

Near Neighbour Searching in High Dimensional Metric Space

David Marshall

Supervisor:

Dr. Shyjan Mahamud (NICTA)

A subthesis submitted in partial fulfillment of the degree of
Master of Information Technology (eScience) at
The Department of Computer Science
Australian National University

November 2006

© David Marshall

Typeset in Palatino by $\text{T}_{\text{E}}\text{X}$ and $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X} 2_{\epsilon}$.

Figures using R, Dia, GIMP, and Postscript

Except where otherwise indicated, this thesis is my own original work.

David Marshall
11 November 2006

Acknowledgments

The ANU's eScience program has provided me with the motivation to return to school. NICTA has made available the expert supervision to help me complete this project.

Dr. Shyjan Mahamud (NICTA) has provided invaluable guidance on the key matters and important related work.

Dr. Alistair Rendell (ANU) helped keep me on schedule and provided critical feedback in the area of experimentation and communication.

My wife has made it possible, through her encouragement and support.

Abstract

This study begins with a review of the literature concerning nearest-neighbour searching of high-dimensional space with application to computer vision. Scale-Invariant Feature Transform (SIFT) descriptors are identified as a powerful method of object classification that motivates the study of efficient methods of near-neighbour searching. The KD-tree algorithm, as well as numerous variations, suffer from the “curse of dimensionality” at the level of dimensions (128) of SIFT descriptors. Locality Sensitive Hashing (LSH) has received much attention as a powerful means of overcoming, to some extent, the problem of high-dimensionality. The hybrid spill-tree (SP-Tree) demonstrated that earlier spatial data structures designed originally to solve the exact nearest neighbour query can be adapted to solve the approximate query, with significant performance implications.

The spilling aspect of SP-Tree was a significant motivation for this research. LSHB adapts LSH with spilling. It is shown that LSHB performs 10% to 20% better than LSH in terms of query speed, but uses 2-3 times or more memory. LSH and LSHB are demonstrated to be effective at near-neighbour searching in 128-dimensional space. KD-Tree is adapted to the near (rather than nearest) neighbour problem. The effect of data attributes such as “density” and distribution are explored. The comparison of algorithms designed for different variations of the exact/approximate near/nearest neighbour problem is discussed.

Contents

Acknowledgments	v
Abstract	vii
1 Introduction	1
1.1 Motivation	1
1.2 Background	3
1.2.1 Nearest Neighbour Search in Computer Vision	3
1.2.2 The Importance of High-Dimensional Space	4
1.2.3 Time/Space Complexity	5
1.3 Objectives	5
2 Related Work	7
2.1 Scale Invariant Feature Transform	7
2.2 Structures and Algorithms	9
2.2.1 KD-Tree	9
2.2.2 Best-Bin-First	10
2.2.3 Sphere-Rectangle-Tree (SR-Tree)	11
2.2.4 LSH (Locality Sensitive Hashing)	11
2.2.5 Hybrid Spill-Tree	13
2.2.6 Comparisons	14
2.3 Acceleration Using Programmable Graphics Processing Unit	14
3 Adapting LSH with Spilling	17
3.1 Problem Definition	17
3.2 (r, c) -Near-Neighbour KD-Tree	18
3.3 LSHB	19
3.3.1 Rationale	19
3.3.2 Algorithm Description	20
3.4 Implementation and Experimental Design	22
3.5 Experimentation	23
4 Conclusions	27
4.1 Suitability of LSH and LSHB at 128-Dimensions	27
4.2 Use of Spilling in LSHB	27
4.3 The Effect of Data “Density” and Distribution	27
4.4 Future Work	28

A	Key Abstracts	29
B	Example Applications	33
C	Other References	35
D	KD-Tree Algorithms	39
	Bibliography	43

Introduction

1.1 Motivation

Computer vision, in particular object classification, has become a vital component of many important technologies that exist today. Some examples that many Australians have already benefited from include:

- **Security** (SmartGate - face recognition for Qantas aircrew [Australian Customs Service]. See Figure 1.1.)
- **Health** (InViVo - 3D ultrasound imaging [Fraunhofer Institute for Computer Graphics]. See Figure 1.2.)
- **Safety** (Safe-T-Cam - monitoring heavy vehicle speed and fatigue [RTA, NSW]. See Figure 1.3.)



Figure 1.1: The SmartGate system uses computer vision to identify Qantas aircrew and enhance the security of air travel.

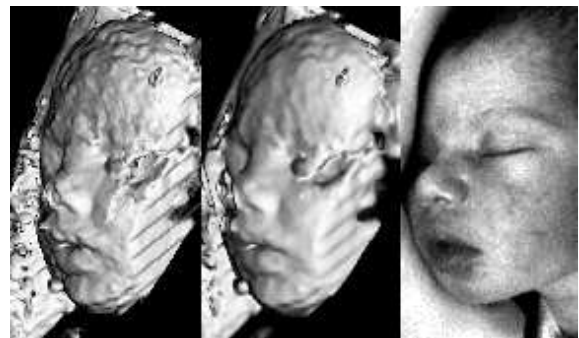


Figure 1.2: The InViVo system uses computer vision to provide diagnostic capabilities for unborn children.

These examples are meant to show applications of computer vision in general, not necessarily nearest-neighbour approaches in particular. Measuring similarity between images is a key aspect of object classification. The nearest neighbour approach to finding similarities between images is a powerful tool for the situation where little



Figure 1.3: The Safe-T-Cam system uses computer vision to monitor heavy vehicle lane positioning and speed, improving the safety of road travel.

or no prior knowledge of the dataset exists. Prior knowledge could include: Knowing what the class of interest is, such as human faces; or Knowing image characteristics such as distance to subject.

The motivation for this research is based on the use of nearest neighbour methods of feature comparison within the field of computer vision. The applicability of these issues, however, extend beyond computer vision. Nearest neighbour methods, in general, are non-parametric. In other words, they are most useful when there is little or no prior knowledge of the data distribution. Parametric methods could also be applied to computer vision, if there is sufficient prior knowledge. Both parametric and non-parametric methods could be applied to fields other than computer vision, such as data mining. Figure 1.4 shows where the focus of this paper fits within the larger scope of object classification. The bottom row gives examples of what could fall into each category. The list of examples is neither exhaustive nor prescriptive. Rather, they provide a general indication of the kind of problem suited to a method.

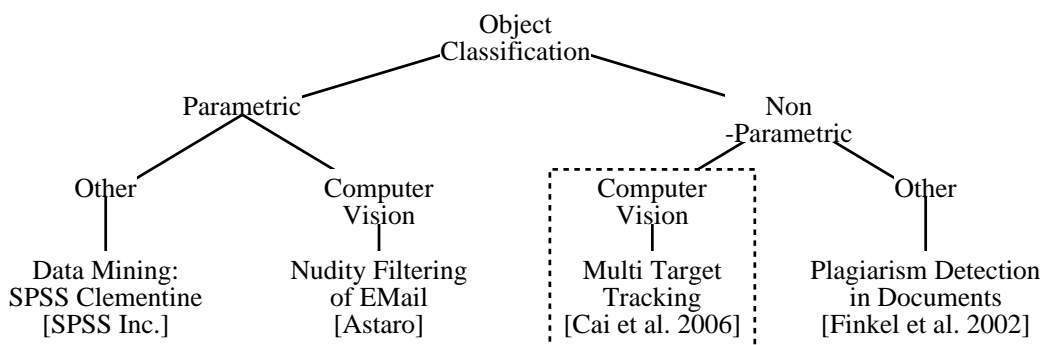


Figure 1.4: In this paper, neighbour searching is considered as a non-parametric method of computer vision. This area is set apart from the wider range of classification by a dotted box.

1.2 Background

There are two general varieties of the nearest-neighbour search problem:

(Nearest neighbor) Given a set P of points in a d -dimensional space R^d , construct a data structure which given any query point q finds the point in P with the smallest distance to q . [Shakhnarovich et al. 2006a]

(c-Approximate nearest neighbor) Given a set P of points in a d -dimensional space R^d , construct a data structure which given any query point q , reports any point within distance at most c times the distance from q to p , where p is the point in P closest to q . [Shakhnarovich et al. 2006a]

Simple two-dimensional examples include:

- finding the nearest airport to a plane, for an emergency landing
- finding the nearest pizzeria to a customer's home, for a pizza delivery

The following are more substantial examples where a nearest-neighbor approach to estimating similarity can be particularly effective. Details can be found in Appendix B.

- Multiple target tracking, such as of players in sport. [Cai et al. 2006]
- Urban surveillance. [Oliveira et al. 2006]
- Video indexing. [Bhagavathy and Saban 2004]
- Fractal image compression. [Tong and Wong 2002]

The extension of the concept of Euclidean distance in 2D and 3D space to higher dimensional space provides an effective comparison of items in these sorts of domains. In this review, particular regard will be given to the performance of nearest neighbour searching in a large database of Scale Invariant Feature Transform (SIFT) descriptors. SIFT descriptors are useful in support of many object classification tasks. They are described in section 2.1.

1.2.1 Nearest Neighbour Search in Computer Vision

A common sub-task of many more complex computer vision problems is that of matching an interesting “feature” from an image to its closest match in a set of features. A feature might be an edge (a change in intensity) or something more complex, such as a pattern of changing intensity across several pixels. What is required is a means of comparing whole features simultaneously. If each pixel in the feature is considered as a “dimension” of space, the feature can be plotted as if it were a point in that space.

In **metric space**, there is a valid concept of distance between points. If we treat two features as points in space, we can determine the distance between them. The lesser the distance between them, the more similar they are in appearance. Figure 1.5 is a simple visual example. Three features of two pixels each are compared.

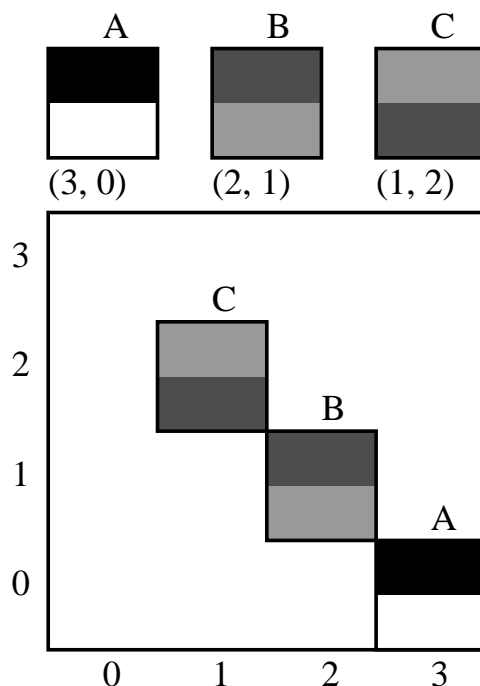


Figure 1.5: Representing (white, light-grey, dark-grey, black) as (0, 1, 2, 3) we plot simple 2-pixel features as points. The top pixel is the x-dimension, and the bottom pixel is the y-dimension. By calculating the distances \overline{AB} and \overline{AC} , we can determine that B is closer to A than C is.

