TRIE METHODS FOR TEXT AND SPATIAL DATA ON SECONDARY STORAGE

by

HEPING SHANG

School of Computer Science
McGill University
Montréal, Québec
Canada

January 2001

A dissertation
submitted to the Faculty of Graduate Studies and Research
of McGill University
in partial fulfillment of the requirements for
the Degree of Doctor of Philosophy

Abstract

This thesis presents three trie organizations for various binary tries. The new trie structures have two distinctive features: (1) they store no pointers and require two bits per node in the worst case, and (2) they partition tries into pages and are suitable for secondary storage. We apply trie structures to indexing, storing and querying both text and spatial data on secondary storage. We are interested in practical problems such as storage compactness, I/O efficiency, and large trie construction.

We use our tries to index and search arbitrary substrings of a text. For an index of 100 million keys, our trie is 10% – 25% smaller than the best known method. This difference is important since the index size is crucial for trie methods. We provide methods for dynamic tries and allow texts to be changed. We also use our tries to compress and approximately search large dictionaries. Our algorithm can find strings with k mismatches in sublinear time. To our knowledge, no other published sublinear algorithm is known for this problem.

Besides, we use our tries to store and query spatial data such as maps. A trie structure is proposed to permit querying and retrieving spatial data at arbitrary levels of resolution, without reading from secondary storage any more data than is needed for the specified resolution. The trie structure also compresses spatial data substantially. The performance results on map data have confirmed our expectations: the querying cost is linear in the amount of data needed and independent of the data size in practice. We give algorithms for a set of sample queries including geometrical selection, geometrical join and the nearest neighbour. We also show how to control query cost by specifying an acceptable resolution.

Résumé

Cette thèse présente trois méthodes de structuration de tries binaires. Ces nouvelles structures de trie ont deux caractéristiques distinctives: (1) elles ne requièrent pas de pointeurs tout en utilisant qu'un maximum de deux bits par noeuds, et (2) elles permettent de paginer les tries de manière à pouvoir les stocker en mémoire secondaire. Nous appliquons ces structures de trie à l'indexage, la sauvegarde, et l'interrogation de textes et de données spatiales conservées en mémoire secondaire. Nous nous sommes intéressés à des problèmes pratiques de représentation compacte des données, d'efficacité des entrées/sorties, et de construction de gros tries.

Nous utilisons nos tries pour indexer et effectuer des recherches de sous-chaînes dans un texte. Pour un index de 100 millions de clés, notre trie est de 10 à 25% plus petit que celui obtenue par la méthode la plus connue. Cette différence est importante puisque la grosseur de l'index est cruciale pour les méthodes utilisant des tries. Nous proposons des méthodes pour des tries dynamiques et permettons des changements aux textes. Nous utilisons aussi nos tries pour compresser de gros dictionnaires et faire des recherches approximatives dans ceux-ci. Notre algorithme peut trouver des chaînes de caractères ayant k différences avec l'argument de recherche en temps sous-linéaire. À notre connaissance, aucun autre algorithme sous-linéaire n'a été publié pour ce problème.

De plus, nous utilisons nos tries pour stocker et interroger des données spatiales comme des cartes. Une structure de trie est proposée pour permettre l'interrogation et l'extraction de données spatiales à des niveaux de résolution arbitraires, sans avoir à lire de la mémoire secondaire plus de données qu'il n'en faut pour la résolution spécifiée. La structure de trie compresse aussi les données spatiales de façon significative. Les resultats des tests effectués sur des cartes ont confirmé nos attentes: le coût des requêtes est linéaire par rapport au nombre de données nécessaires, et est indépendant de la grosseur des données en pratique. Nous donnons des algorithmes pour un ensemble de requêtes qui inclut la selection géométrique, la jointure géométrique, et la recherche du plus proche voisin. Nous montrons aussi comment contrôler le coût des requêtes en spécifiant un niveau de résolution acceptable.

Acknowledgements

I would like first and foremost to express my gratitude to my supervisor and mentor, Professor Tim Merrett, whose support and encouragement were indispensable throughout my doctoral program. He contributed a great deal of his time, effort and thought to the work presented in this dissertation. Professor Merrett showed dedication to his students, his profession and his family. He and his wife, Mary Ann, have been role models for me in my life. During the years of my study in the program, I also received generous financial support from him, without which it would be impossible for me to complete this program. I consider myself fortunate to have been working in association with him and the bond we have formed over the years will be a source of my strength and courage in the years to come.

I am grateful to the School of Computer Science and IRIS (Centres of Excellence) for their financial support, and to my thesis committee members, Prof. Monty Newborn, Prof. Nathan Friedman and Prof. Luc Devroye, for their comments. Thanks also go to Prof. Frank Wm. Tompa, the external examiner, for his extremely thorough examination of this thesis and many constructive suggestions, to Ms. Vicki Keirl for her patience and readiness to provide administrative help.

I wish to thank all my friends during my years at McGill and Montréal for the joy and fun we shared. Special mention should be made of Samir Douik, Luminita Stancu, Xiaoyan Zhao, David Bremner, Josée Turgeon, André Clouâtre, Yu Miao and Chenfeng Huang.

Thanks must also go to my parents, my late father-in-law, and my extended family for their love and constant support.

Finally, I give my special thanks to my wife, Xiaohui, for her devotion to the family, and my son, Jimmy, for the cheer and love we share in the family.

To

Zou Xiaohui,

high school sweetheart, wife and friend who
has made this thesis possible.

Contents

Abstract **iii**

Acknowledgements **v**

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Trie Methods . . . 3

1.2.1 Trie Applications . . . 5

1.2.2 Trie Parameters . . . 7

1.2.3 Trie Representations . . . 8

1.3 Text Searching . . . 12

1.3.1 Exact Text Searching . . . 13

1.3.2 Approximate String Matching . . . 16

1.4 Spatial Data Structures . . . 17

1.4.1 Multi-dimensional Point Structures . . . 17

1.4.2 Non-point Structures . . . 19

1.4.3 Summary . . . 20

1.5 Thesis Outline . . . 21

2 Trie Organization **23**

2.1 Pointerless Representations . . . 23

2.1.1 *FuTrie* . . . 24

2.1.2 *OrTrie* . . . 25

2.1.3 *PaTrie* . . . 26
2.2 Trie Partitioning . . . 27
2.3 Trie Searching . . . 29
2.4 Trie Construction . . . 29
2.4.1 *FuTrie* Construction . . . 31
2.4.2 *OrTrie* Construction . . . 33
2.4.3 *PaTrie* Construction . . . 33
2.5 Summary . . . 36

3 Exact Text Searching **37**
3.1 Text Trie . . . 37
3.2 Statistics on Text Tries . . . 39
3.2.1 Measured Distributions . . . 39
3.2.2 Estimated Performance . . . 43
3.3 Text Trie Construction . . . 46
3.3.1 Dynamic Text . . . 47
3.3.2 Sistring Sorting . . . 49
3.4 Experimental Results . . . 54
3.4.1 Text Trie Sizes . . . 55
3.4.2 Search Times . . . 56
3.4.3 Construction Times . . . 56
3.5 Other Trie Searches . . . 57
3.6 Summary . . . 61

4 Approximate String Matching **62**
4.1 String Similarity . . . 62
4.1.1 Edit Distance . . . 63
4.1.2 Dynamic Programming . . . 63
4.2 Approximate Searching . . . 65
4.2.1 Observations . . . 65
4.2.2 Algorithm . . . 66

4.3 Experimental Results . . . 69
4.3.1 Dictionary Trie Sizes . . . 69
4.3.2 Search Times . . . 70
4.4 Soundex Searching . . . 72
4.5 Summary . . . 73

5 Spatial Zooming **74**
5.1 Map Data Representation . . . 75
5.1.1 Map Relation . . . 75
5.1.2 Dimension Doubling and *ZoomTrie*s . . . 75
5.1.3 Data Resolution . . . 77
5.2 Displaying Operations . . . 78
5.2.1 *Scan* . . . 79
5.2.2 *Search* . . . 80
5.3 Experimental Results . . . 81
5.3.1 Resolution and Feature Priority . . . 81
5.3.2 Windows on Map . . . 84
5.3.3 Extrapolations . . . 84
5.4 Summary . . . 87

6 Spatial Querying **89**
6.1 Query Categories . . . 90
6.1.1 Geometrical Selection: Examples . . . 90
6.1.2 Geometrical Join: Examples . . . 91
6.2 *ZoomTrie* Search: Algorithms . . . 91
6.2.1 Primitives . . . 91
6.2.2 Linear Predicate Precompiling . . . 94
6.3 *ZoomTrie* Search: Implementations . . . 94
6.3.1 Linear Selection . . . 95
6.3.2 Non-Linear Selection . . . 98
6.3.3 Variable-Resolution Selection . . . 100

6.3.4 Geometrical Join 101
6.4 Experimental Results 102
6.4.1 *ZoomTrie* Trie Sizes 102
6.4.2 Data Compression 103
6.4.3 Search Time 104
6.5 Summary 107

7 Conclusion **108**
7.1 Claim of Originality 108
7.2 Contributions 109
7.3 Future Research 111

Bibliography **112**

List of Tables

1.1 Asymmetric Trie Parameters 8
1.2 Trie Representation Comparison 12
2.1 *FuTrie* Structure 25
2.2 *OrTrie* Structure 25
2.3 *PaTrie* Structure 26
2.4 Paged Trie Structure 28
3.1 Random Trie Parameters 39
3.2 Skip Distributions 42
3.3 Regression Fitting 42
3.4 Comparing Regressions 43
3.5 Binary Trie Comparison 46
3.6 Sorting: Time and Space 54
3.7 *PaTrie* Sizes 55
3.8 *PaTrie* Search Times 56
3.9 *PaTrie* Construction Times 57
4.1 Dynamic Programming 64
4.2 Ukkonen's Cutoff 64
4.3 Dynamic Programming Tables 66
4.4 Dictionary and Trie Sizes 69
4.5 Approximate Search Times 71
5.1 Map Filtering v.s. *ZoomTrie* 81
5.2 *ZoomTrie*s at Various Resolutions and Priorities 83
5.3 Window Search Times 84

6.1 Regression Fitting 103
6.2 Map Overlay Statistics 103
6.3 Line Representation Comparison 103

List of Figures

1.1 Trie Structures 4

2.1 Tries and Bitstrings 24

2.2 Paged *PaTrie* 28

2.3 Page Structure 28

3.1 Text Tries 38

3.2 Text Trie Distributions 41

3.3 Updating *PaTrie* 47

4.1 Dictionary Trie 65

4.2 Approximate Trie Searching 67

4.3 Soundex Searching 72

5.1 Dimension Doubling 76

5.2 Map and *ZoomTrie* 76

5.3 Zooming by *ZoomTrie* 77

5.4 Priority and Window Searching 80

5.5 Memphremagog Road Map 82

5.6 Memphremagog Contour (at Every 50 Feet) Map 82

5.7 Contour Map at Resolution 256×256 85

5.8 Contour Map Zooming 85

5.9 *ZoomTrie*: Trie Nodes v.s. Resolutions 86

5.10 *ZoomTrie*: Search Times v.s. Accessed Nodes 86

6.1 Interval Space 96

6.2 *PR-Trie* for Containment Searching 96

6.3 Edge-Line Crossing 97

6.4 *PR-Trie* for Length Searching 98
6.5 Variable-Resolution Querying 100
6.6 Edge-Edge Crossing 101
6.7 *ZoomTrie* Distributions 102
6.8 *ZoomTrie* Search Times 104
6.9 *ZoomTrie* Join Performance 106

List of Algorithms

2.1 Pointerless Trie: Searching Child Node 30
2.2 *FuTrie*: Appending Key 32
2.3 *PaTrie*: Parsing $List(K_n)$ 35
3.1 Text Trie: Inserting and Deleting 48
3.2 Counter Sort: Generating Initial Runs 52
3.3 Counter Sort: Merging Initial Runs 53
4.1 Dictionary Trie: Approximate Search 68
5.1 *ZoomTrie* Primitives: *Scan* and *Search* 79
6.1 *ZoomTrie*: Geometrical Selection 92
6.2 *ZoomTrie*: Geometrical Join 93
6.3 *ZoomTrie*: Searching the Nearest Edge 99

Chapter 1

Introduction

1.1 Motivation

Suppose we are looking for the word `text` in a thick English dictionary. Suppose the dictionary has a *thumb-index*. By checking the thumb-index, we can immediately locate all the `t` pages. If the dictionary has a secondary thumb-index for these `t` pages, we can then locate all the `te` pages. In general, if a thumb-index is available up to the first x letters, then it is possible, in x lookups, to locate all the pages that contain all the words prefixed with the same first x letters. The lookup of thumb-indices is called *digital search* [Knu73]. The data structure associated with the digital search is a *trie* which comes from the word re*trie*val [Fre60]. Trie structures were first developed by [Bri59], and have been discussed intensively by [Knu73] and in other data structure text books.

Tries (or *digital tree*s [Knu73]) are simple but very powerful data structures. First of all, trie shape is independent of the order in which data are presented to the trie construction algorithms. Trie shape is uniquely determined by its data set. Trie methods do not need various construction algorithms, as tree methods do, to prevent tries from degenerating. Neither do they need various reorganization algorithms, as tree methods do, to keep tries balanced.

However, the most distinctive characteristic of tries is the way they classify the data set into hierarchical groups. In contrast to tree structures which partition the data set according to the data presented, trie structures partition the data set according to the data space. Tries recursively partition the data space into equally sized subspaces. Each subspace corresponds to one subtrie which contains at least one datum refers to the underlying space. When traversing a trie down to a subtrie, we get a focussed view on a certain part (a subspace) of the data space. When walking up from a subtrie, we get a broadened view on a larger portion of the data space. Each subspace or subtrie may contain as much as the whole data set, or as few as one datum. In other words, tries are hierarchical data structures which preserve the scale of each part of the space at each level. Tries group the data sets in terms of resolution (or level of abstraction, or approximation, or remoteness). We call this *trie zooming*.

In text searching, trie zooming has only been interpreted as prefix searching. Tries were used for text searching by Morrison [Mor68] and exploited by Gonnet *et al.* [Gon88, GBYS92, Tom92] for the PAT trie implementation of the electronic version of the *New Oxford English Dictionary* (*New OED*) which covers 20 volumes of print or 600MB. Trie methods give search costs often proportional only to the length of the string being sought, and in the worst case, to the logarithm of the size of the text being searched. No other sublinear methods for full text search are known. However, a major difficulty with tries is the size. For example, the PAT trie for the *OED* has 119 million keys, each starting at a word and continuing until the end of the entire dictionary. If it used two pointers per node, it would take 2×12B×119M=2.9GB, assuming four bytes for pointer and node. This is not acceptable in practice. Minimizing the trie size of such a large text is indispensable.

Trie methods are capable of doing other kinds of text searches (see §3.5) which are either difficult or inefficient over other data structures. However, we still face the challenge to make full use of trie methods for text searching [GBYS92]. Specifically, the connections between trie zooming and the approximate string matching are yet to be established. As we have stated, tries provide different levels of string approximation. This can be useful in approximate string matching.

In spatial data searching, tries have been applied to spatial indexing ranging from the *kd*-trie [Ore82] to various quadtries [Hun78, Sam90]. However, the power of tries provides us not only with efficient methods to navigate to the subspace of interest, but also with data structures to group spatial data at varying degrees of resolution and to store only one copy of the data. Furthermore, since the common prefixes of all data elements are stored only once each, trie structures give substantial data compression. These are important assets in spatial data, where gigabytes and terabytes of data are becoming the norm. For example, each topological map at 1:50,000 resolution provided by Energy, Mines and Resources of Canada, requires 16MB storage space. There are 13,000 such maps currently available, and these cover only half of Canada.

We are interested in practical problems that relate to trie methods for storing and querying bulk and persistent data. The trie size, as we have mentioned, is a major problem. Large trie construction is another important problem. A trie construction algorithm usually requires $n\lg(n)\times t$ time, where n is the total number of data elements and t is the disk access time. For the *New OED*, a naive approach would take 119M$\times$27$\times$20$ms \approx$ 2.0 years, assuming 20ms per disk access. Even with one random disk access per key insertion, the total disk time is still 119M$\times$20$ms \approx$ 27.5 days. This is not acceptable in practice. We need algorithms that work with secondary storage efficiently.

1.2 Trie Methods

A (full) trie is a $|\Sigma|$-*ary* tree[†] in which each link (or edge) has a symbol from the alphabet Σ and each root-to-leaf path corresponds to a key. Here, $|\Sigma|$ is the alphabet size. Selection of subtries at level i is determined only by the ith symbol of the search key, not the whole key. For example, when searching trie (a) of Figure 1.1 for the word `text`, the first letter `t` leads us to the right descendant. The third letter `x` leads us to the right most descendant. Eventually, the search terminates at a leaf node. On the other hand, if we look for the word `tax`, the second letter `a` leads us to a *null*

[†]Assuming no index key is a prefix of another key. This is the case for the *sistrings* of Chapter 3. To prevent a key being a prefix of other keys, we can append either a unique string or a unique symbol, say *null*, after each key. In the latter case, the arity of the trie is $|\Sigma|$+1-ary.

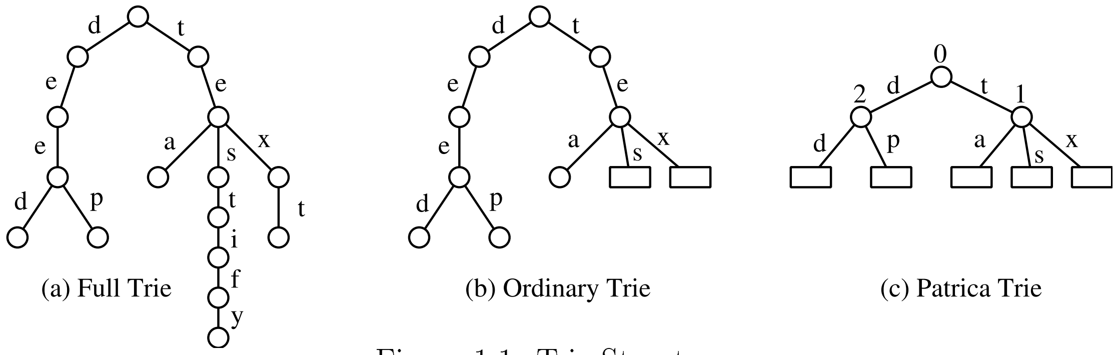


Figure 1.1: Trie Structures

link which means no such word is in the trie. An unsuccessful search terminates at an internal node.

Trie (a) in Figure 1.1 is referred to as *full trie* [CS77] (*pure trie* [Ore82]). There are two other tries. Trie (b) is an *ordinary trie* (*radix search tree, pruned trie* [Knu73, CS77], or simply trie in most literature). Trie (c) is a *Patricia trie* [Mor68] (*collapsed trie, compact trie* [CS77, Szp92]). All three tries in Figure 1.1 are constructed from the same words: `deed, deep, tea, testify`, and `text`.

For an ordinary trie, all single descendant nodes that lead to a leaf node are removed. The pruned links (symbols) are usually stored outside the trie structures and pointed to by pointers in leaf nodes (the rectangular boxes). An ordinary trie is the smallest full trie such that paths truncated at leaves are all pairwise different.

For a Patricia trie, all single descendant nodes are eliminated. To search a Patricia trie, we have to follow the links and numbers in the internal nodes. The number can be either *height*, the level number in the corresponding ordinary trie, or *skip*, the number of removed nodes from the nearest parent which has more than one descendant. Height is the testing symbol position of the search key, and skip is the number of symbols to skip over before the next inspection. Skips are more compact to store than heights. To avoid false matches, a Patricia trie must either store the skipped symbols inside each node (see §2.1.3), or be able to recover them by pointers in the leaf nodes.

Trie structures have many properties. (1) The common prefixes of all key elements are stored only once each. This may give substantial data compression. (2) Trie searching is directed by the search string, and gives search time proportional to the length of search string rather than the trie size. (3) Tries group data according to the data space, not the data presented. Tries preserve the scale of the subspaces at each level, which is a necessary condition for zooming. (4) Trie shapes are uniquely determined by their data presented, not by the orders of key insertions. Tries do not need reorganization algorithms. (5) Tries allow interleaved keys and are suitable for searching multi-keys. (6) Ordinary and Patricia tries are capable of indexing very long, variable length and even *unbounded* key strings.

1.2.1 Trie Applications

Prefix Searching

Many applications require recognizing keywords from a dictionary, and often demand efficient prefix search. Traditional dictionary lookup techniques, such as hashing or tree search, are inadequate because they do not generally allow the search keys to be prefixed or abbreviated. Trie search has been used in many applications: lexical analyzers and compilers [ASU86], pattern recognition [BS89, DTK91], spelling checkers [LEMR89], natural language analysis [TITK88, Jon89], knowledge base retrieval [YKH89], parallel searching [HCE91], and even a custom VLSI chip which can search many tries simultaneously [PZ92].

Text Searching

A great advantage of tries is their potential use in searches which are either difficult or very inefficient over other data structures. Besides prefix searching, tries have been applied to substring searching, proximity searching, range searching, longest repetition searching, most frequent searching [GBYS92, GBY91, ST93], and regular expression searching [BYG89]. We shall give two examples below. Trie methods for text searching will be summarized in §3.5.

The *longest repetition searching* problem is to find two longest and identical substrings in the entire text. This search has been used to recognize and remove repetitions for text compression [ZL77, FG89]. It can also be used to check documents for plagiarism.

The *most frequent searching* problem is to find the most frequently occurring words (or substrings) in a text. This search has been used to generate key phrases in automatic indexing [Jon89]. It can be used to detect frequently occurring subcodes and encode them into macros or machine languages to provide faster performance. Another potential usage of this search is to analyze the personal writing habits of authors (or writing styles in general).

Spatial Data Representation

Ordinary tries have been used to index spatial data, e.g., kd-trie [Ore82], quadtree [Hun78], octree [Mea82], pr-trie [Sam90], ..., etc. Unfortunately, the term *quadtree* (and other related terms) takes more than one meaning. Most often, it refers to a trie structure and hence, should be called *quadtrie*. However, it may also refer to a tree structure, e.g., quad tree in [FB74], and point quadtree in [Sam90]. We shall discuss trie structures for spatial data in §1.4.

Trie Hashing

In *trie hashing* [Lit81], the hash function is defined by an ordinary trie with leaf nodes pointing to buckets (*bucket trie*). Address calculation is carried out by searching the trie.† Trie hashing has been claimed to be one of the fastest access methods (with no more than two disk accesses) for dynamic and ordered files [TB83, Lit85, LNLH91].

Since tries preserve the key order, they are tidy functions [Mer83]. The trie hashing can also be made perfect, i.e., to identify keys with no conflict. Furthermore, when a Patricia trie is used to define the trie hash function, all irrelevant bits of keys are removed from the function. The benefits are twofold: a smaller trie size and a faster access time.

Telecommunications

In telecommunications, messages are usually encoded and transmitted as a sequence of bits. Thus, message decoding becomes typically a trie search process. Another trie

†Trie hashing is not really a hashing method. It takes $\mathcal{O}(\lg n)$ time to find a bucket.

application is to solve communication conflict when a number of spatially isolated and independent sources try to access a single channel. Collision resolution algorithms in [Cap79, Ber84, MF85a] were based on trie search.

1.2.2 Trie Parameters

A number of trie parameters are of interest to us. Trie *depth* is the average path length from the trie root to its leaf nodes. External path length is the sum of path lengths of all leaf nodes. The average trie depth, denoted by A_n, is the expected trie depth of tries with n leaves. A_n gives the average number of symbol inspections made during a *successful* search. Trie *height* (or maximal depth) is the longest path of a trie. The expected trie height, denoted by H_n, is the expected trie height of tries with n leaves. H_n tells the expected worst search time.

The average *unsuccessful* search time is not directly related to A_n. It has been found [Knu73, Szp90] that, for ordinary and Patricia tries, the unsuccessful searches are more likely terminated at internal nodes.

The average value is a rather poor measurement and higher moments are needed. For example, the depth variance provides information on how well a trie is balanced, and the third centralized moment is a measurement of the skewness. Ideally, we would like to know depth distributions. Trie depth has a rich research history [Knu73, Dev82, Dev84, Pit85, KP89, Szp90, Szp91, Jac91].

In practice, the measurement of trie size is as important as the access time. Let trie size, S_n, be the expected number of trie nodes of tries with n leaf nodes. For ordinary tries, S_n has been explored by [Knu73, Reg89, Jac91].

To estimate Patricia trie size, we need to know both S_n and the average *skip* length or skip length distributions. Binary Patricia tries have $S_n = 2n-1$ nodes. No formal discussions were found for S_n of $|\Sigma|$-ary Patricia tries. In [Szp90, Szp91], Szpankowski stated the total number of internal $|\Sigma|$-ary Patricia trie nodes is $n-|\Sigma|-1$. We found this formula valid only when $|\Sigma|=2$ and this was accepted by Szpankowski in a private communication. We do not find in the literature studies on skip length distributions. The only thing we know is that the sum of the skip lengths plus the total number of nodes equals the total number of nodes in the corresponding ordinary trie.

Given a set of n keys, we assume each key $K = k_1k_2...$ is a sequence of symbols from alphabet Σ chosen *independently* at random. Let p_i be the probability of using the ith symbol of Σ. If $p_1 = p_2 = ... = p_{|\Sigma|} = 1/|\Sigma|$, i.e., symbols are uniformly distributed, then the constructed trie is called a *symmetric* trie. Otherwise, it is an *asymmetric* trie. Table 1.1 [Pit85, Reg88, Szp88, Szp90, Szp91] shows some expected asymptotic results for asymmetric tries, where the entropy $h = \sum_{i=1}^{|\Sigma|} p_i \ln 1/p_i$ and $R = \ln \sum_{i=1}^{|\Sigma|} 1/p_i^2$. If symbols are uniformly distributed, $h = R = \ln |\Sigma|$.

	$\lvert\Sigma\rvert$-ary Ordinary Tries	$\lvert\Sigma\rvert$-ary Patricia Tries
Total Nodes, S_n	$n + n/h$	$(n\lvert\Sigma\rvert-1)/(\lvert\Sigma\rvert-1)$ .. $2n-1$
Average Depth, A_n	$\ln(n)\ /h$	$\ln(n)\ /h$
Height, H_n	$2\ln(n)\ /R$	$\ln(n)\ /h$

Table 1.1: Asymmetric Trie Parameters

For binary trees (symmetric), where keys are independent and uniformly distributed, the expected average depth is $1.39\lg(n)$ [Knu73] and the expected height is $2.98\lg(n)$ [Dev87]. Compared with binary tries (symmetric), where A_n=$\lg(n)$ and H_n=$2\lg(n)$, tries are better. In terms of balance property, tries are also better. On average, symmetric tries resemble a complete tree, i.e., an ultimately balanced tree. Symmetric Patricia tries are much better because even H_n is $\log_{|\Sigma|}(n)$. Symmetric tries do not need additional reconstruction to keep them balanced. For asymmetric tries, the situation is slightly different. The entropy, h, changes depth distribution. The more asymmetric the alphabet is, the more skewed a trie is.

So far, most asymptotic results are for the tries whose keys are independent. However, when keys are suffixes from the same text, they are dependent. There is no proper probabilistic model for dependent keys. In §3.2.1 and §6.4.1, we shall show trie parameters measured for the text and spatial data respectively.

1.2.3 Trie Representations

Tabular Representation

The most straightforward implementation of $|\Sigma|$-ary tries is to store each node by an array of $|\Sigma|$ pointers, and a trie by an array of S_n nodes. To save the storage space, all pointers that point to leaf nodes are replaced by pointers pointing to the keys. In other words, leaf nodes are not stored. As a result, a $|\Sigma|$-ary trie is represented by a $|\Sigma|\times(S_n - n)$ table [Fre60, Mor68, Knu73, CS77, RBK89]. In the tabular representation, each table column represents an internal node, and each table entry contains either a column number (internal node), or a *null* pointer (empty), or a pointer to the key (leaf node). The first column is the trie root. To search is to lookup in the table, which takes A_n time on the average to find a key.

Dynamic operations such as changing links and inserting nodes are trivial. Deletion leaves an empty column which can be replaced by the last column. However, the parent node of the last column needs to be changed, too. The parent node can be located by adding a reverse pointer to each node [MF85b]. We propose the following procedure without using the auxiliary structure: (1) search for a keyword by walking down the subtrie rooted in the last column, and (2) search for the keyword once again. The second search passes the node of the last column and its parent.

There is a more subtle implementation which uses three arrays [TY79, FK84, ASU86]. The idea is to shift down each column certain entries and overlap the columns into an array such that no two non-null pointers occupy the same entry. The displacements of columns are stored in the second array. The third array is used to remember column numbers of the pointers in the first array. Even though it is a *NP-complete* problem to minimize the array size [TY79], Tarjan gave a number of effective methods to construct the arrays in size $S_n+|\Sigma|$.

Aoe [Aoe89] reduced the three array implementation to a *double array* structure. By empirical observations, he found that the expected size is indeed $S_n+|\Sigma|$.

Linked List Representation

Tabular representation is prohibitive with large $|\Sigma|$. An alternative is to use dynamic data structures such as linked lists [Bri59, Sus63, AHU83, Jon89, Dun91, HTW92]. In the linked list representation, each trie node is a linked list of outgoing trie links. Each link contains a symbol and a pointer to the left most sibling of the child nodes. In the literature, this data structure is referred to as a *doubly chained tree* after [Sus63]. Conceptually, a doubly chained tree is a binary representation of the $|\Sigma|$-ary trie.

A doubly chained tree is a highly flexible and general structure. Together with dynamic memory allocation techniques, insertions and deletions can be implemented straightforwardly. Even when tries are too big to fit in memory, it is still possible to update tries and keep 100% usage of space at the same time. This can be achieved by using the same trick we suggested for the tabular representation (see the 2nd paragraph of the previous subsection).

A doubly chained tree does not store null outgoing links and therefore requires lesser storage space especially when $|\Sigma|$ gets larger. However, doubly chained trees cannot select a child node in constant time. In the worst case, all outgoing links of a node have to be examined.

Other Representations

In the *compressed trie* [Mal76], each internal node contains a base address and a bit array. Each bit of the bit array indicates whether the corresponding link is a null. All sibling nodes are stored consecutively pointed by the base address of their parent node. *Bitstring* [Ore82] goes further, all nodes are stored consecutively and there are no base addresses. Pointerless representations will be discussed in §2.1.

Severance [Sev74] suggested a number of heuristic implementations which used a tabular approach for the top few levels and doubly chained trees for the remaining levels. The fact is that fan out at the top of tries is much larger than that in the bottom. This also leads to *bucket trie* [Knu73, Lit81]. A bucket trie places b leaf nodes into a bucket, and reduces trie nodes by a factor b. When $b = n$, the extreme case of bucket trie, the whole trie degenerates to a single array of sorted leaf nodes.

This is the case of PAT array [GBYS92]. As the trie nodes are reduced, the binary search of buckets increases. The tradeoff between the bucket size and search time was discussed in [RBK89].

Comparisons

Both tabular and linked list representations require pointers to follow the child nodes. Tables can be searched efficiently but are wasteful in space when they are sparse. Linked lists are more compact but require longer time to be searched. The following calculations we make show the quantity comparison (all on the average) of different representations. We assume tries are symmetric, i.e., all n keywords are distinct and independent, symbols from Σ are uniformly distributed.

The tabular implementation has $(S_n - n) = n/\ln|\Sigma|$ columns and $|\Sigma|$ rows. Totally, it has $2.89n$ table entries (pointers) when $|\Sigma|=2$ and $7.98n$ when $|\Sigma|=26$, the size of English alphabet. Among these entries, S_n of them store pointers pointed either to columns (internal nodes), or to actual data (leaf nodes). The non-null pointer occupancy is $S_n/(|\Sigma|\times(S_n - n)) = (1+\ln|\Sigma|)/|\Sigma|$, e.g., 84.7% when $|\Sigma|=2$ and 16.4% when $|\Sigma|=26$ (very wasteful).

