SCALABLE NEAREST NEIGHBOUR METHODS FOR HIGH DIMENSIONAL DATA

by

Marius Muja

M.Sc., POLITEHNICA University of Timişoara, 2006

B.Sc., POLITEHNICA University of Timişoara, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

The University Of British Columbia

(Vancouver)

April 2013

Abstract

For many computer vision and machine learning problems, large training sets are key for good performance. However, the most computationally expensive part of many computer vision and machine learning algorithms consists of finding nearest neighbour matches to high dimensional vectors that represent the training data. We propose new algorithms for approximate nearest neighbour matching and evaluate and compare them with previous algorithms. For matching high dimensional features, we find two algorithms to be the most efficient: the randomized k-d forest and a new algorithm proposed in this thesis, the priority search k-means tree. We also propose a new algorithm for matching binary features by searching multiple hierarchical clustering trees and show it outperforms methods typically used in the literature. We show that the optimal nearest neighbour algorithm and its parameters depend on the dataset characteristics and describe an automated configuration procedure for finding the best algorithm to search a particular dataset. In order to scale to very large datasets that would otherwise not fit in the memory of a single machine, we propose a distributed nearest neighbour matching framework that can be used with any of the algorithms described in the thesis. All this research has been released as an open source library called FLANN (Fast Library for Approximate Nearest Neighbours), which has been incorporated into OpenCV and is now one of the most popular libraries for nearest neighbour matching.

Preface

All the work in this thesis has been conducted under the supervision of professor David G. Lowe. The following parts of the thesis are based on previously published work:

- Chapter 3 evaluates and compares the approximate nearest neighbour algorithms we found to work best for high dimensional data and proposes an automatic nearest neighbour algorithm configuration procedure. This chapter is based on work presented orally at the International Conference on Computer Vision Theory and Applications (VISAPP) [Muja and Lowe, 2009].
- Chapter 4 introduces and evaluates a new algorithm for efficient nearest neighbour matching of binary features. This chapter is based on work presented orally at the Conference on Computer and Robot Vision (CRV) [Muja and Lowe, 2012] where it received the Best Vision Paper award.

Table of Contents

List of Tables

List of Algorithms

List of Figures

Acknowledgements

First of all, I would like to thank my supervisor David Lowe, who during my time as a graduate student has provided me with invaluable advice and guidance, many years of financial support and has allowed me to explore my own ideas and directions of research. Without David's guidance and support this thesis would not have been possible. I would also like to thank my supervisory committee members, Jim Little and Nando de Freitas, for their advice and feedback.

I would like to thank the colleagues in the Laboratory for Computational Intelligence for the many fruitful conversations we had, in particular to David Meger, Scott Helmer and Ankur Gupta for the numerous useful discussions we had during the years working together and for the fun times we shared while preparing for and competing in the Semantic Robot Vision Challenge. Many thanks to Hoyt Koepke, Jay Turcot and Sancho McCann for being among of the first users of the FLANN library and for providing me with feedback and suggestions on how to improve it.

I am deeply grateful to my loving wife Adriana for her patience, care and support throughout all the years I've spent in graduate school and beyond. I also want to thank my parents, Aurel and Lucia, for encouraging me to pursue my dreams and follow this path, even though they knew it would take me far away from home for a long time.

Chapter 1

Introduction

The most computationally expensive part of many computer vision algorithms consists of searching for the most similar matches to high-dimensional vectors, also referred to as nearest neighbour matching. Having an efficient algorithm for performing fast nearest neighbour matching in large datasets can bring speed improvements of several orders of magnitude to many applications. Examples of such problems include finding the best matches for local image features in large datasets [Lowe, 2004; Philbin et al., 2007], clustering local features into visual words using the k-means or similar algorithms [Sivic and Zisserman, 2003], global image feature matching for human pose estimation [Shakhnarovich et al., 2003], matching deformable shapes for object recognition [Berg et al., 2005] or performing normalized cross-correlation to compare image patches in large datasets [Torralba et al., 2008a]. The nearest neighbour search problem is also of major importance in many other applications, including machine learning, document retrieval, data compression, bio-informatics, and data analysis.

In this thesis we evaluate the most promising nearest-neighbour search algorithms in the literature, propose new algorithms and improvements to existing ones, present a method for performing automatic algorithm selection and parameter optimization and discuss the problem of scaling to very large datasets using compute clusters. We have released all this work as an open source library named FLANN (Fast Library for Approximate Nearest Neighbours).

Figure 1.1: Example of nearest neighbour search in a large dataset of nearly 80 million 32×32 pixel images. The first column contains two query images and the next 3 columns contain the top 16 nearest neighbour matches of each query image in datasets of increasing size (dataset size indicated above each column). It can be observed that as the size the dataset increases (by orders of magnitude) the relevance of the retrieved results is significantly improved. Figure reproduced from [Torralba et al., 2008a], ©2008 IEEE.

1.1 Motivation for Scalable Nearest Neighbour Methods

It has been shown in several papers that using large training sets is key to obtaining good real-life performance from many computer vision methods, particularly from object category recognition algorithms.

The performance improvement brought by using a very large training set of up to 80 million images has been explored in [Torralba et al., 2008a]. An object recognition system consists of two parts, the model and the data. While the large majority of work in the past has been focused on coming up with richer and more complex models, the authors of [Torralba et al., 2008a] show that by using training sets that are orders of magnitude larger than what was typically used in object recognition, simple non-parametric models such as nearest-neighbour classification can obtain results on-par with those of leading class-specific detectors. The performance improvements brought by using larger training sets are exemplified in

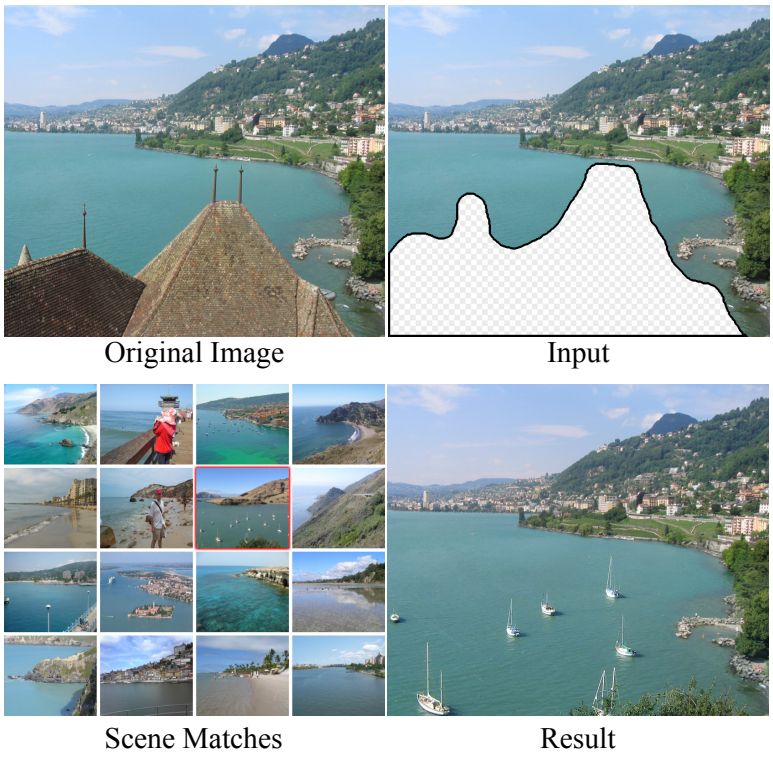

Figure 1.2: Automatic scene completion using millions of photographs. Given an input image and a missing section, simple nearest neighbour techniques and a large training set allow for similar scenes to be found and the missing parts to be seamlessly filled in. Figure reproduced from [Hays and Efros, 2007], ©2007 ACM.

Figure 1.1.

A similar result is demonstrated in [Hays and Efros, 2007], where the authors tackle the task of automatic scene completion using over two million Web gathered images (Figure 1.2). The proposed algorithm seamlessly patches holes in images, by first locating semantically similar images using nearest neighbour matching in a database of gist scene descriptors [Torralba et al., 2003] and then using the most similar matches to extract the missing parts for the image to be completed. The authors note that the results were discouraging when using a smaller dataset of ten thousand images, but increasing the size of the dataset to two million images resulted in a quantitative leap in performance.

