UNIVERSITY OF CALIFORNIA
Santa Barbara

Querying Patterns in High-Dimensional Heterogenous Datasets

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Vishwakarma Singh

Committee in Charge:

Professor Ambuj K. Singh, Chair

Professor Amr El Abbadi

Professor Bangalore S. Manjunath

March 2012

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

The Dissertation of
Vishwakarma Singh is approved:

Professor Amr El Abbadi

Professor Bangalore S. Manjunath

Professor Ambuj K. Singh, Committee Chairperson

December 2011

Querying Patterns in High-Dimensional Heterogenous Datasets

To my parents, my wife, and my entire family.

Acknowledgements

First and foremost, I thank my advisor Prof. Ambuj K. Singh for his endless support and valuable guidance. I would also like to acknowledge the guidance and help of my other committee members: Prof. Amr El Abbadi and Prof. Bangalore S. Manjunath. I am greatly thankful to co-authors of my papers, the members of the Data Mining and Bioinformatics Lab (DBL), and the members of the Center for Bio-Image Informatics for their help. I am sincerely grateful to Prof. Steven K. Fisher, Chris Banna, and Geoffrey P. Lewis for providing biological data. The role of all my teachers in my life is highly appreciable.

I cannot thank my parents enough for everything they have done for me. I would not have been here without their love, support, and vision. I am also infinitely grateful to my wife, Sarita Sarba Nand Singh, without whose love and support this would have been impossible.

Curriculum Vitæ
Vishwakarma Singh

Education

2012	*Doctor of Philosophy (Ph.D.)* Dept. of Computer Science, University of California, Santa Barbara, California, USA.
2011	*Master of Science (M.S.)* Dept. of Computer Science, University of California, Santa Barbara, California, USA.
2003	*Bachelor of Technology (B.Tech.)* Dept. of Computer Science and Engineering, Institute of Technology, Benaras Hindu University, Varanasi, India.

Experience

2009-2011	*Research Assistant* Dept. of Computer Science, University of California, Santa Barbara, California, USA.
2009	*Summer Intern* Dolby Laboratories (Research), San Francisco, California, USA.
2008-2009	*Research Assistant* Dept. of Computer Science, University of California, Santa Barbara, California, USA.
2007	*Teaching Assistant* Dept. of Computer Science, University of California, Santa Barbara, California, USA.
2007	*Summer Intern* Nokia Research Center, Cambridge, Massachusetts, USA.

2006-2007	*Research Assistant* Dept. of Computer Science, University of California, Santa Barbara, California, USA.
2005-2006	*Senior Software Engineer* CenturyLink, Bangalore, India.
2004-2005	*Applications Engineer* Oracle, Hyderabad, India.
2003-2004	*Associate Consultant* Oracle Financial Services Software Ltd., Mumbai, India.

Awards

2008	Award for web based education portal "Click2School", ICS Technology HITEC Entrepreneurship Competition-2008, University of California, Irvine.
2001-2002	Merit award, Institute of Technology, Benaras Hindu University, India.
1999-2003	University Grants Commission merit scholarship, India.

Publications

2012	Finding Skyline Nodes in Large Networks. Arijit Khan, Vishwakarma Singh, and Jian Wu. To appear in *the 3rd International Workshop on Graph Data Management: Techniques and Applications (GDM)*, 2012, Washington, DC, USA.
2011	ProMiSH: Nearest Keyword Set Search in Very High Dimensional Spaces. Vishwakarma Singh and Ambuj K. Singh. *Submitted for review.*
2011	SIMP: Accurate and Efficient Near Neighbor Search in Very High Dimensional Spaces. Vishwakarma Singh and Ambuj K. Singh. *Submitted for review.*
2010	Querying Spatial Patterns. Vishwakarma Singh, Arnab Bhattacharya, and Ambuj K. Singh. *International Conference on Extending Database Technology (EDBT)*, 2010, pages 418-429, Lausanne, Switzerland.

2010	Efficient and Robust Detection of Duplicate Videos in a Large Database. Anindya Sarkar, Vishwakarma Singh, Pratim Ghosh, B. S. Manjunath, and Ambuj K. Singh. *IEEE Transactions on Circuits and Systems for Video Technology (CSVT)*, 2010, Volume 20, Issue 6, pages 870-885.
2010	Geo-Clustering of Images With Missing GeoTags. Vishwakarma Singh, Sharath Venkatesha, and Ambuj K. Singh. *IEEE International Conference on Granular Computing (GrC)*, 2010, pages 420-425, San Jose, USA.
2010	An Algorithm and Hardware Design for Very Fast Similarity Search in High Dimensional Space. Vishwakarma Singh and Wenyu Jiang. *IEEE International Conference on Granular Computing (GrC)*, 2010, pages 426-431, San Jose, USA.
2010	Profile Based Sub-Image Search in Image Databases. Vishwakarma Singh and Ambuj K. Singh. *UCSB Technical Report, 2010-20*.
2008	Efficient Computation of Statistical Significance of Query Results in Databases. Vishwakarma Singh, A. Bhattacharya, and Ambuj K. Singh. *Proceedings of the 20th international conference on Scientific and Statistical Database Management*, 2008, pages 509-516, Hong Kong, China.

Abstract

Querying Patterns in High-Dimensional Heterogenous Datasets

Vishwakarma Singh

The recent technological advancements have led to the availability of a plethora of heterogenous datasets, e.g., images tagged with geo-location and descriptive keywords. An object in these datasets is described by a set of high-dimensional feature vectors. For example, a keyword-tagged image is represented by a color-histogram and a word-histogram. Analyzing these datasets gives better insights into the processes generating the datasets, opens new frontiers of scientific research, and fuels development of life-changing products.

An effective mechanism for exploring these heterogenous datasets is querying. One such kind of query is a pattern query. Given a heterogenous dataset and a query, the task here is to find a set of objects which are constrained by a relationship and satisfy the query. For example, given a dataset of keyword-tagged objects, a useful pattern query is to find a set of similar objects that contains a given set of keywords.

Querying patterns in high-dimensional heterogenous datasets brings about a new set of computational challenges. High performance algorithms to efficiently

and accurately query patterns are presented in this thesis. First, a scalable algorithm, SIMP, is described for accurately querying near neighbors in a high-dimensional dataset. SIMP significantly outperforms the state-of-the-art techniques. Next, a novel algorithm, ProMiSH, is proposed for efficiently querying patterns by keywords. ProMiSH has a speed-up of more than four orders over the state-of-the-art techniques. Then, an algorithm, QUIP, is described for querying patterns by example in a spatial dataset, e.g., geographical maps. QUIP offers an improvement of 87% in running time over the baseline approach. Next, an algorithm for querying patterns by example in a temporal dataset is described. It specifically solves the problem of finding duplicate videos. The proposed algorithm yields a practical query time for video duplicate detection. Finally, a scalable method to compute statistical significance of results of a multi-object query is discussed. Statistical significance or *p-value* provides a more useful criterion for ranking the results of a query.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The recent technological advancements have helped in a fast-paced generation of high-dimensional heterogenous datasets. For example, mobile computing devices have simplified the generation of images and videos tagged with geo-location, timestamp, and descriptive keywords. Similarly, high-throughput imaging devices have led to the creation of large maps of biological and geographical systems, which are often annotated with descriptive keywords and spatial relationships. An object in these datasets is represented by a high-dimensional feature vector. For example, a grayscale image is represented by a 256-dimensional color-histogram. Similarly, a document is represented by a histogram of hundreds of words.

Many of these datasets are heterogenous, i.e., the features of an object in the dataset are not directly comparable. For example, a keyword-tagged image has two kinds of features: a visual feature and a text feature. These two features of an image are not directly comparable, and hence the dataset is heterogenous. Simi-

larly, a dataset of geo-tagged news articles is heterogenous. The word-histogram and the geo-location of the news article are not directly comparable.

These heterogenous datasets are rich sources of valuable information. An effective mechanism for exploring these datasets is querying. One such kind of query is a pattern query. Given a heterogenous dataset and a query, the task here is to find a set of objects which are constrained by a relationship and satisfy the query. The retrieved set of objects which are constrained by a relationship is called a *pattern*.

An example of a pattern query is described next. Consider a dataset of geographical maps. These maps, being very large, are split into tiles of fixed size. Each tile is represented by a high-dimensional visual feature vector and its spatial location in the map. A query, as shown in Figure 1.1(a), consists of a set of tiles. The goal here is to retrieve a connected set of tiles which is visually similar to the query. A result is shown by a red box in Figure 1.1(b). The set of tiles in the result satisfies the connectivity relationship, and hence forms a pattern.

Three kinds of pattern queries are described in this thesis. The first query finds a set of keyword-tagged neighboring objects in a multi-dimensional feature space that contains a given set of keywords. This paradigm of querying patterns is called *querying patterns by keywords*. The second query finds a set of connected objects similar to a given set of connected objects in a spatial dataset. The

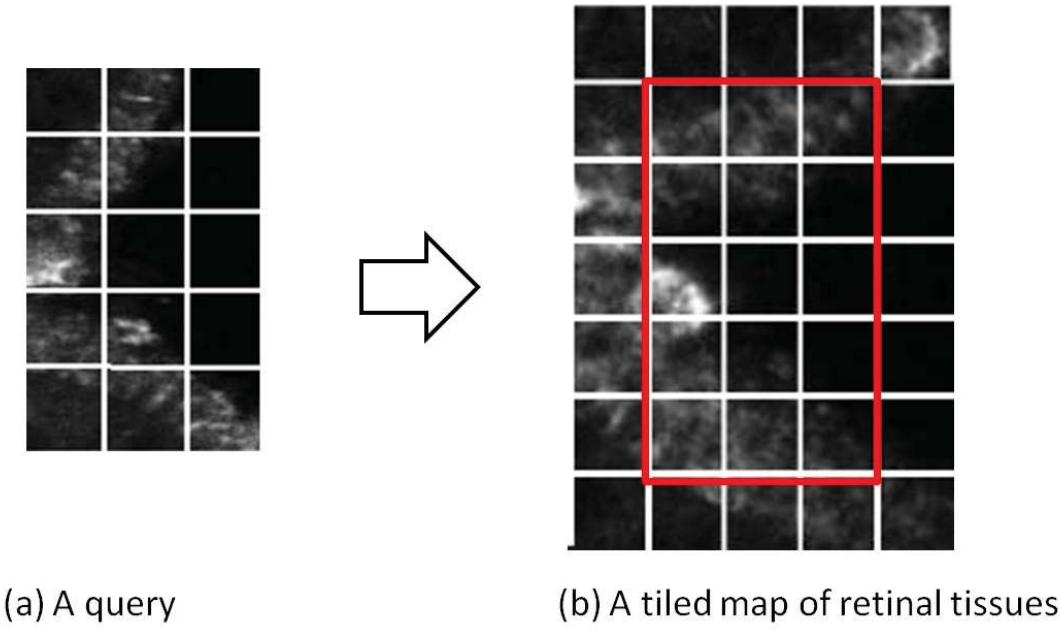


Figure 1.1: (a) A query formed by a spatial arrangement of the tiles of retinal maps. (b) A dataset of tiles of retinal maps. The map shows retinal tissue labeled with peanut-agglutinin conjugated to a fluorescent probe. The set of tiles bounded by the red box is a match for the query.

third query finds a set of time-sequenced objects similar to a given set of time-sequenced objects in a temporal dataset. The last two queries fall into the category of *querying patterns by example*.

To answer the pattern queries discussed above, there is a need to develop retrieval mechanisms. Typically, each object in the dataset is transformed into a multi-dimensional feature vector. Then, a measure of similarity between objects is designed. Next, all the vectors in the dataset are stored in an index. A given query object is also transformed into a feature vector. Finally, objects similar to the query object are efficiently retrieved from the dataset using the index.

An example of a typical search application, a map service, is described next. A map service helps to find a location nearest to a query location. Here, each location in the dataset is represented by a vector of its latitude and longitude. All the locations in the dataset are indexed into an R-Tree [52]. Nearness between two locations is measured by their Euclidean distance. A query location is also represented by its latitude and longitude. Finally, the location nearest to the query location is obtained by a best-first search [56] using the R-Tree [52].

The querying technique described above, though highly successful for many domains, suffers from many limitations. First, it works only for datasets of a single modality, e.g., a dataset of color-histograms of images. Second, it supports only pair-wise object comparisons. Third, it answers only single-object queries,

e.g., querying documents similar to a given document. Fourth, the state-of-the-art indexing techniques [11, 26, 52] retrieve results accurately and efficiently only for datasets of dimensions up to 20 [128]. In addition, the methods [49, 59] giving efficiency for high-dimensional datasets fail to guarantee 100% result accuracy.

The limitations of the state-of-the-art methods, as discussed above, make them unsuitable for querying patterns in high-dimensional heterogenous datasets. A method for querying patterns should handle heterogenous datasets, should support similarity between sets of objects, and should provide both efficiency and accuracy in high-dimensions. Therefore, there is a need to develop new methods to answer pattern queries in high-dimensional heterogenous datasets.

1.1 Thesis Statement and Contributions

In this thesis, it has been shown that: *"Patterns can be queried accurately and efficiently in high-dimensional heterogenous datasets."*

Novel index structures and search algorithms for efficiently and accurately querying patterns are presented in this thesis. The performance of the proposed algorithms was validated using a set of metrics on multiple high-dimensional real datasets. A brief summary of all the queries discussed in the thesis is given in Table 1.1.

Chapter No.	Query types	Description
2	r-near neighbor queries	The dataset consists of objects represented by multi-dimensional feature vectors. The dimension of the dataset ranges between 32 and 256. A query consists of an object and a radius r. The result is the collection of all the points within a distance r from the query object.
3	Querying patterns by keywords	Each object in the dataset is represented by a multi-dimensional feature vector and a set of keywords. The dimension of the dataset ranges between 2 and 100. A query consists of a set of keywords. A result is a set of neighboring objects in the multi-dimensional feature space that contains all the query keywords.
4	Querying patterns by example in a spatial dataset	Each object in the dataset is represented by a multi-dimensional feature vector and a spatial location. The dimension of the dataset ranges between 3 and 13. A query consists of a set of connected objects as shown in Figure 1.1(a). A result is a connected set of objects which is similar to the query as shown in Figure 1.1(b).
5	Querying patterns by example in a temporal dataset	The dataset consists of time-sequenced objects where each object is represented by a multi-dimensional feature vector. The dimension of the dataset ranges between 1 and 8192. A query is a set of time-sequenced objects. A result is again a set of time-sequenced objects that is similar to the query. An example of such a query is video duplicate detection.

Table 1.1: A brief description of all the query types discussed in this thesis.

Near neighbor queries form a vital step in many pattern mining and querying tasks. A novel technique, SIMP, is discussed in Chapter 2 for querying r-near neighbors in very high-dimensional spaces. SIMP efficiently queries r-near neighbors for all the query ranges with 100% quality guarantee. SIMP creates multiple 2-dimensional projections of the data space relative to random points. It uses a

hash-based approach to take an intersection of the 2-dimensional projections to efficiently determine the neighbors. Empirical comparisons on three real datasets of dimensions between 32 and 256 and sizes up to 10 million show a superior performance of SIMP over LSH [31], Multi-Probe LSH [87], LSB tree [122], and iDistance [60]. Scalability tests on real datasets of dimensions up to 256 and sizes up to 100 million establish that SIMP scales linearly with the query range, the dataset dimension, and the dataset size.

Querying patterns by keywords in a dataset of keyword-tagged objects is introduced in Chapter 3. Each object is represented by a multi-dimensional feature vector and a set of keywords. A new algorithm, ProMiSH, is proposed for querying a set of neighboring objects in a multi-dimensional feature space that contains a given set of query keywords. ProMiSH uses random projections and hash-based index structures to query results, and achieves high scalability and speed-up. Empirical studies, both on real and synthetic datasets, show that ProMiSH has a speed-up of more than four orders over the state-of-the-art tree-based techniques. Scalability tests on datasets of sizes up to 10 million and dimensions up to 100 for queries of sizes up to 9 show that ProMiSH scales linearly with the dataset size, the dataset dimension, the query size, and the result size.

Querying patterns by example in a spatial dataset is presented in Chapter 4. Each object is represented by a multi-dimensional feature vector and a spatial

location. Given a spatial pattern, the task here is to retrieve similar patterns from the dataset. A novel algorithm, QUIP, is described for querying the spatial patterns. QUIP uses a dynamic programming based scoring scheme to measure the similarity between patterns. QUIP has two index-based scalable search strategies: TARS and SPARS. Experimental results on real image datasets show that TARS offers an 87% improvement for small queries, and SPARS a 52% improvement for large queries in running time, as compared to the baseline approach. Qualitative tests on real datasets achieve precision of more than 80%.

Querying patterns by example in a temporal dataset is discussed in Chapter 5. It specifically solves the problem of querying duplicate videos. First, a new non-metric distance measure is proposed to find the similarity between a query and a database video. Then, a novel search algorithm based on pre-computed distances and pruning techniques is described to efficiently find duplicate videos. Experiments on a database of $38,000$ videos, worth $1,600$ hours of content, show that the duplicate videos for queries of duration 60 seconds are retrieved in 0.032 seconds with a high accuracy of 97.5%.

Queries, such as database similarity searches, return results satisfying certain properties of distances or scores. For domain scientists, the absolute values of scores are seldom sufficient. Statistical significance or *p-value* of the result is a more useful criterion. An efficient method to calculate the approximate p-value

of a multi-object result, when the distribution of scores for the database objects is non-parametric, is presented in Chapter 6. Experimental evaluations on large databases show that the method is practical, runs five orders of magnitude faster than the basic approach, and has an error of less than 5% in p-value computation.

This thesis concludes with a summary of the proposed pattern queries and the corresponding retrieval methods. An insight into the impact of this research work and challenges of the future tasks is also provided in the conclusions.

Chapter 2

SIMP: Accurate and Efficient Near Neighbor Search in High-Dimensional Spaces

Search for near neighbors of a given query object in a collection of objects is a fundamental operation in numerous applications, e.g., multimedia similarity search, data mining, information retrieval, and pattern recognition. The most common model for search is to represent objects in a high-dimensional attribute space, and then retrieve points near a given query point using a distance measure. In this chapter, we specifically study the problem of r-near neighbor (r-NN) queries. These queries retrieve all the points within a distance r from the query point. Near neighbor search in high-dimensional spaces still remains an open problem.

Existing techniques solve this problem efficiently only for the approximate cases [31, 49, 122]. These solutions are designed to solve r-near neighbor queries

for a fixed query range or a set of query ranges with probabilistic guarantees, and then extended for nearest neighbor queries. Solutions supporting a set of query ranges suffer from prohibitive space cost [49]. There are many applications which are quality sensitive and need to efficiently and accurately support near neighbor queries for all query ranges [119, 86, 18, 120]. In this chapter, we propose a novel indexing and querying scheme called *Spatial Intersection and Metric Pruning* (SIMP). It efficiently supports r-near neighbor queries in very high-dimensional spaces for all query ranges with 100% quality guarantee and with practical storage costs. Our empirical studies on three real datasets having dimensions between 32 and 256 and sizes up to 10 million show a superior performance of SIMP over LSH, Multi-Probe LSH, LSB tree, and iDistance. Our scalability tests on real datasets having as many as 100 million points of dimensions up to 256 establish that SIMP scales linearly with the query range, the dataset dimension, and the dataset size.

2.1 Introduction and Motivation

Let $\mathcal{U}$ be a dataset of N points in a d-dimensional vector space $\mathcal{R}^d$. Let $d(.,.)$ be a distance measure over $\mathcal{R}^d$. An r-NN query is defined by a query point $q \in \mathcal{R}^d$ and a search range r from q. It is a range query whose result set contains all the

points p satisfying $d(q, p) \leq r$. An r-NN query is useful for constructing near neighbor graphs, mining of collocation patterns [113], and mining density based clusters [39]. An r-NN query can also be repeatedly used with increasing query ranges, starting with an expected range, to solve the nearest neighbor family of problems [49, 60, 73].

In order to achieve a practical performance in query processing, data points are indexed and searched using an efficient data structure. A good index should be space and time efficient, should yield accurate results, and should scale with the dataset dimension and size. There are many well-known indexing schemes in the literature, mostly tree-based [11, 26, 52], for efficiently and exactly solving near neighbor search in low dimensions. It is known from the literature that the performances of these methods deteriorate and become worse than sequential search for sufficiently large number of dimensions due to the curse of dimensionality [128]. iDistance [60] is a state-of-the-art method for an exact r-NN search. It has been shown to work well for datasets of dimensions as high as 30. A major drawback of iDistance is that it works well only for clustered data. It incurs expensive query costs for other kinds of datasets and for very high-dimensional datasets.

There are many applications which need efficient and accurate near neighbor search in very high dimensions. For example, content based multimedia similarity search uses state-of-the-art 128-dimensional SIFT [86] feature vectors. Another

example is DNA sequence matching which requires longer seed length (typically 60-80 bases) to achieve higher specificity and efficiency while maintaining sensitivity to weak similarities [18]. A common practice is to use dimensionality reduction [110] techniques to reduce the dimensions of the dataset before using an index structure. These techniques being lossy transformations do not guarantee optimal quality. These techniques are also not useful for a number of datasets, e.g., strings [139] and multimedia [86, 87], which have intrinsically high dimensionality.

State-of-the-art techniques for r-NN search in very high dimensions trade-off quality for efficiency and scalability. Locality Sensitive Hashing (LSH) [59, 49] is a state-of-the-art method for approximately solving r-NN query in very high dimensions. LSH, named *Basic*-LSH here, solves (r_0, ϵ)-neighbor problem for a fixed query range r_0. It determines whether there exists a data point within a distance r_0 of query q, or whether all points in the dataset are at least a distance $(1+\epsilon)r_0$ away from q. In the first case, *Basic*-LSH returns a point within distance at most $(1+\epsilon)r_0$ from q. *Basic*-LSH constructs a set of L hashtables using a family of hash functions to fetch near neighbors efficiently. *Basic*-LSH is extended, hereby named *Extended*-LSH, to solve ϵ-approximate nearest neighbor search by building several data structures for different values of r. *Extended*-LSH builds a set of L hashtables for each value of r in $\{r_0, (1+\epsilon)r_0, (1+\epsilon)^2 r_0, \cdots, r_{max}\}$, where r_0 and

r_{max} are the smallest and the largest possible distance between the query and the data point respectively.

LSH based methods and their variants, though efficient and scalable, lack 100% quality guarantee because of their probabilistic nature. In addition, they are unable to support queries over flexible ranges. *Basic*-LSH is designed for a fixed query range r_0, and therefore yields poor result quality for query ranges $r > r_0$. *Extended*-LSH suffers from prohibitive space costs as it maintains a number of index structures for different values of r. The space usage of even *Basic*-LSH becomes prohibitive for some applications like biological sequence matching [18] where a large numbers of hashtables are required to get a satisfactory result quality. Recently, Lv et al. [87] improved *Basic*-LSH to address its space issue with a novel probing sequence, called Multi-Probe LSH. However, Multi-Probe LSH does not give any guarantee on quality or performance. Tao et al. [122] proposed a B-tree based index structure, LSB tree, to address the space issue of *Extended*-LSH and the quality issue of *Basic*-LSH for query ranges which are any power of 2. Nonetheless, LSB tree is an approximate technique with a high space cost.

We find that none of the existing methods simultaneously offer efficiency, 100% accuracy, and scalability for near neighbor queries over flexible query ranges in very high-dimensional datasets. These properties are of general interest for any search system. In this chapter, we propose a novel in-memory index structure

and querying algorithm called **SIMP** (**S**patial **I**ntersection and **M**etric **P**runing). It efficiently answers r-NN queries for any query range in very high dimensions with 100% quality guarantee and has practical storage costs. SIMP adopts a two-step pruning method to generate a set of candidate near neighbors for a query. Then it performs a sequential search on these candidates to obtain the true near neighbors.

The first pruning step in SIMP is named *Spatial Intersection Pruning*(**SIP**). SIMP computes multiple 2-dimensional projections of the high-dimensional data. Each projection is computed with respect to a different reference point. SIMP partitions each 2-dimensional projection into grids. It also computes the projection of the query answer space. SIMP generates a set of candidates for a r-NN query by an intersection of the 2-dimensional grids and the projection of the query answer space. It preserves all the true neighbors of a query by construction. A hash based technique is used in this step to gain space and query time efficiency. The second step of pruning is called *Metric Pruning* (**MP**). SIMP partitions the dataset into tight clusters. It uses triangle inequality between a candidate, candidate's nearest cluster center, and the query point to further filter out false candidates.

We also design a statistical cost model to measure the performance of SIMP. We show a superior performance of SIMP over state-of-the-art methods iDistance and p-stable LSH on three real datasets having dimensions between 32 and 256

and sizes up to 10 million. We also compared SIMP with Multi-Probe LSH and LSB tree on two real datasets of dimensions 128 and 256 and sizes 1 million and 1.08 million respectively. We observed that SIMP comprehensively outperforms both these methods. Our scalability tests on real datasets of sizes up to 100 million and dimensions up to 256 show that SIMP scales linearly with the query range, the dataset dimension, and the dataset size.

Our main contributions are: (a) a novel algorithm that solves r-NN queries for any query range with 100% quality in a very high-dimensional search space; (b) statistical cost modeling of SIMP; and (c) extensive empirical studies. We discuss related work in Section 2.2. We develop our index structure and query algorithm in Section 2.3. A statistical cost model of SIMP is described in Section 2.4. We present experimental results in Section 2.5. We describe schemes for selecting the parameters of SIMP in Section 2.6.

2.2 Literature Survey

Near neighbor search is well solved for low dimensional data (usually less than 10). Gaede et al. [44] and Samet et al. [110] present a survey of these multidimensional access methods. All the indexing schemes proposed in the literature fall into two major categories: space partitioning and data partitioning. Berchtold

et al. [13] partition the space using a Voronoi diagram and answer a query by searching for the cell in which the query lies. Space partitioning trees like KD-Tree recursively partition the space on different dimensions. Data partitioning techniques like R-Tree [52], M-Tree [26] and their variants enclose relatively near points in Minimum Bounding Rectangles or Spheres and recursively build a tree. The performance of these techniques deteriorates rapidly with an increase in the number of data dimensions [128, 17].

For very high dimensions, space filling curves [76] and dimensionality reduction techniques are used to project the data into low dimensional space before using an index. Weber et al. [128] proposed VA-file to compress the dataset by dimension quantization and minimize the sequential search cost. Jagadish et al. [60] proposed **iDistance** to exactly solve r-NN queries in high dimensions. The space is split into a set of partitions and a reference point is identified for each partition. A data point p is assigned an index key based on its distance from the nearest reference point. All the points are indexed using their keys in a B+-Tree. iDistance performs well for clustered data of dimensions up to 30. The metric pruning of SIMP is inspired from this index structure.

Near Neighbor search is efficiently but approximately solved in very high dimensions [49, 5]. Locality sensitive hashing (LSH) proposed by Indyk et al. [59] provides a sub-linear search time and a probabilistic bound on the result qual-

ity for approximate r-NN search. LSH uses a family of hash functions to create hashtables. It concatenates hash values from k hash functions to create a hash key for each point. It uses L hashtables to improve the quality of search. LSH hash functions put nearby objects in the same hashtable bucket with a higher probability than those which are far apart. One can determine near neighbors by hashing the query point and retrieving elements stored in the bucket containing the query. Many families of hash functions have been proposed [49, 22, 31, 4] for near neighbor search. **p-stable LSH** [31] uses vectors, whose components are drawn randomly from a p-stable distribution, as a family of hash functions for l_p norm. As discussed in Section 2.1, LSH suffers from the quality and the space issues.

Many improvements and variants [9, 97, 101, 87, 37, 63, 122, 78, 118] have been proposed for the LSH algorithm to solve the approximate near neighbor search. Lv et al. [87] proposed a heuristic called **Multi-Probe LSH** to address the space issue of *Basic*-LSH [49]. They designed a novel probing sequence to look up multiple buckets in hashtables of *Basic*-LSH. These buckets have a high probability of containing the near neighbors of a query. Multi-Probe LSH does not have any quality guarantee and may need a large number of probes to achieve a desired quality, thus making it inefficient.

A B-tree based index, called **LSB** tree (a set of LSB trees is called an LSB forest), was proposed by Tao et al. [122] for near neighbor search in relational databases to simultaneously address the space issue of *Extended*-LSH [59] and the quality issues of *Basic*-LSH for query ranges which are any power of 2. Each d-dimensional point is transformed into an m-dimensional point by taking its projection on m p-stable hash functions similar to p-stable LSH [31]. Points are indexed using a B-Tree on their z-order values which are obtained by partitioning the m-dimensional space with equi-width bins. Near neighbor search is carried by obtaining points based on the length of the longest common prefix of z-order. LSB tree is an approximate technique with weak quality guarantees and can have prohibitive costs. It is noteworthy that all the LSH based techniques create index structures independent of the data distribution.

A cost model for near neighbor search using partitioning algorithms was provided by Berchtold et al. [12]. An M-Tree cost model was presented by Ciaccia et al. [27]. Weber et al. [128] and Böhm et al. [17] developed these ideas further.

2.3 Algorithm

We develop the idea of SIMP using a dataset $\mathcal{U}$ of N points in a d-dimensional vector data space $\mathcal{R}^d$. Each point p has a unique identifier. We use Euclidean

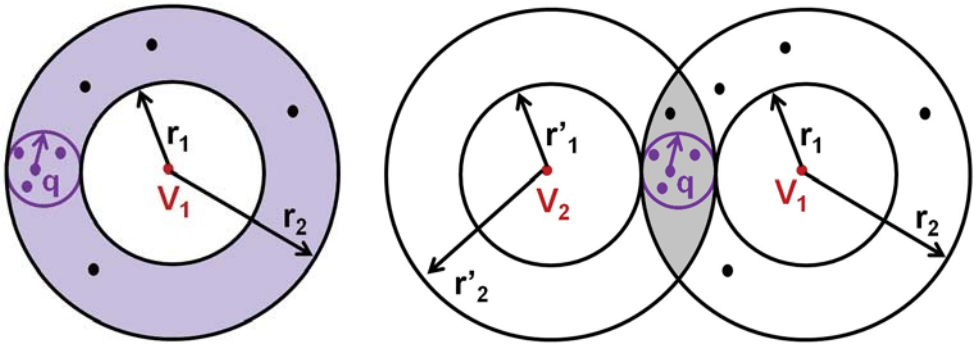
$q(r_q)$: A query with point q and search range r_q
v: A viewpoint
n_v: Number of viewpoints
$\mathcal{G}(v)$: Polar grid of viewpoint v
N: Number of data points in $\mathcal{U}$
$h(s)$: A hashtable of SIMP
p: A data point
$\mathbf{o}$: Origin of the data space
s: Signature of a hashtable
θ: Angle of a data point p relative to a viewpoint v and its angular vector ov
S: Percentage of the data points obtained by an algorithm as candidate for a query (selectivity)
C: Candidate set obtained by an algorithm for a query
r: Distance of a point p from a viewpoint v
b: Bin id of a point in a polar grid
P: Number of probes used by Multi-Probe LSH
L: Number of hashtables in the index structure
k : Size of a hash signature s
d: Dimension of the dataset
w_r: Radial width between rings of a polar grid
w_θ: Angle between radial vectors of a polar grid
n_z: Number of mballs used for Metric Pruning

Table 2.1: A descriptive list of notations used in the chapter.

metric to measure the distance $d(.,.)$ between a pair of points in $\mathcal{R}^d$. We take $\mathbf{o}$ as the origin of the data space. An r-NN query $q(r_q)$ is defined by the query point q and the search range r_q. The answer set of the query $q(r_q)$ contains all the points of the dataset $\mathcal{U}$ which lie within a hyper sphere of radius r_q and center at q. This hyper sphere is the answer space of the query $q(r_q)$. We describe all notations used in this chapter in Table 2.1.

2.3.1 Preliminaries

We first explain the idea of intersection for *Spatial Intersection Pruning (SIP)* that effectively prunes false candidates. Then we describe how intersection is performed using multiple projections of the data points, each relative to a different reference point, to find candidates. We show that the projection relative to a random viewpoint is locality sensitive: a property that helps SIMP effectively prune false candidate by intersection. Finally, we explain the idea of *Metric Pruning (MP)*.

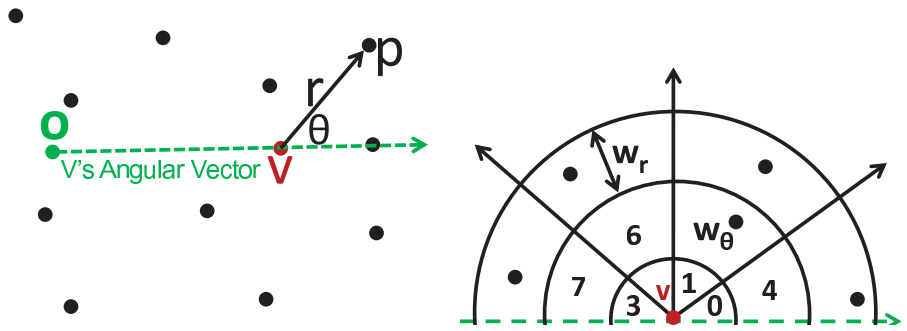


(a) One viewpoint (b) Two viewpoints

Figure 2.1: Spatial Intersection Pruning. (a) Answer space of query $q(r_q)$ is bounded within the distance range $[r_1, r_2]$ relative to viewpoint v_1. Shadowed region contains all the candidates of query $q(r_q)$ relative to viewpoint v_1. (b) Shadowed region contains all the candidates of query $q(r_q)$ relative to two viewpoints v_1 and v_2. Intersection over two viewpoints gives better pruning of the false candidates.

We explain SIP using Figure 2.1. Let v_1 be a randomly chosen reference point, called a **viewpoint**, in the d-dimensional data space $\mathcal{R}^d$. We compute distance

r=$d(v_1, p)$ for each point p in the dataset relative to v_1. Let the answer space of a query $q(r_q)$ be contained in the distance range $[r_1, r_2]$ from v_1 as shown in Figure 2.1. All the points having their distances in the range $[r_1, r_2]$ from v_1 form the set of candidate near neighbors for the query $q(r_q)$. We show the region containing the candidates in shadow in Figure 2.1(a). Let v_2 be another viewpoint. Let $[r'_1, r'_2]$ be the distance bounding range for the answer space of the query $q(r_q)$ relative to v_2. Now, the true near neighbors of the query $q(r_q)$ must lie in the intersection of the range $[r_1, r_2]$ and $[r'_1, r'_2]$ as shown with shadowed region in Figure 2.1(b). We see that an intersection over two viewpoints bounds the answer space more tightly, and thus achieves better pruning of false candidates.



(a) An example dataset (b) Polar grid relative to v

Figure 2.2: (a) A dataset with data space origin **o** and a viewpoint v. Values r and θ for point p are computed relative to v and its angular vector ov. (b) Partition of the data space using equi-width w_r rings and equi-angular w_θ radial vectors relative to viewpoint v and v's angular vector ov. Each bin is given a unique id that places a canonical order on the bins.

We describe how the intersection is performed using an example dataset shown in Figure 2.2(a). Let v be a viewpoint. The vector $\mathbf{o}v$ joining origin $\mathbf{o}$ to the viewpoint v is called v's **angular vector**. We compute distance r=$d(v, p)$ and angle θ for each point p in the dataset relative to v and $\mathbf{o}v$. The range of the distance r is $[0, r_{max}]$, where r_{max} is the distance of the farthest point in the dataset from v. The angle θ lies in the range $[0°, 180°]$ and is computed as

$$\theta = cos^{-1}(\mathbf{o}v.vp/(d(\mathbf{o}, v) \times d(v, p))) \tag{2.1}$$

Thus, we get the projection of the d-dimensional dataset onto a 2-dimensional polar (r, θ) space relative to v. We partition this polar space into grids using equi-width w_r rings and equi-angular w_θ radial vectors relative to the viewpoint v as shown in Figure 2.2(b). This partitioned data space is called v's **polar grid** $\mathcal{G}(v)$. For a given query $q(r_q)$, we compute the projection q' of the query point q relative to the viewpoint v and the angular vector ov. Then, the projection of the answer space of the query $q(r_q)$ relative to v and $\mathbf{o}v$, named **qball**, is a circle with radius r_q and center q'. All the true neighbors of $q(r_q)$ are contained in this qball. We make a choice to use polar coordinates because both of its dimensions, distance r and angle θ, reflect an aggregate value of all the original dimensions in $\mathcal{R}^d$. Any other coordinate system can be similarly used in place of polar coordinates.

Let the qball of the query $q(r_q)$ be enclosed within the bounding range B=$\{[r_1, r_2],[\theta_1, \theta_2]\}$ relative to a viewpoint v as shown in Figure 2.3. We find a set of bins

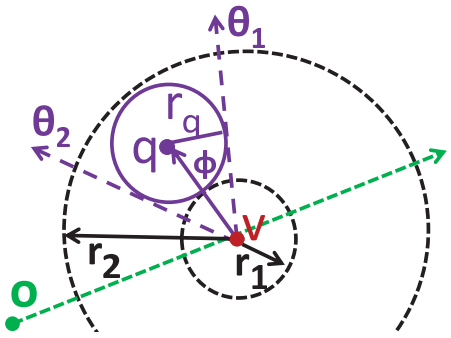


Figure 2.3: Bounding range B=$\{[r_1, r_2],[\theta_1,\theta_2]\}$ of qball of a query $q(r_q)$ relative to a viewpoint v and angular vector $\mathbf{o}v$.

of polar grid $\mathcal{G}(v)$ that encloses the bounding range B. Points contained in these bins form the set of candidates for the query $q(r_q)$ relative to v. For a set of n_v viewpoints, a candidate set is obtained for each viewpoint independently. An intersection of the candidate sets obtained from the n_v viewpoints gives the final set of candidates.

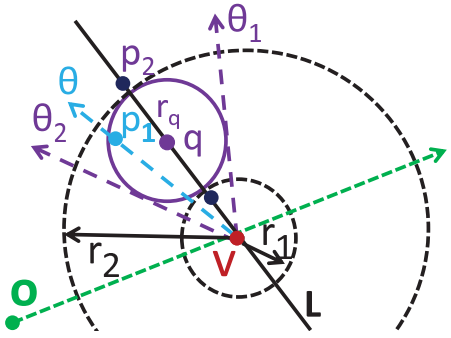


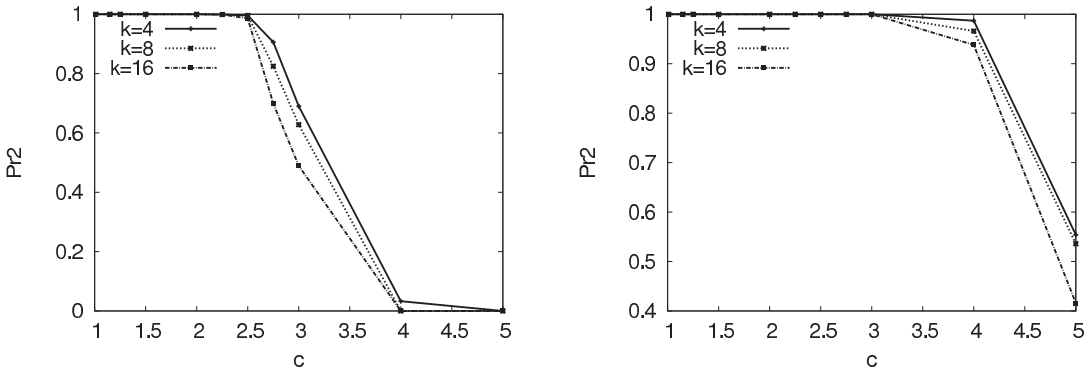
Figure 2.4: We see that point p_2 having $d(q, p_2) > r_q$ lies outside the bounding range of the qball of query $q(r_q)$ relative to viewpoint v lying on the line L. Point p_1 having $d(q, p_2) \leq r_q$ always lies within the bounding range of the qball of query $q(r_q)$ for any viewpoint v.

We next show that the projection relative to a random viewpoint v is locality sensitive. Let v be a randomly chosen viewpoint from the data space and $\mathcal{G}(v)$ be its polar grid. Let the qball of a query $q(r_q)$ be enclosed within the bounding range B=$\{[r_1, r_2], [\theta_1, \theta_2]\}$ relative to v. The bounding range B is computed as shown in Section 2.3.3. Let p be a point at distance r=$d(v, p)$ from v and at an angle θ from v's angular vector ov. Point p is chosen as a candidate if $r_1 \leq r \leq r_2$ and $\theta_1 \leq \theta \leq \theta_2$. If n_v randomly chosen viewpoints are used for intersection, then a point p is a candidate only if it lies in the intersection of the bounding ranges obtained from each of the viewpoints independently. Let Pr_1 be the probability that p is selected as a candidate when $r \leq r_q$. Let Pr_2 be the probability that that p is selected as a candidate when $r > r_q$. Probabilities are computed with respect to the random choices of viewpoints. We say that the projection with respect to a random viewpoint is locality sensitive if $Pr_2 < Pr_1$.

Lemma 1. *The projection of points on a polar (r, θ) space relative to a random viewpoint is locality sensitive.*

Proof. Let points p_1 and p_2 be such that $d(q, p_1) \leq r_q$ and $d(q, p_2) > r_q$. Let p_1 be at an angle θ_{p_1} and p_2 be at an angle θ_{p_2} relative to the angular vector ov of a random viewpoint v.

Point p_1 satisfies $r_1 \leq d(q, p_1) \leq r_2$ and $\theta_1 \leq \theta_{p_1} \leq \theta_2$ for any viewpoint v by construction as shown in Figure 2.4. Therefore, Pr_1=1.



(a) 128-dimensional data (b) 256-dimensional data

Figure 2.5: Values of Pr_2 obtained for varying c, where $d(q, p)=c \times r_q$, and varying number n_v of viewpoints used for spatial intersection on 128 and 256 dimensional real datasets. The value of Pr_1 is always 1.

Next we show that $Pr_2 < 1$ by geometric considerations using Figure 2.4. We draw a line L passing through q and p_2 as shown in Figure 2.4. We draw the enclosing rings of the query $q(r_q)$ at radii $r_1=(d(v, q) - r_q)$ and $r_2=(d(v, q) + r_q)$ from a randomly chosen viewpoint v on the line L. We see that the point p_2 lies outside the enclosing ring, i.e., $d(v, p_2) < r_1$ or $d(v, p_2) > r_2$. This is true for any viewpoint lying on the line L. Therefore, $Pr_2 < 1$ □

We empirically observed that the probability Pr_2 rapidly decreases with an increase in the distance of a point p from the query. This also implies that the probability of a false near neighbor being selected as candidate decreases with its distance from the query point. To derive these results, we computed the values of Pr_2 for a query $q(r_q)$ for varying c, where $d(q, p)=c \times r_q$, using Monte Carlo

methods. We also computed values of Pr_2 for different number of viewpoints n_v used for intersection. We used two real datasets of dimensions 128 (SIFT) and 256 (CHist) for the simulation. We describe these datasets in Section 2.5. We performed simulation using 10 million random viewpoints. We observed that the value of Pr_2 decreases at a fast rate with increasing value of c, as shown in Figure 2.5. For example, the value of Pr_2 is zero for 128-dimensional dataset for c=4 and n_v=4. The value of Pr_1 is always 1.

SIMP achieves a high rate of pruning of false near neighbors due to Lemma 1 and the intersection of the qball of a query with polar grids of multiple viewpoints. This makes SIMP a very efficient method for r-NN search with 100% quality guarantee. In this chapter, we develop a suitable data structure and search algorithm using hashing to implement the idea of SIP.

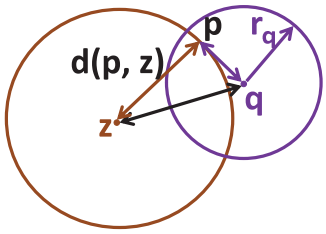


Figure 2.6: Metric Pruning. z is the nearest mcenter of point p. p is a candidate for $q(r_q)$ only if $r_q \geq d(q, p) \geq \mid d(p, z) - d(q, z) \mid$.

To get extra performance, we augment SIP with metric pruning. This pruning is based on triangle inequality and is carried with respect to data clusters. To

obtain a good pruning, these clusters need to be quite small. As a result, we cluster all the data points into n_z tight clusters where a cluster is named an **mball**. We name the center of an mball as an **mcenter**. Let z be the nearest mcenter to a point p. From triangle inequality, we know that a point p is a candidate only if $r_q \geq d(q, p) \geq \mid d(p, z) - d(q, z) \mid$, as shown in Figure 2.6. The nearest mcenter z of each point p and the distance $d(p, z)$ are pre-computed for an efficient pruning using triangle inequality at runtime.

2.3.2 Index Structure

The index structure of SIMP consists of data structures for efficient processing of both SIP and MP. A set of hashtables and bit arrays constitute the index structure of SIP. The index structure of MP consists of three associative arrays.

We first discuss the construction of the index structure for SIP. We randomly choose n_v viewpoints $V=\{v_i\}_{i=1}^{n_v}$ from the d-dimensional dataset. We construct a polar grid for each of the n_v viewpoints. For each polar grid $\mathcal{G}(v)$, a data point p is assigned the id of the bin of the polar grid in which it lies. The bin id of a point p having distance $r=d(v,p)$ and angle θ relative to a viewpoint v and v's angular vector ov is

$$b = (\lfloor r/w_r \rfloor \times (\lfloor 180^\circ/w_\theta \rfloor + 1)) + \lfloor \theta/w_\theta \rfloor.$$

For example, if we take w_r=50 and w_θ=45° for creating a polar grid and a point p at r=70 and θ=40° from the viewpoint, then the bin id of p is $1 \times 5 + 0$=5. The distance r of any point p from a viewpoint v lies between [0, r_{max}], where r_{max} is the distance of the farthest point in the dataset from v. The angle θ of any point p relative to ov lies between [0°, 180°].

Polar grids incur a high space cost if stored separately. A polar grid can be stored in an array whose entry at index i is the collection of the data points in the bin i of the polar grid. A point is stored only by its identifier in the array. This gives a space cost of $n_v \times N$ for n_v polar grids and a dataset of size N. This space cost may become unmanageable for large values of n_v and N. For example, if N is 100 million points and n_v=100, then the total space cost of the index is 40GB.

We develop a hash based index structure to reduce the index space. We assume without loss of generality that n_v=$k \times L$ for some integers k and L. We use a value of k=4 for SIMP. We split n_v viewpoints into L groups, each of size k. A set of k viewpoints is called a *signature* s=$\{v_1, \cdots, v_k\}$. Thus, we construct L signatures s_i such that $(s_i \cap s_j)$=$\emptyset$ for any two signatures s_i and s_j, and $\cup_{i=1}^{L} s_i = V$. We create a hashtable $h(s)$ for each signature s. We generate a k-size $\{b_1 \cdots b_k\}$ hash key for each data point p for a signature s by concatenating the bin ids of p obtained from the polar grids of the viewpoints $v \in s$. All the data points are hashed into the hashtable $h(s)$ using their k-size keys by a standard hashing technique. A

point is stored by its identifier in the hashtable. Each hashtable $h(s)$ defines an intersection over k polar grids. The space cost of L hashtables is $L \times N$, which is k times less than the space cost of storing n_v polar grids ($n_v \times N$).

We describe the creation of hashtables with an example. Let V=$\{v_1, v_2, v_3, v_4\}$ be a set of n_v=4 viewpoints. We create two signatures s_1=$\{v_1, v_2\}$ and s_2=$\{v_3, v_4\}$ for k=2. We create hashtables $h(s_1)$ and $h(s_2)$ for signatures s_1 and s_2 respectively. Let the bin ids of a point p in the polar grids of viewpoints v_1, v_2, v_3, and v_4 be b_1, b_2, b_3, and b_4 respectively. Then, point p is hashed into $h(s_1)$ and $h(s_2)$ using keys b_1b_2 and b_3b_4 respectively.

We also maintain a bit array called *isEmpty* for each polar grid $\mathcal{G}(v)$. The size of this bit array is equal to 1 if r_{max}=0 and is equal to $((\lfloor r_{max}/\ w_r \rfloor + 1) \times (\lfloor 180°/w_\theta \rfloor + 1))$ if $r_{max} > 0$. The actual memory footprint of a bit array is negligible. A bin's id in a polar grid $\mathcal{G}(v)$ is its index in the bit array of $\mathcal{G}(v)$. All the bits corresponding to empty bins of a polar grid $\mathcal{G}(v)$ are marked true.