The double array implementation uses two arrays. Each array has $(S_n + |\Sigma|)$ elements [TY79]. The total number of array elements is $4.89n$ when $|\Sigma|=2$ and $2.61n$ when $|\Sigma|=26$. As the case of the tabular representation, leaf nodes need not be stored since the pointers to them can be set directly to the actual keys. Therefore, there are n null pointers.

The doubly chained tree requires two pointers for each of S_n nodes, e.g., $4.89n$ pointers when $|\Sigma|=2$ and $2.61n$ when $|\Sigma|=26$. Among the pointers, $n/\ln|\Sigma|$ are null pointers, e.g., 29.5% are null pointers when $|\Sigma|=2$ and 11.7% when $|\Sigma|=26$.

The expected search time of the doubly chained tree is $\frac{1}{2}dn/|\Sigma|$, where d is the average number of child nodes of internal nodes. Since $dn/\ln|\Sigma| = n-1+n/\ln|\Sigma|$, we have $d \approx \ln|\Sigma|+1$. When $|\Sigma|=2$, this is $1.76\lg n$, slower than a binary search. When $|\Sigma|=26$, this is $0.94\lg n$, slightly better than a binary search but $0.94/0.21=4.5$ times slower than the tabular representation.

Table 1.2 shows comparison among the three representations. As we can see, with a binary alphabet, the tabular representation takes no more space than the doubly chained tree, and still has 15.3% null pointers. With a 26-letter alphabet, the tabular representation requires 3.1 times more space, but is 4.5 times faster as compared with the doubly chained tree. The double array representation combines the merits of the other two representations. However, all these trie implementations take more storage space than the keyword set itself. We need more compact trie representations.

Implementations	Tabular	Doubly Chained Tree	Double Array
Total Entries	$\lvert\Sigma\rvert n/\ln\lvert\Sigma\rvert$	$2n+2n/\ln\lvert\Sigma\rvert$	$2n+2n/\ln\lvert\Sigma\rvert$
Null Pointers	$(\lvert\Sigma\rvert-1)n/\ln\lvert\Sigma\rvert-n$	$n/\ln\lvert\Sigma\rvert$	n
Disk Accesses	$\log_{\lvert\Sigma\rvert} n$	$(\ln n+\log_{\lvert\Sigma\rvert} n)/2$	$\log_{\lvert\Sigma\rvert} n$

Examples		Tabular	Doubly Chained Tree	Double Array
Total (Null/Total)	$\lvert\Sigma\rvert$=2	$2.89n$ (15.3%)	$4.89n$ (29.5%)	$4.89n$ (20.5%)
	$\lvert\Sigma\rvert$=26	$7.98n$ (83.6%)	$2.61n$ (11.7%)	$2.61n$ (38.3%)
Disk Accesses	$\lvert\Sigma\rvert$=2	$\lg n$	$1.76\lg n$	$\lg n$
	$\lvert\Sigma\rvert$=26	$0.21\lg n$	$0.94\lg n$	$0.21\lg n$

Table 1.2: Trie Representation Comparison

Binary Tries

There are several reasons to use binary tries. (1) All keys inside computers are binary numbers. (2) Binary tries are simple in both concept and implementation, e.g., tries in [Lit81] and doubly chained trees are implemented as binary tries. (3) The expected null pointers in binary tries are minimal. This saves not only the storage space but also the number of branching tests. As we have seen, the ratio of null pointer occupancy in the tabular representation increases monotonically when $|\Sigma|$ increases. In the doubly chained tree representation, however, only the youngest child has a null pointer. The null pointer ratio decreases when $|\Sigma|$ increases. (4) The number of nodes in binary Patricia tries is minimal. We shall examine binary trie representations in §3.2.2.

1.3 Text Searching

Text information is very different from common data applications, and conventional database methods do not help in this case. For example, queries such as "find the word `trie` in a text" can be answered by searching a keyword list (or inverted file [Knu73]). But other queries such as "find the phrase `trie method`", "how many words did Shakespeare introduce into the language between 1610–11 [BY89]", "which English words may correspond to the misspelled word `exsample`", involve either searching the whole text or using advanced index structures.

1.3.1 Exact Text Searching

Given a text T of length n and a pattern P of length r, the exact text searching problem is to find the occurrences of P within T. A pattern can be as simple as a keyword, a substring, or as complicated as a regular expression, the longest repetition, the most used word, etc. Text searching may also be stated as to determine whether T is in the language specified by $\Sigma^* P \Sigma^*$, where Σ is the alphabet of the text. Orthogonal to the pattern, the occurrences can be the left most one, all of them, or simply pattern frequency. There are four basic search techniques: (1) sequential search, (2) tree search, (3) trie search, and (4) hashing.

Sequential Search

A sequential search (or *linear search, brute-force search, full text scanning* [Knu73, Gon83, Fal85]) is to check P at every position of T. This straightforward method has no space overhead but requires rn comparison for the unsuccessful search, and $rn/2$ for the average successful search. It is a time consuming process. The general method is to construct a finite automaton M from pattern P, and simulate M on T. The simulation takes $\mathcal{O}(rn)$ comparison if M is *nondeterministic*, or $\mathcal{O}(2^r + n)$ if M is *deterministic* with 2^r states [AHU83].

Some algorithms are more efficient than the automaton approach. The idea behind the *Knuth-Morris-Pratt* algorithm [KMP77] is to use knowledge of the previous symbol comparison. When a mismatch occurs, the position in P yields enough information to recreate the text previously scanned. Thus, by preprocessing P and keeping information in an auxiliary table, we can slide P to the right as far as possible. This algorithm takes $\mathcal{O}(r + n)$ comparison. The *Aho-Corasick* algorithm [AC75] combines this idea with the automaton approach. Their algorithm can search a set of strings simultaneously.

Inspecting pattern P from right to left generates more information [BM77]. Suppose we are comparing T with P from right to left, one symbol at a time. If the current testing symbol $s \notin P$, we do not need to check symbols of T before s and can align P with the next symbol of s. This is the *Boyer-Moore* algorithm [BM77] which takes $c(r+n)$ comparison. Here $c < 1$ and gets smaller when r increases. Horspool [Hor80] demonstrated that *Boyer-Moore* algorithm is indeed an astonishingly fast method for text searching. It even outperforms hardware with built-in search instructions. However, this algorithm has two problems: (1) it finds only the first occurrence, and (2) it may back up through the text. This adds annoying complications for secondary storage.

Some large text files, such as dictionaries and encyclopedias, do not change or are updated at a very low frequency. It is worthwhile to preprocess these files and build indices for them to speed up the search time. Indexing generates an index file which contains a set of keywords and pointers to the text. The search is fast but is often restricted to the words in the *control dictionary*. The following three search techniques belong to this category.

Tree Search

Tree search (*search by key comparison*) is based on key order. *Binary search* of inverted file [Knu73] or PAT array [GBYS92] is a typical tree search. It takes, at most, $r\lg(n)$ symbol comparison to search an index with n keywords, and $\mathcal{O}(n\lg n)$ time to build an index. The PAT array is a sequence of index points sorted according to the text that they point to. Instead of using a control dictionary, it allows to index every possible suffix of a text, and therefore, to search for any arbitrary substring of the text.

Binary search uses an *implicit* binary tree which makes insertions and deletions rather more expensive. This leads us to store keywords in *explicit* trees. A major difficulty with trees is that they may be degenerated with a certain insertion order. A degenerate tree takes $\mathcal{O}(rn)$ symbol comparison. We need algorithms to construct fairly balanced trees, or reorganization methods when trees are badly balanced. The *AVL tree* [AVL62] is a height balanced tree with such a property: any two subtrees at a common node differ in height no more than one. The *B-tree* [BM72] is a balanced $(2m+1)$-ary tree with two properties: (1) each internal node (except the root) has at least $m+1$ descendants, and (2) all leaf nodes appear at the same level. The *2-3 tree* [AHU83] is the special case of *B-tree* with $m=1$. The *Prefix B-tree* [BU77] is a *B-tree* which uses prefixes of keywords as keys.

Trees have received wide attention in the literature and are relatively well understood. Tree structures permit keywords to be added or deleted dynamically, and still remain balanced. Most tree methods take $\mathcal{O}(\lg n)$ time, in the worst case, to insert or delete a keyword.

Trie Search

Instead of comparing a whole key, trie search (*search by key decomposition*) makes use of the digital property of the keywords. It views a keyword as a string and inspects symbols in the string one by one. Trie search can find a pattern in r comparison, which is independent of the index size n. Furthermore, since keywords are stored along the path, not inside the node, tries can handle very long keywords. As we have mentioned in §1.2.2, tries do not need reorganization algorithms. Trie search for text documents will be summarized in §3.5.

Hashing Methods

Hashing is a direct access method which locates keywords by address calculation. To hash is to redistribute access space from a large keyword space to a small storage space. Ideally, we would like to have a hash function which reduces the space of all possible keywords to the space of the presented keywords without conflicts, i.e., a *minimal perfect* hash function. If this was the case, we could retrieve any keyword in constant time.

[CHK85] presented a practical algorithm to build perfect hash tables. In addition to allowing constant retrieval time, hash tables can be updated in constant expected time, and therefore be built incrementally in time $\mathcal{O}(n)$. However, the tables take more than $20n$ bytes of storage. [FHCD92] gave an algorithm to build, practically, a minimal, perfect and *order preserving* table. The method avoided the common problem of wasted space and time. The address calculation takes no more than three accesses to the hash table. The table itself takes a little more than $4n$ bytes, where n, the number of keywords, may go over one million. However, this algorithm works only for static keywords.

Signature files [FC87] is another example of hashing based text searching. In a simple signature file, text words or phrases are hashed into bit patterns, called *word signature*. Word signatures are either used as keywords for a hash function, or stored in a separated file, called *signature file*. A signature file usually takes less than 10% of the text size. We scan the signature file for the querying word signature. However, a positive answer does not necessarily mean that the querying word is in the text. We can either verify it or accept it as a fact. In the latter case, there may be a small number of incorrect answers, or *false drops*. The probability of errors can be controlled by choosing an adequate length of the signature. Signature searching can reject many non-qualifying strings and, in practice, provide a tenfold speedup over sequential searches. However, it is a $\mathcal{O}(n)$ search method.

1.3.2 Approximate String Matching

Misspelling detection, corruption-correction in communication and pattern recognition, the DNA sequence analysis in genetic science require non-exact string matching. The k *approximate string matching* problem specifies, in addition to the given set of n keywords (or n substrings of a text) and the pattern string P of length r, the parameter k of differences (insertions, deletions, substitutions, and/or transpositions) allowed in a match. Various algorithms have been developed to solve the k approximate string matching problem [HD80, SK83, Kuk92].

The basic approach is to search keywords for the *minimum edit distance* using the *dynamic programming technique*. §4.1 will give a short introduction to these two concepts. In Chapter 4, we shall propose an approximate search algorithm which combines the dynamic programming technique with the trie method. Trie methods have only been previously used as an alternative to dynamic programming to improve search time. Algorithms in [MT77, Dun81] use heuristic searches on trie structures and examine a small subset of trie branches. But their algorithms check only restricted typographic errors.

Knuth [Knu73] suggested using two indices, one in the prefix order and the other in the suffix order (reversing keywords). Misspelled keywords agree up to half or more their length in one of the two indices when only one error occurs. No theoretical or empirical results concerning this method are reported. *Soundex* [OR22] is a commonly adopted technique for spelling checkers. The goal is to reduce words into some codes that tend to bring sound-similar keywords together. Soundex codes classify keywords into equivalence classes, and hence can be searched by the exact searching techniques. Baeza-Yates and Perleberg [BYP92] gave an algorithm based on counting symbols of the text, which takes time proportional to the text length, independent of r and k when all symbols in P are different.

The *n-gram* technique is often used in approximate searching for text recognition [Har72, KST92]. The idea is to break keywords down to smaller segments. If a keyword has only one or two mismatches, most of its segments are correct. With a table that contains all the segments and the associated keywords, we may trace back to the right keyword(s).

1.4 Spatial Data Structures

Spatial data are points, lines, etc., in a multi-dimensional space. Usually in an n-dimensional space, data between 0 to n dimensions are acceptable spatial data. Data structures for retrieving alphanumeric data are not adequate for them because range query on multi-keys is one of the common operations. However, to give details on spatial data structures is not within our scope. Comprehensive surveys can be found in [Ben75, Ooi90, Sam90]. Also, this thesis deals only with vector representations of spatial data; we do not review data structures for raster images. We start with some structures for multi-dimensional point data.

1.4.1 Multi-dimensional Point Structures

The *kd-tree* [Ben75] is a generalization of the one dimensional binary tree. The first level discriminator is the the first attribute of the data. All data with the first attribute values less than, equal to or greater than a certain value go to the left subtree, root or the right subtree respectively. The second level discriminators are the second attribute. The second level nodes are constructed according to the values of the second attribute. This process cycles recursively among all the attributes until there is only one datum left. Range search with kd-tree is straightforward. The kd-tree has been the subject of intensive studies and many variants have been proposed to improve the performance such as clustering, storage efficiency and balancing.

The *kd-trie* [Ore82] is a generalization of the one dimensional binary trie. The binary trie divides the space (not the data presented) by successive powers of two. This corresponds to using the first bit to determine if the datum is in the first or second half of the space, using the second bit to determine if it is in the first or second half of this subspace, and so on. A kd-trie is a binary trie with keys that consists of the data coordinates interleaved bit-by-bit. A point *quadtrie* [FB74] is a 4-ary trie using the interleaved keys.

In multi-dimensional hashing, data space is divided into disjoint regions. Data contained in one region are stored in one or few buckets. The *grid file* [NHS84, Hin85] partitions k-dimensional space into orthogonal grids. The grid boundaries on each dimension are stored in k one dimensional arrays, called *scales*. Bucket addresses are stored in a k-dimensional array, called *grid directory*. Scales are much smaller and can be stored in memory. Consulting the scales, we can find the subscripts to the grid directory and then find the bucket address. A major problem is the storage for the directory. *Multipaging* [MO82] uses k tidy functions to replace the k-dimensional directory. *EXCELL* [Tam82] requires grids to be of equal size. It simplifies the grid partition but uses larger directories.

Another approach is to organize k-dimensional data according to a certain linear order. The idea is to map the data set from k-dimensional space to one dimensional space, and then to use a point data structure such as a B-tree to index them. The commonly used orders (recursive space filling curves, locational keys) are: *Z-order* [OM84], *Hilbert order* and *Sierpinski order* [NB94]. The *Linear quadtree* [Gar82] and *QUILT* [SSN90] used this approach.

1.4.2 Non-point Structures

Spatial features such as roads and lakes in maps consist of point sets. They usually do not form any fixed shape. It is expensive to perform queries on their exact location and extent, and hence we often use *minimum bounding rectangles* (MBR) or other conservative approximations [BHKS93] to filter and approximate irregular shapes. However, we still need data structures to handle both location and extent. Access methods for non-point data can be classified into three groups [Ooi90]:

Transformation Features in a k-dimensional space are represented as points in a higher ($> k$) dimensional space [SK88]. Coordinates for extent are taken as different dimensions. For edges, intervals or MBRs, it could be called, more distinctively, *dimension doubling*. More generally, we speak of *dimension raising* [MS94].

Clipping Data space is partitioned into pairwise disjoint subspaces. If a feature intersects with a set of subspaces, either its identifier or the feature itself is included (duplicated) in each of the subspaces.

Overlapping Data space is covered by a set of rectangle scheme such that features are totally covered by one of the rectangle schemes. The rectangle schemes may overlap with each others.

R-tree [Gut84] is a generalization of the one dimensional B-tree, and hence it is height balanced. In an R-tree, each leaf node contains a pointer to an MBR, and each internal node contains a rectangle scheme that covers all the rectangles in the subtree. In searching, the decision whether to visit a subtree depends on whether its rectangle scheme overlaps the query region. It is common that several rectangle schemes overlap the query region, and this results in the traversal of several subtrees. Minimization of the overlaps of the rectangle scheme as well as the coverage of these rectangles is of primary importance in R-tree construction.

The *segment tree* [Sam90] is an example of clipping. A *segment tree* is a one dimensional region quadtree for intervals. Its nodes contain an interval identifier if and only if the interval it covers is contained in the interval indicated and the interval of its parent node is not contained in the interval. In other words, each identifier may be stored in many tree nodes (clipping). When a segment tree is searched, only the nodes that intersect with the query interval are visited.

RR quadtree [Sha86] is a quadtrie structure that uses the clipping technique for rectangles. It splits unpartitioned data space into quadrants until the subspace either intersects with just one rectangle, or covers a set of rectangles which overlap each other. All rectangles are associated with leaf nodes. When none of the rectangles overlap, each node of RR quadtree contains a part of one rectangle. The storage requirements for RR quadtree are very high.

PLOP-Hashing [KS88] is a grid file extension for non-point data. The method is a multi-dimensional dynamic hashing based on Piecewise Linear Order Preserving (PLOP) hashing. *PLOP-Hashing* partitions data space into orthogonal grids and uses k binary trees to replace the scales in grid file. Binary trees map order information oriented along axes to grid numbers. Two extra values are stored in each leaf node to bind the objects whose centroid are in the corresponding grid. Merrett [MD85] combined the multipaging and clipping techniques to represent diagrams which consist of sequences of small edges.

1.4.3 Summary

Spatial searching requires multi-key searching. However, features often cover irregular areas in multi-dimensional space and cannot be solely represented by point locations. Conventional data structures may not be suitable to non-point data.

Besides the technique to approximate irregular shapes by MBRs, three major techniques have been used to handle non-point data. In the transformation technique, non-point data become point data of a higher dimensional space. It requires no alteration of data structures. However, spatially close data may be torn apart in the higher dimension space. As a consequence, searching may be slow, especially when we deal with secondary storage. The most important property of the clipping technique is that data structures used can be direct extensions of point data structures. However, it requires extra storage and becomes more expensive in insertion and deletion. To reduce duplications, feature identifiers are usually maintained in the structures and features are stored in another file. This results in additional disk access. For the overlapping technique, maintaining the "minimal" overlap is very difficult. Ineffective overlapping schemes tend to overlap and results in searching more paths.

Both trees and tries are hierarchical structures. Tree methods such as B-trees, kd-trees and R-trees [Gut84] are height balanced trees to limit the worst-case performance. However, trees divide the data often by the medians of the data presented. They do not preserve the scale of the data space at each level. Trie methods recursively divide the data space in half, and thus preserve the scale of subspaces at each level. They allow us to read tries down to a certain depth, and retrieve only this subset of the file.

1.5 Thesis Outline

This thesis is organized as follows. Chapter 1 shows the motivation of the thesis and the problem domains. It introduces trie methods and their applications, parameters and representations in general. This chapter also gives a literature overview on text searching and spatial data structures.

After the introduction, we present the underlying trie structures which are used throughout the thesis. There are four topic chapters, one for each trie application: exact text searching, approximate string matching, map displaying, and spatial data querying. All applications deal with very large collections of data, e.g., gigabytes or terabytes. Two problems are discussed in great detail: the efficient use of storage and the I/O performance.

Chapter 2 presents three trie structures: *FuTrie* for binary full trie, *OrTrie* for binary ordinary trie, and *PaTrie* for binary Patricia trie. It deals with the problems of maintaining trie structures in secondary storage. Our goal is to partition a trie into pages and to operate at the cost of only few disk accesses and a small amount of data transport. Problems related to large trie construction are also considered.

Chapter 3 examines trie methods for text searching. The major problem with tries is that their sizes can be even larger than the text. Recent work on tries has focussed on reducing the trie size. We show that our *PaTrie* index achieves size factors of less than 3 for 100 million keys, as compared with 3.4 for the best previous method.

Chapter 4 discusses approximate string matching. The discussion is restricted to search keywords for the best match, e.g., the spelling check problem. The cost of the algorithm we present is independent of the dictionary size. This is the first known algorithm that achieves the time complexity.

Chapter 5 describes *ZoomTrie*, a trie structure for storing spatial data such as maps. *ZoomTrie* gives a continuous zoom, say, from the full details of a map of many gigabytes of data, up to a mere outline, while storing only one copy of the map and reading only the amount of data to be displayed. The discussion focuses on polyline maps. But the method can be applied to any set of multi-dimensional homogeneous features.

Chapter 6 demonstrates that *ZoomTrie* can be used not only in the ubiquitous operations of displaying and plotting, but also in geometrical queries and other spatial data processing. With examples, we show that *ZoomTrie* can be used as an index structure to answer various queries. The idea is to examine data in the order of increasing resolution, and for each resolution level, some part of search space is eliminated from the consideration. The algorithms and results shown in this chapter are for line queries: line-point, line-line and line-region. But they can be extended for point queries and region queries.

Chapter 7 summarizes the contributions of this thesis and future research that we propose.

Chapter 2

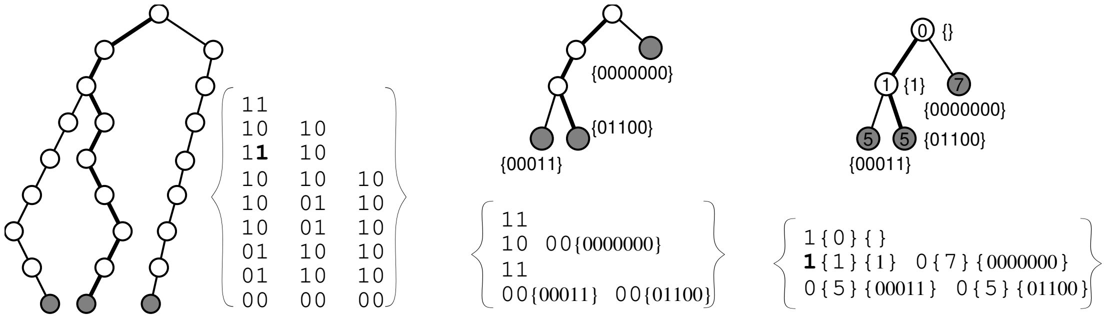
Trie Organization

In this chapter, we shall present three pointerless trie structures: *FuTrie* for the binary full trie [CS77], *OrTrie* for the binary ordinary trie [Fre60] and *PaTrie* for the binary Patricia trie [Mor68]. Our trie organizations have two distinctive features: (1) they store no pointers and require no more than two bits per node, and (2) they partition tries into pages and are suitable for secondary storage. Throughout this thesis, we shall use our trie structures either as auxiliary structures for indexing data, or as main structures for storing and indexing data.

Problems related to the new trie structures such as trie partitioning, trie searching and static trie construction will be discussed in this chapter. From this chapter on, we shall consider binary tries exclusively except for some examples in chapter 4.

2.1 Pointerless Representations

The pointerless trie representations of this section and the partition strategy of the next section are based on the work by Orenstein [Ore82, Ore83]. We shall extend his trie representation for Patricia trie and use one bit plus either a skip or a start for each node. We shall also make the pointerless trie representations capable of storing complete data elements in contrast to the previous versions that store only partial data elements.



(a) *FuTrie* (b) *OrTrie* (c) *PaTrie*

Figure 2.1: Tries and Bitstrings

2.1.1 *FuTrie*

A *FuTrie* is a binary tree whose nodes do not store information and whose links are labelled with *0* for the left links and *1* for the right links. The branching decision for internal nodes at level i is made according to the ith bit of the search string. If the test bit is *0*, the search goes to the left descendant, or else it goes to the right descendant. Each root-to-leaf path corresponds, one to one, to a key string. For example, Figure 2.1 (a) shows a *FuTrie* constructed over key strings *00000011, 00101100*, and *10000000*. The darkened path gives the sequence of *llrlrrll*, where *l* means *left* and *r* means *right*. This corresponds to *00101100*.

Orenstein represented the *FuTrie* using *bitstring* [Ore82], a pointerless structure. The bitstring is a list of trie nodes organized from the root level to the lowest leaf level, and from the left most node of a level to the right most node. Each trie node has two bits. Bit pairs `11`, `10` and `01` represent an internal node with two descendants, one left descendant only and one right descendant only respectively. `00` represents an external node (leaf). Figure 2.1 (a) also shows the bitstring of the *FuTrie*.

Each **on bit** of the bitstring represents an outgoing link. The jth on bit at level i is the link to the jth node of level i+1. The child node has a displacement of $2j-2$ bits. For example, in the bitstring of Figure 2.1 (a), the bold bit at level 3 is the second on bit. So it is the link to the second node of the next level. The child node is the $2j-1$st and the $2j$th bits at level 4. However, in order to access the grand-child nodes, we have to scan $2j$ bits to count the on bits of level 4.

Table 2.1 defines the *FuTrie* structure.

PointerlessTrie	::=	Array [ ] of *TrieLevel*
TrieLevel	::=	Array [ ] of *TrieNode*
TrieNode	::=	$\left\{ \begin{array}{ll} 11 & \swarrow\searrow \\ 10 & \swarrow \\ 01 & \searrow \\ 00 & \bullet \end{array} \right\}$

Table 2.1: *FuTrie* Structure

2.1.2 *OrTrie*

An *OrTrie* is a *pruned FuTrie* in which all node chains that lead to leaves have been pruned. Figure 2.1 (b) shows the *OrTrie* transformed from 2.1 (a). The darkened path, *001*, gives only a prefix of key *00101100*. To recover the whole key string, each *OrTrie* leaf stores either a pointer (*start*) to the key, or the remaining bits of the key (*suffix*). Note that leaves are placed as high as possible in the *OrTrie*, and that a trie does not continue below the levels at which a subtrie contains only one key.

OrTrie leaves are varying in size. If all keys are of the same length, say d bits each, the suffix length at level i will be $d-i+1$ bits long. However, for variable length keys, we need a counter to remember the suffix length.

For *OrTrie*, the bitstring and the implicit addressing are the same as for *FuTrie*, except that the suffix information is excluded when the bits are being counted.

Table 2.2 defines the *OrTrie* structure.

PointerlessTrie	::=	Array [ ] of *TrieLevel*
TrieLevel	::=	Array [ ] of *TrieNode*
TrieNode	::=	$\left\{ \begin{array}{lr} 11 & \swarrow\searrow \\ 10 & \swarrow \\ 01 & \searrow \\ 00 \left\{ \begin{array}{l} \{\text{length}\}\{\text{suffix}\} \\ \text{or } \{\text{start}\} \end{array} \right\} & \bullet \end{array} \right\}$

Table 2.2: *OrTrie* Structure

2.1.3 *PaTrie*

A *PaTrie* is also a *pruned FuTrie*. This time, not only the node chains leading to leaves but also all the internal node chains are pruned. Figure 2.1 (c) shows the *PaTrie* transformed from 2.1 (a). *PaTrie* has only n–1 internal nodes, where n is the total number of leaves (or keys). It gives a much shallower trie.

PaTrie adds costs of storing either *height* — the level number in the corresponding *FuTrie*, or *skip* — the number of removed one-way nodes from the parent, in each internal node. For example, the internal nodes in Figure 2.1 (c) are skips. To choose an outgoing link , the height tells which bit of the search key to inspect, while the skip tells how many bits from the last inspected bit to skip over. Skips are more compact to store than heights, but we shall also use the height later.

If we use the *PaTrie* as an index structure, we need only to store skips in the internal nodes and starts (pointers to the data) in the external nodes. The complete key strings can be recovered by following starts to keys. However, if we want to use *PaTrie* as a data structure to store the entire keys, we need to remember the skipped key bits in both the internal and external nodes. For example, for each node of Figure 2.1 (c), the second number in the parentheses is the pruned key bits, and the first number is the length (in bits) of the second number.

We extend the bitstring to represent *PaTrie*s. Since *PaTrie* nodes have either two descendants or none, one bit is sufficient to distinguish them, i.e., `1` for the internal node and `0` for the external node. In addition to the bit, each internal node has a skip, and optionally, the skipped key bits. Each external node has a start, or alternatively, a length counter and the remaining key bits (suffix).

Table 2.3 defines the *PaTrie* structure.

$$
\begin{array}{rcl}
\textit{PointerlessTrie} & ::= & \text{Array [ ] of } \textit{TrieLevel} \\
\textit{TrieLevel} & ::= & \text{Array [ ] of } \textit{TrieNode} \\
\textit{TrieNode} & ::= & \left\{ \begin{array}{ll}
1 \left\{ \begin{array}{l} \{\text{skip}\}\{\text{substring}\} \\ \text{or } \{\text{skip}\} \end{array} \right\} & \swarrow\searrow \\
0 \left\{ \begin{array}{l} \{\text{length}\}\{\text{suffix}\} \\ \text{or } \{\text{start}\} \end{array} \right\} & \bullet
\end{array} \right\}
\end{array}
$$

Table 2.3: *PaTrie* Structure

Each **on bit** represents two outgoing links. The jth on bit at level i connects to the $2j-1$st node and the $2j$th node of level $i+1$. The left child is the $2j-1$st bit. The right child is the $2j$th bit. The bits for the skips and suffixes are not counted. For example, the bold bit in Figure 2.1 (c) is the first on bit of level 2. So, its left child is the first bit of level 3, and the right child is the second bit if we do not count the bits inside the parentheses.

2.2 Trie Partitioning

Unfortunately, to find the jth on bit involves scanning the bitstring. This scanning has to be done for every level and is expensive. To avoid scanning the entire trie, we slice the trie into layers of k levels each, and then chop each layer into pages of subtries. In each page, the child nodes of each level are either entirely on or entirely off that page. In other words, links can only cross the horizontal boundaries of pages, not the vertical boundaries.

The partitioned trie restricts the sequential search within the pages, and reads no more than one page per layer per search. This partition strategy was originally proposed by Orenstein [Ore83] for *FuTrie*s and *OrTrie*s. It also works for *PaTrie*s. Figure 2.2 gives an example of a paged *PaTrie* with k=3.

Each page in Figure 2.2 has two integers, $Tcount$ and $Bcount$. The two counters contain the number of links into and out of the page layer. The counting stops right before the page they belong to. $Tcount$ and $Bcount$ are used to find, for example, the right link of node X, i.e., the second link leaving the page. This link is also the 2+18=20th link ($Bcount$) out of the page layer. The fourth page of the next page layer contains link 20. Link 20 is also the 20−16=4th link ($Tcount$) into the fourth page.

Once again, to avoid scanning page layers for $Tcount$s, we collect $Tcount$s of the sibling pages into the parent page. As shown in Figure 2.3, each page contains a $Bcount$ and a set of $Tcount$s of the child pages and pointers to the pages. Table 2.4 defines the paged pointerless trie structure.

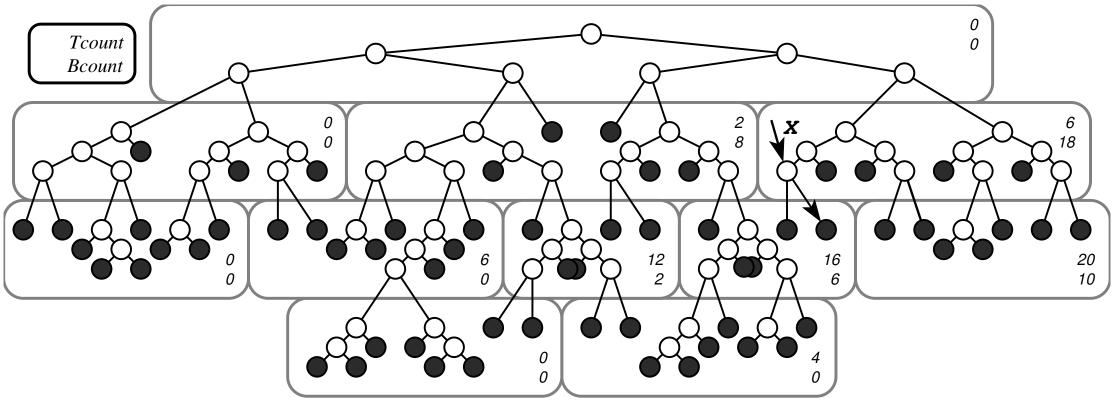


Figure 2.2: Paged *PaTrie*

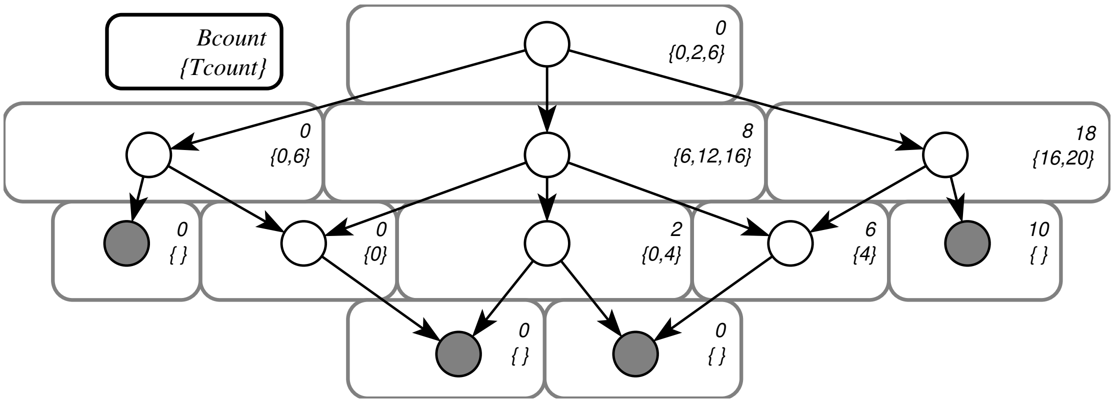


Figure 2.3: Page Structure

PagedTrie ::= Array [ ] of *TriePage*
TriePage ::= Record
  Bcount : *Integer*;
  LinkTo : Array [ ] of Record
    Tcount : *Integer*;
    Page : $\uparrow$*TriePage*;
  End
  SubTries : *Bitstring*;
End

Table 2.4: Paged Trie Structure

2.3 Trie Searching

There are four variables associated with the node being searched. Variable *iLevel* is the trie level of the node, $jNode$ the node number in the level, $Tnode$ the total nodes of the level, and *OutLink* the number of outgoing links counted from the left most node of the level. If j is the on bit number of the node, we have j equal *OutLink* for *FuTrie* and *OrTrie* or $\frac{1}{2}$*OutLink* for *PaTrie* (see §2.1).