1.2.2 The Importance of High-Dimensional Space

A two-pixel feature, although useful as a visual demonstration, is fairly trivial. Typically, interesting features are much larger. As the number of dimensions increase, the concept and means of computing distance between points remains valid. High-dimensional space can, however, have a negative impact on the performance of nearest neighbour searching algorithms that were designed with simpler 2D or 3D space in mind.

The degraded performance caused by increasing dimensions can sometimes be overcome by knowing something about the distribution of points. One dimension might provide greater discrimination than another, or two dimensions might be highly correlated, for instance. When this is the case, a parametric method might be employed. Such prior knowledge of the dataset distribution is not always possible, however. This review will focus on non-parametric methods that are useful when there is little or no prior knowledge of the dataset.

1.2.3 Time/Space Complexity

A brute force approach to finding a nearest neighbour would take $O(n)$ time to evaluate each of n points in a set. Since there might be millions of points, and assessment might be expensive, time can become a serious issue.

Consider also that a 128-dimensional point, using 8 bit integers occupies 128 bytes. 1 million points would take 125 MB. 32 million points would take 4 GB. At this number, available memory could become a serious issue as well. Nearest neighbour search algorithms can be assessed on their use of memory, storage, and processing power.

1.3 Objectives

The power of a near-neighbour approach to object classification motivates the research of efficient methods for performing such a search. The next chapter will review the existing methods and issues associated with such problems, as well as its importance in general. Some potential opportunities are identified which merit in-depth investigation. This will set the stage for describing the research which has been performed.

Related Work

In order to classify images via features, one must first extract the features. An overview will be given of a feature extraction method that motivates the research of efficient near-neighbour searching. After introducing Scale Invariant Feature Transform (SIFT), a selection of important nearest neighbour search methods will be described, including KD-Tree, KD-Tree variants, and Locality Sensitive Hashing (LSH). The review ends with a discussion of how a parallel Graphics Processing Unit (GPU) implementation could benefit all of these nearest neighbour methods. Although not a core concern of this research effort, the growing prevalence of programmable GPUs warrants a brief discussion.

Appendix A contains the abstracts of papers that were found to be of key importance to this review. Appendix C contains some brief comments about other references that were of lesser importance or not relevant on closer inspection.

2.1 Scale Invariant Feature Transform

Primarily as a means of motivating the research of efficient near-neighbour searching in high dimensions, we will now discuss a powerful means of feature extraction.

Lowe introduced the Scale-Invariant Feature Transform (SIFT) descriptor [Lowe 1999] in 1999. The basic idea is to extract interesting features from an image and be able to compare them to template features, regardless of a change in scale or orientation.

The following is a summary of the major steps in the process of extracting SIFT descriptors:

1. **Scale-space extrema detection:** The first stage of computation searches over all scales and image locations. It is implemented efficiently by using a difference-of-Gaussian function to identify potential interest points that are invariant to scale and orientation.
2. **Keypoint localization:** At each candidate location, a detailed model is fit to determine location and scale. Keypoints are selected based on measures of their stability.
3. **Orientation assignment:** One or more orientations are assigned to each keypoint location based on local image gradient directions. All

future operations are performed on image data that has been transformed relative to the assigned orientation, scale, and location for each feature, thereby providing invariance to these transformations.

4. **Keypoint descriptor:** The local image gradients are measured at the selected scale in the region around each keypoint. These are transformed into a representation that allows for significant levels of local shape distortion and change in illumination.

[Lowe 2004]

The scale-space referred to here is the series of progressively blurred (using a Gaussian kernel) images. Image gradients are changes in intensity. (The images are converted to grey-scale before processing.) Figure 2.1 shows a simplified visualisation of these steps.

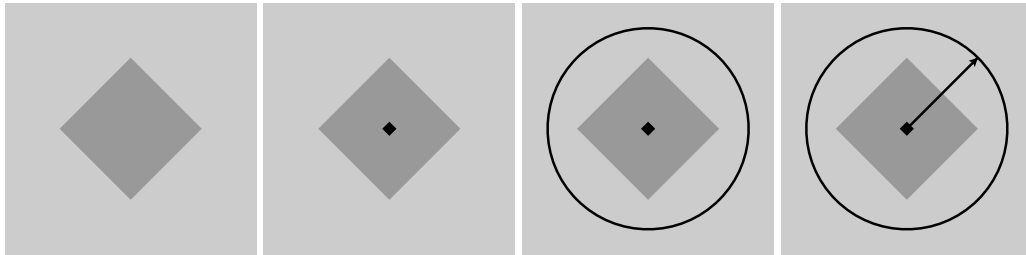


Figure 2.1: A simple representation of the steps in SIFT: Take an input image; identify a point of interest; determine the scale of the feature; and determine the feature's orientation.

Figure 2.2 shows template objects on top, a candidate image in the middle, and resulting SIFT descriptor matches on the bottom.



Figure 2.2: Model images of planar objects are shown in the left column of images. Recognition results to the right show model outlines and image keys used for matching.
[Lowe 1999]

The SIFT descriptor, for the purpose of point-comparison, ultimately ends up as a 128-dimensional vector. This method is particularly strong for object recognition because of its relative invariance to changes in scale and orientation. The resulting

high-dimensional descriptor, however, means that many of the existing algorithms would suffer from the “curse of dimensionality”.

The “curse of dimensionality” is frequently referred to in the literature. It is a term coined by [Bellman 1961] and, in general terms, describes the exponentially growing difficulty of performing various types of spatial analysis in high dimensions. Köppen [Köppen] provides a worthwhile discussion of the curse. In a simple sense, this “curse” is sometimes described as the tendency for points in high dimensional space to become equi-distant from each other. The tendency for points to be equi-distant is more applicable to random/uniform distributions than to realistic data. This becomes apparent when applying LSH and LSHB to actual SIFT data. When using the SIFT data, accuracy improved significantly. This is likely the result of clusters of features belonging to actual objects. A photograph tends to be a composite of many distinct objects. The intent of SIFT descriptors is to help differentiate between such objects. As such, the descriptors tend to be clustered rather than uniformly distributed.

2.2 Structures and Algorithms

2.2.1 KD-Tree

The originally proposed KD-Tree algorithm [Bentley 1975] stores each point in a set of k -dimensional points in a node of a binary tree. At each level of the tree, a different dimension is used to partition the child points. At a given node, the remaining points are divided into those with discriminating dimension less than and greater than the partitioning point. Figure 2.3 shows example data and Figure 2.4 shows the same data stored in a kd-tree.

Bentley suggested a version of the kd-tree [Freidman et al. 1977] which is optimized by carefully choosing the discriminating dimension and partitioning value based on the dimension with maximum spread. (ie. The dimension with greatest difference between lowest and highest values.) This version also controls the depth at which the division of nodes terminates by allowing a bucket-size of nodes at termination, rather than dividing nodes until there are no more nodes remaining. This version is recorded in algorithmic notation at appendix D.

A search, using this structure, works recursively. Starting from the root, the search will first proceed down to the terminal node within which the query point would belong if it was being inserted.

Consider a hyperrectangle that defines the boundary of the current node in the search tree. Every time the search moves up or down a node in the tree, the hyperrectangle is updated to reflect the current boundary.

Consider a hypersphere centred at the query point, the radius of which is the distance to the closest point found so far. Every time the search visits a terminal node, it will search the bucket of points at that node and update the closest match (and therefor shrink the hypersphere).

If, after searching a bucket, the hypersphere has a radius less than the distance to the closest boundary of the hyperrectangle - we say that the “ball is within bounds”.

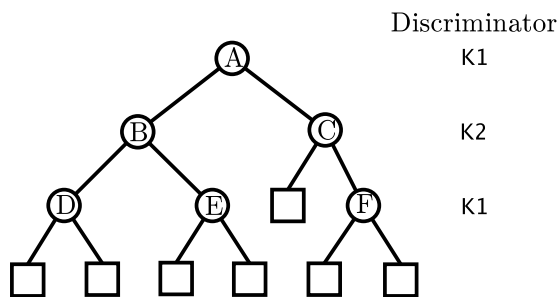
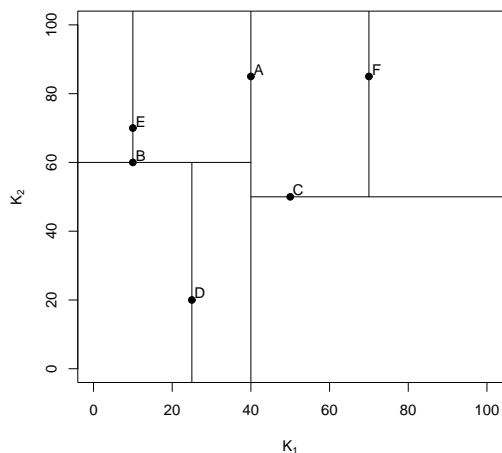


Figure 2.4: KD-Tree (binary tree) of example data partitioned in Figure 2.3 [Bentley 1975]

Figure 2.3: 2D graph of example data, showing a space recursively subdivided by partitions through median points. [Bentley 1975]

If the ball is within bounds, that means that no closer match can be found in another part of the search tree - and the search terminates. If the ball is not within bounds, the recursion will backtrack up the tree and visit other nodes. This search process is demonstrated in Figures 2.5, 2.6, and 2.7.

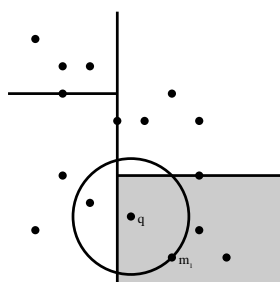


Figure 2.5: Finding the closest point in the first bucket searched

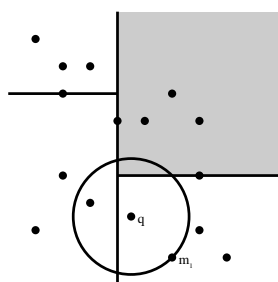


Figure 2.6: Searching a bucket where ball overlaps

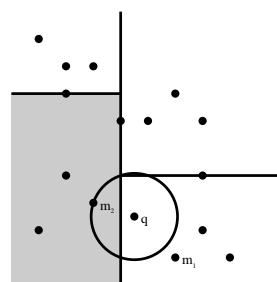


Figure 2.7: Searching another overlapping bucket and finding the nearest neighbour

2.2.2 Best-Bin-First

The Best-Bin-First (BBF) approach, used in [Beis and Lowe 1997], is a KD-Tree variant and reduces search time by maintaining a priority queue of bins that are not visited.