The index structures of MP are created as follows. We split all the data points into n_z mballs. We assign a unique identifier to each mcenter. We store mcenters in an associative array using their identifiers as keys. We store the identifier and the distance of the nearest mcenter for each data point in associative arrays using the point's identifier as key.

2.3.3 Search Algorithm

In this section, we present the search algorithm for SIMP to retrieve candidates for a query $q(r_q)$ using the data structures of SIP and MP. SIMP performs a sequential scan on these candidates to obtain the near neighbors of the query.

In the SIP step, SIMP first finds the nearest viewpoint v_1 to the given query point q. Then it obtains hashtable $h(s)$ corresponding to v_1, i.e., $v_1 \in s$. SIMP computes the keys of the buckets of $h(s)$ containing the candidates of the query $q(r_q)$ as follows. For a query $q(r_q)$, SIMP finds a set of bins enclosing the qball of the query from the polar grid of each of the k viewpoints $v \in s$. SIMP takes a cartesian product of these k sets to get the keys of the buckets containing candidates. The union of the data points in these buckets gives the set of candidates from the hashtable $h(s)$. For example, let viewpoints $\{v_1, v_2\}$ be the signature s of hashtable $h(s)$ for k=2. Let $\{b_{11}, b_{12}\}$ be the set of bins of polar grid $\mathcal{G}(v_1)$ and $\{b_{21}, b_{22}\}$ be the set of bins of $\mathcal{G}(v_2)$ enclosing the qball of the query $q(r_q)$. The union of the data points contained in the buckets of $h(s)$ having keys $\{b_{11}b_{21}, b_{11}b_{22}, b_{12}b_{21}, b_{12}b_{22}\}$ gives the set of candidates from $h(s)$.

We explain here the method to determine a set of bins of a polar grid $\mathcal{G}(v)$ that encloses the qball of a query $q(r_q)$ relative to a viewpoint v. We first compute the bounding range B=$\{[r_1, r_2], [\theta_1, \theta_2]\}$ of the qball relative to the viewpoint v as shown in Figure 2.3. Let $d(q, v)$ be the distance of the query point q from the

Algorithm 1 getBins

In: $q(r_q)$: query, $d(q,v)$: distance of query from the viewpoint v
In: θ: angle of the query relative to ov
In: A: *isEmpty* bit array of $\mathcal{G}(v)$
1: $BC \leftarrow \phi$ /* ids of enclosing bins */
2: $[r_1, r_2], [\theta_1, \theta_2] \leftarrow$ bounding range of the qball of $q(r_q)$
3: startBinR $\leftarrow \lfloor r_1/w_r \rfloor$, endBinR $\leftarrow \lfloor r_2/w_r \rfloor$
4: startBin$\theta \leftarrow \lfloor \theta_1/w_\theta \rfloor$, endBin$\theta \leftarrow \lfloor \theta_2/w_\theta \rfloor$
5: **for all** $r_i \in$ [startBinR, endBinR] **do**
6: **for all** $\theta_i \in$ [startBinθ, endBinθ] **do**
7: $BC \leftarrow BC \cup (r_i \times (\lfloor 180°/w_\theta \rfloor + 1) + \theta_i)$
8: **end for**
9: **end for**
10: **for all** $b \in BC$ **do**
11: **if** $A[b]$ is True **then**
12: $BC \leftarrow BC \setminus b$
13: **end if**
14: **end for**
15: **return** BC

viewpoint v. The values of r_1 and r_2 are obtained as follows:

$$r_1 = \begin{cases} d(q,v) - r_q & \text{if } r_q < d(q,v) \\ 0 & \text{if } r_q \geq d(q,v) \end{cases} \tag{2.2}$$

$$r_2 = d(q,v) + r_q \tag{2.3}$$

We find the aperture ϕ of the qball of $q(r_q)$ relative to v as follows:

$$\phi = 2 \times sin^{-1}(r_q/d(q,v)). \tag{2.4}$$

Algorithm 2 SIMP

In: $q(r_q)$: query, $\mathcal{H}$: set of hashtables
In: Z: array containing the nearest mcenter of each data point
In: pz: array of distances of the points from their nearest mcenter
1: $C \leftarrow \emptyset$ /*candidate set */
2: $v_1 \leftarrow$ nearest viewpoint to q
3: $h(s) \leftarrow \mathcal{H}(v_1 \in s)$
4: $Y \leftarrow [1]$
5: **for all** $v \in s$ **do**
6: $BC \leftarrow$ getBins($q(r_q)$, $d(q,v)$, θ, $isEmpty$)
7: $Y \leftarrow Y \times BC$
8: **end for**
9: **for all** $key \in Y$ **do**
10: $C \leftarrow C \cup$ points $\in h[key]$
11: **end for**
12: /* Metric Pruning */
13: $qz \leftarrow [\,]$ /* list of distances of mcenters z from q*/
14: **for all** $c \in C$ **do**
15: $z \leftarrow$ c's nearest mcenter from Z
16: **if** $qz[z] \neq$ Null **then**
17: $d' \leftarrow qz[z]$
18: **else**
19: $d' \leftarrow qz[z] \leftarrow d(q,z)$
20: **end if**
21: **if** $|\, pz[c] - d' \,| > r_q$ **then**
22: $C \leftarrow C \setminus c$
23: **end if**
24: **end for**
25: /*Sequential search */
26: **for all** $c \in C$ **do**
27: **if** $d(q,c) > r_q$ **then**
28: $C \leftarrow C \setminus c$
29: **end if**
30: **end for**
31: **return** C

We determine angle θ of the query point q relative to viewpoint v and its angular vector $\mathbf{o}v$ using Equation 2.1. The values of θ_1 and θ_2 are given by

$$\theta_1 = \begin{cases} \theta - \phi/2 & \text{if } \phi/2 \leq \theta \\ 0^\circ & \text{if } \phi/2 > \theta \\ 0^\circ & \text{if } r_q \geq d(q,v) \end{cases} \tag{2.5}$$

$$\theta_2 = \begin{cases} \theta + \phi/2 & \text{if } (\phi/2 + \theta) < 180° \\ 180° & \text{if } (\phi/2 + \theta) \geq 180° \\ 180° & \text{if } r_q \geq d(q, v) \end{cases} \tag{2.6}$$

We compute the bins from a polar grid $\mathcal{G}(v)$ using the bounding range B={[r_1, r_2],[θ_1, θ_2]} as described in Algorithm 1. We first obtain the set of bins enclosing the qball of the query for radial partitioning and angular partitioning independently in steps 3 and 4 respectively. We iterate through these bins to compute a set of bins of $\mathcal{G}(v)$ that encloses the qball in steps [5-9]. We remove empty bins from this set using *isEmpty* bit array of $\mathcal{G}(v)$ in steps [10-14].

In the MP step, SIMP uses triangle inequality between a candidate obtained from SIP step, candidate's nearest mcenter, and the query point q. It retrieves the nearest mcenter z of a candidate and the distance to the mcenter $d(p, z)$ from the index. SIMP computes the distance $d(q, z)$ between the mcenter z and the query point q. SIMP discards a candidate if $r_q <\mid d(p, z) - d(q, z) \mid$.

We describe the execution of SIMP using Algorithm 2. Spatial intersection pruning is performed in steps [1-11]. SIMP finds hashtable $h(s)$ whose signature s contains the nearest viewpoint v_1 to the query point q in steps [2-3]. For each viewpoint v in signature s, SIMP obtains a set of bins BC enclosing the qball of the query $q(r_q)$ in step 6. SIMP computes a set Y of hash keys by a cartesian

product of the set of bins BC in step 7. SIMP takes a union of the points in the buckets of hashtable $h(s)$ corresponding to the hash keys Y in steps [9-11]. Next, SIMP applies metric pruning in steps [13-24]. For each candidate, SIMP gets the identifier of its nearest mcenter z from a pre-computed array Z in step 15. SIMP computes the distance of the query q from z if it is not previously computed; otherwise it retrieves the distance from an array qz in steps [16-20]. A candidate is tested using the triangle inequality in step 21. Finally, all the true neighbors are obtained by computing the actual distance of each of the candidates from the query point in steps [26-29].

Extension to nearest neighbor search: The SIMP algorithm for r-NN search can be extended for top-k nearest neighbor search using the approach proposed by Andoni et al. [49]. For a dataset, an expected distance $E(r)$ of top-k nearest neighbors from query points is estimated under the assumption that the query distribution follows the data distribution. We start the r-NN search with r=$E(r)$. If no nearest neighbor is obtained, then we repeat SIMP with range $((1+c) \times r)$ until at least k points are retrieved.

2.4 Statistical Cost Modeling and Analysis

We develop statistical models to measure the query costs of SIMP. For a query $q(r_q)$, the number of buckets of a hashtable probed in steps [5-11] and the number of candidates C to which distance is computed in steps [26-30] of Algorithm 2 define the cost of SIMP. We develop models to find the expected number of buckets of a hashtable $h(s)$ probed and the expected number of candidates obtained for a query $q(r_q)$.

Data distribution: For a viewpoint v and a point p, let $r{=}d(p,v)$ be the distance of p from v and θ be the angle of p relative to v's angular vector ov. Let $\mathcal{P}_v(r, \theta)$ be the spatial probability mass function of the data space relative to the viewpoint v. A hashtable $h(s)$ defines an intersection over k viewpoints. Let $[r, \theta]$ represent the list $[r_i, \theta_i]_{i=1}^{k}$. We represent a point p relative to k viewpoints of a hashtable $h(s)$ as $p([r, \theta])$, where $[r, \theta]$ is the list of distances and angles of p relative to all the viewpoints $v \in s$.

$$p([r, \theta]) = [\cup_{i=1}^{k}(r_{v_i}, \theta_{v_i}) \text{ for all } v_i \in s] \tag{2.7}$$

Let $\mathcal{P}([r, \theta])$ be the joint spatial probability mass function of the data space over k viewpoints. Let Q be the query space. The joint spatial probability mass function $\mathcal{Q}([r, \theta])$ of the query space is taken to be similar to the data space mass function

$\mathcal{P}([r, \theta])$. All the expectations are computed with respect to query mass function $\mathcal{Q}([r, \theta])$. A query $q(r_q)$ is represented as $q(r_q, [r, \theta])$ relative to k viewpoints.

Expected number of hash buckets probed: We compute the set of bins BC of a polar grid $\mathcal{G}(v)$ enclosing the qball of the query $q(r_q)$ using Algorithm 1. All the bits of *isEmpty* bit array are taken to be false. The total number of buckets $Y([r,\theta])$ of a hashtable $h(s)$, whose signature s has k viewpoints, probed for a query $q(r_q, [r, \theta])$ is given by

$$Y([r, \theta]) = \prod_{i=1}^{k} \mid BC_{v_i} \mid$$

The expected number of buckets probed in a hashtable $h(s)$ is obtained by taking a sum over the query space Q with respect to the query mass function $\mathcal{Q}([r, \theta])$.

$$E(Y) = \sum_{x=1}^{|Q|} Y([r, \theta]_x) \times \mathcal{Q}([r, \theta]_x) \tag{2.8}$$

Expected number of candidates: To obtain the expected number of candidates $E(C)$, we first derive the probability of a random point p being chosen as a candidate by spatial intersection pruning. The bounding range B=$\{[r_1, r_2],[\theta_1], \theta_2]\}$ of the qball of a query $q(r_q)$ relative to a viewpoint v is obtained as discussed in Section 2.3.3. For a viewpoint v, the probability that p is selected as a candidate is

$$\begin{aligned} Pr_v(p \text{ is candidate}) &= Pr(p \in B) \\ &= \sum_{\theta_1}^{\theta_2} \sum_{r_1}^{r_2} \mathcal{P}_v(r, \theta). \end{aligned}$$

For k viewpoints, a random point p is a candidate only if p lies in the intersection of the bounding ranges of all the k viewpoints. Let $\{B_1, \cdots, B_k\}$ be the bounding ranges with respect to k viewpoints. Then, the probability that p is a candidate is

$$Pr_v(p \text{ is candidate}) = Pr(p \in B \cap \cdots \cap B_k).$$

The intersection of the bounding ranges of k viewpoints is not independent. Therefore, to obtain the probability $Pr_v(p \text{ is candidate})$, we compute the bounding volume of the qball of query $q(r_q)$ for each viewpoint v independently. The bounding volume for a viewpoint v is obtained by a cartesian product of the bounding distance and angular ranges of the qball relative to v. The joint bounding volume of k viewpoints is obtained by

$$\text{Vol} = \prod_{i=1}^{k} [r_1^i, r_2^i] \times [\theta_1^i, \theta_2^i].$$

The probability that a random point p is a candidate, if k polar grids are used, is obtained by taking a sum of the joint spatial probability mass function $\mathcal{P}([r, \theta])$ of the data space over the joint volume

$$Pr^{SIP}(p \text{ is candidate}) = \sum_{Vol} \mathcal{P}([r, \theta]). \tag{2.9}$$

Next, we derive the probability that a random point p is chosen as a candidate by metric pruning. The distance probability distribution of points of a data space

relative to a point p_1 is given by

$$\mathcal{F}_{p_1}(r) = Pr(d(p_1, p) \leq r).$$

The probability that a random point p, having the nearest mcenter z, is a candidate for a query $q(r_q)$ is

$$\begin{aligned} Pr^{MP}(p \text{ is candidate }) &= Pr(\mid d(p,z) - d(q,z) \mid \leq r_q) \\ &= \mathcal{F}_q(r_q) \end{aligned} \tag{2.10}$$

where $\mathcal{F}_q$ is the distance distribution of $\mid d(p,z) - d(q,z) \mid$ relative to q. It is not feasible to compute $\mathcal{F}_q$ at runtime or store it for all possible queries. Therefore, we approximate it with $\mathcal{F}_{z_q}$, which is the distance distribution of $\mid d(p,z) - d(z_q,z) \mid$ relative to the nearest mcenter z_q of the query point q.

The probability of a random point p being chosen as a candidate for query $q(r_q, [r, \theta])$ using both SIP and MP is:

$$\begin{aligned} Pr(p_{candidate}) = Pr^{MP}(p \text{ is candidate}) \\ \times Pr^{SIP}(p \text{ is candidate}) \end{aligned} \tag{2.11}$$

The total number of candidates C for a query $q(r_q, [r, \theta])$ is given by C=$Pr(p$ is cand$)\times$ N, where N is the dataset size. The expected number of candidates $E(C)$ is obtained by taking a sum over the query space Q with respect to query mass function

$\mathcal{Q}([r, \theta])$.

$$E(C) = \sum_{x=1}^{|Q|} Pr(p \text{ is candidate }) \times \mathcal{Q}([r, \theta]_x) \times N \tag{2.12}$$

It is worth noting that Equation 2.12 gives the expected number of candidates $E(C)$ for Algorithm SIMP. If the distance probability distribution ${\mathcal{F}'}_q$ of the data points relative to the query point q is known, then the actual number of candidates is ${\mathcal{F}'}_q(r_q) \times N$.

We empirically verified the robustness of our model for $E(C)$ using error ratio ξ. If $E(C_a)$ is the average number of candidates obtained empirically, then

$$\xi = \mid E(C_a) - E(C) \mid / E(C_a) \tag{2.13}$$

We computed $\xi(\%)$ for multiple query ranges r_q and various number of mballs n_z on two real datasets of dimensions 128(SIFT) and 256 (CHist) which are described in Section 2.5. The values of $\xi(\%)$ are shown in Table 2.2. We see that $\xi(\%)$ on both the datasets is less than 1% for all query ranges r_q and the number of balls n_z.

Space complexity: SIMP has a linear space complexity in the dataset size N. We compute the memory footprint of SIMP for a d-dimensional dataset having N points. Let the space usage of a word be W bytes. Let each dimension of a point take one word. Then, the space cost of the dataset is $(N \times d \times W)$ bytes. Let a point identifier take one word. A point is stored by its identifier in a hashtable.

d=128			d=256		
r_q	n_z=5,000	n_z=15,000	r_q	n_z=5,000	n_z=15,000
50	0.947	0.947	300	0.703	0.693
100	0.953	0.952	400	0.698	0.687
150	0.954	0.954	500	0.739	0.733
200	0.954	0.954	600	0.743	0.734

Table 2.2: Error ratio $\xi(\%)$ for expected number of candidates $E(C)$ for varying query ranges r_q and varying number of mballs n_z.

Therefore, the space required for L hashtables is $(L \times N \times W)$ bytes. The space cost of n_z mcenters is $(n_z \times d \times W)$ bytes. $(N \times log_2(d_{max}) + N \times log_2(n_z))$ bytes are required to store the distance and the identifier of the nearest mcenter for each point. d_{max} is the maximum distance of a point from its nearest mcenter. We fairly assume that $log_2(d_{max}) \leq W$ and $log_2(n_z) < W/2$. Therefore, the total memory footprint is $N \times W \times (d + L + (n_z/N)d + 1 + 1/2)$ bytes $= a \times N \times W \propto N$. Here, a is a constant proportional to d and W is a constant. Thus, we see that the space complexity of SIMP is $O(N)$.

2.5 Empirical Evaluations

We empirically evaluated the performance of SIMP on five real datasets. We compared SIMP with four alternative methods: (1) p-stable LSH [31], (2) Multi-Probe LSH [87], (3) LSB tree [122], and (4) iDistance [60]. All these methods are briefly described in Section 2.2. p-stable LSH and iDistance are state-of-the-art

methods for an approximate and an exact search respectively, while Multi-Probe LSH and LSB tree have been recently proposed. We first introduce the datasets and the metrics used for measuring the performance of the algorithms. Then we describe the query workload for our experiments and the construction of an specific instance of SIMP index. Next we describe the performance comparison of SIMP with the alternative methods. Finally, we show scalability results of SIMP on datasets having as many as 100 million points.

Dataset description: We used 5 real datasets of various dimensions and sizes for our experiments. The first real dataset, called *SIFT*, contains 128-dimensional 1 million SIFT [86] feature vectors extracted from real images [61]. SIFT is a state-of-the-art feature used for content based image retrieval and object recognition. The second dataset, called *SIFT10M*, and the third dataset, called *SIFT100M*, has 10 million and 100 million 128-dimensional SIFT feature vectors of real images respectively. We obtained these three datasets from INRIA Holiday dataset[1]. All pairs distance distribution of 100,000 SIFT feature vectors is shown in Figure 2.7. The distance distribution of all the feature vectors in SIFT10M dataset relative to a randomly chosen feature vector is shown in Figure 2.8. The distance in Figure 2.7 and Figure 2.8 is scaled to have a maximum value of 1.

[1] http://lear.inrialpes.fr/jegou/data.php

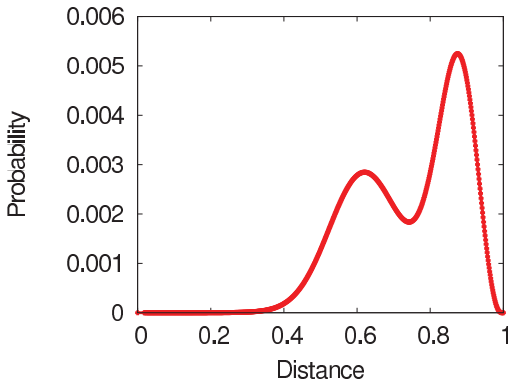


Figure 2.7: All pairs distance distribution of 100,000 128-dimensional SIFT feature vectors.

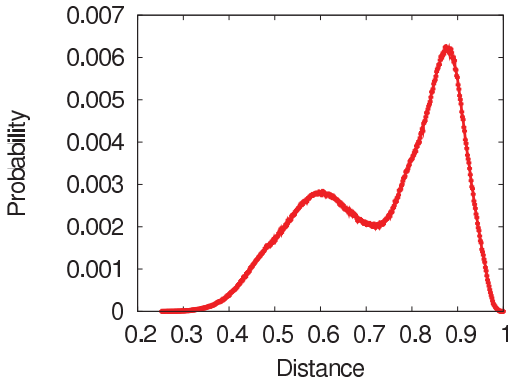


Figure 2.8: The distance distribution of all the feature vectors in SIFT10M dataset relative to a randomly chosen feature vector.

The fourth real dataset, called *CHist*, has 256-dimensional 1,082,476 color histograms of images. For this, we downloaded random images from Flickr[2]. We transformed each image into gray-scale. Then we extracted a 256-dimensional

[2]http://www.flickr.com/

histogram from each image by counting the number of occurrences of each color in the image. The fifth dataset, called *Aerial* [116], has 10 million points of 32 dimensions. We obtained 82,282 gray-scale aerial images from the Alexandria Digital Library[3]. These aerial images are satellite images and air photos of different regions of California. The size of these images varies from 320×160 pixels to 640×480 pixels. We split each of the aerial images into non-overlapping tiles of size 32×32 pixels. The total number of tiles obtained are 10,625,200. We computed a 32-dimensional histogram of the pixel values of each tile in a manner similar to Color Structure Descriptor [90].

Performance metrics: We measured the performance of the algorithms using following metrics: (1) *recall*, (2) *selectivity*, (3) *query time*, and (4) *space usage*. These metrics validate the quality of results, the efficiency, and the scalability of the algorithms. *Recall* measures the result quality of an algorithm. It is the ratio of the number of the true neighbors retrieved by an algorithm to the total number of the true neighbors in the dataset for a query. The true near neighbors of a query in a dataset are obtained using sequential search. iDistance and SIMP have 100% recall. The efficiency of the algorithms are measured by their selectivity, query time, and space usage. *Selectivity* of an indexing scheme is the percentage of the data points in a dataset for which the actual distance from the query is computed.

[3]http://www.alexandria.ucsb.edu/

Query time is the elapsed CPU time between the start and the completion of a query. We verify the space efficiency of the algorithms by computing the memory footprints of their index structures. The main factors governing the query cost of SIMP are the dataset size N, the dataset dimension d, and the query range r_q. We verify the scalability of SIMP by computing its query time for varying values of N, d, and r_q. We observed a sequential search time of 311ms for SIFT, 603ms for CHist, 774ms for Aerial, 3,209ms for SIFT10M, and 28,000ms for SIFT100M.

Query workload: We randomly picked 1,000 query points from each of the SIFT, CHist, and Aerial datasets. We used the queries of SIFT also for both SIFT10M and SIFT100M. Each result on a dataset is reported as an average over all its query points. We performed experiments for multiple query ranges r_q for each query point q. We used query ranges r_q= {50, 100, 150, 200} for SIFT, Aerial, SIFT10M, and SIFT100M and query ranges r_q={300, 400, 500, 600} for CHist. For a dataset, query ranges are chosen such that at least 90% of its data points have their top-1 nearest neighbors within the largest query range. We computed the cumulative mass function of distances of top-1 nearest neighbors of a large set of random query points from each dataset. We found that more than 90% of the queries of SIFT, Aerial, SIFT10M, and SIFT100M have their top-1 nearest neighbor within a query range of 200. The same was true for the query range of 600 for CHist.

SIMP index: Here we describe a specific construct of SIMP index. The viewpoints for SIMP are picked randomly from the dataset. SIMP uses a fixed signature size of k=4 for its hashtables. The number of hashtables L is decided based on the memory constraints. We used two values of L=$\{1, 25\}$ for our experiments. SIMP requires values of w_r and w_θ to create polar grids. A fixed value of w_θ=45° is used for creating the polar grids. The value of w_r is learned for each dataset by training. For our experiments, we chose a training query range r_0 and also randomly picked a set of query points from each dataset. Then we measured the performance of SIMP for a set of values of w_r using r_0 and the query points. We chose the value of w_r which produced the best result for the dataset. We used k-means clustering to find mballs and mcenters for metric pruning (MP). For our experiments, we used n_z=$5,000$ mballs for metric pruning.

All the experiments were performed on Debian GNU/Linux 5.0 and quad-core Intel(R) Xeon(R) CPU $5,140$@2.33GHz with 4MB cache. All the programs were implemented in Java. We used Java Hotspot 64-bit (16.3 build) Server VM.

2.5.1 Performance comparison with p-stable LSH and iDistance

We present the performance comparison of SIMP with p-stable LSH [31] and iDistance [60] on SIFT, CHist, and Aerial datasets. We first describe the settings

of the algorithms used for comparative studies. Then we give empirical evidences to show that SIMP is much superior than LSH on the result quality. Next we empirically show that SIMP is much more efficient than iDistance for all datasets and query ranges. We also show that SIMP scales linearly with the dataset dimension d and the query range r_q. The scalability of SIMP with the dataset size is shown in Section 2.5.3. Finally, we compare the space efficiency of these algorithms.

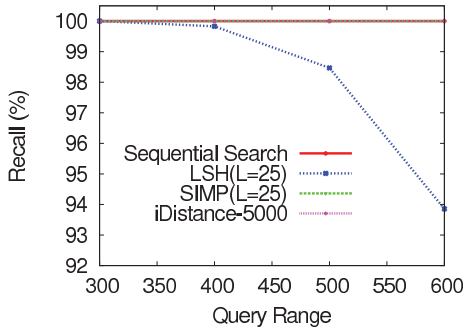
We used following settings of the algorithms for comparison. We implemented p-stable LSH similar to Datar et al. [31] for Euclidean norm. The parameters of LSH are the number of hashtables L, the number of hash values k' concatenated to generate a hash key, and the bin-width w used to bucket the projected values. We used the same number of hashtables L=25 for both LSH and SIMP. We learned the values of k' and w for LSH. We chose the query range r_0=50 for SIFT and Aerial and the query range r_0=300 for CHist for learning the parameters k' and w of LSH and the parameter w_r for SIMP. We measured the performance of LSH for a set of values of k' and w on each dataset using the training query range r_0 and a set of randomly picked query points. We chose the values of k' and w for which LSH had 100% recall with the least query time. For LSH, we learned the values w=1,700 and k'=8 on CHist, w=350 and k'=8 on SIFT, and w=250 and k'=8 on Aerial. For SIMP, we learned the values w_r=300 on CHist, w_r=30 on

SIFT, and w_r=100 on Aerial. We used n_z=5,000 mballs for SIMP. We used the mcenters of these mballs as reference points for iDistance.

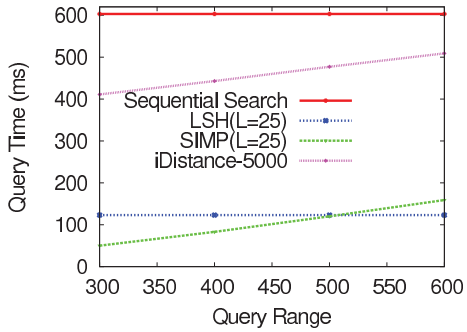
We observed that SIMP always guarantees 100% quality compared to LSH whose quality falls below 50% for larger query ranges. We show the performance of the algorithms on CHist, SIFT, and Aerial datasets in Figures 2.9, 2.10, and 2.11 respectively. We see that SIMP and iDistance have 100% recall on datasets of all sizes and dimensions and for all query ranges. Unlike SIMP, the recall of LSH falls rapidly with an increase in the query range. LSH had a recall of only 21.6% for r_q=200 on Aerial dataset.

Our empirical results show that SIMP has a superior performance than iDistance on datasets of all sizes and dimensions and for all query ranges. Both the methods always yield 100% result quality but SIMP significantly outperforms iDistance in efficiency. We see from Figures 2.9, 2.10, and 2.11 that iDistance has larger query time and selectivity than SIMP on all the datasets. This difference in performance widens with an increase in the dimension of the datasets and the query range. SIMP had a selectivity of 5% compared to 42% selectivity of iDistance on CHist dataset for the query range r_q=300 as seen in Figure 2.9.

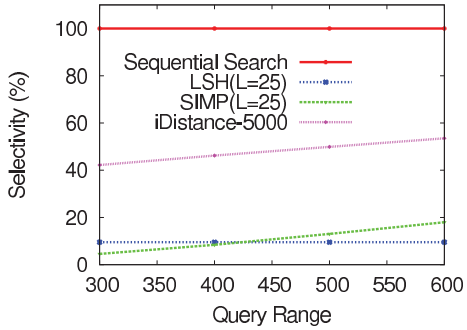
We observed that the selectivity and the query time of SIMP grows linearly with the dataset dimension d and the query range r_q. We see from Figures 2.11 and 2.10 that SIMP has a selectivity of 0.2% and 0.7% on 32-dimensional Aerial



(a) Recall

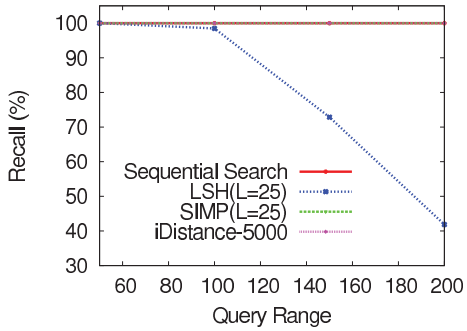


(b) Query Time

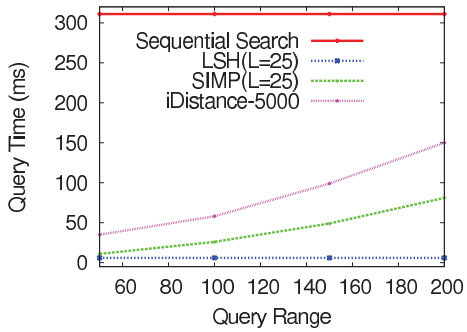


(c) Selectivity

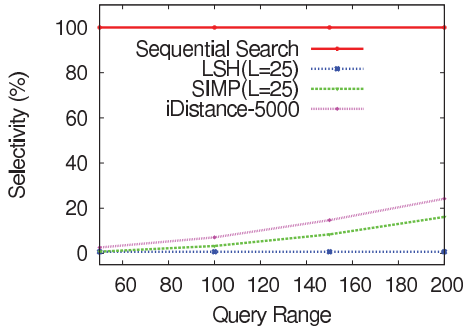
Figure 2.9: Comparative study of performance of SIMP with p-Stable LSH and iDistance on 256-dimensional CHist dataset.



(a) Recall

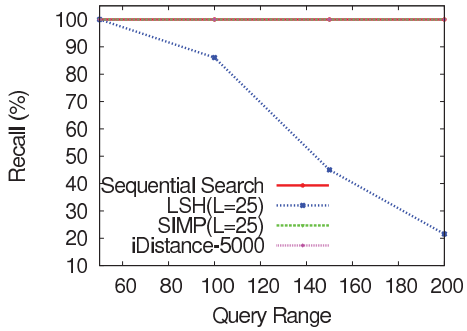


(b) Query Time

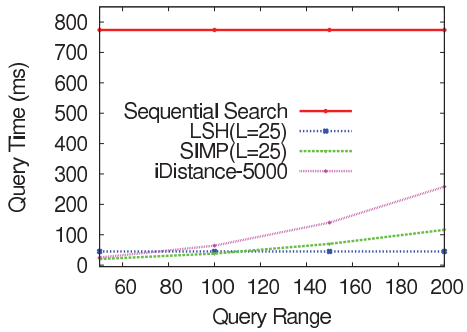


(c) Selectivity

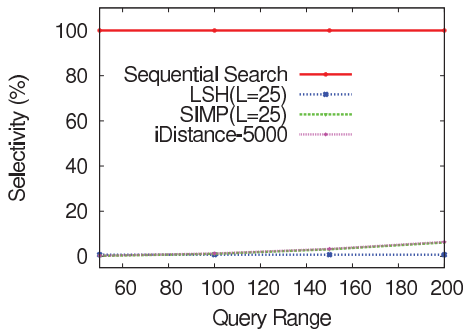
Figure 2.10: Comparative study of performance of SIMP with p-Stable LSH and iDistance on 128-dimensional SIFT dataset.



(a) Recall



(b) Query Time



(c) Selectivity

Figure 2.11: Comparative study of performance of SIMP with p-Stable LSH and iDistance on 32-dimensional Aerial dataset.

and 128-dimensional SIFT datasets respectively for the query range 50. This shows that the selectivity of SIMP grows linearly with d. From Figure 2.9, we see that the selectivity of SIMP grows from 5% for r_q=300 to 17% for r_q=600 on CHist dataset. This validates that the selectivity of SIMP grows linearly with r_q. The small values of the selectivity of SIMP on all the datasets verify that SIMP effectively prunes false candidates using its index structure. The linear behavior of SIMP with d and r_q is further confirmed by its query time on all the three datasets. We see from Figure 2.9 that the query time of SIMP grows linearly from 50ms for r_q=300 to 150ms for r_q=600 on CHist. The query time of SIMP also increases only linearly with d.

It is evident from the empirical results that SIMP is a superior alternative for an accurate and efficient r-NN search. p-stable LSH and SIMP have similar performance for the training query range r_0. With an increase in the query range $r_q > r_0$, the search quality of p-stable LSH falls sharply, whereas SIMP gives 100% result quality with only a linear increase in the search cost. Further, p-stable LSH is inefficient for queries with $r_q < r_0$. Thus, we see that LSH index structure created for a fixed query range r_0 can not handle queries with varying query ranges accurately and efficiently. SIMP performs much better than iDistance across all the datasets of various dimensions. The performance difference between SIMP and iDistance grows with an increase in the dimension of the dataset. The

poor performance of iDistance is because of its multiple searches on the B+ Tree and high selectivity. iDistance searches B+ tree for each mball intersected by an r-NN query.

Space cost comparison: We now discuss the space costs of SIMP, p-stable LSH, and iDistance. We compute memory footprints of the algorithms for CHist dataset using parameters W=4, N=1,082,476, L=25, d_{max}=22,500, and n_z=5,000. d_{max} is the maximum distance of any point from its nearest mcenter. The space cost of the dataset for each algorithm is $(N \times d \times W)$=1,109MB. The memory footprint of LSH index is $(N \times W \times L)$=108MB and SIMP index is $(N \times W \times L) + (n_z \times d \times W + N \times log_2(d_{max}) + N \times log_2(n_z))$=117MB. The extra space usage of SIMP over p-stable LSH is from the data structures of the metric pruning. This overhead remains constant for any value of L. The index structure of iDistance needs a total of 14MB space. Each entry of iDistance needs 32 bits for storing its distance from nearest mcenter as key and 32 bits to store a pointer to a child node or a data object. Each leaf node also needs 8 bytes for storing pointers of its neighboring nodes. iDistance needs 5.12MB for storing 5,000 mcenters. We take 512 entries per node for B+ tree of iDistance. For L=1, the memory usage of SIMP and iDistance is the same.

2.5.2 Performance comparison with Multi-Probe LSH and LSB Tree

Here we describe the performance comparison of SIMP with Multi-Probe LSH [87] and LSB tree [122] on 128-dimensional SIFT and 256-dimensional CHist datasets. We first describe the settings of the algorithms used for the comparison. Then we present results to show that SIMP significantly outperforms both Multi-Probe LSH and LSB tree on all the datasets for all the query ranges. Finally, we compare the space efficiency of the algorithms.

Dataset	t	f	p_2	m	L	H_{max}	u	B	w
SIFT	217	15	0.61	24	354	478019	18	4096	4
CHist	22500	23	0.61	26	521	1.3327E8	27	4096	4

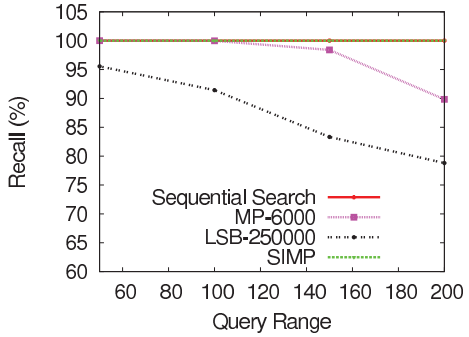
Table 2.3: Parameters of LSB Tree and LSB Forest for two real datasets.

We used a similar index structure for Multi-Probe LSH as p-stable LSH. We implemented the Query-Directed probing sequence algorithm for Multi-Probe LSH as it was shown to perform better than the step-wise probing sequence [87]. Multi-Probe search algorithm does not have a terminating condition, whereas LSB search algorithm has a terminating condition for top-k search. Therefore, we made the following choices for the termination of Multi-Probe and LSB Tree in order to have a meaningful comparison, based on both quality and efficiency, with SIMP for r-NN queries. We terminated Multi-Probe after a fixed number of probes P. LSB

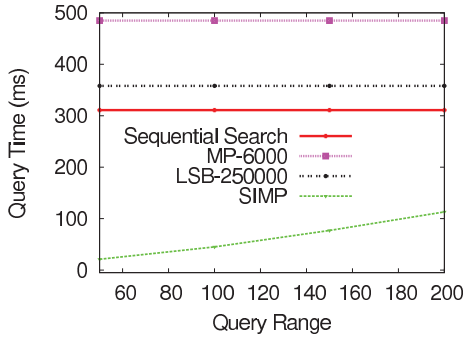
search was terminated for a fixed selectivity S that is the percentage of the dataset explored by LSB search as candidates. We did not compare against LSB forest because of its high space overhead, which can be observed from its parameters in Table 2.3 and has also been noted by the authors. We used L=1 hashtable for both Multi-Probe and SIMP. We used n_z=5,000 mballs for SIMP. We learned the values w_r=300 and w_r=50 on CHist and SIFT datasets respectively for SIMP. We learned the values w=1,700 and k'=8 on CHist and w=350 and k'=8 on SIFT for Multi-Probe.

A performance comparison of the algorithms on SIFT and CHist datasets are shown in Figures 2.12 and 2.13 respectively. We measured the performance of Multi-Probe for P=6,000 probes and LSB tree for the selectivity S=25% on SIFT dataset. We used P=1,500 probes for Multi-Probe LSH and a selectivity S=40% for LSB tree on CHist dataset. Figures 2.12 and 2.13 show that the recall of both Multi-Probe LSH and LSB tree decreases with an increase in the query range, while SIMP has 100% recall for all query ranges. Multi-Probe had a recall of 90% and LSB had a recall of 79% for the query range 200 on SIFT dataset as seen from Figure 2.12. These results verify that SIMP always yields superior result quality than Multi-Probe and LSB.

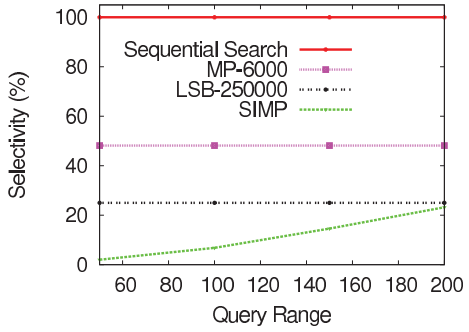
We empirically found that SIMP is much more efficient than Multi-Probe and LSB tree. We see from Figures 2.12 and 2.13 that both Multi-Probe and LSB



(a) Recall

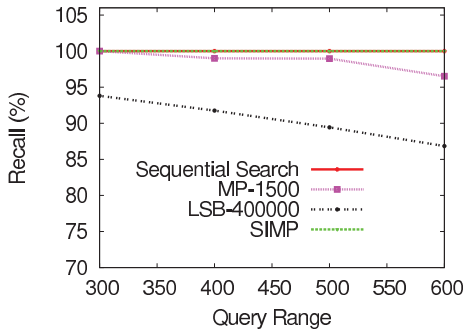


(b) Query Time

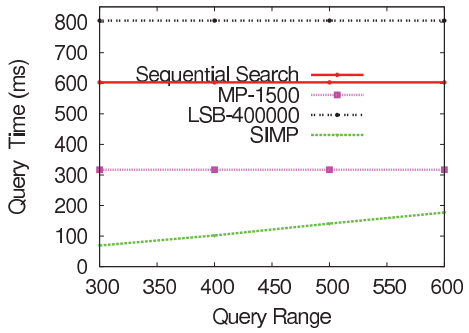


(c) Selectivity

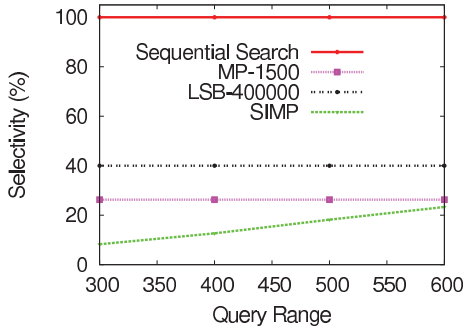
Figure 2.12: Comparative study of performance of SIMP with Multi-Probe LSH (MP) and LSB Tree on 128-dimensional SIFT dataset.



(a) Recall



(b) Query Time



(c) Selectivity

Figure 2.13: Comparative study of performance of SIMP with Multi-Probe LSH (MP) and LSB Tree on 256-dimensional CHist dataset.

have larger selectivity and query time than SIMP for all query ranges r_q on both the datasets. For r_q=50 on SIFT dataset, Multi-Probe was 24 times slower and had 24 times more selectivity than SIMP. For r_q=50 on SIFT dataset, LSB was 17 times slower and had 12 times more selectivity than SIMP. SIMP was also significantly better than LSB Tree and Multi-Probe on CHist dataset.

Our empirical evidences show that SIMP, that guarantees 100% quality, is a superior alternative for an accurate and efficient r-NN search over Multi-Probe LSH and LSB tree. Multi-Probe is a heuristic with no performance and quality guarantees. A large number of probes improves the result quality of Multi-Probe but worsens its efficiency. Multi-Probe yields a high query time for a large number of probes because of the high cost of the computation of the probing sequence and a high selectivity. The poor performance of LSB can be mainly attributed to two reasons. First, m-dimensional points obtained after projections are indexed into a B-Tree based on their z-order values. The value of m increases with an increase in the database size and dimension for constant values of the other parameters. It is well known from the literature that the performance of tree-based indices deteriorate for large dimensions, and so does B-Tree based LSB tree. The value of m is 24 for SIFT and 26 for CHist as seen in Table 2.3, which are sufficiently large to make the LSB tree inefficient. Second, its query time increases with an increase in candidate size because of a large number of bit operations required for

computing LLCP between two z-orders. The size of a z-order value is $m \times u$=24 $\times$ 18=432 bits for SIFT dataset and 26 $\times$ 27=702 bits for CHist dataset.

Space cost comparison: Here, we discuss the space costs of each of the algorithms using parameters d=256, N=1,082,476, W=4, L=1, and n_z=5,000. The space cost of the dataset for each algorithm is 1,109MB. The memory footprint of Multi-Probe index is 4.5MB and SIMP is 13MB. The hashtables of Multi-Probe LSH and SIMP store only the identifier of a point, which takes one word (W bytes). The extra space usage of SIMP over Multi-Probe is again from the data structures of metric pruning. We compute the memory required by LSB tree using the parameters of CHist shown in Table 2.3. We use 512 entries per node for LSB. Each entry of LSB tree stores a z-order value (to compute LLCP) as key and a pointer to a child node or a data object. A z-order value needs $m \times u$=702 bits for storage and a pointer needs 32 bits of storage. LSB tree also needs to store forward and backward pointers of the immediate neighbors of each leaf node. Thus, the total space required for LSB is 100MB. For L=1, we find that the memory footprint of LSB is at least 7 times worse than SIMP and 22 times worse than Multi-Probe. For L=2, LSB takes 200MB whereas the space cost of SIMP and Multi-Probe increase only by 4.5MB (for storing an extra hashtable) to 17.5MB and 9MB respectively.

2.5.3 Large Scale Performance Evaluation

We validated the scalability of SIMP on 128-dimensional real datasets of sizes up to 100 million. Our stress tests reveal that SIMP scales linearly with the dataset size and the query range at very high dimensions. We already showed in Section 2.5.1 that SIMP scales linearly with the dataset dimension. We also further compared SIMP with p-stable LSH for a large workload of $14,700$ queries on 128-dimensional SIFT10M dataset having 10 million points. This test again confirmed that SIMP efficiently and accurately queries near neighbors for any query range, while p-stable LSH has a very poor recall for query ranges larger than the training query range r_0. We used a value of w_r=30 and n_z=$5,000$ for SIMP for these studies.

We computed the selectivity and the query time of SIMP on SIFT, SIFT10M, and SIFT100M datasets to verify its scalability. We computed these values for varying number of hashtables L=$\{1, 25\}$ and varying query ranges r_q. We show the selectivity and the query time of SIMP in Figures 2.14 and 2.15 respectively. Each result is an average over $1,000$ random queries. These results reveal that SIMP has similar selectivity for the dataset of any size for a given r_q and L. This property implies that SIMP scales linearly with the dataset size. This is further confirmed by the query time of SIMP. We see from Figure 2.15 that the query time of SIMP increases approximately 10 times with 10 times increase in the dataset

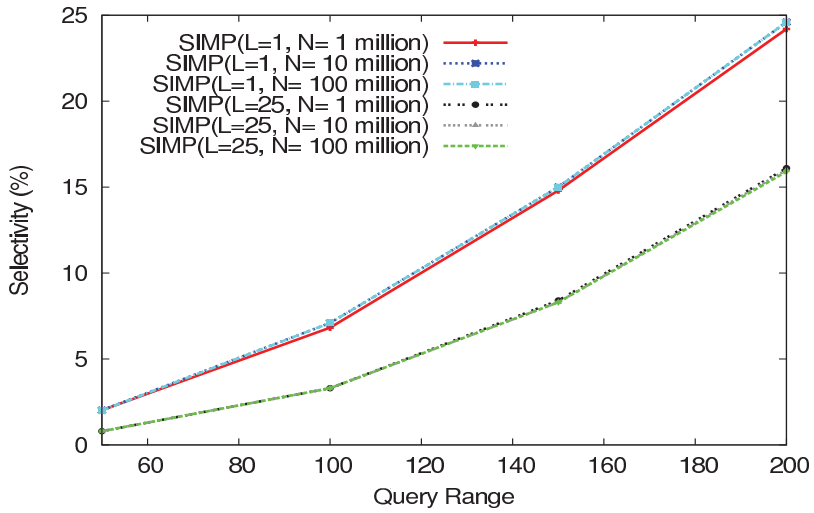


Figure 2.14: Selectivity of SIMP on 128-dimensional real datasets of varying sizes for varying number of hashtables L.

for a given r_q and L. We also observed that the query time of SIMP has a linear behavior with the query range.

SIMP had a query time of 0.4 seconds on 100 million points for r_q=50 and L=25 compared to 28 seconds of sequential search. For SIFT100M dataset, we observed by random sampling that every point has at least one near neighbor within the query range 50. This shows that SIMP can be used to efficiently and accurately find the nearest neighbors in very large datasets of very high dimensions.

The comparative results of SIMP with p-stable LSH on SIFT10M dataset for 14,700 random queries is shown in Figure 2.16. We used a value of L=25 for both

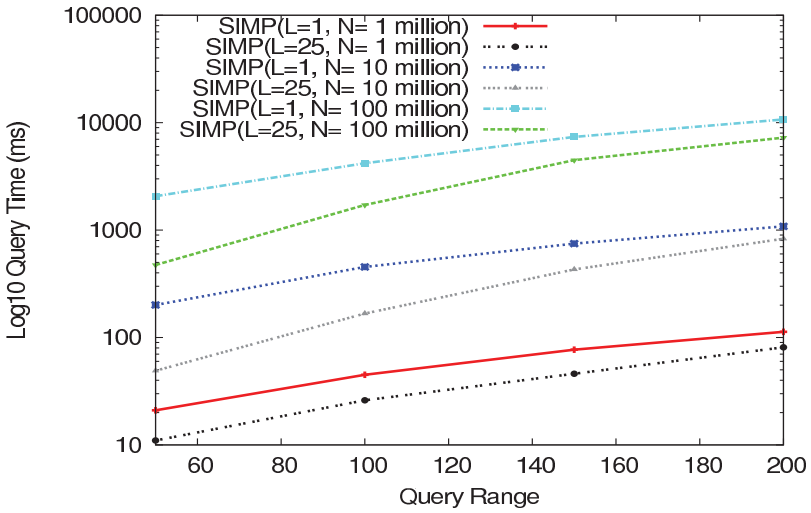
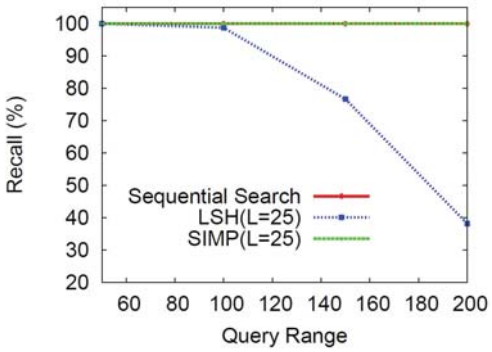
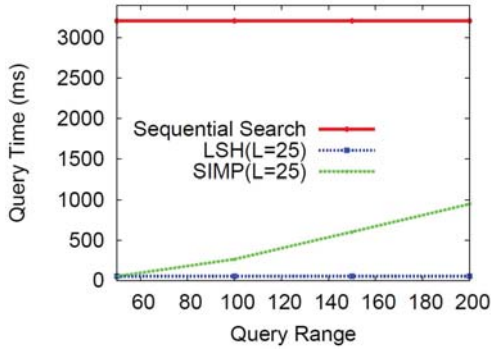


Figure 2.15: Query time of SIMP on 128-dimensional real datasets of varying sizes for varying number of hashtables L.