To find the right child node (if it exists), we scan to *OutLink* nodes in level *iLevel*+1. A new level starts when we reach the last node ($Tnode$) of the previous level. $Tnode$ is the total outgoing links of the previous level.

If *iLevel* is the last level of the page, we need to search the *LinkTo* list for the child page. *Bcount*+*OutLink* is the outgoing link in terms of the whole trie. The result minus $Tcount$ of the child page is the incoming link in term of the child page. *LinkTo.Page* points to the child page. Algorithm 2.1 spells out the searching procedure for the paged pointerless trie.

2.4 Trie Construction

Large trie construction is a problem. As we have pointed out in §1.1, even with one random access per key insertion (a very minimal requirement), the total disk time for constructing a trie of n=100 million keys would be 100M×20ms ≈ 23.1 days, assuming 20ms per disk access. This is not acceptable in practice. We need construction algorithms tuned for secondary storage.

Another problem with this dynamic or updating approach is that we cannot achieve fully occupied pages. Usually a dynamic method may require as much as doubled space requirement reported in the static method. Updating *FuTrie*s or *OrTrie*s is straightforward and will not be discussed. Updating *PaTrie*s will be discussed in §3.3.1

To build the whole trie, we can do much better than the incremental algorithm by careful use of sorting. This approach is based on two theorems. *Theorem 1*: the list of ordered keys is equivalent to the list of leaf nodes obtained by *inorder* traversal of

Algorithm 2.1 Pointerless Trie: Searching Child Node

```
Type Anode = Record
               Page :  ↑TriePage;
               iLevel, jNode :  Integer;                  /* node location */
               Tnode, OutLink :  Integer;              /* level information */
               skip, start, height :  Integer;         /* node information */
            End;

Procedure GetChild( var n : Anode; which);
  begin
    Position := n.OutLink;                     /* calculate the child position */
    if (which = left) and (n has right child) then
      Position := Position - 1;

    while (n.jNode < n.Tnode) do           /* scan to the last node of iLevel */
      m := the next node;
      n.jNode := n.jNode + 1;
      n.OutLink := n.OutLink + OutLinks( m);

    if ((n.iLevel mod k) = 0) then         /* child fall out the current page */
      Position := Position + n.Page↑ Bcount;                   /* in whole trie */
      Find i such that:                       /* search for the connected page */
        (n.Page↑ LinkTo.[i].Tcount ≤ Position) and
        (n.Page↑ LinkTo.[i+1].Tcount > Position);

      Position := Position - n.Page↑ LinkTo[i].Tcount;             /* in page */
      n.OutLink := n.OutLink + n.Page↑ Bcount
                             - n.Page↑ LinkTo[i].Tcount;
      n.Page := n.Page↑ LinkTo[i].Page;                  /* read the new page */

    n.Tnode := n.OutLink;                                /* start a new level */
    n.iLevel := n.iLevel + 1;
    n.jNode := 0; n.OutLink := 0;
    while (n.jNode < Position) do                     /* scan to the child node */
      m := Read the next node;
      n.jNode := n.jNode + 1;
      n.OutLink := n.OutLink + OutLinks( m);
  end;
```

the trie structures. *Theorem 2*: given such an ordered list, we can uniquely construct a trie structure. We shall state formally and prove the two theorems in §2.4.3 for the *PaTrie* structure. However, the proofs are also valid for the *FuTrie* and *OrTrie* because all three tries can be transformed from one to another. When a trie is constructed from the ordered list, it grows only in one direction. This avoids random disk access.

In the following discussion, we assume no key string is a prefix of another key. String comparison uses lexicographic ordering.

2.4.1 *FuTrie* Construction

We construct *FuTrie* as a whole by scanning the sorted key strings. In general, the procedure is: (1) sort the key strings in a lexicographic order, (2) treat each key string as a *FuTrie* by writing it vertically and changing `0` to `10`, and `1` to `01`, and (3) append each key string to the *FuTrie* under construction. For example, given *00000011, 00101100*, and *10000000*, after steps (1) and (2), we have:

$$\begin{bmatrix} \text{key}_1 & \mathit{00000011} \\ \text{key}_2 & \mathit{00101100} \\ \text{key}_3 & \mathit{10000000} \end{bmatrix} \Longrightarrow \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \Longrightarrow \begin{bmatrix} 10 \\ 10 \\ 10 \\ 10 \\ 10 \\ 10 \\ 01 \\ 01 \end{bmatrix} \begin{bmatrix} 10 \\ 10 \\ 01 \\ 10 \\ 01 \\ 01 \\ 10 \\ 10 \end{bmatrix} \begin{bmatrix} 01 \\ 10 \\ 10 \\ 10 \\ 10 \\ 10 \\ 10 \\ 10 \end{bmatrix}$$

Step (3) merges, one by one, *FuTrie*s of the sorted key strings from the smallest key to the largest key. The merge procedure scans both the previous and the current key strings from top down until the bits of the two strings are different, i.e., the bits of the previous key string is `10` and the bits of the current key string is `01` (because of lexicographic ordering). At this moment, the merge procedure goes to the second phase: (3.1) change bits `10` of the *FuTrie* in constructing to bits `11`, (3.1) copy the remaining bits of the current key string to the *FuTrie* in constructing until the end of the key string. Continuing from the above example, after step (3), we have:

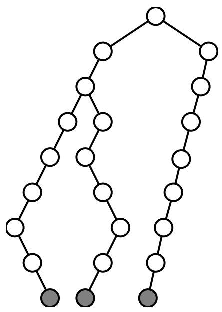
Algorithm 2.2 *FuTrie*: Appending Key

```
Var last :  Array [] of Integer;                                  /* last node of trie T */
Procedure FuTrieMerge( var T:FuTrie; s:string);
  begin
    rootLevel := 1; leafLevel := length(s)+1;
    for i := rootLevel to leafLevel-1 do                          /* shared path */
      if T[i][last[i]] = 10 and s[i] = 1 then
        bifurcation := i;
        break;

    T[ bifurcation][ last[bifurcation]] := 11;                    /* bifurcation point */
    for i := bifurcation+1 to leafLevel-1 do                      /* appended path */
      last[i] := last[i] + 1;
      T[i][ last[i]] := FuTrie(s)[i][1];                          /* treat s as a FuTrie */

    last[ leafLevel] := last[ leafLevel]+1;                       /* create leaf node */
    T[ leafLevel][ last[leafLevel]] := 11;
  end;
```

$$\Longrightarrow \begin{bmatrix} 10 & \\ 10 & \\ 11 & \\ 10 & 10 \\ 10 & 01 \\ 10 & 01 \\ 01 & 10 \\ 01 & 10 \end{bmatrix} \begin{bmatrix} 01 \\ 10 \\ 10 \\ 10 \\ 10 \\ 10 \\ 10 \\ 10 \end{bmatrix} \Longrightarrow \begin{bmatrix} 11 & & \\ 10 & 10 & \\ 11 & 10 & \\ 10 & 10 & 10 \\ 10 & 01 & 10 \\ 10 & 01 & 10 \\ 01 & 10 & 10 \\ 01 & 10 & 10 \end{bmatrix} \Longleftrightarrow$$



Algorithm 2.2 shows the merge procedure. In *FuTrieMerge*, trie nodes are appended (not inserted) to each level. The constructed nodes will no longer be referred to except the last one of each level. Therefore, only the current string and the last node of each level need to stay inside the main memory. The rest of the constructed trie can be dumped to the disk.

Trie paging can be done during the merge. When nodes in a page layer exceed a preset limit (say 8KB), we create a trie page and write it to the disk. To prevent links from crossing the vertical boundaries of a page, we need to check the next string. Only the binary nodes may cause links to cross the vertical boundaries.

Our algorithm reads the ordered keys once and writes the trie once. Assume 16 bytes (4 coordinates, see §5.1.2) per key, $16n$ bytes for the *FuTrie*, $20\mu s$ seek time, $1\mu s$ transfer rate, and 8K bytes per I/O buffer, this method takes:

$$\frac{(16+16)n}{8K} \times \frac{20+8}{3600 \times 10^3} = 3.1 \text{ hours}$$

of I/O time (excluding the sorting time) to build a *FuTrie* of $n = 10^8$ strings.

2.4.2 *OrTrie* Construction

The *OrTrie* can be constructed in the same way as the *FuTrie*, except that we do not construct subtries after the last binary nodes. Instead, we construct leaf nodes with the truncated suffixes. We shall not elaborate on the algorithm.

2.4.3 *PaTrie* Construction

The following theorem establishes that adjacent keys in the inorder traversal of a *PaTrie* are ordered and have common prefixes up to bit h, where h is the height at the internal node falling between them in the traversal. It follows that a *PaTrie* has a unique list representation. The second theorem shows that this list representation gives a unique *PaTrie*.

Let $K_n = s_1, s_2, ..., s_n$, be a set of $n>1$ key strings. Let s_i^j be the jth bit of string s_i, $List(K_n)$ be a list $[s_1, h_1, s_2, h_2, ..., h_{n-1}, s_n]$ such that K_n is ordered and h_i is the length of the longest common prefix of s_i and s_{i+1}. Let $PaTrie(K_n)$ be a *PaTrie* constructed over K_n. We assume the *PaTrie* stores heights in the internal nodes and starts in the external nodes (see §2.1.3). Let $Trav($ *PaTrie*$)$ be the list of starts and heights of the inorder traversal of the *PaTrie*.

Theorem 1 *For any* (s_i, h_i, s_{i+1}), $(1 \le i < n)$, *in* $Trav(PaTrie(K_n))$, *we have:*

$$\begin{cases} s_i^j = 0 & \text{if } j = h_i \\ s_{i+1}^j = 1 & \text{if } j = h_i \\ s_i^j = s_{i+1}^j & \text{if } 1 \le j < h_i \end{cases} \tag{A}$$

Proof:

- When $n = 2$, $Trav(PaTrie(K_2)) = [s_1, h_1, s_2]$. Since h_1, the height of the only internal node, is the discriminant bit position of s_1 and s_2, and s_1 is the left leaf node, (A) is true for $[s_1, h_1, s_2]$.
- Suppose $\forall_{1 \leq i < n-1}(s_i, h_i, s_{i+1}) \in Trav(PaTrie(K_{n-1})) \longrightarrow$ (A).
- For any $Trav(PaTrie(K_n))$, let h be the height of $PaTrie(K_n)$'s root. Let $PaTrie(K_\ell)$ be the left subtrie, and $PaTrie(K_r)$ be the right subtrie. We have: $\ell + r = n$, and $Trav(PaTrie(K_n)) =$

  $$[Trav(PaTrie(K_\ell)), h, Trav(PaTrie(K_r))] = [s_1, ..., s_\ell, h, s_{\ell+1}, ..., s_{\ell+r}]$$

  By hypothesis, (A) is true for all $[s_i, h_i, s_{i+1}]$, except $[s_\ell, h, s_{\ell+1}]$.
  By definition, h is the discriminant bit position for all $s_x \in K_\ell$ and $s_y \in K_r$.
  Hence, (A) is true for (s_ℓ, h, s_r). □

Corollary: $\forall_{PaTrie(K_n)} Trav(PaTrie(K_n)) = List(K_n)$.

Proof: (A) implies strings in $Trav(PaTrie(K_n))$ are totally ordered, i.e., $s_1 < s_2 < ... < s_n$, and h_i of $Trav(PaTrie(K_n))$ is the discriminant bit position of s_i and s_{i+1}. By definition, $Trav(Patr(K_n)) = List(K_n)$. □

Theorem 2 $\forall_{n>1} List(K_n) \longrightarrow$ *unique* $PaTrie(K_n)$

Proof:

- When $n = 2$, $List(K_2) = [s_1, h_1, s_2]$ has only one *PaTrie*.
- Suppose $\forall_{1<i<n} List(K_i) \longrightarrow$ unique $PaTrie(K_i)$.
- For any $List(K_n) = [s_1, h_1, s_2, ..., h_{n-1}, s_n]$, let $h \in List(K_n)$ be the smallest height. Let $List(K_\ell) = [s_1, h_1, s_2, ..., h_{\ell-1}, s_\ell]$ and $List(K_r) = [s_{\ell+1}, h_{\ell+1}, ..., s_n]$. Since any height of a *PaTrie* is strictly larger than the height of its Parent nodes, h must exist and it is the height of the root. Therefore, we have: $List(K_n) = Trav(PaTrie(K_n)) =$

  $$[Trav(PaTrie(K_\ell)), h, Trav(PaTrie(K_r))] = [List(K_\ell), h, List(K_r)]$$

  By the hypothesis, both $PaTrie(K_\ell)$ and $PaTrie(K_r)$ are unique.
  Hence $PaTrie(K_n)$ is unique. □

Algorithm 2.3 *PaTrie*: Parsing $List(K_n)$

```
Procedure PaTrieBuild( var T:PaTrie; list:List(Kn));
  Var heightStack :  Stack of Integer;  trieStack :  Stack of PaTrie;
      left, right :  PaTrie;  currItem :  Integer;
  begin;
    currItem := read List;              /* List(Kn) = 0, s1, h1, s2, ..., h(n-1), sn, 0 */
    push( currItem, heightStack);
    currItem := read next List;
    do forever
      if (currItem is a start) then
        push( currItem, trieStack);                  /* create an one node trie */
        currItem := read next List;
      else
      if (both currItem and top of heightStack is 0) then
        T := pop( trieStack);                   /* end of PaTrie construction */
        break;
      else                                         /* current item is a height */
      if (top of heightStack) < currItem then                /* no decision yet */
        push( currItem, heightStack);
        currItem := read next List;
      else repeat                                           /* find a subtrie */
        left := pop( trieStack);
        right := pop( trieStack);
        convert heights of left and right to skip;
        newTrie := left↙pop(heightStack)↘right;
        push( newTrie, trieStack);
      until (top of heightStack) <= (previous height);        /* no more subtries */
  end;
```

Corollary: $PaTrie(K_n)$ is unique. *Proof:* $List(K_n)$ is unique for any given K_n. □

We preprocess key strings and construct the *PaTrie* as a whole. The procedures are: (1) sort K_n and produce $List(K_n)$, (2) treat $List(K_n)$ as an expression by interpreting heights as operators and starts as operands, (3) parse $List(K_n)$, and (4) page the constructed *PaTrie*.

As we have proved in Theorem 2, constructing *PaTrie* from the $List(K_n)$ is a special case of parsing expressions with operator precedence [ASU86]. Here we have only binary operators and no ambiguity (*PaTrie* is unique). The higher precedence corresponds to the larger height. Algorithm 2.3 shows the procedure of *PaTrie* construction. After parsing, a *PaTrie* is represented as a list of node levels. As for paging *FuTrie*, *PaTrie* can be paged in a single pass.

The I/O cost of Algorithm 2.3 can be calculated as follows. The input is n heights and n starts. The height size is bounded by $\lg(H_n)$. The start size is $\lg n$. When a trie level becomes too big, we write the level to the disk and replace it by a page pointer. The output is n starts, $(n-1)$ heights, and $2n$ bits plus negligible pointers. Assuming $20\mu s$ seek time, $1\mu s$ transfer rate, and $8K$ bytes per page, the cost is:

$$\frac{2n(\lg n + \lg(16 \times 8) + 1)}{8 \times 8K} \times \left(\frac{1}{B} + 1\right) \times \frac{20 + 8}{60 \times 10^3} = 50.4 \text{ minutes}$$

of I/O time (excluding the sorting time) to build a *PaTrie* of $n = 10^8$ keys and 16 bytes each. Here B is the page size in 8KB, large enough for $1/B \ll 1$.

2.5 Summary

We have proposed a set of three new trie organizations. In the next four chapters, we shall apply them to applications in exact text searching, approximate string matching, map data displaying, and spatial data querying in general. All these applications deal with bulk data, and no matter whether we use tries as index structures or data structures, the trie size is always a critical issue. Our trie organizations require one bit per node for the *PaTrie*, two bits per node for the *FuTrie* or *OrTrie*, and there are no pointers. Our representations are smaller than all known methods.

Another problem with tries is that they are often too large to be stored in the main memory. We have given a method to partition tries into pages for secondary storage. To search our trie down to level i, we need to read i/k trie pages. Here k is a preset number and $k = 8$ in all our implementations.

Even with one random disk access per key insertion, the total I/O time will be 23 days for building a trie of 100M keys. We have presented algorithms which minimize random disk access. Our algorithms carefully use the order properties among the key strings, and construct tries as a whole. The expected I/O time is 3.1 hours for constructing a *FuTrie* of 100 million keys, and 50.4 minutes for a *PaTrie*.

Chapter 3

Exact Text Searching

In this chapter, we shall examine trie index structures in searching very large texts. Trie methods give search costs which are often proportional only to the length of the string being sought, and in the worst case, to the logarithm of the size of the text being searched. For very large texts, trie methods are indispensable.

Index tries for text searching have been used by Morrison [Mor68] and exploited by Gonnet *et al.* [GBYS92, Tom92] for the implementation of the electronic version of the *New OED*. However, a major difficulty with tries is that the index generated can be even larger than the text. For example, Morrison's Patricia index could be eighteen times the size of the text.

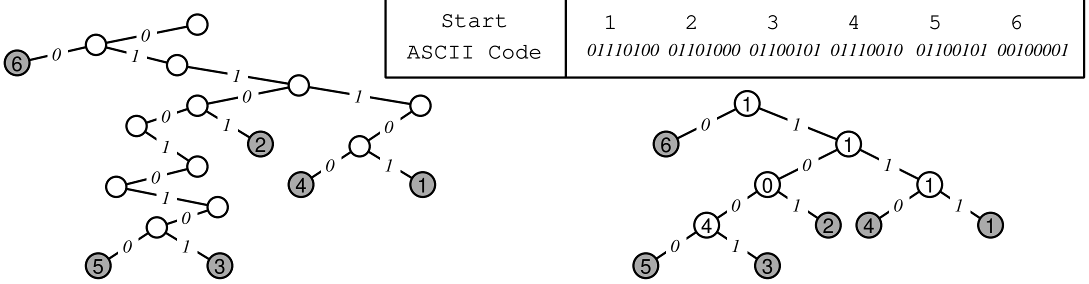
When applying *PaTrie* to index 100 million keys, our experiments show size factors of less than 3, as compared with 3.4 for the best previous method. Our measurements also show expected access costs of 0.1 second, and construction times of 18 to 55 hours, depending on the text characteristics. We shall show that our index structure can handle dynamic texts, and will give new algorithms for text trie inserting and deleting. This chapter is an extended version of [MS93a].

3.1 Text Trie

We follow Gonnet in using semi-infinite strings (or *sistring* [Gon88]), in which a text is viewed as a very long sequence of letters without concern for the structure. Sistrings are suffixes of the text. For example, sistrings in text `there!` are: `there!`, `here!`, `ere!`, `re!`, `e!`, and `!`. If a text is *infinite* in length (by appending null characters after the text), all sistrings are semi-infinite and can be uniquely identified by their starting positions (*start*) in the text.

We define *text trie* as an *OrTrie* or *PaTrie* built over sistrings of the text. In text tries, leaf nodes contain starts (pointers to the text). Sistrings are truncated when their prefixes become unique in the text. The trie will not grow below the level where the search path identifies a unique sistring in the text. External nodes are placed as close to the root as possible. Figure 3.1 shows the text tries for text `there!`.

Text	t	h	e	r	e	!
Start	1	2	3	4	5	6
ASCII Code	01110100	01101000	01100101	01110010	01100101	00100001



(a) *OrTrie* Index (b) *PaTrie* Index

Figure 3.1: Text Tries

Text tries may have to become quite deep to distinguish between similar sistrings. For example, we must go to ten bits to distinguish `ere!` from `e!`. As we have stated in §1.2.2, the *PaTrie* gives a much shallower trie, but adds the cost of storing height or skip information at each internal node. Figure 3.1 (b) shows a *PaTrie* with skip information in the internal nodes.

In this chapter, we assume texts are terminated with a unique symbol, say a null character, and a search can find a string beginning at any byte of the text. Thus, a text of N bytes has exactly n=N distinguishable sistrings. Sometimes, we might ask to search for text strings beginning only at word starts. Since words average five characters in length and are delimited by blanks, a text would have about n=$N/6$ sistrings, and result in a much smaller text trie.

3.2 Statistics on Text Tries

Text statistics will help us show that index size is a significant problem, not just in the worst case, but for normal usage of languages such as English. This is true even for the methods used for the *New OED*. This section is primarily about *PaTrie*s, but the statistics that follow will include *OrTrie*s, so that we can show that the cost of storing skips is more than offset by the reduced height.

3.2.1 Measured Distributions

To estimate text trie performances, we need to know: (1) the total numbers of trie nodes, S_n, for the trie size, (2) the average trie depth, A_n, for the average access time, and (3) trie height (or the maximum depth), H_n, for the expected worst access time. Trie parameters have been analyzed by many people (see §1.2.2). Most theoretical results are based on the assumption that all keys are independently and uniformly distributed, and all symbols of the keys are also uniformly distributed (symmetric or random trie). Table 3.1 are expected asymptotic results for binary tries.

	Total Nodes, S_n	Average Depth, A_n	Height, H_n
Ordinary Trie	$2.44n$	$\lg n$	$2\lg n$
Patricia Trie	$2n-1$	$\lg n$	$\lg n$

Table 3.1: Random Trie Parameters

Unfortunately, sistrings from natural language texts are not uniformly distributed. For example, `e` is much more common than `d` in English. Nor are they independently distributed. All sistrings are suffixes of the same text. Worst of all, sistrings are context related. Theoretical analyses fail to model context dependency within such a large scope. Therefore, we must measure these quantities on actual texts. We shall find that the theoretical values provide extremely lower bounds.

For five texts of 4.5 to 9.5MB each, we picked 20 segments of 1MB each at random from each text, and constructed tries for sistrings beginning at each byte. The texts are: *Shakespeare's Complete Works*, provided by Oxford University Press for NeXT Inc.; *The King James' Bible*, provided by ftp from akbar.cac.washington.edu; Section One of `man` pages for UNIX^{tm} from Solbourne Computer Inc.; C source programs selected randomly from a departmental teaching machine; and *Webster's Ninth New Collegiate Dictionary*, provided by NeXT Inc.

We calculated A_n and H_n for both *OrTrie*s and *PaTrie*s, and S_n for *OrTrie*s only. For both tries, we found the cumulative distribution of depths for n=1 million sistrings, that is, the proportion of nodes lying at or above a given depth in the trie. Finally, to estimate *PaTrie* size, we need to know *skip length*. There is no theoretical analysis for it in the literature. So, we found the cumulative distribution of skips for n=1 million sistrings, that is, the proportion of skip lengths that are less than or equal to a given number of bits. This we used to estimate the average skip sizes. The results are plotted in Figure 3.2.

A qualitative inspection of these results shows that the texts can be ranked for S_n, A_n, H_n and skip sizes of *PaTrie*s. From best to worst: (1) *Shakespeare's Complete Works*, (2) The *King James' Bible*, (3) `man` pages from UNIX, (4) C source programs, and (5) *Webster's Ninth New Collegiate Dictionary.*

Webster was significantly worse than the others in all of these cases. The reason is that we copied the dictionary redundantly from the NeXT by following all pointers, even if they lead to the same text. So our copy has many long, single repetitions. It is apparent that this is just what is bad for tries, but excellently handled by *PaTrie*s. So we expect *Webster* to behave well under *PaTrie*s, because it is effectively much shorter than the other texts, and that is what we find.

For H_n and A_n of *PaTrie*s, the rankings of the texts are almost consistent. The UNIX `man` pages are noticeably worse than the others. *Webster* is the best, as expected, and, together with the *Bible*, has significantly lower H_n.

As we can see from Figure 3.2, most of skip lengths, say 75%, are less than 60 and can be stored in counters of 6 bits long. However, some of them require much larger counters. One of the best methods to compress the skew skips is to encode them by *Huffmann* encoding [Knu68] (which requires one more pass over the data in order to construct the codes). The first part of Table 3.2 shows, both in bits, the maximum skip length and the average skip length by *Huffmann* encoding. The second part shows skip ranges and the corresponding quantities: ℓ — skip counter length, ρ — *Huffmann* code length, and δ — total skip percentage in the range. For example, for

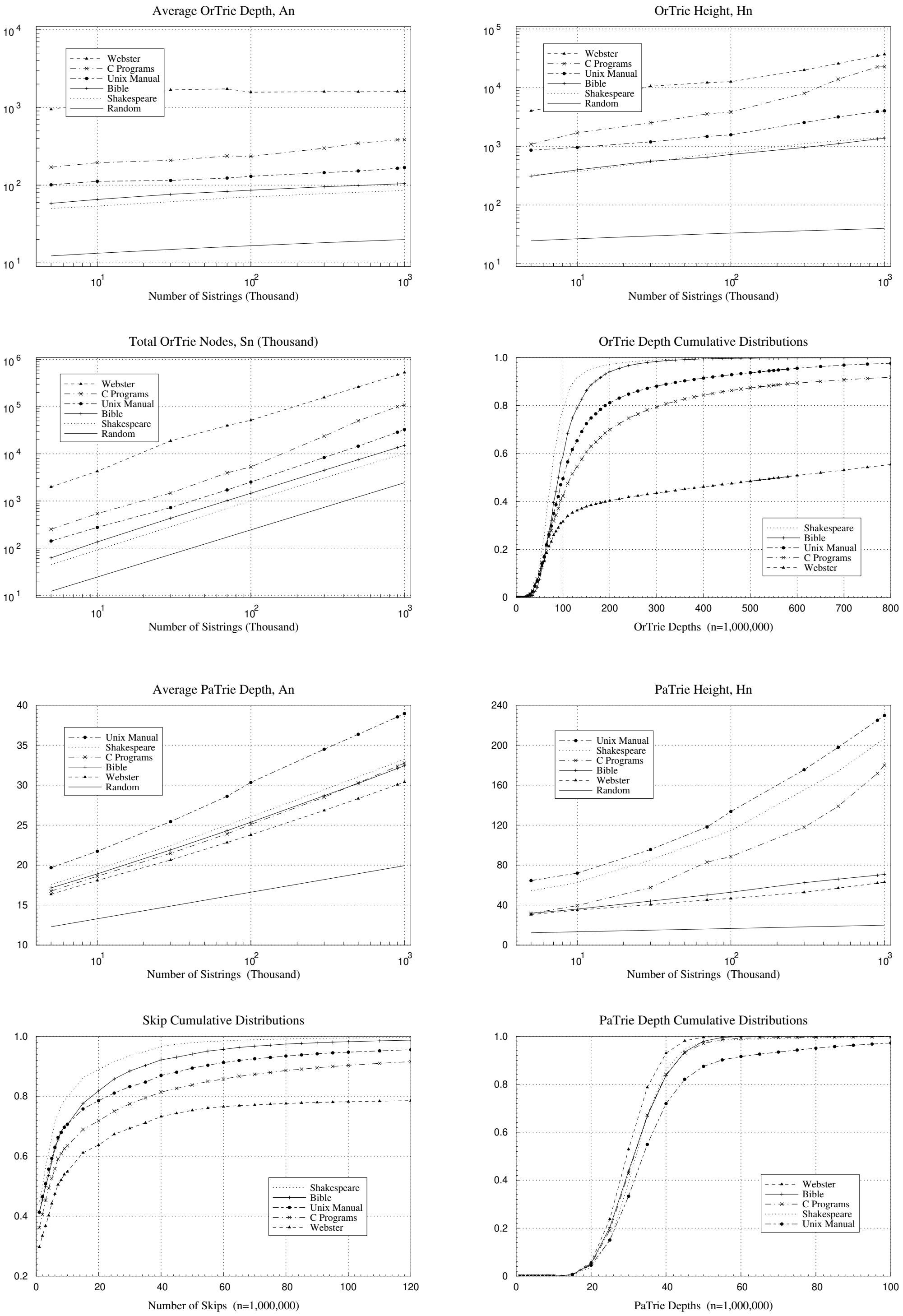


Figure 3.2: Text Trie Distributions

Skip Length (bit)		Shakespeare	Bible	Unix Man	C Program	Webster
Maximum length		11	11	14	16	16
Average Length		4.24	4.79	5.36	6.55	7.98
Skip	Length	Skip Distribution (*Huffmann* Code Length)				
	l	δ (ρ)	δ (ρ)	δ (ρ)	δ (ρ)	δ (ρ)
0	0	30.85% (2)	30.26% (2)	27.82% (2)	24.29% (2)	20.39% (2)
1	0	14.06% (3)	10.95% (3)	13.50% (3)	11.85% (3)	9.35% (4)
2	0	4.64% (4)	4.19% (5)	5.18% (4)	4.36% (4)	3.73% (4)
3 .. 4	1	11.46% (3)	8.18% (3)	9.17% (3)	8.86% (4)	6.77% (4)
5 .. 8	2	15.65% (3)	14.61% (3)	12.21% (3)	11.54% (3)	11.84% (3)
9 .. 16	3	10.20% (3)	10.70% (3)	8.47% (3)	8.64% (4)	9.77% (4)
17 .. 32	4	7.54% (3)	10.67% (3)	7.66% (4)	8.97% (3)	8.48% (4)
33 .. 64	5	4.27% (5)	6.67% (4)	7.85% (4)	8.02% (4)	6.48% (4)
65 .. Largest	$\lceil lg(Larg) \rceil$	1.33% (5)	3.77% (5)	8.14% (4)	13.47% (3)	23.19% (2)

Table 3.2: Skip Distributions

Regressions on Measured Data	Average Depth, A_n		Height, H_n		Total Nodes, S_n
	OrTrie	*PaTrie*	*OrTrie*	*PaTrie*	*OrTrie*
Random	lg n	lg n	2 lg n	lg n	2.44n
Shakespeare	4.7 lg n - 8.0	2.1 lg n - 8.0	151 lg n - 1637	20.4 lg n - 212	10.2n - 13037
Bible	6.0 lg n - 13.7	2.0 lg n - 7.9	138 lg n - 1472	5.2 lg n - 33	15.1n - 32635
Unix Manual	8.5 lg n - 6.8	2.6 lg n - 12.4	425 lg n - 4886	22.5 lg n- 230	32.6n - 560816
C Program	28.5 lg n - 202.0	2.1 lg n - 9.5	2756 lg n - 37061	19.2 lg n - 219	111.4n - 3464685
Webster	68.1 lg n + 364.0	1.8 lg n - 6.5	4120 lg n - 50590	4.1 lg n - 21	529.1n + 58419

Table 3.3: Regression Fitting

skip lengths between "5..8" inclusive, we need a count (ℓ) of 2 bits to identify the 4 different skips. In the *Shakespeare*, 15.65% (δ) of skips are between 5 and 8 in length, and the corresponding *Huffmann* code length (ρ) is 3 bits. The average skip length is calculated by $\Sigma\delta(\ell+\rho)$.