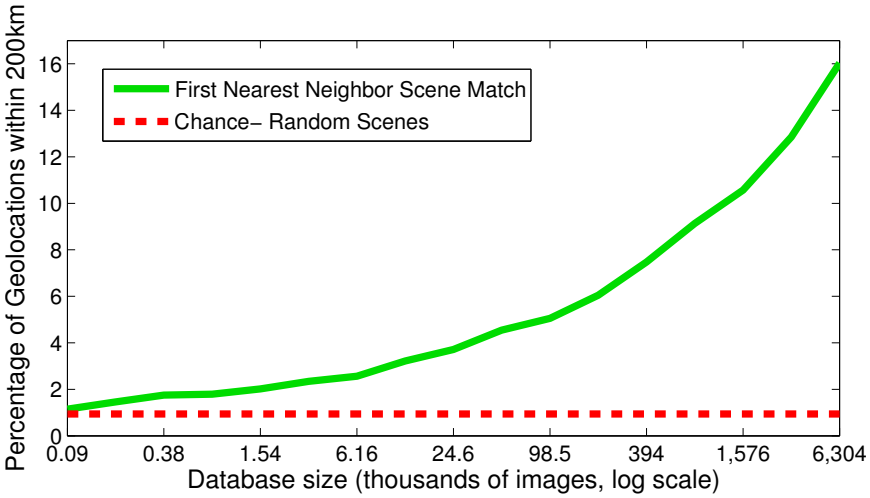

Figure 1.3: Geolocation performance across dataset size of system from [Hays and Efros, 2008], ©2008 IEEE. The percentage of correct geolocations within 200km increases from chance level for small dataset to 18 times as much for the full dataset.

Using a very large database of over 6 million GPS-tagged images, Hays and Efros [2008] propose a system capable of estimating geographic information from an arbitrary image. As previously, simple nearest neighbour matching of an assortment of different descriptors (tiny images, colour and texton histograms, line features, gist features, geometric context) in a very large dataset resulted in very encouraging performance (up to 30 times better than chance). Figure 1.3 illustrates the large impact the size of the dataset has on the overall performance. For small dataset sizes, the performance is just slightly better than chance, but increasing the size of the dataset (by orders of magnitude) leads to a significant performance increase.

Content Based Image Search (CBIS) is another area where using large datasets has a big impact on the quality of the results obtained. In CBIS, a database of images is searched using an image as a query (as opposed to text or some other image meta-information) and the results are database images containing the same or similar objects/scenes as the query image. In [Philbin et al., 2007] the authors present a CBIS system which, when provided with a query in the form of a region

Figure 1.4: Examples of content based images search. The first column contains the query objects, the rest of the columns show the ranked top images retrieved by the system. Figure reproduced from [Philbin et al., 2007], ©2007 IEEE.

of an image, returns a ranked list of images that contain the queried object from a large dataset of over 1 million images (Figure 1.4).

All the above examples show how results are significantly improved when using large training sets. Today the Internet is a vast resource of such training data [Deng et al., 2009], readily available to researchers. However, when working with large datasets, as in the cases mentioned above, the performance of the algorithms employed quickly becomes a key issue and the need for efficient algorithms that can handle such large datasets becomes apparent.

Performing nearest neighbour search is typically a bottleneck in many computer vision applications, so there's a high incentive to make it as efficient as possible in order to speed up the overall algorithm. When working with high dimensional features, as with most of those encountered in computer vision applications (image patches, local descriptors, global image descriptors), there is currently no known generic nearest neighbour search algorithm that is exact and more efficient than linear search. To obtain a speed improvement, many practical applications are forced to settle for an approximate search, where not all the neighbours returned are exact, some are approximate but typically still close or in the same vicinity as

the exact neighbours. In practice it's common for approximate nearest neighbour search algorithms to provide more than 95% correct neighbours and still be two or more orders of magnitude faster than linear search.

In many cases the nearest neighbour search is just a part of a larger application containing other approximations and there is little to no loss in performance from using approximate neighbours in place of exact ones.

1.2 Definitions and Notation

In this thesis we are concerned with the problem of efficient nearest neighbour search in metric spaces. A metric space is a set M on which the notion of distance is defined, $d\colon M \times M \to \mathbb{R}$, and the following hold for $\forall x, y, z \in M$:

- Non-negativity: $d(x,y) \geq 0$,
- Symmetry: $d(x,y) = d(y,x)$,
- Identity: $d(x,y) = 0 \iff x = y$,
- Triangle inequality: $d(x,z) \leq d(x,y) + d(y,z)$.

The nearest neighbour search in a metric space can be defined as follows: given a set of points $P = \{p_1, p_2, ..., p_n\}$ in a metric space M and a query point $q \in M$, find the element $NN(q,P) \in P$ that is the closest to q with respect to a metric distance $d\colon M \times M \to \mathbb{R}$:

$$\mathrm{NN}(q,P) = argmin_{x \in P}\, d(q,x).$$

The *nearest neighbour problem* consists of finding a method to pre-process the set P such that the operation $NN(q,P)$ can be performed efficiently.

We are often interested in finding not just the first closest neighbour, but several closest neighbours. In this case, the search can be performed in several ways, depending on the number of neighbours returned and their distance to the query point: *K-nearest neighbour search* where the goal is to find the closest K points from the query point and *radius nearest neighbour search*, where the goal is to find all the points located closer than some distance R from the query point.

We define the *K-Nearest Neighbour* (KNN) search more formally in the following manner:

$$\mathrm{KNN}(q,P) = A,$$

where A is a set that satisfies the following conditions

$$|A| = K, A \subseteq P,$$

$$\forall x \in A, y \in P - A, d(q,x) \leq d(q,y).$$

The K-nearest neighbour search has the property that it will always return exactly K neighbours (if there are at least K points in P).

The *Radius Nearest Neighbour* (RNN) search can be defined as follows:

$$\mathrm{RNN}(q,P) = \{p \in P, d(q,p) < R\}.$$

Depending on how the value R is chosen, the radius search can return any number of points between zero and the whole dataset. In practice, passing a large value R to radius search and having the search return a large number of points is often very inefficient. *Radius K-Nearest Neighbour* (RKNN) search, is a combination of K-nearest neighbour search and radius search, where a limit can be placed on the number of points that the radius search should return:

$$\mathrm{RKNN}(q,P) = A,$$

such that

$$|A| = K, A \subseteq P,$$

$$\forall x \in A, y \in P - A, d(q,x) < R \text{ and } d(q,x) \leq d(q,y).$$

1.3 Contributions and Thesis Outline

In this thesis we focus on the topic of efficient approximate nearest neighbour search in large datasets. The work presented in the thesis contains several original contributions which can be summarized as follows:

- We have devised a new approximate nearest neighbour search algorithm inspired by the vocabulary tree of Nister and Stewenius [2006] and the GNAT data structure of [Brin, 1995]. The new algorithm, which we call the priority search k-means tree, was found to perform the best for particular datasets.

- We have performed extensive tests comparing the randomized kd-tree algorithm of [Silpa-Anan and Hartley, 2008] to other methods. We have found it to work very well for high dimensional points such as visual local features (such as SIFT, SURF), in some cases being the best algorithm to use for particular datasets.

- We have shown that the optimum algorithm and its parameters are highly dependent on a number of factors such as the data dimensionality, dataset size and statistics. We have proposed an algorithm configuration procedure capable of choosing the optimum algorithm and parameters for a particular dataset while taking into account factors such as index build time, search time and memory usage.

- Binary descriptors are becoming more popular in computer vision due to a series of advantages they have over the more classic vector-based descriptors in terms of computation time, storage cost and matching efficiency. However many nearest neighbour algorithms that work well for vector features are not readily suitable for binary features. We have proposed a new algorithm for fast approximate matching of binary features, based on hierarchical decomposition and clustering of the feature space, and showed it outperforms hashing based methods which are typically used for this purpose.

- It has been shown in the literature that scaling to very large datasets can bring a significant performance boost in many applications. We have addressed the issues brought by scaling to large datasets that would not fit in the main memory of a single machine and have proposed and tested a simple way of distributing the nearest neighbour search operations to a compute cluster.

The thesis is organized into six chapters. We start with an introduction in which we motivate the importance of efficient approximate nearest neighbour search al-

gorithms (current chapter). In Chapter 2 we give the necessary background and discuss the related work.

