This is to certify that the
dissertation entitled

THE HYBRID DIGITAL TREE AND ITS APPLICATIONS TO
GENOMIC SEQUENCE DATABASES

presented by

QIANG XUE

has been accepted towards fulfillment
of the requirements for the

Doctoral degree in Computer Science

[illegible]

Major Professor's Signature

12-16-05

Date

THE HYBRID DIGITAL TREE AND ITS APPLICATIONS TO GENOMIC SEQUENCE DATABASES

By

Qiang Xue

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science and Engineering

2005

ABSTRACT

THE HYBRID DIGITAL TREE AND ITS APPLICATIONS TO GENOMIC SEQUENCE DATABASES

By

Qiang Xue

This dissertation focuses on index structures, search algorithms, and applications for large string databases whose indexes cannot fit entirely in the main memory (RAM). String searching is a classic research topic that has received increasing attention in recent years, due to the rapid growth of digital text collections (strings) and the fast expansion of application range and complexity. Traditional string indexing approaches are either RAM-based or disk-based. The RAM-based structures perform poorly when a database index size exceeds that of the available RAM. On the other hand, disk-based structures do not take full advantage of the available RAM, which may result in overwhelmed Input/Output (I/O) operations. In this dissertation, a novel indexing approach, the Hybrid Digital tree (HD-tree), is proposed. The HD-tree index contains two parts: the RAM-index and the disk-index. The RAM-index resides in the RAM to minimize the disk accesses; while the disk-index maintains the rest of the index on disks so that large databases can be indexed.

The first half of this dissertation focuses on index structures. The HD-tree is proposed after investigating existing indexing techniques. Construction and search algorithms for the HD-tree are developed, and characteristics of the tree structure are discussed. The HD-tree is applied to prefix and substring searches, and is compared with the Prefix B-tree. The comparison shows that the HD-tree not only reduces I/O operations by a factor of two to three, but also reduces the total query processing time by one order of magnitude. The HD-tree is also applied to approximate string matching based on the Hamming distance, where the performance of the HD-tree surpasses that of the M-tree and the linear-scan approach.

In the second half of this dissertation, the HD-tree is applied to indexing and searching genomic sequence databases, such as the entire GenBank protein sequence database. Since the GenBank data is massive, using the standard method to generate an HD-tree index takes dozens of hours. Therefore, the Sort-Merge method is proposed to reduce the construction time by an order of magnitude. Sequence search algorithms using scoring matrices are developed for the HD-tree. Compared with BLAST, a popular sequence search tool, the HD-tree not only reduces query time by a factor of four, but also finds more valid results for short queries. Finally, the HD-tree is applied to sequence searches using the Profile Hidden Markov Model (PHMM), where it shows great success. Compared with one of the most popular PHMM search tools, HMMER, the HD-tree is orders of magnitude faster for short queries.

In the appendix, the research of approximate q-gram matching in genomic sequence databases is presented. It is shown that searching genomic sequence databases using longer query word length and larger Hamming distance in the filtering stage provides an excellent opportunity for optimizing the search cost, while improving the quality of the search. This result provides further support and motivation for developing advanced indexing schemes, such as the HD-tree, for large genomic sequence databases.

In summary, this dissertation not only develops a new tree structure for string indexing, but also successfully applies the structure to real applications. According to comparisons with existing techniques, the proposed data structure, the HD-tree, is promising for indexing and searching large string databases, especially genomic sequence databases.

To God
and
my Parents: Kongwen Xue and Fangying Zhu

Acknowledgements

Completing this dissertation is the biggest challenge in my life. Without the help of numerous people, I cannot succeed.

My first thanks go to my adviser, Dr. Sakti Pramanik, who committed years in training and helping me to be a qualified researcher. He never gave up on me, although I was slow in learning. Many times, he invited me to his house during weekends to provide extra help. He offered valuable insight and suggestions in guiding my research direction. He read my paper and dissertation drafts and edited my grammar errors without a complaint. I am also thankful for Dr. Pramanik's family, who welcomed me warmly whenever I visited and treated me many nice meals.

My thanks also go to Dr. James Cole, a member of my PhD committee. As a biologist, Dr. Cole provided the knowledge and insight I needed to apply the HD-tree to genomic sequence databases. He also offered many suggestions in finding applications and designing experiments. Without his help, the second half of the dissertation is not possible. I am also thankful to Dr. Qiang Zhu and Dr. Gang Qian, who helped me in developing the structure and algorithms of HD-tree, and publishing the research.

I appreciate the help from other members of my PhD committee: Dr. Herman Hughes, Dr. William McCarthy, and Dr. Jon Sticklen. They encouraged me in the process of finishing this dissertation and offered valuable revision suggestions. I am especially thankful that Dr. Hughes, though retired, flew from Atlanta to attend my defense meeting.

I am indebted to my parents for encouraging me to pursue this degree, and supporting me in every step along the way. They sacrificed much more than most parents in helping me to be successful. I shared many tears and laughs over this dissertation with my brothers and sisters in Christ. I cannot imagine how could I continue the study without their prayers and encouragements. Among them, Caleb Kelly and

Steve Herwaldt played the most important role.

During years of study at MSU, I was financially supported by a number of resources such as the CSE department. Steve Tuckey from the writing center at MSU provided valuable help me on my writings and presentations. I have made many friends who offered enormous help and encouragements. For these, I have not stopped giving thanks.

I do not intend to list all the names I need to give thanks. However, in my heart, I am grateful for each of them.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

The study of data structures and algorithms has been a fundamental research in the field of computer science since the era of electronic computing. The increase in raw computing power cannot discount the significance of discovering efficient data structures and algorithms. After all, the faster the computing equipment, the more is the gain from human intelligence. This chapter presents the research needs in string indexing, the basic concepts of string matching, and research challenges in string databases. Existing techniques and their limitations are briefly discussed, so that the motivations for the proposed hybrid indexing approach is clear. The contributions and overview of this dissertation are presented at the end of this chapter.

1.1 Research Needs

Electronic text (string) collections have increased dramatically over the last decade, from megabytes of dictionaries, to gigabytes of genomic sequences, to terabytes of web documents. Such expansion has not stopped accelerating as information technologies continue to develop. Hence, string indexing techniques become more and more essential to process the massive available information. Many applications, such as computational biology [1, 2, 3], signal processing [4, 5], and information retrieval [6, 7, 8], must process complex queries (e.g., approximate string matching) on the growing amount of text collections. Due to the limited size of the internal memory (i.e., Random Access Memory, or RAM), partial or entire indexes have to be stored on the external memory (disk). The resulting input/output communication between fast RAM and slow disk creates a major performance bottleneck. In order to reduce search cost and improve query performance, new ideas must be developed to design efficient data structures and search algorithms for processing such large text collections. Motivated by this fact, the focus of this dissertation is on designing and analyzing index structures and search algorithms for large string databases, such as genomic sequence databases.

1.2 Basic Concepts for String Matching

A *string* consists of a series of symbols (or characters) chosen from an alphabet Λ of size $|\Lambda|$. The letters and strings are assumed to have a lexicographic order. In this dissertation, lower-case letters are used to denote symbols from Λ (e.g., a, b and c), while lower-case Greek letters are used to denote strings (e.g., α, β and γ). The combinations of strings or symbols indicate string concatenation. For example, $\alpha\beta$ is the concatenation of α and β, while αb is the concatenation of α and b. Given a string $\alpha = a_1...a_n$ of length $|\alpha| = n$, $a_1...a_i$ is called a *prefix* of α, $a_j...a_n$ a *suffix* of α, and $a_i...a_j$ a *substring* of α. For simplicity, a database is considered to be a set of records with the form $\Upsilon_i = (\kappa_i, \Delta_i)$, where κ_i is a unique string and Δ_i the descriptive information of κ_i, such as a statistic, a string position, or a pointer to another location where such information can be found. Since this dissertation focuses on the issues of string indexing, Δ_i is usually ignored in discussions (i.e., not strictly distinguishing a record and a string).

This dissertation divides string matching problems into two categories: non-approximate string matching and approximate string matching, based on their computational complexity. Non-approximate string matching includes exact, prefix, substring, and range searches, which are used in applications such as web search engines, relational databases, and E-businesses. Given a database containing strings $\kappa_1...\kappa_n$, an exact search, $ExactSearch(\alpha)$, retrieves κ_i such that $\kappa_i = \alpha$, $1 \leq i \leq n$; a prefix search, $PrefixSearch(\alpha)$, retrieves κ_i where α is a prefix of κ_i; a sub-string

Database	Queries
disk	ExactSearch(hybrid) = {hybrid}
hybrid	
index	PrefixSearch(in) = {index, insert, intersting}
insert	
interesting	
prefix	SubstringSearch(in) = {index, insert, intersting, string, substring}
string	
structure	
substring	RangSearch(in, out) = {index, insert, interesting, prefix}
suffix	

Figure 1.1: Examples of Non-Approximate String Matching

search, $SubstringSearch(\alpha)$, retrieves κ_i where α is a substring of κ_i; and a range search, $RangeSearch(\alpha, \beta)$, retrieves κ_i such that $\alpha \leq \kappa_i \leq \beta$. Examples of non-approximate string matching are shown in Figure 1.1.

Approximate string matching is the string matching with "errors," and the earliest references are from the late 1960s (signal processing) and the 1970s (computational biology). The general form of approximate string matching is to find the positions of a text where a given pattern occurs, allowing a limited number of "errors" in the matches [9]. Various error models have been developed to define how different two strings are, by measuring a "distance" between the two strings. A user-defined distance is used to identify one string as the erroneous variant of another. The distance measures are further discussed in Section 4.5.1.

1.3 Growth of String Databases

String databases and string matching techniques are used in many applications. The following sections introduce the rapid growth in a few applications, such as signal processing, relational databases, E-businesses, and computational biology.

1.3.1 Signal Processing

Signal processing is used in speech recognition to determine text messages transmitted in audio signals. Approximate matching is critical in this application, because parts of a speech may be lost or mispronounced. At the same time, since the physical transmission of signals is error-prone, error correction is important to restore the correct message from a possible error introduced during the transmission. One of the most popular distance measurements, known as the *Levenshtein distance* [4] (also called *edit distance*) was originally developed for the purpose of error correction. The fast growing of multi-media databases demands the ability of searching the content of audio data, and the increasing interest in wireless networks keeps looking for stronger error correcting methods. Hence, approximate string matching in signal processing is a very active research area.

1.3.2 Relational Databases

The rapid growth of the Internet, the increase in online transaction processing, and the expansion of large database applications have contributed significantly to the data explosion of relational databases, such as consumer relation management, national white pages, and digital libraries. In an recent survey (September 14, 2005) of the world's largest and most heavily used databases by Winter Corporation, the size of the largest commercial database tops the 100 TB mark, and has increased three-fold since the 2003 survey [10]. The largest Windows data warehouse is 19.5 TB and the largest number of rows/records is 2.8 trillion. For the first time, the peak workload on a system exceeds 1 billion SQL statements per hour.

In relational databases, a large portion of queries deal with strings (e.g., searching names or addresses). In a majority of cases, users do not expect an exact match of the query string, therefore, prefix search, substring search, and range search are important. How to index and search these massive string data efficiently and effectively is a challenging research issue.

1.3.3 E-businesses

The exponential growth of the Internet continues to encourage many traditional businesses to enter the electronic business (E-business) realm. Consequently, managing electronic documents is essential for the success of E-business. For example, an important application in E-business is to provide electronic product catalogs (E-catalogs) for buyers to locate and select products [11]. Consider a large E-store, such as Amazon.com, the E-catalog may contain millions of products and may receive hundreds of queries per second. In order to provide prompt query response, the E-catalog data have to be indexed properly. In recent years, XML (eXtensive Markup Language), which is based on plain text, has become a standard for representing and exchanging documents on the Internet. Almost all recent E-business standards are based on XML. Since the amount of XML data is large and increasing, efficient storage and query of XML documents is a growing challenge in computer science research [12].

1.3.4 Computational Biology

Computational biology has experienced tremendous growth over the past decade. It is known that genetic information is encoded in DNA and protein sequences. DNA sequences are represented as strings using a four-letter alphabet {A,C,G,T}. Protein sequences are represented as strings using a twenty-letter alphabet, where each letter represents an amino acid. Searching over these strings is a fundamental operation for problems such as assembling DNA chains from the pieces obtained by experiments, looking for given features in DNA chains, or determining how different two genomic sequences are. In such applications, exact match is of little use due to mutations of genomic sequences. Therefore,

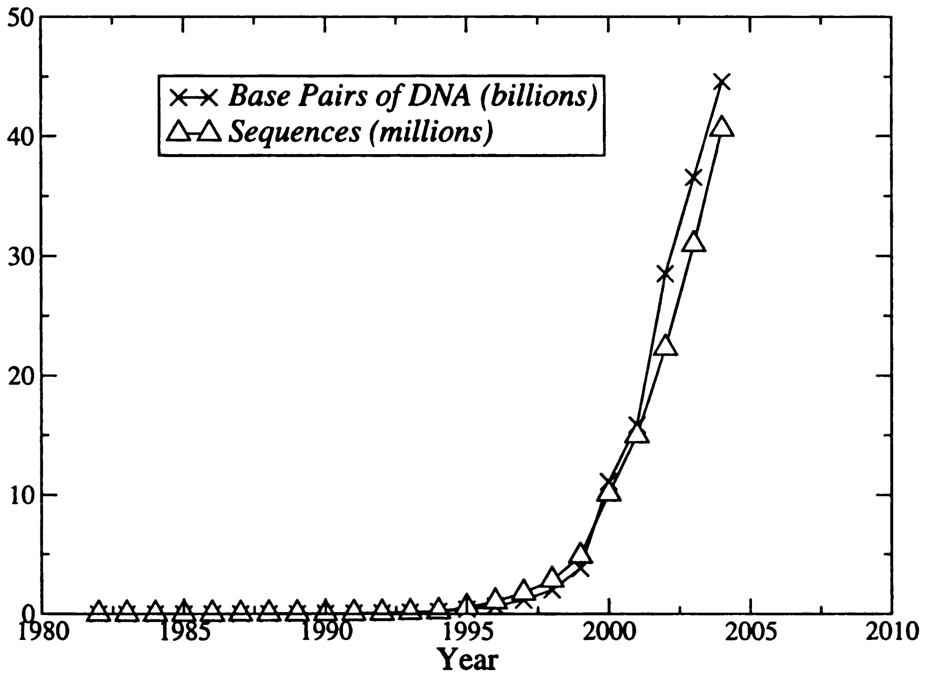


Figure 1.2: Growth of GenBank (1982-2004)

approximate string matching becomes critical in computational biology.

There are several public sequence databases, such as GenBank [13], Swiss-PORT [14], EMBL Nucleotide Sequence Database [15], and DNA Data Bank of Japan [16]. These databases have seen exponential growth in recent years, partly due to the well-known human genome project [17]. Figure 1.2 shows the growth chart of GenBank. As of April 2004, GenBank contains approximately 44 billion base pairs, and 40 million sequences. The rapid expansion of genomic sequence databases and the complexity of sequence matching tasks demand the development of efficient and effective data structures and search algorithms.

1.4 Memory Hierarchy

In order to be cost-effective, computer systems usually contain a hierarchy of memory levels, where each memory level has different cost and performance characteristics. The lowest level consists of CPU registers and caches that are built with the fastest but most expensive memory. Above this lowest level is the internal memory, which is also called random-access memory (RAM). At a higher level, inexpensive but slower magnetic disks are used for external mass storage. Finally, even slower but larger-capacity devices such as tapes and optical disks are used for archival storage. Figure 1.3 illustrates a typical memory hierarchy and its characteristics.

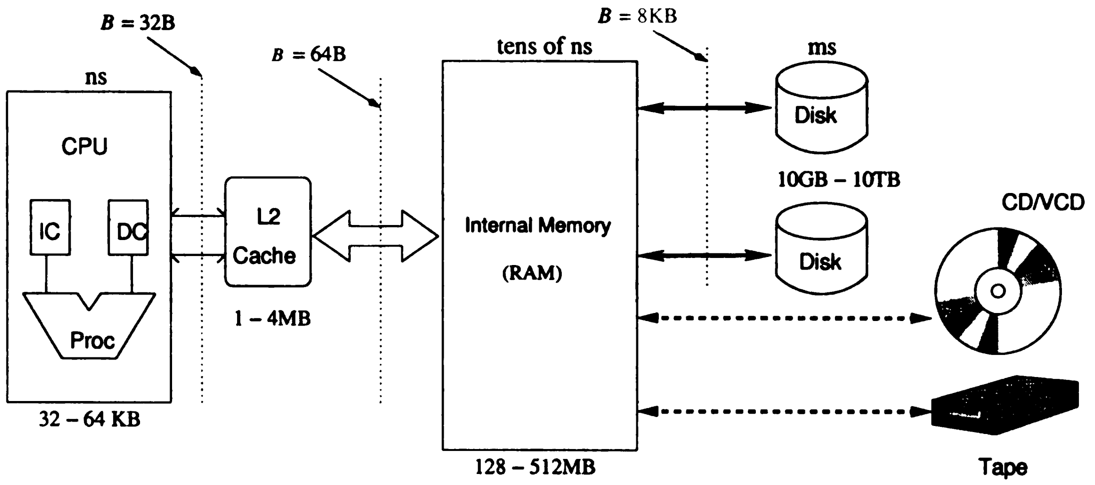


Figure 1.3: The memory hierarchy of a typical uniprocessor system. Below each memory level is the range of typical sizes of that memory level. The value of B at the top of the figure indicates the block transfer size between two adjacent levels of the hierarchy. Sizes are given in units of bytes (B), kilobytes (KB), megabytes (MB), gigabytes (GB), or terabytes (TB).

In modern programming languages, the notion of Virtual Memory allows the program address-space to be far larger than what can fit in the RAM. Programmers usually assume that all memory references require the same access time. In many cases, such an assumption does no harm, especially when the data sets are small. However, when large address spaces span multiple levels of the memory hierarchy, the assumption of equal access time may not reflect the actual behavior of the program. This is because accessing data in the lowest level of memory is orders of magnitude faster than accessing data in higher levels. For example, accessing a CPU register takes nanoseconds (10^{-9} seconds), and accessing RAM takes tens of nanoseconds, but the latency of accessing data from a disk is several milliseconds (10^{-3} seconds), which is about one million times slower than that of a CPU register. Since the latency and bandwidth of memory chips are improving more quickly than those of disks (roughly 60% increase per year in processor performance, and 20% increase per year in disk performance [18]), the access gap is continually growing. Therefore, in applications that process massive amounts of data, the input/output communication (I/O) between levels of memory, especially between the RAM and disks, often becomes a bottleneck. As these trends continue, the I/O bottleneck grows ever worse without some changes in the fundamentals of data storage.

1.5 Existing Indexing Structures and Their Limitations

Indexing structure is a topic studied throughout the development of computer science. The well-known B-tree structure [19], which is the basis of most disk-based indexing structures, was proposed in 1972. The digital tree, which is widely used in string matching, can be traced back to the 1960s [20]. Modern databases contain various types of data, such as pictures, audios, videos, and spatial data; yet textual data (i.e., strings) are still the major components of database systems. How to efficiently index these strings was, is, and will remain a critical issue for database performance.

Over the past few decades, many data structures have been proposed for string indexing. These data structures can be divided into two categories: disk-based structures and RAM-based structures. The first category includes inverted files [7], Prefix B-trees [21] and String B-trees [22, 23, 24]. The second category includes various structures based on digital trees (also known as *tries*), such as Patricia tries [20], suffix trees [25, 26], suffix arrays [27] and PAT trees [28].

Since string indexes for large applications are often too massive to fit entirely in the RAM, disk space must be used to store the indexes. However, the latency of accessing data from a disk is much slower than that from the RAM. Therefore, the resulting I/O communication can be the major performance issue. The two important measures that are normally used to evaluate the performance of disk-based data structures are the number of I/Os to answer a query and the storage utilization of disk blocks.

Among disk-based data structures, inverted files are popular for keyword-based searches. However, it is difficult to perform substring and similarity searches using inverted files. B-trees [19] and their variations [29], are well known balanced multi-way search trees for manipulating dynamic data on the disk. They are very efficient in handling fixed length keys (e.g., integers). However, the performance of B-trees degrades dramatically for variable length keys (e.g., strings), since the fan-out (i.e., the number of children) of an internal node depends on the number of strings stored in the node. The Prefix B-tree is designed to improve the performance of string indexing by using the shortest unique prefixes as separators within an internal node. The String B-tree uses the Patricia trie inside its internal nodes to provide the same worst-case performance as the B-tree. However, since the String B-tree stores indexed strings in a separate file, it generally requires more disk accesses than the Prefix B-tree. These disk-based indexing techniques do not require RAM. To use the large amount of available RAM, they rely on caching mechanisms that are usually not optimized for individual data structure. Therefore, there is a need for disk-based data structures to efficiently use the available RAM.

The RAM-based structures are useful for indexing strings in the RAM where string queries are performed. Patricia tries and PAT/suffix trees are particularly effective in handling relatively small amount of text. However, as the database size increases, it is no longer feasible to keep the entire trie in the RAM. Various methods have been proposed to reduce the sizes of tries, such as efficient implementations of trie nodes [30] and encoded representations of tries. For example, PaTries [31] and PAT-trees [28, 32] are variants of Patricia tries, in which Jacobson's encoding is used to reduce space requirement. The compressed trie structures trade space with computational complexity. However, for very large databases, it is still not practical to fit the corresponding indexes in the RAM. Another approach is to page tries on disks. Because of the unbalanced topology of a trie, it is shown to be difficult and inefficient to page a trie on disk. For example, in the worst case, a downward path of k nodes will be stored in $\Omega(k)$ different pages [24]. Paging tries is especially expensive for dynamic indexing, where inserting or deleting an m-length string may take $\Theta(m)$ page splits or merges [24]. Updating operations will also cause the storage utilization to degrade quickly. In [33], it is reported that the storage utilization is 43% for 238KB dynamic text and 38% for 5.55MB dynamic text. Therefore, it is concluded that RAM-based index structures are not suitable for indexing large string databases whose indexes cannot fit entirely in the RAM.

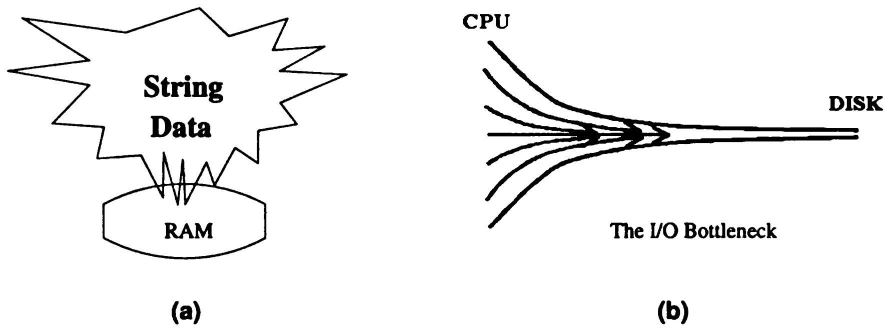


Figure 1.4: Limitations of Disk-based and RAM-based Index Structures

In summary, disk-based structures can index large databases but usually do not fully utilize the available RAM and may result in I/O bottleneck (see Figure 1.4b). On the other hand, RAM-based structures are efficient for string matching problems. However, as database size increases, indexes may become too large to fit in the RAM (see Figure 1.4a), and RAM-based structures perform poorly when database index size exceeds that of the available RAM.

1.6 The Hybrid Approach

According to the above discussion, both RAM-based and disk-based data structures have their strengths and limitations. Since our goal is to index large string databases with the aim of performing complex queries efficiently, the existing data structures cannot provide satisfactory performance. Therefore, a novel approach, the Hybrid Digital tree (HD-tree), is proposed. The basic idea of the HD-tree is to keep the internal nodes (similar to those in a digital tree) in the RAM to minimize the number of I/Os, while maintaining the leaf nodes (which hold the database strings) on disks to maximize the capability of the tree for indexing a large database. Strings stored in a leaf node share the same prefix. The internal nodes are built on these prefixes and are used to guide the search to the leaf node(s) containing the query answer(s). Unlike a traditional digital tree, the parent of a leaf node in the HD-tree allows a set ("range") of multiple prefixes so that indexed strings with different prefixes may share the same leaf node (i.e., disk block) to improve storage utilization. Moreover, unlike the traditional concept of range, the above prefix "range" of a node may not be "continuous", so that the storage utilization can be further improved.

It is known that traditional disk-based trees, such as Prefix B-trees, may use the available RAM to cache their internal nodes, so that the number of disk I/Os may be reduced. However, the HD-tree is different from this approach as follows: First, an internal node of disk-based trees is a disk block, which is usually several kilobytes in size. However, an internal node of the HD-tree is a data structure (i.e., a trie node), which is usually several bytes in size. Second, the internal nodes of disk-based trees are stored on disks and have to be read into the RAM whenever necessary. However, all internal nodes of the HD-tree are kept in the RAM, so that no disk I/Os are required to access these internal nodes.

The HD-tree is shown to be efficient for both non-approximate and approximate string matching. Using the HD-tree, the number of I/Os is optimal for prefix and substring searches. Although hashing techniques can also achieve optimality for exact searches, it cannot be used effectively in prefix, substring, and approximate string searches. It is observed that for a given database size, a small amount of RAM improves the performance of the HD-tree significantly. Since the data structure of internal nodes in an HD-tree is similar to that of tries, the HD-tree not only has a data compression property that is not supported by other disk-based structures such as the Prefix B-tree, but also achieves great success in approximate string matching. The HD-tree supports various approximate string matching based on the Hamming distance [34, 9], simple edit distance [4], general edit distance using a scoring matrix [35], and the profile hidden Markov model [36]. The HD-tree has shown to be effective in indexing and searching large genomic sequence databases such as the entire GenBank protein sequence database.

1.7 Contributions

This dissertation not only studies two important areas of string databases: indexing structures and approximate string matching, but also deals with real world applications in genomic sequence databases. A novel index structure, the Hybrid Digital tree (HD-tree), is proposed. The HD-tree is a RAM/disk-based tree that incorporates and extends indexing strategies of the digital trees and B-trees, taking advantages of their strengths in search performance and index capability. The HD-tree is compared with the Prefix B-tree using real textual data from Text REtrieval Conference (TREC) collections [37]. Queries are generated with different cluster levels to study the effectiveness of the HD-tree. It is shown that given random distinctive queries, the number of disk I/Os is reduced by more than 60%, while the query time is reduced by one order of magnitude. The HD-tree is also applied to approximate string matching based on the Hamming distance, where the HD-tree outperforms existing techniques such as the M-tree [38] and the linear scan approach [39, 40].

The HD-tree is applied to genomic sequence databases. Due to the non-structured feature of genomic sequence data, and the largeness of the database, the standard approach of building HD-tree takes many hours. Heuristics are developed to reduce tree construction time in an order of magnitude. Hence, the HD-tree index can be created for the entire GenBank protein database in reasonable time (e.g., 3-4 hours). Algorithms are developed for genomic sequence search using scoring matrices [35, 41]. The HD-tree is compared with the well-established sequence search algorithm, BLAST [3, 42]. For short protein sequence queries (e.g., insulin), the HD-tree is not only four times faster than BLAST, but also able to find more valid query results. The speed improvement of the HD-tree is even more impressive in sequence search using the Profile Hidden Markov Models (PHMMs), where heuristic algorithms are not applicable. Experiments are conducted on both synthetic and real queries. The HD-tree is shown to be orders of magnitude faster than HMMER, a popular PHMM search tool, for short queries.

Besides index structures and search algorithms, in the appendix, the research of approximate q-gram matching in genomic sequence databases is presented. It is shown that searching genomic sequence databases using longer word length and larger Hamming distance in the filtering stage provides an excellent opportunity for optimizing the search cost while improving the quality of the search. This result is another proper justification of developing advanced indexing schemes, such as the HD-tree, for large genomic sequence databases.

1.8 Overview of the Dissertation

The first half of the dissertation focuses on the index structures for large string databases. In Chapter 2, some existing techniques for string indexing are covered. In Chapter 3, the structure of the Hybrid Digital tree (HD-tree) and the algorithms to build the HD-tree are presented. In Chapter 4, the behavior of the HD-tree is discussed and algorithms for prefix searches and approximate string matching based on the Hamming distance are presented. The comparisons between the HD-tree and other index techniques, such as the Prefix B-tree for prefix search and the M-tree for approximate string matching based on the Hamming distance, are also discussed in Chapter 4.

The second half of the dissertation focuses on applying the HD-tree to genomic sequence databases. In Chapter 5, the background on genomic sequence analysis is covered. In Chapter 6, the techniques to index and search genomic sequence databases using the HD-tree is presented, and the performance of the HD-tree is compared with that of BLAST. In Chapter 7, sequence searches based on PHMMs are introduced, algorithms to search PHMMs using the HD-tree are presented, and the performance of the HD-tree is compared with HMMER, a popular PHMM search tool.

Finally, conclusions and future work are discussed in Chapter 8. In appendix, the research on approximate q-gram matching in genomic sequence databases is presented.

PART ONE

THE HYBRID DIGITAL TREE

Chapter 2: Existing Indexing Techniques

String indexing is an increasingly important task in computer science. In the past a few decades, many data structures have been proposed. All data structures can be characterized in two main ways: based on how a search is performed (hashing, complete key, or digital decomposition) and based on where they are used (in the RAM or on a disk).

Based on how a search is performed, there are three basic categories: hashing, search trees, and digital trees. Hashing maps a key to an integer in a given range (e.g., extendible hashing [43]). It "randomizes" the key order, and is able to perform an exact search very fast. In search trees, the complete value of a key is used to direct the search (e.g., B-trees [19] and Prefix B-trees [21]). In a digital tree (known as a trie, pronounced "try" [44], or radix search tree), keys are decomposed as a sequence of digits or alphabetic characters to direct the search (e.g., Patricia tries [20], suffix trees [26], PAT trees [28], and suffix arrays [27]). Based on where the structure is used, the above data structures can be divided into two categories: RAM-based and disk-based. The first category (RAM-based) includes various structures based on digital trees, such as Patricia tries, suffix trees and suffix arrays. The second category includes the extendible hashing [43], the Prefix B-tree [21], and the String B-tree[24]. In following sections, these data structures will be introduced as the background for the rest of this dissertation.

2.1 Extendible Hashing

The common element of all hashing algorithms is a predefined hash function

$$hash(N \text{ possible keys}) \rightarrow (0, 1, ..., M-1) \tag{2.1}$$

that maps N keys (e.g., strings) to M hash addresses (i.e., integers) in a uniform manner. Since it is possible that two keys may map into one address, how to resolve the collision makes hashing algorithms differ from each other. Most traditional hashing methods have a statically allocated table and are designed to handle only a fixed range of N. When N becomes large, the static hash table becomes infeasible, due to the increased space requirement. Therefore, dynamic hashing structures (e.g., extendible hashing) are required to handle widely varying values of N.

As shown in Figure 2.1, extendible hashing contains a directory and a set of disk blocks storing the keys. Assume M is sufficiently large and the directory consists of a table (i.e., array) of 2^d pointers, where d is a non-negative integer and each pointer points to a disk block. Keys are assigned to a table location corresponding to the d least significant bits of its hash value. The value of d is called *global depth.* It is set to the smallest value for which each block has at most B keys assigned to it. A lookup operation takes at most two I/Os: one to access the directory, and the other to access the block containing the item. If the directory fits in the RAM, only one I/O is needed.

To minimize storage utilization, several table locations may share one disk block if the total number of keys assigned to these table locations are less than B. These table locations have the same k least significant bits in their corresponding hash value. The value of k is called the *local depth.* It is chosen to be as small as possible so that the keys assigned to these table locations fit into a single disk block. Therefore, each disk block has its own local depth. Note that local depth is less then or equal to the global depth.

If a disk block overflows after a new key is inserted, the block needs to be split and the keys within the block need to be redistributed. If the local depth of the overflowing block is less than the global depth, only the block's local depth k and the corresponding pointers in the directory need to be modified. If the local depth of the overflowing block is equal to the global depth, the global depth is increased by one, and the directory doubles in size. The is how extendible hashing adapts to a growing N. The pointers in the new directory are initialized to point to the appropriate disk blocks. These disk blocks themselves do not need to be changed because of directory doubling, except for the block that overflows. An example of extendible hashing is shown in Figure 2.1.

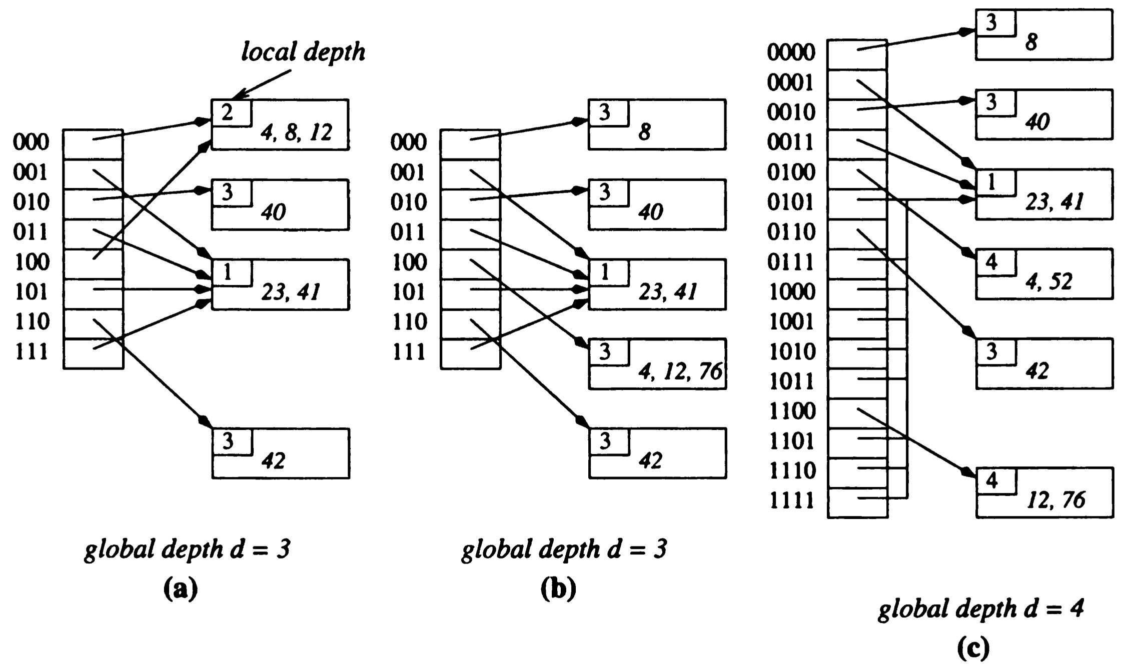


Figure 2.1: Extendible hashing with block size $B = 3$. The keys are the numbers inside a block. The hash address of a key consists of its binary representation. For example, the hash address of key 12 is '...001100'. (a) After insertion of the keys 4, 8, 12, 23, 40, 41, 42. (b) Insertion of 76 into directory location 100 causes the block with local depth 2 to split into two blocks with local depth 3. (c) Insertion of 52 causes the block with to split into two blocks with local depth 4. The directory doubles in size and the global depth *d* is increased to 4.

In extendible hashing, at least $\Omega(n/B)$ (Ω gives lower bounds) blocks are needed to store the directory. On average, the directory uses $\Theta(N^{1+1/B}/B^2)$ (Θ gives exact order) blocks [45]. For practical values of N and B, the $N^{1/B}$ is a small constant, typically less than 2. The expected number of disk blocks required to store the keys is asymptotically $n/ln2 \approx n/0.69$ [46]. Besides the extendible hashing, other dynamic hashing schemes include linear hashing [47] and spiral hashing [48]. More detailed surveys and analysis for dynamic hashing can be found in [49, 50].

Hashing works well for exact searches in the average case. However, it does not support sequential searches such as retrieving keys in a specified range, and other advanced searches such as a substring search. A more effective approach for sequential searches is to use search trees, which are explored next.

2.2 B-trees and Prefix B-trees

B-trees [19] and B-tree variations [29] are well known balanced multi-way search trees used for manipulating dynamic data on disks. Each node of a B-tree is a disk block that can store $\Theta(B)$ pointers and keys, where B is the disk block size. A B-tree of order m ($m = \Theta(B)$) satisfies the following properties:

1) Each node has at most m children.

2) Each internal node (i.e., non-leaf node), except for the root, has at least m/2 children.

3) The root has at least 2 children, unless it is a leaf.

4) All leaves appear on the same level.

5) An internal node with k children contains $k - 1$ keys.

A node that contains r keys and $r + 1$ pointers can be represented as that in Figure 2.2, where $k_1 < k_2 < ... < k_r$ and P_i points to the sub-tree for keys between k_i and k_{i+1}. Searching in a B-tree starts from the root and requires fetching at most one node at each level into the RAM. For example, after the node in Figure

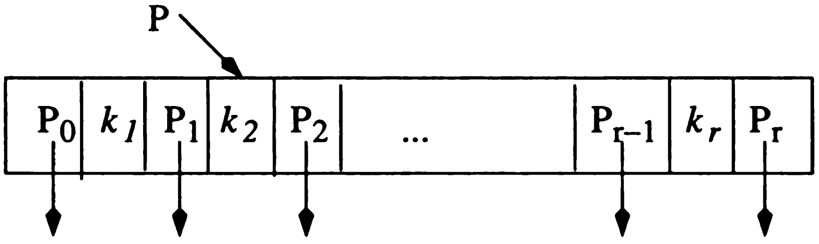


Figure 2.2: A B-tree Node

2.2 has been read into the RAM, the given query key, k, is searched among k_1, k_2, ..., k_r. If the search is successful, the desired key is found; otherwise, assuming $k_i < k < k_{i+1}$, the node indicated by P_i is fetched and the search process is repeated. The pointer P_0 is used if k is less than k_1, and P_r is used if k is greater than k_r. The search is unsuccessful if the pointer is null.

If a new key is inserted into a B-tree of order m, where all leaves are at level l, the new key is inserted into the appropriate node on level $l-1$. If the node overflows (i.e., contains m keys), it splits into two nodes (see Figure 2.3) and inserts the key, $k_{\lceil m/2 \rceil}$, into the parent of the original node. If the splitting causes the parent node to overflow, the parent node splits, and so on. A splitting can thus propagate up to the root, and the tree grows in height only when the root splits. Deletions are handled in a similar way by merging nodes.

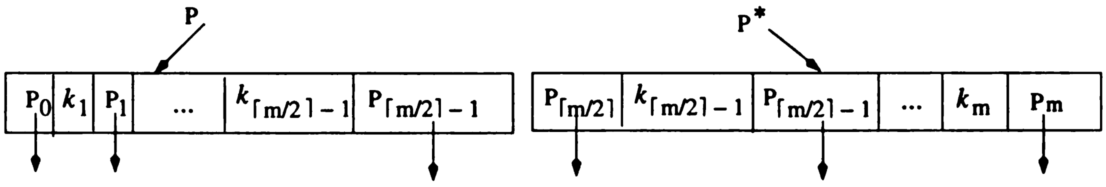

P
P*
P_0
k_1
P_1
...
$k_{\lceil m/2 \rceil - 1}$
$P_{\lceil m/2 \rceil - 1}$
$P_{\lceil m/2 \rceil}$
$k_{\lceil m/2 \rceil - 1}$
$P_{\lceil m/2 \rceil - 1}$
...
k_m
P_m

Figure 2.3: B-tree Splitting

The complexity of B-trees has been studied thoroughly, and their behavioral boundaries have been determined. Given a B-tree of order m, assume that there are N keys, the height, h, of the B-tree is bounded by:

$$h \leq 1 + \log_{\lceil m/2 \rceil}(\frac{N+1}{2}). \tag{2.2}$$

When a new key is being inserted, the average number of nodes, s, that need to be split is bounded by:

$$s \leq 1 + \frac{N-1}{\lceil m/2 \rceil - 1}. \tag{2.3}$$

Therefore, the average number of splits while building a tree of N keys is less than $1/(\lceil m/2 \rceil) - 1)$ per insertion [51].