Table 3.3 shows the results of regression fits of the data of Figures 3.2 for A_n, H_n and S_n. Comparing with theoretical results, we can find random tries are much better than any of our actual text tries. Regression formulas will be used to estimate trie performances. Table 3.4 shows the values of the regression formulas for 1MB texts, and for comparisons, the values (in parentheses) actually measured on these texts.

n = 1,000,000 Expected (**Measured**)	Average Depth, A_n		Height, H_n		Total Nodes, S_n
	OrTrie	*PaTrie*	*OrTrie*	*PaTrie*	*OrTrie*
Random	20	20	40	20	2.44 M
Shakespeare	86 (85)	34 (33)	1373 (1408)	195 (206)	10.2 (10.2) M
Bible	106 (105)	32 (32)	1279 (1383)	71 (71)	15.1 (15.1) M
Unix Manual	163 (168)	39 (39)	3585 (4022)	219 (230)	32.0 (32.9) M
C Program	366 (385)	32 (33)	17870 (22651)	164 (180)	107.9 (108.1) M
Webster	1721 (1622)	29 (30)	31528 (36955)	61 (63)	529.2 (531.8) M

Table 3.4: Comparing Regressions

3.2.2 Estimated Performance

With text trie parameters on hand, we now can estimate text trie performances. The costs of trie searching are proportional to A_n and H_n in the average case and the expected worst case† respectively. We give only calculations for the average costs. In the following calculations, we use N for the text size in bytes, and n for the number of sistrings. We usually set $N=n$ and assume $N=100M=10^8$. But sometimes, we let $N=6n=600$M for comparison with the *New OED* work which involves a text of 600MB and $n=119$M sistrings, each beginning at a word.

†Given a degenerate trie, the worst search time is n which is of no interest to us. We are interested in average tries and the average performances.

A simple *PaTrie* implementation requires two pointers per node. For n sistrings, there are $2n{-}1$ trie nodes, and therefore, $\lg 2n$ bits per pointer. Each start requires $\lg N$ bits to address the N bytes of the text. The average skip length lies between 4.24 bits and 7.98 bits for the measured texts, as shown in Table 3.2. Thus an implementation with two pointers takes

$$\frac{n}{8}\left(2\lg 2n + \lg N + \left\{\begin{array}{c}4.24\\7.98\end{array}\right\}\right) = \left\{\begin{array}{c}10.7\\11.2\end{array}\right\} n \text{ bytes}$$

for a N=100MB file, about eleven times the text size. In this and following calculations, the upper numbers in the braces are for the best measured text and the bottom numbers are for the worst measured text.

As shown in Table 3.3, we have the shortest $A_n = 1.8 \lg n$ for *Webster*, and the tallest $A_n = 2.6 \lg n$ for UNIX manual pages. Thus, for a disk with 20ms average seek time, and n=100M sistrings, the average access time is

$$\left(\left\{\begin{array}{c}1.8\\2.6\end{array}\right\}\lg n - \left\{\begin{array}{c}6.5\\12.4\end{array}\right\}\right) = \left\{\begin{array}{c}41.3\\56.7\end{array} \text{accesses}\right\} = \left\{\begin{array}{c}0.83\\1.13\end{array}\right\}\text{seconds.}$$

We can remove one pointer by storing trie nodes in, say preorder, so that left descendents will be immediate neighbours of their parent nodes and so need no pointers. This saves us $\frac{1}{8}\lg 2n = 3.4n$ bytes. We can also shorten start by using *text pages* as the target of the start pointers, instead of individual bytes. If the text is stored in pages of 4KB, the starts are shorter by 12 bits each. Both improvements together reduce the index size to

$$\frac{n}{8}\left(\lg 2n + \lg\frac{N}{4K} + \left\{\begin{array}{c}4.24\\7.98\end{array}\right\}\right) = \left\{\begin{array}{c}5.8\\6.3\end{array}\right\} n \text{ bytes}$$

for a N=100MB file, about six times the text size. This implementation reduces access time by half because one of the two descendents will be retrieved in the same time as its parent node.

The PAT array [GBYS92] has no pointers at all, and is not even a tree. It stores nothing but starts. The idea is to store these starts in a lexicographical order of the sistrings they point to, and to use them for a binary search of the text. The PAT array requires $\frac{1}{8}(n \lg N) = 3.3n$ bytes for a N=n=100M file. This would seem the absolutely minimal index for addressing any character in a text. We find that it is not. The major consideration is that PAT array cannot use the text paging improvement to reduce the start size. This is because the binary search must make a comparison with the text each time it looks up a start, and therefore, the start must point to the starting bytes of the text, not merely to a page.

Because of the extra comparison, PAT array takes $2 \lg n$ accesses for a search. This is much more expensive than searching a well-made tree, with reference to the text only from the leaf nodes. For an index of 100M sistrings, this is 27 accesses to the text and 27 accesses to the PAT array. However, if we read a page of 2^{10} starts, the last ten lookups of the PAT array will all be on the same page, so the cost will be only about 18 accesses to the PAT array, and a total of 45 accesses. A better improvement can be found in [MS93b].

The *OrTrie* implementation requires two bits per node, plus n starts. As shown in Table 3.3, we have the smallest $S_n = 10.2n$ for *Shakespeare*, and the largest $S_n = 529.1n$ for *Webster*. Combined with our text paging technique, an *OrTrie* index has

$$\frac{n}{8}\left(\lg \frac{N}{4K} + 2 \times \left\{ \begin{array}{c} 10.2 \\ 529.1 \end{array} \right\}\right) = \left\{ \begin{array}{c} 4.4 \\ 134.1 \end{array} \right\} n \text{ bytes}$$

for a N=100MB file. Thus, the index size is no smaller than PAT arrays, and can be immensely larger.

The average time of *OrTrie* searching is A_n/k, where $k = 8$ is the number of levels in a page. For a disk of $20ms$ average seek time, an index of n=100M sistrings, and A_n of Table 3.3, the average access time is

$$\frac{1}{8}\left(\left\{ \begin{array}{c} 4.7 \\ 68.1 \end{array} \right\} \lg n + \left\{ \begin{array}{c} -8.0 \\ 364.0 \end{array} \right\}\right) = \left\{ \begin{array}{c} 14.6 \\ 271.7 \end{array} \text{accesses} \right\} = \left\{ \begin{array}{c} 0.28 \\ 5.43 \end{array} \right\} \text{seconds.}$$

Finally, for the *PaTrie* implementation, we must store n skips, n starts, and one bit for each of the $2n-1$ nodes. So, it takes

$$\frac{n}{8}\left(2 + \lg \frac{N}{4K} + \left\{ \begin{array}{c} 4.24 \\ 7.98 \end{array} \right\}\right) = \left\{ \begin{array}{c} 2.6 \\ 3.1 \end{array} \right\} n \text{ bytes}$$

for a N=100MB file, about three times the text size, and less than the PAT array.

The average access time of a *PaTrie* index is

$$\frac{1}{8}\left(\left\{\begin{array}{c}1.8\\2.6\end{array}\right\}\lg n-\left\{\begin{array}{c}6.5\\12.4\end{array}\right\}\right)=\left\{\begin{array}{c}5.2\\7.1\end{array}\text{accesses}\right\}=\left\{\begin{array}{c}0.10\\0.14\end{array}\right\}\text{seconds.}$$

Table 3.5 summarizes the calculations made throughout this section for various implementations. We assume that n=100M=10^8 and that sistrings start at each byte. For timing, we assume a disk with $20ms$ expected seek time and $1\mu s$ transfer time per byte. The text pages are 4KB where applicable. The I/O buffers are 4KB for PAT arrays and 1KB for both *OrTrie*s and *PaTrie*s. The "best" and "worst" refer to the five measured texts.

Implementations	Expected Trie Sizes (bytes)		Expected # Disk Accesses			
			Average		Maximum	
	best	worst	best	worst	best	worst
2 pointers	$10.7n$	$11.2n$	41.3	56.7	88.0	367.9
1 pointer	$5.8n$	$6.3n$	20.7	28.3	44.0	184.0
PAT array	$3.3n$		45			
OrTrie	$4.4n$	$134.1n$	14.6	271.7	274.4	7362.6
PaTrie	$2.6n$	$3.1n$	5.2	7.1	11.0	46.0

Table 3.5: Binary Trie Comparison

In this section, we have determined formulas for various trie performances. Each formula uses one or two of the four trie parameters, i.e., average trie depth A_n, trie height H_n, total trie node S_n and average skip length. We used statistics from actual text to fit these parameters. In §3.4, we shall show experiments on *PaTrie*s for these texts and demonstrate that the formula for trie size has an error < 4%.

3.3 Text Trie Construction

We consider texts that have no changes, or low update frequency such as historical data and dictionaries. For this type of texts, it is much more efficient to add all the data at once than, say, inserting sistrings by their text order (see §2.4.3). The problem is how to sort numerous and extremely long sistrings. But first, we give an algorithm to handle small changes to a text.

3.3.1 Dynamic Text

We start with an algorithm that builds or updates *PaTrie*s dynamically. The example of Figure 3.3 shows the insertion of `a` at start 7 of `there!` to give `there!a`. The new *PaTrie* is shown in solid lines, and the old, where it differs, in dashed lines. The new start, 7, is lexicographically between starts 6 and 5, but shares a subtrie with starts 5 and 3.

Text	t	h	e	r	e	!	a
Start	1	2	3	4	5	6	7
ASCII Code	*01110100*	*01101000*	*01100101*	*01110010*	*01100101*	*00100001*	*01100001*

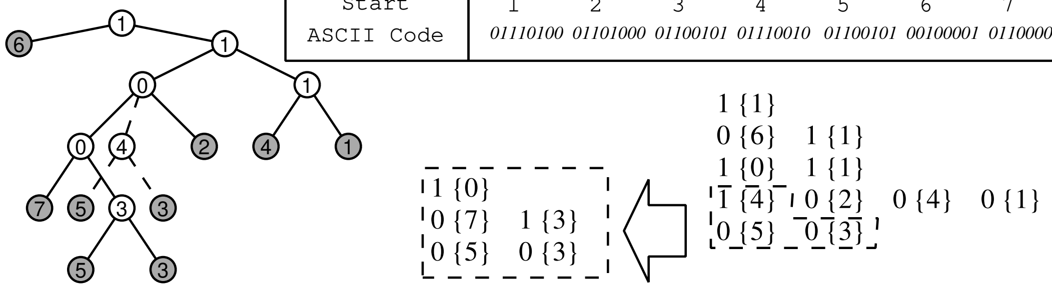


Figure 3.3: Updating *PaTrie*

We see that while the positions of the entries may be altered within their levels, or may change levels, the changes to entries are localized. In the example, we have changed the node `1{4}` to `1{0}`, have moved down the nodes below it, and have inserted a new level for the new leaf and internal node. Algorithm 3.1 shows the procedures for sistring insertion and deletion.

If a new text is added to anywhere but at the end of the old text, the starts will be out of order, or many starts will have to be changed (but only by a constant offset). This ceases to hold, however, if the starts point to pages rather than to byte locations. In that case, text can be added to a new page without changing pointers in any old text page. Thus, all insertions cause only local changes.

Another problem of inserting a text anywhere but at the beginning of the old text is that it changes sistrings *before* the insertion point (sistrings are suffixes). To insert at arbitrary position, we must first find a unique sistring before this position: this sistring will distinguish all preceding sistrings from each other. Than we delete the text from the unique sistring to the insertion point, and prepend this deleted text to

Algorithm 3.1 Text Trie: Inserting and Deleting

```
Procedure SistringIns( Start:integer; Text:string; Var T:PaTrie);
  Var Xnode, Pnode, Cnode : Anode;
  begin
    Xnode := a leaf node of T by searching Text[Start];       /* search for sistring */
    if Text[Start] = Text[Xnode.start] then
      error;                                                  /* sistring already in T */
    prefix := (longest common prefix of Text[Start] and Text[Xnode.start]);
    Xnode := a node of T by searching prefix;
    Cnode.start := Start;                                     /* create a new leaf node */
    Pnode.skip := length(prefix)-(height of Xnode's parent);
    if (Xnode is an internal node) then
      Xnode.skip := Xnode.height-length(prefix);              /* modify skip */

    insert Pnode into the bit-string before Xnode;
    shift subtrie(Xnode) entirely down one level;     /* replace subtrie(Xnode) by */
    if Text[Start+length(prefix)+1] = 0 then
      insert Cnode before Xnode;                          /* Cnode↙Pnode↘Xnode */
    else
      insert Cnode after Xnode;                           /* Xnode↙Pnode↘Pnode */
  end;

Procedure SistringDel( Start:integer; Text:string; Var T:PaTrie);
  Var Xnode, Pnode, Cnode : Anode;
  begin
    Xnode := a leaf node of T by searching Text[Start];       /* search for sistring */
    if Text[Start] <> Text[Xnode.start] then
      error;                                                  /* sistring is not in T */

    if (Xnode is the Root of T) then
      T := empty;                                   /* Xnode is the only node of T */
    else
      Pnode := (The parent node of Xnode);
      Cnode := (The sibling node of Xnode);               /* Xnode↙Pnode↘Cnode */
      Cnode.skip := Cnode.skip+Pnode.skip+1;
      delete Xnode and Cnode;                 /* replace Xnode↙Pnode↘Cnode by */
      shift subtrie(Cnode) entirely up one level;             /* subtrie(Cnode) */
  end;
```

the text to be inserted. A similar consideration applies if the text to be inserted is in fact a *replacement* for some part of the original text.

Fortunately, we do not have to search far for unique sistrings. Because a unique sistring maps from the root to a leaf of the trie representing the text, a unique sistring will start no earlier than the trie height, H_n, before the insertion point. We expect to search only A_n bits, the average trie depth.

To build a *PaTrie* by updating or incrementally inserting sistrings is prohibitive (see §2.4). Essentially Algorithm 3.1 takes $A_n/8$ accesses per insertion $\times n$ insertions $\times$, say $20ms$ per disk access, or

$$\frac{n \lg n}{8} \times \frac{20 \times 10^{-3}}{3600 \times 24} \times \left\{ \begin{array}{c} 1.8 \\ 2.6 \end{array} \right\} = \left\{ \begin{array}{c} 138 \\ 200 \end{array} \right\} \text{days}$$

I/O time to build a *PaTrie* index of n=100 million sistrings.

3.3.2 Sistring Sorting

As we have demonstrated in §2.4.3, random disk access is minimized when tries are built as a whole. Algorithm 2.3, thereafter, is provided to build a *PaTrie* from a $List(K_n)$. Here, K_n is a set of n sistrings and $List(K_n)$ is a list of sorted sistrings alternatively with height information. We now show how to generate the $List(K_n)$ from a given text.

To sort numerous and extremely long sistrings, we have to seek for external sort techniques. In general, an external sorter will: (1) chop data into pieces, (2) sort each piece inside RAM (called *initial run*), and (3) merge all pieces together. RAM size is critical since the larger the RAM capacity is, the fewer the initial runs will be, and the less the sorting time.

Sistrings are too long to be stored entirely inside RAM. We can only sort starts. To do this, RAM must be divided to hold both starts and one segment of the text. We create an initial run by internally sorting the starts corresponding to the text in RAM. If we fit n' starts (about 3 bytes each) into RAM together with the text segment ($n' \times N/n$ bytes), for 20MB of RAM, the expected run lengths will be

$$\frac{20}{3+N/n} = \left\{ \begin{array}{ll} 5M \text{ sistrings} & \text{if } N = n = 100M \\ 2.2M \text{ sistrings} & \text{if } N = 6n = 600M \end{array} \right\} = \left\{ \begin{array}{c} 20 \\ 46 \end{array} \right\} \text{initial runs.}$$

Replacement selection [Knu73] can generate runs twice the size of RAM. However, to sort sistrings, we have to store both text and starts in RAM. Our experiments show that the extra text in RAM is 4 times larger than for quicksort, and that leads to initial runs of twice the number of starts in RAM. (These factors, 4 and 2, deviate by less than 1.5% over all our texts.) The number of initial runs is 18 for N=n=100M, and 68 for N=6n=600M. This is comparable to quicksort for N=n, but substantially worse for N/n=6. Our experiments also confirm Knuth's assertion [Knu73] that quicksort is three times faster than the heapsort used in replacement selection. So, we generate initial runs using quicksort on a fixed amount of text held in RAM. The next two sections show how to merge the initial runs.

Prefix Sort

Our first approach is simple, but requires a very large temporary workspace. In the next section, we give a method which is better in both space and I/O time.

At the output time of initial runs, we append to each start enough bytes from the text to distinguish it from any other start. Let us call this H bytes. If all sistrings are distinguishable by these H bytes, the merge procedure simply reads the output of initial runs, calculates the heights at the same time by comparing neighbours, and outputs $List(K_n)$, the starts and heights. For H=50B and N=n=100M, this method takes $(H+\frac{1}{8}\lg N)n = 5.3$GB temporary storage space.

The time required to do the merge and the height calculation is the time to read the initial runs and write the output. All reading and writing can be done sequentially. If using buffers of 64KB, the merge procedure costs

$$\frac{n}{64K}\left(H + \frac{3}{8}\lg N\right) \times \frac{20+64}{3600 \times 10^3} = 2.2 \text{ hours.}$$

The corresponding quantities for N=6n=600MB are 5.4GB and still 2.2 hours. However, 50 bytes does not fully resolve all sistrings. From the results of cumulative *PaTrie* depth in §3.2.1, we have the following percentages of sistrings unresolved by

H=50: 1% for both *Shakespeare* and *Bible*, 8% for UNIX, 14% for C Programs, and 42% for *Webster*. In the best case, we must do direct access to 1% of the text, which is one million accesses at, say $20ms$ each, or 5.6 hours. Thus, to generate $List(K_n)$, this method takes ten hours of I/O times (over a day for N/n=6).

Counter Sort

To avoid re-reading the text during the merge phase, Gonnet *et al* suggested an algorithm [GBYS92] to include counters in the initial runs. The counters tell the number of starts in all previous runs which are between two adjacent starts of the current run. We extend their method by including height information, two per start, to give the positions of the bits that distinguish the start from its predecessor and successor in the current and all previous runs.

This changes the above run-size calculations. Starts are $\lg N$ bits and heights cannot exceed $\lg 8N$. For N=n=100M, they take about 10 bytes. Thus, a 20MB RAM will hold

$$\frac{20}{10+N/n} = \left\{ \begin{array}{ll} 1.8M \text{ sistrings} & \text{if } N = n = 100M \\ 1.3M \text{ sistrings} & \text{if } N = 6n = 600M \end{array} \right\} = \left\{ \begin{array}{c} 56 \\ 77 \end{array} \right\} \text{ initial runs.}$$

The workspace to hold the initial runs will be 1GB for N=n=100M sistrings.

Algorithm 3.2 captures this discussion. We illustrate it for the text `thero!` (`o` is *01101111* and is introduced to keep the third initial run interesting). The initial runs are:

$$\begin{array}{lcccc} \text{Run 1} & & {}^{0}2^{4} & & {}^{4}1^{0} \\ \text{Run 2} & {}_{0} & {}^{0}3^{5} & {}_{1} & {}^{4}4^{6} \\ \text{Run 3} & {}_{0} & {}^{0}6^{2} & {}_{2} & {}^{6}5^{6} \end{array}$$

This is interpreted as follows. The main entries are starts, the subscript entries are counters, and the superscript entries are heights. Run 1 has no counters. Thus, in run 3, the first counter, ${}_{0}$, tells us that no start in any previous run (1 or 2) points to a sistring sorting before it is pointed to by the first start, 6, in run 3. The second counter, ${}_{2}$, in run 3, tells us that two sistrings come between those at starts 6 and 5 (they are at starts 3 and 2).

Algorithm 3.2 Counter Sort: Generating Initial Runs

```
Type Run = Array [] of Record
                         Counter, Start : Integer;
                         Lheight, Rheight : Integer;        /* Max Heights */
                       End;
Procedure CountInitial( Text:string; R:file of Run);
  Var StartBuf : Run;
      StartBase, CurrRun : integer;
  begin
    StartBase := 0;
    CurrRun := 1;
    while StartBase < N do                      /* sort until no more text */
      read( Text, TextBuf);                      /* read in one piece text */
      for i := 1 to Length(TextBuf) do
        StartBuf[i].Start := i;                          /* initial starts */

      QuickSort( StartBuf);          /* sort starts in lexicographical order */
      Let S_i be the sistring pointed by StartBuf[i].Start;
      for i := 1 to Length(TextBuf) do                 /* initial counters */
        StartBuf[i].Counter := 0;
        StartBuf[i].Lheight := Height(S_i, S_{i+1});
        StartBuf[i].Rheight := StartBuf[i].Lheight;

      for (each sistring Text[x] in run [1..CurrRun-1]) do       /* scan Text */
        find i such that                       /* search position of Text[x] */
          (S_{i-1} < Text[x] < S_i);
        StartBuf[i].Counter := StartBuf[i]+1;
        StartBuf[i].Lheight := Max(startBuf[i].Lheight, Height(Text[x],S_i));
        StartBuf[i].Rheight := Max(startBuf[i-1].Rheight, Height(S_{i-1},Text[x]));
        StartBuf[i].Start := StartBuf[i].Start+StartBase;

      write( R, StartBuf);                       /* output one initial run */
      CurrRun := CurrRun + 1;                  /* prepare for the next run */
      StartBase := StartBase + Length(TextBuf);
  end;
```

Algorithm 3.3 Counter Sort: Merging Initial Runs

```
Procedure CountMerge( R:file of Run; L:List(K_n));
  Var OutSoFar, OutPosi :Array [1..TotalRun] of integer;
      Height, Start, LastHeight, CurrRun : integer;
  begin
    OutSoFar := 0;
    OutPosi := 1;
    LastHeight := 0;
    for i := 1 to n do                              /* for each of n starts */
      CurrRun := TotalRun;
      j := OutPosi[CurrRun];
      while CurrRun>1 and R[CurrRun][j].Count<>OutSoFar[j] do
        CurrRun := CurrRun + 1;          /* run that has the start to be output */
        j := OutPosi[CurrRun];
      Height := Max( LastHeight,R[CurrRun][j].Lheight);
      Start := R[CurrRun][j].Start;
      LastHeight := R[CurrRun][j].Rheight;
      OutSoFar := OutSoFar + 1;
      OutSoFar[CurrRun] := 0;
      OutPosi[CurrRun] := j + 1;                    /* prepare next output */
      Write( L, Height, Start);             /* output one height-start pair */
    Write( L, 0);                /* final output: 0, s_1, h_1, s_2, ..., h_{n-1}, s_n, 0 */
  end;
```

The height, ^{2}, after 6 in run 3, is the bit position distinguishing the sistring at 6 (`!`) from that at 3 (`e`), its successor. The height, ^{6}, before 5, distinguishes the sistring at 2 (`h`) from that at 5 (`o`). The height, ^{4}, after 5, distinguishes `o` at 5 from `r` at 4.

We see that each run considers only the runs before it. To do this, it must scan the text for all previous runs, that is, from the beginning to the position corresponding to the present run. In all, this is (# runs)/2 or 28 passes for N=n=100M, and 39 for N=6n=600M.

The cost of generating the initial runs is dominated by these passes of the text file. With 64KB text buffer, it costs

$$\frac{(20+64)\,N}{64 \times 36 \times 10^8} \times \left\{ \begin{array}{c} 28 \\ 39 \end{array} \right\} = \left\{ \begin{array}{ll} 1.0 \text{ hours} & \text{if } N = n = 100M \\ 8.5 \text{ hours} & \text{if } N = 6n = 600M \end{array} \right\}.$$

Algorithm 3.3 merges the initial runs with counters and heights in a single pass.

The merge phase looks at the first entry in each initial run. Continuing with our example, we see that the merge picks the start 6 because of the $_0$ counter before it and the fact that of the two entries with a zero counter, the 6 has the larger run number. The height, ^{2}, is also output. It then must look for two starts from earlier runs, because of the counter, $_2$, in run 3: it takes 3^5 from run 2, because of the counter $_0$ before it; and 2 from run 1 because the counter $_1$ in run 2 makes us look in run 1. The two starts needed by the counter $_2$ in run 3 are now found, so $^6 5^4$ is output from run 3. The one start needed by the counter $_1$ in run 2 has also been found, so we output ^{6} from run 2, then finally 1 from run 1. The merged output is $6^2 3^5 2^6 5^4 4^6 1$.

Writing and reading the workspace of 1GB takes about 44 minutes assuming buffers of 64KB. This does not add much to the time required to sort the starts.

The costs in time and space are summarized in Table 3.6 for a N=100MB text, assuming 20ms seek time and 1μs transfer time per byte. For a text of N=600MB but n=100M sistrings, such as the *New OED*, the time for the counter sort will be 39 hours. Let us compare it with the statement of Gonnet *et al* [GBYS92] that the *New OED* index can be built "during a weekend". What is new here is that our version includes height information and thus builds the faster pointerless *PaTrie*, not just a PAT array.

	Prefix Sort		Counter Sort	
	Time	Space	Time	Space
Initial Runs	2.2 hours	5.3GB	1.0 hours	1.0GB
Merge	7.8 hours		44 minutes	

Table 3.6: Sorting: Time and Space

3.4 Experimental Results

In §3.4.1, we partially check the *PaTrie* size formula (see §3.2.2) with actual indices of 1, 2, and 4M sistrings extracted from each of the five texts. We give calculated sizes for the purpose of comparing with *OED*. In §3.4.2, we compare the calculated access time against the measurement time. We find a factor of 2 for indices of one million sistrings. The discrepancy is due to the CPU time (on our relatively slow NeXT) which the formula does not take into consideration. We don't know the CPU time for indices of 100 million sistrings. But we can expect the factor is the same as the measurement, and so we use them to compare with *OED*. In §3.4.3, we combine the measured time with the calculated time to extrapolate construction time for *PaTrie*s for comparing with *OED*. All the measurements were carried out on a NeXT (*M68030*, 28MB RAM, and 25MHz clock) with two disks (13.5ms average seek time, and 0.5μs data transfer per byte).

3.4.1 Text Trie Sizes

Instead of *Huffmann* encoding for skip information, which requires skip distributions, we propose two simpler methods. Method 1: skip counters are all large enough to hold the largest skip. For example, according to Table 3.2, the largest skip length for *Shakespeare* is 11 bits. As we have seen, more than 80% of skips are less than 16 (4 bits), and $(11-4)n = 7n$ bits are wasted. Method 2: skip counters are all 5 bits long. When a skip length is larger than 5 bits, we set the counter to all `1`s, and allocate another skip counter to hold the largest skip. The *skip* column of Table 3.7 shows the average length of this method on the measured texts. As compared with *Huffmann* method shown in Table 3.2, this method increases the counter size by no more than two bits.

Index Trie Size	Skip (bit)	N = n = 1 M		N = n = 2 M		N = n = 4 M		N = 100 M
		Calculated	Measured	Calculated	Measured	Calculated	Measured	Calculated
Shakespeare's Works	5.72	1.97 MB	1.99 MB	4.18 MB	4.21 MB	8.86 MB	8.95 MB	284.0 MB
King James' Bible	6.27	2.03 MB	2.10 MB	4.32 MB	4.34 MB	9.14 MB	9.25 MB	290.9 MB
Unix Manual Pages	7.35	2.17 MB	2.12 MB	4.59 MB	4.66 MB	9.68 MB	10.08 MB	304.4 MB
C Programs	8.60	2.33 MB	2.39 MB	4.90 MB	4.97 MB	10.30 MB	10.59 MB	320.0 MB
Webster Dictionary	9.91	2.49 MB	2.47 MB	5.23 MB	5.29 MB	10.96 MB	10.95 MB	336.4 MB
Note: Calculated: $n\times(2+\lceil \lg(N)/4K \rceil+\text{Skip})/8$ and excluding paging overhead ≈2%;								

Table 3.7: *PaTrie* Sizes

Table 3.7 shows *PaTrie* sizes as measured and as calculated using the formula in §3.2.2. The discrepancy does not exceed 4%. We also show the calculated size for texts of 100M sistrings, one sistring per byte: we have not built indexes of this size. For both implementation and calculation, text and index pages are 4KB each.

3.4.2 Search Times

Table 3.8 shows the successful and unsuccessful search time as measured for each of the five text tries. It also shows the successful search time as calculated. Unlike successful searches which always terminate at the external nodes, unsuccessful searches may stop at the internal nodes. The internal nodes are closer to the root than the external nodes. Therefore, unsuccessful searches are faster than successful searches. This was confirmed by our measurements. There is no analysis for unsuccessful searches.

Index Search Time	N = n = 1 M				N = n = 100 M
	Measured	Measured	Calculated	Factor	Calculated
Webster Dictionary	102.43	116.10	56.26	2.06	79.54
King James' Bible	115.58	121.40	62.08	1.96	87.30
C Programs	118.12	123.33	62.08	1.99	89.24
Shakespeare's Works	116.72	125.67	65.96	1.91	93.12
Unix Manual Pages	144.40	145.45	75.66	1.92	110.58
	Unsuccessful	Successful Search Time (ms.)			
Note:	Calculated: $(A_n / 8) \times (13.5 \text{ ms.} + 4{*}0.5 \text{ ms.}) = 1.94 \times A_n$ (ms.);				

Table 3.8: *PaTrie* Search Times

Table 3.8 shows a discrepancy factor of 2 between the measured time and the calculated time. Our trie method requires substantial CUP time for bit masking which is not included in the formula of §3.2.2. We expect the discrepancy factor will remain the same for larger indices.

3.4.3 Construction Times

Table 3.9 shows the costs of *PaTrie* constructions by the two sorting methods outlined in §3.3.2, the prefix sort and the counter sort techniques.

Index Construction Time N = n = 4 M	Shakespeare	Bible	Unix Man	C Program	Webster	Calculated
Prefix Sort (minutes)	35	37	52	80	110	9
Counter Sort (hours)	5.4	5.4	5.7	5.5	5.7	0.12

Extrapolation To N = n = 100 M	Shakespeare	Bible	Unix Man	C Program	Webster	Calculated
Prefix Sort (hours)	17.7	18.7	26.3	40.4	55.5	2.2
Counter Sort (days)	140.6	140.6	148.4	143.2	148.4	0.04

Table 3.9: *PaTrie* Construction Times

The first part of the table shows the times for sorting 4M sistrings, starting at each byte. We scaled down RAM size to 28MB$\times 4/100 = 1.2$MB so that we can assume 28MB RAM when sorting 100M sistrings. The prefix sort takes 248MB workspace and running time ranges from 35 minutes (*Shakespeare*) to 1.83 hours (*Webster*). The counter sort needs a smaller workspace of 96MB and tighter execution time, between 5.4 and 5.7 hours. The calculated times based on the formula of §3.3.2 serve only as loose lower bound since CPU time such as bits comparing and counter setup are not counted.

The second part of Table 3.9 shows extrapolation for 100MB texts, assuming prefix sort algorithm takes $c_1 n \lg n$ time and counter sort algorithm takes $c_2 n^2$ time, where c_1 and c_2 are constant coefficients. The extrapolation tells that, to sort 100M sistrings, the predicted times for prefix sort range from 18 to 56 hours with an overhead of 5.3GB, and the predicted times for counter sort are about 5 months with only 2.4GB. However, the counter sort may not be impractical if we had a computer which is 50 times faster than our NeXT because the predicted I/O time is quite small.

3.5 Other Trie Searches

In this section, we shall examine other trie algorithms [BY89, BYG89, GBYS92, ST93] for exact text searching. Except for the k longest common substring search, all these algorithms have done before. We mention them here to show that they can also be done by our method. For simplicity, we shall mainly use the *FuTrie* structure, which differs from *OrTrie* or *PaTrie* structures only in implementation matters. Let K_n be a set of n strings, which can either be independent of keywords, or be sistrings from a text. We assume a finite alphabet size.