This priority is based on the distance to the partition from the query point. In this way, back-tracking is prioritised to the bins that are closest to the query point. The idea is that bins that are closer have a higher probability of yielding a closer neighbour. This priority queue would enable the search to proceed from Figure 2.5 to Figure 2.7, skipping Figure 2.6.

2.2.3 Sphere-Rectangle-Tree (SR-Tree)

The defining feature of the SR-Tree [Katayama and Satoh 1997] is that it uses the intersection of a bounding rectangle and a bounding sphere for regions in the tree, as shown in Figure 2.8. This intersection provides a tighter boundary. A tighter bounding of regions reduces the amount of back-tracking required, because the search sphere (discussed in 2.2.1) will overlap fewer regions of the search tree.

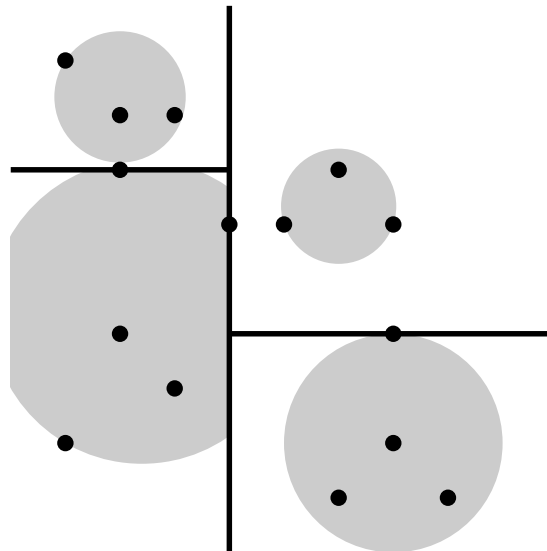


Figure 2.8: With the SR-Tree, subdivided points are more tightly bounded by the intersection of a sphere and a rectangle.

2.2.4 LSH (Locality Sensitive Hashing)

With hashing in general, a small change in data can have a dramatic effect on that data's hash value. The purpose of LSH (introduced in [Lowe 1999]) is for a change in location to result in a proportional change in the hash value. (Hence, the hash function is "locality sensitive".) A formal definition of what makes a hash locality sensitive is as follows. Consider a query point q and distances r_1 and r_2 as in Figure 2.9.

If a point falls within distance r_1 from q , its probability of hashing to the same bucket as q should be $> p_1$. If a point falls further away from q than distance r_2 , its probability of hashing to the same bucket as q should be $\leq p_2$. In order for the hash

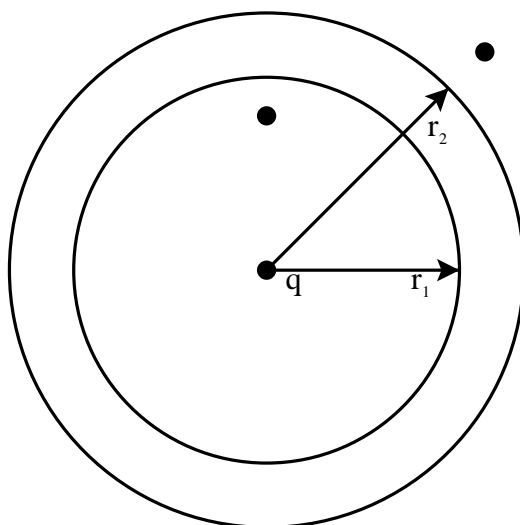


Figure 2.9: The basic premise of LSH is that a point within distance r_1 should be hashed to the same bucket as query point q with a high probability, and a point with distance greater than r_2 should be hashed to the same bucket as query point q with a low probability. Such a hash is considered “locality sensitive”.

function to be valuable, p_1 should be significantly greater than 0.5 and p_2 should be significantly less than 0.5.

It turns out that a binary hash function based on a random hyperplane is locality sensitive. Figure 2.10 shows how two points from the same side of the line have a higher probability of being close to each other than two points from opposite sides.

If the query point is in the vicinity of the hyperplane, there is an increased chance that its nearest neighbour will be across the plane rather than on the same side. This could result in an error. The proportion of space in the region next to the plane is small compared to the region further away from the plane, however. Here, then, is the trade-off. If an increased number of binary hash functions are used to divide the space into buckets, the number of points that need to be searched will be decreased. At the same time, however, increasing the number of hyperplanes increases the proportion of space that is near to a border and therefore the chance of error.

Consider the two extremes: If the number of hyperplanes is 0, every point must be searched and the accuracy will be 100%. With enough hyperplanes, each bucket will contain only a single point. In that case, many query points would fall in empty buckets and the accuracy would be nearly 0%. Effective use of LSH, then, requires choosing the number of binary hashes such that search time is minimized while accuracy is kept within an acceptable range.

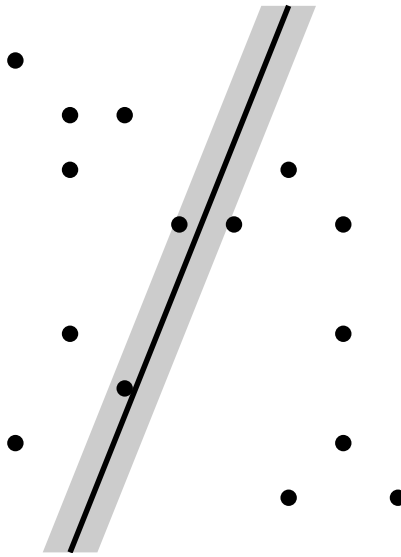


Figure 2.10: Binary Hash function based on random hyperplane: Points will be hashed to one side of the plane or the other. Points falling near the plane (the shaded region) have a greater chance of causing error.

2.2.5 Hybrid Spill-Tree

A Metric-Tree (M-Tree) is similar to a KD-Tree. It differs most significantly by how it partitions at each node. Where a KD-Tree partitions on a single dimension at each node, an M-Tree partitions with a hyperplane. That said, the Spill-Tree (SP-Tree) is a variant of the M-Tree where the boundaries of child nodes are allowed to overlap. (The children can share points.)

The main reason that exact nearest neighbour search using a KD or M-Tree takes much longer than approximate search is the back-tracking to verify that a near neighbour found early on is in fact the nearest. [Liu et al. 2005] estimates the back-tracking to take up to 95% of the search time. Simply eliminating the back-tracking, however, results in a high error rate. The overlapping of child nodes introduced by the SP-Tree reduces this error at the expense of some duplicate assessment of nodes. The effect of this overlapping is shown in Figure 2.11

In their hybrid SP-Tree, [Liu et al. 2005] use a combination of overlapping and non-overlapping nodes as well as random projection.¹Random projection “projects” the data from a high-dimensional space into a lower-dimensional space, thus reducing complexity at some cost to accuracy. They view the LSH approach as “a variant of random projection” [Liu et al. 2005]. Their results showed the hybrid SP-Tree to perform between 2.5 and 31 times faster than LSH at the same level of accuracy.

¹It is worth emphasizing that the hybrid SP-Tree employs a mixture of techniques, including spilling and random projection.

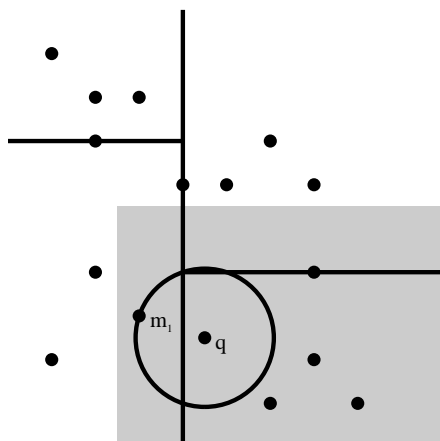


Figure 2.11: The SP-Tree takes a basic KD or M-Tree and extends it with the concept of “spilling”. Points near to a partition will be stored on both sides, in the tree. The search is made more efficient, at the expense of increased storage space.

2.2.6 Comparisons

The KD-Tree algorithm [Bentley 1975] has been in use for about 30 years. As such, it is fairly widely known and understood. Simplicity and ease of implementation can be advantages, but high dimensionality puts pressure on the KD-Tree algorithm and provides motivation to seek a more efficient alternative.

The BBF [Beis and Lowe 1997] and SR-Tree [Katayama and Satoh 1997] algorithms each provide a performance improvement over KD-Tree. Those improvements are not mutually exclusive. The SR-Tree could benefit from the use of a priority queue, and BBF could benefit from tighter bounding.

LSH [Lowe 1999] provides a significant improvement, at high dimensions, over KD-Tree, BBF, and SR-Tree. It has become a popular approach over the past 5 years. The SP-Tree algorithm [Liu et al. 2005], however, has been shown to out-perform LSH. This is accomplished, in part, by allowing the regions of the tree structure to overlap and share data points.

2.3 Acceleration Using Programmable Graphics Processing Unit

The focus of the previously discussed spatial data structures is to reduce the number of points that must be measured. Each algorithm uses its particular method to organize the points into a number of “buckets”. Ultimately, each of these algorithms will end up performing a linear scan on the entire contents of one or more buckets. All of these methods would, therefore, benefit from acceleration of the linear scan portion of the search.

Although not a central concern of this research, a brief discussion of Graphics Processing Units (GPUs) seems warranted. Modern GPUs contain multiple parallel

channels for processing vertexes and fragments. The GeForce 6800 Ultra, for instance, has 16 parallel fragment processors. It also has 256 MB of texture memory. This would allow approximately 2,000,000 SIFT descriptors to be downloaded to the GPU at a time.

There are 4 main alternatives for creating fragment programs on a GPU:

- Assembly Language for the particular GPU.
- HLSL (Microsoft's shading language, applicable to DirectX)
- GLSL (applicable to OpenGL)
- Cg (optimized for NVidia GPUs, but applicable to the widest range of Operating Systems)

Cg is currently the most mature and widely applicable (higher level) shading language, and would be an appropriate choice for GPU implementation. The syntax of all three higher level shading languages are similar, however, and the choice is primarily a matter of operating system and GPU manufacturer.

The linear scan is well suited to a parallel implementation, because the dataset can be divided into multiple independent subsets and evaluated concurrently. [Bustos et al. 2006] found a linear scan nearest neighbour search to run an average 6.4 times faster on a GPU vs a CPU. They used a Pentium IV 3.0Ghz CPU and a GeForce 6800 Ultra GPU. Their fragment programs were implemented using Cg. Other algorithms, such as LSH and SP-Tree, should also benefit from a partial GPU implementation.