Since the birth of the B-tree, many approaches have been developed to improve the basic B-tree structure. For example, in a B^+-tree [51, 29], all keys are stored in leaves. The uppper levels, which are organized as a B-tree, consist only of an index as a roadmap to enable rapid location of a key. The leaves of a B^+-tree are linked together in order to facilitate range queries and sequential access. In B*-trees [29], splitting is postponed by "sharing" the overflowing node's data with one of its adjacent siblings. The overflowing node needs to be split only if the adjacent sibling is also full. When this happens, a new node is created. Data from the overflowing node and its full sibling are evenly redistributed among the three nodes, making each of them approximately 2/3 full. This method reduces the number of times new nodes must be created and thus increases the storage utilization. Assume random insertions. In regular B-trees, it is shown that the average storage utilization of nodes is $\ln 2 \approx 69\%$ [46, 52], while in B*-trees, the average storage utilization increases to about $2 \ln(3/2) \approx 81\%$ [53].

Although the B-tree was initially designed for fixed-length keys, such as integers, the basic idea can be used for variable-length keys, such as strings. However, if the keys are variable-length strings, the number of keys stored in each tree node may not reach the upper limit, m, before the node becomes full. Consequently, the height of the tree is not bounded, as in Equation 2.2. As the key length increases, the number of keys per node decreases; therefore, the height of the tree increases. Thus, the performance of the B-tree degrades, since more internal nodes must be accessed.

In order to increase the number of keys stored in internal nodes, the Prefix B-tree is proposed in [21]. In conventional B-trees, a separator has to be a key (e.g., $k_{\lceil m/2 \rceil}$ in Figure 2.3) . Assume a group of keys is: { "abstract", "common", "define", "longlonglongword", "moment", "people" }. In order to split the group, the key "longlonglongword" has to be used as the separator. However, Prefix B-tree removes such limitation, and uses the shortest unique prefix of a key as a separator. For example, any shortest string between "define" and "longlonglongword" in lexicographic order (e.g., 'e' or 'f') can be a separator (see Figure 2.4). This method increases the number of keys stored in internal nodes. However, it can fail when keys have a long common prefix, since they are adjacent to each other in lexicographic order. In [54], head compression is used to factor out a common prefix from all keys in an internal node. In [55], another compression scheme is adopted. The idea is: if a key begins with the same n characters as its immediate predecessor, the key is stored with its first n characters replaced by integer n. This approach saves space but it does not prevent a key from having many characters in the rest of its positions. Besides the storage issue, heuristics to improve searching performance within a Prefix B-tree node can be found in [55].

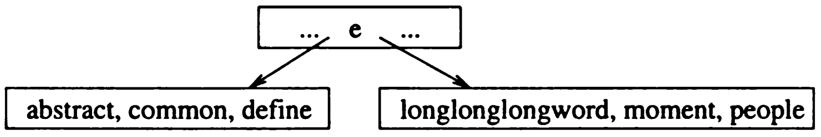


Figure 2.4: An Example of the Shortest String Separator in a Prefix B-tree

2.3 Tries

Tries (pronounced "try", and derived from "information re*trie*val" [44]), or digital trees, are recursive tree structures that use digital decomposition (i.e., decompose strings as a sequence of digits or characters) to represent a set of strings and to direct the search. The basic idea is similar to the thumb index on a large dictionary, where from the first letter of a given word, the pages containing all words beginning with that letter can be located immediately.

A trie can be defined as the following: assume S is a set of strings, and $\Lambda = \{a_i\}_{i=1}^{r}$ is the alphabet (a special symbol \$ is included to represent the end of a string), then the trie associated to S is defined recursively by the rule:

$$trie(S) = (trie(S|a_1), ..., trie(S|a_r)), \tag{2.4}$$

where $S|a_i$ represents the subset of S consisting of strings that start with a_i and stripped of their initial letter a_i. The recursion is halted as soon as S contains one string. A trie only maintains the minimal prefix set of strings necessary to distinguish all the strings of S.

A basic trie can be represented as an M-way tree, where each node is a vector of M components corresponding to symbols in Λ. Each edge is labeled with a symbol that leads to the next node. Each node on level l represents the set of all keys that begin with a certain sequence of l characters. The sequence is the concatenation of the labels traversing from root to a leaf node. An example of a trie structure is shown in Figure 2.5.

Since tries represent strings along the paths in the tree, not in the nodes, considerable compression can be achieved by sharing paths. The height of a trie is the number of nodes in the longest path from the root to an external node. On average, the height of a trie is logarithmic for any square-integrable probability distribution [56]. Tries provide potentially faster access than search trees, since one comparison may lead to a large fan-out (up to $|\Lambda|$) [43].

Compacted tries compresses the unary paths (i.e., each node on the path has only one child) of the tries by storing the labels along the paths into trie nodes [51].

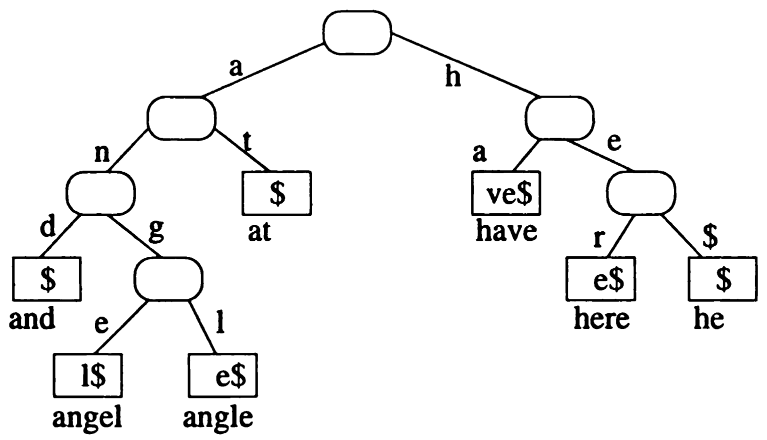


Figure 2.5: An Example of a Trie

Patricia tries ("Practical Algorithm To Retrieve Information Coded In Alphanumeric") are binary tries where the individual bits of the keys are used to decide on the branching [20]. Each internal node of the Patricia trie has an indication of which bit is to be used for branching. This may be given by an absolute bit position or by a count of the number of bits to skip.

2.4 Suffix Trees and Suffix Arrays

A suffix (semi-infinite string) is a substring, which starts from a position in a text and continuing toward the end as far as necessary to make the substring unique within the text. A common method to accelerate string searching is to index all *suffixes* of a text using a trie. The resulting trie is known as a *suffix tree* [25]. For each suffix, a logic pointer pointing to the starting position of the suffix is stored at the leaf node of the suffix tree. The starting position can be either at a character or at a word. To make sure that no suffix in the text is a prefix of another suffix, a unique character that is not in Λ is appended at the end of the text. An example of a suffix tree is shown in Figure 2.6, where # represents the unique character.

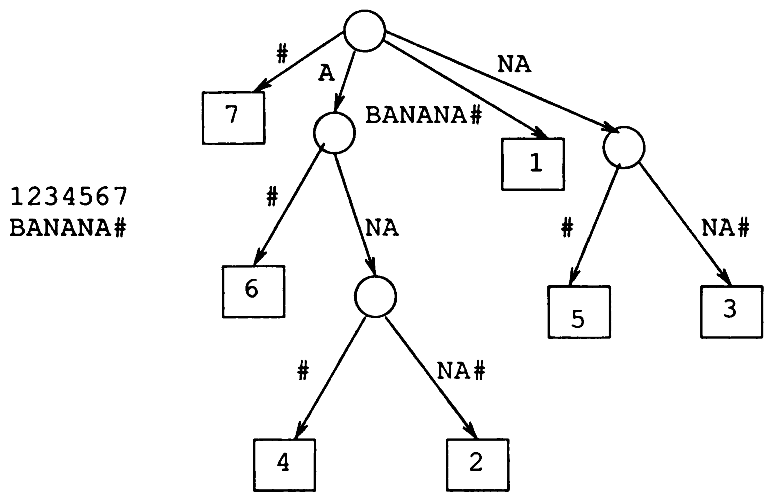


Figure 2.6: An Example of a Suffix Tree for "BANANA". Pointers to the suffix position are shown in leaf node.

Classical algorithms construct a suffix tree for a string of length n in $O(n \log |\Lambda|)$ time and $O(n)$ space (O gives upper bounds) [25, 26, 57]. A recent algorithm removes the dependence on alphabet size [58]. Because the suffix tree indexes all possible suffixes of the text, it occupies a great deal of space, e.g., 17 bytes per index point. Using Patrica trees in a similar setting, 12 bytes are required for each index point. PaTries [31] and PAT-trees [32, 28] are variants of Patricia tries, in which the Jacobson's encoding [59] of the tries is used to reduce space. The average space requirement for each index point in such variants is 6 bytes.

Suffix trees are powerful data structures. However, they use much space. Even though compression techniques help to reduce the size, suffix trees built on large text may easily exceed RAM size. Therefore, suffix trees have to be stored on disk. As shown in [24], since suffix trees have an unbalanced topology that is text-dependent, it is difficult to apply suffix trees on disk efficiently and dynamically. It is also shown in [33] that dynamically paging suffix trees on disk leads to decreasing storage utilization, e.g., 43% for 238KB text and 38% for 5.55MB text.

Suffix arrays store all the text suffixes in lexicographic order by their pointers. They are very space-efficient because only one pointer per suffix is stored. Suffix arrays are inherently static. They can be applied on external memory by partitioning the index into disk blocks. The performance on external memory degenerates when the text collection becomes large and changes over time. Searching in a suffix array requires $O(\log_2 N)$ number of disk accesses. This binary search may perform poorly because of the number of random disk accesses. To reduce the number of disk accesses, the *supra-index* has been used as the first step of the search [7]. The supra-index is a sampling of one out of b suffix array entries, where for each sample, the first l suffix characters are stored in the supra-index.

2.5 String B-trees

In a conventional B-tree, $\Theta(B)$ keys are stored in each internal node. However, if the keys are variable-sized text strings, the keys (i.e., string) can be arbitrarily long, and there may not be enough space to store $\Theta(B)$ strings per node, even using the heuristics adopted by the Prefix B-tree. Consequently, the performance of the tree degrades as the average fan-out (i.e., the number of children) of internal nodes decreases. The worst case of the B-tree indexing variable-length strings is not bounded as that of the B-tree indexing fixed length integers. In order to provide bounded fan-outs, a straightforward approach is to stored $\Theta(B)$ pointers to the $\Theta(B)$ strings in each internal node. However, accessing each string within the internal node during the search may require one disk access in the worse case. Therefore, the number of I/Os required for search is usually too high to make this approach useful. To solve this issue, the String B-tree is proposed so that not only the fan-out of an internal node is bounded by $\Theta(B)$, but also the number of I/Os required for searching an internal node is bounded by a constant number. Such feature is achieved by using a data structure similar to the Patricia trie within an internal node, and store keys in a separate file. The Patricia trie is used to determine the pointer to follow by accessing the file once. The resulting query time to search in a String B-tree for a string of l characters is therefore $O(\log_B N + l/B)$. Insertions and deletions can be done in the same I/O bound.

String B-tree provides a theoretical guaranteed worst-case performance for string searches. However, since the String B-tree stores strings in a separate file, two disk I/Os are required for searching an internal node. In leaf nodes, extra disk I/Os are required to access query answers. The performance of the String B-tree is hence usually worse than that of the Prefix B-tree. This is the reason why the Prefix B-tree is selected to compare with the HD-tree in Chapter 4.4.

Chapter 3: The HD-Tree

The HD-tree adopts a hybrid RAM/disk-based structure, in which leaf nodes are stored on the disk so that a large database can be indexed, and internal nodes are kept in the RAM to achieve greater efficiency. The HD-tree incorporates and extends some indexing strategies of the digital tree and the B*-tree [29], taking advantage of their strengths in search performance, compression capability, and storage utilization. The structure and construction algorithms of the HD-tree are presented in the following sections. Besides the notation and assumptions introduced in Section 1.2, symbol $\sharp$ is a special auxiliary symbol such that $\sharp \notin \Lambda$ and $\sharp < c$ for any $c \in \Lambda$. Assume Υ is a set of letters, functions $MIN(\Upsilon)$ and $MAX(\Upsilon)$ yield the smallest and greatest element in Υ, respectively.

3.1 Basic Structure

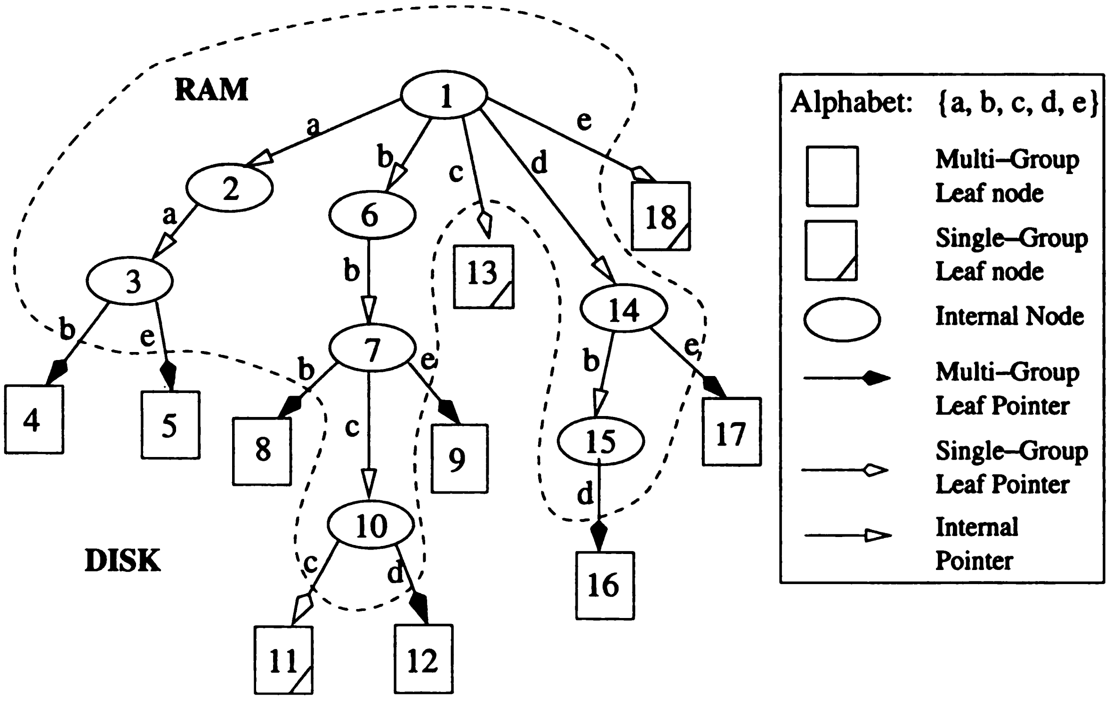


Figure 3.1: An HD-tree

The HD-tree is an unbalanced and ordered tree (see the example in Figure 3.1). An internal node, δ, of the HD-tree contains a list of pairs $L(\delta) = \{(a_1, P_1), (a_2, P_2), ..., (a_m, P_m)\}$, where P_i is a pointer to its child node; $a_i (1 \leq i \leq m)$ is a letter from Λ, called the *label* of P_i; and $a_1 < a_2 < ... < a_m$, such that the pointers are ordered according to their labels. The order of siblings (the nodes who have the same parent) are determined by the pointers. For example, the left sibling of node 6 is node 2, while the right sibling of node 6 is node 13. Leaf nodes, which are implemented as disk blocks, contain the suffixes of indexed strings. The *path string* of a tree node is the concatenation of the labels along the path traversing from the root to the node. The path string of the root is empty. Since an HD-tree node can be uniquely identified by its path string, a path string is also called an *id-string* (i.e., *identification string*) of the corresponding node. Let $ID(\delta)$ denote the id-string of a tree node δ. In Figure 3.1, $ID(2) = a$, $ID(9) = bbe$, and $ID(15) = db$.

3.2 HD-tree Properties

An HD-tree must satisfy two basic properties, which determine the proper leaf nodes for the indexed strings.

<u>PROPERTY</u> 1 *For each internal node, δ, in an HD-tree, $ID(\delta)$ is a common prefix of all strings contained in any leaf node in the sub-tree with δ as the root.*

Property 1 is similar to that of a digital tree. However, the id-string of a leaf node δ' in an HD-tree represents one or more prefixes (i.e., a set or "range" of prefixes) which strings in the leaf node, δ', may have. Let $PS(\delta')$ be the prefix-set of a leaf node δ'. If $|PS(\delta')| = 1$, all strings in δ' share the same common prefix in $PS(\delta')$. Such a leaf node is called a *Single-Group Leaf* (SGL). If $|PS(\delta')| > 1$, δ' may contain several groups of strings, where the strings in each group share a prefix which is different from the prefix of another group. Such a leaf node is called a

Multi-Group Leaf (MGL). The reason for using SGL and MGL is to improve disk utilization; otherwise, some large groups of strings may hinder the grouping of small groups. Based on Property 1, each prefix in $PS(\delta')$ differs only in the last letter. An internal node in an HD-tree may have three types of pointers: (1) Internal Pointer (IP) to an internal node; (2) Single-Group Leaf Pointer (SGLP) to an SGL; and (3) Multi-Group Leaf Pointer (MGLP) to an MGL.

In a traditional index tree such as the B-tree, any key, k, within a given range is kept in one node. This strategy is incorporated into an HD-tree by storing the keys having their prefixes within a "range" (i.e., a set) in the same leaf node. The reason to adopt this strategy in an HD-tree is based on the following observation: the group of keys with one prefix may be too small, and multiple such groups may fit in a leaf node (disk block), which can improve storage utilization.

A key range in a traditional index tree is continuous in that no key between the two boundaries of the range can be excluded. However, the prefix "range" (called the prefix-set) in the HD-tree may not be continuous because one or more prefixes between the two boundaries (minimum and maximum prefixes) of the range may be excluded. The reason to allow the exclusion of some prefixes from the range is that their corresponding key groups may be too large to share one leaf (block) with others. In such cases, one or more separate leaves (disk blocks) are used to store the group of keys corresponding to such a prefix of the large group and keep the remaining small groups of keys (with prefixes within the "range") in another leaf node.

The prefix-set, $PS(\delta')$, for an SGL, δ', contains the unique prefix $ID(\delta')$, i.e., $PS(\delta') = \{ID(\delta')\}$. For example, in Figure 3.1, node 11 is an SGL with $PS(11) = \{bbcc\}$; that is, all the strings in this node have the common prefix, *bbcc*. It is the task of the tree building algorithms to determine which node is an SGL. For example, a percentage of the free space in a leaf node is used in the HD-tree to determine an SGL.

Unlike an SGL, whose prefix-set is directly presented by its id-string, the prefix-set of an MGL needs to be derived as follows: let δ' be an MGL, and δ be the parent node of δ' containing list $L(\delta) = \{(a_1, P_1), \dots, (a_k, P_k), \dots, (a_m, P_m)\}$, where $m > 0$ and P_k is the pointer to δ'. Let $\beta = ID(\delta)$, the prefix-set of MGL δ' is defined as:

$$PS(\delta') = \{\beta c \,|\, c \in \Upsilon_{P_k}\}, \tag{3.1}$$

where Υ_{P_k} is a set of letters obtained through the following steps:

$$\begin{aligned}
1:&\ \ \Upsilon'_{P_k} = \{a_i \,|\, (a_i, P_i) \in L(\delta), a_i < a_k, P_i \text{ is an MGLP}\};\\
2:&\ \ \textbf{if } (\Upsilon'_{P_k} \text{ is empty}\ )\ b' = \sharp; \textbf{else } b' = MAX(\Upsilon'_{P_k});\\
3:&\ \ \Upsilon_\Lambda = \{a \,|\, a \in \Lambda, b' < a \le a_k\};\\
4:&\ \ \Upsilon''_{P_k} = \{a_j \,|\, (a_j, P_j) \in L(\delta), b' < a_j < a_k,\\
&\qquad\qquad P_j \text{ is an IP or SGLP }\};\\
5:&\ \ \Upsilon_{P_k} = \Upsilon_\Lambda - \Upsilon''_{P_k}.
\end{aligned}$$

The set, Υ_Λ, contains all letters (i.e., a continuous range) between the last letter, b', of the id-string of the closest left sibling MGL (if any, otherwise $b' = \sharp$) and the last letter, a_k, of the id-string of the current leaf, δ'. However, any letter between b' and a_k that is used as a label for a pointer to an internal node or an SGL is excluded from Υ_Λ. The letters in the resulting set, Υ_{P_k}, are used as the last letters for the prefixes in $PS(\delta')$, which may not be continuous (i.e., some prefixes within the range may be excluded). The following two examples are given to illustrate the steps to find the prefix-set of a leaf node.

Example 1 Find the prefix-set of node 9 in Figure 3.1.

In Figure 3.1, the internal node, 7, contains a list $L(7) = \{(b, P_1^8), (c, P_2^{10}), (e, P_3^9)\}$, where P_i^n pointing to node n. The steps to find the prefix-set of node 9 are as follows:

$$\begin{array}{llll} 1: & \Upsilon'_{P_3^9} = \{b\}; & 2: & b' = b; \\ 3: & \Upsilon_\Lambda = \{c, d, e\}; & 4: & \Upsilon''_{P_3^9} = \{c\}; \\ 5: & \Upsilon_{P_3^9} = \{d, e\}. & & \end{array}$$

Since $\beta = ID(7) = bb$, $PS(9) = \{bbd, bbe\}$, where *bbc* is excluded from $PS(9)$, the group of keys with prefix *bbc* are stored in the leaf nodes 11 and 12.

Example 2 Find the prefix-set of node 12 in Figure 3.1.

In Figure 3.1, the internal node, 10, contains a list $L(10) = \{(c, P_1^{11}), (d, P_2^{12})\}$. P_1^{11} is an SGLP, and the steps to find $PS(12)$ are as follows:

$$\begin{array}{llll} 1: & \Upsilon'_{P_2^{12}} = \{\}; & 2: & b' = \sharp; \\ 3: & \Upsilon_\Lambda = \{a, b, c, d\}; & 4: & \Upsilon''_{P_2^{12}} = \{c\}; \\ 5: & \Upsilon_{P_2^{12}} = \{a, b, d\}; & & \end{array}$$

Since $\beta = ID(10) = bbc$, $PS(12) = \{bbca, bbcb, bbcd\}$. Prefix *bbcc* (which is greater than *bbcb* and smaller than *bbcd*) is excluded from $PS(12)$. Therefore, $PS(12)$ is not a continuous range of prefixes.

<u>Property</u> 2 *Each leaf node, δ', in an HD-tree keeps all the strings with a prefix in its prefix-set, $PS(\delta')$.*

Based on the previous discussion on the prefix-set, Property 2 of the HD-tree guarantees that any key is placed in one and only one leaf node of an HD-tree. In a traditional index tree (e.g., the B-tree), not every key in the range of a leaf node must appear in the node. If a user wants to search for such a key, the search algorithm will report that the key is not in the database. Similarly, each prefix in the prefix-set of an MGL does not necessarily have a corresponding key in the MGL.

In the HD-tree, keys are stored in leaf nodes. Since the prefix of a key can be found along the path from the root to a leaf node, to duplicate the prefix in the leaf node is not necessary. Therefore, only suffixes are stored in leaf nodes physically. However, an MGL, δ', has more than one prefix. If we only store the suffix (after removing its prefix in $PS(\delta')$) of a key in the node, we would lose the corresponding relationship between the suffix and the prefix of the key. As previously discussed, each prefix in $PS(\delta')$ differs only in the last letter. To resolve the above ambiguity, the suffix of a key is saved together with the last letter of its prefix in $PS(\delta')$. To be consistent, the keys in an SGL are also saved in the same way. Note that the key compression feature (i.e., sharing the common prefixes at the internal nodes and storing only suffixes in the leaf nodes) in the HD-tree increases its capability to index large databases for given RAM and disk sizes.

Because the HD-tree guarantees that a key can be kept in one and only one leaf node, several features can be derived regarding the optimality of the tree to support various types of string queries. Assume that all internal nodes of the HD-tree are kept in the RAM. The first feature is:

FEATURE 1 *Let Q be an exact, prefix, or substring search. If there exists an answer(s) to Q, the HD-tree is optimal for Q. Only the leaf node(s) containing the answer(s) will be accessed. If Q has no answer, at most one disk access is required.*

For a range string query that finds any string, α, such that $\beta_1 \leq \alpha \leq \beta_2$, where β_1 and β_2 are the boundary strings, the second proposition is:

FEATURE 2 *If both boundary strings are in the answer set for a range string query, the HD-tree is optimal for the query.*

Feature 2 only gives a sufficient condition for the optimality of the HD-tree when processing a range query. In other words, even if one or both boundary strings are not in the answer set, the HD-tree can still be optimal as long as the leaf node for the missing boundary string has at least one answer to the query. If the leaf node(s) for the boundary string(s) contains no answer to the query, the access to that leaf node(s) (disk block) is an extra access(s). Hence, the third feature is:

FEATURE 3 *At most two extra disk accesses are required to process a range query using the HD-tree.*

After the basic structure and properties of the HD-tree are defined, it is important to develop algorithms to build the HD-tree according to the structure and properties.

3.3 Building the HD-Tree

To build an HD-tree, algorithms are needed for insertion, deletion, and update. A deletion is the reverse of an insertion, while an update can be implemented by a deletion followed by an insertion. In this dissertation, only the insertion issues and its related algorithms are discussed. Note that *IP* (Internal Pointer), *SGLP* (Single-Group Leaf Pointer), and *MGLP* (Multi-Group Leaf Pointer) are used in algorithm descriptions.

3.3.1 Insertion Procedure

The insertion procedure inserts a new string, κ, into a given HD-tree, where $\kappa = k_1 ... k_n$, $k_i \in \Lambda$, and $1 \leq i \leq n$. The special symbol,$\sharp$, is appended at the end of a string to prevent the string becoming any id-string in the given HD-tree. The insertion process uses the prefix, $k_1 \dots k_i$, to find an internal node whose id-string is equal to $k_1 \dots k_i$. Then the letter, k_{i+1}, is used to find a qualified leaf node. The insertion continues within the leaf node using the rest of the string, $k_{i+1} \dots k_n$. The root of an HD-tree is at level 1. Given an internal node, δ, at level l, k_l is used to determine the next pointer to follow. Algorithm 1 (HD-Insert) is a recursive procedure. It first finds a leaf node, δ', to accommodate κ. Once δ' is found, κ is inserted into δ'. If δ' overflows (i.e., the node size exceeds the disk block capacity) as a result of the insertion, Algorithm 2 (HD-OverflowProc) is invoked. When a string is inserted into an empty HD-tree, an empty internal node (the root) is created before invoking Algorithm HD-Insert.

ALGORITHM 1 : **HD-Insert(κ, l, δ)**

Input: (1) a new string $\kappa = k_1 ... k_n$; (2) the current level l; (3) the current internal node δ, where $L(\delta) = \{(a_1, P_1), \ldots (a_k, P_k), \ldots (a_m, P_m)\}$.

Output: an updated HD-tree

Method:

```
1.  for $i = 1$ to $m$ do
2.   if $a_i == k_l$ and $P_i$ is an IP then
3.     call HD-Insert($\kappa$, $l + 1$, $\delta_{P_i}$); return;
4.   else if $a_i \geq k_l$ and $P_i$ is a SGLP
     or MGLP then
5.     compute $\Upsilon_{P_i}$ as shown in Section 2.2;
6.     if $k_l \in \Upsilon_{P_i}$ then
7.       read $\delta'_{P_i}$ from disk into $\delta'$; goto 18;
8.     end if;
9.   end if;
10. end for;
11. for $j = m$ to 1 do
12.  if $P_j$ is a MGLP then
13.    read $\delta'_{P_j}$ from disk into $\delta'$;
       $a_j = k_l$; goto 18;
14.  end if;
```

```
15. end for;
16. create an empty leaf node $\delta'_P$ as an MGL;
17. add $(k_l, P)$ into $L(\delta)$
    where $P$ is a $MGLP$ pointing to $\delta'$;
18. add $k_l ... k_n$ into $\delta'$;
19. if $\delta'$ overflows then
20.   call HD-OverflowProc($\delta$, $\delta'$);
21. end if;
22. write $\delta'$ back to disk;
23. return;
* $\delta_P$ indicates the tree node pointed to by $P$.
```

Algorithm HD-Insert follows internal pointers down the tree as far as possible (steps 1-4). Then, if there exists an SGL or MGL that can accommodate the search key, the node is read into the RAM as the chosen leaf node (steps 5-11). Otherwise, the right most MGL node is chosen (steps 13-15) and its range is expanded (step 16). If no leaf node is qualified to accommodate the key (e.g., only internal nodes are available), a new leaf node is created and attached to its parent node (steps 20-21). Finally, the key is inserted into the chosen leaf node (step 22). If the leaf node overflows after the insertion, Algorithm HD-OverflowProc is invoked (step 24); otherwise, the leaf node is written back to disks (step 26).

3.3.2 Overflow Processing

As more keys are inserted into a leaf node, the leaf node may exceed its capacity to accommodate more keys (i.e., overflow). Before describing the algorithm to process an overflow leaf node, the concepts and notation used in the description are first introduced. In HD-trees, only suffixes of the original strings are stored in a leaf node (See step 18 in Algorithm HD-Insert). These suffixes are called

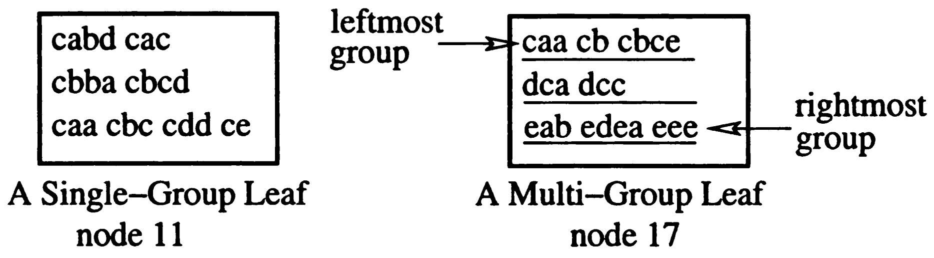


Figure 3.2: Examples of the SGL and the MGL.

suffix-strings. A *group* is a set of suffix-strings whose first letters are the same. The common first letter of a group is called the *group-head.* Groups are ordered by their corresponding group-heads. The first group is called the *leftmost group* and the last group is called the *rightmost group.* In other words, given a leaf node, δ', whose parent is δ, a group in δ' contains strings having the same prefix, βc, where c is the group-head and $\beta = ID(\delta)$. As discussed in Section 3.1, an SGL contains only one group and an MGL contains one or more groups. For example, in Figure 3.2, node 11 is an SGL which contains one group whose group-head is *c*. Node 17 is an MGL which contains three groups, whose group-heads are *c*, *d* and, *e*, respectively.

3.3.3 Linked Leaf Nodes

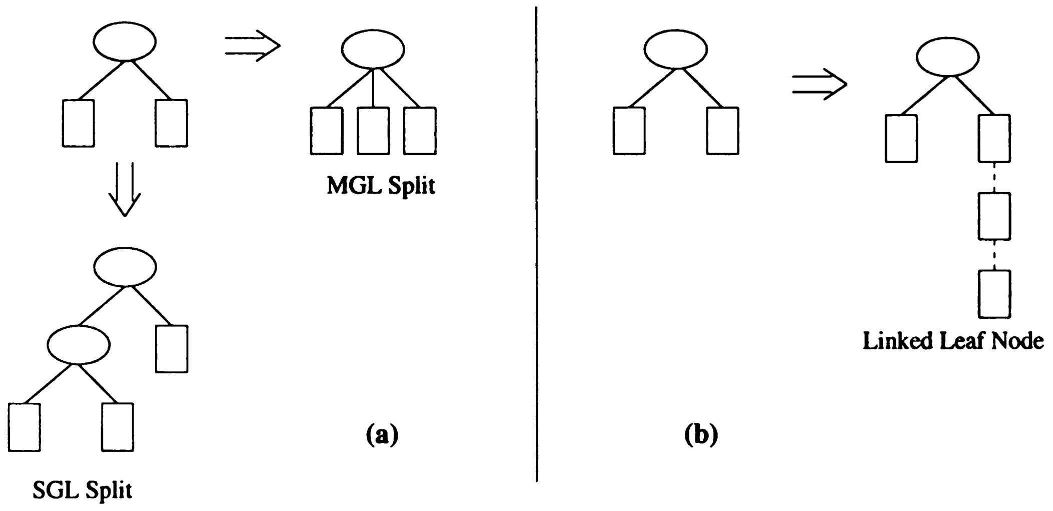


Figure 3.3: Examples of the HD-tree Growth

The HD-tree keeps track of the current available RAM whenever adding or deleting an internal node (not shown explicitly in the algorithms). When RAM is available, the tree grows by creating internal nodes through overflow processing and splitting (see Figure 3.3a). When there is no available RAM, the tree stops creating new internal nodes. Hence, if a leaf node exceeds the disk block size after inserting a string, an extra disk block is linked to the original disk block to accommodate the overflowing data (see Figure 3.3b). Consequently, a search within the leaf node accesses all linked disk blocks. Using this approach, the HD-tree works with any size of RAM.

ALGORITHM 2 : **HD-OverflowProc(δ, δ')**

```
Input: (1) an internal node $\delta$, where
$L(\delta) = \{(a_1, P_1), \dots (a_k, P_k), \dots (a_m, P_m)\}$;
(2) an overflow leaf node, $\delta'$, pointed to by $P_i$ in $L(\delta)$.
Output: an updated HD-tree
Method:
1. if the current RAM is not enough to create
       a new internal node then
2.   link a new disk block to $\delta'$; return;
3. end if;
4. if $\delta_{P_i}$ contains only one group then
5.   create an internal node $\delta_x$;
6.   remove $(a_i, P_i)$ from $L(\delta)$;
7.   add $(a_i, P')$ into $L(\delta)$ where P' is an IP
     pointing to $\delta_x$;
8.   remove the first letter $a_i$ from each
      suffix-strings in $\delta'$;
9.   add $(b, P'')$ into $L(\delta_x)$ where $b$ is the greatest
```

```
        group-head in current $\delta'$ and $P''$ is a
        MGL pointer to $\delta'$;
10.     **if** $\delta'$ still overflows **then**
11.       **call HD-OverflowProc**($\delta_x$, $\delta'$);
12.     **else**
13.       write ($\delta'$) back to disk; **return**;
14.     **end if**;
15.   **else**
16.     **call HD-Split**($\delta$, $\delta'$, *user-defined threshold*);
17.   **end if**;
18.   **return**;
```

Algorithm 2 (HD-OverflowProc) is a recursive procedure to handle overflow leaf nodes. If the overflow leaf node, δ', is an SGL, an internal node is created and the HD-tree grows one level down on the corresponding branch (steps 1-6). HD-OverflowProc may be invoked again if δ' continues to overflow (steps 7-8). If δ' is an MGL, Algorithm 3 (HD-Split) is invoked (step 14).

3.3.4 Split Heuristics

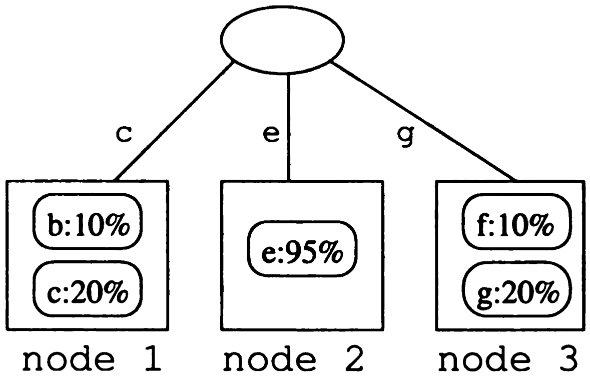


Figure 3.4: Illustration of a large group hindering a possible merge

In the HD-tree, suffix-strings in overflow leaf nodes are split by group. If a node is split into two as soon as it overflows (called *SSplit* Algorithm), the resulting storage utilization is very low. This low storage utilization has two main causes. First, since leaf nodes are ordered by the labels of leaf node pointers, a leaf node containing a large (in size) group may hinder the possible merging of left and right siblings. For example, in Figure 3.4, leaf node 2 contains a large group 'e' (95% of the block size). Group 'e' cannot be stored in either leaf node 1 or 3, since it will cause overflow. If group 'e' does not exist, leaf node 1 and 3 can be merged into one leaf node, which will increase storage utilization. Second, since splitting is done by group, it may divide groups in an unbalanced way. The key range (i.e., the prefix-set) of a leaf node keeps shrinking without any possibility of expanding. Consequently, many underflow leaf nodes (where the storage utilization is less then 50%) are created that may be merged with siblings. Figure 3.5 illustrates a series of splitting that creates two underflow leaf nodes (b2 and b3), which could be merged into one node.

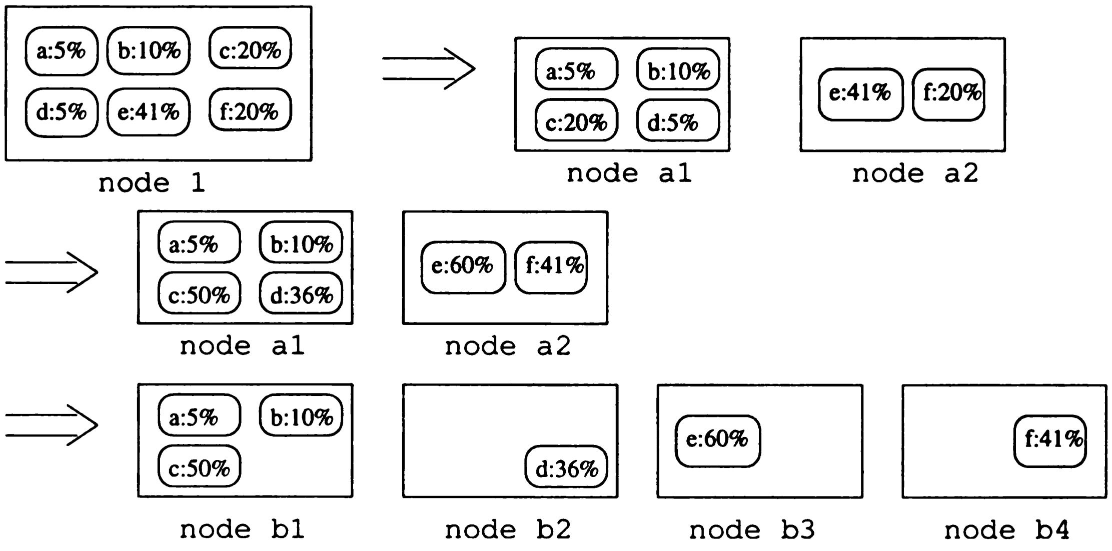


Figure 3.5: Illustration of the underflow leaf nodes generated by simple split (*SSplit*)

According to the above two observations, in order to improve the storage utilization, two heuristics are used: (1) an SGL is created if the size of a group is greater than a user-defined threshold T (e.g., 85% of the disk block size), (2) after an a large group is moved out of an overflow leaf node into an SGL, and before an overflow node is split, groups may be moved to qualified siblings to avoid splitting. The first heuristic is to move a large group into an SGL, so that it does not interfere with the grouping of an MGL. The second heuristic helps to expand or shrink the key range (i.e., the prefix-set) of a leaf node in order to improve the storage utilization.