In Chapter 3 we present the randomized k-d forest and the priority search k-means tree algorithms, perform extensive experiments comparing these algorithms, show that the optimum algorithm is dependent on factors such as input data dimensionality, dataset structure and desired search accuracy. We also propose such an approach for automatically choosing the best algorithm and optimum parameters for each problem instance.

In Chapter 4 we discuss the problem of nearest neighbour matching for binary features and propose a new algorithm which we show to be very efficient, scalable and provides as good or better results than previous algorithms typically used for this purpose.

Scaling to larger datasets has been shown to lead to better performance in a variety of image recognition tasks. When working with very large datasets the memory capacity of most computers quickly becomes a limiting factor when using a single computer. In Chapter 5 we discuss the difficulties of scaling nearest neighbour matching to very large datasets, present a simple solution for distributing the nearest neighbour matching to multiple machines in a compute cluster and conduct several experiments showing the scalability and efficiency of this approach.

Finally, in Chapter 6 we offer conclusions on the topic of approximate nearest neighbour matching and present several promising avenues for future work.

1.4 The FLANN Library

The work in this thesis has been made publicly available as an open source library named FLANN (Fast Library for Approximate Nearest Neighbours) [1] [Muja and Lowe].

FLANN is used by a large number of both research and industry projects (e.g., [Cummins and Newman, 2009], [Havlena et al., 2009a], [Havlena et al., 2009b], [Quigley et al., 2009], [Turcot and Lowe, 2009]), and we have received very positive feedback from the community.

At the moment FLANN is probably the most well known nearest neighbour

[1] http:\people.cs.ubc.ca/~mariusm/flann

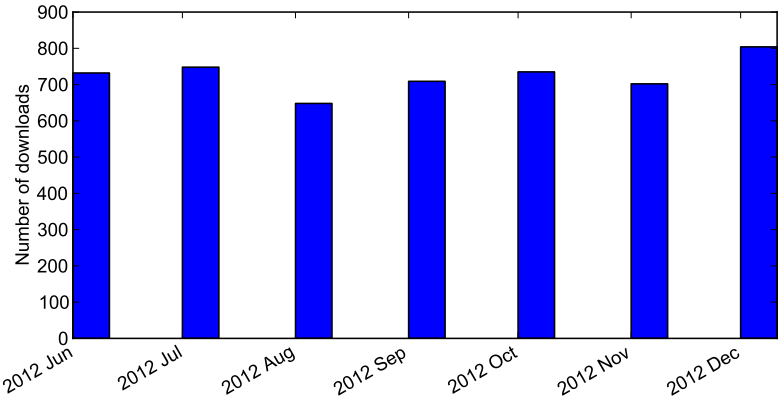

Figure 1.5: Number of FLANN downloads from the library web page. This graph does not include downloads from other sources such as the Linux distributions that package FLANN or its use in OpenCV.

library in the computer vision community, in part due to its inclusion in OpenCV [Bradski and Kaehler, 2008], a popular open source computer vision library. FLANN is used internally by several OpenCV modules, such as the 'features2d', 'calib3d' and 'stitching' modules. FLANN is also a main dependency for the Point Cloud Library (PCL) project, and used extensively throughout the entire library.

FLANN has also been packaged by most of the mainstream Linux distributions such as Debian, Ubuntu, Fedora, Arch, Gentoo and their derivatives.

As with all open-source software, it is difficult to estimate the true number of users due to the fact that there are multiple sources from where people can get the library: download it directly from the library web page, download the source code from the FLANN repository, use the version included in OpenCV or install the package included in the Linux distribution they use. Figure 1.5 shows the number of distinct downloads from the FLANN web page during the second half of 2012. Although not every download necessarily represents an actual FLANN installation, the graph can be used to gauge the interest in the library. We estimate that a large number of users use the version included in OpenCV, so this graph represents only a fraction of the actual users.

1.5 Chapter Summary

In this chapter we have introduced and formally defined the problem of nearest neighbour search and motivated the importance of having efficient algorithms for solving it.

We have shown that for some problems, in particular object and scene recognition, having very large training sets allows for good performance to be obtained using simple non-parametric algorithms. With large training sets, the need for efficient algorithms becomes a key issue. Nearest-neighbour search is part of most computer vision applications and many times it is used as an inner loop inside a more complex algorithm, making it a bottleneck in those applications. Since for high dimensional spaces we don't know any generic exact algorithm that is faster than linear search, it is necessary to use approximate algorithms to speed up the nearest neighbour search.

We conclude the chapter by introducing the FLANN open source library which is a result of the work performed for this thesis.

Chapter 2

Background

In Chapter 1 we show that nearest-neighbour search is a fundamental part of many computer vision algorithms. Consequently, a large body of previous work has been devoted to this area and many nearest neighbour algorithms have been published in the literature. This chapter presents an overview of previous work.

2.1 Nearest Neighbour Matching Algorithms

The large number of nearest neighbour algorithms proposed in the literature can be categorized in many ways: algorithms that perform exact search or approximate search, algorithms that work in metric spaces or require Euclidean spaces, algorithms based on hierarchical space decompositions/partitioning trees or based on various hashing/embedding methods. There is some overlap between these categories and variants of the same algorithm can be part of multiple categories (for example the same algorithm can perform exact search or can be adapted to perform approximate search).

In this section we present some of the best known nearest neighbour algorithms in the literature.

Nearest neighbour algorithms based on partitioning trees have been shown to work effectively in many applications. They recursively decompose the search space into subspaces and construct one or more trees using this decomposition, trees which are then searched to find the nearest neighbours of the query points.

Various algorithms that fall into this category differ on how the space decomposition is made and on how the tree exploration is performed.

K-D Trees and Related Algorithms

Probably the most widely used partitioning tree nearest neighbour algorithm is the *k-d tree* [Bentley, 1975; Friedman et al., 1977], which constructs a binary tree by decomposing the search space using axis aligned hyperplanes that split the data points along the axis with the highest variance. A balanced binary tree is obtained by taking the split value as the median point on the split axis. During search, the tree is traversed from the root to the closest leaf to locate a first nearest neighbour candidate and then traversal is continued by backtracking until all the nodes that may contain nearest neighbour candidates have been explored. During tree exploration a large number of branches which can be determined to be further away than the best neighbour candidate can be skipped (see Figure 2.1).

The k-d tree data structure is very effective for low dimensional features, when during the tree exploration many subspaces can be ignored due to being further away than the best nearest neighbour candidate. However, as the dimensionality increases, more and more of the tree needs to be explored to the point where the

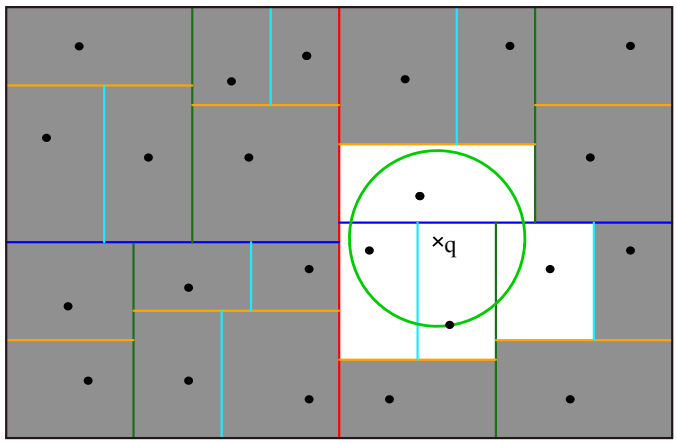

Figure 2.1: Example of a k-d tree in 2D. During tree exploration most of the tree branches can be pruned away because their associated subspaces are further away than the best neighbour candidate found.

majority or even the entire tree needs exploring, making the algorithm no more efficient than linear search.

The decreased performance with the increase in dimensionality, also known under the name of *curse of dimensionality*, is not specific only to the k-d tree, it affects all nearest neighbour search algorithms (as well nearly any algorithm that organizes and analyses high dimensional data). In fact, in high dimensional spaces, no exact nearest neighbour algorithm is currently known to be more efficient than linear search in the general case. This has caused a lot of interest in approximate nearest neighbour (ANN) search algorithms that return a fraction of the exact neighbours (and approximate neighbours for the rest), but which are much faster, in many cases by several orders of magnitude, than linear search.