Keyword Search

Starting at the root, we search $FuTrie(K_n)$, the *FuTrie* structure constructed over K_n, by following the pattern string. If the search ends at a leaf node, the pattern exists. Otherwise the search fails. This search requires $\mathcal{O}(\ell)$ time in the worst case, where ℓ is the length of the pattern.

We can do better when searching for a set, P_m, of m pattern strings. The search is equivalent to superimposing $FuTrie(P_m)$ with $FuTrie(K_n)$. Only the patterns whose leaf nodes overlap with the leaf nodes of $FuTrie(K_n)$ are in K_n. This search requires $\mathcal{O}(m)$ time in the worst case. Both algorithms are independent of n, the size of K_n.

Prefix Search

This is to search K_n for strings with a given prefix. The search is the same as the keyword search except that it can be terminated at an internal node. All the strings inside the subtrie of the internal node are the answers. This search requires $\mathcal{O}(\ell+k)$ time in the worst case, where ℓ is the pattern length and k is the answer size.

In general, we can search for a set of prefixes. Let P_m be a set of m prefixes specified by the *prefixed regular expression* [BYG89]. We superimpose the $FuTrie(P_m)$ with $FuTrie(K_n)$. Only the strings whose root-to-leaf path overlaps with the leaf nodes of $FuTrie(P_m)$ are in K_n. This search requires $\mathcal{O}(m+k)$ time in the worst case.

Regular Expression Search

We first construct a *DFA* (or a *NFA*) machine for a given regular expression, and then simulate the automaton along with a *depth-first* search of $FuTrie(K_n)$. For each trie node which associates with a final state, we accept the whole subtrie and stop searching down that subtrie. Since this algorithm does not need outgoing transitions for final states, it takes sublinear search time on average [BYG89].

Proximity Search

This search is to find all places where one string is at a fixed (given by the user) number of characters away from the other string. We search $FuTrie(K_n)$ for the two pattern strings, and then sort the two answer sets by text position (starts). The final answers are obtained by merging the two sorted sets[†]. Let k_1 and k_2 be the respective answer set sizes, this algorithm requires $\mathcal{O}(k_1 \lg k_1 + k_2 \lg k_2)$ time. Better solutions can be found in [GBYS92].

Range Search

This is to search K_n for all the strings within a range of two pattern strings. We search *FuTrie*(K_n) to find the search paths of the two pattern strings, and then collect all the subtries between (and including) the two search paths. This search takes $\mathcal{O}(\ell+k)$ time in the worst case, where ℓ is the maximum string length of the two patterns and k is the numbers of answer strings.

The Most Common Substring Search

This is to search a text for the most commonly used string, e.g., find the most common word of a text. We build a *FuTrie* over every possible sistring of the text, and add a counter to each internal node to indicate the size of its subtrie. To find the most common word is equivalent to searching for the largest subtrie whose search path begins with a space and ends with a second space. This can be achieved by a simple traversal of the *FuTrie* which takes at most $\mathcal{O}(n/k)$ time. Here k is the number of words found in the text [GBYS92].

The Longest Common Substring Search

When K_n are keywords, we are searching for the longest prefix shared by two keywords. When they are sistrings starting at characters, we are searching for the longest repetition of the text. This search is equivalent to finding the lowest internal node (within a subtrie) of *OrTrie*(K_n). By adding an extra bit to each internal node to indicate which side has the tallest subtrie, we can find the lowest internal node in $\mathcal{O}(H_n)$ time in the worst case, where H_n is the height of *OrTrie*(K_n).

However, it is not necessary to keep the extra bit. Since our tries are organized by levels, we have no difficulty in finding the lowest internal nodes. By scanning the

[†]If exact locations are not maintained, we need subsequent examination of each matched page.

bit-string in reverse, we can walk our tries from leaves to the root. We shall not elaborate on the algorithm in this thesis.

In general, we can use tries to solve the k *longest common substring* problem. The algorithm can be described as follows. (1) Build an *OrTrie* over all possible sistrings of the k strings. (2) Color the trie nodes. We paint an internal node *black* if its subtrie contains sistrings from each of k strings. Otherwise we paint it *white*. (3) Find the lowest black node. This algorithm takes $\mathcal{O}(n)$ time to color and search the *OrTrie*. To our knowledge, this is the first algorithm of its kind.

3.6 Summary

We finish this chapter by comparing our text tries with three other index methods for exact text searching: signature files, inverted files and PAT arrays. Signature files [FC87], use hashing techniques and are 10% to 20% of the text size. They have small storage overhead, but require linear search time. Furthermore, they may return some answers that do not match the query.

An inverted file [Knu73] is a sorted list of keywords with pointers pointing to the text. The storage overhead of inverted files may vary from 30% to 100% of the text size depending on the data structure and the number of indexed keywords. The search time for keywords is logarithmic. Similar performances can be achieved by PAT arrays [GBYS92]. However, PAT arrays have the advantage over inverted lists in efficient searching of substrings. In such a case, the indexes have 340% storage overhead.

Our text tries are smaller than PAT arrays. Trie methods can be used in other searches as we have shown in §3.5, which are either difficult or inefficient over inverted files or PAT arrays. More importantly, our tries take much fewer random disk accesses than PAT arrays. (Minimizing random disk access is an especially crucial issue when using optical disks, which have very slow random access time.)

Chapter 4

Approximate String Matching

In this chapter, we shall use tries to solve the *k differences approximate string matching* problem. We shall focus on dictionary lookup related applications, such as spelling checkers, in which one searches a keyword list or a dictionary for the pattern string which may have k ($k>0$) spelling errors. If k is very large, say larger than the longest keyword, any keyword qualifies as a match since every letter can be a mistake. Obviously, this is not an interesting problem for spelling checkers. Damerau [Dam64] found that 80% misspellings are single errors, i.e., either a letter extra, a letter missing, a letter wrong, or two letters reversed. In other words, with an approximate search of k=1, a spelling checker can find the right keywords for 80% of misspellings. We restrict the approximate search to few mistakes, say $k \leq 3$.

This thesis is primarily on searching bulk data on secondary storage. However, our proposed trie structures work also with small data. As a result, we do not illustrate in this chapter with large data.

4.1 String Similarity

The degree of string similarity is often measured in terms of the *minimum edit distance* — the minimum number of edit operations to change one string into another. Finding the minimum edit distance is an optimization problem and is often solved by the *dynamic programming* technique. In the next two sections, we will give a brief introduction to edit distance and dynamic programming.

4.1.1 Edit Distance

Minimum edit distance [WF74] (or Levenshtein distance [Lev66]), $D(P_m, W_\ell)$, is the minimum number of mismatches between the pattern string $P_m{=}p_1p_2...p_m$ and the target string $W_\ell{=}w_1w_2...w_\ell$ over an alphabet Σ. A mismatch is defined as: (1) a symbol in W corresponds to no symbol in P, (2) a symbol in P corresponds to no symbol in W, (3) a symbol in P corresponds to a different symbol in W, or (4) two adjacent symbols in P correspond to two reversed symbols in W. *Insertion, deletion, substitution and transposition* are four corresponding edit operations to revise the mistakes.

Formally, our k approximate searching problem is to find strings in some set, K_n, of n strings such that they have at most k mismatches, or are the best match for the pattern P. That is,

(1) $[\ W \mid W \in K_n \wedge D(P,W) \leq k\ ]$, and

(2) $[\ W \mid W \in K_n\ \wedge \forall_{W' \in K_n}(D(P,W') \geq D(P,W))\ ]$.

4.1.2 Dynamic Programming

Minimum edit distance can be recursively defined as follows:

$$D(P_i, W_j) = \begin{cases} \infty & \text{if } i < 0 \vee j < 0 \\ i + j & \text{if } i = 0 \vee j = 0 \\ \min \begin{pmatrix} D(P_i, W_{j-1}) + 1 \\ D(P_{i-1}, W_j) + 1 \\ D(P_{i-1}, W_{j-1}) + d_{i,j} \\ D(P_{i-2}, W_{j-2}) + d_{i-1,j} + d_{i,j-1} + 1 \end{pmatrix} & \text{if } i > 0 \wedge j > 0 \end{cases}$$

Here $d_{i,j} = 0$ if $p_i = w_j$, or 1 if $p_i \neq w_j$, and $p_0 = w_0 = \phi$.

In order to find the minimum distance, we need to invoke D four times with both subscripts decreased by no more than two. Hence, a brute-force evaluation of $D(P_m, W_\ell)$ must take $\mathcal{O}(2^{\min(m,\ell)})$ time. However, there are only $m{\times}\ell$ possible $D(P_i, W_j)$ for $1{\leq}i{\leq}m$ and $1{\leq}j{\leq}\ell$. The dynamic programming algorithm [Sel80] evaluates $D(P_m, W_\ell)$ by storing every possible $D(P_i, W_j)$ in a $m{\times}\ell$ table. Table 4.1 shows a $2{\times}3$ dynamic programming table for $P{=}$`ab` and $W{=}$`bbc`.

	$w_0 = \phi$	$w_1 = b$	$w_2 = b$	$w_3 = c$
$p_0 = \phi$	0	1	2	3
$p_1 = a$	1	$D(P_1, W_1)$	$D(P_1, W_2)$	$D(P_1, W_3)$
$p_2 = b$	2	$D(P_2, W_1)$	$D(P_2, W_2)$	$D(P_2, W_3)$

$\Longrightarrow$

	ϕ	b	b	c
ϕ	0	1	2	3
a	1	1	2	3
b	2	1	1	2

Table 4.1: Dynamic Programming

Furthermore, it is not necessary to evaluate each of $m \times \ell$ entries. Ukkonen [Ukk85] proposed an algorithm to reduce the table evaluations. His algorithm works as follows: Let C_j be the maximum i such that $D(P_i, W_j) \leq k$ for the given j (C_j=0 if no $D(P_i, W_j) \leq k$). Given a C_{j-1}, compute $D(P_i, W_j)$ up to $i \leq C_{j-1}$+1, and then set C_j to the largest i (0$\leq i \leq C_{j-1}$+1) such that $D(P_i, W_j) \leq k$. Chang [CL92] proved that this algorithm evaluates $\mathcal{O}(k^2)$ expected entries. As shown in Table 4.2, for P=`adfd` and W=`acdfbdf` of 4×7=28 entries, Ukkonen's algorithm evaluates only 15 entries when k=1.

	ϕ	a	c	d	f	b	d	f
ϕ	0	1	2	3	4	5	6	7
a	1	0	1	2	3	4	5	6
d	2	1	1	1	2	3	4	5
f	3	2	2	2	1	2	3	4
d	4	3	3	2	2	2	2	3

$\Longrightarrow$

	ϕ	a	c	d	f	b	d	f
ϕ	0	1	2	3	4	5	6	7
a	1	0	1	2	3	4		
d	2	1	1	1	2	3		
f	3		2	2	1	2		
d	4					2		
	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7

Table 4.2: Ukkonen's Cutoff

Initially, we have C_0=1. We evaluate the first column up to row C_0+1=2. The largest row such that $D(P_i, W_1) \leq 1$ is 2, i.e., C_1=2. Therefore, we evaluate the second column up to row C_1+1=3. The largest row such that $D(P_i, W_2) \leq 1$ is 2, i.e., C_2=2. Therefore, we evaluate the third column up to row C_2+1=3, and further to have C_3=2, C_4=3, and C_5=0. C_5=0 indicates that it is impossible to change any prefix of `adfd` to `acdfb` in less than one operation. In other words, we get $D(P_4, W_7)>1$ for sure. We need to evaluate more entries only if we want to know $D(P_4, W_7)$.

In the following sections, we assume all keywords end with a unique *end-of-word* symbol, e.g., the null symbol, so that no keyword in a dictionary is a proper prefix of another keyword. All keywords will be pairwise distinguishable.

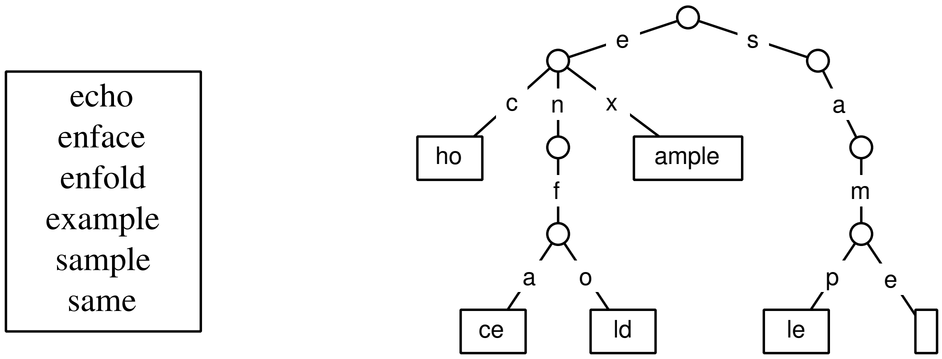


Figure 4.1: Dictionary Trie

4.2 Approximate Searching

Figure 4.1 shows a *dictionary trie* constructed over a set of six keywords. To make our discussion simpler, we use $|\Sigma|$-ary tries, where $|\Sigma|$ is the alphabet size. What we shall show is that the k approximate searching can be carried out by a depth-first traversal with cutoffs on the dictionary trie. The search will maintain a dynamic programming table during the traversal, and evaluate one column of the table per trie node. The result of the evaluation will tell whether the search should continue or the traversal should be cut off. The algorithm is based on the following two observations.

4.2.1 Observations

Observation I

Suppose we are searching keywords in Figure 4.1 for the best match to pattern string `sane`. To find the minimum distances of all keywords from `sane`, we need to evaluate six tables, one for each keyword. Table 4.3 shows three of them. For each table, the entries of the ith column depend only on entries of the $j \leq i$ th column, or the first i letters of the keyword. Keywords `sample` and `same` have the same prefix `sam`, and therefore, share the table entries up to the third column. And so does the first column of keywords `echo`, `enface`, `enfold` and `example`, the first three columns of keywords `enface` and `enfold`. In general, given a traversal path of length x, all the

	ϕ	s	a	m	p	l	e
ϕ	0	1	2	3	4	5	6
s	1	0	1	2	3	4	5
a	2	1	0	1	2	3	4
n	3	2	1	1	2	3	4
e	4	3	2	2	2	3	3
column:		1	2	3	4	5	6

	ϕ	s	a	m	e
ϕ	0	1	2	3	4
s	1	0	1	2	3
a	2	1	0	1	2
n	3	2	1	1	2
e	4	2	2	2	1
		1	2	3	4

	ϕ	e	n	$f \cdots$
ϕ	0	1	2	$3 \cdots$
s	1	1	2	
a	2	2	2	
n	3	3	2	
e	4	3	3	
		1	2	$3 \cdots$

Table 4.3: Dynamic Programming Tables

table entries for keywords inside the subtrie are identical up to the xth column.

With a depth-first traversal of the dictionary trie, the observation enables us to find the minimum distance to each of the keywords. Since each path from the root is a prefix shared by all keywords inside the subtrie, the corresponding columns of the dynamic programming tables are identical and need to be evaluated only **once**.

Observation II

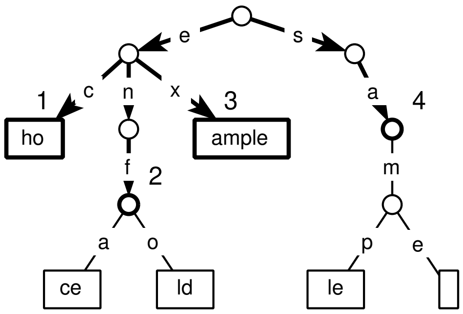
In the last table of Table 4.3, all entries of the second column are >1. If we are searching for keywords with k=1 mismatch, we can stop evaluating this table because for sure the distance between `sane` and `enface` or `enfold` will be >1. In the same way, after evaluating the fourth column of table `sample`, we can stop the evaluation because all entries of the column are >1.

This observation tells us that, if all entries for a trie path are $>k$, we can stop searching down the subtrie, because no word in the subtrie will have a distance $\leq k$. This is equivalent to cutting off the traversal of a subtrie when C_j=0 (see §4.1.2).

4.2.2 Algorithm

Suppose we have a misspelled word P=`exsample` and a dictionary trie as shown in Figure 4.1. We want to find all the keywords with k=1 mismatch. Figure 4.2 shows some intermediate results of the algorithm.

After evaluating $D(P, \texttt{ech})$, we find that entries on the third column are all ≥ 2. According to observation II, no keyword W with the prefix `ech` can have $D(P, W) \leq 1$.



Pattern String:	exsample	k ≤ 1

Depth First	String	Distance	Action
Search Path 1:	ech	≥ 2	reject
Search Path 2:	enf	≥ 2	cutoff
Search Path 3:	example	= 1	accept
Search Path 4:	sa	≥ 2	cutoff

Figure 4.2: Approximate Trie Searching

Since the search is on a leaf node, we reject keyword `echo` and continue the traversal. After evaluating $D(P, \texttt{enf})$, we know, once again, no keyword W with the prefix `enf` can have $D(P, W) \leq 1$, and therefore, there is no need to search down this subtrie. We cut off the subtrie and continue the traversal. Since `ech` and `enf` share the same prefix `e`, we copy the first column of `ech` when evaluating `enf` (observation I). After evaluating the search path 3, we find $D(P, \texttt{example}) = 1$. The traversal stops when the subtrie of search path 4, `sa`, has been cut off.

The search algorithm is essentially a depth-first traversal of a trie with cutoffs. Given a node n in the trie, the root-to-n path, $w_1 w_2 ... w_x$, is the longest prefix shared by all strings in $SubTrie(n)$. If changing $w_1 w_2 ... w_x$ to any possible prefix of the pattern costs more than k, there will be no string in $SubTrie(n)$ that has $\leq k$ mismatches with the pattern string. Hence, there is no need to search down to $Subtrie(n)$. A cutoff happens. Each letter w_j $(1 \leq j \leq x)$ on the path will cause evaluation of the jth column of the table. We use Ukkonen's algorithm to minimize the table evaluations. Algorithm 4.1 spells this out.

Algorithm 4.1 can also be used to search for the best match (a keyword with the shortest edit distance). This time, we first set k to a small number, say 5. Each time we find a better string, i.e., a string with a distance $d<k$, we replace k by d. k decreases monotonically during the search. A good initial k can be the edit distance to the string that shares the longest common prefix with P. This guarantees that k is never too small.

Algorithm 4.1 Dictionary Trie: Approximate Search

```
Var T :array [0..m][0..ℓ] of integer;          /* DP table, T[i,0]=i and T[0,j]=j */
    C :array [0..ℓ] of integer;                                /* C0 = C[0] = k */
    P, W :string;                                /* pattern and target string */
    k :integer;

Procedure ApproxMatch( n : Anode);
  begin
    if (n <> nil) then
      if (n is a leaf node) then
        for each symbol W[j] in the suffix do
          if Dist( j) = 0 then
            return                                      /* more than k mismatch */
        output W;
      else                                                  /* depth-first search */
        if (n.iLevel is aligned to a symbol) then
          if Dist( j) = 0 then                          /* W[j] is current symbol */
            return;                                                    /* cutoff */
        ApproxMatch( LeftChild( n));
        ApproxMatch( RightChild( n));
  end;

Function Dist( j :integer) :integer;
  begin                                                 /* evaluate one column */
    C[j] := 0
    for i := 1 to Min( C[j-1]+1, length(P)) do
      d1 := if (W[j] = P[i]) then 0 else 1;
      T[i,j] := Min( T[i-1,j-1]+d1, Min(T[i-1,j], T[i,j-1])+1);
      if (i > 1 and j > 1) then
        d2 := if (W[j] = P[i-1]) then 0 else 1;
        d3 := if (W[j-1] = P[i]) then 0 else 1;
        T[i,j] := Min( T[i,j], T[i-2,j-2]+d2+d3+1);
      if (T[i,j] <= k) then                                      /* update Cj */
        C[j] := i;

    return( C[j]);
  end;
```

4.3 Experimental Results

We have examined the size and search time of the pointer and pointerless trie representations in §1.2.3 and §3.2.2. Tries in those sections were used as index structures in which leaf nodes store pointers pointing to the actual data. Tries in this chapter are used to organize and store the data set, and therefore, the skipped symbols will be stored inside the trie structures. In this section, we shall look at the performance issues of this kind of trie structures.

We built *OrTrie*s to store three dictionaries: (1) dictionary used by UNIX *look* program, (2) *Webster* dictionary for NeXT, and (3) all words from (1) and (2). Words were separated by a new-line character. Table 4.4 shows the sizes of the three dictionaries and the corresponding *OrTrie*s. The search times were measured on a 25MHz NeXT with 28MB memory.

Dictionary	#Words (N)	File Size (n)	*OrTrie* Size	#Nodes (S_n)	Depth (A_n)
Unix (look)	25,144	0.21 MB	0.11 MB	156,634	43.5 (bits)
Webster	234,936	2.49 MB	1.15 MB	1,796,319	62.3 (bits)
Combined	240,009	2.53 MB	1.17 MB	1,821,125	61.9 (bits)

Table 4.4: Dictionary and Trie Sizes

4.3.1 Dictionary Trie Sizes

We use *OrTrie* to store the keyword list. A *OrTrie* requires two bits per node and a suffix string in each leaf node. When scanning to a leaf node, the search algorithm needs to know how long the suffix is. It needs a counter of, in the worst case, $\lceil$lg(max-suffix-length)$\rceil$ bits long. We may apply some compression techniques such as *Huffmann* encoding described in §3.2.1 to compressing the counters. In the following calculations, we assume to have counters of lg(average-suffix-length) bits on the average. According to the measurement, the average suffix length is 15.67 bits, the maximum suffix length is 139 bits, and the total number of trie nodes is $S_n = 7.16n$. Without considering the overhead of paging *OrTrie*s as described in §2.2 (which is less than 2% of the total trie size), a *OrTrie* takes:

$$\frac{n}{8}\left(2\times 7.16+15.67+\left\{\begin{array}{c}\lceil\lg 139\rceil\\ \lg 15.67\end{array}\right\}\right)=\left\{\begin{array}{cc}4.75 & \text{worst}\\ 4.24 & \text{average}\end{array}\right\}n \text{ bytes.}$$

In our implementation, the counters are large enough to indicate the longest suffix length. Comparing the actual trie sizes, as shown in Table 4.4, with the calculated sizes, we found the the discrepancies do not exceed 3%. If keywords are stored sequentially without any structure, the three dictionaries take 210KB, 2.49MB and 2.53MB respectively. Our *OrTrie*s compress these keyword lists by 48%, 54% and 54% respectively.

We could also use *PaTrie*, which requires one bit per node, reduces the total nodes to $2n-1$, but needs additional skip counters. For comparison, we found the average skip length (5.65), the maximum skip length (119 for *Webster*) and the suffix length parameters (the same as for *OrTrie*) based on the tested dictionaries. A *PaTrie* takes:

$$\frac{n}{8}\left(2+5.65+\left\{\begin{array}{c}\lceil\lg 119\rceil\\ \lg 5.65\end{array}\right\}+15.67+\left\{\begin{array}{c}\lceil\lg 139\rceil\\ \lg 15.67\end{array}\right\}\right)=\left\{\begin{array}{cc}4.79 & \text{worst}\\ 3.72 & \text{average}\end{array}\right\}n \text{ bytes.}$$

The skip counter plays the most important role in reducing the *PaTrie* size. If we simply let each skip counter hold the largest skip length, i.e., $\lceil\lg 119\rceil$, the *PaTrie* size will exceed the *OrTrie* size.

4.3.2 Search Times

Let $\rho(k)$ be an average number of columns evaluated before assuring that $D(P,W) > k$. $\rho(k)$ has two properties: (1) $\rho(k) > k$ if k is less than the target length, (2) $\rho(k) = \mathcal{O}(k)$ [Ukk85, CL92].

$\rho(k)$ relates to the search time. It indicates, on the average, how deep the search goes. If $\rho(k)$ is less than the average trie depth, the dynamic programming will take no more than $\rho(k)|\Sigma|^{\rho(k)}$ expected time, which is independent of the dictionary size. Here $|\Sigma|$ is the alphabet size. However, this expected worst time is a very loose upper bound even if for the small k which we are considering. We will measure $\rho(k)$.

We randomly picked up 14 words from each of the three dictionaries and modified them by 1, 2 and 3 edit operations based on randomly chosen positions. Thus, we had three sets of strings for each dictionary, 14 strings for each set, and at least one word from each dictionary has 1, 2 or 3 mismatches. We compared each of the strings with linear search and *OrTrie* search of the dictionaries. Linear search takes the same amount of I/O time for any k search. I/O time for *OrTrie* search is expected to be proportional to the number of accessed trie nodes. *OrTrie* search reads in fewer nodes for small k search. The measured results are shown in Table 4.5.

	Dictionary	$\rho(k)$	Linear Time	*OrTrie* Time	Accessed / Total
$k = 1$	Unix (look)	2.17	2.9 (*sec.*)	1.0 (*sec.*)	2.7 % (nodes)
	Webster	2.17	27.9 (*sec.*)	2.2 (*sec.*)	0.3 % (nodes)
	Combined	2.21	28.9 (*sec.*)	3.0 (*sec.*)	0.4 % (nodes)
$k = 2$	Unix (look)	3.30	4.6 (*sec.*)	3.1 (*sec.*)	15.2 % (nodes)
	Webster	3.32	44.9 (*sec.*)	9.8 (*sec.*)	2.8 % (nodes)
	Combined	3,35	49.4 (*sec.*)	10.9 (*sec.*)	3.1 % (nodes)
$k = 3$	Unix (look)	4.40	6.6 (*sec.*)	8.6 (*sec.*)	41.5 % (nodes)
	Webster	4.50	66.1 (*sec.*)	28.5 (*sec.*)	11.6 % (nodes)
	Combined	4.49	84.8 (*sec.*)	29.2 (*sec.*)	11.7 % (nodes)

Table 4.5: Approximate Search Times

We measured the ratio of the searched trie nodes against the total nodes. The results show that for $k = 1$, less than 3% of *OrTrie* nodes are searched. For large tries, the ratio is getting smaller, e.g., 0.4% for the dictionary of 240,000 words. Suppose we have a dictionary of 100MB. The *OrTrie* representation will shrink the dictionary to 50MB. 0.4% of 50MB is 200KB. That is to say, to search a 100MB dictionary for words of one mismatch results in reading 200KB information. For those searched nodes, most of them are near the trie root and are physically clustered in terms of trie pages.

In both searches, CPU times increase substantially due to dynamic programming. *OrTrie* search gets even worst because of its extensive bit masking. And therefore, it does not obtain one hundred percent speedup for $k = 1$. A faster computer should improve the search time accordingly.

4.4 Soundex Searching

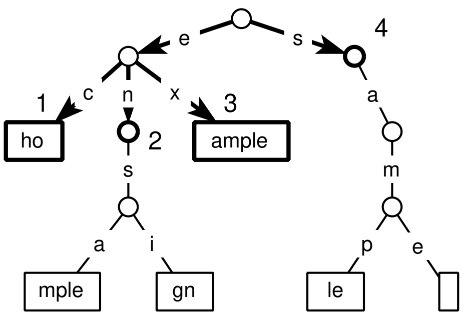
Spelling checkers based on the minimum edit distance work well for typographic misspellings. However, they often fail to detect phonetic errors. For example, `exsample` and `example` have one mismatch, but `nacherlly` and `naturally` have four. This section will explain how to adapt Algorithm 4.1 to detect phonetic misspellings.