Adapting LSH with Spilling

3.1 Problem Definition

There is a potentially confusing range of problem definitions in the literature on “neighbour” searching. They are named with terms such as: “nearest-neighbour”, “ ϵ -nearest-neighbour”, and “(R,c)-near-neighbour”. Before embarking on a project to improve upon an algorithm, it is best to be very deliberate about the category of problem that is to be solved.

	exact	approximate
nearest	KD-Tree	
near		LSH

Table 3.1: A neighbour search problem can be classified as “near” vs “nearest” and “exact” vs “approximate”. KD-Tree, as originally designed, solves the exact-nearest problem, whereas LSH solves the approximate-near problem

Moving from “nearest” to “near” and from “exact” to “approximate” are both relaxations of the problem. Relaxing the problem, where appropriate, allows one to relieve the “curse of dimensionality” and find something useful in a shorter time. Further, these problems can be solved with a probability of error $\delta \geq 0$.

In the “nearest” neighbour search, the distance R of concern is a function of the distance to the actual nearest neighbour. Nearest also implies finding a point with least distance to the query point. In the “near” neighbour search, the distance R is an input parameter to the search. (ie. It is some threshold distance within which neighbours are near enough to be interesting.) Near implies only finding a point (possibly among many) closer than a given distance to the query point.

In a computer vision object classification application, one might apply the near-neighbour search to a training set of images. The training set would contain images that have been labeled as being of the target class or not. All of the features would be extracted from a candidate image. Then, each feature would queried in the training set. Due to the noise inherent in such images, exact results are not required and an approximation to some level of accuracy is acceptable. The class of the candidate

image could be estimated based on an assessment of the positive and negative results from the near-neighbour searches.

In the “exact” search, a neighbour is found with distance $\leq 1 \cdot R$. In the “approximate” search, a neighbour is found with distance $\leq c \cdot R$, where $c \geq 1$.

The original KD-Tree, for example, solves the exact-nearest-neighbour problem. If we took a strategy of terminating the back-tracking in KD-Tree early, we would be solving the approximate-nearest-neighbour problem, with $\delta > 0$. LSH solves the approximate-near-neighbour problem, with some constant $\delta > 0$.

If we are to compare neighbour search algorithms, we must be aware of the category of problem they solve. Most existing neighbour search algorithms can be used to solve a different variety of neighbour search, either through modification of the algorithm or assessment of accuracy. For example, the metric-tree (exact-nearest) was modified (to hybrid spill-tree) in [Liu et al. 2005] to solve approximate-nearest. Also in [Liu et al. 2005], an implementation of LSH (approximate-near) was used to solve approximate-nearest. In this case, the assessment of LSH accuracy was modified to suit the approximate-nearest problem and allow comparison of hybrid spill-tree to LSH at an equal error level. Further, the measure of approximation was the average (across queries) percent difference between distance to found vs nearest neighbour.

The studies of [Liu et al. 2005] provided the original motivation to investigate the utility of applying spilling to LSH. It would seem most appropriate to make such an investigation on the basis of the problem for which LSH was designed, approximate-near.

3.2 (r, c)-Near-Neighbour KD-Tree

The standard KD-Tree and Metric-Tree algorithms were implemented for solving the exact nearest-neighbour problem. The adaptation of Metric Tree [Liu et al. 2005] to solve the approximate nearest-neighbour suggested that KD-Tree could be similarly adapted to solve the approximate-near-neighbour problem. The essential adaptation, in this case, is to allow the KD-Tree search to terminate once a near neighbour has been found. With the approximate-near-neighbour problem, there is no need to back-track in the tree and verify that the found neighbour is nearest. Results for an exact near neighbour search of KD-Tree, compared to a linear scan, can be seen in Figure 3.1.

Without taking further measures, such as solving the approximate rather than exact near-neighbour problem, KD-Tree does no better than a linear scan at any significant dimensionality. ($d \geq 8$) We will now proceed with an explanation of the rationale behind adapting LSH with spilling (LSHB).

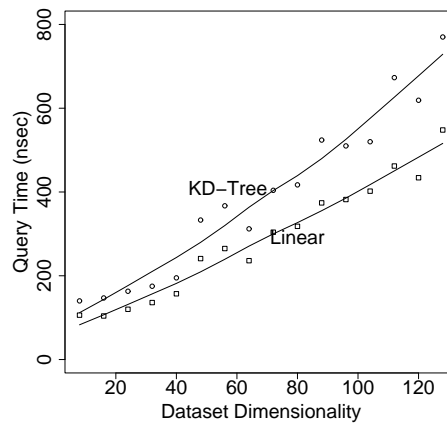


Figure 3.1: Query Time as a function of Dataset Cardinality. An exact-near KD-Tree search performs no better than a linear scan at > 8 dimensions.

3.3 LSHB

3.3.1 Rationale

Figure 3.2 shows the standard LSH scheme. In this case, two 2-bit hashes are used. (ie. $L=2, k=2$) p_1 and p_2 are both stored $L=2$ times. q hashes to $L=2$ buckets, and those buckets will be searched until a near neighbour is found or until the search terminates. p_1 hashes to one of the same buckets as q .

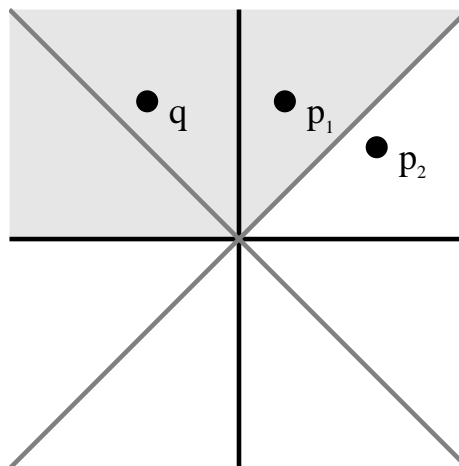


Figure 3.2: LSH might use two 2-bit hashes. The grey lines are one 2-bit hash, and the black lines are a second hash. The shaded region represents the search space, given query point q . p_1 will be examined, but p_2 will not.

Figure 3.3 shows the LSHB scheme. Again, a 2-bit hash is used... but only one. (ie. $L=1, k=2$) With LSHB, L is always 1. p_1 is stored 2 times, because it falls within the “buffer” of one bit of the hash. p_2 is only stored 1 time. p_1 hashes to the same bucket as q . Query points only hash to 1 bucket. They are not subject to buffering.

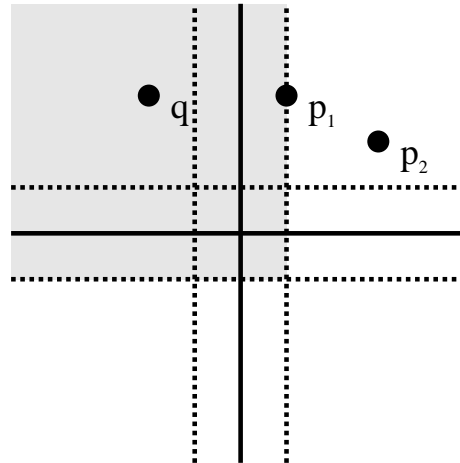


Figure 3.3: LSHB might use a 2-bit hash. The solid lines represent the hash, and the dotted lines show the buffer zone. The shaded region represents the search space, given query point q . p_1 will be examined, but p_2 will not.

The buffering in LSHB serves a similar purpose to the multiple (L) hashes of LSH. They both serve to reduce the probability δ of not reporting a near neighbour when there is one. The rationale was that LSHB would answer queries in a shorter time, with equal error rate δ .

3.3.2 Algorithm Description

The LSHB algorithm is identical to the LSH algorithm except for the buffered nature of the hashes and the insertion of points into the search structure. Once the search structure has been built, queries are processed in the same way as with LSH.

In general, the individual “bits” of the hashes can be described as surfaces that assess points as either falling on one side of the surface or the other. In this way, each bit or surface in the hash contributes a “0” or a “1” to the hash.

The significant difference, with LSHB, is that a point may be stored on both sides if it is close enough to the surface. “Close enough” (ie. the buffer distance) is a euclidean distance and is chosen so as to achieve the desired error rate.

One specific type of binary hash, that is effective yet simple to implement, is the hyper-plane. In the LSH scheme, a hyper-plane might be defined as the plane bisecting two points. The two points, for example, could be two randomly chosen points from the dataset. This would guarantee that the binary hash is “non-trivial”. (ie. that it has 1 or more points on both sides)

With LSHB, we must additionally be able to assess how close the points are to the plane. We don't need to know this distance absolutely, however, only if it is within the buffer distance. Here, the design of the algorithm loses some generality in order to take advantage of the specific nature of a hyper-plane. Instead of using two random points to define the hyper-plane, it is defined by three points as follows (considering buffer distance = b):

- Select a random point C .
- Select a random point A , at distance $2 \cdot b$ from C .
- Calculate the point B , along a line from A through C and at distance $2 \cdot b$ from C .

Thus, C (the center point) falls on a hyperplane bisecting points A and B . The significance of choosing points A and B at distance $2 \cdot b$ from C is illustrated in Figures 3.4 and 3.5. As with LSH, a point that is closer to A than B falls on side A of the plane. Additionally, a point that is closer to C than A is within distance b of the plane.

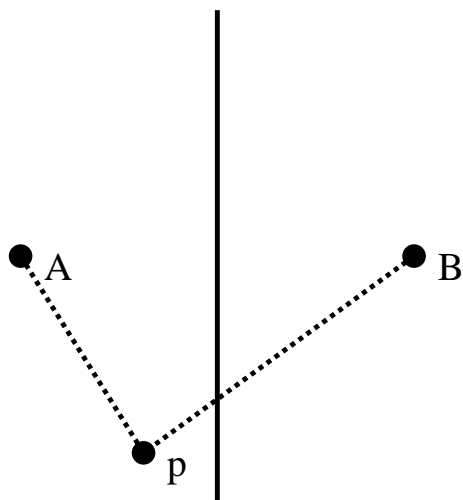


Figure 3.4: In the LSH scheme, a binary hash is defined by points A and B and a bisecting hyperplane (represented by the line). A query point q will hash to one side of the hyperplane or the other regardless of proximity to the plane.

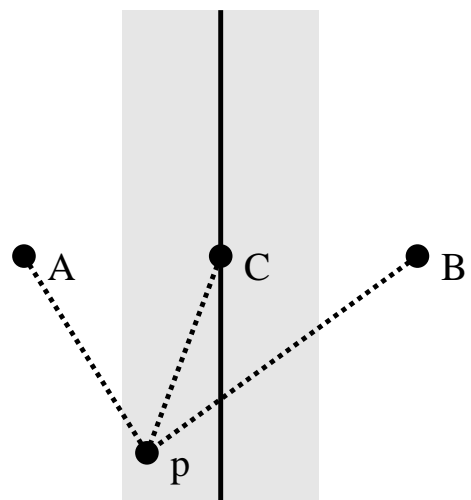


Figure 3.5: In the LSHB scheme, a binary hash is defined by points A , B , and midpoint C and a bisecting hyperplane (represented by the line) passing through C . Data points falling within the shaded region will be hashed to both sides of the plane. A query point q on one side of the plane will be compared to points on that side, as well as points within the shaded region.

