To the Graduate Council:

I am submitting herewith a thesis written by Melanie Katherine Hughey entitled "An Implementation of kd-trees for Improved Query Matching with Latent Semantic Indexing." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Michael W. Berry

Dr. Michael W. Berry, Major Professor

We have read this thesis
and recommend its acceptance:

Shirley V Browne

Accepted for the Council:

Associate Vice Chancellor and Dean of The Graduate School

# An Implementation of kd-trees for Improved Query Matching with Latent Semantic Indexing

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Melanie Katherine Hughey
May 1998

# Acknowledgments

I thank my advisor, Dr. Michael Berry, for his encouragement and guidance throughout this entire thesis project. I also thank Dr. Shirley Browne and Dr. David Straight for serving on my thesis committee. I am grateful to Todd Letsche for being available and willing to answer endless questions about the original LSI++ software. Thanks to Dian Witter Martin for helping me understand the LSI concept and for forging the thesis path ahead of me. I also thank Dr. David Mount and Sunil Arya of the University of Maryland for generously providing the kd-tree code. Finally, I thank my parents for their unwavering love and support and for believing I would succeed even in my most doubtful moments.

# Abstract

Efficient information searching and retrieval methods are needed to navigate the ever increasing volumes of digital information. Traditional lexical information retrieval methods are inefficient and often return inaccurate results. To overcome problems such as polysemy and synonymy, concept-based retrieval methods have been developed. One such method is Latent Semantic Indexing (LSI), a vector-space model, which uses the singular value decomposition (SVD) of a term-by-document matrix to represent terms and documents in k-dimensional space. As with other vector-space models, LSI is an attempt to exploit the underlying semantic structure of word usage in documents. During the query matching phase of LSI, a user's query is first projected into the term-document space, and then compared to all terms and documents represented in the vector space. Using some similarity measure, the nearest (most relevant) terms and documents are identified and returned to the user. The current LSI query matching method requires that the similarity measure be computed between the query and every term and document in the vector space. In this thesis, the kd-tree searching algorithm is used within a recent LSI implementation to reduce the time and computational complexity of query matching. The kd-tree data structure stores the term and document vectors in such a way that only those terms and documents that are most likely to qualify as nearest neighbors to the query will be examined and retrieved.

# Contents


**1 Introduction** **1**

1.1 Motivation 1

1.2 Problem and Goals 5

1.3 Overview 5

**2 Overview of the kd-tree Algorithm** **6**

2.1 Querying in the LSI Information Retrieval Model 6

2.2 Finding Nearest Neighbors 7

2.3 Building the kd-tree 8

2.4 Searching the kd-tree 10

2.5 An LSI Example with kd-trees 12

**3 Implementation of the kd-tree Algorithm** **20**

3.1 Querying in the LSI Information Retrieval Model 21

3.2 kd-tree Implementation 24

**4 The Performance and Effectiveness of the kd-tree Algorithm** **31**

4.1 Evaluation of Performance 32

4.2 Evaluation of Effectiveness 39

**5 Conclusions and Future Work** **42**

**Bibliography** **43**

**Vita** **46**

# List of Tables


| | | |
|---|---|---|
| 2.1 | The labels and titles for a small medical topics collection | 13 |
| 2.2 | Original LSI results for the query blood pressure | 19 |
| 2.3 | kd-tree LSI results for the query blood pressure | 19 |
| 3.1 | Member functions added to implement the kd-tree algorithm | 29 |
| 3.2 | Externally accessible functions added to implement the kd-tree building and searching functions | 30 |
| 4.1 | Document collections used to evaluate kd-tree algorithm performance | 33 |
| 4.2 | The number of LSI factors used for each document collection tested | 34 |
| 4.3 | The wall clock time in seconds for various numbers of factors when searching for the 5-nearest neighbors to a query in the MINIMED document collection using the original LSI++ query matching method and the kd-tree query matching method | 35 |
| 4.4 | The wall clock time in seconds for various numbers of factors when searching for the 5-nearest neighbors to a query in the MEDLARS and CCE document collections using the LSI++ query matching method and the kd-tree query matching method | 35 |
| 4.5 | The wall clock time in seconds for various numbers of factors when reading a kd-tree from disk and building a kd-tree from scratch for the MINIMED, MEDLARS and CCE document collections | 37 |
| 4.6 | The top 5 documents returned by the kd-tree algorithm for the MEDLARS document collection using 50 LSI factors | 41 |
| 4.7 | The top 5 documents returned by the original LSI++ query matching method for the MEDLARS document collection using 50 LSI factors | 41 |

# List of Figures


| | | |
|---|---|---|
| 1.1 | The rank-k approximation, $A_k$, to the term-by-document matrix $A$ | 3 |
| 2.1 | Representing the partitioning of records in the kd-tree | 10 |
| 2.2 | Finding nearest neighbors to the query | 12 |
| 2.3 | A two-dimensional LSI model of the medical article titles collection along with the query, blood pressure | 15 |
| 2.4 | The distribution of terms in the buckets of the term kd-tree | 16 |
| 2.5 | The distribution of documents in the buckets of the document kd-tree | 17 |
| 3.1 | Query matching using the kd-tree algorithm [Let96] | 23 |
| 3.2 | The LSI++ inheritance structure [Let96] | 25 |
| 4.1 | The speed improvement of the kd-tree algorithm over the original query matching method for the MINIMED, MEDLARS and CCE document collections | 38 |
| 4.2 | The precision and recall for the query crystalline lens including humans using the MEDLARS document collection | 40 |

# Chapter 1

# Introduction

As digital libraries of information grow to consume terabytes of memory, efficient methods for information retrieval are in greater demand. Several approaches to information retrieval are based on the vector-space model. These vector-space information retrieval models typically represent documents, terms and queries in the same multidimensional space, and apply some similarity measure to perform query matching (i.e., retrieval of the closest terms or documents to the query). The most desirable vector-space information retrieval models are those capable of retrieving documents that don't necessarily contain any of the search terms in the query. Documents with similar semantic content to the query can then be retrieved by computing the distance between the query vector and each document vector in the term-document space [BC87]. One such vector-space information retrieval model, which is the focus of this work, is Latent Semantic Indexing.

## 1.1 Motivation

Latent Semantic Indexing (LSI) is a conceptual-based, vector-space model of information retrieval [BF96, LB97, BDO95, DDF+90]. LSI is capable of exploiting the underlying semantic structure of word usage in documents to address problems such as polysemy

(multiple meanings of words) and synonymy (use of synonyms). LSI determines the semantic content of documents in a collection using the truncated singular value decomposition (SVD) of a term-by-document matrix

$$A = [a_{ij}],$$

where, $a_{ij}$ is the frequency that term $i$ occurs in document $j$ [BDO95]. Term weighting is often used in information retrieval to increase or decrease the importance of terms within documents or across the entire document collection [Dum91]. For example, one may define

$$a_{ij} = L(i, j) * G(i),$$

where $L(i, j)$ is the local weight for term $i$ in document $j$ and, $G(i)$ is the global weight for term $i$ across the entire document collection.

The SVD [GL96] of the term-by-document matrix A is given by

$$A = U\Sigma V^T,$$

where the diagonal matrix $\Sigma$ contains the singular values of $A$, and the columns of the orthogonal matrices $U$ and $V$ contain the left and right singular vectors, respectively, corresponding to the diagonal elements of $\Sigma$. The elements of left and right singular vectors are used by the LSI model to encode the terms and documents, respectively, in the $k$-dimensional space. Specifically, the rank-$k$ approximation to $A$ defined by

$$A_k = U_k\Sigma_k V_k^T,$$

is constructed from the first $k$ columns of the orthogonal matrices $U$ and $V$ (denoted by $U_k$ and $V_k$, respectively) and the first $k$ values of $\Sigma$ (see Figure 1.1). Each row of $U_k\Sigma_k$ and $V_k\Sigma_k$ specifies the $k$ coordinates for a term and document, respectively, in the LSI vector-

Figure 1.1: The rank-k approximation, $A_k$, to the term-by-document matrix $A$.

space model [DDF+90].

Once the truncated SVD of the original term-by-document matrix is computed and the terms and documents are encoded in $k$-dimensional space, any query must be projected into the space so that, using some similarity measure, the $m$ most-relevant terms and documents may be found and returned to the user. A query can be projected into the term-document space by forming, what is referred to as, a *pseudo-document* representation of the query in the term-document space [BDO95, DDF+90, LB97]. Specifically, the query $q$ can be represented as

$$q = q_0^T U_k \Sigma_k^{-1},$$

where $(q_0^T U_k)$ is essentially the sum of weighted term vectors corresponding to the indexed terms (or keywords) found in the query. As discussed in [DDF+90], $q$ is the centroid of the term vectors specified by the user's query.

