2.5 times in the worst case, faster than the list-tries.
- The search time within the double-array was from 1.2 to 3.1 times faster in the average case, and from 1.5 to 5.2 times faster in the worst case, than that of the list-tries. The retrieval time of the double-array was constant with respect to the number of keys stored, but for list tries the search time increased as the number of keys that were stored, increased.
- The algorithms presented in [12] were suitable for information retrieval systems in which the frequency of appending keys was higher than that of deleting keys, thus allowing the redundant space created by deletion to be exhausted by the subsequent insertions.
- The trie, based on the double-array structure, was implemented very efficiently, using only about 300 lines of C!

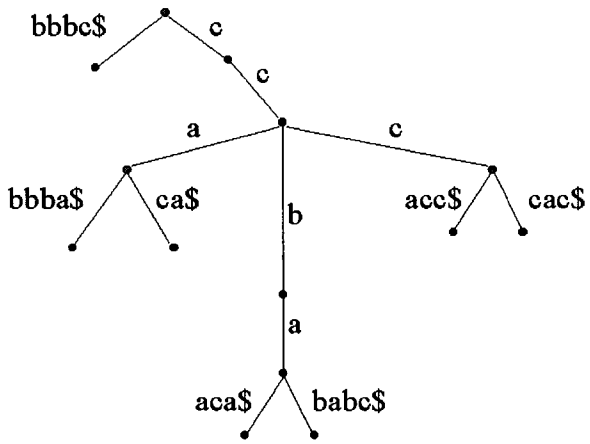


Figure 2.7: The compact trie for the set {bbbc, ccabbba, ccaca, ccbaaca, ccbababc, cccacc, ccccac} corresponding to the trie given in Figure 2.1.

2.2.2 Compact Tries and Patricia tries

The second group of concepts introduced to reduce the size of tries deals with reducing the *number* of trie *nodes*. In a standard trie, all characters of all strings are represented by pointers between the trie nodes. However, in natural-language applications, the trie nodes near the leaf levels tend to be sparse.

A simple way to reduce the number of trie nodes is to omit chains of nodes that have only a single descendant, and thus lead to a leaf. We refer to this variant of a trie, in which tailing "sticks" caused by single-descendant nodes are eliminated [69, 148], as a *compact* trie. Figure 2.7 shows an example of a trie constructed from the following seven strings for the alphabet $\{a, b, c\}$: $\{bbbc, ccabbba, ccaca, ccbaaca, ccbababc, cccacc, ccccac\}$

The number of leaves is $n + 1$, where n is the number of input strings. Furthermore, in the leaves, we may store either the strings themselves or pointers to the strings.

Another form of compaction for tries, which involves reducing the number of nodes, is the so-called PATRICIA-trie, where PATRICIA stands for *Practical Algorithm To Retrieve Information Coded In Alphanumeric*, introduced in 1968 by Morrison [103]. This structure is a variant of tries which eliminates the waste of space caused by having only one descendent, and is achieved by collapsing one-way branches into a single node. The structure finds a number of applications

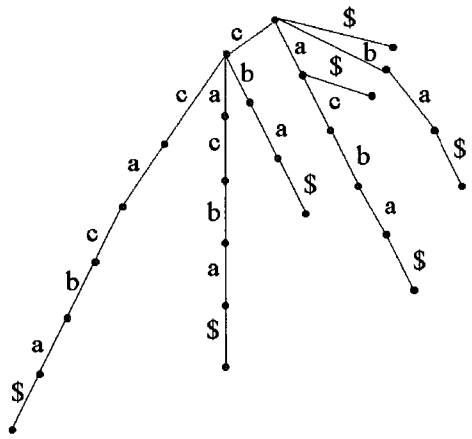


Figure 2.8: a suffix tree constructed from the string "ccacba".

inside suffix trees, making the latter a popular application.

One of the most important applications of tries and Patricia trees is the use of suffix trees and compact suffix trees (also called PAT) [149]. A suffix tree is a trie and a compact suffix tree is a Patricia trie in which the words are all the suffixes of a given string [149]. The suffix tree [66] is a trie in which we store references to all *sistrings* of the test string in a trie. A sistring [11] (semi-infinite string) is a substring of the text, defined by its starting position and continuing to the right as far as necessary to make the string unique. A document of length n may be viewed as if it is consisting of n sistrings, one sistring at each position of the text. Figure 2.8 shows an example of a suffix tree constructed from the string "ccacba". A suffix tree is a data structure that is extremely efficient for string matching, and can be achieved if matched against an entire data set in $O(n)$ time, where n is the length of the text stored in the tree. This data structure is used for a wide range of applications like approximate string matching, compression schemes, and generic sequences, and we will not even attempt to survey the field here, as this is not the primary focus of the Thesis.

Patricia tries [37, 69, 103, 66], which generalize the fundamental idea of collapsing sticks into a single node, are an important variant of a trie that has been widely analyzed [37, 129, 144]. All the unary nodes are collapsed in a Patricia trie, and not merely those that lead directly to a leaf node as in the compact trie. Single-descendant nodes are eliminated by keeping a counter at each branch to indicate which element of the key (for example which character position in

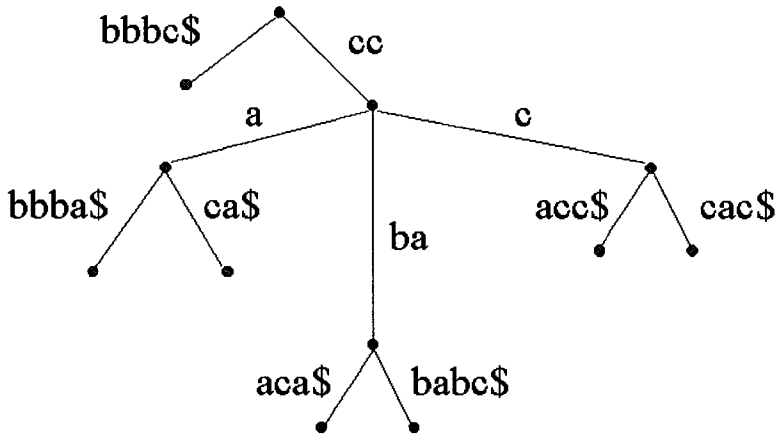


Figure 2.9: The Patricia trie for the set {bbbc, ccabbba, ccaca, ccbaaca, ccbababc, cccacc, ccccac} corresponding to the trie given in Figure 2.1.

a string) determines the next branching. During a search, some character positions can be skipped, since they do not discriminate between strings stored in the next levels of the trie. To avoid false matches, a final comparison has to be performed between the query string and the string found at a leaf node where the search terminates. Alternatively, the substring that is representing the single branch can be stored instead of a counter, as described above, leading to an alternative implementation known as a *compact* Patricia trie. Both implementations lead to a decrease in the number of trie nodes, a saving that is partly offset by a more complex structure and more complex traversals. Sedgewick [136] and Knuth [82] described various techniques for implementing the search and insertion operations in Patricia tries. Figure 2.9 shows an example of a trie constructed from the following seven strings for the alphabet $\{a, b, c\}$: $\{bbbc, ccabbba, ccaca, ccbaaca, ccbababc, cccacc, ccccac\}$.

For the binary Patricia trie, where there are only two symbols in the alphabet, and the number of leaves is n, the number of internal nodes is $n - 1$, and the height of the Patricia trie is bounded by n. Thus, the size of the Patricia trie does not depend on the length of the strings, but only on the *number* of strings. Observe that the height of the ordinary trie or the compact trie are not necessarily bounded by n.

The optimization resulting from the Patricia trie depends on how many nodes have a single descendant. Based on tests conducted on their data set, the authors of [69] reported the following results:

- 31% of all nodes in a compact trie are single-descendant.
- Even with a reduction in the number of nodes to 30%, compared to the compact trie, a Patricia trie would be much larger than a TST for a given collection of words.
- The Patricia trie is more complex than a compact trie, and is thus likely to yield, on average, higher access costs. The single-descendant nodes in a compact trie are rarely visited, although over 30% of the nodes are single-descendant. Thus, there were less than 4% of the nodes that were actually visited during the processing of their data.

The performance of algorithms on tries and Patricia tries depends on the shape of the underlying trees [37]. The number of internal nodes counts the number of pointers needed to store the data structure, whereas the external path length is related to the depth of the insertion of a new word in the trie. The shape itself strongly depends on the *source*, or the *way* the words that are inserted in the structure, are produced. As mentioned earlier, the two most elementary source models are the so-called *memoryless* sources, where symbols in words are each emitted independently of the previous one, and the *Markovian* sources, where the probability of emitting a symbol depends solely on a bounded part of the past history.

The main parameters of Patricia tries have been studied for these classical sources and the results for path length and depth of Patricia tries were obtained in [129, 144].

The size and path length of standard and hybrid tries have also been studied extensively in the context of dynamical sources. Bourdon [37] extended these results to the study involving the parameters of Patricia tries when compared in the same context. Their probabilistic model was the so-called Bernoulli model of size n denoted by (B_n, S), which considers all the possible sets of fixed cardinality n consisting of independent infinite source words. The aim of [37] was to analyze the behavior of the size and the path length of the Patricia trie as the cardinality, n, increased indefinitely.

The results that Bourdon [37] obtained can be summarized as follows:

- In the Bernoulli model (B_n, S) relative to a dynamical source S, the average values of the size $E_n(S_p)$ and the path length $E_n(L_p)$ of Patricia tries built with n words have the

following asymptotic behavior:

$$\begin{aligned} E_n[S_p] &\approx \frac{1}{H_s} n[1 - C_{1S}] \\ E_n[R_p] &\sim \frac{1}{H_s} n \log n - n[\frac{\gamma + C_{2S}}{H_S} - C_{Sf}], \end{aligned}$$

where, H_s denotes the entropy of the source S, C_{1S} and C_{2S} are constants depending solely on the mechanism of the source, while C_{Sf} is a constant depending on both the source and the initial density.

Their results compared favorably with those obtained by Clement *et al.* [46].

- In the Bernoulli model (B_n, S) relative to a dynamical sources S, the average values of the size $E_n(S_p^*)$ and the path length $E_n(L_p^*)$ of standard tries built with n words have the following asymptotic behavior:

$$\begin{aligned} E_n[S^*] &\approx \frac{1}{H_s} n, \\ E_n[R^*] &\sim \frac{1}{H_s} n \log n - n[\frac{\gamma}{H_S} - C_{Sf}]. \end{aligned}$$

Their results exhibited a different asymptotic behavior for Patricia tries and standard tries. They also derived the values of some so-called "correction" terms, namely C_{1S} and C_{2S}, and showed that the average value of the size of the Patricia trie is actually better than the average size of the standard trie.

In [24] Baeza-Yates and Gonnet presented what was probably the first reported algorithms to achieve sublinear expected time in the application of searching for any regular expression in a text using constant or logarithmic expected time for some restricted regular expressions. The main idea motivating their results was to simulate the finite automaton represented by the query over a Patricia trie describing the text. The authors of [24] ran the automaton on all the paths of the Patricia trie from the root to the leaves, stopping when possible. The time savings came from the fact that each edge of the tree is traversed at most once, and that every edge represented pairs of symbols in many places of the text.

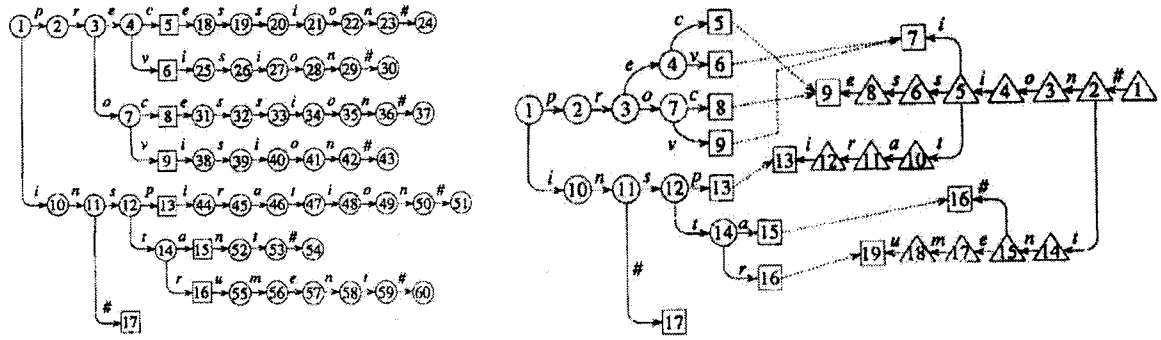


Figure 2.10: An example (adapted from [140]) for the trie and the corresponding two-trie for the words {precession, prevision, procession, provision, inspiration, instant, instrument, in}.

2.2.3 Two-Trie

Aoe *et al.* [64, 101, 140] described an algorithm for the compaction of a trie that is applicable to a dynamic set of keys. They named the new trie structure the "two-trie". The two-trie can be considered as an alternative strategy to reduce the number of nodes needed for the trie. It follows the same concept of the compact trie, as in the previous section, to reduce the number of trie nodes by omitting chains of nodes that have only a single descendant, and thus lead to a leaf.

However, the difference here is the way of storing these chains. The essential idea is to construct *two* tries, one for the front and the second for the rear compressions of keys. This structure and the resulting algorithms were based on maximizing the benefits of strings that share common prefixes *and suffixes*. The rear trie is actually used for storing these chains of nodes that have only a single descendant leading to a leaf. The leaves of the front trie are connected to the leaves of the rear trie *via* a link. The RE trie needs to traverse transitions in the opposite direction for the recognition of a key, and in both directions for updating a key. Figure 2.10 shows an example (adapted from [140] and also given in Chapter 4) for the trie and the corresponding two-trie.

The concepts involved are similar to those used in a Directed Acyclic Word-Graph (DAWG) [90]. It differs from the latter in that the two-trie approach can uniquely determine the information corresponding to keys, which is something that the DAWG cannot. The space savings

of such a two-trie structure compared to a compact trie [69], was reported to be about 20%, whereas the insertion and retrieval time of a two-trie was reported to be similar to that of a (non-compacted) standard trie [140].

We will study this structure in more detail in Chapter 4.

2.2.4 Compact-Balanced Tries (CB-tries)

In a binary trie, the amount of storage required in secondary memory increases with the number of keys to be stored [141, 142]. Nicodeme *et al.* [110] referred to a method proposed by Kouacou-Kouadio to compress the trie into a *Compact trie* representation by using a bit-map, based on the performance of a scan along the nodes of the trie with a preorder traversal. The Compact Trie representation is composed of a bit-map and a pointer-list. The bit-map is the sequence of 0's and 1's obtained by the preorder traversal of the trie. The pointer list associates a pointer to each leaf of the trie. The drawback of the Compact tries [110] is that it requires sequential processing for all nodes along the bit-map and the pointer-list, which, in turn, has a negative impact on the processing time as the number of keys increases. Also, updating the structure implies locking the whole structure, which is a serious drawback for parallel processing.

The Compact Trie based on a bit-map representation of the tries, is simple and powerful, although it is not a segmented structure [110]. Nicodeme *et al.* presented the Compact-Balanced Tries (CB-Tries) [110] which is a natural consequent of the Compact Trie. They showed how to split the compact trie in a segmented and flexible structure of B-tree type. A trie is split into pieces and each piece is represented in a compact way, by a node comparable to a B-tree node. An edge key is generated at each trie or subtrie splitting, and a corresponding starting depth is calculated, where the edge key is the key value of the splitting point. The edge depth indicates the number of bits of the binary representation of the edge key that have to be taken into account before returning back to the ordinary processing of the bit-map. The data structure used to handle the CB-Tries is quite analogous to the one useful for handling the Compact trie. Figure 2.11 (taken from [110]) shows an example of a Compact trie and the corresponding CB-trie.

The conclusions obtained from Nicodeme *et al's* paper [110] can be summarized as follows:

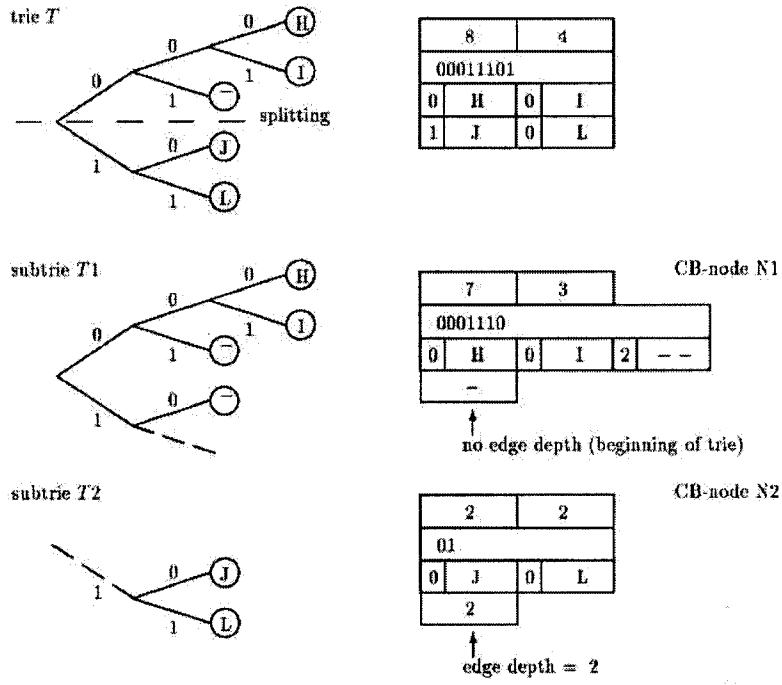


Figure 2.11: An example of a Compact trie and the corresponding CB-trie and the corresponding nodes.

- CB-tries require between one and two bytes to represent a key.
- At the block-node level of the structure, a binary search operation is not possible, and only sequential processing is permitted. The sequential-processing is done locally and performed using in-core processing.
- The price paid for the excellent compaction results is the poor performance of the bit-map handling, and the increase in the complexity of the algorithms for insertion, deletion and searching in the trie.
- CB-tries could lead to efficient usage in a parallel machine.
- CB-tries offer compactness and all B-tree properties, and can be a secure choice for implementation of memory databases.
- CB-tries can also manage the clustering of ordered indices without any special difficulty.

- The worst case of insertion corresponds to pairs of keys having their N first bits in common. In this case, the number of bits used to represent these keys, in the bit-map representation, is approximately $2N$. Therefore, if N is large, and if there is large number of such pairs of keys, a Patricia-tree-like method would locally provide a greater space saving than the bit-map representation. In the worst case, this would require about the same amount of memory in a CB-trie as in a classical B-tree.

2.2.5 Compact Binary Tries

This method, as we see, is very closely related to the method described in the previous section for Compact-Balanced tries. It involves the same idea of splitting the trie in order to decrease the bit map representation of the whole trie. The difference involves the bit map representation and the splitting process. In [141, 142], the authors referred to a method proposed by Jonge *et al.* [75] to compress the binary trie into a compact bit stream by traversing the trie in a pre-order manner. This compact bit stream is called the pre-order bit stream which consists of three elements: *treemap*, *leafmap* and *B_TBL*. *Treemap* represents the state of the tree and can be obtained by a pre-order tree traversal, emitting '0' for every internal node and '1' for every leaf visited. *Leafmap* represents the state (dummy or not) of each leaf and can be obtained by traversing the tree in pre-order, and so if the leaf is a dummy, the corresponding bit is set to a '0', otherwise the bit is set to '1'. B_TBL stores each of the bucket addresses.

The problem with this method is that the cost of searching and updating keys in large key sets is high. In [141, 142], the authors proposed a new method by which they are able to avoid the increased time, even though the dynamic key sets are extremely large for these hierarchical structures. This method separates the binary trie into smaller binary tries of certain depth. These separated tries are numbered and connected by pointers. The authors of [141, 142] refer to this binary trie (separated in this manner) as the Hierarchical Binary Digital Search tree (HBDS-tree). The pre-order bit stream is created and controlled for each of the separated tries. By using this improved method, each process can be enhanced, because unnecessary scanning of the pre-order bit stream for each separated trie can be omitted.

The authors of [141, 142] also applied the new method for the three binary trie organizations explained earlier, namely the standard trie, the compact trie, and the Patricia trie studied in

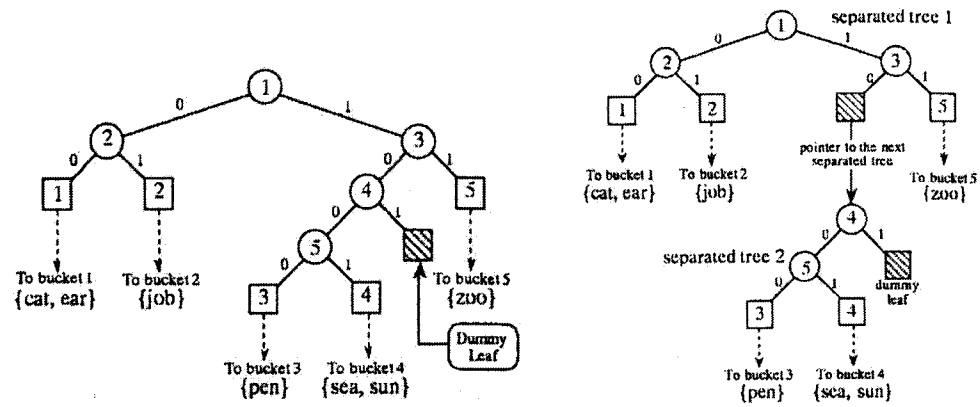


Figure 2.12: The BDS-tree for a set of keys and the corresponding HBDS-tree. This figure is taken from [141].

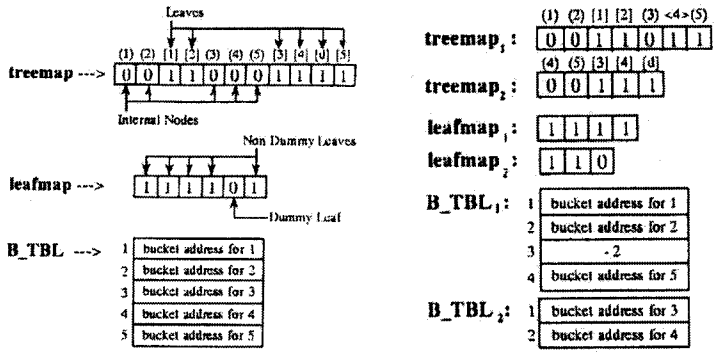


Figure 2.13: The pre-order bit stream for the BDS-tree and the HBDS-tree corresponding to Figure 2.12. The figure is taken from [141].

Sections 2.2.1 and 2.2.2. The new trie structures that they proposed, store no pointers and require only a single bit per node in the worst case. Figure 2.12 shows a BDS-tree for the set of keys {cat, car, job, pen, sea, sun, zoo}, as shown in [141]. Figure 2.13 shows the corresponding pre-order bit stream, also taken from [141].

A Patricia trie [143] gives the shallowest trie by eliminating all single descendant nodes. However, as the number of registered keys increases, the storage required also increases. Figure 2.14 (adapted from [143]) shows an example of the Patricia BDS-tree. In [139, 143] the authors applied a new pre-order bit stream compression method for the Patricia trie, which is essentially the previous method, with some changes in one of the elements of the bit stream. They called

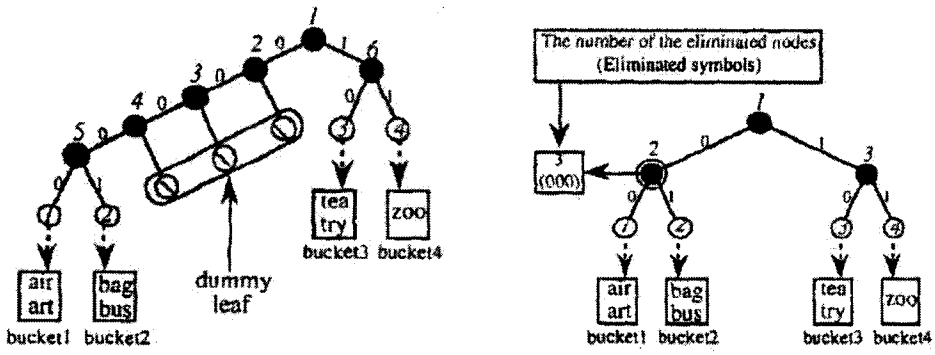


Figure 2.14: An example of the Patricia BDS-tree. (Left) The ordinary trie. (Right) The Patricia BDS-tree. The figure is taken from [143].

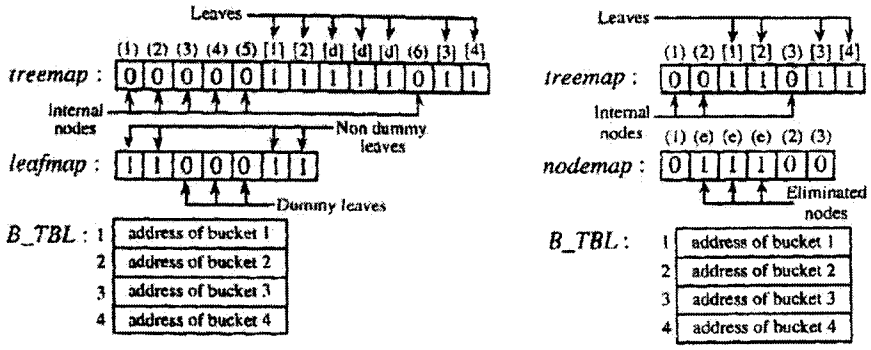


Figure 2.15: The pre-order bit stream for the ordinary trie and the Patricia BDS-tree corresponding to Figure 2.14. The figure is taken from [143].

the resulting trie the compact Patricia trie. The new pre-order bit stream for Patricia BDS-trees consists of *treemap*, *B_TBL* and *nodemap* instead of *leafmap*. *Treemap* and *nodemap* are obtained in the same way as in Jonge's methods [75]. *Nodemap* is of the same length as the number of internal nodes in the ordinary BDS-Tree. Figure 2.15 (also adapted from [143]) shows the corresponding bit stream presentation for Figure 2.14.

In [139], the authors compared the space and time effectiveness of each pre-order bit stream based on the ordinary and the Patricia BDS-trees assuming the following scenario. They assumed that the ordinary BDS-tree has the structure of the complete tree, and that the maximum depth of the complete tree is h. Moreover, the ratio of dummy leaves stored in the complete ordinary BDS-tree is $D \times 100\%$, where $0 \leq D \leq 1$. The theoretical and experimental results presented in [139] can be summarized as follows:

- $treemap_o$, described for the Ordinary BDS-tree, requires $2^{h+1}-1$ bits, since it is composed of the whole nodes of the complete ordinary BDS-tree.

- $treemap_p$, described for the Patricia BDS-tree, requires $(1-D)2^{h+1}-1$ bits, since it is composed of the number of all nodes in the complete ordinary BDS-tree minus twice the number of dummy leaves in the complete ordinary BDS-tree.

- $leafmap$, described for the Ordinary BDS-tree, requires 2^h bits, since it corresponds to the number of leaves in the complete ordinary BDS-tree.

- *nodemap*, described for the Patricia BDS-tree, requires 2^h-1 bits, since it is the number of internal nodes in the complete ordinary BDS-tree.

- The "rate of decrease in *treemap*" is $D+\frac{D}{2^{h+1}-1} \approx D$, where $D/(2^{h+1}-1) \longrightarrow 0$ as h becomes large. It can thus be seen that as the number of dummy leaves in the ordinary BDS-tree increases, the rate of decrease in *treemap* also increases, leading to a smaller *treemap* for the Patricia BDS-tree.

- As for the time complexity, with regard to the search of the ordinary BDS-tree, in the worst case, the entire tree must be scanned. Thus, the complexity is $O(2^h)$. Regarding the update and deletion, the worst scenario is the case where the leaf to be processed is located at the leftmost position, and so the complexity is $O(2^h-h)$.

- The time efficiency of each process associated with the Patricia BDS-tree becomes ($D\times 100$) times faster than the ordinary BDS-tree, and thus the complexity for search in the Patricia BDS-tree is $O(D2^h)$, and the cost for the update and deletion are $O(D(2^h-h))$.

- If the key set becomes large and the value of h becomes large, the time efficiencies of the Patricia BDS-tree and the ordinary BDS-tree are lowered. By using the improvement method of separating the tree structure hierarchically [141, 142], the time efficiency can be greatly improved.

- The experimental results for comparing the Patricia BDS-tree and ordinary BDS-tree confirm the theoretical results and demonstrate that the Patricia BDS-tree is more compact than the tree created by the method of Jonge *et al.* [75]. The storage requirement to register a single key is between 2.4 and 2.5 bits, and thus the Patricia BDS-tree can lead

to more compact indices than the B-tree and B+-tree. The average retrieval time can be improved by the hierarchical method.

In conclusion, the theoretical and experimental results, showed that this method generates 40-60 percent shorter new bit-streams than the traditional bit streams proposed by Jonge *et al.* [75].

2.2.6 Level-Compressed Trie (LC-trie) and Level-Path-Compressed Trie (LPC-trie)

Andersson *et al.* [9, 10] introduced and analyzed a new method to reduce the search cost in tries. They called the new structure the "level-compressed-trie", or LC-trie. The LC-trie inherits the good properties of binary tries with respect to neighbor and range searches, while the external path length is significantly decreased. The method is briefly described as follows: The i highest complete m levels of a trie are replaced by a single node of degree m^i; the compression is repeated in the sub-tries and the replacement is made top-down. All the strings stored are assumed to be prefix-free, i.e. no string is a prefix of another string, which could be easily done by adding a string terminator. Figure 2.16 (adapted from [9]) shows an example for a trie and the corresponding LC-trie.

The following results can be shown to be true for LC-tries [9, 10]:

- For uniformly distributed data, the expected average depth of an LC-trie is $\Theta(\log^* n)$, where $\log^* n$ is the iterated logarithm function, $\log^* n = 1 + \log^*(\log n)$, for any $n > 1$, and $\log^* n = 0$, otherwise. As opposed to this, the expected average depth of a trie containing n independent random strings is $\Theta(\log n)$.
- For the Patricia trie, the expected average cost is still $\Theta(\log n)$. The level compression technique gives a much better improvement over the Patricia technique, at least for uniformly distributed data.
- The LC-trie improves on the uncompressed and path-compressed tries for other large classes of distributions.

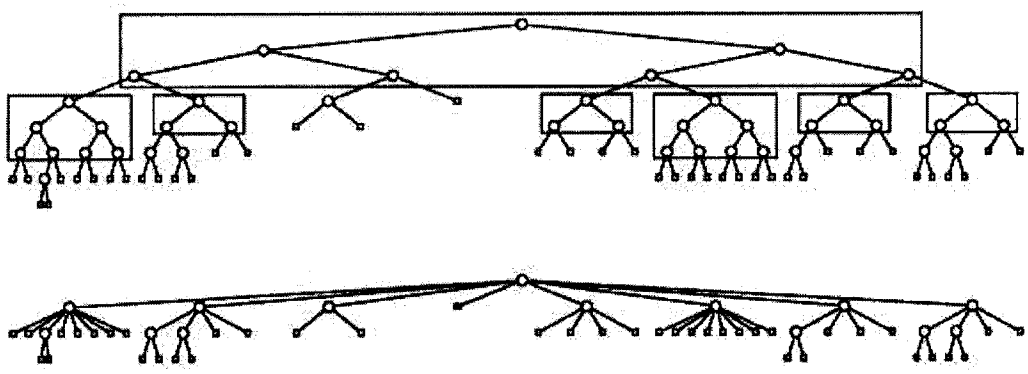


Figure 2.16: A binary trie and the corresponding LC-trie. Compressed levels are marked by rectangles. This figure was adapted from [9].

- In particular, for an independent random sample from a Bernoulli-type process with character probabilities which are not all equal, the average expected depth of the LC-trie is $\Theta(\log \log n)$, and in certain restricted cases as low as $\Theta(\log^* n)$.

In [111, 112] Nilsson *et al.* proposed and analyzed a related dynamic variant of the LC-trie which is the Level Path Compression LPC-trie. These are digital binary tries that use level compression in addition to the path compression of Patricia tries. Figure 2.17, obtained from [111], shows an example for a trie and the corresponding LPC-trie.

The LPC-trie is well suited for modern language environments with efficient memory allocation and garbage collection. One of the difficulties [111, 112] when implementing a dynamic compressed trie structure is that a single update operation may cause a large and costly restructuring of the trie. The authors of [111, 112] showed how to make the restructuring in a dynamic trie more efficient by introducing a relaxed criterion for level compression. This relaxed level compression also reduces the depth further, while using a similar amount of memory. They relaxed the criterion for level compression by allowing the compression to take place even when a subtrie is only partly filled. The price they paid is the potential increased storage requirements.

The experimental results included in [111, 112] for compression methods of dynamic tries, showed that:

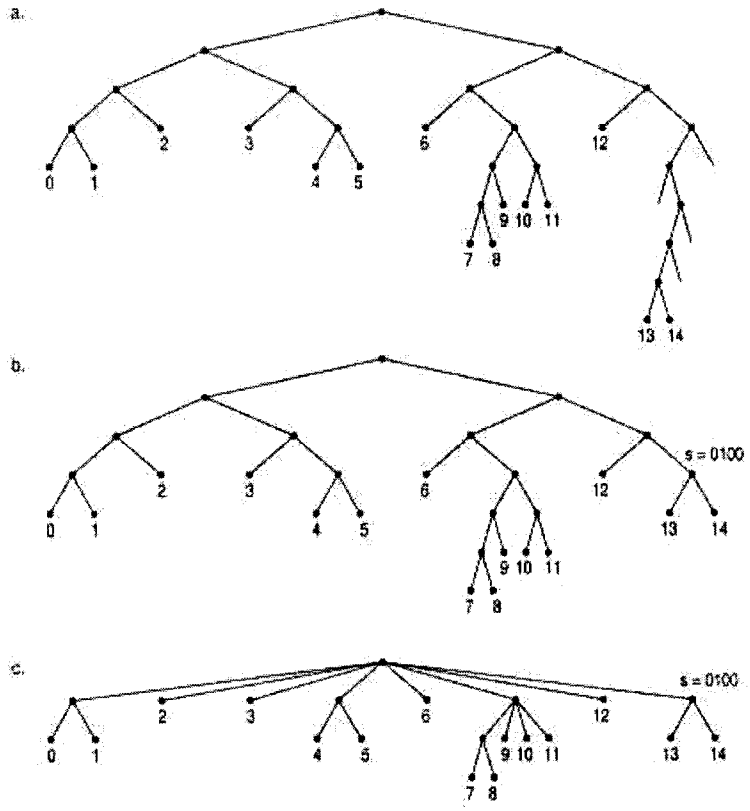


Figure 2.17: (a) A binary trie; (b) A path compressed trie (Patricia tre); (c) The corresponding LPC-trie. This figure has been adapted from [111].

- The average depth of the LPC-trie is much less than that of the balanced BST, resulting in better search times.
- The time to perform the update operations and the space requirements were similar to those of the BST.
- The LPC-trie is a good choice for an order preserving data structure when very fast search operations are required.
- The disadvantage of level-compressed tries is that the level-compressed nodes may have large out-degrees, which makes functional updates extremely expensive.

In [11], Nilsson *et al.* discussed how the suffix tree can be used for string searching. They

showed how an array implementation combined with binary encoding, data compression, path compression, and level compression can be used to build an efficient, compact and fast implementation of a suffix tree. They implemented the suffix tree as a level-compressed Patricia tree. It was shown in [11] that such a structure could be constructed in $O(n \log |A|)$ time and $O(n)$ space, where n is the size of the text, and $|A|$ is the alphabet size.

The experimental results done in [73] for compression methods for functional tries, stated that the path-, width-, and level-compressed trie is the ideal choice for a functional main memory index structure. The experiments showed that PWL-trie was the best with respect to the trie size, the average path length, the copy cost, and the search and update performance when compared with different kinds of compression methods for tries. The goal of level compression was to reduce the depth of the trie, and this can be applied to a path- and width-compressed trie as long as the cost of copying a node is reasonable, as in functional tries. An update may lead to copying the entire search path from the root to the leaf subject to the update.

2.2.7 Order-Containing Trie (O-trie)

Another approach to reduce the size of a trie is to change the order in which the letters of a string are tested. Comer and Sethi viewed a string of length n as a tuple of n attributes [50, 48]. The idea they introduced was to change the order of attributes to influence the size of the result trie. Unfortunately, the problem of determining which order of attributes leads to the smallest trie is NP-complete [50, 48] and is only useful for a static set of keys. Comer and Sethi [50, 48] proposed heuristics to minimize a static trie by locally optimizing the order of testing the attributes. He proposed the order-containing trie (or O-trie) as a space-efficient implementation of a trie. The name came from the fact that the order of testing the attributes had to be contained in the trie itself. A static trie was built and then the attribute testing in subtrees was reordered to reduce the size of the trie.

In [49] Comer proposed a "greedy" heuristic for constructing low-cost tries. The heuristic was built on choosing the attributes in an order which minimizes the number of nodes in each level. But clearly, that achieved only a local optimization and did not guarantee a global optimization. This method requires at most $O(n|s|^2)$ time for computation. However, the greedy method may perform badly, because it cannot guarantee success.

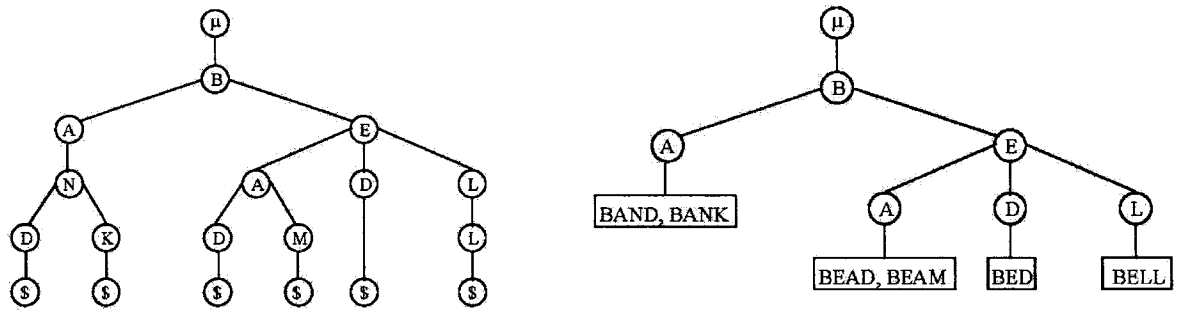


Figure 2.18: The variable-depth trie for the words; {band, bank, bead, beam, bed, bell}. (Left) the standard trie. (Right) the corresponding variable-depth trie. This figure is adapted from [130].

In [50], the authors have demonstrated that:

- Determining tries that are minimal in terms of storage space is NP-complete for all alphabet sizes $|A|$, $|A| \geq 2$ for full tries, and $|A| \geq 3$ for compact tries, Patricia tries, and compact O-tries.
- Determining a minimal average access time is NP-complete for all alphabet sizes $|A| \geq 2$.

2.2.8 Variable-Depth Trie

A variable depth trie index consists of a set of chains from the root whose lengths may vary [130]. The tabular implementation for the trie is the fastest implementation because it provides a direct link between a node and all its children. However, a trie index requires considerable memory space, because each node is assigned a column in the trie matrix, and there could be several null pointers. The idea of a variable depth trie index is illustrated in Figure 2.18 adapted from [130]. The left hand side of the figure shows the original trie and the right hand side shows the corresponding Variable-Depth trie for the words:{band, bank, bead, beam, bed, bell}.

In the variable-depth trie, a dictionary is partially indexed such that each chain from the root identifies a subset of the words, all having the same prefixes. The number of links in the chain is called the *length* of the chain, and it indicates its indexing depth. A variable-depth trie index

consists of a set of chains from the root whose lengths may vary. Thus determining whether a word exists in the trie or not involves traversing a chain from the root to reach a smaller set of words in which a binary or Fibonacci search can be performed. The main objective in the variable-depth Trie index construction is to have an index within the available memory space such that the dictionary is partitioned into subsets that are as small as possible.

The experimental study conducted in [130] showed that:

- Searching dictionary files using variable-depth tries can be as much as six times faster than using a pure binary search strategy.
- The use of a trie structure leads to a greater reduction in the search time from that of the pure binary search when the size of the file grows large.
- The investigations on optimization problems involving the trade-off between space and search time in real-world systems employing dictionary files, permitted the development of a taxonomy of variable-depth trie index optimization problems. This could serve as the basis for appropriate modelling of the trie index problem and the construction of a suitable trie index to a dictionary file, given the constraints and the objectives in a real-world situation.
- The savings in time was essentially due to an efficient partitioning of the file into subsets for binary search, thereby reducing the time spent in binary search, which accounted for a significant fraction of the total search time.
- Constructing a trie index using a variable-depth trie method needs an appropriate parameter setting for the index construction, and requires a sensitivity analysis on the parameter settings before the appropriate values can be chosen.

2.2.9 *b*-Trie

One of the many other variations of the trie is the *b*-trie [81, 149]. Szpankowski [81, 149] proposed the *b*-trie in which a leaf is allowed to hold as many as b strings. The idea is very similar to the variable-depth trie presented in the previous Subsection. Figure 2.19 presents an example of a 3-trie from [81].

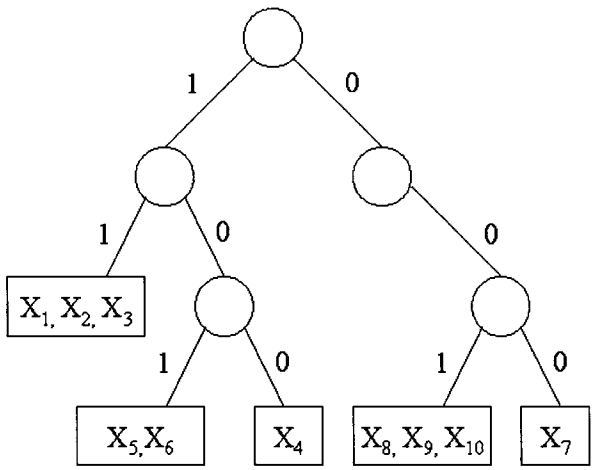


Figure 2.19: A 3-trie from [81] built from the strings $X^1 = 11000\ldots$, $X^2 = 11100\ldots$, $X^3 = 11111\ldots$, *and* $X^4 = 1000\ldots$, $X^5 = 10111\ldots$, $X^6 = 10101\ldots$, $X^7 = 00000\ldots$, $X^8 = 00111\ldots$, $X^9 = 00101\ldots$, $X^10 = 00100\ldots$

Many applications for which the b-trie is useful are listed in [81]. The b-trie is particularly useful in algorithms for extendible hashing in which the capacity of a page or other storage unit is b. The b-trie is also useful in applications of lossy compression based on an extension of the Lempel-Ziv lossless schemes. In these applications the parameter b is large and may depend on the number of strings stored in the trie.

The analytical work of Knessl and Szpankowski in [81] explored the height of tries for strings under different distributions, and include the case where multiple strings are held at a single trie leaf. They established bounds on the height of these tries and showed that under a Markovian model these structures will be relatively flat, with no branches exceeding $C \log n$ in length, where n is the number of strings, and C is determined by the distribution.

2.2.10 Burst Trie

The burst trie [69] is a recent variant of the trie, mainly constructed for applications that depend on the efficient management of large sets of distinct strings in memory. One of these applications is the construction of indices for text databases where a record is held for each distinct word in the text. The memory requirement for the burst trie is no more than the memory requirement

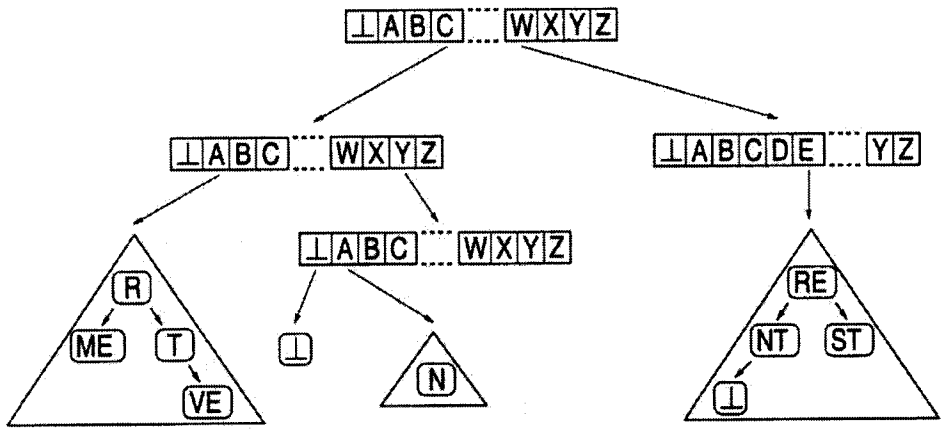


Figure 2.20: An example of a burst trie (adapted from [69]) with BSTs used to represent containers.

for a binary tree, and it is as fast as a trie, even though it is not as fast as a hash table. The burst trie maintains the strings in a sorted or nearly-sorted order.

The authors of [69] describe burst tries and explore the parameters that govern their performance for a large data set. They experimentally determine good choices of parameters, and compare burst tries to other structures used for the same task, with a variety of data sets. These experiments show that the burst trie is particularly effective for the skewed frequency distributions common in text collections, and dramatically outperforms all other data structures for the task of managing strings while maintaining the sorted order. In [68] the authors also tested the performance of burst trie for a small data set, and conclude according to the work in [68] and [69] that the burst trie is suitable for managing sets of hundreds to millions of distinct strings, and for the input of hundreds to billions of occurrences efficiently.

Little work has been done to achieve the self-adjusting of tries based on the underlying access distribution. To the best of our knowledge, the only work pertaining to this, is the burst tries [69, 158] which starts with a single container (implemented in [69, 158] as a BST with a move-to-front heuristic). When the number of nodes in the container starts to be large, (as per a pre-defined criterion), it "bursts" to form a node of the trie that points to smaller containers, and so on. Figure 2.20 (from [69]) shows an example for a burst trie with BSTs used to represent the respective containers.

Although it is an elegant data structure which is very efficient for short strings with skewed distributions, the disadvantage of the burst trie, as we see, is that it needs a number of parameters which are to be adjusted. These may have to be varied according to the data for which the burst trie is used. Also, burst tries do not seem to work efficiently if the lengths of the strings are large. Indeed, the advantage of burst tries is mainly based on the fact that words most commonly used are short. Burst tries are also related implicitly to tries when multiple strings are held at a single trie leaf, as can be seen from the b-trie [149] discussed in Section 2.2.9.

2.2.11 Cache-Efficient Tries

In [1], Acharya *et al.* presented a new trie structure where the structures of the trie nodes consider the cache characteristics of the system. The idea is that they use different data structures to represent different nodes in a trie, and the changes of the fanout at a node are adapted by dynamically switching the data structure to represent the node. The size and the layout of individual data structures are determined based on the size of the symbols in the alphabet as well as characteristics of the cache(s). When taking cache characteristics into consideration, their algorithms out-perform alternatives that are otherwise efficient. These algorithms derive their performance advantage primarily by making better use of the memory hierarchy, and by invoking the node adaptation only during insertion.

Acharya *et al.* [1] classified the prior work on adapting trie structures elegantly into four approaches listed here: The **first** approach focuses on reducing the number of instructions executed by reducing the number of nodes and levels in the trie. The **second** approach views tries as collections of m-way vectors and focuses on reducing the space used by these vectors using a list-based representation or a sparse-matrix representation. The **third** approach focuses on the data structures used to represent trie nodes with the goal of reducing both the space required and the number of instructions executed. **Finally**, more recent results consider optimizing tries used for address lookups in network routers. The algorithms proposed by these researchers focus on reducing the number of memory accesses by reducing the number of levels in the trie and the fanout at the individual nodes. The approach in cache-efficient tries is closest to these approaches because they share the goal of reducing memory accesses. There are, however two main differences:

- First of all, these algorithms assume that the"symbols" to be matched at each level (so to speak) are not fixed, and that the strings that constitute the trie can be arbitrarily subdivided. This assumption is valid for applications that they focus on, namely those involving the IP address lookup in network routers. Additionally, for cache-efficient tries, they assume a fixed alphabet which is applicable to most other applications for which tries are used.

- Secondly, these algorithms focus on restructuring the trie, while cache-efficient tries focus on selecting the data structure for the individual nodes.

2.3 Conclusion

This Chapter surveyed the different variations of tries, from a fairly comprehensive literature survey. For each variant in the Chapter, we described its form, the objective for which it was designed, and provided an example for this structure. It also showed how the search was accomplished, mentioned its complexity, and illustrated the applications for which the structure was designed.

As can be seen from this survey, the work concerning tries is extensive, and also on-going. The importance of tries for the wide range of applications in which it can be used has also been mentioned. In this Thesis we shall concentrate on the applications of tries for information retrieval (exact matching) and for syntactic pattern recognition (approximate matching).

Chapter 3

Self-Adjusting Ternary Search Tries

3.1 Introduction

The primary challenge in implementing tries[1] is to avoid using excessive memory for the trie nodes that are nearly empty [31]. When it comes to implementation [45], several options are possible depending on the decision structure chosen to guide the descent in the subtrees. The space can be saved by changing the representation of the trie node. One way to implement tries efficiently is to represent each node in the trie as a Binary Search Tree (BST). When the nodes are implemented as BSTs, the trie is called "Ternary Search Tree (TST)", as explained in Chapter 2.

A *Ternary Search Trie* (TST) is a highly efficient dynamic dictionary structure applicable for strings and textual data. The strings are accessed based on a set of access probabilities and are to be arranged using a TST. We consider the scenario where the probabilities are not known *a priori*, and are time-invariant. Our aim is to adaptively restructure the TST so as to yield the best access or retrieval time. There are numerous methods that have been proposed in the case of lists and BSTs, but in the case of the TST, currently, the number of reported adaptive schemes are few.

[1]The new work presented here has been published in the *Proceedings of ACMSE'2005, the 2005 ACM South Eastern Conference*, Kennesaw, Georgia, March 2005 [20] and a more extensive journal version has been published in the *Computer Journal* [22].