Arya et al. [1998] modify the original k-d tree algorithm to use it for approximate matching. They impose a bound on the solution accuracy by using the notion of $(1+\varepsilon)$-*approximate* nearest neighbour: a point $p \in X$ is an $(1+\varepsilon)$-approximate nearest neighbour of a query point $q \in X$, if $dist(p,q) \leq (1+\varepsilon)dist(p^*,q)$ where p^* is the true nearest neighbour. The authors also propose the use of a priority queue to speed up the search in a tree by visiting tree nodes in order of their distance from the query point. This method of approximating the nearest neighbour search is also referred to as "error bound" approximate search.

Another way of approximating the nearest neighbour search is by limiting the time spent during the search, or "time bound" approximate search. This method is proposed in [Beis and Lowe, 1997] who use a k-d tree data structure and stop the search early after examining a fixed number L_{max} of leaf nodes. In practice this time-constrained approximation criterion has been found to give better results than the error-constrained approximate search.

Silpa-Anan and Hartley [2008] propose the use of multiple randomized k-d trees as a means to speed up approximate nearest-neighbour search. [Muja and Lowe, 2009] examine the randomized k-d trees data structure and perform more exhaustive tests which show that it's a very effective data structure for matching high dimensional data.

Sproull [1991] proposes the PCA-tree in which the partitioning hyperplanes don't have to be aligned with the coordinate axis, but can have arbitrary orientations. The partitioning hyperplanes are picked to be perpendicular on the principal

eigenvector of the covariance matrix in order to obtain a better decomposition of the search space. The paper shows that on some datasets such a decomposition requires fewer than half of the terminal nodes having to be examined, compared to the axis aligned decomposition. Although the PCA-tree provides a better space decompositions than the kd-tree, by choosing non-axis aligned splitting hyperplanes, searching a PCA-tree is more costly because it requires projecting the query point onto the direction perpendicular on the splitting hyperplane. This projection is a straightforward $O(1)$ operation for the k-d tree decomposition, but requires an inner product (consisting of $O(d)$ multiplications and $O(d)$ additions) for a non-axis aligned decomposition. Due to this overhead, in practice, for many high dimensional problems such as local features matching, kd-trees achieve better accuracy in a given search time.

Dasgupta and Freund [2008] introduce the random projection trees (RP-trees) in which the splitting hyperplanes are chosen to be random directions on the unit hypersphere and instead of splitting at the median, a small amount of "jitter" is added to the splitting value. The paper shows that RP-trees automatically adapt to the intrinsic low dimensionality structure in the data, without having to explicitly learn this structure. RP-trees also have the overhead of projecting the query point onto the axis perpendicular on the splitting hyperplane, making them less efficient than k-d trees in practice, for many high dimensional problems.

Jia et al. [2010] also try to improve on the way the k-d tree partitions the space by using non axis aligned splitting hyperplanes. Similar to the PCA tree, the splitting hyperplanes are chosen to be perpendicular on directions with high data variance constructed as a weighted axis combination using only weights from the set $\{-1,0,1\}$. Using only the weights -1, 0 and 1 allows for fast evaluation of the query projection during tree search by reducing an inner product operation to a simple sum. In our experiments, we haven't found this to be any more efficient than a randomized k-d tree decomposition, as the overhead of evaluating multiple dimensions during search overcame the benefit of the better space decomposition.

Clustering Partitioning Trees

The algorithms presented so far in this section construct binary trees by using hyperplanes to recursively split the data points. Another type of partitioning trees have been proposed in the literature that are constructed by recursively partitioning the data points using various clustering algorithms.

Fukunaga and Narendra [1975] propose a partitioning tree constructed by clustering the data points using the k-means algorithm into k disjoint groups and then recursively doing the same for each of the groups. The resulting hierarchical k-means tree requires that the data points exist in a vector space because the mean of each cluster needs to be computed.

Brin [1995] proposes a similar tree, called GNAT (Geometric Near-neighbour Access Tree), in which some of the data points are used as the cluster centres instead of computing the cluster mean points. The resulting partitioning tree does not require the points to be located in an Euclidean space in which the dimensions can be accessed independently, instead it only assumes the points are located in a metric space.

Moore [2000] proposes the anchors hierarchy, a data structure and algorithm to efficiently partition data points located in a generic metric space and construct a ball-tree like metric tree. It shows that this data structure can be used to accelerate some statistical learning algorithm even for high dimensional data.

Yianilos [1993] introduces another metric partitioning tree, the *vp-tree* (vantage point tree). Similar to the k-d tree, the vp-tree is a binary tree, but instead of using coordinate values for dividing the points, the vp-tree uses the distance to a selected vantage point. The points are split in two sets, the ”near” and ”far” points and the procedure is repeated for each set in order to construct the tree. Kumar et al. [2008] show that vp-trees can be used to efficiently compute exact nearest neighbours for image patches.

Cover trees [Beygelzimer et al., 2006] are another data structure designed to accelerate nearest neighbour search in metric spaces. A cover tree is a leveled tree structure where each point represents a node and resides on a numbered level. The nodes of the tree must obey a set of invariants that dictate how nodes need to be separated on each level, which nodes on higher levels can be parents of nodes on lower

levels and how the levels are nested to cover the dataset at finer scales as the tree is descended. Cover trees have been shown to accelerate exact nearest neighbour search for spaces with low intrinsic dimensionality, defined in terms of a measure called the expansion constant of the space, and provide theoretical guarantees on the performance based on this expansion constant. Cover trees can also be adapted for ε-*approximate* search, however we have found that in practice for large dimensional points such as visual local features (SIFT, SURF, GIST) their performance is much lower than that of other approximate nearest neighbour methods such as the randomized kd-trees or the priority search k-means tree.

Liu et al. [2004] propose a metric tree that allows an overlap between the children of each node, called the spill-tree. The reason for the overlap is to mitigate the boundary effect of a point being located just outside the boundary of the nearest cells during the tree exploration. Although this approach has some benefits we have found that it requires significantly more memory while the performance is lower than that of the randomized k-d trees.

Approximate search is involved when there is a data dimensionality reduction step that projects the data points into a lower dimensionality space. The projection into the lower dimensionality space typically introduces a distortion of the relations between the points and, as a consequence, the search (exact or approximate) in the lower dimensionality space becomes approximate in the original space. Liu et al. [2004] use this technique of projecting into a lower dimensionality space, but perform multiple rounds of random projections to reduce the error introduced by the dimensionality reduction and increase the probability of finding the correct neighbour.

Mikolajczyk and Matas [2007] evaluate the nearest neighbour matching performance for several tree structures, including the k-d tree, the hierarchical k-means tree, and the agglomerative metric tree (ball tree). They conclude that the k-d trees have very good performance on near queries and very low on far ones, while the performance of metric trees is more consistent. We have obtained similar results in Chapter 3.

Hashing-based Nearest Neighbour Techniques

Perhaps the best known hashing based nearest neighbour technique is the locality sensitive hashing (LSH) [Andoni and Indyk, 2008]. LSH can also be regarded as a form of probabilistic dimensionality reduction. LSH uses a large number of hash functions with the property that the hashes of vector points that are close to each other are likely to also be close. At query time the hashes of the query vectors are computed and the dataset points whose hashes fall within the same bins as the query hashes are examined.

Multi-probe LSH [Lv et al., 2007] was proposed to improve the high storage costs of LSH due to the fact that it requires a large number of hash tables to achieve good performance. By probing multiple buckets that are likely to contain query matches, multi-probe LSH manages to reduce the number of hash tables required by up to an order of magnitude.

In [Bawa et al., 2005] the authors propose the LSH Forest, a data structure which replaces each hash table in the classic LSH algorithm with a prefix tree. The prefix trees can hold hash keys of variable length, allowing the LSH forest to adapt better to the data, compared to the classic LSH for which the key length parameter needs to be hand tuned. The LSH Forest is shown to outperform LSH for similarity search in the text domain.

The performance of hashing methods is highly dependent on the quality of the hashing functions they use. Consequently, a large body of research has been targeted at improving hashing methods by using data-dependent hashing functions computed using various learning techniques: parameter sensitive hashing [Shakhnarovich et al., 2003], spectral hashing [Weiss et al., 2008], randomized LSH hashing from learned metrics [Jain et al., 2008], kernelized LSH [Kulis and Grauman, 2009], learnt binary embeddings [Kulis and Darrell, 2009], shift-invariant kernel hashing [Raginsky and Lazebnik, 2009], semi-supervised hashing [Wang et al., 2010], optimized kernel hashing [He et al., 2010] and complementary hashing [Xu et al., 2011].