Finally, a similarity measure (e.g., cosine) is used to determine the nearest neighbors of the *pseudo-document* $q$ within the $k$-dimensional term-document space. Cosines between the query vector and the term and document vectors can be used to produce rank-ordered lists of the *closest* terms and documents with respect to geometry and semantics.

## 1.2 Problem and Goals

This thesis presents the application of a searching technique based on kd-trees to reduce the time and computational complexity of finding the $m$-nearest term and document vectors to the query vector in the LSI-generated k-dimensional space.

## 1.3 Overview

The following chapters discuss the LSI++ implementation, and the theory, implementation, and performance evaluation of the kd-tree searching algorithm. Chapter 2 reviews the LSI++ software environment and describes the kd-tree algorithm in detail. Chapter 3 presents the implementation of the kd-tree algorithm within the LSI++ software environment, and Chapter 4 evaluates the effectiveness and performance of the kd-tree algorithm for improved query matching. Finally, Chapter 5 comprises a summary of the results obtained with the kd-tree algorithm and a few recommendations for future work.

# Chapter 2

# Overview of the kd-tree Algorithm

The kd-tree algorithm introduced by Bentley [Ben75] provides a method for searching a file containing $n$ records. Each record is represented by a vector of $k$ dimensions and the $m$ closest matches or nearest neighbors to a specified query vector (according to a distance measure $d$) are returned. The goal of the kd-tree algorithm is to reduce the number of records examined while searching for the query's nearest neighbors. The kd-tree algorithm is primarily a two step procedure with the kd-tree data structure built from a data file of $n$ records in step one and kd-tree traversal to find the closest match or matches to the query in step two [Ben75, FBF77].

## 2.1 Querying in the LSI Information Retrieval Model

In the LSI++ Information Retrieval (IR) model [LB97, Let96], queries are composed of either terms or both terms and documents. Documents are added to a query using relevance feedback [LB97, Let96]. Relevance feedback enhances the precision and recall of a user's query by adding to the query the indexed terms of the added documents. The query is projected into the term-document space by using the pseudo-document representation discussed in Section 1.1. The pseudo-document is generated from any user-specified

terms and/or documents and is referred to as the query vector. The query vector is then compared to a subset of all term and document vectors using some distance measure and a sorted, ranked list of term and document vectors nearest (or most similar) to the query is returned.

## 2.2 Finding Nearest Neighbors

One method of finding nearest neighbors using the LSI IR model is to compute the cosine between the query vector and every term and document vector represented in the $k$-dimensional space (see Section 1.1). The term and document vectors are then sorted according to the cosine with the query vector and the $m$-closest vectors returned.

Assuming that only a small subset of term and document vectors will be returned for an arbitrary query, computational time and space (i.e., memory) are wasted in computing a distance measure for term and document vectors that are not at all close to the query vector. The kd-tree algorithm provides a method of organizing the term and document vectors before the query matching phase so that the $m$-nearest neighbors to the query vector can be determined without computing a distance measure (e.g., cosine) for every term and document vector represented in the term-document space [Ben75, FBF77].

## 2.3 Building the kd-tree

The kd-tree is a binary tree with storage requirements proportional to the number of records to be searched. In this implementation of the kd-tree, a record is a term or document represented by a vector of length $k$. The construction of a kd-tree is the same for term and document vectors, thus, without loss of generality, the following discussion of kd-tree building will focus on document vectors. The kd-tree is structured as follows. There are two types of tree nodes: interior nodes and leaf nodes. Each interior node represents, but does not store, a subset of the document vectors. Each interior node has a left child and a right child. Each child represents a subset of the document vectors represented by its parent node. The subset of document vectors represented by each child node is determined by a *partitioning rule*. Interior nodes represent subsets of vectors by storing a *bounds array* specific to the subset. Specifically, the bounds array represents the geometric boundaries (within the $k$-dimensional vector space) of the subset of vectors at each interior node and is used to search for nearest neighbors. The leaf nodes, also called *buckets*, actually store document vectors. The same partitioning rule used for interior nodes, and an integer $n$ specifying the number of records each bucket will hold, determines the subset of vectors stored in each bucket.

The partitioning rule introduced above defines a discriminating dimension $d$ whose value is used as the partitioning value. This rule is best illustrated with an example. The root of the tree represents the entire set of document vectors in a $k$-dimensional vector space. This set must be partitioned into two subsets, each subset will be represented by the left and

right children of the root. To determine which vectors will belong to which child, each vector is examined. Each vector in the *k*-dimensional vector-space model with value at dimension *d* (the discriminating dimension) less than the partitioning (or cutting) value, will belong to the left child. Each vector with value at dimension *d* greater than the partitioning (or cutting) value, will belong to the right child. The partitioning rule is then applied recursively to each child node and its subset of vectors. Partitioning continues until the number of vectors in a subset is equal to or less than the specified bucket size. When this occurs the current subset of vectors is given storage in a leaf node.

The partitioning rule is applied at every internal node as the tree is built. Since every internal node in the tree represents a subset of the original set of document vectors, a new discriminating dimension and partitioning value must be chosen relative to the current subset. In this implementation of the kd-tree, known as the *optimized kd-tree* [FBF77], the discriminating dimension for the current subset of vectors is the dimension with the largest spread in values. The partitioning value is set at the median of these values [FBF77]. Figure 2.1 illustrates the partitioning of a set of records and how these partitions are represented in the kd-tree. Nodes A, B and C illustrate internal nodes that represent the geometric boundaries of subsets of term or document vectors while nodes D, E, F and G illustrate leaf nodes or buckets that store subsets of vectors.

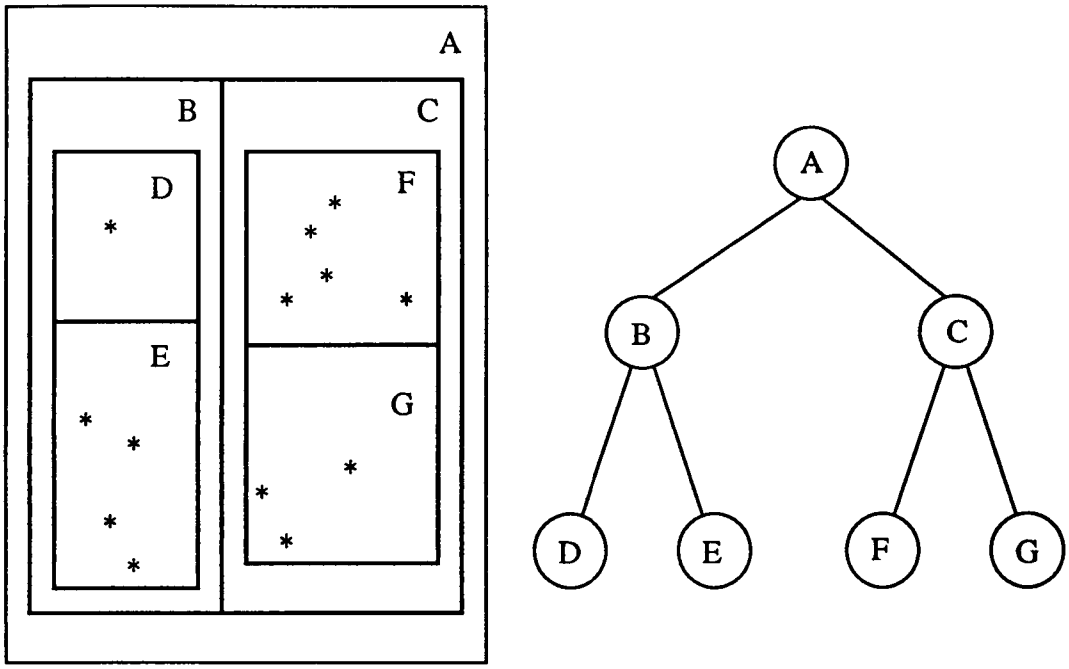


Figure 2.1. Representing the partitioning of records in the kd-tree.

## 2.4 Searching the kd-tree

The kd-tree search is implemented as a *recursive* procedure that begins at the root of the tree and traverses the nodes until a bucket is reached where a subset of term or document vectors are compared to the query vector and the nearest neighbor(s) found. Again, since the traversal of the kd-tree is the same for terms and documents, for simplicity, the following discussion will focus on the traversal of a kd-tree composed of document vectors. At each internal node (including the root), the cutting value of the current node is compared against the query vector to determine if the left or right child should be visited. When a leaf node is reached, a similarity measure (e.g., cosine) is computed between the query vector and every document vector in the bucket. Throughout the search, a list of the *m*-

closest documents found so far and their distances from the query is maintained. The list is updated if a vector is found to be closer than the farthest vector from the query in the list.