More specifically, we consider the problem of performing a sequence of access operations on a set of strings $S = \{s_1, s_2, ..., s_N\}$. We attempt to solve this problem by representing the set of strings by using the above mentioned TST [45]. We assume that the strings are accessed based on a set of access probabilities $P = \{p_1, p_2, ..., p_N\}$, and are to be arranged using the TST. We also assume that P is not known *a priori*, and that it is time-invariant[2]. In essence, we attempt to achieve efficient access and retrieval characteristics by not performing any specific estimation process.

In this Chapter, we demonstrate how we can apply two self-adjusting heuristics that have been previously used for BSTs, to TSTs. These heuristics are, namely, the **splaying** [3, 145] and the **Conditional Rotation** [3, 44] heuristics. We have also applied another balancing strategy used for BSTs, to TSTs, which is the basis for the **Randomized** search trees, or Treaps [13, 53, 154]. In an earlier paper [145], Oommen and his co-authors showed that the conditional rotation heuristic is the best for the BST when the access distribution is skewed. In this Chapter, the results demonstrate that the conditional rotation heuristic is the best when compared to other heuristics that are considered in this part of the Thesis. The heuristic has the ability to learn when to stop "self-adjusting", i.e., whenever the tree will not benefit by further adjustments.

In [69], Heinz *et al.* have shown through experimentation that TSTs are slower and less compact than *hash* tables and *Burst* tries. As opposed to this, our intention is to examine different balancing and self-adjusting heuristics that were previously used for the BSTs, and to apply *them* to TSTs. The advantage of this is to allow the TST to adapt itself according to the data without any parameter adjustments. The principal objective of our study is to demonstrate the improvements gained by different heuristics when applying them to TSTs, and to compare this with the original TSTs.

The growth in cache size and CPU speed has led to the development of cache-aware algorithms, of which a crucial feature is that the number of random memory accesses must be kept small. All the papers we consider here possess the same framework, namely that they don't

[2]The body of literature available for non-stationary distributions is scanty. Little work has been done for lists where the distribution changes in a Markovian manner. Otherwise, almost all the work for non-stationary environments involves the non-expected case analysis, namely the amortized model of reckoning. Since this is not the primary focus of this Chapter, we believe that, in the interest of brevity, and of being concise, it is better to not visit these aspects here.

consider caching. The main aim of our heuristic is to try to have the most likely items in the "top" nodes of the TST, which could then be kept in the cache so that the number of random memory accesses will be small. From that perspective, the data used in our experiments should really to considered to be of a "moderate" size, namely, that of a size in which the data can be entirely cache resident.

The organization of this Chapter is as follows: Section 3.2 briefly reviews the TST, the main structure that we will use, and describes how the rotation operation is achieved in the TST. Section 3.3 gives a brief literature survey about the self-adjusting heuristics used for BSTs that we will apply to the TST. Section 3.4 describes, in detail, how each of these methods can be applied to the TST. Section 3.5 gives the experimental setup and the different data sets used in the experiments, describes the experiments done, and presents a comparative survey of results. Section 3.6 briefly explains other heuristics that have been applied to the BST and that can be applied to the TST. Section 3.7 concludes the Chapter.

3.2 Ternary Search Tries (TSTs)

TSTs are efficient and easy to implement. They offer substantial advantages over both BST and digital search tries. They are also superior to hashing in many applications for the following reasons:

- A TST [30, 45] may be viewed as an efficient implementation of a trie that gracefully adapts to handle the problem of excessive memory for tries at the cost of slightly more work for the "full" nodes.
- TSTs combine the best of two worlds: The low space overhead of BSTs and the character-based time efficiency of tries.
- TSTs are usually substantially faster than hashing for unsuccessful searches[3].
- TSTs grow and shrink gracefully; hash tables need to be rebuilt after large size changes.
- TSTs support advanced searches, such as partial-matches and near-neighbor searches.

[3]This claim depends on properties such as the load factor of the hash table.

- TSTs support many other operations, such as traversal, so as to report the items in a sorted order.

As mentioned in Chapter 2, the number of nodes in a TST is constant for a given input set, independent of the order in which the nodes are inserted [30]. The time for an access in the TST [145] is bounded by the length of the string searched for, plus the number of left and right edges traversed. Thus, to minimize the access time we want to keep the depths of the so-called "solid subtrees" small. Using this, we can extend the standard BST techniques to the TST, as we will see in the next sections.

3.2.1 Rotation in TST

TSTs are amenable [145] to the same restructuring primitive as BSTs, namely the *rotation* operation, which takes $O(1)$ time, and which rearranges left and right children but not their middle children. This operation has the effect of raising (or promoting) a specified node in the tree structure while preserving the lexicographic order of the elements. There are two types of rotations: *right* rotations and *left* rotations depending on whether the node to be raised is a left or right child of its parent respectively. Figure 3.1 shows the rotation operations as applicable to the TST.

The properties of rotations performed at node i are:

1. The subtrees rooted at i_L and i_R remain unchanged.
2. After a rotation is performed, i and the parent of i, $P(i)$, interchange roles i.e., i becomes the parent of $P(i)$.
3. Except for $P(i)$, nodes that were ancestors of i before a rotation, remain as ancestors of i after it.
4. Nodes that were not ancestors of i before a rotation do not become ancestors of i after it.
5. The middle nodes are not affected.

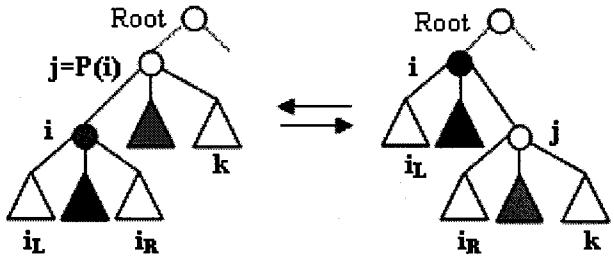


Figure 3.1: Rotation operations as applicable to the TST.

3.3 Self-Adjusting Heuristics

In the literature, a lot of attention has been given to trees with bounded height, known as balanced trees. The reason for this is obvious, since the worst-case access time is proportional to the height of the tree. However, balanced trees [8, 65] are not as efficient if the access pattern is nonuniform; furthermore, they also need extra space for storage of the "balance" information [145]. These trees are all designed to reduce the worst-case time per operation. However, in some typical applications of search trees, the system encounters a sequence of access operations, and what matters is the total time required for the sequence and not the individual times required for each access operation. In such applications, a better goal is to reduce the "amortized times" of the operations, which is the average time of an operation in a worst-case sequence of operations. In other applications, one seeks to minimize the average asymptotic cost of the data structure.

One way to obtain amortized average case efficiency is to use a "self adjusting" data structure [3]. The structure can be in any arbitrary state, but after each access operation (or a set of access operations) a restructuring rule is applied, which, in turn, is intended to improve the efficiency of future operations.

Self adjusting data structures have several possible advantages [3] over balanced, or otherwise explicitly, constrained structures. These advantages, cited from [3], are:

1. Ignoring constant factors, self adjusting structures are never much worse than constrained structures, and since they adjust according to usage, they can be much more efficient if the access distribution is skewed.

2. They often need less space, since no balance or other constraint information is stored.

3. Their access and update algorithms are conceptually simple, and very easy to implement.

Although these advantages have been cited in [3], they have to be qualified[4]. With regard to the first advantage, it is true that, in general, adaptive structures possess these advantages, but for some structures, for example, for splay trees, there is a reduction in the worst-case complexity, from $O(n^2)$ to $O(n \log n)$ total cost. But in this case the possible savings are by a constant factor, bounded from above by the alphabet size. With regard to the second advantage, adaptive structures only require less space than cost-balanced structures, but not less than other structures, in general. Indeed, because these structure require parent pointers, they generally require more space than unbalanced structures. Finally, the third advantage cited in [3], is arguable, as anyone who has attempted to teach adaptive structures to undergraduate students can attest.

Self adjusting structures have two possible disadvantages:

1. They require more local adjustments, especially during accesses, but explicitly constrained structures need adjusting only during updates, and not during accesses.

2. Individual operations within a sequence can be expensive, which may be a drawback in a real-time application.

As we shall see presently, numerous methods have been proposed for the self-adjusting of lists and binary search trees. For tries, most of the work that has been done, has attempted to change the structure of the nodes of the tries, or even change the structure of the trie itself, in order to save more storage (compression) and/or to render the access time more efficient [1, 9, 28, 29, 103]. A little work has been done in the self-adjusting of tries based on the underlying access distribution. To the best of our knowledge, the only work directly pertaining to this is the Burst trie [69, 158] mentioned earlier in Chapter 2. The Burst trie starts with a single container, implemented in [69, 158] as a BST with a move-to-front heuristic. When the number of nodes in the container starts to be large, (as per a pre-defined criterion), it "bursts" to form a node of the trie that points to smaller containers, and so on. Although it is an elegant

[4]We are grateful to anonymous Referee of [22] for pointing out these qualifications.

data structure which is very efficient for strings with skewed distribution, the disadvantage of the Burst trie, as we see it, is that it needs a number of parameters which are to be tuned or adjusted. These may have to be varied according to the data for which it is used. The worst case for the Burst trie is where the strings are of equal lengths and have a relatively flat probability distribution. For text data, its performance depends on the fact that words most commonly used are short, and are stored wholly within the access trie. Rare strings are held in containers, but because they are typically long, there is no loss of efficiency [69]. The cost of searching the container is offset by the time saving of not having to traverse a large number of trie nodes.

Indeed, if we work with the assumption that the structure of the node is not permitted to be changed in any restructuring operation, then, to the best of our knowledge, there is no work reported for creating and manipulating self-adjusting TSTs. The only work reported involves the extension of the splaying operation introduced by Sleator and Tarjan in [145] for BSTs, which has since been applied for muti-way trees [138, 159].

In [69], Heinz *et al.* have shown through experimentation, that TSTs are slower and less compact than *hash* tables and *Burst* tries. We attempt to examine different balancing and self-adjusting heuristics that were previously used for BSTs, and to apply them to TSTs. The advantage of this is to allow the TST to adapt itself according to the data without any parameter adjustments.

The problem of constructing efficient BSTs has been extensively studied. The optimal algorithm due to Knuth [82], uses dynamic programming and produces the optimal BST using $O(n^2)$ time and space. Alternatively, Walker and Gotlieb [157] have used dynamic programming and divide-and-conquer techniques to yield a nearly-optimal BST using $O(n)$ space and $O(n \log n)$ time.

Self-adjusting BSTs are closely related to the subject of self-organizing lists. A self-organizing list is a linear list that rearranges itself such that after a long enough period of time, it tends towards the optimal arrangement with the most probable element at the head of the list, while the rest of the list is recursively ordered in the same manner. Many memory-less schemes have been developed to reorganize a linear list dynamically [16, 33, 67, 70, 82, 96]. Among these are the move-to-front rule [82, 96] and the transposition rule [132]. These rules, their extensions, and their analytic properties are discussed extensively in the literature [16, 33, 67, 70, 82, 96].

Schemes involving the use of extra memory have also been developed; a review of these is found in [70]. The first of these uses counters to achieve estimation. Another is a stochastic move-to-rear rule [120] due to Oommen and Hansen, which moves the accessed element to the rear with a probability which decreases each time the element is accessed. A stochastic move-to-front [120] and a deterministic move-to-rear scheme [119] due to Oommen *et. al.* have also been reported.

The primitive tree restructuring operation used in most BST schemes is the well known **Rotation** [2]. A few memory-less tree reorganizing schemes which use this operation have been presented in the literature among which are the **Move-to-Root** and **Simple Exchange** rules. These rules are analogous in spirit to the move-to-front and transposition rules respectively for linear lists. The Move-to-Root Heuristic was historically the first self-organizing BST scheme in the literature [4] and is due to Allen and Munro. It is both conceptually simple and elegant. Each time a record is accessed, rotations are performed on it in an upwards direction until it becomes the root of the tree. The idea is that a frequently accessed record will be close to the root of the tree as a result of it being frequently moved to the root, and this will minimize the cost of the search and the retrieval operations. Allen and Munro [4] also developed the Simple Exchange rule, which rotates the accessed element one level towards the root, similar to the transposition rule for lists. Contrary to the case of lists, where the transposition rule is better than the move-to-front rule [132], they show that whereas the Move-to-Root scheme has an expected cost that is within a constant factor of the cost of a static optimum BST, the simple exchange heuristic does not have this property. Indeed, it is provably bad.

Sleator and Tarjan [145] introduced a technique, which also moves the accessed record up to the root of the tree using a restructuring operation called **splaying** which is a multi-level generalization of rotation. Their structure, called the splay tree, was shown to have an amortized time complexity of $O(\log n)$ for a complete set of tree operations which included insertion, deletion, access, split, and join. This heuristic is rather ingenious. It is a restructuring move that brings the accessed node up to the root of the tree, and also keeps the tree in a symmetric order, and thus an in-order traversal would access each item in order from the smallest to the largest. Additionally it has the interesting side effect of tending to keep the tree in a form that is nearly height-balanced apart from also capturing the desired effect of keeping the most frequently accessed elements near the root of the tree. The heuristic is somewhat similar to the Move-to-Root scheme, but whereas the latter has an asymptotic average access time within a

constant factor of the optimum when the access probabilities are independent and time invariant, the splaying operation yields identical results even when these assumptions are relaxed. More explicit details and the analytic properties of the splay tree with its unique "two-level rotations" can be found in [44, 145]. Splaying at a node x of depth d takes $\Theta(d)$ time. That is, the time required is proportional to the time needed to access the item in x. Splaying not only moves x to the root, but roughly halves the depth of every node along the access path. On an n-node splay tree all the standard search tree operations have an amortized time bound of $O(\log n)$ per operation. Thus splay trees are as efficient as balanced trees when the total running time is the measure of interest. In a splay tree, whenever a node is accessed via standard operations, it is splayed, thus making it the root. More details about splaying will be shown when it is applied to the TST in Section 3.4.1.

The disadvantage of the splay tree is that the cost of the access operation is high due to the large amount of restructuring operations done. Also, the splaying rule always moves the record accessed up to the root of the tree. This means that if a nearly optimal arrangement is reached, a single access of a seldomly-used record will disarrange the tree along the entire access path [13, 44] as the element is moved upwards to the root. Thus the number of operations done on every access is exactly equal to the depth of the node in the tree, and these operations are not merely numeric computations (such as those involved in maintaining counters and "balancing functions") but rotations. Thus the splaying rule can be very expensive. Moreover [13], this is undesirable in a caching or paging environment where the write-operations involved in the restructuring will "dirty" memory locations or pages that might be otherwise stay "clean".

To overcome the disadvantages of splaying, Cheetham *et. al.* [44] proposed a heuristic that concentrates on these problems. They introduced a distinct heuristic to reorganize a BST so as to asymptotically arrive at an optimal form. It requires three extra memory locations per record. Whereas the first counts the number of accesses to that record, the second counts the number of accesses to the subtree rooted at that record, and the third evaluates the **Weighted Path Length** (WPL) of the subtree rooted at that record. The paper [44] specifies an optimal updating strategy for these memory locations. More importantly, however, it also specifies how an accessed element can be rotated towards the root of the tree so as to minimize the overall cost of the tree. Finally, unlike most of the algorithms that are currently in the literature, this move is not done on every data access operation. It is performed if and only if the overall WPL

of the resulting BST decreases. Cheetham *et. al.* [44] also presented a space-optimized version requiring only a **single** additional counter per node. Using this memory location they designed a scheme identical to the one described above. The details of applying this method is shown in [44]. We will discuss it in more detail when we apply it to the TST.

One Balancing strategy for BSTs, that we will apply to the TST, is Randomized Search trees, or Treaps [13, 53, 154]. Let X be a set of n items, each of which has associated with it a key and a priority. The keys are drawn from some totally ordered universe, and so are the priorities [13]. The two ordered universes need not be the same. A treap for X is a rooted binary tree with node set X that is arranged in *inorder* with respect to the keys and in *heaporder* with respect to the priorities, where the latter items are to be understood as follows:

- ***Inorder*** means that for any node x in the tree $y.key \leq x.key$ for all y in the left subtree of x and $x.key \leq y.key$ for y in the right subtree of x.
- ***Heaporder*** means that for any node x with parent z the relation $x.priority \leq z.priority$ holds.

It is easy to see that for any set X such a treap exists. The item with largest priority becomes the root, and the allotment of the remaining items to the left and right subtree is then determined by their keys. Put differently, the treap for an item set X is exactly the binary search tree that results from successively inserting the items of X in the order of their decreasing priorities into an initially empty tree using the usual leaf insertion algorithm for the BSTs. Given the key of some item $x \in X$, the item can easily be located in the treap using the usual search tree algorithm.

We now address the question of updates: The *insertion* of a new item z into T can be achieved as follows: At first, using the key of z, attach z to T in the appropriate leaf position. At this point the keys of all the nodes in the modified tree are in inorder. However, the heaporder condition might not be satisfied, i.e. z's parent might have a smaller priority than z. To reestablish heaporder, we simply rotate z up as long as it has a parent with smaller priority (or until it becomes the root). *Deletion* of an item x from T can be achieved by "inverting" the insertion operation: First locate x, then rotate it down until it becomes a leaf, where the decision to rotate left or right is dictated by the relative order of the priorities of the children

of x. Finally, the leaf is "clipped away" or deleted.

The random search tree for X is defined [13] to be a treap for X where the priorities of the items are independent, identically distributed continuous random variables.

3.4 Proposed Restructuring Methods for Ternary Search Tries (TST)

In this section we demonstrate how we can apply two self adjusting heuristics that have been previously used for BSTs, to TSTs. These heuristics are the **splaying** [3, 145] and the **Conditional Rotation** [3, 44] heuristics. We have also applied another balancing strategy used for BSTs, to TSTs, which is the basis for the **Randomized** search trees, or Treaps [13, 53, 154]. The splaying method has already been proposed and has been (albeit, less formally) applied to the TST in [145], but the other two schemes which we suggest are newly proposed for the TST. All the pieces of pseudo-code for applying the suggested techniques to TSTs are given in Appendix A.

3.4.1 Splaying for TSTs

Sleator and Tarjan [145] introduced the idea of extending the splaying as a restructuring heuristic after each access for the TST. We will call the resulting data structure the *Splay-TST*.

To splay [145] at a node x, we proceed up the access path from x, one or two nodes at a time, carrying out the same splaying steps as in the BST, with the additional condition that whenever x becomes a middle child, we continue from $p(x)$ instead of from x. Formally, the following splaying step is repeated until x is the root of the current BST (Figure 3.2):

- ***Case 1 (zig)***: If $p(x)$, the parent of x, is the root of the tree or the middle node of its parent, and x is not a middle child of $p(x)$, rotate the edge joining x with $p(x)$. If x is a middle child of $p(x)$, then splay $p(x)$.

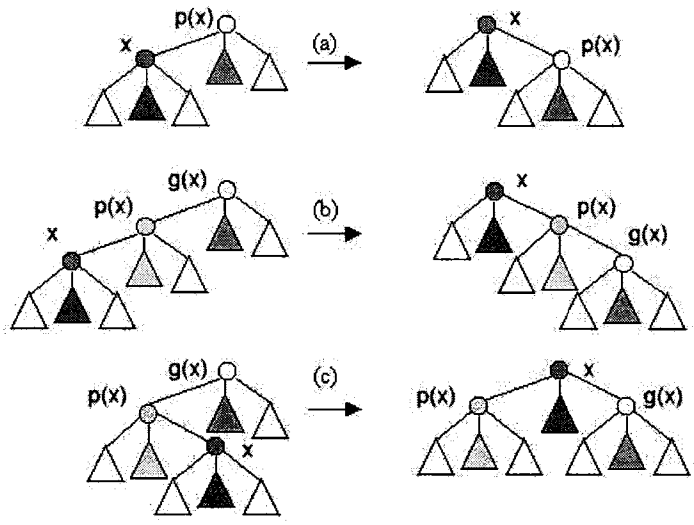


Figure 3.2: Different cases for the splaying operation for the TST. Node x is the current node involved in the splaying. The cases are: (a) Zig. (b) Zig-Zig. (c) Zig-Zag.

- ***Case 2 (zig-zig)***: If $p(x)$ is not the root, and x and $p(x)$ are both left or right children, rotate the edge joining $p(x)$ with its grandparent, $g(x)$, and then rotate the edge joining x with $p(x)$. If x is a middle child of $p(x)$, then splay $p(x)$.
- ***Case 3 (zig-zag)***: If $p(x)$ is not the root, and x is a left child and $p(x)$ a right child, or vice-versa, rotate the edge joining x with $p(x)$, and then the edge joining x with the new $p(x)$. If x is a middle child of $p(x)$, then splay $p(x)$.

These steps are performed from the bottom of the TST towards the root until the recursion terminates with Case (1). Whenever we reach the root of a current BST during the splay process, we continue splaying for the parent node unless the root of the current BST is the root of the whole TST.

To state (without proof) Sleator and Tarjan's results for the BST, we introduce the following notation. Let $w(i)$ be the weight of a node x_i, which is an arbitrary positive value, subjectively assigned to the nodes. Also let $s(i)$ be the size of a node x_i, which is the sum of the individual weights of all the records in the subtree rooted at x_i. If p_i is the access probability of x_i, the following are true for a sequence of m accesses on an n-node splay tree:

Theorem 3.1. *(Balance theorem) The total access time is* $O((m+n)\cdot\log n + m)$. □

Theorem 3.2. *(Static optimality theorem) If every record is accessed at least once, the total access time is*

$$O(m + \sum_{i=1}^{n} p_i \cdot \log(\tfrac{m}{p_i})).$$

□

Theorem 3.3. *(Static finger theorem) If* f *is any fixed item, the total access time is*

$$O(n\cdot\log n + m + \sum_{j=1}^{m}\log(|i_j - f| + 1)).$$

where the items are assumed to be numbered 1 through n *in a symmetric order, and the sequence of accessed items is* $i_1, i_2, \ldots, i_m$. □

Theorem 3.4. *(Working set theorem) Suppose the accesses are numbered from 1 to* m *in the order in which they occur. For any time instant* j *at which the record accessed is* x_j*, let* $t(j)$ *be the number of distinct items accessed before* j*, subsequent to the last access of item* x_j*, or subsequent to the beginning of the sequence if* j *is the first access to* x_j*. Then the total access time is:*

$$O(n\cdot\log n + m + \sum_{j=1}^{m}\log(t(j)+1)).$$

□

Theorem 3.5. *(Unified theorem) The total time of a sequence of* m *accesses on an* n*-node splay tree is*

$$O(n\cdot\log n + m + \sum_{j=1}^{m} \log\ min\ \{\tfrac{m}{s_{i_j}}, |i_j - f| + 1, t(j)+1\}),$$

where f *is any fixed item, and* $t(j)$ *is defined as in Theorem 3.4 above.* □

Sleator and Tarjan also derived upper bounds for each of the operations supported by the splay tree. In the interest of brevity, we shall only present the upper bounds for the access operation. The details of the other operations can be found in [145].

Let W be the total weight of the tree included in the access operation. For any item i in a tree T let $i-$ and $i+$ denote the item preceding and the item following i, respectively, in the symmetric order, where $w(i-) = \infty$, if $i-$ is undefined, and $w(i+) = \infty$ if $i+$ is undefined. Then, the upper bound on the amortized time complexities of the access operation is:

$$3\log(\frac{W}{w(i)}) + 1 \text{ if } i \in t;$$

$$3\log(\frac{W}{min\{w(i-),w(i+)\}}) + 1 \text{ if } i \notin t;$$

Theorems 3.1 through 3.4 are particularly important, because they imply that:

1. Over a long enough sequence of accesses, a splay tree is as efficient as any type of uniformly balanced tree.
2. A splay tree is as efficient as any fixed search tree, including the optimum tree for the given access sequence.
3. Splay trees support accesses in the vicinity of a fixed finger with the same efficiency as finger search trees.
4. The most recently accessed items, which can be conceptually imagined to form a "working set", are the easiest to access.

Theorem 3.5 unifies all of the above into a single result.

The amortized analysis of the Splay-TST is given in [145], and it shows that the amortized time to access a string S in a Splay-TST is $O(|S| + \log M)$, where M is the number of strings stored in the tree.

3.4.2 Randomized TSTs

As we saw in Section 3.3, random BST adjustments are done during insertions, and no adjustments are done during searching. The price that we pay, of course, is the extra memory requirement for the priority of each node. In [13], the authors showed that it is possible to implement randomized search trees in such a manner that no priorities are stored explicitly. They offered three different methods. The first uses hash functions to generate priorities on demand. The second method stores the nodes of the tree in a random permutation and uses the node addresses themselves as priorities. The last method recomputes priorities from subtree sizes. In our implementation we explicitly utilized stored priorities, because the access time was our primary consideration.

The randomization is usually used when the distribution of the data accesses is uniform, and the items are all equally likely to be accessed. This heuristic is included in order to study the effect of randomization on TSTs, and to see how its performance is compared with the performance of other heuristics for uniformly distributed data sets. The effect of randomization for non-uniform distributions is typically studied using a few traditional distributions such as the Zipf's, exponential etc. We shall discuss these aspects in a subsequent section when the properties of TSTs are experimentally verified.

Aragon and Seidel [13] analyzed a number of interesting quantities related to Randomized Binary Search Trees (RBSTs) and derive their expected values. Some of the quantities are:

- $D(x)$, the depth of node x in a RBST, which is the number of nodes in the path from x to the root.
- $S(x)$, the size of the subtree rooted at node x, which is the number of nodes contained in that subtree.

Let n be the number of nodes in an RBST. Then the following lemmas and theorem (whose proofs are omitted) are stated in [13].

Lemma 3.1. *Let T be the treap of n items, and let $1 \leq i, j \leq n$. Then, assuming that all the priorities are distinct, x_i is an ancestor of x_j in T iff among all x_h, with h between i and j, the item x_i has the largest priority.* □

Theorem 3.6. *Let $1 \leq l < n$. In a randomized search tree of n nodes the following expectations hold:*

- $E_x[D(x_l)] < 1 + 2 \cdot \ln n$.
- $E_x[S(x_l)] < 1 + 2 \cdot \ln n$. □

Lemma 3.2. *In a randomized search tree with $n > 1$ nodes, we have for index $1 \leq l \leq n$ and any $c > 1$,*

$P_r[D(x_l) \geq 1 + 2c \ln n] < 2(n/e)^{-c \ln(c/e)}$. □

This balancing method can also be applied to the TST, if we take into consideration the fact that searching for a string S in a TST is as if we search for each character of the string in a *separate* BST. So, each of these BSTs could be balanced using the "Random Tree" property.

In other words, during insertion of S in a TST, each character c in S is assigned a random priority, and then inserted in its corresponding BST, T. At this time, if the character is found in T, we skip to the next character, and no adjustments are done in T. If the character is not found, an insertion is done at the proper leaf position, so as to maintain the inorder property. In order to keep the heaporder property, we simply rotate c up as long as it has a parent with smaller priority, or until it becomes the root of *its* BST, T. This is done for every character of the string during the insertion process.

The search is done using the usual search algorithm of the TST. We will call the resulting data structure the *Rand-TST*.

As shown in [13] and the theorems presented above, the expected cost of searching a RBST is $O(\log(n))$. As a simple extension, the expected cost of searching the Rand-TST is $O(|S| + \log(M))$, where $|S|$ is the size of the string searched for, and M is the total number of strings stored in the tree.

3.4.3 Conditional Rotations for TSTs

As mentioned, the basic philosophy we have adopted to achieve adaptive TST restructuring is to adopt the same adaptive restructuring scheme for the tree associated with *every* BST along the search path of the string. We shall now see how we can adapt the conditional rotation heuristic for this purpose. In this heuristic, the rotation is not done on every data access operation. It is performed if and only if the overall WPL of the entire BST rooted at the *current middle* node decreases. So the net effect is as if we are invoking the same scheme for every BST that represents a character in the string. Clearly, the maximum number of rotations performed is $\mid S \mid$, where S is the string to be inserted in the TST. We will call the resulting data structure the *Cond-TST.*

The reader must observe the difference between the current scheme and the original conditional rotation scheme introduced in [44]. In the latter there was only one BST which was adaptively restructured. In this case, there is a BST at *every* node, and this restructuring is done only if the "estimated" cost of the *middle* node for *that* BST decreases. Thus the convergence results are true for the TST as a consequence of the result being true for every single BST of every node of the TST. Observe too that in this way, we are trying to diminish the problem associated with splaying, which does too many rotations during the access operation. We also try to gain the advantages of self adjusting strategies and the advantages of balancing strategies with the additional expense of only a single extra memory location per node.

In [44] Cheetham *et. al.* developed two algorithms for this heuristic when applied to the BST. The first requires three extra integer memory locations for each record in the BST, and the second, the space-optimized version, requires only a single extra memory location. We now discuss the properties of the first one, and later discuss the optimized version, but with respect to the TST. We will use the optimized algorithm to test the heuristic for the TST.

For each node i of every BST t_c in the TST, T, we include three extra memory locations. The current BST t_c is the BST rooted at the c^{th} middle node traversed during the access. The three memory locations are:

- ***First counter*** $\alpha_i(n)$: total number of accesses of a node i in t_c up to and including time n.

- ***Second counter*** $\tau_i(n)$: the total number of accesses to the subtree, of t_c, rooted at node i. Clearly $\tau_i(n)$ satisfies:

$$\tau_i = \sum_{j \in T_i} \alpha_j \quad \forall j \in t_c \; of \; T. \tag{3.1}$$

- ***Third counter*** $\kappa_i(n)$: the WPL of the subtree, of t_c, rooted at node i at time instant n. Let $\lambda_i(n)$ be the path length of i from the root of t_c. Then:

$$\kappa_i = \sum_{j \in T_i} \alpha_j \cdot \lambda_j \quad \forall j \in t_c \; of \; T. \tag{3.2}$$

Equivalently,

$$\kappa_i = \sum_{j \in T_i} \tau_j \quad \forall j \in the \; BST \; t_c \; of \; T. \tag{3.3}$$

These quantities (α, τ and κ) are maintained for every node in t_c, and are updated after every access operation after we find the character sought for in t_c. In order to avoid a traversal of the entire BST, t_c, for updating τ and κ, Cheetham *et. al.* noted the following recursively computable properties for the quantities τ and κ applied to t_c:

$$\tau_i = \alpha_i + \tau_{iL} + \tau_{iR} \tag{3.4}$$

$$\kappa_i = \alpha_i + \tau_{iL} + \tau_{iR} + \kappa_{iL} + \kappa_{iR} \tag{3.5}$$

where iR and iL denote the right and left children of the node i.

These properties imply that to calculate τ and κ for any node, it is necessary to look only at the values stored at the node and at the left and right children of the node.

Observe that in order to update τ, only a single top-down pass on the TST is required. This can be done by incrementing the τ values as we traverse down the access path. Using Equations (3.4) and (3.5), we present a simple scheme to calculate the α, τ, and κ fields of the nodes of t_c after each access. The recursions motivating the scheme are highlighted in the following theorem.

Theorem 3.7. *Let i be any arbitrary node in the TST, T. Node i stores the current character*

of the string we search for. At this time, we have finished searching the current BST t_c, and then we move to the next BST rooted at the middle child of i. On accessing $i \in T$, the following updating schemes for α, τ, and κ are valid whether or not a rotation is performed at i.

- ***Updating*** α
  We need to update α only at node i as per:

$$\alpha_i = \alpha_i + 1$$

- ***Updating*** τ
  We need to update the nodes in the access path from the root of the current BST to the node before the parent $P(i)$ according to the rule:

$$\tau_j = \tau_j + 1$$

  If a rotation is performed, i and $P(i)$ are updated by applying:

$$\tau_{P(i)} = \alpha_{P(i)} + \tau_{P(i)L} + \tau_{P(i)R}$$

  followed by

$$\tau_i = \alpha_i + \tau_{iL} + \tau_{iR}$$

  If no rotation is performed, they are updated as other nodes in the path.

- ***Updating*** κ
  We need to update the nodes in the access path from the root of the current BST to node i by applying:

$$\kappa_j = \alpha_j + \tau_{jL} + \tau_{jR} + \kappa_{jL} + \kappa_{jR}$$

  from node i upwards to the root of the current BST.

Proof: The proof of these updating rules follows from the corresponding proofs of the respective updating rules for the case of the BST in [44]. Since the updating is done for independent BSTs in the TST, T, they are true for *every* BST, and so they are true for the entire TST. The arguments are not included here to avoid repetition. □

The main contribution of Cheetham *et. al.* [44] is their proposal of a criterion function θ_i which can be used to determine whether a rotation should be applied at node i or not. Let θ_i be $\kappa_{P(i)} - \kappa'_i$, where the primed quantity is a post-rotational quantity (i.e., it is the value of the specified quantity after the rotation has been performed). The criterion function, θ_i reports whether performing a rotation at node i will reduce the κ-value at $P(i)$ or not. This is of interest because it can be shown that the κ-value of the *entire* current BST is reduced by a rotation if and only if θ_i is reduced by the *local* rotation at i (referred to as a κ-lowering rotation). The result is also true for the TST and is formally given below.

Theorem 3.8. *Any κ-lowering rotation performed on any node in the current BST, t_c, will cause the weighted path length of t_c to decrease, and so this will decrease the accumulated WPL of the middle nodes along the search path of the entire TST, T.*

Proof: This theorem is true for every BST of T as shown in [44], and so it is true, in particular, for t_c. Since we decrease the WPL of every BST whose root is a middle node in the search path, the accumulated WPL of the middle nodes will decrease, and so the result will also be true for the entire TST, T. □

Assuming that the average path length of the current BST with n nodes is $k_1 \log n$, the original algorithm requires $2k_1 \log n$ time, and $3k_1 n$ space over and beyond what is required for the tree itself.

Now that we are familiar with the original algorithm, we shall examine a space-optimized version that requires only n extra memory locations and $\log n$ time for every BST. This version is based on the observation that the values of α_i and κ_i are superfluous because the information stored in the α_i values is also stored in the τ_i values, and the values of κ_i can be expressed in terms of τ_i and α_i, as in Equation (3.5).

To see the effect of any tree restructuring operation, the correct method would be to perform the operation and to compare the cost before and after it is done. The reason for this is that the

operation can be done anywhere in the tree, but the effect of the operation must be observed at the root of the tree. Cheethman *et. al.* [44] were able to show the existence of a function ψ_i, which can be locally computed and simultaneously "forecast" the effect at the global level. As in [44], we state a new criterion function, ψ_i, defined only in terms of τ_i and α_i and show that a local evaluation of ψ_i is sufficient to anticipate whether a rotation will *globally* minimize the cost of the overall TST or not. If $\psi_i > 0$, then the rotation should be done as it will globally minimize the cost of the whole TST. This result is formally stated as follows.

Theorem 3.9. *Let i be the accessed node of the current BST t_c, in the TST, T, and let $\kappa_{P(i)}$ be the weighted path length of the BST rooted at the parent $P(i)$ if no rotation is performed on node i, and i is not the middle node of $P(i)$. Let i_L and i_R be the left and right children of i respectively and $B(i)$ be the brother node of i. $P(i)$ will be assumed null if i is the middle child or the root of T. Let κ'_i be the weighted path length of the BST rooted at node i if the rotation is performed. Furthermore, let ψ_i be defined as follows:*

$$\psi_i = \alpha_i + \tau_{iL} - \alpha_{P(i)} - \tau_{B(i)} \text{ if } i \text{ is a left child;}$$

$$\psi_i = \alpha_i + \tau_{iR} - \alpha_{P(i)} - \tau_{B(i)} \text{ if } i \text{ is a right child.}$$

Then, if $\theta_i = \kappa_{P(i)} - \kappa'_i$,

$$\psi_i \geq 0 \text{ if and only if } \theta_i \geq 0.$$

Proof: This is one of the fundamental results in [44]. Typically, when a restructuring is to be done, the operation will cause the cost of the entire data structure to change. However, Cheetham *et al* showed that in the case of the conditional rotation, it suffices to do a local computation so as to determine whether the global, overall cost of the tree will decrease. This computation involves the quantity ψ_i for the BST. If ψ_i is positive, it can be shown that the cost of the overall BST decreases.

The question of extending this concept to the TST is actually quite straightforward. Since the actual contents of the nodes are unchanged, and since the TST is comprised of individual

BSTs which are "disjoint", we can determine whether we will achieve a global optimization by merely computing the value of ψ_i for each separate BST. Since the cost of every BST can be guaranteed to be decremented whenever the corresponding ψ_i is positive, and since the decrease of the cost of any one BST does not effect the increase/decrease of the cost *of any other* BST, it is straightforward to see that the cost of the overall TST is also decreased whenever all the ψ_is are positive. But since the other ψ_js, (other than the one that is local to the current BSTs) are unchanged, the fact that the current ψ_i is positive suffices to guarantee that the global cost of the overall TST also decreases. Hence the theorem! □

Observe that we need not maintain the κ fields because ψ_i requires only the information stored in the α and τ fields. This implies that after the search for the desired record (requiring an average of $O(\log n)$ time) and after performing any reorganization (which takes constant time), the algorithm does not need to update the κ values of the ancestors of i. Also note that at node i, the α values may be expressed in terms of the τ values, i.e. $\alpha_i = \tau_i - \tau_{iL} - \tau_{iR}$, and need not be explicitly stored. Therefore, for every node of every BST in the TST, we need only one extra memory location, and a second pass of the current BST is not required. Since $\alpha_{P(i)} = \tau_{P(i)} - \tau_i - \tau_{B(i)}$, Then ψ_i can now be evaluated as follows:

$$\psi_i = 2\tau_i - \tau_{iR} - \tau_{P(i)} \text{ if i is a left child;}$$

$$\psi_i = 2\tau_i - \tau_{iL} - \tau_{P(i)} \text{ if i is a right child.}$$

Consequently, the modified algorithm, requires only n extra memory locations and $O(|S| + \log(M))$ time, where $|S|$ is the size of the string searched for, and M is the total number of strings stored in the TST.

3.5 Experimental Results

The adaptive restructuring mechanisms introduced in this Chapter, which are applicable to the TST, have been rigorously tested so as to evaluate their relative performance. In all brevity, we

mention that the performance of the new strategy for the cond-TST introduced here, is, in our opinion, quite remarkable.

The principal objective of our study was to demonstrate the improvements gained by different heuristics when they were applied to TSTs, and to compare them with the original TST. Additionally, we intended to investigate the performance of the different adjusting heuristics when the access distributions varied. The experiments compare the performance of the four data structures presented in the previous sections, namely the TST, the Splay-TST, the Rand-TST, and the Cond-TST. The comparison is made in terms of the running time measured in seconds, and the space requirements in MBs. For our comparisons we conducted four sets of experiments on four benchmark data sets as explained below.

3.5.1 Data Sets

Four benchmark data sets were used in our experiments. Each data set was divided into two parts: a *dictionary* and *test documents*. The dictionary was made up of the words or sequences that had to be stored in the TST. The test documents were the "files", for which the words were searched for in the corresponding dictionary.

The four dictionaries we used[5] were as follows:

- Dict[6]: This is a dictionary file used in the experiments done by Bentley and Sedgewick in [30].
- The *Webster's Unabridged* Dictionary: This dictionary was used by Clement *et. al.* [1, 45] to study the performance of different trie implementations including the TST.
- 9-grams dictionary: This dictionary consisted of the unique words found in the Genomic data used by Heinz *et. al.* [69] for testing the Burst trie. The genomic data is a collection of nucleotide strings, each being typically, thousands of nucleotides in length. It is parsed into shorter strings by extracting n-grams of length 9, which are utilized to locate regions where a longer inexact matched may be found. The alphabet size is four characters.

[5]We requested from the authors of the Burst trie paper [69] the data that they used. Unfortunately, the data was copyright protected and we did not have the financial resources to purchase its license.

[6]The actual dictionary can be downloaded from http://www.cs.princeton.edu/ rs/strings/dictwords.

- NASA dictionary: The dictionary consisted of unique words extracted from the *NASA* file. This is a large data set that is freely available, and consists of a collection of file names accessed on some web servers[7].

The corresponding four test documents were:

- Dict documents: These were five artificially created documents from the dictionary described above. Each document was created with a large number of strings that followed a certain distribution. These documents were used to simulate five types of access distributions, namely the Zipf's, the exponential, the two families of the wedge distribution, and the uniform distribution.
- Herman Melville's novel *Moby Dick*: This complete text [1, 45] is available on the WWW. The average length of the words in *Moby Dick* was 5.4 characters.
- Genome[8]: The Genome document did not have the same skewed distribution that is typical of text, as in the case of the first two sets. It was fairly uniform, with even the rarest n-grams occurring hundreds of times, and did not even show much locality. This data earlier demonstrated a poor performance in the Burst trie structure [69]. The document consisted of the parsed 9-grams obtained from the genome data, and was used to test the case where the strings searched for were of equal lengths.
- NASA document: This document showed characteristics that were different from the English text and had strings of relatively larger lengths than the other data sets. As we shall see, the strings had the property that they shared common prefixes, which, in turn, gave a perfect TST when it was used to store strings of this type.

The statistics of these data sets are shown in Table 3.5.1, and snapshots of the different dictionaries used are given in Table 3.5.1. The first two data sets have very similar dictionaries, which are actually English words, and so we show only a snapshot for the dictionary of Dict data set. As mentioned earlier, since the data used in our experiments is entirely cache resident, all these sets, really, should be considered to be of "moderate" size.

[7]This dictionary can be obtained from: http://cg.scs.carleton.ca/ morin/teaching/tds/src/nasa.txt.gz. We are grateful to Prof. Pat Morin, from Carleton University, for providing us with this source.

[8]This file could be obtained from: http://goanna.cs.rmit.edu.au/~rsinha/resources/data/set6_genome.zip

	Dict	Webster	Genome	NASA
TST & Size of dictionary	232KB	944KB	2.75MB	548KB
Size of document	1.3GB	1.02GB	301MB	2.34GB
number of words in dictionary	25,481	91,479	262,024	15,700
number of words in document	150000000	185408000	31623000	78494851
min word length	1	1	9	1
max word length	22	21	9	99

Table 3.1: Statistics of the data set used in the experiments.

Dict	Genome	NASA
spite	ccacggacc	$/cgi-bin/imagemap/countdown70?181,275$
et	gactggcca	$/shuttle/missions/sts-70/sts-70-patch-small.gif$
I'll	ggattgaaa	$/shuttle/missions/sts-68/news/sts-68-mcc-11.txt$
disulfide	tgtttattg	$/shuttle/missions/sts-70/o-ring-problem.gif$
windbag	ttggtgaag	$/shuttle/missions/sts-70/movies/woodpecker.mpg$
Eleazar	cagcatgct	$/shuttle/resources/orbiters/endeavour.gif$
radiography	ttcaatcat	$/shuttle/missions/sts-68/news/sts-68-mcc-12.txt$
marmalade	acaacgaaa	$/shuttle/missions/sts-68/sts-68-patch-small.gif$
concatenate	acggtgcca	$/images/op-logo-small.gif$
rib	gtctccgtc	$/shuttle/missions/sts-73/mission-sts-73.html$
sore	taataagtc	$/shuttle/missions/sts-73/sts-73-patch-small.gif$
grantee	actgcgaaa	$/shuttle/missions/sts-71/images/KSC-95EC-0911.gif$
Jolla	atcctggtt	$/shuttle/technology/sts-newsref/sts_asm.html$
taper	atcgcctct	$/shuttle/technology/images/srb_mod_compare_6-small.gif$
Almaden	gtgaagtcg	$/shuttle/technology/images/srb_mod_compare_1-small.gif$
yearn	gtaatggtt	$/images/shuttle-patch-logo.gif$

Table 3.2: Snapshots of the strings in Dict, Genome and NASA data sets.

3.5.2 Methodology

We considered four sets of experiments. Each set corresponded to each of the data sets. In each experiment, we stored the dictionary in the TST, and the strings in the corresponding document were used in the search process. The words of the dictionary were stored in the order that they appeared in the dictionary. For each data set, 100 parallel experiments were done for a large number of accesses so that a statistically dependable ensemble average could be obtained. In this Chapter, we have tested only the schemes for cases of successful access operations; the primary aim being that of demonstrating the improvements gained by different heuristics when applying them to TSTs themselves.

To obtain the simulated five documents for the first data set, the words of the dictionary were first randomly arranged in a second file, and then the resultant random file was used to generate a large search file for each distribution. Finally, the resultant search files were utilized to search each data structure in such a way that the same identical search was made for each of them. Subsequently, 150,000,000 accesses were performed on each tree, each accessed item being randomly chosen. The "randomness" of the accesses was specified by the following four types of probability distributions:

1. Zipf's distribution. In this case, the individual probabilities obey the following:

$$p_i = \frac{k_1}{i}, \text{ where } k_1 = 1/\ (\sum_{i=1}^{n} \frac{1}{i}).$$

2. Exponential distribution. Here the individual probabilities obey:

$$p_i = \frac{k_2}{2^i}, \text{ where } k_2 = 1/\ (\sum_{i=1}^{n} \frac{1}{2^i}).$$

3. Two types of wedge distributions. In both these cases $\{p_i\}$ satisfied:

$$p_i = k_3 \cdot (n - i + 1) \text{ where } k_3 = 1/\ (\sum_{i=1}^{n} i).$$

4. Uniform distribution. Here the individual probabilities obey:

$$p_i = \frac{1}{n}.$$

The difference between the two wedge distributions is that the first had the probabilities chosen in such a way that the items, when listed in order from the highest access probability to the lowest, were also in a lexicographic order from the smallest item to the largest. The second distribution was chosen such that the items had an inverse relationship between their probability mass orderings and their lexicographic orderings.

In all these cases, to make the documents realistic, the actual records are queried by the *exact* Zipf's, Exponential and Wedge distributions respectively, as defined by the above equations, whenever the precision of the machine architecture permitted it. However, as the index of the records increases (decreases in the case of the second type of wedge distribution), it is clear that the probability mass will keep decreasing. Clearly, beyond a certain limit, the probability of accessing a record will fall below the underflow limitation of the machine. Subsequent to this cut-off point, the distribution was approximated by setting all the remaining probabilities to have the smallest non-underflow value permitted in the exact distribution. We have thereby effectively rendered the distribution exact whenever possible, and forced it to be uniform thereafter as the index of the record increases (decreases in the case of the second type of wedge distribution).

The other three sets of experiments compared the performance of the data structures on the other three real documents. First, the words in each dictionary were inserted (as above) as they appeared in the dictionary on each data structure. Then, against this dictionary, the words in the corresponding document were searched for in their order of occurrence in the document. Again, 100 parallel experiments were done for a large number of accesses, so that a statistically dependable ensemble average could be obtained.

3.5.3 Analysis of Results

We now present the results obtained for each data set. The results can be summarized according to each data set as follows:

- *Dict data set*: Figures 3.3 through 3.6 show the results for the different simulated documents. Each figure shows the accumulated time needed for successful search operations for different number of words, and for different probability distributions. From the results we see that the Cond-TST is the best. For example, for 140 million search operations with

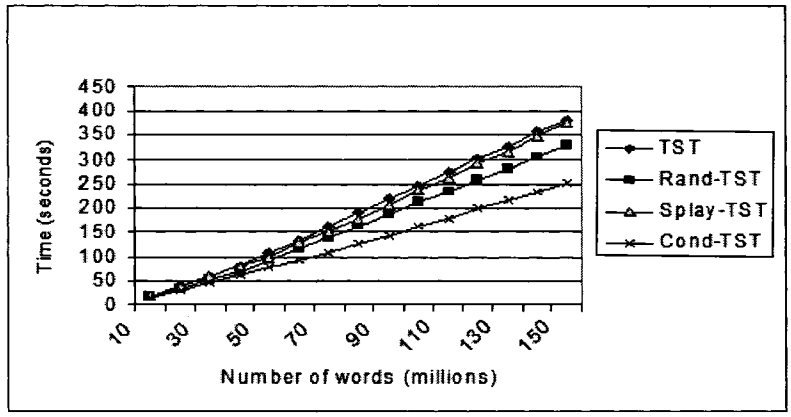


Figure 3.3: The accumulated time in seconds needed for all the data structures studied for various numbers of words, when the *Dict* dictionary was used, and the document was skewed by the Zipf's distribution.

Zipf's distribution, the Cond-TST took 235 seconds, the Splay-TST took 347 seconds, and the Rand-TST required 306 seconds. The original TST (without any adjusting) took 356 seconds[9]. Observe Figure 3.7 which shows the time needed for the last 50 million successful searches.

- *Webster data set*: Figure 3.8 shows the results for the second set of experiments which was done on real-life data set. The results confirm that the Cond-TST is the best with respect to the data structures used in the comparison. For example, for 140 million search operations, the Cond-TST took 159 seconds, the Splay-TST took 261 seconds, and the Rand-TST required 225 seconds. The original TST (without any adjusting) took 173 seconds.

- *Genome data set*: In this case, the cond-TST gave a comparable time with respect to the TST. This data set gave a bad performance for the Burst trie with respect to the TST, and that was why we used this data set to see show how our heuristic compared to the TST. Figure 3.9 presents the results, from which we can see that, for 30 million search operations, the Cond-TST took 88 seconds, the Splay-TST took 117 seconds, and

[9]The time shown here may be seen to be longer than the time given by the Burst trie. The time depends on the data and the implementation. Our main goal in this Chapter was to see the effect of adjusting heuristics on the TSTs, and all the heuristics were built using the same implementation for the basic TST. Also, the time for parsing and printing the results are included with the time presented in the results.