The different LSH algorithms provide theoretical guarantees on the search quality and have been successfully used in a number of projects, however our experiments show that for real data they are usually outperformed by algorithms us-

ing space partitioning structures such as the randomized k-d trees and the priority search k-means tree.

Hashing methods are particularly convenient in case of binary data points, compared to the vector based data points, which require an extra step of binarization. Hashing algorithms have been shown to be effective at matching binary visual features, which have recently been gaining popularity in the computer vision community. In Chapter 4 we compare the multi-probe LSH algorithm with a newly proposed hierarchical clustering algorithm on a dataset of binary features.

Nearest Neighbour Graph Techniques

Nearest neighbour graph methods build a graph structure in which points are vertices and edges connect each point to its nearest neighbours. When a query point is given, this graph is explored in different manners in order to get closer to the nearest neighbour of the query point. In [Sebastian and Kimia, 2002] the authors select a few well separated elements in the graph as "seeds" and start the graph exploration from those seeds in a best-first fashion. Similarly, Hajebi et al. [2011] perform a best-first exploration of the k-NN graph, but using a hill-climbing strategy and picking the starting points at random. They present recent experiments that compare favourably to randomized KD-trees, so the proposed algorithm should be considered for future evaluation and possible incorporation into FLANN.

The nearest neighbour graph methods suffer from a quite expensive construction of the k-NN graph structure. Wang et al. [2012] improve the construction cost by building an approximate nearest neighbour graph. This is accomplished by hierarchically and randomly dividing the data into subsets and building exact neighbourhood graphs for each subset. The procedure is repeated several times to generate multiple neighbourhood sub-graphs which are combined to obtain a more accurate approximate neighbourhood graph.

2.2 Nearest Neighbour Search in Large Datasets

Nister and Stewenius [2006] present a fast method for nearest-neighbour feature search in very large databases. Their method is based on accessing a single leaf node of a hierarchical k-means tree similar to that proposed by Fukunaga and

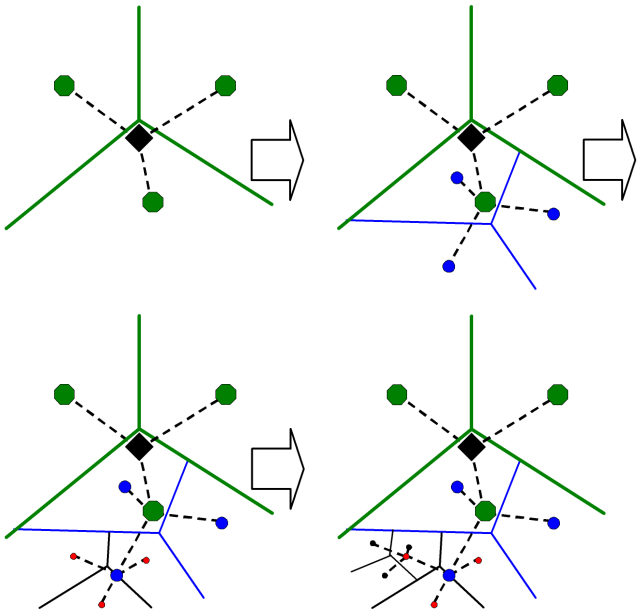

Figure 2.2: Vocabulary Tree of [Nister and Stewenius, 2006] @2006 IEEE exemplifying a hierarchical k-means decomposition with branching factor 3.

Narendra [1975]. Due to the fact that tree is traversed a single time without any backtracking, a large number of neighbours have to be retrieved in order to obtain good overall performance. Figure 2.2 illustrates the k-means hierarchy used to traverse to a leaf node.

In [Leibe et al., 2006] the authors propose an efficient method for clustering and matching features in large datasets. They compare different partitional and agglomerative clustering methods such as k-means and agglomerative clustering and compare the quality of the resulting clusters. They propose a mixed partitional-agglomerative clustering algorithm that is more efficient while producing clusters of similar quality to that of an agglomerative clustering algorithm. They propose a ball-tree data structure for fast matching in high dimensional feature spaces.

Torralba et al. [2008b] use a boosting algorithm and restricted Boltzmann machines to compute a small binary code for each element of the dataset. The use of small codes allows them to perform experiments on datasets containing millions

of images using a standard PC. Weiss et al. [2008] compute similar binary codes by using a different technique (spectral hashing) which they show to be superior to that of [Torralba et al., 2008b]. Jégou et al. [2008] use binary signatures to refine quantized SIFT descriptors in a bag-of-features image search framework.

Jégou et al. [2010] introduce a product quantization approach as a way of computing short codes. The idea behind product quantizers is to construct a quantizer of the original space as a Cartesian product of lower dimensionality quantizers and represent each vector as short code composed of the quantization indices of its subparts. The authors show that this approach outperforms those of [Torralba et al., 2008b] and [Weiss et al., 2008].

In [Schindler et al., 2007] the authors show that the size of the dataset can be increased while maintaining good query performance by building a vocabulary using only the most informative features, effectively increasing the discriminative power of the visual words. The authors also propose a different search algorithm in a hierarchical k-means tree, called Greedy N-Best Paths (GNP), which follows multiple branches at each level, however they don't prioritize which branches are explored first by their proximity to the query vector.

Philbin et al. [2007] present a large-scale object retrieval system that, when provided with a query in the form of a region of an image, returns a ranked list of images that contain the same object from a large dataset of over 1 million images. The authors focus on scalable methods of building a visual vocabulary and compare two different implementations, the hierarchical vocabulary of [Nister and Stewenius, 2006] and an approximate flat vocabulary built using randomized kd-trees. The results show that the flat approximate vocabulary outperforms the hierarchical vocabulary.

In [Torralba et al., 2008a] the authors explore how recognition performance can be improved by increasing the amount of training data and ask themselves how big the dataset has to be to robustly perform recognition using simple nearest-neighbour schemes. They collect a dataset of nearly 80 million tiny images (32 x 32 pixels), representing a dense sampling of the non-abstract nouns in Wordnet [Stark and Riesenfeld, 1998], and show that for richly represented classes, simple non-parametric methods such as nearest-neighbours give similar performance to class specific detectors.

Deng et al. [2009] introduce ImageNet, an ongoing effort to create a large scale ontology of images built upon the structure of WordNet, aiming to populate the majority of the 80,000+ noun synsets of WordNet with an average of 500-1000 full-resolution images, resulting in tens of millions of annotated images organized by the semantic hierarchy of WordNet. The authors show the usefulness of ImageNet through a series of experiments including object recognition, image classification and object localization experiments and confirm the findings of Torralba et al. [2008a], that performance can be significantly improved by enlarging the dataset.

2.3 Nearest Neighbour Algorithms for Binary Features

Matching binary features is of increasing interest in the computer vision community with many binary visual descriptors being recently proposed in the literature (BRIEF [Calonder et al., 2010], ORB [Rublee et al., 2011], BRISK [Leutenegger et al., 2011]). Binary features are attractive because of several advantages they have over the more established vector based features such as SIFT [Lowe, 2004] and SURF [Bay et al., 2006]: they are often very cheap to compute, usually are more compact to store, and matching them with the Hamming distance can be done efficiently by using fast bitwise operations.

There is also a lot of work published on the topic of learning short binary codes through Hamming embeddings from different feature spaces. Salakhutdinov and Hinton [2009] introduce the notion of semantic hashing when they learn a deep graphical model that maps documents to small binary codes. When the mapping is performed such that close features are mapped to close codes (in Hamming space), the nearest neighbour matching can be efficiently performed by searching for codes that differ by a few bits from the query code. If the resulting codes are small enough to be used as indices (addresses) in a hashing table, this can be done very efficiently.

Torralba et al. [2008b] use a similar approach by learning compact binary codes from images with the goal of performing real-time image recognition on a large dataset of images using limited memory. Weiss et al. [2008] formalize the requirements for good codes and introduce a new technique for efficiently computing binary codes. Other papers in which a Hamming embedding is computed from SIFT

visual features are those of Jégou et al. [2008] and Strecha et al. [2010].