As detailed above, at internal node P, the recursive search procedure is called on P's child node that represents the subset of document vectors that lie closest to the query vector in the $k$-dimensional term-document space. If the difference between the query vector's value at discriminating dimension $d$ and the cutting (or partitioning) value of the current node is less than zero, then the left child is visited. Otherwise the right child is visited. The discriminating dimension and cutting value are determined during the tree-building process discussed in Section 2.3. When control returns to node P, the bounds array of the non-visited child is compared to the list of closest documents found so far. If the bounds array for the non-visited child intersects the sphere encompassing the query with radius $r$, then the non-visited child is visited . Hence, buckets that do not intersect this sphere are not examined. The query radius $r$ is computed as

$$r = | q_d - y_d |,$$

where $q_d$ is the $d$-th component of the query vector and $y_d$ is the $d$-th component of the vector corresponding to the closest document found so far.

For each bucket visited, the cosine with the query and each document in the bucket is computed. If necessary, the list of the $m$-closest documents found so far is updated. Figure 2.2 illustrates (in two dimensions) the situation in which a neighboring bucket must be

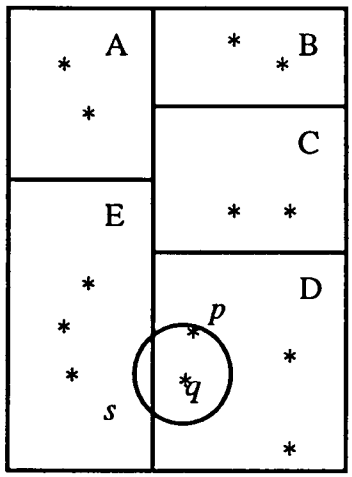


Figure 2.2. Finding nearest neighbors to the query.

visited. Point $q$ represents the query vector, point $p$ is the vector representing the closest document to the query found so far. Circle $s$ is centered at the query and has radius equal to the distance between the query and the closest document found so far. As shown, the circle overlaps buckets D and E so all vectors in both buckets must be examined in order to insure that the true nearest neighbors have been found [FBF77].

## 2.5 An LSI Example with kd-trees

The following example demonstrates the effectiveness of the kd-tree algorithm in comparison to the original LSI++ method of finding the nearest neighbors to a given query. Table 2.1 lists the document titles for a small collection of medical research articles featured in

Table 2.1: The labels and titles for a small medical topics collection. Keywords are in boldface.

| Label | Titles |
|---|---|
| d1 | **study** of **depressed patients** after **discharge** with regard to **age** of onset and **culture** |
| d2 | **culture** of pleuropneumonia like organisms found in vaginal **discharge** of **patients** |
| d3 | **study** showed **oestrogen** production is **depressed** by ovarian irradiation |
| d4 | cortisone rapidly **depressed** the secondary **rise** in **oestrogen** output of **patients** |
| d5 | boys tend to react to death anxiety by acting out **behavior** while girls become **depressed** |
| d6 | changes in children's **behavior** following hospitalization studied a week after **discharge** |
| d7 | surgical technique to **close** ventricular septal defects |
| d8 | chromosomal **abnormalities** in **blood culture** and bone marrow from leukaemic **patients** |
| d9 | **study** of christmas **disease** with **respect** to **generation** and **culture** |
| d10 | insulin not responsible for matabolic **abnormalities** accompanying a prolonged **fast** |
| d11 | **close** relationship between high **blood pressure** and vascular **disease** |
| d12 | mouse kidneys show a decline with **respect** to **age** in the ability to concentrate the urine during a water **fast** |
| d13 | **fast** cell **generation** in the eye lens epithelium of **rats** |
| d14 | **fast rise** of cerebral oxygen **pressure** in **rats** |
| d15 | **behavior** of **rats** after detected **rise** in **oestrogen** |
| d16 | **depressed patients** who feel the **pressure** to **fast** |

previous work by Berry and Fierro [BF96]. The keywords or terms for this collection are shown in bold and represent words that appear more than once in the document collection. Figure 2.3 illustrates a rank-2 LSI approximation of the term-document associations within the medical titles collection along with the query **blood pressure**. This vector-space model is generated by computing a truncated singular value decomposition (SVD) of the term-by-document matrix (see Section 1.1) for the medical titles collection, where each matrix element represents the frequency of term $i$ in document $j$ [DDF+90, BD095]. Local (log) and global (entropy) weighting functions were applied to each element of the matrix to normalize the weight of terms $i$ in document $j$ and the weight of term $i$ within the entire document collection, respectively [BDO95]. After the query is represented as a pseudo-document and projected into the term-document space (see Section 1.1), the LSI model then searches the space for the nearest neighbors or most similar terms and documents to the query. In the original LSI model, nearest neighbors are found by computing the cosine between the query and *every* term and document vector represented in the k-dimensional term-document space.

The terms and documents must then be ranked according to their cosines with the query. Finally, the $m$-closest terms and documents to the query are returned to the user where $m$ is specified by the user. In contrast, applying the kd-tree algorithm within the LSI++ model reduces the number of vectors for which the cosine must be calculated and thus reduces the time and space complexity of finding the query's nearest neighbors. Figures 2.4 and 2.5 illustrate the buckets or leaf nodes of the kd-trees built for the term and document vec-

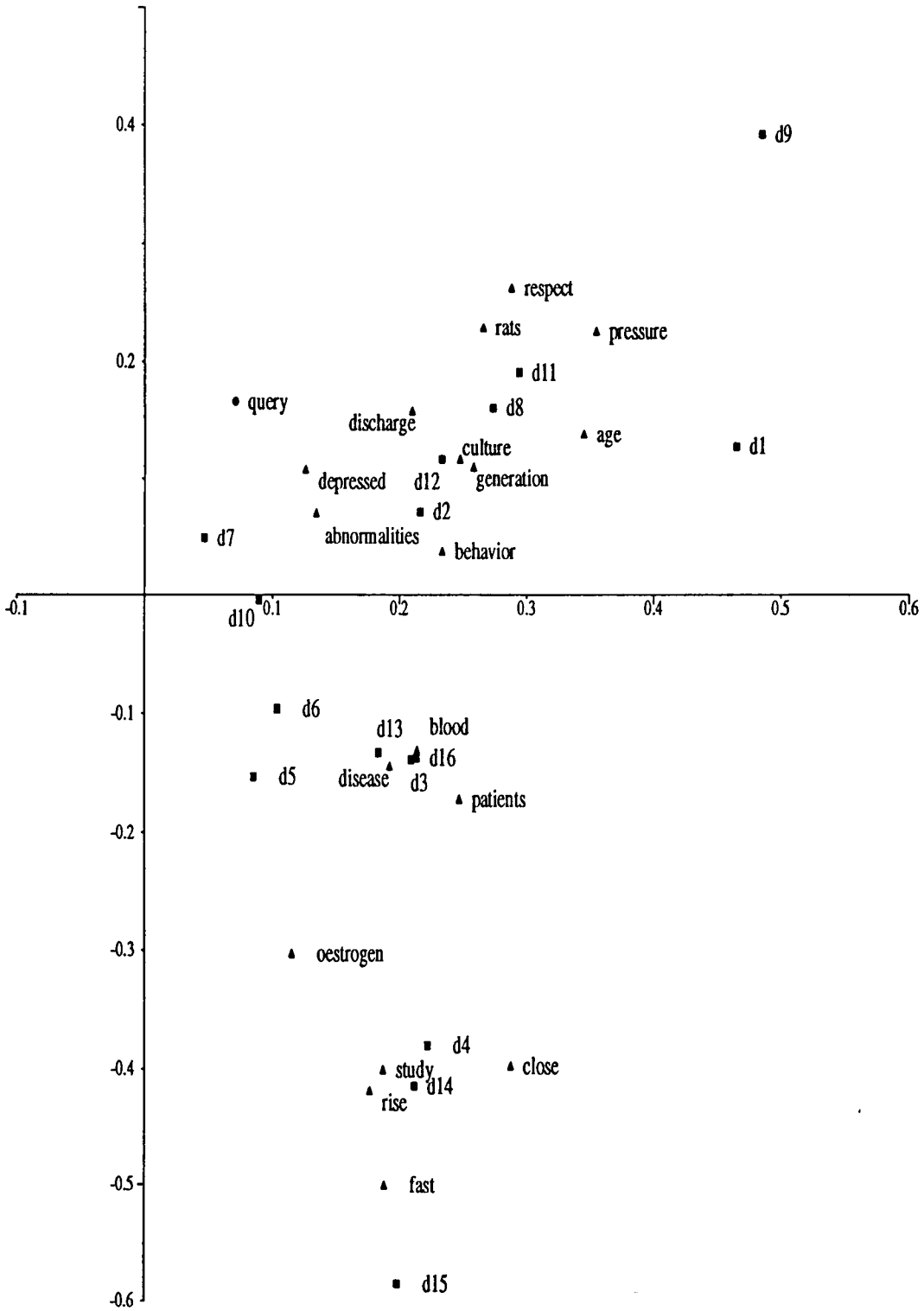


Figure 2.3: A two-dimensional LSI model of the medical article titles collection along with the query, **blood pressure**.