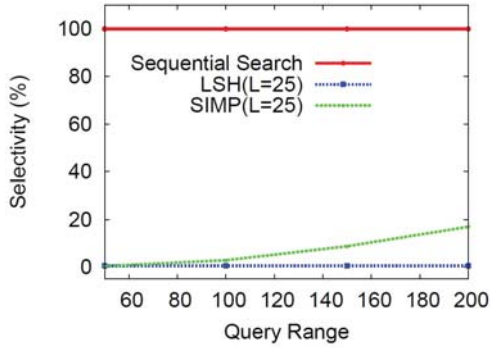
SIMP and LSH. We learned the values of w=350 and k'=8 for LSH using r_0=50. We observed that SIMP is 60 times faster than sequential search for r_q=50. For r_q=50, LSH had a query time of 54ms and a selectivity of 0.59% compared to a query time of 53ms and a selectivity of 0.48% for SIMP. We found that recall of LSH fell to 38.10% for r_q=200 unlike SIMP which had 100% recall for all query ranges.



(a) Recall



(b) Query Time



(c) Selectivity

Figure 2.16: Comparative study of performance of SIMP with p-Stable LSH on 128-dimensional 10 million SIFT points.

2.5.4 Effectiveness of Pruning Criteria

We performed experiments on CHist dataset to study the pruning effectiveness of spatial intersection pruning (SIP) and metric pruning (MP) for varying query ranges. We show the results in Figure 2.17 for n_z=5,000 and Figure 2.18 for n_z=15,000. We observed that the total pruning achieved by SIMP decreases with an increase in the query range for a given number of mballs n_z. SIP being the first step contributes the most to the pruning. MP provides an additional pruning over SIP. We also observed that the contribution of MP increases with an increase in the query range.

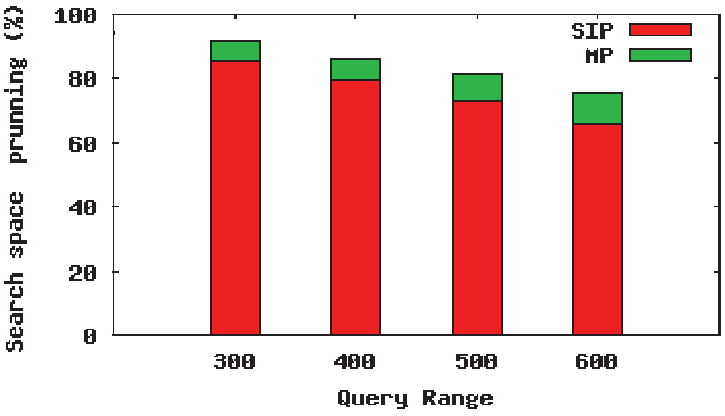


Figure 2.17: Data pruning (%) obtained by SIP and MP pruning steps of SIMP using n_z=5,000 mballs for varying query ranges on CHist dataset.

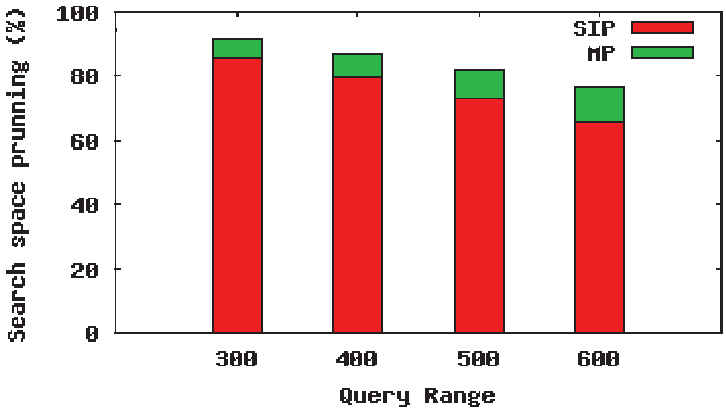


Figure 2.18: Data pruning (%) obtained by SIP and MP pruning steps of SIMP using n_z=15,000 mballs for varying query ranges on CIIist dataset.

2.6 Parameter Selection for SIMP

The tunable parameters of SIMP are the number of hashtables L, the radial bin-width w_r of polar grids, and the number of mballs n_z used for metric pruning. The parameters L and w_r play a similar role for SIMP as the number of hashtables and the bin-width of p-stable LSII. The parameter w_r is learned by training SIMP on a dataset using a training query range r_0. Though SIMP outperforms existing techniques even for L=1 hashtable, a better performance is achieved by using a larger number of hashtables. The value of L can be determined based on the available memory. The mcenters of SIMP play a similar role as the reference points of iDistance. The number of mballs n_z should be determined based on the data distribution. It can be computed by fixing a value of Root Mean Square Error

for each cluster. It can also be learned by the methods proposed by Jagadish et al. [60] for iDistance. We suggest to use a value of n_z=5,000.

2.7 Conclusions

In this chapter, we proposed SIMP for answering r-NN queries in a high-dimensional space. SIMP offers both 100% accuracy and efficiency for any query range unlike state-of-the-art methods. SIMP uses projection, spatial intersection, and triangle inequality to achieve a high rate of pruning, and thus gains high performance. We efficiently implemented the spatial intersection approach by hashing. We also developed statistical cost models to measure SIMP's performance. SIMP captures data distribution through its viewpoints and mcenters. We empirically showed a better performance of SIMP over p-Stable LSH and iDistance on three real datasets of dimensions 32, 128, and 256 and sizes 10 million, 1 million, and 1.08 million respectively. We also showed a much superior performance of SIMP over Multi-Probe LSH and LSB tree on two real datasets of dimensions 128 and 256 and sizes 1 million and 1.08 million respectively. We empirically validated on the datasets of sizes up to 100 million and dimensions up to 256 that SIMP scales linearly with the query range, the dataset size, and the dataset dimension.

Chapter 3

Querying Patterns by Keywords in Multi-Dimensional Datasets

Keyword-based pattern search in text rich multi-dimensional datasets facilitates many novel applications and tools. This chapter introduces querying patterns by keywords. We consider a dataset of objects that have keywords and are embedded in a vector space, and queries that ask for the tightest groups of points satisfying a given set of keywords. We name these queries *nearest keyword set search* (NKS) queries. We propose ProMiSH (Projection and Multi Scale Hashing) that uses random projection and hash-based index structures to query results, and achieves high scalability and speed-up. We present an exact and an approximate version of the algorithm. Our empirical studies, both on real and synthetic datasets, show that ProMiSH has a speed-up of more than four orders over state-of-the-art tree-based techniques. Our scalability tests on datasets of sizes up to 10 million and dimensions up to 100 for queries of sizes up to 9 show

that ProMiSH scales linearly with the dataset size, the dataset dimension, the query size, and the result size.

3.1 Introduction

Objects (e.g., images, documents, or chemical compounds) are characterized by a collection of relevant features, and are commonly represented as points in a high dimensional attribute space. For example, images (documents) are represented by vectors of their colors (words). These objects very often have descriptive text information associated with them, for example, descriptive tags in images or chemical compounds or names of geo-locations in maps. In this chapter, we consider multi-dimensional datasets where each data point has a set of keywords. The presence of keywords allows for the development of new tools for querying and exploring these multi-dimensional datasets.

A variety of queries on text-rich datasets have been studied in the literature, such as spatial keyword search [33], spatial preference query [133], and location-specific keyword search [55, 142]. In this chapter, we study *nearest keyword set search* (NKS) queries on text rich multi-dimensional datasets. An NKS query is a set of user provided keywords. The top-1 result of an NKS query is a set of data points which contains all the query keywords and the points form the tightest

cluster in the high dimensional space. Figure 3.1 illustrates an NKS query. The high dimensional points in the dataset are represented by circles. Each point has a unique identifier and is tagged with a set of keywords. For a query Q=$\{a, b, c\}$, the set of points {7, 8, 9} contains all the query keywords $\{a, b, c\}$ and are nearest to each other compared to any other set of points containing these query keywords. Therefore, the set of points {7, 8, 9} is the top-1 result for the query Q.

NKS queries are useful for many applications. These can be used to extend existing image (web) search engines where images (web pages) are represented by their feature vectors in high dimensional spaces. If there is no image (web page) that contains all the query keywords given by an user, then the search engine can return a set of the most similar images (web pages) which contains all the query keywords. The similarity between images (web pages) is measured by the distance between their feature vectors. The smaller the distance between images (web pages), the higher the similarity between them. NKS queries are also helpful for extending existing map services[1]. A person moving into a new place may be interested in finding an apartment that is very near to a school and also to a hospital using a map service. Here the query Q is $\{apartment, school, hospital\}$ and the nearness between places is measured by the distance between their geo-

[1]http://maps.google.com

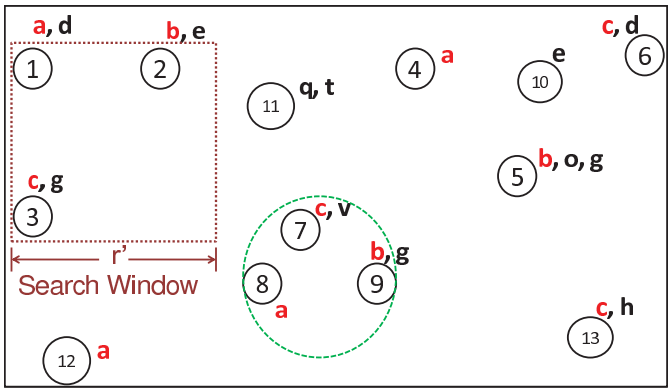


Figure 3.1: An example of an NKS query on a keyword tagged multi-dimensional dataset. Query is Q={a, b, c}. The top-1 result is the set of points {7, 8, 9}. This figure also shows an example of a localized search: a sliding window of side length r' prunes unwanted candidates like $\{2, 12, 13\}$.

attributes. NKS queries are also useful for enhancing GIS systems[2] and for geo-tagging of objects and regions [137].

Query Definition: Let $\mathcal{D} \subset \mathcal{R}^d$ be a d-dimensional dataset having N points. Each point $o \in \mathcal{D}$ has a unique identifier (id). Each point is also tagged with a set of keywords $\sigma(o)$={$v_1, .., v_t$} $\subseteq \mathcal{V}$, where $\mathcal{V}$ is a dictionary of size U of all the unique keywords in $\mathcal{D}$. We use L_2 (Euclidean norm) to measure distance between any two points $o_i, o_j \in \mathcal{D}$, i.e., $dist(o_i, o_j) = ||o_i - o_j||_2$. We measure the nearness of a set of points A by the maximum distance between any two points in A, called

[2]http://www.geabios.com

$\mathcal{D}$: A dataset
$\mathcal{V}$: A dictionary of unique keywords in $\mathcal{D}$
Q : A set of keywords comprising a query
o : A point in $\mathcal{D}$
v : A keyword
$N(v)$: Number of points in $\mathcal{D}$ having keyword v
N : Number of points in $\mathcal{D}$
U : Number of unique keywords in $\mathcal{D}$
q : Number of keywords in query Q
d : Number of dimensions of a point
t : Number of keywords per point
k : Number of top results
w_0 : Initial bin-width for hashtable
m : Number of unit random vectors used for projection
L : Number of Hashtable-Inverted Index structures
s : A scale value
r : Diameter of a set of points
z : A d-dimensional unit random vector

Table 3.1: A descriptive list of notations used in the chapter.

diameter $r(A)$.

$$r(A) = \max_{\forall o_i, o_j \in A} ||o_i - o_j||_2$$

A relatively small value of $r(A)$ implies that the points in A are very near to each other or the corresponding objects are very similar to each other. A q-size NKS query Q=$\{v_{Q1}, ..., v_{Qq}\}$ has q unique keywords provided by a user. Set $A \subseteq \mathcal{D}$ is a possible result, called a *candidate*, of Q if it contains points for all the query keywords, i.e., $Q \subseteq \bigcup_{o \in A} \sigma(o)$, and no subset of A does so. We allow overlapping candidates. If $\mathcal{S}$ is the set of all candidates of Q, then a result of Q is the candidate

A^* such that

$$A^* = \arg\min_{A \in \mathcal{S}} r(A).$$

A top-k NKS query retrieves k candidates having the least diameters. If two candidates have equal diameter, then they are further ranked by their cardinality.

A naive solution for an NKS query can incur unmanageable costs. One solution is to use an inverted index to find all points in the dataset which contain at least one query keyword. For a query Q of size q, selected points can be grouped by query keywords into q groups. Then one can generate all the candidates by a cartesian product of the q groups, and pick the top-k candidates with the least diameters. Let $N(v)$ be the number of points tagged with a query keyword v. The number of candidates explored by the naive method would be $\prod_{i=1}^{q} N(v_i)$ and the cost of search for top-1 result in a d-dimensional dataset would be $d \times \prod_{i=1}^{q} N(v_i)$. This search cost can be very large for large values of q or $N(v)$.

A search method using a data partitioning tree-based index was proposed by Zhang et al. [135, 137] to solve NKS queries on multi-dimensional datasets. The performance of this algorithm deteriorates sharply with an increase in the dimension of the dataset as the pruning techniques become ineffective. Our empirical results show that this algorithm may take hours to terminate for a high dimensional dataset having only few thousands points. That a tree-based algorithm

does not scale with an increase in the dimension of the dataset was also noted by the authors.

NKS queries are useful for many applications as discussed earlier. These applications use datasets of number of dimensions. For example, map services use location information which is two dimensional, multi-attribute geo data [81] are represented using feature vectors having dimensions in tens, and feature vectors of images and documents have dimensions in hundreds. Therefore, there is a need for an efficient algorithm that scales linearly with the dataset dimension. An algorithm also needs to yield practical query times on large datasets.

We propose the ProMiSH (*Projection and Multi-Scale Hashing*) algorithm to efficiently solve NKS queries. We present an exact (*ProMiSH-E*) and an approximate (*ProMiSH-A*) version of the algorithm. ProMiSH-E always efficiently retrieves the true top-k results, and therefore has 100% accuracy. ProMiSH-A is much more time and space efficient but returns results whose diameters are within a small approximation ratio of the diameters of the true results. Both the algorithms scale linearly with the dataset dimension, the dataset size, the query size, and the result size. Thus, ProMiSH possesses all the three desired characteristic of a good search algorithm: 1) high quality of results (accuracy), 2) high efficiency, and 3) good scalability.

ProMiSH-E uses a set of hashtables and inverted indices to perform a localized search of results. Hashtables are created by projecting points onto unit random vectors, and then splitting the lines of projected values into overlapping bins of equal width. ProMiSH-E hashtables are inspired from Locality Sensitive Hashing (LSH) [31], which is a state-of-the-art method for nearest neighbor search in high dimensional spaces. The index structure of ProMiSH-E supports accurate search, unlike LSH-based methods that allow only approximate search with probabilistic guarantees. ProMiSH-E creates hashtables at multiple bin-widths, called scales. A search starts with the hashtable at the lowest scale and sequentially proceeds to higher scales, until a termination condition is met. A search in a hashtable yields subsets of points that contain query results.

ProMiSH-E explores a subset of points obtained from a hashtable using a novel pruning based strategy. Points in the subset are grouped by query keywords. These groups are ordered based on their pairwise inner joins. A greedy method is proposed to obtain the ordering of the groups as the computation of optimal ordering is NP-hard. Finally, results are obtained by an efficient multi-way distance join of the groups. ProMiSH-A is an approximate variation of ProMiSH-E to achieve even more space and time efficiency.

We evaluated the performance of ProMiSH on both real and synthetic datasets. We used state-of-the-art Virtual bR*-Tree [137] as a reference method for com-

parison. The empirical results show that ProMiSH consistently outperforms Virtual bR*-Tree on datasets of all dimensions. The difference in performance of ProMiSH and Virtual bR*-Tree grows to more than four orders of magnitude with an increase in the dataset dimension, the dataset size, and the query size. Our scalability tests on datasets of sizes up to 10 million and dimensions up to 100 for queries of sizes up to 9 show that ProMiSH scales linearly with the dataset size, the dataset dimension, the query size, and the result size. Our datasets had as many as 24,874 unique keywords and a data point was tagged with a maximum of 14 keywords. The space cost analysis of the algorithms show that ProMiSH-A is much more space efficient than both ProMiSH-E and Virtual bR*-Tree.

Our main contributions are: (1) a novel multi-scale index structure for scalable answering of NKS queries, (2) an efficient candidate generation technique from a subset of points, and (3) extensive empirical studies. We present a detailed literature survey in Section 3.2. We discuss preliminary ideas about our index structure and algorithm in Section 3.3. We design our index structures in Section 3.4. An exact search algorithm (ProMiSH-E) that finds relevant subsets of points is described in Section 3.5. Section 3.6 discusses how answers are generated from the subsets. We propose an approximate algorithm (ProMISH-A) and derive its upper approximation ratio bound in Section 3.7. We analyze the cost of ProMiSH in Section 3.8. Finally, we present empirical results in Section 3.9.

3.2 Literature Survey

A variety of queries, semantically different from our NKS queries, have been studied in literature on text-rich spatial datasets. Location-specific keyword queries on the web and in the GIS systems [142, 55, 123, 71] were answered using a combination of R-Tree [52] and inverted index. Felipe et al. [33] developed IR^2-Tree to rank objects from spatial datasets based on a combination of their distances to the query locations and the relevance of their text descriptions to the query keywords. IR^2-Tree is an R-Tree which contains a signature of the textual content of a subtree in the subtree's root. Cong et al. [28] integrated R-tree and inverted file to answer a similar query as Felipe et al. [33] using a different ranking function. Martins et al. [91] computed text relevancy and location proximity independently, and then combined the two ranking scores. Cao et al. [20] recently proposed algorithms for spatial group keyword query defined by a query location and a set of query keywords. They proposed to retrieve a group of spatial web objects such that the group's keywords cover the query's keywords and the objects in the group are nearest to the query location and have the lowest inter-object distances. Other keyword-based queries on spatial datasets are aggregate nearest keyword search in spatial databases [83], top-k preferential query [133], finding top-k sites in a spatial data based on their influence on feature points [131], opti-

mal location queries [38, 136], and retrieving top-k prestige-based relevant spatial web objects [19].

Our NKS query is similar to the m-closest keywords query of Zhang et al. [135]. They designed bR*-Tree based on a R*-tree [10] that also stores bitmaps and minimum bounding rectangles (MBRs) of keywords in every node along with points MBRs. The candidates are generated by a priori algorithm [2]. They prune unwanted candidates based on the distances between MBRs of points or keywords and the best found diameter. Their pruning techniques become ineffective with an increase in the dataset dimension as there is large overlap between MBRs due to the curse of dimensionality. This leads to an exponential number of candidates and large query times. A poor estimation of starting diameter further worsens the performance of their algorithm. bR*-Tree also suffered from a high storage cost, therefore Zhang et al. modified bR*-Tree to create Virtual bR*-Tree [137] in memory at run time. Virtual bR*-Tree is created from a pre-stored R*-Tree which indexes all the points, and an inverted index which stores keyword information and path from the root node in R*-Tree for each point. Both, bR*-Tree and Virtual bR*-Tree, are structurally similar, and use similar candidate generation and pruning techniques. Therefore, Virtual bR*-Tree shares similar performance weaknesses as bR*-Tree.

Many tree-based indices, e.g., R-Tree [52] and M-Tree [26], have been proposed for an efficient near neighbor search in high dimensional spaces. Tree-based indices fail to scale to dimensions greater than 10 because of the curse of dimensionality [128]. VA-file [128] and iDistance [60] provide better scalability with the dataset dimension. However, the task of designing an efficient method for solving NKS queries by adapting VA-file or iDistance is not obvious.

Methods based on random projections [62] and hashing [72, 49, 31, 122] have come to be state-of-the-art methods for an efficient near neighbor search in datasets of high dimensions. Datar et al. [31] used random vectors constructed from p-stable distributions to project points, and then computed hash keys for the points by splitting the line of projected values into disjoint bins. They concatenated hash keys obtained for a point from m random vectors to create a final hash key for the point. All points were indexed into a hashtable using their hash keys. Our index structure is inspired from the same.

Multi-way distance joins of a set of multi-dimensional datasets, each of which is indexed into a R-Tree, have been studied in literature [103, 102]. As discussed above, a tree-based index fails to scale with the dimension of the dataset. Further, it is not straightforward to adapt these algorithms if every query requires a multi-way distance join only on a subset of the points of each datasets.

3.3 Preliminaries

In this section, we first describe how a localized search makes answering NKS queries efficient. Then we discuss the idea of projection used to create our index structure. We also present two lemmas which ensures that the exact search algorithm ProMiSH-E always finds the true top-k results. Finally, we empirically show using a statistical model that our index structure supports a very efficient and accurate search. A list of commonly used notations in the chapter is presented in Table 3.1.

We illustrate a **localized search** with an example. Figure 3.1 shows a query Q=$\{a, b, c\}$ and its top-1 result set $\{7, 8, 9\}$. We see from the figure that the number of points tagged with keyword a is 4, keyword b is 3, and keyword c is 4. Therefore, the total number of candidates explored by a naive method would be 48. One of the candidates explored by the naive method is the set $\{2, 12, 13\}$. For a localized search, we create a sliding window of side length r'. We anchor the window at a point having a query keyword, e.g., point 1 having keyword a in Figure 3.1. We perform a search only on the points which lie in the overlapping region of the window and the data space. This search using a window of side length r' is repeated for every point in the dataset that contains a query keyword. It can be seen that such a localized search with a reasonable estimate

of r' prunes unpromising candidates like $\{2, 12, 13\}$, making the search efficient. A search can start with a small sliding window of side length r'. Then, r' can be progressively increased with every iteration until a candidate is found in one of the overlapping regions. In this chapter, we design a suitable index structure and a search technique to implement this idea.

We use the idea of projection of points on unit random vectors to create our index structure. The projection of any two points o_1 and o_2 in $\mathcal{R}^d$ is defined by $o_1.o_2 = \sum_{i=1}^{d} o_{1i} \times o_{2i}$. We project all the points in a dataset $\mathcal{D}$ onto a unit random vector z. We split the line of projected values into overlapping bins of equal width w as shown in Figure 3.2. Each point is assigned a hash key based on the bin in which it lies. Since the line is split into overlapping bins, each point lies in two bins, and therefore gets two hash keys. For example, the line of projected values T in Figure 3.2 has been split into overlapping bins {x1, x2, x3, y1, y2, y3}. Point o lies in bins x1 and y2, and therefore gets two hash keys corresponding to each of the bins. We assign hash keys to a point o as follows:

$$\mathbf{h}_1(o) = \lfloor \frac{z.o}{w} \rfloor \tag{3.1}$$

$$\mathbf{h}_2(o) = \lfloor \frac{z.o - \frac{w}{2}}{w} \rfloor + C \tag{3.2}$$

where C is a constant to distinguish values of $\mathbf{h}_1$ and $\mathbf{h}_2$.

ProMiSH generates m-size signatures for a point o by concatenating its hash keys obtained from m unit random vectors. Since a point gets two hash keys from each unit random vector, a total of 2^m signatures are generated for it. For example, let z_1 and z_2 be two unit random vectors for m=2. Let the hash keys of a point o be $\{x_1, y_1\}$ from z_1 and $\{x_2, y_2\}$ from z_2. ProMiSH creates 2^2 2-size signatures $\{x_1x_2, x_1y_2, y_1x_2, y_1y_2\}$ for o by concatenation. These signatures are used to index points into a hashtable. Each bucket of the hashtable contains a subset of points in the dataset. To achieve efficiency, ProMiSH performs a search in each promising buckets of the hashtable independently for answering an NKS query.

Next we discuss two lemmas which guarantee that ProMiSH-E always retrieves the true top-k results for an NKS query using the index structure.

Lemma 2. *Let $\mathcal{R}^d$ be a d-dimensional Euclidean space. Let z be a vector uniformly picked from a unit (d-1)-sphere such that $z \in \mathcal{R}^d$ and $||z||_2 = 1$. For any two points o_1 and o_2 in $\mathcal{R}^d$, we have $||o_1 - o_2||_2 \geq ||z.o_1 - z.o_2||_2$.*

Proof. Since, an Euclidean space with dot product is an inner product space, we have

$$\begin{aligned} ||z.o_1 - z.o_2||_2 &= |z.(o_1 - o_2)| \\ &\leq ||z||_2 \times ||o_1 - o_2||_2 \\ &= ||o_1 - o_2||_2 \text{ since } ||z||_2 = 1 \end{aligned}$$

The inequality follows from Cauchy-Schwarz inequality. □

Lemma 3. *If a set of points $A = \{o_1, .., o_n\}$ in $\mathcal{R}^d$ with diameter r is projected onto a d-dimensional unit random vector z, and the line is split into overlapping bins of equal width $w \geq 2r$, then all the points of set A are contained in one of the bins.*

Proof. From Lemma 2 and the definition of diameter, we have $\forall o_i,\ o_j \in A,\ |z.o_i - z.o_j| \leq ||o_i - o_j|| \leq r$. Therefore, the span of projected values of the points in set A, i.e., $\max(z.o_1, ..., z.o_n) - \min(z.o_1, ..., z.o_n)$, is $\leq r$. Since the line is split into overlapping bins of width $2r$, it follows from the construction, as shown in Figure 3.2, that a line segment of width r is fully contained in one of the bins. Hence, all the points in set A will lie in the same bin. □

We illustrate here with an example how Lemma 3 guarantees retrieval of true results. For a query Q, let the diameter of its top-1 result be r. We project

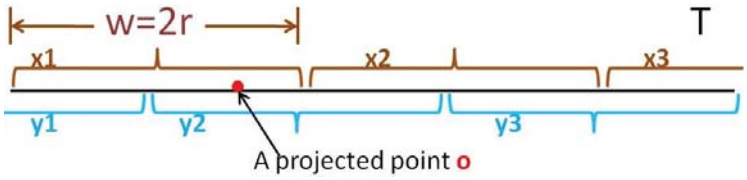
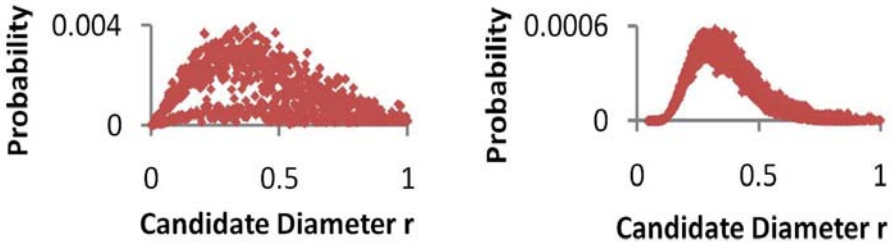


Figure 3.2: Division of projected values of points on a unit random vector into overlapping bins of equal width w=2r.

all the data points in $\mathcal{D}$ on a unit random vector and split the projected values into overlapping bins of bin-width $2r$. Now if we perform a search in each of the bins independently, then Lemma 3 guarantees that the top-1 result of query Q is certainly found in one of the bins.

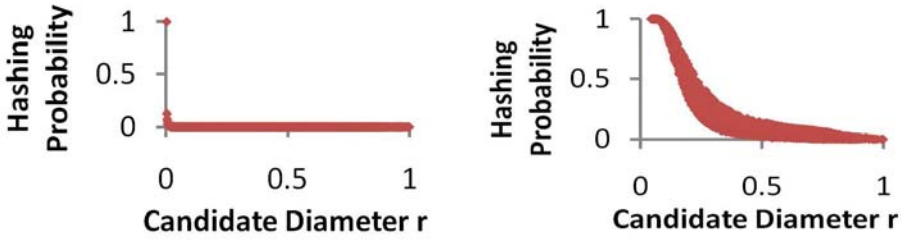


(a) d=2 (b) d=16

Figure 3.3: Probability mass functions f_r of diameters of candidates of a query of size 3 on a 2-dimensional and a 16-dimensional real datasets.

Dataset Dimension d	2	4	8	16	32
Percentage Ratio ($\frac{N_p}{N_n}$)	0.007	0.3	5.8	22	47

Table 3.2: Percentage ratio of the expected number of candidates N_p to the total number of candidates N_n of a query.



(a) d=2 (b) d=16

Figure 3.4: Values of $Pr(A|r)^2$ for varying diameters of candidates of a query of size 3 on a 2-dimensional and a 16-dimensional real datasets.

Statistical Model: We develop a statistical model to compute the expected number of candidates explored by ProMiSH in a hashtable. We show using this model that ProMiSH effectively prunes the false candidates. Let $\mathcal{D}$ be a d-dimensional dataset of size N where each point o is tagged with one keyword. Let Q be an NKS query with q keywords $\{v_{Q1}, ..., v_{Qq}\}$. Let set $A^* \subset \mathcal{D}$ with diameter r^* be the top-1 result of query Q.

Let f_v be the probability mass function of the keywords $v \in \mathcal{V}$. Using f_v, we get the number of points tagged with a query keyword v_Q as $N(v_Q) = f_v(v_Q) \times N$. Therefore, the total number of candidates for query Q in $\mathcal{D}$ is

$$N_n = \prod_{i=1}^{q} f_v(v_{Qi}) \times N \tag{3.3}$$

Let f_r be the probability mass function of diameters of candidates of Q. Then, the total number of candidates of Q having diameter r is given by

$$N_r = f_r(r) \times N_n \tag{3.4}$$

We select all the points in $\mathcal{D}$ which contain at least one query keyword v_Q. We project these points on a unit random vector z. We split the line of projected values into overlapping bins of equal width $w = 2r^*$. Let A be a candidate of query Q with diameter r. Let $Pr(A|r)$ be the conditional probability that A is fully contained within a bin. This probability is computed for random unit vectors. If this process is repeated on m independent unit random vectors, then the joint probability that a candidate A is contained in a bin in each of the m vectors is $Pr(A|r)^m$. A hashtable of ProMiSH is created using m unit random vectors. Therefore, the expected number of candidates explored by ProMiSH in a hashtable is

$$N_p = \sum_r Pr(A|r)^m \times N_r \tag{3.5}$$

We empirically computed the probability mass function f_r, the probability $Pr(A|r)^m$, and the ratio of N_p to N_n. We used real datasets of varying dimensions each having N=1 million points for our experiments. We describe these real datasets in Section 3.9. We used randomly selected queries of size q=3. We show probability mass functions f_r of diameters of candidates of a query Q on datasets of dimensions d=2 and d=16 in Figure 3.3. We computed the diameters of all the candidates of a query Q in the dataset to obtain f_r and r^*. The diameters of the candidates were scaled to lie between 0 and 1. We show values of $Pr(A|r)^2$ for varying diameters of candidates of a query Q on datasets of dimensions d=2 and

d=16 in Figure 3.4. To compute $Pr(A|r)$, we randomly chose a candidate A of diameter r. We projected all the points of A on one million unit random vectors. Then we computed the number of vectors on each of which all the points in A lie in the same bin.

We make following observations from the above studies: (a) the diameters of the candidates of a query have a heavy-tailed distribution, and (b) the value of $Pr(A|r)^m$ decreases exponentially with an increase in the diameter of the candidate of a query. The first observation implies that a large number of the candidates have diameters much larger than r^*. The second observation implies that the candidates with diameter larger than r^* have much smaller chance of falling in a bin than A^*, and thus being probed by ProMiSH. Therefore, most of the false candidates, i.e., candidates with diameters larger than r^*, are effectively pruned out by ProMiSH using its index structure.

We show percentage ratio of N_p to N_n in Table 3.2 for datasets of varying dimensions. Each ratio was computed as an average over 50 random queries. We observe from Table 3.2 that ProMiSH prunes more than 99% of the false candidates for datasets of low dimensions, e.g., d=2. For datasets of high dimensions, e.g., d=32, more than 50% of the false candidates get pruned.

3.4 Index for Exact search

The index structure of ProMiSH-E has two main data structures. The first data structure is a keyword-point inverted index $\mathcal{I}_{kp}$ which indexes all the points in the dataset $\mathcal{D}$ using their keywords. $\mathcal{I}_{kp}$ is shown with a broken maroon rectangle in Figure 3.5. The second data structure consists of multiple hashtables and their corresponding inverted indices. We call a hashtable $\mathcal{H}$ together with its corresponding inverted index $\mathcal{I}_{khb}$ as the $\mathcal{HI}$ structure.

We create a hashtable $\mathcal{H}$ as follows. We randomly choose m d-dimensional unit vectors. For each unit random vector z, we compute the projection $z.o$ for each point o in $\mathcal{D}$. Next we split the line of projected values using overlapping bins of width w as shown in Figure 3.2. We compute hash keys for each point using Equations 3.1 and 3.2. Each point o gets two hash keys $\{b_{1i}, b_{2i}\}$ from each unit random vector z_i. Thus, we have m pairs of hash keys for each data point o. We take a cartesian product of these m pairs of hash keys to generate 2^m signatures for each point o. A signature $sig(o)$=$\{b_{j1}, ..., b_{jm}\}$ of a point o contains a hash key from each of the m pairs. We hash each point o using each of its 2^m signatures as hash key into the hashtable $\mathcal{H}$. A signature $sig(o)$ of a point o is converted into a hashtable bucket identifier (bucket id) using a standard hash function, e.g., $(\sum b_{ji} * pr_i)\%hashtable\_size$, where pr_i is a random prime number.

For each hashtable $\mathcal{H}$, we create a corresponding inverted index $\mathcal{I}_{khb}$. For each bucket of $\mathcal{H}$, we compute the union of keywords of its points. Then we index each bucket of the hashtable $\mathcal{H}$ against each of the unique keywords it contains in the inverted index $\mathcal{I}_{khb}$.

We show a $\mathcal{HI}$ structure in Figure 3.5 with a broken blue rectangle. We create $\mathcal{HI}_s$ structures for increasing bin-width w=$w_0 2^s$, where w_0 is initial bin-width and $s \in \{0, ..., L-1\}$ is the scale. If $pMax$ is the maximum span of projected values of points on any unit random vector, then

$$L = \lceil log_2 \left(\frac{pMax}{w_0} \right) \rceil \tag{3.6}$$

We do not create a $\mathcal{HI}$ index structure for $w \geq pMax$ as it puts the whole dataset $\mathcal{D}$ in a single bin.

3.5 Exact Search (ProMiSH-E)

Here, we describe the ProMiSH-E algorithm. A search starts with the $\mathcal{HI}$ structure at scale s=0. ProMiSH-E finds buckets of hashtable $\mathcal{H}$, each of which contains all the query keywords, using the inverted index $\mathcal{I}_{khb}$. Then ProMiSH-E explores each of the selected buckets using an efficient pruning based technique to generate results. ProMiSH-E terminates after exploring $\mathcal{HI}$ structure at the

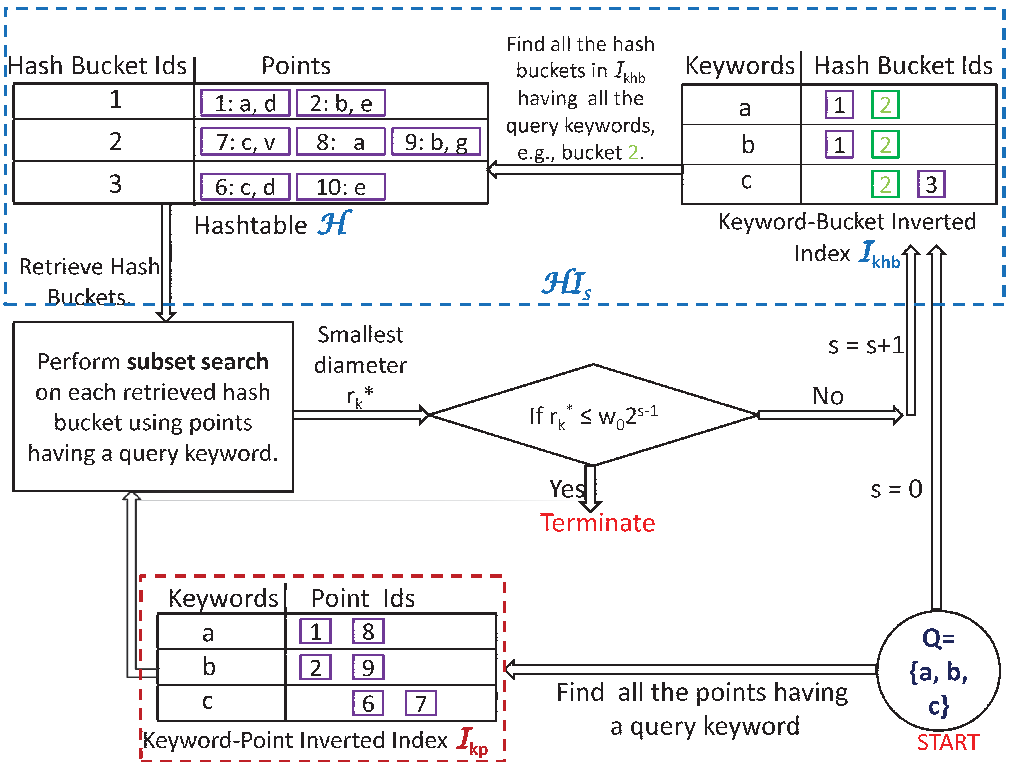


Figure 3.5: Index structure and flow of execution of ProMiSH.

smallest scale s such that the kth result has the diameter $r_k^* \leq w_0 2^{s-1}$. Figure 3.5 shows a flow of execution of ProMiSH-E.

Algorithm 3 details the ProMiSH-E algorithm. ProMiSH-E maintains a bitset BS. For each $v_Q \in Q$, ProMiSH-E retrieves the list of points corresponding to v_Q from $\mathcal{I}_{kp}$ in step 4. For each point o in the retrieved list, ProMiSH-E marks the bit corresponding to o's identifier in BS as true in step 5. Thus, ProMiSH-E finds all the points in $\mathcal{D}$ which are tagged with at least one query keyword. Next the search continues in the $\mathcal{HI}$ structures, beginning at s=0. For any given scale s,

Algorithm 3 ProMiSH-E

In: Q: query keywords; k: number of top results
In: w_0: initial bin-width
1: $PQ \leftarrow [e([\,], +\infty)]$: priority queue of top-$k$ results
2: HC: hashtable to check duplicate candidates
3: BS: bitset to track points having a query keyword
4: **for all** $o \in \cup_{\forall v_Q \in Q} \mathcal{I}_{kp}[v_Q]$ **do**
5: $BS[o] \leftarrow$ true /* Find points having query keywords*/
6: **end for**
7: **for all** $s \in \{0, ..., L-1\}$ **do**
8: Get $\mathcal{HI}$ at s
9: $E[\,] \leftarrow 0$ /* List of hash buckets */
10: **for all** $v_Q \in Q$ **do**
11: **for all** $bId \in \mathcal{I}_{khb}[v_Q]$ **do**
12: $E[bId] \leftarrow E[bId] + 1$
13: **end for**
14: **end for**
15: **for all** $i \in (0, ..., SizeOf(E))$ **do**
16: **if** $E[i] = SizeOf(Q)$ **then**
17: $F' \leftarrow \emptyset$ /* Obtain a subset of points */
18: **for all** $o \in \mathcal{H}[i]$ **do**
19: **if** $BS[o] =$ true **then**
20: $F' \leftarrow F' \cup o$
21: **end if**
22: **end for**
23: **if** checkDuplicateCand(F', HC) = false **then**
24: searchInSubset(F', PQ)
25: **end if**
26: **end if**
27: **end for**
28: /* Check termination condition */
29: **if** $PQ[k].r \leq w_0 2^{s-1}$ **then**
30: Return PQ
31: **end if**
32: **end for**
33: /* Perform search on $\mathcal{D}$ if algorithm has not terminated */
34: **for all** $o \in \mathcal{D}$ **do**
35: **if** $BS[o] =$ true **then**
36: $F' \leftarrow F' \cup o$
37: **end if**
38: **end for**
39: searchInSubset(F', PQ)
40: Return PQ

ProMiSH-E accesses the $\mathcal{HI}$ structure created at the scale in step 8. ProMiSH-E retrieves all the lists of hash bucket ids corresponding to keywords in Q from the inverted index $\mathcal{I}_{khb}$ in steps [10-11]. An intersection of these lists yields a set of hash buckets each of which contains all the query keywords in steps [12-16]. For the example in Figure 3.5, this intersection yields the bucket id 2. For each selected hash bucket, ProMiSH-E retrieves all the points in the bucket from hashtable $\mathcal{H}$. ProMiSH-E filters these points using bitset BS to get a subset of points F' in steps [17-22]. Subset F' contains only those points which are tagged with at least one query keyword. F' is a potential set and is explored further.

Subset F' is checked if it has not been explored earlier using *checkDuplicate-Cand* (Algorithm 4) in step 23. Since each point is hashed using 2^m signatures, duplicate subsets may be generated. If F' has not been explored earlier, then ProMiSH-E performs a search on it using *searchInSubset* (Algorithm 5) in step 24. Results are inserted into a priority queue PQ of size k. Each entry $e([\,], r)$ of PQ is a tuple containing a set of points and the set's diameter. PQ is initialized with k entries, each of whose set is empty and the diameter is $+\infty$. Entries of PQ are ordered by their diameter. Entries having equal diameter are further ordered by the size of their set. A new result is inserted into PQ only if its diameter is smaller than the kth smallest diameter in PQ. If ProMiSH-E does not terminate

after exploring the $\mathcal{HI}$ structure at the scale s, then the search proceeds to $\mathcal{HI}$ at the scale $(s+1)$.

Algorithm 4 checkDuplicateCand

In: F': a subset; HC: hashtable of subsets
1: $F' \leftarrow sort(F')$
2: $pr1$: list of prime numbers; $pr2$: list of prime numbers;
3: **for all** $o \in F'$ **do**
4: $pr_1 \leftarrow$ randomSelect($pr1$); $pr_2 \leftarrow$ randomSelect($pr2$)
5: $h_1 \leftarrow h_1 + (o \times pr_1)$; $h_2 \leftarrow h_2 + (o \times pr_2)$
6: **end for**
7: $h \leftarrow h_1 h_2$;
8: **if** isEmpty($HC[h]$)=false **then**
9: **if** elementWiseMatch(F', $HC[h]$) = true **then**
10: Return true;
11: **end if**
12: **end if**
13: $HC[h]$.add(F');
14: Return false;

ProMiSH-E terminates when the kth smallest diameter r_k in PQ becomes less than or equal to half of the current bin-width $w{=}w_0 2^s$ in steps [29-31]. Since $r_k \leq \frac{w_0 2^s}{2}$, Lemma 3 guarantees that each of the true candidates are contained in a bin of the hashtable, and therefore have been explored. If ProMiSH-E fails to terminate after exploring $\mathcal{HI}$ at all the scale levels $s \in \{0, ..., L-1\}$, then it performs a search on the complete dataset $\mathcal{D}$ in steps [34-39].

Algorithm *checkDuplicateCand* (Algorithm 4) uses a hashtable HC to perform a duplicate check for a subset F'. Points in F' are sorted using their identifier. Two separate standard hash functions are applied to the identifier of the points in the sorted order to generate two hash values in steps [2-6]. Both the hash values

are concatenated to get a hash key h for the subset F' in step 7. The use of multiple hash functions helps to reduce hash collisions. If HC already has a list of subsets at h, then an element-wise match of F' is performed with each subset in the list in steps [8-9]. Otherwise, F' is stored in HC using key h in step 13.

3.6 Search in a Subset

We design an algorithm for finding the tightest clusters in a subset of points. A subset is obtained from a hashtable bucket as explained in Section 3.5. Points in the subset are grouped based on the query keywords. Then all the promising candidates are explored by a multi-way distance join of these groups. The join uses r_k, the diameter of the kth result obtained so far by ProMiSH-E, as the distance threshold.

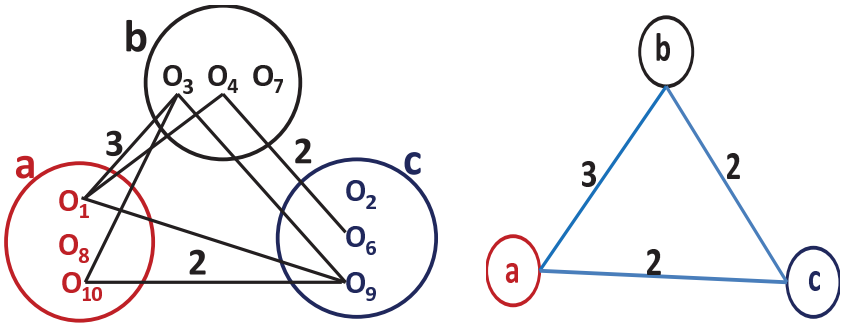
We explain a multi-way distance join with an example. A multi-way distance join of q groups $\{g_1, \cdots, g_q\}$ finds all the tuples $\{o_{1,i}, \cdots, o_{x,j}, o_{y,k}, \cdots, o_{q,l}\}$ such that $\forall x, y$: $o_{x,j} \in g_x$, $o_{y,k} \in g_y$, and $||o_{x,j} - o_{y,k}||_2 \leq r_k$. Figure 3.6(a) shows groups $\{a, b, c\}$ of points obtained for a query Q=$\{a, b, c\}$ from a subset F'. We show an edge between a pair of points of two groups if the distance between the points is at most r_k, e.g, an edge between point o_1 in group a and point o_3 in group b. A

multi-way distance join of these groups finds tuples $\{o_1, o_3, o_9\}$ and $\{o_{10}, o_3, o_9\}$. Each tuple obtained by a multi-way join is a promising candidate for a query.

3.6.1 Group Ordering

A suitable ordering of the groups leads to an efficient candidate exploration by a multi-way distance join. We first perform a pairwise inner joins of the groups with distance threshold r_k. In inner join, a pair of points from two groups are joined only if the distance between them is at most r_k. Figure 3.6(a) shows such a pairwise inner joins of the groups $\{a, b, c\}$. We see from Figure 3.6(a) that a multi-way distance join in the order $\{a, b, c\}$ explores 2 true candidates $\{\{o_1, o_3, o_9\}, \{o_{10}, o_3, o_9\}\}$ and a false candidate $\{o_1, o_4, o_6\}$. A multi-way distance join in the order $\{a, c, b\}$ explores the least number of candidates which is 2. Therefore, a proper ordering of the groups leads to an effective pruning of false candidates. Optimal ordering of groups for the least number of candidate generation is NP-hard [58].

We propose a greedy approach to find the ordering of groups. We explain the algorithm with a graph in Figure 3.6(b). Groups $\{a, b, c\}$ are nodes in the graph. The weight of an edge is the count of point pairs obtained by an inner join of the corresponding groups. The greedy method starts by selecting an edge having the least weight. If there are multiple edges with the same weight, then an edge is



(a) Pairwise inner joins (b) A graph representation

Figure 3.6: (a) a, b, and c are groups of points of a subset F' obtained for a query Q=$\{a, b, c\}$. A point o in a group g is joined to a point o' in another group g' if $||o-o'|| \leq r_k$. Examining the groups in the order $\{a, c, b\}$ generates the least number of candidates by a multi-way join. (b) A graph of pairwise inner joins. Each group is a node in the graph. The weight of an edge is the number of point pairs obtained by an inner join of the corresponding groups.

selected at random. Let the edge ac, with weight 2, be selected in Figure 3.6(b). This forms the order set $(a - c)$. The next edge to be selected is the least weight edge such that at least one of its nodes is not included in the order set. Edge cb, with weight 2, is picked next in Figure 3.6(b). Now the order set is $(a - c - b)$. This process terminates when all the nodes are included in the order. $(a - c - b)$ gives the ordering of the groups.

Algorithm 5 describes how the groups are ordered. The kth smallest diameter r_k is retrieved form the priority queue PQ in step 1. For a given subset F' and a query Q, all the points are grouped using query keywords in steps [2-5]. A pairwise inner join of the groups is performed in steps [6-18]. An adjacency list

AL stores the distance between points which satisfy the distance threshold r_k. An adjacency list M stores the count of point pairs obtained for each pair of groups by the inner join. A greedy algorithm finds the order of the groups in steps [19-30]. It repeatedly removes an edge with the smallest weight from M till all the groups are included in the order set *curOrder*. Finally, groups are sorted using *curOrder* in step 30.