The idea behind the Soundex system [OR22, Knu73] is to reduce strings into a code in which strings that are sounding similar (in English) will have an identical code. The Soundex code consists of the first letter of encoding strings followed by a sequence of digits (often truncated to 3). Digits are assigned to letters as follows:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 0 1 2 0 0 2 2 4 5 5 0 1 2 6 2 3 0 1 0 2 0 2
```

Zeros are removed and repeated digits are reduced to a single digit. For example, the Soundex code for `exsample` is `e2205140` ⇒ `e2514` and `example` is `e205140` ⇒ `e2514`.

Let $S{=}p_1d_1d_2..d_\ell$ be the Soundex code for pattern string $P{=}p_1p_2...p_m$ ($\ell{<}m$). Algorithm 4.1 is a depth-first search of *OrTrie*. For a given node $n{\in}OrTrie$, the root-to-n path, $w_1w_2...w_x$ ($x{\leq}m$), is the longest prefix shared by all strings in $SubTrie(n)$. The Soundex code, $w_1d'_1d'_2..d'_i$ ($i{\leq}\ell$), is also the longest Soundex prefix for all these strings. If $w_1d'_1d'_2..d'_i$ is not a prefix of S, then no Soundex code of strings in $SubTrie(n)$ matches S. We can stop searching down $SubTrie(n)$.



Pattern String:	exsample	e2514

Depth First	String	Soundex	Action
Search Path 1:	echo	e2	reject
Search Path 2:	en	e5	cutoff
Search Path 3:	example	e2514	accept
Search Path 4:	s	s	cutoff

Figure 4.3: Soundex Searching

Figure 4.3 shows search example for Soundex code `e2514`. When a search goes down to a leaf node, e.g., `ho`, we have a complete Soundex code. If this code is identical to the pattern code `e2514`, such as `example` $\Rightarrow$ `e2514`, the string will be accepted. If they are different, such as `echo` $\Rightarrow$ `e2` $\neq$ `e2514`, the string will be rejected. When searching an internal node, we get a partial (prefix) Soundex code, e.g., `s`. If this code is a prefix of `e2514`, we need to traverse down the subtrie to check the remaining code sequence. If not, we stop traversing down the subtrie since every Soundex code of strings in the subtrie cannot be equal to the sought Soundex code.

This code is either a prefix of `e2514`, and therefore needs to traverse down the subtrie to check the remaining code sequence, or not. For example, `s` is not a prefix of `e2514`. Since every Soundex code of strings in this subtrie is started with `s`, no string can be accepted, and therefore, we stop traversing down to this subtrie during the search.

4.5 Summary

Tries have been used for a long time as a dictionary structure for exact keyword lookup. In this chapter, we have expanded the exact trie search to approximate searching and Soundex search. Our trie structure, *OrTrie*, also compresses the keyword lists up to 54%. And the search is carried out directly upon the structure without any decompression operation.

The k approximate search algorithm of this chapter is a combination of the trie method and the dynamic programming technique. It stores keywords in a trie, and finds the approximate keywords by depth-first traversing the trie, and at the same time, evaluating a dynamic programming table to provide cutoffs of the traversal. The expected worst time is $\mathcal{O}(k|\Sigma|^k)$. This search time is independent of the dictionary size when $k \ll A_n$, the average depth of the dictionary trie. To our knowledge, no other published algorithm achieves this time complexity.

Chapter 5

Spatial Zooming

In this chapter, we shall propose a trie method to store and display map data. A major issue in representing and displaying large quantities of map data is how to change resolution, or level of abstraction, or remoteness, or *zooming*. The proposed trie structure permits displaying a map at any desired level of detail after reading from the file only the amount of data to be displayed. It gives a continuous zoom, say, from the full details of a digital map of many gigabytes of data, up to a mere outline, while storing only one copy of the map.

We assume map data are sequences of coordinate vectors. This chapter will focus on displaying two dimensional maps. However, the method under discussion works for other pictorial data, such as points, minimal bounding rectangles, triangulated polygons, cubes, k dimensional line segments, etc. Chapter 6 will discuss spatial data queries in general.

This chapter uses *FuTrie* only for spatial zooming because (1) it is the simplest trie and (2) it gives the exact data requested. Both *OrTrie* and *PaTrie* give much better compression (see §6.4.2). *OrTrie* will be used in Chapter 6 for spatial searching and other applications. *PaTrie*, however, does not improve data compression much further and requires a more complicated construction procedure. We do not consider *PaTrie* in spatial applications. This chapter is an extended version of [MS94].

5.1 Map Data Representation

5.1.1 Map Relation

A map contains many objects, both simple and complex. Simple objects are point data, such as houses, monuments, and elevations. They may have associated names, descriptors or values. Complex objects are collections of points, such as contours, coastlines, rivers, roads and boundaries. Points may be connected by cubic splines or quadratic segments, but are usually linked by straight edges. We are concerned with non-point data, i.e., the "complex objects".

We assume a map is represented as a set of edges. We think of it as a relation $MAP(Priority, P_s, P_e)$. Attribute $Priority$ is an integer and is used to distinguish map features in terms of importance. For example, major highways are more important than unpaved roads, and 100-foot contours are considered more important than 10-foot contours. More important features are given smaller priority numbers. Larger priority numbers are used in such a way as to include the smaller numbers, so that when low priority features are selected, the important ones will also be displayed. We do not always show priority, and when we do, we represent it as m bits, $p_{1-m}p_{2-m}...p_0$.

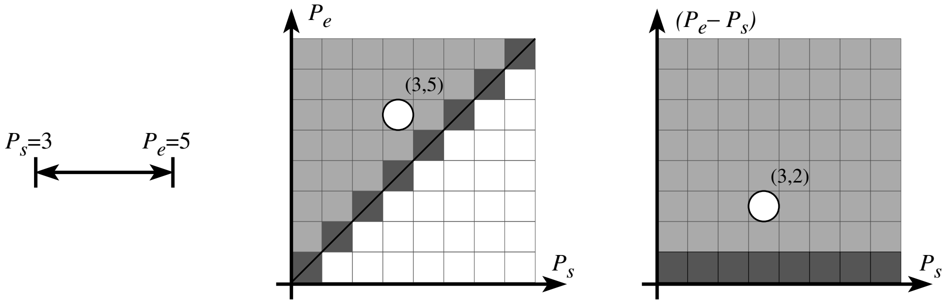
Attribute P_s and P_e are two k-dimensional points, and can be interpreted as two ending points of an interval (k=1), or an edge (k=2), or a line segment ($k \geq 2$), or two diagonal points of a rectangle (k=2), or a cube (k=3), etc. We assume that each coordinate is an integer of d-bits long.

5.1.2 Dimension Doubling and *ZoomTrie*s

We transform each (P_s, P_e) into a point, called *geometrical key*. A geometrical key, $K = p_1p_2...p_{2kd}$, is a bit string formed by interleaving bits of the $2k$ coordinates. For example, when k=2, the coordinates of two vertices can be derived as:

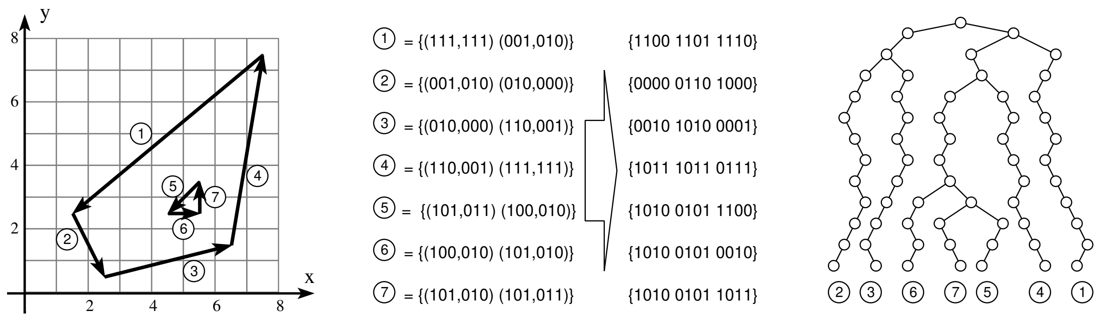
$$\begin{array}{lllllll} P_s & = & (x, y) & = & (x_1x_2...x_d, y_1y_2...y_d) & = & (p_1p_5p_9...p_{4d-3},\ p_2p_6p_{10}...p_{4d-2}) \\ P_e & = & (x', y') & = & (x'_1x'_2...x'_d, y'_1y'_2...y'_d) & = & (p_3p_7p_{11}...p_{4d-1},\ p_4p_8p_{12}...p_{4d}). \end{array}$$

If $(x, y) = (\mathit{111}, \mathit{111})$ and $(x, y) = (\mathit{001}, \mathit{010})$, we have $K = \mathit{1100\ 1101\ 1110}$ (see Figure 5.2 (b)). In *kd-tree* [Ben75], bit interleaving becomes *cyclic discriminators*. In



(a) Interval (3,5) (b) Start-End Rep. (c) Start-Length Rep.

Figure 5.1: Dimension Doubling



(a) A Map (b) Bit Interleaving (c) *ZoomTrie*

Figure 5.2: Map and *ZoomTrie*

Z-order [OM84], all points are connected by lexicographical order of their geometrical keys.

The transformation of a k-dimensional edge to a $2k$-dimensional point is called *transformation to parameter space* [NH85], *T-schemes* [SK88], or simply *transformation.* For edges, intervals, or rectangles oriented along the axes, it could be called, more distinctively, *dimension doubling* [MS94]. There are several approaches to dimension doubling. We illustrate two of them in Figure 5.1. Given an interval $(P_s, P_e) = (3, 5)$ at resolution 2^3 (a), we can represent it as a point (shown as a white disc) by using the two end points (b), or the start point and the length (c). The places where intervals can appear are shaded. The dark shading corresponds to the intervals of zero length.

We shall use the end-point representation, without claiming that it is the best. For example, the start-length representation usually compresses the data more, and occupies the full quadrant. However, other representations differ only in detail.

A *ZoomTrie* is a trie built from geometrical keys. Figure 5.2 shows a map (a), bit interleaving (b) and the *FuTrie* implementation of *ZoomTrie*.

5.1.3 Data Resolution

We define *edge at resolution* 2^r as an edge with each coordinate specified to the first r bits. For example, given a 2-dimensional edge (P_s, P_e) at the full resolution, the corresponding edge at resolution 2^r $(1 \le r \le d)$ is (P_s^r, P_e^r) and

$$\begin{aligned}(P_s, P_e) &= ((x_1x_2...x_d, y_1y_2...y_d),\ (x'_1x'_2...x'_d, y'_1y'_2...y'_d)) \quad = (P_s^d, P_e^d)\\ (P_s^r, P_e^r) &= ((x_1x_2...x_r, y_1y_2...y_r),\ (x'_1x'_2...x'_r, y'_1y'_2...y'_r)).\end{aligned}$$

To show an edge (P_s, P_e) on a display of $2^r \times 2^r$ $(1 \le r \le d)$ pixels, we need to scale each of the four coordinates, say x, by

$$x' = \left\lfloor x \times \frac{2^r}{2^d} \right\rfloor = \left\lfloor \frac{x}{2^{d-r}} \right\rfloor.$$

Operation $\lfloor x/2^{d-r} \rfloor$ is equivalent to removing the last $d-r$ bits of x, or retrieving the first r bits. This is equivalent to show (P_s^r, P_e^r). Figure 5.3 shows the map of Figure 5.2 at quarter resolution 2^1, half resolution 2^2 and full resolution 2^3.

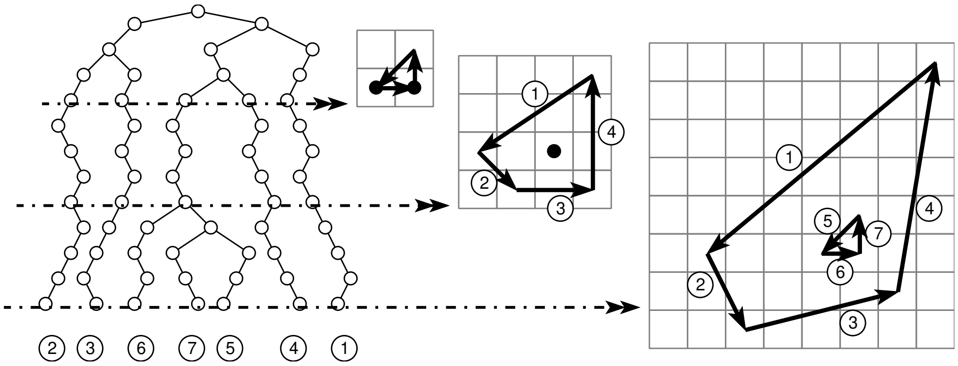


Figure 5.3: Zooming by *ZoomTrie*

To display (P_s, P_e) at an arbitrary resolution $r' \times r'$ $(1 \le r' \le 2^d)$, we scale x by

$$x'' = \left\lfloor x \times \frac{r'}{2^d} \right\rfloor = \left\lfloor \frac{x}{2^{d-\lceil \lg r' \rceil}} \times \frac{r'}{2^{\lceil \lg r' \rceil}} \right\rfloor \approx \left\lfloor \left\lfloor \frac{x}{2^{d-\lceil \lg r' \rceil}} \right\rfloor \times \frac{r'}{2^{\lceil \lg r' \rceil}} \right\rfloor.$$

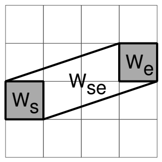
Since $\frac{1}{2} < (r'/2^{\lceil \lg r' \rceil}) \le 1$, the error of the approximate x'' is less than one pixel. In other words, if we show $(P_s^{r''}, P_e^{r''})$, the error is less than one pixel, where $r'' = d - \lceil \lg r' \rceil$. We shall only discuss resolution at 2^r.

Each (P_s^r, P_e^r) covers a set of edges. For example, a 2-dimensional edge $(P_s, P_e) = ((x, y), (x'y')) \subseteq (P_s^r, P_e^r)$ *if and only if*

$$\underbrace{x_1x_2...x_r0...0}_{d} \leq x \leq \underbrace{x_1x_2...x_r1...1}_{d}, \quad \underbrace{y_1y_2...y_r0...0}_{d} \leq y \leq \underbrace{y_1y_2...y_r1...1}_{d}$$
$$\underbrace{x'_1x'_2...x'_r0...0}_{d} \leq x' \leq \underbrace{x'_1x'_2...x'_r1...1}_{d}, \quad \underbrace{y'_1y'_2...y'_r0...0}_{d} \leq y' \leq \underbrace{y'_1y'_2...y'_r1...1}_{d}.$$

In general, a k-dimensional edge (P_s^r, P_e^r) defines a region $W^r = W_s^r \bigcup W_e^r \bigcup W_{se}^r$, where

$$\begin{aligned} W_s^r &::= \underbrace{p_1p_3...p_{2kr-1}00...0}_{kd} \leq P_s \leq \underbrace{p_1p_3...p_{2kr-1}11...1}_{kd} \\ W_e^r &::= \underbrace{p_2p_4...p_{2kr}00...0}_{kd} \leq P_e \leq \underbrace{p_2p_4...p_{2kr}11...1}_{kd} \\ W_{se}^r &::= \text{convex hull of } W_s^r \text{ and } W_e^r \text{ minus } W_s^r \text{ and } W_e^r \end{aligned}$$



All edges $(P_s^q, P_e^q) \subseteq W^r$ of higher resolutions $(q \geq r)$ are not distinguishable at resolution 2^r. Furthermore, $P_s^q \subseteq W_s^r$ and $P_e^q \subseteq W_e^r$. (P_s^r, P_e^r) gives an abstraction, or a zoom out, or an approximation view of covered edges (for both location and extension). The higher the resolution (the closer we look), the more precise the view of these edges. We shall use these ideas and notations in this and next chapters.

5.2 Displaying Operations

Given a MAP relation, we want to display the whole map (1), the map at resolution 2^r (2), the map with $Priority$ up to P (3) and a map region inside a rectangle window W (4). Formally, we are searching MAP relation for:

$$\begin{aligned} &(1) \quad \{(P_s, P_e) \mid (Priority, P_s, P_e) \in MAP\}, \\ &(2) \quad \{(P_s^r, P_e^r) \mid (Priority, P_s, P_e) \in MAP \ \wedge \ 1 \leq r \leq d\}, \\ &(3) \quad \{(P_s, P_e) \mid (Priority, P_s, P_e) \in MAP \ \wedge \ Priority \leq P\}, \\ &(4) \quad \{(P_s, P_e) \mid (Priority, P_s, P_e) \in MAP \ \wedge \ (P_s, P_e) \subseteq W\}. \end{aligned}$$

In the next two sections, we shall define two primitives: *Scan*() and *Search*(), as shown in Algorithm 5.1, for displaying operations.

Algorithm 5.1 *ZoomTrie* Primitives: *Scan* and *Search*

```
Procedure Scan( n : Anode; r : integer);
  begin
    if (n <> nil) then
      if (n.i_level <= m+4r) then                /* m is priority length in bits */
        Scan( LeftChild(n), r);
        Scan( RightChild(n), r);
      else
        output (P_s^r, P_e^r);                   /* find an edge */
  end;

Procedure Search( n : Anode; r : integer; P:predicate);
  begin
    if (n <> nil) then
      if (n.i_level <= m+4r+1) then              /* for each (P_s^r, P_e^r) ∈ SubTrie(n) */
        if ¬P(Priority) or ∃_{(P_s,P_e)∈SubTrie(n)} ¬P(P_s, P_e) then
          Search( LeftChild(n), r, P);           /* keep searching down */
          Search( RightChild(n), r, P);
        else
          Scan( n, r);                           /* collect edges ∈ SubTrie(n) */
  end;
```

5.2.1 *Scan*

Given a trie node n and a resolution 2^r ($1 \leq r \leq d$), function *Scan*(n, r) collects paths from the root to each node $x \in SubTrie(n)$ at level $m+4r+1$. Since paths of the top $m+4r+1$ levels represent all possible edges of resolution 2^r and their priorities, the function returns all $(P_s^r, P_e^r) \in SubTrie(n)$. For the *FuTrie* implementation of *ZoomTrie*, *Scan*() collects (P_s^r, P_e^r) without reading any unnecessary bits and more than one edges which are not distinguishable at the resolution.

With this primitive, we can draw a whole map (1) by invoking *Scan*(*root*, d), or a map at resolution 2^r (2) by invoking *Scan*(*root*, r). As shown in Figure 5.3, *Scan*(*root*, 1) for quarter resolution, *Scan*(*root*, 2) for half resolution, and *Scan*(*root*, r=d=3) for full resolution.

5.2.2 *Search*

Let n be a trie node and $\mathcal{P}$ be a predicate which is either $Priority \leq P$ or $(P_s, P_e) \subseteq W$. Here P is a $Priority$ and W is a rectangle window on the map. Function *Search*($n, r, \mathcal{P}$) collects nodes, say x, such that all $(Priority, P_s, P_e) \in SubTrie(x)$ satisfies $\mathcal{P}$, and some $(Priority, P_s, P_e) \in SubTrie(Parent(x))$ does not. In other words, *Search*() finds all largest subtries such that all keys within them satisfy $\mathcal{P}$. For each node x, *Search*() invokes *Scan*(x, r) to collect the edges.

We now can draw maps with priority $\leq P$ (3) by *Search*(*root*, *d*, $Priority \leq P$), or within window W (4) by *Search*(*root*, *d*, $(P_s, P_e) \subseteq W$). Each *Scan*() within a *Search*() can be changed to draw at a lower resolution in obvious ways. Algorithm 5.1 shows *Scan*() and *Search*().

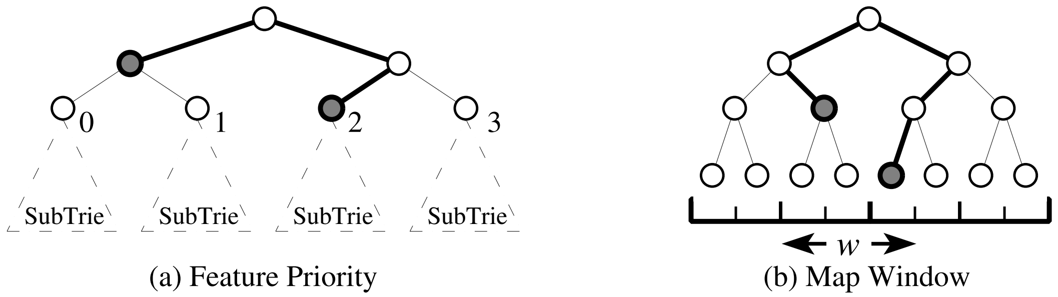


Figure 5.4: Priority and Window Searching

The search operations are illustrated in Figure 5.4. In (a), the top of a *ZoomTrie* is shown with paths to subtries of $Priority \leq 2$ highlighted. *Search*() identifies the two shaded nodes, and *Scan*() extracts all paths from the two subtries thus found and passes them to the draw routine. In (b), window W covers $\frac{3}{8} \cdots \frac{5}{8}$ of the map. *Search*() identifies the two shaded nodes and invokes *Scan*() to draw edges within the two corresponding subtries.

The window on the map specifies a region which can be described by a *PR quadtree* [Sam90] or *PR-Trie*. The *PR-Trie* has two different leaf nodes: black if the corresponding part of the space is contained in a region of interest; white if otherwise. Internal nodes lead to at least one black leaf and one white node. In fact, the *Search*() operation identifies and collects nodes that correspond to the black nodes of the window's *PR-Trie*. This operation can be made much faster by superimposing a precompiled *PR-Trie* on the *ZoomTrie*, instead of recomputing condition $\mathcal{P}$ for every path. Chapter 6 will give a general discussion on *ZoomTrie* searching.

5.3 Experimental Results

Maps used in the experiment are road and contour overlays extracted from 31H1, "Memphremagog", 1:50,000, from Energy, Mines and Resources of Canada. The road map has n=46,313 short edges and the contour map has n=483,063. All coordinates are 16 bit integers and the maximum resolution is, therefore, 64K×64K. Figure 5.6 and 5.5 show the road map and the contour map with contours at every 50 feet.

First, we compared *ZoomTrie*s with a simple filtering technique which read all the data but drew only visible edges. That is, after drawing an edge, it read the subsequent points in the sequence until the displacement was great enough to span at least one pixel. Then it drew the resultant edge. The data for this simple filtering was highly compressed, being stored as *differentials* after the first point in each sequence. Table 5.1 shows that *ZoomTrie*s also compress the data, but by a factor of about two less than the simple method. Both drawings are a little slow on our 25MHz NeXT, requiring waits in the order of minutes. *ZoomTrie* performs even worse because of the extensive bit masking operations. Data compression will be examined in §6.4.2.

	Simple Filter		*ZoomTrie*	
	File Size	Drawing Time	File Size	Drawing Time
Roads	0.22MB	25.4 *sec.*	0.31MB	93.2 *sec.*
Contours	1.7MB	112 *sec.*	3.2MB	957 *sec.*

Table 5.1: Map Filtering v.s. *ZoomTrie*

5.3.1 Resolution and Feature Priority

The results in Table 5.1 show *ZoomTrie*s in their poorest guise. The drawings in that case are presented to the full resolution of 64K×64K pixels. We now present *ZoomTrie*s for various resolutions, still displaying the whole map. For the two maps

Figure 5.5: Memphremagog Road Map

Figure 5.6: Memphremagog Contour (at Every 50 Feet) Map

using *ZoomTrie*s, Table 5.2 shows the time required to process the file without actually drawing it, as well as the total amount of time including drawing. It also indicates the number of nodes of the trie that are processed.

Resolution (pixels)	Contour with Priority 0	1	2	3	Road	(seconds)
512×512	12.1	25.3	56.6	107.3	13.5	Display+Search
	6.2	13.3	28.1	49.0	6.9	Search Only
	88K	187K	406K	733K	103K	Accessed #Nodes
1K×1K	22.2	46.5	108.4	218.8	23.1	Display+Search
	12.1	24.9	56.6	110.1	13.3	Accessed #Nodes
	178K	376K	855K	1651K	196K	Accessed #Nodes
2K×2K	33.8	71.1	168.5	351.6	35.1	Display+Search
	21.0	43.4	99.9	204.5	21.8	Search Only
	313K	662K	1544K	3118K	332K	Accessed #Nodes
4K×4K	45.0	94.4	226.4	476.9	46.8	Display+Search
	30.7	64.2	152.1	320.3	32.1	Search Only
	476K	1004K	2374K	4941K	499K	Accessed #Nodes
64K×64K	88.4	185.6	552.1	956.9	93.2	Display+Search
	74.4	155.5	388.0	820.6	78.3	Search Only
	1166K	2458K	5897K	12638K	1234K	Accessed #Nodes

Table 5.2: *ZoomTrie*s at Various Resolutions and Priorities

We see that the *ZoomTrie* is faster than simple filtering for resolutions up to a megapixel, in the case of the road map (which does not compress so much for the simple filter), and up to 512×512 pixels in the case of the contour map. We also see that the contour map, being ten times larger, is ten times slower than the road map.

Table 5.2 also shows the time for various priorities. We have assigned four levels of priority to the contour map: every 100 feet, every 50 feet, every 30 or 50 feet, and every 10 feet. These priorities are numbered 0, 1, 2, and 3, respectively. At priority 0, all resolutions are faster than the simple filter; at priority 1, resolutions up to 16 megapixels are faster; and at priority 2, resolutions up to one megapixel are faster.

We do not usually display at more than one megapixel, although we may plot at 16 megapixels. Figure 5.6 plots the map at resolution 4K×4K and with contours at every 50 feet. Figure 5.7 plots the same map but at a lower resolution, 256×256 pixels. It took only 14 seconds to draw the map. Note that zigzag patterns are visible on the map.

We could also have assigned priorities to the road map, determined by the number of digits in the route number. Similarly, rivers and coastlines, political boundaries, etc. could have been assigned priorities.

5.3.2 Windows on Map

Finally, we built the windowing algorithm, which requires an implementation of *search* operation in addition to *scan*. We picked four square windows about $\frac{1}{16}$th of the map area to represent various data complexities. Figure 5.8 shows one such zoom window on the bottom left corner of the contour map. The map region was plotted at resolution 4K×4K and with contours at every 10 feet.

Table 5.3 shows the processing time (without drawing) averaged over these four windows. We see that the larger numbers are $\frac{1}{16}$th of the corresponding results in Table 5.2.

Resolution	Contour with Priority (seconds)				Road
(pixels)	0	1	2	3	(seconds)
256×256	1.0	1.3	1.8	2.3	0.8
512×512	1.0	1.8	3.0	4.3	1.1
1K×1K	2.0	2.8	5.3	8.5	1.4
2K×2K	2.3	4.0	8.0	14.3	1.8
4K×4K	3.3	5.8	11.8	21.8	2.5
64K×64K	6.3	13.0	26.8	52.0	4.8

Table 5.3: Window Search Times

5.3.3 Extrapolations

The aggregation of all the measurements described above gives the plotting of the number of nodes against the map resolution, and drawing and processing time against the number of nodes shown in Figures 5.9 and 5.10.

We see that the number of nodes is initially exponential in the resolution, and then linear when it passes the trie height. The processing time is linear in the number of

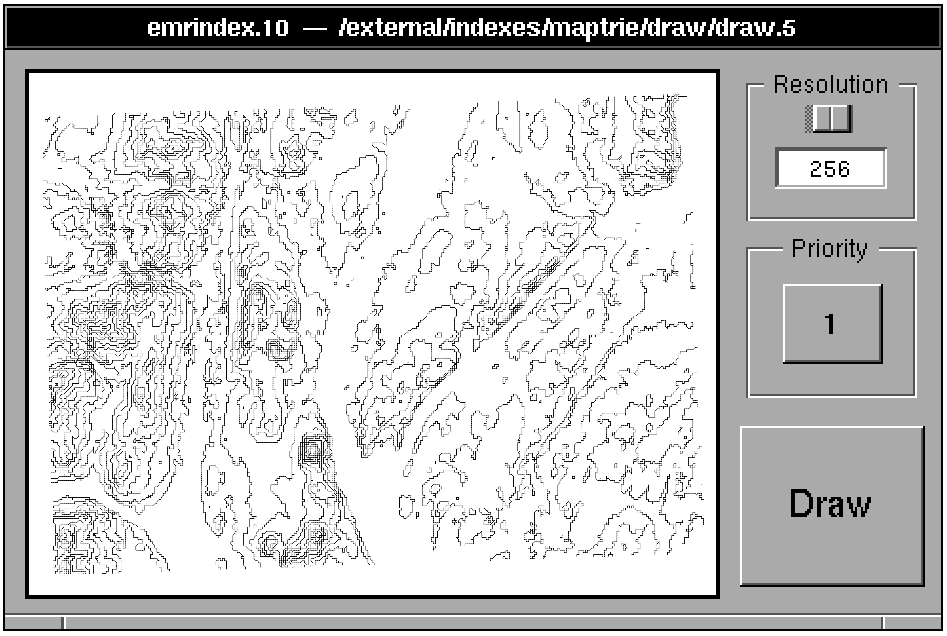


Figure 5.7: Contour Map at Resolution 256×256

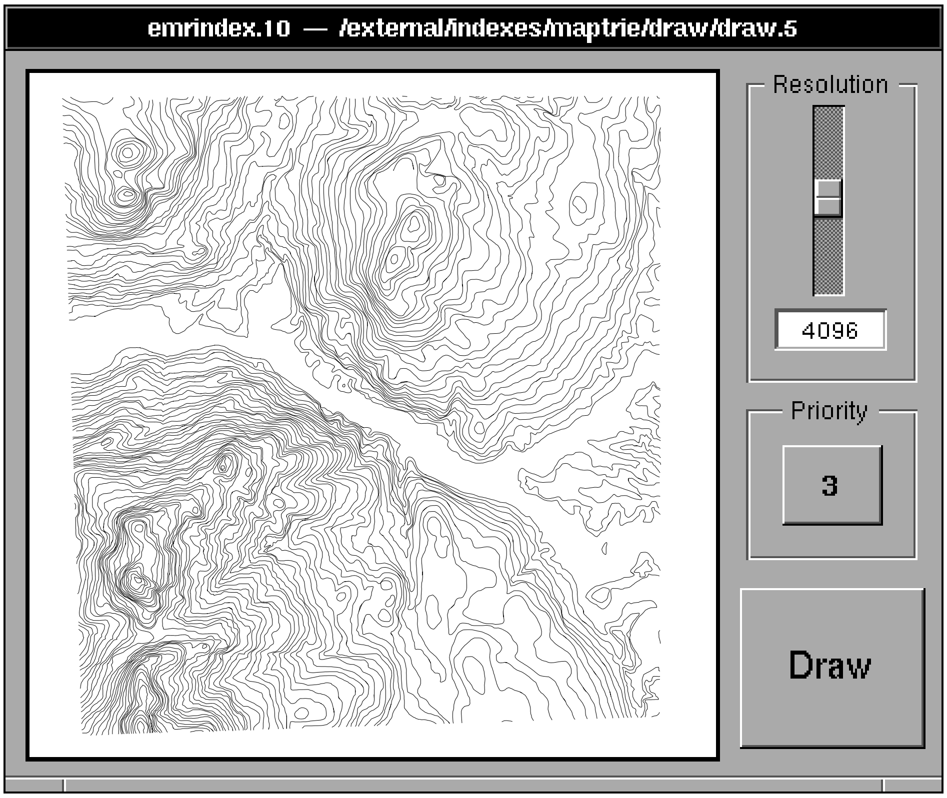


Figure 5.8: Contour Map Zooming

nodes. We made the linear regression fit the time data, and obtained

Scan() alone	$0.065 \times n + 0.195$
Search() and *scan*()	$0.074 \times n + 1.525$

where n is total leaf nodes in thousands.

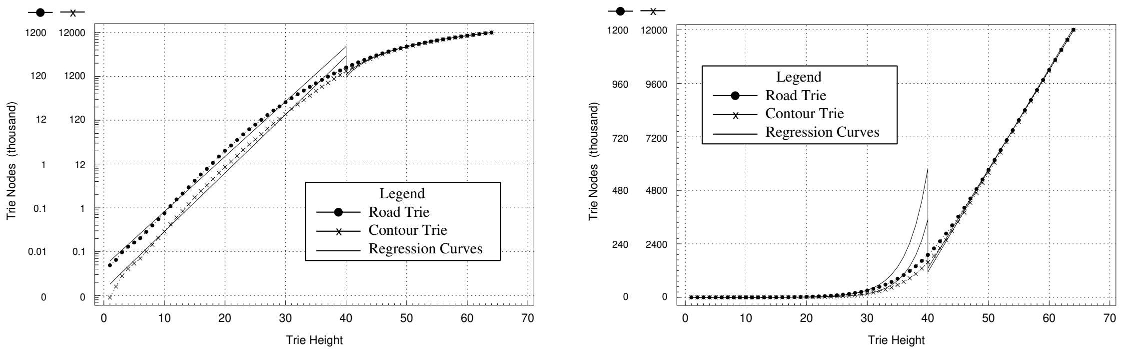


Figure 5.9: *ZoomTrie*: Trie Nodes v.s. Resolutions

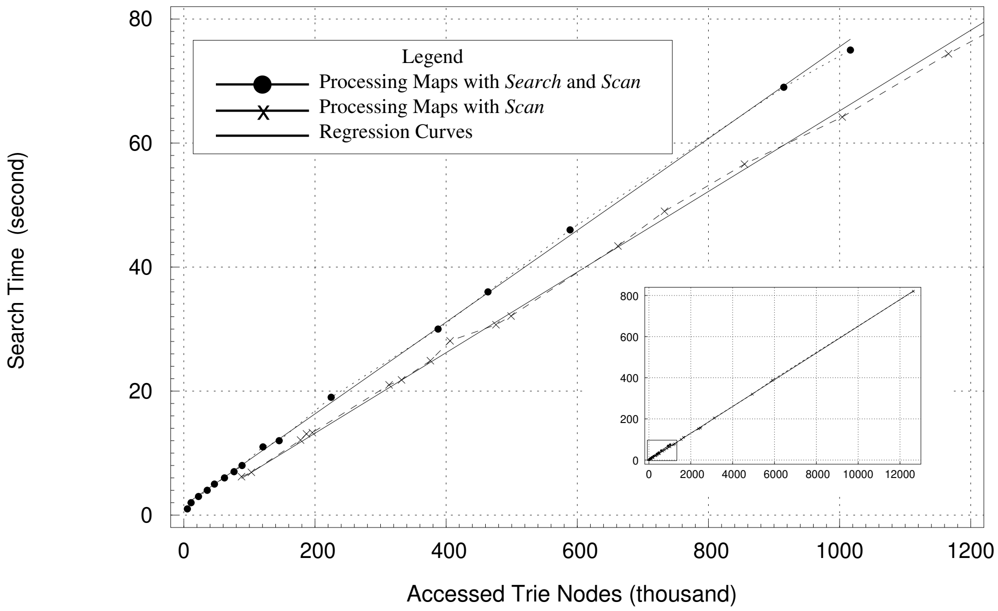


Figure 5.10: *ZoomTrie*: Search Times v.s. Accessed Nodes

The search time depends entirely on how much data is displayed, which in turn depends on the resolution and on the priorities selected. The search time does not depend on the size of the source file. This is just what we would expect. If we had all 13,000 maps for Canada at 1:50,000, the file would be some 80GB (assuming 32 bits per coordinate). But if it were plotted at 4K$\times$4K resolution, showing the kind of detail illustrated in Figure 5.6, it would require the same amount of time (assuming the same data distribution), i.e., about eight minutes on our workstation. By contrast, the same data displayed through a filter would require reading 40GB (the compression is about twice that of the *ZoomTrie*), which would take at least six hours on our machine.

The exponential-to-linear variation of nodes with resolution is also expected. The resolution is just the trie depth. The upper levels of the *ZoomTrie* resemble a complete binary tree, which grows exponentially. After a certain level, most paths will not bifurcate, so the number of nodes grows linearly.

For our 16 bits per coordinate, we found level 40 to give a reasonable break between the exponential and linear pieces. This corresponds to a resolution of 2^{10} in each coordinate, or 1K$\times$1K pixels. We did exponential and linear fits asymptotically, as shown in Figure 5.9. This gives the following normalized results (which must be multiplied by the total number of trie nodes).

	Exponential Part	Linear Part
Contour Map	$2^{(0.443\times r-19.5)}$ $(1\leq r<40)$	$0.038\times r-1.43$ $(40<r)$
Road Map	$2^{(0.417\times r-17.7)}$ $(1\leq r<40)$	$0.037\times r-1.37$ $(40<r)$

5.4 Summary

We have described a trie representation for map data which allows us to display maps at arbitrary levels of resolution, without reading from secondary storage any more data than is needed for the specified resolution. The basic technique can automatically match the amount of data retrieved with the number of pixels to be displayed. A simple refinement, which requires independent classification of features in hierarchies, permits selection of features in terms of importance. This selection can be independent of the resolution, or can be linked to it.

In order to show a map at different levels of detail, one could store several versions of the map in a hierarchy. For example, Energy, Mines and Resources of Canada provides a series of topological maps. At 1:250,000, a thousand maps are needed to cover the country. At the next level, 1:50,000, the linear resolution increases by 5, the area resolution increases by 5^2, and twenty-five thousand maps are needed. Compared with the *ZoomTrie* representation, this solution has four disadvantages. (1) The data are stored redundantly, once for each level. (2) The zoom is discontinuous and permits only a few levels of resolution. (3) The map is cut to pieces, and additional efforts are required to align them up. (4) The only way to adjust for the pixel size at each level is still to filter the input rather than avoid reading it.

Chapter 6

Spatial Querying

In this chapter, we shall investigate *ZoomTrie* search in general. It is important to establish that *ZoomTrie* can be used not only for the ubiquitous operations of display and plot but also for geometrical queries and other spatial data processing.

For spatial data structures, two basic issues have to be addressed: the efficient use of storage and the ease of locating objects based on the spatial proximity. *ZoomTrie* stores the common prefixes of all data elements only once each, which gives substantial spatial data compression. *ZoomTrie* enables us to work at various resolutions without having to filter a large data file or store different copies of the same data at different levels of detail. The two properties make *ZoomTrie* an efficient data structure to store and index spatial data.

Tries recursively partition a data space into equally sized subspaces. Each subspace contains at least one datum. If a subspace is outside the query region, then it does not contain answer data and need not be searched. If a subspace is inside the query region, then all the data in the subspace are answers and what we need do is to collect them. Otherwise, we have to partition the subspace by searching down the trie structures. As we shall show, this method takes $\mathcal{O}(n^{\frac{1}{2}})$ time in the worst case for all interval queries [All83], where n is the total number of intervals.

The algorithms and results of this chapter are for line queries: line-point, line-line and line-region. However, they are also suitable for point queries and can be generalized to region and other queries.

6.1 Query Categories

Let *GEO* be a set of (P_s, P_e). By analogy with the relational algebra [Cod70, Mer83], we define two types of queries: *Geometrical selection* which selects edges from a *ZoomTrie* built over one *GEO*, and *Geometrical join* which selects edge pairs from two *ZoomTrie*s. We focus on the predicate whose truth is determined for each edge without comparison with other edges. We call it a *linear predicate*. Similarly, the truth of a *quadratic predicate* is determined by an edge pair without comparing with other pairs. We formulate queries using predicate $\mathcal{P}$ on edges. That is

- $\left[\ (P_s, P_e) \ \mid\ (P_s, P_e) \in GEO \wedge \mathcal{P}(P_s, P_e)\ \right]$,
- $\left[\ (P_s, P_e, P'_s, P'_e) \ \middle|\ \begin{array}{c} (P_s, P_e) \in GEO \wedge (P'_s, P'_e) \in GEO' \\ \wedge\, \mathcal{P}(P_s, P_e, P'_s, P'_e) \end{array}\ \right]$.

6.1.1 Geometrical Selection: Examples

Linear Predicate

L0: Retrieve each edge in *GEO*:

$$\mathcal{P}(P_s, P_e) ::= (P_s, P_e) \in GEO$$

L1: Find intervals that are contained in interval I:

$$\mathcal{P}(P_s, P_e) ::= (I_s \leq P_s \leq I_e) \wedge (I_s \leq P_e \leq I_e)$$

L2: Find edges that connect to point P:

$$\mathcal{P}(P_s, P_e) ::= (P_s = P) \vee (P_e = P)$$

L3: Find edges that intersect with line L:

$$\mathcal{P}(P_s, P_e) ::= (P_s, P_e)\ \underline{intersect}\ L$$

L4: Find edges that are contained in region R:

$$\mathcal{P}(P_s, P_e) ::= (P_s, P_e)\ \underline{inside}\ R$$

L5: Find edges that are longer than ℓ:

$$\mathcal{P}(P_s, P_e) ::= \underline{distance}(P_s, P_e) \geq \ell$$

Non-Linear Predicate

NL1: Find the nearest edge to point P:

$$\mathcal{P}(P_s, P_e) ::= \forall_{({P'}_s,{P'}_e)\in GEO}(\underline{mindist}(P,({P'}_s,{P'}_e)) \geq \underline{mindist}(P,(P_s,P_e)))^{\dagger}$$

NL2: Find the longest edge:

$$\mathcal{P}(P_s, P_e) ::= \forall_{({P'}_s,{P'}_e)\in GEO}(\underline{distance}({P'}_s,{P'}_e) \leq \underline{distance}(P_s,P_e))$$

6.1.2 Geometrical Join: Examples

Quadratic Predicate

Q1: Find edge pairs such that one connects to the other:

$$\mathcal{P}(P_s, P_e, P'_s, P'_e) ::= P_e = P'_s$$

Q2: Find edge pairs such that one intersects with the other:

$$\mathcal{P}(P_s, P_e, P'_s, P'_e) ::= (P_s, P_e)\ \underline{intersect}\ ({P'}_s, {P'}_e)$$

6.2 *ZoomTrie* Search: Algorithms

*ZoomTrie*s in this chapter are implemented by the *OrTrie*. In contrast to the *FuTrie* implementation, an *OrTrie* has no chain nodes after the last binary node. The truncated paths are stored inside leaf nodes. This increases data compression. But we may retrieve more bits than necessary when search down to the leaf node.

6.2.1 Primitives

We define two primitives: *Scan*() and *Search*(), as shown in Algorithm 6.1, for linear predicates. *Scan*(n) collects all edges within a subtrie rooted at n. *Search*(*root*, 1, $\mathcal{P}$) finds all largest subtries such that they are as close to the root as possible and all edges inside them satisfy the predicate $\mathcal{P}$. *Search*() checks each $\mathcal{P}(P_s^r, P_e^r)$ $(1 \leq r \leq d)$ in the increasing order of resolution. The evaluation, as shown in Algorithm 6.1, yields three possible answers:

† $\underline{mindist}(P,(P_s,P_e))$ is the minimum distance from point P to edge (P_s,P_e).

Algorithm 6.1 *ZoomTrie*: Geometrical Selection