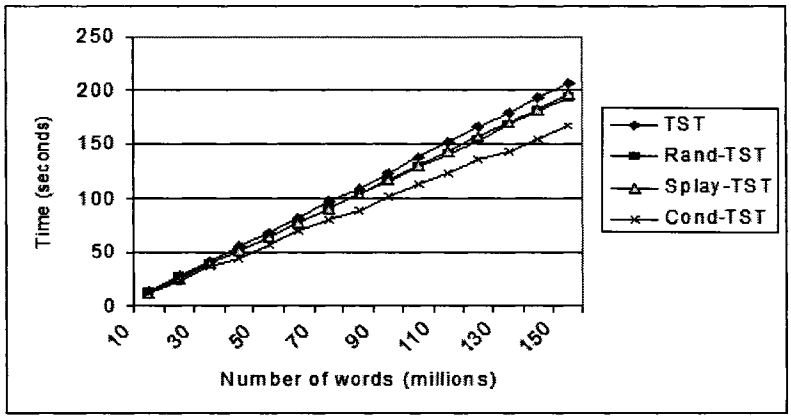


Figure 3.4: The accumulated time in seconds needed for all the data structures studied for various numbers of words, when the *Dict* dictionary was used, and the document was skewed by the Exponential distribution.

the Rand-TST required 88 seconds. The original TST (without any adjusting) took 85 seconds.

- *NASA data set*: This data set was used to test the case when the data itself was stored in a manner favorable to the TST, and for which the data itself led to the best possible TSTs. Since the strings had so many common prefixes, the dictionary led to a nearly optimal TST. Figure 3.10 shows the result of comparing the new heuristics with the basic TST. The results show that the Cond-TST has the ability to learn when to stop "self-adjusting", and to learn that for some data that tree will not need adjusting at all! In this case, the negative side effect for the splaying and randomizing the TST become apparent, because they disturb the original data so as to lead to a TST that is inferior to the one originally stored. For example, for 70 million search operations, the Cond-TST took 190 seconds, the Splay-TST took 275 seconds, and the Rand-TST required 270 seconds. The original TST (without any adjusting) took 206 seconds.

From the results presented above, we see that the Cond-TST has the ability to learn the characteristics of the data set and to adjust the TST according to these characteristics. It overcomes the side effect of the Burst tries which need parameter adjustments, and the disadvantage of the splaying heuristic that takes a very long time for adjusting the structure even when it is not needed. Also, the Cond-TST is much better than the randomized heuristics which are only

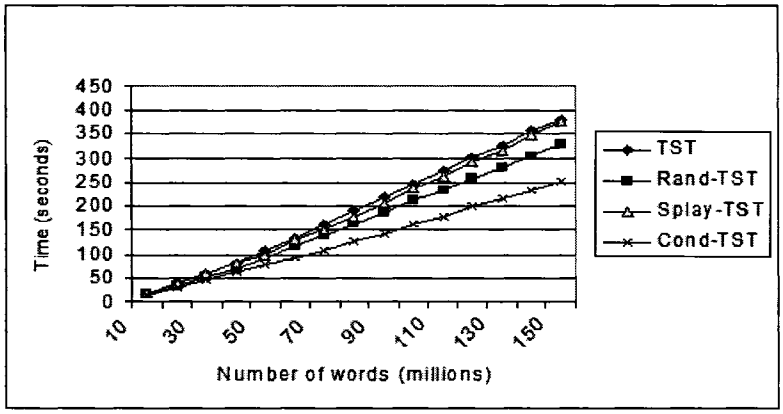


Figure 3.5: The accumulated time in seconds needed for all the data structures studied for various numbers of words, when the *Dict* dictionary was used, and the document was skewed by the wedge distribution.

Heuristics	Dict	Webster	Genome	NASA
TST & Splay-TST	3,183,960	11,053,536	20,967,168	2,724,432
Rand-TST & Cond-TST	3,714,620	12,895,792	24,461,696	3,178,504

Table 3.3: Space required, in KB, by the different data structures for the different dictionaries used.

suitable for uniformly distributed data sets. The *minor* disadvantage of the Cond-TST is the marginal extra space requirement needed for storing the access information. This disadvantage is minimal (i.e., it requires only linear extra space) considering the availability of inexpensive memory in present-day systems. The space requirements are summarized in Table 3.3.

3.6 More Related Work

The literature also records various schemes which adaptively restructure the BST with the aid of additional memory locations. The two outstanding schemes in this connection are the **monotonic tree** and Mehlhorn's D-Tree [33, 82]. The monotonic tree is a dynamic version of a tree structuring method suggested by Knuth [82] as a means to structure a nearly-optimal static tree. The static monotonic tree is arranged in such a manner that the most probable

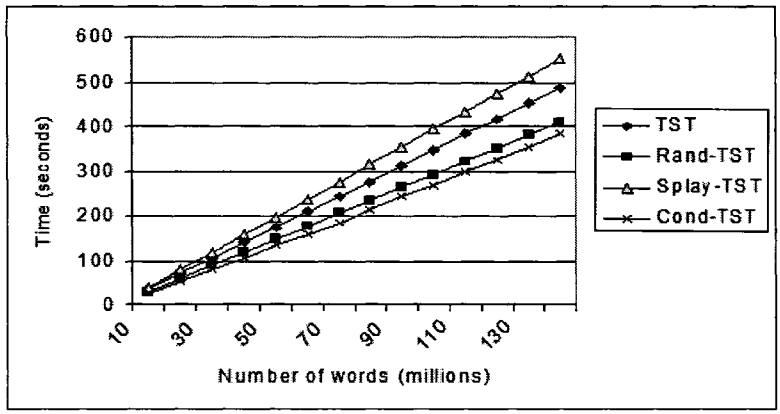


Figure 3.6: The accumulated time in seconds needed for all the data structures studied for various numbers of words, when the *Dict* dictionary was used, and the document was skewed by the Uniform distribution.

key is the root of the tree, and the subtrees are recursively ordered in the same manner. The static version of this scheme behaves quite poorly [97]. Walker and Gotlieb [157] have presented simulation results for static monotonic trees, and these results also indicate that this strategy behaves quite poorly when compared to the other static trees known in the literature.

Bitner suggested a dynamic version of this scheme [33], which could be used in the scenario when the access probabilities are not known *a priori*. Each record has one extra memory location, which "counts" the number of accesses made to it. The reorganization of the tree after an access is then very straightforward. When a record is accessed, its counter is incremented, and then the record is rotated upwards in the tree until it becomes the root of the tree, or it has a parent with a higher frequency count than itself. Over a long enough sequence of accesses, this will, by the law of large numbers, converge to the arrangement described by the static monotonic tree. Although this scheme is intuitively appealing, Bitner determined bounds for the cost of a monotonic tree, and showed that it is largely dependent on the entropy, $H(S)$ of the probability distribution of the keys. If $H(S)$ is small, then the monotonic tree is nearly optimal; but if $H(S)$ is large, it will behave quite poorly. Bitner also stated a result, proving that the expected entropy of a randomly chosen probability distribution is $\log(N) - \ln\ 2$, which is nearly the maximum entropy attainable. He concluded from this, that on the average the monotonic tree scheme will behave poorly . The experimental results [44] support this viewpoint.

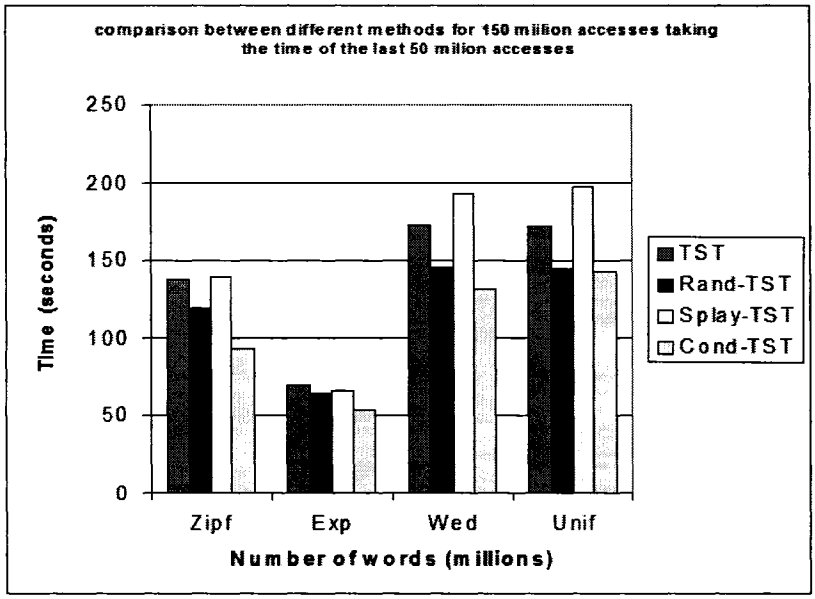


Figure 3.7: A comparison between the different data structures when various *Dict* documents were accessed against the *Dict* dictionary for 150 million accesses, when the ensemble average of the time required for the last 50 million accesses was measured.

As opposed to the above, Mehlhorn's D-tree is a BST scheme significantly different from the Monotonic Tree. At every node the D-tree maintains counters which record the weights of the two subtrees at the node, where the weight of a subtree is defined as the number of leaves in that subtree. D-trees permit multiple leaves for the same object, for indeed, each time an object is searched, the number of leaves referring to that object is increased by unity. The searching technique in the scheme ensures that all searches will be properly directed to corresponding objects at the leaf level. In any actual implementation of the D-tree, both search-time and space can be saved by coalescing a significant number of leaves into a single "super-leaf". The D-tree uses the weights of the two children subtrees at each node as the input parameters to a balancing function which provides a numeric measure for how "balanced" the subtree is. If the balancing function of any node in the tree exceeds this threshold while executing a search for a record, single and/or double rotations are executed at strategic nodes along the search path which ensure that the D-tree remains balanced after the rotations are executed. This scheme is closely related to the $BB[\alpha]$-trees also described by Mehlhorn. The latter uses the weight (defined as in the D-tree) of the subtrees of a tree as a method of quantifying how balanced the tree is. A node in the $BB[\alpha]$-tree scheme is considered to be balanced if the balancing function,

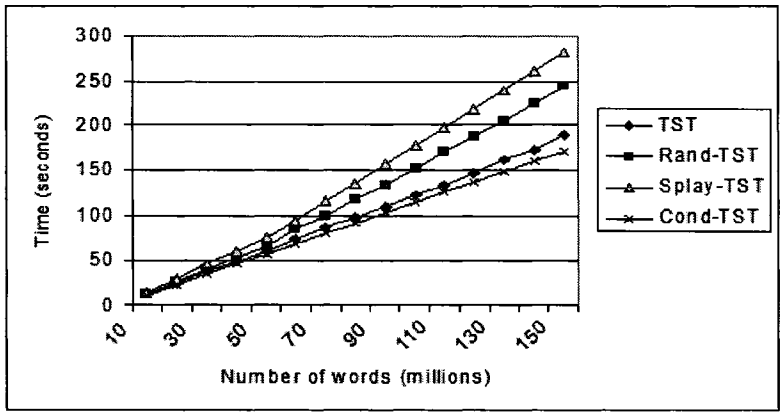


Figure 3.8: The accumulated time in seconds needed for all the data structures studied for various numbers of words in *Moby Dick* was searched against the *Webster's Unabridged Dictionary* in the order of occurrence in the novel.

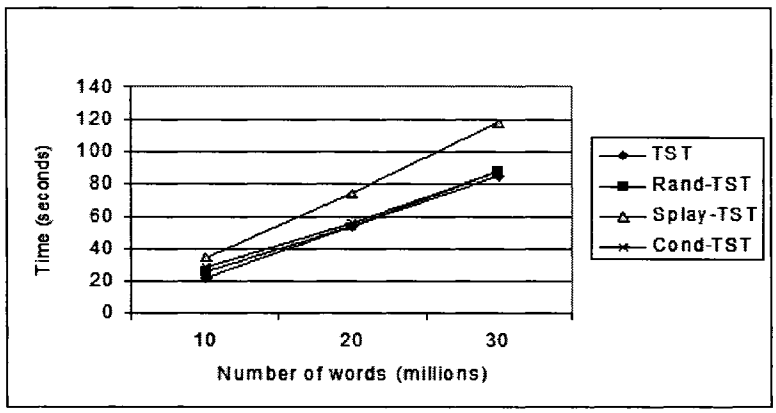


Figure 3.9: The accumulated time in seconds needed for all the data structures studied for various numbers of words in the *Genome* document when searched against the *Genome* dictionary in the order of occurrence in the document.

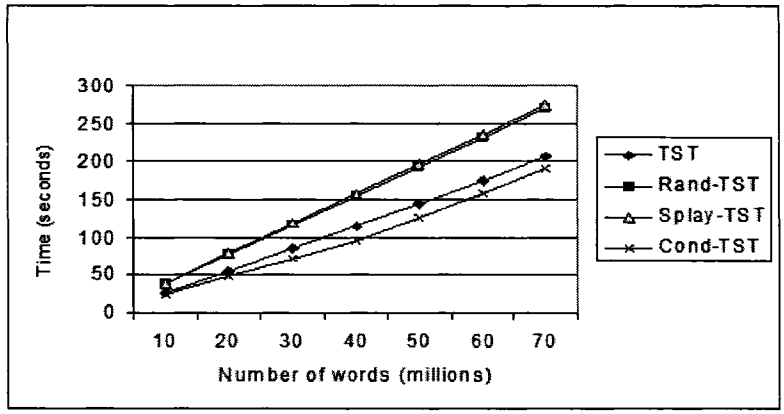


Figure 3.10: The accumulated time in seconds needed for all the data structures studied for various numbers of words in the *NASA* document when searched against the *NASA* dictionary in the order of occurrence in the document.

which takes the weight of the left and right subtrees as parameters, returns a value which is bounded by the variable α. If the tree is reckoned to be "unbalanced", just as in the case of the AVL-tree [82], the $BB[\alpha]$-tree reorganizes the nodes using single or double rotations.

All these techniques that have been used previously for BSTs can now be applied to TSTs. Such a study is open. In this study, we have chosen three techniques that were previously demonstrated to be among the most superior ones when applied to BSTs. This has motivated us to use them for TSTs. We have also shown that their performance is superior (with respect to time) when compared to TSTs without any balancing heuristics.

3.7 Conclusion

In this Chapter we applied two self-adjusting heuristics that were previously used for BSTs, to TSTs. These heuristics were, namely, the **splaying** (Splay-TST) and the **Conditional Rotation** (Cond-TST) heuristics. We also applied another balancing strategy used for BSTs, to TSTs, which was the basis for the **Randomized** search trees, or Treaps, (Rand-TST). The formal properties of the data structure have been stated. Numerous experiments were done to investigate the performance of the different adjusting heuristics when applied to TSTs so as to see how they perform in the context of different probability distributions characterizing various

benchmark data sets. The comparison (made on "moderate-size" data sets) was measured in terms of the running time, quantified in seconds.

From the results, we unequivocally conclude that the Cond-TST is the best scheme that can be used to improve the performance of the original TST, and that it has the ability to learn when to stop "self-adjusting" whenever the tree will not need further adjustments.

Chapter 4

Dual-Tries

4.1 Introduction

As stated in Chapter 2, the high memory usage of tries has long been recognized as a serious problem, and many techniques have been proposed to reduce their memory requirements. Proposals for modified tries that address the issue of high memory usage can be broadly classified in two groups [69]: Reduction in trie node *size*, and reduction in the *number* of trie nodes. In the previous Chapter (Chapter 3), we proposed self-adjusting techniques for TSTs where we utilized the fact that a TST is a representation for the trie data structure that helps to decrease the *space* by modifying the representation of a node to be a BST.

This Chapter addresses the problem of reducing the size of the trie by reducing the *number* of nodes it contains. It generalizes the whole concept of using tries by also incorporating the concept of "direction" from both storage and processing perspectives. The Chapter introduces a new data structure, namely the *Dual-Trie* (DT), that incorporates the idea of using two tries, one for the front of the string and the other for the rear.

The difference between the DT and the Two-Trie (Section 2.2.3) described in [64, 101, 140] will be outlined in this Chapter. However, both structures share a common goal, namely that of decreasing the storage space of the trie by trying to decrease the number of nodes needed to build the trie, while attempting to maintain the same search time for the standard trie. This

decrease in the number of nodes is a result of making simultaneous use of the common suffixes and the common prefixes of the words stored in the dictionary.

The organization of this Chapter is as follows: Section 4.2 reviews the two-trie data structure proposed by Aoe *et al.* [64, 101, 140] and shows the difference between the two-trie and the compact trie. Section 4.3 describes, in detail, the proposed Dual-Trie (DT) data structure. It starts by presenting the main idea, and then proceeds to describe the DT structure itself, after which we present an example for the DT. Section 4.4 describes the different operations that can be done on the DT that render it a fully dynamic data structure for storing strings. Section 4.5 gives the experimental setup and the different data sets used in the experiments, describes the experiments done, and presents a comparative survey of the results. Section 4.6 shows how we can apply the self-adjusting techniques that were used in the previous Chapter to increase the performance of the DT in various situations, and presents the pertinent experimental results. Section 4.7 concludes the Chapter.

4.2 Two-trie

Aoe *et al.* [64, 101, 140] described an algorithm for the compaction of a trie, applicable to a dynamic set of keys. They named the new trie structure the "two-trie". It follows the same concept of the compact trie (see Section 2.2.2) to reduce the number of trie nodes, and this is done by omitting chains of nodes that have only a single descendant, and thus lead to a leaf. However, the difference here is the way in which these chains are stored. The essential idea is to construct *two* tries, one for the front compression of keys, and the second for the rear. The rear trie is actually used for storing these chains of nodes that have only a single descendant leading to a leaf. This structure and the resulting algorithms were based on maximizing the benefits of strings that share common prefixes *and suffixes*. In Chapter 2 (Section 2.2.3), we gave an overview about the two-trie [64, 101, 140] data structure. More details will be summarized presently.

The idea utilized here is similar to a Directed Acyclic Word-Graph (DAWG) [90]. It differs from the latter in that the two-trie approach can uniquely determine the information corresponding to keys, an issue which the DAWG cannot handle. The space savings of such a two-trie

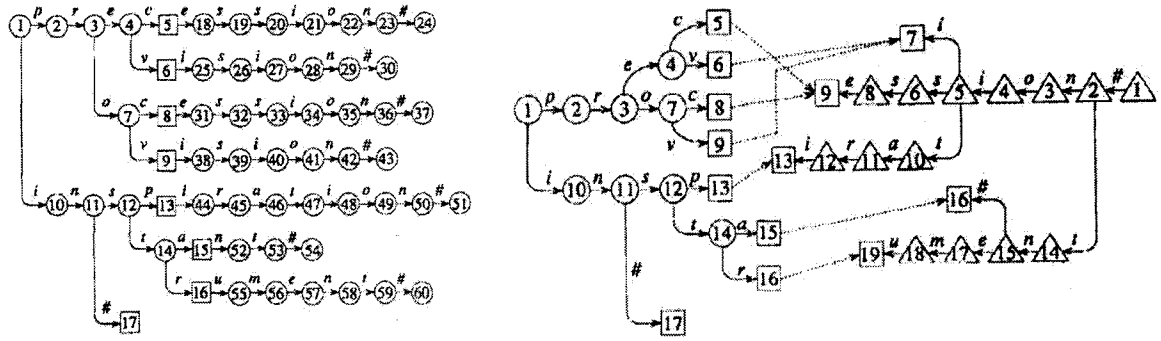


Figure 4.1: An example (adapted from [140]) for the trie and the corresponding two-trie for the words {precession, prevision, procession, provision, inspiration, instant, instrument, in}.

structure, when compared to a compact trie [69], was reported to be about 20%, whereas the insertion and retrieval time of a two-trie was reported to be similar to that of a (non-compacted) standard trie [140]. Hence, with respect to compact tries, a two-trie structure is not practical for a large set of keys.

The strategy to recognize a string in the two-trie structure, is as follows [140]: A trie for the front strings, called the FR trie, is constructed for front compression, and a similar trie for the set of reversed strings for the rear strings, called the RE trie, is constructed for rear compression. The accepting states of the FR and RE tries are the states associated with recognizing front strings and the reverse strings of the rear strings respectively. An accepting state of the FR trie is called a *Separate* state. The accepting node is the first node in the original trie that can lead to a single chain of nodes that have only single descendants leading to a leaf, and that is why it corresponds to only a single string. For key retrieval, separate states on the FR trie must be connected to the corresponding accepting state to be advanced successfully on the RE trie. The RE trie needs to traverse transitions in the opposite direction for the recognition of a key, and in both directions for updating a key. A Separate state is the first state which distinguishes the object key from the other competing keys. Thus, the information for the keys can be attached to separate states from the FR trie. Figure 4.1 shows an example[1] (adapted from [140]) for the trie and the corresponding two-trie.

Shishibori *et al.* [140] found that:

[1]This Figure appears both here and in Section 2.2.3, to render the Chapter self-complete.

- The time complexity for retrieving a key s is $O(|s|)$.
- For insertion and deletion, the time complexity is $O(|A| + |s|)$, where $|A|$ is the cardinality of the alphabet.

Shishibori *et al.* [140] also presented two other implementations of two-tries using a list structure and a double-array [12] structure to obtain superior compression. The list structure was suitable for applications that require frequent updates of the two-trie, but it leads to a slower retrieval. To compensate for this, the double-array structure was presented, leading to a faster implementation. In this case:

- The time complexity of retrieving a key in the list structure is $O(|A||s|)$.
- The time complexity of deletion and insertion in the list structure is $O(|A||s|)$.
- For the double-array, the time complexity of retrieving a key is $O(|s|)$.
- It was difficult to evaluate the time complexity for updating a key merely in terms of $|s|$ and $|A|$. By introducing a new parameter h, which is the total number of empty locations of the double-array, inserting a key takes $O(h|A||s|)$. This demonstrated that the time complexity for updating the double-array structure was more than that required for the list structure. The experimental observations showed that the internal parameter h is a very small value.
- The time complexity of deleting a key in the double-array structure is $O(|A||s|)$.
- If the internal parameter h of the double-array can be neglected, then the complexity of the list structure and the double-array structure depends on the number of elements used for representing each node, and the number of bytes representing those elements.

4.3 The Proposed Dual-Trie (DT) Sructure

As we saw in the previous Section, the Two-trie presented a way by which the number of trie nodes can be reduced by omitting chains of nodes that have only a single descendant, and thus

lead to a leaf. This is done by storing these chains in another trie in the reverse direction. The second trie helps to reduce the number of nodes by making use of the common suffixes that exist in these chains. The problem with the two-trie structure is that if the number of keys is large, then the compression of the single descent nodes will be less beneficial.

4.3.1 Main idea

The main idea for the proposed Dual-Trie (DT), comes from the fact that we need to maximize the benefits of strings that share common prefixes *and suffixes* without being restricted only to common suffixes in single descent chains that lead to leaves. Let X to be a string, of length N, in a dictionary H, which will be stored in a trie, T. The idea is to divide the lengths of the strings in the dictionary according to some ratio $f : r$, where f represents the portion of the string of length $\lfloor N * (f/(f+r)) \rfloor$ that is stored in the Front (FR) trie and r represents the rest of the string to be stored in the Rear (RE) trie, but in the reverse order.

When searching a string in the DT, we divide the string in the same ratio $f : r$, and then we search for the first part in the FR trie and for the reverse of the second part in the RE trie.

4.3.2 The DT Structure

Each node (including a leaf) in the FR trie represents a certain prefix for some words in the dictionary, and each node (including a leaf) in the RE trie represents a certain suffix for some words in the dictionary, except that it is in the reverse order. The problem here is that a leaf in the FR may not be uniquely matched with a leaf in the RE trie. A leaf that represents the first part of a string stored in the FR trie can correspond to more than one leaf that represent the (second) reversed portions of some (possibly other) strings stored in the RE trie, strings that share the same prefix represented by that leaf in the FR trie, and vice versa.

To solve this problem, we assign a unique code to each leaf in the FR (RE) trie. When the second part of the string is stored in the RE (FR) trie, the leaf node for the RE (FR) trie may store a group of codes that corresponds to the possible prefixes (suffixes) that are represented by those leaves in the FR (RE) trie, and which represent the first parts of the strings that have

this suffix (prefix). In this Chapter, we assume that we will assign unique codes to the leaves of the FR trie, and the possible group of codes are stored in the RE trie. These unique codes can be maintained by quite simply using unique integer values, and by maintaining a single additional quantity which is the largest code that has already been assigned. During deletion, there will be some codes that may not be used any more. These codes can be stored for future use in a linked list or a stack. During insertion, if a new code is needed, one of these codes can be reused again. If there is no such available code, a new code is generated by incrementing the integer associated with the largest code used so far.

This group of codes that are stored at each leaf of the RE trie can be stored in different ways, for example, in a list, a hash table[2], or in a BST. BSTs are much faster and have a superior representation as they can be easily balanced, and they also have a high search performance. The total number of nodes that will be stored in these BSTs will be equal to the total number of strings in the dictionary. Each node in the BSTs corresponds to a unique string in the dictionary and can be used to store data that corresponds to this string for a future retrieval. This node also assists the system in knowing whether the string exists in the dictionary or not.

4.3.3 Example

Figure 4.2 shows an example for a standard trie and the corresponding DT structure. The structures are built from 24 n-grams of length 6 for Genomic data. The figure shows both the FR and the RE tries. The BSTs for the codes are assumed to be stored within the leaves of the RE trie. The examples show the case when the DT structure is built for an $f : r$ ratio of unity. From the figure we see a great saving in the number of the trie nodes for the DT structure. The standard trie will need 62 trie nodes to store the 24 words while the DT structure will need only 34 trie nodes. From the example, it is clear that the two-trie built from this group of words will not lead to any saving in the space. This is the case where there is no single descent nodes leading to a leaf, which is also the case for which the benefits of the two-trie cannot appear.

To search for the string *gtcatc*, we first divide it into two equal parts, *gtc* and *atc*. We then

[2]Hash tables are known to be very efficient in searching, because only a constant time will be needed to access a key, although they are more complex if there are a lot of updates. Other techniques, such as Dynamic Perfect Hashing [7], can be used, and this is one of the avenues for future research.

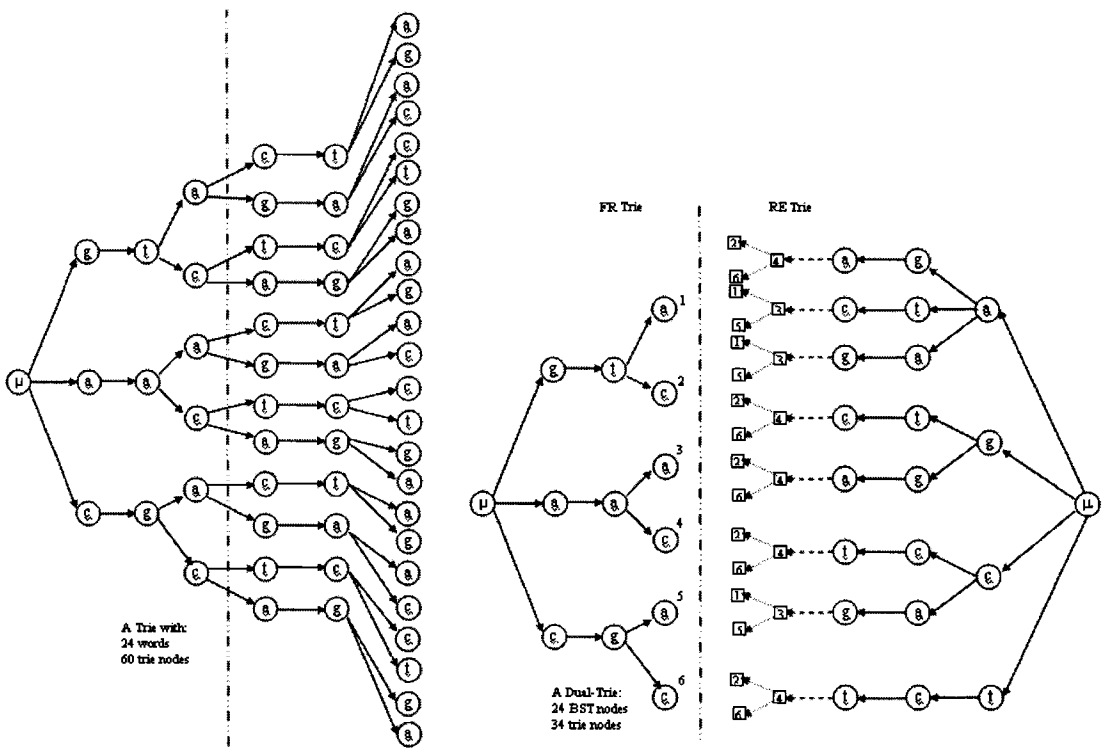


Figure 4.2: An example for a standard trie (left) and the corresponding Dual-Trie (right) for 24 n-grams of length 6 for Genomic data.

search for *gtc* in the FR trie, using the same technique applicable for searching the standard trie, and retrieve the code stored at the leaf node, which in this case equals '2'. After this, we search for the reverse of the string *atc* in the RE trie, where we reach a leaf. Finally, we search the BST stored at this leaf for that code which has already been retrieved from the FR trie. When that code is found, it means the original string is found in the dictionary.

4.4 Operations on DT

We advocate that the DT data structure can be considered as a fully dynamic structure for storing strings, which supports Insertion, Retrieval and Deletion operations. In this Section, we will discuss these different operations in detail, present their respective algorithm, and finally give an analysis about each of their time complexities. The complete theoretical analysis for the

space complexities for the DT structure (for different $f : r$ ratios) will be left as a future work. The complete empirical analysis for the space complexities for different $f : r$ ratios will be given in Section 4.5.

4.4.1 Insertion

Given the ratio $f : r$ and the input string X of length N, the insertion operation starts by dividing the string X into two parts. The first part, $frontX$, will be of length $\lfloor N * (f/(f+r)) \rfloor$ and the second part, $rearX$, will be of length $N - |frontX|$. Observe that we simultaneously operate on two tries, the FR trie and the RE trie. The string $frontX$ will be inserted into the FR trie following the same procedure for inserting any string in a standard trie, when the nodes are implemented as an array of size equals the size of the Alphabet. The string $rearX$ will be reversed and inserted in the RE trie, also using the same procedure for inserting any string in a standard trie.

Each distinct string, $frontX$, will be given a code, called $prefixCodeX$, that will be stored at the corresponding leaf of the FR trie. Then, when storing the string $rearX$ in the RE trie, the corresponding code $prefixCodeX$ will be inserted at the BST stored at the corresponding leaf for the string $rearX$ in the RE trie. In this way, the FR and the RE tries are linked by these codes, and by the BSTs stored at each leaf of the RE trie. We also need to store at each leaf of the FR trie a counter, $refRE$, that shows the number of BSTs which maintain the same $prefixCodeX$. This counter will be incremented each time a new string, which has the same prefix $frontX$, has to be inserted. This counter will be important later in the deletion process. Note that the BSTs will maintain only distinct codes, because if any code is repeated, it will imply that the string already exists in the DT structure.

The pseudo code for inserting a string in a DT is given in Algorithm 4.1. For simplicity, we assume that only distinct strings are inserted.

Algorithm 4.1 Algorithm DT-Insert

Input: A distinct string X of length N with data $dataX$ to be inserted in the Dual-Trie DT.
Output: The resultant DT.
Method:

1: $frontX$ = the prefix of X of length $\lfloor N * (f/(f+r)) \rfloor$
2: Insert $frontX$ in $DT.FR$.
3: **if** $frontX$ is a new prefix **then**
4: $prefixCodeX$ = a new code.
5: Store $prefixCodeX$ with the corresponding leaf of $DT.FR$.
6: Set $refRE = 1$ at that leaf.
7: **else**
8: $prefixCodeX$ = the code stored at the corresponding leaf of $DT.FR$.
9: Increment $refRE$ stored at that leaf.
10: **end if**
11: $rearX$ = reverse$(X - frontX)$.
12: Insert $rearX$ in $DT.RE$.
13: Insert $prefixCodeX$ in the BST stored at the corresponding leaf of $DT.RE$.
14: Store $dataX$ in the corresponding BST node.
15: **Return** {The new DT.}
16: **End DT-Insert**

4.4.2 Retrieval

We now consider the retrieval operation for the DT. Given the ratio $f : r$ and the string X of length N, the search process for the string X will start by dividing the string again in the same ratio $f : r$. Again, the first part, $frontX$, will be of length $\lfloor N * (f/(f+r)) \rfloor$ and the second part, $rearX$, will be of length $N - |frontX|$. The search process continues by searching the $frontX$ string in the FR trie and retrieving the corresponding code, $prefixCodeX$, that will be found at the corresponding leaf. It continues then by reversing the $rearX$ string and search for it in the RE trie starting from its root. The corresponding, pefix code, $prefixCodeX$, will be used to search the BST stored at the corresponding leaf of the RE trie, determining if the string exists or not.

The pseudo code for retrieving a string in a DT is given in Algorithm 4.2.

Algorithm 4.2 Algorithm DT-Search

Input: A string X of length N to be searched for in the Dual-Trie DT.
Output: The string found or not. If it is found, the algorithm will return the data, $dataX$, stored with that string.
Method:

1: $frontX$ = the prefix of X of length $\lfloor N * (f/(f+r)) \rfloor$
2: Search $frontX$ in $DT.FR$.
3: **if** $frontX$ is not found **then**
4: **Return** {X is not found.}.
5: **else**
6: $prefixCodeX$ = the code stored at the corresponding leaf of $DT.FR$.
7: **end if**
8: $rearX$ = reverse($X - frontX$).
9: Search $rearX$ in $DT.RE$.
10: **if** $rearX$ is not found **then**
11: **Return** {X is not found.}.
12: **else**
13: Search $prefixCodeX$ in the BST stored at the corresponding leaf of $DT.RE$.
14: **if** $prefixCodeX$ is not found **then**
15: **Return** {X is not found.}.
16: **else**
17: Retrieve $dataX$ stored at the corresponding BST node.
18: **end if**
19: **end if**
20: **Return** {X is found, $dataX$.}
21: **End DT-Search**

4.4.3 Deletion

The deletion process will again start by dividing the string X to be deleted in the same given ratio $f : r$ and then proceeding to search for $frontX$ in the FR trie. Here, we glean the advantage of storing the counter $refRE$. When the $frontX$ is found in the FR trie, we encounter two cases:

- $refRE > 1$: This means that the prefix $frontX$ is referred by more than one leaf in the RE trie. Since this implies that this code is still in use, no change is needed in the FR trie.
- $refRE = 1$: This scenario implies that this is the last time the prefix $frontX$ will be needed, and consequently all single descent nodes that lead to this leaf in the FR trie will

Algorithm 4.3 Algorithm DT-Delete

Input: A string X of length N to be deleted from the Dual-Trie DT, where X is assumed to be already stored in DT.
Output: The updated DT.
Method:

1: $frontX$ = the prefix of X of length $\lfloor N * (f/(f+r)) \rfloor$
2: Search $frontX$ in $DT.FR$.
3: $prefixCodeX$ = the code stored at the corresponding leaf of $DT.FR$.
4: Decrement $refRE$ stored at that leaf.
5: **if** $refRE = 0$ **then**
6: Delete all single descent nodes till that leaf of $DT.FR$.
7: **end if**
8: $rearX$ = reverse$(X - frontX)$.
9: Search $rearX$ in $DT.RE$.
10: Delete $prefixCodeX$ from the BST stored at the corresponding leaf of $DT.RE$.
11: **if** The corresponding BST is empty **then**
12: Delete all single descent nodes till that leaf of $DT.RE$.
13: **end if**
14: **Return** {The new DT.}
15: **End DT-Delete**

have to be removed.

The deletion process proceeds by searching for the reverse of $rearX$ in the RE trie. As soon as $rearX$ is found, we have two cases for the size, $|BST_x|$, of the corresponding BST stored at that leaf:

- $|BST_x| > 1$: This means that the suffix $rearX$ is still needed for other string(s). In this case no change is needed in the RE trie.
- $|BST_x| = 1$: This means that this is the last time the suffix $rearX$ will be needed and all single descent nodes that lead to this leaf in the RE trie will have to be removed. At the same time, we also purge the BST_X that was stored at that leaf.

The pseudo code for deleting a string from a DT is given in Algorithm 4.3. For simplicity, we will assume that any string to be deleted is already stored in it.

4.4.4 Analysis

In this section will give the analysis for the different time complexities in the worst case for the different operations discussed in the previous section.

Theorem 4.1. *The worst case time complexity for inserting, searching, or deleting a string X of length N from a DT that stores a dictionary H of size $|H|$ is equal to $O(N)$ plus the time taken to insert, search, or delete a code respectively from the data structure used to store the group of codes in the leafs of the second trie.*

Proof. We will discuss the complexity for each operation separately by analyzing the corresponding algorithm. The analysis will only be shown for the "expensive" steps. All other steps that are not mentioned are known to have constant time complexities.

- Insertion: In Algorithm 4.1, it can be seen that the time complexity of the insertion process is much simpler than that of the Two-trie. In our case, we do not have to burden ourselves with issues involving updating the "Separate" nodes. So, quite simply, we state that the time complexity for inserting string X of length N will be $O(N)$ (Steps 2 and 12), in addition to the time needed for inserting the $prefixCodeX$ in the BST of the corresponding node of the RE trie (Step 13). The time for inserting this code will depend on the size of the corresponding BST which remains to be analyzed in the average case. In the worst case this will be $O(\log |H|)$.

- Search: In Algorithm 4.2, the time needed for searching the string X of length N will be exactly $O(N)$ (Steps 2 and 9) plus the time needed to search for the code in the corresponding BST (Step 13). Again, the time for searching this code will depend on the size of the corresponding BST, which is $O(\log |H|)$ in the worst case. Observe that the time for searching for codes will be insignificant for small BSTs, as will be empirically demonstrated in Section 4.5.

- Deletion: In Algorithm 4.3, the time taken for deletion will again be the time required for searching for the string (Steps 2 and 9) plus the time required for deleting single descent paths (Steps 6 and 12). So this time will be $O(N)$ plus the time needed to search for and delete a single code from the BST (Step 10).

□

From the above Theorem we can conclude that using a good data structure like balanced BSTs or hash tables for storing the group of codes can greatly enhance the time performance for all the operations available for the DT structure. This will be dominant in the case when the number of codes stored per leaf is large. The time for manipulating these codes can vary from $O(1)$ when hash tables are used to $O(|H|)$ when lists are used. In the case when BSTs are used for storing codes, this work can be further enhanced as will be explained in Section 4.6.

4.5 Experimental Results

The proposed DT data structure has been rigorously tested so as to evaluate its relative performance when compared to the trie and the two-trie [64, 101, 140] data structures.

The principal objective of our study was to compare the performance of the DT with that of the two-trie. The comparison was done for different ratios $f : r$, where f corresponds to the ratio of the front part of the stings that are stored in the FR trie, and r is the ratio of the string that is stored in the RE trie. Two sets of experiments were conducted as described below:

- Set I: The first set was done by increasing the FR trie ratio with respect to the RE trie ratio, i.e., by setting $r = 1$ and f taking the values $1 - 9$.
- Set II: The second set of experiments was done by increasing the RE trie ratio with respect to the FR trie ratio, i.e., by setting $f = 1$ and r taking the values $1 - 9$.

The aim of these two sets of experiments, was to test the effect of increasing one part of the string with respect to the other compared with the scenario of being both equal. The experimental results are reported for the case when the group of codes that can be stored at each leaf of the RE trie are represented as BSTs. This is because the list representation yielded a very poor performance especially when the number of codes was large. Again we encounter two cases: Either we give codes for the prefixes and store these codes at the leaves of the RE trie, or we give codes for the suffixes and store these codes at the leaves of the FR trie. In our

work, we chose the former, because we believe that the later results (although reversed) will be similar.

For our comparisons we conducted the same four sets of experiments on the four benchmark data sets as explained in Chapter 3, where each data set was divided into two parts: a *dictionary* and the corresponding *test documents*. The four dictionaries were: Dict (25,481 words), Webster's unabridged (91,479 words), 9-grams of the unique words found in the Genomic data (262,024 words), and the NASA (15,700 words) dictionaries with the same corresponding document files. The Dict dictionary is used to test the performance of different distributions for the corresponding artificially created documents. The Wester's unabridged dictionary represents a larger dictionary used to test the search performance of the novel *Moby Dick*. The Genome data set represents the case when the single descent paths that lead to the leaves are not significant, and is the case where the DT should show a great improvement in the storage space of the trie. The NASA data set represents the case of longer strings where there will be a larger probability of obtaining single descent paths. This is the case where the advantages of compressed tries appears, as in the case of two-tries, and where the DT should show a performance similar to the compressed trie.

The difference here is that we have reported only the total time taken to conduct the full search for a total of 45,000,000 accesses and not for a different number of words to be searched. We did this so that we could test the effect of changing the structure of the trie on the total time performance, and not for different number of searches as done in Chapter 3.

In each experiment, we stored the dictionary in the DT, and the strings in the corresponding document were used in the search process. The words of the dictionary were stored in the order that they appeared in the dictionary. For each data set, 100 parallel experiments were done for a large number of accesses so that a statistically dependable ensemble average could be obtained. Again, we tested only the schemes for cases of successful access operations; the primary aim being that of demonstrating the changes in the performance of the DT for different $f : r$ ratios and when compared against the standard trie and the two-trie data structures. The analysis of the results follows.

Dict	Trie	Two-trie	f=1	f=2	f=3	f=4	f=5	f=6	f=7	f=8	f=9
# nodes	107185	50351	23547	30364	37738	43096	43964	50569	53821	56636	56636
# FR nodes	0	43058	11901	27109	36375	42388	43414	50240	53543	56408	56408
# BSTs	0	0	9838	3125	1350	704	548	328	277	227	227
max. |BST|	0	0	60	420	580	1017	1017	2709	3583	4445	4445
Zipf time	56	55	71	171	405	>>	>>	>>	>>	>>	>>
Exp. time	34	36	37	47	87	>>	>>	>>	>>	>>	>>
Unif. time	72	66	91	227	609	>>	>>	>>	>>	>>	>>

Table 4.1: Space and time performance for the different data structures when the **Dict** dictionary of 25,481 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the f portion ranging from 1 to 9, when r is set to unity. The symbol ">>" means that the time taken is relatively much more than the other corresponding times reported in the table.

4.5.1 Analysis of Results

We now present the results obtained for each data set. The results can be summarized according to each data set as follows:

- *Dict data set*: Tables 4.1 and 4.2 show the results for the different simulated documents. Each Table shows the accumulated time needed for successful search operations for a total number of 45,000,000 words, and for different probability distributions. From the results we see that the ratio 1 : 1 is the best for this dictionary. The DT structure in this case will have 23,547 trie nodes plus 25,481 BST nodes. We know that the space required for BST nodes is much smaller than that for trie nodes. For the two trie, the total number of nodes is 50,351 trie nodes. This represents a saving of 53% in the number of trie nodes. The results also shows that this dictionary has more common prefixes than common suffixes, and that is why increasing the f portion yields a better space saving than increasing the r portion. The time performance of the DT is comparable to the the time performance of the two-trie. The DT structure shows a relatively poor time performance when the number of nodes per BST is increased, a situation which can be adjusted as will be shown in Section 4.6.

- *Webster data set*: Tables 4.3 and 4.4 show the results for the second set of experiments

Dict	Trie	Two-trie	r=1	r=2	r=3	r=4	r=5	r=6	r=7	r=8	r=9
# nodes	107185	50351	23547	33222	44215	52317	57787	62093	66110	69661	72717
# FR nodes	0	43058	11901	2753	879	417	237	124	70	58	54
# BSTs	0	0	9838	18833	21800	23399	24525	25110	25335	25411	25445
max. |BST|	0	0	60	33	15	11	10	10	4	3	3
Zipf time	56	55	71	57	56	56	56	57	58	58	58
Exp. time	34	36	37	37	36	36	36	36	37	37	37
Unif. time	72	66	91	71	71	71	71	71	73	73	75

Table 4.2: Space and time performance for the different data structures when the **Dict** dictionary of 25,481 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the r portion ranging from 1 to 9, when f is set to unity.

done on a real-life data set. The results are comparable to those shown for the artificially created data for the Dict dictionary. From the results it is also seen that the equal ratio is the best when it concerns space performance. In this case, the number of trie nodes for the DT structure will be 58,441 in comparison with 174,275 tries nodes for the two-trie structure. This represents a saving of 66% in the number of trie nodes. Also, the time performance of the DT is comparable to the time performance of the two-trie. Here, the DT structure shows a poorer time performance when the number of nodes per BST is increased. This time performance can also be adjusted, as will be shown in Section 4.6.

- *Genome data set*: In this case, the DT structure shows an exceptional space and time performance in comparison with the two-trie data structure. The results are shown in Tables 4.5 and 4.6. For an equal $f : r$ ratio, the number of trie nodes stored in the DT structure will be only 1,706 in comparison with 350,113 trie nodes for the two-trie structure. *This represents a saving of* 99% *in the number of trie nodes*!

  For an equal ratio, the time performance is comparable to the two-trie, although it is not as good as for the latter. This can be solved by increasing the r ratio, and in this way the number of nodes stored in each BST will be smaller. Alternatively, we could also increase the f ratio, which is better in space performance than increasing the r ratio, and store the BSTs with the nodes of the FR trie. In this case the time performance will be very similar to that of the two-trie. For example, in Table 4.6, when $r = 6$, the total number of

Webster	Trie	Two-trie	f=1	f=2	f=3	f=4	f=5	f=6	f=7	f=8	f=9
# nodes	369086	174275	58441	82567	110071	130182	135879	155867	166641	176745	176750
# FR nodes	0	161922	29907	75922	107615	128958	134988	155418	166310	176486	176496
# BSTs	0	0	24767	6444	2439	1220	888	447	330	258	253
max. |BST|	0	0	536	1636	3256	4218	4551	6384	9276	12403	12403
time	37	38	58	292	843	>>	>>	>>	>>	>>	>>

Table 4.3: Space and time performance for the different data structures when the **Webster** dictionary of 91,479 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the f portion ranging from 1 to 9, when r is set to unity. The symbol ">>" means that the time taken is relatively much more than the other corresponding times reported in the table.

Webster	Trie	Two-trie	r=1	r=2	r=3	r=4	r=5	r=6	r=7	r=8	r=9
# nodes	369086	174275	58441	93771	135593	167003	188784	204943	217617	230051	241926
# FR nodes	0	161922	29907	5787	1686	628	270	181	122	53	27
# BSTs	0	0	24767	59087	74305	81031	85140	87797	89372	90225	90754
max. |BST|	0	0	536	59	30	14	11	6	6	7	4
time	37	38	58	44	43	42	42	42	42	42	42

Table 4.4: Space and time performance for the different data structures when the **Webster** of 91,479 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the r portion ranging from 1 to 9, when f is set to unity.

trie nodes will be 87,386, which represents a saving of 75% in the number of trie nodes. The time performance in this case, is the same as the two-trie and the standard trie. A further advantage for the latter can be gleaned by applying various balancing strategies to the BSTs.

- *NASA data set*: This data set was used to test the case of longer strings which could possibly contain many common suffixes and prefixes. The DT structure shows the same positive benefits as that of the two-trie. This is the case where we have many single descent nodes with common suffixes, which is the scenario in which the two-trie will show great benefits. The only difference is that we need to adjust the $f : r$ ratio to get the optimum value. The results are shown in Tables 4.7 and 4.8. In Table 4.7, when $f = 7$, the DT

Genome	Trie	Two-trie	f=1	f=2	f=3	f=4	f=5	f=6	f=7	f=8	f=9
# nodes	611549	350113	**1706**	5546	5546	21866	21866	21866	21866	87386	87386
# FR nodes	0	349465	**341**	5461	5461	21845	21845	21845	21845	87381	87381
# BSTs	0	0	**1024**	64	64	16	16	16	16	4	4
max. |BST|	0	0	**256**	4096	4096	16384	16384	16384	16384	65531	65531
time	82	84	**121**	180	179	2408	>>	>>	>>	>>	>>

Table 4.5: Space and time performance for the different data structures when the **Genome** dictionary of 262,084 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the f portion ranging from 1 to 9, when r is set to unity. The symbol ">>" means that the time taken is relatively much more than the other corresponding times reported in the table.

Genome	Trie	Two-trie	r=1	r=2	r=3	r=4	r=5	r=6	r=7	r=8	r=9
# nodes	611549	350113	1706	5546	21866	87386	87386	**87386**	87386	87386	87386
# FR nodes	0	349465	341	85	21	5	5	**5**	5	5	5
# BSTs	0	0	1024	4096	16384	65536	65536	**65536**	65536	65536	65536
max. |BST|	0	0	256	64	16	4	4	**4**	4	4	4
time	82	84	121	107	90	86	86	**85**	85	85	86

Table 4.6: Space and time performance for the different data structures when the **Genome** dictionary of 262,084 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the r portion ranging from 1 to 9, when f is set to unity.

structure shows some benefits in space savings with respect to the two-trie, and an 18% saving in the number of trie nodes. This means that this data set has longer common prefixes at the beginning. The time performance is also comparable to the two-trie, and can be further improved, as will also be shown in Section 4.6.

4.6 Self-Adjusting Dual-Tries

As we explained before, the relations between the FR trie and the RE trie were built by incorporating BSTs. The number of BSTs and the number of nodes stored in each BST depend on

NASA	Trie	Two-trie	f=1	f=2	f=3	f=4	f=5	f=6	f=7	f=8	f=9
# nodes	97800	56481	183530	109046	76921	62155	55280	49113	**46184**	46191	46549
# FR nodes	0	34163	5542	13953	20607	26051	29740	34192	**37540**	40350	42343
# BSTs	0	0	15506	14817	13649	12312	11579	9593	**5905**	4467	3221
max. |BST|	0	0	17	54	97	110	149	222	**278**	509	422
time	98	99	114	115	118	119	121	121	**122**	127	132

Table 4.7: Space and time performance for the different data structures when the **NASA** dictionary of 15,699 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the f portion ranging from 1 to 9, when r is set to unity.