Performing approximate nearest neighbour search by examining all the points in a Hamming radius works efficiently when the distance between the matching codes is small. When this distance increases, the number of points in the Hamming radius gets exponentially larger, making the method impractical. This is the case for visual binary features such as BRIEF or ORB where the minimum distance between matching features can usually be larger than 20 bits. In cases such as this exploring the Hamming radius of the query point is impractical and other nearest neighbour matching techniques must be used.

Because computing Hamming distances between binary features can be done very efficiently, in many previous publications linear search is the method of choice for matching binary features. This is only practical for small to medium size datasets and does not scale to large datasets, where it quickly becomes a bottleneck.

Locality Sensitive Hashing (LSH) [Gionis et al., 1999] has been shown to be effective for fast matching of binary features in [Rublee et al., 2011] where it is used to match ORB visual descriptors.

Min-hash [Chum et al., 2008, 2009] is an instance of Locality Sensitive Hashing for sets, in which the elements are compared using the Jaccard similarity (the ratio between the cardinality of the intersection and cardinality of the union of the sets). Zitnick [2010] uses the min-hash technique for matching a newly introduced binary patch descriptor, transforming each binary descriptor into a set by considering the indices assigned to 1 in the descriptor as the set elements.

Norouzi et al. [2012] introduce a way of performing exact K-nearest neighbour search in Hamming spaces by building multiple hash tables on disjoint substrings of the binary codes. By using short disjoint substrings (less than 32 bits), the matching in the multiple hashing tables is done efficiently by direct indexing with the substring codes. The proposed algorithm is shown to have sub-linear run-time behaviour for uniformly distributed codes.

Many algorithms for fast matching typically used for vector-based descriptors, such as kd-trees or k-means trees, are not readily suitable for binary descriptors because they assume the features exist in Euclidean space where dimensions of the features can be individually averaged. It's possible to use these methods for

binary features by considering them in a vector space where each bit is a numerical value and match them using the Manhattan distance, however doing so results in high dimensional features with lower matching performance compared to that of the original binary features matched with the Hamming distance.

Since the Hamming distance is a metric distance, algorithms designed to work in metric spaces, such as metric trees, can be used for matching binary features. Muja and Lowe [2012] propose the use of multiple hierarchical decomposition trees for matching binary features and show that this approach can outperform hashing methods such as multi-probe LSH, as will be described in Chapter 4.

Trzcinski et al. [2012] show that the Voronoi diagrams in binary spaces have thick boundaries, meaning that a large fraction of points is equidistant from two random points, leading to bad performance from many ANN algorithms, which assume points can be well clustered together. Based on this observation, they propose two algorithms optimized for search in binary spaces: the Parc-trees which is based on a hierarchical decomposition of the space similar to that of [Muja and Lowe, 2012], but without a common priority queue for searching, and the Uniform LSH, a variation of LSH in which the bits that form the keys are distributed more uniformly.

2.4 Automatic Configuration of Nearest Neighbour Algorithms

There have been hundreds of papers published on nearest neighbour search algorithms, but there has been little systematic comparison to guide the choice among algorithms and set their internal parameters. One reason for this is that the relative performance of the algorithms varies widely based on properties of the datasets, such as dimensionality, correlations, clustering characteristics, and size.

For optimum performance, the algorithm used and its parameters should be tuned for each case. There has been extensive research on *parameter tuning* methods, or in a more general sense *algorithm configuration*, however these methods have not been applied for finding optimum parameters for nearest neighbour algorithms. In practice, and in most of the nearest neighbour literature, setting the algorithm parameters is usually a manual process carried out by using various heuristics

and rarely more systematic approaches.

The use of a systematic approach for automatically choosing the optimal nearest neighbour algorithm and its parameters has many benefits such as the possibility of comparing different algorithms in a more meaningful way, allowing users to select the optimum algorithm for a specific problem and allowing users not familiar with different algorithms to use them with the optimum parameters instead of the default ones (which in many situations are far from optimal).

The difficulty of finding optimum parameters for an algorithm depends mainly on the number and type of the parameters (categorical, ordinal, continuous). When there are few parameters, usually the most convenient is to pick a limited set of possible values for each parameter and try all possible combinations of parameters, approach also known as *grid search* or *full factorial design*. When the experiments involve only a selected subset of all the possible combinations, the approach is known as *fractional factorial design*. With an increase in the number of parameters, the grid search is no longer feasible due to the exponential growth in the number of possible parameter combinations. Luckily, most nearest neighbour search algorithms have a relatively small number of parameters, making the grid search approach feasible [Muja and Lowe, 2009].

Bawa et al. [2005] show that the performance of the standard LSH algorithm is critically dependent on a parameter k that determines the length of the hashing key. The optimum value for this parameter depends on multiple factors, such as the data distribution, size of dataset and the number of nearest neighbours, and needs to be tuned for each case when one of these parameters changes beyond a certain threshold. A new algorithm, LSH Forest, is proposed that eliminates some of the data dependent parameters of LSH.

Muja and Lowe [2009] show that the randomized k-d trees and the hierarchical k-means algorithms' performance also depends on a series of parameters such as the number of parallel trees (for randomized k-d trees) or branching factor (for hierarchical k-means). The optimum parameters are shown to be dependent on data distribution as well. An automatic algorithm configuration method is proposed that can determine the optimum algorithm and its parameters by combining grid search with a finer grained Nelder-Mead downhill simplex optimization process [Nelder and Mead, 1965].

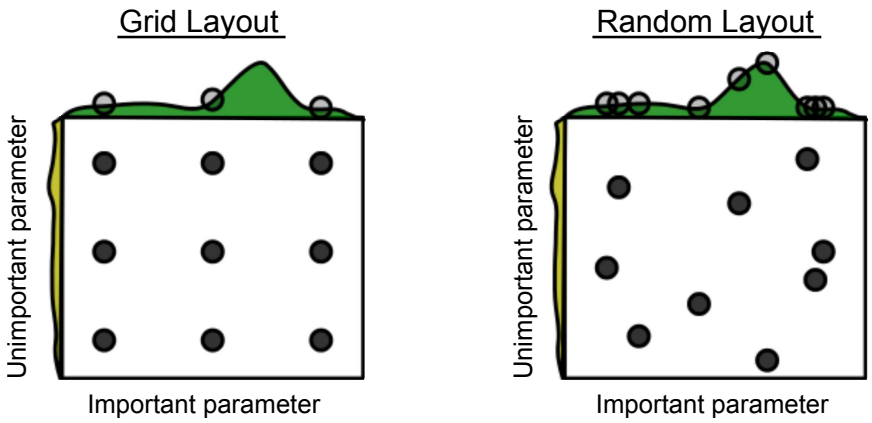

Figure 2.3: Example from [Bergstra and Bengio, 2012] illustrating why random search is preferable to grid search for parameter spaces with low effective dimensionality. With grid search the important dimensions are tested in fewer places than with random search.

In the remainder of this section we present several methods for parameter tuning, previously proposed in the literature, that could be used for nearest neighbour algorithms. [Hutter, 2009] can be consulted for an in-depth review of the algorithm configuration literature.

Coy et al. [2001] introduce a search based parameter optimization procedure for a set of problems by running a parameter optimization routine on a small subset (analysis set) and then combining (averaging) the resulting parameters. For each problem instance in the analysis set they perform a set of experiments using full or fractional factorial design with two values for each parameter, followed by a gradient search on the response surface approximated from the experimental runs.

Adenso-Diaz and Laguna [2006] propose the CALIBRA system that uses a similar approach of experimental design, followed by a local search. In a first phase it uses a full factorial design with two values per parameter, followed by a second local search phase which evaluates nine parameter configurations around the best configuration found so far in a increasingly denser pattern. Once a local optimum is found the search is restarted with a coarser grid, until a maximum number of total experiments is performed. The system is limited at optimizing a maximum of five parameters and can only handle ordinal parameter types.

Bergstra and Bengio [2012] show that, except for very small parameter spaces, random search can be a more efficient strategy for parameter optimization than grid search or manual parameter selection. The reason is that some parameters are more important to tune than others and while grid search has an even coverage in the high dimensional parameter space, when projected into the lower sub-dimensional space of the more important parameters, it has a poor coverage. An intuitive representation of this can be seen in Figure 2.3.