```
Procedure Search( n : Anode; r : integer; P :predicate);
  begin
    if (n <> nil) then
      if (n is not a leaf node) and (n.i_level <= 2kr) then
        Search( LeftChild(n), r, P);          /* for each (P_s^r, P_e^r) in the subtrie */
        Search( RightChild(n), r, P);
      else
        if ∀(P_s,P_e)∈SubTrie(n) P(P_s, P_e) then
          Scan( n);                              /* collect edges ∈ SubTrie(n) */
        else
        if ∀(P_s,P_e)∈SubTrie(n) ¬P(P_s, P_e) then
          return;                              /* ignore all edges ∈ SubTrie(n) */
        else                                    /* has to be an internal node */
          Search( n, r+1, P);         /* check each (P_s^{r+1}, P_e^{r+1}) ∈ SutTrie(n) */
  end;

Procedure Scan( n : Anode);
  begin
    if (n <> nil) then
      if (n is not a leaf node) then          /* for each (P_s, P_e) ∈ SubTrie(n) */
        Scan( LeftChild(n));
        Scan( RightChild(n));
      else
        output (P_s, P_e)                                       /* find an edge */
  end;
```

- $\mathcal{P}(P_s, P_e)$ is true for all $(P_s, P_e) \subseteq (P_s^r, P_e^r)$. We collect every $(P_s, P_e) \subseteq (P_s^r, P_e^r)$ by invoking *Scan*().
- $\mathcal{P}(P_s, P_e)$ is false for all $(P_s, P_e) \subseteq (P_s^r, P_e^r)$. We ignore all $(P_s, P_e) \subseteq (P_s^r, P_e^r)$ by stopping searching down the subtrie (cutoff).
- Otherwise, i.e., some of (P_s, P_e) satisfy $\mathcal{P}$ and others do not. We have to increase the resolution by searching down the subtrie.

Similarly, as shown in Algorithm 6.2, *Product*() and *Join*() are two primitives for quadratic predicates. *Product*(n, m) returns the *Cartesian* product of all edges in the two subtries rooted at n and m respectively. *Join*($n, m, 1, \mathcal{P}$) finds all the largest

Algorithm 6.2 *ZoomTrie*: Geometrical Join

```
Procedure Join( n, m : Anode; r : integer; P :predicate);
  begin
    if (n <> nil) and (m <> nil) then
      if (n is not a leaf node) and (n.i_level <= 2kr)
        Join( LeftChild(n); m, r, P);        /* for each (P^r_s, P^r_e) ∈ SubTrie(n) */
        Join( RightChild(n); m, r, P);
      else
      if (m is not a leaf node) and (m.i_level <= 2kr)
        Join( n, LeftChild(m); r, P);        /* for each (P'^r_s, P'^r_e) ∈ SubTrie(m) */
        Join( n, RightChild(m); r, P);
      else
        if ∀_{(P_s,P_e)∈SubTrie(n) ∧ (P'_s,P'_e)∈SubTrie(m)} P((P_s, P_e), (P'_s, P'_e)) then
          Product( n, m);                    /* output SubTrie(n) × SubTrie(m) */
        else
        if ∀_{(P_s,P_e)∈SubTrie(n) ∧ (P'_s,P'_e)∈SubTrie(m)} ¬P((P_s, P_e), (P'_s, P'_e)) then
          return;                            /* stop searching both subtries */
        else
          Join( n, m, r+1, P);               /* search down both subtries */
  end;

Procedure Product( n, m : Anode);
  begin
    if (n <> nil) and (m <> nil) then
      if (n is not a leaf node) then         /* for each (P_s, P_e) ∈ SubTrie(n) */
        Product( LeftChild(n), m);
        Product( RightChild(n), m);
      else
      if (m is not a leaf node) then         /* for each (P'_s, P'_e) ∈ SubTrie(m) */
        Product( n, LeftChild(m));
        Product( n, RightChild(m));
      else
        output( (P_s, P_e), (P'_s, P'_e));   /* output the edge pair */
  end;
```

subtrie pairs such that all edge pairs from them satisfy $\mathcal{P}$. Since *Join*() compares all edges in the two *ZoomTrie*s, it is an $\mathcal{O}(n^2)$ algorithm. However, a join such as merging two *ZoomTrie*s can be executed in linear time complexity.

6.2.2 Linear Predicate Precompiling

A *PR-Trie* is a trie structure for representing points and regions [Sam90]. It has two types of leaves: (1) black leaf if the corresponding subspace is inside the region of interest, and (2) white leaf if the corresponding subspace is outside the region of interest. An internal node leads to at least one black leaf, and so corresponds to a space overlapping the region of interest.

A linear predicate specifies a set of $2k$ dimensional points. The set forms the region of interest, and therefore can be represented by a *PR-Trie*. With such a *PR-Trie*, the *ZoomTrie* selection becomes a search for the keys covered by the black leaves. The procedure is: (1) superimpose the *PR-Trie* onto the *ZoomTrie*, (2) remove *PR-Trie* node if there is no corresponding *ZoomTrie* node, and (3) traverse the superimposed *ZoomTrie*. For black leaves, all edges inside the subtrie satisfy $\mathcal{P}$. We collect them by invoking *Scan*(). For white leaves, no edge inside the subtrie satisfies $\mathcal{P}$. We cut off the subtries.

One way to convert a linear predicate to the *PR-Trie* representation is to enumerate all possible answers and then to construct the trie. But this is too expensive. For some linear predicates, such as the examples shown in §5.1.3, *PR-Trie*s can be constructed more directly. In these cases, searches can be made much faster by precompiling the linear predicates into *PR-Trie*s, rather than recomputing the condition for every (P_s^r, P_e^r).

6.3 *ZoomTrie* Search: Implementations

The purpose of this section is to apply the four primitives in §6.2.1 to linear and quadratic predicates in §6.1. The discussion will focus on predicate evaluation, i.e., to check whether $\mathcal{P}$ is true or false for all $(P_s, P_e) \subseteq (P_s^r, P_e^r)$. In addition, we consider the two non-linear predicates in §6.1 and propose a special algorithm for them.

6.3.1 Linear Selection

L0: Retrieve each edges in *GEO*. This query invokes *Scan*(*root*) only.

L1: Find intervals that are contained in interval I:

$$\mathcal{P}(P_s, P_e) ::= (I_s \leq P_s \leq I_e) \wedge (I_s \leq P_e \leq I_e)$$

Intervals are transformed into point data in 2-dimensional space when constructing a *ZoomTrie*. Figure 6.1 shows an example of six intervals, `a`, `b`, `c`, `d`, `e` and `f` (a), their transformation (see §5.1.2) in 2-dimensional space (b), and the *OrTrie* implementation of *ZoomTrie* (c).

There are thirteen possible relationships between two intervals. Figure 6.2 (a) shows the relationships (using Allen's notation [All83]) to resolution 2^3. The labeled regions show the relationships to the given interval, the `=` sign. The eight squares on the diagonal indicate the five possible point-interval relationships.

Query **L1** consists of four interval relationships, i.e., `d, s, f` and `=`, which in general is a triangular region with apex at the `=` sign and base along the diagonal. This is shown in Figure 6.2 (b), where `I` $= (2, 5)$ is the interval in question, and a, b, c, d, e and f are the candidate intervals for containment. Figure 6.2 (c) shows the *PR-Trie* for this containment relation and (d) shows the result of superimposing (c) with Figure 6.1 (c). By superimposing, we mean to traverse both tries simultaneously. The test to go left (or right) requires a left (or right) descendant in both tries. If we come to a white leaf of the *PR-Trie*, no edge in the corresponding sub-*ZoomTrie* satisfies $\mathcal{P}$. If we come to a black leaf, all edges in the subtrie are answers.

In higher dimensions, intervals are rectangles aligned with the axes, and are represented in the same way as edges, so this query can be modified to give any of the other relationships between intervals in any number of dimensions.

L2: Find edges that connect to point P:

$$\mathcal{P}(P_s, P_e) ::= (P_s = P) \vee (P_e = P)$$

When searching *ZoomTrie* down to a node at level $2kr{+}1$, we have a path $p_1 p_2 ... p_{kr}$ which is an edge (P_s^r, P_e^r) and defines two rectangular regions W_s and W_e (see §5.1.3).

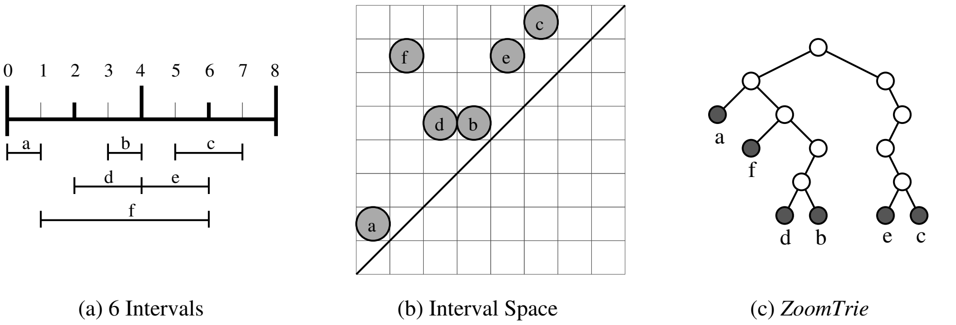


Figure 6.1: Interval Space

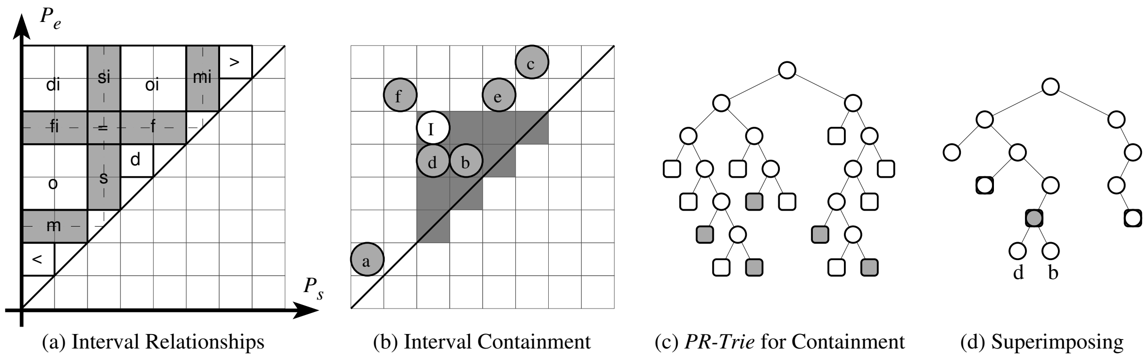
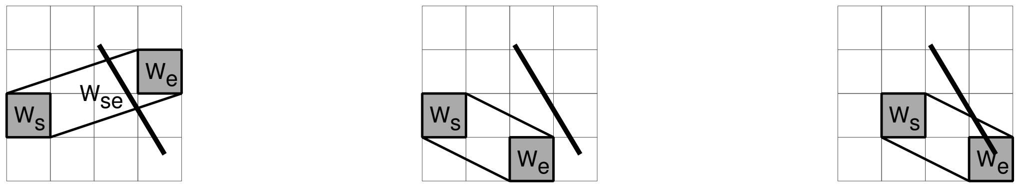


Figure 6.2: *PR-Trie* for Containment Searching



(a) All Crossing (b) None Crossing (c) Some Crossing

Figure 6.3: Edge-Line Crossing

If neither P $\underline{inside}$ W_s nor P $\underline{inside}$ W_e, i.e., the first r bits between P and P_s^r, P and P_e^r are different, no ending point of $(P_s, P_e) \subseteq (P_s^r, P_e^r)$ can be identical with P. We can cut off the search. Otherwise, we have to search the subtrie.

L3: Find edges that intersect with line L:

$$\mathcal{P}(P_s, P_e) ::= (P_s, P_e)\ \underline{intersect}\ L$$

When k=2, relationship between L and W^r (see §5.1.3) of (P_s^r, P_e^r) is:

$$\begin{aligned} \forall_{(P_s,P_e)\subseteq(P_s^r,P_e^r)}\mathcal{P}(P_s, P_e) &\equiv L\ \underline{intersect}\ W_{se}^r \\ \forall_{(P_s,P_e)\subseteq(P_s^r,P_e^r)}\neg\mathcal{P}(P_s, P_e) &\equiv L\ \underline{not\ intersect}\ W^r \end{aligned}$$

or otherwise, i.e., L touches W_s^r or W_e^r. We cannot tell whether they intersect in this case. We have to increase the resolution by calling *Search*(). Figure 6.3 shows the three situations.

L4: Find edges that are contained in region R:

$$\mathcal{P}(P_s, P_e) ::= (P_s, P_e)\ \underline{inside}\ R$$

For a given (P_s^r, P_e^r), we have

$$\begin{aligned} \forall_{(P_s,P_e)\subseteq(P_s^r,P_e^r)}\mathcal{P}(P_s, P_e) &\equiv W^r\ \underline{inside}\ R \\ \forall_{(P_s,P_e)\subseteq(P_s^r,P_e^r)}\neg\mathcal{P}(P_s, P_e) &\equiv W^r\ \underline{not\ intersect}\ R \end{aligned}$$

If R is convex (a special case), the predicate can be decomposed into a conjunction of two components, i.e., $\mathcal{P}(P_s, P_e) ::= P_s\ \underline{inside}\ R \wedge P_e\ \underline{inside}\ R$. For a given (P_s^r, P_e^r), if both P_s^r and P_e^r are inside R, then all $(P_s, P_e) \subseteq (P_s^r, P_e^r)$ satisfy $\mathcal{P}(P_s, P_e)$. If one of them is outside R, then no edge inside satisfies $\mathcal{P}(P_s, P_e)$. Otherwise, (P_s^r, P_e^r) overlaps with R and we have to check them in higher resolutions.

This example illustrates superimposition that involves three tries. We traverse the three tries simultaneously. On odd steps, the test to go left (or right) requires a left (or right) descendant of the first *PR-Trie* and the *ZoomTrie*. We do not traverse the second *PR-Trie* on odd steps. On even steps, the test requires a descendant of the second *PR-Trie* and the *ZoomTrie*. As before, we do not traverse the first *PR-Trie* on even steps. If two adjacent steps stay on black leaves of the *PR-Trie*s, it means every edge in the sub-*ZoomTrie* satisfies $\mathcal{P}$. If a step arrives at a white leaf, no edge in the subtrie satisfies $\mathcal{P}$.

L5: Find edges that are longer than ℓ:

$$\mathcal{P}(P_s, P_e) ::= \underline{distance}(P_s, P_e) \geq \ell$$

For the interval space, $\mathcal{P}$ specifies a triangular region, as shown in Figure 6.4 (a). This figure also shows the space to resolution 2^2 (b), and the *PR-Trie* (c).

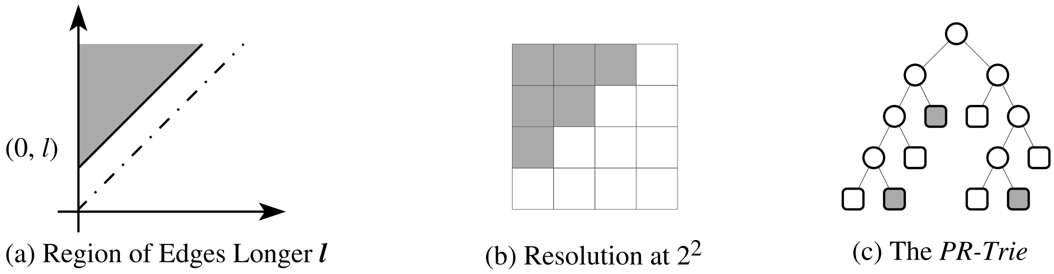


Figure 6.4: *PR-Trie* for Length Searching

For higher dimensions, this query becomes another example of a decomposable *PR-Trie*. Each dimension produces a *PR-Trie* that is the same as shown in Figure 6.4. We traverse the *ZoomTrie*, and at the same time, cycle among the *PR-Trie*s.

6.3.2 Non-Linear Selection

NL1: Find the nearest edge to point P:

$$\mathcal{P}(P_s, P_e) ::= \forall_{(P'_s, P'_e) \in GEO}(\underline{mindist}(P, (P'_s, P'_e)) \geq \underline{mindist}(P, (P_s, P_e)))$$

This is an optimization query which involves comparing edges with others. This query does not fall under the general primitives described in §6.2.1. We introduce Algorithm 6.3 (with `Dist` being previously initialized) to solve this query.

Algorithm 6.3 *ZoomTrie*: Searching the Nearest Edge