3.4 Implementation and Experimental Design

In order to effectively show the differences in performance between LSH and LSHB, one must use a dataset and error rate δ such that $L \geq 2$. If $L = 1$ and $b = 0$, the two algorithms are equivalent.

We must set $\delta > 0$ in order to make a non-trivial comparison. LSH cannot guarantee $\delta = 0$ unless either all or none of the points in the dataset are near neighbours to all query points. As $\delta \rightarrow 0$, $L \rightarrow \infty$. LSHB could guarantee $\delta = 0$ with $b = r \cdot c$, but such a large b (especially at higher dimensions) dramatically increases storage requirements in practice. (This is demonstrated in Section 3.5.)

In this implementation of LSH, parameters k and L were not optimized automatically. They were set to suit the purposes of the experiments, such as achieving a specified accuracy. With LSHB, the parameter b was similarly chosen to give the desired accuracy. In an application, one would find an optimal k (and associated L or b) that gives the best query performance at the desired level of accuracy. This might be accomplished via sample queries.

It takes longer to process a query with no near neighbour, so lower density will have a tendency to increase mean processing times.

It has been stated that LSH “works well on data that is extremely high-dimensional but sparse”. [Datar et al. 2004] It is worth noting the effect of sparseness (or density) on the performance of an algorithm. When data has a high density, a higher proportion of queries result in a near-neighbour. Since it takes longer to process queries having no near-neighbour, low density data favours algorithms with a lower bound on finding no near-neighbour.

It can be difficult to accurately determine p_1 and p_2 in the LSH scheme. p_1 and p_2 could be used to compute δ , the guaranteed maximum probability of reporting no near-neighbour when there is one. In practice, it was found to be simpler to disregard p_1 and p_2 and find the optimal k and L for a particular training set. This presumes using a mean error rate, rather than a theoretical maximum error rate.

With LSHB, the need to hash points to multiple buckets lead to the most errors during design and implementation. Consequently, this took a disproportionate amount of design/development/testing time.

3.5 Experimentation

For all the experiments, the problems fit well within main memory. The experiments were performed on a Dell laptop with an Intel Core Duo 2.0GHz CPU and 2GB of DDR2 667MHz SDRAM. The operating system was Fedora Core 5. All code was written in C and compiled using GCC. In all graphs where the performance of LSH and LSHB are compared, the comparison is done with approximately equal error rates. (It was difficult to precisely control the error rate and there was some fluctuation, but not such that there was any bias.)

Figures 3.7, 3.8, and 3.6 show the effect of changing the buffer size of the LSHB algorithm. The “relative buffer” shown on the x-axis is the buffer size as a percentage of $(\frac{r-c}{dimensions})$

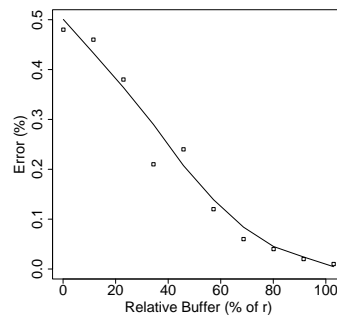


Figure 3.6: Error rate as a function of LSHB buffer size, where buffer is shown as a percentage of r (defining “near”). The error rate decreases to 0% as the buffer size increases from 0 to 100% of r .

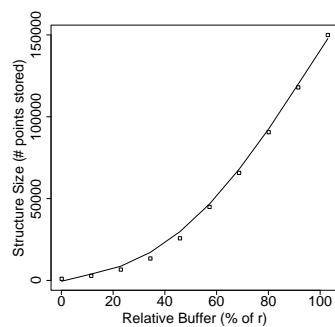


Figure 3.7: Storage Size as a function of LSHB buffer size, where buffer is shown as a percentage of r (defining “near”). Storage decreases as the buffer size increases from 0 to 100% of r .

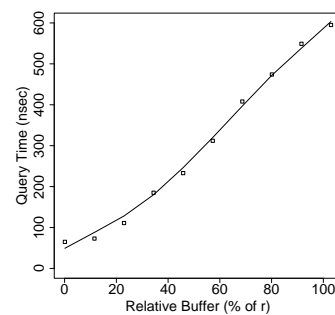


Figure 3.8: Query time as a function of LSHB buffer size, where buffer is shown as a percentage of r (defining “near”). Queries take longer as the buffer size increases from 0 to 100% of r .

Both synthetic and realistic datasets were used in these experiments. The synthetic data was generated uniformly at random, with the required dataset size and dimensionality. This allowed a study of LSHB's relative effectiveness with increasing dimensionality and dataset size. Random-uniform data, however, has a distribution unlike data from real applications and is not sufficient for drawing conclusions about a method's effectiveness in practice.

One usually shows how an algorithm performs with increasing dataset cardinality (n). Simply increasing n , while holding (r,c) constant, increases the density. (ie. the percentage of queries that will return a near-neighbour) It is difficult to calculate (r,c) to achieve a specific density. Instead of trying to hold density constant, the relative performance of standard LSH and LSHB is emphasized in Figures 3.9 and 3.10, rather than the specific shape of the performance curve.

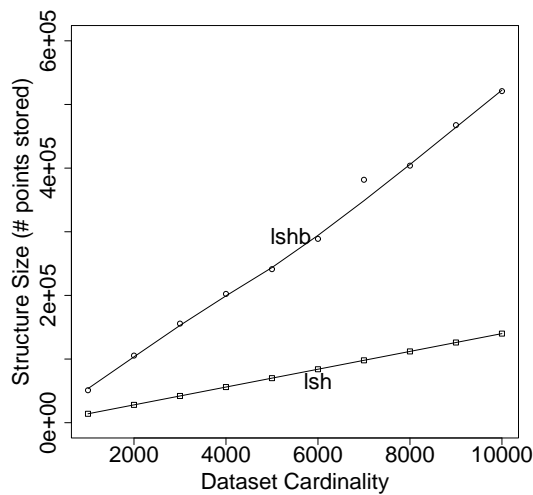


Figure 3.9: Storage size as a function of dataset cardinality. LSHB requires consistently greater space than LSH.

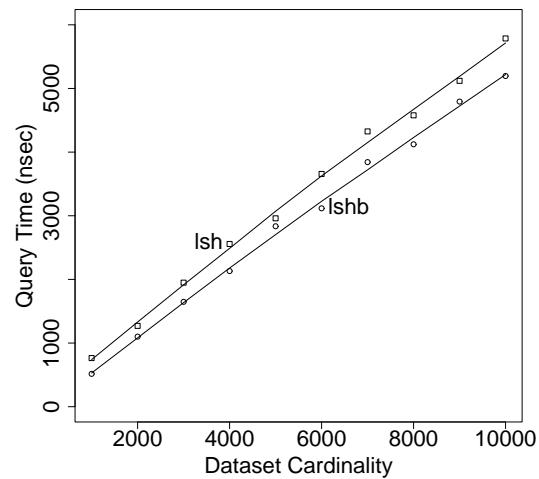


Figure 3.10: Query time as a function of dataset cardinality. LSHB answers queries in approximately 10% less time than LSH.

With near-neighbour algorithms, it is also common to show query performance against increasing dimensionality. Again, increasing dimensionality while holding (r,c) constant has an effect on density... in this case, decreasing it. As with was the case when showing performance in relation to dataset size, density will be allowed to decrease and the relative performance of the two algorithms will be emphasized rather than the shape of the curve. It is also difficult to maintain a constant error rate while changing other parameters so, as much as possible, error rates are kept equal between algorithms but allowed to fluctuate in Figures 3.11 and 3.12.

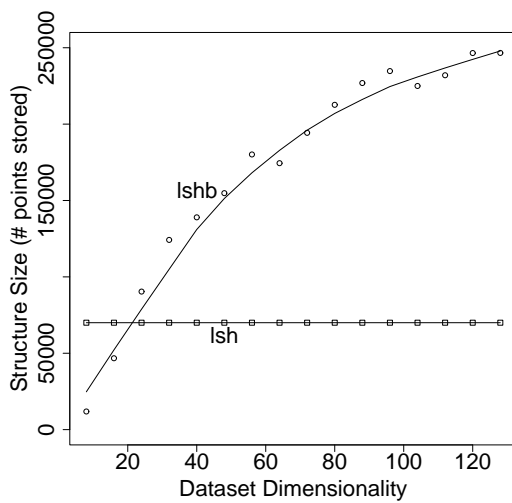


Figure 3.11: Storage size as a function of dataset dimensionality. LSH requires a constant amount of storage space, whereas LSHB requires a growing amount of storage.

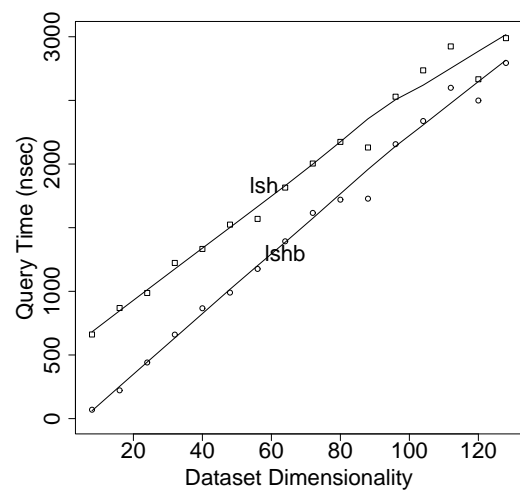


Figure 3.12: Query time as a function of dataset dimensionality. LSHB consistently answers queries in less time than LSH.

After demonstrating the relative performance of the two methods with respect to dataset size and dimensionality using synthetic data, it is important to make a more convincing argument with realistic data. The motivation and focus of this study was SIFT feature descriptors for computer vision, so realistic SIFT data was chosen as a measure of effectiveness. Demonstration binaries have been made available [David G. Lowe] for SIFT feature extraction, so they were used in conjunction with the cars [INRIA LEAR] dataset. The features for each image in the set were extracted and stored in individual files. For each query, a single descriptor was selected from one image and compared to the entire set of points for one other image. The results are shown in Figures 3.13 and 3.14.