<u>Algorithm</u> 3 : **HD-Split**(δ, δ', T)

```
Input: (1) an internal node $\delta$, where
$L(\delta) = \{(a_1, P_1) \dots (a_k, P_k) \dots (a_m, P_m)\}$;
(2) an overflow MGL $\delta'$ pointed to by $P_i$ in $L(\delta)$;
(3) a threshold $T$.
Output: an updated HD-tree
Method:
 1. if $\delta'$ contains a group $g_x$ whose size is
        greater than $T$ then
 2.   create an empty leaf node $\delta'_x$ as an SGL;
 3.   move $g_x$ from $\delta'$ into $\delta'_x$;
 4.   add $(a_x, P_x)$ into $L(\delta)$ where $a_x$ is the
       group-head of $g_x$ and $P_x$ is a SGLP pointing
       to $\delta'_x$;
 5.   adjust$(a_i, P_i)^*$;
 6. end if;
 7. if a left/right MGL sibling $\delta_{P_y}$ of $\delta'$ has
        space to accommodate the leftmost/rightmost
      group $g$ in $\delta'$ then
 8.   read $\delta'_{P_y}$ from disk;
 9.   move $g$ from $\delta'$ into $\delta'_{P_y}$;
```

```
10.   adjust($a_i, P_i$) and ($a_y, P_y$);
11. end if;
12. if $\delta'$ overflows then
13.   if $\delta'$ contains only one group then
14.     call HD-OverflowProc($\delta$, $\delta'$);
15.   else
16.     create an empty leaf node $\delta'_z$ as an MGL;
17.     move groups from $\delta'$ into $\delta'_z$ one by one in
        increasing order until $\delta'_z$ is more than half
        full or there is only one group left in $\delta'$;
18.     add ($a_z, P_z$) into $L(\delta)$;
19.     adjust ($a_i, P_i$) and ($a_z, P_z$);
20.   end if;
21. end if;
22. write new and modified leaf node(s) to disk;
23. return;
```

* **adjust**(a_i, P_i) *is a procedure which sets* a_i *to the largest group-head in the leaf node* $\hat{\delta}'_{P_i}$ *and marks* P_i *as SGLP or MGLP correspondingly.*

In Algorithm HD-Split, heuristic (1) is implemented in steps 1-6, while heuristic (2) is implemented in steps 7-11. If δ' still overflows after the two heuristics are applied, δ' is split and a new leaf node is created to accommodate some groups in δ' (steps 12-17).

Figure 3.6 shows a few examples of HD-tree splitting and the linked leaf nodes after inserting more strings into the original HD-tree (see Figure 3.1). For example, node 11 in Figure 3.1 grows into an internal node 11 and two leaf nodes: 19 and 20. Node 17 splits into two leaf nodes: 17 and 21. Node 18 grows to three linked leaf nodes: 18a, 18b, and 18c, which happens when the RAM is not available for further

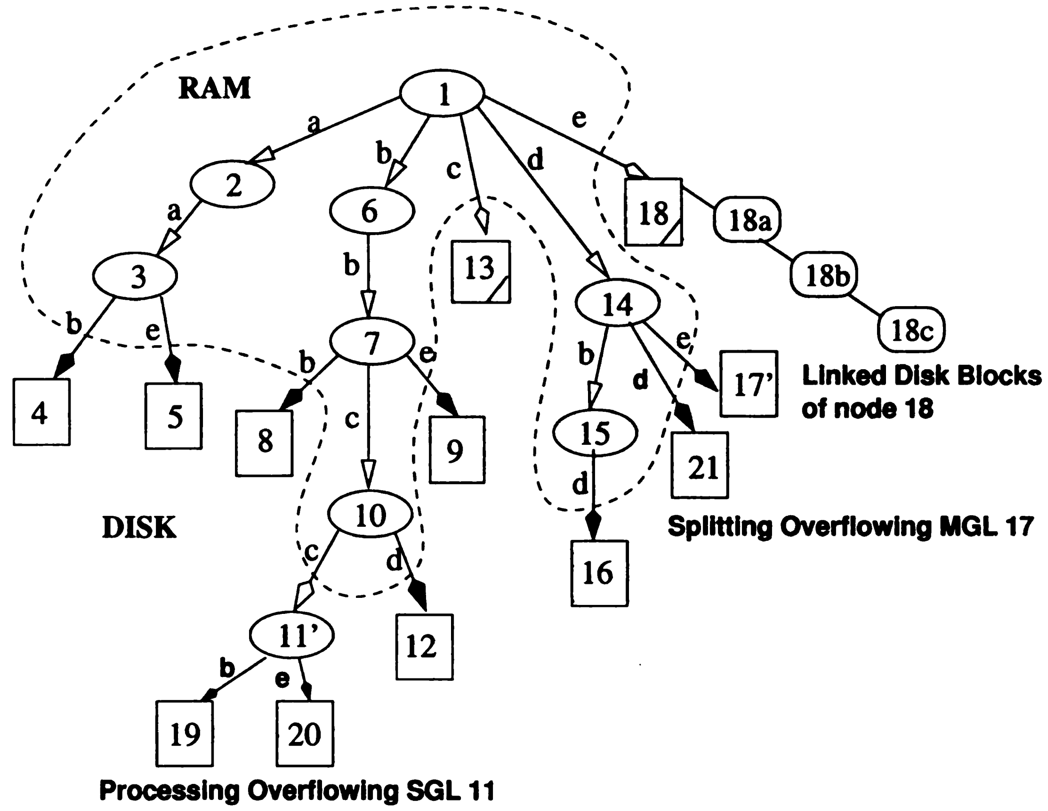

RAM
DISK
1
2
3
4
5
6
7
8
9
10
11'
12
13
14
15
16
17'
18
18a
18b
18c
19
20
21
Linked Disk Blocks of node 18
Splitting Overflowing MGL 17
Processing Overflowing SGL 11

Figure 3.6: Examples of HD-tree splitting and linked leaf nodes

splitting.

After the HD-tree is built, algorithms are needed to search the HD-tree for different types of queries. The next chapter presents the search algorithms and the performance of the HD-tree.

Chapter 4: HD-tree Behavior

Once the HD-tree is built, various queries can be conducted using the tree. In this chapter, algorithms for prefix search and approximate string matching based on the Hamming distance are presented. Characteristics of the HD-tree are discussed. The experimental results show that HD-tree outperform existing data structures, such as the Prefix B-tree for prefix searches and the M-tree for approximate string matching based on the Hamming distance.

4.1 Prefix Search

The HD-tree is able to handle exact, prefix, and sub-string searches. In this section, Algorithm 4 (HD-PrefixSearch) is presented for prefix searches. Other string queries (see Section 1.2) can be performed using the prefix search algorithm. For example, $ExactSearch(\alpha)$ is equivalent to $PrefixSearch(\alpha\sharp)$. $RangeSearch(\alpha, \beta)$ can be done by first finding the leaf nodes storing α and β using $PrefixSearch(\alpha\sharp)$ and $PrefixSearch(\beta\sharp)$, respectively, then sequentially access all the leaf nodes between these two leaf nodes. Similar to the approach used in [24], the HD-tree handles sub-string searches within a set of strings $\{\kappa_1, ..., \kappa_n\}$ by indexing all suffix strings of $\kappa_1, ..., \kappa_n$. Therefore, $SubstringSearch(\alpha)$ can be performed by $PrefixSearch(\alpha)$ on these suffix strings.

```
ALGORITHM 4 HD-PrefixSearch($\kappa$, $l$, $\delta$)
    Input: (1) the query string $\kappa = k_1 ... k_n$;
    (2) the current level $l$;
    (3) the current internal node $\delta$, where
    $L(\delta) = \{(a_1, P_1) \ ... \ (a_k, P_k) \ ... \ (a_m, P_m) \}$;
    Output: query result(s)
```

Method:

```
1.  if $l > n$ then
2.    return all strings in leaf nodes under $\delta$;
3.  end if;
4.  for $i = 1$ to $m$ do
5.    if $a_i == k_l$ and $P_i$ is an $IP$ then
6.      return HD-PrefixSearch( $\kappa$, $l + 1$, $\delta_{P_i}$);
7.    else if ($P_i$ is an $SGLP$ and $a_i == k_l$)
          or ( $P_i$ is an $MGLP$ and $a_i >= k_l$) then
8.      read $\delta'_{P_i}$ from disk
9.      retrieve strings in $\delta'_{P_i}$ which have a
        prefix $k_l ... k_n$;
10.   end if;
11. end for;
12. return;
```

Algorithm HD-PrefixSearch starts from the root of an HD-tree, which is at level 1, and traverses down the tree as far as possible (steps 4-6). If a leaf node is encountered, the search will read the leaf node from disks and linearly search the strings in the leaf (steps 7-10). If the prefix is shorter than the id-string of a node (see Section 3.1), all the strings in the sub-tree are the answers (steps 1-3). Figure 4.1 shows the paths of searching "aa" (node $1 -> 2 -> 3 -> 4$, 4a, 4b, 5) and "bbcda" (node $1 -> 6 -> 7 -> 10 -> 12$).

4.2 Data Sources for Experiments

The performance of the HD-tree on prefix searches are tested using real textual data. A sample database, WSJ1, is generated from the Wall Street Journal (entire

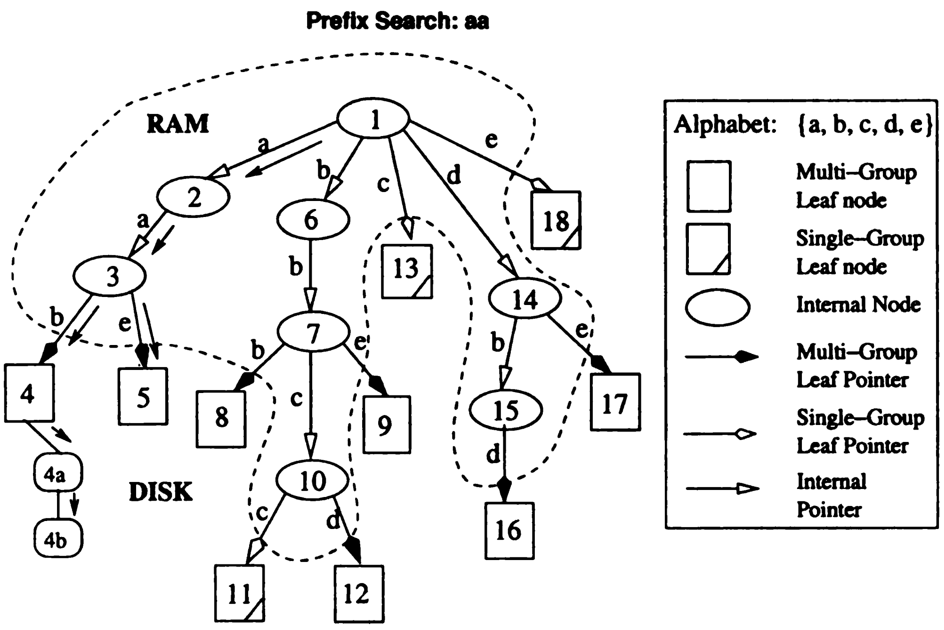


Figure 4.1: Examples of String Searches

year of 1991), which is a part of the Text REtrieval Conference (TREC) collection [37]. Markup tags are removed, texts are split into segments of 5MB each, and unique prefixes of the suffix strings at word boundaries are extracted for every segment. If a prefix string is longer than 32, only the first 32 letters are kept to reduce storage requirement. WSJ1 can be used for keyword-based document searches [7]. Similarly, database WSJ2 is generated from the same data source, except that suffix strings start at letters (not spaces). WSJ2 is used for sub-string searches [28]. The sample query set, Q1, is generated by randomly selecting keywords from the Wall Street Journal (1991). The sample query set, Q2, is generated by randomly selecting substrings from WSJ2. The sample database, GENO, is generated using DNA sequences from GenBank (see Chapter 6 for GenBank Overview). Overlapping words (strings) of length 28 (i.e., a fixed window size of 28 shifting from the beginning to the end of a sequence by one letter at a time) are extracted from these DNA sequences. Conducting similarity searches on these overlapping strings is useful for finding homologous regions in genomic sequence databases [60]. To increase the efficiency, every two symbols from the

DNA alphabet, $\{A, C, G, T\}$, are encoded into one symbol. Hence, the string length becomes 14. Each of these databases consists of 15 million strings and each string is associated with a four-byte integer containing position information. WSJ1 is used for prefix searches, while GENO is used for approximate string matching based on the Hamming distance.

Statistics of these databases and query sets are shown in Table 4.1, where *key#* indicates the number of database keys; *max*, *min*, and *avg* are the maximum, minimum, and average length of keys, respectively. The query performance (the number of I/Os) in the experimental results is the average among the 100 queries for text databases and 1000 queries for DAN sequence databases.

Table 4.1: Statistics of sample databases and queries

DB	*size*	*key#*	*min*	*max*	*avg*
WSJ1	260.0MB	15M	3	33	14.18
WSJ2	251.8MB	15M	2	33	13.60
GENO	257.5MB	15M	14	14	14
Q1	856B	100	4	20	8.56
Q2	902B	100	4	10	9.02
Q3	14000B	1000	14	14	14

4.3 HD-Tree Behavior

Experiments are conduced to analyze the behavior of the HD-tree and evaluate its performance by comparing it with existing techniques. The HD-tree is implemented using C++ and experiments are conducted on a PC with 512MB RAM and 1.8GHz Pentium 4 processor, running Linux OS. The disk block size used in the experiments is 4096 bytes.

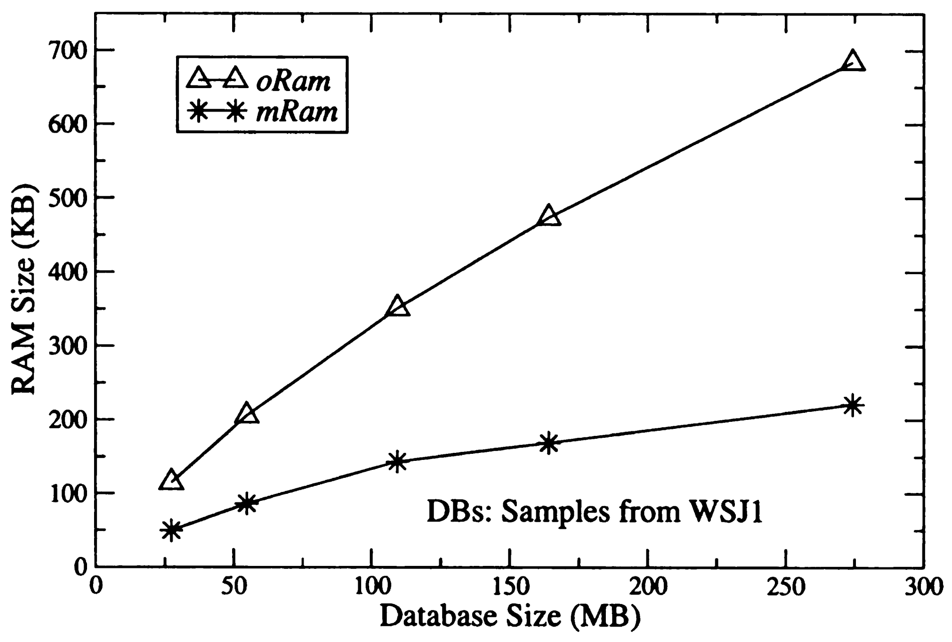


Figure 4.2: RAM Usage of HD-trees

Table 4.2: RAM/disk Usage of HD-trees

DB	27.39	54.75	109.4	164.1	219.2	274.3
RAM	0.1123	0.2009	0.3428	0.4623	0.5716	0.6679
disk	32.08	57.04	96.95	130.5	161.1	188.2
total	32.19	57.24	97.3	131	161.7	188.9
ratio(%)	117.5	104.5	89.93	79.83	73.77	68.87

DB, *RAM*, *disk*, *total*: MB; Databases: Samples from WSJ1; *ratio* = total/DB; ALN=0

4.3.1 RAM and Disk Usage

Figure 4.2 illustrates the RAM usage of the HD-tree, where *mRam* is the minimum RAM size needed to achieve the minimal-IO (i.e., the optimal I/O achieved by the HD-tree), and *oRam* is the minimum RAM size required to achieve near-optimal performance. Table 4.2 shows the RAM, disk, and total (RAM+disk) size of HD-trees. Note that the ratio between the total size of an HD-tree and the corresponding database size decreases as the database size increases. This performance gain is achieved by the compression feature of the HD-tree.

4.3.2 Split Heuristics

A set of experiments is designed to show the effectiveness of the split heuristics for building an HD-tree. Table 4.3 shows the comparison of the storage utilization (using one disk block for each leaf node) between the SSplit, which is a $B^{+}tree$-like approach, and the HD-Split (see Section 3.3.4). Note that the HD-Split adopted two heuristics to improve the storage utilization. One heuristic is to distinguish the SGL from the MGL, which allows the prefix range to be "discontinuous." The other heuristic is to move groups to the left or right sibling to avoid a split, so that the prefix set of an MGL is dynamically adjusted. It is shown that the HD-Split increases the disk utilization by more than 40%, which indicates the effectiveness of the grouping mechanism in the HD-Split.

Table 4.3: Split heuristics on storage utilization

DBSize(MB)	50	100	150	200	250
SSplit	45.7	44.8	44.6	44.5	44.1
HD-Split	65.1	63.5	63.1	62.7	62.6
Improve	42.5	41.7	41.5	40.1	42.0

Databases: Samples from WSJ1, Table value: %

4.3.3 Threshold Phenomena

As described in Section 3.3.3, using linked disk blocks, the HD-tree is scalable for any RAM size. Figure 4.3 shows the relationship between the average number of links (ANL) and the available RAM size as the percentage of the database size (RAM/DB). The ANL is the total number of linked disk blocks divided by the number of linked leaf nodes. An ANL value of zero means that each leaf node occupies one disk block. As shown in Figure 4.3, the ANL decreases as the RAM/DB increases. Note that there exists a threshold (where the curve becomes flat) in the figure. The threshold is almost invariant of database sizes.

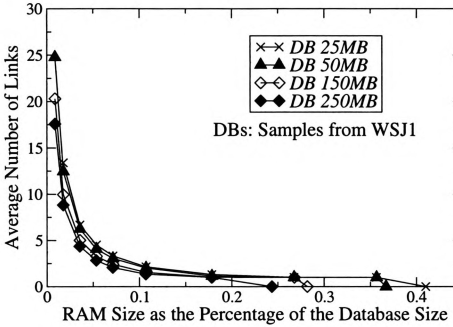


Figure 4.3: The relationship between the ANL and the available RAM as the percentage of the database size.

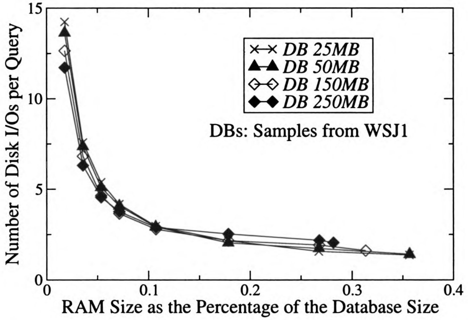


Figure 4.4: The relationship between the number of I/Os and the available RAM when the answer size is fixed.

When ANL is greater than zero (i.e., the linked disk blocks are used), the query performance of the HD-tree is shown to be closely related to the ANL. Curves in Figures 4.4 and 4.5, where the number of I/Os rather than ANL is used, are similar to those in Figure 4.3. The threshold phenomenon is due to the logarithmic nature of the tree (i.e., lower level contains less nodes). As the HD-tree grows, adding the same amount of RAM (i.e., increasing a certain number of leaf nodes) has a decreased impact on the selectivity of the tree (i.e., the total number of leaf nodes). Therefore, when the available RAM is limited with respect to the database size, it is important to allocate enough RAM at the threshold point where the RAM is most effectively utilized.

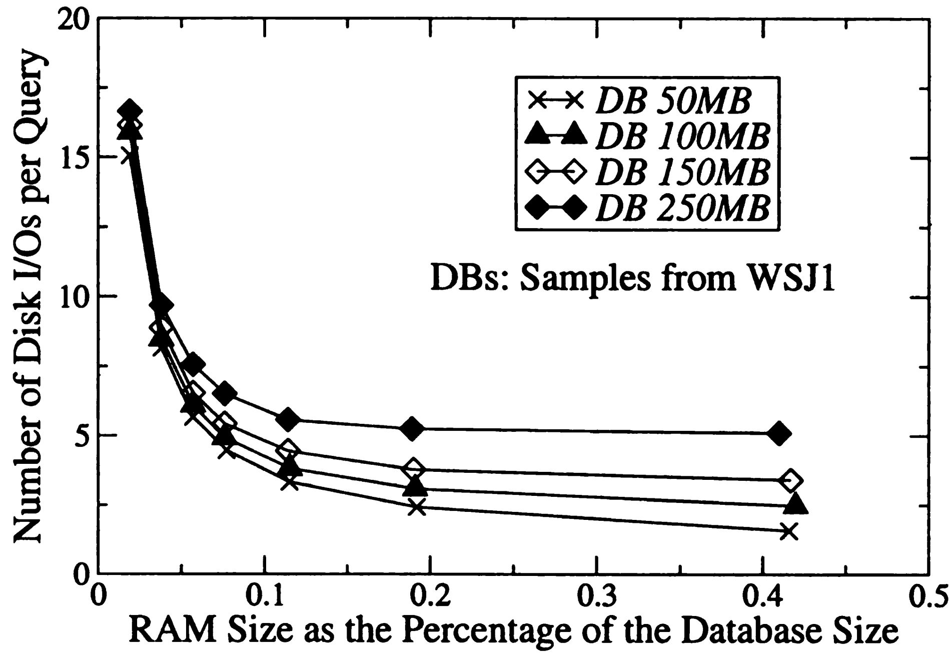


Figure 4.5: The relationship between the number of I/Os and the available RAM when the answer size changes.

4.4 Comparisons with the Prefix B-tree

In this section, the performance of the HD-tree is evaluated by comparing it with that of the Prefix B-tree. The Prefix B-tree is widely adopted by database systems and has been shown to be a practical technique for indexing large string databases. The Prefix B-tree used is the experiments was implemented by the popular Berkeley DB [61], which is an open source database system. As a disk-based index structure, the Prefix B-tree does not require any RAM, while the HD-tree requires a certain amount of RAM to keep its internal nodes. For a fair comparison, the same amount of RAM used by the HD-tree is provided for the Prefix B-tree as a cache. The caching algorithm is based on the popular LRU (least-recently-used) heuristic, which is used by almost all commercial database systems because of its simplicity and effectiveness. The LRU algorithm keeps recently accessed internal nodes in the RAM to reduce the number of disk I/Os.

The disk I/Os are compared between the HD-tree and the Prefix B-tree using 1000 queries with different numbers of distinctive queries. This set of experiments is designed to evaluate the effect of the locality of the query results on the performance of the HD-tree and the Prefix B-tree. The queries are generated as follows: (1) generate a certain number of distinct queries to form a query pool; (2) randomly generate 1000 queries from the query pool. In one extreme case, the 1000 queries are all the same (i.e., one distinctive query), i.e., all the 1000 queries return the same results. As the number of distinctive queries increases, the level of localities in the query results reduces. The other extreme is when all 1000 queries are different.

As shown in Figure 4.6, the performance of the Prefix B-tree is better when the number of distinctive queries are small. However, as the number of distinctive queries increases, the performance of the Prefix B-tree deteriorates quickly. The two curves cross between 10 and 20 distinct queries, where the HD-tree starts to outperform the Prefix B-tree. For 1000 distinctive queries, the HD-tree is almost three times better than the Prefix B-tree in term of the number of disk I/Os. The results show that the performance of the Prefix B-tree using the LRU caching mechanism is very susceptible to the locality of the query results. On the other

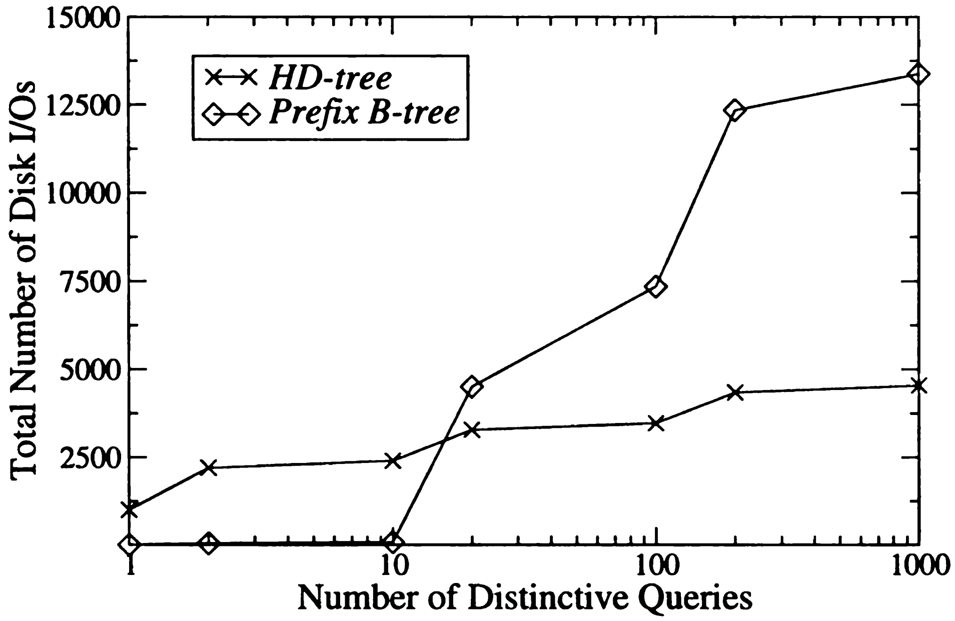


Figure 4.6: I/O comparison for different query localities; average query length is 6.

hand, the HD-tree is robust to distinctive queries. It is concluded that the HD-tree performs better as queries become more distinctive. In the following comparisons, 1000 distinctive random queries are used.

In Figures 4.7 and 4.8, the performance of the HD-tree and the Prefix B-tree is compared for different RAM sizes. In Figure 4.7, it is shown that the HD-tree not only reduces the number of I/Os, but also uses the RAM more effectively than the caching mechanism adopted by the Prefix B-tree. For example, as the RAM increases from 250KB to 1.6MB, the HD-tree reduces more than 50% of I/Os, but the Prefix B-tree only reduces less than 20% of I/Os. For the given database WSJ1 (252MB) and 1.6MB of RAM, the HD-tree reaches its optimal status where each leaf node occupies only one disk block. In Figure 4.8, more RAM to the HD-tree is served as a cache which is the same as that of the Prefix B-tree. It is shown that the HD-tree is continually better than the Prefix B-tree when the RAM is largely available.

In Figure 4.9, the number of I/Os are compared for different query lengths. It is shown that the HD-tree performs increasingly better than the Prefix B-tree as the

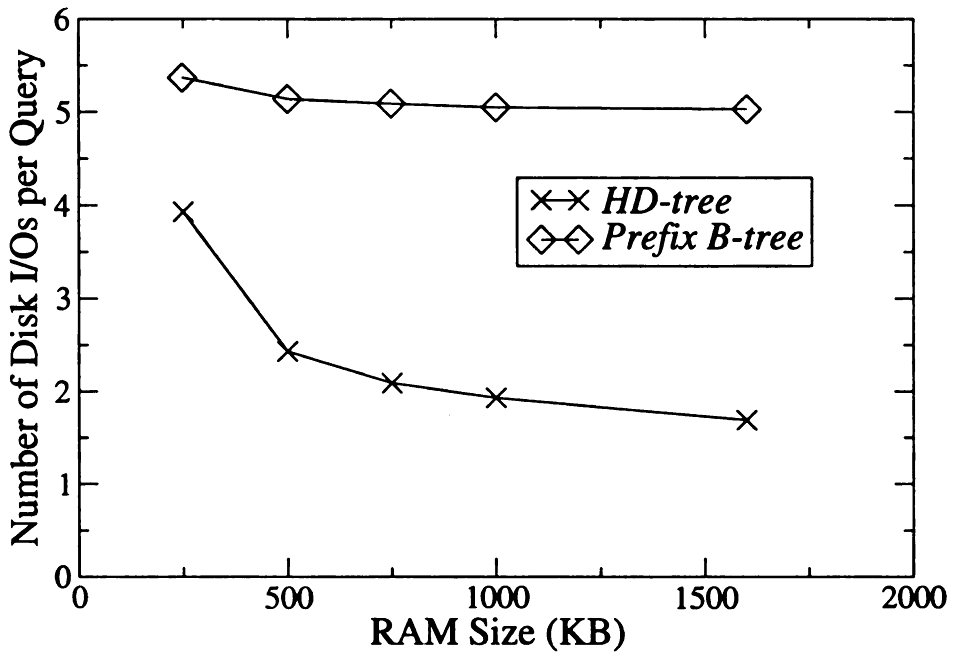


Figure 4.7: I/O comparison for different RAM sizes; average query string length is 8.

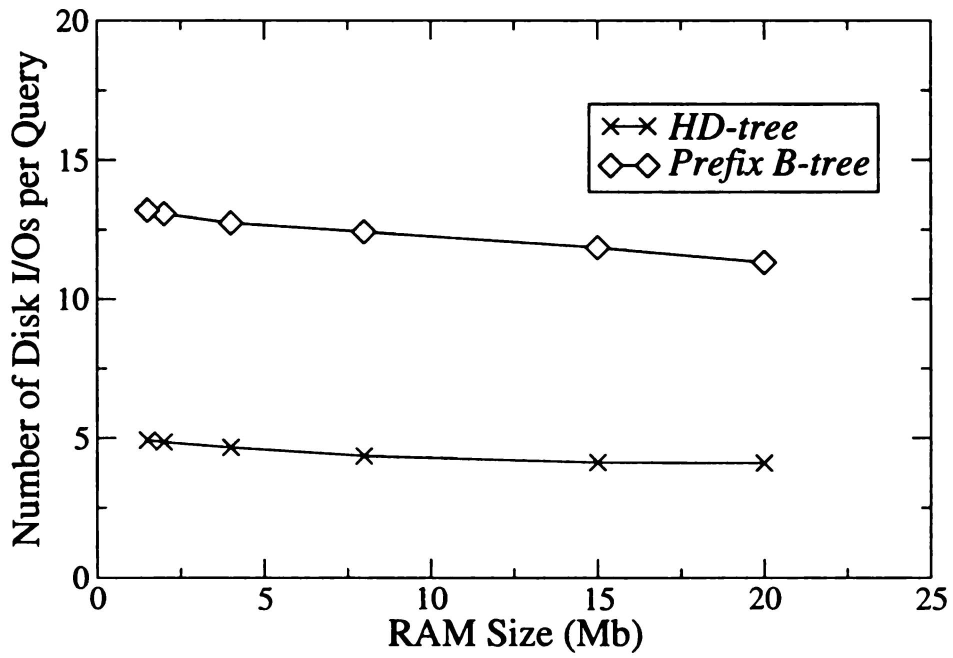


Figure 4.8: I/O comparison for different RAM sizes; average query string length is 6.

query string length increases. Since the Prefix B-tree uses the same amount of RAM as that of the HD-tree to cache internal nodes, it is concluded that the hybrid RAM/disk-based index structure (e.g., the HD-tree) is better than the disk-based structure combined with caching (e.g., the Prefix B-tree plus LRU caching), especially when queries are more distinctive.

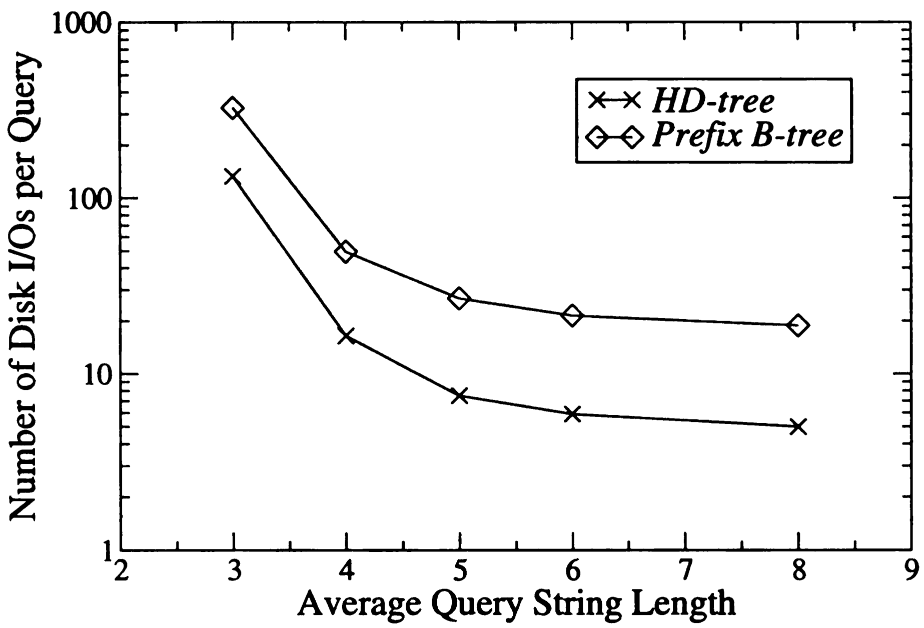


Figure 4.9: I/O comparison for different query lengths; y-axis is in Logarithmic scale.

Finally, the HD-tree is compared with the Prefix B-tree in terms of total running time including both the RAM processing time and the I/O time. The experiments are conducted in the same computing environment (a Linux PC with 512MB RAM and 1.8GHz Pentium 4 processor). Figure 4.10 shows the running time of the HD-tree and the Prefix B-tree for 1000 queries with different numbers of distinctive queries. It is noticed that the actual running time of the HD-tree is comparable to that of the Prefix B-tree even when the 1000 queries are the same. The reason is that since a large amount of RAM is available, the operating system provides LRU caching for the HD-tree as well. The HD-tree is shown to be increasingly faster than the Prefix B-tree as the number of distinctive queries

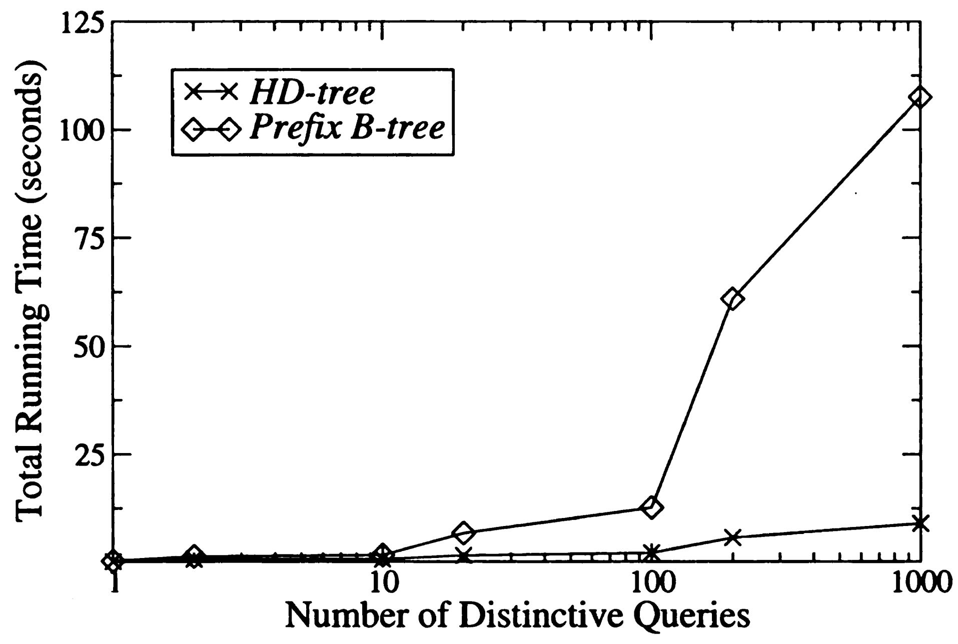


Figure 4.10: Running time comparison; average query string length is 6.

increases. For 1000 distinctive queries, the HD-tree is more than one magnitude faster than the Prefix B-tree.

4.5 Approximate String Matching Based on the Hamming Distance

Most disk-based index structures, such as the Prefix B-tree, cannot efficiently perform approximate string matching. However, since the HD-tree uses a trie-based structure for internal nodes, it has a great potential to perform well in approximate string matching. In this section, the search algorithms and experimental results for approximate matching based on the Hamming Distance are presented. More complex string matching issues and applications are discussed in Chapter 6 and 7.

4.5.1 Distance Measure

In order to perform approximate string matching, a distance measure is required. *Levenshtein distance* [4], which is the most popular distance measure, allows user to deletion, insertion, or substitution of a symbol in two matching strings. If different operations have different costs, or if the costs depend on the symbols involved, it is called *general edit distance.* The general edit distance is powerful enough for a wide range of applications, such as genomic sequence analysis. If all the operations cost one, it is called *simple edit distance* (or just *edit distance*), denoted as $edist(\alpha, \beta)$, where α and β are strings. Edit distance is the minimum number of operations to make two strings equal. For example, *edist*(*"string"*, *"stingy"*) = 2. If only substitution is allowed at cost one, it is known as Hamming distance [62], denoted as $hamming(\alpha, \beta)$. In order to compute the Hamming distance, the two strings must have the same length. For example, *hamming*(*"string"*, *"stingy"*) = 4 and *hamming*(*"string"*, *"strict"*) = 2.

4.5.2 Search Algorithm

ALGORITHM 5 **HD-HammingSearch(κ, l, max, δ)**

```
Input: (1) the query string $\kappa = k_1...k_n$; (2) the current level $l$;
(3) the maximum Hamming distance $max \geq 0$;
(4) the current internal node $\delta$,
where $L(\delta) = \{(a_1, P_1) \ ... \ (a_k, P_k) \ ... \ (a_m, P_m)\}$;
Output: query result
Method:
 1. for $i = 1$ to $m$ do
 2.   if $a_i \neq k_l$ and $max \geq 1$ and $P_i$ is an $IP$ then
 3.     return HD-HammingSearch(
            $\kappa$, $l + 1$, $max - 1$, $\delta_{P_i}$);
```

```
4.  else if $a_i == k_l$ and $P_i$ is an $IP$ then
5.    return HD-HammingSearch(
          $\kappa$, $l+1$, $max$, $\delta_{P_i}$);
6.  else if $P_i$ is an $SGLP$ and $a_i \neq k_l$
          and $max == 0$ then
7.    return $NULL$;
8.  else if $P_i$ is an $MGLP$ or $SGLP$
9.    read $\delta'_{P_i}$ from disk;
10.   retrieve suffix strings $\alpha$ in $\delta'_{P_i}$
        where $HDistance(\alpha, k_l...k_n) \leq max$
11.  end if;
12. end for;
13. return;
```

Algorithm 5 (HD-HammingSearch) starts from the root of an HD-tree, which is at level 1, and traverses down the tree as far as possible (steps 2-5). Note that the maximum Hamming distance decreases if a mismatch is found while going down the tree (step 3). Once a leaf node is encountered (steps 6-10), the search may continue within the leaf node (steps 9-10), or stop at step 7 if the condition in step 6 is satisfied.