NASA	Trie	Two-trie	r=1	r=2	r=3	r=4	r=5	r=6	r=7	r=8	r=9
# nodes	97800	56481	183530	266731	309132	336130	352359	365212	376649	381297	386301
# FR nodes	0	34163	5542	2107	1172	774	545	395	299	232	195
# BSTs	0	0	15506	15632	15666	15675	15683	15684	15688	15685	15690
max. |BST|	0	0	17	5	4	3	3	2	4	3	3
time	98	99	114	116	121	121	123	125	127	142	150

Table 4.8: Space and time performance for the different data structures when the **NASA** dictionary of 15,699 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the r portion ranging from 1 to 9, when f is set to unity.

the $f : r$ ratio and on the total number of strings in the dictionary that are stored in the DT. The larger the dictionary, the greater the total number of nodes in the BSTs, because the total number of nodes in all the BSTs is exactly the number of strings in the dictionary. Thus, the larger the f ratio, the greater the number of nodes in the FR trie, and the number of nodes in the RE trie and the number of BSTs are correspondingly smaller. As the number of stored BSTs decreases, the number of nodes stored in each structure increases, and furthermore, as the size of the BSTs increases, the load (in time) to the DT structure also increases. Thus, in this case, applying balancing strategies will help to reduce the load of the BSTs. Alternative solutions can be achieved by always storing the BSTs at the trie with a larger number of nodes. In this way, the total number of BST nodes will be distributed in a larger number of BSTs.

From the results above, we observe that for most of the time, equal ratios, i.e., $f = r$, or

nearly-equal ratios give the greatest saving in space. From the Tables above we see for these ratios, we still have BSTs in which the number of nodes is in the range $17 - 536$, in which case balancing strategies can play a very positive role.

As the number of nodes stored in each BST increases, we observe that it is also expedient to utilize balancing or self-adjusting strategies, implying that all the techniques mentioned in Chapter 3 can now be readily applied. Consequently, for each BST we can apply the same heuristics, namely, the **splaying** [3, 145] and the **Conditional Rotation** [3, 44] heuristics. We have also applied the same balancing strategy which is the basis for the **Randomized** search trees, or Treaps [13, 53, 154], which can help in increasing the search performance when the BSTs are large. This will be empirically verified presently.

We repeated the same experiments done earlier to measure the performance of the DT after applying self-adjusting techniques to the BSTs that are used to store the groups of codes that may be needed at each leaf of the RE trie. The self-adjusting techniques tested were: The Splaying, the Conditional Rotation, and the Treap heuristics described in Chapter 3. The results are shown only for the case of increasing the f portion, as this is the case where BSTs will have a larger number of nodes. Tables 4.9 through 4.12 summarize the results for each dictionary and for each balancing strategy. The tables show a significant improvement for the time performance.

From the results above, we can summarize the following recommendations for the DT structure:

- We recommend that we set $f > r$ when we have the number of common prefixes greater than the number of common suffixes.
- Conversely, we recommend that we set Make $r > f$ when we have the number of common suffixes greater than the number of common prefixes.
- We recommend that we set $r = f$ when we have the number of common suffixes and the number of common prefixes to be approximately equal.
- We believe that it is advantageous to always store the BSTs for codes with the trie containing a larger number of nodes. In this way, a smaller number of nodes will be stored *within* each BST.

Technique	Distribution	f=1	f=2	f=3	f=4	f=5	f=6	f=7	f=8	f=9
Treap	**Zipf**	62	73	78	85	83	90	88	87	88
	Exp.	38	44	43	40	41	55	48	40	42
	Unif.	79	94	101	105	106	125	114	113	113
Cond.	**Zipf**	60	65	72	70	74	71	73	74	74
	Exp.	39	41	56	52	56	40	54	49	49
	Unif.	86	97	120	123	127	106	121	123	122
Splay	**Zipf**	62	70	77	81	82	87	88	90	90
	Exp.	39	39	38	38	38	39	38	39	39
	Unif.	86	112	125	132	134	141	145	148	148

Table 4.9: Time performance for the dual-trie when applying different adjusting techniques and when the **Dict** dictionary of 25,481 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the f portion ranging from 1 to 9, when r is set to unity.

- In case the number of nodes in the BSTs is large, we recommend that the algorithm involves a balancing strategy to help increase the time performance.
- The DT structure is very suitable for storing Bio-informatics data, which generally has a smaller alphabet size and where the strings have many common prefixes and suffixes. Also, the standard trie built for this kind of data will have a rather small portion of single descent nodes, which eliminate the advantage for storing them in the two-trie data structure. The results actually shows a saving of 99% in the number of trie nodes when they are stored in the DT structure.
- The DT structure shows the same performance for the compressed trie structures when the data (like the NASA data set) has a high number of single descent nodes. This is the case when the data has longer strings.

4.7 Conclusion

In this Chapter, we introduced a new structure, namely the Dual-Trie (DT) which introduced the concept of direction to reduce the space of the trie by decreasing the number of nodes. This

Technique	f=1	f=2	f=3	f=4	f=5	f=6	f=7	f=8	f=9
Treap	50	59	64	67	68	70	73	73	72
Cond.	46	48	52	56	57	53(11)	53(15)	53(26)	54(24)
Splay	47	52	56	60	59	63	64	64	64

Table 4.10: Time performance for the dual-trie when applying different adjusting techniques and when the **Webster** dictionary of 91,479 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the f portion ranging from 1 to 9, when r is set to unity. The values in brackets show the time taken for the learning phase of the Conditional rotation heuristic.

Technique	f=1	f=2	f=3	f=4	f=5	f=6	f=7	f=8	f=9
Treap	120	146	140	152	152	151	151	179	165
Cond.	105	118	118	134	134	134	135	157(160)	153(15)
Splay	150	186	185	203	203	225	203	203	203

Table 4.11: Time performance for the dual-trie when applying different adjusting techniques and when the **Genome** dictionary of 262,084 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the f portion ranging from 1 to 9, when r is set to unity. The values in brackets show the time taken for the learning phase of the Conditional rotation heuristic.

could be done by dividing the strings in the dictionary into two portions according a ratio $f : r$. The two portions are then stored in two different tries. The first one was stored in a forward trie, FR, and the reverse of the second one was stored in the so-called, RE trie. Such a modelling enabled us to get the maximum benefits of both the common prefixes and the common suffixes for the strings in the given dictionary. The two tries were linked by giving codes to the leaves of one trie, and by storing these codes in the corresponding leaves of the second trie in BSTs. These codes were then stored in the second trie only when both the prefix and suffix constituted a word in the dictionary. As the number of nodes stored in the BSTs increased, we suggested the application of balancing heuristics to increase the time performance.

The new structure demonstrated significant benefits for compressing data, gave a very good space performance, and at the same time, maintained the same time performance as the two-trie. It also yielded a high space performance for data involving Bio-informatics. For data of

Technique	f=1	f=2	f=3	f=4	f=5	f=6	f=7	f=8	f=9
Treap	102	100	99	99	99	101	101	105	104
Cond.	102	102	101	99	100	101	101	102	102
Splay	103	100	100	100	101	104	104	109	109

Table 4.12: Time performance for the dual-trie when applying different adjusting techniques and when the **NASA** dictionary of 15,699 words is used. The time is measured in seconds for the total time needed for 45 million accesses. The performance for the dual-trie is shown for different values of the f portion ranging from 1 to 9, when r is set to unity.

this type, the results demonstrated an improvement of 99% saving in the number of trie nodes with respect to the two-trie data structure. The DT structure was also shown to be profitable for storing a large number of strings, when the compact tries yielded a poor space performance. Finally, it was suitable also for data containing longer strings, in which case, the number of single descent nodes was large, and the performance of the DT structure had the same advantages as the two-trie data structure.

Part III

Tries for Syntactic Pattern Recognition

Chapter 5

Notations and Basic-Concepts

5.1 Introduction

The main objective of this Chapter is to collect all the important concepts and notations needed for this part of the Thesis. We will not present any new results here. We assume that the reader has a basic understanding of the design and analysis of algorithms, data structures, basic AI search techniques, and basic text algorithms.

The Chapter is organized as follows: Section 5.2 provides the basic notations, the problem definitions, and the respective applications. Section 5.3 describes the basic concepts for inexact string-to-string matching, the basic distance functions used, and the dynamic equations for calculating the string distances. Section 5.4 briefly reviews the various AI search techniques that we can benefits from in searching the trie space. Section 5.5 describes the basic assumptions that we use throughout this part of the Thesis. Finally, Section 5.6 concludes the Chapter.

5.2 Notation and The Problem Definition

This section defines the basic notations that we will be using throughout this part of the Thesis. It also defines the problems that this part of the Thesis addresses, and finally gives the

applications in which the proposed methods can be used.

5.2.1 Notations

In terms of notation, A is a finite alphabet, H is a finite (but possibly large) dictionary, T is the trie used to store the dictionary, H. M is the length of the noisy word Y and N is the length of a word in the dictionary, X. The left derivative of order one of any string $Z = z_1 z_2 \dots z_k$ is the string $Z_p = z_1 z_2 \dots z_{k-1}$. The left derivative of order two of Z is the left derivative of order one of Z_p, and so on. Table 5.1 summarizes the basic notations that we use in this part of the Thesis.

The machine used for most of the experimental results is PC Pentium 4 of 3.2GHz with a 1GB of RAM and a 80GB local disk. The machine was not performing other "heavy" tasks when the experiments were running. We measured the performance in terms of the number of operations (additions and minimizations). Although evaluating the performance in terms of time is very implementation dependent, sometimes, in the interest of benchmarking, we also provide the results in terms of the time utilized.

5.2.2 The Problem

This part of the thesis focuses on the problem of dictionary-based syntactic pattern recognition, when the dictionary is stored as a trie. For approximate string matching, the entire space for the trie has to be searched to get the nearest neighbors in the dictionary that are related to the noisy word. In this Thesis we are trying to enhance the search of the trie space by applying both blind and directed (heuristic) Artificial intelligence (AI) search techniques. We also enhance a heuristic search technique and apply the new technique to approximate string matching.

More specifically, we consider the traditional problem involved in the syntactic Pattern Recognition (PR) of strings, namely that of recognizing garbled words (sequences). Let Y be a misspelled (noisy) string obtained from an unknown word X^*, which is an element of a finite (but possibly, large) dictionary H, that is stored as a trie T. Y is assumed to contain Substitution, Insertion and Deletion (SID) errors and we are trying to obtain an appropriate

Name	Meaning	Domain
A	Alphabet	Finite set
$\|A\|$	Alphabet size	$\mathtt{N} \geq 2$
λ	The null symbol	-
$d()$	inter-symbol distance	$A \times A \longrightarrow \mathtt{N}$ or $\mathtt{R}$, ≥ 0
$D()$	Edit distance function	$A^* \times A^* \longrightarrow \mathtt{N}$ or $\mathtt{R}$, ≥ 0
H	A dictionary	A^*
T	The trie storing H	A^*
μ	The empty string	-
Y	A noisy word	A^*
X	A word in H	A^*
X^+	The nearest neighbor to Y in H	A^*
N	Length the word X	$\mathtt{N}$
M	Length of the noisy string Y	$\mathtt{N}$
Z_p	The left derivative of order one of Z	A^*
Z_g	The left derivative of order two of Z	A^*
K	Number or percentage of allowed errors	$\mathtt{N}$ or $\mathtt{R}$, ≥ 0
N_H	Number of words in H	$\mathtt{N}$, ≥ 0

Table 5.1: Primary variables and terms used in this part of the Thesis

estimate X^+ of X^*, by processing the information contained in Y.

5.2.3 Applications

The first references to this problem [105] are from the sixties and seventies, where the problem appeared in a number of different fields. At that time, the main motivation for this kind of search came from computational biology, signal processing, and text retrieval. These are still the largest application areas that motivate the research in this problem. The application domains where the methods presented in this part of the Thesis can be utilized are numerous and include:

1. **The Internet:** When searching the Internet, it is often the case that the user enters the word to be searched incorrectly. Our results can be used to achieve a proximity search of the Internet in order to locate sites and documents that contain words which closely match the one entered. They can thus be used to greatly enhance the power of search

engines.

2. **Processing of Biological Sequences:** Our results can be used to locate subsequences in sequences when the former are inaccurately represented. Thus, they have potential applications in the human genome project, in the detection of targets for diseases, and ultimately in the drug-design process.

3. **Speech Recognition:** If the waveform associated with a speech signal is processed to yield a string of phonemes, our results can be used to process the phoneme sequence in order to recognize the speech utterance or speaker.

4. **Spelling Correction:** Our results can be used to achieve the automatic correction of misspelled strings (substrings) in a document.

5. **Keyword Search:** When searching libraries and collections, it can occur that the user enters the keyword with spelling or phonetic errors. Our results can be used to search the library or respective collection by first determining the keywords which best match the entered string, and then executing the search.

6. **Optical Character Recognition:** If the digital pixels associated with a sequence of handwritten or printed characters are processed to yield a string of syntactic primitives, our results can be used to process the primitives sequence to recognize the words represented by the handwriting or sequence of printed characters.

7. **Applications in Communication Theory:** Finally, our results can be used for designing and recognizing fast convolution codes if their noisy versions are processed. They can thus be also used in communication channels for detecting symbols by finding the "most-likely" noiseless sequence.

5.3 String Distance

5.3.1 String-to-String Distances

Given the two strings X and Y, the distance between X and Y, $D(X, Y)$ is defined as [104]:

Definition 5.1. *The distance $D(X,Y)$ between two strings X and Y is the cost of the minimum-cost sequence of operations that transform Y into X. The cost of a sequence of operations is the sum of the cost of individual inter-symbol operations. The cost of an operation is considered a positive real number. If it is not possible to transform Y into X we say that $D(X,Y) = \infty$.*

In text searching applications the operations[1] of most interest are [99, 104]:

- *Insertion*: insert a new letter a into Y. An insertion operation on the string $Y = VW$ consists in adding a letter a, thus converting Y into $X = VaW$.
- *Deletion*: delete a letter a from Y. A deletion operation on the string $Y = VaW$ consists in removing a letter a, thus converting Y into $X = VW$.
- *Substitution*: replace a letter a in Y by letter b. A replacement operation on the string $Y = VaW$ consists in replacing a letter for another, thus converting Y into $X = VbW$.
- *Transposition*: swap two adjacent letters a and b in Y. A transposition operation on the string $Y = VabW$ consists in swapping two adjacent letters, thus converting Y into $X = VbaW$.

The operations are assigned costs, such that the more likely operations are cheaper. The goal of the optimization is to minimize the total cost. The possible costs that can be assigned to these operations are given in the next subsection.

If the distance function $D(X,Y)$ turns out to be symmetric (i.e., $D(X,Y) = D(Y,X)$ and if $D(X,Y) < \infty$, then $D()$ is a metric, i.e., it satisfies the following axioms [104]:

- $D(X,X) = 0, \ \forall \ X$
- $D(X,Y) > 0, \ \forall \ X$
- $D(X,Y) = D(Y,X), \ \forall \ X, \ Y$
- $D(X,Y) \leq D(X,Y) + D(Y,Z), \ \forall \ X, \ Y, \ Z$

  The last property is called the "triangular inequality".

[1] For the rest of the Thesis, we will assume that we are editing Y to get X.

The most commonly used distance functions (there are many others) are [99, 104]:

- *Levenshtein or Edit distance* [86, 87]: This distance permits insertions, deletions and substitutions. The costs can be of a 0/1 form or general. In the case of general costs, the distance is called the General Levenshtein Distance (GLD). For the case in which all non-zero costs are unity, the distance is symmetric, and $D(X,Y)$ satisfies $0 \leq D(X,Y) \leq \max(|X|,|Y|)$. In the literature, this problem is often called "String matching with K differences".

- *Hamming distance* [134]: This distance allows only substitutions. When the substitution cost is unity, the distance is symmetric, and it is finite whenever $|X| = |Y|$. In this case $D(X,Y)$ satisfies $0 \leq D(X,Y) \leq |X|$. In the literature this problem is often called "String matching with K mismatches".

- *Episode distance* [52]: This distance permits only insertions. In the literature this problem is often called "episode matching", since it models the case in which a sequence of events occurs within a short period. This distance is not symmetric, and it may not be possible to convert Y into X. Hence, when the cost of the insertion is unity, it turns out that $D(X,Y)$ is either $|X| - |Y|$ or ∞.

- *Longest Common Subsequence distance* [109]: This distance permits only insertions and deletions. The name of this distance comes from the fact that it measures the length of the longest pairing of characters that can be made between both strings, so that the pairing respects the order of the letters. The longer the pairing, the smaller the distance. When the unit operator cost is unity, the distance is symmetric and $0 \leq D(X,Y) \leq |X| + |Y|$.

When the distance is symmetric, one can imagine that the changes can be made on either X or Y. Insertions in X are thus the same as deletions from Y and vice versa, and replacements can be made in any of the two strings to match the other. In this Thesis we assume that all the operations are done on Y to transform it to the string X.

5.3.2 Computing String Distances

From now on, we will assume that we are using the *Levenshtein Distance* as the distance function for calculating the distance between the two strings and we will simply refer to it as the *Edit Distance.*

The inter-symbol distances $d(x, y)$ can be of a 0/1 sort, parametric or entirely symbol dependent. If they are parametric [42, 124]:

$$\begin{aligned} \textit{Substitution}: \quad & d(x,y) = r \quad if\ x \neq y \\ & = 0 \quad if\ x = y \\ \textit{Deletion and Insertion}: \quad & d(\lambda,x) = d(x,\lambda) = 1 \quad \forall\ x. \end{aligned}$$

If they are symbol dependent [76, 134], they are usually assigned in terms of the inter-symbol confusion probabilities as:

$$\begin{aligned} \textit{Substitution}: \quad & d(x,y) = -\ln[Pr(x \rightarrow y) \div Pr(x \rightarrow x)] \\ \textit{Insertion}: \quad & d(x,\lambda) = -\ln[Pr(x\ is\ deleted) \div Pr(x \rightarrow x)] \\ \textit{Deletion}: \quad & d(\lambda,x) = Kd(x,\lambda) \end{aligned} \tag{5.1}$$

where K is an empirically determined constant.

In all of these cases, the primary Dynamic Programming (DP) rule used in computing the string distance $D(X, Y)$ is:

$$\begin{aligned} D(x_1x_2\ldots x_i,\ y_1y_2\ldots y_j) = \min\ [\ & t_1 : \{D(x_1x_2\ldots x_{i-1},\ y_1y_2\ldots y_{j-1}) + d(x_i,y_j)\}, \\ & t_2 : \{D(x_1x_2\ldots x_i,\ y_1y_2\ldots y_{j-1}) + d(\lambda,y_j)\}, \\ & t_3 : \{D(x_1x_2\ldots x_{i-1},\ y_1y_2\ldots y_j) + d(x_i,\lambda)\}]. \end{aligned} \tag{5.2}$$

The proof of (5.2) is straightforward as in [104]. For two non-empty strings of length i and j, this can be proven by inductively assuming that all the edit distances between shorter strings have already been computed. Since there are only three possible operations, the distance $D(X_{1\ldots i}, Y_{1\ldots j})$ can be computed as the minimum of three easily computed terms:

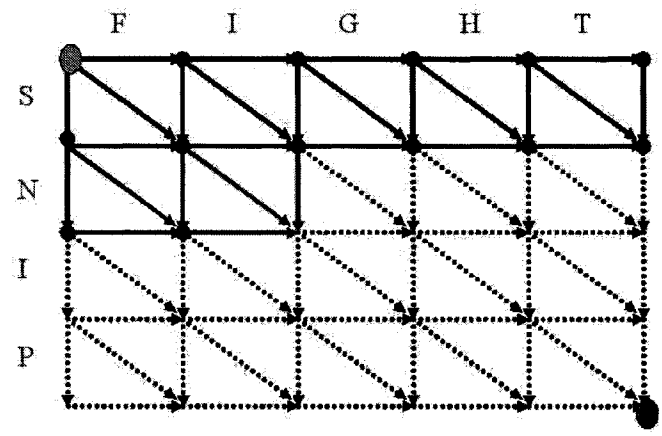


Figure 5.1: An example of the calculation flow in the Dynamic Programming Matrix DPM when it is used to compute the edit distance $D(snip, fight)$ where a row-wise left-to-right traversal is used.

- t_1: The cost of substituting y_j with x_i plus the previously calculated value $D(X_{1...i-1}, Y_{1...j-1})$.
- t_1: The cost of deleting y_j plus the previously calculated value $D(X_{1...i}, Y_{1...j-1})$.
- t_1: The cost of inserting x_i plus the previously calculated value $D(X_{1...i-1}, Y_{1...j})$.

To calculate the edit distance between two strings X and Y using Equation (5.2), a two dimensional matrix called the Dynamic Programming Matrix (DPM) (of size $M \times N$), is needed. The DP strategy must fill DPM in such a way that for any entry $DPM_{(i,j)}$, the upper ($DPM_{(i-1,j)}$), left ($DPM_{(i,j-1)}$), and upper-left ($DPM_{(i-1,j-1)}$) neighbors of this entry are computed *a priori*. This is easily achieved by either a row-wise left-to-right traversal or a column-wise top-to-bottom traversal. Figure 5.1 shows an example by which we can compute the edit distance $D(snip, fight)$ when a row-wise left-to-right traversal is used.

The algorithm needs $O(|X||Y|)$ time in the worst and average case. However the space required is only $O(\min(|X|, |Y|))$. This is because, in the case of a column-wise processing, only the previous column must be stored in order to compute the new one. Also, the sequence of operations performed to transform Y into X can be easily recovered from the matrix.

Recognition using distance criteria for the dictionary-based approximate string matching problem is obtained by essentially evaluating the string in the dictionary which is "closest" to

the noisy one as per the metric under consideration.

5.4 Artificial Intelligence Search Techniques

We now present the AI search techniques that will be used in this part of the Thesis. The task of a search algorithm is to find a solution path through a problem space. Search algorithms should keep track of the paths [91, 126] from a Start to a Goal node, because these paths contain the series of operations that lead to the solution of the problem.

This part of the Thesis is concerned with incorporating search techniques in dictionary-based approximate string matching when the dictionary is stored as a trie. That is why the main graph that has to be searched, possesses a trie structure. We will presently explain the possible search techniques applicable when a trie is used in the representation of the problem's search space.

5.4.1 Blind or Uninformed Techniques

Some search techniques are classified as being blind or uninformed [60] in the search, since the order in which nodes are selected is unaffected by the information concerning the goal node or the unexplored region between the explored nodes and the goal node. The search is conducted by the systematic application of simple rules. We will present two such "blind search" techniques.

Depth-first search: The strategy followed by a depth-first search is, as its name implies, to search "deeper" in the graph whenever possible. In a depth-first search, the search is conducted from the starting node as far as it can be carried until either a leaf or the goal is encountered. At each node some arbitrary rule (e.g., "take the left-most path") is used to direct the search. If a dead end is encountered, the search is backed up one level (this operation is called "backtracking"), and the next left-most branch is taken. This entire process is repeated until all nodes are explored. Figure 5.2 shows how the depth-first search applies to a trie example.

Breadth-first search: This strategy is a form of exhaustive search in which each of the nodes is systematically explored at each level until the goal node is attained. The search is conducted on a level-by-level basis. Figure 5.3 shows how the breadth-first search applies to a

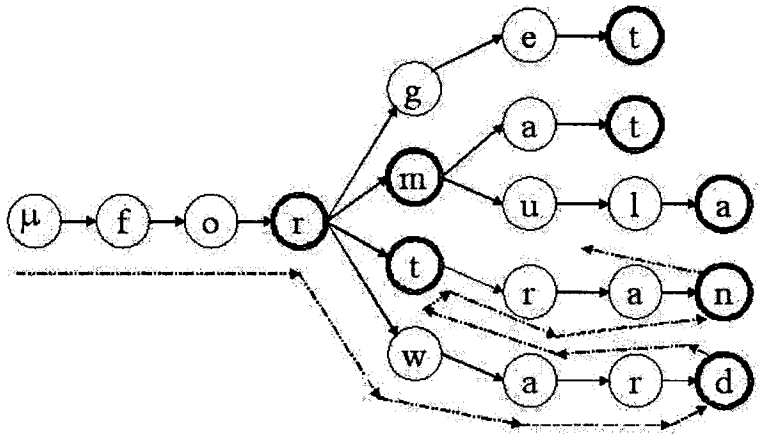


Figure 5.2: The depth-first search path on a trie example.

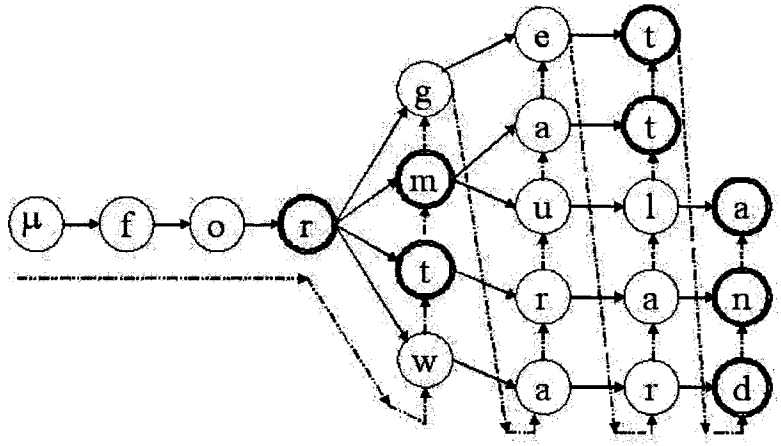


Figure 5.3: The breadth-first search path on a trie example.

trie example.

If the goal is known [59], the algorithm stops at the first goal state found. In this case, the depth-first and the breadth-first searches could return different solutions to the problem. The depth-first always finds the first solution, reading the tree from left to right, whereas breadth first finds the shallowest solution.

When we consider the case of having costs associated with solutions [59], we cannot stop when we have found the first solution. Instead, we must keep track of the best solution(s) found so far with their costs, and continue the search in case there is a better solution further on.

Indeed, we have to continue until the whole space is exhausted.

In approximate matching the goal is not known *a priori*, and besides, we have costs associated with the solutions. Consequently, the whole trie will have to be searched so as to obtain the nearest possible solution. The blind or uninformed search suffers from the high time complexity especially when the problem is approximate, but the accuracy is optimal as the entire space will be searched. To get better time complexities, we need to use more informative or Heuristic search techniques to prune the search space, with the probable sacrifice of a small accuracy.

5.4.2 Heuristic search

Heuristic search takes the advantage of the fact that most problem spaces provide, at relatively small computational cost, some information that makes a distinction among the nodes in terms of their likelihood of leading to a goal node. The search is called *directed* or *informed* [60], if the knowledge of the unexplored region, including the information on the goal node, is used to guide the search. This information is called a "heuristic evaluation function".

AI problem solvers employ heuristics in two basic situations [91, 126]:

- A problem may not have an exact solution because of inherent ambiguities in the problem statement or the available data. In this case heuristics are used to choose the most likely solution.
- A problem may have an exact solution, but the computational cost of determining it may be prohibitive. Heuristics attack this complexity by guiding the search along the most "promising" path through the space. By eliminating unpromising nodes and their descendants from further consideration, a heuristic algorithm can "defeat" this combinatorial explosion, and find an acceptable solution.

A heuristic is only an informed guess of the next step to be taken in solving a problem [91, 126], which is often based on experience or intuition. Because heuristics use limited information, they can seldom predict the *exact* behavior of the state space further along the search. A

heuristic can lead a search algorithm to a suboptimal solution, or fail to find any solution at all. This is an inherent limitation of a heuristic search.

Heuristics and the design of algorithms to implement heuristic search have long been a core concern of AI research. Game playing and theorem proving are two of the oldest applications in AI; both of these require heuristics to *prune* spaces of possible solutions. It is not feasible to examine every inference that can be made in a domain of mathematics, or to investigate every possible move that can be made on a chessboard, and thus a heuristic search is often the only practical answer. It is useful to think of heuristic algorithms as consisting of two parts: the heuristic measure and an algorithm that uses it to search the state space.

In the context of this part of the Thesis, whenever we refer to heuristic search, we imply those methods that are used for solving the problems possessing ambiguities or which are inherently extremely time consuming. In both these cases, the Thesis utilizes a heuristic that efficiently prunes the space and that also leads to a good (but potentially suboptimal) solution. We will presently give an overview of only the techniques that are used in the work of the Thesis. For other techniques like simulated annealing, Tabu search, and Genetic algorithms, the reader is requested to refer to the established literature, such as [83, 91, 126].

Hill-Climbing search: The simplest way to implement a heuristic search is through a *Hill-Climbing* (HC) procedure [91, 126]. Although HC is basically uninformed on the features of the unexplored region, it makes use of local knowledge about any particular node. HC strategies expand the current node in the search space and evaluate its children. The best child is selected for further expansion, and neither its siblings nor its parent are retained. The search halts when it reaches a node that is better than any of its children.

Because it keeps no history, the algorithm cannot recover from failures. A major problem of HC strategies is their tendency to become "stuck" at local maxima/minima. In spite of its limitations, HC can be used effectively if the evaluation function is sufficiently informative so as to avoid local optima. Also, the HC strategies have a high ability of pruning the search space, even if the problem is ambiguous.

Best-first search: Heuristic search can also use more informed schemes [91, 126], provided by the so-called best-first search. The best-first search, considers the most "promising" node. This is acceptable when an exact match is considered, with the hope that it will reach the

goal much faster, and it will also stop as soon as the goal is attained. But if the problem is ambiguous, the algorithm may have to search the entire space to find the nearest goal, or to yield the best candidates that can *approximate* the goal.

Beam search: To overcome the problems of HC, and to provide more pruning than the best-first search, researchers have proposed other heuristics such as the *Beam Search*[2] (BS) [133]. In BS, the process retains q nodes, rather than a single node as in HC, and these are stored in a single pool. The process then evaluates them using the objective function. At each iteration, all the successors of all the q nodes are generated, and if one is the goal, the process is terminated. Otherwise, the process selects the best q successors from the complete list and repeats the process. This BS avoids the combinatorial explosion problem of the Breadth-First search by expanding only the q most promising nodes at each level, where a heuristic is used to predict which nodes are likely to be closest to the goal, and to pick the best q successors.

One potential problem of the BS is that the q nodes chosen tend to quickly lack diversity. The major advantage of the BS, however, is that it increases both the space and time efficiency dramatically.

The literature includes many applications in which the BS pruning heuristic has been used. These applications include: handwriting recognition [57, 88, 94], Optical Character Recognition (OCR) [58], word recognition [85], speech recognition [34, 89], information retrieval [160] and error-correcting Viterbi parsing [5].

5.4.3 Branch and Bound (BB) search

In AI, whenever we encounter a search space, the latter can be searched in a variety of ways such as (as explained above) by invoking a breadth-first search, a depth-first search or even a best-first search scheme, where, in the latter, the various paths are ranked by using an appropriate heuristic function. If the cost function is associated solely with the final state, we can make no improvement to the algorithm without further heuristic guidance.

However, if the path has a cost [59], as in the case of approximate string matching, we can do

[2]In the context of this Thesis, whenever we speak about Beam Search, we are referring to a *local* beam search, which is a combination of an AI-based local search and the traditional beam search [133] methodologies.

somewhat better. We assume that the cost always increases with the path length. So, consider the case when we have found a solution g with a cost, $C(g)$. We proceed to look for further solutions. Assume that we are about to examine a node, n, at the end of a path, p, from the root of the tree being searched. Now, n and any node below n will have a cost of at least $C(p)$, so if $C(p) > C(g)$, it is not worth pursuing this path further as all the nodes below will exceed the current cost. The algorithm resulting from this insight is called a *Branch and Bound* (BB) search[3].

Such a scheme may be invoked at the expense of more processing operations *per* node, and possibly additional storage for storing some local indices. But what we gain is that we can prune numerous unnecessary paths, and thus save enormous redundant computations without sacrificing any accuracy at all.

5.5 Assumptions

We list some assumptions that hold for this part of the Thesis:

- This part of the Thesis focuses on the problem of dictionary-based syntactic pattern recognition.
- The dictionary is stored as a trie, where the entire space for the trie has to be searched to get the nearest neighbors in the dictionary that are related to the noisy word.
- This Thesis is most concerned with the Levenshtein (or generalized Levenshtein) distance [86, 87] which we also call by its alternative name "edit distance", and it is denoted by $D(.,.)$.
- The costs are of the form $0/1$, or they can, in the case of GLD, be general costs.
- We devise methods for cases in which the number of errors is known or not known *a priori*.

[3]The process of not having to explore the descendants of a particular node in the search tree is referred to as "fathoming" in the literature. In this Thesis we use "bounding" and "fathoming" synonymously.

- Although transposition errors are of interest, we will not consider them in this Thesis because a transposition can be simulated with an insertion and a deletion, although the cost may be different.
- The length of the words stored in the dictionary is assumed to be larger than 4.
- We assume that we are editing from the noisy word Y to the word X, where X is a word in the dictionary.
- In this part we care about solving the best match problem, but all the methods can be easily extended to include the set of "best" candidates.

5.6 Conclusion

In this Chapter we reviewed all the important concepts and notations needed for this part of the Thesis. The Chapter started with the basic notations and the problem definitions and applications. It then gave the basic concepts in approximate string matching for the inexact string-to-string matching problem. It continued with a review of the different AI search techniques that will be used throughout this part of the Thesis. The Chapter ended with a listing of some assumptions that will be made for this part of the Thesis.

Chapter 6

State of the Art: Trie-based Syntactic PR

6.1 Introduction

Approximate string matching is fundamental to text processing, because we live in an error-prone world. The main objectives of this Chapter are to give an overview of the state-of-the-art in the field, obtained from a survey of the previous work on approximate string matching. The Chapter starts with a classification of the field, and then gives an overview of each item in this classification. Being the primary focus of the Thesis, we will concentrate mainly on dictionary-based approximate string matching techniques, and more specifically, when the trie is used to store the dictionary. The literature contains hundreds (if not thousands) of associated papers. Excellent recent surveys about the field can be found in [47, 105]. As the main objective of the Thesis is to optimize IR and PR using tries, in this part of the Thesis we concentrate on optimizing PR using tries, and more specifically on the problem of approximate string matching.

The Chapter organization is as follows: Section 6.2 presents an overall classification of the approximate string matching problem. Section 6.3 gives an overview of the different pieces of work done in string-to-string based approaches, which are actually the basic techniques used in

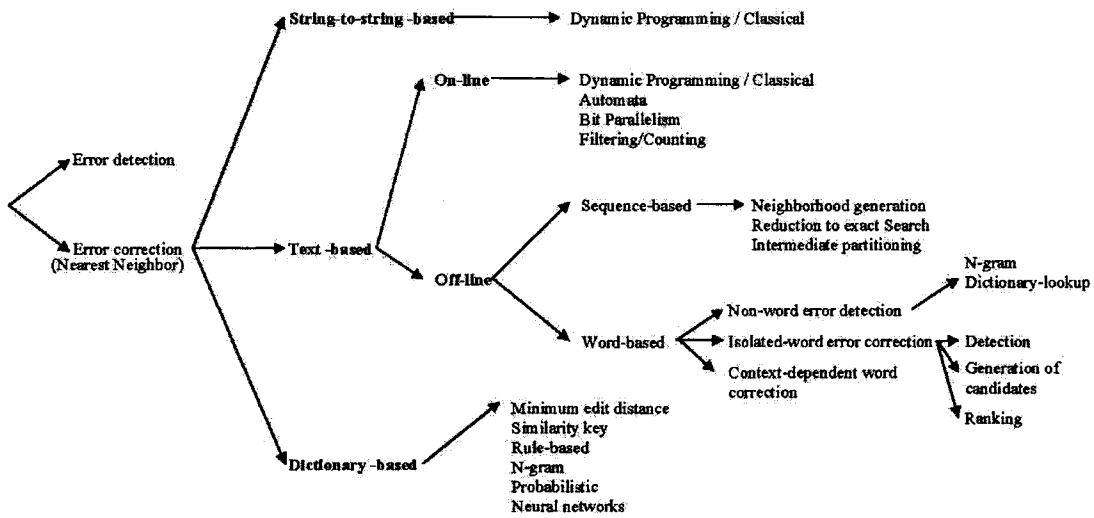


Figure 6.1: A broad classification of approximate string matching approaches.

all the other approaches. Section 6.4 presents the different results available for text-based approaches, and more specifically the various results for on-line and off-line text-based approaches. Off-line approaches, and more specifically word-based approaches, are more related to the work done in the Thesis. Section 6.5 describes the related work done in dictionary-based approaches. Section 6.6 describes more specifically the dictionary-based approaches for the instances when the trie is used as the data structure to store the dictionary.

6.2 Classifications

We can study the approximate string matching problem from two different perspectives: On the one hand, given a pattern, P, we want to calculate the distance between this pattern and a string, x, or retrieve all the strings from the text or dictionary that are near to P. On the other hand, we want to correct the pattern P, whether it is just a single string or a substring of the text. Whether we need to make corrections or to get the nearest neighbor string, we are broadly working in three main areas, as shown in Figure 6.1:

- **String-to-string-based approaches**: These are the approaches that solve the problem

of calculating the distances between any two strings, and which use and compute the metric between them.

- **Text-based approaches**: These are approaches that deal with the distance between a given noisy string and a text. The text is available, and the approaches are adapted to consider searching in different parts of the text at the same time. The techniques used here can also be classified into two approaches:
  - *On-line text-based* approaches, where the text is not available *a priori*, which is why it cannot be preprocessed to yield a better performance.
  - *Off-line text-based* approaches, where the text is available *a priori* and can be preprocessed for better performance by building an index on the text to facilitate the searching. In [104], Navarro divided the approaches using two classes of indices: one of them was able to solve the general problem using *sequence-retrieving* indices, and the other, called *word-retrieving* indices, was able to solve the restricted case using indices on a natural language that retrieves entire words matching the pattern.
- **Dictionary-based approaches**: These are approaches that deal with the distance between a given noisy string and the words in the dictionary so as to get the nearest neighbor one, or a group of best candidates. These approaches are actually the main objective of this part of the Thesis, and more specifically when the dictionary is stored as a trie.

When our aim is to process erroneous strings, the field can be classified into two main approaches: error detection and error correction. In [84], Kukich showed that efficient techniques have been devised for detecting strings that did not appear in a given word list, dictionary, or lexicon. Correcting a misspelled string is a much harder problem. In [84], the author gave two classifications for error correcting, which were either interactive spelling checkers or those which permit automatic correction. The latter task is much more demanding, and it is not clear how far existing correction techniques can fully automate correction. Kukich also gave a classification for automatic word correction as three increasingly broader problems:

- Nonword error detection: In this case computations are directed primarily toward exploring efficient pattern-matching and string comparison techniques for deciding whether an input

string appears in a predefined word list or dictionary. The two main techniques that have been explored for nonword error detection [84] are n-gram analysis and dictionary lookup.

N-grams are n-letter contiguous subsequences of words or strings. In general, n-gram error detection techniques work by examining each n-gram in an input string and looking it up in a pre-compiled table of n-gram statistics to ascertain either its existence or its frequency. Strings that are found to contain nonexistent or highly infrequent n-grams are identified as probable misspellings. N-gram techniques usually require either a dictionary or a large corpus of text in order to pre-compile the n-gram table.

Dictionary look-up techniques work by simply checking to see if an input string appears in a dictionary. If not, the string is flagged as a misspelled word.

- Isolated-word error correction: These are general and special-purpose correction techniques devised to achieve the goal of text recognition, which is to accurately reproduce input text, requiring that output errors must be both detected and corrected. The problem of isolated-word error correction entails three subproblems: detection of an error, generation of candidate corrections, and ranking of candidate corrections. Some techniques omit the ranking, and delegate the final selection to the user. Other techniques combine all three sub-processes into a single step by computing a similarity or probability measure between an input and each word, or some subset of words in the dictionary.

- Context-dependent word correction: These methods started in the early 1980's with the development of automatic natural language-processing models, and interest in them has recently been rekindled with the development of statistical language models.

Kukich [84] gave a more specific classification for Dictionary-based methods which dominate spelling correction techniques. A convenient way to organize the approaches is to group them into six main classes:

- Minimum edit distance: Here we compute the minimum (edit) distance between a misspelled string and a dictionary entry.

- Similarity key techniques: The notion behind similarity key techniques is to map every string into a key such that similarly spelled strings have identical or similar keys. When a

key is computed for a misspelled string it will provide a pointer to all similarly spelled words (candidates) in the lexicon. Similarity key techniques have a speed advantage because it is not necessary to directly compare the misspelled string to every word in the dictionary.

- Rule-based techniques: These are algorithms or heuristic programs that attempt to represent knowledge of common error spelling patterns in the form of rules for transforming misspellings into valid words. The candidate generation process consists of applying all applicable rules to a misspelled string and of retaining every valid dictionary word that results. Ranking is frequently done by assigning a numerical score to each candidate based on a predefined estimate of the probability of having made the particular error that the invoked rule corrected.

- N-gram-based techniques: These have been used in spelling correctors as access keys into a dictionary for locating candidate corrections, and as lexical features for computing similarity measures. They have also been used to represent words and misspellings as vectors of lexical features to which traditional and novel vector distance measures can be applied to locate and rank candidate corrections. Some n-gram-based correction techniques execute the processes of error detection, candidate retrieval, and similarity ranking in three separate steps. Other techniques collapse the three steps into a single operator.

- Probabilistic techniques: N-gram techniques led naturally to probabilistic techniques in both the text recognition and spelling correction paradigms. Two types of probabilities have been exploited, those which involve transition probabilities and confusion probabilities. Transition probabilities represent probabilities that a given letter will be followed by another given letter, and confusion probabilities are estimates of how often a given letter is mistaken or substituted for another given letter. These are typically source dependent. Because different OCR devices use different techniques and features to recognize characters, in such applications, each device will typically have its own unique confusion distribution. Combining probabilistic information with dictionary techniques results in significantly better error correction ability. A more efficient and widely used method for combining transition and confusion probabilities is the dynamic-programming technique called the Viterbi algorithm [61]. A disadvantage of this method is that the highest probability string is not always a valid word. Oommen *et. al* [122] presented a technique for probabilistically correcting noisy words with a very high accuracy which attained the

information theoretic bound. The problem with this technique is the cubic time performance, as the calculation is done on a string-to-string basis, using an $O(N^3)$ computational scheme.

- Neural nets: Neural nets are likely candidates for spelling correctors because of their inherent ability to do associative recall based on incomplete or noisy input. Because they can be trained using actual spelling errors, they have the potential to adapt to the specific error patterns of their user community, thus maximizing their correction accuracy for that population.

Kukich [84] also argued that most application-specific design considerations are related to three main issues (where the general description of the issues is per [84]):

- Lexicon issues: These include such things as lexicon size and coverage, rates of entry of new terms into the lexicon, and whether morphological processing, such as affix handling, is required.

- Computer-human interface issues: These include considerations as to whether a real-time response is needed, whether the computer can solicit feedback from the user, and how much accuracy is required on the first guess, etc. These issues are especially important where trade-offs must be made to balance the need for accuracy against the need for quick response time.

- Spelling error pattern issues: These include issues such as what constitutes the most common errors, how many errors tend to occur within a word, and whether errors tend to change word length. Other issues consider if misspellings are typographical (e.g., the → teh, or spell → speel) where the writer or typist knows the correct spelling but simply makes a motor coordination slip, cognitively (e.g., receive → recieve) where we encounter an error presumed to be a misconception or lack of knowledge on the part of the writer or typist, or phonetically based (e.g., obyss → obiss) where we encounter a special class of cognitive errors in which the writer substitutes a phonetically correct but orthographically incorrect sequence of letters for the intended word. In general, these errors can be characterized by rules or by probabilistic tendencies. Spelling error pattern issues have had, perhaps, the greatest impact on the design of correction techniques. Most studies

of spelling error patterns were done for the purpose of designing correction techniques for nonword errors (i.e., those which have no exact match in the dictionary), and so most findings relate only to nonword as opposed to real-word errors (i.e., those that result in another valid word).

In the subsequent sections we will present a (brief) literature review about each of these approaches, but we will concentrate more on the dictionary-based approaches, and more specifically when the dictionary is stored as a trie.

6.3 String-to-String-Based Approaches

Damerau [40, 115, 116, 134] was probably the first researcher to observe that most of the errors found in strings were either a single substitution, insertion, deletion or a reversal (transposition) error. Thus the question of computing the dissimilarities between strings was reduced to that of comparing them using these edit transformations. In much of the existing literature, the transposition operation has been modelled as a sequence of a single insertion and deletion.

The first breakthrough in comparing strings using the three (the SID) edit transformations was the concept of the Levenshtein metric introduced in coding theory [86], and its computation as shown in Chapter 5. Many researchers, among whom are Wagner and Fisher [155], generalized it by using edit distances which are symbol dependent. The latter distance goes by many names, but we shall call it the Generalized Levenshtein Distance (GLD). One of the advantages of the GLD is that it is a metric if the inter-symbol distances obey triangular inequality constraints [40, 114, 134, 147]. The GLD has also been studied for parameterized [42, 124] inter-symbol distances. Wagner and Fisher [155] and others [134, 146] also proposed an efficient algorithm for computing this distance by utilizing the concepts of dynamic programming (as shown in Chapter 5). This algorithm is optimal for the infinite alphabet case. Various amazingly similar versions of the algorithm are available in the literature, a review of which can be found in [40, 134, 147]. Masek and Paterson [95] improved the algorithm for the finite alphabet case, and Ukkonen [151] designed solutions for cases involving other inter-*substring* edit operations. Related to these algorithms are the ones used to compute the Longest Common Subsequences (LCS) of two strings (Hirschberg [71], Hunt and Szymanski [72] and others [40, 134, 147]).

String correction using GLD-related criteria has been done for noisy strings [40, 128, 134, 147], substrings [78, 134, 147], and subsequences [116], and also for strings in which the dictionaries are treated as grammars [134, 147, 156]. Besides these, various probabilistic methods have also been studied in the literature [39, 80, 128]. Indeed, more recently, probabilistic models which *attain the information theoretic bound* have also been proposed [121, 122].

All the algorithms proposed earlier for estimating X^+, require the separate evaluation of the edit distance between Y and every element of $X \in H$. However, they do not generally utilize the information it has obtained in the process of evaluating any one $D(X_i, Y)$, to compute any other $D(X_j, Y)$. Suppose X_i and X_j have the same prefix $X^{(P)} = a_1 a_2 ... a_p$. Then, previous algorithms would compute the distance $D(a_1 a_2 ... a_p, Y)$ for both of X_i and X_j, and would thus unnecessarily repeat the same comparisons and minimizations for the substring $a_1 a_2 ... a_p$ and *all its prefixes.* Thus, the previous algorithms usually have many redundant computations, but they can be used for both text-based and dictionary-based approaches.

6.4 Text-Based Approaches

In text-based approaches [99] a short pattern string P, of length m and a large text T, of length n, are given, where both the pattern and the text are sequences of characters from the alphabet A with $m \ll n$. The problem consists of finding one (or more generally all) of the exact occurrences of a pattern P in a text T. The approximate string searching problem is a generalization of the exact string searching problem, which involves finding substrings S of a text string close to a given pattern string such that $D(P, S) \leq K$. The solutions to this problem differ if the algorithm has to be on-line (that is, the text is not known in advance) or off-line (the text can be preprocessed).

6.4.1 On-line text-based approaches

An online approximate string searching algorithm [99] consists of two phases: the *preprocessing* phase in P and the *searching* phase of P in T. The *preprocessing* phase involves gathering information about the pattern which can be used for fast implementation of primitive operations

in the searching phase or of constructing a finite automaton that recognizes all strings at a distance at most K from the pattern. The *searching* phase consists of scanning the text or the construction of an array in order to find all approximate occurrences of the pattern in the text. In [105], Navarro presented a survey that focuses on online searching when the text could not be preprocessed to build an index on it. His goal in this survey was to explain the basic tools of approximate string matching. The work for on-line text-based approximate string matching under the edit distance can be divided into four classifications based on the different approaches used to solve the problem: dynamic programming/classical, automata, bit parallelism, and filtering/counting algorithms. For more references to each approach we refer the interested reader to [99].

6.4.2 Off-line text-based approaches

If the text is large and has to be searched frequently, even the fastest on-line algorithms are not practical, and preprocessing of the text becomes necessary. Starting from 1992, indexing text for approximate string matching was considered one of the main open problems in this area. In [104], Navarro divided the approaches into two classes of indices; one of them was able to solve the general problem *sequence-retrieving* indices, and the other was able to solve the restricted case of an index on a natural language that retrieves whole words that match the pattern, called *word-retrieving* indices. These word-retrieving indices are identical to the indices used for dictionary-based approaches except that for the text-based ones we have to store the list of occurrences of the words in the text under consideration. An inverted index can be built in $O(n)$ time by keeping the vocabulary in a trie data structure and storing the list of occurrences at the leaves.

For sequence-retrieving indices, there are three approaches [108]: neighborhood generation, reduction to exact searching, and intermediate partitioning. In [47], Cole *et. al* presented a survey on index-based searching, i.e., the process of building a persistent data structure (an index) on the text to later speed up the search. In the *neighborhood generation* approach, the user generates and searches for, using an index, all the strings that are at distance k or less from the pattern (their neighborhood). *Partitioning* into exact searching, selects patterns that must appear unaltered in any approximate matching occurrence, uses the index to search

for those substrings, and checks the text areas surrounding them. *Intermediate partitioning* extracts substrings from the pattern that is searched for, allowing fewer errors than by using neighborhood generation. Assuming that the errors occur in the pattern, or in the text, leads to radically different approaches.

In [104], Navarro gave a good background about the work done in word-retrieving indices. He mentioned that the first proposal for a word-retrieving index, called *Glimpse*, was due to Manber and Wu [93]. Here, the text was logically divided into "blocks", and the index stored all the different words of the text (the vocabulary). For each word, the list of the blocks where the word appeared was maintained. In order to search for a word allowing errors, an on-line approximate search algorithm (such as be Agrep [162]) was run over the vocabulary. Then, for every block where a matching word was present, a new sequential search was performed over that block, again using Agrep. The search in the vocabulary was inexpensive because it was small compared to the text size. In [14], Araujo *st. al.* took the approach of *full inversion* in an index called *Igrep* [14]. For each word, the list of all its occurrences in the text was maintained and the text was never accessed. The search on the vocabulary was done using the bit-parallel algorithm in [25], but the second phase of the search changed as soon as the matching words in the vocabulary were identified, at which juncture all their lists were merged. In [137], Shang and Merettal used a trie to arrange all the words of a dictionary and reported improvements over Igrep (the best known algorithm) for $K = 1$. We will present a more detailed literature review about dictionary-based approaches in Section 6.5.