Figure 2.4: The distribution of terms in the buckets of the term kd-tree. Non-visited buckets are shaded.

**d1** study of depressed patients after discharge with regard to age of onset and culture

**d2** culture of pleuropneumonia like organisms found in vaginal discharge of patients

**d3** study showed oestrogen production is depressed by ovarian irradiation

**d4** cortisone rapidly depressed the secondary rise in oestrogen output of patients

**d5** boys tend to react to death anxiety by acting out behavior while girls become depressed

**d6** changes in children's behavior following hospitalization studied a week after discharge

**d7** surgical technique to close ventricular septal defects

**d8** chromosomal abnormalities in blood culture and bone marrow from leukaemic patients

**d9** study of christmas disease with respect to generation and culture

**d10** insulin not responsible for matabolic abnormalities accompanying a prolonged fast

**d11** close relationship between high blood pressure and vascular disease

**d12** mouse kidneys show a decline with respect to age in the ability to concentrate the urine during a water fast

**d13** fast cell generation in the eye lens epithelium of rats

**d14** fast rise of cerebral oxygen pressure in rats

**d15** behavior of rats after detected rise in oestrogen

**d16** depressed patients who feel the pressure to fast

Figure 2.5: The distribution of documents in the buckets of the document kd-tree. Non-visited buckets are shaded.

tors of the collection and the nodes visited in order to find the nearest terms and documents to the query. As illustrated, only a subset of term and document vectors are visited, and therefore require cosine calculation with the query. This reduction in computation is due to the use of the kd-tree which detects term and document vectors that are not likely to qualify as nearest neighbors to the query and eliminates them from consideration during tree traversal. In this particular example composed of the 18 term vectors and 16 document vectors, only 15 term vectors and 13 document vectors are actually visited. Compared to the original LSI++ implementation, the addition of the kd-tree algorithm results in the effective elimination of 6 of 34 possible cosine calculations. For collections containing tens of thousands of terms and documents, the savings in distance-to-query calculations would be of greater significance.

Tables 2.2 and 2.3 illustrate the query results returned from the original LSI++ implementation and the kd-tree LSI++ implementation, respectively. The terms and documents returned by either model are the same. Hence, this implementation of the kd-tree in the LSI IR model significantly reduced (without loss of accuracy) the number of cosine calculations necessary to return the *m*-nearest neighbors to the query.

Table 2.2: Original LSI results for the query **blood pressure.**

| **Terms** | **Cosine** | **Documents** | **Cosine** |
|---|---|---|---|
| blood | 0.94 | d11 | 0.86 |
| pressure | 0.91 | d8 | 0.70 |
| close | 0.76 | d7 | 0.70 |
| disease | 0.53 | d16 | 0.65 |
| abnormalities | 0.51 | d14 | 0.38 |

Table 2.3: kd-tree LSI results for the query **blood pressure.**

| **Terms** | **Cosine** | **Documents** | **Cosine** |
|---|---|---|---|
| blood | 0.94 | d11 | 0.86 |
| pressure | 0.91 | d8 | 0.70 |
| close | 0.76 | d7 | 0.70 |
| disease | 0.53 | d16 | 0.65 |
| abnormalities | 0.51 | d14 | 0.38 |

# Chapter 3

# Implementation of the kd-tree Algorithm

The kd-tree algorithm can be implemented in any IR model where data records and queries are represented as vectors in a $k$-dimensional term-document space and a query matching produces the $m$-closest records (term and document vectors) to the query. In this thesis, the kd-tree algorithm is implemented within the LSI++ software environment as discussed in [Let96]. Using LSI++, the user enters a query, and the $m$ terms and documents most similar related to the query are returned to the user. Prior to this incorporation of the kd-tree query matching method, the $m$-closest terms and documents to the query were found by exhaustively computing the cosine between the query vector and every term and document vector in the document collection (see Section 2.2). For very large document collections, this method of finding nearest neighbors to the query is expensive and somewhat inefficient (see Section 2.2).

By reducing the number of cosine calculations necessary to find the nearest neighbors to the query, the kd-tree algorithm can reduce the time and space complexity of query matching in the LSI IR model. Given that finding nearest neighbors to a query is the central focus of many IR models, incorporating the kd-tree algorithm into any vector-space IR model should speed-up the process of finding and returning relevant information for the

user.

## 3.1 Querying in the LSI Information Retrieval Model

The implementation of the kd-tree algorithm within the LSI++ software environment is transparent to the user. As described by Letsche [Let96], the LSI++ query can be represented as a six tuple of the form

$$q = < b, r, n, t, d, f >,$$

where

$b$ indicates whether to return a ranked list of terms, documents, or both,

$r$ represents the full path to the document collection to be searched,

$n$ represents the number of terms, documents, or both to return,

$t$ represents a set of keywords (terms) used to specify the query,

$d$ represents a set of documents used in relevance feedback, and

$f$ is the current number of dimensions or factors used to represent the query, term and document vectors.

After the user has entered a query and the vectors representing terms and documents have been loaded from disk, the search for the $m$-closest terms and documents to the query is invoked. This search is accomplished through the following 4 steps:

1. Generate the query vector or *pseudo-document* with the query terms and relevance feedback documents as described in Section 2.5.

2. Build or load from disk the term kd-tree and document kd-tree as described in Section 2.3. If the user specifies that only documents should be returned, then only the document kd-tree is built or loaded from disk.

3. As illustrated in Section 2.4, traverse each kd-tree by comparing the query vector to the interior node cutting value and boundaries in the search for buckets containing term and document vectors which qualify as nearest neighbors to the query vector. When leaf nodes are reached, calculate the cosine between the query vector and every term or document in the bucket. If the user specifies that only documents should be returned, only the document kd-tree is traversed.

4. As a kd-tree(s) is traversed, maintain a linked list each of $m$-closest terms or documents to the query, where $m$ is specified by the user. Again, if the user specifies that only documents should be returned, then only one linked list is needed.

Step 1 is consistent with the original LSI++ specification described in [Let96]. In Step 2, available term and document trees built for the current number of LSI factors are loaded into memory from disk, avoiding the time and space expense of building new kd-trees. Figure 3.1 illustrates the query matching process described in the steps above.