4.5.3 Comparisons

One straightforward method to perform similarity searches based on the Hamming distance is to employ the linear scan. Assume that the database is stored sequentially on disk without fragments, which boosts its performance by a factor of 10. The performance of the linear scan is proportional to 10% of the database size. This benchmark is used in [39, 40]. As shown in Figure 4.11, for Hamming distances ≤ 1 and ≤ 2, the HD-tree with RAM as small as 50KB outperforms the 10% linear scan. For Hamming distance ≤3, the HD-tree outperforms the 10% linear scan when the RAM size is more than 150KB. Figure 4.12 shows that as the database size increases, the HD-tree is increasingly more efficient than the 10% linear scan.

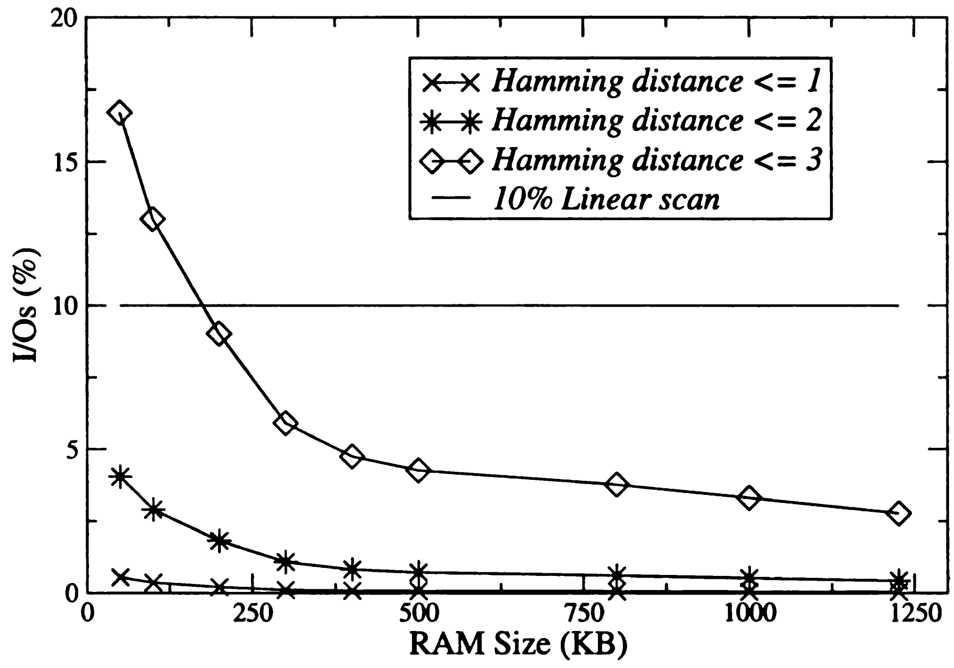


Figure 4.11: I/O comparison for Similarity searches as RAM size increase; the y-axis is the number of I/Os as the percentage of the total disk blocks occupied by the database.

Table 4.4 shows the performance comparison between the HD-tree and the M-tree, a disk-based metric tree for similarity searches [38]. It is shown that the number of I/Os using the HD-tree is much less than that using the M-tree. Since the M-tree is a pure disk based structure, it does not have any RAM requirement. If the RAM is available, the M-tree can cache the top level tree nodes to reduce the number of I/Os. However, the performance of the M-tree does not improve much as the RAM size increases. On the other hand, a small amount of RAM can boost the performance of the HD-tree dramatically. For Hamming distance ≤2, the M-tree takes an extra 560KB RAM to cache the second level of the tree, but the performance is improved less than 1%. However, 16KB RAM helps the HD-tree to reduce the number of I/Os from 640 to 214, i.e., the performance is improved by a

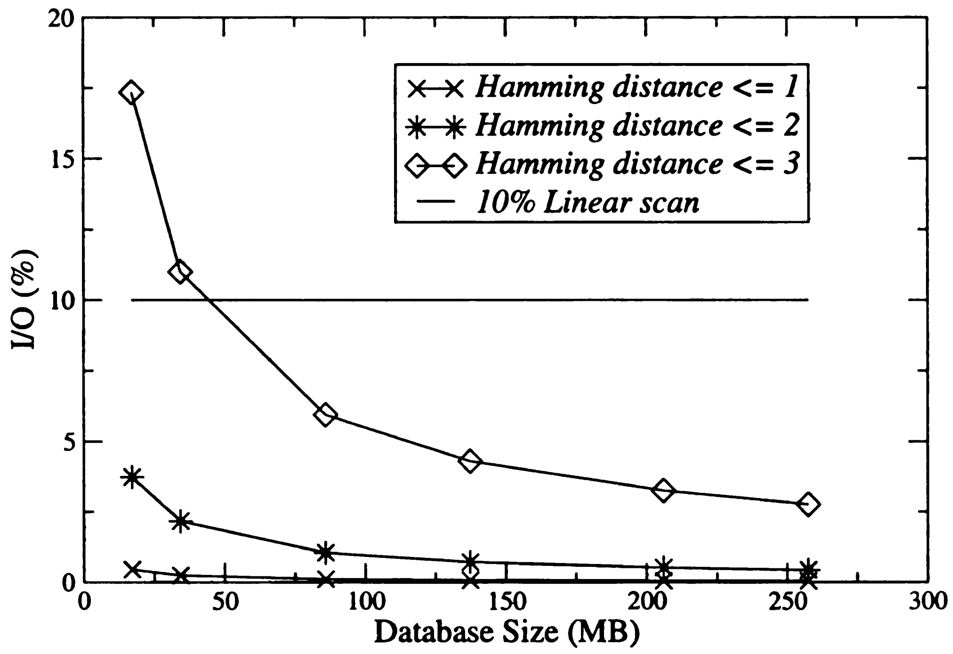


Figure 4.12: I/O comparison for Similarity searches as the Database size increases, the y-axis is the number of I/Os as the percentage of the total disk blocks occupied by the database.

factor of two.

The potential of the HD-tree is not limited in various types of queries discussed in this chapter. In fact, the HD-tree is more useful is approximate string matching based on general edit distance. The potential of the HD-tree is further explored in the second half of this dissertation.

Table 4.4: the HD-tree vs. the M-tree

$HD-tree$				$M-tree$			
ram	io			ram	io		
KB	≤ 1	≤ 2	≤ 3	KB	≤ 1	≤ 2	≤ 3
4	125	641	1684	0***	790	1420	2480
20	27	198	802	4**	789	1419	2479
32	18	143	616	564*	649	1279	2339

***: no cache; **: cache the root; *: cache top two levels

PART TWO

INDEXING AND SEARCHING GENOMIC

SEQUENCE DATABASES

Chapter 5: Genomic Sequence Analysis

The publication of the first working draft of the entire human genome sequence in February 2001 is considered to be a milestone of scientific research. The journal, Nature, describes this event as "Unravelling the three billion or so base pairs of our entire DNA has been compared to the landing on the moon, the splitting of the atom and even the invention of the wheel [63]." The broadly available genomic sequences provide a great opportunity to advance our understanding of the role of genetic factors in human health and disease, and to apply this insight rapidly to drug development, disease prevention, and genetic tests [64]. The massive amount of sequence information requires careful storage, organization, and analysis. Therefore, *bio-informatics*, which includes recording, analyzing, and searching of nucleotide and protein sequences, is an emerging and prominent research field, where biology, computer science, and information technology merge into a single discipline.

Since the inception of the Human Genome Project, which was completed in 2003, hundreds of other genome sequence projects on microbes, plants, and animals have been completed or are in progress [17]. Advances in molecular biology and sequencing equipment have allowed the increasingly rapid sequencing of large portions of the genomes. For example, a new technology developed at the 454 Life Sciences Corporation may achieve a 100-fold increase in DNA sequencing speed over current technology [65]. This technique could allow one person using one machine to easily sequence the 3 billion base pairs in the human genome within a hundred days. As the genomic sequence database continues to grow exponentially, the current popular linear-scan-based searching method will sooner or later become infeasible. Index-based approaches, such as the HD-tree, could become the choice of sequence searching in the future. In this chapter, existing techniques used in sequence analysis are presented, and the following two chapters will discuss indexing and searching genomic sequence databases using the HD-tree.

5.1 Introduction to Sequence Analysis

It is known that *DNA* (deoxyribonucleic acid) stores complete instructions for all the cellular functions of an organism. The primary structure of DNA is represented as strings using a four-letter alphabet, { A,C,G,T }, where each letter represents a *nucleotide*. *RNA* (polynucleotides) is a single-stranded molecule composed of nucleotide sequences that is similar to the double-stranded DNA. RNA helps to transfer information from DNA to the protein-forming system of the cell. Three-letter combinations of the nucleotide form *messenger RNA* (mRNA), which transcribes amino acids. Amino acids in turn can be combined to create proteins. Therefore, the structure of protein is represented as a string using a twenty-letter alphabet, each letter corresponds to one amino acid. The letters in nucleotide (DNA) or protein sequences are known as *base pairs* (or *residues*).

Most sequence databases consist of long strings of nucleotides and/or amino acids. Each sequence of nucleotides or amino acids represents a particular gene or protein (or section thereof). There are also databases which include taxonomic information, such as the structural and biochemical characteristics of organisms. Sequence databases provide scientists with a wealth of information. However, the power of a sequence database comes not from the collection of information, but in its analysis, which is the most pressing task in bio-informatics. Scientific research has shown that all genes share some common elements. A new sequence often has significant similarity to a sequence which is already known. Therefore, part of the information about the structure and function of the known sequence can be transfered to the new sequence. If two sequences are related, they are homologous, and one sequence is a homologue of another.

The common applications of sequence analysis include:

(1) finding genes in DNA sequences of various organisms;

(2) aligning similar proteins and generating phylogenetic trees;

(3) clustering protein sequences into families to develop of protein models; and

(4) developing methods to predict the structure and/or function of newly discovered proteins and structural RNA sequences [36].

Table 5.1: Two pairwise alignments to a fragment of human alpha globin: hba_human

(a)	hba_human	`GSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKL`
		`G+ +VK+HGKKV  A+++++AH+D++ +++++LS+LH  KL`
	sequence_a	`GNPKVKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKL`
(b)	hba_human	`GSAQVKGHGKKVADALTNAVAHVDDMPNALSALSD----LHAHKL`
		`GS+ + G +   +D L  ++ H+ D+  A +AL D    ++AH+`
	sequence_b	`GSGYLVGDSLTFVDLL--VAQHTADLLAANAALLDEFPQFKAHQE`

All these applications involve the most basic sequence analysis task: determining if two sequences are homologous. This is usually done by first aligning the sequences (or parts of them), and then deciding whether that alignment is more likely because the sequences are homologous, or just by chance. Table 5.1 shows an example of two pairwise alignments [36], where (a) implies a clear similarity, and (b) is most likely unrelated. In the central line of each alignment, identical positions are indicated with letters, and "related" positions with plus signs ("Related" pairs are those which have a positive score in a substitution matrix, which is discussed in Section 5.3.). Within aligned sequences, symbol '-' represents a gap due to insertion or deletion.

In the task of finding homologous sequences, the key issues are:

(1) the type of alignments to be considered;

(2) the scoring system used to rank alignments;

(3) the method used to find alignments; and

(4) the method used to evaluate the significance of alignments.

The focus of this dissertation is on issue (3), and other issues are resolved by existing techniques. In the following sections, these issues are discussed in detail.

5.2 Alignment Type

There are three types of sequence alignment: pairwise alignment, structural alignment, and multi-sequence alignment.

Pairwise alignment is the most commonly used alignment type. It is used to find a homologue of a gene (protein or DNA), or a gene-product in a database of known examples. Pairwise alignment can be done locally or globally. A *local alignment* finds related regions within sequences. They can consist of a sub-sequence within each sequence. For example, positions 22-32 of sequence X might be aligned with positions 64-74 of sequence Y. A global alignment between two sequences is an alignment in which all the characters in both sequences participate in the alignment. That is, both sequences have to be aligned from beginning to end. Global alignment is useful mostly for finding closely-related sequences. Local alignment is more flexible than global alignment and has the advantage to find related regions that appear in a different position in the two proteins. This is not possible with global alignment methods. Global and local alignment can refer to query and database sequences, respectively. For example, both query and database sequences are global, or global on query sequence and local on database sequence.

Structural alignment is mostly used for proteins. It is a form of alignment to establish equivalences between two or more protein structures based on their fold. Because protein structure is more conserved than protein sequence, structural alignments can be more reliable, especially when the sequences have diverged so much that simple sequence comparison cannot detect their similarity.

Multi-sequence alignment, as shown in Table 5.2 [66], is an extension of pairwise alignment. It incorporates more than two sequences into an alignment and aligns all of the sequences in a specified set. Multi-sequence alignments is used to identify related regions between sequences. It is also very useful in generating profile hidden hidden Markov models to search sequence databases for more distant homologues (see Chapter 7).

Table 5.2: An example of multi-sequence alignment: eight fragments from immunoglobulin sequences

VTISCTGSSSNIGAG-NHVKWYQQLPG
VTISCTGTSSNIGS--ITVNWYQQLPG
LRLSCSSSGFIFSS--YAMYWVRQAPG
LSLTCTVSGTSFDD--YYSTWVRQPPG
PEVTCVVVDVSHEDPQVKFNWYVDG--
ATLVCLISDFYPGA--VTVAWKADS--
AALGCLVKDYFPEP--VTVSWNSG---
VSLTCLVKGFYPSD--IAVEWESNG--

5.3 Scoring System

A key element in evaluating the quality of a pairwise sequence alignment is to assess whether a given alignment constitutes evidence for homology by a process of mutation. The basic mutational processes are *substitutions*, *insertions*, and *deletions*. Substitutions change residues in a sequence, while insertions and deletions add and remove residues, respectively. Insertions and deletions are together referred to as *gaps*. *Conservative substitutions* are generally defined as amino acid replacements that preserve the structure and functional properties of proteins. In order to distinguish homologous alignments from random alignments, a scoring scheme is needed, where a "*substitution matrix*" is often used to assign a score for aligning any possible pair of residues.

Early sequence analysis used a unitary scoring matrix, where all matches and mismatches are scored or penalized the same. The unitary scoring matrix is equivalent to the simple edit distance (see Section 4.5). Although unitary scoring matrix is sometimes used for DNA and RNA comparisons, it is not appropriate for protein alignments since it ignores the mutation and structure relations between different amino acids. Years of research in protein sequence analysis has shown that matches and mismatches among different amino acid pairs require different scores, and various substitution matrices have been developed to reflect these different scores [67, 35, 41].

Substitution matrices are generally presented as log-odds matrices, where each score in the matrix is the logarithm of an *odds ratio* [36]. The odds ratio is the ratio of the number of times residue 'A' is observed to replace residue 'B', divided by the number of times residue 'A' would be expected to replace residue 'B' randomly. A positive score in the matrix indicates a pair of residues that replace each other more often than expected by chance. This is evidence in favor of the aligned sequences being homologous. Meanwhile, negative scores in the matrix are evidence against the sequences being homologous.

The process of computing log-odds scores is shown as follows. Assume a pair of sequences, α and β, of lengths m and n, respectively. Let a_i be the ith symbol in α, and b_j be the jth symbol in β. These symbols come from some alphabet Λ (e.g., the twenty amino acids). Symbols from Λ are also denoted by low-case letters like a, b. For now, only ungapped (i.e., no gaps) global pairwise alignments are considered.

Assume two sequences are drawn from a random match model R, where any symbol a occurs independently with frequency f_a. Hence the probability of matching two sequences is:

$$P(\alpha,\beta|R) = \prod_i f_{a_i} \prod_j f_{b_j}. \tag{5.1}$$

In the alternative match model H, where the two sequences are homologous, aligned pairs of residues occur with a joint probability p_{ab}. The value p_{ab} can be thought of as the probability that the residues a and b have each independently been derived from some unknown original residue, c (c might be the same as a and/or b). The probability for the alignment is:

$$P(\alpha,\beta|H) = \prod_i p_{a_i b_i}. \tag{5.2}$$

The ratio of these two probabilities is the odds ratio:

$$\frac{P(\alpha,\beta|H)}{P(\alpha,\beta|R)} = \frac{\prod_i p_{a_i b_i}}{\prod_i f_{a_i} \prod_i f_{b_i}} = \prod_i \frac{p_{a_i b_i}}{f_{a_i} f_{b_i}}. \tag{5.3}$$

In order to arrive at an additive scoring system, the logarithm of this ratio, known as the log-odds ratio, is computed as:

$$S = \sum_i s(a_i, b_i), \tag{5.4}$$

where

$$s(a_i, b_i) = \log(\frac{p_{a_i b_i}}{f_{a_i} f_{b_i}}) \tag{5.5}$$

is the log likelihood ratio (i.e., score) of pair (a, b) occurring as an aligned pair, as opposed to an unaligned pair.

In the 1970's, Dayhoff pioneered this approach to derive the well known PAM (Point Accepted Mutations) family of substitution matrices [67]. In Dayhoff's method, all of the proteins are aligned in several families. Then, phylogenetic trees (a graphical means to depict the relationships of a group of organisms) are constructed for each family. Each phylogenetic tree is examined for the substitutions found on each branch. This leads to a table of the relative frequencies with which amino acids replace each other. This table is combined with the relative frequencies of each amino acids in the proteins studied to compute the PAM matrices. Each PAM matrix is associated with a number, which is the number of mutations per 100 amino acids in the sample protein data. For example, PAM30 assumes the occurrence of 30 point mutations per 100 amino acids (or 300 nucleotides) in the gene. An example of a substitution matrix, PAM30, is shown in Table 5.3. Using a substitution matrix, the total score assigned to an alignment is a sum of scores for each aligned pair of residues, plus scores for each gap. Such additive scoring schemes assume mutations occur independently at different positions in a sequence, which is a reasonable approximation for DNA and protein sequences [36].

Table 5.3: The PAM30 Substitution Matrix

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	6	-7	-4	-3	-6	-4	-2	-2	-7	-5	-6	-7	-5	-8	-2	0	-1	-13	-8	-2
R	-7	8	-6	-10	-8	-2	-9	-9	-2	-5	-8	0	-4	-9	-4	-3	-6	-2	-10	-8
N	-4	-6	8	2	-11	-3	-2	-3	0	-5	-7	-1	-9	-9	-6	0	-2	-8	-4	-8
D	-3	-10	2	8	-14	-2	2	-3	-4	-7	-12	-4	-11	-15	-8	-4	-5	-15	-11	-8
C	-6	-8	-11	-14	10	-14	-14	-9	-7	-6	-15	-14	-13	-13	-8	-3	-8	-15	-4	-6
Q	-4	-2	-3	-2	-14	8	1	-7	1	-8	-5	-3	-4	-13	-3	-5	-5	-13	-12	-7
E	-2	-9	-2	2	-14	1	8	-4	-5	-5	-9	-4	-7	-14	-5	-4	-6	-17	-8	-6
G	-2	-9	-3	-3	-9	-7	-4	6	-9	-11	-10	-7	-8	-9	-6	-2	-6	-15	-14	-5
H	-7	-2	0	-4	-7	1	-5	-9	9	-9	-6	-6	-10	-6	-4	-6	-7	-7	-3	-6
I	-5	-5	-5	-7	-6	-8	-5	-11	-9	8	-1	-6	-1	-2	-8	-7	-2	-14	-6	2
L	-6	-8	-7	-12	-15	-5	-9	-10	-6	-1	7	-8	1	-3	-7	-8	-7	-6	-7	-2
K	-7	0	-1	-4	-14	-3	-4	-7	-6	-6	-8	7	-2	-14	-6	-4	-3	-12	-9	-9
M	-5	-4	-9	-11	-13	-4	-7	-8	-10	-1	1	-2	11	-4	-8	-5	-4	-13	-11	-1
F	-8	-9	-9	-15	-13	-13	-14	-9	-6	-2	-3	-14	-4	9	-10	-6	-9	-4	2	-8
P	-2	-4	-6	-8	-8	-3	-5	-6	-4	-8	-7	-6	-8	-10	8	-2	-4	-14	-13	-6
S	0	-3	0	-4	-3	-5	-4	-2	-6	-7	-8	-4	-5	-6	-2	6	0	-5	-7	-6
T	-1	-6	-2	-5	-8	-5	-6	-6	-7	-2	-7	-3	-4	-9	-4	0	7	-13	-6	-3
W	-13	-2	-8	-15	-15	-13	-17	-15	-7	-14	-6	-12	-13	-4	-14	-5	-13	13	-5	-15
Y	-8	-10	-4	-11	-4	-12	-8	-14	-3	-6	-7	-9	-11	2	-13	-7	-6	-5	10	-7
V	-2	-8	-8	-8	-6	-7	-6	-5	-6	2	-2	-9	-1	-8	-6	-6	-3	-15	-7	7

There are several recent attempts to construct scoring matrices based on observed amino acid substitutions. The BLOSUM (BLOcks SUbstitution Matrix) family of matrices are one of these newly developed log-odds scoring matrices [41]. Unlike PAM matrices, which are developed from global alignments, BLOSUM matrices are based on local multi-sequence alignments of more distantly related sequences. These ungapped alignments are obtained from protein families called BLOCKS database [68]. The first stage of building the BLOSUM matrix is to cluster (group) sequences that are identical in more than $d\%$ of their amino acids. This is done to avoid bias of the result in favor of a certain protein. The matrix built from blocks with more than $d\%$ of similarity is called BLOSUMd. For example, the matrix built using sequences with more than 62% similarity is called BLOSUM62. The second stage is to compute the probability of amino acids in each column of the multiple alignments. Finally, the log odd ratio is calculated and rounded to the nearest integer.

In general, different substitution matrices are tailored to detect similarities among sequences that are diverged by differing degrees. It is shown that the BLOSUM62 matrix is among the best for detecting most weak protein similarities [41]. However, the BLOSUM series does not include any matrices suitable for short queries (e.g., < 50). Therefore, PAM matrices are recommended instead [69].

5.3.1 Gap Penalties

Sequence alignments involve gaps, which are caused by insertions or deletions of residues (see Table 5.1). Gaps are expected to be penalized. The cost associated with a gap of length l is given either by a *linear score*,

$$g(l) = -lo, \tag{5.6}$$

or an *affine score*,

$$g(l) = -o - (l-1)e, \tag{5.7}$$

where o is called the *gap-open* penalty, and e is called the *gap-extension* penalty. The gap-open penalty is usually greater than the gap-extension penalty. This allows long insertions and deletions to be penalized less than they would be by the linear gap cost. For example, for queries shorter than 35 residues, the recommended gap-open and gap-extension penalties are -9 and -1, respectively [69].

5.4 Alignment Algorithms

Given a scoring system, an algorithm is needed to find an optimal alignment for a pair of sequences. Dynamic programming, which solves a problem by caching sub-problem solutions rather than recomputing them, is essential to sequence analysis. Dynamic programming algorithms are guaranteed to find the optimal scoring alignment(s). Since log-odds ratio is used in the scoring scheme, the optimal alignment has the highest score. The following sections introduce two most commonly used dynamic programming algorithms in sequence analysis: the Needleman-Wunsch algorithm[70] and the Smith-Waterman algorithm [71].

5.4.1 Needleman-Wunsch Algorithm

The Needleman-Wunsch algorithm performs a global alignment on two sequences. It is the first instance of dynamic programming being applied to biological sequence comparison. An improved version is introduced in [72]. The idea of the Needleman-Wunsch algorithm is to build up an optimal alignment using previous optimal alignment for shorter sub-sequences. Given two sequences: $\alpha = a_1 ... a_m$ and $\beta = b_1 ... b_n$, assume an $(m+1) \times (n+1)$ table T, where $T(i, j)$ (the cell at ith row and jth column of T) is the score of the best alignment between the subsequence $a_1...a_i$ and $b_1...b_j$. $T(i,j)$ can be built recursively by first initializing $T(0,0) = 0$, then proceeding to fill the table from top left to bottom right (see Figure 5.1). The score of $T(i,j)$ is obtained as follows:

$$\begin{aligned} T(i,0) &= -io \\ T(0,j) &= -jo \\ T(i,j) &= max \begin{cases} T(i-1,j-1) + s(a_i,b_i) \\ T(i-1,j) - o \\ T(i,j-1) - o \end{cases}, \end{aligned} \tag{5.8}$$

where $s(a_i,b_j)$ is the substitution score of a_i and b_j, o is the the gap-open penalty, and the linear gap score is used. The values in the top row, $T(0,j)$, represent alignments of $b_1...b_j$ to all gaps in α. Likewise, values in the left-most column, $T(i,0)$, represent alignments of $a_1...a_i$ to all gaps in β. The value in the table cell, $T(m,n)$, is by definition the best score for aligning α and β. The Needleman-Wunsch algorithm takes $O(mn)$ time and $O(mn)$ memory. Since m and n are usually comparable, the algorithm is said to be $O(n^2)$.

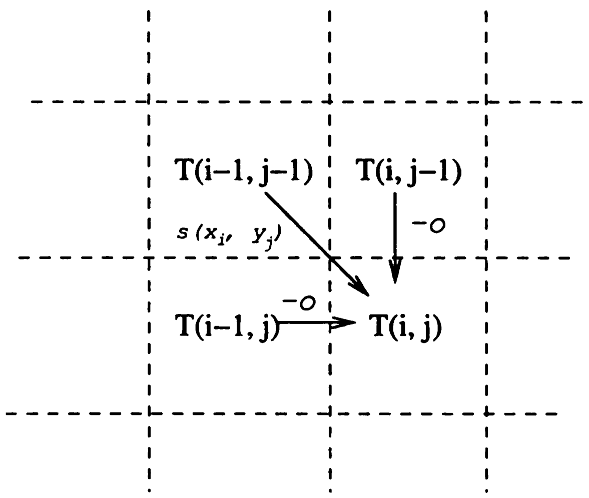


Figure 5.1: Compute a Cell in a Dynamic Programming Table

5.4.2 Smith-Waterman Algorithm

The Smith-Waterman algorithm is a well-known algorithm for performing local sequence alignment. It is similar to the Needleman-Wunsch algorithm, except a few changes which enable the Smith-Waterman algorithm to find optimal local alignments. First, since a local alignment may start at any position, $T(i,j)$ is allowed to take the value 0, if all other options have value less than 0. This corresponds to starting a new alignment. Hence, the score of $T(i,j)$ is obtained as follows:

$$\begin{aligned} T(i,0) &= 0 \\ T(0,j) &= 0 \\ T(i,j) &= max \begin{cases} 0 \\ T(i-1,j-1) + s(a_i,b_i) \\ T(i-1,j) - o \\ T(i,j-1) - o \end{cases} \end{aligned} \tag{5.9}$$

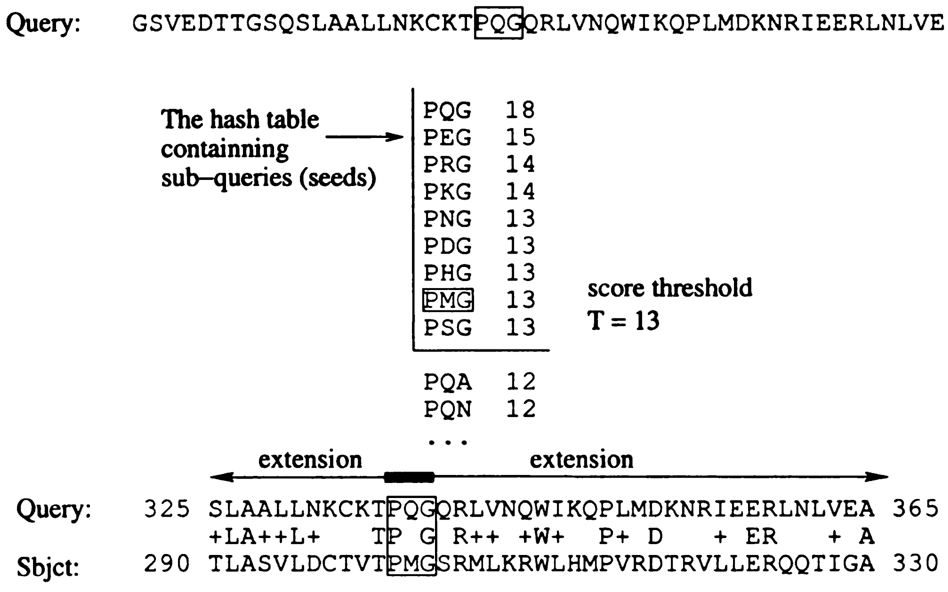
Note that the top row, $T(0,j)$, and the left-most column, $T(i,0)$, are filled with zeros, instead of $-jo$ and $-io$ in the Needleman-Wunsch algorithm.

The second change is that an alignment can end anywhere in the table. Therefore, instead of taking the value in $T(m,n)$ for the best score, the algorithm looks for the highest score of $T(i,j)$ over the whole table. In order for Smith-Waterman algorithm to work, the expected score for a random alignment must be negative. Otherwise, a long alignment between random unrelated sequences will have high scores due to their length, and the optimal local alignment would be likely to be masked by a longer but incorrect alignment. At the same time, there must be some $s(a,b)$ greater than 0. Otherwise, the algorithm cannot find any alignment, since zero is always chosen as the maximum score at each cell. The scoring matrices discussed in Section 5.3 satisfy these two conditions.

The Needleman-Wunsch and Smith-Waterman algorithms are guaranteed to find the optimal global and local alignments, respectively. However, they require $O(mn)$ time and space. If a large number of sequences are searched, time rapidly becomes an issue. Therefore, there are many attempts to develop heuristic alignment algorithms with the aim of increasing speed under limited sacrifice of sensitivity (i.e., the optimal alignment may be missed). Two of the best-known heuristic algorithms are BLAST [3, 42] and FASTA [73, 74].

5.4.3 BLAST

The BLAST (Basic Local Alignment Search Tool) is designed for finding high scoring local alignments between a query sequence and a target database. The basic idea of BLAST is that true alignments are very likely to contain a short segment of identities (as in DNA sequences), or very high scoring matches (as in protein sequences). These short segments can be used as "seeds" to find a good longer alignment. By keeping the seed segments short, it is possible to make a hash table for all possible seeds from the query sequence and use the hash table to find the matching segments in the target database.



Query: GSVEDTTGSQSLAALLNKCKTPQGQRLVNQWIKQPLMDKNRIEERLNLVE
The hash table containning sub–queries (seeds)
PQG 18
PEG 15
PRG 14
PKG 14
PNG 13
PDG 13
PHG 13
PMG 13
PSG 13
score threshold T = 13
PQA 12
PQN 12
...
extension
extension
Query: 325 SLAALLNKCKTPQGQRLVNQWIKQPLMDKNRIEERLNLVEA 365
+LA++L+ TP G R++ +W+ P+ D + ER + A
Sbjct: 290 TLASVLDCTVTPMGSRMLKRWLHMPVRDTRVLLERQQTIGA 330

Figure 5.2: The BLAST Search Algorithm

In Figure 5.2, assuming protein sequences, the BLAST algorithm looks for words (seeds) of length W (default = 3) that score at least T (default = 13) using a substitution matrix. Words in the database that score T or greater (called High Scoring Pair or HSP) are extended in both directions using an algorithm similar to the Smith-Waterman algorithm. The extension stops when the score of the extended alignment falls below a threshold. The extended alignment that meets certain criteria (see Section 5.5) will then be reported by BLAST.

5.4.4 FASTA

As shown in Figure 5.3, FASTA uses four steps to find local high scoring alignments:

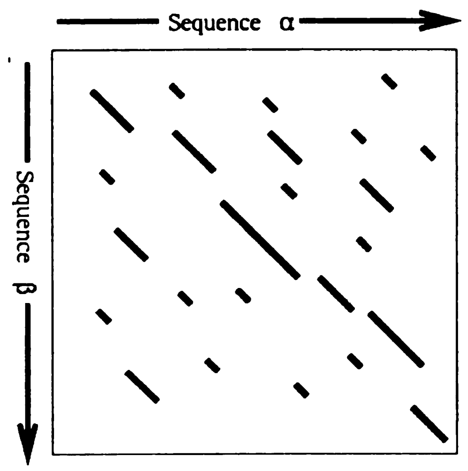
(1) Identify all exact matches of length k (k-tuples) or greater between the two sequences α and β. Speed is achieved by employing a hash table. For example, for proteins, if $k = 3$, there are 8000 (20^3) possible k-tuples. Each element of an array, A, of length 8000 is set to represent one of these k-tuples. Sequence α is scanned once and the location of each k-tuple in α is recorded in the corresponding element of A. Sequence β is then scanned. By referring to the locations of all k-tuples in α, matches that are common to α and β are identified.

(2) If two k-tuples are present on the same diagonal, then the difference between their starting position (offset) is also the same. The diagonals with the most significant number of matches are identified. The best diagonals are extended to find maximal scoring ungapped regions.

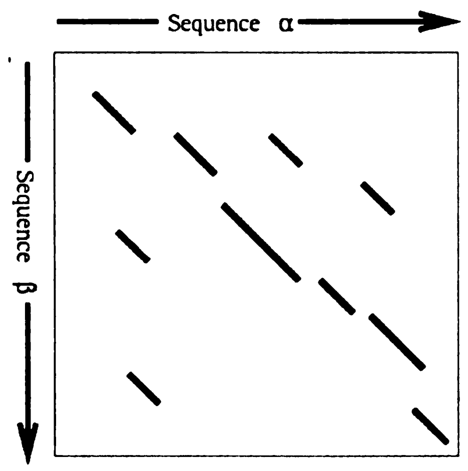
(3) If there are several initial regions above a user-defined score, then those that could form a longer alignment are joined, allowing for gaps and a score x is calculated with a penalty for each gap. These candidate alignments are

ranked by x.

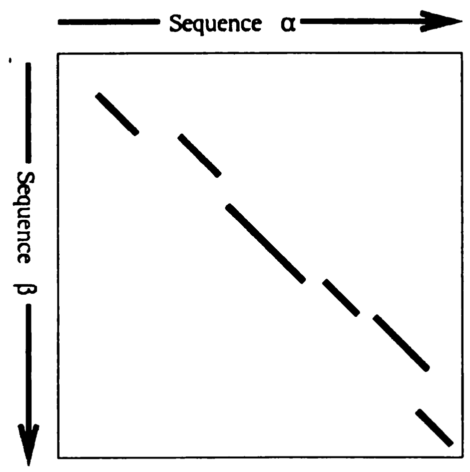
(4) The highest scoring candidate alignments are realigned using dynamic programming over a narrow band of the high scoring diagonal to produce an alignment with the final score.



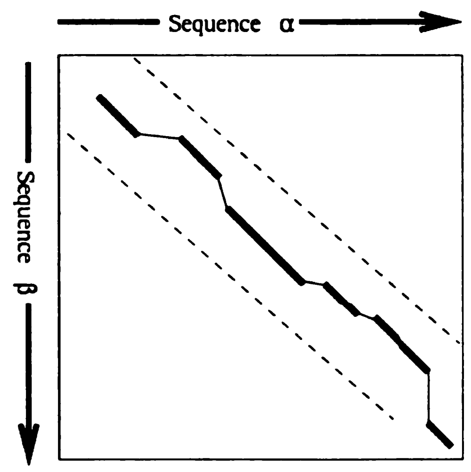
(a) Step 1: Find Identical Regions



(b) Step 2: Locate and Extend Diagonals



(c) Step 3: Join Regions



(d) Step 4: Apply Dynamic Programming

Figure 5.3: The FASTA Algorithm

5.5 Evaluating Alignments

Once an optimal alignment is found, the next task is to determine if the alignment is a biologically meaningful alignment (i.e., constitutes evidence for homology), or just the best alignment between two entirely unrelated sequences. The most common method is based on the statistical approach of calculating the chance of a match score greater than a randomly observed value (i.e., the sequences are unrelated).

Statistics for the scores of local alignments, which are the focus of this dissertation, are well understood [3]. It is known that the asymptotic distribution of the maximum, M_N, of a series of N independent random variables has the form

$$P(M_N \leq x) \simeq exp(-KNe^{\lambda(x-\mu)}) \tag{5.10}$$

for some constants K, λ [36, 75]. This form of distribution is called the *extreme value distribution* or EVD. For local ungapped alignment, the appropriate EVD is derived analytically in [3].

Given sufficiently large sequence lengths m (query) and n (database), the expected number of random matches with score at least S is given by the formula

$$E(S) = Kmne^{\lambda S}, \tag{5.11}$$

where λ is the positive root of

$$\sum_{a,b} f_a f_b e^{\lambda s(a,b)} = 1, \tag{5.12}$$

and K is a constant which is dependent only on f_a and $s(a,b)$. $E(S)$ is called the

E-value for the score S. The probability of a match having score greater than S is

$$P(\alpha > S) = 1 - e^{-E(S)}, \tag{5.13}$$

where $P(\alpha > S)$ is called *P-value.* In practice, E-value is usually used as the measurement to determine the significance of an alignment. The E-value is approximately how many alignments having the given score or higher would expect to be found in the given database by chance. Therefore, the smaller the E-value, the more significant (i.e., more likely to be homologous) is the alignment.

The statistics discussed above have a solid theoretical foundation only for ungapped local alignments. However, many computational experiments and some analytic results strongly suggest that the same theory applies as well to gapped alignments [76, 42, 77]. For ungapped alignments, the statistical parameters can be calculated from the substitution scores and the background residue frequencies of the sequences being compared. For gapped alignments, these parameters must be estimated from a large-scale comparison of random sequences. In [76], the values of λ and K are provided for a range of standard protein alignment scoring schemes, using a large amount of randomly generated sample data.

Chapter 6: Indexing and Searching Genomic Sequence Databases

As shown in Chapter 5, genomic sequence databases have seen rapid growth in recent years. There are three major sequence databases: the DNA DataBank of Japan (DDBJ), the European Molecular Biology Laboratory (EMBL), and the GenBank. All three databases participate in the International Nucleotide Sequence Database Collaboration (INSDC). They increase in size and exchange data on a daily basis.

6.1 Overview of GenBank

GenBank [13] is a genetic sequence database hosted by the National Center for Biotechnology Information (NCBI) at the National Institutes of Health (NIH). It is an annotated collection of publicly available DNA and protein sequences. GenBank contains millions of sequences, submitted from individual laboratories or large-scale sequencing projects. As shown in Figure 1.2 (see Chapter 1), GenBank grows at an exponential rate. In a recent press release (August 22, 2005), the INSDC announced that the DNA sequence database had exceeded 100 gigabases (i.e., approximately 100GB) [78]. In this dissertation, the entire GenBank non-redundant protein sequence database (as of April, 2004) is chosen as the testbed for the HD-tree. This database contains approximately 1.9 million sequences, and 661 million residues.

6.2 Existing Techniques

To support efficient searches in genomic sequence databases, many algorithms (systems) have been developed. Based on how the search is conducted, these systems can be divided into two categories: linear-scan-based and index-based. Linear-scan-based systems include FASTA and BLAST, which are described in Chapter 5. Index-based systems, such as BLAT [79] and CAFE [80, 81], perform a query using a pre-built index of the database. Although linear-scan-based systems are faster than index-based systems for smaller databases, as the size of the genomic sequence databases continually increases, index-based systems are more and more appealing.