3.6.2 Nested Loops with Pruning

We perform a multi-way distance join of the groups by nested loops. For example, consider the set of points in Figure 3.6. Each point $o_{a,i}$ of group a is checked against each point $o_{b,j}$ of group b for the distance predicate, i.e., $||o_{a,i} - o_{b,j}||_2 \leq r_k$. If a pair ($o_{a,i}$, $o_{b,j}$) satisfies the distance predicate, then it forms a tuple of size 2. Next this tuple is checked against each point of group c. If a point $o_{c,k}$ satisfies the distance predicate with both the points $o_{a,i}$ and $o_{b,j}$, then a tuple ($o_{a,i}$, $o_{b,j}$, $o_{c,k}$) of size 3 is generated. Each intermediate tuple generated by nested loops satisfies the property that the distance between every pair of its points is at most r_k. This property effectively prunes false tuples very early in the join process and helps to gain high efficiency. A candidate is found when a tuple of size q is generated. If a candidate having a diameter smaller than the current value of r_k is found, then the priority queue PQ and the value of r_k are updated.

Algorithm 5 searchInSubset

In: F': subset of points; Q: query keywords; q: query size
In: PQ: priority queue of top-k results
1: $r_k \leftarrow PQ[k].r$ /* kth smallest diameter */
2: $SL \leftarrow [(v, [\,])]$: list of lists to store groups per query keyword
3: **for all** $v \in Q$ **do**
4: $SL[v] \leftarrow \{\forall o \in F' : o$ is tagged with $v\}$ /* form groups */
5: **end for**
6: /* Pairwise inner joins of the groups*/
7: AL: adjacency list to store distances between points
8: $M \leftarrow 0$: adjacency list to store count of pairs between groups
9: **for all** $(v_i, v_j) \in Q$ such that $i \leq q,\ j \leq q,\ i < j$ **do**
10: **for all** $o \in SL[v_i]$ **do**
11: **for all** $o' \in SL[v_j]$ **do**
12: **if** $||o - o'||_2 \leq r_k$ **then**
13: $AL[o, o'] \leftarrow ||o - o'||_2$
14: $M[v_i, v_j] \leftarrow M[v_i, v_j] + 1$
15: **end if**
16: **end for**
17: **end for**
18: **end for**
19: /* Order groups by a greedy approach */
20: *curOrder* $\leftarrow [\,]$
21: **while** $Q \neq \emptyset$ **do**
22: $(v_i, v_j) \leftarrow$ removeSmallestEdge(M)
23: **if** $v_i \notin$ *curOrder* **then**
24: *curOrder*.append(v_i); $Q \leftarrow Q \setminus v_i$
25: **end if**
26: **if** $v_j \notin$ *curOrder* **then**
27: *curOrder*.append(v_j); $Q \leftarrow Q \setminus v_j$
28: **end if**
29: **end while**
30: sort(SL, *curOrder*) /* order groups */
31: *findCandidates*(q, AL, PQ, Idx, SL, *curSet*, *curSetr*, r_k)

The new value of r_k is used as distance threshold for future iterations of nested loops.

We describe answer exploration by nested loops using Algorithm 6 (*findCandidates*). Nested loops are performed recursively. An intermediate tuple *curSet* is checked against each point of group $SL[Idx]$ in steps [2-23]. First, it is determined

using AL whether the distance between the last point in *curSet* and a point o in $SL[Idx]$ is at most r_k in step 3. Then the point o is checked against each point in *curSet* for the distance predicate in steps [5-15]. The diameter of *curSet* is updated in steps [9-11]. If a point o satisfies the distance predicate with each point of *curSet*, then a new tuple *newCurSet* is formed in step 17 by appending o to *curSet*. Next a recursive call is made to *findCandidates* on the next group $SL[Idx + 1]$ with *newCurSet* and *newCurSetr*. A candidate is found if *curSet* has a point from every group. A result is inserted into PQ after checking for duplicates in steps [26-33]. A duplicate check is done by a sequential match with the results in PQ. For a large value of k, a method similar to Algorithm 4 can be used. If a new result gets inserted into PQ, then the value of r_k is updated in step 18.

3.7 Approximate Search (ProMiSH-A)

We first discuss ProMiSH-A which is more space and time efficient than ProMiSH-E. Then we show using a statistical model that ProMiSH-A retrieves results within a small approximation ratio of the true results with a high probability.

Algorithm 6 findCandidates

In: q: query size; SL: list of groups
In: AL: adjacency list of distances between points
In: PQ: priority queue of top-k results
In: Idx: group index in SL
In: $curSet$: an intermediate tuple
In: $curSetr$: an intermediate tuple's diameter

```
1: if Idx ≤ q then
2:   for all o ∈ SL[Idx] do
3:     if AL[curSet[Idx-1], o] ≤ r_k then
4:       newCurSetr ← curSetr
5:       for all o' ∈ curSet do
6:         dist ← AL[o, o']
7:         if dist ≤ r_k then
8:           flag ← true
9:           if newCurSetr < dist then
10:            newCurSetr ← dist
11:          end if
12:        else
13:          flag ← false; break;
14:        end if
15:      end for
16:      if flag = true then
17:        newCurSet ← curSet.append(o)
18:        r_k ← findCandidates(q, AL, PQ, Idx+1, SL, newCurSet, newCurSetr, r_k)
19:      else
20:        Continue;
21:      end if
22:    end if
23:  end for
24:  return r_k
25: else
26:  if checkDuplicateAnswers(curSet, PQ) = true then
27:    return r_k
28:  else
29:    if curSetr < PQ[k].r then
30:      PQ.Insert([curSet, curSetr])
31:      return PQ[k].r
32:    end if
33:  end if
34: end if
```

The index structure and the search method of ProMiSH-A are variations of ProMiSH-E, therefore we describe only the differences. The index structure of

ProMiSH-A differs from ProMiSH-E only in the way the line of projected values of points on a unit random vector is split. ProMiSH-A splits the line into non-overlapping bins of equal width, unlike ProMiSH-E which splits the line into overlapping bins. Therefore, each data point o gets one hash key from a unit random vector z in ProMiSH-A. A signature $sig(o)$ is created for each point o by the concatenation of its hash keys obtained from each of the m unit random vectors. Each point is hashed using its signature $sig(o)$ into a hashtable at a given scale.

The search technique of ProMiSH-A differs from ProMiSH-E in the initialization of priority queue PQ and the termination condition. ProMiSH-A starts with an empty priority queue PQ, unlike ProMiSH-E whose priority queue is initialized with k entries. ProMiSH-A checks for a termination condition after fully exploring a hashtable at a given scale. It terminates if it has k entries in its priority queue PQ. Since each point is hashed only once into a hashtable of ProMiSH-A, it does not perform a subset duplicate check or a result duplicate check.

Approximation ratio: An approximation ratio $\rho \geq 1$ is defined as the ratio of the diameter of the result reported by ProMiSH-A r to the diameter of the true result r^*, i.e., $\rho = \frac{r}{r^*}$. Next we describe a model to obtain probabilistic approximation ratio bound for ProMiSH-A.

Let $\mathcal{D}$ be a d-dimensional dataset and Q=$\{v_{Q1}, \cdots, v_{Qq}\}$ be an NKS query. The total number of candidates N_r of query Q having diameter r is given by Equation 3.4. We project all the points in dataset $\mathcal{D}$, which contain at least one query keyword v_Q, onto a unit random vector z. We split the line of projected values into non-overlapping bins of equal width w. Let $Pr(A|r)$ be the conditional probability for random unit vectors that a candidate A of query Q having diameter r is fully contained within a bin. For m independent unit random vectors, the joint probability that a candidate A is contained in a bin in each of the m vectors is $Pr(A|r)^m$. A hashtable of ProMiSH-A is created using m unit random vectors. Therefore, the probability that no candidate of diameter r is retrieved by ProMiSH-A from the hashtable is $(1-Pr(A|r)^m)^{N_r}$. Let the diameter of the top-1 result of query Q be r^*. Then the probability $P(r')$ that at least one candidate of any diameter r, where $r^* \leq r \leq r'$, is retrieved by ProMiSH-A is given by

$$P(r') = 1 - \prod_{r=r^*}^{r'} (1 - Pr(A|r)^m)^{N_r}. \tag{3.7}$$

For a given constant λ, where $0 \leq \lambda \leq 1$, we can compute the smallest value of r' from Equation 3.7 such that $\lambda \leq P(r')$. The value $\rho^*=\frac{r'}{r^*}$ gives an upper bound on the approximation ratio of the results returned by ProMiSH-A with the probability λ. We empirically computed ρ^* for queries of size q=3 for different values of λ using this model. We used a 32-dimensional real dataset having 1 million points described in Section 3.9 for our study. For a set of randomly chosen

queries of size 3, we computed the values of N_r and $Pr(A|r)^2$. We used projections on 1 million random vectors and a bin-width of w=100 for computing $Pr(A|r)^2$. We obtained the approximation ratio bound of ρ^*=1.4 and ρ^*=1.5 for λ=0.8 and λ=0.95 respectively.

3.8 Cost Analysis of ProMiSH

In this section, we discuss the time and the space cost of ProMiSH. We use the following settings for the cost analysis. Let $\mathcal{D}$ be a dataset having N d-dimensional points each of which is tagged with t keywords. Let the space cost of a point's identifier, a dimension of a point, and a keyword be E bytes individually. Let U be the number of unique keywords in $\mathcal{D}$. Let Q=$\{v_{Q1}, \cdots, v_{Qq}\}$ be an NKS query of size q. We assume that the data points are uniformly distributed across all the keywords. Therefore, the total number of the data points tagged with a keyword v is

$$N(v) = N \times (\frac{t}{U})$$

Time cost: Let the index structure of ProMiSH-E be comprised of $\mathcal{HI}$ structures at L scale levels where the value of L is obtained by Equation 3.6. Let $\mathcal{H}_s$ be the hashtable at scale s. We assume without any loss of generality that the hashtable $\mathcal{H}_s$ is created using m=1 unit random vector. Let *pSpan* be the span

of the projected values of the data points on the unit random vector. We assume that the data points tagged with a keyword v are uniformly distributed on the line of projected values. ProMiSH-E divides the the line of projected values into overlapping bins to compute the hash keys of the points using a bin-width of $w{=}w_0 2^s$. Therefore, the number of the data points having keyword v lying in a bucket b of $\mathcal{H}_s$ is

$$\begin{aligned} N(vb) &= N(v) * w/pSpan \\ &= N(v)/2^{L-s} \end{aligned}$$

We first compute the cost of search in a bucket b of $\mathcal{H}_s$. The cost of pairwise inner joins for a query Q of size q for d-dimensional data points is $(N(vb) \times q)^2 \times d/2$. Nested loop enumerates the candidates by looking up the pre-computed distances between the points from the adjacency list. Therefore, the worst case cost of the nested loop is $N(vb)^q$. The total cost of search in a bucket b of the hashtable $\mathcal{H}_s$ is

$$T(bs) = ((N(vb) \times q)^2 \times d/2) + N(vb)^q$$

The total number of buckets in $\mathcal{H}_s$ of ProMiSH-E is 2^{L-s+1}. Therefore, the cost of search in $\mathcal{H}_s$ is

$$T(\mathcal{H}_s) = 2^{L-s+1} \times T(bs)$$

ProMiSH-A divides the line of projected values into non-overlapping bins. The total number of buckets in $\mathcal{H}_s$ of ProMiSH-A is 2^{L-s}. The cost of search in the hashtable $\mathcal{H}_s$ of ProMiSH-A is

$$T(\mathcal{H}_s) = 2^{L-s} \times T(bs)$$

We show the query time of ProMiSH for NKS queries on multiple real and synthetic datasets in Section 3.9.

Space cost: The index structure of ProMiSH consists of the *keyword-point* inverted index $\mathcal{I}_{kp}$ and L pairs of hashtable $\mathcal{H}$ and *keyword-bucket* inverted index $\mathcal{I}_{khb}$. The space cost of $\mathcal{I}_{kp}$ is $S(\mathcal{I}_{kp}) = (N \times E \times t)$ bytes. For ProMiSH-E, each point is hashed into a hashtable $\mathcal{H}$ using 2^m signatures, therefore a hashtable takes $S_E(\mathcal{H}) = (2^m \times N \times E)$ bytes. For ProMiSH-A, each point is hashed using only one signature, therefore a hashtable takes $S_A(\mathcal{H}) = (N \times E)$ bytes. The space cost of a $\mathcal{I}_{khb}$ inverted index is $S(\mathcal{I}_{khb}) = (U \times M \times log_2 M/8)$ bytes, where M is the number of buckets in hashtable $\mathcal{H}$. The total space cost of the index of ProMiSH-E is $S(\mathcal{I}_{kp}) + S_E(\mathcal{H}) + S(\mathcal{I}_{khb})$. The total space cost of the index of ProMiSH-A is $S(\mathcal{I}_{kp}) + S_A(\mathcal{H}) + S(\mathcal{I}_{khb})$. We show the ratio of the index space of ProMiSH to the dataset space in Section 3.9.4.

3.9 Empirical Evaluations

We empirically evaluated the performance of ProMiSH-E and ProMiSH-A on synthetic and real datasets. We used recently introduced Virtual bR*-Tree [137] as a reference method for comparison. We briefly described Virtual bR*-Tree in Section 3.2. Virtual bR*-Tree was proposed as an improvement over bR*-Tree [135]. We first introduce the datasets and the metrics used for measuring the performance of the algorithms. Then we discuss the quality results of the algorithms on real datasets. Next we describe comparative results of ProMiSH-E, ProMiSH-A, and Virtual bR*-Tree on both synthetic and real datasets. We also report scalability test results of ProMiSH on both synthetic and real datasets. Finally, we present a comparison of the space usage of all the algorithms.

Id	Dataset Size (N)	Dictionary Size U	Average t
1	10,000	5,661	12
2	30,000	6,753	13
3	50,000	7,101	13
4	70,000	7,902	14
5	1 Million	24,874	11

Table 3.3: Description of real datasets of five different sizes.

Datasets: We used both synthetic and real datasets for experiments. Synthetic data was randomly generated. Each component of a d-dimensional synthetic point was chosen uniformly from [0-10,000]. Each synthetic point was randomly tagged with t keywords. A dataset is characterized by its (1) size, N; (2) dimen-

sionality, d; (3) dictionary size, U; and (4) the number of keywords associated with each point, t. We created various synthetic datasets by varying these parameters for our empirical studies.

Our NKS query extends the functionality of the existing keyword-based image search engines as discussed in Section 3.1. An extended search engine returns a group of similar images which contains all the user provided keywords as a result. Based on this application, we used images having descriptive tags as real datasets. We downloaded images with their textual keywords from Flickr[3]. We transformed each image into grayscale. We created a d-dimensional dataset by extracting a d-dimensional color histogram from each image. Each data point was tagged with the keywords of its corresponding image. We describe real datasets of five different sizes used in our empirical studies in Table 3.3. The largest real dataset had $24,874$ unique keywords and each point in it was tagged with 11 keywords. A query for a dataset was created by randomly picking a set of keywords from the dictionary of the dataset. A query is parameterized by its size q.

Performance metrics: We measured the performance of the algorithms by their *approximation ratio*, *query time*, and *space usage*. These metrics help us evaluate the quality of results (accuracy), the efficiency, and the scalability of the search algorithms.

[3]http://www.flickr.com/

We measured the quality of results of an algorithm by its approximation ratio [49, 122]. For $1 \leq i \leq k$, if r_i is the ith diameter in top-k results retrieved by an algorithm for a query Q and r_i^* is the true ith diameter, then the approximation ratio of the algorithm for top-k search is given by $\rho(Q) = (\sum_{i=1}^{k} \frac{r_i}{r_i^*})/k$. The smaller the value of $\rho(Q)$, the better is the quality of the results returned by the algorithm. The least value of $\rho(Q)$ is 1. We report the *average approximation ratio (AAR)* for a query of a given size, which is the mean of the approximation ratios of 50 queries.

We validated the time efficiency of the algorithms by measuring their query time. The index structure and the dataset for each method reside in memory. Therefore, the query time measured as the elapsed CPU time between the start and the completion of a query gives a fair comparison between the methods. A query was executed multiple times and the average execution time was taken as its query time. Finally, we report the query time for a query size q as an average of 50 different queries. The query time of a search algorithm mainly depends on the dataset size N, the dataset dimension d, and the query size q. Therefore, we validated the scalability of the algorithms by computing their query time for varying values of N, d, and q. We verified the space efficiency of an algorithm by computing the ratio of its index memory footprint to the dataset memory footprint.

Methods implementation: We implemented all the methods in Java. For Virtual bR*-Tree, we fixed the leaf node size to 1,000 entries and other nodes' sizes to 100 entries. Virtual bR*-Tree finds only the smallest subset, therefore we used k=1 for ProMiSH for a fair comparison. We used the value of m=2 and L=5 to create the index structure of ProMiSH-E and ProMiSH-A. For a dataset, if $pMax$ is the maximum span of projected values of data points on any unit random vector, then a value of w_0=$\frac{pMax}{2^L}$ was used as the initial bin-width.

All the experiments were performed on a machine having Quad-Core Intel Xeon CPU@2.00GHz, 4,096 KB cache, and 98 GB main memory and running 64-bit Linux version 2.6.

3.9.1 Quality Test

We validated the result quality of ProMiSH-E, ProMiSH-A and Virtual bR*-Tree by their average approximation ratio (AAR). ProMiSH-E and Virtual bR*-Tree perform an exact search. Therefore, they always retrieve the true top-k results, and have AAR of 1. We used the results returned by them as the ground truth. Figure 3.7 shows AAR computed over top-5 results retrieved by ProMiSH-A for varying query sizes on two 32-dimensional real datasets. We observe from Figure 3.7 that AAR of ProMiSH-A is always less than 1.5. This low AAR allows

ProMiSH-A to return practically useful results with a very efficient time and space complexity.

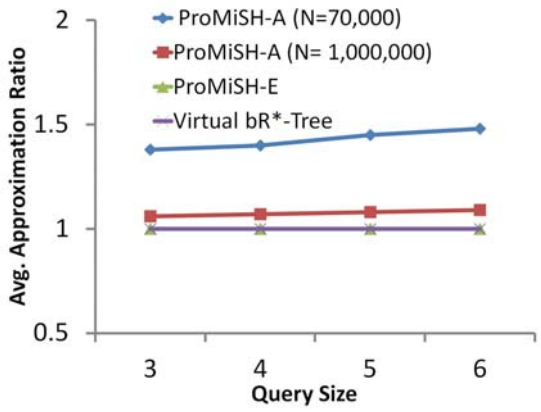


Figure 3.7: Average approximation ratio of ProMiSH-A for varying query sizes on 32-dimensional real datasets of various sizes.

3.9.2 Efficiency on Synthetic Datasets

We performed experiments on multiple synthetic datasets to verify the efficiency and the scalability of ProMiSH. We first discuss the comparison of query times of Virtual bR*-Tree, ProMiSH-A, and ProMiSH-E for varying dataset dimensions d, dataset sizes N, and query sizes q. We found that ProMiSH performs at least four orders of magnitude better than Virtual bR*-Tree. We also show results of the scalability tests of ProMiSH for varying values of N, d, q, and the result size k. Our scalability results reveal a linear performance of ProMiSH with

N, d, q, and k. All the query times were measured in milliseconds (ms) and are shown in log scale in all the figures.

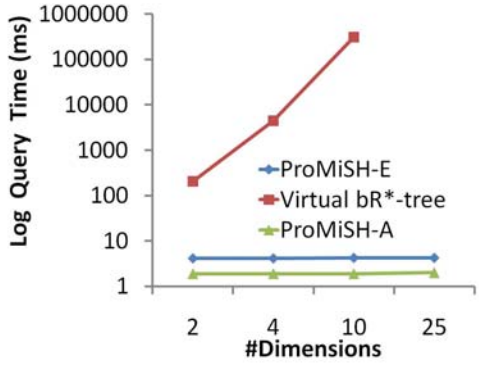


Figure 3.8: Query time comparison of algorithms for retrieving top-1 results for queries of size q=5 on synthetic datasets of varying dimensions d. Values of N=100,000, t=1, and U=1,000 were used for each dataset.

The query times of ProMiSH-E, ProMiSH-A, and Virtual bR*-Tree for retrieving top-1 results for queries of size 5 on datasets of varying dimensions d are shown in Figure 3.8. We used a dataset of 100,000 points where each point was tagged with t=1 keyword using a dictionary of size U=1,000. For the dataset of dimension 25, ProMiSH-A completed in 1.8 ms and ProMiSH-E took only 4.2 ms. Conversely, results for Virtual bR*-Tree could not be obtained since it ran for more than 5 hours. We observed that ProMiSH not only significantly outperforms Virtual bR*-Tree on datasets of all dimensions but the difference in performance also grows to more than five orders with an increase in the dataset dimension.

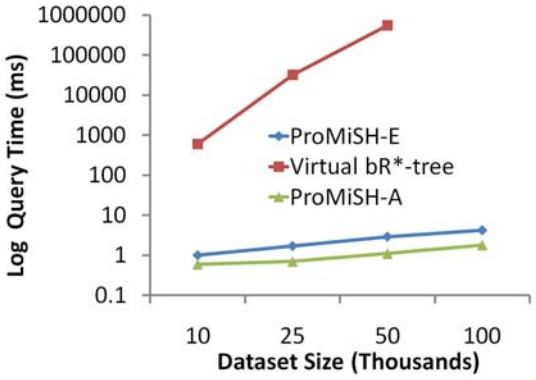


Figure 3.9: Query time comparison of algorithms for retrieving top-1 results for queries of size q=5 on 25-dimensional synthetic datasets of varying sizes N. Values of t=1 and U=1,000 were used for each dataset.

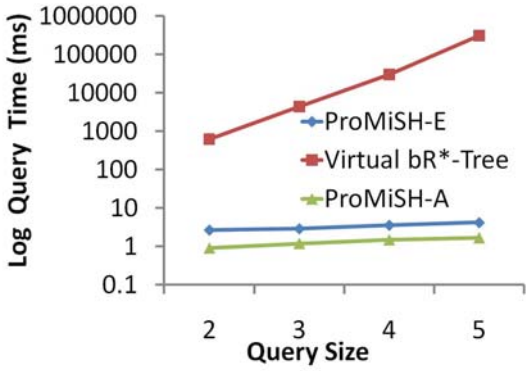


Figure 3.10: Query time comparison of algorithms for retrieving top-1 results for queries of varying sizes q on a 10-dimensional synthetic dataset having 100,000 points. Values of t=1 and U=1,000 were used for the dataset.

We show the query times of all the three algorithms on 25-dimensional datasets of varying sizes N for retrieving top-1 results for queries of size 5 in Figure 3.9. Each dataset used a dictionary of size U=1,000 and t=1 keyword per point. The query time of Virtual bR*-Tree could not be obtained for the dataset of size

N=100,000 as it failed to finish even after 5 hours of execution. We report the query times of all the three algorithms for retrieving top-1 results for queries of varying sizes q on a 10-dimensional synthetic dataset of size N=100,000 in Figure 3.10. Each data point was tagged with t=1 keyword using a dictionary of size U=1,000. For a query of size 5, ProMiSH-A had a query time of 1.7 ms, ProMiSH-E had a query time of 4.2 ms, and Virtual bR*-Tree had a query time of 305 seconds. We again observed that ProMiSH not only significantly outperforms Virtual bR*-Tree but the difference in performance grows to five orders of magnitude with an increase in the dataset size and the query size.

All the above results show that ProMiSH has a linear increase in its query time with a linear increase in the dataset size N, the dataset dimension d, and the query size q. In contrast, Virtual bR*-Tree fails to scale with q, d, and N. These results confirm that the pruning criteria of Virtual bR*-Tree, as discussed in Section 3.2, becomes ineffective with an increase in the dimension of the dataset. This leads to an exponential generation of potential candidates and large query times.

Next we present scalability results of ProMiSH-E and ProMiSH-A on large synthetic datasets of varying dimensions for large query sizes and varying result sizes. Each dataset used a dictionary of size U=200. A point in each dataset was tagged with t=1 keyword. Figure 3.11 shows the query times of both algorithms

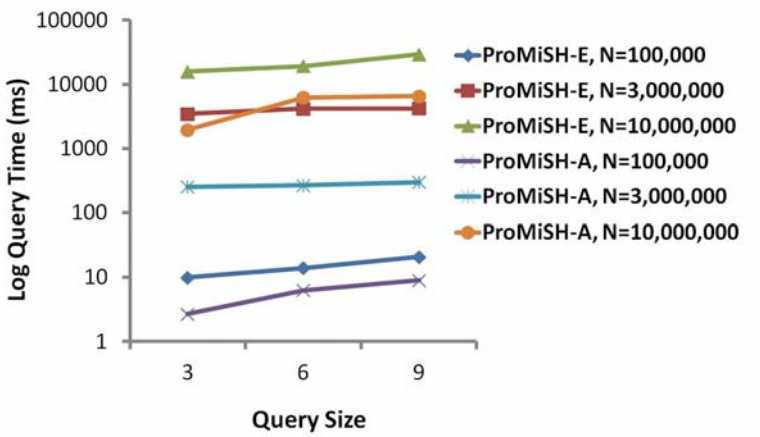


Figure 3.11: Query time analysis of ProMiSH algorithms for retrieving top-1 results for queries of varying sizes q on 25-dimensional synthetic datasets of varying sizes N. Values of t=1 and U=200 were used for each dataset.

for retrieving top-1 results for queries of varying sizes q on 25-dimensional datasets of varying sizes N. ProMiSH-E had a query time of 29 seconds and ProMiSH-A had a query time of 6 seconds for queries of size 9 on a dataset of 10 million points. We observed that ProMiSH-A is an order of magnitude faster than ProMiSH-E for queries of all sizes. We see from Figure 3.11 that ProMiSH scales linearly with the query size and the dataset size.

Figure 3.12 shows the query times of ProMiSH-E and ProMiSH-A for retrieving top-1 results for queries of varying sizes on 3 million size datasets of varying dimensions. ProMiSH-E had a query time of 4.7 seconds and ProMiSH-A had a query time of 0.3 seconds for queries of size q=9 on a 100-dimensional dataset having N=3 million points. ProMiSH-A is an order of magnitude faster than ProMiSH-E on datasets of all dimensions. We observed that both algorithms scale

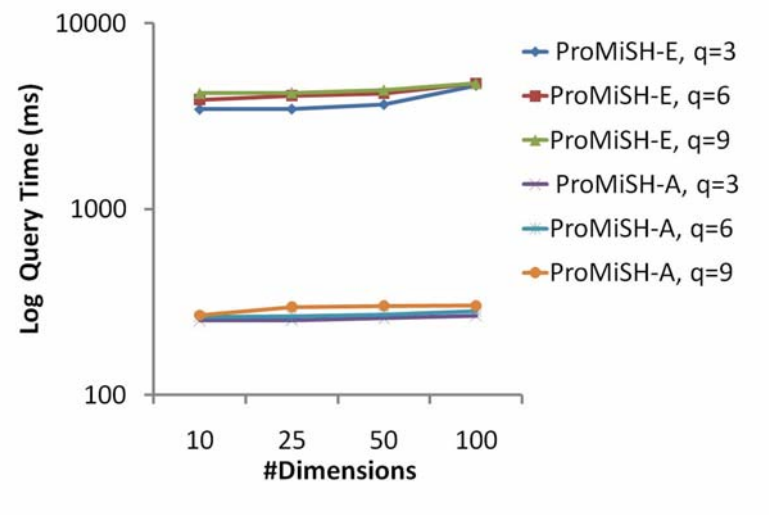


Figure 3.12: Query time analysis of ProMiSH algorithms for retrieving top-1 results for queries of varying sizes q on large synthetic datasets of varying dimensions d. Values of N=3 million, t=1, and U=200 were used for each dataset.

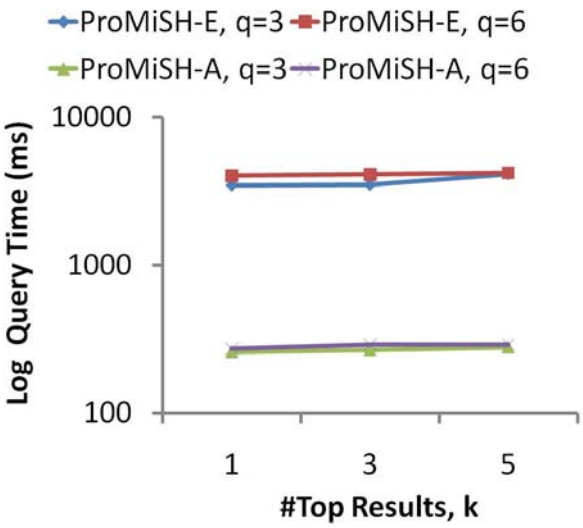


Figure 3.13: Query time analysis of ProMiSH algorithms for retrieving top-k results for queries of sizes 3 and 6 on a 50-dimensional synthetic dataset of size N=3 million. Values of t=1 and U=200 were used for the dataset.

linearly with dimension d of the dataset. Figure 3.13 shows the query times for retrieving top-k results for queries of varying sizes q on a 50-dimensional dataset having 3 million points. It reveals a linear performance of both algorithms for increasing k. ProMiSH-A is an order of magnitude better than ProMiSH-E for any result size k. All these tests show that the query time of ProMiSH scales linearly with the dataset size, the dataset dimension, the query size, and the result size.

3.9.3 Efficiency on Real Datasets

We evaluated the efficiency and the scalability of ProMiSH on multiple real datasets. We first discuss query time comparisons of alternative algorithms for varying dataset dimensions d, query sizes q, and dataset sizes N. We observed that ProMiSH is at least four orders of magnitude better than Virtual bR*-Tree. We also discuss scalability tests of ProMiSH-E and ProMiSH-A for varying values of q, d, and the result size k. These scalability results on a dataset of size 1 million reveal a linear query time performance of ProMiSH with q, d, and k. All the query times were measured in milliseconds (ms) and are shown in log scale in all the figures.

We show the query times of all the three algorithms on N=50,000 size real datasets of varying dimensions d for retrieving top-1 results for queries of size q=4

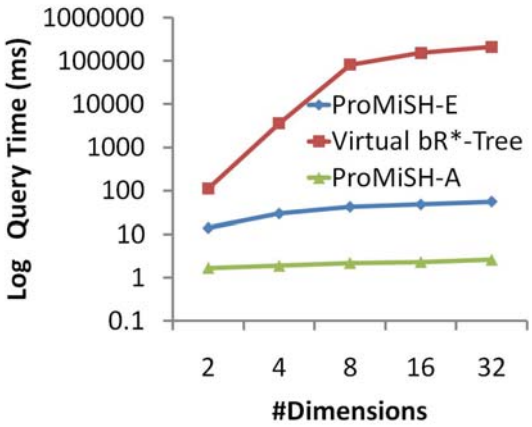


Figure 3.14: Query time comparison of algorithms for retrieving top-1 results for queries of size q=4 on real datasets of varying dimensions d and size N=50,000.

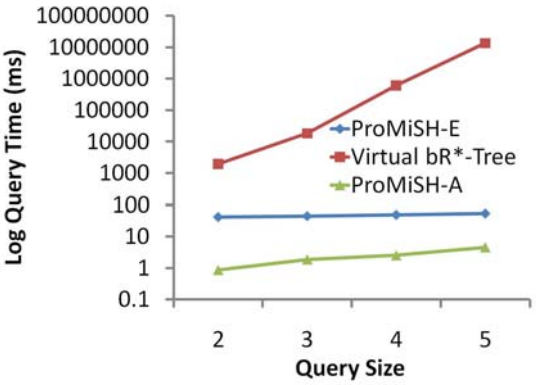


Figure 3.15: Query time comparison of algorithms for retrieving top-1 results for queries of varying sizes q on a 16-dimensional real dataset of size N=70,000.

in Figure 3.14. ProMiSH-A had a query time of 3 ms, ProMiSH-E had a query time of 55 ms, and Virtual bR*-Tree had a query time of 210 seconds for 32-dimensional dataset. The comparison of query times for retrieving top-1 results for queries of varying sizes q on a 16-dimensional real dataset of size N=70,000 is shown in Figure 3.15. ProMiSH-A had a query time of 5 ms, ProMiSH-E had

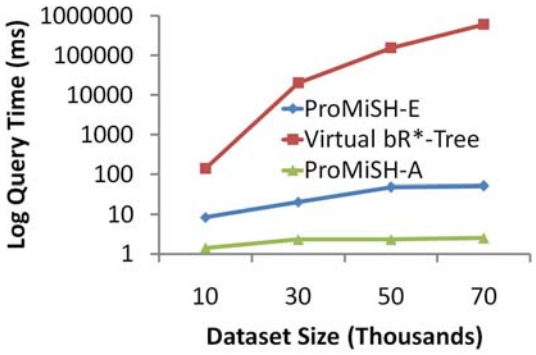


Figure 3.16: Query time comparison of algorithms for retrieving top-1 results for queries of size q=4 on 16-dimensional real datasets of varying sizes N.

a query time of 54 ms, and Virtual bR*-Tree had a query time of 13,352 seconds for queries of size q=5. The comparison of query times on 16-dimensional real datasets of varying sizes N for retrieving top-1 results for queries of size q=4 is shown in Figure 3.16. ProMiSH-A had a query time of 3 ms, ProMiSH-E had a query time of 49 ms, and Virtual bR*-Tree had a query time of 608 seconds for a dataset of size N=70,000.

The above results show that ProMiSH significantly outperforms state-of-the-art Virtual bR*-Tree on real datasets of all dimensions and sizes and on queries of all sizes. ProMiSH-E is five orders of magnitude faster than Virtual bR*-Tree for queries of size q=5 on a 16-dimensional real dataset of size 70,000. ProMISH-E is also at least four orders of magnitude faster than Virtual bR*-Tree for a queries of size q=4 on a 32-dimensional real dataset of size 50,000. ProMiSH-A always has an order of magnitude better performance than ProMiSH-E. Similar to the

observations on synthetic datasets, we find that the difference in query time of ProMiSH and Virtual bR*-Tree grows to multiple orders of magnitude with an increase in the dataset size N, the dataset dimension d, and the query size q. In addition, the query time performance of ProMiSH-A and ProMiSH-E is linear with the dataset size, the dataset dimension, and the query size, unlike Virtual bR*-Tree whose performance deteriorates sharply. This again confirms that the pruning criteria of Virtual bR*-Tree is ineffective for high dimensional datasets.

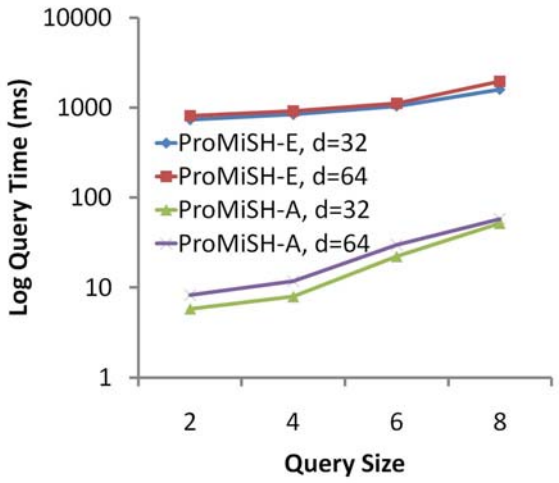


Figure 3.17: Query time analysis of ProMiSH algorithms for retrieving top-1 results for queries of varying sizes q on real datasets of varying dimensions and size N=1 million.

We performed stress tests of ProMiSH on real datasets having 1 million points of dimensions 32 and 64. Figure 3.17 shows the query times of ProMiSH-A and ProMiSH-E for varying query sizes. ProMiSH-A had a query time of 58 ms and ProMiSH-E had a query time of 1,592 ms for queries of size q=8 on 64-dimensional

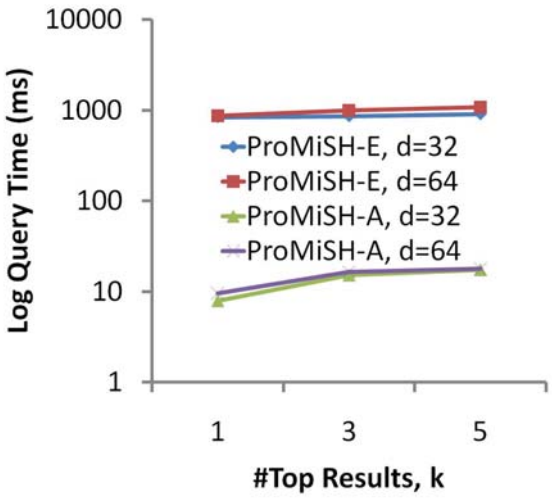


Figure 3.18: Query time analysis of ProMiSH algorithms for retrieving top-k results for queries of size q=4 on real datasets of varying dimensions and size N=1 million.

real datasets. Figure 3.18 shows the query times of ProMiSH for retrieving top-k results for queries of size q=4. ProMiSH-A had a query time of 18 ms and ProMiSH-E had a query time of 1,084 ms for top-5 results on 64-dimensional datasets. Figures 3.17 and 3.18 verify that the query time of ProMiSH increases linearly with d, q, and k.

Our evaluations on large real datasets of high dimensions establish that ProMiSH scales linearly with the dataset size, the dataset dimension, the query size, and the result size. ProMiSH also yields practical query times on large datasets of high dimensions, and is therefore useful for answering real time queries.

3.9.4 Space Efficiency

We evaluated the space efficiency of ProMiSH by computing the memory footprint of its index. For ProMiSH-E and ProMiSH-A, we used the space cost formulation from Section 3.8. Here, we first describe the space cost of Virtual-bR* Tree in terms of the dataset and the index parameters. Then we give the ratio of the index space to the dataset space for all the three algorithms for varying dataset parameters. Let the space cost of a point's identifier, a dimension of a point, and a keyword be E bytes individually. Let $\mathcal{D}$ be a dataset having N d-dimensional points each of which is tagged with t keywords. Let U be the number of unique keywords in $\mathcal{D}$. The dataset has a space cost of $S(\mathcal{D})$=$((d+t)\times\ N\ \times\ E)$ bytes.

The index structure of Virtual bR*-Tree comprises of a R*-Tree, an inverted index, and a query specific bR*-Tree. Let the number of children per node in R*-Tree be x. Let the total number of nodes in R*-Tree be N_R. The space cost of R*-Tree is $((2 \times d + x) \times E \times N_R)$ bytes. The inverted index stores a point's identifier and its path from the root node in R*-Tree. Therefore, the space cost of the inverted index is $((log_x N + 1) \times t \times E \times N)$ bytes. For a query of size q, the space cost of bR*-Tree is $((2 \times d \times E + 2 \times d \times E \times q + x \times E + U/8) \times N_R)$ bytes.

We investigated the ratio of the index space to the dataset space for all the three algorithms using their space cost formulation. We used the following values

of the parameters: E=4 bytes, m=2, M=10,000, L=5, x=100, q=5, and t=1. We show the ratios for varying values of d, N, and U in Table 3.4 for ProMiSH-E, Table 3.5 for ProMiSH-A, and Table 3.6 for Virtual bR*-Tree. For datasets of low dimensions, e.g., d=8, we observe that ProMiSH-E has the highest ratios, whereas ProMiSH-A has the lowest ratios. For datasets of high dimensions, e.g., d=128, we observe that ProMiSH-E and Virtual bR*-Tree have comparable ratios, whereas ProMiSH-A again has the lowest ratios.

The index space of ProMiSH is independent of the dimension of the dataset, whereas the dataset space grows linearly with the dimension. Therefore, the space ratio of ProMiSH decreases with dimension. The index space of Virtual bR*-Tree also grows with dimension. Therefore, ProMiSH has a lower space ratio than Virtual bR*-Tree for a high dimensional dataset.

	N=10 million		N=100 million	
d	U=100	U=1,000	U=100	U=1,000
8	2.8	3.0	2.8	2.8
16	1.4	1.6	1.5	1.5
32	0.7	0.8	0.8	0.8
64	0.4	0.4	0.4	0.4
128	0.2	0.2	0.2	0.2

Table 3.4: Ratio of the index space to the dataset space for ProMiSH-E for varying N, d, and U.

	N=10 million		N=100 million	
d	U=100	U=1,000	U=100	U=1,000
8	0.7	0.9	0.7	0.7
16	0.4	0.5	0.4	0.4
32	0.2	0.2	0.2	0.2
64	0.09	0.1	0.09	0.09
128	0.05	0.06	0.05	0.05

Table 3.5: Ratio of the index space to the dataset space for ProMiSH-A for varying N, d, and U.

	N=10 million		N=100 million	
d	U=100	U=1,000	U=100	U=1,000
8	0.9	0.9	0.9	0.9
16	0.5	0.5	0.6	0.6
32	0.3	0.3	0.4	0.4
64	0.2	0.2	0.3	0.3
128	0.2	0.2	0.2	0.2

Table 3.6: Ratio of the index space to the dataset space for Virtual bR*-Tree for varying N, d, and U.

3.10 Conclusions

In this chapter, we proposed solutions for the problem of top-k nearest keyword set search in multi-dimensional datasets. We developed an exact (ProMiSH-E) and an approximate (ProMiSH-A) methods using hashtables and inverted indices. We also proposed an efficient solution to find results from a subset of data points. Our empirical results show that ProMiSH is faster than state-of-the-art tree-based technique, having performance improvements of multiple orders of magnitude. These performance gains are further emphasized as dataset size and dimension are

increased, as well as for large query sizes. ProMiSH-A has the fastest query time among all the compared algorithms. We empirically observed a linear scalability of ProMiSH with the dataset size, the dataset dimension, the query size, and the result size. We also observed that ProMiSH yield practical query times on large datasets of high dimensions for queries of large sizes.

Chapter 4

Querying Spatial Patterns

Spatial data are common in many scientific and commercial domains such as geographical information systems and gene/protein expression profiles. Querying for distribution patterns on such data can discover underlying spatial relationships and suggest avenues for further scientific exploration. In this chapter, we study querying spatial patterns by example. Given a spatial pattern, the task here is to retrieve similar patterns from the dataset.

Supporting spatial pattern retrieval requires not only the formulation of an appropriate scoring function for defining relevant connected subregions, but also the design of new access methods that can scale to large databases. In this chapter, we propose a solution to this problem of querying significant subregions on spatial data provided as raster images. We design a scoring scheme to measure the similarity of subregions. All the raster images are tiled and each alignment of the query and a database image produces a tile score matrix. We show that

the problem of finding the best connected subregion from this matrix is NP-hard and develop a dynamic programming heuristic. With this heuristic, we develop two index-based scalable search strategies, TARS and SPARS, to query patterns in large data repositories. Experimental results on real image datasets show that TARS offers an 87% improvement for small queries, and SPARS a 52% improvement for large queries in running time, as compared to linear search. Qualitative tests on real datasets achieve precision of more than 80%.

4.1 Motivation

Spatial data arise in various domains such as geographical information systems, biology, environmental management, and IC fabrication. Often, the distribution of a spatial attribute of interest (e.g., population density, contamination rate, vegetation growth, protein expression, etc.) is captured using a raster image [96, 112, 108]. Such an example is shown in Figure 4.1 which displays the population density map of Afghanistan[1]. The color of each pixel is associated with a particular value of the population density. In biological and medical images, pixel intensity represents the distribution of tissues, gene, or proteins. Figure 4.2 shows the fluorescent microscopy image of a cross section of a feline retina [43]. The intensity of a pixel reveals the distribution of peanut-agglutinin, a lectin found in the retina.

[1] http://sedac.ciesin.columbia.edu/gpw/

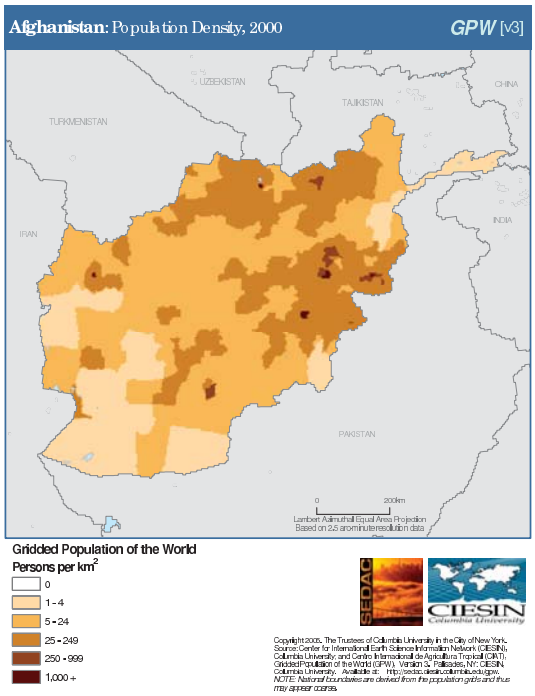


Figure 4.1: Population density map of Afghanistan.

Ever since John Snow's analysis of cholera outbreaks that resulted in finding a contaminated water pump in London in 1854 [125], the analysis of spatial data distributions has been a popular avenue of scientific inquiry. Revelation of similarity in demographic patterns helps us correlate and understand various geographic factors affecting population growth. Similarity in vegetation pattern discovered by querying aerial images can help relate climate cycle and land formation at various places on Earth. In retinal images, the similarity in spatial patterns may offer new insights into biological processes.

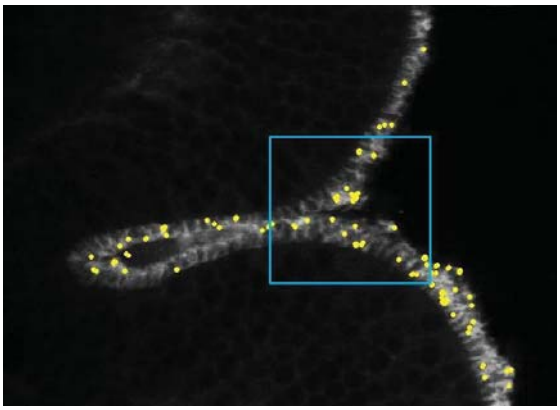


Figure 4.2: Example of a biologically interesting spatial pattern (best viewed in color). The marked pattern highlights a fold of the retinal tissue labeled with peanut-agglutinin conjugated to a fluorescent probe. Yellow dots are the point of interests detected using affine covariant region technique [93] of computing local descriptor.

In this chapter, we propose to search for a specified pattern in a database of spatial distributions represented as raster images. A query pattern can be described either by specifying a local distribution or by marking a rectangular region of interest on a given image as shown in Figure 4.2. The database may consist of population density maps, biological images, remote sensing images, or raster images of any other domain. The task is to find sub-regions of images in the database that are similar to the specified query pattern and are meaningful.

Our problem statement is close to sub-image search for natural images [106]. The methods developed in this domain use local descriptors [94] that are computed around few key points of interest as shown by yellow circles in Figure 4.2. These descriptors are obtained from a small number of neighborhood pixels around de-

tected points of interest and are designed to provide robustness for photometric, scale, viewpoint and affine changes for natural image matching. State-of-the-art SIFT descriptors [86] are histograms of the gradients of the sampled points around the key point. Sampled points are divided into 4×4 grids. Gradients of the sampled points in each grid are summarized using 8 bin orientation histograms. Histograms of all the grids are concatenated to yield a SIFT descriptor of size 128 for the key point. Local descriptors till now have had only limited success (around 66% precision [106]) because of the difficulty of coping with all the image variations. Raster images do not have challenges of photometric, viewpoint and affine changes. For spatial distributions in raster images, the resolution is known, and this permits easy normalization. Points of interest in natural images are computed using pixel intensity gradient and, therefore, may be absent in a large portion of an image as seen in Figure 4.2 making it impossible to carry out sub-image search for those regions. These points are also less than sufficient to summarize all the useful information in an image. Instead of just focusing on key points as for natural images, a solution to the proposed problem needs to capture the information over the entire useful part of an image (foreground), e.g., summarize each pixel representing population density in Figure 4.1 using a histogram.

Our method tiles the query image and database images into atomic units. Then, a domain-based scoring function is used to score an alignment of two atomic

units. Finally, the score is aggregated over a connected region to find the best match. The idea of a match is illustrated in Figure 4.3. A query is aligned with each image in the database under all possible translations. Each alignment generates a matrix of scores, both positive and negative, between corresponding tiles. Positive scores denote foreground matches while a negative score means that a background tile of the query is matched to a database tile. A connected subregion over the matrix identifies a meaningful matching subregion. Scores over all possible connected subregions can be used to define answers to range and nearest-neighbor queries. The generality of the solution and the identification of best connected subregions are the unique aspects of our design.