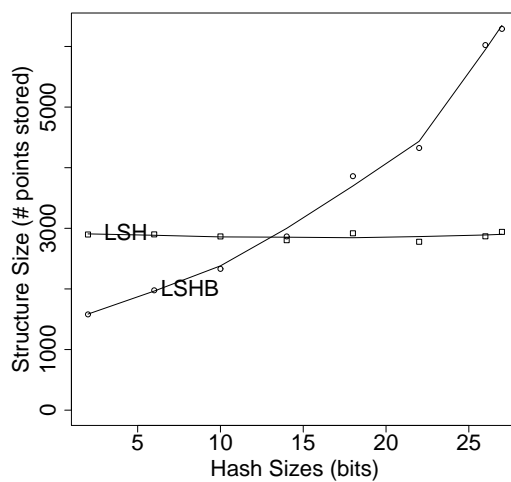


Figure 3.13: Storage size as a function of hash complexity (bits). LSH requires a constant amount of storage space, whereas LSHB requires a growing amount of storage.

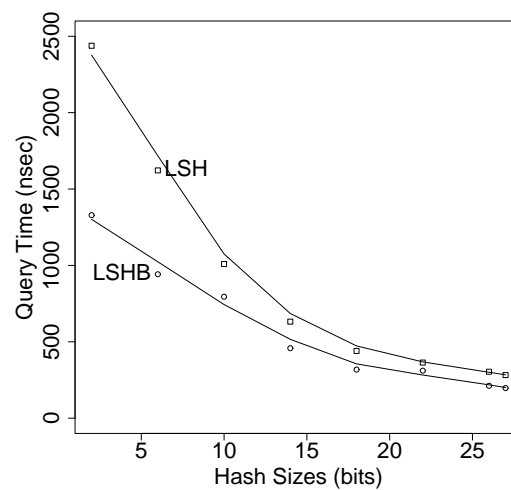


Figure 3.14: Query time as a function of hash complexity (bits). LSHB consistently answers queries in less time than LSH.

Conclusions

The primary objectives of this research were achieved with promising results. Some additional issues were discovered during the process that are also worth noting.

4.1 Suitability of LSH and LSHB at 128-Dimensions

Unlike some algorithms which suffer the “curse of dimensionality”, LSH and LSHB continue to outperform a linear scan at high dimensions such as 128. Either LSH or LSHB are appropriate near-neighbour search algorithms in the context of SIFT data.

4.2 Use of Spilling in LSHB

Allowing points to “spill” onto both sides of a binary (hyper-plane) hash when they are near the plane provides a performance advantage, at a cost of increased memory usage. Where sufficient memory is available (2-3 times or more), LSHB outperforms LSH on SIFT data query speed by 10% to 20%. Since LSH cannot simply improve its performance beyond the optimal (k, L) by using more memory, this is an important point.

4.3 The Effect of Data “Density” and Distribution

High density data lead to faster query performance for both LSH and LSHB. Uniform/random (synthetic) data proved to be more challenging for both algorithms. Natural data (with inherent clustering) resulted in greater accuracy.

4.4 Future Work

As a support tool for research such as this, it could be beneficial to have a synthetic data generator that would allow a mixture of distributions. Such a generator could take as input certain parameters to define clustering/distribution, numbers of points, etc. With such a tool, one could conveniently vary parameters such as dataset size and dimensionality while still having a dataset that is closer to being “real”.

This implementation did not employ hashing of the buckets, which would have allowed parameter k to be larger. Using 128D SIFT descriptors, the experiments showed that k appeared to be approaching optimal but the optimal k was beyond the constraints of this implementation. It could be useful, in a future study, to implement bucket hashing and be able to find and use the optimal k .

One assumption of LSH (and its error guarantee δ) is that hash functions are chosen randomly. This is the way it was done in these implementations. With realistic data, one might be able to improve the accuracy vs speed trade-off by selecting combinations of hash functions that result in faster queries and greater accuracy.

The error rates chosen for the various comparisons in this study were somewhat arbitrary, even though they served their purpose. It would be interesting, and perhaps more convincing, to follow this study with an actual implementation of an object classifier. Then, LSH could be compared to LSHB with more tangible constraints and outcomes. (Such as... what is the maximum accuracy that each can achieve while classifying video frames at 15fps and using a limited amount of memory?)

Key Abstracts

The following studies are of key importance in the area of SIFT and nearest neighbour search.

Nearest-Neighbor Methods in Learning and Vision: Theory and Practice (Chapter 1 - Introduction) [Shakhnarovich et al. 2006a]

Abstract: The nearest-neighbor (NN) problem occurs in the literature under many names, including the best match or the post office problem. The problem is of significant importance to several areas of computer science, including pattern recognition, searching in multimedia data, vector compression, computational statistics, and data mining. For many of these applications, including some described in this book, large amounts of data are available. This makes nearest-neighbor approaches particularly appealing, but on the other hand it increases the concern regarding the computational complexity of NN search. Thus it is important to design algorithms for nearest-neighbor search, as well as for the related classification, regression, and retrieval tasks, which remain efficient even as the number of points or the dimensionality of the data grows large. This is a research area on the boundary of a number of disciplines: computational geometry, algorithmic theory, and the application fields such as machine learning. This area is the focus of this book, which contains contributions from researchers in all of those fields.

An Algorithm for Finding Best Matches in Logarithmic Expected Time [Freidman et al. 1977]

Abstract: An algorithm and data structure are presented for searching a file containing N records, each described by k real valued keys, for the m closest matches or nearest neighbors to a given query record. The computation required to organize the file is proportional to $kN \log N$. The expected number of records examined in each search is independent of the file size. The expected computation to perform each search is proportional to $\log N$. Empirical evidence suggests that, except for very small files, this algorithm is considerably faster than other methods.

Shape Indexing Using Approximate Nearest-Neighbour Search in High-Dimensional Spaces [Beis and Lowe 1997]

Abstract: Shape indexing is a way of making rapid associations between features detected in an image and object models that could have produced them. When model databases are large, the use of high-dimensional features is critical, due to the improved level of discrimination they can provide. Unfortunately, finding the nearest neighbour to a query point rapidly becomes inefficient as the dimensionality of the feature space increases. Past indexing methods have used hash tables for hypothesis recovery, but only in low-dimensional situations. In this paper, we show that a new variant of the k-d tree search algorithm makes indexing in higher-dimensional spaces practical. This Best Bin First, or BBF, search is an approximate algorithm which finds the nearest neighbour for a large fraction of the queries, and a very close neighbour in the remaining cases. The technique has been integrated into a fully developed recognition system, which is able to detect complex objects in real, cluttered scenes in just a few seconds.

Object Recognition from Local Scale-Invariant Features [Lowe 1999]

Abstract: An object recognition system has been developed that uses a new class of local image features. The features are invariant to image scaling, translation, and rotation, and partially invariant to illumination changes and affine or 3D projection. These features share similar properties with neurons in inferior temporal cortex that are used for object recognition in primate vision. Features are efficiently detected through a staged filtering approach that identifies stable points in scale space. Image keys are created that allow for local geometric deformations by representing blurred image gradients in multiple orientation planes and at multiple scales. The keys are used as input to a nearest-neighbor indexing method that identifies candidate object matches. Final verification of each match is achieved by finding a low-residual least-squares solution for the unknown model parameters. Experimental results show that robust object recognition can be achieved in cluttered partially-occluded images with a computation time of under 2 seconds.

Nearest-Neighbor Methods in Learning and Vision: Theory and Practice (Chapter 3 - Locality-sensitive hashing using stable distributions) [Shakhnarovich et al. 2006b]

Abstract: In this chapter, we introduce and analyze a novel locality-sensitive hashing family. The family is defined for the case where the distances are measured according to the l_s norm, for any $s \in [0,2]$. The hash functions are particularly simple for the case $s = 2$, i.e., the Euclidean norm. The new family provides an efficient solution to (approximate or exact) randomized near neighbor problem.

Distinctive Image Features from Scale-Invariant Keypoints [Lowe 2004]

Abstract: This paper presents a method for extracting distinctive invariant features from images that can be used to perform reliable matching between different views of an object or scene. The features are invariant to image scale and rotation, and are shown to provide robust matching across a substantial range of affine distortion, change in 3D viewpoint, addition of noise, and change in illumination. The features are highly distinctive, in the sense that a single feature can be correctly matched with high probability against a large database of features from many images. This paper also describes an approach to using these features for object recognition. The recognition proceeds by matching individual features to a database of features from known objects using a fast nearest-neighbor algorithm, followed by a Hough transform to identify clusters belonging to a single object, and finally performing verification through least-squares solution for consistent pose parameters. This approach to recognition can robustly identify objects among clutter and occlusion while achieving near real-time performance.

An Investigation of Practical Approximate Nearest Neighbor Algorithms [Liu et al. 2005]

Abstract: This paper concerns approximate nearest neighbor searching algorithms, which have become increasingly important, especially in high dimensional perception areas such as computer vision, with dozens of publications in recent years. Much of this enthusiasm is due to a successful new approximate nearest neighbor approach called Locality Sensitive Hashing (LSH). In this paper we ask the question: can earlier spatial data structure approaches to exact nearest neighbor, such as metric trees, be altered to provide approximate answers to proximity queries and if so, how? We introduce a new kind of metric tree that allows overlap: certain datapoints may appear in both the children of a parent. We also introduce new approximate k-NN search algorithms on this structure. We show why these structures should be able to exploit the same random-projection-based approximations that LSH enjoys, but with a simpler algorithm and perhaps with greater efficiency. We then provide a detailed empirical evaluation on five large, high dimensional datasets which show up to 31-fold accelerations over LSH. This result holds true throughout the spectrum of approximation levels.

A Graphics Hardware Accelerated Algorithm for Nearest Neighbor Search [Bustos et al. 2006]

Abstract: We present a GPU algorithm for the nearest neighbor search, an important database problem. The search is completely performed using the GPU: No further post-processing using the CPU is needed. Our experimental results, using large synthetic and real-world data sets, showed that our GPU algorithm is several times faster than its CPU version.

Example Applications

The following studies provide examples of using a nearest neighbour approach to feature comparison.

Robust Visual Tracking for Multiple Targets [Cai et al. 2006]

Abstract: We address the problem of robust multi-target tracking within the application of hockey player tracking. The particle filter technique is adopted and modified to fit into the multi-target tracking framework. A rectification technique is employed to find the correspondence between the video frame coordinates and the standard hockey rink coordinates so that the system can compensate for camera motion and improve the dynamics of the players. A global nearest neighbor data association algorithm is introduced to assign boosting detections to the existing tracks for the proposal distribution in particle filters. The mean-shift algorithm is embedded into the particle filter framework to stabilize the trajectories of the targets for robust tracking during mutual occlusion. Experimental results show that our system is able to automatically and robustly track a variable number of targets and correctly maintain their identities regardless of background clutter, camera motion and frequent mutual occlusion between targets.