Searching homologous regions in a genomic sequence database is usually conducted in two stages: the filtering stage and the alignment stage. The filtering stage detects candidate regions which are likely to be homologous. The alignment stage then examines these regions in detail and reports the regions which are indeed homologous according to some criteria (e.g., the E-value introduced in Section 5.5). Due to the unstructured nature of genomic sequences, words (i.e., sub-strings) of length L (also called q-grams) is often used for indexing and searching in the filtering stage [42, 82, 79, 74]. Words can be either *overlapping* (i.e., a fixed window size of L shifting from the beginning to the end of a sequence by one letter at a time) or non-overlapping (see Figure 6.1). Hits (positions of the words in the genomic sequence) are located by matching query words with database words. Each hit may produce a candidate region. Dynamic programming is used in the alignment stage to find the true homologous regions from the candidate regions [71, 83].

Sequence:	`ADEGBCDEA`
Overlapping words:	`ADE, DEG, EGB, GBC, BCD, CDE, DEA`
Non–overlapping words:	`ADE, GBC, DEA`

word length = 3

Figure 6.1: Overlapping versus Non-overlapping Words

Among index-based methods, BLAT builds an index in the RAM using non-overlapping words (length of 4 or 5 for protein sequences). BLAT is shown to be faster than BLAST; however, it is less sensitive. CAFE uses inverted files [7] to index overlapping words (length of 3 for protein sequences). Candidate regions are generated by searching these overlapping words. Heuristics, such as FRAMES [80], are used to reduce the number of candidate regions passed to the alignment stage.

Suffix trees (see Section 2.4) have been used in genomic sequence searches for small databases [2, 9, 84]. Suffix trees can be built in the RAM within $O(n)$ space and $O(n)$ time [57]. However, constructing an suffix tree larger than the available RAM is a challenging task [85, 9, 84]. Therefore, algorithms using suffix trees have not been successfully applied to large genomic sequence databases. In [84], a new way of creating suffix trees in excess of available RAM size for genomic sequences is proposed. Multiple passes over sequences are adopted and each pass processes a sub-range of suffixes. The suffix tree is used to search genomic sequences. However, the search dose not return the high scoring alignments, but the positions of the potential matches. It is concluded that using the suffix tree for genomic sequence searches can significantly reduce the computation compared to the standard dynamic programming methods [84].

In [86], another index structure, suffix sequoia, is proposed to index protein sequences. Suffix sequoia indexes overlapping words (length of 5 is used for 471M residues). Positions of these overlapping words are stored on disks. A data structure, called *bitmap*, combined with *offset files* are used to locate the position list of an overlapping word stored on disks (*position files*). The index size of suffix sequoia is relatively small, which is just over 4 bytes per residue of sequences. This is because only the positions of the overlapping words are stored. Sequence names are indexed, so that given a position, the corresponding sequence name is returned. The search of suffix sequoia returns the sequence names which contains a overlapping word that when aligned with an overlapping query word, may achieve a score above certain threshold. Therefore, the search results of suffix sequoia are not high scoring alignments, such as those returned by BLAST.

The HD-tree is a better approach than these existing index-based methods. First, the HD-tree uses longer overlapping words, which eliminate the filtering stage and reduce the search time for short queries. Second, the HD-tree is guaranteed to find the optimal alignments according to user-defined search criteria, since the HD-tree searches each overlapping word. Finally, the HD-tree is also applicable to more complex sequence searches, such as the profile hidden Markov model (see Chapter 7), which none of these index-based methods have attempted to do.

6.3 Creating the HD-tree Using the Sort-Merge Method

The success of the HD-tree in prefix searches (see Chapter 4) encourages the application of this structure in approximate string matching, especially in the area of genomic sequence analysis. Since the matching of a sequence may start at any position, overlapping words of length L are used to index genomic sequence databases [87, 82, 60]. This increases the size of string data by at least L times. The standard approach (Brute-Force) of creating an HD-tree is to insert one string at a time. Therefore, at least one disk access may be required. Due to the largeness of the GenBank protein sequence database, using the Brute-Force approach to create an index may result in an overwhelming number of disk accesses and an unreasonable large amount of time. For example, in one experiment, it takes approximately 30 minutes to index 375MB of overlapping words generated from protein sequences containing 15 million residues. In order to index the entire GenBank protein sequence database (661 million residues) within a reasonable time (e.g., several hours), heuristics must be developed to speed up the index process.

The HD-tree is an ordered tree; therefore, if the inserted strings are in sorted order, the tree can be built quickly from left to right. Based on this observation, the *Sort-Merge method* is developed to build the HD-tree for large string databases in the following steps:

1. The string database is divided into segments (e.g., 50MB) depending on the available RAM size. Each segment is loaded into the RAM, and an array of suffix pointers pointing to each position is created in the RAM.

2. An internal sorting process (e.g., quicksort [51]) is conducted so that these suffix pointers are sorted in the lexicographic order of the suffix strings they are pointing to. Each suffix string is associated with a 4-byte integer that is the position of this suffix string in the sequence database.

3. These sorted suffix strings are written into temporary files on disks. Only the first L characters of these suffix strings are reserved. For the suffix strings having the same first L characters, their positions are linked into a position list.

4. After all segments are processed, the temporary files containing the sorted strings are merged into one file using an external merge algorithm [51]. The the final file contains the sorted suffix strings for the entire database.

5. Create partial HD-trees (i.e, sub-trees) in the RAM from the sorted suffix strings. These partial HD-trees are written to disks when they exceed a user-defined threshold (e.g., 32MB).

6. Merge these partial HD-trees into the final HD-tree.

The process of creating an HD-tree index from the GenBank protein sequence database using the Sort-Merge method is shown in Figure 6.2.

Since the inserted strings are in sorted order, the standard splitting algorithm (see Algorithm 3 in Section 3.3.4) is modified to increase the storage utilization.

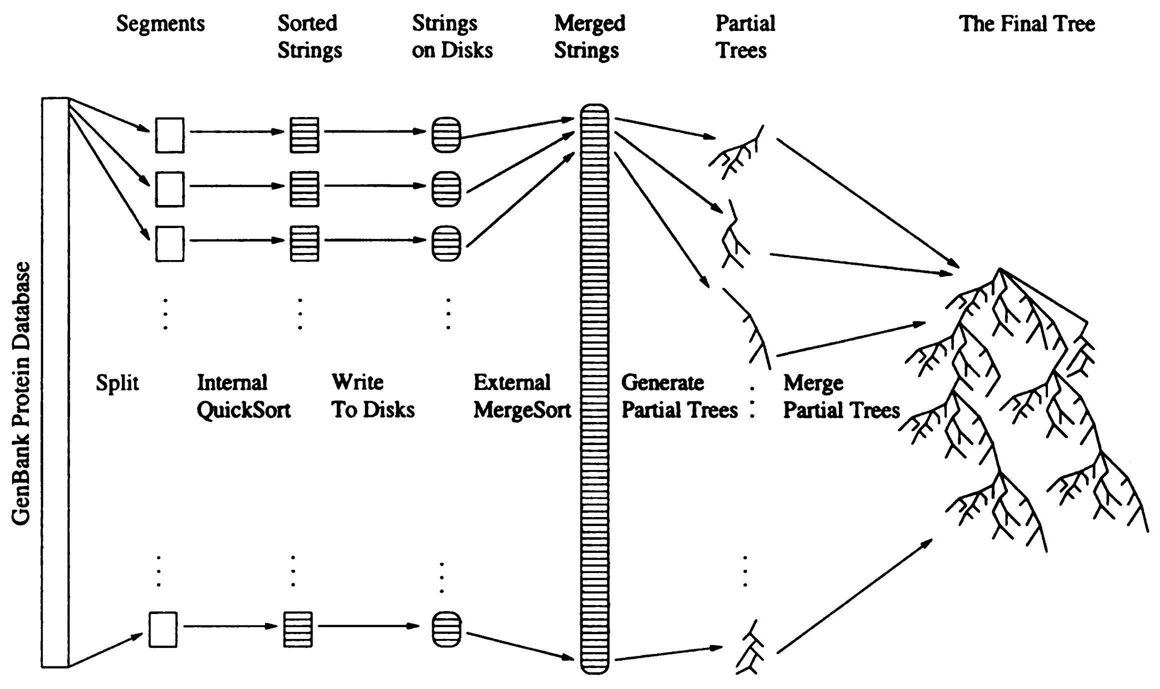


Figure 6.2: The Sort-Merge method of creating the HD-tree from the GenBank protein sequence database

The basic idea is to fill up a node as much as possible. If the current leaf node overflows, only the right-most unit is moved to the newly-created leaf node (see Figure 6.3b). Note that in the standard HD-tree splitting algorithm, units in the overflowing leaf node are evenly distributed as much as possible between the overflowing leaf node and the newly-created leaf node (see Figure 6.3a).

Using the Sort-Merge method, the construction time of the HD-tree is greatly reduced, and storage utilization is also improved. Figure 6.4 compares the construction time between the Brute-Force method and the Sort-Merge method. It is shown that the Brute-Force construction time is increased significantly when the number of indexed words reaches 15 million. This is because the size of the HD-tree containing 15 million words (approximately 375MB) exceeds the available RAM size in this example. When the HD-tree is smaller than the available RAM, leaf nodes are cached in the RAM by the operating system, therefore the actual number of disk accesses is relatively small. However, once the RAM cannot hold the entire

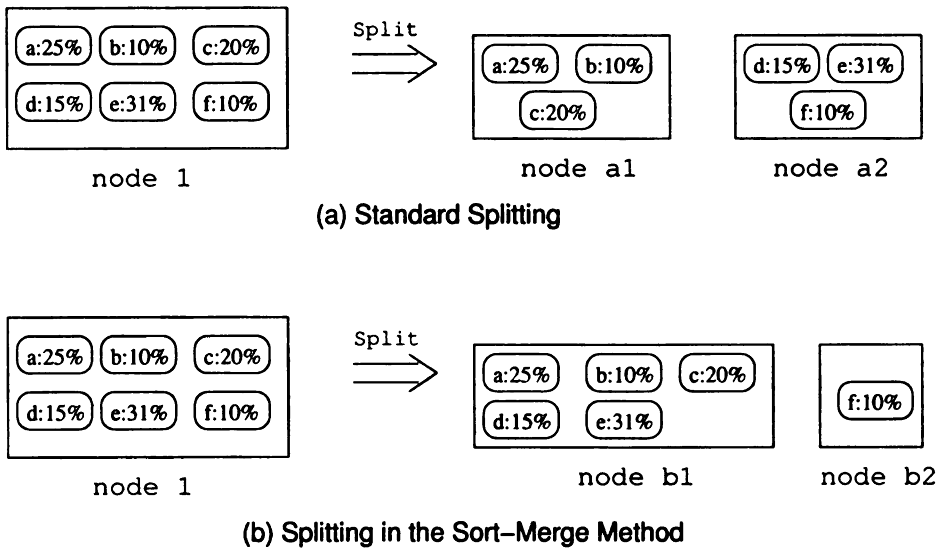


Figure 6.3: Split heuristic in the Sort-Merge method

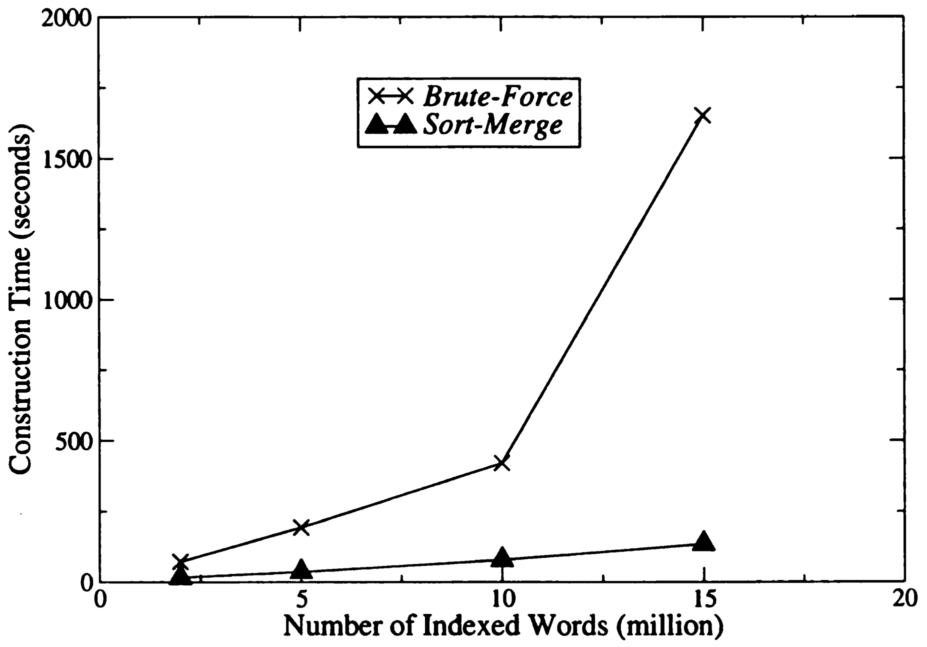


Figure 6.4: Improvement in Construction Time using the Sort-Merge Method. Overlapping words of length 20 are used.

tree, some leaves has to be stored on disks. Hence, each subsequent insertion may require at least one disk access (i.e., the total number of disk access may be proportional to the number of overlapping words), which increases the construction time significantly. On the other hand, using the Sort-Merge method, the number of disk accesses is roughly proportional to the sizes of the overlapping words (the sorting process) and the HD-tree (the merging process). The Sort-Merge method is almost 10 times faster than the Brute-Force method for large string databases such as the GenBank protein sequence database. As shown in Figure 6.5, storage utilization of the Sort-Merge method is consistently over 85%, which is improved significantly over the Brute-Force method. This improvement is due to the splitting heuristic which fills up a leaf node as much as possible.

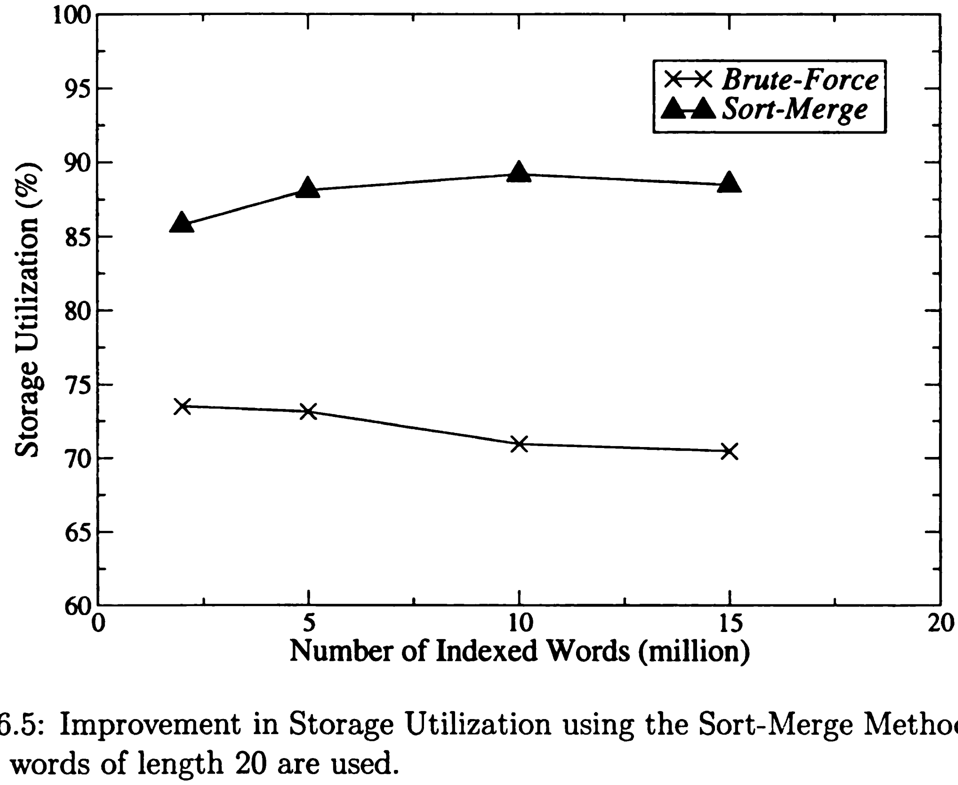


Figure 6.5: Improvement in Storage Utilization using the Sort-Merge Method. Overlapping words of length 20 are used.

Typically, some strings in a leaf node share a common prefix. Therefore, another heuristic is to use *difference-encoding* [55] to store strings in leaf nodes. Difference-encoding stores strings in lexicographic order. It uses a one-byte integer to indicate the number of prefix characters that are repeated from the previous string (i.e., the common prefix). Figure 6.6 shows an example of this encoding scheme. The difference-encoding can reduce not only the storage requirement of leaf nodes, but also the number of characters to be compared when searching strings stored in a leaf node sequentially, since the length of the common prefix is stored.

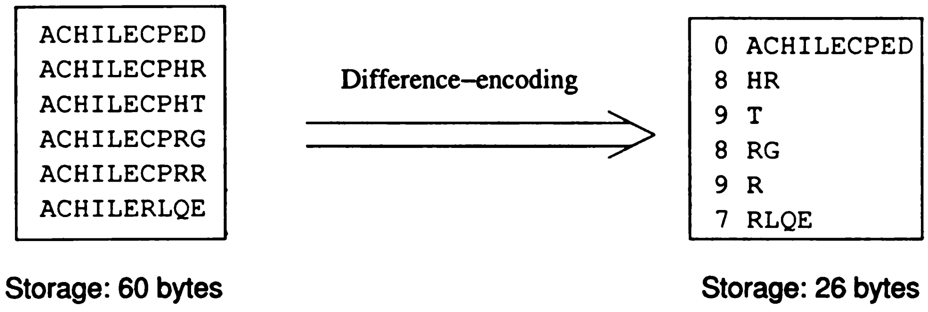


Figure 6.6: An example of the Difference-encoding

6.3.1 Discussion

The Sort-Merge method of the HD-tree uses similar construction strategy (i.e., build partial trees) as that in [84] (Hunt's method) to build large suffix index. However, the two methods have the following differences: First, the Hunt's method is to create a large persistent suffix tree exceeds the available RAM size. The suffix tree is stored on disks as a disk-image of the RAM. Therefore, it is not considered as a disk-based structure. The Sort-Merge method is to create an HD-tree index for overlapping words (i.e., prefixes of length L for all suffix keys). The HD-tree is a hybrid RAM/disk-based data structure which uses disk blocks as leaf nodes. Second, the Hunt's method uses multiple passes to create partial suffix trees, while the Sort-Merge method divides the database sequences into smaller segments to create partial HD-trees.

Since the implementation of the Hunt's method is not available, the experimental results provided in the paper are shown in Table 6.1. Although the machine and the database are not the same, the construction time of the Hunt's method seems to be longer than that of the HD-tree, and the index size seems to be larger. This is partly because the Hunt's method creates the complete suffix tree, which is space and time consuming. However, as the overlapping word length increases, the construction time and space requirement of the Sort-Merge method may exceed that of the Hunt's method.

Table 6.1: Hunt's Suffix Tree versus the HD-tree

	Database	Hardware	RAM	Space	Time
Hunt's Suffix Tree	268M DNA	Enterprise 450 SUN	2GB	19GB	13.5h
HD-tree, suffixes of length 20	661M Protein	Pentium 1.8G PC	516MB	10.6GB	3.41h

The HD-tree contains RAM-index and disk-index. The RAM-index of the HD-tree is the top levels of the suffix tree in the Hunt's method. The disk-index of the HD-tree contains the suffixes of the overlapping words. These suffixes enable the HD-tree to quickly find answers for short queries (i.e., less than the length of overlapping words). Using Hunt's method, however, if a query cannot be answered in the suffix tree (which happens often), in order to find an answer (i.e., a high scoring alignment), the search must follow the pointer stored in leaf nodes to the original database sequences. This operation is very costly if the sequence database is not reside in the RAM, since at least one random disk access is require for a possible match. This is likely to be the reason why the Hunt's method does not return high scoring alignments, but only the positions which may be a potential match.

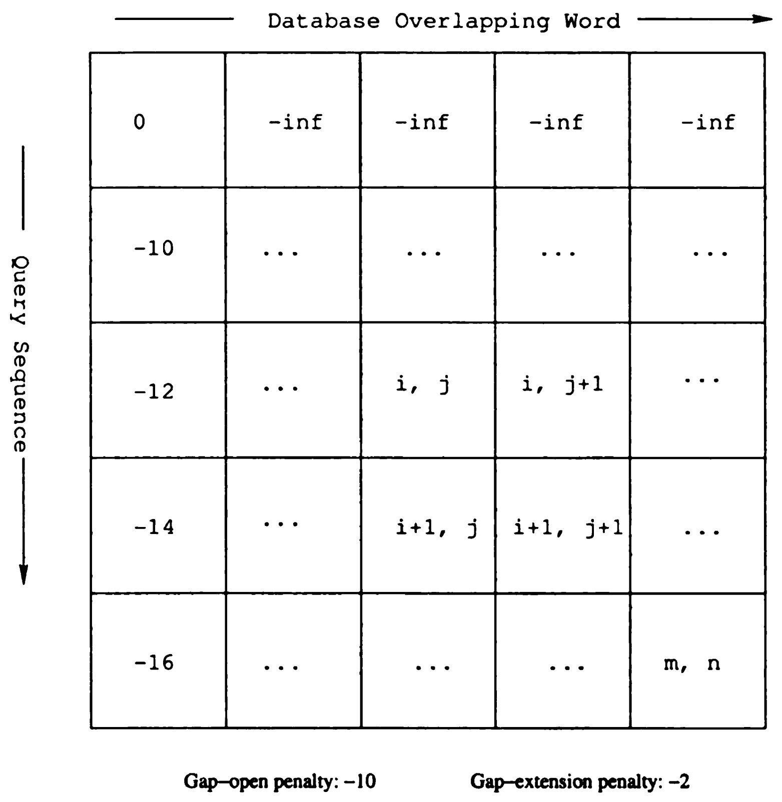
6.4 HD-tree Search Algorithm

Using suffix trees and suffix arrays to perform approximate string matching has been studied for years [88, 57, 2, 85]. Most of these researches are focused on simple edit distance (also called k-difference problem), where a constant cost (e.g., unitary cost) is used for insertion, deletion, and substitution. Since genomic sequence searches use substitution matrices and affine gap cost model (see Section 5.3), algorithms based on simple edit distance cannot be directly applied.

The HD-tree indexes overlapping words of length L. Therefore, it can be viewed as a suffix tree with maximum hight of L. The algorithm of sequence search in the HD-tree is an extension of the dynamic programming algorithm applied to a suffix tree for approximate string matching based on the edit distance [2, 57, 84]. Substitution matrices and affine gap cost model are integrated in the algorithm. Imagine the need to obtain the best matching score, $score(\alpha, \beta)$, between sequences $\alpha = a_1 ... a_m$ and $\beta = b_1 ... b_n$. A table of $(m+1) \times (n+1)$ cells is created. Cell $C_{i,j}$ (i.e., ith row and jth column) contains the value $c_{i,j}$, which is the maximum achievable score by matching $a_1 ... a_i$ and $b_1 ... b_j$. The following equations define the calculation of $c_{i,j}$ (i.e., the dynamic programming):

$$
\begin{aligned}
o &= \text{the gap-open penalty (negative value)} \\
e &= \text{the gap-extension penalty (negative value)} \\
m_{a,b} &= \text{the substitution score for symbol } a \text{ and } b \\
gap(C_{i,j}, C_{i',j'}) &= \begin{cases} o \text{ if from cell } C_{i,j} \text{ to cell } C_{i',j'} \text{ opens a gap} \\ e \text{ if from cell } C_{i,j} \text{ to cell } C_{i',j'} \text{ extends a gap} \end{cases} \\
c_{0,0} &= 0 \qquad\qquad (6.1) \\
c_{i,0} &= o + e * i, \ 0 < i \le m \\
c_{0,j} &= -inf, \ 0 < j \le n \\
c_{i,j} &= \max \begin{cases} c_{i-1,j} + gap(C_{i-1,j}, C_{i,j}), \\ c_{i,j-1} + gap(C_{i,j-1}, C_{i,j}), \\ c_{i-1,j-1} + m_{a_i,b_j} \end{cases}
\end{aligned}
$$

In the above equations, $c_{m,n}$ is the $score(\alpha, \beta)$, and $c_{i,0}$ is the score between $a_1 ... a_i$ and empty string. $c_{0,j}$ is initialized to $-inf$ (i.e., negative infinity), so that gaps at the beginning of an overlapping word are not allowed. This is because the score of an alignment with beginning gaps is always less than that of the same alignment without the beginning gaps. Since the HD-tree indexes all overlapping words, the alignment with beginning gaps is a redundant result of the same alignment without the beginning gaps. Figure 6.7 shows a dynamic programming table in the HD-tree.


Database Overlapping Word
Query Sequence
Gap-open penalty: -10
Gap-extension penalty: -2

0	-inf	-inf	-inf	-inf
-10	...	...	...	...
-12	...	i, j	i, j+1	...
-14	...	i+1, j	i+1, j+1	...
-16	...	...	...	m, n

Figure 6.7: An Example of a Dynamic Programming Table Used in the HD-tree.

Recall that in the HD-tree, each tree node at level x corresponds to an id-string, $a_1 ... a_x$, which is the concatenation of the labels along the path from the root to the tree node (see Section 3.1). Since HD-tree indexes overlapping words, every potential match can be found by traversing the HD-tree from root to leaves. Therefore, query answers can be found by starting at the root and following every branch, until a match is found or the id-string of the current tree node cannot be the prefix of a possible match (i.e., any string under this sub-tree is not a answer, and the sub-tree is abandoned for the search).

Given a query string, $q = q_1 ... q_k$, and a minimum matching score, S, an algorithm is designed to determine the matching score between q and string α', using the dynamic programming described in Equation 6.1. In order to traverse the HD-tree and abandon a sub-tree that does not contain query answers, the algorithm must be able to meet the following criteria: (a) considering the string α' incrementally, (b) determining when $score(q, \alpha') \geq S$, and (c) determining when $score(q, \alpha'\beta') < S$ for any string, β' (see Section 1.2 for string notation).

Assume a table of $(k+1) \times (h+1)$ is created, where k is the query length, and h is the height of the HD-tree. Starting from the root, the algorithm descends recursively by every branch of the HD-tree. When descending by a branch labeled by letter a, the algorithm appends a to current string, α', and compute a table column corresponding to a (see Figure 6.8). There are three possibilities for a given α':

(1) If the $score(q, \alpha') \geq S$, all the leaves of the current sub-tree are reported as answers.

(2) If $score(q, \alpha'\beta') < S$ for any string, β', the sub-tree corresponding to α' is abandoned immediately.

(3) Otherwise, the algorithm continues recursively descending through the HD-tree.

In order to fulfill criterion (c), the heuristic to determine whether $score(q, \alpha'\beta') < S$ for any string, β', is as follows: assume the cell $C_{i,|\alpha'|}$ contains the maximum score in column $|\alpha'|$ (i.e., cells $C_{0,|\alpha'|}$ to $C_{k,|\alpha'|}$), then the maximum achievable score equals to $score(q, \alpha' q_{i+1} ... q_k)$ (i.e., the rest of matching is an exact match). If $score(q, \alpha' q_{i+1} ... q_k) < S$, then for any string, β', $score(q, \alpha'\beta') < S$, where

$$score(q, \alpha' q_{i+1}...q_k) = score(q_1...q_i, \alpha') + score(q_{i+1}...q_k, q_{i+1}...q_k). \qquad (6.2)$$

The HD-tree sequence search algorithm is shown in Algorithm 6. Array $msc[1...k]$ is precomputed, where $msc[i] = score(q_{i+1}...q_k, q_{i+1}...q_k)$. String $\alpha = a_1...a_x$ is the id-string of the HD-tree node, N, at level x. $c_{i,j}$ is the value in the ith row and jth column of the table C. In the algorithm, the computation of line 1 is based on the dynamic programming described in Equation 6.1. Lines 2 to 3 correspond to criterion (b), and lines 5 to 8 correspond to criterion (c). The algorithm is recursive and the number of table columns is at most the height of the HD-tree.

ALGORITHM 6 **HD-SeqSearch(Table C, HDtreeNode N, String $\alpha = a_1...a_x$)**

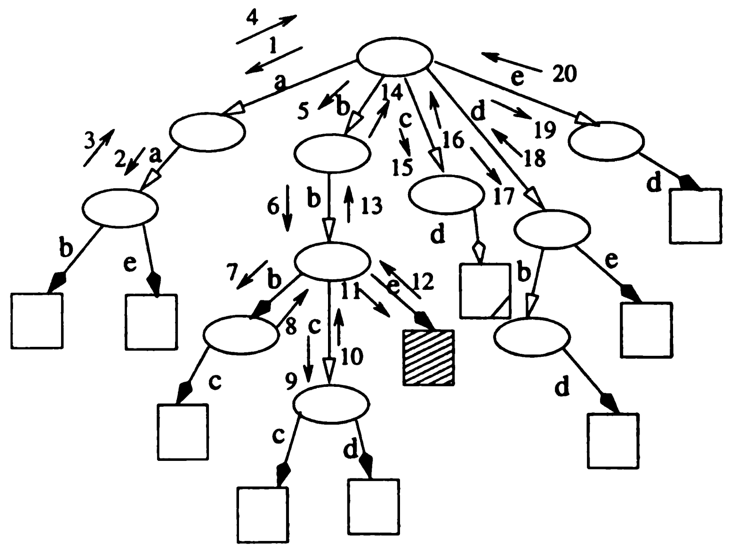
```
Input: (1) a query string $q = q_1...q_k$ (2) the minimum score $S$
Output: matched strings
Method:
 1. Compute column $x$ of the table $C$ using $a_x$, the resulting table is $C'$;
 2. if $c_{k,x} \geq S$ then
 3.     Report all the leaves below the tree node $N$;
 4. else
 5.     $c_{i,x} = max(c_{0,x}...c_{k,x})$;
 6.     if ($c_{i,x} + msc[i] < S$) then
 7.         return;
 8.     end if
 9. else
10.     for each child $N'$ labeled $b$
11.         if($N'$ is a leaf node);
```

```
12.           search strings in $N'$;
13.       else
14.           HD-SeqSearch($C'$, $N'$, $\alpha b$);
15.       end if
16.   end for
17. end if
```

An example of traversing the HD-tree and computing the dynamic programming table for a sequence search is shown in Figure 6.8. Compared with the standard dynamic programming method (which computes n columns), the HD-tree significantly reduces the computation. For example, in Figure 6.8, only 10 columns are computed (this does include the columns computed within the shaded leaf node). This is because: (1) since the HD-tree indexes overlapping words, computing one table column at a tree node is equivalent to computing multiple table columns in a standard dynamic programming; (2) a sub-tree may be abandoned if it is impossible to satisfy the search criterion in the sub-tree (lines 6 and 7 in Algorithm 6).

6.5 Comparisons with BLAST

BLAST [3, 42] is the most commonly used tool in bio-informatics. It is used by GenBank to provide genomic sequence searches. Although there are more sensitive algorithms (such as FASTA [73, 74]), BLAST is much faster and, in practice, has proved sensitive enough to detect the moderate sequence similarities that imply homology. In our experiments, the local BLAST (version 2.2.6 for Linux, downloaded from the NCBI web site) is used for searching the GenBank protein sequence database with the following parameters: word size is 3, scoring matrix is PAM30, E-value is 100, gap-open penalty is -9, and gap extension penalty is -1 (see Section 5.3 for the meaning of these parameters). These parameters are



Substitution Matrix:

	a	b	c	d	e
a	10	9	-5	-5	-5
b		15	-5	1	-5
c			10	-5	-5
d				10	-5
e					10

Gap open: −10
Gap extension: −2

Query: bbee Closeness: 80%

Maximum Score = 15 + 15 + 10 + 10 = 50
Minimum Score = 50 * 80% = 40
Array msc = {35, 20, 10, 0}

$\xrightarrow{1}$

		a
	0	−inf
b	−10	9
b	−12	−1
e	−14	−3
e	−16	−5

9 + 35 > 40
continue

$\xrightarrow{2}$

		a	a
	0	−inf	−inf
b	−10	9	−1
b	−12	−1	18
e	−14	−3	8
e	−16	−5	6

18 + 20 < 40
return

$\xrightarrow{5}$

		b
	0	−inf
b	−10	15
b	−12	5
e	−14	3
e	−16	1

15 + 35 > 40
continue

$\xrightarrow{6}$

		b	b
	0	−inf	−inf
b	−10	15	5
b	−12	5	30
e	−14	3	20
e	−16	1	18

30 + 20 > 40
continue

$\xrightarrow{7}$

		b	b	b
	0	−inf	−inf	−inf
b	−10	15	5	3
b	−12	5	30	20
e	−14	3	20	25
e	−16	1	18	15

25 + 10 < 40
return

$\xrightarrow{9}$

		b	b	c
	0	−inf	−inf	−inf
b	−10	15	5	3
b	−12	5	30	20
e	−14	3	20	25
e	−16	1	18	15

25 + 10 < 40
return

$\xrightarrow{11}$

		b	b	e
	0	−inf	−inf	−inf
b	−10	15	5	3
b	−12	5	30	20
e	−14	3	20	40
e	−16	1	18	30

40 + 10 > 40
search leaf (shaded)

$\xrightarrow{15}$

		c
	0	−inf
b	−10	−5
b	−12	−15
e	−14	−17
e	−16	−19

−5 + 35 < 40
return

$\xrightarrow{17}$

		d
	0	−inf
b	−10	1
b	−12	−11
e	−14	−13
e	−16	−15

1 + 35 < 40
return

$\xrightarrow{19}$

		e
	0	−inf
b	−10	−5
b	−12	−15
e	−14	−17
e	−16	−19

−5 + 35 < 40
return

Figure 6.8: An example of a genomic sequence search using the HD-tree. Arrow shows the traversing path. Each downward arrow computes one column (the right-most) of the table, and upward arrows do not compute matrix. "continue", "return", and "search leaf" correspond to lines 14, 7, and 12 in Algorithm 6, respectively.

recommended by BLAST for searching queries shorter than 35 residues [69].

Although sequence analysis involves long queries, (e.g., hundreds or thousands of residues), this dissertation focuses on short queries, such as some insulin protein sequences. Short queries are also useful in primer and probe design packages, where the intent is not to find related genes or gene segments, but to find regions of sequence that might cause cross-priming or cross-hybridization. Therefore, detecting even relatively short homologous regions is useful. Another application of short queries is to find motifs (i.e., recurring patterns) in DNA and protein sequences [89, 90] and use motifs to find active regions of proteins [91]. Motifs are usually short, Figure 6.9 shows the distribution of motif lengths using a motif extraction algorithm, MEX, from 7000 enzyme sequences [91].

It is realized that the index methods in [84, 86] are related to the HD-tree. However, they do not provide valid query results (see Section 6.2). Therefore, the real sequence search tool, BLAST, is chosen as the comparison target.

The performance of the HD-tree is compared with that of BLAST in the context of the quality of the query result and the query processing time. For fair comparison of query time, all experiments are conducted on a Linux PC with 512MB RAM and 1.8GHz Pentium 4 processor. In the experiments, the entire query is aligned (i.e., global on query) to each possible position of database sequences (i.e., local on database sequences). Queries of length 10 (2 sequences), 14 (8 sequences), and 18 (8 sequences) are selected from the GenBank protein sequence database. Each query has the word 'insulin" in their annotation. It is noted that these selected queries may not be insulin sequence, but may be insulin-related or insulin-like protein sequences. Only these real queries are used (synthetic queries are used in Chapter 7), so that query quality can be measured as follows. Since the ideal answer set is unknown, the quality of the query result is measured by the number of returning sequences containing the word "insulin" in their annotations.

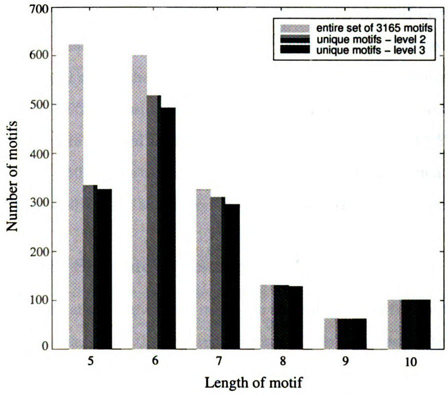


Figure 6.9: Motif length distribution using MEX from 7000 enzyme sequences

This approximation is based on a reasonable assumption that sequences having "insulin" in their annotations would most likely to be homologous.

6.5.1 Closeness of HD-tree Queries

The HD-tree uses *closeness* as a query parameter. The closeness is a percentage value to measure the similarity between two sequences. The usage of the closeness is as follows. Given a query sequence and a scoring matrix, an exact match achieves the maximum score. The maximum score multiplied by the closeness is the minimum score that each query answer must achieve, so that these query answers will satisfy the similarity defined by the closeness. For example, for a given query "ADEGBC", using the PAM30 substitution matrix (see Section 5.3), the maximum score achieved by an exact match is 50 (i.e., *score*("*ADEGBC*" , "*ADEGBC*") = 50). If the closeness is 80%, only those alignments that achieve the score of 40 ($50 * 80\% = 40$) and above are considered as the query answers.

6.5.2 Index Size and Construction Time

Since the database is given, the index size and construction time of the HD-tree is affected by the length of the overlapping words. Table 6.2 presents the index sizes and construction times of the HD-tree for different lengths of overlapping words. It is shown that as word length increases, the construction time and the index size increases accordingly.

For the given GenBank protein sequence database, BLAST takes 26 minutes to process the raw data. It creates 7 files and the total size is approximately 1.45GB. The processing time of BLAST is not to build index, but to re-organize the raw data so that BLAST can search the database more efficiently. On the other hand, for the same database using overlapping word length of 20, it takes the HD-tree approximately 2.43 hours to sort the overlapping words, and one hour to create the

Table 6.2: Index Size, construction time and storage utilization of the HD-tree for the GenBank protein sequence database

Word Length	RAM (MB)	Disk (GB)	SortTime (hour)	MergeTime (hour)	TotalTime (hour)	ASU* (%)
12	10.29	6.60	1.73	0.62	2.35	87.83
15	12.54	8.02	2.00	0.73	2.73	87.40
18	14.85	9.47	2.21	0.88	3.10	87.03
20	16.42	10.46	2.43	0.98	3.41	86.79

ASU: average storage utilization

tree from these sorted overlapping words. Although the construction time of the HD-tree is about three hours longer than that of BLAST, if a large amount of queries are conducted after the index is built, the amortized construction time is not significant.

The length of overlapping words affects the query performance of the HD-tree. If the query length is close to or longer than that of a overlapping word, the HD-tree may not be able to determine if the word is the query answer or not, since appending more residues at the end of the word may achieve higher score. In this situation, the HD-tree must follow the position information associated with the word to continue the search in the original database sequences. This operation is expensive and not recommended in the HD-tree (see Section 8.2 for the future work related this issue). Therefore, the HD-tree requires the query length to be shorter than the length of overlapping words.

Figure 6.10 shows the level distribution of the HD-tree in the RAM for different overlapping word lengths. It is shown that increasing the overlapping word length leads to an increased total HD-tree levels (i.e., the area under a curve) in the RAM. This in turn increases the possibility to reduce the number of disk accesses, since more sub-trees may be abandoned during the search. However, as shown in Table 6.2, increasing the overlapping word length results in larger index size and longer construction time. Therefore, it is recommended to choose the longest overlapping word according to available resources. In the following experiments, overlapping words of length 20 are used.