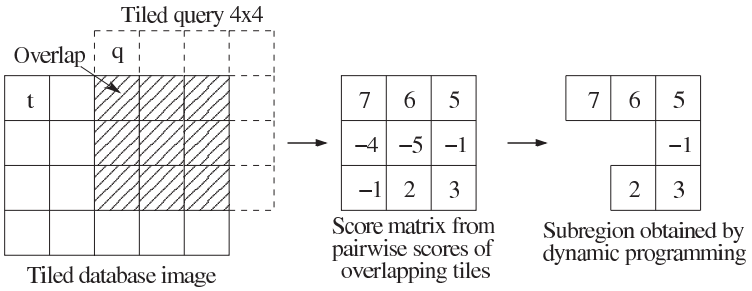


Figure 4.3: A 4×4 query is overlapped with a database map. For each tile in the 3×3 overlapped region, a score for the match is computed. Dynamic programming is run on the score matrix to obtain the maximal scoring connected subregion.

Once we adopt the score and subregion based idea for retrieval of high-quality answers, the next challenge is one of scalability. How to identify the best subregions over millions of alignments? Clearly, a region-by-region search design will

not work. So, how to develop access methods and index structures that can find the best subregions without examining all of them? Our solution to the scalability problem is two-fold: (i) development of an index structure that works with our definition of score, and (ii) design of two new algorithms that use the index structure to find the best subregions in an efficient manner.

The idea of finding the best connected subregion in a matrix that maximizes the sum of piecemeal scores is itself of theoretical and practical interest. We show that this problem is NP-hard. This necessitates appropriate heuristics that examine not all but a subset of connected subregions. We develop a dynamic programming based solution and characterize the class of subregions that is examined by this heuristic. Our access methods are unaffected by how the best connected subregions in an alignment are identified; they work correctly with any such heuristic.

In a nutshell, our contributions in this chapter are as follows:

- We develop a score-based framework for identifying the best connected subregions for a given query region with tile-based decomposition (Section 4.3).
- We develop index structure based access methods to query the best matching subregions efficiently. The first method, TARS, is instance optimal but traverses the index multiple times and, therefore, performs better for small

queries. The second method, SPARS, makes a single pass through the index and is suited for large queries (Section 4.4).

- We study the computational complexity of finding the highest scoring subregion. We show that this problem is NP-hard by reduction from the Thumbnail Rectilinear Steiner Tree problem [45]. We develop an efficient dynamic programming heuristic and characterize the class of subregions explored by this heuristic (Section 4.3).

- We empirically show scalability (Section 4.5) and quality (Section 4.5.4) of our methods on two real datasets.

4.2 Related Work

Query by example for images, called Content-Based Image Retrieval (CBIR), has been extensively studied. Region-Based Image Retrieval (RBIR) systems extend CBIR by making the search sensitive to different regions of an image. A survey on the recent methods of CBIR and RBIR can be found in [32]. Most of the RBIR systems use automatic or manual region segmentation in order to characterize regions and then compute a one-to-one or many-to-one mapping to match query regions to those in the database [7, 21, 98, 126, 127]. Weakness of the segmentation based methods lies is their incapability to handle region queries that

partially extend across various segments or regions of an image. Malki et al. [88] avoided segmentation by using a multi-resolution quadtree [41] to organize images. Their method had no constraint of connected pattern and had equal weight for foreground and background. Sub-image retrieval using local descriptors [94] has been addressed recently for natural images [70, 106] but cannot be extended for querying patterns in raster images as discussed in Section 4.1.

Tiling is the most common way of storing raster data [124] in spatial DBMS. Image tiling at varying scales was used by Svetlana et al. [77] for recognizing natural scene categories using full image matching. Tiles were also used by [14] to partition images into clusters in the color space.

Methods for querying similar images based on full image matching has also been developed for aerial [89, 115] and biomedical [105, 35] images separately. Baumann et al. [8] proposed a web-enabled service over a multidimensional DBMS, used as storage, for interactive navigation and SQL based querying on raster images. Their system does not support pattern querying. Vinhas et al. [124] also proposed DBMS system for handling raster image in spatial databases. OLAP techniques were exploited by [50] to speed up aggregate query processing in raster image databases. New geo-raster operations with array algebra is proposed in [51]. Zhang et al. [138] developed index structure for spatio-temporal aggregation over streaming raster images for a given query region. They first split the images

into tiles and then computed aggregate for each tiles. Gertz et al. [47] proposed a data and query model by extending Image Algebra to formulate and answer queries over geospatial image data. Pajarola et al. [100] provides a compression technique for large raster images and designs methods to support spatial range queries directly over compressed images. Hadjieleftheriou et al. [53] address the problem of querying a user defined movement patterns in space and time from a large collection of spatiotemporal trajectories. Yankov et al. [132] develop best-match searching algorithm for two-dimensional shapes.

Our work is the first to support pattern querying on geographic maps like demography, pollution, etc. The technique is generic enough to be extended to raster images of other domains like biology, medicine, etc. It differs from image retrieval techniques by supporting pattern querying without image segmentation into regions or objects, developing score based similarity measure, finding the best connected match, and discriminating between foreground and background to discover meaningful patterns.

4.3 Sub-Region Similarity

In this section, we discuss feature extraction from images, define similarity measure between a pair of image tiles, and then extend the idea of similarity

between tiles to regions. Then, we show that the computation of the optimal score between two sub-regions is NP-hard. Finally, we develop a dynamic programming heuristic to compute a good alignment.

Each raster image in the database is split into tiles [124]. All the pixel values in a tile is summarized as a histogram. Dimension of the histogram equals the number of discrete levels of pixel values which is later reduced by a dimensionality reduction technique (PCA [104]) for efficiency. Finally, our database DB consists of the feature vectors of these tiles. We also tile the query image Q and obtain feature vectors similar to a database image tile. We search for similar regions for a given query image in a feature vector space. We use L_1 norm as the measure of distance between a pair of histograms. Raster images can be of varying resolution or scale. In this chapter, we assume that a given database consists of raster images of the same scale. If the images are of varying resolution, they can be preprocessed and normalized to the same resolution as scale is known.

4.3.1 Scoring Function

We measure similarity between a pair of tiles using a scoring function. Our scoring function is a monotonically decreasing function of the distance between the feature vectors of a query tile q and an image tile t. Scoring function is defined

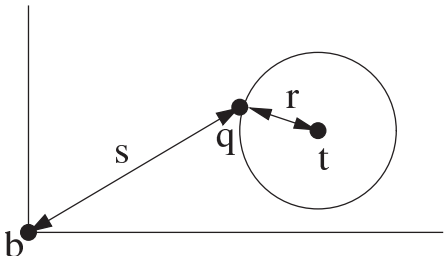


Figure 4.4: Scoring a query tile q against a database tile t. b denotes the perfect "background" tile. $score(q,t) = s - \lambda r - c$.

as

$$score(q,t) = f(q) - g(d(q,t)) - c \tag{4.1}$$

where f is a function based on domain knowledge, g is a monotonically increasing function, d is the distance between tiles and c is a constant. The score can be positive or negative. The intent of the scoring function is to discriminate between foreground (region of interest) and background. A query tile with little or no information forms the background and, therefore, should get a negative or low score no matter how good the match is. A tile with more pattern information is a part of *region of interest* (ROI) and should get a high score when matched with a similar database image tile. The function $f(q)$ measures whether the tile q is in a ROI.

4.3.2 Instance of Scoring Function

The scoring function template described in Section 4.3.1 is broadly applicable to a number of scoring functions. Here, we give a specific instance based on the idea of *log-odds*. For this model, we assume that the tile space consists of two distributions. The first distribution is that of foreground tiles from database which we call the *true distribution*. We model this as an exponential distribution[2]. For a database tile t and a query tile q (Figure 4.4), if $r = d(q,t) =$ the distance between the feature values of q and t, then we can characterize this distribution as

$$P(q|true\ distribution) = \lambda_1 e^{-\lambda_1 r}. \tag{4.2}$$

The second distribution is that of background tiles in the database, which we call the *background distribution*. We postulate a perfect background tile b (Figure 4.4) and an exponential background distribution centered at b. If $s = d(q,b)$ then

$$P(q|background\ distribution) = \lambda_2 e^{-\lambda_2 s}. \tag{4.3}$$

The score of a query tile q matching a database tile t is given by the *log-odds ratio*:

$$\begin{aligned} score(q,t) &= ln\frac{P(q|true\ distribution)}{P(q|background\ distribution)} \\ &= \lambda_2 s - \lambda_1 r + ln(\lambda_1/\lambda_2) \end{aligned} \tag{4.4}$$

[2]The reasons for choosing this distribution are three-fold: first, data observation, second, its simplicity, and third, its utility in capturing small variations over related images.

Since scoring is only used to discriminate between foreground and background matches and the actual value is not important, the scores can be conveniently translated and scaled with constants. Denoting λ_1/λ_2 by a constant λ, then scaling by λ_2, and finally translating the score gives

$$score(q,t) = s - \lambda r - c \tag{4.5}$$

$$= d(q,b) - \lambda d(q,t) - c \tag{4.6}$$

where λ and c are independent constants. Comparing Eq. (4.6) with Eq. (4.1), we see that $f(q) = d(q,b)$ and $g(d(q,t)) = \lambda\, d(q,t)$.

We name the above scoring function *Discriminator Function*. We can make the following observations from Eq. (4.6): (i) When distance to a database tile is kept invariant, a query tile with less background has a higher score, (ii) For a particular query tile, a more distant database tile has a lower score. We show the advantage of this scoring function over simple distance measures in Section 4.3.5.

4.3.3 Score of an Overlapping Region

Once we have a model to measure the similarity between a pair of tiles, we next consider how to measure similarity between two regions. The alignment or the overlap of a query image Q with a database image produces a score matrix of pairwise aligned tiles, as depicted in Figure 4.3. The score of the alignment

is defined as the score of a *connected subregion* that has the *maximal* possible *cumulative* score. We are interested in the alignment of a single pattern and, hence, the justification for finding a single connected region. The maximal scoring subregion may include negative scores and may not be rectangular in shape, as shown in Figure 4.3.

The best match in a database of images is found by considering all possible alignments, i.e., translations of the query image over each database image. This is illustrated in Figure 4.5 where three alignments are shown.

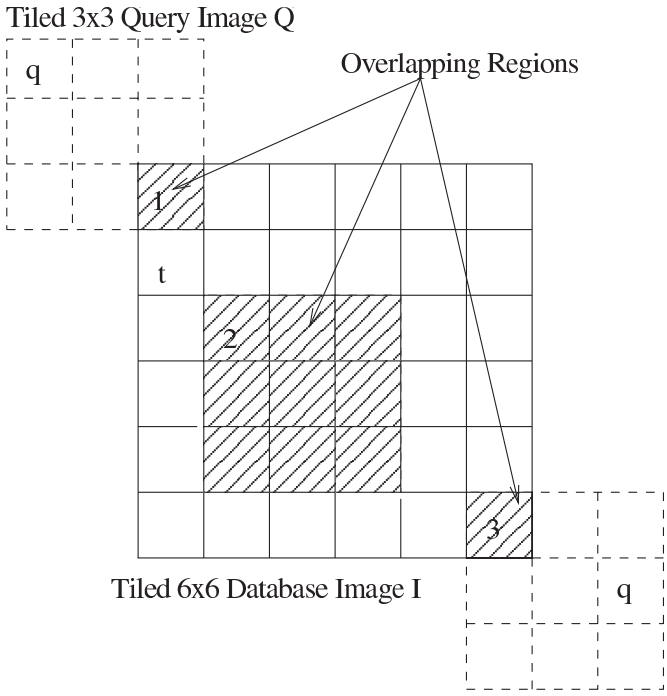


Figure 4.5: Overlapping regions found by translation of a query image Q on a database image I at 3 alignments.

4.3.4 NP-Completeness Proof

We next prove that the problem of finding the maximal scoring subregion in a score matrix is NP-hard. We prove this by showing that the corresponding decision problem is NP-complete. We first define the "graph" analog of the matrix problem as follows: *Given a graph representation $G = (V, E)$ of a matrix, with weight $w(v)$ on each vertex $v \in V$ corresponding to the entry in the matrix, is there a connected subgraph of weight $\geq W$?* We denote this problem by Maximal Weighted Connected Subgraph or MWCS.

Theorem 1. Maximal Weighted Connected Subgraph (MWCS) *is NP-complete, for a matrix graph of degree at most 4.*

Proof. MWCS is in NP since the weight of a connected subgraph can be computed in polynomial time.

For reduction, we use the Rectilinear Steiner Tree (RST) problem that is known to be NP-complete [46]. The RST problem asks: Given a set of n terminal points that are embedded in an integer grid in a plane, is there a spanning tree of total length at most l such that the vertices of the spanning tree are the input points of the set and the grid points, where the length of an edge is the L_1 distance between the corresponding vertices?

There is a special case of the RST problem known as the THUMBNAIL RECTILINEAR STEINER TREE (TRST) problem [45]. The TRST problem restricts the terminal points to an $m \times m$ grid. The TRST problem remains NP-complete even when m is bounded by a polynomial of n [46].

Given an instance of the TRST, we construct an instance of the MWCS as follows: We first find the bounding box of the points of the TRST, i.e., the $m \times m$ grid. Then, we replace each terminal point by a vertex of weight $w \gg l$. At each grid point that is not already occupied by the n terminal points, we place a vertex with weight 0. Between a pair of consecutive vertices on the same grid line (e.g., on the half-grid positions), we place a vertex with weight -1. Each vertex is connected to only to its horizontal and vertical neighbors, thus producing a matrix graph. Figure 4.6 shows an example of the construction. The original points are shown by double circles. The construction takes time polynomial in m, and hence polynomial in n, and the graph G thus constructed is planar with degree at most 4.

We claim that the original TRST on n points has a rectilinear Steiner tree of length $\leq l$ if and only if the MWCS graph has a connected subgraph of weight $\geq W = n.w - l$.

Only if: Assume that there is a Steiner tree of length at most l. By definition, it spans all the terminal points and is connected. Note that for a length l path

between two points, there are exactly l vertices of weight -1. The vertices corresponding to the n terminal points have a weight of w each. Therefore, the weight of this tree is at least $n.w - l$. Figure 4.6 shows such a Steiner tree in solid lines.

If: Any connected subgraph of weight at least $n.w - l$ in G must include all the n vertices of weight w and at most l vertices of weight -1. There is no way to connect two vertices of weight ≥ 0 without passing through a vertex of weight -1. Therefore, the length of this path is at most l, since otherwise, the connected subgraph would have included more than l vertices of weight -1. Also, if the subgraph has the maximal weight, it is a tree, since, if it is not, at least one pair of vertices has more than one path between them. Removing that path increases the weight of the tree by the absolute weight of the negatively-weighted vertices in the path. Therefore, this subgraph defines a Steiner tree for the original n points. An example of such a subgraph is shown in Figure 4.6 in solid lines. □

4.3.5 Dynamic Programming Heuristic

Now, we design a dynamic programming (DP) heuristic as an alternative to examining all possible subregions for finding the maximal score. Assume that the score in cell $C(i, j)$ of the score matrix is denoted by $s(i, j)$. The DP starts from one of the corner cells of the score matrix. For discussion purposes, assume that it starts at cell $C(0, 0)$ in Figure 4.7. Next, it proceeds by first moving towards the

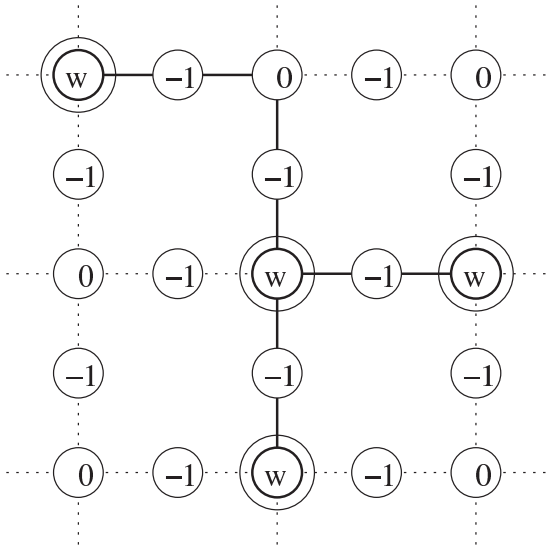


Figure 4.6: Construction from Thumbnail Rectilinear Steiner Tree instance to Maximal Weighted Connected Subgraph (MWCS) instance. The double lined vertices are the original terminal points. The solid lines represent the optimal solution of both the problems.

right ($\rightarrow$) and calculates a subregion corresponding to each cell in 0^{th} row. Then it goes to 1^{st} row 0^{th} cell $C(1,0)$ by moving in the top ($\uparrow$) direction in the score matrix (akin to a row-scan order). DP completes this iteration when it reaches the top-most and right-most cell in the matrix. In Figure 4.7, this cell is $C(2,2)$.

Suppose, $R(i,j)$ is a maximal scoring sub-region that has its top-right corner at the cell $C(i,j)$. Also, suppose $s(i,j)$ denotes the score of $C(i,j)$ and $S(i,j)$ denotes the maximal score for the subregion $R(i,j)$. DP examines 4 possibilities to find the maximal score of $R(i,j)$: (i) the score of the cell itself, (ii) the score of the cell plus the maximal score for the bottom subregion, (iii) the score of the

cell plus the maximal score for the left subregion, and (iv) the score of the cell plus the maximal scores for the bottom and the left subregions. Since the bottom and the left subregions can intersect, the score of the intersecting region should be subtracted from the cumulative scores of the two subregions so that it is not counted twice.

The DP algorithm computes the following recurrence relation to find all the subregions in the score matrix and their scores:

$$S(i,j) = \max \begin{cases} s(i,j) \\ s(i,j) + S(i,j-1) \\ s(i,j) + S(i-1,j) \\ s(i,j) + S(i,j-1) + S(i-1,j) \\ \qquad -S\big(R(i,j-1) \cap R(i-1,j)\big) \end{cases} \tag{4.7}$$

The corresponding subregions maintained for the 4 cases are, respectively:

$$R(i,j) = \begin{cases} C(i,j) \\ C(i,j) \cup R(i,j-1) \\ C(i,j) \cup R(i-1,j) \\ C(i,j) \cup R(i,j-1) \cup R(i-1,j) \end{cases} \tag{4.8}$$

To improve the overall score, DP executes the above logic starting from all the 4 corner cells with the following combinations of moves: (i) Starting at bottom-left cell and moving in $\uparrow$ and $\rightarrow$ direction, (ii) Starting at bottom-right cell and

moving in $\uparrow$ and $\leftarrow$ direction, (iii) Starting at top-left cell and moving in $\downarrow$ and $\rightarrow$ direction, (iv) Starting at top-right cell and moving in $\downarrow$ and $\leftarrow$ direction. It returns the subregion having the maximum score of all these 4 possibilities. Such a subregion explored by DP on a score matrix is illustrated by Figure 4.3.

Running Time: For a score matrix of size $m \times n$, for each cell, the DP computes the maximal score for the subregion ending at that cell. Calculating the scores for each cell requires finding an intersection of the largest scoring subregions on its bottom and left. This requires a running time of $O(mn)$ in the worst case. Thus, the total running time of the DP algorithm is $O(m^2n^2)$. For a particular score matrix, the DP needs to be run from all the 4 corners, which is constant. Thus, the worst case running time for the DP is *quadratic* in the size of the score matrix.

Class of Subregions Examined: The DP algorithm does not (and cannot!) investigate all the possible connected subregions; it chooses the maximal scoring connected subregion from only a certain class of shapes. Next, we analyze the class of such shapes. Consider only right ($\rightarrow$) and top ($\uparrow$) moves starting at left-bottom corner. The maximal scoring subregion $R(i, j)$ for cell $C(i, j)$ will include another cell $C(i', j')$ only if $C(i', j')$ is included in either $R(i, j-1)$ or $R(i-1, j)$. Similarly, the subregions $R(i, j-1)$ and $R(i-1, j)$ contain only those cells that

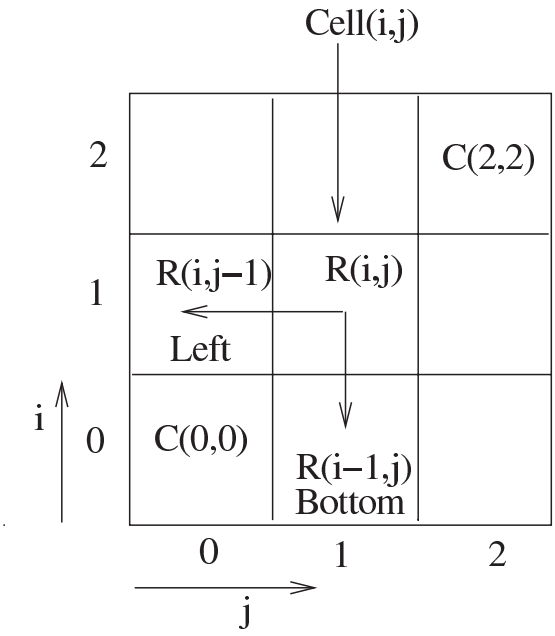


Figure 4.7: DP forms sub-region $R(i, j)$ by looking at scores of $C(i, j)$, $R(i-1, j)$ and $R(i, j-1)$.

3,1 (−1)	3,2 (−1)	3,3 (40)	3,4 (−90)
2,1 (−1)	2,2 (10)	2,3 (1)	2,4 (35)
1,1 (−1)	1,2 (−1)	1,3 (10)	1,4 (−1)

Figure 4.8: Example of a shape not captured by DP. The scores are shown in brackets. The optimal solution consists of the cells $(3, 3)$, $(2, 2)$, $(2, 3)$, $(2, 4)$ and $(1, 3)$ having scores 40, 10, 1, 35 and 10 respectively.

are towards the left and bottom of them. Therefore, by induction, if cell $C(i', j')$ is included in $R(i, j)$, then $C(i', j')$ must be at the left and bottom of $C(i, j)$. No cell which is towards the right or top of $C(i, j)$ will be included in $R(i, j)$. As an example, consider the cell $(2, 4)$ in Figure 4.8. It can consider only the cells (i, j) where $i \leq 2$ and $j \leq 4$, i.e., all cells to its left and bottom. It cannot consider any other cell (e.g., cell $(3, 3)$ in the figure). The DP ends at the top-right corner. However, the maximal scoring subregion may end at any cell, and not necessarily at the top-right corner cell. Thus, for example, if the maximal scoring sub-region ends at $(3, 3)$, then the nature of DP forbids it to consider cells $(1, 4)$, $(2, 4)$ and $(3, 4)$ (Figure 4.8). Hence, even though the optimal solution for the example in Figure 4.8 consists of all the five shaded cells, this DP will find only the region consisting of cells $(1, 3)$, $(2, 2)$, $(2, 3)$ and $(3, 3)$ as the answer. The shaded region given in Figure 4.8 cannot be obtained by DP starting from any corner.

Since the DP is run from the four corners in four sets of moves, the subregions captured are of four types: containing cells (i) towards bottom and left, (ii) towards bottom and right, (iii) towards top and left, and (iv) towards top and right.

Formally, the shapes for the class of such subregions can be characterized in the following way. For a particular shape P, a cell $C(i, j)$ *sinks* another cell $C(i', j')$, denoted by $C(i, j) \triangleleft C(i', j')$, if $C(i, j)$ can be reached from $C(i', j')$ in P

by taking one of the four combinations of moves described earlier. For example, in Figure 4.8, cell $(3,3)$ sinks cell $(2,3)$ for right and top moves since it can be reached from $(2,3)$ by this move combination. For the same move combination, it does not sink cell $(2,4)$ as it cannot reached from $(2,4)$ using only right and top moves. A cell $C(i,j)$ *sinks* a shape P, denoted by $C(i,j) \triangleleft P$, if and only if for all cells $C(i',j')$ belonging to P, $C(i,j)$ sinks $C(i',j')$, i.e.,

$$C(i,j) \triangleleft P \iff \forall C(i',j') \in P, \quad C(i,j) \triangleleft C(i',j') \tag{4.9}$$

A particular shape P can be captured by DP if and only if there exists a cell $C(i,j) \in P$ that sinks P. Combining all the 4 sets of moves as mentioned earlier characterizes the entire set of shapes captured by DP. Examples of shapes captured are: $\sqcap, \square$, etc. Shapes that cannot be captured include $+, \times$. We also present few examples of the count of the shapes captured by DP for varying number of grids. DP captures all the possible 14 shapes for a 2×4 grid. DP captures 31 of the 38 possible shapes for a 3×3 grid.

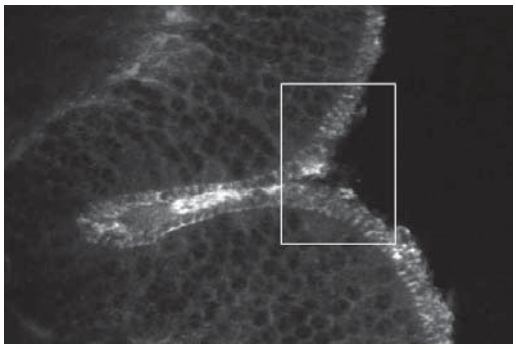
Advantages of Score Based Similarity: We performed quality experiments to compare our score based similarity measure with distance based measures. We used the *Discriminator* scoring function, described in Section 4.3.2, to measure the similarity between a pair of tiles. We compared the first result retrieved by our similarity measure to the sum of L_1 distance measured between correspond-

ing histogram of tiles of the overlapping region. One such result over biological images is shown in Figure 4.9. We can see that a simple distance measure fails to discriminate between foreground and background and hence generates more false matches. Our similarity measure maximizes the score of the best matching subregion and performs better.

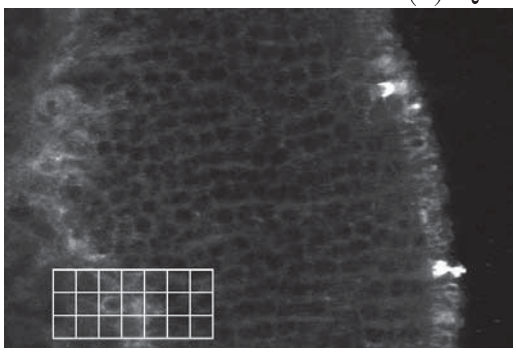
4.4 Query Algorithms

In this section, we discuss linear search and two new query algorithms to find the top-k similar regions from a database. These strategies are general enough to work with other scoring schemes and heuristics. The first algorithm is a naïve linear search through the database. The other two algorithms use a multi-dimensional index structure to prune the search space to achieve efficiency and scalability. In the ensuing discussion, the size of a query image Q and a database image I is defined in terms of the number of constituent tiles. We take the size of Q to be n.

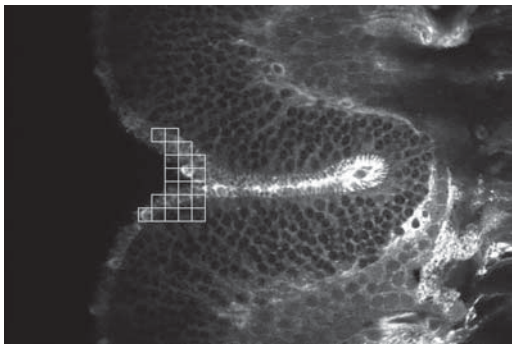
The *Linear Search* algorithm searches through all the possible overlaps to find the top-k matching regions. It translates the query image over all the database images and computes a score for each of them. It maintains a priority queue of the results to find the k highest scoring regions. Since the number of possible



(a) Query pattern



(b) Result with distance-based scheme on entire region



(c) Result with scoring-based scheme and sub-region finding

Figure 4.9: (a) Example of a biologically interesting pattern. The marked pattern highlights a fold of the retinal tissue labeled with peanut-agglutinin conjugated to a fluorescent probe. (b) Retrieved result when distance-based matching on entire region is used. (c) Retrieved result when score-based matching on subregions is used.

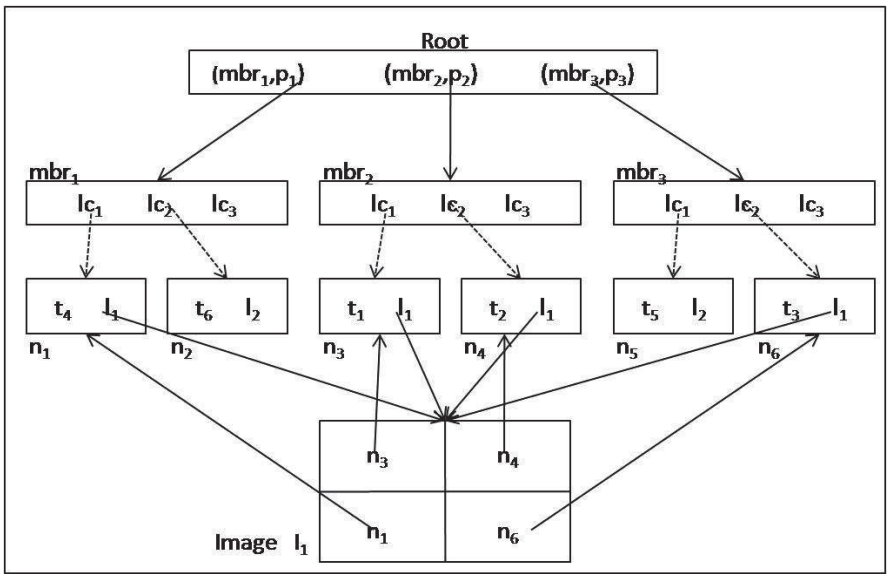


Figure 4.10: Index structure. Image I_1 maintains pointers to leaf nodes of its tiles. Leaf nodes maintain pointer to I_1.

overlaps increases linearly with increase in image and database size, this method does not scale and is impractical for large queries and database sizes.

To make the search scalable and efficient, we next propose two algorithms TARS and SPARS. Both algorithms use an index on the feature vectors of the image tiles to query nearest neighbors for a given tile. We can use any R-tree [52] (data-partitioning) like index structure for this. We choose bulk loadable STR-Tree [80] as the index because of its simplicity and availability. Each leaf node of the index is an entry of the form *lmbr(t, I)* where t is the feature vector of an image tile and I is a pointer to its parent image. Each non-leaf node has a list containing an entry per child of type (MBR, child-pointer) where MBR is

Algorithm 7 TARS

In: T: tree root, Q: list of query tiles
In: k: number of top results
Out: RQ: priority queue of top-k matches

```
 1: RQ ← [(−, −∞)]
 2: T ← +∞
 3: BS ← φ : bit-vector for explored overlap region
 4: while T > GetHead(RQ).s do
 5:    for all q_i ∈ Q do
 6:       nnTile[i] ← GetNextNN(q_i)
 7:    end for
 8:    for all q_i ∈ Q do
 9:       org ← FindOverlapRegion(nnTile[i],q_i)
10:       if org not flagged in BS then
11:          sm ← GetScoreMatrix(Q,org)
12:          e(rg,s) ← DP(sm)
13:          flag org in BS
14:          Insert(RQ, e(rg,s))
15:       end if
16:       sm ← score matrix of overlapping Q with nnTile
17:       T ← score of DP(sm)
18:    end for
19: end while
```

the minimum bounding box and child-pointer is the pointer to the child node respectively. Each database image I is a two-dimensional array of pointers to the *lmbrs* containing its tiles as shown in Figure 4.10. This structure allows for full access from a tile to its parent image and vice versa. We can easily find the row and column position of a tile from the image array to find an overlap.

4.4.1 TARS (Threshold Algorithm for Regionbased Search)

The algorithm TARS formulates the region retrieval query as a top-k aggregate query. It views the given query image as a multi-component object where each of its tile $q_i \in Q, \forall i = 1, \ldots, n$ constitutes its independent components. Similarly, it views DB as a list of multi-component objects. Each DB object is a set of connected tiles from a DB image. It uses sub-region similarity function as aggregate function. Query image (query object) is overlapped with a connected set of tiles from an image (DB object), a score matrix is obtained by computing pairwise tile similarity, and then maximum similarity score is computed using DP on the score matrix. The goal in this setting now is to retrieve the top-k maximum scoring DB objects.

Algorithm TARS adopts a strategy similar to the Threshold Algorithm (TA) [40] to solve this aggregate query. For each q_i, TARS views that all the database tiles are ranked in a decreasing order of their scores with q_i. Overlapping regions are determined by the tiles from the ranked lists. Table 4.1 shows a sorted view of the database for a query image consisting of two tiles. Maximum similarity score computed using DP is monotonic. For two score matrices a and b obtained by overlapping a query object with two different DB objects, if a has tile-wise larger scores than b, then the score of DP on a will be larger than b. This monotonic property is used to terminate TARS.

q_1	q_2
(t_3,I_2,10)	(t_1,I_1,9)
(t_2,I_1,8)	(t_2,I_1,7)
(t_1,I_1,2)	(t_3,I_2,6)

Table 4.1: Sorted access of tiles for a given query (q_1, q_2) in TARS.

TARS performs incremental nearest-neighbor searches for each q_i on the index structure to get sorted access to the database tiles. It starts by accessing the first nearest neighbor t_{i_1} for each q_i (steps 5-7 of Algorithm 7). Then, for each q_i, it finds the overlapping region of Q with a database image I ($t_{i_1} \in I$) such that q_i aligns with t_{i_1} in the overlap. Figure 4.11 shows how Q having 4 tiles overlaps with an image I also having 4 tiles such that q_1 aligns with t_1.

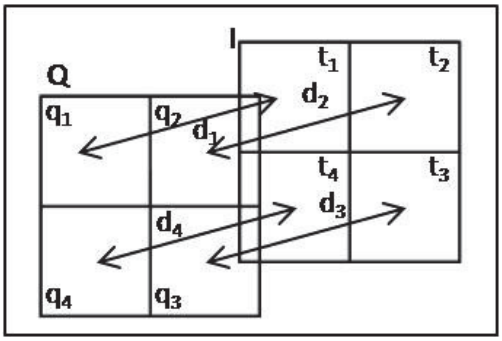


Figure 4.11: Overlap of query image Q with database image I such that q_1 aligns with t_1.

The algorithm then uses DP to find the maximum scoring subregion rg and its score s for each such overlap org (steps 11-12). It inserts all the results $e(rg,s)$ in a priority queue RQ of size at most k. The entries in RQ are sorted based on their

scores. Once RQ has k regions, a result is inserted only if its score is more than the k^{th} current region with the least score. In order to prevent multiple processing of the same database region, TARS flags the explored overlapping regions in a bit-vector BS.

At the end of the first iteration, TARS builds a score matrix by aligning each q_i with t_{i_1}. The DP score on this score matrix is the *threshold* score T. The threshold score is an upper bound on the scores of all the regions that have not yet been explored; this is because all the tiles to be accessed in the next iteration by each q_i have scores lower or at best equal to the current t_i's and the DP score is monotonic. This threshold score is updated after every iteration of the algorithm. The algorithm proceeds to the next iteration only if T is greater than the least score in RQ. As TARS proceeds, T decreases and the algorithm terminates with optimal results.

The performance of algorithm TARS worsens with increase in query size. It is instance optimal but it traverses the index structure separately for each $q_i \in Q$ to access the database tiles in sorted order. The cost of this multiple nearest-neighbor traversal grows quickly with increasing query size. To avoid this scalability problem, we next propose a technique SPARS that finds the top-k regions by performing a single traversal through the index structure and has better performance than TARS for large queries.

4.4.2 SPARS (Single Pass Region-based Search)

Algorithm SPARS is a novel top-k aggregate query algorithm which makes a single traversal through the index tree. It finds the top scoring regions by performing a *best-first search* [56]. It maintains a priority queue BQ to find the next best node to process. When the algorithm encounters a leaf node, it explores an actual region in the database corresponding to its image tile. Similar to TARS, it maintains a priority queue RQ of the top-k regions.

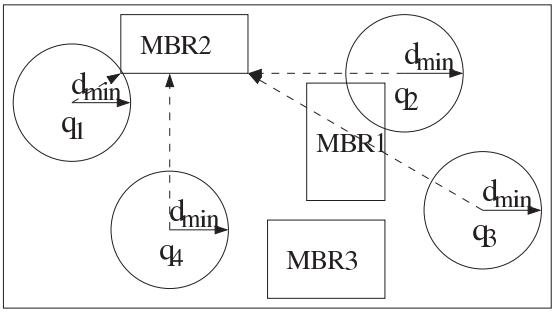


Figure 4.12: MBR and its nearest query tile. q_1 is nearest to MBR_2 with distance d_{min}.

The search for top-k regions starts at the root node of the index, which is the first entry in BQ with $+\infty$ score. The algorithm next processes each of its children. If the child is a non-leaf index node mbr, then it computes a score for it as follows (outlined in steps 8-12 of Algorithm 8). It determines the minimum distance between any query tile and the node $d_{min} = \min_i d(q_i, mbr)$, as shown in Figure 4.12. Then, it computes a score matrix by aligning each $q_i \in Q$ with

Algorithm 8 SPARS

In: T: tree root, Q: list of query tiles
In: k: number of top results
Out: RQ: priority queue of top-k matches

```
 1: n ← size(Q)
 2: RQ ← [(−, −∞)]
 3: BQ: queue of intermediate entities← [(T, +∞)]
 4: e(mbr,s) ← GetHead(BQ)
 5: while e.s ≥ GetHead(RQ).s do
 6:    if e is of type (mbr,s) then
 7:       for all child node cn in e.mbr do
 8:          if cn is mbr then
 9:             d_min ← GetMinDistance(Q,mbr)
10:             sm ← GetScoreMatrix(Q,[d_min])
11:             e(rg,s) ← DP(sm)
12:             Insert(BQ,e(mbr,s))
13:          else
14:             /*if cn is a lmbr*/
15:             q_j ← query tile nearest to lmbr
16:             e(rg,s) ← GetMaxSubRg(lmbr,q_j,Q)
17:             RQ.Insert(e(rg,s))
18:             for all q_i in Q and i ≠ j do
19:                d_min ← GetMinDistance(q_i,lmbr)
20:                sm ← GetScoreMatrix(Q,[d_min])
21:                e(lmbr,q_i,s) ← DP(sm)
22:                Insert(BQ,e(lmbr,q_i,s))
23:             end for
24:          end if
25:       end for
26:    else
27:       /*if e is of type (q_j,lmbr,s)*/
28:       e(rg,s) ← GetMaxSubRg(lmbr,q,Q)
29:       Insert(RQ,e(rg,s))
30:    end if
31:    e(mbr,s) ← GetHead(BQ)
32: end while
33: return RQ
```

virtual image tiles having a distance d_{min} from q_i as shown in Figure 4.13. The score s of the node is the score of the maximum scoring subregion found using DP

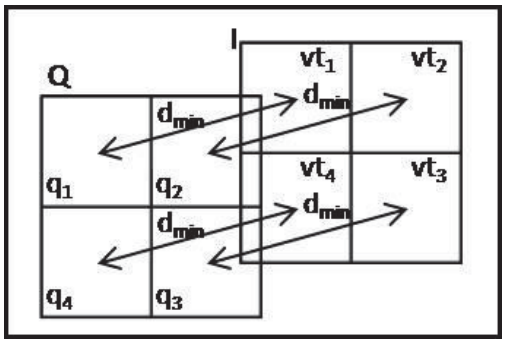


Figure 4.13: Overlap of query image Q with virtual tiles $(vt_1, vt_2, \ldots)$ at distance d_{min}.

on this score matrix. SPARS inserts the *mbr* along with the score s as an element *e(mbr,s)* in BQ. It inserts e into BQ only if RQ has less than k elements or s is greater than the minimum score in RQ.

If the child is a leaf node *lmbr*, then the algorithm finds the nearest query tile q_i to tile t of *lmbr* and computes the minimum distance $d_{min} = \min_i d(q_i, lmbr)$. It explores the actual image region for the (q_i,t) alignment using Algorithm 9 as illustrated in steps 15-17 of Algorithm 8. Algorithm *GetMaxSubRg* finds the overlap of query Q with image I pointed by *lmbr* by aligning q_i with t. Since the same overlapping region can be encountered later for a query and image tile pair, *GetMaxSubRg* maintains a bit-vector BS to flag the explored regions; this prevents multiple processing of the same database region. Algorithm *GetMaxSubRg* returns the maximum scoring subregion rg of the overlap and the corresponding score s

using DP. The result *e(rg,s)* is inserted into RQ only if RQ has less than k elements or s is greater than the minimum score in RQ.

After processing the alignment of q_i with t, we still need to process the other alignments corresponding to other query tiles and t. The SPARS algorithm delays exploring these alignments in order to save computation cost. It calculates a score s of the *lmbr* for each $q_j, j \neq i$ using the same method discussed for a *mbr* (outlined by the steps 18-23 of Algorithm 8). It finds the minimum distance d_j between q_j and tile t in *lmbr*. It overlaps the query image with a virtual image such that the distance between each aligned pair of tiles is d_j. DP is run on this score matrix to compute score s of the maximum scoring subregion for the overlap. SPARS inserts elements e(*lmbr*,q_j,s) in BQ for each q_j only if RQ has less than k elements or s is greater than the minimum score in RQ. The algorithm explores these regions during the access of the elements from BQ as outlined by steps 28-29 of Algorithm 8.

SPARS proceeds by accessing the current highest scoring element from BQ and terminates when the lowest score in RQ is greater than the highest score in BQ.

Pruning Strategy: The scoring function $s(q,t)$ is a monotonically decreasing function of $d(q,t)$, as discussed in Section 4.3.1. The aggregate score of an overlap

Algorithm 9 GetMaxSubRg (SPARS)

In: $lmbr$: leaf node, q_i: query tile, Q: query tiles list
Out: e: entity of maxsubregion and score
1: BS: bit-vector for explored overlap region
2: $org \leftarrow$ FindOverlapRegion($lmbr$,q_i)
3: **if** org not flagged in BS **then**
4: $dm \leftarrow$ GetDistanceMatrix(org)
5: $sm \leftarrow$ GetScoreMatrix(Q,dm)
6: e(rg,s) $\leftarrow$ DP(sm)
7: flag org in BS
8: **end if**
9: **return** e(rg,s)

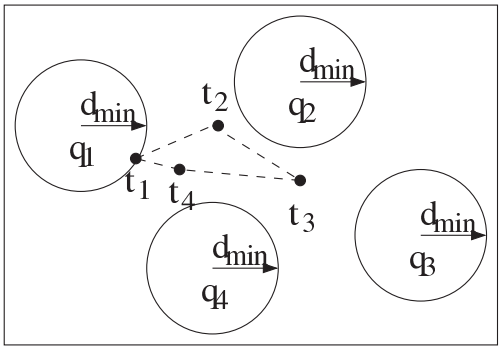


Figure 4.14: Tiles of the overlapping region for q_1 aligning with t_1 lie at distances greater than d_{min}.

is also monotonic with respect to individual scores of the score matrix of an overlap. With this monotonicity property, the following lemma holds. SPARS uses this lemma to prune the search space.

Lemma 4. *The score $S(Q, I)$ of the overlap of a query image Q with an image I with $d(q_i, t_i) = r,\ \forall i$, is an upper bound on the score $S(Q, I')$ of all possible overlaps of Q with image I' provided $d(q_i, t'_i) \geq r,\ \forall i$.*

From this lemma, we see that the score s of a node (mbr or $lmbr$) at a minimum distance d_{min} from the query tiles is an upper bound on the score of all nodes whose minimum distance is greater than d_{min}. We visualize such an example in Figure 4.12 where MBR_2 is at a distance of d_{min} but MBR_3 is at a greater (minimum) distance from all the query tiles. Therefore, score of MBR_3 will be less than MBR_2. The score s is also an upper bound on the score of an actual overlapping region if the distance between the corresponding tiles of Q and I have distance greater than d_{min}. We have such a scenario in Figure 4.14 in which tile t_1 finds q_1 as its nearest neighbor. All the tiles of the corresponding overlap as shown in Figure 4.11 lie at a distance greater than d_{min}. Therefore, the score of this overlap is less than the score s of a node. These facts justify that the score of an element in BQ is an upper bound on all the nodes and regions that have not been explored and are ranked lower in BQ. At any point during the search, SPARS has already explored a hypersphere of radius d_{min} centered at each query tile if the next candidate from BQ has a minimum distance d_{min} from all the query tiles.

SPARS processes the nodes in decreasing order of their scores. It explores all nodes having score greater than the least score in RQ since they are potential candidates to yield regions with higher score. It terminates the search once the

minimum score in RQ becomes more than the highest score in BQ. Thus, this pruning strategy ensures an optimal result for SPARS.

4.5 Experimental Studies

In this section, we first empirically analyze the performance and efficiency of our access methods. Then, we present detailed quality analysis of our new similarity measure with visual results. We used Java 5.2 as our implementation language. We performed experiments on a 3.2 GHz, 4 GB memory PC running Debian Linux 4.0.

4.5.1 Dataset Preparation

We used two large real image datasets belonging to raster image family to empirically analyze the efficiency and scalability of our algorithms. The first dataset consists of 112,045 gray-scale images of various tissues and layers of retina [43] from different experimental conditions. Multiple molecular probes such as lectins and antibodies were used to examine the localizations of specific protein expression in retinal cells and the expression patterns of these proteins in different layers of retina. The fluorescence tagged probes were imaged by immunohistochemistry using confocal microscopes. We used the magnification of these images to scale

them to a standard magnification using the CubicFilter from GraphicsMagick[3]. Our second dataset consists of 82,282 gray-scale aerial images from the Alexandria Digital Library[4]. These are satellite images and air photos of different regions of California. The size of the images in both datasets varied from 320×160 pixels to 640×480 pixels.

Reduced dimension	Energy retained	
	Retinal dataset	Aerial dataset
3	85.73%	81.21%
6	96.14%	93.38%
13	98.94%	97.55%

Table 4.2: Percentage energy remaining after PCA.

Retinal dataset		Aerial dataset	
Image count	Region count	Image count	Region count
112,045	10,004,850	82,282	10,625,200
56,241	5,000,050	37,037	5,000,000
33,762	3,000,000	21,744	3,000,000
11,112	1,000,100	5,560	1,000,000

Table 4.3: Database sizes of retinal and aerial images.

We split the images into non-overlapping tiles of size 32×32 pixels and compute feature vector for each tile. The feature vector of each tile is a histogram of its pixel values similar to CSD feature vector [90]. To enhance efficiency, we performed PCA [104] on these feature vectors to reduce the dimensionality. The number of

[3] http://www.graphicsmagick.org/
[4] http://www.alexandria.ucsb.edu/

principal components retained and the corresponding energy preserved is shown in Table 4.2. The index structure was built on this transformed data. We used the *Discriminator* scoring function, described in section 4.3.2, to measure the similarity between a pair of tiles. We discuss the choice of parameters for the scoring function later in Section 4.5.4.

The parameters that are crucial to the performance of the access methods are: (i) Query size, n (ii) Database size, N, and (iii) Dimensionality of the feature vector, dim. Query size n is defined as the number of constituent tiles in the query image. Database size N is defined as the number of images. The number of images and possible overlapping regions obtained by translation for varying N is described in Table 4.3 for both retinal and aerial datasets. For each experiment, we used 100 randomly picked queries from the dataset. All the reported time measurements are averaged over these 100 queries for top-10 results.

4.5.2 Performance Comparison of the Algorithms

We experimented with varying query sizes to compare the performance of the algorithms. We use the largest datasets of size $N = 112,045$ for retinal images and $N = 82,282$ for aerial images with $dim = 6$ for this experiment. Our results show that both TARS and SPARS outperformed linear search on both the datasets, as shown by Figures 4.15 and 4.16. Comparing TARS with SPARS on the aerial

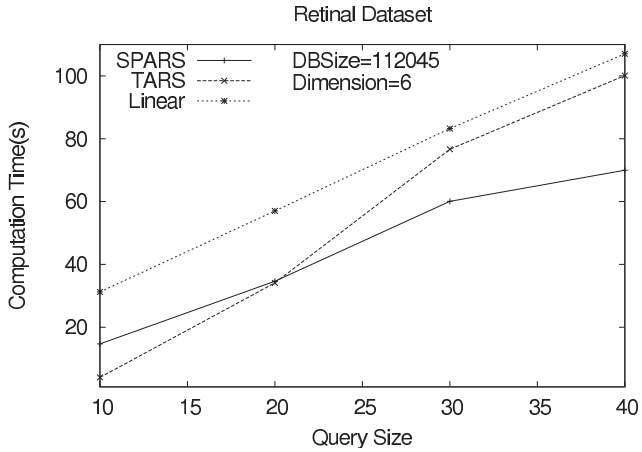


Figure 4.15: Effect of query size on the performance of the algorithms for retinal images.