Baeza-Yates and Navarro [26] proposed two speed-up techniques for on-line approximate searching in large indexed textual databases when the search is done on the vocabulary of the text. The *first* proposal required approximately 10% extra space and exploited the fact that consecutive strings in a stored dictionary tend to share a prefix. The *second* proposal required even more additional space. The proposal here was to organize the vocabulary in such a way as to avoid the complete on-line traversal. The organization, in turn, was based on the fact that they sought only those elements of H which are at an edit distance of at most K units from the given query string. Clearly, the efficiency of this method depends on the number of errors allowed. Although the method showed improvement over the scheme of Shang and Merettal (when using the trie), it only works if the edit distance follows the triangular inequality rule, and if K is known *a priori*.

The author of [104] suggested a limited depth-first search on a suffix tree. Since every substring started at the root of the suffix tree, it was sufficient to explore every path starting at the root, descending by every branch to the place where it could be seen that that branch did not represent the beginning of an occurrence of the pattern. Navarro [104] implemented the same algorithm replacing the dynamic programming with a bit-parallel algorithm, and the results demonstrated that it was up to 150 times faster even though it could not work in more complex setups. Another hybrid indexing approach was presented in [106], which was based on a suffix array being combined with a partitioning of the pattern. This was essentially a hybrid between the extremes of the suffix tree traversal and a filtering strategy used for locating exact fragments of the pattern.

Four important different data structures are used in the literature [47] for processing text. They all roughly serve the same purpose but possess different space/time tradeoffs, varying from being more powerful with respect to computation to less advantageous with respect to space. *Suffix trees* permit searching for any substring of the text, *Suffix arrays* permit the same operations but are slightly slower, *q-Grams* permit searching for any text substring not longer than q, and finally, *q-samples* permit the same operations but only for some text substrings.

6.5 Dictionary-Based Approaches

The dictionary-based approaches are quite related to the word-retrieving approaches except that the dictionary-based approaches potentially include a much larger number of words than just the vocabulary in the text, and so the index will be larger and include only the search phase in the vocabulary. Typically, the vocabulary is very small compared to the text, as per Heaps law which states that the vocabulary for a text of n words, grows as $O(n^{\beta})$, where $0 < \beta < 1$, and β is between 0.4 and 0.6. Dictionary-based approaches can also be related to correcting misspelled words in a text by retrieving the nearest words to the misspelled word from the dictionary.

In general, minimum edit distance algorithms require m string-based comparisons between the misspelled string and the dictionary, where m is the number of dictionary entries [84]. However some shortcuts have been devised to minimize the search time required. Proceeding with the assumption that deletions are the most common type of error, Mor and Fraenkel [102]

achieved an efficient response time for a full-text information retrieval application by storing every word in the dictionary $|x| + 1$ times, where $|x|$ was the length of the word, each time omitting one letter, after which, they invoked a hash function to look up the misspellings.

To correct an error in a string, we can operate in a reverse manner. In reverse techniques, a candidate set is produced by first generating every possible single-error permutation of the misspelled string and then the dictionary is checked to see if any of these permutations make up valid words [84]. This means that given a misspelled string of length n and an alphabet of size 26, the number of strings that must be checked against the dictionary is $26(n + 1)$ for the insertions, plus n for the possible deletions, plus $25n$ for possible substitutions, and plus $n - 1$ for possible transpositions. This is a total of $53n + 25$ strings, assuming there is only one error in the misspelled string!

Most of the time-efficient methods currently available, require that the maximum number of errors is known *a priori*, and these schemes are optimized for the cases when the edit distance costs are of a 0/1 form. In [55], Du and Chang proposed an approach to design a very fast algorithm for approximate string matching which divided the dictionary into partitions according to the lengths of the words. They limited their discussion to cases in which the error distance between the given string and its nearest neighbors in the dictionary was "small".

Bunke [41] proposed the construction of a finite state automaton for computing the edit distance for *every* string in the dictionary. These automata are combined into one "global" automaton that represents the dictionary, which in turn, is used to calculate the nearest neighbor for the noisy string when compared against the active dictionary. This algorithm requires time which is linear in the length of the noisy string. However, the number of states of the automaton grows exponentially. Unfortunately, the algorithm needs excessive space, rendering it impractical. For example, for the English alphabet with 26 characters, the minimum number of possible states needed is 29,619 for processing a single string in the dictionary!

Oflazer [113] also considered another method that could easily deal with very large lexicons. The set of all dictionary words is treated as a regular language over the alphabet of letters. By providing a deterministic finite state automaton recognizing this language, Oflazer suggested that a variant of the Wagner-Fisher algorithm can be designed to control the traversal through the automaton in such a way that only those prefixes which could potentially lead to a correct

candidate X^+ (where $GLD(X^+, Y) < K$) would be generated. To achieve this, he used the notion of a **cutoff** edit distance, which measures the minimum edit distance between an initial substring of the incorrect input string, and the (possibly partial) candidate correct string. The cutoff-edit distance required the *a priori* knowledge of the maximum number of errors found in Y and also required that the inter symbol distances are of 0/1 sort, or a maximum error value when general distances are used.

The literature also reports some methods that have proposed a filtering step so as to decrease the number of words in the dictionary that need to be involved in the calculations. One such method is "the similarity" keys method [135] that offers a way to select a list of possible correct candidates in the first step. This correction procedure, proposed in [135], can be argued to be a variant of Oflazer's approach [113]. Given the input word Y and a bound k, they first compute a finite state automaton, A, that accepts exactly all words $X \in H$, where the Levenshtein-distance between X and Y does not exceed k. A is called the Levenshtein-automaton for Y. Both automata are traversed in parallel, to obtain the intersection of the languages of the two automata as the list of "correction candidates". A speed up technique for [135] was later proposed in [98], which had its applications in the area of natural language processing, where the background dictionary was very large, and where the fast selection of an approximate set of correction candidates was important. This scheme introduced a new concept of "the universal Levenshtein automaton" of fixed degree k. In this scheme the dictionary was modelled as a deterministic finite state automaton, and the basic method was essentially a parallel backtracking traversal of the universal Levenshtein automaton and the dictionary automaton. In this case, the time required depends merely on the permitted number of edit operations involved in the distance computations.

A host of optimizing strategies has also been reported in the literature for methods which model the language probabilistically using N-grams, and for Viterbi-type algorithms [6, 36, 61, 153]. Clearly, these methods do not explicitly use a "finite-dictionary" (trie or any other) model, and so we believe that it is futile to survey them here. The same is also true for methods that are applicable for error correct parsing [5, 127] and grammatical inference [100], where the dictionary is represented by the language generated by a grammar whose production probabilities are learnt in the "training" phase of the algorithm. These methods are not directly related to the central thrust of our finite-dictionary results. We therefore refer the reader to the latter references for

a survey on the state-of-the-art in these avenues.

6.6 Trie-Based Approaches

As mentioned earlier, the *trie* is a data structure that can be used to store a dictionary when the dictionary is represented as a set of words [62]. Tries offer text searches with costs that are independent of the size of the document being searched. The data is represented not in the nodes but in the path from the root to the leaf. Thus, all strings sharing a prefix will be represented by paths branching from a common initial path. Figure 6.2 shows an example of a trie for a simple dictionary of words {for, form, fort, forget, format, formula, fortran, forward}. The figure illustrates the main advantage of the trie as it maintains only the minimal prefix set of characters that is necessary to distinguish all the elements of H. The trie has the following features:

1. The nodes of the trie correspond to the set of all the prefixes of H.
2. If X is a node in the trie, then X_p, the left derivative of order one, will be the parent node of X, and X_g, the left derivative of order two, will be the grandparent of X.
3. The root of the trie will be the node corresponding to μ, the null string.
4. The leaves of the trie will all be words in H, although the converse is not true.

A suffix trie is a trie in which all the suffixes of the text string have been inserted. A suffix tree achieves $O(n)$ worst-case space and construction time by compressing unary paths of the suffix trie. A Directed Acyclic Word-Graph (DAWG) [90] is the minimal automaton that recognizes all the substrings of the text and is obtained by compressing the suffix tree via the identification of all the final states.

Tries have been explored as an alternative to dynamic-programming techniques for improving the search time in minimum edit distance algorithms. Muth and Tharp [62] devised a heuristic search technique in which a dictionary of correct spelling was stored in a binary trie. The search examined a small subset of the database, which were selected branches of the tree. The

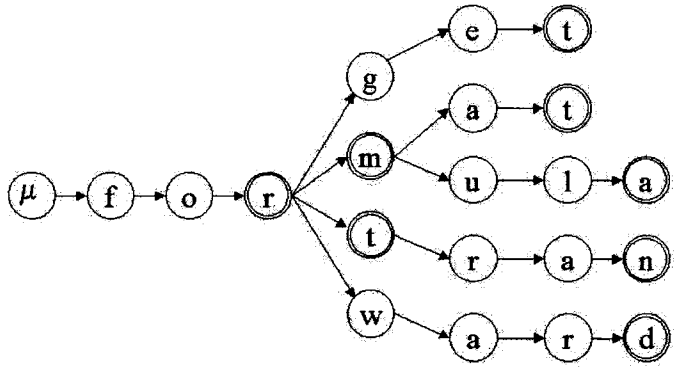


Figure 6.2: An example of a dictionary stored as a trie with the words {for, form, fort, fortran, formula, format, forward, forget}.

assumption they made was that the first part of the word was correct (i.e., there were some correct characters before the error). The search was done character by character and with each successful match the search continued down through the tree, until a terminal node was reached or a match was not possible. The assumption was then made that the current character was erroneous, which might not be a correct assumption. The process continued by trying to infer the type of error. If the exact nature of error could be inferred, the subsequent search down the trie could proceed in the proper manner.

Dunlavey [56] described an algorithm called SPROOF, which stored a dictionary as a trie. The search proceeded by visiting the nodes of the trie in increasing edit distance until the candidate corrections at the leaves were reached. A similar technique was used by Boivie [35] in his *da* dictionary assistance program, which in turn became the underlying algorithm for Taylor's [150] *grope* spelling corrector. The trie has also been used to achieve the approximate string matching of DNA fragments [92].

With regard to traversal, the trie can be considered as a graph, which can be searched using any of the possible search strategies applicable to AI problems. The literature includes two possible strategies that have been applied to tries, namely the Breadth First Search strategy [76, 118] and the Depth First Search strategy [137], also currently recognized as an "industrial benchmark" [131]. The main concern of this part of the Thesis is to optimize the traversal of the trie by using different AI techniques. The choice of the AI technique depends on the error

model and main objectives for the applications in which the trie is used as the data structure to store the dictionary.

6.6.1 Breadth-First Trie-based Scheme

Rather than work with inter-string edit distances, let us now consider how we can achieve the simultaneous computation of the distances for all the words in H. This is based on the principles introduced by Kashyap *et.al.* in [76], who developed a recursive procedure to compute $D(X,Y)$ for all the relevant prefixes in the entire dictionary. This involved only a fixed finite number of the prefixes of X and the left derivative of Y. They introduced a new distance measure, $D_1(X,Y)$, (an intermediate computational tool called a *pseudodistance*) between X and Y. The measure $D_1(X,Y)$ has the desirable properties that it can be computed "recursively" and that the final distance $D(X,Y)$ can be obtained from *it* using only a single additional symbol comparison. $D_1(X,Y)$ is the minimum sum of individual edit distances associated with the edit operations needed to transform a received Y to X *given that the last symbol of* X *was not inserted during editing.*

The relationship between $D_1(X,Y)$ and $D(X,Y)$ is formalized below:

Theorem 6.1. *Let* $X = x_1x_2....x_N$ *and* $Y = y_1y_2....y_M$. *Let* X_p *be the left derivation of* X *of order one. Then:*

$$D(X,Y) = \min[D_1(X,Y), \{D_1(X_p,Y) + d(x_N,\lambda)\}]. \tag{6.1}$$

Proof: The result is proved in [76]. □

Let $Y^{(K)}$ be the prefix of Y of length K. We now report the recursive properties of the pseudodistances between an arbitrary string $X \in A^*$ and $Y^{(K)}$.

Theorem 6.2. *Let* $X = x_1x_2....x_N$ *and* $Y = y_1y_2....y_M$. *Then, if* X_p *and* X_g *are the left derivatives of* X *of orders one and two respectively, and if* b *and* c *are elements of* A, *the following pseudodistance equalities are true:*

1. *When* $|X| = 0$*:*

$$D_1(\mu, Y^{(K+1)}) = D_1(\mu, Y^{(K)}) + d(\lambda, y_{K+1}). \tag{6.2}$$

2. *When* $X = b$*, since* b *is not an inserted symbol:*

$$\begin{aligned} D_1(b, Y^{(K+1)}) = \min \; [\; & \{D_1(b, Y^{(K)}) + d(\lambda, y_{K+1})\}, \\ & \{D_1(\mu, Y^{(K)}) + d(b, y_{K+1})\}]. \end{aligned} \tag{6.3}$$

3. *When* $X = X_1bc$ *(where* $|X| \geq 2$*) with* $|X_1| \geq 0$*, since* c *is not inserted:*

$$\begin{aligned} D_1(X_1bc, Y^{(K+1)}) = \min \; [\; & \{D_1(X_1bc, Y^{(K)}) + d(\lambda, y_{K+1})\}, \\ & \{D_1(X_1b, Y^{(K)}) + d(c, y_{K+1})\}, \\ & \{D_1(X_1, Y^{(K)}) + d(b, \lambda) + d(c, Y^{(K+1)})\}]. \end{aligned} \tag{6.4}$$

Proof: The proof is quite involved and given in [76]. □

Observe that in the above expressions the number of terms included to obtain the distance between X_1bc and $Y^{(K)}$ is exactly *three*, and is more efficient than the method applicable for Regular Languages [156]. Although these computations involving the pseudodistance restricts the noisy channel to avoid the deletion of two consecutive symbols of X^* (which was justified in [76], where it was shown that in the case of independent noise, in the worst case, 99.84% *of the erroneous strings* will *not* contain two consecutive deletions), in practice, the accuracy is not affected by this assumption. It is, indeed, a much less restrictive assumption than what is traditionally used in the field, namely that of limiting the *number* of errors in Y.

The interesting feature of utilizing the $D_1(X, Y)$ measure is that it leads to a Breath First traversal on the trie as opposed to the Depth First traversal given in [137]. This approach gave more optimization to the trie traversal when used for approximate string matching and when the error model is general and the maximum number of errors is not known *a priori.*

6.6.2 Depth-First Trie-based Scheme

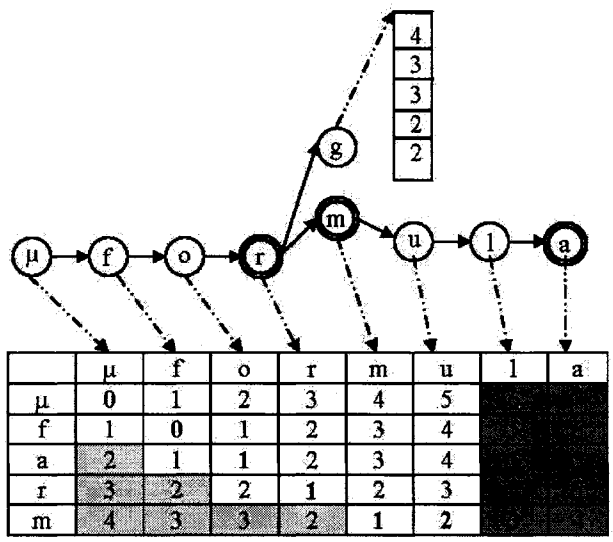
Shang *et al.* [137, 131] used the trie data structure for both exact and approximate string searching. Their trie-based method had a cost which is independent of the document size. They proposed a k-approximate match algorithm using Levenshtein distances on a text represented as a trie, which performed a depth-first search on the trie using the matrix involved in the dynamic programming equations used in [155]. Besides the trie, they also had to maintain a matrix to represent the Dynamic Programming (DP) matrix required to store the results of the calculations. The trie representation compressed the common prefixes into overlapping paths, and the corresponding column (in the DP matrix) had to be evaluated only *once*. The insight is that the trie representation of the text drastically shortens the DP. If a $m \times n$ DP matrix was used to match a given Pattern, Y_m, with the text, X_n, there would have to be a new table for for each suffix in X (or prefix if X is a word in the dictionary). Sharing of common prefixes in a trie structure saves not only index space but also search time.

Furthermore, the Ukkonen cutoff can be used to terminate unsuccessful searches very early, as soon as the difference exceeds k. Change and Lawler [43] showed that Ukkonen's algorithm evaluated $O(k)$ columns, which implies that searching a trie will be done only down to depth $O(k)$. If the fanout of a trie is $\sum$, the trie method needs to evaluate only $O(k|\sum|^k)$ DP table entries. Ukkonen [152] proposed an algorithm to reduce the table evaluations. His algorithm worked as follows:

- Let G_j be the maximum row number i such that $D(X_i, Y_j) \leq k$ for the given j ($G_j = 0$ if there was no such i).
- Given G_{j-1}, compute $D(X_i, Y_j)$ up to $i \leq G_{j-1} + 1$.
- Set G_j to the largest i ($0 \leq i \leq G_{j-1} + 1$) such that $D(X_i, Y_j) \leq k$.

Chang and Lawler [43] proved that this algorithm evaluates $O(k^2)$ expected entries.

As a result of the Ukkonen cutoff, the authors of [137] presented an interesting observation, where, if all the entries of a column are greater than k, no word with the same prefix can have a distance $\leq k$. They thus concluded that we can stop searching down the subtrie. This observation reveals that it is not necessary to evaluate every suffix in the trie, and thus many



	μ	f	o	r	m	u	l	a
μ	0	1	2	3	4	5		
f	1	0	1	2	3	4		
a	2	1	1	2	3	4		
r	3	2	2	1	2	3		
m	4	3	3	2	1	2		

Figure 6.3: An example of the calculation done on the same trie shown in Figure 6.2, when the calculation is done for the path corresponding to the word "formula" where the noisy word is "farm", and $K = 1$. One additional branch is shown to demonstrate that only *one* column is to be calculated per node. The shaded area represents the entries that will not be calculated if the Ukkonen cutoff and the observations in [137] are used.

subtries will be bypassed. In the extreme case, the one which involves the exact search, all but one of the subtries are trimmed.

Figure 6.3 shows an example of the calculation done on the same trie shown in Figure 6.2, when the calculation is done for the path corresponding to the word "format" where the noisy word is "farm", and $K = 1$. One additional branch is shown to demonstrate that only *one* column is to be calculated per node. The shaded area represents the entries that will not be calculated if the Ukkonen cutoff and the observations in [137] are used.

The authors of [137] presented their method from the perspective of full-text retrieval, for which both the index and the text had to be stored. In applications such as spelling checkers, the text is a dictionary, and need not be stored separately from the index. They compared their work experimentally with *agrep* [162], and showed that tries outperform *agrep* significantly for small k, the number of mismatches. Since *agrep* has a complexity which is linear in k, and the search complexity for tries is exponential in k, *agrep* is expected to become better than tries for large k.

6.7 Conclusion

In this Chapter we presented the state-of-the-art for the work done in approximate string matching. We first gave a classification of the different approaches done in the field, and then presented an overview of each of the items. The overview about dictionary-based approaches was more detailed, and more specifically, highlighted the techniques in which the trie was used as the data structure to store the dictionary.

It is clear that with such a variety of different approaches used to solve the same problem, it is difficult to select an appropriate algorithm for every specific problem encountered in approximate string searching. Although the theoretical analysis given in the literature is useful, it is important that the theory is accompanied by exclusive experimental comparisons.

In the following Chapters we will show how to apply different AI techniques (those presented in Chapter 5) to optimize the trie-based traversal when used for approximate string matching and when the Levenshtein distances are used. The choice of the technique depends on the error model and distance functions that will be used.

Chapter 7

Breadth-First-Trie Based Scheme

7.1 Introduction

In this Chapter[1], we show how we can optimize non-sequential PR computations by incorporating a Breadth-First Search (BFS) scheme (described in Section 6.6) on the underlying graph structure. Although the new scheme marginally restricts the types of errors found in Y (like most researchers do), in practice, there is no loss of PR accuracy. The Chapter shows how these searches can be effectively implemented using a new data structure called the Linked List of Prefixes (LLP), applicable for arbitrary inter-symbol distances. The latter permits *level-by-level* traversal of the trie (as opposed to traversal along the "branches"). The BFS-LLP-based algorithm for the syntactic PR of strings has been rigorously tested on three benchmarks dictionaries. The algorithm was tested by recognizing noisy strings generated using the model discussed in [121] on each of these dictionaries. The main contribution is that our new approach can be used for Generalized Levenshtein distances and not for just 0/1 costs. Additionally, our method is applicable when all possible correct candidates need to be known, and not just the best match, which is the case where the so called "cutoffs" cannot be used in the Depth-First Search (DFS) trie-based technique [131, 137]. We also show that further improvements can be gained by introducing the knowledge of the maximum number or percentage of errors in Y. All

[1]The work done in this Chapter was published in the *Proceedings of the Joint IARR International Workshops SSPR 2004 and SPR 2004*, Libon, Portugal, August 2004 [118], and a more detailed journal version is currently being reviewed [117].

our claims have been verified experimentally, as will be explained presently.

The Chapter is organized as follows: Section 10.2 describes the proposed method for obtaining the estimated string X^+ for a noisy string Y, and shows how the newly proposed data structure, the Linked List of Prefixes (LLP), can be used in the computations. It also describes the structure of the LLP and describes the algorithm involved in obtaining X^+ by using the BFS-LLP-based method. Section 7.3 presents the improvement that can be gained by introducing the knowledge of the maximum number or percentage of errors, K, in Y. Section 7.4 concentrates on the experiments done to test the new strategy and lists the results. Finally, Section 7.5 concludes the Chapter.

7.2 Procedure and Data Structure for Obtaining X^+

The concept of using FSMs in string matching [51] is well established for exact strings. Wagner [156] and others (as shown in Chapter 6) extended it for inexact-PR, but as explained earlier, the computation was more expensive. The methods used in [76] were less expensive because the FSM was a trie, and thus cycle-free. We shall now use the trie-based dynamic programming rules in conjunction with a new structure, the Linked List of Prefixes (LLP), to make the computations feasible, and to efficiently obtain X^+.

7.2.1 Tries and the Linked Lists of Prefixes (LLP)

The problem with a DFS strategy on the trie is that there does not appear to be an easy way to prune nodes in the trie without having *a priori* bound on the number of errors found in Y. The alternative strategy is a Breadth-First-Search (BFS), where the nodes in the trie are traversed level-by-level. But to achieve this, these nodes must be maintained in an optimal manner so that they, their parent and successors can be accessed quickly, We explain now how this can be done.

To calculate the best estimate X^+, what we need is to divide the dictionary into its sets of prefixes. Each set $H^{(p)}$ is the set of all the prefixes of H of length less than or equal to p, for

$1 \leq p \leq N_m$, where N_m is the length of longest word in H. The trie itself divides the prefixes and the dictionary in the way we want, as each sub-trie starting from the root to level p corresponds to all the prefixes in the set $H^{(p)}$. Additionally, the trie itself represents the FSM that we need, but the problem in the trie structure (with no additional information) is that while it can be implemented in different ways, it is not easily traversed to yield the desired calculations. So what we need is a data structure that facilitates the trie traversal, and gives us a unique data structure that can always be used to effectively compute the pseudodistances for the prefixes. Such an efficient data structure, called the Linked Lists of Prefixes (LLP), is now proposed. The LLP can be built on the top of the trie, by implementing the trie as linked lists and connecting all lists in the same level together with the LLP structure, or separately. In this Chapter we consider the case when the LLP is built separately. This is because of the fact that the trie has many representations and we don't want to restrict it only to the case when the trie has to be implemented as a linked list, which may take more time when the exact match is considered in the same application.

The LLP consists of a linked list of levels, *where each level is a level in the corresponding trie.* Each level, in turn, consists of a linked list of all prefixes that have the same length p. The levels are ordered in an increasing order of the length of the prefixes, exactly as in the case of the trie levels. Figure 7.1 shows the corresponding LLP for a trie example. Each node in the linked lists is actually a pointer to the node of the trie itself, and so we can access the parent nodes in the trie in a straightforward manner, as will be seen in the algorithm presently. The values of D_1 used during calculations is stored in the trie nodes. Thus, while the LLP is a data structure used to facilitate the traversing of the trie in the proposed string correction algorithm, it simultaneously allows us to calculate the distance between a noisy string Y and all the strings in the dictionary stored in the trie, using the BFS technique instead of the DFS technique described in [137].

The pseudo-code for constructing the LLP from a trie is given in figure 7.1.

Algorithm 7.1 Algorithm Build-LLP

Input: The dictionary in the form of trie T.

Output: The LLP L which stores n, an array that stores the number of nodes in each level, *lists*, the lists of prefixes where each level corresponds to a level in the trie T and each node in the list is a pointer to a node in the trie, and N, the number of levels.

Method:

```
1: level = 0
2: L → lists[level][1] = T → root
3: L → n[level] = 1
4: level = level + 1
5: notdone = true
6: while notdone do
7:    counter = 0          // count the number of nodes
8:    notdone = false
9:    for each node t_n in L → lists[level − 1] do
10:      {t_n is also a node in the trie}
11:      for each child ct_n for the node tn in the trie T do
12:         if ct_n ≠ NULL then
13:            L → lists[level][counter] = ct_n
14:            counter = counter + 1
15:            notdone = true
16:         end if
17:      end for
18:   end for
19:   L → n[level] = counter
20:   level = level + 1
21: end while
22: L → N = level
23: return L
24: End Algorithm Build-LLP
```

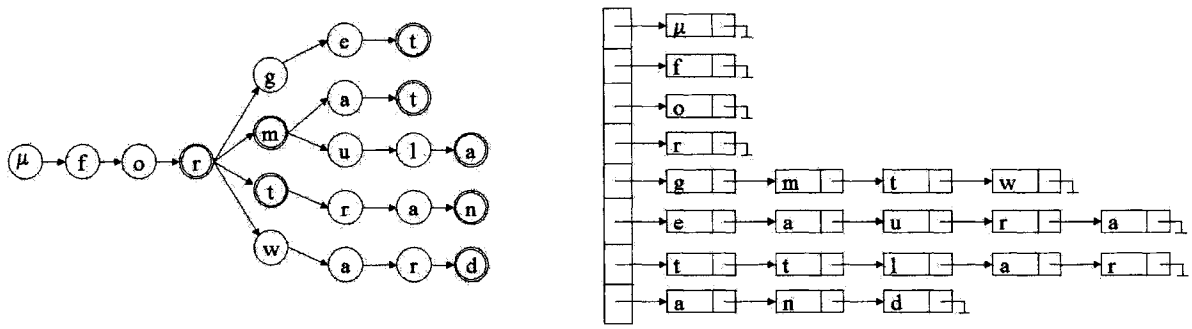


Figure 7.1: The corresponding LLP (right) for the trie (left) with words {for, form, fort, fortran, formula, format, forward, forget}.

7.2.2 The Procedure for Obtaining X^+

Let U be a prefix corresponding to some node in the trie T. It can be seen from Equation (6.4) that if the pseudodistance between a certain node U and $Y^{(K+1)}$ has to be computed, it can be done with merely the knowledge of the pseudodistances between each of U, U_p (the parent of U), Ug (the grandparent of U), and the string $Y^{(K)}$ respectively. This justifies the algorithm given below for calculating X^+, the estimate of the correct string for the noisy string Y. For any $U \in T$ let u_f be the last symbol and u_{pf} be the last symbol of U_p. In each node of the trie we store two values for the pseudodistances, namely the previous value of D_1 and its new value, $newD_1$, (both of which are initialized to ∞ for every node in the trie, or rather, in the conceptual LLP). Observe that we need both of these quantities because the computation of D_1 will require the "old" values of D_1 calculated in the previous iteration, which we refer to as $D_1(U, Y^{K-1})$. We then proceed in a BFS-manner along the trie from the root towards the leaves, and at each iteration fill in the distances for nodes at the same level and for nodes which are two levels deeper than at the previous iteration. Indeed, it is at this stage that the LLP becomes useful. Rather than work with *set based operations* as in [76], the LLP permits the access to nodes *level by level*, while details of the parents and grandparents are gleaned from the trie itself. The pseudo-code for the computation that formalizes this along the trie and LLP are given in Algorithm 7.2. In terms of notation, we let L be the LLP corresponding to the trie T, for which N_m is the length of the longest word in the dictionary, H.

Algorithm 7.2 Recognize-Using-LLP

Input: The dictionary stored in the form of a trie T and the corresponding LLP L, the noisy string Y, and the matrix of elementary symbol edit distances d. The list W, the list of nodes in T that corresponds to the words in the dictionary, H.

Output: The estimated correct string X^+.

Method:

1: $t_n = L \rightarrow lists[0][1]$ //the root of the trie
2: $t_n \rightarrow D_1 = 0$
3: $M = |Y|$
4: $N_m = L \rightarrow N$
5: **for** $K = 1$ to M **do**
6: **if** $2K < N_m$ **then**
7: $S = 2K$
8: **else**
9: $S = N_m$
10: **end if**
11: **for** $level = 0$ to S **do**
12: **for** $i = 1$ to $L \rightarrow n[level]$ **do**
13: $\{t_n$ corresponds the prefix we process$\}$
14: $t_n = L \rightarrow lists[level][i]$
15: $u_f = t_n \rightarrow character$
16: $u_{pf} = t_n \rightarrow parent \rightarrow character$
17: $D_1 = t_n \rightarrow D_1$
18: $D_{1p} = t_n \rightarrow parent \rightarrow D_1$
19: $D_{1g} = t_n \rightarrow parent \rightarrow parent \rightarrow D_1$
20: **if** D_1 or D_{1p} or $D_{1g} < \infty$ **then**
21: $q_1 = D_1 + d(y_k, \lambda)$
22: $q_2 = D_{1p} + d(y_k, u_f)$
23: $q_3 = D_{1g} + d(\lambda, u_{pf}) + d(y_k, u_f)$
24: $t_n \rightarrow new\_D_1 = Min[q_1, q_2, q_3]$
25: **end if**
26: **end for**
27: **end for**
28: Update all the values of D_1 to new_D_1 (this can be done by using a flag that we flip each iteration showing which of D_1 or new_D_1 to use in calculations and which to store in).
29: **end for**
30: $\{$the last step where we consider all the words in the dictionary$\}$
31: **for** every string U that corresponds to a node W **do**
32: $w \rightarrow D_1 = Min[D_1, D_{1p} + d(\lambda, u_f)]$
33: **end for**
34: **return** $X^+ =$ the string U in W with the minimum value of D_1
35: **End Algorithm Recognize-Using-LLP**

Figure 7.2 shows an example of the calculations done to obtain X^+ for the noisy word $Y = port$. The figure shows clearly how the LLP facilitates the calculations for pseudodistances through the trie using Equation (6.4), and yet permits the calculations to be achieved level-by-level. Each sub-figure shows the results of calculations for each value of K, the length of the prefix of the noisy string Y, processed so far. Observe that at each subsequent step, two more levels of the LLP are considered. This is distinct from the DFS scheme in which, for each symbol of Y, the distance for every node along the whole path of the trie must be updated. In each sub-figure, the values for the pseudodistances are calculated (which are initialized to ∞ if they are not computed) using the values in the previous sub-figure. Finally, the last sub-figure shows the terminating step in the calculations which uses the "dictionary-words" list and Equation (6.1). From it we see that the word "fort" has the smallest distance value, which allows us to assign $X^+ = fort$. The inter-symbol distances used in the calculations are obtained as per Equation (5.1) and using the confusion probabilities as in [76, 134]. The formal result concerning the correctness of the scheme follows.

Theorem 7.1. *The PR scheme for computing X^+ using the LLP is correct.*

Proof: Rather than complicate the proof with unnecessary symbolism (which would only add to the "pomp" and not to the content), we present the arguments in a straightforward manner.

Performing pseudodistance computations by traversing the LLP is the same as performing computations on the trie and its sets of prefixes. This follows from the fact that we will perform BFS-based computation in the trie nodes level by level, and the nodes in each level of the LLP exactly correspond to the same corresponding nodes in the corresponding level of the trie. Thus performing computations level by level in the LLP is exactly equivalent to performing computations, depth-by-depth, in the underlying trie. Consequently, when we are finished with the computation, the string in H which is chosen as X^+, will be exactly the same string that a *set-based* algorithm, operating on the trie, would yield. The result follows. □

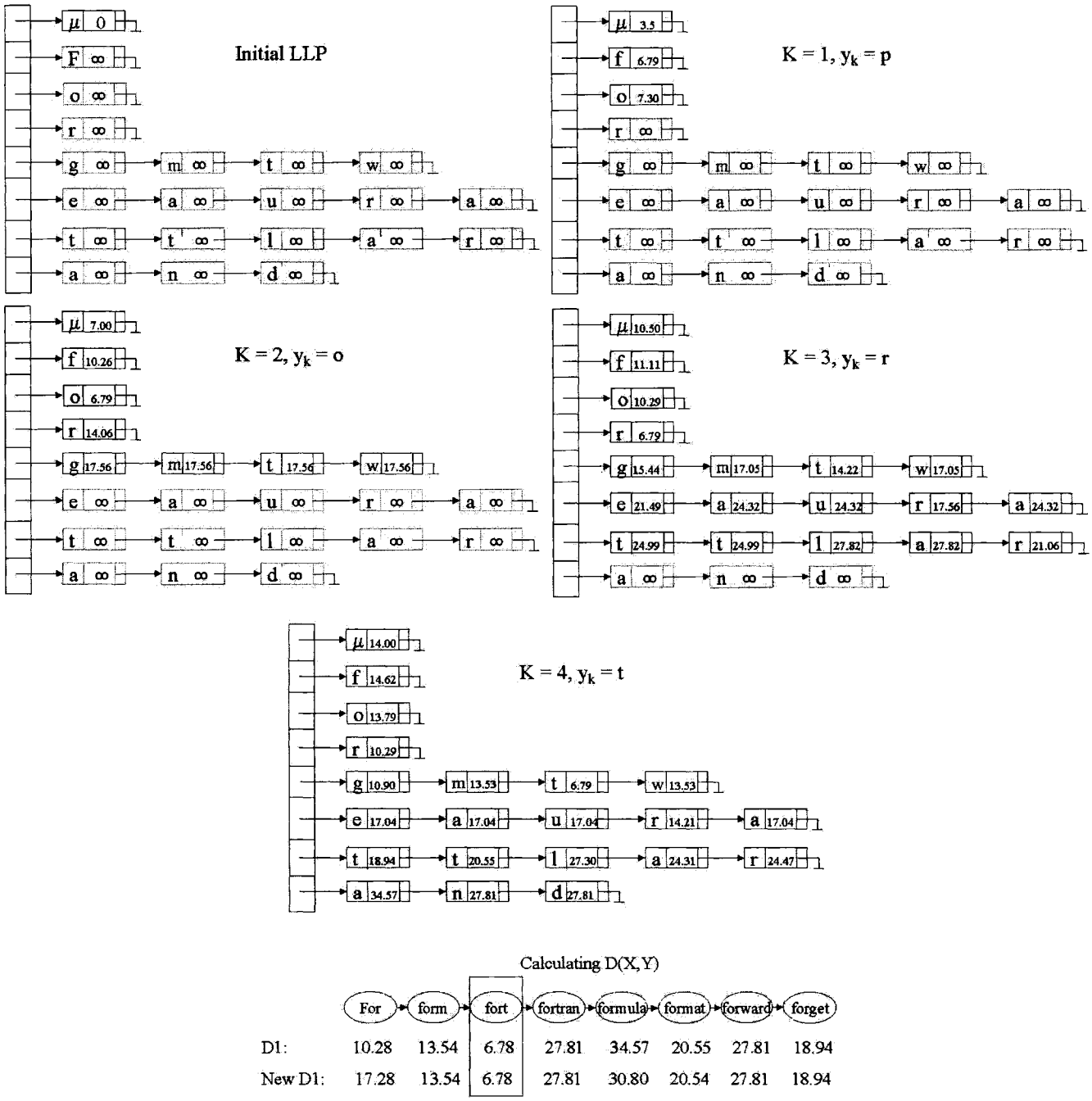


Figure 7.2: An example for the calculations done to obtain X^+ for the noisy word $Y = port$ which corresponds to the original word $U = fort$. The computation uses the LLP which corresponds to the trie with words {for, form, fort, fortran, formula, format, forward, forget}. The distances are obtained as described in the text.

7.3 Optimizing Computations When K is Known

All the techniques described previously in this Chapter, did not assume an *a priori* knowledge of the maximum number of errors permitted in Y, and considered this information to be unknown. While this is quite advantageous (as will be demonstrated presently), further advantages can be gleaned if knowledge of the maximum number of errors is known *a priori* which is assumed customarily in the well-accepted algorithms [131, 137]. In this section, we propose an optimization technique that can be applied to both the BFS and the DFS trie-based techniques, when the maximum number of errors K is known *a priori*. We have two cases here: either K is the maximum number of errors, or K is the maximum percentage of errors that can be in Y, the noisy string. The optimization is based on the fact that the lengths of the strings stored in the dictionary are also known *a priori*. We have the following cases:

- If K is the maximum number of errors in Y: In this case the maximum depth $\triangle$ we need to investigate in any search (DFS or BFS) is $M + K$.
- If K is the maximum percentage of error in Y: The maximum depth $\triangle$ we need to investigate in any search is $M + M * K$ which equals $M(1 + K)$.

This optimization is applied to the BFS-trie-based technique by introducing stopping calculations at a level whose index is greater than $\triangle$. For example, in Figure 7.2, if K equals unity, the last level to be considered is level number 5, as M equals 4. For the DFS-trie-based technique, the optimization is done along the paths of the search, and in every path, we don't go any deeper than the level $\triangle$.

7.4 Experimental Results

To investigate the power of our new method with respect to computation we conducted various experiments on three benchmark dictionaries. The results obtained were (in our opinion) remarkable with respect to the gain in the number of computations needed to get the best

Table 7.1: Statistics of the data sets used in the experiments.

	Eng	Dict	Webster
Size of dictionary	8KB	225KB	944KB
number of words in dictionary	964	24,539	90,141
min word length	4	4	4
max word length	15	22	21

estimate X^+. By computations we mean the number of addition and minimization operations needed. The BFS-trie-based scheme was compared with the DFS-trie-based work for approximate matching [137] when the maximum number of errors was not known *a priori*.

Three benchmark data sets were used in our experiments. Each data set was divided into two parts: a *dictionary* and the corresponding *noisy file*. The dictionary was composed of the words or sequences that had to be stored in the Trie. The noisy files consisted of the strings which were searched for in the corresponding dictionary. The three dictionaries we used, were as follows:

- *Eng*[2]: This dictionary consisted of 946 words obtained as a subset of the most common English words [54] augmented with words used in computer literature. The length of the words was greater than or equal to 4, and the average length of a word was approximately 8.3 characters.
- *Dict*[3]: This is a dictionary file used in the experiments done by Bentley and Sedgewick in [30].
- *Webster's Unabridged* Dictionary: This dictionary was used by Clement *et. al.* [1], [45] to study the performance of different trie implementations.

The statistics of these data sets are shown in Table 7.4 and the alphabets of all the dictionaries are the 26 lower case English alphabet.

[2]This file is available at `www.scs.carleton.ca/~oommen/papers/WordWldn.txt`.
[3]The actual dictionary can be downloaded from http://www.cs.princeton.edu/ rs/strings/dictwords.

Three sets of corresponding noisy files were created for five specific error characteristics, where the latter means the maximum percentage of errors per word. The five error percentages tested were for 30%, 40%, 50%, 60%, and 70%, referred to by the five sets SA, SB, SC, SD, and SE respectively.

Each of the five sets, SA, SB, SC, SD, and SE, were generated using the noise generator model described in [121]. The distribution used in the error generation is as follows. We assumed that the number of insertions was geometrically distributed with parameter $\beta = 0.7$. The conditional probability of inserting any character $a \in A$, given that an insertion occurred, was assigned the value 1/26, and the probability of deletion was set to be 1/20. The table of probabilities for substitution (typically called the confusion matrix) was based on the proximity of character keys on the standard QWERTY keyboard and is given in [121][4]. The statistics associated with each of the five sets are given in Tables 7.3, 7.4, and 7.5 for each of the three dictionaries used, and a subset of some of the original words and their noisy versions are given in Table 7.2. Some of the words in the dictionary are very similar even before garbling, words such as "official" and "officials"; "attention", "station" and "situation". These are words whose noisy versions can themselves easily be mis-recognized.

The two algorithms, the DFS-trie-based and our algorithm, BFS-trie-based, are tested with the five sets of noisy words for each of the three dictionaries[5]. We report the results obtained in terms of the number of computations (additions and minimizations) and the accuracy for the five sets in Tables 7.6, 7.7, and 7.8 for each of the three dictionaries respectively. The numbers are shown in millions. The results show the significant benefits of the BFS-based method with respect to the number of computations, which at the same time maintains the same accuracy. For example, for the *Webster* dictionary, for the SA set, the number of operations for DFS-trie-based is 1,123,109, and for the BFS-trie-based method is 880,897, which represents a saving of 21.56%. For the *Dict* dictionary, for the SA set, the number of operations for DFS-trie-based is 73,443, and for the BFS-trie-based method is 59,346, which represents a saving of 19.19%. The

[4]It can be downloaded from `www.scs.carleton.ca/~oommen/papers/QWERTY.doc`.

[5]A BFS technique was earlier shown in [118] to be much more superior to a method which computes X^+ using sequential comparison between every $X \in H$ and Y. The fact that it is also uniformly superior to a DFS-method is what we demonstrate here.

calculations were done on a Pentium V processor, 3.2 GHZ.

To test the optimization done when the maximum number or percentage of error is known, we ran the experiments again for the optimized case. The results are shown in Table 7.9 for just the *Dict* dictionary, as the results obtained for the other dictionaries represented approximately the same improvement (measured in percentages). For example, for the SA set, the number of operations for DFS-trie-based is 71,654, and for the BFS-trie-based method is 57,450, which represents a saving of 19.82%.

To investigate further, the advantage gained by techniques presented here, other sets of noisy files were generated for each dictionary where the number of errors equals 1, 2 and 3. The results are also shown for the *Dict* dictionary as the other results show the same behavior. The results are shown in Table 7.10 for the case when the number of errors is not known *a priori*, and in Table 7.11 for the optimized case where the maximum number of errors is known (assumed). In every single case the BFS method is noticeably superior. For example, when the number of errors equals unity, and if this is not known *a priori*, the number of operations for the DFS-trie-based is 72,115, and for the BFS-trie-based method is 57,756, which represents a saving of 19.91%. When the number of errors is known (and equals unity), the number of operations for the DFS-trie-based is 66,545, and for the BFS-trie-based method is 52,956, which represents a saving of 20.4%.

7.5 Conclusion

In this Chapter we presented a feasible solution for the problem of estimating a transmitted string X^* by processing the corresponding string Y, which was a noisy version of X^*, an element of a finite (but possibly, large) dictionary H, when the whole dictionary was considered simultaneously. The method used a BFS on the trie representation of the dictionaries (described in Section 6.6) using the simultaneous dynamic programming equation relating the distances between the prefixes of Y and the prefixes of all the words in H. Since the set of all the prefixes used in the computations could not be used profitably, we proposed an enhancement by the

introduction of a new data structure called the Linked Lists of Prefixes (LLP), which could be constructed when the dictionary was represented using a trie. The LLP was an enhanced, but modified, representation of the trie, which could be used to facilitate the "dictionary-based" dynamic programming calculations.

This BFS-trie-based algorithm for the syntactic PR of strings has been rigorously tested and the results showed significant benefits, with respect to number of computations, of up to 21% when compared to DFS-trie-based algorithm.

Original word (dictionary)	Noisy word	# errors
afternoon	amternoon	1
appeared	appeqyryed	3
average	saovarage	4
business	buszidhanews	5
children	cmhdildeen	3
computers	comcuhtegzrs	5
construction	yconykqgjntructixofjn	10
cooperation	unccwpogpewrvatoot	10
critical	cvrhitical	3
department	depvwayutmehnt	5
developments	neuedlsmomewntezs	11
emergence	ewegence	2
everywhere	waerywhsrze	4
externally	pextecrnwilrfmkly	8
foreign	gkforwogn	4
greater	grerlater	2
immediate	imzpjjedpceiate	7
implemented	irsmopqelmsenydtead	9
influence	onwfluegpnce	4
interesting	mincbtebresqqltoixnpr	11
judgment	rjudgmeqnjht	4
machines	mcbinmekud	6
millions	unumillisintns	7
offensive	focgdfxenfseivore	9
powerful	powwgeortfl	5
procedures	procdurex	2
reported	repkortped	2
responsible	srespdopbbsinqtldze	11
service	sdrrvirce	3
situation	sxittuiadtbuxicoln	9
soldiers	oldiesrs	2
systems	sgystdzemtws	6
throughout	fthriuyghout	3
vessels	lgetsyspels	5
written	qritte	2

Table 7.2: A subset of the dictionary, some noisy strings, from the set SE of the dictionary called Eng, and their characteristics.

Errors	SA	SB	SC	SD	SE
Number of insertions	483	722	912	1089	1131
Number of deletions	248	272	293	306	307
Number of substitutions	430	450	535	546	559
Total number of errors	1161	1444	1740	1941	1997
Average % error	19.76	24.58	29.62	33.04	33.99
Maximum % error per word	30.00	40.00	50.00	60.00	70.00

Table 7.3: Noise statistics of the set SA, SB, SC, SD, and SE that correspond to the **Eng** dictionary.

Errors	SA	SB	SC	SD	SE
Number of insertions	14463	19922	26777	30479	33359
Number of deletions	7274	8165	8721	9141	9323
Number of substitutions	13008	14830	16244	16544	16953
Total number of errors	34745	42917	51742	56164	59635
Average % error	19.15	23.66	28.52	30.96	32.87
Maximum % error per word	30.00	40.00	50.00	60.00	70.00

Table 7.4: Noise statistics of the set SA, SB, SC, SD, and SE that correspond to the **Dict** dictionary.

Errors	SA	SB	SC	SD	SE
Number of insertions	58465	79980	105276	122015	133809
Number of deletions	31086	35439	38328	39446	40353
Number of substitutions	55936	64005	68819	70551	71785
Total number of errors	145487	179424	212423	232012	245947
Average % error	18.74	23.11	27.36	29.88	31.68
Maximum % error per word	30.00	40.00	50.00	60.00	70.00

Table 7.5: Noise statistics of the set SA, SB, SC, SD, and SE that correspond to the **Webster** dictionary.

Oper	SA		SB		SC		SD		SE	
	DFS	BFS	DFS	BFS	DFS	BFS	DFS	BFS	DFS	BFS
Add	57.9	55.9	60	58.7	61.6	60.8	63.1	62.9	63.5	63.4
Min	39.5	28.4	40.9	29.8	42.0	30.8	43.0	31.9	43.3	32.1
Total	97.4	84.3	100.9	88.5	103.6	91.6	106.1	94.8	106.8	95.5
Sav	**13.44**		**12.28**		**11.58**		**10.65**		**10.58**	
Acc	94.5	94.3	92.5	91.7	88.1	89.0	87.2	87.6	86.4	85.0

Table 7.6: The experimental results obtained from each of the five sets for the **Eng** dictionary. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the BFS-trie-based method obtains over the DFS-trie-based method.

Oper	SA		SB		SC		SD		SE	
	DFS	BFS	DFS	BFS	DFS	BFS	DFS	BFS	DFS	BFS
Add	43706	39364	44765	40772	46224	42710	46984	43721	47610	44554
Min	29737	19982	30443	20686	31416	21655	31923	22161	32340	22577
Total	73443	59346	75208	61458	77640	64365	78907	65882	79950	67131
Sav	**19.19**		**18.28**		**17.09**		**16.50**		**16.03**	
Acc	88.8	88.8	84.8	84.2	81.1	81.3	79.7	79.9	78.9	78.9

Table 7.7: The experimental results obtained from each of the five sets for the **Dict** dictionary. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the BFS-trie-based method obtains over the DFS-trie-based method.

Oper	SA		SB		SC		SD		SE	
	DFS	BFS	DFS	BFS	DFS	BFS	DFS	BFS	DFS	BFS
Add	668990	584556	683275	603555	707927	628293	714927	645552	723991	657615
Min	454119	296341	463642	305840	476077	318209	484743	326839	490786	332870
Total	1123109	880897	1146917	909395	1184004	946502	1199670	972391	1214777	990485
Sav	**21.56**		**20.70**		**20.05**		**18.94**		**18.46**	
Acc	85.8	86.3	81.2	81.6	77.7	78.1	76.5	76.7	76.0	76.0

Table 7.8: The experimental results obtained from each of the five sets for the **Webster** dictionary. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the BFS-trie-based method obtains over the DFS-trie-based method.

Oper	SA		SB		SC		SD		SE	
	DFS	BFS	DFS	BFS	DFS	BFS	DFS	BFS	DFS	BFS
Add	41644	38135	43496	40151	45441	42402	46415	43532	47145	44422
Min	30010	19315	31329	20339	32681	21477	33357	22048	33856	22495
Total	71654	57450	74825	60490	78122	63879	79772	65580	81001	66917
Sav	**19.82**		**19.15**		**18.23**		**17.79**		**17.38**	
Acc	88.8	88.8	84.8	84.2	81.1	81.3	79.7	79.9	78.9	78.9

Table 7.9: The experimental results obtained from each of the five sets for the **Dict** dictionary for the optimized case when the percentage of error is considered known *a priori*. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the BFS-trie-based method obtains over the DFS-trie-based method.

Oper	Error = 1		Error = 2		Error = 3	
	DFS	BFS	DFS	BFS	DFS	BFS
Add	38694	35104	43391	40202	48633	46394
Min	27851	17852	31202	20401	34835	23497
Total	66545	52956	74593	60603	83468	69891
Sav	**20.4**		**18.8**		**14.8**	
Acc	91.7	92.1	80.5	79.9	69.5	69.4

Table 7.10: The experimental results obtained for each of the three different errors for the **Dict** dictionary for the optimized case when the number of error is not known. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the BFS-trie-based method obtains over the DFS-trie-based method.