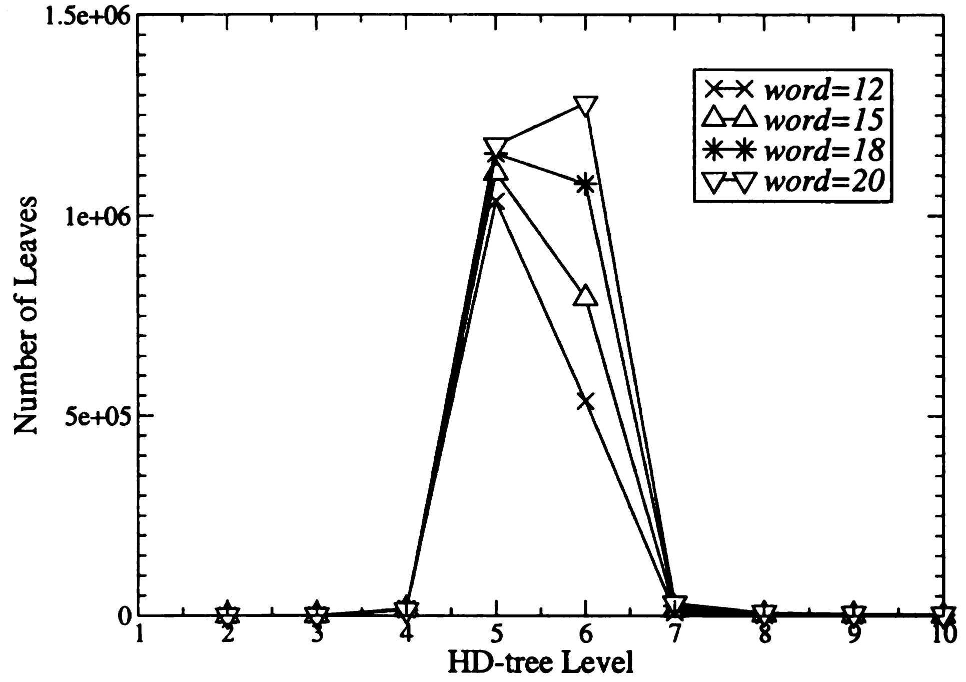


Figure 6.10: Distribution of HD-tree Levels in the RAM. Levels greater than 10 are not shown.

6.5.3 Quality of Query Results

In Table 6.3, the query result of the HD-tree is compared with that of BLAST. The column "Common" indicates the total number of insulin sequences found by both the HD-tree and the BLAST. The column "HD-only" and "BLAST-only" indicate the total number of insulin sequences found only by the HD-tree and BLAST, respectively. As shown in the table, the quality of the HD-tree query results increases as the corresponding closeness decreases. This is because the minimum score required for a matching decreases as closeness decreases. Therefore, alignments with lower matching score (i.e., distantly related) are returned as query answers. When the closeness is low enough, the HD-tree not only finds all the insulin sequences found by BLAST, but also finds some insulin sequences that

BLAST cannot find. For example, for query length of 14, at the closeness level of 40%, the HD-tree returns 119 extra insulin sequences that BLAST cannot find.

Table 6.3: Query quality comparison

QueryLen	Closeness (%)	Common	HD-only	BLAST-only
10	90	8	0	0
10	80	8	0	0
10	70	8	0	0
10	60	8	1	0
10	50	8	1	0
10	40	8	1	0
10	30	8	72	0
14	90	28	4	197
14	80	128	4	97
14	70	179	12	46
14	60	209	22	16
14	50	219	29	6
14	40	225	119	0
14	30	225	206	0
14	20	225	280	0
18	90	17	2	16
18	80	25	3	8
18	70	26	4	7
18	60	33	4	0
18	50	33	67	0
18	40	33	69	0
18	30	33	112	0
18	20	33	131	0

In Figure 6.11, the number of insulin sequences among the top N results is compared between the HD-tree and BLAST. These query results are sorted by their matching scores. It is shown that the HD-tree consistently outperforms the BLAST. For example, among the top 1000 results, the HD-tree returns 35 more insulin sequences than that of the BLAST. Again, this shows that the HD-tree achieves better query quality than BLAST.

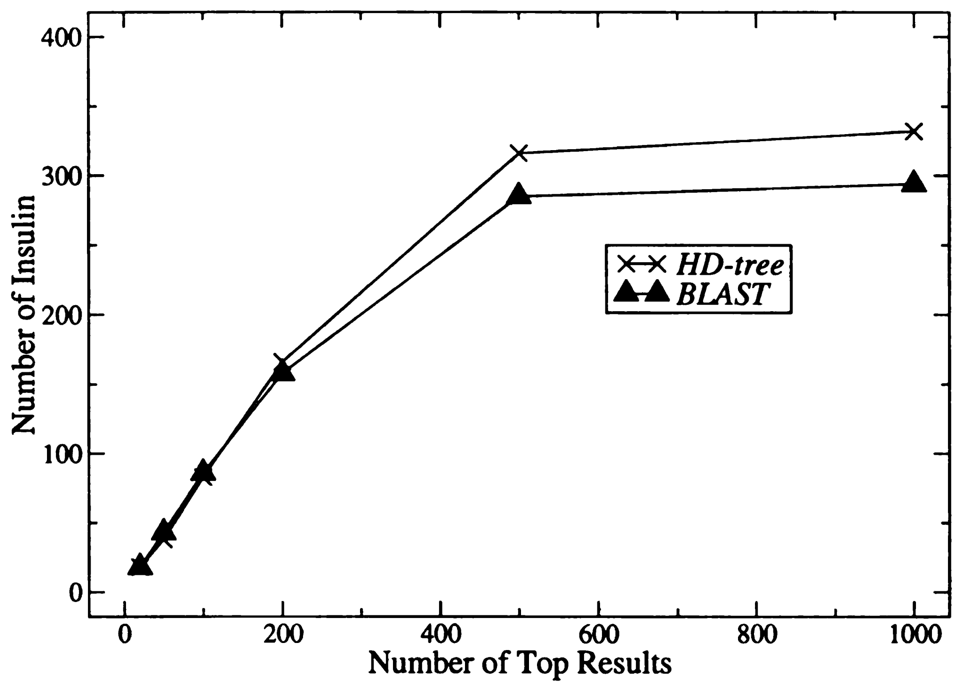


Figure 6.11: Number of insulin sequences among top N results. Query length is 14.

6.6 Query Time

Table 6.4: BLAST query time

Query Length	Query Time (seconds)
10	37
14	46.25
18	45.75

As shown in Table 6.4, the query time of BLAST is relatively stable. This is because each query has to scan through the entire database. The query time of the HD-tree increases as the closeness increases. However, as shown in Figure 6.12, the HD-tree is consistently faster than BLAST. Even at the closeness level of 40%, where the quality of the HD-tree surpasses that of the BLAST, the HD-tree is still four times faster than BLAST. The query performace of the HD-tree is affected by query length. According to HD-tree search algorithm (Algorithm 6) described in Section 6.4, for a given closeness, as the query length increases, the value of $msc[i]$ increases. Consequently, the possibility to abandon a sub-tree decreases, and more

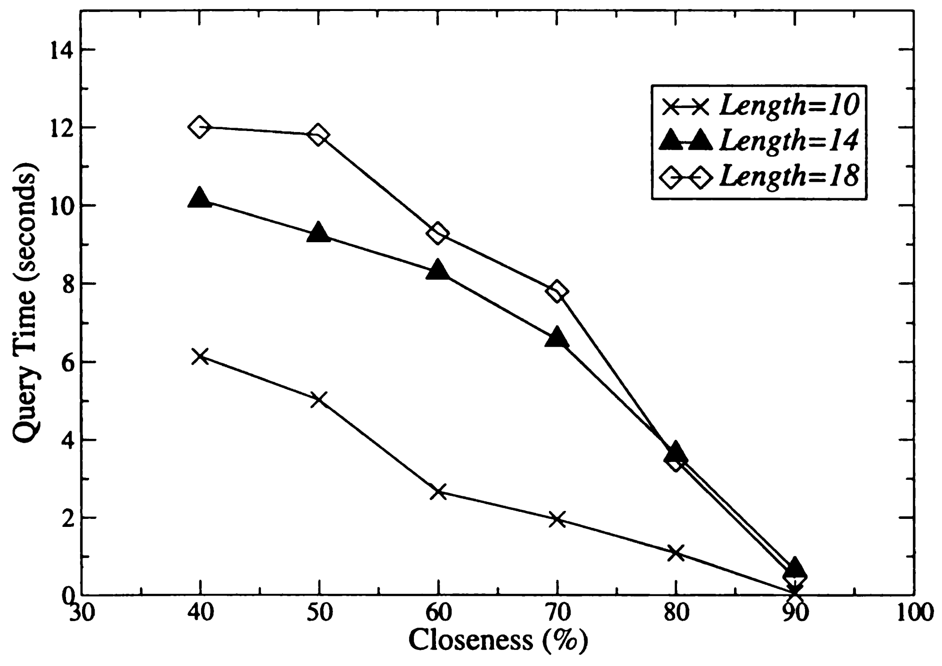


Figure 6.12: HD-tree query time

leaf nodes are accessed, which increases the query time.

According to the experiment results shown above, it can be concluded that, for short queries (e.g., query length < 20) using appropriate closeness, the HD-tree outperforms BLAST not only in speed, but also in quality of query results. The potential of the HD-tree is not limited to this achievement. In the next chapter, the HD-tree is applied to more complex sequence search tasks, where each position of a query sequence uses different scoring criteria.

Chapter 7: Sequence Search Using the Profile Hidden Markov Model

The *Profile Hidden Markov Model* (PHMM) has received increasing attention in the field of protein homology detection, since profile-based methods are much more sensitive in detecting distant homologous relationships than pairwise methods [92, 93, 94, 95]. Because of the computational complexity involved in PHMM searches, heuristic alignment algorithms, such as BLAST and FASTA, have not been successfully applied in this area. Pure dynamic-programming-based systems are often used for PHMM searches. However, these dynamic-programming-based systems are very time consuming. For instance, it may take approximately 15 minutes to search a short model of length 12 in the GenBank protein sequence database. The HD-tree is able to reduce the PHMM search time significantly without reducing the quality of search results. In this chapter, sequence analysis using PHMM is introduced. Algorithms for searching the HD-tree using PHMM are proposed. Finally, the HD-tree is compared with HMMER [96], a popular implementation of PHMM for protein sequence analysis. It is shown that the HD-tree is orders of magnitude faster than HMMER for short queries.

7.1 Profile Analysis

Profile analysis has long been a useful tool in finding and aligning distantly related sequences [97]. A *profile* is a description of the consensus (e.g., probability of a residue) of a multi-sequence alignment (see Section 5.1) from a group or "family" of homologous sequences. It uses a position-specific scoring system to capture the information of conservation at various positions in a multi-sequence alignment. This makes it a much more sensitive method for searching genomic sequences than pairwise methods (e.g., BLAST or FASTA) that use a position-independent scoring system. Profile-based methods need mathematical theory to support the meaning and derivation of the scores [94]. The *Hidden Markov Model* (HMM) is a type of probabilistic models that is generally applicable to time series or linear sequences. It has been widely applied to speech recognition since the 1970s [98]. In 1994, HMM was first introduced into profile-based sequence analysis as a coherent theory [99].

7.2 Markov Chain

A *Markov chain* is a sequence of random variables (X_1, X_2, X_3, ...), having the property that, given the present, the future is conditionally independent of the past [100]. In other words,

$$P(X_t = j | X_0 = i_0, X_1 = i_1, ..., X_{t-1} = i_{t-1}) = P(X_t = j | X_{t-1} = i_{t-1}). \quad (7.1)$$

Therefore, the probability of a Markov sequence, $\alpha = a_1 a_2 ... a_m$, is

$$\begin{aligned} P(\alpha) &= P(a_m, a_{m-1}, ..., a_1) \\ &= P(a_m | a_{m-1}, ..., a_1) P(a_{m-1} | a_{m-2}, ..., a_1) ... P(a_1) \\ &= P(a_m | a_{m-1}) P(a_{m-1} | a_{m-2}) ... P(a_1) \end{aligned} \quad (7.2)$$

7.3 The Hidden Markov Model

The hidden Markov model contains a finite set of states. Transitions among the states are governed by a set of probabilities, called *transition probabilities.* The state sequence is called the *path*, π. The path itself follows a Markov chain, so that the probability of a state depends only on the previous state. The transition probability from state k to state l is written as:

$$t_{k,l} = P(\pi_i = l | \pi_{i-1} = k). \tag{7.3}$$

To mark the beginning and end of the model, $t_{0,k}$ and $t_{l,0}$ are used to represent the beginning and end transition probability, respectively. Each state of HMM can produce a symbol from a distribution over all possible symbols. Therefore, the probability of producing a symbol, a, at state k is defined as:

$$e_k(a) = P(a_i = a | \pi_i = k). \tag{7.4}$$

This probability is called *emission probability.* Assume an HMM is used to generate a sequence. At any state, a residue is emitted from the state's emission probability distribution. The next state is chosen according to the state's transition probability distribution. The model then generates two strings of information. One is the underlying state path, produced by transmitting from state to state. The other is the observed sequence, where each residue is emitted from one state in the state path. The name, "hidden Markov model", comes from the fact that the state sequence is a Markov chain and is "hidden" from observers. Only the symbol sequence is directly observed. The joint probability of an observed sequence, α, and a state path, π, is:

$$P(\alpha, \pi) = t_{0,\pi_1} \prod_{i=1}^{m} e_\pi(a_i) t_{\pi_i,\pi_{i+1}}, \tag{7.5}$$

where π_{m+1} equals to 0, and represents the end of the model.

Figure 7.1 shows an example of a simple HMM. Starting in the initial state, 1, the next state is chosen with transition probability $t_{1,1}$ (i.e., staying in state 1) or $t_{1,2}$ (i.e., moving to state 2). Then a residue is generated with an emission probability associated with the current state (e.g., generate a G with $p_1(G)$). The transition or emission process is repeated until the end state is reached. In this way, a hidden state path, and an observed symbol sequence are produced.

In summary, an HMM specifies the following four properties:

(1) the symbol alphabet, Λ, containing $|\Lambda|$ different symbols (e.g., $\Lambda = \{A,C,G,T\}$ for DNA, $|\Lambda| = 4$);

(2) the number of states, N, in the model;

(3) emission probabilities, $e_i(a)$, for each state, i, that sum to one over $|\Lambda|$ symbols, a: $\sum_a e_i(a) = 1$; and

(4) transition probabilities, $t_{i,j}$, for each state, i, going to any other state (including itself), j, that sum to one over N states: $\sum_j t_{i,j} = 1$.

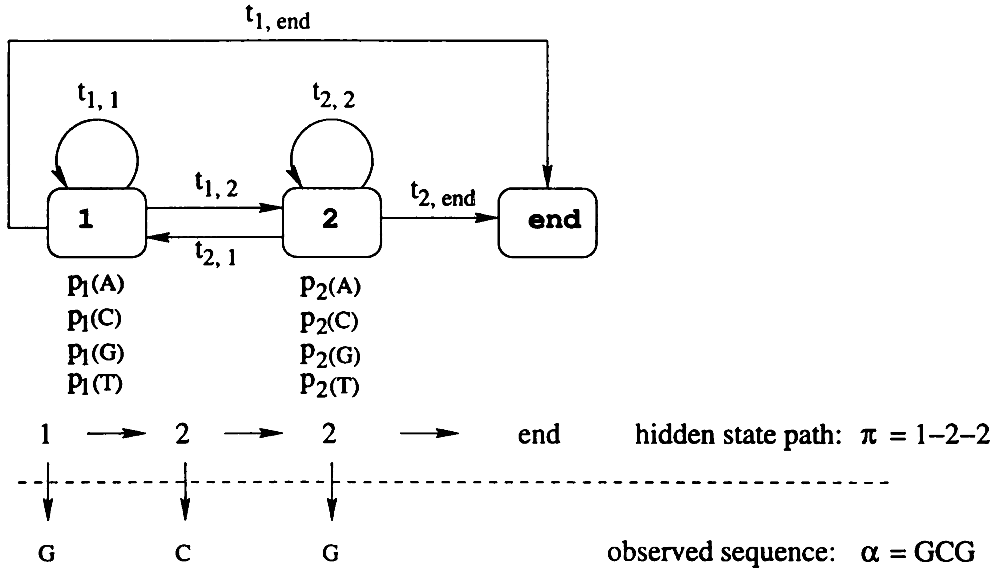


Figure 7.1: A simply HMM. The joint probability $P(\alpha, \pi) = t_{1,1}\ t_{1,2}\ t_{2,end}\ p_1(G)\ p_2(C)\ p_3(G)$. Note that another state path (1-1-2) could have generated the same symbol sequence with a probably different joint probability.

7.4 The Most Probable Path

Given a particular sequence, α, and a HMM, there are many state paths, π, may generate α. However, the probabilities of generating α from each path are very different. The path having the highest probability to generate the given sequence, α, is called the *most probable path*:

$$\pi^* = \underset{\alpha}{argmax}\, P(\alpha, \pi), \tag{7.6}$$

where *argmax* returns the maximum parameter, π, which generates the maximum value of $P(\alpha, \pi)$. The most probable path, π^*, is what people are usually interested in. It can be found recursively by the Viterbi algorithm [36]. Suppose the probability, $v_k(i)$, of the most probable path ending in state k with observation a_i is known. Then the probability, $v_l(i+1)$, of the most probable path ending in state l can be calculated for observation a_{i+1} as:

$$v_l(i+1) = e_l(a_{i+1})\, \underset{k}{max}(v_k(i)a_{k,l}). \tag{7.7}$$

Since all sequences have to start in the beginning state, 0, the initial condition is that $v_0(0) = 1$. By keeping backward pointers, the actual state sequence (i.e., the path) can be found by backtracking. Assume the transition probability to end state is $t_{k,0}$, the Viterbi algorithm is:

ALGORITHM 7 : **Viterbi**

```
**Input:** (1) a HMM of length $L$; (2) a sequence $\alpha = a_1 ... a_m$.
**Output:** the most probable path $\pi^*$.
Initialization ($i = 0$):
    $v_0(0) = 1$, $v_k(0) = 0$ for $k > 0$.
Recursion ($i = 1 ... L$):
```

$$v_l(i) = e_l(a_i)\max_k(v_k(i-1)t_{k,l});$$

$$ptr_i(l) = \operatorname{argmax}_k(v_k(i-1)t_{k,l}).$$

Termination:

$$P(x, \pi^*) = \max_k(v_k(L)t_{k,0});$$

$$\pi^* L = \operatorname{argmax}_k(v_k(L)t_{k,0}).$$

Traceback ($i = L \ldots 1$):

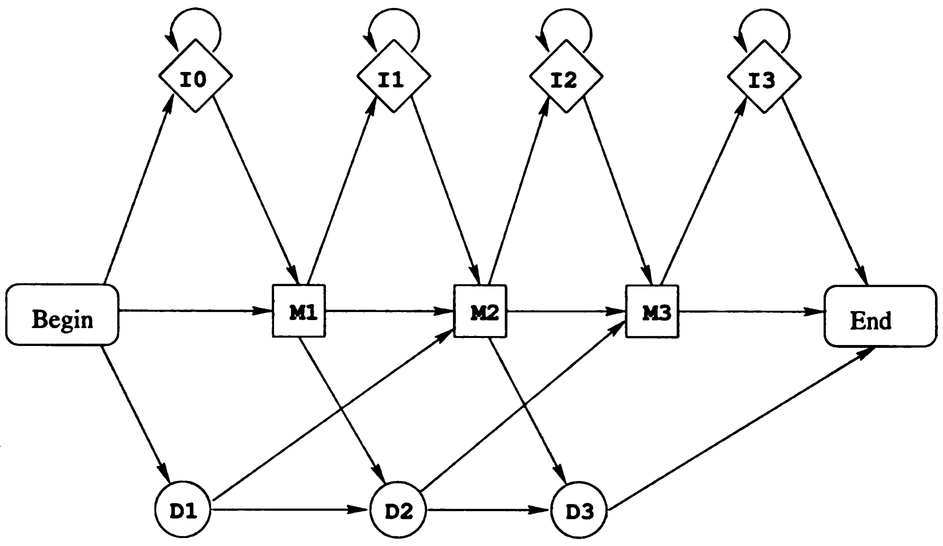
$$\pi^*_{i-1} = ptr_i(\pi^*_i).$$

7.5 The Profile Hidden Markov Model

Functional biological sequences typically come in families. Just as a pairwise alignment captures the relationship between two sequences, a multi-sequence alignment can show how the sequences in a family relate to each other. It is desirable to provide a consensus model for a multi-sequence alignment, so that the relationship between an new sequence and the family can be identified. In [99], a particular type of HMMs is introduced. This type of HMMs is well suited for representing profiles of multi-sequence alignments, and is called the Profile Hidden Markov Model (PHMM).

Unlike the general HMMs, PHMMs are strongly linear, left-right models. There are three states at each consensus column of a multi-sequence alignment: "match", "insert", and "delete". A "match" state models the distribution of residues allowed in a column. An "insert" and "delete" state at each column models insertion and deletion of one or more residues between this column and the next, respectively. A small PHMM corresponding to a short multi-sequence alignment is shown in Figure 7.2. Algorithms for generating a PHMM from a multi-sequence alignment are described in [36].

In order to increase computing speed (since sum operation is usually faster



The Multi–Sequence Alignment

1	2	3
C	A	D
C	G	E
C	A	Y
C	A	D
C	K	Y

Emission Probability:

M1: C=6/25, other=1/25

M2: A=4/25, G=2/25, K=2/25, other=1/25

M3: D=3/25, E=2/25, Y=3/25, other=1/25

Figure 7.2: A small PHMM (right) representing a small multiple alignment of five protein sequences (left) with three consensus columns. Squares represent match states (M1-M3). The 20 emission probabilities are calculated using Laplace's rule (i.e., each missing residue is counted one). Insert states (diamonds I0-I3) also have 20 emission probabilities (assume to be the same as the background distribution). Delete states (circles labeled D0-D3) are "mute" states that have no emission probabilities. State transition probabilities are shown as arrows.

than product) and resolve underflow problems due to small values of probabilities, the probability parameters in a PHMM are usually converted to additive log-odds scores [92, 101]. If the emission probability of a match state is p_a for residue a, and the expected background frequency of residue a in the sequence database is f_a, the score for residue a at this match state is $\log(p_a/f_a)$. Therefore, the scores for aligning a residue to a profile match state are comparable to that of the traditional position-independent scoring system (see Section 5.3).

In position-independent scoring systems, an insertion or deletion of an residue, a, is scored with the affine gap penalty: $g(l) = -o - (l-1)e$, where l is the gap length, o is the gap-open penalty, and e is the gap-extension penalty. In a PHMM, for an insertion of length l, there is one state transition for entering into an insert state (called M-I transition), $l-1$ state transitions for each subsequent insert state (called I-I transition), and one state transition for leaving the insert state (called I-M transition). The cost of the M-I transition is $\log t_{M,I}$, where $t_{M,I}$ is the state transition probability for moving from the match state to the insert state. In the same way, the costs of the I-I transition and M-I transition are $\log t_{I,I}$ and $\log t_{I,M}$, respectively. This is akin to the affine gap penalty, with the gap-open penalty as $\log t_{M,I} + \log t_{I,M}$, and the gap-extension penalty as $\log t_{I,I}$. However, in a PHMM, these gap costs are not arbitrary numbers. Since PHMMs have a cost for the transition from a match state to a match state that has no counterpart in position-independent scoring systems, the probability of a transition to an insert state is linked to the probability of a transition to a match state, If the gap cost is reduced by raising the transition probability, $t_{M,I}$, toward 1.0, the probability of the M-M transition, $t_{M,M}$, falls toward zero, and thus the cost for sequences without an insertion approaches negative infinity. Therefore, there is a trade-off in choosing the state transition probabilities so that the cost for the sequences having an insertion is balanced against the cost for the sequences without insertion. This is an example of why PHMMs are useful and non-trivial.

Additionally, in PHMMs, an inserted residue is associated with the emission probabilities of an insert state. If these emission probabilities are the same as the background residue frequency, the score of the inserted residue is $\log f_a/f_a = 0$. In position-independent scoring systems, inserted residues have no cost besides the affine gap penalty. This zero cost assume that insertions in protein structures have the same residue distribution as proteins in general. This assumption is usually wrong, since insertions tend to be seen most often in surface loops of protein structures, and so have a bias toward hydrophilic residues [92]. PHMMs can capture this information in emission distributions of the insert state, which increases the sensitivity of sequence searching using PHMMs.

7.6 Viterbi Equations

An important usage of PHMMs is to detect potential membership in a family by obtaining significant matches of a sequence to a given PHMM. The search can be done by the Viterbi equations [36], which are related to Algorithm 7 described in Section 7.4.

Let $V_j^M(i)$ be the log-odds score of the best path of matching a sub-sequence, $a_{1...i}$, and the sub-model up to state j, ending with a_i being emitted by state M_j. Similarly, $V_j^I(i)$ and $V_j^D(i)$ are the scores of the best path ending in a_i being emitted by I_j and D_j, respectively. Then the Viterbi equations are:

$$
\begin{aligned}
V_j^M(i) &= \log\frac{e_{M_j}(a_i)}{f_{a_i}} + max\begin{cases} V_{j-1}^M(i-1) + \log t_{M_{j-1},M_j} \\ V_{j-1}^I(i-1) + \log t_{I_{j-1},M_j} \\ V_{j-1}^D(i-1) + \log t_{D_{j-1},M_j} \end{cases} \\
V_j^I(i) &= \log\frac{e_{I_j}(a_i)}{f_{a_i}} + max\begin{cases} V_j^M(i-1) + \log t_{M_j,I_j} \\ V_j^I(i-1) + \log t_{I_j,I_j} \end{cases} \\
V_j^D(i) &= max\begin{cases} V_{j-1}^M(i-1) + \log t_{M_{j-1},D_j} \\ V_{j-1}^D(i-1) + \log t_{D_{j-1},D_j} \end{cases}
\end{aligned} \tag{7.8}
$$

In order to allow an alignment to start and end in a delete or insert state, in Figure 7.2, the "Begin" state is represented as M_0, and $V_0^M(0)$ is set to 0. The "End" state is represented as M_{L+1}, and the $V_j^M(i)$ without the emission term is used to calculate $V_{L+1}^M(n)$ as the final score for matching the HMM with the sequence, $\alpha = a_{1...n}$.

7.7 Searching PHMM Using the HD-tree

The algorithm to search PHMM in the HD-tree is related to the algorithm described in Section 6.4. Assume a query model, $q = q_{1...k}$, and a minimum matching score, S. Starting from the root, the algorithm descends recursively by every branch of the HD-tree. When descending by a branch labeled by the letter, a, the algorithm adds a to the current string, α'. Assume $phmm(q, \alpha')$ is the highest achievable score by matching q and α'. There are three possibilities for the given α':

(1) If the $phmm(q, \alpha') \geq S$, all the leaves of the current sub-tree are reported as answers.

(2) If $phmm(q, \alpha'\beta') < S$ for any string, β', the branch is abandoned

immediately.

(3) Otherwise, the algorithm continues recursively descending through the HD-tree.

The score of the best path to match the sub-sequence, $\alpha' = a_{1...i}$, and the query model, $q = q_1...q_k$, is computed using the Viterbi Equation 7.8. Program 1 is the pseudo code (C++ style) of the PHMM search algorithm at each tree node in the HD-tree. The above three possibilities (1), (2), and (3) are determined at lines 59, 61, and 63, respectively. Lines 11 to 21, 28 to 32, and 41 to 49 correspond to the computations of $V_j^M(i)$, $V_j^M(i)$, and $V_j^M(i)$ in the Viterbi equations, respectively. The variables used in the program are explained as follows:

aa is the symbol (amino acid) at the current tree node;

k is the length of the model;

-INFTY is the negative infinity value;

sc is the current best log-odds score;

gtsc(XXX, j) is the given log-odds score of the transition probability (TMM: match to match, TIM: insert to match, TDM: delete to match, etc.) from $(j-1)$th state to j state of the model;

gmsc(aa, j) and **gisc(aa, j)** are the given log-odds scores of the emission probabilities of **aa** at jth match and insert state of the model, respectively;

gmmx(i, j), **gimx(i, j)**, and **gdmx(i, j)** are the best scores at the jth match, insert, and delete state for sub-sequence $\alpha' = a_1...a_i$, respectively (i.e., $V_j^M(i)$, $V_j^M(i)$, and $V_j^M(i)$ in Equation 7.8).

max is the maximum matching score of sub-sequence, α', to the given PHMM;

max_j is the jth position where the maximum score is achieved;

maxsc[j] is the maximum achievable score of any sub-sequence to the sub-model, $q_j ... q_k$. It is the counterpart of the *msc*[*i*] in Algorithm 6 (see Section 6.4). Similar to the *msc*[*i*], the *maxsc*[*j*] is used to abandon a sub-tree which does not contain a potential match (see lines 57 to 59). The *maxsc*[*j*] is precomputed using the Program 1, except that **gmsc(aa, j)** (line 18) and **gisc(aa, j)** (line 46) are replaced by the maximum log-odds scores of the emission probabilities at jth match and insert state of the model, respectively.

<u>PROGRAM</u> 1 : PHMM Search

```
01: PhmmSearch(uchar aa, int i)
02: {
03:    int sc = 0;
04:    int rt = UNKNOWN;
05:    int max    = -INFTY;
06:    int max_j  = 0;
07:    gmmx(i, 0) = gimx(i, 0) = gdmx(i, 0) = -INFTY;
08:    for (int j = 1; j <= k; j++)
09:    {
10:       /* match state */
11:       gmmx(i, j)  = -INFTY;
12:       if ((sc = gmmx(i-1, j-1) + gtsc(TMM, j-1)) > gmmx(i, j))
13:          gmmx(i, j) = sc;
14:       if ((sc = gimx(i-1, j-1) + gtsc(TIM, j-1)) > gmmx(i, j))
15:          gmmx(i, j) = sc;
16:       if ((sc = gdmx(i-1, j-1) + gtsc(TDM, j-1)) > gmmx(i, j))
17:          gmmx(i, j) = sc;
```

```
    if (gmsc(aa, j) != -INFTY)
       gmmx(i, j) += gmsc(aa, j);
    else
       gmmx(i, j) = -INFTY;
    if (gmmx(i, j) > max)
    {
       max = gmmx(i, j);
       max_j = j;
    }
    /* delete state */
    gdmx(i, j) = -INFTY;
    if ((sc = gmmx(i, j-1) + gtsc(TMD, j-1)) > gdmx(i, j))
       gdmx(i, j) = sc;
    if ((sc = gdmx(i, j-1) + gtsc(TDD, j-1)) > gdmx(i, j))
       gdmx(i, j) = sc;
    if (gdmx(i, j) > max)
    {
       max = gdmx(i, j);
       max_j = j;
    }
    /* insert state */
    if (j < k)
    {
       gimx(i, j) = -INFTY;
       if ((sc = gmmx(i-1, j) + gtsc(TMI, j)) > gimx(i, j))
          gimx(i, j) = sc;
       if ((sc = gimx(i-1, j) + gtsc(TII, j)) > gimx(i, j))
```

```
            gimx(i, j) = sc;
         if (gisc(aa, j) != -INFTY)
            gimx(i, j) += gisc(aa, j);
         else
            gimx(i, j) = -INFTY;
         if (gimx(i, j) > max)
         {
            max = gimx(i, j);
            max_j = j;
         }
      }
   }
   curmax = max + maxsc[max_j];
   if (curmax < S)
      rt = ABANDON;
   else if (sc >= S)
      rt = FOUND;
   else
      rt = CONTINUE;

   return rt;
}
```

7.8 The HMMER Package

HMMER is a freely distributable package (current version is 2.3.2) for protein sequence analysis using PHMM [96]. It contains a set of programs useful for building and searching PHMMs. HMMER is hosted at Washington University at St. Louis, and is one of the most popular packages used by biologists to detect distant homologous relationships using PHMMs. HMMER is used to search for sequences that belong to a known protein family constructed from a multi-sequence alignment. The protein family, like most protein families, is so diverse that a BLAST search may fail to report even the known members in the family. HMMER is also used in automated annotation of the domain structure of proteins, and automated construction and maintenance of large multi-sequence alignment databases [102].

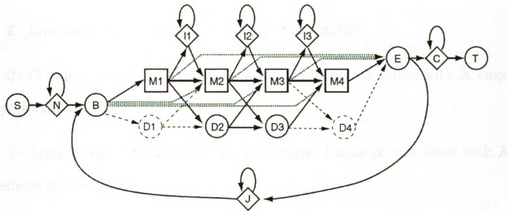


Figure 7.3: The "Plan 7" architecture of HMMER. Squares indicate match states; diamonds indicate insert states; circles indicate delete states and special states; arrows indicate state transitions.

Figure 7.3 shows the current HMMER "Plan 7" model architecture [102]. There are 7 transitions per node in the main model. Unlike the standard model in Figure 7.2, Plan 7 has five special states: 'S', 'N', 'C', 'T', and 'J'. When combined with entry probabilities from 'B' state and exit probabilities to 'E' state, these special states control unique features of the model. For instance, how likely the model is to generate various sorts of local or multi-hit alignments. The abbreviations used in Figure 7.3 are explained as follows [102]:

Mx: Match state x. Has K ($K = 20$ for protein sequences) emission probabilities.

`Dx`: Delete state x. Non-emitter.

`Ix`: Insert state x, Has K emission probabilities.

`S`: Start state. Non-emitter.

`N`: N-terminal unaligned sequence state. Emits on transition with K emission probabilities.

`B`: Begin state (for entering main model). Non-emitter.

`E`: End state (for exiting main model). Non-emitter.

`C`: C-terminal unaligned sequence state. Emits on transition with K emission probabilities.

`J`: Joining segment unaligned sequence state. Emits on transition with K emission probabilities.

In traditional pairwise alignments, distinction is made between global Needleman-Wunsch and local Smith-Waterman algorithms. However, in the Plan 7 architecture, local versus global alignment in HMMER is controlled by transition probabilities. For example, local alignments with respect to the model are achieved by non-zero state transition probabilities from the begin state, 'B', to internal match states, and from internal match states to the end state, 'E' (see dotted lines in Figure 7.3). Local alignments with respect to the sequence are achieved by non-zero state transitions on the flanking insert states, 'N' and 'C'. More than one hit to the PHMM per sequence is achieved by a cycle of non-zero transitions through the special insert state, 'J'.

7.9 HMMER Plan 7 in the HD-tree

The HD-tree adopts the Plan 7 architecture of the HMMER package, so that local, global, and multi-hit alignments can be easily controlled by transition probabilities in PHMMs. Using the same architecture also allows better comparison between HMMER and the HD-tree. To implement the Plan 7 architecture, Program 1 has to be modified to include the extra states. Therefore, Program 2 is inserted between line 15 and line 16 of Program 1, and Program 3 is inserted between line 56 and line 57 of Program 1. The variables used in Program 2 and Program 3 are explained as follows:

`bsc[j]` is the given log-odds score of the transition probability 'E' state to jth state;

`esc[j]` is the given log-odds score of the transition probability from jth state to 'E' state;

`gxsc(XXX, MOVE|LOOP)` is the given log-odds scores of the transition probability at the state, 'N', 'E', 'C', or 'J', where "MOVE" refers to the transition N-B, E-C, C-T, or J-B; and "LOOP" refers to the transition N-N, E-J, C-C, or J-J;

`gxmx(i, XXX)` is the best log-odds score of the transition probability from ith state to the state, 'B', 'E', 'C', 'J', or 'N' (e.g., XMN represents the transition probability to 'B' state).

<u>PROGRAM</u> 2 : PHMM Search for HMMER Plan 7, A

```
01:   if ((sc = gxmx(i-1, XMB) + bsc[j]) > gmmx(i, j))
02:       gmmx(i, j) = sc;
```

<u>PROGRAM</u> 3 : PHMM Search for HMMER Plan 7, B

```
/* N state */
gxmx(i, XMN) = -INFTY;
if ((sc = gxmx(i-1, XMN) + gxsc(XTN, LOOP)) > -INFTY)
   gxmx(i, XMN) = sc;
/* E state */
gxmx(i, XME) = -INFTY;
for (int j = 1; j <= k; j++)
   if ((sc =  gmmx(i, j) + esc[j]) > gxmx(i, XME))
      gxmx(i, XME) = sc;
/* J state */
gxmx(i, XMJ) = -INFTY;
if ((sc = gxmx(i-1, XMJ) + gxsc(XTJ, LOOP)) > -INFTY)
   gxmx(i, XMJ) = sc;
if ((sc = gxmx(i, XME)   + gxsc(XTE, LOOP)) > gxmx(i, XMJ))
   gxmx(i, XMJ) = sc;
/* B state */
gxmx(i, XMB) = -INFTY;
if ((sc = gxmx(i, XMN) + gxsc(XTN, MOVE)) > -INFTY)
   gxmx(i, XMB) = sc;
if ((sc = gxmx(i, XMJ) + gxsc(XTJ, MOVE)) > gxmx(i, XMB))
   gxmx(i, XMB) = sc;
/* C state */
gxmx(i, XMC) = -INFTY;
if ((sc = gxmx(i-1, XMC) + gxsc(XTC, LOOP)) > -INFTY)
   gxmx(i, XMC) = sc;
if ((sc = gxmx(i, XME) + gxsc(XTE, MOVE)) > gxmx(i, XMC))
   gxmx(i, XMC) = sc;
```

```
sc = gxmx(i, XMC) + gxsc(XTC, MOVE);
```

7.10 Comparisons

In this section, the HD-tree is compared with HMMER for sequence searching using PHMMs. The entire GenBank protein database is served as the sample database. The HD-tree is created using overlapping words of length 20. Experiments are conducted on a Linux PC with 512MB RAM and 1.8GHz Pentium 4 processor. Queries are generated from the popular PFAM database. Both synthetic and real queries are used in the experiments. Since PHMMs provide the parameters to computer E-value (see Section 5.5), the HD-tree is able to accept E-value as a search criterion besides the closeness defined in Section 6.5.1.

7.10.1 PFAM

PFAM (Protein FAMilies) is a large collection of multi-sequence alignments and PHMMs, covering many common protein families [103]. Genome projects, including both the human and fly, have used PFAM extensively for large scale functional annotation of genomic data [104]. PFAM version 18.0 (August 2005) is used in the experiments. It contains alignments and models for 7973 protein families, based on the Swissprot 47.0 and SP-TrEMBL 30.0 protein sequence databases [105]. PFAM is constructed by first distinguishing a stable curated "seed" alignment of a small number of representative sequences, then using HMMER to make a model of the seed, then searching the database for homologues using the model, and automatically producing the full alignment by aligning every sequence to the seed [103].

7.10.2 Synthetic Queries

The first set of experiments is conducted using synthetic queries, which are generated from long PHMMs in PFAM. The procedure of generating the synthetic queries is as follows. Assume a long PHMM, M_r, of length L. Transition and emission probabilities related to match, insert, and delete states of a synthetic query, M_s, of length l are copied from the ith to $(i+l)$th state of M_r. Transition probabilities from 'B' state to any state in M_s are copied from the first l states of M_r, which transition probabilities from any state to 'E' state in M_s are copied from the last l states of M_r. These synthetic queries are then run through a program, **`hmmcalibrate`**, provided by HMMER package. **`hmmcalibrate`** takes a PHMM and empirically determines parameters (such as λ in Equation 5.11) that are used to make searches more sensitive by calculating more accurate E-values (see Section 5.5).