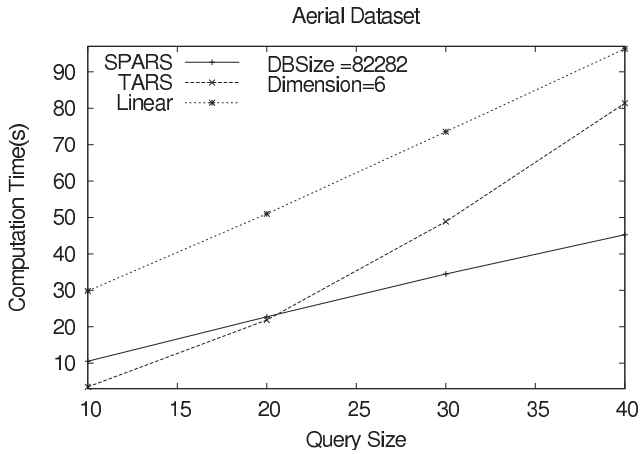


Figure 4.16: Effect of query size on the performance of the algorithms for aerial images.

dataset, we found TARS to be 3 times faster for query size 10 but slower by 2 times for query size of 40 than SPARS (Figure 4.16). The same behavior was

noticed for the retinal dataset where TARS was faster by 3.6 times for $n = 10$ but slower by 1.4 times for $n = 40$ than SPARS (Figure 4.15).

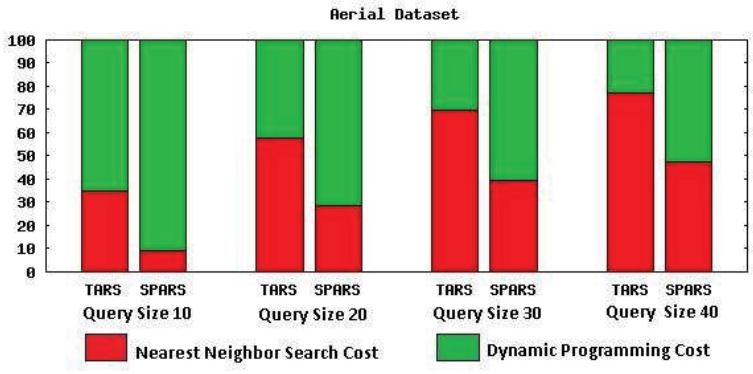


Figure 4.17: Percentage split of NN and DP time for varying query sizes for TARS and SPARS for aerial images.

SPARS performs better than TARS for query sizes of more than 20. We attribute this change in performance of TARS to its multiple traversal through the index structure, as discussed in Section 4.4.1. We measured the average of total nearest-neighbor search cost *NN* and dynamic programming *DP* cost for varying query sizes for both the algorithms TARS and SPARS. We present the percentage of time spent by each algorithm on NN and DP in Figure 4.17. We observe that the *NN* cost increases faster in TARS compared to SPARS as query size increases. TARS is instance optimal [40] and, therefore, it performs better than linear search and SPARS for smaller query sizes when the cost of this multiple traversal is not

high. As this cost increases with increase in query size, its performance is poorer as compared to SPARS.

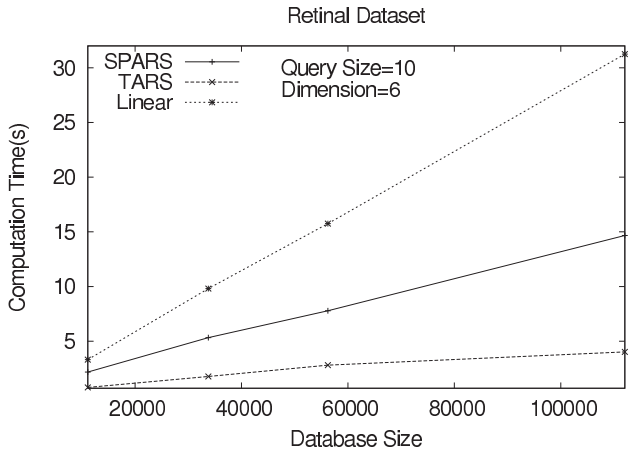


Figure 4.18: Performance of algorithms for varying database sizes of retinal images for query size 10.

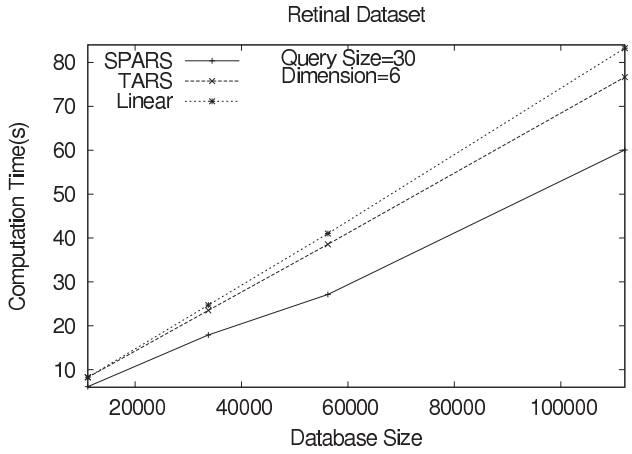


Figure 4.19: Performance of algorithms for varying database sizes of retinal images for query size 30.

We next experimented with varying database sizes for the retinal dataset to confirm the above behavior of the algorithms. For a query size of $n = 10$ and $dim = 6$, we found TARS to be more than 2.7 times faster than SPARS across the database sizes as shown Figure 4.18. SPARS is more than 1.5 times faster than linear search. The performance difference increases with increase in database size. Our other experiment with $n = 30$ and $dim = 6$ found SPARS to be 1.3 time better than TARS and more than 1.3 times better than linear search (Figure 4.19).

We see from the results discussed above that TARS is a better algorithm than the other two for $n \leq 20$ whereas SPARS is better for $n > 20$. TARS saves more than 87% of the query time for $n = 10$ on both the datasets for the largest size. SPARS has a saving of 34% on retinal and 52% on aerial for $n = 40$ on the largest datasets. The average query time of TARS is approximately $4\,s$ on a database of size $N = 112,045$ and a query size of $n = 10$. The average query time for SPARS on the same database is $70\,s$ for query size of 40.

4.5.3 Performance Analysis of SPARS

We next performed detailed analysis of the behavior of the algorithm SPARS for varying n (query size), N (database size) and dim (dimension) on both the datasets. The performance results of SPARS on both datasets for varying N and dim are shown in Figures 4.20 and 4.21. The dataset size is 56,241 for retinal

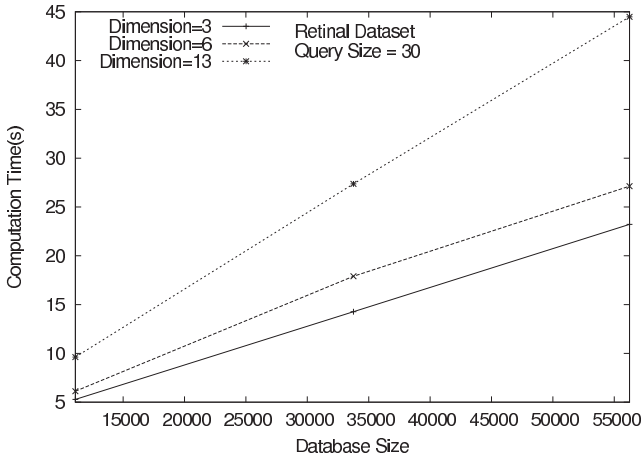


Figure 4.20: Effect of database size and dimension on the performance of SPARS on retinal images.

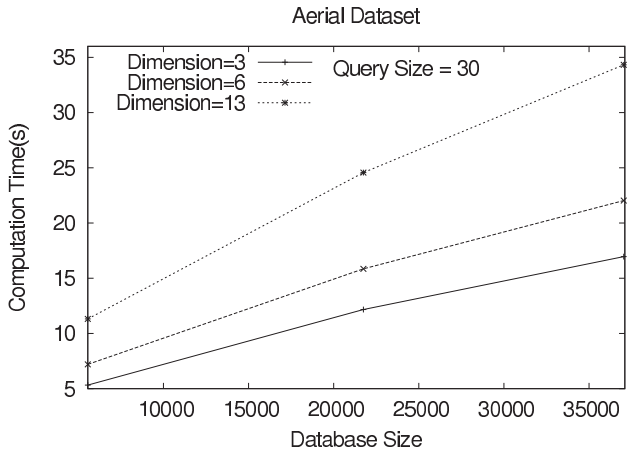


Figure 4.21: Effect of database size and dimension on the performance of SPARS on aerial images.

images and 37,037 for aerial images. SPARS scales linearly for a given query size across varying database sizes and dimensions. The performance of SPARS on both datasets for varying n and dim is shown in Figure 4.22. SPARS exhibited linear

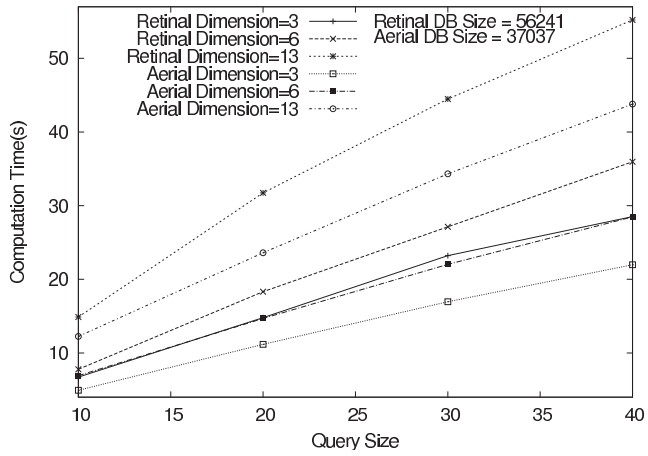


Figure 4.22: Effect of query size and dimension on the performance of SPARS on retinal and aerial images.

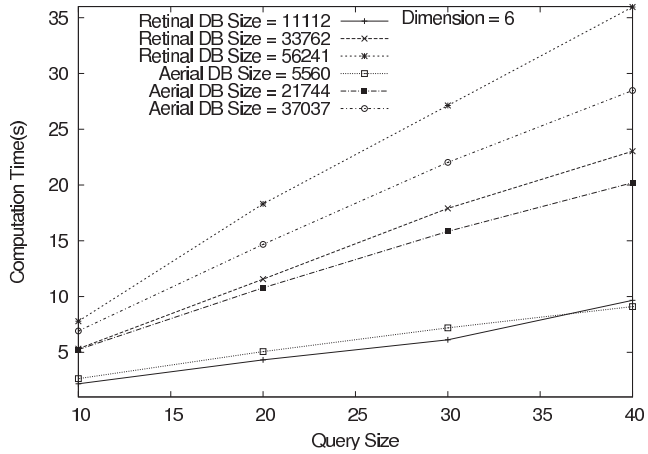


Figure 4.23: Performance of SPARS for varying query sizes and database sizes of retinal and aerial images.

scalability for a given database size across varying query sizes and dimensions. Experiments with varying N and n for $dim = 6$ on both datasets also revealed a linear scalability performance as shown in Figure 4.23.

The exhaustive set of empirical results discussed above confirms a sub-linear scalability for SPARS across varying query size and database size compared to a linear scan of the database. Its scalability is also sub-linear for low dimensions compared to linear scan. This establishes the scalability and efficiency of our algorithm.

4.5.4 Quality Analysis

Dataset	Images	Queries	λ	c	Precision
PA Retinal	80	18	1	23000	80.3%
NF Retinal	37	8	1	23000	82.5%
Aerial	550	7	1	115000	88%

Table 4.4: Datasets used for quality analysis, corresponding parameter values for scoring function, and precision measures.

In this section we analyze the quality of our similarity measure and describe the datasets used for experiments.

Dataset Preparation: We used 3 different datasets to verify the quality of our new similarity measure. From each dataset, we chose interesting regions for querying. For each query, regions in images were manually tagged as a true or a false match. Since the process is manually intensive, we used small datasets as shown in Table 4.4. PA and NF datasets are confocal microscopic images of

cross-sections of feline retina labeled with the lectin peanut-agglutinin and anti-neurofilament antibody respectively. Aerial dataset consists of satellite images of Beverly Hills in California.

Parameter Learning and Precision: Here, we discuss the choice of parameters for the *Discriminator* scoring function described in Section 4.3.2. We use pure black as background for retinal images which is true for the most of the real microscopic images. We use pure black for aerial images also for the purpose of simplicity, though, it can have other backgrounds. Background for other domains need to be determined from knowledge and training. We take the sum of all the pixels in a tile as its distance from background. Distance between tiles is measured using L_1 norm.

We learn the parameter λ and c using manual training with an approach similar to Walrus [98]. For each query, we measured *top-k* precision where $k = 5$ and precision is the ratio of true matches to total matches. We trained the PA dataset on 10 queries. Highest precision (82.0%) was achieved for $\lambda = 1$ and $c = 23000$. These parameters gave an accuracy of 78.6% for 8 other queries over PA dataset. For the same parameter values, 8 queries on NF dataset gave precision of 82.5%. Training and testing on 7 aerial image queries had a precision of 88% for $\lambda = 1$ and $c = 115000$. We summarize the results in Table 4.4. The same parameter

values were used for scalability and efficiency measurements in Section 4.5.2. Our size of the training set was limited by the manually intensive nature of the task.

Finally, we present results for a set of queries from these three datasets in Figure 4.24. All the match for retinal images have been validated by domain scientists and found to be significantly interesting. As shown in the first row of Figure 4.24 , query and the results are examples of biologically interesting fold of retinal tissues labeled with peanut-agglutinin.

4.6 Conclusions

In this chapter, we addressed the problem of querying significant subregions in raster images. We designed a generic scoring scheme to measure similarity between a query image and an image region. We tiled the images to represent a region as a collection of tiles, and each overlap between a query and a database image as a matrix of scores. We proved that the problem of finding a connected subregion of maximal score in a score matrix is NP-hard and then developed a dynamic programming heuristic to score an overlapping region. With this similarity measure, we proposed two index-based scalable search strategies TARS and SPARS for querying in a large repository. These strategies are general enough to work with any scoring scheme and heuristic. We empirically analyzed the perfor-

mance of these algorithms on datasets of 112,045 retinal images and 82,282 aerial images. We save more than 87% search time on small queries using TARS and up to 52% search time on large queries with SPARS on these datasets as compared to linear search. It should be noted that our heuristic for finding the best connected subregions and our access methods for top-k queries (TARS and SPARS) are independent of each other. We demonstrate the quality of our similarity measure (more than 80% precision) with analysis over two real datasets. The ability to extract significant subregions (connected regions with highest score) can have a significant impact on analyzing raster images. Future work includes the formulation of other heuristics for finding similar subregions that have bounded approximation errors on quality and the formulation of other domain-specific scoring functions.

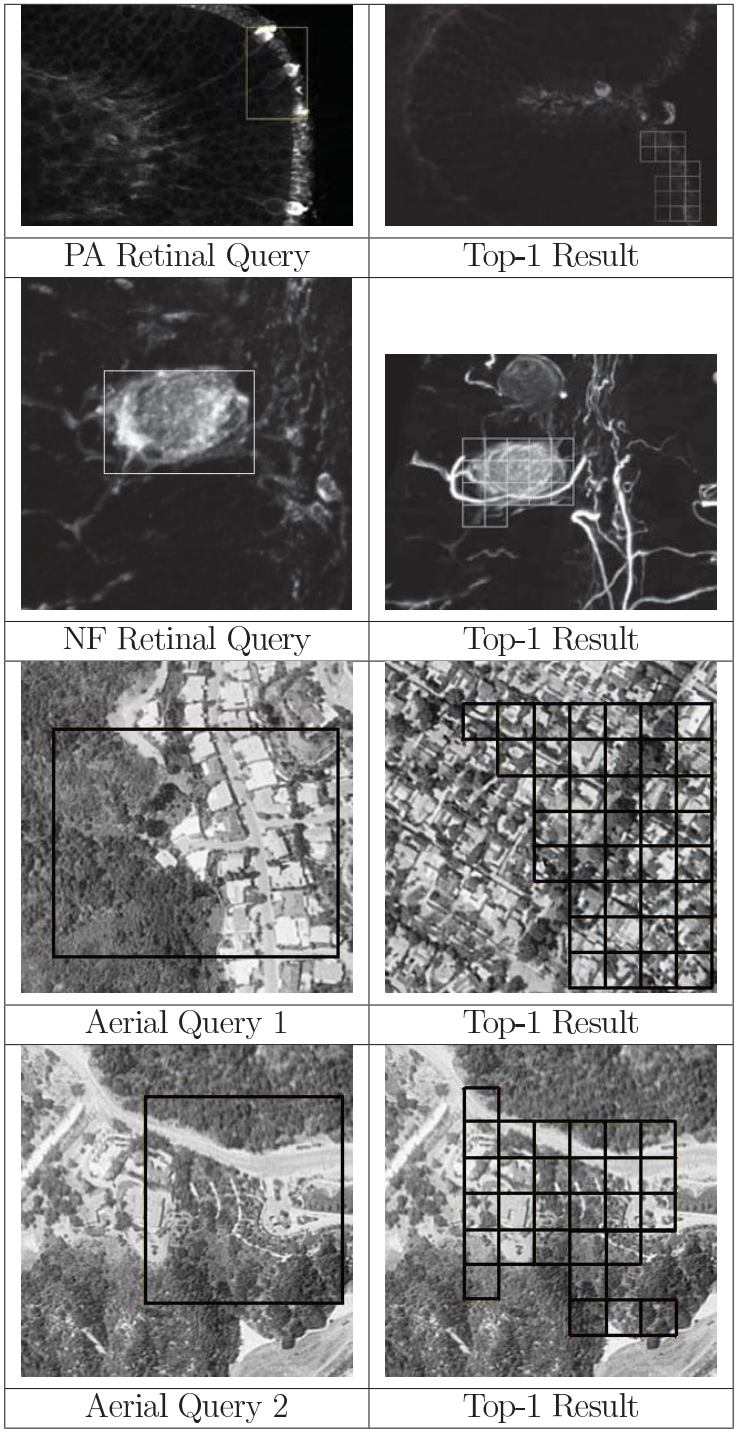


Figure 4.24: Top-1 result for various queries from three real datasets.

Chapter 5

Querying Patterns in Multi-Dimensional Temporal Datasets

Querying patterns in temporal datasets can reveal interesting temporal behaviors and suggest avenues for further scientific exploration. This chapter discusses pattern queries in video datasets. A video is a time series of images. We specifically study the problem of duplicate video detection in this chapter.

We present an efficient and accurate method for duplicate video detection in a large database using video fingerprints. We have empirically chosen the Color Layout Descriptor, a compact and robust frame-based descriptor, to create fingerprints which are further encoded by vector quantization. We propose a new non-metric distance measure to find the similarity between the query and a database video fingerprint and experimentally show its superior performance over other distance measures for accurate duplicate detection. Efficient search can not

be performed for high dimensional data using a non-metric distance measure with existing indexing techniques. Therefore, we develop novel search algorithms based on pre-computed distances and new dataset pruning techniques yielding practical retrieval times. We perform experiments with a database of 38000 videos, worth 1600 hours of content. For individual queries with an average duration of 60 seconds (about 50% of the average database video length), the duplicate video is retrieved in 0.032 seconds, on Intel Xeon with CPU 2.33GHz, with a very high accuracy of 97.5%.

5.1 Introduction

Copyright infringements and data piracy have recently become serious concerns for the ever growing online video repositories. Videos on commercial sites e.g., www.youtube.com, www.metacafe.com, are mainly textually tagged. These tags are of little help in monitoring the content and preventing copyright infringements. Approaches based on *content-based copy detection (CBCD)* and *watermarking* have been used to detect such infringements [64, 75]. The watermarking approach tests for the presence of a certain watermark in a video to decide if it is copyrighted. The other approach (CBCD) finds the duplicate by comparing the fingerprint of the query video with the fingerprints of the copyrighted videos. A

fingerprint is a compact signature of a video which is robust to the modifications of the individual frames and discriminative enough to distinguish between videos. The noise robustness of the watermarking schemes is not ensured in general [75], whereas the features used for fingerprinting generally ensure that the *best match in the signature space remains mostly unchanged even after various noise attacks.* Hence, the fingerprinting approach has been more successful.

We define a "duplicate" video as the one *consisting entirely of a subset of the frames in the original video - the individual frames may be further modified and their temporal order varied.* The assumption that a duplicate video contains frames only from a single video has been used in various copy detection works, e.g., [85],[114],[130]. In [130], it is shown that for a set of 24 queries searched in YouTube, Google Video and Yahoo Video, 27% of the returned relevant videos are duplicates. In [23], each web video in the database is reported to have an average of five similar copies - the database consisted of 45000 clips worth 1800 hours of content. Also, for some popular queries to the Yahoo video search engine, there were two or three duplicates among the top ten retrievals [85].

In Figure 5.1, we present the block diagram of our duplicate video detection system. The relevant symbols are explained in Table 5.1. The database videos are referred to as "model" videos in the chapter. Given a model video V_i, the decoded frames are sub-sampled at a factor of 5 to obtain T_i frames and a p dimensional

feature is extracted per frame. Thus, a model video V_i is transformed into a $T_i \times p$ matrix Z^i. We empirically observed in Section 5.3 that the Color Layout Descriptor (CLD) [68] achieved higher detection accuracy than other candidate features. To summarize Z^i, we perform k-means based clustering and store the cluster centroids $\{X_j^i\}_{j=1}^{F_i}$ as its fingerprint. The number of clusters F_i is fixed at a certain fraction of T_i, e.g., a fingerprint size of 5x means that $F_i = (5/100)T_i$. Therefore, the fingerprint size varies with the video length. K-means based clustering generally produces compact video signatures which are comparable to those generated by sophisticated summarization techniques as discussed in [99]. In [1], we have compared different methods for keyframe selection for creating the compact video signatures.

The duplicate detection task is to retrieve the best matching model video fingerprint for a given query fingerprint. The model-to-query distance is computed using a new non-metric distance measure between the fingerprints as discussed in Section 5.4. We also empirically show that our distance measure results in significantly higher detection accuracy than traditional distance measures (L_1, partial Hausdorff distance [54, 57], Jaccard [25] and cosine distances). We design access methods for fast and accurate retrieval of duplicate videos. The challenge in developing such an access method is two-fold. *Firstly, indexing using such distances has not been well-studied till date* - the recently proposed distance based hashing

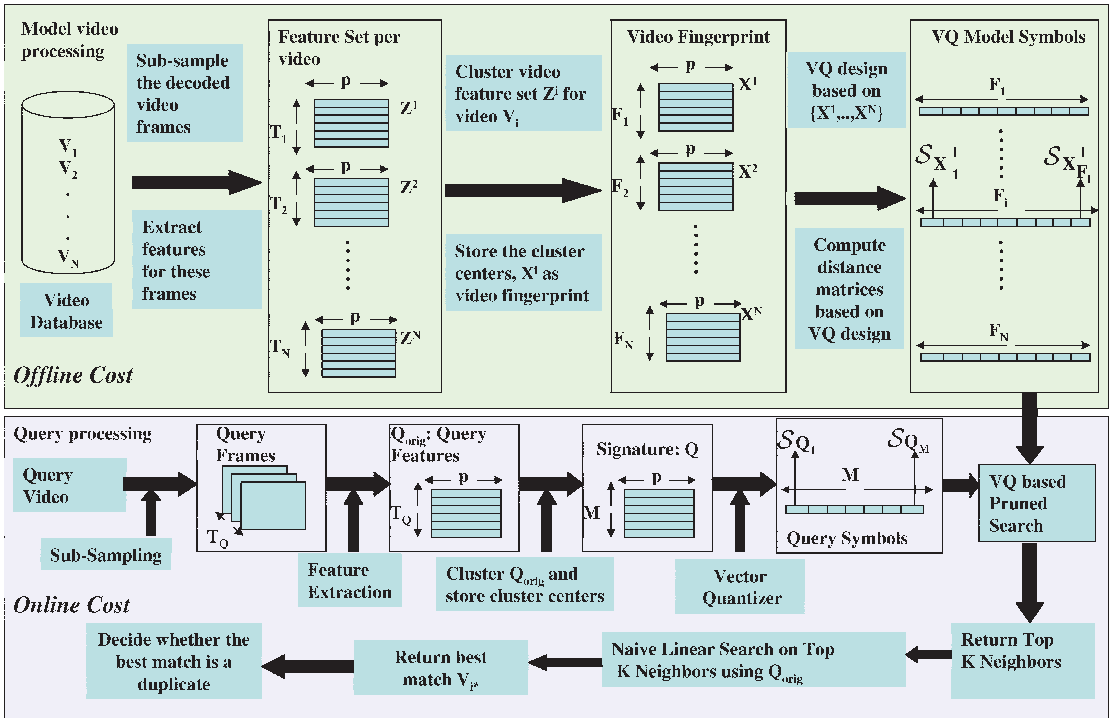


Figure 5.1: Block diagram of the proposed duplicate detection framework - the symbols used are explained in Table 5.1.

[6] performs dataset pruning for arbitrary distances. Secondly, video fingerprints are generally of high dimension and varying length. *Current indexing structures (M-tree [26], R-tree [52], kd tree [11]) are not efficient for high-dimensional data.*

To perform efficient search, we propose a two phase procedure. The first phase is a coarse search to return the top-K nearest neighbors (NN) which is the focus of the chapter. We perform vector quantization (VQ) on the individual vectors in the model (or query) fingerprint X^i (or Q) using a codebook of size U (=

Notation	Definition
N	Number of database videos
V_i	i^{th} model video in the dataset
V_{i*}	Best matched model video for a given query
p	Dimension of the feature vector computed per video frame
$Z^i \in \mathbb{R}^{T_i \times p}$, T_i	Z^i is the feature vector matrix of V_i, where V_i has T_i frames after temporal sub-sampling
$X^i \in \mathbb{R}^{F_i \times p}$, F_i	X^i is the k-means based signature of V_i, which has F_i keyframes
X^i_j	j^{th} vector of video fingerprint X^i
U	Size of the vector quantizer (VQ) codebook used to encode the model video and query video signatures
$Q_{orig} \in \mathbb{R}^{T_Q \times p}$	Query signature created after sub-sampling, where T_Q refers to the number of sub-sampled query frames
$Q \in \mathbb{R}^{M \times p}$	Keyframe based query fingerprint, where M is the number of query keyframes
C_i	i^{th} VQ codevector
$\overrightarrow{x_i}$, $\overrightarrow{q}$	$\overrightarrow{x_i}$ is VQ based signature of V_i, while $\overrightarrow{q}$ is VQ based query signature
$\mathcal{S}_{X^i_j}$	VQ symbol index to which X^i_j is mapped
$\mathcal{A}$	Set of N "model signature to query signature" distances
$\mathbb{D} \in \mathbb{R}^{U \times U}$	Inter VQ-codevector distance matrix, for L_1 distance between VQ codevectors
$\mathbb{D}^* \in \mathbb{R}^{N \times U}$	Lookup table of shortest distance values from each VQ-based model signature to each VQ codevector
$\mathbb{C}(i)$	Cluster containing the video indices whose VQ-based signatures have the i^{th} dimension as non-zero
$\|\|\overrightarrow{x_1} - \overrightarrow{x_2}\|\|_p$	L_p norm of the vector $(\overrightarrow{x_1} - \overrightarrow{x_2})$.
$\|E\|$	Cardinality of the set E
ℓ	Fractional query length = (number of query frames/number of frames in the original model video)

Table 5.1: Glossary of notations

8192) to generate a sparse histogram-based signature $\overrightarrow{x_i}$ (or $\overrightarrow{q}$). This is discussed in Section 5.5.2. Coarse search is performed in the VQ-based signature space. Various techniques proposed to improve the search are the use of pre-computed

information between VQ symbols, partial distance based pruning, and the dataset pruning as discussed in Section 5.5. The second phase uses the unquantized features (X^i) for the top-K NN videos to find the best matching video V_{i^*}. The final module (Section 5.7) decides whether the query is indeed a duplicate derived from V_{i^*}.

The computational cost for the retrieval of the best matching model video has two parts (Figure 5.1).

1) **Offline cost** (model related) - consists of the un-quantized model fingerprint generation, VQ design and encoding of the model signatures, and computation and storing of appropriate distance matrices.

2) **Online cost** (query related) - the query video is decoded, sub-sampled, keyframes are identified, and features are computed per keyframe - these constitute the query **pre-processing cost**. In this chapter, we report the **query time** - this comprises of the time needed to obtain k-means based compact signatures, perform VQ-based encoding on the signatures to obtain sparse histogram-based representations, compute the relevant lookup tables, and then perform two-stage search to return the best matched model video.

The chapter is organized as follows. Section 5.2 contains some relevant previous work. Feature selection for fingerprint creation is discussed in Section 5.3. Section 5.4 introduces our proposed distance measure. The various search algo-

rithms, along with the different pruning methods, are presented in Section 5.5. The dataset creation for this task is explained in Section 5.6.1. Section 5.6.2 contains the experimental results while Section 5.7 describes the final decision module which makes the "duplicate/non-duplicate decision".

Main Contributions

• We propose a new non-metric distance function for duplicate video detection when the query is a noisy subset of a single model video. It performs better than other conventional distance measures.

• For the VQ-based model signatures retained after dataset pruning, we reduce the search time for the top-K candidates by using suitable pre-computed distance tables and by discarding many non-candidates using just the partially computed distance from these model video signatures to the query.

• We present a dataset pruning approach, based on our distance measure in the space of VQ-encoded signatures, which returns the top-K nearest neighbors (NN) even after pruning. We obtain significantly higher pruning than that provided by distance based hashing [6] methods, trained on our distance function.

In this chapter, the terms "signature" and "fingerprint" have been used interchangeably. "Fractional query length" (ℓ in Table 5.1) refers to the fraction of the model video frames that constitute the query. Also, for a VQ of codebook size U, the 1-NN of a certain codevector is the codevector itself.

5.2 Literature Survey

A good survey for video copy detection methods can be found in [75]. Many schemes use global features (e.g., color histogram computed over the entire video) for a fast initial search for prospective duplicates [130]. Then, keyframe-based features are employed for a more refined search.

Keypoint based features: In an early duplicate detection work by Joly *et al.* [66], the keyframes correspond to extrema in the global intensity of motion. Local interest points are identified per keyframe using the Harris corner detector and local differential descriptors are then computed around each interest point. These descriptors have been subsequently used in other duplicate detection works [64, 65, 74, 75]. In [130], PCA-SIFT features [69] are computed per keyframe on a host of local keypoints obtained using the Hessian-Affine detector [92]. Similar local descriptors are also used in [141], where near-duplicate keyframe (NDK) identification is performed based on matching, filtering and learning of local interest points. A recent system for fast and accurate large-scale video copy detection, the Eff^2 Videntifier [29], uses Eff^2 descriptors [79] from the SIFT family [86]. In [140], a novel measure called Scale-Rotation Invariant Pattern Entropy (SR-PE) is used to identify similar patterns formed by keypoint matching of near-duplicate image pairs. A combination of visual similarity (using global histogram for coarser

search and local point based matching for finer search) and temporal alignment is used to evaluate video matching for duplicate detection in [121]. VQ based techniques are used in [24] to build a SIFT-histogram based signature for duplicate detection.

Global Image Features: In some approaches, the duplicate detection problem involves finding the similarity between sets of time-sequential video keyframes. A combination of MPEG-7 features such as the Scalable Color Descriptor, Color Layout Descriptor (CLD) [68] and the Edge Histogram Descriptor (EHD) has been used for video-clip matching [15], using a string-edit distance measure. For image duplicate detection, the Compact Fourier Mellin transform (CFMT) [36] has also been shown to be very effective in [48] and the compactness of the signature makes it suitable for fingerprinting.

Entire Video based Features: The development of "ordinal" features [16] gave rise to very compact signatures which have been used for video sequence matching [95]. Li *et al.* [82] used a binary signature to represent each video, by merging color histogram with ordinal signatures, for video clip matching. Yuan *et al.* [134] also used a similar combination of features for robust similarity search and copy detection. UQLIPS, a recently proposed real-time video clip detection system [114], uses RGB and HSV color histograms as the video features. A localized color histogram based global signature is proposed in [85].

Indexing Methods: Each keyframe is represented by a host of feature points, each having a descriptor. The matching process involves comparison of a large number of interest point pairs which is computationally intensive. Several indexing techniques have been proposed for efficient and faster search. Joly *et al.* [66] use an indexing method based on the Hilbert's space filling curve principle. In [64], the authors propose an improved index structure for video fingerprints, based on Statistical Similarity Search (S^3) where the "statistical query" concept was based on the distribution of the relevant similar fingerprints. A new approximate similarity search technique was proposed in [67] and later used in [74, 65] where the probabilistic selection of regions in the feature space is based on the distribution of the feature distortion. In [141], an index structure LIP-IS is proposed for fast filtering of keypoints under one-to-one symmetric matching. For the Videntifier [29] system, the approximate NN search in the high-dimensional database (of Eff^2 descriptors) is done using the NV-tree [78], an efficient disk-based data structure.

Hash-based Index: The above mentioned indexing methods are generally compared with locality sensitive hashing (LSH) [49, 31], a popular approximate search method for L_2 distances. Since our proposed distance function is non-metric, LSH cannot be used in our setup as the locality sensitive property holds only for metric distances. Instead, we have experimented with the recently proposed distance

based hashing (DBH) [6] scheme, which can be used for arbitrary distance measures.

Final Duplicate Confirmation: From the top retrieved candidate, the duplicate detection system has to validate whether the query has indeed been derived from it. The keyframes for a duplicate video can generally be matched with the corresponding frames in the original video using suitable spatio-temporal registration methods. In [65, 75], the approximate NN results are post-processed to compute the most globally similar candidate based on a registration and vote strategy. In [74], Law-To *et al.* use the interest points proposed in [66] for trajectory building along the video sequence. A robust voting algorithm utilizes the trajectory information, spatio-temporal registration, as well as the labels computed during the off-line indexing to make the final retrieval decision. In our duplicate detection system, we have a "distance threshold based" (Section 5.7.1) and a registration-based framework (Section 5.7.2) to determine if the query is actually a duplicate derived from the best-matched model video.

The advantages of our method over other state-of-the-art methods are summarized below.

• In current duplicate detection methods, the query is assumed to contain a large fraction of the original model video frames. Hence, the query signature, computed over the entire video, is assumed to be similar to the model video signature.

This assumption, often used as an initial search strategy to discard outliers, does not hold true when the query is only a small fraction (e.g., 5%) of the original video. For such cases, the query frames have to be individually compared with the best matching model frames, as is done by our distance measure. As shown later in Figures. 5.3 and 5.8, we observe that our proposed distance measure performs much better than other distances for duplicate detection for shorter queries.

- We develop a set of efficient querying techniques with the proposed distance measure which achieves much better dataset pruning than distance based hash (DBH) - DBH is the state-of-the-art method for querying in non-metric space.

- In [65, 75], the registration step is performed between query frames and other model frames to confirm whether the query is a duplicate derived from the model video. In our distance computation procedure, we also end up computing which model vector serves as the best match for a query vector - this inter-vector correspondence helps in faster identification of the best matching model keyframe for a given query keyframe (discussed in Section 5.7.2). This frame-to-frame correspondence is needed for effective registration.

5.3 Feature Extraction

Candidate Features

We performed duplicate video detection with various frame based features - CLD, CFMT [36], Localized Color Histogram (LCH) [85] and EHD [129]. The LCH feature divides the image into a certain number of blocks and the 3D color histogram is computed per block. E.g., if each color channel is quantized into 4 levels, the 3D histogram per image block has $4^3 = 64$ levels. If the image is divided into two partitions along each direction, the total LCH feature dimension is $4^3 \times 2^2 = 256$. To study the variation of detection accuracy with signature size, we have considered the LCH feature for dimensions 256 and 32 ($2^2 \times 2^3 = 32$). For the frame-based features, we use our proposed distance measure, which is explained in Section 5.4.

We also considered video features (computed globally, i.e. over the entire video and not per key-frame). One such global feature used is the m-dimensional histogram obtained by mapping each of the 256-dimensional LCH vectors to one of m codebook vectors (this signature creation is proposed in [85]), obtained after k-means clustering of the LCH vectors. We have experimented with $m = 20$, 60, 256 and 8192, and L_2 distance is used. The other global feature is based on a combination of the ordinal and color histograms [134]. Both the ordinal and color

histograms are 72-dimensional (24 dimensions along each of the Y, Cb and Cr channels) and the distance measure used is a linear combination of the average of the distance between the ordinal histograms and the minimum of the distance between the color histograms, among all the 3 channels.

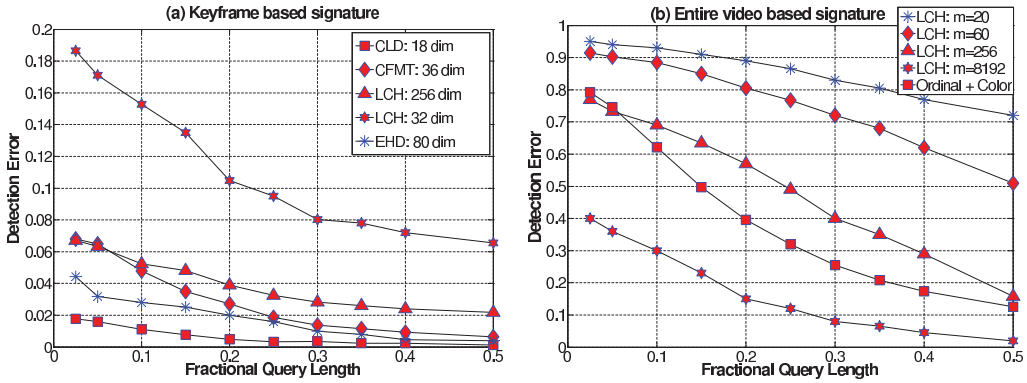


Figure 5.2: Comparison of the duplicate video detection error for (a) keyframe based features and (b) entire video based features: the query length is varied from 2.5% to 50% of the actual model video length. The error is averaged over all the query videos generated using noise addition operations, as discussed later in Section 5.6.1. The model fingerprint size used in (a) is 5x.

Experimental Setup and Performance Comparison

We describe the duplicate detection experiments for feature comparison. We use a database of 1200 video fingerprints and a detection error occurs when the best matched video is not the actual model from which the query was derived. The query is produced using one of various image processing/noise addition methods, discussed in Section 5.6.1. The query length is gradually reduced from 50% to 2.5% of the model video length and the detection error (averaged over all noisy queries)

is plotted against the fractional query length (Figure 5.2). For our dataset, a fractional query length of 0.05 corresponds, on an average, to 6 sec of video $\approx$ 30 frames, assuming 25 frames/sec and a sub-sampling factor of 5. Figure 5.2(a) compares frame based features while Figure 5.2(b) compares video based features.

In our past work [111], we have shown that the CFMT features perform better as video fingerprints than SIFT features for duplicate detection. Here, it is seen that for duplicate detection, 18-dim CLD performs slightly better than 80-dim EHD, which does better than 36-dim CFMT and 256-dim LCH (Figure 5.2(a)). Due to the lower signature dimension and superior detection performance, we choose 18-dim CLD feature per keyframe for fingerprint creation. It is seen that for a short query clip, the original video histogram is often not representative enough for that clip leading to higher detection error, as in Figure 5.2(b). Hence, using a global signature with L_2 distance works well only for longer queries.

We briefly describe the CLD feature vector and also provide some intuition as to why it is highly suited for the duplicate detection problem. The CLD signature [68] is obtained by converting the image to a 8×8 image, on averaging, along each (Y/Cb/Cr) channel. The Discrete Cosine Transform (DCT) is computed for each image. The DC and first 5 (in zigzag scan order) AC DCT coefficients for each channel constitute the 18-dimensional CLD feature. The CLD feature is compact and captures the frequency content in a highly coarse representation of the image.

As our experimental results suggest, different videos can be distinguished even at this coarse level of representation for the individual frames. Also, due to this coarse representation, *image processing and noise operations, which are global in nature, do not alter the CLD significantly* so as to cause detection errors; thus, the feature is robust enough. Significant cropping or gamma variation can distort the CLD sufficiently to cause errors - a detailed comparison of its robustness to various attacks is presented later in Table 5.7. Depending on the amount of cropping, the 8×8image considered for CLD computation can change significantly, thus severely perturbing the CLD feature. Also, significant variations in the image intensity through severe gamma variation can change the frequency content, even for an 8×8 image representation, so as to cause detection errors.

Storage-wise, our system consumes much less memory compared to methods which store key-point based descriptors [65, 29]. The most compact key-point based descriptor is the 20-dim vector proposed in [65] where each dimension is represented by 8 bits and 17 feature vectors are computed per second. The corresponding storage is 10 times that of our system (assuming 18-dimensional CLD features per frame where each dimension is stored as a double, 25 frames/sec, temporal sub-sampling by 5, 5% of the sub-sampled frames being used to create the model signature).

5.4 Proposed Distance Measure

Our proposed distance measure to compare a model fingerprint X^i with the query signature Q is denoted by $d(X^i, Q)$ [1] (5.1). This distance is the sum of the best-matching distance of each vector in Q with all the vectors in X^i. In (5.1), $\|X^i_j - Q_k\|_1$ refers to the L_1 distance between X^i_j, the j^{th} feature vector of X^i and Q_k, the k^{th} feature vector of Q. Note that $d(\cdot, \cdot)$ is a quasi-distance.

$$d(X^i, Q) = \sum_{k=1}^{M} \left\{ \min_{1 \le j \le F_i} \|X^i_j - Q_k\|_1 \right\} \tag{5.1}$$

What is the motivation behind this distance function? We assume that each query frame in a duplicate video is a tampered/processed version of a frame in the original model video. Therefore, the summation of the best-matching distance of each vector in Q with all the vectors in the signature for the original video (X^i) will yield a small distance. Hence, the model-to-query distance is small when the query is a (noisy) subset of the original model video. Also, this definition accounts for those cases where the duplicate consists of a reordering of scenes from the original video.

A comparison of distance measures for video copy detection is presented in [54]. Our distance measure is similar to the Hausdorff distance [57, 54]. For our problem, the Hausdorff distance $h(X^i, Q)$ and the partial Hausdorff distance

[1] For ease of understanding, the quasi-distance measure $d(\cdot, \cdot)$ is referred to as a distance function in subsequent discussions.

$h_P(X^i, Q)$ are interpreted as:

$$h(X^i, Q) = \max_{1 \le k \le M} \left\{ \min_{1 \le j \le F_i} \|X_j^i - Q_k\|_1 \right\} \tag{5.2}$$

$$h_P(X^i, Q) = \underbrace{P^{th}\text{largest}}_{1 \le k \le M} \left\{ \min_{1 \le j \le F_i} \|X_j^i - Q_k\|_1 \right\} \tag{5.3}$$

For image copy detection, the partial Hausdorff distance (5.3) has been shown to be more robust than the Hausdorff distance (5.2) in [54]. We compare the performance of $h_P(X^i, Q)$ (5.3) for varying P, with $d(X^i, Q)$, as shown in Figure 5.3, using the same experimental setup as in Section 5.3. It is seen that the results using $d(X^i, Q)$ are better - *the improved performance is more evident for shorter queries.*

Intuitively, why does our distance measure perform better than the Hausdorff distance? In (5.2) (or (5.3)), we first find the "minimum query frame-to-model video" distance for every query frame and then find the maximum (or P^{th} largest) among these distances. *Thus, both $h(X^i, Q)$ and $h_P(X^i, Q)$ effectively depend on a single query frame and model video frame, and errors occur when this query (or model) frame is not representative of the query (or model) video.* In our distance function (5.1), $d(X^i, Q)$ is computed considering all the "minimum query frame-to-model video" terms and hence, the effect of one (or more) mismatched query feature vector is compensated.

Dynamic time warping (DTW) [107] is commonly used to compare two sequences of arbitrary lengths. The proposed distance function has been compared to DTW in [1], where it is shown that DTW works well only when the query is a continuous portion of the model video and not a collection of disjoint parts. *This is because DTW considers temporal constraints and must match every data point in both the sequences.* Hence, when there is any mismatch between two sequences, DTW takes that into account (thus increasing the effective distance), while the mismatch is safely ignored in our distance formulation.

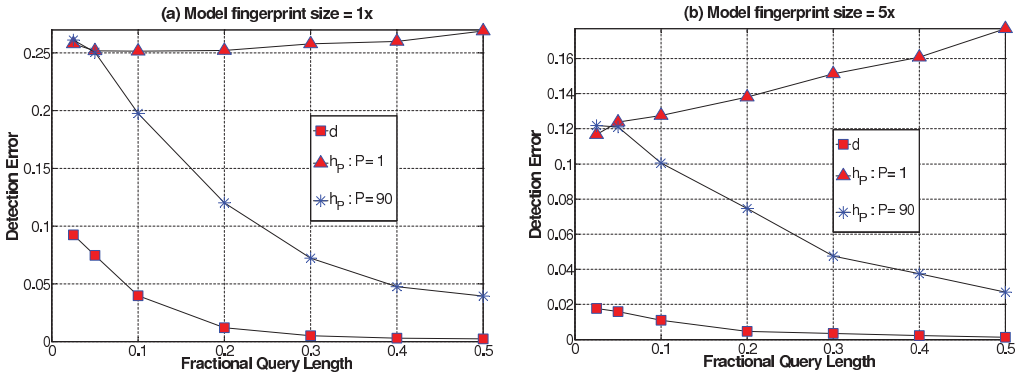


Figure 5.3: Comparison of the duplicate video detection error for the proposed distance measure $d(\cdot,\cdot)$ (5.1) and the Hausdorff distances: here, $(h_p : P = k)$ refers to the partial Hausdorff distance (5.3) where the k^{th} maximum is considered.

5.5 Search Algorithms

In this section, we develop a two-phase approach for fast duplicate retrieval. The proposed distance measure (5.1) is used in our search algorithms for duplicate

detection. First, we discuss a *naive linear search algorithm* in Section 5.5.1. Search techniques based on the vector quantized representation of the fingerprints that achieve speedup through suitable lookup tables are discussed in Section 5.5.2. Algorithms for further speedup based on dataset pruning are presented in Section 5.5.3.

We perform temporal sub-sampling of the query video to get a signature Q_{orig} having T_Q vectors (see Figure 5.1 and Table 5.1). The initial coarse search (first pass) uses a smaller query signature Q, having M ($M < T_Q$) vectors. Q consists of the cluster centroids obtained after k-means clustering on Q_{orig}. When $M = (5/100)T_Q$, we refer to the query fingerprint size as 5x. The first pass returns the top-K NN from all the N model videos. The larger query signature (Q_{orig}) is used for the second pass to obtain the best matched video from these K candidates using a naive linear scan. As the query length decreases, the query keyframes may differ significantly from the keyframes of the actual model video; hence, the first pass needs to return more candidates to ensure that the actual model video is one of them.

A naive approach for the search is to compute all the N model-to-query distances and then find the best match. This set of N distances is denoted by $\mathcal{A}$ (5.4). We speedup the coarse search by removing various computation steps involved in

$\mathcal{A}$. For the purpose of explaining the speedup obtained by various algorithms, we provide the time complexity breakup in Table 5.2.

$$\mathcal{A} = \{d(X^i, Q)\}_{i=1}^{N} = \left\{ \sum_{k=1}^{M} \left\{ \min_{1 \le j \le F_i} \|X_j^i - Q_k\|_1 \right\} \right\}_{i=1}^{N} \tag{5.4}$$

5.5.1 Naive Linear Search (NLS)

The Naive Linear Search (NLS) algorithm implements the two-pass method without any pruning. In the first pass, it retrieves the top-K candidates based on the smaller query signature Q by performing a full dataset scan using an ascending priority queue L of length K. The priority queue is also used for the other coarse search algorithms in this section to keep track of the top-K NN candidates. The k^{th} entry in L holds the model video index ($L_{k,1}$) and its distance from the query ($L_{k,2}$). A model signature is inserted into L if the size of L is less than K or its distance from the query is smaller than the largest distance in the queue. In the second pass, NLS computes the distance of the K candidates from the larger query signature Q_{orig} so as to find the best matched candidate. The storage needed for all the model signatures $= O(N\bar{F}p)$, where $\bar{F}$ denotes the average number of vectors in a model fingerprint.