Oper	Error = 1		Error = 2		Error = 3	
	DFS	BFS	DFS	BFS	DFS	BFS
Add	43706	39364	44765	40772	46224	42710
Min	29737	19982	30443	20686	31416	21655
Total	73443	59346	75208	61458	77640	64365
Sav	**19.19**		**18.28**		**17.09**	
Acc	88.8	88.8	84.8	84.2	81.1	81.3

Table 7.11: The experimental results obtained for each of the three different errors for the **Dict** dictionary for the optimized case when the number of error is considered known *a priori*. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the BFS-trie-based method obtains over the DFS-trie-based method.

Chapter 8

A Look-Ahead Branch and Bound Pruning Scheme

8.1 Introduction

In the Artificial Intelligence (AI) domain, Branch and Bound schemes are used when we want to prune paths that have costs above a certain threshold. These techniques have been applied to prune, for example, game trees.

In this Chapter[1], we present a new Branch and Bound pruning strategy that can be applied to dictionary-based approximate string matching when the dictionary is stored as a trie. The new strategy attempts to look ahead at each node, c, before moving further, by merely evaluating a certain local criterion at c. The search algorithm according to this pruning strategy will not traverse inside the $subtrie(c)$ unless there is a "hope" of determining a suitable string in it. In other words, as opposed to the reported trie-based methods [76], [137], the pruning is done *a priori* even before embarking on the edit distance computations. The new strategy depends highly on the variance of the lengths of the strings in H and makes use of the fact that the length

[1]Some of the results in this Chapter have been published in the *Proceedings of the 4th International Conference on Computer Recognition Systems CORES'05*, Rydzyna Castle, Poland, 22-25 May, 2005 [19]. The corresponding journal paper is currently being reviewed [17].

of the strings to be compared are known *a priori*. It combines the advantages of partitioning the dictionary according to the string lengths, and the advantages gleaned by representing H using the trie data structure. The results demonstrate a marked improvement (up to 30% when costs are of a 0/1 form, and up to 47% when costs are general) with respect to the number of operations needed on three benchmark dictionaries. This high improvement is at the expense of just storing two extra memory locations for each node in the trie. Also, if the length of the noisy word is very far from all the acceptable words in the dictionary, i.e., those which can give an edit error smaller than K, the edit distance computations for this noisy word can be totally pruned with only a single comparative test. All of these concepts will be outlined presently.

The Chapter is organized as follows: Section 8.2 reviews the different possible cutoffs that can be applied to tries. Section 8.3 describes, in detail, the new LHBB scheme, and presents the algorithm for obtaining X^+. Section 8.4 describes how the new technique can be used with general costs. Section 8.5 presents the experiments done and provides the results that demonstrate the benefits of the new method. Section 8.6 concludes the Chapter.

8.2 Tries and Cutoffs

As mentioned earlier, tries offer text searches with costs which are independent of the size of the document being searched. The data is represented not in the nodes but in the path from the root to the leaf. Thus all strings sharing a prefix will be represented by paths branching from a common initial path. Figure 8.1 shows an example of a trie for a simple dictionary of words {For, Form, Fort, Fortran, Forma, Forget, Format, Formula, Forward}[2].

Also, as we know, Shang *et al.* [137] used the trie data structure for exact and approximate string searching. They presented a trie-based method whose cost is independent of the document size, and proposed a k-approximate match algorithm on a text represented as a trie, which performs a Depth First Search (DFS) on the trie. The insight they provided was that the trie

[2]The trie in this Figure is slightly different from the one in Figure 7.1. A new word "forma" is added to the dictionary to better illustrate the difference between applying the new technique and applying Ukkonen's cutoff.

representation of the text drastically reduces the Dynamic Programming (DP) computations. The trie representation compresses the common prefixes into overlapping paths, and the corresponding column (in the DP matrix) needs to be evaluated only once. All of this was discussed in more detail in Chapter 5.

In [137], the authors applied a known pruning strategy called Ukkonen's cutoff [151] (described in Section 6.6.2) to abort unsuccessful searches. For example, in Figure 8.1, if the noisy word is Y = "fwt", Ukkonen's cutoff will force searching in any path to terminate prematurely, whenever the prefixes to be examined cannot lead to Y with an error less than K. This means that the paths that cannot lead to a solution can be pruned, and thus the method limits the search to a portion of the search space. So, for example, if $K = 2$, the path for the word "fortran" will be cut off after doing the calculations at node r, and so no more search will be done at the trie rooted at node r. Figure 8.2 shows the pruning done when applying the Ukkonen's cutoff technique. Chang and Lawler [43] showed that Ukkonen's algorithm evaluated $O(K)$ expected DP table entries. If the fanout of the trie is Σ, the trie method needs to evaluate only $O(K|\Sigma|^K)$ expected DP table entries, which are independent of the number of noisy words we are searching for. Their experiments showed that their method significantly out-performs the nearest competitor for $K = 0$ and $K = 1$, which are arguably the most important cases. They also compared their work experimentally with *agrep*, a software package for Unix that implements the algorithm presented in [162]. This is an extension (for a numeric scheme) for the exact string matching algorithm developed by Baeza-Yates and Gonnet [27]. Also, a similar cutoff technique, called the edit distance cutoff, was used in [113], to devise error-tolerant finite-state recognizers.

Most of the dictionaries used in string correction contain strings of different lengths. This variation in the string lengths could help in excluding many strings from the computation of the corresponding edit distances when compared against the noisy word, as strings of this length couldn't have possibly given rise to the given noisy string. Indeed, this conclusion is because the difference in their lengths is more than the number of errors allowed. This property was used in [55] to *partition* the dictionary and eliminate the words to be compared in the dictionary. A set is built from all possible partitions, and a *string-to-string* correction technique was used to get

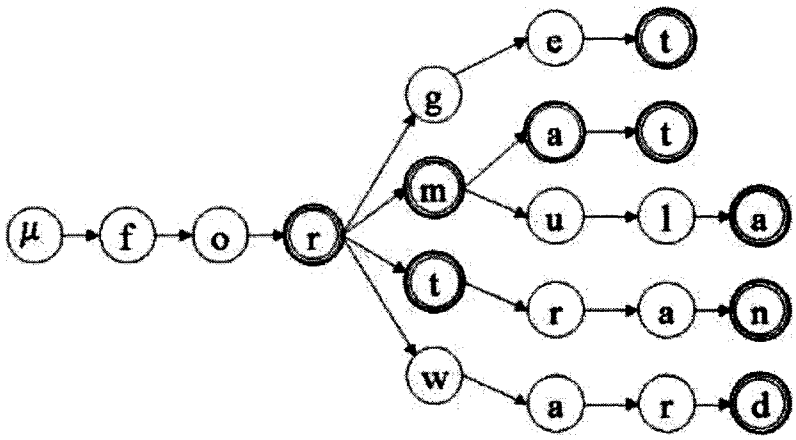


Figure 8.1: An example of a dictionary stored as a trie with the words {For, Form, Fort, Fortran, Forma, Forget, Format, Formula, Forward}.

the best match. The authors of [55] limited their discussion to cases in which the error distance between the given string and its nearest neighbors in the dictionary was small. The problem with this method is that this set can be quit large for larger values of K, and can thus include the whole dictionary. This could lead to string-to-string comparisons for a large partition of the dictionary, or even the whole dictionary itself. Another drawback of this method is that two words sharing common prefixes, but which reside in different partitions, will necessitate redundant computations for the entire common segments.

8.3 Look-Ahead Branch and Bound Scheme

Given the fact that the dictionary is stored in a trie, any PR-related search for a word in H will have to search the entire trie. To reduce the computational burden, we shall now show how we can use concepts in AI to "reduce" the portion of the search space investigated. We do this by invoking the principles of Branch and Bound (BB) strategies.

In AI, whenever we encounter a search space, the latter can be searched in a variety of ways such as by invoking a BFS, a DFS or even a Best-First Search scheme, where, in the latter, the

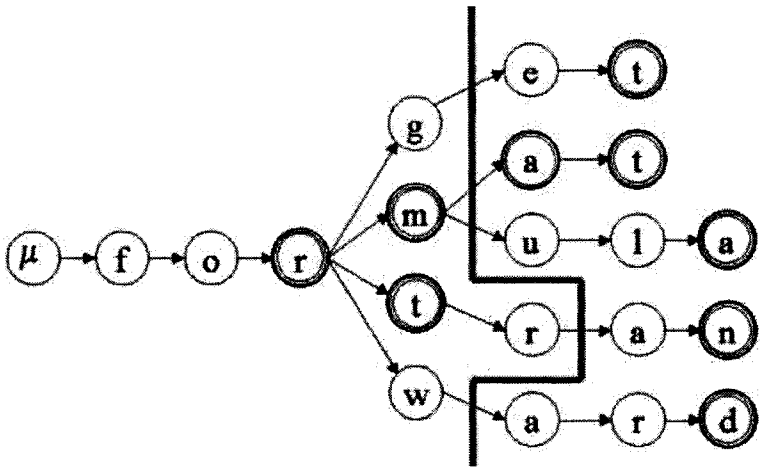


Figure 8.2: The cutoff done for the trie example when applying Ukkonen's cutoff, for Y = "fwt" and $K = 2$.

various paths are ranked by using an appropriate heuristic function. But if the search space is very large, BB techniques can be used to prune the search space. This is done by estimating the costs of the various potential paths with a suitable heuristic, and if the cost of any path exceeds a pre-set threshold, this path (or branch) is pruned, and the search along this path is aborted. What we lose are that we need more processing operations *per* node, and possibly additional storage for storing some local indices. But what we gain is that we can prune numerous unneeded paths, and thus save enormous redundant computations.

In the present case, we now investigate how we can eliminate searching along some of the paths of the *trie*. Thus, we effectively map the trie into the "search tree" of an AI algorithm, and seek a suitable heuristic to achieve the pruning. The heuristic that we propose has three characteristics, namely, it has a static component, a dynamic component, and finally, it must be of a look-ahead sort, as opposed to the cut-off methods already proposed [113], [151]. Indeed, the edit distance cutoff used in [113] and Ukkonen's cutoff used in [137], depend on the *a posteriori* evaluation of the edit distances even as we process more characters from prefixes of strings in the dictionary. In other words, in these schemes, the pruning is invoked only *after* calculating the edit distance of the prefix being currently processed, and results only when there is no possible conversion from this prefix to the noisy word in hand. We will now examine each of the components of our BB heuristic.

8.3.1 The look-ahead component

The idea that we advocate, is to prune from the calculations the sub-tries in which the strings stored are not within a pre-defined acceptable condition. The lengths of the string stored in *subtrie*(c) can be directly related to the maximum number (percentage) of errors allowed, and thus can simplify the equations and the condition that has to be tested per node even before we traverse the path. The maximum error can give an indication about the maximum and minimum lengths of the strings allowed.

We propose a strategy by which we will not traverse the *subtrie*(c) unless there is a "hope" of determining a suitable string in it, where the latter is defined as the string that could be garbled into Y with less than K errors. Stating that a *subtrie*(c) has to be pruned, implies that the minimum possible errors of all the substrings (to transform them into Y) that are stored in *subtrie*(c) is bigger than K. So in our new heuristic, because the maximum error can be known *a priori*, and because the lengths[3] of the strings in H are also known *a priori*, we can look ahead at each node c, and decide whether we have to prune the *subtrie*(c). If we do, we are guaranteed that all the strings stored will not possibly lead to Y with less than K errors.

8.3.2 The dynamic component

The lengths of the *prefixes* to be processed can also be directly related to the maximum error, K. The maximum and minimum allowed lengths for all strings stored in a *subtrie*(c) are easily related to the length of Y, M, and to the error K, as:

$$max(length(X^+)) \leq M + K$$

Further, if we are at node c and the length of the prefix calculated so far is N', and the length of any string in *subtrie*(c) is N'', this constraint can be re-written as:

[3]Observe that our method is quite distinct from the dictionary partitioning strategy which is also based on string lengths [55].

$$max(N' + N'') \leq M + K$$

Since N' is constant per node c, this means:

$$max(N'') \leq M - N' + K \tag{8.1}$$

Similarly, since K is the *absolute* number of errors,

$$min(N'') \geq M - N' - K \tag{8.2}$$

Using these dynamic equations for the minimum and maximum lengths allowed for string eligible to be X^+, we can easily test at each node if the lengths of the suffixes stored are within these acceptable ranges, namely, $min(N''), max(N'')$.

8.3.3 The static component

To test if we are within acceptable ranges for the potential candidates for X^+, we need to store the information needed for these calculations within each node, so that the conditions can be tested locally (and quickly) within the corresponding node. Fortunately, this information is already known *a priori* and is easily calculated and stored. More specifically, we need to store two values at each node of the trie, which are:

- *Maxlen*: A value stored at a node which indicates the length of the path between this node and the most distant node representing an element of the dictionary H. This is actually the length of the largest suffix for all the suffixes stored in the subtrie rooted at this node.
- *Minlen*: A value stored at a node which indicates the length of the path between this node and the least distant node representing an element in H. This is actually the length of the smallest suffix for all the suffixes stored in the trie rooted at this node.

8.3.4 The overall heuristic

At each node of the trie, before we do any further computations, we test the following conditions, referred to as the LHBB conditions:

(a) $Minlen > M - N' + K$ *obtained by negating Eq.* (8.1), or

(b) $Maxlen < M - N' - K$ *obtained by negating Eq.* (8.2).

If (a) or (b) is true, it means there is no hope of finding a solution within the present subtrie, and so we prune the calculations for the subtrie. If K represents the percentage of error in Y, then K, in the equations above, can be replaced by $K * M$. The LHBB, as its name implies, first looks forward at each node, and sees if it is expected to perform any further calculations. If at any time we reach a string X in the dictionary (which is thus an accepting node), we accept the string if the $D(X, Y) \leq K$.

Consider, for example, the same trie in Figure 8.1, where the noisy word Y = "fwt". By applying the LHBB, for $K = 2$, the path for the word "fortran" will be pruned before doing the edit distance calculations at node t, and so no further search will be done at the trie rooted at node t. But since node t is an accepting node, we need to calculate its edit distance. This thus saves two levels of computations for the edit distance for the trie rooted at node t with respect to the previous method. The path for the word "forget", however, will be pruned before doing edit distance calculations at node g. Since g is not an accepting node, it will be also pruned from further calculations. Figure 8.3 shows the pruning done when applying the LHBB technique only.

The LHBB can also be used in combination with the Ukkonen's cutoff used earlier for tries. The LHBB requires only the testing of the above conditions using the values stored locally within each node. Figure 8.4 shows the pruning when applying both techniques.

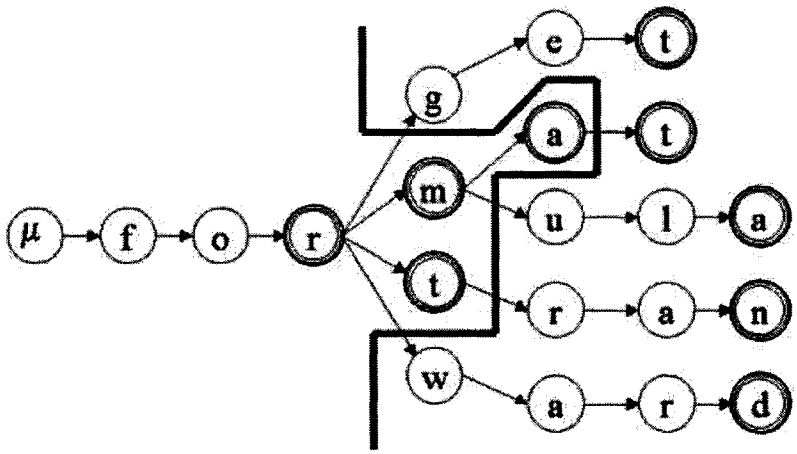


Figure 8.3: The cutoff done for the trie example when applying only the LHBB technique, for Y = "fwt" and $K = 2$.

8.3.5 Algorithm for Obtaining X^+ Using LHBB

In this section, we present the algorithm for obtaining X^+, by pruning using LHBB in trie-based calculations. The algorithm follows the steps of the trie method except that it includes the LHBB pruning (see Algorithm 8.1). The lines indicated by asterisks show the modified part. Also, computing the *Maxlen* and *Minlen* values is fairly straightforward, and can be done during the construction of the trie, as the strings are inserted one by one. When inserting a string in the trie, we already know the length of this string, and hence the values of *Maxlen* and *Minlen* need to be adjusted only for the nodes along the path included in the insertion, which can be done by comparing their old values with the length of the newly inserted string.

8.4 A Look-Ahead BB Scheme for General Costs

When general costs are used, relating the computed (or anticipated) edit distance to the maximum edit distance or the maximum number of errors is not possible, and so the Ukkonen's cutoff cannot be used. In this case, as far as we know, the only available technique that can be used to prune the trie is the one we propose. This is because the new technique can serve as

Algorithm 8.1 Algorithm LHBB

Input: A Dictionary stored as a Trie T with the *maxlen* and *minlen* values stored at each node and a noisy string Y to be searched. N' is the length of the prefix processed so far, and *ed* is the Edit distance matrix used for calculating the edit distances column by column.
Output: The candidate correct string X^+.
Method:

1: $N = 0$
2: start from the root $tn = root$ of T.
3: **while** tn is not NULL **do**
4: get the next child for tn in the DFS path and make tn points to it.
5: increment N'.
6: **if** no more children for tn **then**
7: return to the parent node.
8: decrement N'.
9: go to line 3.
10: **end if**
11: {**********LHBB**********}
12: {Test the LHBB condition}
13: $N\_min = M - K - N'$ {adjust the condition boundaries}
14: $N\_max = M + K - N'$
15: **if** $tn \rightarrow Maxlen < N\_min$ *or* $t \rightarrow Minlen > N\_max$ **then**
16: **if** tn is accept node **then**
17: calculate the $ed[N']$ column for this node using Ukkonen's cutoff.
18: **if** $ed[N'][M] < K$ **then**
19: Put the string represented by this node in the acceptable candidate list.
20: **end if**
21: **end if**
22: return to the parent node. {prune any extra search in the trie rooted at node tn}
23: decrement N'.
24: go to line 3.
25: **else**
26: {**********}
27: {Do the Ukkonen's cutoff calculations}
28: calculate the $ed[N']$ column for node tn.
29: min = min edit distance for column $ed[N']$
30: {Test the edit distance cutoff condition}
31: **if** $min > K$ **then**
32: return to the parent node. {prune any extra search in the trie rooted at node tn}
33: decrement N'.
34: go to line 3.
35: **else**
36: **if** tn is accept node and $ed[N'][M] < K$ **then**
37: Put the string represented by this node in the acceptable candidate list.
38: **end if**
39: **end if**
40: **end if**
41: **end while**
42: **Return** X^+ = the string with minimum *ed* for the strings in the candidate list.
43: **End Algorithm LHBB**

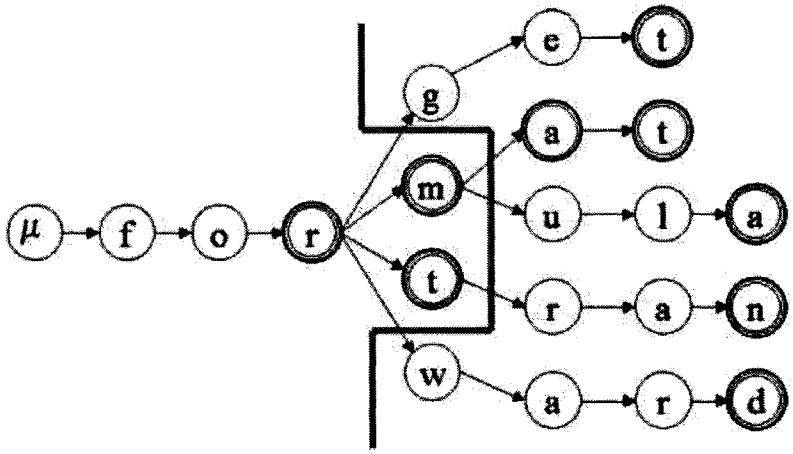


Figure 8.4: The cutoff done for the trie example when applying both Ukkonen's cutoff and the LHBB technique, for Y = "fwt" and $K = 2$.

a direct link between the length of the strings in the dictionary and the maximum number of errors, independent of the costs that are assigned for the errors, while similarly using the same LHBB conditions.

When the maximum number of errors is used as a criterion, a further improvement can be done by pruning the paths if the length of the prefix calculated so far is larger than $M + K$, and this (as a replacement for Ukkonen's cutoff) can be used in conjunction with the LHBB technique. Other enhancements can be applied when only the best match is required. In this case, we can "cut off" the subtrie at any node if the *min* value (the minimum edit distance value in any column during the calculation, which is the minimum edit distance value to change any prefix in H to Y) is larger than the edit distance of the nearest neighbor word found so far. If it is, we can prune the trie at this node. An analogous technique was also applied in [137] when the best match was required. We shall see that when applying the new technique, the enhanced algorithm yields an even better performance.

8.5 Experimental Results

To investigate the power of our new method with respect to computation we conducted various experiments. The results obtained were remarkable with respect to the gain in the number of computations needed to get the best estimate X^+. By computations we mean the addition and minimization operations needed, including the minimization operations required for calculating the LHBB criterion. The LHBB scheme was compared with the original trie-based work for approximate matching [137] when the edit distance costs were of a 0/1 form and of a general form.

The three benchmark data sets used in our experiments were the same as those used in Section 7.4. Again, each data set was divided into two parts: a *dictionary* and the corresponding *noisy file*. The dictionary was composed of the words or sequences that had to be stored in the Trie. The noisy files consisted of the strings which were searched for in the corresponding dictionary.

Three sets of corresponding noisy files were created using the noise generator model described in [121], and in each case, the files were created for a specific error value. The three error values tested were for $K = 1,\ 2,\ and\ 3$, as is typical in the literature [113], [137].

The two methods, Trie (the original method) [137] and our scheme, LHBB, were tested for the three sets of noisy words. We report below a summary of the results obtained in terms of the number of computations (additions and minimizations) in millions.

We conducted *two* set of experiments: The *first* set of experiments was when the costs are of a 0/1 form, and the *second* set of experiments was when the costs were general and were generated from the table of probabilities for substitution (typically called the confusion matrix), which was based on the proximity of character keys on the standard QWERTY keyboard and is given in [121][4]. The conditional probability of inserting any character, given that an insertion occurred, was assigned the value 1/26; and the probability of deletion was set to be 1/20.

[4]It can be downloaded from `www.scs.carleton.ca/~oommen/papers/QWERTY.doc`.

Operation	Eng		Dict		Webster	
	Trie	LH	Trie	LH	Trie	LH
Additions	5.2	3.5	550	390	3,360	2,224
Improvement	32.69		29.09		33.80	
Minimizations	5.5	4.2	575	454	3,489	2,552
Improvement	23.63		21.04		26.85	
Total	10.7	7.7	1,125	844	6,849	4,776
Improvement	**28.03**		**24.97**		**30.26**	
Time	0	0	16	9	73	56
Improvement	**0**		**43.75**		**23.28**	

Table 8.1: The results obtained in terms of the number of operations (additions and minimizations) needed when the maximum number of errors $K = 1$, and the costs are of a 0/1 form. The figures given are in Millions. The time shown is in seconds, and the *total* improvement obtained is in "bold" face.

8.5.1 Experimental Setup I: 0/1 Costs

In Tables 8.1, 8.2, and 8.3, the results show the significant benefits of the LHBB scheme with up to 30% improvement. For example, for the *Websters* dictionary, when $K = 1$, the number of computations is $6,849$ million and $4,776$ million respectively, which represents an improvement of 30.26%. The improvement decreases as the number of errors increases, a condition that can be expected because as K increases, more neighbors have to be tested, which in turn implies that more parts of the trie have to be examined. By studying the results we see that the improvements are quite prominent even for $K = 2$ *and* 3. The improvement is more than 20%, which is considerable compared to what can be achieved by the state-of-the-art trie methods. Additionally, observe that the search is still bounded by the $O(K|\Sigma|^K)$ DP tables entries, because we use the trie to store the dictionary. This is in contrast to the method discussed in [55], where the set R becomes so large as K increases, and the method is reduced to the tedious corresponding sequential string-to-string comparison techniques.

Further improvement can be obtained when the algorithm is only searching for the best match. We can then apply the same strategy for the same dictionaries when $K = 1$, 2, *and* 3. The results are shown in Table 8.4 for the *Dict* dictionary, as the results for the other dictionaries are almost identical. The results show the significant benefits of the LHBB scheme with up

Operation	Eng		Dict		Webster	
	Trie	LH	Trie	LH	Trie	LH
Additions	18.1	12.3	3,654	2,648	24,901	16,773
Improvement	32.04		27.64		32.64	
Minimizations	19.1	14.5	3,809	3,048	25,830	19,077
Improvement	24.08		20.13		26.14	
Total	37.2	26.8	7,463	5,696	50,805	35,850
Improvement	**27.95**		**23.81**		**29.43**	
Time	1	1	134	81	615	495
Improvement	**0**		**39.55**		**19.51**	

Table 8.2: The results obtained in terms of the number of operations (additions and minimizations) needed when the maximum number of errors $K = 2$, and the costs are of a 0/1 form. The figures given are in Millions. The time shown is in seconds, and the *total* improvement obtained is in "bold" face.

to 26% improvement (compare with Table 8.1). For example, when $K = 1$, the number of computations is $1,120$ million and 819 million respectively, which represents an improvement of 26.87%. The improvement decreases as the number of errors increases.

8.5.2 Experimental Setup II: General Costs

The second set of experiments was conducted when the costs were general as explained above. In this case Ukkonen's cutoff cannot be applied as the maximum number of errors cannot be related to the edit distance costs any more. The results are shown in Table 8.5 for the *Dict* dictionary, and not included for the other dictionaries, as their results are relatively the same. The results show the significant benefits of the LHBB scheme with up to 47.98% improvement. For example, when $K = 1$, the number of computations is $66,545$ million and $34,614$ million respectively, representing an improvement of 47.98%.

When the best match is required and the inter-symbols costs are general, we can apply the same strategy using the same dictionaries for the cases when $K = 1,\ 2,\ and\ 3$. The results are shown in Table 8.6 for the *Dict* dictionary (the results for the other dictionaries are omitted). The results show the significant benefits of the LHBB scheme with up to 42.59% improvement.

Operation	Eng		Dict		Webster	
	Trie	LH	Trie	LH	Trie	LH
Additions	37.1	27.6	12,117	9,205	93,373	68,133
Improvement	25.60		24.03		29.27	
Minimizations	38.8	31.8	12,613	10,430	99,852	76,530
Improvement	18.04		17.30		23.35	
Total	75.9	59.4	24,730	19,635	196,190	144,666
Improvement	**21.73**		**20.60**		**26.26**	
Time	2	1	306	261	2,327	1,899
Improvement	**50.00**		**14.70**		**18.39**	

Table 8.3: The results obtained in terms of the number of operations (additions and minimizations) needed when the maximum number of errors $K = 3$, and the costs are of a 0/1 form. The figures given are in Millions. The time shown is in seconds, and the *total* improvement obtained is in "bold" face.

Operation	K=1		K=2		K=3	
	Trie	LH	Trie	LH	Trie	LH
Additions	548	389	3,511	2,527	11,131	8,338
Improvement	29.01%		28.03%		25.09%	
Minimizations	572	430	3,657	2,779	11,569	9,033
Improvement	24.82%		24.01%		21.92%	
Total	1,120	819	7,168	5,306	22,700	17,371
Improvement	**26.87%**		**25.98%**		**23.48%**	

Table 8.4: The results obtained in terms of the number of operations (additions and minimizations) needed when maximum number of errors, K is 1, 2, *and* 3, the costs are of 0/1 from, the best match optimization is applied, and the *Dict* dictionary is used. The figures given are in Millions. The *total* improvement obtained is in "bold" face.

For example, when $K = 1$, the number of computations is $16,827$ million and $9,661$ million respectively, which represents an improvement of 42.59%. If we compare the results of Table 8.5 and Table 8.6, we will find that the best match optimization add further improvement of 72% for the LH when $K = 1$ and the costs are general. This best match improvement is not significant when the costs are of a 0/1 form because in this case Ukkonen's cutoff is already applied.

Operation	K=1		K=2		K=3	
	Trie	LH	Trie	LH	Trie	LH
Additions	38,694	19,412	43,391	29,070	48,633	37,990
Improvement	49.83%		33.01%		21.88%	
Minimizations	27,851	15,202	31,202	22,411	34,835	28,917
Improvement	45.42%		28.17%		16.98%	
Total	66,545	34,614	74,593	51,481	83,468	66,907
Improvement	**47.98%**		**32.19%**		**19.84%**	

Table 8.5: The results obtained in terms of the number of operations (additions and minimizations) needed when maximum number of errors, K is 1, 2, *and* 3, the costs are general, and the *Dict* dictionary is used. The figures given are in Millions. The *total* improvement obtained is in "bold" face.

Operation	K=1		K=2		K=3	
	Trie	LH	Trie	LH	Trie	LH
Additions	8,236	4,503	11,412	7,715	16,990	12,821
Improvement	45.33%		32.40%		24.54%	
Minimizations	8,591	5,158	11,900	8,734	17,675	14,378
Improvement	39.96%		26.61%		18.65%	
Total	16,827	9,661	23,312	16,449	34,665	27,199
Improvement	**42.59%**		**29.44%**		**21.54%**	

Table 8.6: The results obtained in terms of the number of operations (additions and minimizations) needed when maximum number of errors K is 1, 2, *and* 3, the costs are general, the best match optimization is applied, and the *Dict* dictionary is used. The figures given are in Millions. The *total* improvement obtained is in "bold" face.

8.6 Conclusion

In this Chapter, we presented a new Branch and Bound (BB) scheme that can be applied to approximate string matching using tries, which we called a *Look-Ahead* Branch and Bound scheme or the LHBB-trie pruning strategy. The new scheme made use of the information about the lengths of the strings stored in the dictionary and assumed that the maximum number of errors was known *a priori*. As a result, the lengths of the strings could be related to the maximum number (percentage) of errors, K. The heuristic that we proposed, worked specifically on a trie and had three characteristics, namely a static component, a dynamic component, and finally, it was of a look-ahead sort, as opposed to the cutoff methods already proposed in [113], [151].

The new LHBB pruning could also be used together with Ukkonen's cutoff technique [151].

Several experiments were conducted using three benchmarks dictionaries for noisy sets involving different error values, $K = 1,\ 2,\ and\ 3$. The results demonstrated a significant improvement, with respect to the number of operations needed for approximate searching using tries, which could be even as high as 30%. We also demonstrated how we could extend the latter when the costs were general, in which case improvements of up to 47% were obtained, when compared with the DFS-trie-based algorithm where the Ukkonen's cutoff could not be used.

Chapter 9

Clustered-Beam-Search

9.1 Introduction

This Chapter[1] shows how we can optimize non-sequential syntactic PR computations by incorporating heuristic search schemes used in AI into the approximate string matching problem. First, we present a new technique enhancing the Beam Search (BS), which we call the Clustered Beam Search (CBS), and which can be applied to any tree searching problem[2]. We then apply the new scheme to achieve approximate string matching when the dictionary is stored as a trie. The trie is implemented as a Linked List of Prefixes (LLP) as shown earlier. The latter permits *level-by-level* traversal of the trie (as opposed to traversal along the "branches"). The newly-proposed scheme can be used for Generalized Levenshtein distances, and also when the maximum number of errors is not given *a priori*. It has been rigorously tested on three benchmarks dictionaries by recognizing noisy strings generated using the model discussed in

[1]Patent applications have been filed to protect the intellectual property and the results contained in this Chapter [21]. Also, some of the results from this Chapter have appeared in the *Proceedings of ICAPR2005, the 2005 International Conference on the Advances of Pattern Recognition*, in Bath, United Kingdom, August 2005 [18]. This talk was the Plenary talk of the Conference. The extended journal version of this work is accepted and will be published in the *IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics*, in June 2006 [23].

[2]The new scheme can also be applied to a general graph structure, but we apply it to the trie due to the dominance of the latter in our application domain, approximate string matching.

[121], and the results have been compared with the acclaimed standard [131], the Depth-First-Search (DFS) trie-based technique [137]. The new scheme yields a marked improvement (of up to 75%) with respect to the number of operations needed, and at the same time maintains almost the same accuracy. The improvement in the number of operations increases with the size of the dictionary. The CBS heuristic has also been compared with the performance of the original BS heuristic when applied to the trie structure, and the experiments again show an improvement of more than 91%. Furthermore, by marginally sacrificing a small accuracy in the general error model, or by permitting an error model that increases the errors as the length of the word increases (to be explained presently), an improvement of more than 95% in the number of operations can be obtained.

The layout of the Chapter is as follows: Section 9.2 describes the enhancement, the proposed scheme, namely, the Clustered Beam Search (CBS), and presents the new proposed algorithm, and discusses its complexity. The section also presents an example. Section 9.3 describes how the CBS can be applied to approximate string matching. It presents the corresponding heuristic functions, the data structures used, and the algorithm. Section 9.4 concentrates on the experiments done to test the new scheme, and presents the results along with a detailed discussion. Section 9.5 proposes additional optimization possibilities of the scheme if the error model is modified, and Section 9.6 concludes the Chapter.

9.2 Proposed Clustered Beam Search (CBS)

We propose a new heuristic search strategy that can be considered as an enhanced scheme for the BS. The search will be done level-by-level for the tree structure[3]. By "level" we mean the nodes at the same depth from the root. At each step we maintain $|A|$ small priority queues, where A is the set of possible clusters that we can follow in moving from one node to its children[4]. It is well known in the theory of PR that the word cluster is associated with a distance or dissimilarity

[3]For the case where the graph is a general structure (and not a tree), we need to maintain the traditional *Open* and *Close* lists that are used for Best First Search [91, 126].

[4]For example, A can be the set of letters in the alphabet in the case of the trie.

measure. Strictly speaking, the way we gather the nodes is referred to as a "bucket" in the traditional data structures literature. Since the latter term is overused, we shall take the liberty of referring to the subset of nodes in any collection as a "cluster". The queues maintain only the best first q nodes corresponding to each cluster, and in this way the search space is pruned to have only $q|A|$ nodes in each level.

CBS is like BS in that it considers only some nodes in the search and discards the others from further calculations. Unlike BS, it does not compare all the nodes in the same level in a single priority queue, but rather compares only the nodes in the same cluster. The advantage of such a strategy is that the load of maintaining the priority queue is divided into $|A|$ priority queues. In this way, the number of minimization operations needed will dramatically decrease for the same number of nodes taken from each level.

As we increase q, the accuracy increases and the pruning ability decreases. When the evaluation function is informative, we can use small values for q. As q increases, the cost associated with maintaining the order of the lists may overcome the advantage of pruning. The possibility of including a larger number of nodes per level increases with the new CBS scheme when compared to the BS leading to increased accuracy.

9.2.1 The Proposed CBS Algorithm and its Complexity

The pseudo code for the proposed algorithm is shown in Algorithm 9.1. The saving of the CBS over the BS appears in step 8, in which we only maintain a small queue. The result regarding the complexity follows.

Theorem 9.1. *Let A be the set of clusters for which we can decide to branch at each node of a tree of maximum depth h, and let c be the search cost associated with each node. Also, if q is the beam width for the CBS, then $q|A|$ is the beam width of the BS, where $|A|$ is the number of clusters. Then, the time complexity for the CBS will be $O(c.h(q|A|^2)\log(q))$, and for the BS will be $O(c.h(q|A|^2)\log(q|A|))$. Also, if the $|A|$ queues can be processed in parallel, the time complexity for the CBS will be $O(c.h(q|A|)\log(q))$.*

Proof. The maximum number of the best selected nodes after pruning at any level is $q|A|$, and the maximum number of children to be expanded for each of these nodes is $|A|$. Thus, the number of children considered for pruning at the next level is bounded by $q|A|^2$. In the CBS, the $q|A|^2$ nodes will be divided among $|A|$ clusters. For each cluster, we need to maintain a priority queue containing q nodes with an individual cost of $O(q|A|\log(q))$. The search cost for each level of the tree, will thus be the product of the total cost for maintaining the $|A|$ queues and the cost per node, which, in turn, will be $O(c(q|A|^2)\log(q))$. Arguing as above for all the h levels, the worst cost will be $O(c.h(q|A|^2)\log(q))$. If the $|A|$ queues can be maintained in parallel, the cost will be reduced to $O(c.h(q|A|)\log(q))$.

This is in contrast with the BS where we have to maintain best $q|A|$ nodes in just a single queue with an associated cost of $O(q|A|\log(q|A|))$. Similarly, the total search cost will then be $O(ch(q|A|^2)\log(q|A|))$. □

The benefits of CBS increases as $|A|$ increases while the performance of the BS will decrease.

9.2.2 Example

Figure 9.1 shows an example for a single level of the calculations. The clusters are $\{a, b, c\}$ and $q = 1$. Thus, for the BS, the number of minimum nodes taken from the queue is $q \times |A| = 3$, and for the CBS we take one minimum node from each queue which is also 3 nodes. So the number of nodes pruned is the same, but the calculations are much more optimized. In the example[5] we see that the number of operations needed for the BS is 12 and for the CBS is 3, when an insertion sort is used. The shaded area shows the best nodes that are kept in the queues. This (possibly trivial) example shows improvements of 75%, which is significant even for this small set. The empirical results in Section 9.4 demonstrate an improvement of up to 92% when applied to approximate string matching, where the number of nodes is much larger.

[5]The choice of the nodes does not have to be the same, but rather can be different. The results also show that the accuracy for CBS is higher due to the fact that we can increase q in the CBS much more than in the BS.

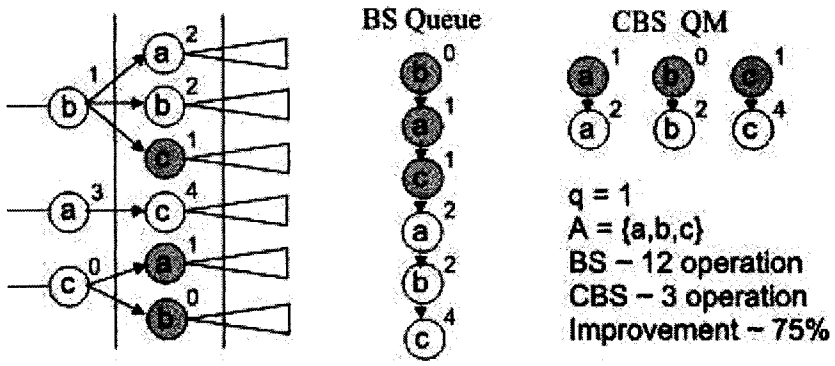


Figure 9.1: An example showing the benefits of the CBS over the BS.

9.3 The CBS for Approximate String Matching

For approximate string matching, the problem encountered involves both ambiguities and excessive time requirements to process a dictionary. Observe that there is no exact solution for the noisy string that one is searching for, and at the same time the process is time consuming because one has to search the entire space to find it. We thus seek a heuristic to determine the nearest neighbor to the noisy string, and one which can also be used to prune the space. The ambiguity of the problem can be resolved by several methods as explained previously. One of the methods is the Depth-First-Search trie-based heuristic that uses the dynamic equations and edit distance calculations described in the next section.

We attempt to describe a heuristic search for the approximate string matching problem and to also prune the search space when the inter-symbol distances are general, and the maximum number of errors cannot be known *a priori*. We will first present the heuristic measure, the data structures used to facilitate the calculations, and finally, the algorithm as applied to approximate string matching.

9.3.1 The Heuristic measure

In string-processing applications, the traditional distance metrics quantify $D(X, Y)$ as the minimum cost of transforming one string X into Y. This distance is intricately related to the costs

Algorithm 9.1 CBS

Input:
a. The tree to be searched, T, with clusters, A.
b. The matrix $Q[|A| \times q]$ to maintain the priority queues.
c. The beam width, q, considered for each cluster.
d. The goal we need to search for.
Output: Success or Failure.
Method:

1: Form $|A|$ single-element queues, $Q[1, \ldots, |A|]$, where each queue $Q[i]$ contains the child in cluster i of the root node.
2: **while** Q is not empty and the goal is not found **do**
3: Determine if any of the elements in Q is the goal.
4: **if** goal found **then**
5: **return** Success.
6: **else**
7: **for** each Q[i], where $1 \leq i \leq |A|$ **do**
8: Sort the nodes originally in $Q[i]$ in terms of the heuristic, keeping only the first best q nodes.
9: Replace each of the q nodes by its successors, and add each successor to the end of the corresponding $Q[c]$ according to its cluster c.
10: **end for**
11: **end if**
12: **end while**
13: **return** Failure.
14: **End Algorithm CBS**

associated with the individual edit operations, the SID operations, as shown in Chapter 5. These inter-symbol distances can be of a form 0/1, parametric, or entirely symbol dependent.

Two possible heuristic functions that can be used to measure the similarity between strings, can also be used as a measure to prune the search space. These measures are:

- Heuristic function F_1: The *Edit distance* $D(X, Y)$

  F_1 can be computed using the dynamic programming rule:

$$D(x_1x_2\ldots x_N,\ y_1y_2\ldots y_M) = \min\ [\ \{D(x_1x_2\ldots x_{N-1},\ y_1y_2\ldots y_{M-1}) + d(x_N, y_M)\},$$
$$\{D(x_1x_2\ldots x_N,\ y_1y_2\ldots y_{M-1}) + d(\lambda, y_M)\},$$

$$\{D(x_1x_2\ldots x_{N-1},\ y_1y_2\ldots y_M) + d(x_N, \lambda)\}],(9.1)$$

where $X = x_1x_2\ldots x_N$ and $Y = yx_1y_2\ldots y_M$.

Recognition using distance criteria is obtained by essentially evaluating the string in the dictionary which is "closest" to the noisy one as per the metric under consideration.

These dynamic equations are exactly the ones that are used for the DFS-trie-based technique [137], where the actual inter-symbol costs are of a form 0/1, and the transposition evaluation is added to the dynamic equation. Both the trie and the matrix are needed in the calculations. When the DFS is used in the calculations, the reader will observe that only a single column will have to be calculated at any given time, and this depends solely on the previously calculated column already stored in the matrix (thus preserving the previous calculations). This was all explained earlier in more detail in Chapter 6.

- Heuristic function F_2: The *Pseudo-distance* $D_1(X, Y)$

  The second heuristic function, F_2, is the pseudo-distance proposed by Kashyap *et al.* [76]. For each character in Y, the pseudo-distance is calculated for the whole trie (level-by-level), and for each character of Y that is processed, we consider two additional levels of the trie (as shown in Chapter 7).

9.3.2 Characteristics of the Heuristic Functions

The first heuristic function, F_1, seems to be very effective for a CBS as will be seen presently. The problem with using F_2 as a pruning measure is that it needs an additional parameter that has to be tuned, in addition to the parameter q. This additional parameter is the length of the prefix of Y that has to be processed after which we can start applying the pruning strategy in order to avoid early removal of entire portions of the trie. Indeed, if we used the pseudo-distance as a measure for pruning, we believe that it will not yield the same accuracy as when the measure $D(X, Y)$ is used, except if excessive tuning is permitted, thus rendering it impractical.

9.3.3 Data Structures Used

There are two main data structures used to facilitate the computations:

- *The Linked List of Prefixes (LLP)*: To calculate the best estimate X^+, we need to divide the dictionary into sets of prefixes. Each set $H^{(p)}$ is the set of all the prefixes of H of length equals to p, where $1 \leq p \leq N_m$, and N_m is the length of longest word in H. More precisely, we want to process the trie level-by-level. The trie divides the prefixes and the dictionary in desired way, and further represents the FSM. The problem in the trie structure is that it can be implemented in different ways, and it is not easily traversed level-by-level. We need a data structure that facilitates the trie traversal, and one that also leads to a unique representation which can always be used to effectively compute the edit distances or pseudo-distances for the prefixes. For this purpose, we have used the same data structure, the Linked Lists of Prefixes (LLP), as in Chapter 7.

  Within each entry of the LLP, we keep a link to the column information that is needed for the calculations required for the next children. In this way, we need to maintain this column information for only the first *childq* nodes of each level. This column link is set to NULL if the node is already pruned. The storage requirement for the LLP is the same as the trie, in addition to the links between children in the same level, and between the different levels themselves and the links to the column information.

- *The Queues Matrix (QM)*: This matrix structure is used during the pruning done for each level in the LLP. A newly initialized QM matrix is needed for each level. The matrix, QM (of dimension $|A| \times q$), can be used to maintain the $|A|$ priority queues and to keep pointers to the best q nodes in each cluster. Each entry in the matrix keeps a pointer to a node in the LLP, and all the pointers in the matrix will be to nodes in the same level. The space required for this matrix is $O(q|A|)$[6].

The two data structures are used simultaneously, in a conjunctive manner, to achieve the pruning that is needed. This will be illustrated in more detail in the next section.

[6]This can be easily extended to include all nodes of different levels.

9.3.4 Applying the CBS

The pseudo code, as shown in Algorithm 9.2, illustrates how the CBS can be applied to approximate string matching, namely the CBS-LLP-based scheme. It also shows how the proposed data structures, presented in the previous section, can be used to facilitate the calculations. The LLP helps to maintain the list of the children of the best nodes and to achieve the pruning expediently. Moving the nodes in the same lists will not affect the trie order at all, but helps us to effectively maintain information about which nodes are to be processed and which are to be discarded. The QM also helps us to maintain the queues.

9.4 Experimental Results

To investigate the power of our new method with respect to computation we conducted various experiments on three benchmark dictionaries. The results obtained were very significant with respect to the gain in the number of computations needed to get the best estimate X^+. As before, the computations were comprised of the number of addition and minimization operations needed. The CBS-LLP-based scheme was compared with the acclaimed DFS-trie-based work for approximate matching [137] when the maximum number of errors was not known *a priori*.

The three benchmark data sets that were used in our experiments were those used in the previous chapters, namely Eng[7], Dict[8], and the Webster's Unabridged Dictionary. Each data set was divided into two parts: a *dictionary* and the corresponding *noisy file*. The dictionary was, as indicated earlier, composed of the words or sequences that had to be stored in the trie, and the noisy files consisted of the strings which were searched for in the corresponding dictionary. For all dictionaries we removed words of length smaller than or equal to 4.

Three sets of corresponding noisy files were created using the noise generator model described in [121], and in each case, the files were created for three specific error characteristics, where the latter means the number of errors per word. The three error values tested were for 1, 2, and 3

[7]This file is available at `www.scs.carleton.ca/~oommen/papers/WordWldn.txt`.
[8]The actual dictionary can be downloaded from `http://www.cs.princeton.edu/~rs/strings/dictwords`.

Algorithm 9.2 CBS-LLP-based Scheme

Input:

a. The dictionary, H, represented as a Linked Lists of Prefixes, LLP, with alphabet, A.

b. The matrix QM to maintain the priority queues.

c. The width of the beam considered for each alphabet q.

d. The noisy word, Y, for which we need to find the nearest neighbor.

Output: X^+, the nearest neighbor string in H to Y.

Method:

1: Start from the first level in LLP, which contains the root of the trie.
2: Initialize *minnode* to null, which stores the node representing the string that is nearest neighbor to Y so far.
3: Initialize *childq* to 1, which is the number of nodes to be considered in the current level.
4: **while** *childq* $\neq 0$ **do**
5: Initialize QM to be empty.
6: **for** for each node n in the *childq* nodes of the current level **do**
7: Get the character c represented by node n.
8: Calculate the edit distances $D(X, Y)$, for the string represented by node n and using the column information stored in the parent of n.
9: Add n to $QM[c]$ if it is one of the best q nodes already in the list according to the distance value. If the distance value of n is equal to one of the q nodes, one of the solutions we use is to extend the $QM[c]$ to include n.
10: **if** n is an accept node, i.e., a word in the dictionary **then**
11: Compare $D(X, Y)$ with the minimum found so far and if it has lower edit distance value, store n in *minnode*.
12: **end if**
13: **end for**
14: Move all the children of the best nodes in QM to the beginning of the next level in LLP, if any, and store their number in *childq*.
15: Increment current level to the next level.
16: **end while**
17: **return** the string X^+, corresponding to the path from *minnode* to the root.
18: **End Algorithm CBS-LLP-based Scheme**

referred to by the three sets SA, SB, and SC respectively.

Each of the three sets, SA, SB, and SC was generated using the noise generator model described in [121]. We assumed that the number of insertions was geometrically distributed with parameter $\beta = 0.7$. The conditional probability of inserting any character $a \in A$ given that an insertion occurred, was assigned the value $1/26$ and the probability of deletion was $1/20$. The table of probabilities for substitution (typically called the confusion matrix) was based on the proximity of character keys on the standard QWERTY keyboard and is given in [121][9].

The two algorithms, the DFS-trie-based and our algorithm, CBS-LLP-based, were tested with the three sets of noisy words for each of the three dictionaries. We report the results obtained in terms of the number of computations (additions and minimizations) and the accuracy for the three sets in Tables 9.1, 9.2, and 9.3 for each of the three dictionaries respectively. The calculations were done on a Pentium V processor, 3.2 GHZ. Figure 9.2 shows a graphical representation of the results. The figures compare both time and accuracy. The numbers are shown in millions. The results show the significant benefit of the CBS-based method with respect to the number of computations, while maintaining excellent accuracy. For example consider the *Webster* dictionary, for the SA set, and $q = 100$: the number of operations for DFS-trie-based is 1,099,279, and for the CBS-LLP-based method is 271,188 representing a savings of 75.3%, and a loss of accuracy of only 0.5%. For the *Dict* dictionary, for the SA set, $q = 100$, the number of operations for the DFS-trie-based is 72,115, and for the CBS-LLP-based method is 44,254: and this represents a savings of 36.6%, and a loss of accuracy of only 0.2%. When $q = 50$, the number of operations for the CBS-LLP-based method is 21,366, representing a savings of 70.4%, with a loss of accuracy of only 0.5%. There is always a trade-off between time and accuracy but the loss of accuracy here is negligible compared to the "phenomenal" savings in time.