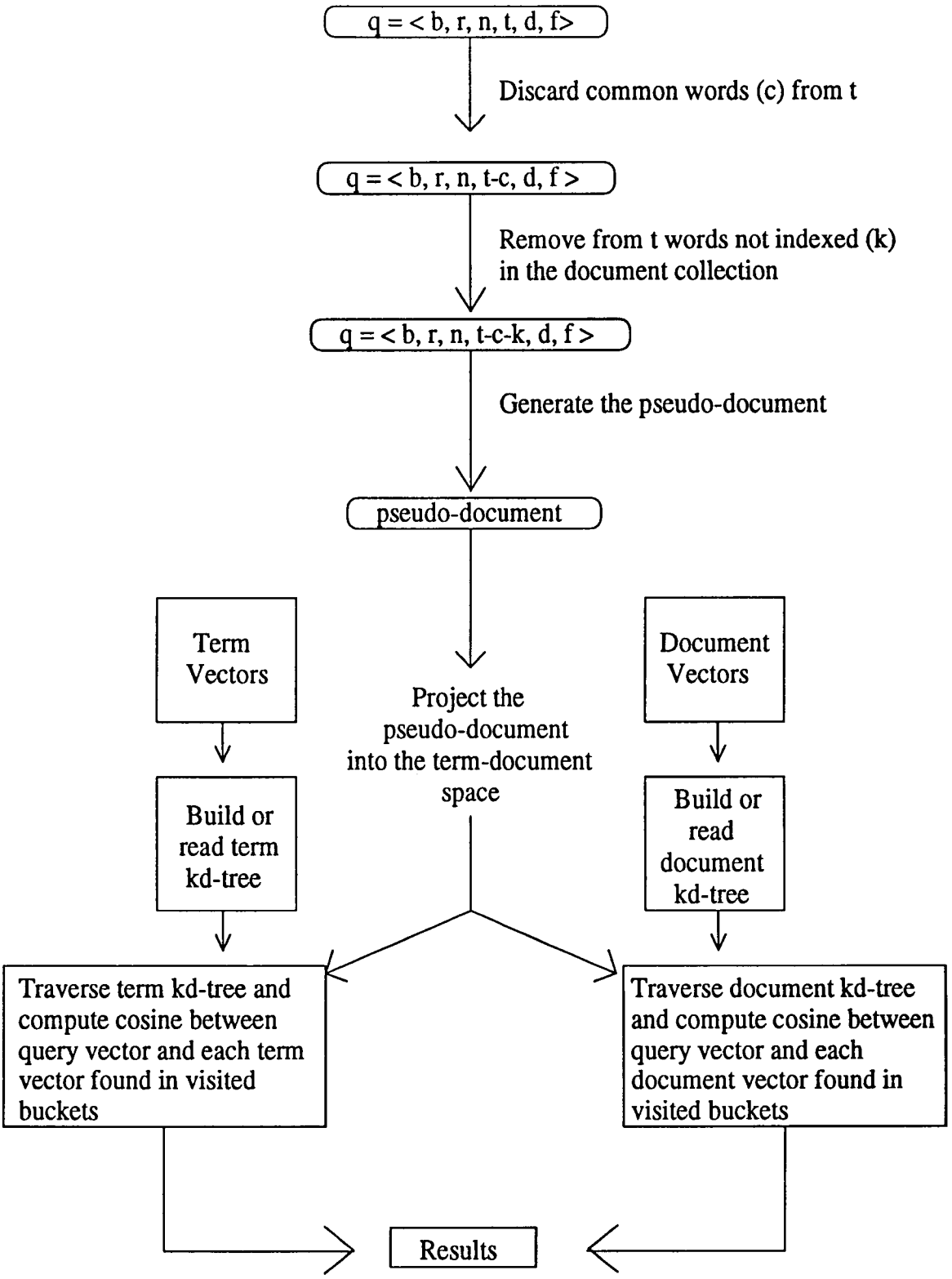


Figure 3.1: Query matching using the kd-tree algorithm [Let96].

## 3.2 kd-tree Implementation

The LSI++ software environment was developed in C++ to exploit the languages's encapsulation and inheritance features. The original LSI++ implementation provides two user interfaces: the client-server interface which is accessible from the World Wide Web (WWW) browser, and the command line interface.

Though the LSI++ software environment has both serial and distributed versions [Let96], the kd-tree algorithm was implemented only in the serial version. This serial version is designed to operate in-memory or out-of-memory depending on system requirements at the time of execution. When sufficient memory is available, all term and document vectors are loaded into memory prior to the query matching phase. Otherwise, the out-of-memory version of LSI++ requires that the term and document vectors be loaded into memory incrementally and discarded from memory later. The current kd-tree implementation of LSI++ does not make full use of the out-of-memory implementation, and so all term and document tree structures are built and stored in memory. However, the out-of-memory kd-tree implementation still reduces memory usage by storing the required singular values and singular vectors on disk. As in the original out-of-memory implementation [Let96], these floating-point arrays are loaded into memory incrementally and discarded after all query processing is completed.

The LSI++ information retrieval package was designed and implemented in C++ with both independent and inherited class structure as demonstrated in Figure 3.2. The imple-

mentation of the kd-tree algorithm required the addition of several independent C++ classes to the original LSI++ software. The classes and routines that implement the kd-tree algorithm are based on kd-tree software used in previous research by Arya and Mount [AM93, Ary95]. Where needed, the new classes were simply included as members of the various classes employed in the original LSI++ implementation. An overview of the kd-tree LSI++ class structure follows.

The *lsiServer* and *lsiSerial* classes implement the serial in-memory and serial out-of-memory versions of LSI++, respectively. These classes inherit from the *lsi* base class. The *query* class is the base class for the *coreQuery* and *outCoreQuery* classes. *CoreQuery* is an *lsiServer* class object while *outCoreQuery* is an *lsiSerial* class object. The *kd-tree*, *point* and *min_k* class objects have been added as new members of the *coreQuery* and *outCoreQuery* classes and collectively perform the kd-tree building, kd-tree searching and query matching functions. The *termVector*, *docVector*, and *singValues* classes provide the

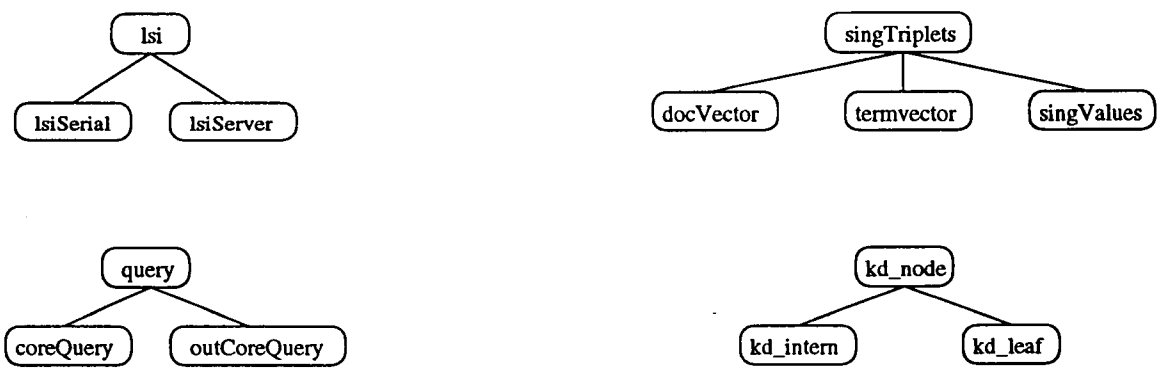


Figure 3.2: The LSI++ inheritance structure [Let96].

term and document vectors and singular values used in the query matching process. The base class *singTriplets* performs vector retrieval.

The addition of classes and functions to implement the kd-tree algorithm begins at the *lsiServer* and *lsiSerial* class level. For both classes, member function **writeTrees** determines the appropriate path and file names for the term and document trees so that they can be written to disk before the LSI++ program exits. **WriteTrees** calls the **writeDocTree** and **writeTermTree** member functions of the *coreQuery* and *outCoreQuery* classes described below.

The *coreQuery* and *outCoreQuery* objects are invoked within the *lsiServer* and *lsiSerial* classes, respectively. The *outCoreQuery* and *coreQuery* member functions translate the user query into a query vector and perform query matching. While the *coreQuery* class functions load all term, document, and singular vectors directly into memory, the *outCoreQuery* class functions reduce memory usage as discussed in Section 3.1. For both classes, member functions **writeDocTree** and **writeTermTree** open file pointers to the document and term kd-tree files, respectively, and pass them to the kd-tree member function **writeTree** so that the final term and document trees may be written to disk (if they do not already exist).

The *kd-tree* class is invoked within the *coreQuery* and *outCoreQuery* classes, and provides an entry point to the kd-tree data structure. Member function **Appx_k_Near_Neigh** initiates a kd-tree search for the nearest neighbors to a query. Member functions **getTree** and

**writeTree** are used to read and write a kd-tree from/to disk, respectively. The *kd-tree* class is specified as a friend class to the *kd_node* class, the base class of *kd-tree* elements, described below.

The *kd_node* class is the base class for the *kd_intern* and *kd_leaf* classes. The *kd_node* class defines the virtual member function **ann_search** used to traverse the interior and leaf nodes of the kd-trees. Class *kd_tree* described above is specified as a friend to the *kd_node* class.

The *kd_intern* and *kd_leaf* classes are inherited from the *kd_node* class and are invoked by the friend function **rkd_tree** during the kd-tree building phase. The *kd_intern* class is used to represent internal nodes within the kd-trees. Inherited function **ann_search** controls the traversal of internal nodes and determines which internal nodes should be visited by calculating the distance between the query and the cutting value of the current internal node as discussed in Section 2.4. This function also determines if the non-visited sibling of the current internal node should be visited by computing the distance between the query and the bounding box of the sibling node (see Section 2.4). The *kd_intern* class also defines the member function **writeNode** for writing internal nodes to disk. Friend function **read_in_tree** is called from an empty *kd_tree* root (an internal node) to continue the process of reading a kd-tree from disk.