The HD-tree is not a heuristic search algorithm such as BLAST. Therefore, all results are found as long as the search criterion (e.g., the E-value) is satisfied. Experiments have shown that the HD-tree finds all the results returned by HMMER. Therefore, the query quality of the HD-tree is the same as that of HMMER. Table 7.1 shows the average query time with respect to query lengths and E-values using the HD-tree and HMMER. For the HD-tree, the results are the averages of 100 synthetic queries unless clearly stated. However, since HMMER is very slow (10-15 minutes per query), only 10 synthetic queries are used for HMMER to generate the average. It is shown that the HD-tree is orders of magnitude faster than HMMER for queries shorter than 12. As the query length increases, the performance of the HD-tree decreases faster than that of HMMER. Therefore, it is not recommended to use the current HD-tree for queries longer than 12 (see Section 8.2 for potential solutions for long queries).

Besides the E-value, the HD-tree also uses closeness (see Section 6.5.1) to

Table 7.1: HD-tree versus HMMER, E-value = 10

Qlen	7	8	9	10	11	12
HMMER	605.1	723.1	672.0	741.6	859.4	878.9
HD-tree	0.007	0.012	0.927	5.631	38.760	154.970

Table value: Query time in seconds.

evaluate the degree of similarity between two sequences. E-value and closeness are exchangeable for a given query. Table 7.2 shows the relationship between the closeness and the E-value for the synthetic queries. It is shown that for the query of length 7, even a near-exact match (i.e., closeness = 98%) may result in E-value of at least 10. This is one reason why short queries are executed very fast using the HD-tree. In order to return more results for short queries, either the E-value is to be increased or the closeness has to be decreased.

Table 7.2: Closeness versus E-value

Qlen	Eval	AvgC	Eval	AvgC	Closeness	AvgEval	Closeness	AvgEval
7	10	98.0	100	93.8	60	3019.0	90	674.7
8	10	96.4	100	86.0	60	1348.0	90	209.2
9	10	90.6	100	75.1	60	628.6	90	69.8
10	10	82.8	100	65.0	60	295.5	90	21.5
11	10	72.8	100	55.2	60	134.5	90	5.8
12	10	63.2	100	47.5	60	64.8	90	1.8

Qlen: Query Length; *Eval*: E-value; *AvgEval*: Average E-value; *AvgC*: Average Closeness.

Figure 7.4 illustrates the relationships between query time and query length for given closenesses. It is shown that the query time decreases as the query length decreases or closeness increases. The trend is similar to that of the HD-tree in pairwise alignments using position-independent scoring matrices (see Section 6.6). In order to show the relationship between query time and disk accesses, Table 7.3 provides the statistics for queries of length 10. It is shown that the query time is closely related to the number of disk accesses, which reflects the pruning power (i.e., the ability to abandon a sub-tree in a search) of the HD-tree. Table 7.4 presents the number of dynamic programming columns computed in sequence searches using the

HD-tree, where "Ratio" is the number of columns computed by the HD-tree divided by the number of columns computed by regular dynamic programming. Similar comparison is conducted in [84] to show that using suffix index can significantly reduce the number of columns to be computed. Same conclusion can be made from Table 7.4.

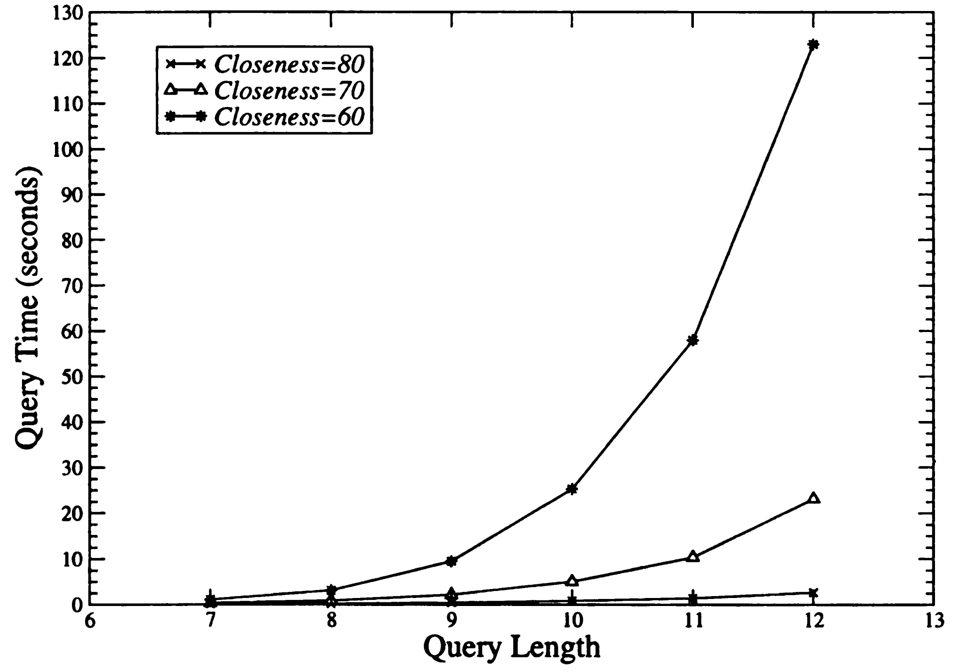


Figure 7.4: HD-tree query time for synthetic PHMMs

Table 7.3: HD-tree query time for synthetic PHMMs; Query length = 10

Closeness (%)	Qtime (seconds)	DiskAcc	AccPerc (%)	AvgE-value
90	0.14	48	0.001	21.46
80	0.88	299	0.011	51.77
70	5.04	1989	0.072	125.4
60	25.29	11153	0.407	295.5
50	100.43	57781	2.107	684.8

Qtime: query time; *DiskAcc*: the number of disk accesses; *AccPerc*: the percentage of accessed leaves.

Table 7.4: Computation of dynamic programming table columns using synthetic PHMMs, Closeness = 60%

PHMM Length	Column Computed	Ratio
7	10103	1.52844E-05
8	35335	5.34569E-05
9	117850	0.000177882
10	348271	0.000526885
11	899827	0.001361312
12	2573057	0.003892673

7.10.3 Analyzing Query Time

To further analyze the performance difference between the HD-tree and HMMER, the query time is divided into CPU time and disk access time. Table 7.5 shows the statistics of the query performance using 10 synthetic queries with different query lengths for both the HD-tree and HMMER. The original GenBank protein database, which includes sequence annotations, contains 239732 disk blocks (4KB each block). The total length of the sequences is approximately 661 million. The HD-tree generated from the database contains 2741767 leaf nodes (i.e., disk blocks). In Table 7.5, DiskAccNum is the average number of disk blocks accessed by the HD-tree. DiskAccTime is the time spent on reading data from disks. DiskAccP1 is the DiskAccNum as the percentage of the total number of leaf nodes. DiskAccP2 is the DiskAccNum as the percentage of the number of disk blocks occupied by the database, which HMMER accesses sequentially. DptColumn is the number of Dynamic Programming Table (DPT) column computed by the HD-tree, which includes the DPT columns computed for searching strings within leaf nodes. DptRatio is DptColumn divided by 661 million (i.e., the number of DPT column computed by HMMER).

As shown in Table 7.5, for different query lengths, the majority of HD-tree query time (approximately 90% or above) is spent on reading disks. However, more than 93% of HMMER query time is spent on CPU computation. Therefore, it can be concluded that the HD-tree is an I/O-bound approach and HMMER is a CPU-bound approach.

HMMER is a pure dynamic-programming-based method. It sequentially reads all database sequences, and compute one DPT column for each residue. For any query, reading the database takes approximately 40 seconds. The computation of DPT (approximately 661 million columns) dominates the query time. Therefore, the query time is relatively consistent, except some variations due to the post-processing of query results. The computational complexity of HMMER is $O(mn)$, where m is the query length and n is the database size.

Table 7.5: Analyzing query time. Both the HD-tree and HMMER use 10 synthetic queries, and E-value = 10.

QueryLen	7	8	9	10	11	12
HD-tree						
CpuTime	0.001	0.001	0.004	0.028	1.258	9.094
CpuTime%	10.0	8.3	5.6	4.9	7.9	10.5
DiskAccTime	0.009	0.011	0.067	0.543	14.720	77.420
DiskAccTime%	90.0	91.7	94.4	95.1	92.1	89.5
DiskAccNum	1	1	21	229	11304	76975
DiskAccP1 (%)	0.000	0.000	0.001	0.008	0.412	2.807
DiskAccP2 (%)	0.000	0.000	0.009	0.096	4.715	32.109
DptColumn	101	132	721	7855	395785	2697946
DptRatio	1.528E-8	1.997E-8	1.091E-7	1.188E-6	5.988E-5	4.082E-3
Closeness	99.0	97.0	88.7	77.1	67.3	59.5
MaxError	0.19	0.32	2.28	4.29	6.95	10.36
HMMER						
CpuTime	567.45	681.41	632.01	700.70	819.72	839.34
CpuTime%	93.78	94.23	94.05	94.48	95.38	95.50
DiskAccTime	37.65	41.69	39.99	40.90	39.68	39.56
DiskAccTime%	6.22	5.77	5.95	5.52	4.62	4.50

On the other hand, as an index-based method, the HD-tree significantly reduces DPT computation. This reduction is the result of two reasons. First, since the HD-tree indexes all overlapping word using trie structure, it avoids the repeated DPT computations for the subsequences (at different positions) having the same prefix. Second, during the tree traversal, a sub-tree may be abandoned (i.e., pruned) if a possible match is not possible within the sub-tree. For example, for query length of 7, the DPT column computed by the HD-tree is only 101, which is negligible compared to 661 million columns computed by HMMER. As the query length increases, the computation of DPT increases. However, even when the query length is 12, the DPT columns computed by the HD-tree is still less than 0.1% of the DPT columns computed by HAMMER. Therefore, the CPU time of the HD-tree is not the major factor of the the query performance. However, since the HD-tree stores leaf nodes on disks, the query time is dominated by the disk access time, which is proportional to the number of disk accesses (i.e., leaf-node accesses). Since the internal nodes of the HD-tree is a trie, each tree node represents a string (i.e., a prefix of overlapping words). A sub-tree can be pruned (i.e., do not access the leaf nodes in this subtree) if the best alignment between the query and the prefix string representing the sub-tree has exceeded the maximum error (i.e., the maximum score minus the minimum score. See Section 6.5.1.). The maximum error is determined by both the closeness and the length of query sequence, and the closeness plays a more important role. As shown in Table 7.5, the average closeness decreases as the query length increases, hence the maximum error (MaxError in Table 7.5) increases. The increased maximum error leads to decreased pruning power. For a given query, assume a sub-tree, T, is pruned for a maximum error of ε. If ε is increased, the search has to continue within T. Therefore, only sub-trees of T may be pruned, and the pruning power is reduced. It is observed from the experimental results that for a given HD-tree and a fixed E-value, the number of disk accesses increases exponentially as the query length increases. Therefore, as the query length increases, HMMER will sooner or later outperform the HD-tree.

7.10.4 Real Queries

In order to compare the HD-tree with HMMER using real PHMMs, experiments are conducted on PHMMs obtained from the PFAM database directly. Table 7.6 shows the query time of the HD-tree and HMMER using these real PHMMs, where "Seq" is a given number to distinguish different queries having the same length. Although the overall performance trend is similar to that in synthetic queries, it is observed that the running time varies among queries having the same length. This may due to the fact that the HD-tree uses the prefix of a query to reduce the search space. Therefore, the composition of a query affects the performance of the HD-tree. For example, if the first a few states in a PHMM has more pruning power, the HD-tree tends to be faster.

In summery, according to the results from both synthetic and real data, the HD-tree is shown to be much faster than HMMER in PHMM search for short queries. In Chapter 6, the HD-tree also outperforms BLAST for pairwise alignment using position-independent scoring matrices. As genomic sequence databases continue to grow, the benefit of using an index-based approach, such as the HD-tree, is more and more appealing than the linear-scan-based approach, such as HMMER and BLAST, especially for complex sequence analysis tasks such as PHMM searches.

Table 7.6: Query time for real PHMMs; E-value = 10

Qlen	Seq	HD-tree (seconds)	Closeness	HMMER (seconds)
7	1	0.386	74	563
8	1	0.028	79	741
8	2	0.363	67	704
8	3	0.036	99	951
8	4	0.002	99	741
8	5	0.128	72	709
9	1	4.661	57	660
9	2	0.001	99	692
9	3	4.871	58	671
9	4	0.001	99	757
10	1	0.168	81	740
10	2	8.909	53	731
10	3	0.612	76	724
10	4	9.464	61	951
10	5	3.168	70	728
10	6	0.001	99	717
11	1	5.685	47	809
11	2	0.312	81	856
11	3	5.312	66	821
11	4	7.000	64	786
11	5	1.112	66	844
12	1	36.252	45	898
12	2	160.322	48	875
12	3	52.374	53	905
12	4	229.911	41	843
12	5	211.286	34	892

Qlen: query length; *Seq*: query sequence number.

Chapter 8: Conclusions and Future Work

8.1 Conclusions

There is an increasing demand for efficient indexing techniques to support various types of queries (e.g., prefix searches and approximate string matching) on large string databases, such as genomic sequence databases. Most existing string indexing techniques are either RAM-based or disk-based. RAM-based index structures are not suitable for large databases when only a limited amount of RAM is available. Disk-based structures, on the other hand, can index large databases but usually do not fully utilize the available RAM.

In this dissertation, a novel hybrid RAM/disk-based structure, the Hybrid Digital tree (HD-tree), is proposed. The HD-tree takes advantage of the strengths of both RAM-based and disk-based structures. It contains two parts: the RAM-index and the disk-index. The RAM-index uses the trie structure, and resides in the RAM to minimize the disk accesses; while the disk-index maintains the rest of the index on disks so that large databases can be indexed. Algorithms for constructing and searching the HD-tree are developed. The HD-tree not only scales well with the size of the RAM and the database, but also is efficient for various types of queries. Experiments are conducted to compare the HD-tree with existing techniques. In comparisons with the Prefix B-tree for prefix searches, the HD-tree not only reduces I/O operations by more than 60%, but also reduces the total query processing time by one order of magnitude. The HD-tree also outperforms the linear scan and the M-tree for approximate string matching based on the Hamming distance.

The HD-tree is very useful in solving real-world applications, such as searching genomic sequence databases. The proposed Sort-Merge method successfully reduced the standard HD-tree construction time (the Brute-Force method) by an order of magnitude, so that the entire GenBank protein sequence database can be indexed in few hours. Algorithms are developed to perform sequence searches using the HD-tree. In the application of searching homologous sequences using position-independent scoring matrices, the HD-tree is not only four times faster than BLAST, a popular heuristic sequence search tool, but also able to find more homologous sequences for short queries. The speed improvement of the HD-tree is even more impressive in sequence search using Profile Hidden Markov Models (PHMMs), where heuristic algorithms are not applicable. The HD-tree is shown to be orders of magnitude faster than HMMER, a popular PHMM search tool, for short queries, while maintaining the same query quality as that of HMMER.

The major contribution of this dissertation is the application of index-based approximate string matching for genomic sequence databases. This is a prominent research area in the field of bio-informatics. Due to the complexity of advanced sequence searches, such as the profile hidden Markov model (PHMM) search, dynamic programming over the entire sequence database is used by popular search tools such as HMMER. However, as the genomic sequence databases continually grow rapidly, index-based approaches will sooner or later replace linear-scan-based approaches.

Traditional Disk-based string index structures, such as Prefix B-trees and String B-trees, are not applicable in approximate string matching, although they may index large string databases. This is because these Disk-based structures use string as the index unit (unlike the trie which decomposes a string into letters). An internal node contains a set of strings and represents sub-ranges of the entire search space. Depending on which sub-range the query string belongs to, the search continues in a child node having smaller sub-ranges. Such tree structure is effective for exact matches or prefix matches, since the query results belong to one sub-range. However, for approximate string matching, the query results may belong to many sub-ranges across the entire search space. The sub-ranges defined by internal nodes of Prefix B-trees or String B-trees are not small enough to rule out the possibility of finding a match within a sub-range (i.e., a sub-tree). Therefore, the tree loses its pruning power, and the entire tree may have to be accessed for a query.

On the other hand, trie-based index structures decompose a string into a sequence of letters and build index letter by letter. Therefore, the sub-ranges are much more refined than those in B-trees, and a search may be able to prune a sub-tree if a match is not possible within the sub-tree. Trie-based structures, such as suffix trees, are effective in approximate string matching by significantly reducing the computation of dynamic programming table [84]. However, the suffix tree is a RAM-based structure and requires a large amount of RAM, which makes it infeasible to index the entire GenBank protein database in the RAM. In [84], a method is proposed for creating suffix trees (Hunt's suffix tree) in excess of available RAM size. Disk space is served as the image of RAM for the suffix tree and accessing the suffix tree relies on the operating system to page in/out the disk image. Since the suffix tree stores pointers (i.e., suffix positions) in leaf nodes, a search may not be completed unless it follows the pointer to the original sequence database, which is costly.

The HD-tree, is a combination of both the RAM-based and disk-based structures. It uses trie-based structure to index overlapping words, so that it is as efficient as the suffix tree in reducing the computation of dynamic programming table. The HD-tree stores partial overlapping words in leaf nodes, so that a search of short queries (shorter than the length of overlapping words) does not need to access original sequence databases. Unlike the disk-based suffix tree, the HD-tree uses a small amount of RAM to store internal nodes of the tree, and groups strings having the same prefix in leaf nodes (i.e., generate some clustering information). These RAM-based internal nodes can prune the leaf nodes (i.e., disk blocks) that do not contain the query answer. Hence the number of disk accesses is significantly reduced. Since Hunt's suffix tree does not have a mechanism to group strings into disk blocks (i.e., lack of clustering information), it requires more disk accesses than the HD-tree. Therefore, a hybrid RAM/disk-based index structure such as the HD-tree is a promising approach for indexing and searching large string databases, especially genomic sequence databases.

8.2 Future Work

The success of the HD-tree in genomic sequence searches using PHMM encourages continual research in this area. One of the biggest challenges is to support long queries. As shown in previous chapters, performance of the HD-tree degrades as query length increases. Therefore, a new method for employing the HD-tree needs to be developed for long queries.

One promising solution for long queries is to use two search stages: filtering and alignment. In the filtering stage, the HD-tree is served as a filter to locate the potential homologous regions in the database using short sub-queries (e.g., length of 7 to 11 residues) obtained from a long query. In the alignment stage, these potential regions can be extended in both directions and be searched against the long query using dynamic programming to find the final results. This strategy is similar to that of heuristic algorithms, such as BLAST. However, most heuristic algorithms rely on hashing techniques for filtering, which makes PHMM searches very difficult. The HD-tree, on the other hand, provides an index structure that supports complex PHMM searches. At the same time, a potential homologous region returned by HD-tree searches can be more informative than that of existing heuristic algorithms, due to longer sub-query length (7 to 11 residues versus 3 to 4 residues). This will likely reduce the number of candidate regions in the alignment stage.

In the above two-stage approach, how to generate sub-queries effectively and efficiently is an important research issue. In a genomic sequence (or PHMM), some sub-regions are more informative than others. These highly informative sub-regions have been used to look for motifs (i.e., recurring patterns) in DNA and protein sequences [89, 90]. According to the Shannon theory, the *entropy* (i.e., mean amount of information) in a message can be computed as:

$$H = -\sum_{i=1} p(m_i) \log p(m_i), \tag{8.1}$$

where $p(m_i)$ is the probability of the message component, m_i [106]. For a PHMM, the emission probabilities of each symbol at each state (position) is known. Therefore, the *information content* (also called relative entropy) of a PHMM of length L can be computed as:

$$I = \sum_{j=1}^{L} \sum_{i=1}^{|\Lambda|} f_{i,j} \log \frac{f_{i,j}}{p_i}, \tag{8.2}$$

where $f_{i,j}$ is the emission probability of ith symbol in Λ, and p_i is the background frequency of the ith symbol [89, 107, 108]. The information content can then be used to find the highly informative sub-regions in a long PHMM. However, the number and length of sub-queries, and the search criteria for these sub-queries (such as closeness and E-value) all affect query time and quality. Therefore, developing the appropriate method to generate sub-queries will be an important part of future work for long-query searches using the HD-tree.

Once sub-queries are generated and query results are produced, the next step is to construct candidate homologous regions from the search results of sub-queries.

One approach is to consider each search result as a candidate region, similar to that in BLAST. Another approach is to form candidate regions based on the position of initial search results, similar to that in FASTA. The second approach is likely to be faster, but may sacrifice sensitivity, since it employs a heuristic method for selecting candidate regions from initial sub-query search results. Algorithms need to be developed and experiments need to be conducted to find the best approach to generate candidate regions, which is another important part of the future work.

In summary, the HD-tree developed in this dissertation is shown to be a valuable approach for indexing and searching large string databases. It is successfully applied to genomic sequence databases for short queries, especially in the profile hidden Markov model searches. Yet, the potential for using the HD-tree for long queries needs to be explored further so that it can be a more powerful tool for the growing field of genomic sequence analysis.

APPENDIX

Appendix A: Approximate Q-gram Matching in Genomic Sequence Databases

Searching a genomic sequence database usually begins with selecting a set of candidate regions, a stage called filtering. Local alignments on these candidates are then performed to find true homologous regions. Exact matching of q-grams (substrings or words of length q) has been used by popular systems, such as the BLAST, for the filtering stage. However, if a smaller q value is used, exact matching can result in a very large candidate set, leading to low search efficiency. On the other hand, with a larger q value for exact matching, search efficiency improves but the accuracy of the search is significantly reduced. As the size of genomic sequence databases increases, the situation may get even worse. In this appendix, the application of approximate q-gram matching based on the Hamming distance (see Section 4.5) is analyzed for the filtering stage. According to the experimental results on GenBank nucleotide databases, it is concluded that approximate matching based on a combination of larger q value and longer Hamming distance, is much more efficient and effective than exact matching. A theoretical model is developed to further analyze the performance of approximate matching.

A.1 Introduction

genomic sequence databases are widely used to assist molecular biologists in understanding the biochemical function, chemical structure and evolutionary history of organisms. Given a query sequence, a basic operation on these databases is to locate the homologous regions within existing genomic sequences. During the past decade, genomic sequence databases have been growing rapidly in size. The size of GenBank [109], a popular collection of publicly available DNA sequences, increased from 217,102,462 residues (base pairs) and 215,273 sequences in 1994, to 44,575,745,176 residues and 40,604,319 sequences in 2004 [110]. At the same time, there is an increasing demand for searches on genomic sequence databases.

To support efficient searches on genomic sequence databases, many algorithms (systems) have been developed in the past decade. Based on how the search is conducted, these systems can be divided into two categories: linear-scan-based and index-based. Linear-scan-based systems, such as FASTA [73, 74] and BLAST [3, 42], compare a query sequence with all the sequences in the database. FASTA is considered as the most accurate (sensitive) system, while BLAST is more popular and faster but less sensitive. Index-based systems, such as BLAT [79], CAFE [80, 81], and Suffix Sequoia [111, 86] perform a query using a pre-built index of the database. Although linear-scan-based systems are faster than index-based systems for smaller databases, as the size of the genomic sequence databases continually increases, index-based systems are more and more appealing for their efficiency.

Searching homologous regions in a genomic sequence database is usually conducted in two stages: the filtering stage and the alignment stage. The filtering stage detects candidate regions which are likely to be homologous. The alignment stage then examines these regions in detail and reports the regions which are indeed homologous according to some criteria. The resulting homologous regions are also called "high-scoring segment pairs" or HSPs [42]. Because of the unstructured nature of genomic sequences, the q-gram (substring or word of length q) is often used as a basic indexing/search unit in the filtering stage [42, 82, 79, 74]. Q-grams can be either overlapping (i.e., a fixed window size of L shifting from the beginning to the end of a sequence by one letter at a time) or non-overlapping. Hits (q-gram positions in the genomic sequence) are located by matching query q-grams with database q-grams. Each hit may produce a candidate region. Dynamic programming and scoring matrices are used in the alignment stage to find the true homologous regions among the candidate regions [83, 71]. These algorithms are so costly that the alignment stage may take more than 90% of the total processing time. Therefore, the quantity and the quality of candidate regions produced by the alignment stage are very important for improving the overall efficiency of the searching algorithm. Heuristics, such as the two-hit method in [42], have been developed to reduce the number of candidate regions based on original hits. Besides q-grams, suffix keys are also used as an index/search unit in the filtering stage [112]. Compared to q-gram based approaches, these suffix-key-based methods take more time and space; however, they are more sensitive in finding matches that have relatively low similarity to the query [112, 111].

In the filtering stage, both FASTA and BLAST use a hashing technique to sequentially search overlapping query q-grams against overlapping database q-grams. On the other hand, BLAT builds an index based on non-overlapping q-grams in memory. It uses either exact matching or approximate matching with at most one mismatch in the filtering stage to locate candidate regions. BLAT was shown to be faster than popular existing tools, such as BLAST; however, it is less sensitive. CAFE uses inverted files [7] to index genome databases. Overlapping q-grams are used in CAFE. Heuristics, such as FRAMES [80], are used to reduce the number of candidate regions passed to the alignment stage. Compared with FASTA, CAFE is shown to be faster in searching GenBank nucleotide databases with comparable accuracy [80].

This appendix focuses on the performance of the filtering stage using q-grams. Most existing systems use exact matching of q-grams to find the hits. For example, BLAST provides the options of using different word length q. The shorter q, the higher the sensitivity is; however, the number of resulting candidate regions also increases dramatically. For example, for a typical BLAST search, 61,926,143 hits might be found using $q = 7$, while only 271,083 hits would be found using $q = 11$. On the other hand, homologous regions that would be found using $q = 7$ can be missed using $q = 11$. Using exact matching, it is difficult to further reduce the number of hits while increasing the sensitivity. Since HSPs can be viewed as the results of approximate matching, using approximate matching based on the Hamming distance in the filtering stage may produce lower number of hits as well as higher sensitivity.

Approximate matching on string databases has been studied extensively [9]. Index-based q-gram methods were proposed to reduce the search cost for large genome databases [113, 114]. In BLAT, one mismatch has been shown to be effective in finding low similarity HSPs; however, the system does not provide the option to use more than one mismatch. Approximate q-gram matching with more than one mismatch has not been adopted widely in searching genome databases.

The goal of this appendix is to investigate the effect of applying larger word length (q) and higher Hamming distance (i.e., the number of mismatches) to the filtering stage. Experiments were conducted on both the E. coli and the entire GenBank nucleotide databases to investigate the performance (cost and sensitivity) of various combinations of word length and Hamming distance. A theoretical model is developed to further analyze the performance. Both experimental results and theoretical analysis show that approximate matching using longer word length and larger Hamming distance can achieve both lower cost and higher sensitivity than exact matching for the filtering stage.

The rest of this appendix is organized as follows. Methodology and experimental results are discussed in Section A.2.2, the theoretical model is presented in Section A.3, and conclusions and future work are given in Section A.4.

A.2 Filtering Based on Approximate Matching

A.2.1 Motivation

The motivation for using approximate matching of q-grams for the filtering stage is inspired by the following observations. Since HSPs, especially ones with relatively low similarity, may have evenly distributed mismatches, it is likely that exact matching may not be able to find a hit (using larger q) within such HSPs. Let q/h represent approximate matching of q-grams within Hamming distance of h. Exact matching of q-grams is represented as q/0. For example, in Figure A.1(a), the HSP cannot be found by word size of 7 or larger. However, the HSP can be found by approximate matching of 13/1. On the other hand, for a region (assume it is not an HSP) in Figure A.1(b), two hits will be reported by 7/0. However, the region will be passed by 13/1. Therefore, it is possible that lower cost (number of hits) and higher sensitivity can be achieved by using proper q/h combination. Since approximate matching has two adjustable variables, q and h, it is more flexible than exact marching, where only word length q can be tuned.

```
ttgatgatgtcatagtatgc      attgatgatgtcatctta
|||||| |||||| ||||||      |||||||    |||||||
ttgatgctgtcatcgtatgc      attgatggctacatctta

         (a)                      (b)
```

Figure A.1: Example alignments

A.2.2 Methodology

BLAST is the most commonly used tool in bio-informatics. Although there are more sensitive algorithms, nucleotide BLAST (BLASTN) is much faster and, in practice, has proved sensitive enough to detect the moderate sequence similarities that imply homology. In addition to its more formal use in detecting evolutionarily related sequences, due to its speed and availability, local BLASTN is used as a core component in several primer and probe design packages, where the intent is not to find related genes or gene segments, but to find regions of sequence similarity that might cause cross-priming or cross-hybridization. For these uses, even relatively short regions of similarity will be of concern. BLASTN (version 2.2.6 for Linux, downloaded from the NCBI web site) is used to generate the standard HSP answer set by searching a database with word size of 7 and mismatch penalty of -1. Queries are 30 probes from an actual oligonucleotide micro-array; each of them has 70 residues. Sequences in standard FASTA format were downloaded from the GenBank in May 2003. The non-redundant nucleotide database, contained 1,751,987 sequences and 8,542,465,976 residues; the E. coli database contained 400 sequences and 4,662,239 residues.

A program is developed to simulate the filtering stage. Overlapping q-grams from a query sequence are compared with overlapping q-grams from the database sequences. Given a Hamming distance h, a hit is recorded if the Hamming distance between a query q-gram and a database q-gram is within distance h. Assume the standard HSPs are provided. If a hit lies within a standard HSP, it is a true hit, otherwise, it is a false hit. An HSP is considered to be found by the program if at least one hit lies within a standard HSP. Each false hit generates a candidate region, while any number of true hits within a HSP generates only one candidate region. Therefore, the search cost measured by alignments (i.e., the number of candidate regions) is computed by the number of false hits plus the number of HSPs found. Sensitivity is measured by the number of HSPs found divided by the number of standard HSPs.

A.2.3 Experimental Results

Two sets of experiments are conducted using various q and h combinations on both the E. coli database and the entire GenBank nucleotide database. To fully understand the behavior of approximate matching for different HSPs, four HSP categories are defined for the E. coli database and two HSP categories for the GenBank nucleotide database based on the length of the HSPs and their similarity to the query. The categories are shown in Table A.1.

Table A.1: Categories of HSPs returned by Probes

	E. coli				
Category		I	II	III	IV
	total	$L^* \geq 20$	$L^* \geq 40$	$L^* \geq 40$	$20 \leq L^* \leq 40$
			$S^+ \leq 80$	$S^+ \leq 80$	
HSPNumber	482	387	52	43	192
ActualS	[63, 100]	[63, 95]	[63, 89]	[63, 80]	[69, 80]
ActualL	[15, 70]	[20, 70]	[40, 70]	[40, 70]	[25, 40]

	GenBank		
Category		V	VI
	total	$L^* \geq 40$	$20 \leq L^* \leq 40$
		$S^+ \leq 80$	$S^+ \leq 80$
HSPNumber	2690	205	
ActualS	[65, 100]	[65, 80]	[76, 80]
ActualL	[21, 70]	[40, 70]	[35, 40]

L^*: HSP length; S^+: Similarity in percentage.

The sensitivities of each q/h combination are calculated based on these HSP categories. Figures A.2-A.6 show the experimental results on the cost-sensitivity relationships for some typical q/h combinations. Note that only q-grams shorter then the minimum HSP length in a HSP category are examined.

Since 11/0 is the default word size for BLASTN, it is used as the benchmark in the following discussions. In Figures A.2-A.6, all q/h combinations that have a better sensitivity and cost than 11/0 are identified within the dotted lines at the upper-left corner of the figures. In Figure A.2, it is shown that 11/0 only finds 22.0% of the total HSPs that are longer than 20, while 20/4 is able to find 77.8%

HSPs with a similar cost. 16/3 is able to find 90.7% of the HSPs with 10 times more cost than 11/0. However, its cost is still about 10 times less than that of 7/0. In each figure, there always exist some combinations of q/h perform better than 11/0 both in cost and sensitivity, such as 17/2 in Figure A.2 and 24/5 in Figure A.5. Moreover, some combinations of q/h such as 20/3 are consistently better than 11/0 in all HSP categories. Such phenomenon indicates that approximate matching is able to achieve better performance in both sensitivity and cost than exact matching. Table A.2 shows three combinations which are always better than 11/0 in cost and sensitivity for all categories. For example, compared with 11/0, on the average, 20/3 improves the sensitivity by 55%, while reduces the cost by 90%. Table A.2 also shows an interesting trend: with properly increasing word length and Hamming distance, sensitivities and costs are improved continuely.

It is observed that with a fixed word length, both cost and sensitivity increases as the Hamming distance increases. With a fixed Hamming distance, both cost and sensitivity decreases as the word length increases. This trend is consistent among all HSP categories; however, the relationship of sensitivity may not be consistent if both Hamming distance and word length changes. For example, the sensitivity of 16/2 is a little better then 13/1 in Figure A.2; however, it is worse than 13/1 in Figure A.3. This indicates that different HSP categories may affect the performance of approximate matching.

In order to achieve certain sensitivity requirement, both q and h have to be adjusted in order to reduce the cost. Table A.3 shows the q/h combinations that achieve the least cost under corresponding sensitivity requirements. Since 7/0 is used to construct the standard answer set, the cost of approximate matching is compared with the cost of 7/0 at different sensitivity levels. It is shown that approximate matching is able to achieve high sensitivity with relatively low cost. For example, in category II, the sensitivity of 30/10 is 98%; however, the cost is