To show the benefits of the CBS over the BS, we show the results when applying the BS for the *dict* dictionary in Table 9.4 and Figure 9.3, when the approximately equivalent width (number of nodes taken per level) is considered for the BS. The width is considered approximately equal when we approximately equate the number of addition operations. The figure shows only the result when applied to set SA, as these results are representative of the other sets. From the

[9]It can be downloaded from `www.scs.carleton.ca/~oommen/papers/QWERTY.doc`.

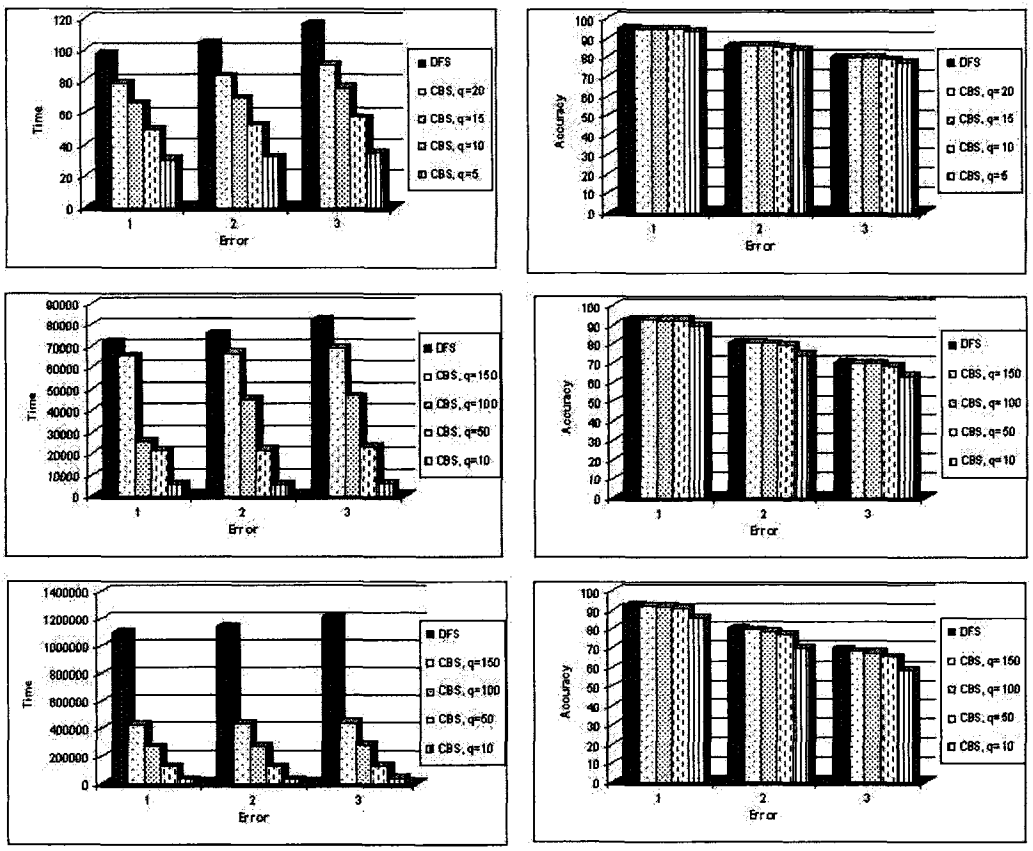


Figure 9.2: The results for comparing the **CBS-LLP**-based method with the **DFS-trie**-based method for (top) the **Eng** dictionary, (middle) the **Dict** dictionary, and (bottom) the **Webster** dictionary. The time is represented by total number of operations in millions.

q	Oper	SA		SB		SC	
		DFS	CBS	DFS	CBS	DFS	CBS
5	Add	58.1	16.2	62.4	17.4	69.6	19.3
	Min	39.6	14.2	42.5	14.9	47.3	16.2
	Total	97.7	30.4	104.9	32.3	116.9	35.5
	Sav	**68.8**		**69.2**		**69.6**	
	Acc	95.1	93.5	85.8	83.8	80.0	77.4
10	Add	58.1	25.2	62.4	27.0	69.6	30.0
	Min	39.6	24.6	42.5	25.7	47.3	27.7
	Total	97.7	49.8	104.9	52.7	116.9	57.7
	Sav	**49.0**		**49.7**		**50.6**	
	Acc	95.1	94.6	85.8	85.4	80.0	79.0
15	Add	58.1	32.4	62.4	34.6	69.6	38.5
	Min	39.6	34.0	42.5	35.4	47.3	37.9
	Total	97.7	66.4	104.9	70.0	116.9	76.4
	Sav	**32.0**		**33.2**		**34.6**	
	Acc	95.1	94.8	85.8	86.0	80.0	79.8
20	Add	58.1	37.7	62.4	40.4	69.6	44.8
	Min	39.6	41.6	42.5	43.3	47.3	46.2
	Total	97.7	79.3	104.9	83.7	116.9	91.0
	Sav	**18.83**		**20.2**		**22.2**	
	Acc	95.1	94.8	85.8	86.0	80.0	79.9

Table 9.1: The experimental results obtained from each of the three sets for the **Eng** dictionary. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the **CBS-LLP**-based method obtains over the **DFS-trie**-based method.

figures the reader will observe the significant decrease in the operations needed when the same ordering technique is used for arranging all the priority queues. The results are shown only for $q = 10$ and $q = 50$, because if we increase q it will yield poor results for the BS which is much worse than when $q = 50$. For example, the number of operations for the BS-LLP-based method is 256,970, and for the CBS-trie-based method is 21,366, representing a savings of 91.7% in the total number of operations with accuracy 92.3%. In this case, the number of operations for the BS method is much more than the 72,115 operations of the DFS-trie-based method. From Table 9.2, we see that we can increase q in the CBS-LLP-based method to 100 and get an accuracy of 92.6 with savings of 36.6% in the total number of operations with respect to DFS-trie-based method. This is not feasible by merely applying the BS method.

q	Oper	SA		SB		SC	
		DFS	CBS	DFS	CBS	DFS	CBS
10	Add	42909	2619	45337	2756	49319	2995
	Min	29206	2854	30825	2937	33479	3092
	Total	72115	5473	76162	5693	82798	6087
	Sav	**92.4**		**92.5**		**92.6**	
	Acc	92.8	89.5	80.5	74.4	70.2	63.6
50	Add	42909	7195	45337	7579	49319	8228
	Min	29206	14171	30825	14402	33479	14823
	Total	72115	21366	76162	21981	82798	23051
	Sav	**70.4**		**71.1**		**72.6**	
	Acc	92.8	92.3	80.5	79.4	70.2	68.8
100	Add	42909	11447	45337	12068	49319	13101
	Min	29206	32807	30825	33159	33479	33786
	Total	72115	44254	76162	45227	82798	46887
	Sav	**36.6**		**40.6**		**43.4**	
	Acc	92.8	92.6	80.5	80.2	70.2	69.8
150	Add	42909	14526	45337	15319	49319	16637
	Min	29206	51276	30825	51753	33479	52591
	Total	72115	65802	76162	67072	82798	69228
	Sav	**8.8**		**11.9**		**16.4**	
	Acc	92.8	92.7	80.5	80.4	70.2	70.0

Table 9.2: The experimental results obtained from each of the three sets for the **Dict** dictionary. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the **CBS-LLP**-based method obtains over the **DFS-trie**-based method.

9.5 Optimizing Computations when Changing the Error Model

As we see from the results in the previous section, by marginally sacrificing a small accuracy value for the general error model (by less than 1%) a noticeable improvement can be obtained with respect to time.

However, by permitting an error model that increases the errors as the length of the word increases (i.e., the errors do not appear at the very beginning of the word), an improvement of more than 95% in the number of operations can be obtained, which is, in our opinion, absolutely

q	Oper	SA		SB		SC	
		DFS	CBS	DFS	CBS	DFS	CBS
10	Add	654692	16090	681001	16605	726513	17739
	Min	444587	17153	462126	17444	492467	18215
	Total	1099279	33243	1143127	34049	1218980	35954
	Sav	**96.9**		**97.0**		**97.1**	
	Acc	92.5	86.2	80.3	69.7	69.5	58.3
50	Add	654692	43395	681001	44890	726513	47703
	Min	444587	83549	462126	84212	492467	85772
	Total	1099279	126944	1143127	129102	1218980	133475
	Sav	**88.5**		**88.7**		**89.1**	
	Acc	92.5	90.8	80.3	77.1	69.5	65.5
100	Add	654692	70603	681001	73183	726513	77876
	Min	444587	200585	462126	201764	492467	204401
	Total	1099279	271188	1143127	274947	1218980	28227
	Sav	**75.3**		**75.9**		**76.8**	
	Acc	92.5	91.9	80.3	78.9	69.5	67.57
150	Add	654692	92673	681001	96111	726513	102289
	Min	444587	337323	462126	338184	492467	342062
	Total	1099279	429996	1143127	434295	1218980	444351
	Sav	**60.88**		**62.0**		**63.5**	
	Acc	92.5	92.2	80.3	79.6	69.5	68.4

Table 9.3: The experimental results obtained from each of the three sets for the **Webster** dictionary. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the **CBS-LLP**-based method obtains over the **DFS-trie**-based method.

amazing. This is because if errors are less likely to appear at the very beginning of the word, the quality of pruning, with respect to accuracy, will be more efficient at the upper levels of the tree. Thus we can utilize a small value for q, the width of the beam, and as a result achieve more pruning. All our claims have been verified experimentally as shown in Table 9.5 and in Figure 9.4. The results are shown for $q = 5$, (which is a very small width) demonstrating very high accuracy. For example, for the set SA, the number of operations for the DFS-LLP-based method is 74,167, and for the CBS-trie-based method is just 3,374, representing a savings of 95.5%. This has obviously great benefits if the noisy words received are not noisy at the beginning, in which case we still need to apply approximate string matching techniques. Even here we would like to make use of the exact part at the very beginning of the word, a part which is variant from one

Oper	q = 10		q = 50	
	BS	CBS	BS	CBS
Add	2840	2619	7914	7195
Min	25310	2854	249056	14171
Total	28150	5473	256970	21366
Sav	**80.6**		**91.7**	
Acc	89.9	89.5	92.5	92.3

Table 9.4: The experimental results obtained from each of the three sets for the **Dict** dictionary. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the **CBS-LLP**-based method obtains over the **BS-LLP**-based method when applied to set **SA**.

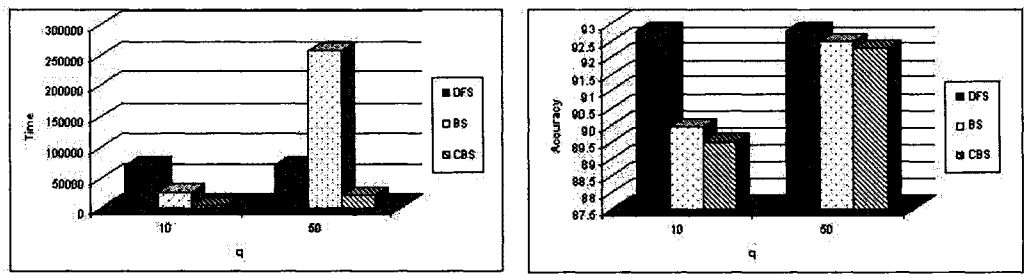


Figure 9.3: The results for comparing the **CBS-LLP**-based method with the **BS-LLP**-based method for the **Dict** dictionary when applied to set **SA**. The time is represented by total number of operations in millions.

word to another, and where we cannot use partitioning to process the noisy word to indirectly apply an exact match strategy.

9.6 Conclusion

In this Chapter we have presented a feasible, fast, AI-based solution for the approximate string matching problem, when the whole dictionary is considered simultaneously.

First, we proposed a new AI-based search called the Clustered Beam Search (CBS) that can be considered an enhancement of the Beam Search (BS) used in AI. The new scheme can be used to search a graph more efficiently with respect to the number of operations needed.

Oper	SA		SB		SC	
	DFS	CBS	DFS	CBS	DFS	CBS
Add	44140	1756	45629	1832	47711	1913
Min	30027	1618	31219	1672	32407	1730
Total	74167	3374	77148	3504	80118	3643
Sav	**95.5**		**95.5**		**95.5**	
Acc	91.7	92.1	91.4	91.7	89.3	89.5

Table 9.5: The experimental results obtained from each of the three sets for the **dict** dictionary. The results are given in terms of the number of operations needed in millions, and the corresponding accuracy. The results also show the percentage of savings (in bold) in the total number of operations that the **CBS-LLP**-based method obtains over the **DFS-trie**-based method when the optimized **error model** is used and $q = 5$.

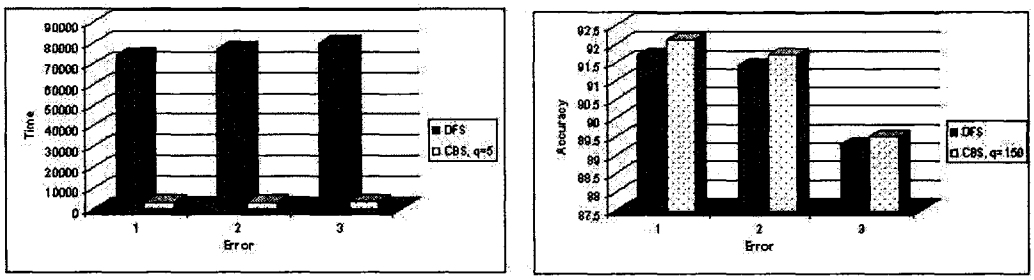


Figure 9.4: The results for comparing the **CBS-LLP**-based method with the **DFS-trie**-based method for the **Dict** dictionary when the optimized **error model** is used and $q = 5$. The time is represented by total number of operations in millions.

The CBS was applied to dictionary-based approximate string matching, where the dictionary is stored using a trie.

Secondly, we proposed a strategy that can be used in conjunction with the LLP representation of the trie explained earlier. The new implementation strategy helps prune the search space more efficiently and dramatically decreases the number of operations needed to get the nearest neighbor to Y.

Thirdly, the CBS-LLP-based approximate string matching has been compared with the benchmark Depth-First trie-based technique proposed by Shang *et.al.* [137] using large and small dictionaries. The results demonstrate a significant improvement with respect to the number of operations needed (up to 75%) while keeping the accuracy comparable to the optimum.

It has also been compared with the BS-LLP-based method to show the benefits of CBS over BS. The results sometimes show improvements of more than 90% when q equaled 50.

Finally the new scheme, CBS-LLP-based, has also been tested for string generation using a new error model, where the error predominantly appears at the end of the string. The results demonstrate a marked improvement of more than 95%, while keeping the accuracy the same.

Chapter 10

Trie-Based Dynamic Matrix Optimization

10.1 Introduction

All SPR problems that deal with sequences encounter the situation where the distance computation has to accomplished. Typically, the distance involves two strings, namely the noisy one to be recognized and a potential exact representation. Such computations are straightforward, and many quite similar algorithms have been proposed to handle this [105]. But when the computations involve a noisy string and a *dictionary* in its entirety, the dynamic programming equations become cumbersome.

In this Chapter, we shall demonstrate that even without utilizing any approximate heuristic, the computation of the true distance can be significantly improved at the cost of maintaining some additional distance information as a "secondary" matrix. Indeed, by not merely maintaining a *string-versus-string* matrix (refer to Chapter 5, Section 5.3 for more details), but a pair of *string-versus-dictionary* matrices, the advantages that we get on three benchmarks dictionaries is up to 48%. The primary contribution of this Chapter, which involves such optimized distance computations, is, to our knowledge, completely novel.

The advantage that we glean is by, first of all, modelling the dictionary as a trie. As we know, the trie is a data structure that offers search costs that are independent of the document size. Tries also combine prefixes together, and so by using tries in approximate string matching [76], [137], if Z and W are two strings stored in the trie and share common prefix, we can utilize the information obtained in the process of evaluating any one $D(Z, Y)$, to compute any other $D(W, Y)$.

We attempt to use the same data structure, the trie, for storing the strings in the dictionary so as to take advantage of the compact calculations for the distance matrix, by utilizing the common paths for the common prefixes as will be explained presently.

The age-old DP algorithm (traditionally attributed to Wagner and Fisher [155]) has a quadratic complexity. The question of whether this algorithm is optimal was solved by Wong and Chandra [161] in 1976. They showed that the Wagner and Fisher algorithm was optimal for the infinite alphabet case. The question then arose of how the distance computation could be enhanced if the alphabet was finite and known *a priori*. To respond to this, Masek and Paterson [95] applied the "four Russians" algorithm by Arlazarov *et. al.* [15] to Wagner and Fisher's string-distance procedure to obtain an $O(n^2/\log n)$ time algorithm for the *finite* alphabet case. Consider now the scenario when we are not dealing with individual words but with computing the distance between a single word and a finite set of words given as a dictionary. Of course a naive strategy will be to individually compute the distance between the given string and every word in the dictionary. As opposed to that, if the dictionary is treated in its entirety, it appears as if the lower bound (for the case when the entire string Y is not known *a priori*) is the DFS bound due to Shang and Merrettal [137]. However, we propose a strategy akin to the finite alphabet case - one which considers the information in Y. We do this by specifically optimizing the computations for the symbols of Y in A, and by not repeating the computations for the symbols of Y which are not in A. We are not aware of any research which has performed such an optimization.

We assume that we are using the 0/1-based Levenshtein distance, $D(X, Y)$, between two strings, X and Y, where the primary dynamic programming rule used in computing the inter-string distance $D(X, Y)$ is:

$$D(x_1 \ldots x_i,\ y_1 \ldots y_j) = \min\ [\ \{D(x_1 \ldots x_{i-1},\ y_1 \ldots y_{j-1}) + d(x_i, y_j)\},$$
$$\{D(x_1 \ldots x_i,\ y_1 \ldots y_{j-1}) + d(\lambda, y_j)\},$$
$$\{D(x_1 \ldots x_{i-1},\ y_1 \ldots y_j) + d(x_i, \lambda)\}]. \qquad (10.1)$$

To optimize this, unlike all previous reported techniques, we introduce a new technique with its associated operations, namely the so-called *Opt-DFS-trie*, for doing the dynamic matrix calculations. The new technique utilizes two DP matrices and is based on the concept of extracting the maximum information of the characters in the noisy string Y. It thus examines the specific character of the corresponding column to be calculated with respect to the characters of Y, and optimizes on the fact that we are doing calculations against the whole dictionary, H, stored in a trie, T. The experimental results presented later show improvements obtained by these methods to be up to 48% for the number of operations (minimizations and additions) with small and large benchmark dictionaries. This high improvement is at the expense of just storing a "secondary" dynamic programming matrix.

The Chapter is organized as follows: Section 10.2 explains in detail the proposed technique. It starts with describing the fundamental idea, and then proceeds to explain the data structures that will be used, and finally, the algorithm and the proof of its correctness. Section 10.3 illustrates the algorithm when the maximum number of errors K, is known *a priori*. Section 10.4 provides an example to illustrate the idea. Section 10.5 presents the experiments done and provides the results that demonstrate the benefits of the new method. Section 10.6 concludes the Chapter.

10.2 Optimizing the DP Matrix Computations

We intend to propose a new optimization scheme for the DP matrix computations when using binary Levenshtein distances, i.e., when the error values are of a 0/1 form. We shall show that

this can lead to more optimization than the results obtained in Chapter 8. Indeed, we shall demonstrate that further optimization can be done when using the DP matrix for calculating the edit distance for a dictionary stored in a trie [137], where all the edit distances are calculated simultaneously for all the words in the dictionary. All of that will be described presently in the following subsections.

10.2.1 Main idea

Let Υ be the set of distinct characters that are found in the noisy word Y. This set can be computed in $O(|Y|)$ time. The nodes in the trie are processed one by one. For the sake of notation, let us assume that each node, n, corresponds to a character, c, in a certain prefix of a word, X, in the dictionary, H. This node has a parent p that corresponds to character c_p. If we traverse the trie in a DFS manner, the calculations will be done in a column-by-column manner in the DP matrix (see Figure 6.3). In this case, the calculations for columns that correspond to the children at the same level of the trie, and for the same parent, can be further optimized. This is because if we are given a prefix X_{i-1}, we know that the next column to be calculated has an index i, and consequently, it turns out that there are a maximum of $|\Upsilon| + 1$ possibilities for calculating this column. This is a consequence of the fact that the only characters that can change the calculations in the columns are the characters that are in Y. This result is summarized in the following theorem.

Theorem 10.1. *Let Y be the noisy word of distinct characters, Υ, that is being compared against a dictionary, H, stored in a trie, T, traversed in a DFS manner. Also, let the individual edit symbol costs be of a 0/1 form. If n_1 and n_2 are nodes at same level i of T associated with characters c_1 and c_2 respectively, such that they are the children of the same parent p, then, the columns that correspond to n_1 and n_2 will be identical if both c_1 and c_2 are not in Υ.*

Proof. Since both n_1 and n_2 share the same parent p, they also share the same prefix and the same previous column information that corresponds to p. Given that c_1 and c_2 are not in Υ, and the fact that costs are of 0/1 form, both characters will have the same inter-symbol costs with respect to the characters, Υ, in Y. So according to the dynamic programming calculations

in Equation (10.1), they will both lead to the same edit distance costs for every entry in the columns and so both columns will be equal. The result follows. □

By virtue of Theorem 10.1, it is clear that we need to calculate only one column, for all nodes of the same parent p which have characters that are not in Υ. This is because of the fact that as per Theorem 10.1, all the characters that are not in Y lead to identical calculations and identical stored values.

The reader should observe that this observation will lead to a great savings in the computation time, as only a maximum of $|\Upsilon| + 1$ columns will have to be calculated for each parent instead of $|A|$ columns which is the alphabet size. Consequently, this will lead to significant benefits when $|A| >> \Upsilon$, and also when the fanout of the nodes is high. This will also lead to a *horizontal* cut-off of the calculations instead of merely the depth-based cutoffs applied in [19] and [137]. This savings will be at the expense of a small additional storage as explained below. To utilize the above information, we will need to store the common "column" information for the characters which are not in Y and which are at the same level and for the same parent. This entails the maintenance of another matrix which will be needed to store the common column information during the processing. As soon as all the children of a parent node are calculated we intend to be able to discard the corresponding common column and replace it with another common column for another parent at the same level, as well as for other paths of calculations. In this way, we will be able to traverse the trie in a depth-first manner and save these column calculations along the path associated with the parent. The maximum number of columns to be stored will then be equal to the maximum length of a word in the dictionary. We shall now explain how each of these steps is implemented.

10.2.2 Data-structures

To explain our strategy, we will first explain the following data-structures which will be needed to do the calculations:

1. *The dictionary*: The dictionary, H, will be stored using the *trie* data structure. An

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	u	r	s	t	w	v	x	y
1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0

Figure 10.1: An example for the tuple representation, V, of Y when the alphabet, A, is the English alphabet and Y equals "farm".

example of this is shown in Figure 8.1, where the nodes of the trie are represented as an array, list, or binary search tree [45]. In this paper, for ease of explanation, we use the array representation.

2. *The vector V*: During the calculation we will need to test whether the character associated with any particular node of a path is in Υ or not. To render this possible, we will represent Y in a *tuple* representation, so as to perform this test in just a single operation. In this case, Y will be represented as a vector, V, of size $|A|$, the alphabet size, where $V[c] = 1$ if c is in Y, and $V[c] = 0$ otherwise. Figure 10.1 shows an example for Y equals "farm".

3. *The two DP matrices*: We shall maintain two DP matrices to achieve the edit distance calculations along a certain path. The first DP matrix, B, is the same as the one traditionally used in [137] for storing columns for *edit distances* along the path, but it stores only those columns that correspond to characters in Υ. The second DP matrix, C, is the same as the one traditionally used in [137], but is required for storing the *common* columns for edit distance calculations along the path, so that they can be effectively reused. Let a_i to be the last symbol in the Prefix X_i, we can formally define B and C as follows[1]:

$$B(i,j) = D(X_i, Y_j)|_{a_i \in \Upsilon} \tag{10.2}$$

$$C(i,j) = D(X_i, Y_j)|_{a_i \notin \Upsilon} \tag{10.3}$$

where i and j correspond to the positions of the characters in X and Y respectively, X_i and Y_i correspond to the prefixes of lengths i and j respectively, and the notation "|" corresponds to the subsequent "given" condition being satisfied.

[1]For any matrix, say H, $H(i,j)$ refers to a single entry in matrix H with indices i and j, whereas $H(i)$ refers to column i in the matrix H

10.2.3 Algorithm and correctness

In this Section, we will present the algorithm for calculating the edit distance when it is optimized with respect to Υ, the characters in Y. To calculate the edit distance for node n with parent p that corresponds to the character c and the preceding character c_p respectively, and prefixes X_i and X_{i-1} respectively, we have the following two cases:

1: $c \in \Upsilon$: In this case, the corresponding column for the node n being considered has to be calculated again using the column previously calculated for p and the corresponding dynamic equations. The results will be stored in column $B(i)$. Again, we have two possible cases for calculating $B(i)$ when using the column information stored for the parent p:

- $c_p \in \Upsilon$: In this case, the column that corresponds to the parent, p, is stored in $B(i-1)$.
- $c_p \notin \Upsilon$: In this case, the column that corresponds to the parent, p, is stored in $C(i-1)$ (as opposed to being stored in $B(i-1)$).

2: $c \notin \Upsilon$: The corresponding column for the node n being considered will be stored in $C(i)$. Again, we have two possible cases for $C(i)$:

- $C(i)$ is *active*: This means that this column has already been calculated and that the edit distance between the prefix X_i and the noisy word Y has already been completed and stored. This scenario implies that we have already, "at hand", the optimization required for using the previously calculated information.
- $C(i)$ is not *active*: This means that this column has to be calculated for the first time using the column information that corresponds to p considering the same two cases in which this column is stored in either $B(i-1)$ or $C(i-1)$ depending on whether $c_p \in \Upsilon$ or not, respectively.

The dynamic equations that will be used for all the cases presented above and their correctness are summarized in the following theorem.

Theorem 10.2. *Let Y be the noisy word of distinct characters, Υ, that is being compared against a dictionary, H, stored in a trie, T, that is traversed in a DFS manner. Consider an arbitrary node n, at level i of T which represents the character c. Then, the edit distance calculated at n between Y and the prefix X_i, with last character c (represented by n), can be effected as follows. The results of the edit distance calculations will be stored in either column $B(i)$ or column $C(i)$ depending on whether $c \in \Upsilon$ or not respectively, and can be calculated using the following dynamic equations:*

***1**: $c \in \Upsilon$: The results will be stored in $B(i)$ as per the following two cases:*

- $c_p \in \Upsilon$:

$$B(i,j) = \min \; [\; \{B(i-1,j-1) + d(x_i, y_j)\}, \; \{B(i,j-1) + d(\lambda, y_j)\}, \; \{B(i-1,j) + d(x_i, \lambda)\}]. \tag{10.4}$$

- $c_p \notin \Upsilon$:

$$B(i,j) = \min \; [\; \{C(i-1,j-1) + d(x_i, y_j)\}, \; \{B(i,j-1) + d(\lambda, y_j)\}, \; \{C(i-1,j) + d(x_i, \lambda)\}]. \tag{10.5}$$

***2**: $c \notin \Upsilon$: The corresponding column for n will be stored in $C(i)$ as per the following two cases:*

- *$C(i)$ is active: No dynamic equations will be needed here.*
- *$C(i)$ is not active: $C(i)$ will be calculated according to the following two cases:*
  - $c_p \in \Upsilon$:

$$C(i,j) = \min \; [\; \{B(i-1,j-1) + d(x_i, y_j)\}, \; \{C(i,j-1) + d(\lambda, y_j)\},$$

$$\{B(i-1,j)+d(x_i,\lambda)\}]. \tag{10.6}$$

* $c_p \notin \Upsilon$:

$$C(i,j) = \min \; [\; \{C(i-1,j-1)+d(x_i,y_j)\}, \; \{C(i,j-1)+d(\lambda,y_j)\}, \; \{C(i-1,j)+d(x_i,\lambda)\}]. \tag{10.7}$$

Proof. From the above equations, we see that the proposed edit distance calculations effectively use the same dynamic equation as in Equation (10.1), but utilize quantities stored in different *matrices* depending on the different cases discussed above. The proof of the correctness of all these cases is based on the fact that the basic dynamic equations of the Levenshtein distance is correct [155] (Equation 10.1) and on Theorem 10.1. We will need to prove that the columns calculated along any path using B and C, are exactly the same as the columns calculated along the same path using only a single DP matrix, D, and the DP Equation (10.1). Since at each step, when we consider the computation of $B(i,j)$ or $C(i,j)$, we consider only the previous column (that corresponds to the parent node), the proof can be easily done by induction. Let us first rewrite the DP Equation (10.1) using the the DP matrix D.

$$D(i,j) = \min \; [\; \{t_1 = D(i-1,j-1)+d(x_i,y_j)\}, \; \{t_2 = D(i,j-1)+d(\lambda,y_j)\}, \; \{t_3 = D(i-1,j)+d(x_i,\lambda)\}]. \tag{10.8}$$

Basic step:

The basic case must consider three levels, namely for the nodes in the trie which correspond to prefixes of lengths 0, 1, and 2.

The case for $i = 0$, which corresponds to the evaluation of $D(0)$, concerns the computation

for the initial column. Thus, the edit distance stored at each entry will basically correspond to the sequence of deletion operations for the prefix of Y that corresponds to this entry. This is the *true* edit distance column $D(0)$, and in the interest of continuity, we store this column in $C(0)$, and make it always active[2].

For $i = 1$, we consider the processing of the children of the root node that will always use $C(0]$ as a previous column. In this case, terms t_1 and t_3 of Equation (10.8) will always use matrix C. We have now two cases for each child that corresponds to a character c:

- $c \in \Upsilon$: The results will be stored in column $B(1)$, and so the term t_2 in Equation (10.8) will use matrix B, and the equation reduces to Equation (10.5).
- $c \notin \Upsilon$: In this case the column should either have been already calculated, or it will be calculated and stored in $C(1)$. These two scenarios are easily flagged by examining whether $C(1)$ is active or not.
  - $C(1)$ is not active: The term t_2 in Equation (10.8) will use matrix B, and the equation reduces to Equation (10.7).
  - $C(1)$ is active: This case occurs only if $C(1)$ is previously calculated for another child whose character is not in Υ. Thus, according to Theorem 10.1, this active column should be equal to the current column under consideration, and in this case $C(1)$ will also be the true edit distance that corresponds to $D(1)$.

The argument for case when $i = 2$, is analogous to the case when $i = 1$. In this scenario, we apply the result with respect to each parent separately, effectively considering each parent to be the root. It is exactly the same when the column for the parent node is stored in $C(1)$, since $c_p \notin \Upsilon$. The difference however is when the parent column is stored in $B(1)$ because $c_p \in \Upsilon$. We again have the same two cases:

- $c \in \Upsilon$: The DP Equation (10.8) will be reduced to Equations (10.4).

[2]Strictly speaking, the same entry should also be made in $B(0)$; but since we can assume that $\lambda \notin \Upsilon$, we store it merely in $C(0)$ to minimize computations. This also satisfies the base condition for Equation (10.7).

- $c \notin \Upsilon$: If $C(2)$ is not active (i.e., it has not been previously calculated), the DP Equation (10.8) will also be reduced to Equations (10.6). If $C(2)$ is active (which means it has been calculated before for another child whose character is not in Υ), then according to Theorem 10.1, this active column should be equal to the current column under consideration and in this case $C(2)$ will also be the true edit distance that corresponds to $D(2)$.

Inductive hypothesis:

At level i, all the columns calculated for the nodes at that level are the true edit distance and correspond to column $D(i)$. They are stored in the corresponding matrices $B(i)$ and $C(i)$ depending on whether their characters are in Υ or not respectively.

Inductive step:

We need to prove that the result holds for $i + 1$. Indeed, if the argument is expanded for as before for all the possible cases, we see that we encounter scenarios similar to the case of $i = 2$. Again, since we consider the theorem for each parent separately, each parent can be conceptually considered to be the root of a conceptual tree. The parent column should be stored in either $B(i)$ or $C(i)$, where, by the inductive hypothesis this column is the true edit distance and corresponds to column $D(i)$. If the parent column is stored in $C(i)$, then we will have two cases for each child that corresponds to character c:

- $c \in \Upsilon$: The results will be stored in column $B(i+1)$ and so the term t_2 in Equation (10.8) will use matrix B, and the equation reduces to Equation (10.5).
- $c \notin \Upsilon$: In this case the column should either have been already calculated, or it will be calculated and stored in $C(i + 1)$. These two scenarios are easily flagged by examining whether $C(i + 1)$ is active or not.
  - $C(i + 1)$ is not active: The term t_2 in Equation (10.8) will use matrix B, and the equation reduces to Equation (10.7).
  - $C(i + 1)$ is active: This case occurs only if $C(i + 1)$ is previously calculated for another child whose character is not in Υ. Thus, according to Theorem 10.1, this

active column should be equal to the current column under consideration, and in this case $C(i+1)$ will also be the true edit distance that corresponds to $D(i+1)$.

When the parent column is stored in $B(i)$ because $c_p \in \Upsilon$, we again have the same two cases:

- $c \in \Upsilon$: The DP Equation (10.8) will be reduced to Equations (10.4).
- $c \notin \Upsilon$: If $C(i+1)$ is not active (i.e., it has not been previously calculated), the DP Equation (10.8) will also be reduced to Equations (10.6). If $C(i+1)$ is active (which means it has been calculated before for another child whose character is not in Υ), then according to Theorem 10.1, this active column should be equal to the current column under consideration and in this case $C(i+1)$ will also be the true edit distance that corresponds to $D(i+1)$.

The theorem follows.

□

In all these cases, to test whether a character is in Υ or not, we merely have to look up V using the character informaion stored at the node being processed. The pseudo-code for calculating the edit distance between a noisy word, Y, against the whole dictionary, H, that is stored in the trie, T, is given in Algorithms 10.1 and 10.2. Algorithm 10.1 is intended to show the trie traversal, and Algorithm 10.2 presents the edit-distance calculations required for a node of the trie.

10.3 Optimization when maximum number of errors K is known

When the maximum number of errors, K, is known *a priori*, the Ukkonen cutoff can be used to terminate unsuccessful searches very early, as soon as the difference exceeds k. Chang and Lawler

Algorithm 10.1 Optimized-DFS-Trie

Input: A Dictionary stored as a Trie T. N' is the length of the prefix processed so far in the dictionary, and B and C are the Edit-distance matrix and the Common-distance matrix respectively, used for calculating the edit distances column by column.
Output: The candidate correct string X^+.
Method:

1: Create the V vector for Y.
2: $N = 0$
3: start from the root $tn = root$
4: Initialize the first column of C and B.
5: Make the first column of C active.
6: **while** tn is not NULL **do**
7: **if** there are no more children for tn **then**
8: **Make column** $C[N'+1]$ **inactive.**
9: Set t_n to be the parent node.
10: Decrement N'.
11: Go to line 6.
12: **else**
13: Make tn point to this child.
14: Increment N'.
15: **end if**
16: ed = edit_distance(N',$V[c]$,$V[c_p]$)
17: **if** tn is an accept node **then**
18: Compare ed with that of the best word found so far, and take the string that corresponds to tn if it is better, and save it in X^+.
19: **end if**
20: **end while**
21: **Return** X^+ = the string with minimum ed.
22: **End Algorithm Optimized-DFS-Trie**

Algorithm 10.2 edit_distance(i, *common*, *commonp*)

Input: i is the column being processed, M the length of Y, and B and C are the Edit-distance matrix and the Common-distance matrix respectively, used for calculating the edit distances column by column. *common* is a flag to indicate if the current column is "common" or not. Similarly, *commonp* is a flag to see if the previous column is common or not.

Output: *ed* the edit distance value.

Method:

```
1: for j = 1 to M do
2:   if common = true then
3:     {Case 2: c is not in Υ}
4:     if C[i] is active then
5:       ed = C[i, M]
6:     else
7:       if commonp = true then
8:         Compute C[i, j] using Equation (10.7)
9:       end if
10:      if commonp = false then
11:        Compute C[i, j] using Equation (10.6)
12:      end if
13:    end if
14:  else
15:    {Case 1: c is in Υ}
16:    if commonp = true then
17:      Compute B[i, j] using Equation (10.5)
18:    end if
19:    if commonp = false then
20:      Compute B[i, j] using Equation (10.4)
21:    end if
22:  end if
23: end for
24: Return ed.
25: End edit_distance
```

[43] showed that Ukkonen's algorithm evaluated $O(k)$ columns, which implies that searching a trie will be done only down to depth $O(k)$. If the fanout of a trie is $\sum$, the trie method needs to evaluate only $O(k|\sum|^k)$ DP table entries. Ukkonen [152] proposed an algorithm to reduce the table evaluations. Please refer to Section 6.6.2 for details.

As a result of the Ukkonen cutoff, the authors of [137] presented an interesting observation, that if all the entries of a column are greater than k, no word with the same prefix can have a distance $\leq k$. They thus concluded that we can stop searching down the subtrie. This observation informs us that it is not necessary to evaluate every suffix in the trie, and thus many subtries will be bypassed. In the extreme case, the one which involves the exact search, all but one of the subtries are trimmed.

Further optimization was also given in Chapter 8 [19] for the DFS process, where we presented a new Branch and Bound pruning strategy (called LHBB) that could be applied to dictionary-based approximate string matching when the dictionary was stored as a trie. The new strategy attempted to look ahead at each node, c, before moving further, by merely evaluating a certain local criterion at c. The search algorithm according to this pruning strategy would not traverse inside the *subtrie*(c) unless there was a "hope" of determining a suitable string in it. In other words, as opposed to the reported trie-based methods [76], [137], the pruning was done *a priori* even before embarking on the edit distance computations. The results in [19] showed improvements of up to 30% with respect to the number of operations.

The currently proposed optimized technique can be included to enhance the LHBB scheme. This will yield even further improvement as most nodes with a smaller fanout will be pruned and the benefits of the new method will be predominant.

The pseudo-code for calculating the edit distance between a noisy word, Y, against the whole dictionary, H, that is stored in the trie, T, is given in Algorithms 10.3. The "bolded" lines are the statements added the original LHBB scheme (in Chapter 8) so as to include the Opt-DFS-trie scheme. What we need to do is to deactivate the common columns for the children that will not be reached again, which results from pruning the trie at their parent node. The algorithm for calculating the edit distances is the same as in Algorithm 10.2.

Algorithm 10.3 Optimized-LHBB-Trie

Input: A Dictionary stored as a Trie T. N' is the length of the prefix processed so far in the dictionary, and B and C are the Edit-distance matrix and the Common-distance matrix respectively, used for calculating the edit distances column by column.
Output: The candidate correct string X^+.
Method:

1: Create the V vector for Y.
2: $N = 0$
3: start from the root $tn = root$
4: Initialize the first column of C and B.
5: Make the first column of C active.
6: **while** tn is not NULL **do**
7: **if** there are no more children for tn **then**
8: **Make column** $C[N'+1]$ **inactive.**
9: Set t_n to be the parent node.
10: Decrement N'.
11: Go to line 6.
12: **else**
13: Make tn points to this child.
14: Increment N'.
15: **end if**
{prune any extra search in the trie rooted at node tn, if LHBB condition is true}
16: **if** LHBB condition is true **then**
17: **make column** $C[N'+1]$ **inactive.**
18: return to the parent node.
19: decrement N'.
20: go to line 6.
21: **else**
22: ed = edit_distance(N',$V[c]$,$V[c_p]$)
{test the Ukkonen's cutoff condition and prune any extra search in the trie rooted at node tn if true}
23: **if** Ukkonen's cutoff condition is true **then**
24: **make column** $C[N'+1]$ **inactive.**
25: return to the parent node.
26: decrement N'.
27: go to line 6.
28: **else**
29: **if** tn is accept node and $ed < K$ **then**
30: Put the string represented by this node in the acceptable candidate list.
31: **end if**
32: **end if**
33: **end if**
34: **end while**
35: **Return** X^+ = the string with minimum ed for the strings in the candidate list.
36: **End Algorithm Optimized-LHBB-Trie**

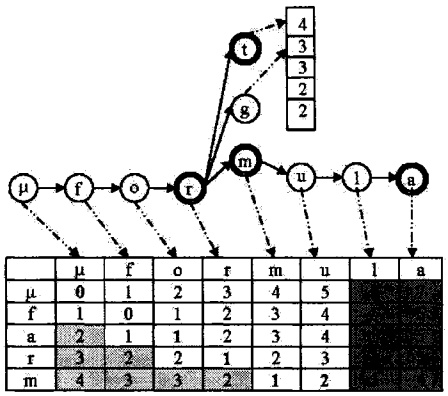
10.4 An example

Figure 10.2 shows an example for the calculations of the corresponding columns for the letters t and g when X is the string "formula" and Y is the string "farm". We see from the figure that the two columns are the same because they are both not in Y, and they both correspond to the same parent, r.

More illustration about the whole calculations between the noisy word "farm" and the whole dictionary stored in the trie of Figure 8.1, is given in Figure 10.3. The common shaded area in all the figures, represents the benefits obtained by utilizing the common prefix calculations for the trie. Path 1, node w represents Case 2 of the calculations, where the DP Equation (10.6) has to be used. Node a, observed after node w, represents the case where the DP Equation (10.5) has to be used. Path 2, is a good example to show the benefits of the new proposed method, where the calculations for node t will not be done again and the edit distance value will be directly retrieved from the active common column previously calculated in Path 1. Path 3, for node m, represents the case where the DP Equation (10.4) has to be used, which is the situation encountered for the traditional normal edit-distance calculations. Path 4, shows also the savings in the common prefix calculations for node m. Path 5, represents the same case as in Path 2 where the common column for g is still active and we can reuse the stored information. Node e after node g represents the case where the DP Equation (10.7) has to be used. For every path, the active row is shown to see how the common columns will be active or inactive after returning from the path of calculations to the nearest parent which still has more children to be processed. In Path 5, we return to the root and all the common columns are set to be inactive except the root, as the root node corresponds to the first column, assumed always to be active and common column.

10.5 Experimental Results

To investigate the power of our new method, namely Optimized Depth-First Search Trie-based method (Opt-DFS-trie), with respect to computation, we conducted various experiments. The



	μ	f	o	r	m	u	l	a
μ	0	1	2	3	4	5		
f	1	0	1	2	3	4		
a	2	1	1	2	3	4		
r	3	2	2	1	2	3		
m	4	3	3	2	1	2		

Figure 10.2: An example which shows a common column between the symbols t and g.

results obtained were remarkable with respect to the gain in the number of computations needed to get the best estimate X^+. By computations we mean the addition and minimization operations needed for calculating the edit distances. The Opt-DFS-trie scheme was compared with the Look-Ahead Branch and Bound (LHBB) described in [19], when the edit distance costs were of a 0/1 form.

We compare the new technique with the LHBB scheme, because, to our knowledge, this is the best reported scheme (till now) that can be applied when the DFS-trie-based is used for dictionary-based computations. The new scheme is actually applied "on top of" the LHBB scheme to demonstrate the added benefits obtained by applying the optimization for the DP matrix calculations.

We have also applied our strategy to the DFS-trie based method, without any cutoffs, to report the benefits obtained when the maximum number of errors is not known *a priori*, and hence no cutoffs can be used. In the latter case, the new technique shows a smaller improvement. This is because as we go lower in the trie, the fanout of the node decreases. As a result, the common distance calculations will not be reused and it is as if no further optimization can be obtained. This is because the optimization here comes from the fact that the common column information is reused over and over again. However, when cutoffs are used, most nodes with small fanout are already pruned during the calculations.

The three benchmark data sets used in our experiments were the same as those used in Section 7.4. Again, each data set was divided into two parts: a *dictionary* and the corresponding *noisy file*. The dictionary was composed of the words or sequences that had to be stored in the Trie. The noisy files consisted of the strings which were searched for in the corresponding dictionary.

Three sets of corresponding noisy files were created using the noise generator model described in [121], and in each case, the files were created for a specific error value. The three error values tested were for $K = 1,\ 2,\ and\ 3$, as is typical in the literature [113], [137].

The two methods, LHBB [19] and our new optimized scheme, Opt-DFS-trie when applied on the LHBB scheme, were tested for the three sets of noisy words. We report below a summary of the results obtained in terms of the number of computations (additions and minimizations) in millions.

In Tables 10.1, 10.2, and 10.3, the results show the significant benefits obtained by applying the optimized scheme over just the LHBB scheme, with up to 48% improvement. For example, for the *Dict* dictionary, when $K = 1$, the number of computations is 436 million and 844 million respectively, which represents an improvement of 48.34%. The improvement decreases as the number of errors increases, which can be expected because as K increases, we will have to go deeper in the trie, and a greater number of nodes with a smaller fanout will be included in the calculations. By studying the results we see that the improvements are quite prominent even for $K = 2\ and\ 3$. The improvement is more than 20%, which is considerable compared to what can be achieved by the state-of-the-art trie methods. Additionally, observe that the search is still bounded by the $O(K|\Sigma|^K)$ expected DP tables entries, because we use the trie to store the dictionary.

Table 10.4 shows the results when the Opt-DFS-trie is compared with, and applied to the original DFS-trie when the maximum number of errors, K is not known *a priori*. The results are shown when K equals to unity. The results for other values of K are the same because the entire trie will have to be traversed any way. For example, for the *Dict* dictionary, when $K = 1$, the number of computations is $75,132$ million and $86,418$ million respectively, which represents an improvement of 13.05%.

Operation	Eng		Dict		Webster	
	OPT	LH	OPT	LH	OPT	LH
Additions	1.9	3.5	186	390	1,116	2,224
Improvement	45.71		52.31		49.82	
Minimizations	2.6	4.2	250	454	1,444	2,552
Improvement	38.09		44.93		43.42	
Total	4.5	7.7	436	844	2,560	4,776
Improvement	**41.56**		**48.34**		**46.39**	

Table 10.1: The results obtained in terms of the number of operations (additions and minimizations) needed when the maximum number of errors $K = 1$ and known *a priori*, the costs are of a 0/1 form, and the new OPT-DFS-trie is applied over the **LHBB** scheme. The figures given are in Millions, and the *total* improvement obtained is in "bold" face.

Operation	Eng		Dict		Webster	
	OPT	LH	OPT	LH	OPT	LH
Additions	8.0	12.3	1,453	2,648	9,590	16,773
Improvement	34.96		45.13		42.82	
Minimizations	10.2	14.5	1,851	3,048	11,894	19,077
Improvement	29.66		39.27		37.65	
Total	18	26.8	5,696	5,696	21,484	35,850
Improvement	**32.08**		**41.99**		**40.07**	

Table 10.2: The results obtained in terms of the number of operations (additions and minimizations) needed when the maximum number of errors $K = 2$ and known *a priori*, the costs are of a 0/1 form, and the new OPT-DFS-trie is applied over the **LHBB** scheme. The figures given are in Millions, and the *total* improvement obtained is in "bold" face.

10.6 Conclusion

In this Chapter, we presented a new optimized scheme for the DP matrix calculations that can be applied to approximate string matching using tries. The optimized technique depends on the fact that we are processing the words of the dictionary simultaneously using the trie data structure, and that we can have common column calculations for the DP matrix along the children of the same parent, where these children are actually different prefixes in the dictionary. This principle led to a horizontal cut-off of the calculations instead of merely the depth-based cut-off applied in [137]. Several experiments were conducted using three benchmarks dictionaries for noisy sets involving different error values, $K = 1,\ 2,\ and\ 3$.

Operation	Eng		Dict		Webster	
	OPT	LH	OPT	LH	OPT	LH
Additions	21.0	27.6	6,122	9,205	45,572	68,133
Improvement	23.91		33.49		33.11	
Minimizations	25.2	31.8	7,347	10,430	53,965	76,530
Improvement	20.75		29.56		29.49	
Total	46.2	59.4	13,469	19,635	99,537	144,666
Improvement	**22.22**		**31.40**		**31.19**	

Table 10.3: The results obtained in terms of the number of operations (additions and minimizations) needed when the maximum number of errors $K = 3$ and known *a priori*, the costs are of a 0/1 form, and the new OPT-DFS-trie is applied over the **LHBB** scheme. The figures given are in Millions, and the *total* improvement obtained is in "bold" face.

Operation	Eng		Dict		Webster	
	OPT	DFS	OPT	DFS	OPT	DFS
Additions	50.1	58.16	37,266	42,909	584,316	654,692
Improvement	13.78		13.15		10.75	
Minimizations	51.0	59.0	37,866	43,509	592,442	662,817
Improvement	13.79		12.96		9.11	
Total	101.1	117.1	75,132	86,418	1,176,758	1,317,509
Improvement	**13.66**		**13.05**		**10.68**	

Table 10.4: The results obtained in terms of the number of operations (additions and minimizations) needed when the maximum number of errors $K = 1$ and is **not** known *a priori*, the costs are of a 0/1 form, and the new OPT-DFS-trie is applied over the **DFS-trie** scheme. The figures given are in Millions. The time shown is in seconds, and the *total* improvement obtained is in "bold" face.

The results demonstrated a significant improvement, with respect to the number of operations needed for approximate searching using tries which could be even as high as 48%. The new technique showed marked benefits when applied in conjunction with the technique described in Chapter 8 [19] because of the fact that the new technique is advantageous when the fanout of the nodes of the trie is high.