The *kd_leaf* class is very similar to the *kd_intern* class with the exception that it allocates storage for term and document vectors. Class *kd_leaf*'s member functions **ann_search**

and **writeLeaf** are used to traverse and store (in memory) leaf nodes, respectively. Class *kd_leaf*'s inherited function **ann_search** is used to determine which, if any, of the current leaf node's term/document vectors should be added to the list of nearest neighbors by computing the cosine between the query vector and every term or document vector stored in the current leaf node. **WriteLeaf** writes the contents of a leaf node to disk. Storage for term or document vectors in a kd_leaf is provided by invoking the *point* class.

The *min_k* class implements the linked list structure that stores (in sorted order) the *m*-closest terms and documents to the query and invokes the *point* class to store term and document vectors. The *min_k* class is invoked as a static global variable during the kd-tree searching phase.

The *point* class is invoked within LSI++ whenever term, document, or query vector storage is needed. As a member of the *coreQuery* and *outCoreQuery* classes, the *point* class provides storage for each term and document vector, the query vector, and the *m*-closest term and document vectors to the query vector. During the kd-tree building phase, the *point* class is invoked by the recursive function **rkd_tree** to provide storage for a subset of vectors in a kd_leaf node. Table 3.1 lists member functions added to the LSI++ software environment to implement query matching based on the kd-tree algorithm.

There are several externally accessible functions utilized in the kd-tree building and searching process. Table 3.2 lists these functions and provides a brief description of each.

Table 3.1: Member functions added to implement the kd-tree algorithm.

| Member Function | Description |
|---|---|
| void **writeTrees**(String *termTreeName*, String *docTreeName*) | Generic function called before the server exits. Assigns file names to term and document trees to be written. Passes these file names to **writeDocTree** and **writeTermTree** functions to begin process of writing term and document kd-trees to disk. |
| void **writeDocTree**(ofstream *&docTreeFile*) | Opens the document kd-tree file for writing. Passes the file pointer to next level so document kd-tree data may be written to disk. |
| void **writeTermTree**(ofstream *&termTreeFile*) | Opens the term kd-tree file for writing. Passes the file pointer to next level so term kd-tree data may be written to disk. |
| void **writeNode**(ofstream *&outfile*, int *dim*) | Writes the contents of an internal kd-tree node to disk file. The variable dim is passed on to the **writeLeaf** function. |
| void **writeLeaf**(ofstream & *outfile*, int *dim*) | Writes the contents of a leaf node to disk file. Variable *dim* provides the dimension of the vectors stored in the leaf node being written. |
| void **ann_search**(float *box_dist*) | A generic recursive function defined as a virtual function in the *kd_node* class. It is inherited by the *kd_intern* and *kd_leaf* classes and defined specifically in each class to perform functions specific to each class. In general, **ann_search** controls the traversal of the internal and leaf nodes of the kd-tree. The variable *box_dist* represents the distance between the query and the bounding box for the current internal node and is used to determine if the non-visited sibling of the current internal node should be visited. |

Table 3.2: Externally accessible functions added to implement the kd-tree building and searching functions.

| Function | Description |
|---|---|
| void **encl_rect**(point **pa*, int *n*, int *dim*, point **bounds*) | Computes the smallest rectangle enclosing a subset set of points (vectors) *pa*. *n* represents the number of vectors in the subset, *dim* the dimension of each vector, *bounds* is a 2-dimensional *point* class array that stores the low and high values at each dimension for the subset of points. |
| void **max_spread**(point * *pa*, int *n*, int *dim*) | Finds the dimension of maximum spread in values for a subset of vectors *pa* containing *n* vectors of dimension *d*. |
| void **median_split**(point **pa*, int *n*, int *d*, float **cv*, int *n_lo*) | Splits a subset of vectors *pa* about it's median at dimension *d* to create two independent subsets of vectors. *cv* represents the returned splitting value. *n_lo* represents the index (or position within the subset of vectors) of the splitting value. |

# Chapter 4

# The Performance and Effectiveness of the kd-tree Algorithm

The original implementation of LSI++ provides accurate information retrieval in an efficient manner. Incorporating the kd-tree algorithm is an attempt to improve the response time performance of the LSI++ IR model by reducing the search time required to find the $m$-nearest neighbors to a query.

This chapter will evaluate both the performance and accuracy of the kd-tree implementation within the LSI++ software environment. The time and computational complexity of building and searching a kd-tree will be examined and compared with the time and computational complexity of the original LSI++ query matching method (i.e., computing the cosine between the query vector and every term and/or document vector in the term-document space). Finally, the accuracy of the kd-tree query matching method will be evaluated and compared to that of the original LSI++ query matching implementation.

## 4.1 Evaluation of Performance

The kd-tree query matching method was compared with the original LSI++ query matching method. Performance timings were obtained on a Sun Ultra1 SPARCstation computer with a 167 MHz processor, 32 KB on-chip cache (16 KB Instruction, 16 KB Data), 512 KB external. cache, and 256 MB of main memory. The three document collections used for performance timings are listed in Table 4.1. The MINIMED collection consists of a set of medical journal article titles [BF96]. The MEDLARS collection [DDF+90] is a larger set of medical journal article titles. The CCE document collection is an on-line version of the Concise Columbia Encyclopedia [cce89]. The MINIMED, MEDLARS, and CCE document collections represent a small, medium, and large document collection, respectively.

Each document collection was tested using the out-of-memory implementation of both the original LSI++ software and the kd-tree LSI++ software. Due to out-of-memory processing overhead (reading and writing from/to disk), the out-of-memory implementation of LSI++ requires more processing time than the in-memory version. However, for building and searching kd-trees, there is no difference between the in-memory and out-of-memory implementations with respect to computational complexity since kd-trees are constructed and searched in exactly the same way for both implementations. Consequently, the out-of-memory implementation of LSI++ was used to obtain the upper bounds on processing time for the LSI++ IR model.

Table 4.1: Document collections used to evaluate kd-tree algorithm performance.

| Document Collection | Number of Terms | Number of Documents | Max Number of Factors Tested |
|---|---|---|---|
| MINIMED | 18 | 16 | 15 |
| MEDLARS | 5831 | 1033 | 90 |
| CCE | 31110 | 15486 | 90 |

*LsiQuery* is the serial out-of-memory driver for both the original and kd-tree implementations of LSI++. *LsiQuery* takes a query as input and invokes the *lsiSerial* class as described in Section 3.2. In the kd-tree implementation of LSI++, the *lsiSerial* class initiates the kd-tree building and searching routines, and returns the query matching results to the user.

As discussed in [FBF77], the kd-tree building and searching time depends on the number of term/document vectors and the number of LSI factors. Since the type of vector (term or document) for which a kd-tree is built does not affect the time or computational complexity of building and searching a kd-tree, the performance timings shown below are those observed when only documents were returned.

Table 4.2: The number of LSI factors used for each document collection tested.

| | 5 | 10 | 15 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| MINIMED | X | X | X | | | | | | | | |
| MEDLARS | | X | | X | X | X | X | X | X | X | X |
| CCE | | X | | X | X | X | X | X | X | X | X |

Performance timings were measured in elapsed wall clock time for each document collection tested in both the original and kd-tree implementations of LSI++. For tests using the MEDLARS and CCE document collections, the initial number of LSI factors was 10 and then incremented by 10 up to a maximum of 90. For MINIMED experiments, the number of factors was set at 5, 10, and 15. Table 4.2 details the testing scenarios described above.

In order to validate the observed kd-tree construction and search times, an understanding of the computational complexity of such events is required. As proven by Friedman, Bentley, and Finkel [FBF77], the time complexity of constructing a kd-tree is proportional to $k\,N \log N$, where $k$ is the dimension of the vector space (i.e., the number of LSI factors) and $N$ is the number of term/document vectors stored in the kd-tree. The time complexity of searching a kd-tree is proportional to $\log N$ where $N$ is the number of term/document vectors stored in the kd-tree.