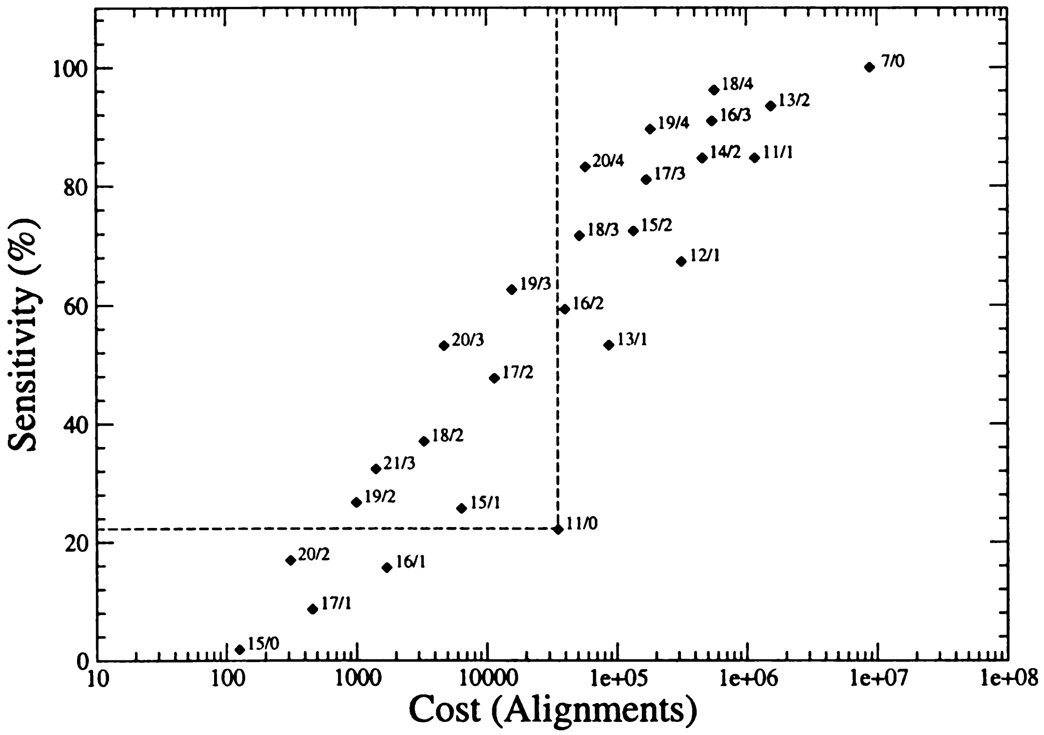


Figure A.2: E. coli, HSP Category I

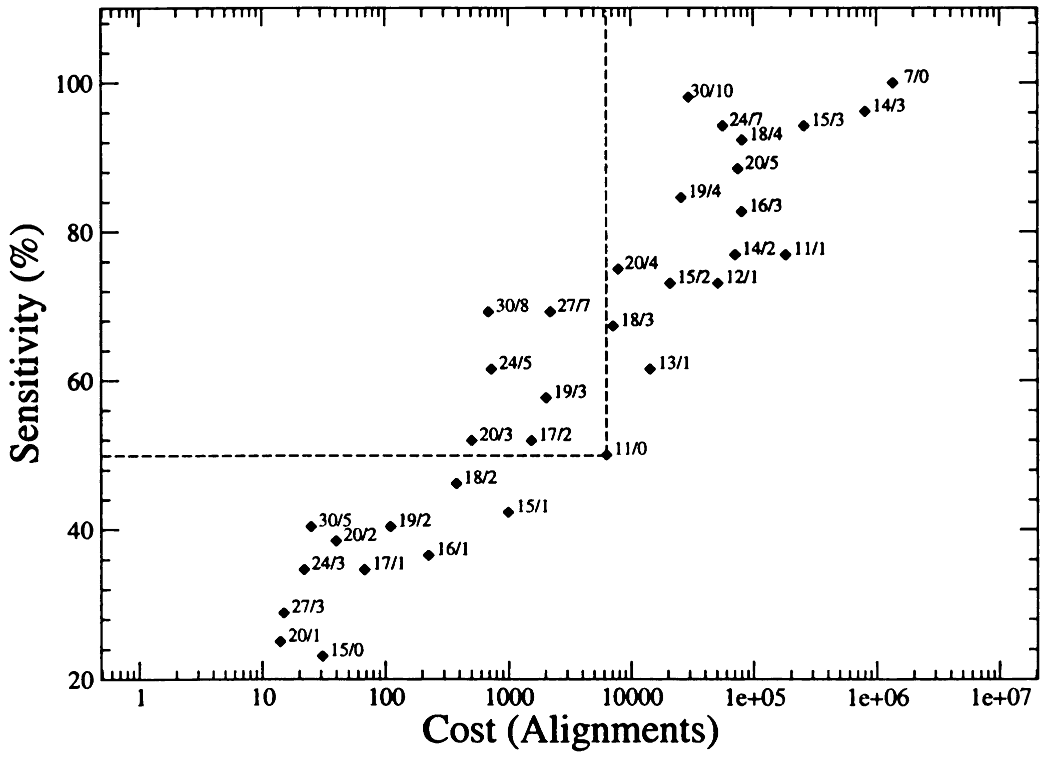


Figure A.3: E. coli, HSP Category II

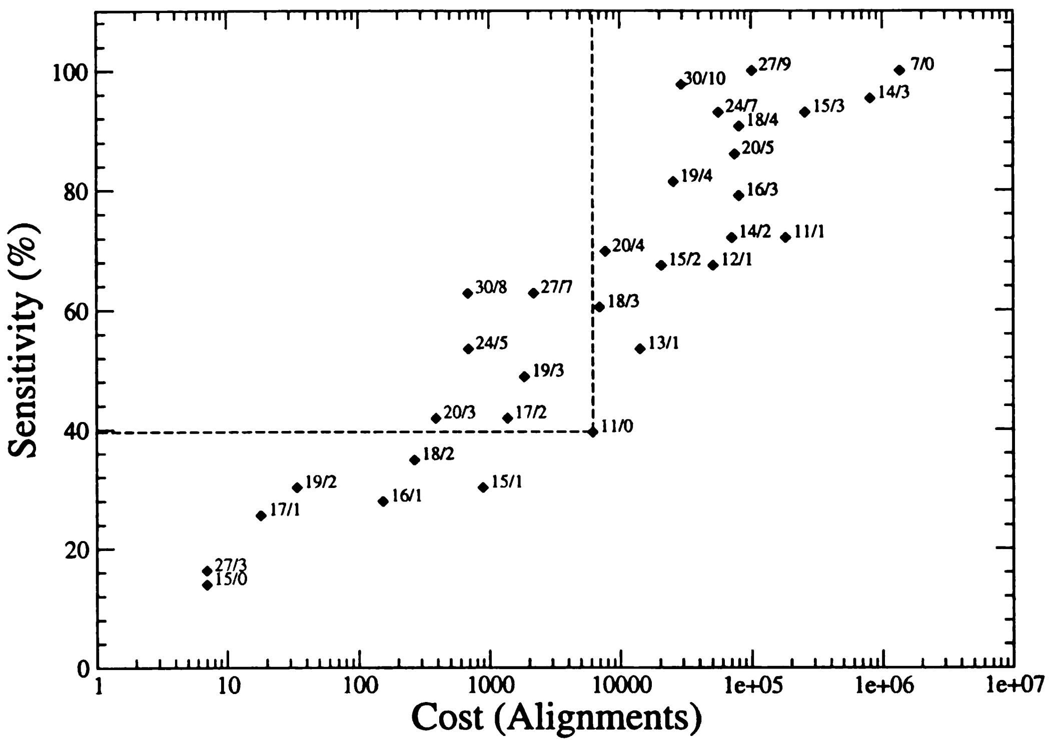


Figure A.4: E. coli. HSP Category III

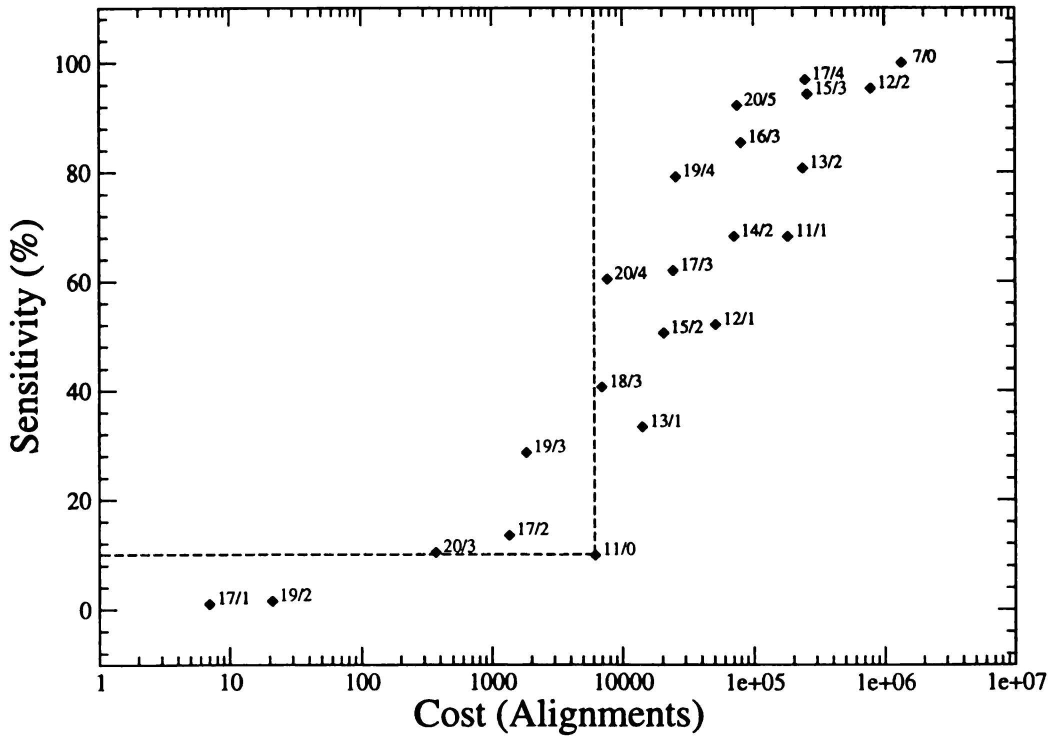


Figure A.5: E. coli, HSP Category IV

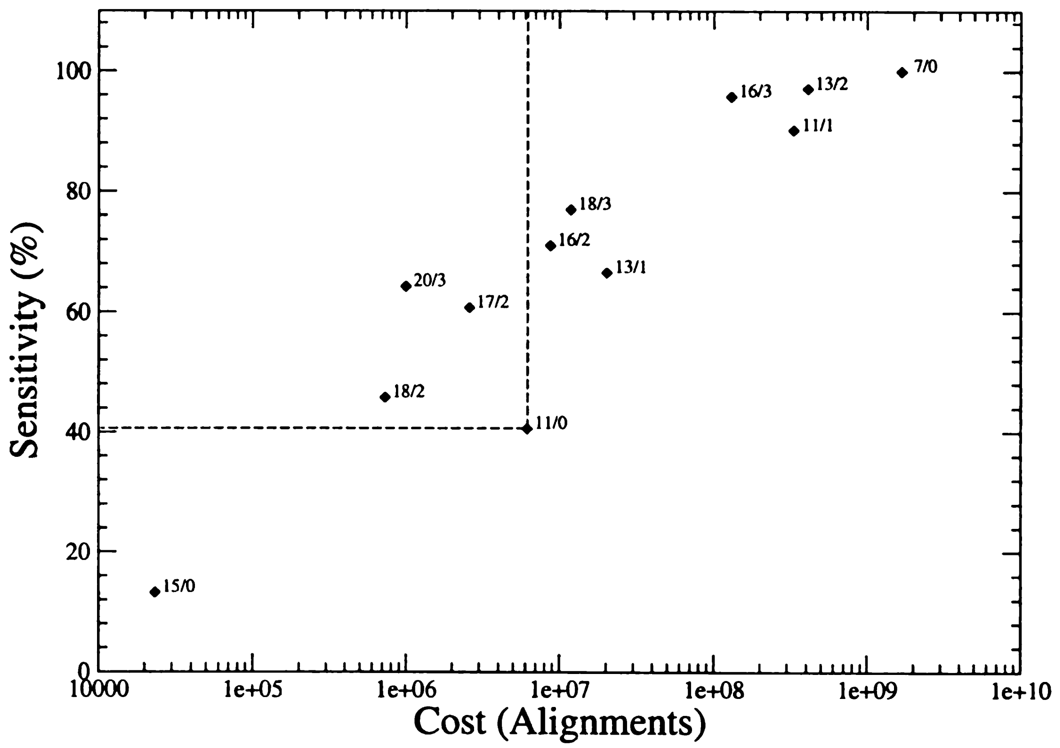


Figure A.6: GenBank. HSP Category V

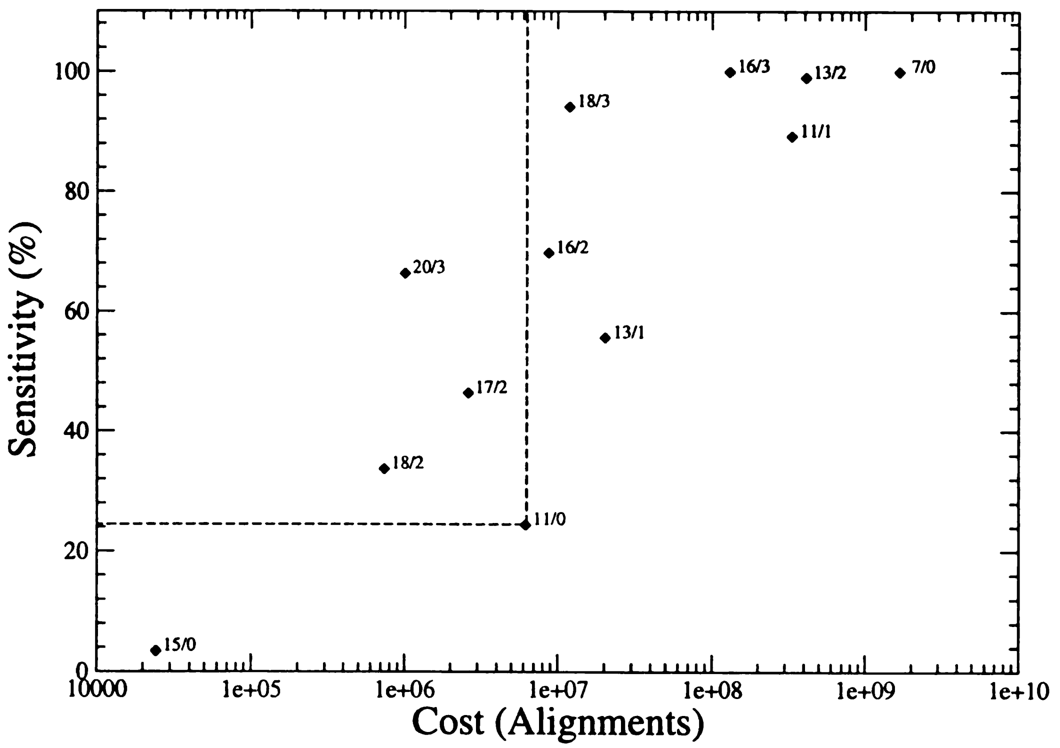


Figure A.7: GenBank, HSP Category VI

only 2% of that using 7/0. In category V, 16/3 finds 96% of HSPs using only 8% cost of 7/0. Note that, since the HSPs returned by 7/0 are used as the standard HSP set to calculate the sensitivities, the sensitivity of 7/0 is always 100%. In real application, there could exist HSPs that cannot be found by 7/0, but can be hit by approximate matching.

Table A.2: Combinations better than 11/0 in both cost and sensitivity

$CT^\sharp$	I		II		III		IV	
q/h	C^-	S^+	C^-	S^+	C^-	S^+	C^-	S^+
14/1	0.61	1.35	0.61	1.08	0.60	1.12	0.60	1.89
17/2	0.24	1.53	0.24	1.04	0.22	1.06	0.22	1.37
20/3	0.08	1.84	0.08	1.04	0.06	1.06	0.06	1.05

$CT^\sharp$	V		VI		Avg.	
q/h	C^-	S^+	C^-	S^+	C^-	S^+
14/1	0.90	1.44	0.90	1.58	0.70	1.41
17/2	0.42	1.50	0.42	1.90	0.29	1.40
20/3	0.16	1.58	0.16	2.72	0.10	1.55

$CT^\sharp$: HSP Category; C^-: Cost / (Cost of 11/0); S^+: Sensitivity / (Sensitivity of 11/0).

As shown in Figure A.2 (HSP $\geq$ 20) and A.3 (HSP $\geq$ 40), the sensitivities of all combinations decrease when HSPs are shorter; however, 11/0 decreases faster than other combinations. It is concluded that approximate matching is less affected by HSP length than exact matching. For relatively low similarities and shorter HSPs, approximate matching has a greater advantage. For example, in Figure A.5, 11/0 only finds 9.9% of the HSPs while 20/4 is able to find 60.4% of the HSPs with similar cost.

To measure the consistency of above observations, experiments are carried out using some typical q/h combinations in a much larger database (i.e., the GenBank nucleotide database). As shown in Figure A.6 and A.7, the performances of the selected q/h combinations have a very similar trend to that of the E. coli database. The sensitivities of all combinations are higher than those in the E. coli database,

Table A.3: Combinations satisfy minimum sensitivity.

$CT^\sharp$	I			II			III		
MS^*	q/h	C^-	S^+	q/h	C^-	S^+	q/h	C^-	S^+
40	20/3	0.00037	0.40	30/5	0.000018	0.40	20/3	0.00029	0.42
60	18/3	0.0053	0.64	30/8	0.00051	0.69	30/8	0.0005	0.63
70	20/4	0.0058	0.78	20/4	0.0058	0.75	19/4	0.02	0.81
80	19/4	0.02	0.88	19/4	0.02	0.85	19/4	0.02	0.81
90	16/3	0.06	0.91	30/10	0.02	0.98	30/10	0.02	0.98

$CT^\sharp$	VI			V			VI		
MS^*	q/h	C^-	S^+	q/h	C^-	S^+	q/h	C^-	S^+
40	18/3	0.0051	0.41	18/2	0.00044	0.46	20/3	0.0006	0.66
60	20/4	0.0057	0.60	20/3	0.00059	0.64	20/3	0.0006	0.66
70	19/4	0.02	0.79	16/2	0.0052	0.71	18/3	0.0071	0.94
80	20/5	0.06	0.92	16/3	0.08	0.96	18/3	0.0071	0.94
90	20/5	0.06	0.92	16/3	0.08	0.96	18/3	0.0071	0.94

$CT^\sharp$: HSP Category; MS^* Minimum sensitivity; C^-: Cost / (Cost of 7/0); S^+: Sensitivity / (Sensitivity of 7/0).

because the GenBank nucleotide database contains more HSPs with higher similarities.

A.3 Theoretical Analysis

A theoretical model is developed to further analyze the performance of approximate and exact matching for the filtering stage. The problem is studied in a probabilistic framework where artificial genomic sequences are generated according to the *Bernoulli model*, which is also used by BLAST for their scoring system. In the Bernoulli model, every symbol of a finite alphabet A is created independently of the other symbols with different probabilities (identical independent distribution, i.i.d.). The following notations are used in the description of our theoretical model:

s: The similarity ratio between two regions.

Λ: the nucleotide alphabet of size $|\Lambda| = 4$.

$P_D(\alpha)$: The distribution of a letter $\alpha \in \Lambda$ in database sequences.

$P_Q(\alpha)$: The distribution of a letter $\alpha \in \Lambda$ in query sequences.

$HDist(\beta_1, \beta_2)$: The Hamming distance between two q-grams: β_1 and β_2.

A.3.1 False Hit Probability (FHP)

According to the Bernoulli model, if one letter is drawn from a query sequence and one letter is drawn from database sequences, the probability that the two letters are identical is $P_m = \sum_{\alpha \in \Lambda} P_D(\alpha) * P_Q(\alpha)$. The probability that two q-grams are identical is $M(q) = (P_m)^q$. Therefore, the probability of h mismatches between two q-grams is:

$$M'(q,h) = \binom{q}{h} \times (P_m)^{q-h}(1-P_m)^h, \tag{A.1}$$

where $\binom{q}{h}$ is the number of combinations of q taken h at a time. The probability that two q-grams are within Hamming distance h is:

$$M_f(q,h) = \sum_{i=0}^{h} \binom{q}{i} \times (P_m)^{q-i}(1-P_m)^i. \tag{A.2}$$

$M_f(q,h)$, called False Hits Probability (FHP), is the probability of matching a query q-gram by chance among database q-grams.

A.3.2 True Hit Probability (THP)

Assume the similarity between two homologous regions, HR_1 and HR_2, is s. Similar to $M_f(q,h)$, for a q-gram, β_1, from HR_1, the probability of having a q-gram β_2 in HR_2 and $HDist(\beta_1, \beta_2) \leq h$ is:

$$M_t(q,h) = \sum_{i=0}^{h} \binom{q}{i} \times s^{q-i}(1-s)^i \tag{A.3}$$

$M_t(q,h)$, called True Hit Probability (THP), is the probability of matching a q-gram within a homologous region. The higher the value of $M_t(q,h)$, the better is the chance that the homologous regions in the database will be selected in the filtering stage, hence, the higher is the sensitivity.

A.3.3 Verifying FHP and THP

To verify the correctness of the theoretical model, the theoretical FHP and THP are compared with corresponding experimental values. The experimental FHP is computed as the number of total hits divided by the search space, which is the number of query q-grams times the number of database q-grams. Table A.4 shows the alphabet distributions of the queries and databases used in our experiments. Table A.5 shows the comparison between theoretical and experimental results of FHP. Query q-grams in this experiment are generated from the same probes as mentioned in Section A.2.2. It is shown that the experimental results are close to the corresponding theoretical FHP values. The errors between theoretical and experimental values are caused by the fact that the alphabet distribution in genomic sequences is not totally independent. When the theoretical FHP is very low, such as that of 20/2, the non-independent factor, which is not included in the model, starts to dominate the number of false hits.

To verify the correctness of the theoretical THP, the HSPs returned by the BLASTN are used as the answer set. The experimental THP is the number of true hits divided by the total number of q-grams within all the HSPs. Since the similarity value varies among HSPs, the final theoretical THP value is normalized by the percentage of each valid similarity in the answer set. As shown in Table A.6, the experimental results are close to the theoretical values, which verifies the correctness of the model.

Table A.4: Alphabet Distribution

	A	T	C	G
Probes	0.26571	0.27571	0.21333	0.24524
E.coli	0.24639	0.24611	0.25404	0.25347
GenBank	0.28837	0.28408	0.21293	0.21462

Table A.5: False Hit Probability (× 1e-10)

q/h	*Theoretical*	*Experimental*
7/0	605101.2	758074.5
11/0	2352	3679.3
12/1	24369.3	30950.8
13/1	5728.4	8692.1
13/2	109008.9	146763.1
16/1	109.8	88.5
16/2	2583.3	3716.1
16/3	37269.7	52180.9
20/1	0.5	0
20/2	15.8	2.1
20/3	290.2	246.4
20/4	3795.5	5326.1

Table A.6: True Hit Probability

q/h	*Theoretical*	*Experimental*
7/0	0.137	0.154
11/0	0.0485	0.0594
13/1	0.13	0.16
16/2	0.195	0.239
18/3	0.287	0.345
20/1	0.0325	0.0475
20/2	0.107	0.135
20/3	0.223	0.273
20/4	0.372	0.443

Query: Probes; DB: GenBank

A.3.4 Discussion Based on Theoretical THP and FHP

According to Equations A.2 and A.3, fixing the word length, both THP and FHP increase as the Hamming distance increases; fixing the Hamming distances, both THP and FHP decrease as the word length increases. This trend is observed in the experimental results. Since THP is the probability of matching one query q-gram with one HSP q-gram, the value of THP is much lower than that of the actual sensitivity since there are multiple trials within a HSP; however, using THP can approximate the trend of sensitivity. Therefore, it is possible to estimate the cost and sensitivity of approximate matching using the model. Note that, for a given database and sensitivity requirement, very low FHP may not be necessary. For example, if the database size is 1e+7 and query sequence is 1e+3, the search space is about 2e+10 (since both strands must be considered), theoretically, any FHP less than 2e-10 generates less than one hit (2e+10 * 2e-10 = 1). Therefore, it is not necessary to further reduce the FHP.

Since approximate matching is more expensive than exact matching, among all combinations which satisfy given complexity and sensitivity criteria, it is cost-effective to chose the shortest word length and hamming distance combination. The model can be used to find the best choice of q/h to satisfy user specified system cost and sensitivity.

A.4 Conclusion

Searching genomic sequence databases becomes increasingly challenging as the rate of increase in database sizes continues to accelerate. Approaches based on q-grams are widely used. Most existing q-gram-based systems use exact matching to locate candidate regions in the filtering stage. Approximate matching beyond one mismatch has not been adopted in most existing q-gram-based systems. In this appendix, it is shown that searching genomic sequence databases using longer word length and larger Hamming distance in the filtering stage provides an excellent opportunity for optimizing the search cost while improving the quality of the search. The encouraging results could be a motivation for researches in efficient calculations of Hamming distance. Moreover, the significant improvement in performance achieved by the Hamming distance-based search opens the possibilities of creating more advanced indexing schemes for large genomic sequence databases, where the number of the Hamming distance computations are minimal, and the cost of the Hamming distance computation in main memory is negligible compared to the cost of secondary memory accesses. Finally, a theoretical model is developed to further analyze the performance of approximate matching. The model can be used to find the best choice of q-gram length and the Hamming distance to satisfy user specified system cost and sensitivity.

Bibliography

[1] E. W. Myers, "An overview of sequence comparison algorithms in molecular biology," Tech. Rep. TR-91-29, University of Arizona, November 1991.

[2] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge: Cambridge University Press, 1997.

[3] S. F. Altschula, W. Gisha, W. Millerb, E. W. Meyersc, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, 1990.

[4] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Doklady Akademii Nauk SSSR*, vol. 163, no. 4, pp. 845–848, 1965 (Russian).

[5] N. R. Dixon and T. B. Martin, *Automatic Speech and Speaker Recognition.* New York, NY, USA: John Wiley & Sons, Inc., 1979.

[6] J. Zobel and P. Dart, "Phonetic string matching: lessons from information retrieval," in *SIGIR 96: Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*, (New York, NY, USA), pp. 166–172, ACM Press, 1996.

[7] R. Baeza-Yates and B. Ribiero-Neto, *Modern Information Retrieval.* Addison Wesley Longman Publishing Co. Inc, 1999.

[8] J. C. French, A. L. Powell, and E. Schulman, "Applications of approximate word matching in information retrieval," in *CIKM 97: Proceedings of the sixth international conference on Information and knowledge management*, (New York, NY, USA), pp. 9–15, ACM Press, 1997.

[9] G. Navarro, "A guided tour to approximate string matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, 2001.

[10] K. Auerbach, "Largest commercial database in winter corp. topten survey tops one hundred terabytes." Web, September 14, 2005. http://www.wintercorp.com/PressReleases/ttp2005_pressrelease_091405.htm.

[11] M. Wang and X. S. Wang, "Optimizing relational store for e-catalog queries: a data mining approach," in *Proceedings of the 2002 ACM symposium on Applied computing*, pp. 1147–1152, ACM Press, 2002.

[12] F. Tian, D. J. DeWitt, J. Chen, and C. Zhang, "The design and performance evaluation of alternative xml storage strategies," *SIGMOD Rec.*, vol. 31, no. 1, pp. 5–10, 2002.

[13] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, and J. Ostell, "Genbank," *Nucleic Acids Research*, vol. 32, no. 1, pp. 23–26, 2004.

[14] A. Bairoch and R. Apweiler, "The swiss-prot protein sequence database and its supplement trembl in 2000," *Nucleic Acids Research*, vol. 28, no. 1, pp. 45–48, 2000.

[15] C. Kanz, P. Aldebert, N. Althorpe, W. Baker, A. Baldwin, K. Bates, P. Browne, A. van den Broek, M. Castro, G. Cochrane, K. Duggan, R. Eberhardt, N. Faruque, J. Gamble, F. G. Diez, N. Harte, T. Kulikova, Q. Lin, V. Lombard, R. Lopez, R. Mancuso, M. McHale, F. Nardone, V. Silventoinen, S. Sobhany, P. Stoehr, M. A. Tuli, K. Tzouvara, R. Vaughan, D. Wu, W. Zhu, and R. Apweiler, "The embl nucleotide sequence database," *Nucleic Acids Research 2005*, vol. 33, 2005.

[16] "Dna data bank of japan," April 21, 2005. http://www.ddbj.nig.ac.jp/.

[17] "Human genome project," April 21, 2005. http://www.ornl.gov/sci/techresources/Human_Genome/home.shtml.

[18] "Scalable i/o project," *Lawrence Livermore National Laboratory*, May 16, 2005. http://www.llnl.gov/icc/lc/siop/.

[19] R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica*, vol. 1, no. 3, pp. 173–189, 1972.

[20] D. R. Morrison, "Patricia practical algorithm to retrieve information coded in alphanumeric," *J. ACM*, vol. 15, no. 4, pp. 514–534, 1968.

[21] R. Bayer and K. Unterauer, "Prefix b-trees," *ACM Trans. Database Syst.*, vol. 2, no. 1, pp. 11–26, 1977.

[22] P. Ferragina and R. Grossi, "Fast string searching in secondary storage: theoretical developments and experimental results," in *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pp. 373–382, Society for Industrial and Applied Mathematics, 1996.

[23] P. Ferragina, *Dynamic Data Structures for String Matching Problems.* PhD thesis, Univerisity of Pisa-Genova-Udine, Pisa, Italy, 1997.

[24] P. Ferragina and R. Grossi, "The string b-tree: A new data structure for string search in external memory and its applications," *J. Assoc. Comput. Mach.*, vol. 46, no. 2, pp. 236–280, 1999.

[25] E. M. McCreight, "A space-economical suffix tree construction algorithm," *J. ACM*, vol. 23, no. 2, pp. 262–272, 1976.

[26] P. Weiner, "Linear pattern matching algorithms," in *14th Annual Symposium on Switching and Automata Theory*, pp. 1–11, IEEE, October 1973.

[27] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," in *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pp. 319–327, Society for Industrial and Applied Mathematics, 1990.

[28] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider, "Lexicographical indices for text: Inverted files vs. pat trees," Tech. Rep. OED-91-01, University of Waterloo, 1991.

[29] D. Comer, "Ubiquitous b-tree," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, 1979.

[30] J. Cl'ement, P. Flajolet, and B. Vall'ee, "The analysis of hybrid trie structures," in *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pp. 531–539, Society for Industrial and Applied Mathematics, 1998.

[31] T. H. Merrett, H. Shang, and X. Zhao, "Database structures, based on tries, for text, spatial, and general data," in *International Symposium on Cooperative Database Systems for Advanced Applications*, pp. 316–324, December 1996.

[32] W. B. Frakes and R. Baeza-Yates, *Information Retrieval: Data Structures & Algorithms.* Prentice Hall, 1992.

[33] D. R. Clark and J. I. Munro, "Efficient suffix trees on secondary storage," in *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, (Atlanta, Georgia, United States), pp. 383–391, Society for Industrial and Applied Mathematics, 1996.

[34] R. W. Hamming, "Error-detecting and error-correcting codes," *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[35] S. F. Altschula, "Amino acid substitution matrices from an information theoretic perspective," *Journal of Molecular Biology*, vol. 219, pp. 555–665, 1991.

[36] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, *Biological sequence analysis: Probabilistic models of proteins and nucleic acids.* Cambridge University Press, 1998.

[37] E. M. Voorhees and D. Harman, "Overview of the sixth text retrieval conference (trec-6)," in *Proceedings of the Sixth Text REtrieval Conference*, pp. 1–24, NIST Special Publication, 1997.

[38] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: an efficient access method for similarity search in metric spaces," in *Proceedings of the 23rd Very Large Data Bases*, (Athens, Greece), pp. 426–435, Auguest 1997.

[39] K. Chakrabarti and S. Mehrotra, "The hybrid tree: An index structure for high dimensional feature spaces," in *Proceedings of the 15th ICDE*, IEEE Computer Society, 1999.

[40] G. Qian, Q. Zhu, Q. Xue, and S. Pramanik, "The nd-tree: A dynamic indexing technique for multidimensional non-ordered discrete data spaces," in *Proceedings of 29th VLDB Conference*, pp. 620–631, 2003.

[41] S. Henikoff and J. G. Henikoff, "Amino acid substitution matrices from protein blocks," *Proceedings of the National Academy of Science USA*, vol. 89, pp. 10915–10919, November 1992.

[42] S. F. Altschul, T. L. Madden, A. A. Schaeffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, "Gapped blast and psi-blast : a new generation of protein database search programs," *Nucleic Acids Research*, vol. 25, pp. 3389–3402, September 1997. http://www.ncbi.nlm.nih.gov/BLAST.

[43] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashing a fast access method for dynamic files," *ACM Trans. Database Syst.*, vol. 4, no. 3, pp. 315–344, 1979.

[44] E. H. Fredkin, "Trie memory," *Communications of the ACM*, vol. 3, no. 9, pp. 490–499, 1960.

[45] P. Flajolet, "On the performance evaluation of extendible hashing and trie searching.," *Acta Inf.*, vol. 20, pp. 345–369, 1983.

[46] H. Mendelson, "Analysis of extendible hashing.," *IEEE Trans. Software Eng.*, vol. 8, no. 6, pp. 611–619, 1982.

[47] P.-Å. Larson, "Performance analysis of linear hashing with partial expansions," *ACM Trans. Database Syst.*, vol. 7, no. 4, pp. 566–587, 1982.

[48] J. K. Mullin, "Spiral storage: Efficient dynamic hashing with constant performance.," *Comput. J.*, vol. 28, no. 3, pp. 330–334, 1985.

[49] R. A. Baeza-Yates and H. Soza-Pollman, "Analysis of linear hashing revisited.," *Nord. J. Comput.*, vol. 5, no. 1, 1998.

[50] R. J. Enbody and H. C. Du, "Dynamic hashing schemes.," *ACM Comput. Surv.*, vol. 20, no. 2, pp. 85–113, 1988.

[51] D. E. Knuth, *The art of computer programming: sorting and searching.*, vol. 3. Addison Wesley, 1998.

[52] A. C.-C. Yao, "On random 2-3 trees.," *Acta Inf.*, vol. 9, pp. 159–170, 1978.

[53] R. A. Baeza-Yates and P.-Å. Larson, "Performance of b+-trees with partial expansions," *IEEE Trans. Knowl. Data Eng.*, vol. 1, no. 2, pp. 248–257, 1989.

[54] D. Lomet, "The evolution of effective b-tree: page organization and techniques: a personal account," *SIGMOD Rec.*, vol. 30, no. 3, pp. 64–69, 2001.

[55] P. J. Weinberger, "Unix b-trees," Tech. Rep. TM-81-11272-1, AT&T Bell Laboratories, 1981.

[56] L. Devroye, "A note on the average depth of tries," *Computing 28*, pp. 367–371, 1982.

[57] E. Ukkonen, "On-line construction of suffix-trees," *Algorithmica*, vol. 14, pp. 249–260, 1995.

[58] M. Farach, "Optimal suffix tree construction with large alphabets.," in *FOCS*, pp. 137–143, 1997.

[59] G. Jacobson, "Succinct static data structures," Tech. Rep. CMU-CS-89-112, Carnegie Mellon University, 1982.

[60] Q. Wang, "Performance projection for disk-based indexing of the genomic databases," Master's thesis, Department of CSE, Michigan State University, 2002.

[61] Sleepycat, "Berkeley db." http://www.sleepycat.com/.

[62] D. Sankoff and J. Kruskal, eds., *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, 1983.

[63] D. Adam, "Draft human genome sequence published," *Nature*, February 2001.

[64] F. S. Collins, E. D. Green, A. E. Guttmacher, and M. S. Guyer, "A vision for the future of genomics research – a blueprint for the genomic era," *Nature*, vol. 422, pp. 835–847, April 2003.

[65] M. Margulies, M. Egholm, W. E. Altman, and etc., "Genome sequencing in microfabri-
cated high-density picolitre reactors," *Nature advanced online publication*, 2005. http://www.nature.com/nature/journal/vaop/ncurrent/abs/nature03959.html.

[66] G. Fuellen, "Multiple alignment," *Complexity International*, vol. 4, 1997.

[67] M. O. Dayhoff, R. M. Schwartz, and B. C. Orcutt, *Atlas of Protein Sequence and Structure*, vol. 5 of *3*, ch. A model of evolutionary change in proteins, pp. 345 – 352. National Biomedical Research Foundation, 1978.

[68] S. Henikoff and J. G. Henikoff, "Automated assembly of protein blocks for database searching," *Nucleic Acids Res.*, vol. 19, pp. 6565–6572, 1991.

[69] "Blast substitution matrices," October 19, 2005. http://www.ncbi.nlm.nih.gov/blast/html/sub_matrix.html.

[70] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, pp. 443–453, 1970.

[71] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequence," *Journal of Molecular Biology*, vol. 147, pp. 195–197, 1981.

[72] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of Molecular Biology*, vol. 162, pp. 705–708, 1982.

[73] W. R. Pearson and D. J. Lipman, "Improved tools for biological sequence comparison," *Proceedings of the National Academy of Science USA*, vol. 85, no. 8, pp. 2444–2448, 1988.

[74] W. R. Pearson, "Flexible sequence similarity searching with the fasta3 program package," *Methods Mol. Biol.*, vol. 132, pp. 185–219, 2000.

[75] E. J. Gumbel, *Statistics of Extremes.* Columbia University Press, 1958.

[76] S. F. Altschul and W. Gish, "Local alignment statistics," *Methods in Enzymology*, vol. 266, pp. 460–480, 1996.

[77] W. R. Pearson, "Empirical statistical estimates for sequence similarity searches," *Journal of Molecular Biology*, vol. 276, pp. 71–84, 1998.

[78] R. Mehnert and K. Cravedi, "Public collections of dna and rna sequence reach 100 gigabases," August 22, 2005. http://www.nlm.nih.gov/news/press_releases/dna_rna_100_gig.html.

[79] W. J. Kent, "Blat the blast-like alignment tool," *Genome Res.*, vol. 12, pp. 656–664, April 2002.

[80] H. E. Williams and J. Zobel, "Indexing and retrieval for genomic databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 1, pp. 63–78, 2002.

[81] H. E. Williams, "Compressed indexing for genomic retrieval," *J. Mathematical Modelling and Scientific Computing*, vol. 9, no. 2, pp. 144–154, 1998.

[82] S. Burkhardt and J. Krkkinen, "Better filtering with gapped q-grams," *Fundam. Inf.*, vol. 56, no. 1,2, pp. 51–70, 2003.

[83] D. J. States, W. Gish, and S. F. Altschul, "Improved sensitivity of nucleic acid database searches using application-specific scoring matrices," *Methods: A companion to Methods in Enzymology*, vol. 3, no. 1, pp. 66–77, 1991.

[84] E. Hunt, M. P. Atkinson, and R. W. Irving, "Database indexing for large dna and protein sequence collections," *The VLDB Journal*, vol. 11, no. 3, pp. 256–271, 2002.

[85] G. Navarro and R. Baeza-Yates, "A hybrid indexing method for approximate string matching," *Journal of Discrete Algorithms (JDA)*, vol. 1, no. 1, pp. 205–239, 2000. Special issue on Matching Patterns.

[86] E. Hunt, "Indexed searching on proteins using a suffix sequoia," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 27, pp. 24–31, 2004.

[87] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron, "q-gram based database searching using a suffix array (quasar)," in *Proceedings of the third annual international conference on Computational molecular biology*, pp. 77–83, ACM Press, 1999.

[88] E. Ukkonen, "Approximate string-matching over suffix trees," in *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching*, pp. 228–242, Springer-Verlag, 1993.

[89] G. Z. Hertz and G. D. Stormo, "Identifying dna and protein patterns with statistically significant alignments of multiple sequences," *Bioinformatics*, vol. 15, no. 7, pp. 563–577, 1999.

[90] K. J. Kechris, E. van Zwet, P. J. Bickel, and M. B. Eisen, "Detecting dna regulatory motifs by incorporating positional trends in information content," *Genome Biology*, vol. 5, no. 7, 2004.

[91] V. Kunik, Z. Solan, S. Edelman, E. Ruppin, and D. Horn, "Motif extraction and protein classification," in *Proceedings of the 2005 IEEE Computational Systems Bioinformatics Conference (CSB05)*, 2005.

[92] S. R. Eddy, "Profile hidden markov models.," *Bioinformatics*, vol. 14, no. 9, pp. 755–763, 1998.

[93] J. Park, K. Karplus, C. Barrett, R. Hughey, D. Haussler, T. Hubbard, and C. Chothia, "Sequence comparisons using multiple sequences detect three times as many remote homologues as pairwise methods," *Journal of Molecular Biology*, vol. 284, no. 4, pp. 1201–1210, 1998.

[94] S. Henikoff, "Scores for sequence searches and alignments.," *Curr Opin Struct Biol*, vol. 6, no. 3, pp. 353–360, 1996.

[95] M. Wistrand and E. L. Sonnhammer, "Improved profile hmm performance by assessment of critical algorithmic features in sam and hmmer," *BMC Bioinformatics*, vol. 6, no. 1, pp. 99–109, 2005.

[96] "Hmmer: profile hmms for protein sequence analysis," October 27, 2005. http://hmmer.wustl.edu/.

[97] M. Gribskov, A. D. McLachlan, and D. Eisenberg, "Profile analysis: Detection of distantly related proteins," vol. 84, pp. 4355–4358, 1987.

[98] L. R. Rabiner, "A tutorial on hidden markov models and selected applicaitons in speech recognition," in *Proceedings of the IEEE*, vol. 77, pp. 257–286, 1989.

[99] A. Krogh, M. Brown, I. S. Mian, K. Sjolander, and D. Haussler, "Hidden markov models in computational biology: Applications to protein modeling," *Journal of Molecular Biology*, vol. 235, no. 1501-1531, 1994.

[100] A. Papoulis, *Probability, Random Variables, and Stochastic Processes, 2nd ed*, ch. Brownian Movement and Markoff Processes, pp. 515–553. McGraw-Hill, 1984.

[101] C. Barrett, R. Hughey, and K. Karplus, "Scoring hidden markov models," *Comput. Applic. Biosci.*, vol. 13, pp. 191–199, 1997.

[102] S. R. Eddy, *HMMER Userś Guide*, October 2003. ftp://ftp.genetics.wustl.edu/pub/eddy/hmmer/CURRENT/Userguide.pdf.

[103] A. Bateman, L. Coin, R. Durbin, R. D. Finn, V. Hollich, S. Griffiths-Jones, A. Khanna, M. Marshall, S. Moxon, E. L. L. Sonnhammer, D. J. Studholme, C. Yeats, and S. R. Eddy, "The pfam protein families database," *Nucleic Acids Res.*, vol. 32, pp. D138–D141, 2004.

[104] A. Bateman, E. Birney, L. Cerruti, R. Durbin, L. Etwiller, S. R. Eddy, S. Griffiths-Jones, K. L. Howe, M. Marshall, and E. L. L. Sonnhammer, "The pfam protein families database," *Nucleic Acids Res.*, vol. 30, no. 1, p. 276280, 2002.

[105] "The pfam database of protein families and hmms," August, 2005. http://pfam.wustl.edu/.

[106] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, pp. 379–423, 1948.

[107] T. D. Schneider, G. D. Stormo, L. Gold, and A. Ehrenfeucht, "Information content of binding sites on nucleotide sequences," *Journal of Molecular Biology*, vol. 188, pp. 415–431, 1986.

[108] M. Tompa, "An exact method for finding short motifs in sequences, with application to the ribosome binding site problem," in *Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology*, pp. 262–271, AAAI Press, 1999.

[109] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, B. A. Rapp, and D. L. Wheeler, "Genbank," *Nucleic Acids Research*, vol. 28, no. 1, pp. 15–18, 2000.

[110] "Genbank statistics," April 21, 2005. http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html.

[111] E. Hunt, "The suffix sequoia index for approximate string matching," Tech. Rep. TR-2003-135, Department of Computing Science, Glasgow University, March 2003.

[112] E. Hunt, R. W. Irving, and M. Atkinson, "Persistent suffix trees and suffix binary search trees as dna sequence indexes," Tech. Rep. TR-2000-63, Department of Computing Science, Glasgow University, October 2000.

[113] H. Hyyrö and G. Navarro, "A practical index for genome searching," in *Proceedings of the 10th International Symposium on String Processing and Information Retrieval (SPIRE 2003)*, LNCS 2857, pp. 341–349, Springer, 2003.

[114] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio, "Indexing methods for approximate string matching," *IEEE Data Engineering Bulletin*, vol. 24, no. 4, pp. 19–27, 2001. Special issue on Managing Text Natively and in DBMSs. Invited paper.