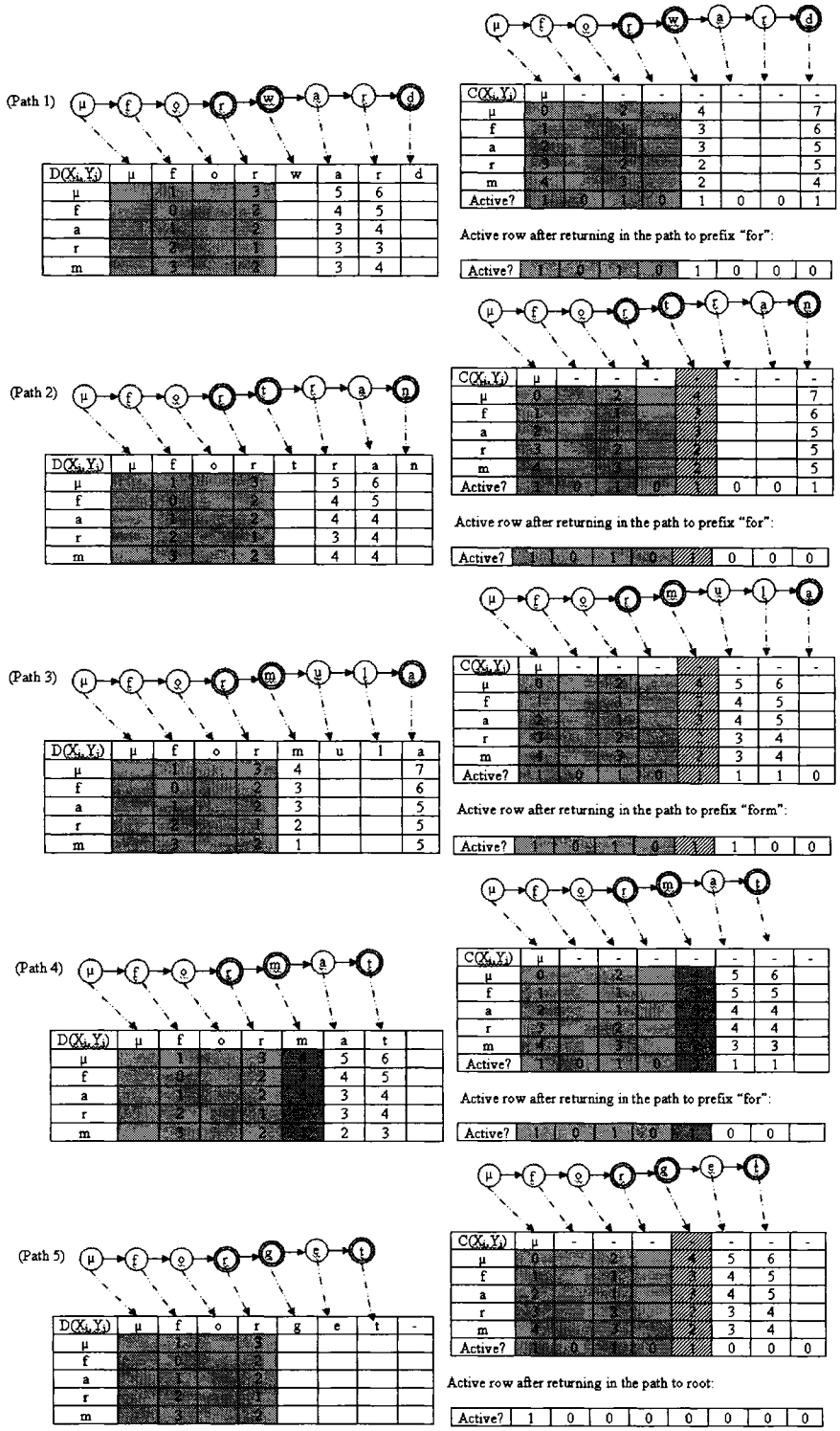


Figure 10.3: An example for edit distance calculations for the noisy word "farm" against the whole dictionary in Figure 8.1.

Part IV

Contributions and Future Directions

Chapter 11

Conclusions

The main aim of this Thesis was to enhance the search performance for strings when they were stored using the *trie* data structure, and to develop methods that work well, in practice, especially for dictionary-based techniques. The enhancing of the search was done for two domains, namely the *exact* and the *approximate* string-searching domains. This was achieved in the "exact" domain in two ways. In the first method, we applied self-adjusting techniques for the trie data structure when nodes of the trie were implemented as binary search trees. In the second method, we incorporated the concept of "direction" by proposing a new representation for the trie, namely the Dual-Trie (DT). With regard to the "approximate" domain, we proposed the use of Artificial Intelligence (AI) search techniques when the strings were stored in a trie, and we also optimized the dynamic programming calculations for edit distances.

The next few sections discuss an overview of the contributions, both to the field of information retrieval and syntactic pattern recognition. We will also catalogue the future possible work for each of these two fields.

11.1 Tries in Information Retrieval

Chapter 2 surveyed the different variations of tries that can be used for storing and retrieving strings, from a fairly comprehensive literature survey. For each variant in the Chapter, we described its form, the objective for which it was designed, and provided an example for the structure. It also showed how the search was accomplished, mentioned its complexity, and illustrated the applications for which the structure was designed.

As can be seen from this survey, the work concerning tries was extensive, and also on-going. The importance of tries for the wide range of applications in which they can be used was also mentioned.

The main problem with the trie is its large memory usage. Proposals for modified tries that address this issue can be broadly placed in two groups [69]: Reduction in the trie node *size*, and reduction in the *number* of trie nodes. In this part of the Thesis, we made some contributions in each of these avenues.

We will first summarize the different contributions done in this direction, and then present the possible future avenues for research.

11.1.1 Self-Adjusting Ternary Search Tries

In Chapter 3, we proposed self-adjusting techniques for Ternary Search Tries (TSTs). The TST is a representation for the trie data structure that is useful for decreasing the space by changing the representation of the node to be a BST. We applied two self-adjusting heuristics, that were previously used for BSTs, to TSTs. These heuristics were, namely, the **splaying** (Splay-TST) and the **Conditional Rotation** (Cond-TST) heuristics. We also applied another balancing strategy used for BSTs, to TSTs, which was the basis for the **Randomized** search trees, or Treaps, (Rand-TST). The formal properties of the data structure were stated, and numerous experiments were done to investigate the performance of the different adjusting heuristics when applied to TSTs. The goal of the experiments was to see how they performed in the context of

different probability distributions characterizing various benchmark data sets. The comparison (made on "moderate-size" data sets) was measured in terms of the running time, quantified in seconds.

From the results we unequivocally concluded that the Cond-TST was the best scheme that could be used to improve the performance of the original TST, and that it had the ability to learn when to stop "self-adjusting", i.e., whenever the tree did not need any further adjustments.

11.1.2 Dual-Tries

Chapter 4 introduced a new structure, namely the Dual-Trie (DT). It followed the direction of reducing the space of the trie by reducing the number of nodes it had. This could be done by dividing the strings in the dictionary into two portions according a ratio $f : r$. The two portions would be stored in two different tries. The first one was stored in the FR trie, and the reverse of the second one was stored in the RE trie. This helped to get the maximum benefits of both the common prefixes and the common suffixes for the strings in a given dictionary. The two tries were linked by giving codes to the leaves of one trie, and by storing these codes in the corresponding leaves of the second trie in BSTs. These codes were stored in the second trie only when both the prefix and suffix constituted a word in the dictionary. As the number of nodes stored in the BSTs increased, some balancing heuristics were applied to increase the time performance.

The new structure showed great benefits for compressing data, gave a very good space performance, and, at the same time, maintained the same time performance as the two-trie. The structure showed high space performance for data involving Bio-informatics. For this kind of data, the results demonstrated an improvement of 99% saving in the number of trie nodes with respect to that of the two-trie data structure. The DT structure was also recommended for storing a large number of strings. In this case, the compact tries showed a poor space performance. The DT structure also showed an improvement for data containing longer strings. In this case, the number of single descent nodes was large, and the performance of the DT structure had the same advantages as the two-trie data structure.

11.1.3 Future Work in Information Retrieval

The possible future work that can be done is as follow:

1. **Complete theoretical analysis of the Dual-Trie structure**:
   In Chapter 4, we introduced the new DT structure, and also submitted an extensive empirical evaluation. An overview of the possible time complexities for the different operations that can be involved with the new structure, was also provided.

   As a future work, we would like to have a complete space analysis of the DT structure. This analysis should consider the different possibilities for the $f : r$ ratio, including the possibility of a situation when the ratio is unity.

   This complete analysis should be compared with the analysis for the two-trie, to prove theoretically, the advantages of the DT structure when compared to the two-trie, and the standard trie data structures.

11.2 Tries in Syntactic Pattern Recognition

In Chapter 5, we reviewed all the important concepts and notations needed for this portion of the Thesis that dealt with Syntactic PR. The Chapter started by listing the basic notations, and the problem definitions and applications. It then gave the basic concepts of approximate string matching for the inexact string-to-string matching problem. It continued with a review of the different AI search techniques that were to be used throughout this part of the Thesis. The Chapter concluded with a listing of some of the assumptions which were to be used.

In Chapter 6, we presented the state-of-the-art for the work done in approximate string matching. We first gave a classification of the different approaches done in the field, and then presented an overview of each of the schemes. The overview on dictionary-based approaches was more detailed, and it more specifically highlighted the techniques applicable when the trie was used as the data structure to store the dictionary.

We also mentioned that with such a variety of different approaches used to solve the same problem, it was difficult to select a single appropriate algorithm that would be suitable for every specific application. Although the theoretical analysis given in the literature is useful, we observed that it was important that the theory was accompanied and justified by extensive experimental comparisons.

We shall now summarize the various contributions made in this part of the Thesis, and then proceed to suggest the possible directions for future work.

11.2.1 Breadth-First-Trie Based Scheme

In Chapter 7, we presented a feasible solution for the problem of estimating a transmitted string X^* by processing the corresponding string Y, which was a noisy version of X^*, an element of a finite (but possibly, large) dictionary H, when the whole dictionary was considered simultaneously. The method used a BFS on the trie representation of the dictionaries (described in Section 6.6) using the simultaneous dynamic programming equation relating the distances between the prefixes of Y and the prefixes of all the words in H. Since the set of all the prefixes used in the computations could not be used profitably, we proposed an enhancement by the introduction of a new data structure called the Linked Lists of Prefixes (LLP), which could be constructed when the dictionary was represented using a trie. The LLP was an enhanced, but modified, representation of the trie, which could be used to facilitate the "dictionary-based" dynamic programming calculations.

This BFS-trie-based algorithm for the syntactic PR of strings was rigorously tested and the results showed significant benefits, with respect to number of computations of up to 21% when compared to DFS-trie-based algorithm.

11.2.2 A Look-Ahead Branch and Bound Pruning Scheme

In Chapter 8, we presented a new Branch and Bound (BB) scheme that can be applied to approximate string matching using tries, which we called a *Look-Ahead* Branch and Bound scheme or the LHBB-trie pruning strategy. The new scheme made use of the information that the lengths of the strings stored in the dictionary were known *a priori*, thus the lengths of the strings could be related to the maximum number of errors, K. The heuristic that we proposed, worked specifically on a trie and had three characteristics, namely a static component, a dynamic component, and finally, it was of a look-ahead sort, as opposed to the cutoff methods already proposed in [113], [151]. The new LHBB pruning could also be used together with Ukkonen's cutoff technique [151].

Several experiments were conducted using three benchmarks dictionaries for noisy sets involving different error values, $K = 1,\ 2,\ and\ 3$. The results demonstrated a significant improvement, with respect to the number of operations needed for approximate searching using tries, which could be even as high as 30%. Other results were shown when the costs are general, in which case improvements of up to 47% were obtained, when compared with the DFS-trie-based algorithm where the Ukkonen's cutoff could not be used. Finally, further improvements were also obtained when the algorithm also utilized a best match optimization strategy.

11.2.3 Clustered-Beam-Search

In Chapter 9, we have presented a feasible, fast, AI-based solution for the approximate string matching problem, when the whole dictionary was considered simultaneously.

First of all, we proposed a new AI-based search scheme called the Clustered Beam Search (CBS) that could be considered as an enhancement of the Beam Search (BS) used in AI. The new scheme can be used to search a graph more efficiently (i.e., with respect to the number of operations needed). The CBS was applied to the dictionary-based approximate string matching problem, where the dictionary was stored using a trie.

Secondly, we proposed a strategy that could be used in conjunction with the LLP representation of the trie explained earlier. The new implementation strategy helped to prune the search space more efficiently, and to dramatically decrease the number of operations needed to get the nearest neighbor to Y.

Thirdly, the CBS-LLP-based approximate string matching was compared with the benchmark Depth-First trie-based technique proposed by Shang *et.al.* [137] using large and small dictionaries. The results demonstrated a significant improvement with respect to the number of operations needed (up to 75%) while keeping the accuracy comparable to the optimum. It has been compared with the BS-LLP-based method to show the benefits of CBS over BS. The results, under certain conditions, sometimes showed improvements of more than 90%.

Finally, the new scheme, CBS-LLP-based, has also been tested for noisy strings obtained by the string generation method using a new error model, where the error predominantly appears at the end of the string. The results demonstrated a marked improvement of more than 95%, while keeping the accuracy the same.

11.2.4 Trie-Based Dynamic Matrix Optimization

In Chapter 10, we presented a new optimized scheme for the Dynamic Programming (DP) matrix calculations that can be applied to approximate string matching using tries. The optimized technique depends on the fact that we are processing the words of the dictionary simultaneously using the trie data structure, and that we can have common column calculations for the DP matrix along the children of the same parent, where these children are actually different prefixes in the dictionary. This permitted us to obtain a horizontal cut-off of the calculations instead of merely the depth-based cut-off derived in [137]. Several experiments were conducted using the three benchmarks dictionaries for noisy sets involving different error values, $K = 1,\ 2,\ and\ 3$.

The results demonstrated a significant improvement, with respect to the number of operations needed for approximate searching using tries, which was as high as 48%. The new technique yielded significant benefits when it was applied in conjunction with the technique described in

Chapter 8 [19] because of the fact that the former took advantage of the special properties of the trie when the fanout of the nodes is high.

11.2.5 Future Work in Syntactic Pattern Recognition

There are many possible avenues for future research. We list them below.

1. **Optimize probabilistic methods**:
   For the General Levenshtein Distance (GLD) the elementary symbol edit distances are symbol dependent. The question of how these elementary distances can be assigned is relatively open. As we saw in Chapter 5, the assignment can be parametric [42, 124] or entirely symbol dependent [76, 134]. The fundamental problem that arises from all these assignment strategies is that the final classified string obtained using such edit distances has no probabilistic significance except in some rather simple cases. Furthermore, if $D(X, Y)$ is the edit distance associated with editing string X to Y, the latter has no explicit relationship to $Pr(X \rightarrow Y)$ except in a few rather trivial cases. Some efforts to extend these, using so called "normalized" distances [125] have been reported, but here too, the probabilistic significance of the resultant distance is yet unknown. The understanding of how the individual patterns from the various classes could have been generated can lead to the design of optimal classifier. This was earlier done in [121] by explicitly modelling the channel, η^*, as a generator whose input is the string X, and whose output is the random string Y.

   As a future work, we would like to extend the methods for probabilistic computations discussed in [121, 122, 123] using the trie data structure and the semi-ring properties discussed in [77, 79]. We explain this below.

   The problem with the probabilistic methods discussed in [122, 123] involves the expensive time requirements, since a cubic matrix is needed for computations. Additionally, the computations are done on a word-by-word basis for the dictionary-based probabilistic approach. We would like to relax the probabilistic equation (for example, by removing

the factorial terms (see [121, 122])) so that we can use a two dimensional matrix, and at the same time apply probabilistic computations to dictionary-based problems when the dictionary is stored as a trie. Thus, we would like to obtain the benefits of saving the computations for common prefixes, while maintaining the same high accuracy of the probabilistic method.

2. **CBS for probabilistic computations**:
   In the context of the above discussion, we would like to apply the new search scheme that we proposed, namely the CBS, in order to achieve approximate string matching when the relaxed probabilistic model is used in the computations instead of the edit distance model presently used. We anticipate that this will yield extremely accurate results, and at the same time maintain the same pruning capability.

3. **Optimize trie-based computations by using Dual-Trie**:
   To optimize the distance-based computations even more, we intend to utilize the property that in most languages, many strings share both common suffixes and prefixes. Up to the present time, all our work was directed at optimizing calculations by taking advantage of only the common prefixes. We intend to extend the work to optimize the computations for both suffixes and prefixes by using updated DP equations on the Dual-Trie data structure given in Chapter 4.

   Our future work in this area will require the update of equations for the DP conditions, and the decision criteria for the nearest neighbor string for a given noisy word. In this context, the "Dual-Trie" is a promising structure, because the essential idea is to construct two tries, one for the prefixes and the other for the "reverse" suffixes of the strings in H. When a noisy word is received, we intend to divide *it* into two parts and to search for each part in the corresponding trie.

   We expect that this work will yield savings in both time and space, because the primary results included in Chapter 4 show a great savings for the Dual-Trie data structure. This is because it combines the optimization in common suffixes and common prefixes by dividing the strings into two parts and storing each half in a separate trie. As we previously stated in Chapter 4, initial experiments seem to show that we can save more than 99% of the

number of nodes used in the trie for Genome data of 262,084 words. We hope to take advantage of this to enhance approximate string matching as well. To solve this problem from this prospective, it appears as if we have to traverse the trie "as a whole" to get the most likely string. Decreasing the number of nodes by 99% should thus definitely lead to a decrease in the search time involved in solving approximate string matching problems. Also, each of the two tries in the Dual-Trie structure is actually the same as the trie data structure, and thus we believe that we can utilize all the new approaches proposed in the Syntactic PR Part of the Thesis.

4. **Two-Level Trie-Based Dynamic Matrix Optimization**:

   The work in Chapter 10 can be extended to include two-level optimization. In other words, by not merely examining the parent, but also the grand-parent of every node, we can optimize the common column calculations two levels up. In this case, because the calculation of any column depends solely on the previous column (parent), the common calculations will propagate through the entire trie and not just through a single node. The overhead for this extension is that we will need an additional third DP matrix to store the common column calculations for the children of the grand-parent. In this case, we can achieve a worst case complexity of $O(K|\Upsilon + 1|^K)$ number of column calculations, which has benefits over the DFS-trie method when the fanout of the nodes is greater than Υ. This in turn leads to a higher probability of reusing the common column information.

   The calculations within the column itself can be further optimized. This comes from the fact that for the columns that correspond to characters in Y, we will not have to recalculate the entire column (with respect to the common column information). We can start the calculations from the position where the corresponding character first appears in Y, a condition which can be quickly inferred by storing this additional information within the vector V (see Section 10.2.2). However, the question of considering whether this will lead to a significant improvement remains open.

Appendix A

Pseudo-code for the Self-Adjusting TSTs

In this Appendix we present the pseudo-code for different algorithms used for the access operations of the TST when the corresponding self adjusting and balancing heuristics, discussed in Chapter 3, are invoked. The presentation of the algorithms follows the style used in the standard MIT-Press text book [51].

The pseudo-code for the access operation supported by the Splay-TST is given in Algorithms A.1 and A.2. The pseudo-code for the insertion operation supported by the Rand-TST is given in Algorithms A.3 and A.4. The pseudo-code for the space-optimized version of the conditional rotation heuristic when applied to the TST, T, is given in Algorithm A.5.

Algorithm A.1 Algorithm Splay-TST-Access

Input: A Ternary search trie T and a string s to be searched for, and whose characters are s_i. The end of string s is marked with \$.

Output: The restructured tree T', and the record A_i.

Method:

```
1: if T is empty then
2:   Return NULL
3: end if
4: if T ↑.key = s_i = $ then
5:   perform Splay on node i of T
6:   Return record A_i
7: else
8:   if T ↑.key = s_i then
9:     perform Splay-TST-Access on T ↑.MiddleChild and with character s_{i+1}
10:  else
11:    if T ↑.key < s_i then
12:      perform Splay-TST-Access on T ↑.RightChild
13:    else
14:      perform Splay-TST-Access on T ↑.LeftChild
15:    end if
16:  end if
17: end if
18: End Algorithm Splay_Tree_Access
```

Algorithm A.2 Splay

Input: A Ternary Search Tree T and a node i in T.
Output: A restructured tree T' with i as root.
Method:

```
1: if i is the root of T then
2:   Return T
3: else
4:   if i the root of the current BST then
5:     Splay P(i)
6:   else
7:     if i is a left child then
8:       if P(i) is the root of T then {Case 1}
9:         Right_Rotate P(i)
10:        Return T
11:      else
12:        if P(i) is the root of current BST then {Case 1}
13:          Right_Rotate P(i)
14:          Splay P(P(i))
15:        else
16:          if P(i) is a left child then {Case 2}
17:            Right_Rotate the parent of P(i)
18:            Right_Rotate P(i)
19:          else
20:            Right_Rotate P(i) {Case 3}
21:            Left_Rotate P(i)
22:          end if
23:          Splay i' which is the post-rotational node i
24:        end if
25:      end if
26:    else {node i is a right child}
27:      if P(i) is the root of T then {CASE 1}
28:        Left_Rotate P(i)
29:        Return T
30:      else
31:        if P(i) the root of the current BST then {Case 1}
32:          Left_Rotate P(i)
33:          Splay the parent of P(i)
34:        else
35:          if P(i) is a Right child then {Case 2}
36:            Left_Rotate the parent of P(i)
37:            Left_Rotate P(i)
38:          else
39:            Left_Rotate P(i) {Case 3}
40:            Right_Rotate P(i)
41:          end if
42:          Splay i' which is the post-rotational node i
43:        end if
44:      end if
45:    end if
46:  end if
47: end if
48: End Algorithm Splay
```

Algorithm A.3 Rand-TST-Insert

Input: A Ternary search trie T and a string s to be inserted for, and whose characters are s_i. The end of string s is marked with \$.
Output: The restructured tree T'.
Method:

```
1: if T is empty then
2:    add new node T with key = s_i with new random priority value
3: end if
4: if T ↑.key = s_i = $ then
5:    we have finished
6: else
7:    if T ↑.key = s_i then
8:       {we don't need Rebalance here}
9:       if T ↑.MiddleChild = NULL then
10:         T ↑.MiddleChild ← add new node T with key = s_i with new random priority value
11:      end if
12:      T ↑.MiddleChild ← perform Rand-TST-Insert on T ↑.MiddleChild but with s_{i+1}
13:   else
14:      if T ↑.key < s_i then
15:         n← T ↑.RightChild
16:         if T ↑.RightChild = NULL then
17:            T ↑.RightChild ← add new node T with key = s_i with new random priority value
18:            n ← T ↑.RightChild
19:            Perform Rebalance on T ↑.RightChild
20:         end if
21:         perform Rand-TST-Insert on n but with s_{i+1}
22:      else
23:         n← T ↑.LeftChild
24:         if T ↑.LeftChild = NULL then
25:            T ↑.LeftChild ← add new node T with key = s_i with new random priority value
26:            n ← T ↑.LeftChild
27:            Perform Rebalance on T ↑.LeftChild
28:         end if
29:         perform Rand-TST-Insert on n but with s_{i+1}
30:      end if
31:   end if
32: end if
33: Return T
34: End Algorithm Rand-TST-Insert
```

Algorithm A.4 Rebalance

Input: A node n in a Ternary search trie T to be rebalanced
Output: The restructured tree T'.
Method:

```
1: if P(n) ≠ NULL and P(n) ↑ .periority > n ↑ .periority and n is not a middle child then
2:    if n is a Right child then
3:       Left_Rotate n
4:    else
5:       Right_Rotate n
6:    end if
7: end if
8: End Algorithm Rebalance
```

Algorithm A.5 Optimized-Cond-TST

Input: A Ternary Search Trie T and a search string S assumed to be in T with characters s_i.
Output: The restructured tree T', and a pointer to record A stored at the end of the string S.
Method:

1: **if** T is empty **then**
2: **Return** NULL
3: **end if**
4: $\text{J} \leftarrow T$
5: $\tau_j \leftarrow \tau_j + 1$ {update τ for the present node}
6: **if** $s_j = s_i$ **then**
7: { We reached the end of the current BST}
8: **if** node j is a left child in the current BST **then**
9: $\psi_j \leftarrow 2\tau_j - \tau_{jR} - \tau_{P(j)}$
10: **else**
11: **if** node j is a right child in the current BST **then**
12: $\psi_j \leftarrow 2\tau_j - \tau_{jL} - \tau_{P(j)}$
13: **else**
14: {j is also the root of current BST}
15: $\psi_j \leftarrow 0$
16: **if** $s_{i+1} \neq$ NULL **then**
17: perform Optimized-Cond-TST on $j\uparrow$.MiddleChild but with s_{i+1}
18: **end if**
19: **end if**
20: **end if**
21: **if** $\psi_j > 0$ **then**
22: rotate node j upwards
23: recalculate $\tau_j, \tau_{P(j)}$
24: **end if**
25: **if** $s_j = s_i =$ \$ **then**
26: {We reached the end of the string}
27: **Return** record A_i
28: **end if**
29: **else**
30: **if** $s_j < s_i$ **then**
31: {search the subtrees}
32: perform Optimized-Cond-TST on $j\uparrow$.RightChild
33: **else**
34: perform Optimized-Cond-TST on $j\uparrow$.LeftChild
35: **end if**
36: **end if**
37: **END Algorithm Optimized-Cond-TST**

Bibliography

[1] A. Acharya, H. Zhu, and K. Shen. Adaptive algorithms for cache-efficient trie search. *ACM and SIAM Workshop on Algorithm Engineering and Experimentation*, pages 296–311, January 1999.

[2] G. M. Adel'son-Velski'i and E.M. Landis. An algorithm for the organization of information. *Sov. Math. Dokl.*, 3:1259–1262, 1962.

[3] S. Albers and J. Westbrook. Self-organizing data structures. *Amos Fiat and Gerhard Woeginger, editors, Online Algorithms: The State of the Art, Springer-Verlag*, pages 31–51, 1998.

[4] B. Allen and I. Munro. Self-organizing binary search trees. *Journal of ACM*, 25:526–535, 1978.

[5] J. C. Amengual and E. Vidal. Efficient error-correcting viterbi parsing. *IEEE Transactions on Communications*, 20(10):1109–1116, October 1998.

[6] J. C. Amengual and E. Vidal. The viterbi algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(10):268–278, October 1998.

[7] M. Dietzfelbinger and A. Karlin and K. Mehlhorn and F. M. Der. Dynamic perfect hashing: Upper and lower bounds. In *SIAM J. Comput.*, volume 23, pages 738–761, 1994.

[8] A. Andersson. General balanced trees. *Journal of Algorithms*, 30:1–28, 1999.

[9] A. Andersson and S. Nilsson. Improved behaviour of tries by adaptive branching. *Information Processing Letters*, 46(6):296–300, June 1993.

[10] A. Andersson and S. Nilsson. Faster searching in tries and quadtrees-an analysis of level compression. *Proceedings of Second Annual European Symp. on Algorithms*, pages 82–93, 1994.

[11] A. Andersson and S. Nilsson. Efficient implementation of suffix trees. *SOFTPREX: Software–Practice and Experience*, 25(2):129–141, 1995.

[12] J. Aoe, K. Morimoto, and T. Sato. An efficient implementation of trie structures. *SOFTPREX: Software–Practice and Experience*, 22(9):695–721, 1992.

[13] C. Aragon and R. Seidel. Randomized search trees. *Proc. 30th Symp. on Foundations of Computer Science*, pages 540–545, 1989.

[14] M. Araujo, G. Navarro, and N. Ziviani. Large text searching allowing errors. *Proceedings of the 4th South American Workshop on String Processing (WSP'97), Carleton University Press*, pages 2–20, June 1997.

[15] V. L. Arlazarov, E. A. Dinic, M. A. Kronod, and I. A. Faradzev. On economic construction of the transitive closure of a directed graph. *(Russian) Doklady Akademii Nauk SSSP*, 194:488–7, 1970.

[16] D.M. Arnow and A.M. Tenenbaum. An investigation of the move-ahead-k rules. In *Congressus Numerantium, Proceedings of the Thirteenth Southeastern Conference on Combinatorics, Graph Theory and Computing*, pages 47–65, Florida, February 1982.

[17] G. Badr and B. J. Oommen. A novel look-ahead optimization strategy for trie-based approximate string matching. To appear in Pattern Analysis and Applications, PAA-0681.

[18] G. Badr and B. J. Oommen. Enhancing trie-based syntactic pattern recognition using ai heuristic search strategies. In *Proceedings of ICAPR2005, the 2005 International Conference on the Advances of Pattern Recognition*, volume I, pages 1–17, Bath, United Kingdom, August 2005. This was a plenary talk of the conference.

[19] G. Badr and B. J. Oommen. A look-ahead branch and bound pruning scheme for trie-based approximate string matching. In *Proceedings of the 4th International Conference on Computer Recognition Systems CORES'05*, pages 87–94, Rydzyna Castle (Poland), May 2005.

[20] G. Badr and B. J. Oommen. On using conditional rotations and randomized heuristics for self-organizing ternary search tries. In *Proceedings of ACMSE'2005, the 2005 ACM South Eastern Conference*, volume 1, pages 109–115, Kennesaw, Georgia, March 2005.

[21] G. Badr and B. J. Oommen. Search-enhanced trie-based syntactic pattern recognition of sequences. 2005. Patent Pending: In pre-examination phase.

[22] G. Badr and B. J. Oommen. Self-adjusting of ternary search tries using conditional rotations and randomized heuristics. *The Computer Journal*, 48(2):200–219, March 2005.

[23] G. Badr and B. J. Oommen. On optimizing syntactic pattern recognition using tries and ai-based heuristic-search strategies. *to appear in IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics*, 36(3), June 2006.

[24] R. Baeza-Yates and G. Gonnet. Fast text searching for regular expressions or automaton searching on a trie. *Journal of the ACM*, 43(6):915–936, April 1997.

[25] R. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In D. S. Hirschberg and E. W. Myers, editors, *Proceedings of the7th Annual Symposium on Combinatorial Pattern Matching*, number 1075, pages 1–23, Laguna Beach, CA, 1996. Springer-Verlag, Berlin.

[26] R. Baeza-Yates and G. Navarro. Fast approximate string matching in a dictionary. *in Proceedings of the 5th South American Symposium on String Processing and Information Retrieval (SPIRE'98), IEEE CS Press*, pages 14–22, 1998.

[27] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. In *Annual ACM-SIGIR Conference on Information retrieval*, pages 168–175, Cambridge, Mass., June 1982.

[28] P. Bagwell. Fast and space efficient trie searches. Technical report, 2000/334, Ecole Polytechnique Fdrale de Lausanne, March 2000.

[29] P. Bagwell. Ideal hash trees. Technical report, October 2001.

[30] J. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369, New Orleans, January 1997.

[31] J. Bentley and R. Sedgewick. Ternary search trees. *Dr. Dobb's Journal*, 1998.

[32] A. Berman. A new data structure for fast approximate matching. *Technical Report 94-03-02, Department of Computer Science and Engineering, University of Washington*, March 1994.

[33] J. R. Bitner. Heuristics that dynamically organize data structures. *SIAM J.Comput.*, 8:82–110, 1979.

[34] E. Bocchieri. A study of the beam-search algorithm for large vocabulary continuous speech recognition and methods for improved efficiency. In *Proceedings of Eurospeech*, volume 3, pages 1521– 1524, Berlin, 1993.

[35] R. H. Boivie. On spelling correction and beyond. *AT&T Bell Labs tech. Mem.*, June 1981.

[36] A. Bouloutas, G. W. Hart, and M. Schwartz. Two extensions of the viterbi algorithm. *IEEE Transactions on Information Theory*, 37(2):430–436, March 1991.

[37] J. Bourdon. Size and path length of patricia tries: dynamical sources context. *Random Structures and Algorithms*, 19:289–315, 2001.

[38] R. Brandais. File searching using variable length keys. *In Proceedings of Western Joint Computer Conference*, 15:295–298, 1959.

[39] P. Bucher and K. Hoffmann. A sequence similarity search algorithm based on a probabilistic interpretation of an alignment scoring system. In *Proceedings of the Fourth International Conference on Intelligent Systems for Molecular Biology, ISMB-*, volume 96, pages 44–51, 1996.

[40] H. Bunke. *Structural and Syntactic Pattern Recognition.* Handbook of Pattern Recognition and Computer Vision. Edited by C.H.Chen, L.F.Pau and P.S.P. Wang, World Scientific, Singapore, 1993.

[41] H. Bunke. Fast approximate matching of words against a dictionary. *Computing*, 55(1):75–89, 1995.

[42] H. Bunke and J. Csirik. Parametric string edit distance and its application to pattern recognition. *IEEE Trans. Systems, Man and Cybern*, SMC-25(1):202–206, January 1993.

[43] W. Chang and E. Lawler. Approximate string matching in sublinear expected time. In *13th Annual Symposium on Foundations of Computer Science*, pages 116–124, St. Louis, Missouri, October 1992. IEEE Computer Society Press.

[44] R.P. Cheetham, B. J. Oommen, and D. T. H. Ng. Adaptive structuring of binary search trees using conditional rotations. *IEEE Transactions on knowledge and data engineering*, 5(4):695–704, August 1993.

[45] J. Clement, P. Flajolet, and B. Vallee. The analysis of hybrid trie structures. In *Proc. Annual A CM-SIAM Symp. on Discrete Algorithms*, pages 531–539, San Francisco, California, 1998.

[46] J. Clement, P. Flajolet, and B. Vallee. Dynamical sources in information theory: A general analysis of trie structures. *INRIA Rapport de recherche 3645*, 1999.

[47] R. Cole, L. Gottieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. *Proceedings of the thirty-sixth annual ACM Symposium on Theory of Computing, Chicago, IL, USA*, pages 91–100, June 2004.

[48] D. Comer. Heuristics for trie index minimization. *ACM Transactions on Database Systems*, 4(3):383 – 395, 1979.

[49] D. Comer. Analysis of a heuristic for trie minimization. *ACM Transactions on Database Systems*, 6(3):513–537, 1981.

[50] D. Comer and R. Sethi. The complexity of trie index construction. *Journal of the ACM*, 24(3):428–440, 1977.

[51] T. H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1990.

[52] G. Das, R. Fleisher, L. Gasieniek, D. Gunopulos, and J. Karlainen. Episode matching. In *Proceedings of CPM'97, number 1264 in LNCS*, pages 12–27, Springer-Verlag, 1997.

[53] L. Devroye. Applications of the theory of records in the study of random trees. *Acta Informatica*, 26:123–130, 1988.

[54] G. Dewey. *Relative Frequency of English Speech Sounds.* Harvard Univ. Press, 1923.

[55] M. Du and S. Chang. An approach to designing very fast approximate string matching algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):620–633, 1994.

[56] M. R. Dunlavey. On spelling correction and beyond. *Communications of the ACM*, 24(9):609–618, September 1981.

[57] J. T. Favata. Offline general handwritten word recognition using an approximate beam matching algorithm. *IEEE Transactions on pattern Analysis and Machine Intelligence*, 23(9):1009–1021, September 2001.

[58] Z. Feng and Q. Huo. Confidence guided progressive search and fast match techniques for high performance chinese/english ocr. In *Proceedings of the 16th International Conference on Pattern Recognition*, volume 3, pages 89–92, Quebec, CANADA, August 2002.

[59] J. Finlay and A. Dix. *An Introduction to Artificial Intelligence.* UCL Press, 1996.

[60] M. Firebaugh. *Artificial Intelligence. A Knowledge-Based Approach.* Boyd and Fraser, 1988.

[61] G. D. Forney. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, March 1973.

[62] Jr. Frank E. Muth and Alan L. Tharp. Correcting human error in alphanumeric terminal input. *Information Processing and Management*, 13(6):329–337, 1977.

[63] E. Fredkin. Trie memory. *Communication of the ACM*, 3:490–499, 1960.

[64] M. Fuketa, T. Sumitomo, M. Shishibori, and J. Aoe. A suffix compression algorithm of tries. *ICCPOL'99: 18th International Conference on Computer Processing of Original Languages*, 18:345–348, 1999.

[65] I. Galperin and R.L. Rivest. Scapegoat trees. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 165–174, Austin, Texas, United States, 1993.

[66] G.H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures in Pascal and C.* Addison Wesley, Reading, MA, 1991.

[67] G.H. Gonnet, J.I. Munro, and H. Suwanda. Exegesis of self-organizing linear search. *SIAM J.Comput.*, 10:613–637, 1981.

[68] S. Heinz and J. Zobel. Performance of data structures for small sets of strings. In *Twenty-Fifth Australasian Computer Science Conference (ACSC2002),ACS*, pages 87–94, Melbourne, Victoria, Australia, 2002.

[69] S. Heinz, J. Zobel, and H. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20(2):192–223, 2002.

[70] H.J. Hester and D.S. Herberger. Self-organizing linear learch. *ACM Computing Surveys*, pages 295–311, 1976.

[71] D. S. Hirschberg. Algorithms for longest common subsequence problem. 24:664–675, 1977.

[72] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Comm. Assoc. Comput. Mach.*, 20:350–353, 1977.

[73] J. Iivonen, S. Nilsson, and M. Tikkanen. An experimental study of compression methods for functional tries. *Submitted to WAAPL'99*, 1999.

[74] P. Jacquet and W. Szpankowski. Analysis of digital tries with markovian dependency. *IEEE Trans. Information Theory, IT-*, 37(5):1470–1475, 1991.

[75] W. D. Jonge, A. S. Tanenbaum, and R. P. Reit. Two access methods using compact binary trees. *IEEE Transactions Softwar Engineering*, 13(7):799–809, 1987.

[76] R. L. Kashyap and B. J. Oommen. An effective algorithm for string correction using generalized edit distances -i. description of the algorithm and its optimality. *Inf. Sci.*, 23(2):123–142, 1981.

[77] R. L. Kashyap and B. J. Oommen. A common basis for similarity and dissimilarity measures involving two strings. *Internat. J. Comput. Math.*, 13:17–40, 1983.

[78] R. L. Kashyap and B. J. Oommen. The noisy substring matching problem. *IEEE. Trans. Software Engg. SE-*, 9:365–370, 1983.

[79] R. L. Kashyap and B. J. Oommen. Similarity measures for sets of strings. *Internat. J. Comput. Math.*, 13:95–104, 1983.

[80] R. L. Kashyap and B. J. Oommen. String correction using probabilistic methods. *Pattern Recognition Letters*, pages 147–154, 1984.

[81] C. Knessl and W. Szpankowshi. A note on the asymptotic behavior of the height in b-trie for b large. *Electronic Journal of Combinatorics*, 7:R39, 2000.

[82] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, Ma., 1973.

[83] D. L. Kreher and D. R. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search.* The CRC press Series on Discrete Mathematics and its Applications, 1999.

[84] K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439, December 1992.

[85] P. Laface, C. Vair, and L. Fissore. A fast segmental viterbi algorithm for large vocabulary recognition. In *Proceeding of ICASSP-95*, volume 1, pages 560–563, Detroit, May 1995.

[86] A. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Phys. Dokl.*, 10:707–710, 1966.

[87] A. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17, 1996.

[88] C. Liu, M. Koga, and H. Fujisawa. Lexicon-driven segmentation and recognition of handwritten character strings for japanese address reading. *IEEE Transactions on pattern Analysis and Machine Intelligence*, 24(11):1425–1437, November 2002.

[89] F. Liu, M. Afify, H. Jiang, and O. Siohan. A new verification-based fast-match approach to large vocabulary continuous speech recognition. In *Proceedings of European Conference on Speech Communication and Technology*, pages 1425–1437, Aalborg, Denmark, September 2001.

[90] C. L. Lucchesi and T. Knowaltowski. Applications of finite automata representing large vocabularies. *Software Practices and Experiences*, 23(1):15–30, 1993.

[91] G. F. Luger and W. A. Stubblefield. *Artificial Intelligence Structure and Strategies for Complex Problem Solving.* Addison-Wesley, 1998.

[92] O. Madani. Fast approximate matching in restriction site mapping. *Master Thesis, Computer Science and Engineering, University of Washington*, June 1996.

[93] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 23–32, San Fransisco, CA, USA, January 1994.

[94] S. Manke, M. Finke, and A. Waibel. A fast technique for large vocabulary on-line handwriting recognition. In *International Workshop on Frontiers in Handwriting Recognition*, pages 437–444, Univeristy of Essex, Wivenboe Park, Colchester, England, September 1996.

[95] W. J. Masek and M. S. Paterson. A faster algorithm computing string edit distances. *J. Comput. System Sci.*, 20:18–31, 1980.

[96] J. McCabe. On serial files with relocatable records. *Operations Research*, 12:609–618, 1965.

[97] K. Mehlhorn. Nearly optimal binary search trees. *Acta Informatica*, 5:287–295, 1975.

[98] S. Mibov and K. Schulz. Fast approximate string matching in large dictionaries. 2002.

[99] P. D. Michailidis and K. G. Margaritis. On-line approximate string searching algorithms: Survey and experimental results. *International Journal of Computer Math.*, 79(8):867–888, 2002.

[100] L. Miclet. Grammatical inference. *Syntactic and Structural Pattern Recognition and Applications*, pages 237–290, 1990.

[101] H. Mochizuki, Y. Hayashi, M. Shishibori, and J. Aoe. A compact and fast structure for trie retrieval algorithm. In *Proc. of 1996 IEEE International Conference on Systems, Man and Cybernetics*, pages 2221–2226, Beijing, China, 1996.

[102] M. Mor and Fraenkel. A hash code method for detecting and correcting spelling errors. *Communication ACM*, pages 935–938, December 1982.

[103] D. R. Morrison. Patricia: a practical algorithm to retrieve information coded in alphanumeric. *Journal of ACM*, 15(4):514–534, 1968.

[104] G. Navarro. Approximate text searching. *Ph.D. Thesis, Department of Computer Science, University of Chile*, December 1998.

[105] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, March 2001.

[106] G. Navarro and R. Baeza-Yates. A hybrid method for approximate string matching. *Journal of Discrete Algorithms*, 0(0):1–35, 2000.

[107] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001.

[108] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing text with approximate q-grams. In *Lecture Notes In Computer Science, Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, pages 350–363, Montreal, Canada, June 2000.

[109] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, pages 443–453, 1970.

[110] P. Nicodeme. Compact balanced tries. In *Proceedings of the IFIP 12th world computer Congress*, volume A-12, Madrid, Spain, September 1992.

[111] S. Nilsson and M. Tikkanen. Implementing a dynamic compressed trie. In K. Mehlhorn, editor, *Proceedings Workshop on Algorithm Engineering*, pages 1–3, Saarbrucken, Germany, 1998.

[112] S. Nilsson and M. Tikkanen. An experimental study of compression methods for dynamic tries. *Algorithmica*, 33(1):19–33, 2002.

[113] K. Oflazer. Error-tolerant finite state recognition with applications to morphological analysis and spelling correction. *Computational Linguistics*, 22(1):73–89, March 1996.

[114] T. Okuda, E. Tanaka, and T. Kasai. A method of correction of garbled words based on the levenshtein metric. *IEEE Trans. Comput. C-*, 25:172–177, 1976.

[115] B. J. Oommen. Constrained string editing. *Information Sciences*, 40(3):267–284, December 1987.

[116] B. J. Oommen. Recognition of noisy subsequences using constrained edit distances. *IEEE Trans. on Pattern Anal. and Mach. Intel.,PAMI-*, 9:676–685, 1987.

[117] B. J. Oommen and G. Badr. Breadth-first search strategies for trie-based syntactic pattern recognition. To appear in Pattern Analysis and Applications, PAA-0481.

[118] B. J. Oommen and G. Badr. Dictionary-based syntactic pattern recognition using tries. In *Proceedings of the Joint IARR International Workshops SSPR 2004 and SPR 2004*, pages 251–259, Libon, Portugal, August 2004.

[119] B. J. Oommen, E. R. Hansen, and J.I. Munro. Deterministic optimal and expedient move-to-rear list organizing strategies. *Theoretical Computer Science*, 74:183–197, 1990.

[120] B. J. Oommen and E.R. Hansen. List organizing strategies using stochastic move-to-front and stochastic move-to-rear operations. *SIAM J. Computing*, 16:705–716, 1987.

[121] B. J. Oommen and R. L. Kashyap. A formal theory for optimal and information theoretic syntactic pattern recognition. *Pattern Recognition*, 31:1159–1177, 1998.

[122] B. J. Oommen and R. K. S. Loke. Syntactic pattern recognition involving traditional and generalized transposition errors: Attaining the information theoretic bound. Submitted for Pubication.

[123] B. J. Oommen and R. K. S. Loke. Pattern recognition of strings with substitutions, insertions, deletions and generalized transposition. *Pattern Recognition*, 30:789–800, October 1997.

[124] B. J. Oommen and R. K. S. Loke. Designing syntactic pattern classifiers using vector quantization and parametric string editing. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-29:881–888, 1999.

[125] B. J. Oommen and K. Zhang. The normalized string editing problem revisited. *IEEE Transactions on Pttern Analysis and Machine Intelligence*, SMC-29B:881–888, December 1999.

[126] J. Pearl. *Heuristics : intelligent search strategies for computer problem solving.* Addison-Wesley, 1984.

[127] J. C. Perez-Cortes, J. C. Amengual, J. Arlandis, and R. Llobet. Stochastic error correcting parsing for ocr post-processing. In *International Conference on Pattern Recognition ICPR-2000*, pages 4405–4408, Barcelona, 2000.

[128] J. L. Peterson. Computer programs for detecting and correcting spelling errors. *Comm. Assoc. Comput. Mach.*, 23:676–687, 1980.

[129] B. Rais, P. Jacquet, and W. Szpankowski. Limiting distribution for the depth in patricia tries. *SIAM Journal on Discrete Mathematics archive*, 6(2):197–213, 1993.

[130] R. RAMESH, A. J. G. BABU, and J. P. KINCAID. Variable-depth trie index optimization: Theory and experimental results. *ACM Transactions on Database Systems*, 14(1):41–74, 1989.

[131] K. M. Risvik. Search system and method for retrieval of data, and the use thereof in a search engine. *United States Patent 6377945 B1*, April 23 2002.

[132] R. L. Rivest. On self-organizing sequential search heuristics. *Comm. ACM*, 19:63–67, 1976.

[133] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2003.

[134] D. Sankoff and J. B. Kruskal. *Time Warps, String Edits and Macromolecules: The Theory and practice of Sequence Comparison.* Addison-Wesley, 1983.

[135] K. Schulz and S. Mihov. Fast string correction with levenshtein-automata. *International Journal of Document Analysis and Recognition*, 5(1):67–85, 2002.

[136] R. Sedgewick. *Algorithms.* 2nd edition, Addison-Wesley, Reading MA, 1983.

[137] H. Shang and T. Merrettal. Tries for approximate string matching. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):540–547, August 1996.

[138] M. Sherk. Self adjusting k-ary search trees. *Journal of Algorithms*, 19(1):25–44, 1995.

[139] M. Shishibori, K. Ando, M. Okuno, and J. Aoe. A key search algorithm using the compact patricia trie. *IEEE International Conference on Intelligent Processing Systems*, pages 1581–1584, 1997.

[140] M. Shishibori, J. Aoe, K. Morimoto, M. Shishibori, and K. Park. A trie compaction algorithm for a large set of keys. *IEEE Trans. on Knowledge and Data Engineering*, 8(3):476–491, 1996.

[141] M. Shishibori, H. Mochizuki, T. Arita, and J. Aoe. An efficient method of compressing binary tries. In *Proc. of 1996 IEEE International Conference on Systems, Man and Cybernetics*, pages 2133–2138, Beijing, China, October 1996.

[142] M. Shishibori, K. Morita, K. Ando, and J. Aoe. The design of a compact data structure for binary tries. In *Lam Woo International Conference Centre, Hong Long Baptist University*, volume 17, pages 606–611, April 1997.

[143] M. Shishibori, M. Okuno, K. Ando, and J. Aoe. An efficient compression method for the patricia trie. In *Proc. of 1997 IEEE International Conference on Systems, Man and Cybernetics*, pages 415–420, Florida, US, October 1997.

[144] R. M. K. Sinha. On partitioning a dictionary for visual text recognition. *Pattern Recognition*, 23(5):497–500, 1990.

[145] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[146] G. A. Stephen. *String Searching.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[147] G. A. Stephen. *String Searching Algorithms*, volume 6. Lecture Notes Series on Computing, World Scientific, Sihgapore, NJ, 2000.

[148] E. Sussenguth. Use of tree structures for processing files. *Communications of the ACM*, 6(5):272–279, 1963.

[149] W. Szpankowski. *Average Case Analysis of Algorithms on Sequences.* John Wiley and Sons, New York, 2001.

[150] W. D. Taylor. Grope. *AT&T Bell Labs tech. Mem.*, 1981.

[151] E. Ukkonen. Algorithm for approximate string matching. *Information and control*, 64:100–118, 1985.

[152] E. Ukkonen. Finding approximate patterns in string. *Journal of Algorithms*, 6:132–137, 1985.

[153] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13:260–269, 1967.

[154] J. Vuillemin. A unifying look at data structures. *Journal of the ACM*, pages 229–239, 1980.

[155] R. Wagner and A. Fischer. The string-to-string correction problem. *Journal of the Association for Computing Machinery (ACM)*, 21:168–173, 1974.

[156] R. A. Wagner. Order-n correction for regular languages. *Comm. ACM*, 17:265–268, 1974.

[157] W. A. Walker and C.C. Gotlieb. A top-down algorithm for constructing nearly optimal lexicographical trees. In *Graph Theory and Computing*, New York, 1972. Academic Press.

[158] H. E. Williams, J. Zobel, and S. Heinz. Self-adjusting trees in practice for large text collections. *Software Practice and Experience*, 31(10):925–939, 2001.

[159] I. H. Witten and T. C. Bell. Source models for natural language text. *International Journal on Man Machine Studies*, 32:545–579, 1990.

[160] J. G. Wolff. A scaleable technique for best-match retrieval of sequential information using metrics-guided search. *Journal of Information Science*, 20(1):16–28, 1994.

[161] C. K. Wong and A. K. Chandra. Bounds for the string editing problem. *J. Assoc. Comput. Mach.*, 23:13–16, 1976.

[162] S. Wu and U. Manber. Fast text searching allowing errors. *Commmunication of the ACM*, 35(10):83–91, October 1992.