The observed timings show that query matching time is greatly improved by the kd-tree algorithm (see Tables 4.3 and 4.4). For example, the kd-tree algorithm returned the 5-nearest neighbors to the query 26 times faster (on average) using 30 LSI factors and 5 times faster (on average) using 90 LSI factors when compared to the original query match-

Table 4.3: The wall clock time in seconds for various numbers of factors when searching for the 5-nearest neighbors to a query in the MINIMED document collection using the original LSI++ query matching method and the kd-tree query matching method.

| MINIMED | | |
|---|---|---|
| Factors | Original | kd-tree |
| 5 | 0.000324 | 0.000163 |
| 10 | 0.0003629 | 0.000166 |
| 15 | 0.0004144 | 0.000177 |

Table 4.4: The wall clock time in seconds for various numbers of factors when searching for the 5-nearest neighbors to a query in the MEDLARS and CCE document collections using the LSI++ query matching method and the kd-tree query matching method.

| | MEDLARS | | CCE | |
|---|---|---|---|---|
| Factors | Original | kd-tree | Original | kd-tree |
| 10 | 0.0226870 | 0.00038505 | 0.3467361 | 0.00074303 |
| 20 | 0.0257340 | 0.00308299 | 0.4029010 | 0.00734699 |
| 30 | 0.0287220 | 0.00509691 | 0.4978090 | 0.01883800 |
| 40 | 0.0317629 | 0.00699198 | 0.5319180 | 0.03234510 |
| 50 | 0.0366569 | 0.00860500 | 0.5779950 | 0.04523000 |
| 60 | 0.0420419 | 0.01031900 | 0.5914341 | 0.08047310 |
| 70 | 0.0476560 | 0.01180990 | 0.6276500 | 0.10078700 |
| 80 | 0.0467719 | 0.01328600 | 0.6799350 | 0.11885200 |
| 90 | 0.0485239 | 0.01483090 | 0.7348529 | 0.14662400 |

ing method for the CCE document collection. The cost of this improvement, however, is the processing time to build the kd-tree. When querying the MEDLARS document collection, 88 percent (0.02 seconds) and 81 percent (0.06 seconds) of the total kd-tree processing time was utilized to build the document kd-trees for queries using 20 and 90 LSI. factors, respectively. For very large data sets (CCE), some processing time may be spared by building kd-trees in advance, storing them to disk, and reading the necessary kd-tree from disk during the query matching process. For example, when querying the CCE document collection, reading the kd-tree from disk was 3.2 and 4.5 times faster than building the document kd-trees for 10 and 90 LSI factors, respectively. For small to medium sized document collections, however, it is more efficient to build the kd-trees at the beginning of the query matching phase (see Table 4.5).

For all three document collections in all dimensions tested, the kd-tree searching algorithm always outperformed the original LSI++ query matching method. Timings observed for the MINIMED, MEDLARS, and CCE collections revealed average speed improvements of 2.2, 4.7 and 11.4, respectively (see Figure 4.1). As illustrated in Figure4.1, the greatest speed improvements are achieved for the smaller numbers of LSI factors and/or for larger document collections.

Table 4.5: The wall clock time in seconds for various numbers of factors when reading a kd-tree from disk and building a kd-tree from scratch for the MINIMED, MEDLARS and CCE document collections.

| | MINIMED | | MEDLARS | | CCE | |
|---|---|---|---|---|---|---|
| Factors | Build | Read | Build | Read | Build | Read |
| 5 | 0.0001711 | 0.0002060 | | | | |
| 10 | 0.0001600 | 0.0002240 | 0.0178779 | 0.0159160 | 0.49716 | 0.23780 |
| 15 | 0.0001780 | 0.0002660 | | | | |
| 20 | | | 0.0232339 | 0.0230050 | 1.11265 | 0.35102 |
| 30 | | | 0.0320081 | 0.0326310 | 1.71434 | |
| 40 | | | 0.0348260 | 0.0407951 | 2.37168 | 0.60457 |
| 50 | | | 0.0420420 | 0.0504421 | 2.81526 | 0.74159 |
| 60 | | | 0.0498980 | 0.0584990 | 3.63846 | |
| 70 | | | 0.0493780 | 0.0667111 | 3.93145 | 1.00664 |
| 80 | | | 0.0552551 | 0.0740150 | 4.95376 | 1.13338 |
| 90 | | | 0.0614200 | 0.0831549 | 5.40047 | 1.20446 |

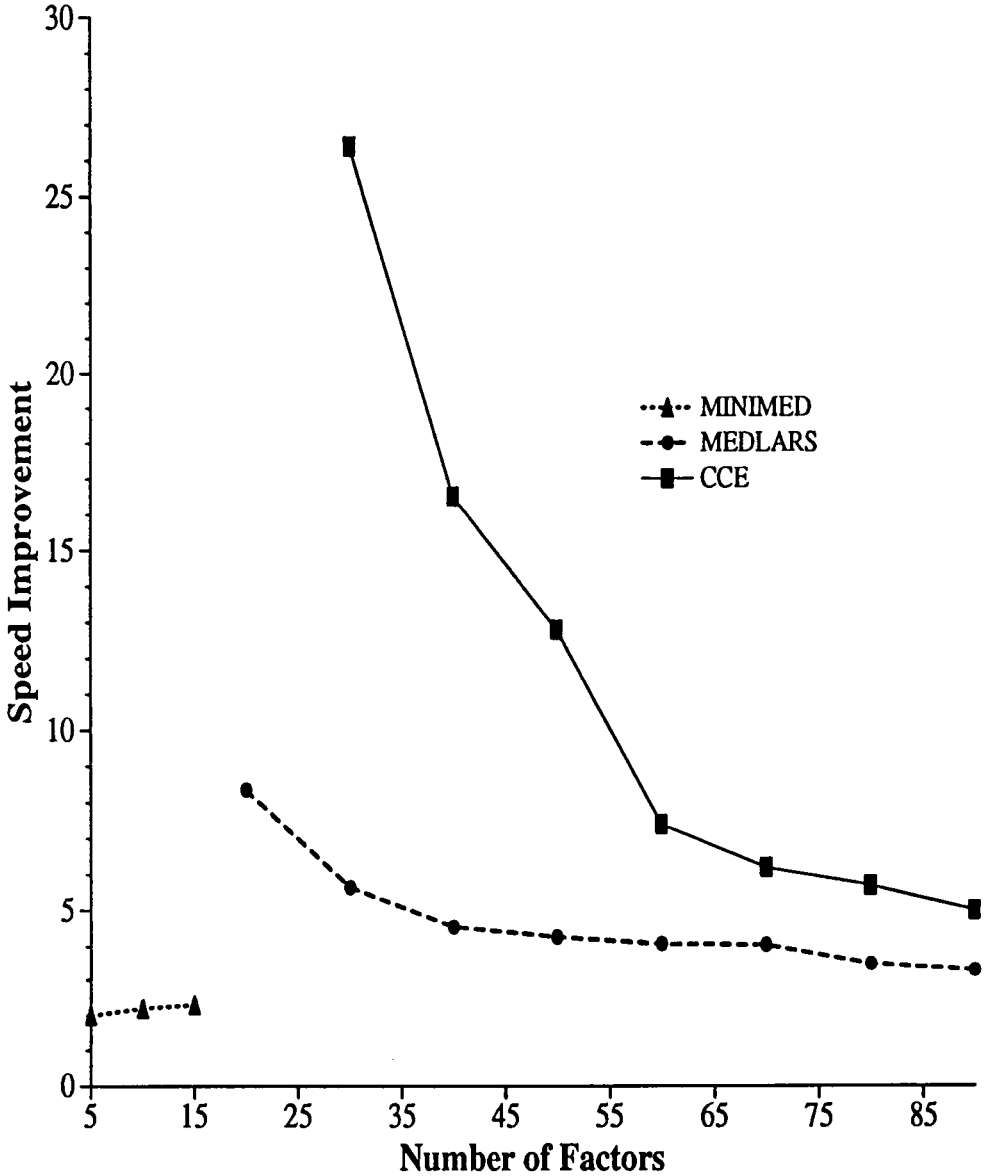


Figure 4.1: The speed improvement of the kd-tree algorithm over the original query matching method for the MINIMED, MEDLARS and CCE document collections.

## 4.2 Evaluation of Effectiveness

The effectiveness of the kd-tree implementation of LSI++ was evaluated in two ways. First, the *precision* and *recall* of query matching using the kd-tree algorithm was measured. Second, the results returned by the kd-tree query matching algorithm were compared to those returned by the original LSI++ query matching method for the same query. *Precision* is the ratio of relevant documents returned to the total number of documents retrieved for a given query. *Recall* is the ratio of the number of relevant documents retrieved to the total possible number of relevant documents for a given query [FBY92]. The relevance of a document is determined by pre-defined relevance judgements associated with a given document collection and a set of queries.