A Video System for Urban Surveillance: Function Integration and Evaluation [Oliveira et al. 2006]

Abstract: This paper is concerned with video sequence analysis for urban area surveillance applications. The aim is to detect, track and classify targets entering a urban scene under varying illumination conditions and distracters. The paper contributions consist in the integration of algorithms for performing the various tasks and in their statistical evaluation. Results are presented on the basis of a benchmark video sequence.

SketchIt: Basketball Video Retrieval Using Ball Motion Similarity. [Bhagavathy and Saban 2004]

Abstract: A prototype basketball video retrieval system is presented in this report. Retrieval is based on the similarity of ball motion in the clip with that in the query. The system uses a query-by-sketch paradigm, where the user provides a sketch of the desired ball trajectory. The video data is pre-processed to make the ball motion invariant to camera translation. The next stage is dimensionality reduction wherein we model the ball motion as a set of parabolic trajectories. An R-tree is used to index these parabolic representations and search for similar trajectories in a low dimension parametric space. The query is processed to obtain its parametric representation, and a nearest neighbor search is performed for similar parabolas. These query results are then post-processed by assigning scores based on various similarity criteria. The system could be extended to other types of videos and moving objects. As a proof of concept, the system was tested for ball trajectories in basketball video.

Adaptive approximate nearest neighbor search for fractal image compression. [Tong and Wong 2002]

Abstract: Fractal image encoding is a computationally intensive method of compression due to its need to find the best match problem is between image subblocks by repeatedly searching a large virtual codebook constructed from the image under compression. One of the most innovative and promising approaches to speed up the encoding is to convert the range-domain block matching problem to a nearest neighbor search problem. This paper presents an improved formulation of approximate nearest neighbor search based on orthogonal projection and pre-quantization of the fractal transform parameters. Furthermore, an optimal adaptive scheme is derived for the approximate search parameter to further enhance the performance of the new algorithm. Experimental results showed that our new technique is able to improve both the fidelity and compression ratio, while significantly reduce memory requirement and encoding time.

Other References

The following papers were reviewed, but found not to be of key importance for one reason or another. In some cases, there is a more current or comprehensive study by the same authors. In other cases, the papers were found not to be as applicable as was initially thought.

Multidimensional binary search trees used for associative searching [Bentley 1975]

This was Bentley's original paper on the KD-Tree. His later paper [Freidman et al. 1977] provides a sufficient and updated reference.

R-trees: a dynamic index structure for spatial searching [Guttman 1984]

This paper dealt with non-point (ie. region) data. It was not applicable to comparing SIFT descriptors. It lead, however, to a more applicable paper on KDB trees. [Robinson 1981]

The K-D-B-tree: a search structure for large multidimensional dynamic indexes [Robinson 1981]

The K-D-B-tree attempts to combine the "multidimensional search efficiency of balanced K-D-trees and the I/O efficiency of B-trees"[Robinson 1981], however it does not overcome the problem of high-dimensions.

Nearest neighbors in high-dimensional spaces [Indyk 2004]

This paper gave an overview of a number of approaches, including a discussion of reducing the computational complexity through approximation.

Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. [Böhm et al. 2001]

This paper provided an overview of numerous nearest neighbour search algorithms. It included a discussion of parallel algorithms that might be an interesting starting point, in conjunction with [Fung 2005] and [Bibby 2005], for investigating a GPU implementation.

Similarity Search in High Dimensions via Hashing [Gionis et al. 1999]

This paper provided a useful overview of LSH.

Approximate nearest neighbors: towards removing the curse of dimensionality [Indyk and Motwani 1998]

This paper included an earlier discussion of LSH, but [Shakhnarovich et al. 2006b] is a more recent presentation.

Nearest Neighbor Queries in Metric Spaces [Clarkson 1997]

This paper discussed an algorithm where performance is dependent on a ratio of the distance between the furthest points and the distance between the nearest points. This algorithm didn't appear to be appropriate to a situation where that ratio was large.

The SR-tree: an index structure for high-dimensional nearest neighbor queries [Katayama and Satoh 1997]

This paper introduces a tree structure similar to the SS-Tree and the R-Tree. Rather than using either a sphere or a rectangle to bound the regions of the tree, it uses an intersection of both. This results in a smaller bounding region.

Two algorithms for nearest-neighbor search in high dimensions [Kleinberg 1997]

This paper was primarily theoretical. No concrete performance comparisons with other algorithms were provided. This paper would be more useful if the objective was to theoretically prove the performance of the particular algorithms involved.

Near Neighbor Search in Large Metric Spaces [Brin 1995]

This paper discussed an improvement over the vp-tree.

Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces [Yianilos 1993]

This paper discussed the vp-tree as an improvement over the kd-tree.

Five balltree construction algorithms [Omohundro 1989]

This paper discussed ball-trees in detail, but did not contrast that approach with other nearest neighbor algorithms.

Fast Pose Estimation with Parameter-Sensitive Hashing [Shakhnarovich et al. 2003]

This paper presented an extension to LSH that determines efficient hashing functions. It might be appropriate to investigate this work further if the objective is to determine an optimal variation of LSH.

Fast Feature Detection with a Graphics Processing Unit Implementation [Bibby 2005]

This paper focused on feature detection rather than similarity measurement. It might be of particular interest if one is considering implementation on a general purpose GPU.

Sharing features: efficient boosting procedures for multiclass object detection [Torralla et al. 2004]

This paper dealt with feature selection in a multi-class detection situation. It was not directly relevant to the subject of similarity measurement.

Approximate nearest neighbor queries in fixed dimensions [Arya and Mount 1993]

This paper discusses the approach of reducing computational complexity by relaxing the requirement for an exact nearest neighbour to an approximate nearest neighbor. An implementation is presented.

An optimal algorithm for approximate nearest neighbor searching fixed dimensions [Arya et al. 1998]

This is an expansion of his earlier work[Arya et al. 1998], but there are more efficient approximate nearest neighbour algorithms to investigate.

[Mahamud 2002]

This paper dealt with determining an optimal distance measure. It could be of further interest if the goal is to improve an implementation.

The Optimal Distance Measure for Object Detection [Mahamud and Hebert 2003]

This paper related to his earlier work [Mahamud 2002] and was not of direct relevance, given the scope of this review.

Algorithms for Nearest Neighbor Search [Indyk 2001]

This was a useful overview presentation of nearest neighbour search algorithms.

 E^2 LSH: Exact euclidean locality-sensitive hashing [Andoni and Indyk 2004]

This web site provides some information about an implementation of LSH, including a manual for the code.

Closest Point Search in High Dimensions [Nene and Nayar 1996]

This paper discussed an approximate search algorithm, but there are more efficient algorithms to investigate.

A Simple Algorithm for Nearest Neighbor Search in High Dimensions [Nene and Nayar 1997]

This paper is related to his earlier work [Nene and Nayar 1996], and demonstrates an improvement over kd-tree.

Computer Vision on the GPU [Fung 2005]

This chapter would be of particular interest if the objective was to implement a vision algorithm on a general purpose GPU.

KD-Tree Algorithms

The following algorithms have been reformatted for the purpose of uniform presentation. They originally appeared in [Freidman et al. 1977].

StartSearch is the root of the recursive search.

STARTSEARCH

- 1 $X_q[1 : k]$ ▷ key values of the query record
- 2 PQD[1:m] ▷ priority queue of the m closest distances encountered at any phase of the search. PQD[1] is the distance to the m^{th} nearest neighbor so far encountered.
- 3 PQR[1:m] ▷ priority queue of the record numbers of the corresponding m best matches encountered at any phase of the search.
- 4 $B_+[1 : m]$ ▷ coordinate upper bounds
- 5 $B_-[1 : m]$ ▷ coordinate lower bounds
- 6 discriminator[1:I] ▷ discriminator at each k-d tree node.
- 7 partition[1:I] ▷ partition value at each k-d treenode. I is the number of internal nodes
- ▷ initialize
- 8 $PQD[1 : m] \leftarrow \infty$
- 9 $B_+[1 : k] \leftarrow \infty$
- 10 $B_-[1 : k] \leftarrow -\infty$
- ▷ search
- 11 SEARCH(root)

Ball Within Bounds detects if it is possible that a closer match might be found outside of the current region of the tree.

BALL WITHIN BOUNDS

- 1 **for** $d \leftarrow 1$ **to** k
- 2 **do if** COORDINATE DISTANCE($d, X_q[d], B_-[d]$) \leq PQD[1]
- 3 or COORDINATE DISTANCE($d, X_q[d], B_+[d]$) \leq PQD[1]
- 4 **then return** FALSE
- 5 **return** TRUE

Bounds Overlap Ball determines if it is possible that a given region of the tree might contain a closer match.

BOUNDS OVERLAP BALL

```

1  sum ← 0
2  for d ← 1 to k
3      do if  $X_q[d] < B_-[d]$ 
4          then ▷ lower than low boundary
5              sum ← sum + COORDINATE DISTANCE(d,  $X_q[d]$ ,  $B_-[d]$ )
6              if DISSIM(sum) > PQD[1]
7                  then return TRUE
8      elseif  $X_q[d] < B_+[d]$ 
9          then ▷ higher than high boundary
10         sum ← sum + COORDINATE DISTANCE(d,  $X_q[d]$ ,  $B_+[d]$ )
11         if DISSIM(sum) > PQD[1]
12             then return TRUE
13 return FALSE

```

Coordinate Distance and Dissim are used to calculate the distance between points.

COORDINATE DISTANCE

▷ Dissimilarity Measure

DISSIM

▷ Dissimilarity Measure

Search is called recursively to visit the tree and find the closest match.

SEARCH(node)

```

1  if node is terminal
2    then ▷ examine records in bucket(node), updating PQD, PQR
3      if BALL WITHIN BOUNDS
4        then done
5        else
6          return
7     $d \leftarrow \text{discriminator}[\text{node}]$ 
8     $p \leftarrow \text{partition}[\text{node}]$ 
9    ▷ recursive call on closer son
10   if  $X_q[d] \leq p$ 
11     then  $temp \leftarrow B_+[d]$ 
12          $B_+[d] \leftarrow p$ 
13         SEARCH(leftson[node])
14     else
15          $temp \leftarrow B_-[d]$ 
16          $B_-[d] \leftarrow p$ 
17         SEARCH(rightson[node])
18          $B_-[d] \leftarrow temp$ 
19   ▷ recursive call on farther son, if necessary
20   if  $X_q[d] \leq p$ 
21     then  $temp \leftarrow B_-[d]$ 
22          $B_-[d] \leftarrow p$ 
23         if BOUNDS OVERLAP BALL
24           then procSearch(rightson[node])
25     else
26          $temp \leftarrow B_+[d]$ 
27          $B_+[d] \leftarrow p$ 
28         if BOUNDS OVERLAP BALL
29           then procSearch(leftson[node])
30          $B_+[d] \leftarrow temp$ 
31   ▷ see if we should return or terminate
32   if BALL WITHIN BOUNDS
33     then done
34     else
35       return

```

Build Tree takes a file containing a complete dataset and builds an optimal KD-Tree.