Hutter et al. [2009] introduce ParamILS, an algorithm configuration method based on iterated local search in the parameter space. ParamILS is initialized with a combination of a default and random parameter configurations and it constructs a chain of local optimum parameter configurations inside a main loop. The parts of this loop are, (i) a solution perturbation to escape from local optima, (ii) an iterative first improvement local search procedure and (iii) an acceptance criterion for accepting better or equally well performing parameter configurations. In addition the search is re-initialized at random with a $p_{restart}$ probability.

2.5 Chapter Summary

In this chapter we summarized related work in the field of nearest neighbour search. Among the papers referenced, we emphasized the type of algorithms used (partitioning tree, hashing, nearest-neighbour graphs), whether it works in a metric space or requires an Euclidean vector space and whether it does exact or approximate search. We have also emphasized the importance and usefulness of approximate nearest neighbour search algorithms in most practical applications.

Next, we presented research related to nearest neighbour matching in large datasets. This included algorithms such as the vocabulary tree, ball-tree structures, k-means trees, short code embeddings using several techniques (spectral hashing, product quantizers). As shown in the introduction chapter, increasing the size of the dataset is key for improved performance in a variety of applications.

We then reviewed existing literature on the topic of fast matching of binary features. Binary features have seen increased popularity in the recent literature due to the fact that they allow fast computation and efficient storage and matching.

Finally, we presented several approaches for algorithm configuration that could

be applied for finding optimum nearest neighbour algorithms for a specific problem by tuning their parameters.

Chapter 3

Fast Approximate Nearest Neighbour Matching

In this chapter we discuss the problem of fast approximate nearest neighbour matching and present the algorithms that we found to be the most effective for efficiently finding nearest neighbour matches for high dimensional points.

For high-dimensional spaces, there are no generic nearest-neighbour search algorithms known that are exact and more efficient than linear search. As linear search is too costly for many applications, this has generated an interest in algorithms that perform approximate nearest-neighbour search, in which non-optimal neighbours are sometimes returned. Such approximate nearest neighbour (ANN) search algorithms can be orders of magnitude faster than exact search, while still providing near-optimal accuracy.

Most ANN algorithms have a way to trade-off between the search time and accuracy (when accepting a lower accuracy the search is faster and when demanding a high accuracy the search will take longer). The trade-off can usually be controlled by one or more algorithm parameters and by varying these parameters we can analyse how well an approximate nearest-neighbour algorithm performs.

To measure the accuracy of an approximate nearest neighbour algorithm we use the term *search precision* or just *precision*, which we define as the fraction of the neighbours returned by the approximate algorithm which are exact neighbours (identical to what linear search would return). The search time of an algorithm

can be measured literally, as the time required to perform a search, however we find that it's better to express it as the time required to perform a linear search divided by the time required to perform the approximate search and we refer to it as the *search speedup* or just *speedup*. Using the ratio between the linear search time and the approximate search time, in addition to providing a useful measure of the speedup of approximate search compared to the exact linear search, has the advantage of being more invariant than the absolute time with respect to things like the speed of the computer running the experiments.

Repeated distance computations between the query point and datasets points are the most expensive part of most nearest neighbour search algorithms, so in order to be more efficient most algorithms try to "look at" as few points in the dataset as possible. To do that and still return a large fraction of the correct neighbours, the search algorithms have various strategies of choosing which points in the dataset to examine and which to ignore.

There are two strategies of controlling the level of approximation during search: *an error bound search* and a *time bound search*. In the error bound case, the requirement of finding the closest neighbour is relaxed and the algorithm returns any point that is within a factor of $(1+\varepsilon)$ of the true nearest neighbour, also known as $(1+\varepsilon)$-approximate nearest neighbour. When using $(1+\varepsilon)$-approximate search, parts of the dataset can be discarded more quickly compared to regular search, resulting in a faster search at the expense of search accuracy. The other strategy, of time constrained search, consists of terminating the search early, typically before a predetermined number of dataset points have been visited. We have found the time constrained search to give better results than $(1+\varepsilon)$-NN search and for this reason we are using this approach in the experiments in this chapter and by default in FLANN (although $(1+\varepsilon)$-approximate search is present as an optional parameter).

We have evaluated many different algorithms for approximate nearest neighbour search on datasets with a wide range of dimensionality. In our experiments, one of two algorithms obtained the best performance, depending on the dataset and desired precision: the *priority search k-means tree* or the *multiple randomized k-d trees*. We show that for a particular dataset the algorithm type and it's parameters have a big impact on the performance and that an automatic algorithm configuration procedure can greatly improve performance.

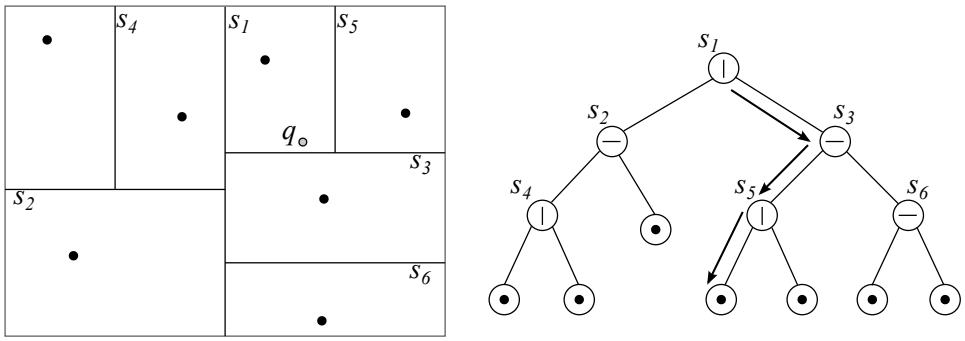

Figure 3.1: Example of a kd-tree decomposition and the associated binary tree data structure. The arrows indicate the way the tree is traversed to locate the first nearest neighbour candidate for the query point q.

3.1 The Randomized k-d Tree Algorithm

This section gives a brief introduction to the classic k-d tree algorithm and describes how it can be used for approximate search. It then presents the multiple randomized k-d tree version which we found to be very effective for some high dimensional problems.

The classic k-d tree

The classical k-d tree [Bentley, 1975; Friedman et al., 1977] is one of the best known nearest-neighbour search algorithms and was shown to be very efficient for low dimensionality data. The k-d tree is a space partitioning data structure build by splitting the data using axis-aligned hyperplanes perpendicular on the axis on which the data has the highest variance. By recursively applying this splitting procedure, a binary tree is constructed in which the inner nodes represent axis-aligned hyperplane and the leaf nodes represent the points in the dataset (see Figure 3.1). Each branch of an inner node corresponds to a rectangular region of the search space bounded by the hyperplanes corresponding to the ancestor nodes.

During search, the tree is traversed from the root to a leaf node by choosing at each inner node the branch representing the region of space containing the query point. The point in the leaf node becomes a first nearest neighbour candidate and the search is continued by backtracking into the tree. Any branches that are further

away then the distance to the nearest neighbour candidate can be ignored as it is certain that they will not contain closer points, and the nearest neighbour candidate is updated any time a point that's closer to the query is located. The search ends when the entire tree has been explored or all the remaining branches can be pruned away due to being too far.

The above algorithm works very well for low dimensional data, managing to quickly ignore most of the tree branches and to find the exact nearest neighbour very efficiently. As the dimensionality increases however, the number of cells that the algorithm has to visit quickly increases (the number of adjacent cells in the tree increases exponential with the data dimensionality) to the point where the algorithm needs to visit nearly all the leaf nodes. When the algorithm has to visit almost all leaf nodes, it becomes even slower than linear search due to the overhead of traversing the tree.

Approximate search in a k-d tree

To obtain a speedup over linear search it is necessary to settle for an approximate nearest-neighbour search algorithm, which can greatly improve the search speed at the cost of not always returning the exact nearest neighbours. The exact k-d tree search can be transformed into an approximate search by using either the error constrained $(1+\varepsilon)$-search or the time constrained search. Below we describe the time constrained search, as we found it to be more effective in practice.

For the approximate search, after descending the tree once, the algorithm doesn't backtrack through the tree as in the case of the exact search. Instead, during the descend phase all the unexplored branches are placed into a priority queue ordered by the distance from the query point to the region corresponding to the branch and after a leaf is reached the search is resumed from the branch on top of the priority queue. The search is terminated when a fixed number of leafs nodes have been visited and the neighbour candidate at that point is returned.