The MEDLARS document collection [DDF+90] includes relevance judgement data that lists specific queries and the documents within the MEDLARS collection that are, according to the relevance judgements, most pertinent to each query. With this data, an evaluation of the precision and recall, and hence effectiveness of the kd-tree query matching method is possible.

Figure 4.2 plots the *precision* versus *recall* observed for both the kd-tree algorithm and the original query matching method using the query **crystalline lens including humans**. Another example of the effectiveness of the kd-tree query matching algorithm is provided by comparing Tables 4.6 and 4.7 which list the results returned for the given query by the kd-tree implementation of LSI++ and the original implementation of LSI++, respectively.

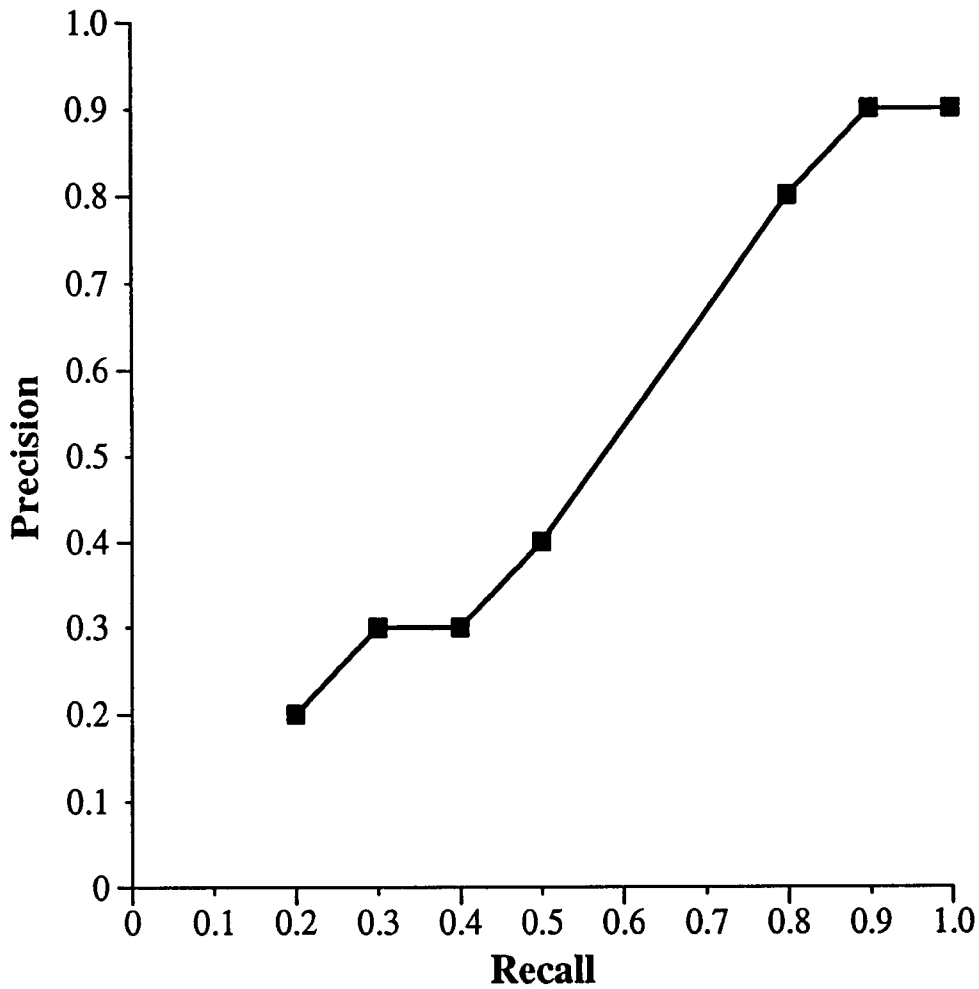


Figure 4.2: The precision and recall for the query **crystalline lens including humans** using the MEDLARS document collection.

Table 4.6: The top 5 documents returned by the kd-tree algorithm for the MEDLARS document collection using 50 LSI factors.

| Query: the crystalline lens in vertebrates including humans | | |
|---|---|---|
| Rank | Document number | Document title |
| 90 | 185 | the identification of lysosomal enzymes in bovine lens epithelium |
| 88 | 184 | an investigation of mitotic control in the rabbit lens epithelium |
| 87 | 142 | the effects of electrophoretically separated lens proteins on lens |
| 86 | 511 | the problem of albuminoid |
| 86 | 500 | studies on the soluble proteins of bovine lens |

Table 4.7 The top 5 documents returned by the original LSI++ query matching method for the MEDLARS document collection using 50 LSI factors

| Query: the crystalline lens in vertebrates including humans | | |
|---|---|---|
| Rank | Document number | Document title |
| 90 | 185 | the identification of lysosomal enzymes in bovine lens epithelium |
| 88 | 184 | an investigation of mitotic control in the rabbit lens epithelium |
| 87 | 142 | the effects of electrophoretically separated lens proteins on lens |
| 86 | 511 | the problem of albuminoid |
| 86 | 500 | studies on the soluble proteins of bovine lens |

# Chapter 5

# Conclusions and Future Work

The kd-tree algorithm does offer significant speed improvement over the original LSI++ approach for finding the most relevant terms and documents to a query. With the kd-tree algorithm, query matching in the LSI model can be accomplished in less time with the same level of retrieval effectiveness. For large document collections, kd-tree query matching was found to be 5 to 26 times faster than the original LSI query matching method (see Section 4.1). The price paid for this speed improvement is the time to build or read from disk the kd-tree structure. However, for consecutive queries of the same document collection with a fixed number of LSI factors, the kd-tree building/reading time is a one-time cost. Only when the user selects a new document collection and/or different number of LSI factors will time be spent building/retrieving a different kd-tree.

The implementation of the kd-tree algorithm within the LSI++ model could be further optimized. Improved query matching could be obtained by: reducing the time for building kd-trees by parallelizing the kd-tree search algorithm, reducing the storage requirements for kd-trees with more efficient use of pointers, combining term and document vectors into one kd-tree per collection/number of LSI factors, or overhauling the algorithm so that only one kd-tree per document collection is needed to find the nearest terms and documents to the query (irrespective of the number of LSI factors).

# Bibliography

# Bibliography


[AM93] Sunil Arya and David M. Mount. Algorithms for Fast Vector Quantization. Proceedings of DCC 1993: Data Compression Conference, IEEE Press, 381-390, 1993.

[Ary95] Sunil Arya. Nearest Neighbor Searching and Applications. PhD Dissertation, University of Maryland, College Park, MD, 1995.

[BC87] N. Belkin and W. Croft. Retrieval techniques. In M. Williams, editor, *Annual Review of Information Science and Technology (ARIST)*, Volume 22, Chapter 4, pages 109-145. Elsevier Science Publishers B.V., 1987.

[BDO95] Michael W. Berry, Susan T. Dumais, and G. O'Brien. Using Linear Algebra for Intelligent Information Retrieval. *SIAM Review*, 37(4):573-595, 1995.

[BF96] Michael W. Berry and Ricardo D. Fierro. Low-rank Orthogonal Decompositions for Information Retrieval Applications. *Numerical Linear Algebra with Applications*, 3(4):301-327, 1996.

[Ben75] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509-517, 1975.

[cce89] *The Concise Columbia Encyclopedia.* Columbia University Press, New York, second edition, 1989.

[DDF+90] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by Latent Semantic Analysis. *Journal of the American Society for Information Science*, 41(2):391-407, 1990.

[Dum91] Susan T. Dumais. Improving the Retrieval of Information from External Sources. *Behavior Research Methods, Instruments & Computers*, 23(2):229-236, 1991.

[FBF77] Jerome H. Friedman, John Louis Bentley, and Raphael Ari Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Transactions on Mathematical Software*, 3(3):209-226, 1977.

[FBY92] W. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures & Algorithms*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[GL96] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, third edition, 1996.

[LB97] Todd A. Letsche and Michael W. Berry. Large-Scale Information Retrieval with Latent Semantic Indexing. *Information Sciences*, 100:105-137, 1997.

[Let96] Todd A. Letsche. Toward Large-Scale Information Retrieval Using Latent Semantic Indexing. Master's Thesis, University of Tennessee, Knoxville, TN, 1996.

# Vita

Melanie Katherine Hughey was born in Atlanta, Georgia on December 8, 1970. She graduated from Shiloh High School in 1989 and received a Bachelor of Science in Geography in 1993 from the University of Georgia in Athens, Georgia. In August 1994, she entered the Computer Science graduate program at the University of Tennessee in Knoxville, Tennessee and received a Master of Science in Computer Science in May 1998.