Time	Operation involved	Complexity
T_{11}	Computing L_1 distance between vectors X_j^i and Q_k : $\|X_j^i - Q_k\|_1$	$O(p)$
$T_{12} = T_{11}.F_i$	Finding the best matched model vector for a query vector : $\min_{1 \le j \le F_i} \|X_j^i - Q_k\|_1$	$O(F_i p)$
$T_2 = M.T_{12}$	Finding the best match for all M frames in Q to compute $d(X^i, Q)$	$O(MF_i p)$
$T_3 = \sum_{i=1}^{N} T_2$	Computing all N model-to-query distances : $\mathcal{A} = \{d(X^i, Q)\}_{i=1}^{N}$	$O(MN\bar{F}p)$
T_4	Retrieve minimum K values from $\mathcal{A}$ to return top-K videos using a priority queue	$O(N \log K)$
T_5	Finding V_{i^*} from top-K videos using larger query signature Q_{orig}	$O(T_Q K\bar{F}p + K)$

Table 5.2: Time complexity of the various modules involved in computing $\mathcal{A} = \{d(X^i, Q)\}_{i=1}^{N}$ (5.4), returning the top-K NN, and then finding the best matched video V_{i^*} from them. $\bar{F} = \sum_{i=1}^{N} F_i/N$ denotes the average number of vectors in a model fingerprint. For the VQ-based schemes, the distance $d(\cdot, \cdot)$ is replaced by the distance $d_{VQ}(\cdot, \cdot)$ (5.9), while the other operations involved remain similar.

5.5.2 Vector Quantization and Acceleration Techniques

From Table 5.2, it is observed that time T_{11} can be saved by pre-computing the inter-vector distances. When the feature vectors are vector quantized, an inter-vector distance reduces to an inter-symbol distance, which is fixed once the VQ codevectors are fixed. Hence, we vector quantize the feature vectors and represent the signatures as histograms, whose bins are the VQ symbol indices. For a given VQ, we pre-compute and store the inter-symbol distance matrix in memory.

We now describe the VQ-based signature creation. Using the CLD features extracted from the database video frames, a VQ of size U is constructed using the Linde-Buzo-Gray algorithm [84]. The distance $d(\cdot, \cdot)$ (5.1) reduces to $d_{VQM}(\cdot, \cdot)$

(5.5) for the VQ-based framework, where $\mathbb{D}$ is the inter-VQ codevector distance matrix (5.6). $C_{\mathcal{S}_{X_j^i}}$ refers to the ${\mathcal{S}_{X_j^i}}^{th}$ codevector, i.e. the codevector to which the VQ maps X_j^i.

$$d_{VQM}(X^i, Q) = \sum_{k=1}^{M} \min_{1 \le j \le F_i} \|C_{\mathcal{S}_{X_j^i}} - C_{\mathcal{S}_{Q_k}}\|_1 = \sum_{k=1}^{M} \min_{1 \le j \le F_i} \mathbb{D}(\mathcal{S}_{X_j^i}, \mathcal{S}_{Q_k}) \tag{5.5}$$

$$\text{where } \mathbb{D}(k_1, k_2) = \|C_{k_1} - C_{k_2}\|_1,\ 1 \le k_1, k_2 \le U \tag{5.6}$$

Let $\overrightarrow{q} = [q_1, q_2, \cdots, q_U]$ denote the normalized histogram-based query signature (5.7) and $\overrightarrow{x_i} = [x_{i,1}, x_{i,2}, \cdots, x_{i,U}]$ denote the corresponding normalized model signature (5.8) for video V_i.

$$q_k = |\{j : \mathcal{S}_{Q_j} = k,\ 1 \le j \le M\}|/M \tag{5.7}$$

$$x_{i,k} = |\{j : \mathcal{S}_{X_j^i} = k,\ 1 \le j \le F_i\}|/F_i \tag{5.8}$$

Generally, consecutive video frames are similar; hence, many of them will get mapped to the same VQ codevector while many VQ codevectors may have no representatives (for a large enough U). Let $\{t_1, t_2, \cdots, t_{N_q}\}$ and $\{n_{i,1}, n_{i,2}, \cdots, n_{i,N_{x_i}}\}$ denote the non-zero dimensions in $\overrightarrow{q}$ and $\overrightarrow{x_i}$, respectively, where N_q and N_{x_i} denote the number of non-zero dimensions in $\overrightarrow{q}$ and $\overrightarrow{x_i}$, respectively.

The distance between the VQ-based signatures $\overrightarrow{x_i}$ and $\overrightarrow{q}$ can be expressed as:

$$d_{VQ}(\overrightarrow{x_i}, \overrightarrow{q}) = \sum_{k=1}^{N_q} q_{t_k} . \left\{ \min_{1 \le j \le N_{x_i}} \mathbb{D}(t_k, n_{i,j}) \right\} \tag{5.9}$$

It can be shown that the distances in (5.5) and (5.9) are identical, apart from a constant factor.

$$d_{VQM}(X^i, Q) = M.d_{VQ}(\overrightarrow{x_i}, \overrightarrow{q}) \tag{5.10}$$

The model-to-query distance (5.9) is same for different model videos if their VQ-based signatures have the same non-zero dimensions. For our database of 38000 videos, the percentage of video pairs (among $\binom{38000}{2}$ pairs) that have the same non-zero indices is merely $3.2 \times 10^{-4}\%$ [1]. A note about our VQ-based signature - since we discard the temporal information and are concerned with the relative frequency of occurrence of the various VQ symbols (one symbol per frame), the signature is similar to the "bag-of-words" model commonly used for text analysis and computer vision applications.

The distance computation involves considering all possible pairs between the N_q non-zero query dimensions and the N_{x_i} non-zero model dimensions. We propose a technique where the distance computation can be discarded based on a partially computed (not all $N_q.N_{x_i}$ pairs are considered) distance - we call it "Partial Distance based Pruning" (PDP) (Section 5.5.2). We then present two VQ-based techniques (VQLS-A in Section 5.5.2 and VQLS-B in Section 5.5.2) which use different lookup tables, utilize PDP for faster search and significantly outperform the NLS scheme.

Partial Distance Based Pruning (PDP)

We present a technique (PDP) that reduces time T_3 (Table 5.2) by computing only partially N model-to-query distances in $\mathcal{A}$. This speedup technique is generic enough to be used for both the un-quantized and the VQ-based signatures. We insert a new distance in the priority queue L if it is smaller than the largest distance in the queue ($L_{K,2}$). The logic behind PDP is that if the partially computed model-to-query distance exceeds $L_{K,2}$, the full distance computation is discarded for that model video.

Let $\hat{d}(X^i, Q, k')$ be the distance between X^i and the first k' vectors of Q - this is a partially computed model-to-query distance for $k' < M$. If $\hat{d}(X^i, Q, k')$ exceeds $L_{K,2}$, we discard the model video signature X^i as a potential top-K NN candidate and save time spent on computing $d(X^i, Q)$, its total distance from the query. Though we spend additional time for comparison in each distance computation (comparing $\hat{d}(X^i, Q, k')$ to $L_{K,2}$), we get a substantial reduction in the search time as shown later in Figure 5.6.

When PDP is used in the un-quantized feature space, we call that method as Pruned Linear Search (PLS). The total storage space required for PLS is also $O(N\bar{F}p)$, like NLS. Since we do not consider all the M vectors of Q in most of the distance computations, we have $m \leq M$ vectors participating, on an average, in the distance computation. Therefore, the time required to compute $\mathcal{A}$, T_3

(Table 5.2) now reduces to $O(mN\bar{F}p)$. The other computational costs are same as that for NLS.

$$\hat{d}(X^i, Q, k') = \sum_{k=1}^{k'} \left\{ \min_{1 \le j \le F_i} \|X_j^i - Q_k\|_1 \right\} \tag{5.11}$$

$$\text{For } k' \le M,\ \hat{d}(X^i, Q, k') \le \hat{d}(X^i, Q, M), \text{ and } \hat{d}(X^i, Q, M) = d(X^i, Q)$$

$$\therefore\ \hat{d}(X^i, Q, k') \ge L_{K,2} \Rightarrow d(X^i, Q) \ge L_{K,2} \tag{5.12}$$

Vector Quantization based Linear Search - Method A (VQLS-A)

In VQLS-A, we pre-compute the inter-VQ codevector distance matrix $\mathbb{D}$ (5.6) and store it in memory. We perform a full search on all the video signatures using $d_{VQ}(\overrightarrow{x_i}, \overrightarrow{q})$ (5.9) to find the top-K NN signatures - however, it directly looks up for a distance between two VQ symbols in the matrix $\mathbb{D}$ (e.g., $\mathbb{D}(t_k, n_{i,j})$ in (5.9)) and hence saves time (T_{11} in Table 5.2) by avoiding the L_1 distance computation. This method also uses PDP for speedup. *PDP in NLS implies searching along a lesser number of query frames. Here, it implies searching along a lesser number of non-zero query dimensions.* Sorting of the non-zero dimensions of the query signature results in improved PDP-based speedup. As in NLS, we maintain an ascending priority queue L.

$$\hat{d}_{VQ}(\overrightarrow{x_i}, \overrightarrow{q}, k') = \sum_{k=1}^{k'} q_{t_k^*} . \left\{ \min_{1 \le j \le N_{x_i}} \mathbb{D}(t_k^*, n_{i,j}) \right\} \tag{5.13}$$

where $q_{t_1^*} \geq q_{t_2^*} \cdots \geq q_{t_{N_q}^*}$ represents the sorted query signature. After considering the first k' non-zero (sorted in descending order) query dimensions, we discard the distance computation if $\hat{d}_{VQ}(\overrightarrow{x_i}, \overrightarrow{q}, k')$ (5.13) exceeds $L_{K,2}$.

The storage requirement for $\mathbb{D}$ is $O(U^2)$. Let the average number of non-zero dimensions in the VQ-based model signatures be F', where $F' = (\sum_{i=1}^{N} N_{x_i})/N$. We need to encode Q before search which incurs a time of $O(MU)$. Since this algorithm uses a constant time lookup of $O(1)$, the complexity of T_2 is reduced to $O(N_q F')$. The time T_3 to compute all the N model-to-query distances, without PDP, is $O(MU + N_q N F')$. Using PDP, the average number of non-zero query dimensions considered reduces to N'_q, where $N'_q < N_q$. The corresponding reduced value of T_3 is $O(MU + N'_q N F')$. The time needed to sort the query dimensions is $O(N_q \log N_q)$, which is small enough compared to $(MU + N'_q N F')$.

Vector Quantization based Linear Search - Method B (VQLS-B)

This method obtains higher speedup than VQLS-A by directly looking up the distance of a query signature symbol to its nearest symbol in a model video signature (e.g., $\{\min_{1 \leq j \leq N_{x_i}} \mathbb{D}(t_k, n_{i,j})\}$ in (5.9)). Thus, the computations involved in both T_{11} and T_{12} (Table 5.2) can be avoided, hence reducing the time to find a model-to-query distance to $O(N_q)$. We pre-compute a matrix $\mathbb{D}^* \in \mathbb{R}^{N \times U}$ where $\mathbb{D}^*(i,k)$ (5.15) denotes the minimum distance of a query vector, represented by

symbol k after the VQ encoding, to the i^{th} model.

$$d_{VQ}(\overrightarrow{x_i}, \overrightarrow{q}) = \sum_{k=1}^{N_q} q_{t_k}.\mathbb{D}^*(i, t_k), \text{ using (5.9)} \tag{5.14}$$

$$\text{where } \mathbb{D}^*(i, t_k) = \min_{1 \le j \le N_{x_i}} \mathbb{D}(t_k, n_{i,j}), 1 \le i \le N, 1 \le k \le N_q \tag{5.15}$$

VQLS-B differs from VQLS-A only in the faster distance computation using $\mathbb{D}^*$ instead of $\mathbb{D}$; the distance $\hat{d}_{VQ}(\overrightarrow{x_i}, \overrightarrow{q}, k')$ is now computed using (5.16) instead of (5.13).

$$\hat{d}_{VQ}(\overrightarrow{x_i}, \overrightarrow{q}, k') = \sum_{k=1}^{k'} q_{t_k^*}.\mathbb{D}^*(i, t_k^*) \tag{5.16}$$

There is an increase in the storage required for lookup - $\mathbb{D}^*$ needs storage of $O(NU)$ but the time T_3 to compute all the distances in $\mathcal{A}$, without PDP, is now reduced to $O(MU+NN_q)$. Using PDP, T_3 reduces to $O(MU+NN_q')$, as explained for VQLS-A in Section 5.5.2. Our experiments do confirm that this method has the lowest query time among all proposed methods (Table 5.8), the only disadvantage being that the storage cost (linear in N) may become prohibitively high for very large datasets.

Storage Reduction for VQLS Methods

For a large codebook size U, the storage cost for the distance matrix $\mathbb{D}$ can be significantly high. The solution is to perform a non-uniform scalar quantization (SQ) on the elements in $\mathbb{D}$. Suppose, we have used a SQ of codebook size U_1. In

that case, we just need to send the quantizer indices (each index needs $\lceil \log_2(U_1) \rceil$ bits) and maintain a table of the U_1 SQ centroids. Depending on the codebook size used, the memory savings can be substantial - without quantization, each element is a double needing 8 bytes = 64 bits. Our experiments have shown that we can do without very high resolution for the distance values and a 3-bit quantizer also works well in general. A low-bit scalar quantizer has also been used for the elements in $\mathbb{D}^*$, where the storage needed is $O(NU)$.

	VQLS-A	VQLS-B
T_3	$O(MU + N'_q N F')$	$O(MU + NN'_q)$
Storage	$U^2.b_{SQ,A}/2 + NF'b_{VQ} + 64.K\bar{F}p + 64.2^{b_{SQ,A}} + 64.18.2^{b_{VQ}}$	$NUb_{SQ,B} + 64.K\bar{F}p + 64.2^{b_{SQ,B}} + 64.18.2^{b_{VQ}}$

Table 5.3: Runtime needed to compute all the model-to-query distances (T_3) and storage (in bits) are compared for VQLS-A and VQLS-B.

We present a quick comparison of the two VQ-based search methods, VQLS-A and VQLS-B, in Table 5.3. The time complexity has already been explained while introducing the methods. Here, we elaborate on the storage complexity. For VQLS-A, the storage cost for $\mathbb{D}$ is $U^2.b_{SQ,A}/2$ where $2^{b_{SQ,A}}$ is the SQ codebook size used to encode the elements in $\mathbb{D}$. The SQ codebook is stored with a cost of $64.2^{b_{SQ,A}}$ bits. The storage cost for all the non-zero dimensions in the model video signatures is $NF'b_{VQ}$ where the CLD features are quantized using a VQ of size $2^{b_{VQ}}$. The storage size for the VQ that is used to encode the CLD features $= (64.2^{b_{VQ}}.18)$ bits $= 9.43$ MB (for $b_{VQ} = 13$). For VQLS-B, the storage cost for

$\mathbb{D}^*$ is $NUb_{SQ,B}$ where $2^{b_{SQ,B}}$ is the scalar quantizer size used to encode the NU members in $\mathbb{D}^*$. The storage cost for the unquantized signatures of the top-K model videos returned by the first pass is $64.K\bar{F}p$, where the video signatures are assumed to have $\bar{F}$ feature vectors on an average.

5.5.3 Search Algorithms with Dataset Pruning

The VQLS schemes described above *consider all the N model videos* to return the top-K NN videos. Further speedup is obtained by reducing the number of model videos accessed during the search. We present two dataset pruning methods for VQ-based signatures. The first method (VQ-M1) guarantees that the same top-K NN videos are returned even after pruning, as using naive linear search. The second method (VQ-M2) is *an approximation of the first and achieves much higher pruning*, though it is not guaranteed to return the correct top-K NN. The model-to-query distance (for the videos retained after pruning) can be computed using VQLS-A or VQLS-B (with PDP), for both VQ-M1 and VQ-M2.

Method VQ-M1

VQ-M1 uses a multi-pass approach for pruning. The logic is that *for a given query, the model videos which are nearest to it are likely to have some or all of the non-zero dimensions, as the query signature itself, as non-zero.*

The pre-computed information needed for VQ-M1 is listed below.

• We store a proximity matrix $\mathbb{P} \in \mathbb{R}^{U \times U}$ which stores the U nearest neighbors, in ascending order, for a certain VQ codevector, e.g., $\mathbb{P}(i,j)$ denotes the j^{th} NN for the i^{th} VQ codevector. For $U = 8192(2^{13})$, the storage cost of $\mathbb{P} = U^2.13$ bits (each of the U^2 terms represents an integer $\in [0, 2^{13}-1]$ and hence, is represented using 13 bits, giving a total storage cost of 109 MB).

• We also maintain a distance matrix $\mathbb{D}' \in \mathbb{R}^{U \times U}$ which stores the NN distances, in ascending order, for each VQ codevector. Here, $\mathbb{D}'(i,j)$ denotes the distance of the $\{\mathbb{P}(i,j)\}^{th}$ codevector from the i^{th} VQ codevector, i.e. $\mathbb{D}'(i,j) = \mathbb{D}(i, \mathbb{P}(i,j))$. We do not need to store $\mathbb{D}'$ explicitly as it can be computed using $\mathbb{D}$ and $\mathbb{P}$.

• We also store U clusters $\{\mathbb{C}(i)\}_{i=1}^{U}$, where $\mathbb{C}(i)$ denotes the cluster which contains those model video indices whose signatures have the i^{th} dimension as non-zero. The storage cost for 8192 clusters containing 38000 videos (the total model video dataset size for our experiments as mentioned in Section 5.6.1) is found to be equal to 6.3 MB.

$$\mathbb{C}(i) = \{j : x_{j,i} > 0,\ 1 \leq j \leq N\} \tag{5.17}$$

We now provide a list of symbols used in VQ-M1 (Algorithm 10) along with their definitions:

• $\mathbb{S}_j$: the set of distinct model videos considered in the j^{th} pass,

• G: the set of non-zero query dimensions, where $G = \{t_1, t_2, \cdots, t_{N_q}\}$,

• d_j^*: the minimum of the distances of all non-zero query dimensions to their j^{th} NN codevectors,

$$d_j^* = \min_{t_k \in G} \mathbb{D}'(t_k, j) \tag{5.18}$$

• A_j: the set of distinct VQ indices which are encountered on considering the first j NN for all the elements in G. Therefore, $(A_j \setminus A_{j-1})$ denotes the set of distinct (not seen in earlier passes) VQ indices encountered in the j^{th} pass, when we consider the j^{th} NN of the elements in G.

We maintain an ascending priority queue L of size K, for the K-NN videos, which is updated after every iteration. In the first iteration, we consider the union of the clusters which correspond to the non-zero query dimensions. We consider all the model videos from this union for distance computation. For the 1^{st} iteration, d_1^* equals 0 and the second iteration is almost always required. In the j^{th} iteration, we find the j-NN codevector of the non-zero query dimensions and the new codevectors (not seen in the earlier iterations) are noted. We obtain the new model videos which have common non-zero dimensions with these newly encountered dimensions and consider them for distance computation. For the j^{th} iteration, we terminate the search for top-K NN if $d_j^* \geq L_{K,2}$ (or if all the N model videos have already been considered). For a formal proof that we are assured of finding the correct top-K NN if $d_j^* \geq L_{K,2}$, see [1]. If the terminating condition

is satisfied at iteration $j=J$, the sequence of model videos considered is given by $\{\mathbb{S}_1, \mathbb{S}_2, \cdots, \mathbb{S}_{J-1}\}$.

We find that the maximum number of iterations (J) needed to obtain all the K-NN for a given query increases with both K and the fractional query length (ℓ), as shown in Table 5.4. For example, from Table 5.4, for K=10 and $\ell = 0.10$, the value of J is 500. Since we consider the j-NN for a VQ codevector at the j^{th} iteration, the number of NN that needs to be stored for each codevector equals the maximum number of iterations (J). Hence, the corresponding storage cost for $\mathbb{P}$ reduces to (500/8192).109 = 6.65 MB. We refer to this fraction (J/U) as $f(K, \ell)$ (a function of K and ℓ) when referring to the effective storage cost of $\mathbb{P}$, as used later in Table 5.8.

K	$J_{avg}(\ell=0.05)$	$J(\ell=0.05)$	$J_{avg}(\ell=0.10)$	$J(\ell=0.10)$	$J_{avg}(\ell=0.50)$	$J(\ell=0.50)$
10	2	450	3	500	31	1300
50	4	750	9	850	62	1600
100	8	950	17	1000	82	1750

Table 5.4: Average J_{avg} (averaging over all queries) and maximum number of iterations J for varying fractional query lengths (ℓ, whose value is shown in parentheses) and K, for $U = 8192$. Both J_{avg} and J increase with K and ℓ.

We compare the dataset pruning obtained using DBH [6], trained using our distance function, with that of VQ-M1 (Table 5.5) [2]. It is observed that the pruning obtained using VQ-M1 is significantly higher than that obtained using DBH.

[2]The DBH implementation is courtesy Michalis Potamias, a co-author in [6].

Algorithm 10 Algorithm for VQ-M1 - here, unique(E) returns the unique (without repeats) elements in E

In: N model video signatures, $\overrightarrow{x_i} \in \mathbb{R}^U,\ 1 \leq i \leq N$

In: the query signature $\vec{q}$, and lookup matrices $\mathbb{P}$ and $\mathbb{D}'$ (along with the lookup tables needed by the distance computation method VQLS-A/B)

Out: Best sequence to search N videos for top-K NN and also top-K NN (model video indices)

Initialization: (1^{st} pass)
$G = \{t_1, t_2, \cdots, t_{N_q}\}$, the non-zero query dimensions
$A_1 = G$, set of 1-NN of elements in G is G itself
$\mathbb{S}_1 = \bigcup_{1 \leq i \leq N_q} \mathbb{C}(t_i)$, set of model videos having at least 1 non-zero dimension from G
$d_1^* = \min_{t_k \in G}\{\mathbb{D}'(t_k, 1)\} = 0$
We maintain an ascending priority queue L of length K, based on the elements in $\mathbb{S}_1$, where $d_{VQ}(\overrightarrow{x_i}, \overrightarrow{q})$ is found using (5.9) or (5.14), depending on whether VQLS-A/B is being used.
End of 1^{st} pass
for $j = 2$ to U **do**
    $d_j^* = \min_{t_k \in G}\{\mathbb{D}'(t_k, j)\}$, minimum distance between non-zero query dimensions to their j^{th} NN
    if $L_{K,2} \leq d_j^*$ **or** $\sum_{k=1}^{j} |\mathbb{S}_k| = N$ (all model videos have been considered) **then**
        break;
    end if
    B_i=$\mathbb{P}(t_i, j)$, $1 \leq i \leq N_q$, B = set of VQ indices which are j^{th} NN of elements in G
    $E = B \setminus A_{j-1}$, E = unique(E), set of VQ indices that are j^{th} NN of elements in G and were not seen in earlier iterations
    $\mathbb{S}_j = \bigcup_{1 \leq i \leq |E|} \mathbb{C}(E_i)$
    $\mathbb{S}_j = \mathbb{S}_j \setminus \bigcup_{1 \leq i < j} \mathbb{S}_i$, set of all model videos not seen in earlier iterations and having at least one element in E as a non-zero dimension
    $A_j = A_{j-1} \cup E$, set of all VQ indices which belong to one of the top j-NN for elements in G
    Update the priority queue L based on the elements in $\mathbb{S}_j$
end for
return the sequences observed so far $\{\mathbb{S}_1, \mathbb{S}_2, \cdots, \mathbb{S}_{J-1}\}$ (assuming that the search terminates at iteration $j = J$) and top-K NN from the priority queue L

For DBH, the pruning obtained depends on the allowed error probability (p_{error}) - we report results for p_{error} of 0.05. As mentioned earlier, we are guaranteed ($p_{error}=0$) to return the top-K NN using VQ-M1.

ℓ	VQ-M1(K = 10)	VQ-M1(K = 50)	VQ-M1(K = 100)	DBH(K = 10)	DBH(K = 50)	DBH(K = 100)
0.05	15.03	20.52	23.90	71.85	78.37	81.97
0.10	21.22	27.23	30.83	73.64	80.93	82.13
0.50	42.04	48.21	51.51	78.16	81.40	83.24

Table 5.5: Comparison of the percentage of model videos retained after dataset pruning for VQ-M1 with that obtained using DBH, for different fractional query lengths (ℓ) and K. For DBH, $p_{error} = 0.05$ is used.

Method VQ-M2

Based on empirical observations, we assume that *the signature of the duplicate video, created from a subset of frames in the original video with noise attacks on the frames, will have common non-zero dimensions with the original model video signature.* Hence, the list of model videos considered for K-NN candidates corresponds to $\mathbb{S}_1$, the sequence of videos returned by the first iteration of VQ-M1. Thus, VQ-M2 is a single iteration process.

This method introduces errors only if there is no overlap between the non-zero dimensions of the query and the original model video, i.e. if the best matched video index $i^* \notin \mathbb{S}_1$. When the noise attacks introduce enough distortion in the feature vector space (*so that non-zero query dimensions may not overlap with*

the non-zero dimensions of the original model signature), a simple extension is to consider P NN ($P > 1$) across each non-zero query dimension. Thus, P should increase with the amount of distortion expected, and pruning gains decrease with increasing P. The number of videos in $\mathbb{S}_1$ and the storage cost for the proximity matrix $\mathbb{P}$ (defined for VQ-M1) depend on P. For $P > 1$, the storage cost for $\mathbb{P}$ is $O(UP)$.

The sequence $\mathbb{S}_1$, for $P \geq 1$, is obtained as follows (for VQ-M1, $\mathbb{S}_1$ corresponds to $P = 1$):

$$\mathbb{S}_1 = \bigcup_{1 \leq i \leq N_q} \mathbb{C}(t_i) \text{ using (5.17), for } P = 1$$

$$\mathbb{S}_1 = \bigcup_{j \in \mathcal{B}} \mathbb{C}(j) \text{ using (5.17), where } \mathcal{B} = \bigcup_{1 \leq i \leq N_q, 1 \leq k \leq P} \mathbb{P}(t_i, k), \text{ for } P \geq 1$$

In Figure 5.4, we compare the dataset pruning obtained for different choices of P, fractional query lengths, and using different number of keyframes for creating the query signatures. Using a higher fraction of query keyframes (M/T_Q), the pruning benefits are reduced, as more model videos are now considered due to the higher number of non-zero query dimensions. The percentage of videos retained after VQ-M2 based pruning is 3% and 7.5%, for 10% length queries, for $P = 1$ and $P = 3$, respectively. From Table 5.5, the corresponding pruning obtained using VQ-M1 varies from 21%-31% as K is varied from 10-100.

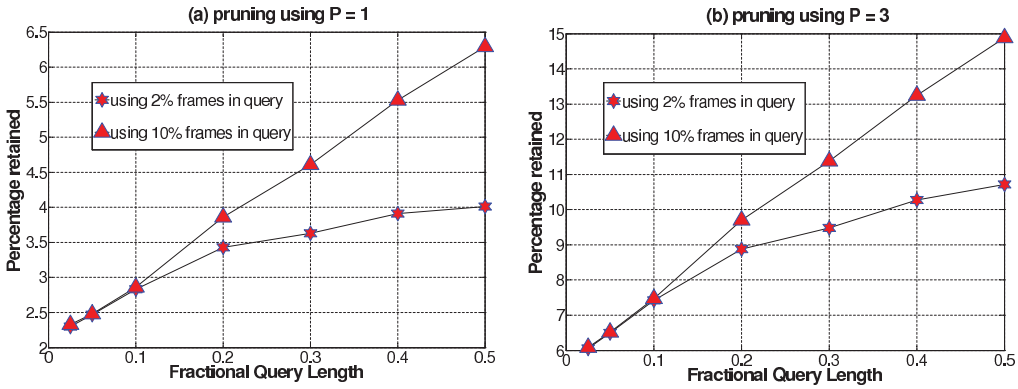


Figure 5.4: Comparison of the fraction of model videos retained after VQ-M2 based pruning, for varying fractional query lengths, and using different sized query signatures. The number of cluster centers for the query is fixed at 2% and 10% of the number of query frames, after temporal sub-sampling, i.e. $M/T_Q = 0.02$ and 0.10 (notations as in Figure 5.1) for the 2 cases. (a) Pruning using P=1. (b) Pruning using P=3.

For dataset pruning, we have presented two methods: VQ-M1 and VQ-M2. We present a quick overview of these methods through Table 5.6, where we compare their runtime and storage requirements. $T_{pr,1}$ and $T_{pr,2}$ refer to the time needed for dataset pruning for VQ-M1 and VQ-M2, respectively.

	T_{pr}	T_3	T_4	Storage
VQLS-A	0	$O(MU + N'_q NF')$	$O(N \log K)$	$U^2.b_{SQ,A}/2 + NF'b_{VQ} + 64.K\bar{F}p + 64.2^{b_{SQ,A}} + 64.18.2^{b_{VQ}}$
VQ-M1(A)	$T_{pr,1}$	$O(MU + N'_q N_{pr,1} F')$	$O(N_{pr,1} \log K)$	$U^2.13.f(K,\ell)$ + 6.3 MB (for $b_{VQ} = 13$) + storage(VQLS-A)
VQ-M2(A)	$T_{pr,2}$	$O(MU + N'_q N_{pr,2} F')$	$O(N_{pr,2} \log K)$	6.3 MB (for $b_{VQ} = 13$) + storage(VQLS-A)

Table 5.6: Comparison of query time (in terms of T_{pr}, T_3, and T_4) and storage (in bits) for VQ-M1 and VQ-M2.

For VQ-M1(A), the additional costs, over that of VQLS-A, needed for pruning are $U^2.13.f(K,\ell)$ (the cost for maintaining the proximity matrix $\mathbb{P}$) and 6.3 MB (the cost for maintaining the 8192 clusters). For VQ-M2(B), the proximity matrix $\mathbb{P}$ is not needed. The number of model videos retained after pruning are denoted by $N_{pr,1}$ and $N_{pr,2}$ for VQ-M1(A) and VQ-M2(A), respectively. This helps to reduce T_3 and T_4, which are defined in Table 5.2. The pruning obtained by VQ-M1 and VQ-M2 are determined at runtime depending on the query and we have numerically compared the pruning achieved using Figure 5.4 and Table 5.5. To reiterate, VQ-M1 is an iterative process (e.g., we need $J \geq 1$ iterations) while VQ-M2 is a one-pass process. Thus, in general, $T_{pr,1} > T_{pr,2}$ and $N_{pr,1} > N_{pr,2}$.

5.6 Experimental Setup and Results

Section 5.6.1 explains the dataset creation for duplicate detection. We have performed a variety of noise attacks and we empirically compare the duplicate detection accuracy over these attacks. Section 5.6.2 presents the comparison of the different speedup techniques proposed for improving the coarse search. Section 5.6.3 shows how our distance measure outperforms other histogram-based distances for VQ-based signatures.

5.6.1 Dataset Generation and Evaluation of Duplication Attacks

Two online video repositories www.metacafe.com and www.youtube.com are crawled to obtain a database of 38000 model videos, worth about 1600 hours of video content. A randomly chosen subset of 1200 videos ($\approx$ 50 hours of content), is used to generate the query videos. We perform various modifications on the decoded query frames for each of these 1200 videos to generate 18 duplicates per video. We empirically observe that the CLD feature is robust to the discussed modifications. The number of duplicates for each noise class is shown in parentheses.

- Gaussian blurring using a 3×3 and 5×5 window, (2)
- Resizing the image along each dimension by a factor of 75% and 50%, respectively, (2)
- Gamma correction by -20% and 20%, (2)
- Addition of AWGN (additive white Gaussian noise) using SNR of -20, 0, 10, 20, 30 and 40 dB, (6)
- JPEG compression at quality factors of 10, 30, 50, 70 and 90, (5)
- Cropping the frames to 90% of their size (1).

The frame drops that are considered can be random or bursty. We simulate the frame drops by creating a query video as a fraction (2.5%-50%) of the model video frames. The duplicate detection accuracy after the individual noise attacks is shown in Table 5.7.

Re-encoding Attacks: The downloaded videos are originally in Flash Video Player (FLV) format and they are converted to MPEG-1 format to generate the query video. We have also re-encoded the video using MPEG-2, MPEG-4, and Windows Media Video (WMV) formats. The CLD feature is robust against global attacks induced by strong AWGN and JPEG compression attacks and hence, robustness is expected against video re-encoding attacks - this is also experimentally verified. For MPEG-4 compressed videos, we experiment with varying frame rates (5, 10, 20, 30, 40 and 80 frames/sec) and the average detection accuracy is 99.25% - the results remain almost constant for different frame rates.

Color to Gray-scale Conversion: We have also converted the model video frames from color to gray-scale to create the query - here, the Y component is slightly modified. For gray-scale videos, we consider the first 6 dimensions of the CLD feature, which correspond to the DCT terms for the Y channel, as the effective signature. *The decision to use 6 or 18 dimensions is made based on whether dimensions 8-12 and 14-18 (AC DCT coefficients for Cb and Cr channels) are all zero*, i.e. it is a gray-scale frame. If frames of a different video are added

to the query video, then as the percentage of inserted frames (from other videos) increases, the detection accuracy decreases significantly as shown in Figure 5.5.

Logo and Caption Insertions: We have also experimented with logo and caption insertions. The initial logo considered is a 60×90 binary patch with 700 pixels (they constitute the logo pattern) being set to 1. We then resize the logo to 5%, 10%, 15% and 20% of the image size. We superimpose the logo pattern on the bottom leftmost part of the image and the image pixels, whose positions coincide with the 1's in the logo, are set to zero (black logo). For the caption insertion, the original block of text can be captured in a 50×850 binary patch where 2050 pixels (constituting the caption) are set to 1. We then resize the caption such that it can span a different number of columns (30%, 50%, 70% and 90% of the image size). The same principle is used to modify the image as in the logo insertion example. The coarseness of the CLD feature explains its relative robustness against logo and caption insertions. The averaging of the entire image to an 8×8 representation dilutes the effect of local changes.

5.6.2 Empirical Evaluation of Various Proposed Algorithms

We analyze the performance of the proposed algorithms for duplicate detection. The final detection accuracy, for a certain query length, is obtained by averaging

Attack	Error	Attack	Error
blur	0.0114	resize	0.0111
JPEG	0.0125	crop	0.0145
(blur+resize)	0.0119	(AWGN+crop)	0.0156
MPEG-2	0.0098	MPEG-4	0.0088
logo (5%)	0.0140	logo (10%)	0.1780
caption (30%)	0.0155	caption (50%)	0.0190
Attack	Error	Attack	Error
gamma	0.0221	AWGN	0.0113
(blur + crop)	0.0154	(resize + crop)	0.0148
(gamma + crop)	0.0243	(AWGN + resize)	0.0128
WMV	0.0076	gray-scale	0.0388
logo (15%)	0.0198	logo (20%)	0.0228
caption (70%)	0.02301	caption (90%)	0.0288

Table 5.7: Detection error obtained using CLD features, for individual noise attacks, averaged over fractional query lengths from 2.5%-50%, and over varying parameters for a given attack, are shown.

over all the (1200×18) noisy queries, where the 18 duplication schemes were introduced in Section 5.6.1.

• Firstly, we show the speedup obtained using PDP, by comparing PLS (NLS + PDP) with NLS, and comparing VQLS-A and VQLS-B schemes, with and without PDP (Figure 5.6). It is also seen that the VQ-based schemes significantly outperform NLS and PLS, that use un-quantized features.

• Secondly, we show the performance improvements obtained using VQ-M1(A) and VQ-M2(A), in place of VQLS-A, and using VQ-M2(B) in place of VQLS-B - these methods achieve additional speedup through dataset pruning (Figure 5.7(a) and 5.7(b)).

Speedup Obtained Using PDP: We show the runtime needed (T_3+T_4 from Table 5.2), with and without PDP for NLS, VQLS-A and VQLS-B schemes, in Figure 5.6(b-1), (b-2) and (b-3), respectively, to return the top-K model videos. T_3 is reduced by using PDP. $T_4 = O(N \log K)$ increases with K and thus, the effective runtime saving decreases as K increases. PDP provides significant runtime saving so that "with pruning: $K = 100$" takes lesser time than "without pruning: $K = 10$". Also, comparing (b-2) and (b-3) with (b-1) in Figure 5.6, we observe that the runtime needed by VQLS-A and VQLS-B (with PDP) is much lower than that for PLS and NLS.

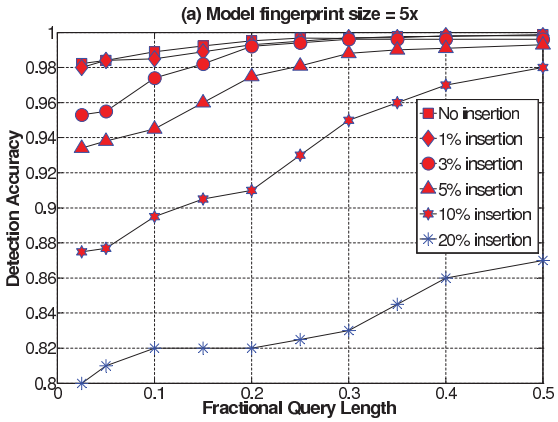


Figure 5.5: Variation of the detection accuracy with varying levels of video clip (from a different video) insertion - a fractional query length of 0.1 means that the query consists of 10% frames present in the (original query + inserted video clip). Model fingerprint size = 5x.

Speedup Obtained through Dataset Pruning: We observe the runtime saving obtained through dataset pruning (using VQ-M1 and VQ-M2) using VQLS-

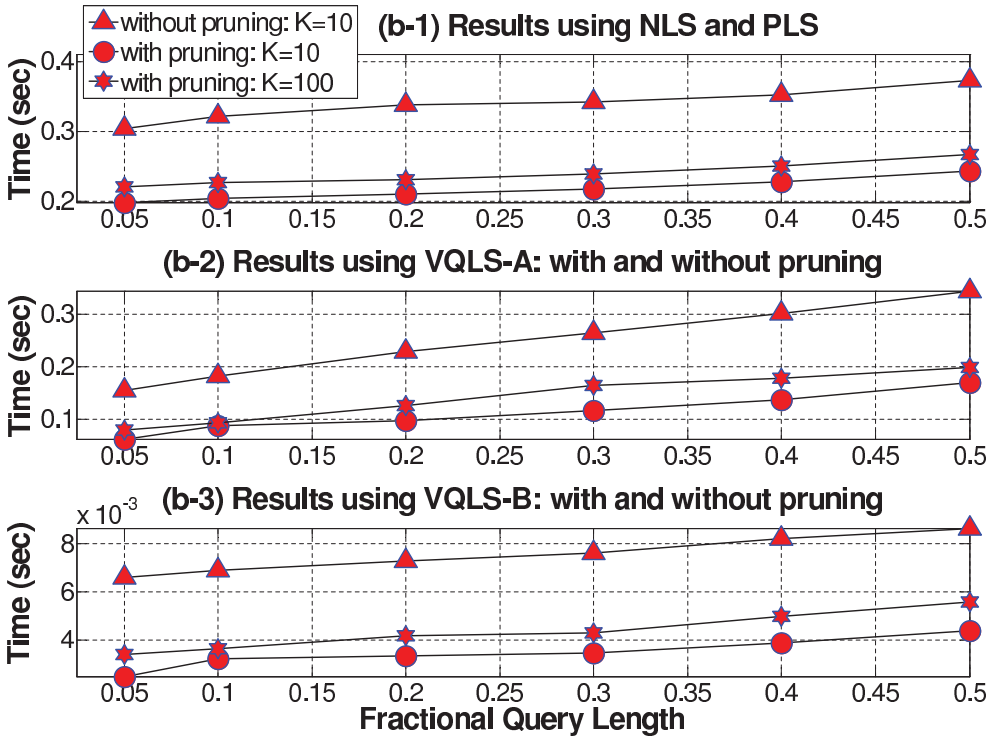


Figure 5.6: Runtime improvements due to PDP are shown for the PLS and VQ-based linear search schemes: (b-1) results using NLS and PLS; (b-2) results using VQLS-A - with and without pruning; and (b-3) results using VQLS-B - with and without pruning. "Pruning/ no pruning" indicates whether or not PDP has been used. Here, runtime $= (T_3 + T_4)$ is the time needed to return the top-K model videos after the first pass.

A and VQLS-B for the model-to-query distance computation, in Figure 5.7(a) and 5.7(b), respectively. PDP is employed for all the methods and "prune/no prune" denotes whether or not we employ dataset pruning methods (VQ-M1 or VQ-M2).

• For VQLS-A, the runtime comparison for the different methods is: VQLS-A > VQ-M1(A) > VQ-M2(A). Hence, using dataset pruning results in significant speedup (Figure 5.7(a)).

• For VQLS-B, the use of the lookup table D^* reduces runtime significantly, so that the time required for the iterative pruning technique (VQ-M1) is higher than the runtime without pruning, especially for higher values of K and longer queries. Hence, for VQLS-B, for fractional query lengths exceeding 0.10, the runtime comparison for the various methods is: VQ-M1(B) > VQLS-B > VQ-M2(B) (Figure 5.7(b)).

Storage and Time Comparison: We present the variation of the detection accuracy with query time, along with the associated storage costs, for the various methods in Table 5.8. The query length (ℓ) considered is 10% of the actual model video lengths. *It is seen that among methods with higher storage costs (using $\mathbb{D}^*$, where storage $\propto N$), VQ-M2(B) has the minimum query time while for methods with lower storage costs (using $\mathbb{D}$, where storage $\propto U^2$), VQ-M2(A) has the minimum query time.* The various values used in Table 5.8 are $\bar{F} = 25$, $F' = 18$, $b_{VQ} = 13$, $b_{SQ,A} = 3$, $b_{SQ,B} = 3$, $N = 38,000$ (dataset size) and $U = 8,192$ (VQ size).

5.6.3 Comparison of Other Histogram based Distances for VQ-based Signatures

We compare our distance measure between the VQ-based signatures with the L_1 distance (d_{L1}), an intersection based distance (d_{int}), the cosine distance (d_{cos})

Index	Method	K	Storage (bits)	Storage (MB)	$\ell = 0.10$	
					Query Time(s)	Accuracy
1	NLS	10	$64.N\bar{F}p$	133.47	0.42	0.989
	NLS	50			0.43	0.994
2	PLS	10	$64.N\bar{F}p$	133.47	0.27	0.989
	PLS	50			0.28	0.994
3	VQLS-A	10	$64.18.2^{b_{VQ}}$ (9.43 MB) +	22.91	0.096	0.883
	VQLS-A	50	$U^2.b_{SQ,A}/2 + NF'b_{VQ}$	23.06	0.102	0.958
	VQLS-A	**100**	$+64.K\bar{F}p + 64.2^{b_{SQ,A}}$	**23.24**	**0.109**	**0.969**
4	VQ-M1(A)	10	$U^2.13.f(K,\ell)$ + 6.3 MB +	35.86, 46.51	0.048	0.883
	VQ-M1(A)	50	$U^2.b_{SQ,A}/2 + NF'b_{VQ}+$	40.67, 50.65	0.065	0.958
	VQ-M1(A)	**100**	$64.K\bar{F}p + 64.2^{b_{SQ,A}}$ + 9.43 MB	**42.85, 52.83**	**0.076**	**0.969**
5	VQ-M2(A)	10	9.43 MB + cluster cost (6.3 MB)	29.21	0.014	0.883
	VQ-M2(A)	50	$+U^2.b_{SQ,A}/2 + NF'b_{VQ}$	29.36	0.020	0.958
	VQ-M2(A)	**100**	$+64.K\bar{F}p + 64.2^{b_{SQ,A}}$	**29.54**	**0.024**	**0.969**
6	VQLS-B	10	9.43 MB +	123.36	0.012	0.883
	VQLS-B	50	$NUb_{SQ,B} + 64.K\bar{F}p$	123.51	0.015	0.958
	VQLS-B	**100**	$+64.2^{b_{SQ,B}}$	**123.69**	**0.019**	**0.969**
7	VQ-M2(B)	10	9.43 MB + cluster cost (6.3 MB)	129.66	0.010	0.883
	VQ-M2(B)	50	$+NUb_{SQ,B} + 64.K\bar{F}p$	130.26	0.013	0.958
	VQ-M2(B)	**100**	$+64.2^{b_{SQ,B}}$	**130.99**	**0.016**	**0.969**

Table 5.8: Comparison of all the 3 parameters - detection accuracy, query time (expressed in seconds), and storage, for the different methods, at varying K. Query time equals $(T_3 + T_4 + T_5)$ (along with the time for k-means-clustering to obtain Q from Q_{orig} and the time for sorting the query dimensions). Unless otherwise mentioned, the elements are stored in "double" format (=64 bits). The storage cost of VQ-M1(A) depends on the fractional query length (ℓ): thus, for $K = 10$, the storage cost equals 35.86 MB for $\ell = 0.10$.

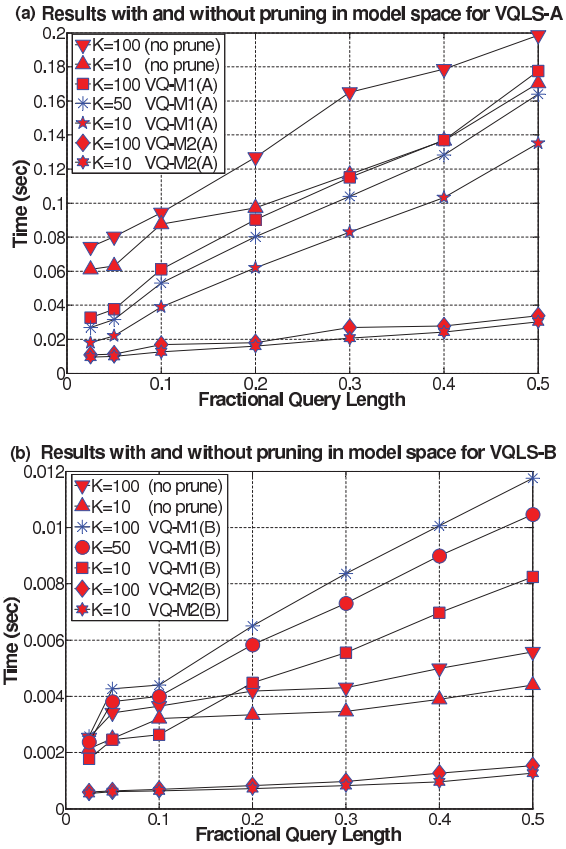


Figure 5.7: Runtime improvements due to pruning in the model video space, for VQLS-A and VQLS-B, are shown. By "*no prune*", we mean that *pruning in model video space (VQ-M1 or VQ-M2) is absent, while PDP is used for all the methods.* Significant runtime savings are obtained for VQ-M1(A) and VQ-M2(A) over VQLS-A (Figure a) and for VQ-M2(B) over VQLS-B (Figure b).(a) Results with and without dataset pruning for VQLS-A. (b) Results with and without dataset pruning for VQLS-B.

and the Jaccard coefficient based distance (d_{Jac}), which was used for copy detection in [25]. The different distance measures are defined here:

$$d_{int}(\overrightarrow{x_i}, \overrightarrow{q}) = 1 - \sum_{k=1}^{U} \min(x_{i,k}, q_k), \quad d_{cos}(\overrightarrow{x_i}, \overrightarrow{q}) = (\sum_{j=1}^{U} x_{i,j} q_j)/(||\overrightarrow{x_i}||_2 . ||\overrightarrow{q}||_2)$$

$$\text{Jaccard coefficient } J_{coeff} = \sum_{k=1}^{U} \frac{\min(x_{i,k}, q_k)}{\max(x_{i,k}, q_k)}, \text{ and } d_{Jac} = -J_{coeff}$$

The performance comparison of the different distance measures (Figure 5.8) shows that the detection accuracy using d_{VQ} is significantly higher than the other distances, especially for small query lengths. *For our proposed measure, the effective distance is the sum of distances between "query vector to best matching vector in model signature".* For traditional histogram-based distances, the effective distance is computed between corresponding bins in the model and query signatures - this distance is small only when the query signature is similar to the entire model signature, which is true mainly for longer queries. Hence, *the advantage of using our asymmetric distance is more obvious for shorter query lengths.*

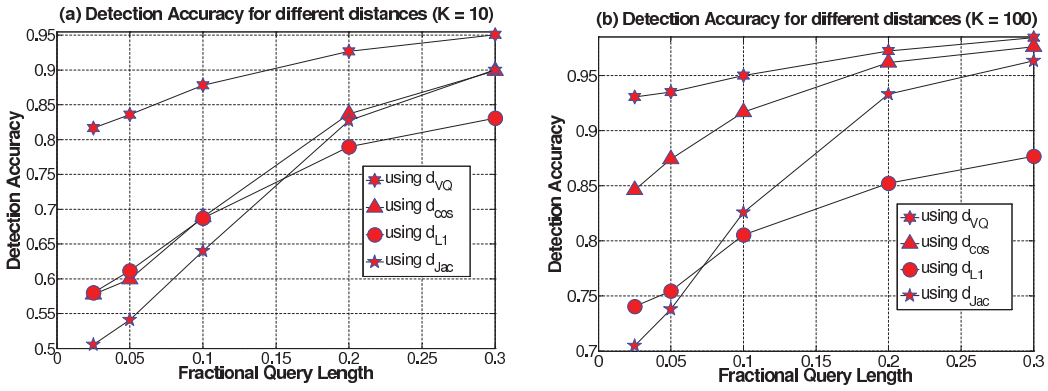


Figure 5.8: Comparison of the detection accuracy obtained using the different VQ based distances, for $K = 10$ and $K = 100$, is shown. Results using d_{int} and d_{L1} are near-identical and so, only d_{L1} based results are shown. Results using d_{VQ} are significantly better than that using d_{cos} (which in turn performs better than d_{L1} and d_{Jac}) at smaller query lengths. (a) Detection results for various distances (k=10). (b) Detection results for various distances (k=100).