```
Procedure NearestEdge( n : Anode; r : integer);
  begin
    if (n <> nil ) then
      if (n is not a leaf node) and (n.i_level <= 2kr) then
        NearestEdge( LeftChild(n), r);          /* for each (P_s^r, P_e^r) in the subtrie */
        NearestEdge( RightChild(n), r);
      else
      if (MinDist(P, (P_s^r, P_e^r)) > Dist) then
        return;                                  /* no closer edge in the subtrie */
      else
        Dist := Min(Dist, MaxDist(P, (P_s^r, P_e^r)));     /* a better estimation */
        if (n is a leaf node) and (mindist(P, (P_s, P_e)) = Dist) then
          Answer := (P_s, P_e);                            /* find a closer edge */
        else                                               /* search down the subtrie */
          NearestEdge( n, r+1);
  end;
```

mindist() is the minimum distance from point P to point P_s, point P_e and the extended line of edge (P_s, P_e). For simplisity, we assume *mindist*() is the minimum distance from P to the ending points of (P_s, P_e). `Dist` is an approximate minimal distance which decreases monotonically during the search. A better initial `Dist` could be set to an ending point which shares the longest prefix bits with P.

MaxDist(P, (P_s^r, P_e^r)) is the distance from point P to the farthest corner points of W_s and W_e. Here W_s and W_e are rectangular regions defined by (P_s^r, P_e^r). *MinDist*(P, (P_s^r, P_e^r)) is the distance from P to the nearest corner points of W_s and W_e. However, if P is contained in W_s or W_e, *MinDist*() is always 0.

NL2: Find the longest edge:

$$\mathcal{P}(P_s, P_e) ::= \forall_{(P'_s, P'_e) \in GEO}(\underline{distance}(P'_s, P'_e) \leq \underline{distance}(P_s, P_e))$$

Algorithm 6.3 can be applied. But the function *MaxDist* needs to be redefined. Let $MaxDist(P_s^r, P_e^r)$ be the maximal distance from a corner point of W_s to a corner point of W_e. By definition, $Distance(P_s, P_e) \leq MaxDist(P_s^r, P_e^r)$ for any $(P_s, P_e) \subseteq (P_s^r, P_e^r)$. So, if $MaxDist(P_s^r, P_e^r)$ is less than the edge length found so far, we can cut off the subtrie. Otherwise, we have to search down the subtrie.

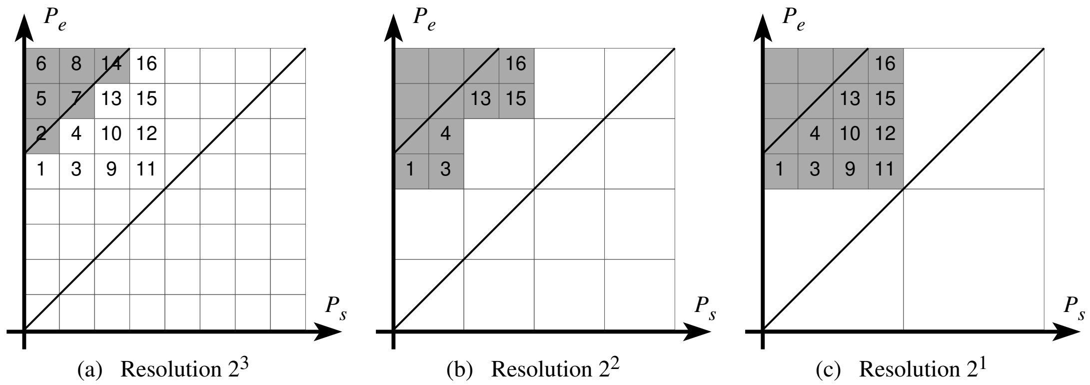


Figure 6.5: Variable-Resolution Querying

6.3.3 Variable-Resolution Selection

A major advantage of *ZoomTries* is that they are scale invariant and can be used to answer queries to any level of resolution. The cost of the query can be controlled by specifying the acceptable accuracy, i.e., to search the tries only down to a prespecified depth. However, we must decide what to do with edges at this resolution that may contain only some of the answer edges. The easiest supposition is to take all contained edges. This makes sense because at least one edge satisfies the query predicate. An alternative supposition is to take all contained edges if the number of satisfied edges exceeds a prespecified threshold.

Let us consider the first supposition and apply it to a query "find intervals that are not shorter than $l = 5$". Figure 6.5 shows the *PR-Trie* for the query at full resolution 2^d=2^3, half resolution 2^{d-1}=2^2, and quarter resolution 2^{d-2}=2^1. The shaded square boxes are answer intervals.

At the full resolution, the *PR-Trie* identifies all possible answers. At the half resolution, six extra intervals satisfy the predicate. Among those intervals, `3` and `15` have length 3. Their errors to l are 2. At the quarter resolution, four more intervals are added to the answer set. Interval `11` has length 1 and its error to l is 4. In general, at resolution 2^r ($r \leq d$), the maximum error of each endpoint is $2^{d-r}-1$ (by ignoring the $d-r$ least significant bits). Therefore, the maximum error for this length query is $2(2^{d-r}-1)$. If we satisfy the answer within a specified error, we can get it much faster, by searching the tries to only the appropriate depth.

6.3.4 Geometrical Join

Q1: Find edge pairs such that one connects to the other:

$$\mathcal{P}(P_s, P_e, P'_s, P'_e) = P_e = P'_s$$

Algorithm 6.2 checks $\mathcal{P}$ for every edge pair of $GEO \times GEO'$ in order of increasing resolution. Each edge pair $(P^r_s, P^r_e) \in GEO$ and $(P'^r_s, P'^r_e) \in GEO'$ defines two rectangular regions W_e and W'_s. If the two regions do not intersect with each other, then the two edges cannot share a same ending point, i.e., $P_e = P'_s$. We cut off the subtrie. Otherwise, we increase resolution r by calling $Join()$.

Q2: Find edge pairs such that one intersects with the other:

$$\mathcal{P}(P_s, P_e, P'_s, P'_e) ::= (P_s, P_e)\ \underline{intersect}\ (P'_s, P'_e)$$

It is hard to visualize the answer space, but the relationships between (P^r_s, P^r_e) and (P'^r_s, P'^r_e) can be classified as shown in Figure 6.6. If both W^r_{se} cross each other (a), then all edges contained in (P_s, P_e) intersect with all other edges contained in (P'^r_s, P'^r_e). If the two W^r are separated (b), then none of their edges intersect. Otherwise W^r_s or W^r_e intersects with the W^r of the other edge (c). In this case, we have to increase the resolution of both edges.

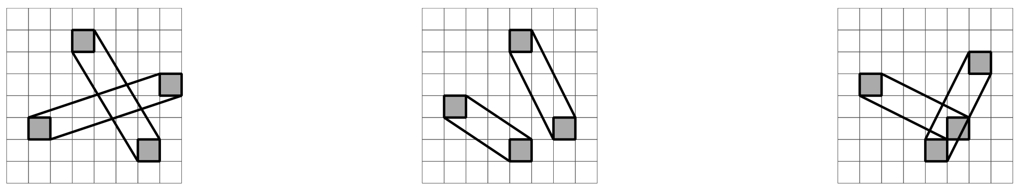

(a) All Crossing (b) None Crossing (c) Some Crossing

Figure 6.6: Edge-Edge Crossing

The consequence of this classification is that we can avoid comparing every possible edge pair when the answer set is less than n^2. Views at lower resolutions give partial answers. Only the edge pairs with ambiguity need further tests at higher resolutions.

6.4 Experimental Results

We built *OrTrie*s over two maps described in §5.3. This section shows *OrTrie* parameters, compression comparisons and search time analyses. All the measurements were carried out on a 25MHz NeXT with 28MB of memory.

6.4.1 *ZoomTrie* Trie Sizes

The *OrTrie* implementation requires two bits per each of S_n nodes, and a suffix string per leaf node. Since all geometrical keys have the same length, the suffix length at level i is of $2kd{-}i{+}1$ bits, and therefore, needs not to be indicated. On an average, a suffix has $2kd{-}A_n$ bits, where A_n is the average *OrTrie* depth. In total, *OrTrie* has $2S_n$ bits for the nodes, $n{\times}(2kd{-}i{+}1)$ bits for the suffixes, and less than 2% of the total size for the overhead of trie paging.

We measured A_n and S_n for the two *OrTrie*s at resolution 2^i ($3{\leq}i{\leq}16$). Figure 6.7 plots the measured results and analytical results for tries built over uniformly and independently distributed numbers (random trie). The case for A_n is worst than logarithm due to data clustering. We give a lower bound fit (the deeper A_n is, the smaller a trie will be). Table 6.1 shows the regression fits of the results.

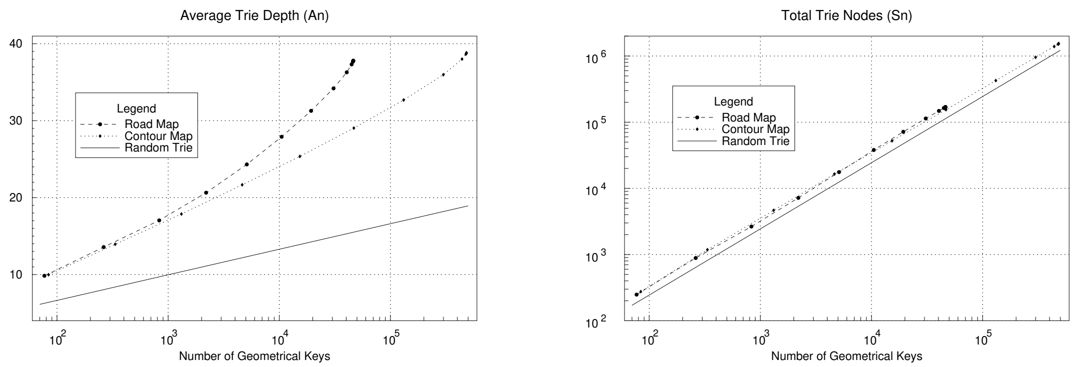


Figure 6.7: *ZoomTrie* Distributions

	Total Nodes (S_n)	Average Depth (A_n)	*OrTrie* Size
Random Trie	$2.44n$	$\lg n$	$2S_n + (2kd - A_n)n$
Road Trie	$3.66n$	$3.17 \lg n - 12.59$	
Contour Trie	$3.17n$	$2.35 \lg n - 6.19$	

Table 6.1: Regression Fitting

6.4.2 Data Compression

For data compression, we compare *OrTrie* with three other line representations which are not necessarily of the same expressive power — (1) set of edges: $(P_1, P_2 - P_1)$, $(P_2, P_3 - P_2), \cdots, (P_{n-1}, P_n - P_{n-1})$, (2) sequences of points: $P_1, P_2, \cdots, P_n$ and (3) sequences of differentials after the first points: $P_1, (P_2 - P_1), (P_3 - P_2), \cdots, (P_n - P_{n-1})$. For natural map data, such as contours and coastlines, edges are usually small. Table 6.2 shows the length statistics on our measured maps. We can see that the average length of differentials takes no more than one third of the coordinate bits. So, we assume that the average line differential is $< 2^{d/3}$, i.e., it takes $\frac{1}{3}d$ bits.

Under the above assumption and k=2 and d=32, the set representation (1) takes $(kd + kd/3)n$=$85.33n$ bits, the sequence representation (2) takes kdn=$64n$ bits, the differential representation (3) takes $kdn/3$=$21.33n$ bits, and the *OrTrie* representation (4), as seen in the previous section, takes $2S_n + (kd + kd/3 - A_n)n$ bits. Table 6.3 shows the sizes of the four representations.

	x Coordinate Maximum	y Coordinate Maximum	Line Differentials Maximum	Line Differentials Average	Line Differentials Std Dev
Road (bits)	40181 (16)	28925 (15)	1420 (11)	19.26 (5)	35.12
Contour (bits)	40147 (16)	29110 (15)	466 (9)	15.22 (4)	12.25

Table 6.2: Map Overlay Statistics

	(1) Set	(2) Sequence	(3) Differential	(4) *OrTrie* Road	(4) *OrTrie* Contour
$n = 10^3$	10.67KB	8KB	2.67KB	9.19KB	9.30KB
$n = 10^6$	10.67MB	8MB	2.67MB	5.23MB	6.36MB
$n = 10^9$	10.67GB	8GB	2.67GB	1.27GB	3.42GB

Table 6.3: Line Representation Comparison

As we have seen, when $n = 10^9$, the road *OrTrie* size is only 12.7% of the set file and 47.6% of the differential file. In the case of the contour map, the *OrTrie* size takes 42.8% of the set file and 128.1% of the differential file. However, we do not need to scan the whole file to restore the coordinates of the last point. In general, the larger the n is, the more compact the *OrTrie* will be. Since the *OrTrie* is designed for secondary storage, it performs well in both compressing and retrieving very large spatial data.

6.4.3 Search Time

We implemented and measured some of queries shown in §6.1. First of all, we expect the search time to be linearly proportional to the number of visited trie nodes. This is confirmed by the measured times plotted in Figure 6.8. For linear and quadratic predicates, we expect the number of visited trie nodes to be proportional to the number of selected (output) geometrical keys. In other words, the search time should not depend on the size of the source file. For non-linear selections, the search should visit only a small amount of the trie nodes.

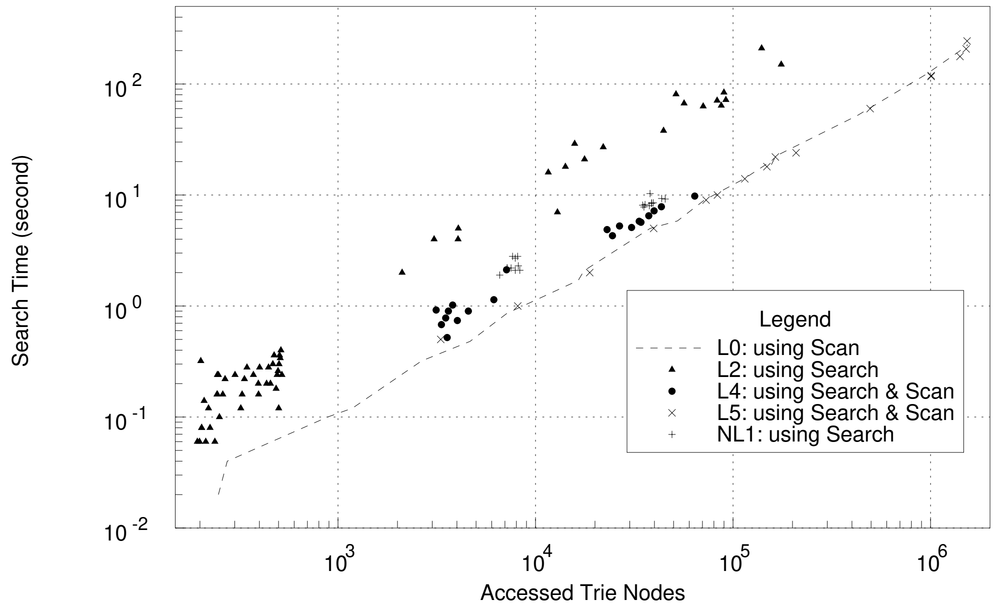


Figure 6.8: *ZoomTrie* Search Times

Selection Time

We measured **L0** by retrieving all the map edges at each resolution 2^r $(3 \leq r \leq d)$. As shown in Table 6.8, the total number of visited nodes is strictly proportional to the number of output keys.

In order to determine whether one ending point of an edge is identical to the given point P, **L2** has to check every bit of the ending point, and hence, invokes *Search*() only. The search time of *Search*() depends on complexity of the predicate. As we can see from Figure 6.8, for the same number of visited nodes, it takes **L2** about an order of magnitude more time than **L0**. This indicates that the search is CPU bound.

For simplicity, we chose region R of **L4** as rectangular windows which covered about $\frac{1}{48}$th of the whole map area. We measured R on ten randomly chosen locations. This query invokes *Search*() to locate R and *Scan*() to collect edges inside R. On the average, each search visited 2.53% of the total nodes for the road *OrTrie* and 2.32% for the contour *OrTrie*. Just as we expected, this is the amount of data required, i.e., $\frac{1}{48} \approx 2.1\%$.

For **L5**, we measured ℓ at ave, $ave{+}\sigma$, $ave{+}2\sigma$, $ave{+}4\sigma$, $ave{+}8\sigma$, $\frac{1}{2}max$, max and $2max$, where ave is the average map edge length, σ is the standard deviation of the length, and max is the longest edge. **L5** takes about the same amount of time as **L0** because of the simplicity of the predicate checking.

NL1 was measured using ten randomly selected points on both maps. On an average, **NL1** visited 4.25% of the total trie nodes for the road *OrTrie* and 2.52% for the contour *OrTrie*. Overall, our algorithm searches less than 5% of the whole file.

Join Times

We constructed *OrTrie*s over edges (from 5 to 2000) randomly selected from the two maps, and then joined them with the original *OrTrie* maps. As we can see from Figure 6.9, the total numbers of searched keys or trie nodes are strictly linearly proportional to output keys. In other words, the join time does not depend on the size of the source file. In general, we expect the join algorithm to have time complexity proportional to the numbers of the output keys, rather than the *OrTrie* sizes.

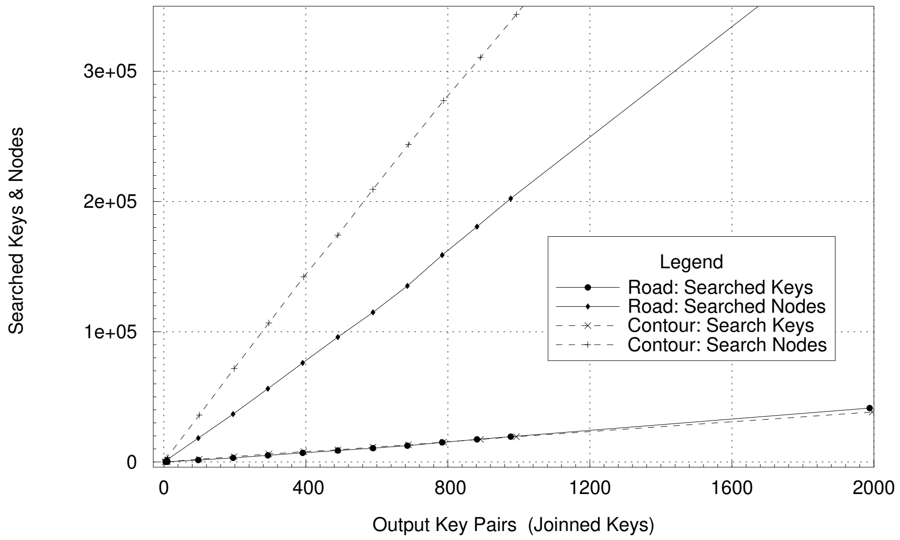


Figure 6.9: *ZoomTrie* Join Performance

The Worst Case Analysis

As we have shown in §6.2.2, queries based on linear predicates can be answered by superimposing *ZoomTrie* on a *PR-Trie*. For interval queries, as shown in Figure 6.2 (a), all possible answers form a single region. It has been proved that *PR-Trie* for a polygon with perimeter p at resolution 2^r has a maximum $\mathcal{O}(p+r)$ nodes [Hun78, Dye82, Sam90]. As a consequence of superimposing *ZoomTrie* on *PR-Trie*, we search only those nodes that are on the boundary of the queried region. The rest of space is either cut off or scanned.

Given n uniformly distributed intervals in 2-dimensional space, there are $\mathcal{O}(n^{\frac{1}{2}})$ intervals on the boundary of a queried region. In other words, we can answer any one of the thirteen interval queries in $\mathcal{O}(n^{\frac{1}{2}})$ worst time. In general, our search method reduces the data dimensions by one, i.e., from $2k$ data space to $2k-1$ search space, in the worst case.

6.5 Summary

This chapter has demonstrated that *ZoomTrie* can be used beyond displaying or plotting operations. We have shown how to query and process spatial data using *ZoomTrie*s, and given general query methods for linear and quadratic predicates, i.e., predicates that seek edges satisfying conditions that do not involve other edges. We have also presented special algorithms for two non-linear predicates, and shown how to specify the resolution acceptable for controlling the query costs. The experimental results confirm our expectations: the query cost depends only on the amount of data needed.

We have observed that for natural map data, e.g., contours and coastlines, edges are very short. That is, ending points will lie very close to the diagonal in the double-dimensional space. This effectively reduces the dimensionality by one. By removing one dimension, *ZoomTrie* compresses the spatial data still further. Our extrapolation shows, when $n = 2^{30}$, *ZoomTrie* size can be as small as 12.7% of the set file, and 47.6% of the differential file. The larger the spatial data is, the better the *ZoomTrie* performs.

Algorithms and results presented in this chapter are for line queries. However, they can be extended for polygon queries. The basic idea is to replace polygons by their minimal bounding rectangles or triangulations. We shall leave this for future research.

We have concentrated on *FuTrie* and *OrTrie* implementation of the *ZoomTrie*. We do not report our experiments with the *PaTrie* implementation because of limitation of space. However, the general conclusions are (1): a *PaTrie* does not give much further compression as comparing with *OrTrie*, (2): a *PaTrie* is more difficult to construct.

Chapter 7

Conclusion

The main objective of this thesis has been to design trie structures for secondary storage and apply them to indexing, storing and querying text and spatial data. This chapter summarizes contributions and the major results of this thesis project. Some future research is also outlined.

7.1 Claim of Originality

To the author's knowledge, the methods and the corresponding experimental results listed below are the original contributions of this thesis:

- *PaTrie*, a pointerless representation for the binary Patricia trie.
- Construction algorithms for very large *FuTrie*, *OrTrie* and *PaTrie*.
- Statistics on text tries — *OrTrie* and *PaTrie*, dictionary tries — *OrTrie* and *PaTrie*, and map tries — *FuTrie*, *OrTrie* and *PaTrie*.
- *PaTrie* for indexing large texts.
- *OrTrie* for storing large dictionaries and k-approximate string matching.
- Trie algorithms for spatial zooming and spatial querying by zooming.

7.2 Contributions

Trie Organization

We have proposed three pointerless structures, *FuTrie*, *OrTrie*, and *PaTrie*, for various binary tries. The data structures have two distinctive features: (1) they store no pointers and require two bits per node in the worst case (*FuTrie*), and (2) they are partitioned by pages and are suitable for secondary storage. Our experimental results have shown that the trie structures have excellent performance in both storage compactness and I/O efficiency. Therefore, the proposed structures are particularly useful in applications that deal with persistent bulk data.

We have investigated large trie constructions for static data. We have mapped the trie construction problem to the well studied sorting problem. In particular, we have proved that Patricia trie construction is a special case of parsing expressions with an operator precedence. Instead of spending 80 hours of computer time to build a Patricia trie of one million leaves (our first experience with large trie construction), we now need less than 10 minutes to build the same trie. We have given two external sorting algorithms for numerous and extremely long sistrings: one requires a large intermediate workspace and the other takes longer running time. The latest report on the PAT array says it can be built over a weekend for the *New OED*. Our *PaTrie* construction takes 18 to 55 hours for a comparable text, but requires more working space than used in earlier work.

Text Searching

We have applied trie methods to indexing very large text documents on secondary storage (text trie). By examining statistics for various text tries, we have concluded that the Patricia trie performs much better than other tries when indexing text files. We have shown that our *PaTrie* implementation is 10% – 25% smaller than the best previous data structure. This difference is important since the index size is crucial to the trie approach. Our search time is several times faster than the competitive trie indexes, and our method retains all the flexibility of the other trie methods. We have also presented methods for dynamic index tries, so that the text may change.

We have demonstrated that dictionary tries for English words are 60% (*OrTrie* representation) to 70% (*PaTrie* representation) smaller than the simple lists of the words. Dictionary tries gives approximate views of words. Combined with the dynamic programming technique, tries are used to solve the k approximate approximate string matching problem. The expected worst time of our algorithm is $\mathcal{O}(k|\Sigma|^k)$, which is independent of the dictionary size and the search string length. Here, we assume k, the number of mismatches, is very small, say less than 4. We have also shown that dictionary tries can be used for Soundex code searching.

Spatial Data Zooming and Querying

We have applied trie methods to representing and indexing spatial data on secondary storage (map trie). We have proposed the *ZoomTrie* structure for map data storing, displaying and querying. *ZoomTrie* permits us to query and retrieve the data at arbitrary levels of resolution, without reading from secondary storage any more data than is needed for the specified resolution. Our performance results on map displaying have confirmed that the processing cost is linear in the amount of data needed and independent of the total data size.

We have described a general *ZoomTrie* query method for linear (and quadratic) predicates that seek edges (and edge pairs) satisfying conditions that do not involve other edges (and edge pairs). We have given specific algorithms for a set of sample queries ranging from geometrical selection and geometrical join, to the nearest neighbour. We have also shown how to specify the acceptable resolution to control the query cost. We have implemented and tested most of the queries. The performance data on map querying has confirmed our expectations: the cost depends only on the amount of data needed.

7.3 Future Research

We have modeled texts as sequences of symbols and provided a trie structure to store extremely long sequences (sistring) in a very compact way and yet to preserve search efficiency. Many applications require storing, searching, and manipulating long sequences, e.g., molecular biology, human speech recognition, file comparison, text editor, etc. Although trie searching for exact subsequences (see §3.5) has been widely explored, the full use of text and dictionary tries nevertheless is still an open problem.

One classical problem of sequence manipulations is to find the longest common substring among k strings (k-LCG problem). The longest repetition search [GBYS92, ST93] solves k-LCG for k=1. We have proposed a solution to arbitrary k in §3.5. We could also extend trie hashing by using a binary Patricia trie as the hash function. Binary Patricia tries retain only the bits that distinguish a key from the others. All irrelevant bits of keys are removed. This will give better hashing performance.

The approximate searching of dictionary tries has many potential applications, e.g., removing duplicated entries from a mailing list in which a name and address may have been written in different forms or with a few misspellings, and finding the right molecule when measurement of atom quantity is inaccurate, etc.

Spatial data in *ZoomTrie*s are organized in a Z-order. It has been noted that a Z-order is not a continuous mapping, i.e., spatially nonadjacent points can become adjacent in the Z-order space. Discontinuity degenerates range query performance, specially when data are stored in secondary storage. Trie methods based on other mapping schemes need to be explored.

We have not mentioned updating spatial data represented by *ZoomTrie*s. The problem boils down to the ability to update tries, which we have discussed for very large text tries. We anticipate no difficulty in making occasional changes to maps. However, we have shown that trie construction methods are much more efficient when all the data are added at once than when a large trie grows one edge at a time.

Bibliography

[AC75] A.B. Aho and M.J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–40, 1975.

[AHU83] A.B. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.

[All83] A.F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–43, 1983.

[Aoe89] J.I. Aoe. An efficient digital search algorithm by using a double-array structure. *IEEE Transactions on Software Engineering*, 15(9):1066–77, 1989.

[ASU86] A.B. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[AVL62] G.M. Adelson-Velskii and E.M. Landis. Doklady akademia nauk SSSR. *English translation in Soviet Math*, 3:1259–63, 1962.

[Ben75] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–17, 1975.

[Ber84] T. Berger. Poisson multiple access for packet broadcast channels. *IEEE Transactions on Information Theory*, IT-30:745–51, 1984.

[BHKS93] T. Brinkhoff, H. Horn, H.P. Kriegel, and R. Schneider. A storage and access architecture for efficient query processing in spatial database systems.

In *Proceedings of 3rd International Symposium, SSD'93*, pages 130–45, Singapore, June 1993.

[BM72] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 13:173–89, 1972.

[BM77] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–72, 1977.

[Bri59] R. De La Briandais. File searching using variable length keys. In *Proceedings of the Western Joint Computer Conference*, volume 15, pages 295–8, New York, 1959. Spartan Books.

[BS89] R.M. Bozinovic and S.N. Srihari. Off-line cursive script word recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(1):68–83, 1989.

[BU77] R. Bayer and K. Unterauer. Prefix B-trees. *ACM Transactions on Database Systems*, 2(1):11–26, 1977.

[BY89] R.A. Baeza-Yates. *Efficient Text Searching*. PhD Dissertation, Computer Science Department, University of Waterloo, May 1989. Research Report CS-89-17.

[BYG89] R.A. Baeza-Yates and G.H. Gonnet. Efficient text searching of regular expressions. In *Proceedings of 16th International Colloquium on Automata, Languages and Programming*, LNCS 372, pages 46–62, Stresa, Italy, July 1989. Springer-Verlag.

[BYP92] R.A. Baeza-Yates and C.H. Perleberg. Fast and practical approximate string matching. In *Proceedings of 3rd Annual Symposium on Combinatorial Pattern Matching*, LNCS 644, pages 185–92, Tucson, Arizona, April 1992. Springer-Verlag.

[Cap79] J.I. Capetanakis. Tree algorithms for packet broadcast channels. *IEEE Transactions on Information Theory*, IT-25(5):505–15, 1979.

[CHK85] G.V. Cormack, R.N.S. Horspool, and M. Kaiserswerth. Practical perfect hashing. *Computer Journal*, 28(1):54–8, 1985.

[CL92] W.I. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proceedings of 3rd Annual Symposium on Combinatorial Pattern Matching* [BYP92], pages 175–84.

[Cod70] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–87, 1970.

[CS77] D. Comer and R. Sethi. The complexity of trie index construction. *Journal of the ACM*, 24(3):428–40, 1977.

[Dam64] F.J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–6, 1964.

[Dev82] L. Devroye. A note on the average depth of tries. *Computing*, 28:367–71, 1982.

[Dev84] L. Devroye. A probabilistic analysis of the height of tries and of the complexity of triesort. *Acta Informatica*, 21:229–37, 1984.

[Dev87] L. Devroye. Branching processes in the analysis of the heights of trees. *Acta Informatica*, 24:277–98, 1987.

[DTK91] A.C. Donwton, R.W.S. Tregidgo, and E. Kabir. Recognition and verification of handwritten and hand printed british postal addresses. *International Journal of Pattern Recognition and Artificial Intelligence*, 5(1-2):265–91, 1991.

[Dun81] M.R. Dunlavey. On spelling correction and beyond. *Communications of the ACM*, 24(9):608, 1981.

[Dun91] J.A. Dundas. Implementing dynamic minimal-prefix tries. *Software Practice and Experience*, 21(20):1027–40, 1991.

[Dye82] C.R. Dyer. The space efficiency of quadtrees. *Computer Graphics and Image Processing*, 19(4):335–48, 1982.

[Fal85] C. Faloutsos. Access methods for text. *ACM Computing Surveys*, 17(1):49–74, 1985.

[FB74] R.A. Finkel and J.L. Bentley. Quad trees – a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[FC87] C. Faloutsos and S. Christodoulakis. Description and performance analysis of signature file methods for office filing. *ACM Transactions on Office Information Systems*, 5(3):237–57, 1987.

[FG89] E.R. Fiala and D.H. Greene. Data compression with finite windows. *Communications of the ACM*, 32(4):490–505, 1989.

[FHCD92] E.A. Fox, L.S. Heath, Q.F. Chen, and A.M. Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35(1):105–121, 1992.

[FK84] M.L. Fredman and J. Komlos. Storing a sparse table with $\mathcal{O}(1)$ worst access time. *Journal of the ACM*, 31(3):538–44, 1984.

[Fre60] E.H. Fredkin. Trie memory. *Communications of the ACM*, 3:490–500, 1960.

[Gar82] I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–10, December 1982.

[GBY91] G.H. Gonnet and R.A. Baeza-Yates. *Handbook of Algorithms and Data Structures: in Pascal and C (2nd ed.)*. Addison-Wesley, Reading, MA, 1991.

[GBYS92] G.H. Gonnet, R.A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In *Information Retrieval: Data Structures and Algorithms*, pages 66–82. Prentice-Hall, 1992.

[Gon83] G.H. Gonnet. Unstructured data bases or very efficient text searching. In *ACM PODS*, pages 117–24, Atlanta, GA, March 1983.

[Gon88] G.H. Gonnet. Efficient searching of text and pictures. Technical Report OED-88-02, Centre for the New OED., University of Waterloo, 1988.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the SIGMOD Conference*, pages 45–57, Boston, June 1984.

[Har72] L.D. Harmon. Automatic recognition of print and script. In *Proceedings of IEEE 60*, pages 1165–76, October 1972.

[HCE91] J.J. Hardwicke, J.H. Connolly, and J. Edwards. Parallel access to an English dictionary. *Microprocessors and Microsystems*, 15(6):291–8, 1991.

[HD80] P.A.V. Hall and G.R. Dowling. Approximate string matching. *Computing Surveys*, 12(4):381–402, 1980.

[Hin85] K. Hinrichs. Implementation of the grid file: Design concepts and experience. *BIT*, 25:569–92, 1985.

[Hor80] R.N. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10:501–6, 1980.

[HTW92] D.M. Hawken, P. Townsend, and M.F. Webster. The use of dynamic data structures in finite element applications. *International Journal for Numerical Methods in Engineering*, 33:1795–811, 1992.

[Hun78] G.M. Hunter. *Efficient Computation and Data Structure for Graphics*. PhD Dissertation, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ, 1978.

[Jac91] P. Jacquet. Analysis of digital tries with Markovian dependency. *IEEE Transactions on Information Theory*, IT-37(5):1407–75, 1991.

[Jon89] L.P. Jones. PORTREP: A portable repeated string finder. *Software Practice and Experience*, 19(1):63–77, 1989.

[KMP77] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *Computer Journal*, 6(2):323–50, 1977.

[Knu68] D.E. Knuth. *Information Structures*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1968.

[Knu73] D.E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.

[KP89] P. Kirschenhofer and H. Prodinger. On the balance property of Patricia tries: External path length viewpoint. *Theoretical Computer Science*, 68:1–17, 1989.

[KS88] H.P. Kriegel and B. Seeger. PLOP-hashing: A grid file without directory. In *Proceedings of 4th International Conference on Data Engineering*, pages 369–76, Los Angeles, February 1988.

[KST92] J.Y. Kim and J. Shawe-Taylor. An approximate string-matching algorithm. *Theoretical Computer Science*, 92:107–17, 1992.

[Kuk92] K. Kukich. Techniques for automatically correcting words in text. *Computing Surveys*, 24(4):377–439, 1992.

[LEMR89] Y.H. Lee, M. Evens, J.A. Michael, and A.A. Rovick. Spelling correction for an intelligent tutoring system. In *Proceedings of Computing in the 90's. The First Great Lakes Computer Science Conference*, pages 77–83, Kalamazoo, MI, October 1989.

[Lev66] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Dokl.*, 6:126–36, 1966.

[Lit81] W. Litwin. Trie hashing. In *SIGMOD 81*, pages 19–29, April 1981.

[Lit85] W. Litwin. Trie hashing: Further properties and performances. In *Proceedings of the International Conference on Foundations of Data Organization and Algorithms*, 1985.

[LNLH91] W.A. Litwin, N.Roussopoulos, G. Levy, and W. Hong. Trie hashing with controlled load. *IEEE Transactions on Software Engineering*, 17(7):678–691, 1991.

[Mal76] K. Maly. Compressed tries. *Communications of the ACM*, 19(7):409–15, 1976.

[MD85] T.H. Merrett and B. Düchting. Relational storage and processing of two-dimensional diagrams. *Computers & Graphics*, 9(3):247–58, 1985.

[Mea82] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–47, 1982.

[Mer83] T.H. Merrett. *Relational Information Systems*. Reston Publishing Co., Reston, VA, 1983.

[MF85a] P. Mathys and P. Flajolet. Q-ary collision resolution algorithms in random access system with free and blocked channel access. *IEEE Transactions on Information Theory*, IT-31(2):217–43, 1985.

[MF85b] T.H. Merrett and B. Fayerman. Dynamic Patricia. In *Proceedings of the International Conference, FODO*, pages 13–20, Kyoto, Japan, May 1985.

[MO82] T.H. Merrett and E.J. Otoo. Dynamic multipaging: A storage structure for large shared data banks. In *Improving Database Usability and Responsiveness*, pages 237–54. Academic Press, New York, 1982.

[Mor68] D.R. Morrison. PATRICIA – Practical Algorithm To Retrieve Information Coded In Alphanumeric. *Journal of the ACM*, 14(4):514–34, 1968.

[MS91] T.H. Merrett and H. Shang. Unifying programming languages and databases: Scoping, metadata, and process communication. In *The*

Third International Workshop on Database Programming Languages: Bulk Types & Persistent Data, pages 139–48, Nafplion, Greece, August 1991.

[MS93a] T.H. Merrett and H. Shang. Trie methods for representing text. In *Proceedings of 4th International Conference, FODO'93*, LNCS 730, pages 130–45, Chicago, Ill, October 1993. Springer-Verlag.

[MS93b] T.H. Merrett and H. Shang. Trie methods for representing text. Technical Report SOCS-93.5, School of Computer Science, McGill University, 1993.

[MS94] T.H. Merrett and H. Shang. Zoom tries: A file structure to support spatial zooming. In *Sixth International Symposium on Spatial Data Handling*, pages 792–804, Edinburgh, 1994.

[MT77] F.E. Muth,Jr and A.L. Tharp. Correcting human error in alphanumeric terminal input. *Information Processing & Management*, 13:329–37, 1977.

[NB94] W.G. Nulty and J.J Bartholdi. Robust multidimensional searching with spacefilling curves. In *Sixth International Symposium on Spatial Data Handling*, pages 805–17, Edinburgh, 1994.

[NH85] J. Nievergelt and K. Hinrichs. Storage and access structures for geometric data base. In *Proceedings of 2th International Conference, FODO'85*, pages 441–55, Kyoto, Japan, May 1985. Plenum Press.

[NHS84] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984.

[OM84] J.A. Orenstein and T.H. Merrett. A class of data structures for associative searching. In *Proceedings of Third ACM SIGACT–SIGMOD Symposium on Principles of Database Systems*, pages 181–90, Waterloo, April 1984.

[Ooi90] B.C. Ooi. *Efficient Query Processing in Geographic Information Systems*. LNCS 471. Springer-Verlag, 1990.

[OR22] M.K. Odell and R.C. Russell. U.S. Patent Numbers, 1,261,167 (1918) and 1,435,663, 1922. U.S. Patent Office, Washington, D.C.

[Ore82] J.A. Orenstein. Multidimensional tries used for associative searching. *Information Processing Letters*, 14(4):150–6, 1982.

[Ore83] J.A. Orenstein. Blocking mechanism used by multidimensional tries. Unpublished letter, February 1983.

[Pit85] B. Pittel. Asymptotical growth of a class of random trees. *The Annals of Probability*, 13(12):414–27, 1985.

[PZ92] T.B. Pei and C. Zukowski. Putting routine tables in silicon. *IEEE Network*, 6(1):42–50, 1992.

[RBK89] 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.

[Reg88] M. Regnier. Trie hashing analysis. In *Proceedings of Fourth International Conference on Data Engineering*, pages 377–81, Los Angeles, CA, February 1988.

[Reg89] M. Regnier. New results on the size of tries. *IEEE Transactions on Information Theory*, IT-35(1):203–5, 1989.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, Mass., 1990.

[Sel80] P.H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms*, 1:359–73, 1980.

[Sev74] D.G. Severance. Identifier search mechanisms: A survey and generalized model. *ACM Computing Surveys*, 6(3):175–94, September 1974.

[Sha86] C.A. Shaffer. *Application of Alternative Quadtree Representations.* PhD Dissertation, Computer Science Department, University of Maryland, College Park, MD, June 1986. Technical Report TR-1672.

[SK83] D. Sankoff and J.B. Kruskal. *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison.* Addison-Wesley, Reading, Mass., 1983.

[SK88] B. Seeger and H.P. Kriegel. Techniques for design and implementation of efficient spatial access methods. In *Proceedings of 14th International Conference on VLDB*, pages 360–72, Los Angeles, August 1988.

[SSN90] C.A. Shaffer, H. Samet, and R.C. Nelson. QUILT: a geographic information system based on quadtrees. *International Journal of Geographical Information Systems*, 4(2):103–31, 1990.

[ST93] A. Salminen and F.W. Tompa. Pat expressions: an algebra for text search. *Acta Linguistica Hungarica*, 41:277–306, 1992-93.

[Sus63] E.H. Sussenguth,Jr. Use of tree structures for processing files. *Communications of the ACM*, 6(5):272–9, 1963.

[Szp88] W. Szpankowski. Some results on v-ary asymmetric tries. *Journal of Algorithms*, 9:224–44, 1988.

[Szp90] W. Szpankowski. Patricia tries revisited. *Communications of the ACM*, 37(4):691–711, 1990.

[Szp91] W. Szpankowski. A characterization of digital search trees from the successful search viewpoint. *Theoretical Computer Science*, 85:117–34, 1991.

[Szp92] W. Szpankowski. Probabilistic analysis of generalized suffix trees. In *Proceedings of 3rd Annual Symposium on Combinatorial Pattern Matching* [BYP92], pages 1–14.

[Tam82] M. Tamminen. The EXCELL method for efficient geometric access to data. In *Proceedings of ACM IEEE 19th Design Automation Conference*, pages 345–51, Las Vegas, Nevada, 1982.

[TB83] L. Torenvliet and P.V.E. Boas. The reconstruction and optimization of trie hashing functions. In *Proceedings of the 12th International Conference on VLDB*, pages 655–660, October 1983.

[TITK88] T. Tokunaga, M. Iwayama, H. Tanaka, and T. Kamiwaki. LangLAB: A natural language analysis system. In *Proceedings of the 12th International Conference on Computational Linguistics*, pages 655–60, Budapest, Hungary, August 1988.

[Tom92] F.W. Tompa. An overview of Waterloo's database software for the *OED*. Technical Report OED-92-01, Centre for the New OED., University of Waterloo, 1992.

[TY79] R.E. Tarjan and A.C.C. Yao. Storing a sparse table. *Communications of the ACM*, 21(11):606–11, 1979.

[Ukk85] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–7, 1985.

[WF74] R.A. Wanger and M.J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–78, 1974.

[YKH89] H. Yokota, H. Kitakami, and A. Hattori. Term indexing for retrieval by unification. In *Proceedings Fifth International Conference on Data Engineering*, pages 313–20, Los Angeles, CA, February 1989.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–43, 1977.