BUILD TREE(file)

```

1  j, d, maxspread, p
2  if SIZE(subfile) ≤ b
3      then return MAKE TERMINAL(file)
4  maxspread ← 0
5  for j ← 1 to k ▷ find coordinate with greatest spread
6      do if SPRADEST(j, file) > maxspread
7          then maxspread ← SPRADEST(j, file)
8              d ← j
9  p ← MEDIAN(d, file)
10 return MAKE NONTERMINAL(d, p,
                        BUILDTREE(LEFT SUBFILE(d, p, file)),
                        BUILDTREE(RIGHT SUBFILE(d, p, file)))

```

Spreadest, Median, Make Terminal, and Make Nonterminal are used by Build Tree.

SPRADEST(j, subfile)

▷ estimated jth key value spread

MEDIAN(j, subfile)

▷ median of jth key values

MAKE TERMINAL(file)

MAKE NONTERMINAL(d, p, leftnode, rightnode)

Bibliography

- ANDONI, A. AND INDYK, P. 2004. *e²lsh*: Exact euclidean locality-sensitive hashing. <http://web.mit.edu/andoni/www/LSH/index.html>. (p.37)
- ARYA, S. AND MOUNT, D. M. 1993. Approximate nearest neighbor queries in fixed dimensions. In *SODA '93: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms* (Philadelphia, PA, USA, 1993), pp. 271–280. Society for Industrial and Applied Mathematics. (p.36)
- ARYA, S., MOUNT, D. M., NETANYAHU, N. S., SILVERMAN, R., AND WU, A. Y. 1998. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM* 45, 6, 891–923. (p.36)
- ASTARO. Astaro security gateway. http://www.astaro.com/features/email_security.
- AUSTRALIAN CUSTOMS SERVICE. Smartgate. <http://www.customs.gov.au/site/page.cfm?u=5552>. (p.1)
- BEIS, J. S. AND LOWE, D. G. 1997. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *CVPR '97: Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)* (Washington, DC, USA, 1997), pp. 1000. IEEE Computer Society. (pp.10, 14, 30)
- BELLMAN, R. 1961. *Adaptive Control Processes: A Guided Tour*. Princeton University Press. (p.9)
- BENTLEY, J. L. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9, 509–517. (pp.9, 10, 14, 35)
- BHAGAVATHY, S. AND SABAN, M. A. E. 2004. Sketchit: Basketball video retrieval using ball motion similarity. In *PCM (2)* (2004), pp. 256–263. (pp.3, 34)
- BIBBY, C. 2005. Fast feature detection with a graphics processing unit implementation. <http://www.robots.ox.ac.uk/~cbibby/publications/paper994.pdf>. (pp.35, 36)
- BÖHM, C., BERCHTOLD, S., AND KEIM, D. A. 2001. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.* 33, 3, 322–373. (p.35)
- BRIN, S. 1995. Near neighbor search in large metric spaces. In *Proc. 21st Inter. Conf. on Very Large Data Bases* (1995), pp. 574–584. (p.36)
- BUSTOS, B., DEUSSEN, O., HILLER, S., AND KEIM, D. A. 2006. A graphics hardware accelerated algorithm for nearest neighbor search. In *International Conference on Computational Science (4)* (2006), pp. 196–199. (pp.15, 32)

-
- CAI, Y., DE FREITAS, N., AND LITTLE, J. J. 2006. Robust visual tracking for multiple targets. URL: <http://www.cs.ubc.ca/~nando/papers/eccv06.pdf>. To be presented at ECCV06. (pp. 3, 33)
- CLARKSON, K. 1997. Nearest neighbor queries in metric spaces. In *Proc. 39th ACM Symp. Theory Comp. (1997)*, pp. 609–617. (p. 36)
- DATAR, M., IMMORLICA, N., INDYK, P., AND MIRROKNI, V. S. 2004. Locality-sensitive hashing scheme based on p -stable distributions. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry (2004)*, pp. 253–262. (p. 22)
- DAVID G. LOWE. Sift demo program. <http://www.cs.ubc.ca/~lowe/keypoints>. (p. 26)
- FINKEL, R. A., ZASLAVSKY, A., MONOSTORI, K., AND SCHMIDT, H. 2002. Signature extraction for overlap detection in documents. In *Proceedings of the 25th Australasian Computer Science Conference. (2002)*, pp. 59–64.
- FRAUNHOFER INSTITUTE FOR COMPUTER GRAPHICS. Invivo. <http://a7www.igd.fhg.de/projects/invivo/invivo.html>. (p. 1)
- FREIDMAN, J. H., BENTLEY, J. L., AND FINKEL, R. A. 1977. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.* 3, 3, 209–226. (pp. 9, 29, 35, 39)
- FUNG, J. 2005. *GPU Gems 2*, Chapter 40: Computer Vision on the GPU, pp. 649–665. Addison Wesley. (pp. 35, 37)
- GIONIS, A., INDYK, P., AND MOTWANI, R. 1999. Similarity search in high dimensions via hashing. In *The VLDB Journal (1999)*, pp. 518–529. (p. 35)
- GUTTMAN, A. 1984. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data (New York, NY, USA, 1984)*, pp. 47–57. ACM Press. (p. 35)
- INDYK, P. 2001. Dimacs summer school tutorial on new frontiers in data mining: Algorithms for nearest neighbor search. <http://dimacs.rutgers.edu/Workshops/MiningTutorial/pindyk-slides.ppt>. (p. 37)
- INDYK, P. 2004. Nearest neighbors in high-dimensional spaces. In J. E. GOODMAN AND J. O'ROURKE Eds., *Handbook of Discrete and Computational Geometry, chapter 39*. CRC Press. 2nd edition. (p. 35)
- INDYK, P. AND MOTWANI, R. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proc. of 30th STOC (1998)*, pp. 604–613. (p. 35)
- INRIA LEAR. Inria car dataset. <http://lear.inrialpes.fr/data>. (p. 26)
- KATAYAMA, N. AND SATOH, S. 1997. The sr-tree: an index structure for high-dimensional nearest neighbor queries. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data (New York, NY, USA, 1997)*, pp. 369–380. ACM Press. (pp. 11, 14, 36)

-
- KLEINBERG, J. M. 1997. Two algorithms for nearest-neighbor search in high dimensions. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing* (New York, NY, USA, 1997), pp. 599–608. ACM Press. (p.36)
- KÖPPEN, M. The curse of dimensionality. <http://citeseer.ist.psu.edu/524717.html>. (p.9)
- LIU, T., MOORE, A. W., GRAY, A., AND YANG, K. 2005. An investigation of practical approximate nearest neighbor algorithms. In L. K. SAUL, Y. WEISS, AND L. BOTTOU Eds., *Advances in Neural Information Processing Systems 17*, pp. 825–832. Cambridge, MA: MIT Press. (pp.13, 14, 18, 31)
- LOWE, D. G. 1999. Object recognition from local scale-invariant features. In *ICCV '99: Proceedings of the International Conference on Computer Vision-Volume 2* (Washington, DC, USA, 1999), pp. 1150. IEEE Computer Society. (pp.7, 8, 11, 14, 30)
- LOWE, D. G. 2004. Distinctive image features from scale-invariant keypoints. *ijcv* 60, 2, 91–110. (pp.8, 31)
- MAHAMUD, S. 2002. *Discriminative distance measures for object detection*. PhD thesis, Carnegie Mellon University School of Computer Science. Chair-Martial Hebert and Chair-Reid Simmons. (p.37)
- MAHAMUD, S. AND HEBERT, M. 2003. The optimal distance measure for object detection. In *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR) (2003)*. (p.37)
- NENE, S. A. AND NAYAR, S. K. 1996. Closest point search in high dimensions. In *CVPR '96: Proceedings of the 1996 Conference on Computer Vision and Pattern Recognition (CVPR '96)* (Washington, DC, USA, 1996), pp. 859. IEEE Computer Society. (p.37)
- NENE, S. A. AND NAYAR, S. K. 1997. A simple algorithm for nearest neighbor search in high dimensions. *IEEE TPAMI: IEEE Transactions on Pattern Analysis and Machine Intelligence* 19, 989–1003. (p.37)
- OLIVEIRA, R., RIBEIRO, P., MARQUES, J., AND LEMOS, J. 2006. A video system for urban surveillance: Function integration and evaluation. URL: http://homepages.inf.ed.ac.uk/rbf/caviar/papers/tfc_iwiamis_article.pdf. International Workshop on Image Analysis for Multimedia Interactive Systems, Lisbon, April 2004. (pp.3, 33)
- OMOHUNDRO, S. M. 1989. Five balltree construction algorithms. Technical report, International Computer Science Institute, Berkeley, CA. (p.36)
- ROBINSON, J. T. 1981. The k-d-b-tree: a search structure for large multidimensional dynamic indexes. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1981), pp. 10–18. ACM Press. (p.35)
- RTA, NSW. Safe-t-cam. <http://www.rta.nsw.gov.au/heavyvehicles/safety/safetcam/index.html>. (p.1)

- SHAKHNAROVICH, G., DARRELL, T., AND INDYK, P. 2006a. *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice*, Chapter 1, pp. 1–10. MIT Press. (pp. 3, 29)
- SHAKHNAROVICH, G., DARRELL, T., AND INDYK, P. 2006b. *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice*, Chapter 3, pp. 55–65. MIT Press. (pp. 30, 35)
- SHAKHNAROVICH, G., VIOLA, P., AND DARRELL, T. 2003. Fast pose estimation with parameter-sensitive hashing. In *ICCV '03: Proceedings of the Ninth IEEE International Conference on Computer Vision* (Washington, DC, USA, 2003), pp. 750. IEEE Computer Society. (p. 36)
- SPSS INC. Spss clementine. <http://www.spss.com/clementine/index.htm>.
- TONG, C. S. AND WONG, M. 2002. Adaptive approximate nearest neighbor search for fractal image compression. *IEEE Transactions on Image Processing* 11, 6, 605–615. (pp. 3, 34)
- TORRALBA, A., MURPHY, K. P., AND FREEMAN, W. T. 2004. Sharing features: efficient boosting procedures for multiclass object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Volume 2 (Washington, DC, June 2004), pp. 762–769. (p. 36)
- YIANILOS, P. 1993. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. 4th ACM Symp. on Discrete Algorithms* (1993), pp. 311–321. (p. 36)