5.7 Duplicate Confirmation

After finding the best matched video V_{i*}, we discuss a distance threshold based (Section 5.7.1) and a registration-based (Section 5.7.2) approach to confirm whether the query is a duplicate derived from V_{i*}.

5.7.1 Distance Threshold based Approach

The training phase to obtain the distance threshold involves finding the 1-NN and 2-NN distances for 1200 query videos, over various noise conditions and query lengths. The distance between X^{i*}, the fingerprint of the 1-NN video V_{i*}, and the larger query signature Q_{orig}, is computed using (5.1) and is normalized by the query length T_Q, so as to make the threshold independent of the query length. Thus, the effective 1-NN distance equals $\{d(X^{i*}, Q_{orig})/T_Q\}$. Since the same 1200 videos were considered as the model videos, the 1-NN always refers to a duplicate video and the 2-NN to a non-duplicate one. Ideally, the threshold δ_s should be such that all the 1-NN (or 2-NN) distances are less (or greater) than it. By equally weighing the probability of false alarm P_{FA} (wrongly classifying the 2-NN retrieval as a duplicate) and missed detection P_{MD} (failing to classify the 1-NN retrieval as a duplicate), the threshold δ_s is empirically set at 230 - distribution of 1-NN and 2-NN distances and illustrative explanation of threshold selection are shown

in [1]. The corresponding P_{FA} and P_{MD} values equal 0.07. Depending on whether the emphasis is on minimizing P_{FA} or P_{MD}, δ_s can be decreased or increased, accordingly.

For verifying the effectiveness of the distance threshold, we repeat the duplicate detection experiments *on an unseen dataset of 1700 videos ($\approx$ 75 hours of video), all of which are different from the model videos.* For each video, 18 duplicates are created as in Section 5.6.1. Using a threshold δ_s of 230, 3% of the videos were classified as "duplicates" - for them, the 1-NN distance is less than δ_s.

For those cases where the query-to-model distance is very close to the threshold δ_s, we use a registration-based approach (Section 5.7.2). The registration method is computationally intensive but is more accurate in determining if the query is indeed a duplicate of the retrieved candidate.

5.7.2 Registration based Approach

In this approach, we need to know which model keyframe should be considered for registration for a given query keyframe. While computing the distance $d(X^{i^*}, Q_{orig})$ in the second pass of the search process, we have already obtained the best matching vector in the model signature ($X^{i^*} \in \mathbb{R}^{F_{i^*} \times p}$) for every query vector in Q_{orig}. What we now need is a way to map every model (query) vector to its corresponding keyframe. This is done as follows. Considering the cluster

centers (X^{i^*}) obtained after k-means clustering on the feature matrix (Z^{i^*}), we can find which vector in Z^{i^*} best matches to a certain vector in X^{i^*} - the frames corresponding to the selected vectors in Z^{i^*} constitute the model keyframes.

Registration method: First, a set of salient points is detected in the respective frames. Then the SIFT feature descriptor is computed locally around those points followed by establishing correspondences between them by computing the distance in the SIFT feature space. As this usually yields a lot of false matches (more than 50% in some scenarios), RANSAC [42] is included in this framework to filter out the bad point correspondences and to get a robust estimate of homography parameters. Finally, we conclude that the query video is a duplicate of V_{i^*} if majority of the query frames (approximately 70% in our case) can indeed be registered with the best matching keyframes in V_{i^*}. This fraction (70%) can be increased or decreased depending on whether the emphasis is on minimizing P_{FA} or P_{MD}.

5.8 Discussion

Here we have addressed the duplicate video detection problem. We empirically selected CLD for fingerprinting as it was robust to the duplication attacks. However, if there is extensive cropping, padding, or rotation/shear, salient point-

based descriptors can be more effective. We developed a new non-metric distance measure which is very effective for short queries. This distance measure has high computational complexity as it computes the distances between all model-to-query keyframe pairs. We reduce the computational cost using pre-computed information, partial distance based pruning and dataset pruning. This distance measure can be explored in other domains which require subset matching. The proposed dataset pruning method has been effective for our distance function and VQ histogram based signatures. It would be interesting to study how well the pruning method generalizes for histogram-based distances.

5.9 Conclusions

The problem of *fast and real-time duplicate detection* in a large video database is investigated through a suite of efficient algorithms. We retrieve the duplicate video for about a minute long query in 0.03 sec with an average detection accuracy of over 97%. Our proposed distance measure is shown to perform very well when the query is a noisy subset of a model video and keyframe-based signatures are used. In the future, we will explore how the duplicate detection system scales to larger sized datasets.

In our problem, we have assumed that the query is entirely constituted from a model video. If, however, a query contains portions of multiple videos, the same asymmetric distance will not be effective. In that scenario, one can consider disjoint windows (of suitable length) of the query video and issue multiple queries. The aim is to identify the model to which a certain query window can be associated. This topic will be explored in future.

Chapter 6

Efficient Computation of Statistical Significance of Query Results in Databases

Queries such as database similarity searches return results satisfying certain properties of distances or scores. However, for domain scientists, the absolute values of scores are seldom sufficient. Statistical significance or *p-value* of the result is a more useful criterion. Further, most database systems support queries that have multiple attributes or objects. The score of the result is an aggregate of the individual scores. The simple way of calculating the p-value by enumerating all random possibilities fails for large database and query sizes. We propose an efficient method to calculate the approximate p-value of a multi-attribute result when the distribution of scores for the database objects is non-parametric. Experimental evaluation on large databases shows that our method is practical, runs 5

orders of magnitude faster than the basic approach, and has an error of less than 5% in p-value computation.

6.1 Motivation and Problem Statement

Many database systems retrieve results based on some distance or score measure between the query object and the database objects. Score is a quantitative measure of the similarity between objects based on multiple attributes. It has been widely used for ranking results in content-based multimedia retrieval systems. However, with the growing interest in analyzing the results of a database similarity query, computing rigorous statistical properties of the results is more meaningful.

Statistical significance helps the domain scientists in understanding the nature of the query and the statistical properties of the database objects. The most well known example is BLAST [3]. A standard measure of statistical significance is the *p-value*. The p-value of score s of a query result from a database is defined as the probability of randomly obtaining a result from the database with a score s or higher for the same query. It is the area under the probability distribution function (pdf) of the scores of random objects greater than s.

For a database management system (DBMS) serving single object queries, the score *pdf* can be characterized or calculated, and so, the p-value can be computed. However, there are database systems of complex objects where each object consist of multiple attributes or components. Such systems support queries with multiple attributes or objects and the score of a result is some aggregate function (e.g., sum) of the individual scores of each query component against its corresponding result component [40]. These queries are common for region based image retrieval (RBIR) systems [32] and information retrieval systems [109]. For example, in an RBIR system, a query region is composed of a number of sub-regions (e.g., tiles) [30, 117]. The database images are also split into sub-regions. Each component sub-region has a corresponding score of its match with a query sub-region. The score of a result is the sum of the individual scores.

For a given query object Q of size r, a random database for computing the p-value can be modeled by considering all possible aggregates of size r composed of components from the database. To find the p-value, we need to calculate the score pdf for this random database. This simple method has a running time that grows exponentially with database size and query size and is, therefore, impractical. In this chapter, we propose and solve the following problem: *"Given a query Q composed of r objects $Q_i, i = 1, \cdots, r$, database objects $D_j, j = 1, \cdots, n$, scoring functions $f_i : Q_i \times D \rightarrow \Re$, compute the p-value of obtaining a score s for a result*

$R = \cup_{i=1}^{r} R_i$, *where* $s = \sum_{i=1}^{r} f(Q_i, R_i)$, *for a random database of objects, each having* r *component objects."*

Methods have been proposed for obtaining a single measure of statistical significance by combining the individual p-values. For example, the method in [34] requires finding the correlation among the attributes, which is done by sampling for large datasets. We adopt a more direct approach. We find the sum pdf of the individual pdfs of the components of the query. Then we calculate the p-value from this sum score pdf. Since score pdf of each component is independent of the other, this pdf is the *convolution* of all the individual pdfs. For most databases, the nature and the parameters of this pdf cannot be computed. We consider such cases where the probability distribution function of the cumulative scores is non-parametric.

6.2 Algorithm

For a multiple object query, the p-value can be found from the sum pdf of its components. The basic approach of calculating the sum pdf is to calculate the pdf of each query component and then find their convolution. Two score pdfs h_1 and h_2 can be convoluted to produce the sum score pdf h: the probability corresponding to score s considers all possible scores s_1 and s_2 from h_1 and h_2

Algorithm PRUNE
Input: Query $Q = \cup_{i=1}^{r} Q_i$, Score s, Database D, Number of bins b
Output: P-value p
1. **for** $i = 1$ to r
2. $\quad D_i :=$ 1-NN(Q_i, D)
3. $\quad h_i :=$ BinHistogram(D_i, b)
4. **end for**
/* σ_i is the sum pdf of bin histograms $1, \cdots, i$ */
5. $\sigma_1 := h_1$
6. **for** $i = 2$ to r
7. $\quad B(\sigma_i) := s - \sum_{j=i+1}^{r} \max(h_j)$
8. $\quad B(h_i) := B(\sigma_i) - \max(\sigma_{i-1})$
9. $\quad B(\sigma_{i-1}) := B(\sigma_i) - \max(h_i)$
10. $\quad \sigma_i :=$ Convolute(all bins $\sigma_{i-1,j} \geq B(\sigma_{i-1})$, all bins $h_{i,k} \geq B(h_i)$)
11. **end for**
12. $p :=$ Sum of probabilities in all bins $\sigma_{r,j} \geq s$

Figure 6.1: The PRUNE algorithm.

such that $s = s_1 + s_2$. The cost of computing this convolution is, thus, *quadratic* in the number of distinct scores in the constituent pdfs. Hence, we can see that the convolution of multiple pdfs incurs a multiplicative cost on the size of the pdfs, and therefore, can be large. Assume that a query has r components, and each component has b distinct score values. The convolution of the first two components requires $b \times b = b^2$ operations and produces up to b^2 distinct scores. Convoluting this result with the third component requires $b^2 \times b = b^3$ operations, and so on. The total running time, therefore, is $b \times b \times \cdots \times b = O(b^r)$.

In order to speed up the p-value computation, we consider the two aspects of the problem—computing the score distribution for each query component and convoluting the distributions—separately. The first sub-problem is handled by pre-processing and maintaining a separate score pdf for each object component in the database. This can be done offline. For each component of the query, we approximate its score pdf by the pdf corresponding to its nearest component in the database. The nearest database component can be retrieved very efficiently by indexing the feature vectors of the objects using R-trees [52].

To efficiently convolute the pdfs and compute the p-value, we developed an approximation technique PRUNE (Figure 6.1). There are three main steps in the algorithm: (i) Use *histograms* to approximate the score probability distribution functions of each query object, (ii) Progressively *cascade* the convolution of query object histograms to obtain the score histogram for the entire query, and (iii) Use *bounds* to convolute the histograms. We next explain each step in detail.

6.2.1 Use of Histograms to Approximate Distributions

Since the cost of convoluting two pdfs is a quadratic function of the number of distinct values in the pdfs, instead of using an actual score pdf, we approximate it by a histogram with a fixed number of bins as shown in step 3 of Algorithm PRUNE (Figure 6.1). For speed, simplicity, and convenience, we choose equi-

width histograms. The whole score range is divided into a fixed number of equi-width bins. The accuracy of the approximation depends on the number of bins maintained. More bins have less error, but higher running time. Section 6.3 considers the effect of the number of bins on the running time and the error in calculating the p-value.

6.2.2 Cascaded Convolution of Histograms

As described earlier, the simple way of directly convoluting r histograms has a time complexity which is exponential in r. To avoid such high costs, we convolute the histograms in a progressive fashion. Initially, the histograms of two query component objects are convoluted to yield another score histogram, which is again binned into b bins. Then, this histogram is convoluted with the next histogram and so on till all the r histograms have been convoluted.

Denoting the i^{th} histogram by h_i and the convolution of histograms up to i components by σ_i, we compute $\sigma_i = \sigma_{i-1} \oplus h_i$ up to $i = r$. Each histogram convolution requires *quadratic* number of operations in terms of the number of bins in the histograms. The total time complexity, therefore, is $O(b^2 r)$. To make it even more efficient, we apply a bounding procedure which is described next.

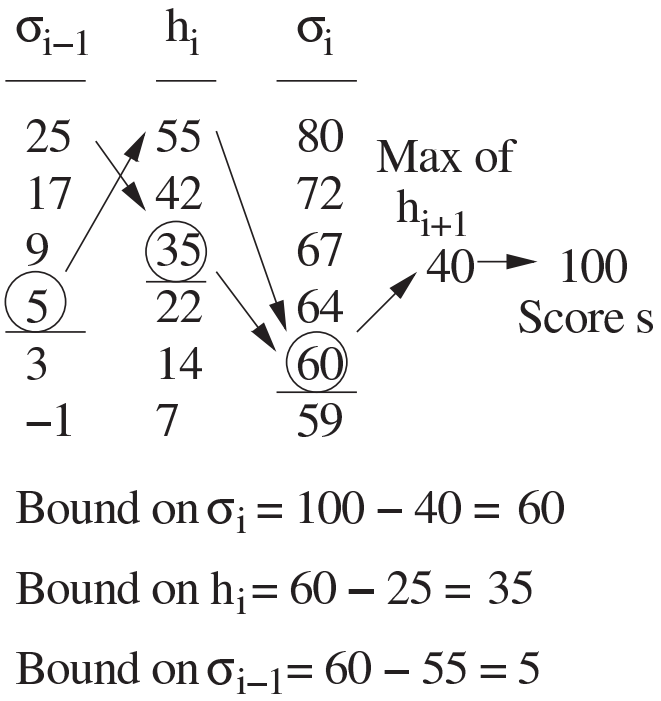


Figure 6.2: Efficient convolution of histograms. $\sigma_{i-1} \oplus h_i = \sigma_i$. The bins below the score thresholds (shown inside circles) can be pruned to save time.

6.2.3 Convolution of Bounded Histograms

The bounding method is based on the observation that computing the p-value for a score s requires counting only those scores that are greater than or equal to s. Scores in the histogram of a query object that cannot add up to s even when combined with the best scores of the histograms of other query objects need not be considered. Therefore, the bins in the histogram whose scores fall below this *threshold score* can be deleted. The bounding method achieves this pruning of

histogram bins by evaluating the threshold score at each stage. This reduces the number of bins, and thus, the running time.

Figure 6.2 shows an example of how such thresholds are computed. Assume that the histogram σ_{i-1} is convoluted with h_i to yield σ_i. Also, assume that the score s for which the p-value is being calculated is 100. If the maximum score in h_{i+1} is 40, then any score below $100 - 40 = 60$ in σ_i cannot add up to s. This is the threshold score for that histogram, and is highlighted in the figure. Thus, all scores below 60 can be deleted from σ_i. By analyzing this bounding behavior backwards for the histograms σ_{i-1} and h_i, it can be seen that such contributing pairs of scores need not be calculated at all. The maximum score in h_i is 55. Since we do not need any score in σ_i that is below 60, all scores below $60-55 = 5$ in σ_{i-1}, when added to any score in h_i will be less than 60, and hence, can be deleted. Continuing this reasoning, all scores below 35 in h_i can be deleted. The threshold scores are highlighted in the figure.

In this example, the number of bins in σ_{i-1} and h_i are reduced from 6 to 4 and 3 respectively. This translates to a saving of $6 \times 6 - 4 \times 3 = 24$ bin convolution operations. Steps 7 to 10 of Algorithm PRUNE (Figure 6.1) apply bounding to the cascaded convolution. As shown in the next section, the overall saving for r histogram convolutions is significant.

Note that the two sources of error in the p-value computation are the use of nearest neighbors and histogram binning. The bounding method does not introduce any error.

6.3 Experiments

In this section, we demonstrate the effectiveness of our PRUNE method over alternate approaches. We explain the empirical results in the context of region-based image retrieval (RBIR) system for a biomedical image database of fluorescent micrographs of feline retinas labeled with different antibodies [43]. The dataset consists of 805,272 tiles. The score between two tiles is a decreasing function of the L_1 distance between the color histogram features of the tiles. The tiles are the component objects in our system. The score of the alignment of a query region to a database region is the sum of the scores of the alignment of the individual tiles. The details of the dataset preparation, the features, the scoring function, and the retrieval system are explained in [117].

6.3.1 Running Time

The basic approach of online computation of score pdfs of each query tile and their convolution yields impractical time. Therefore, we do not consider

it. Instead, we maintain a database of the pre-computed pdf of each database component. We use the following parameters for the analysis of running time: (i) the number of bins in the score histograms, (ii) the query score for which the p-value is computed, measured as a percentage of the maximum score that can be achieved by the query, and (iii) the query size, which is the number of tiles in the query image.

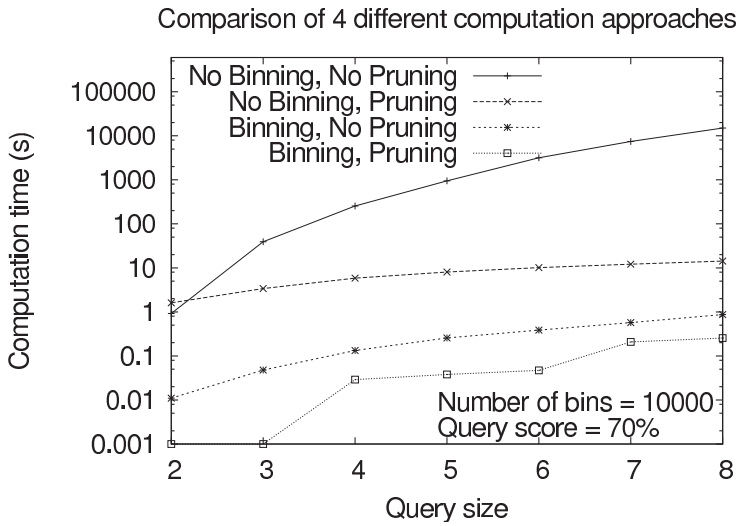


Figure 6.3: Comparison of the various approaches of p-value computation.

First, we compare the four different approaches of computing the p-value: (i) Using actual pdf without pruning, (ii) Using actual pdf with pruning, (iii) Using binned pdf without pruning, and (iv) Using binned pdf with pruning (PRUNE). Figure 6.3 shows their running times for different query sizes. The pruning strategy shows a gain of about 10^3 for a query size of 8 without binning. Binning

improves the computation time by 2 orders of magnitude with pruning and 5 orders of magnitude without pruning. In all cases, the PRUNE strategy finished in practical times—at most 255 ms.

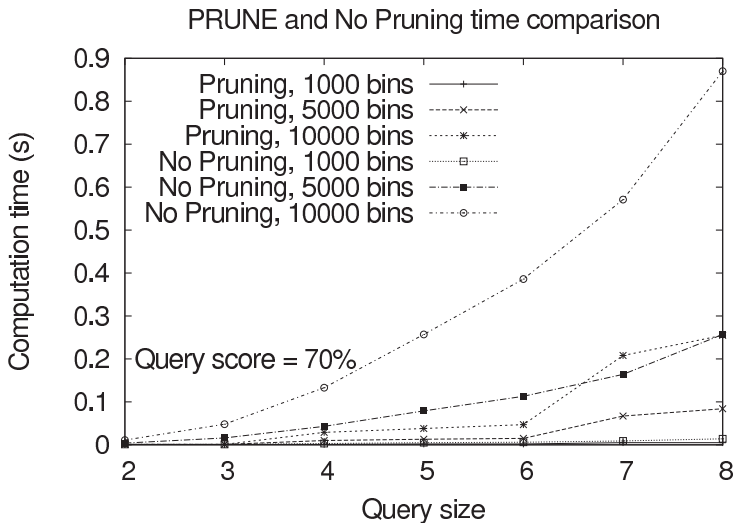


Figure 6.4: The effect of pruning on the running time of p-value computation.

Since the PRUNE strategy outperforms all other approaches we analyze it further with respect to other parameters. Figure 6.4 shows that the efficiency of pruning increases with the increase in query size across varying number of bins in the histogram. Up to medium query sizes of 6, and number of bins 5000, the scalability is linear or better.

The next experiment (Figure 6.5) shows that the pruning strategy performs better when the query score increases, across varying number of bins. When the query score is 80% of the maximum score, the pruning strategy is very effective

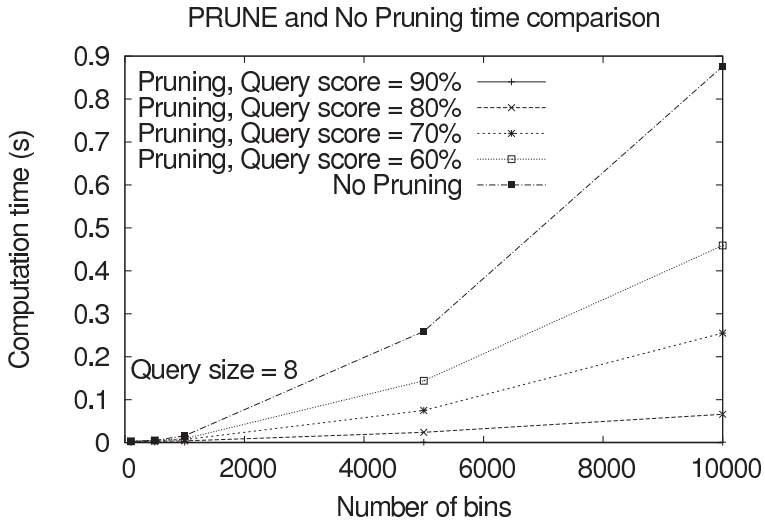


Figure 6.5: The effect of query score and number of bins on the running time of p-value computation.

for all histogram bin sizes. The scalability is better for higher query scores. Thus, the empirical results strongly suggest that our PRUNE method is efficient and practical.

6.3.2 Error

We next performed experiments to measure the error in p-value computation induced by binning. Figure 6.6 shows the error percentage across varying number of bins. When the query size is large, using less number of bins accumulates the error over more number of steps, resulting in more than 20% error. Increasing the

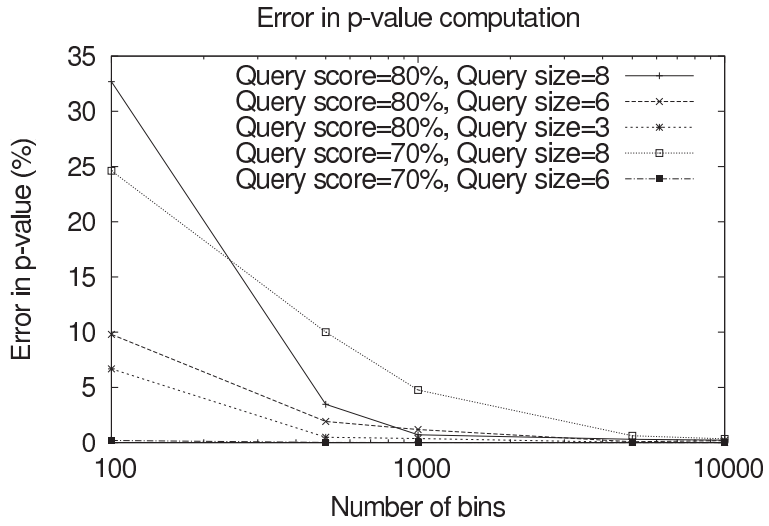


Figure 6.6: The percentage error in p-value computation due to binning.

number of bins to 1000 reduces the error to at most 5%, irrespective of the query size and the query score. This proves the effectiveness of our strategy.

6.4 Conclusions

In this chapter, we defined the problem of efficiently computing the p-value for multi-object query results for non-parametric distributions. We proposed an approximate bounding procedure PRUNE and showed that it is faster than the alternate approaches by more than 5 orders of magnitude with the error in computation less than 5%. Possible future avenues of work include sampling to obtain the score histograms, computing bounds for other aggregate functions like max,

and examining the order in which the component object histograms should be convoluted in order to minimize the number of operations.

Chapter 7

Conclusions

In this thesis, it was shown that patterns can be queried accurately and efficiently in high-dimensional heterogenous datasets. Pattern queries were studied in three types of datasets: (1) datasets of keyword-tagged objects; (2) datasets of objects having spatial relationships; and (3) datasets of objects having temporal relationships. Novel index structures and algorithms were presented to answer these pattern queries. Multiple real datasets of sizes up to 100 million and dimensions up to 256 were used for empirical evaluations. These evaluations showed that the proposed algorithms are highly scalable and accurate.

A novel technique, SIMP, was described in Chapter 2 for accurately and efficiently finding r-near neighbors in high dimensional spaces. Comparative studies on three real datasets of dimensions between 32 and 256 and sizes up to 10 million showed a superior performance of SIMP over the state-of-the-art methods. Empirical studies on real datasets of sizes up to 100 million points and dimensions

up to 256 showed that SIMP scales linearly with the query range, the dataset dimension, and the dataset size.

Querying patterns by keywords in a dataset of keyword-tagged objects was introduced in Chapter 3. The proposed algorithm, ProMiSH, queries a similar set of objects containing a given set of query keywords accurately and efficiently. Empirical evaluations, both on real and synthetic datasets, showed that ProMiSH has a speed-up of more than four orders over the state-of-the-art tree-based techniques. Empirical studies on datasets of sizes up to 10 million and dimensions up to 100 for queries of sizes up to 9 established the scalability of ProMiSH.

Querying patterns by example in a spatial dataset was presented in Chapter 4. A new algorithm, QUIP, was proposed for querying these patterns from the dataset. QUIP has two index-based scalable search strategies: TARS and SPARS. Experimental results on real raster image datasets showed that TARS offers an 87% improvement for small queries, and SPARS a 52% improvement for large queries in running time, as compared to linear search. Qualitative tests on real datasets achieved precision of more than 80%.

Pattern queries in a temporal dataset were explored in Chapter 5. This chapter specifically studied the problem of duplicate video retrieval. A new non-metric distance measure was proposed to find the similarity between a query and a database video. Novel search algorithms based on pre-computed distances and pruning

techniques were presented for efficiently querying duplicate videos. Experiments showed that the proposed technique answers video queries of duration 60 seconds in 0.032 seconds with a high accuracy of 97.5%.

Queries, such as database similarity searches, return results satisfying certain properties of distances or scores. For domain scientists, the absolute values of scores are seldom sufficient. Statistical significance or *p-value* of the result is a more useful criterion. An efficient method to calculate the approximate p-value of a multi-object result was discussed in Chapter 6. Experimental evaluations on large databases showed that the method is practical, runs five orders of magnitude faster than the basic approach, and has an error of less than 5% in p-value computation.

7.1 Impact

Near neighbor queries form a vital step in many querying and mining applications. The state-of-the-art methods for querying near neighbors fail to guarantee both accuracy and efficiency for high-dimensional datasets. Therefore, these applications lack the desired performance for high-dimensional datasets. The availability of an efficient method for near neighbor search, like SIMP, will have a

strong impact on the future success of these applications. SIMP will also accelerate building of new applications for many emerging datasets.

The pattern querying techniques proposed in this thesis will encourage development of many useful products. These techniques will also enhance human capabilities to explore and analyze large repositories of heterogenous datasets.

7.2 Future work

A variety of new datasets, e.g., social networks, offer new opportunities for querying patterns. A user in a social network is represented by her attributes and relationships. This dataset is inherently heterogenous. It will be a challenging future work to develop methods for querying patterns in social networks.

Recent times have also seen the availability of dynamic datasets, i.e., the datasets which change with time. These kinds of datasets offer another venue for designing and querying useful patterns.

Bibliography

[1] http://vision.ece.ucsb.edu/publications/Sarkar_tech_report_DTW_09.pdf.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *J. Molecular Biology*, 215(3):403–410, 1990.

[4] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.

[5] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *SODA*, pages 573–582, 1994.

[6] V. Athitsos, M. Potamias, P. Papapetrou, and G. Kollios. Nearest neighbor retrieval using distance-based hashing. *Proc. of ICDE*, pages 327–336, April 2008.

[7] I. Bartolini, P. Ciaccia, and M. Patella. A Sound Algorithm for Region-Based Image Retrieval Using an Index. In *DEXA Workshop*, pages 930–934, 2000.

[8] P. Baumann. Web-enabled raster gis services for large image and map databases. In *DEXA*, page 870, 2001.

[9] M. Bawa, T. Condie, and P. Ganesan. Lsh forest: self-tuning indexes for similarity search. In *WWW*, pages 651–660, 2005.

[10] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[11] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[12] S. Berchtold, C. Böhm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *PODS*, pages 78–86, 1997.

[13] S. Berchtold, D. A. Keim, H.-P. Kriegel, and T. Seidl. Indexing the solution space: A new technique for nearest neighbor search in high-dimensional space. *IEEE TKDE*, 12(1):45–57, 2000.

[14] S. Berretti, A. D. Bimbo, and E. Vicario. Spatial Arrangement of Color in Retrieval by Visual Similarity. *Pattern Recognition*, 35(8):1661–1674, 2002.

[15] M. Bertini, A. D. Bimbo, and W. Nunziati. Video clip matching using MPEG-7 descriptors and edit distance. In *Proc. of CIVR*, pages 133–142, 2006.

[16] D. N. Bhat and S. K. Nayar. Ordinal measures for image correspondence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 415–423, 1998.

[17] C. Böhm. A cost model for query processing in high-dimensional data. *ACM TODS*, 25:129–178, 2000.

[18] J. Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17:419–428, 2001.

[19] X. Cao, G. Cong, and C. S. Jensen. Retrieving top-k prestige-based relevant spatial web objects. *PVLDB*, 3:373–384, 2010.

[20] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *SIGMOD*, pages 373–384, 2011.

[21] C. Carson, S. Belongie, H. Greenspan, and J. Malik. Blobworld: Image Segmentation Using Expectation-Maximization and Its Application to Image Querying. *PAMI*, 24(8):1026–1038, 2002.

[22] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *ACM STOC*, pages 380–388, 2002.

[23] S. Cheung and A. Zakhor. Estimation of web video multiplicity. In *Proc. SPIE–Internet Imaging*, volume 3964, pages 34–36, 1999.

[24] C. Chiu, C. C. Yang, and C. S. Chen. Efficient and effective video copy detection based on spatiotemporal analysis. In *Ninth IEEE International Symposium on Multimedia*, pages 202–209, Dec 2007.

[25] C. Y. Chiu, J. H. Wang, and H. C. Chang. Efficient histogram-based indexing for video copy detection. *Ninth IEEE International Symposium on Multimedia Workshops, 2007.*, pages 265–270, Dec. 2007.

[26] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, pages 426–435, 1997.

[27] P. Ciaccia, M. Patella, and P. Zezula. A cost model for similarity queries in metric spaces. In *PODS*, pages 59–68, 1998.

[28] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2:337–348, 2009.

[29] K. Dadason, H. Lejsek, F. Asmundsson, B. Jonsson, and L. Amsaleg. Videntifier: identifying pirated videos in real-time. In *Proc. of the 15th International Conference on Multimedia*, pages 471–472. ACM, 2007.

[30] C. Dagli and T. S. Huang. A Framework for Grid-Based Image Retrieval. In *ICPR*, volume 2, pages 1021–1024, 2004.

[31] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry*, pages 253–262, 2004.

[32] R. Datta, J. Li, and J. Z. Wang. Content-Based Image Retrieval: Approaches and Trends of the New Age. In *MIR '05: Int. Workshop on Multimedia Information Retrieval*, pages 253–262, 2005.

[33] I. De Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, pages 656–665, 2008.

[34] R. Delongchamp, T. Lee, and C. Velasco. A Method for Computing the Overall Statistical Significance of a Treatment Effect Among a Group of Genes. *BMC Bioinformatics*, 7, 2006.

[35] T. L. Department, M. Gld, C. Thies, and T. M. Lehmann. Content-based image retrieval in medical applications. In *Procs. Int. Society for Optical Engineering (SPIE)*, pages 312–320, 2000.

[36] S. Derrode and F. Ghorbel. Robust and efficient Fourier-Mellin transform approximations for gray-level image reconstruction and complete invariant description. *Computer Vision and Image Understanding*, 83(1):57–78, 2001.

[37] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li. Modeling lsh for performance tuning. In *CIKM*, pages 669–678, 2008.

[38] Y. Du, D. Zhang, and T. Xia. The optimal-location query. In *SSTD*, pages 163–180, 2005.

[39] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231. AAAI Press, 1996.

[40] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In *PODS*, pages 102–113, 2001.

[41] R. Finkel and J. L. Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4(1):1–9, 1974.

[42] M. A. Fischler and R. C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, 1981.

[43] S. K. Fisher, G. P. Lewis, K. A. Linberg, and M. R. Verardo. Cellular Remodeling in Mammalian Retina: Results from Studies of Experimental Retinal Detachment. *Progress in Retinal and Eye Research*, 24(3):395–431, 2005.

[44] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[45] J. L. Ganley and J. P. Cohoon. The Rectilinear Steiner Tree on a Checkerboard. *ACM Trans. Design Automation of Electronic Systems*, 1(4):512–522, 1996.

[46] M. R. Garey and D. S. Johnson. The Rectilinear Steiner Tree Problem is *NP*-Complete. *SIAM J. on Applied Mathematics*, 32(4):826–834, 1977.

[47] M. Gertz, Q. Hart, C. Rueda, S. Singhal, and J. Zhang. A data and query model for streaming geospatial image data. In *EDBT Workshops*, pages 687–699, 2006.

[48] P. Ghosh, E. D. Gelasca, K. Ramakrishnan, and B. Manjunath. *Duplicate Image Detection in Large Scale Databases*. Book Chapter in Platinum Jubilee Volume, Oct 2007.

[49] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[50] A. G. Gutierrez. Applying OLAP pre-aggregation techniques to speed up query response times in raster image databases. In *ICSOFT*, pages 259–266, 2007.

[51] A. G. Gutierrez and P. Baumann. Modeling fundamental geo-raster operations with array algebra. In *ICDM Workshops*, page 607, 2007.

[52] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[53] M. Hadjieleftheriou, G. Kollios, P. Bakalov, and V. J. Tsotras. Complex spatio-temporal pattern queries. In *VLDB*, pages 877–888, 2005.

[54] A. Hampapur and R. M. Bolle. Comparison of distance measures for video copy detection. In *Proc. of ICME*, pages 737 – 740, Aug 2001.

[55] R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing spatial-keyword (SK) queries in geographic information retrieval (GIR) systems. In *SSDBM*, pages 16–26, 2007.

[56] G. R. Hjaltason and H. Samet. Distance Browsing in Spatial Databases. *ACM Trans. Database Syst.*, 24:265–318, 1999.

[57] D. P. Huttenlocher, G. A. Klanderman, and W. J. Rucklidge. Comparing images using the Hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):850–863, Sept 1993.

[58] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems*, 9:482–502, 1984.

[59] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.

[60] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM TDS*, 30(2):364–397, 2005.

[61] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE TPAMI*, 33:117–128, 2010.

[62] W. Johnson and J. Lindenstrauss. Extensions of lipschitz mappings into a hilbert space. *Contemporary Mathematics*, 26:189–206, 1984.

[63] A. Joly and O. Buisson. A posteriori multi-probe locality sensitive hashing. In *ACM MM*, pages 209–218, 2008.

[64] A. Joly, O. Buisson, and C. Frelicot. Statistical similarity search applied to content-based video copy detection. *Int. Conf. on Data Engineering Workshops*, page 1285, 2005.

[65] A. Joly, O. Buisson, and C. Frelicot. Content-based copy retrieval using distortion-based probabilistic similarity search. *IEEE Transactions on Multimedia*, 9(2):293–306, Feb. 2007.

[66] A. Joly, C. Frelicot, and O. Buisson. Robust content-based video copy identification in a large reference database. In *Int. Conf. on Image and Video Retrieval*, pages 414–424, 2003.

[67] A. Joly, C. Frelicot, and O. Buisson. Feature statistical retrieval applied to content based copy identification. *International Conference on Image Processing, 2004.*, 1:681–684 Vol. 1, Oct. 2004.

[68] E. Kasutani and A. Yamada. The MPEG-7 color layout descriptor: a compact image feature description for high-speed image/video segment retrieval. In *Proc. of ICIP*, volume 1, pages 674–677, 2001.

[69] Y. Ke and R. Sukthankar. PCA-SIFT: A more distinctive representation for local image descriptors. In *Proc. of CVPR*, pages 506–513, 2004.

[70] Y. Ke, R. Sukthankar, and L. Huston. An efficient parts-based near-duplicate and sub-image retrieval system. In *MM*, pages 869–876, 2004.

[71] A. Khodaei, C. Shahabi, and C. Li. Hybrid indexing and seamless ranking of spatial and textual features of web documents. In *DEXA*, pages 450–466, 2010.

[72] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *STOC*, pages 599–608, 1997.

[73] C. A. Lang and A. K. Singh. Faster similarity search for multimedia data via query transformations. *Int. J. Image Graphics*, pages 3–30, 2003.

[74] J. Law-To, O. Buisson, V. Gouet-Brunet, and N. Boujemaa. Robust voting algorithm based on labels of behavior for video copy detection. In *Proc. of the 14th annual ACM International Conference on Multimedia*, pages 835–844. ACM, 2006.

[75] J. Law-To, L. Chen, A. Joly, I. Laptev, O. Buisson, V. Gouet-Brunet, N. Boujemaa, and F. Stentiford. Video copy detection: a comparative study. In *Proc. of CIVR*, pages 371–378. ACM, 2007.

[76] J. K. Lawder and P. J. H. King. Using space-filling curves for multi-dimensional indexing. In *BNCOD*, pages 20–35, 2000.

[77] S. Lazebnik, C. Schmid, and J. Ponce. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *CVPR*, pages 2169–2178, 2006.

[78] H. Lejsek, F. H. Ásmundsson, B. P. Jónsson, and L. Amsaleg. Nv-tree: An efficient disk-based index for approximate search in very large high-dimensional collections. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31:869–883, 2009.

[79] H. Lejsek, F. H. Asmundsson, B. T. Jonsson, and L. Amsaleg. Scalability of local image descriptors: a comparative study. In *ACM MULTIMEDIA '06*, pages 589–598, New York, NY, USA, 2006. ACM.

[80] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A simple and efficient algorithm for R-tree packing. Technical report, Institute for Computer Applications in Science and Engineering (ICASE), 1997.

[81] W. Li and C. X. Chen. Efficient data modeling and querying system for multi-dimensional spatial data. In *GIS*, pages 58:1–58:4, 2008.

[82] Y. Li, J. S. Jin, and X. Zhou. Matching commercial clips from TV streams using a unique, robust and compact signature. In *Digital Image Computing: Techniques and Applications, 2005. DICTA'05. Proceedings 2005*, pages 39–39, 2005.

[83] Z. Li, H. Xu, Y. Lu, and A. Qian. Aggregate nearest keyword search in spatial databases. In *Asia-Pacific Web Conference*, pages 15–21, 2010.

[84] Y. Linde, A. Buzo, R. Gray, et al. An algorithm for vector quantizer design. *IEEE Transactions on communications*, 28(1):84–95, 1980.

[85] L. Liu, W. Lai, X. Hua, and S. Yang. Video Histogram: A Novel Video Signature for Efficient Web Video Duplicate Detection. *Lecture Notes in Computer Science*, 4352:94–103, 2007.

[86] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[87] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *VLDB*, pages 950–961, 2007.

[88] J. Malki, N. Boujemaa, C. Nastar, and A. Winter. Region Queries without Segmentation for Image Retrieval by Content. In *Visual Information and Information Systems (VISUAL)*, pages 115–122, 1999.

[89] B. S. Manjunath and W. Y. Ma. Browsing large satellite and aerial photographs. In *ICIP*, pages 765–768, 1996.

[90] B. S. Manjunath, P. Salembier, and T. Sikora. *Introduction to MPEG-7: Multimedia Content Description Interface*. Wiley, 2002.

[91] B. Martins, M. J. Silva, and L. Andrade. Indexing and ranking in geo-ir systems. In *workshop on GIR*, pages 31–34, 2005.

[92] K. Mikolajczyk and C. Schmid. An affine invariant interest point detector. In *Proc. European Conf. Computer Vision*, pages 128–142. Springer Verlag, 2002.

[93] K. Mikolajczyk and C. Schmid. Scale and affine invariant interest point detectors. *IJCV*, 60(1):63–86, 2004.

[94] K. Mikolajczyk and C. Schmid. A performance evaluation of local descriptors. *PAMI*, 27(10), 2005.

[95] R. Mohan. Video sequence matching. In *Proc. of ICASSP*, pages 3697–3700, 1998.

[96] S. Morain and S. L. Baros. *Raster Imagery in Geographic Information Systems*. ONWARD press, 1996.

[97] R. Motwani, A. Naor, and R. Panigrahi. Lower bounds on locality sensitive hashing. In *SCG '06*, pages 154–157, 2006.

[98] A. Natsev, R. Rastogi, and K. Shim. WALRUS: A Similarity Retrieval Algorithm for Image Databases. *TKDE*, 16:301–316, 2004.

[99] P. Over, A. F. Smeaton, and P. Kelly. The TRECVID 2007 BBC rushes summarization evaluation pilot. In *TVS '07: Proc. of the International Workshop on TRECVID Video Summarization*, pages 1–15. ACM, 2007.

[100] R. Pajarola and P. Widmayer. Spatial indexing into compressed raster images: How to answer range queries without decompression. In *IW-MMDBMS*, page 94, 1996.

[101] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *SODA*, pages 1186–1195, 2006.

[102] D. Papadias, N. Mamoulis, and Y. Theodoridis. Processing and optimization of multiway spatial joins using r-trees. In *PODS*, pages 44–55, 1999.

[103] H.-H. Park, G.-H. Cha, and C.-W. Chung. Multi-way spatial joins using r-trees: Methodology and performance evaluation. In *Symposium on Advances in Spatial Databases*, pages 229–250, 1999.

[104] K. Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2(6):559–572, 1901.

[105] E. G. M. Petrakis and C. Faloutsos. Similarity searching in medical image databases. *TKDE*, 9(3):435–447, 1997.

[106] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *CVPR*, pages 1–8, 2007.

[107] L. Rabiner and B. H. Juang. *Fundamentals of Speech Recognition*. Prentice Hall Signal Processing Series, NJ, 1993.

[108] P. Rigaux, M. Scholl, and A. Voisard. *Spatial Databases: With Application to GIS*. Morgan Kaufmann, 2001.

[109] G. Salton. *Automatic Text Processing: The Transformation Analysis and Retrieval of Information by Computer*. Addison-Wesley, 1988.

[110] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., 2005.

[111] A. Sarkar, P. Ghosh, E. Moxley, and B. S. Manjunath. Video fingerprinting: Features for duplicate and similar video detection and query-based video retrieval. In *Proc. of SPIE, Multimedia Content Access: Algorithms and Systems II*, volume 6820, pages 68200E–68200E–12, 2008.

[112] S. Shekhar and S. Chawla. *Spatial Databases: A Tour*. Prentice Hall, 2003.

[113] S. Shekhar and Y. Huang. Discovering spatial co-location patterns: A summary of results. In *Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pages 236–256, 2001.

[114] H. T. Shen, X. Zhou, Z. Huang, J. Shao, and X. Zhou. UQLIPS: a real-time near-duplicate video clip detection system. In *VLDB*, pages 1374–1377. VLDB Endowment, 2007.

[115] M. Silva, G. Camara, R. Souza, D. Valeriano, and M. Escada. Mining patterns of change in remote sensing image databases. In *ICDM*, page 8, 2005.

[116] V. Singh, A. Bhattacharya, and A. K. Singh. Querying spatial patterns. In *EDBT*, pages 418–429, 2010.

[117] V. Singh, A. Bhattacharya, A. K. Singh, C. Banna, G. P. Lewis, and S. K. Fisher. QUIP: Querying Significant Patterns from Image Databases. Technical report, Dept. of Computer Science, University of California, Santa Barbara, 2007.

[118] V. Singh and W. Jiang. An algorithm and hardware design for very fast similarity search in high dimensional space. In *IEEE Granular Computing (GrC)*, pages 426–431, 2010.

[119] V. Singh and A. K. Singh. Profile based sub-image search in image databases. Technical Report 2010-20, Dept. of Computer Science, University of California, Santa Barbara, October 2010.

[120] V. Singh, S. Venkatesha, and A. K. Singh. Geo-clustering of images with missing geotags. In *IEEE Granular Computing (GrC)*, pages 420–425, 2010.

[121] H. Tan, X. Wu, C. Ngo, and W. Zhao. Accelerating near-duplicate video matching by combining visual similarity and alignment distortion. In *ACM MULTIMEDIA '08*, pages 861–864. ACM, 2008.

[122] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*, pages 563–576, 2009.

[123] S. Vaid, C. B. Jones, H. Joho, and M. Sanderson. Spatio-textual indexing for geographical search on the web. In *SSTD*, pages 218–235, 2005.

[124] L. Vinhas, R. C. M. de Souza, and G. Câmara. Image data handling in spatial databases. In *Brazilian Symposium on GeoInformatics*, 2003.

[125] P. Vinten-Johansen, H. Brody, N. Paneth, S. Rachman, M. Rip, and D. Zuck. *Cholera, Chloroform, and the Science of Medicine: A Life of John Snow*. Oxford University Press, 2003.

[126] J. Z. Wang, J. Li, and G. Wiederhold. SIMPLIcity: Semantics-Sensitive Integrated Matching for Picture LIbraries. *PAMI*, pages 947–963, 2001.

[127] R. Weber and M. Milvoncic. Efficient Region-Based Image Retrieval. In *CIKM*, pages 69–76, 2003.

[128] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, pages 194–205, 1998.

[129] C. Won, D. Park, and S. Park. Efficient use of MPEG-7 edge histogram descriptor. *Etri Journal*, 24(1):23–30, 2002.

[130] X. Wu, A. G. Hauptmann, and C. Ngo. Practical elimination of near-duplicates from web video search. In *Proceedings of the 15th International Conference on Multimedia*, pages 218–227. ACM, 2007.

[131] T. Xia, D. Zhang, E. Kanoulas, and Y. Du. On computing top-t most influential spatial sites. In *VLDB*, pages 946–957, 2005.

[132] D. Yankov, E. Keogh, L. Wei, X. Xi, and W. Hodges. Fast best-match shape searching in rotation-invariant metric spaces. *IEEE Transactions on Multimedia*, 10(2):230–239, 2008.

[133] M. L. Yiu, X. Dai, N. Mamoulis, and M. Vaitis. Top-k spatial preference queries. In *ICDE*, pages 1076–1085, 2007.

[134] J. Yuan, L. Y. Duan, Q. Tian, and C. Xu. Fast and robust short video clip search using an index structure. In *Proceedings of the 6th ACM SIGMM Int. Workshop on Multimedia Information Retrieval*, pages 61–68, 2004.

[135] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *ICDE*, pages 688–699, 2009.

[136] D. Zhang, Y. Du, T. Xia, and Y. Tao. Progressive computation of the min-dist optimal-location query. In *VLDB*, pages 643–654, 2006.

[137] D. Zhang, B. C. Ooi, and A. K. H. Tung. Locating mapped resources in web 2.0. In *ICDE*, pages 521–532, 2010.

[138] J. Zhang, M. Gertz, and D. Aksoy. Spatio-temporal aggregates over raster image data. In *GIS*, pages 39–46, 2004.

[139] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *SIGMOD*, pages 915–926, 2010.

[140] W. Zhao and C. Ngo. Scale-Rotation Invariant Pattern Entropy for Keypoint-Based Near-Duplicate Detection. *IEEE Transactions on Image Processing*, 18(2):412–423, 2009.

[141] W. L. Zhao, C. W. Ngo, H. K. Tan, and X. Wu. Near-duplicate keyframe identification with interest point matching and pattern learning. *IEEE Transactions on Multimedia*, 9:1037–1048, 2007.

[142] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma. Hybrid index structures for location-based web search. In *CIKM*, pages 155–162, 2005.