The use of the priority queue was first proposed in Arya et al. [1998] and it has the advantage that it increases the likelihood of visiting closer neighbours early in the search process. This makes the search more efficient by allowing the algorithm to prune away more branches during the remainder of the search. Because

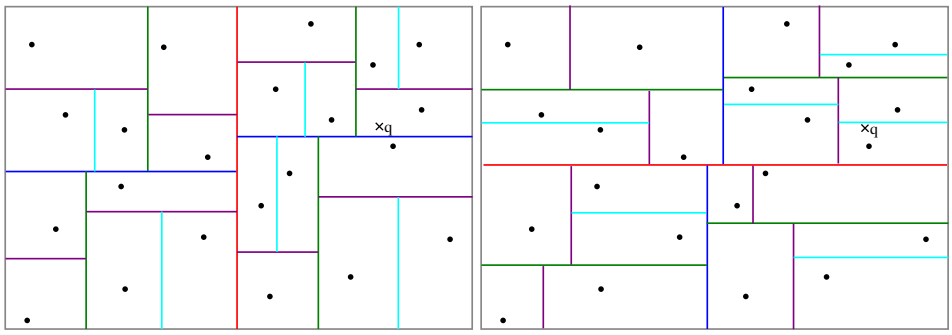

Figure 3.2: Example of randomized kd-trees. The nearest neighbour is across a decision boundary from the query point in the first decomposition, however it's in the same cell in second decomposition.

the search is stopped early, the answer returned is not always the correct nearest neighbour, but the use of the priority queue makes it likely that even when it's not the closest point, it's still a close one.

3.1.1 Algorithm description

When using approximate search in a single k-d tree, the data dimensionality has a significant impact on the accuracy of the results. The season for this can be understood by imagining the case when the query point is close to one of the splitting hyperplanes. In that case the nearest neighbour lies with almost equal probability on the other side of the hyperplane as on the same side as the query point and further exploration of the tree is required before the cell containing the nearest neighbour will be visited. In the case of the approximate search when the exploration tree exploration is stopped early, the cell with the closest neighbour might not get visited at all.

Silpa-Anan and Hartley [2008] proposed the use of multiple randomized k-d trees (or randomized k-d forest) to improve the accuracy of the results when using high dimensional data. As illustrated by Figure 3.2, with multiple randomized k-d tree decompositions when the closest neighbour happens to lie across a decision boundary from the query point, it's possible that in a different decomposition the two points will be in the same cell.

The randomized k-d trees are build in a similar manner to the classic k-d tree

described above, with the difference that the classic kd-tree algorithm splits data on the dimension with the highest variance, but for the randomized k-d trees the split dimension is chosen randomly from the top N_D dimensions with the highest variance. We used the fixed value $N_D = 5$ in our implementation, as this performs well across all our datasets and does not benefit significantly from further tuning.

Algorithm 1 Building the randomized k-d forest

Input: features dataset D, number of trees N
Output: the randomized kd-trees data structure
procedure BUILDRANDOMIZEDKDTREES(D,N)
1: $trees \leftarrow \{\}$
2: **for** $i \leftarrow \{1..N\}$ **do**
3: $trees_i \leftarrow$ BUILDKDTREE(D)
4: **end for**
5: **return** *trees*
procedure BUILDKDTREE(D)
1: **if** $|D| == 1$ **then**
2: CREATELEAFNODE(D)
3: **else**
4: $axesMeans[1..d] \leftarrow$ mean of features in D along each dimension
5: $axesVariances[1..d] \leftarrow$ variance of features in D along each dimension
6: $maxVarianceAxes[1..N_D] \leftarrow$ top N_D dimensions with highest variance
7: $splitDim \leftarrow$ random dimension from $maxVarianceAxes$
8: $splitValue \leftarrow axesMeans[splitDim]$
9: $(D_1, D_2) \leftarrow$ split dataset on $splitDim$ using $splitValue$
10: $T_1 \leftarrow$ BUILDKDTREE(D_1)
11: $T_2 \leftarrow$ BUILDKDTREE(D_2)
12: CREATEINNERNODE($splitDim$, $splitValue$, T_1, T_2)
13: **end if**

When searching the randomized k-d forest, a single priority queue is maintained across all the randomized trees (see Algorithm 2). The priority queue is ordered by increasing distance to the decision boundary of each branch in the queue, so search will visit first the closest leaves from all the trees. Once a data point is visited inside a tree, it is marked so that it will not be re-visited from another tree.

The degree of approximation is determined by the maximum number of leaves to be visited (across all trees) and the search is stopped when this number is

reached, returning the best nearest neighbour candidates found up to that point.

Algorithm 2 Searching the randomized k-d forest

Input: randomized k-d trees T_i, query point Q, number of neighbours K, maximum number of points to examine L

Output: approximate K nearest neighbours of query point

procedure SEARCHKDTREES(T_i,Q,K,L)

```
1:  count ← 0
2:  PQ ← empty priority queue
3:  R ← empty priority queue
4:  for each tree T_i do
5:     call TRAVERSEKDTREE(Q,T_i,PQ,R)
6:  end for
7:  while PQ not empty and count < L do
8:     T ← top of PQ
9:     call TRAVERSEKDTREE(Q,T,PQ,R)
10: end while
11: return K top points from R
```

procedure TRAVERSEKDTREE(Q,T,PQ,R)

```
1:  if T is a leaf node then
2:     add point in T to R
3:     count ← count + 1
4:  else
5:     splitDim ← T.splitDim
6:     splitValue ← T.splitValue
7:     if Q(splitDim) < splitValue then
8:        closestNode ← T.left
9:        otherNode ← T.right
10:    else
11:       closestNode ← T.right
12:       otherNode ← T.left
13:    end if
14:    add otherNode to PQ
15:    call TRAVERSEKDTREE(Q,closestNode,PQ,R)
16: end if
```

Figure 3.3 shows the value of searching in many randomized kd-trees at the same time. It can be seen that performance improves with the number of randomized trees up to a certain point (about 20 random trees in this case) and that

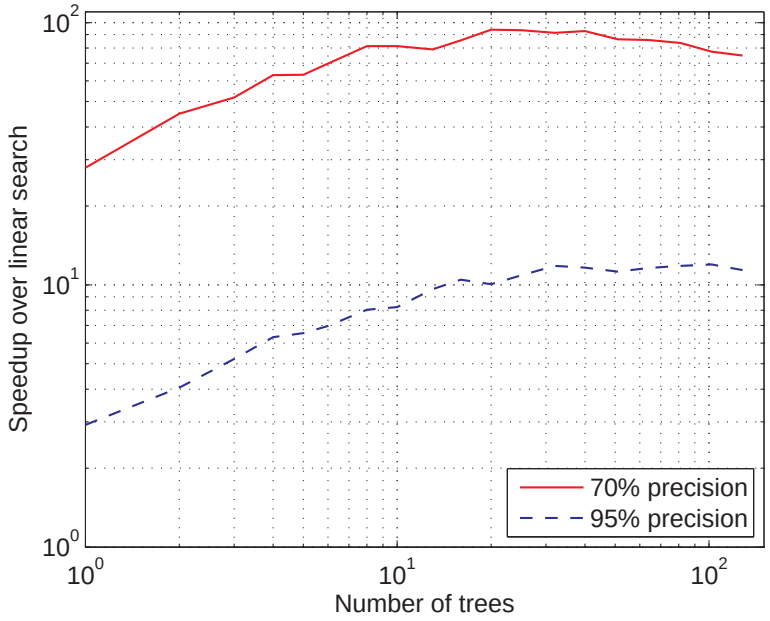

Figure 3.3: Speedup obtained by using multiple randomized kd-trees (100K SIFT features dataset).

increasing the number of random trees further leads to static or decreasing performance. The memory overhead of using multiple random trees increases linearly with the number of trees, so at some point the speedup will not justify the additional memory used.

3.1.2 Analysis

We analyse the complexity of the randomized k-d forest by looking at the index construction time, search time as well as the memory requirements. Depending on the application domain, each of these can be of a high importance. For example, when performing many batch nearest neighbour searches with a fixed dataset that can be built in advance, the target is usually to minimize the search time. For online searches, when the dataset changes often and the trees need to be rebuilt, both the construction time and the search time should be as low as possible. In memory constrained environments, having an index will a low memory overhead is also a key factor.

Construction Time Complexity

To analyse the time complexity for constructing a randomized k-d forest, we first look at the complexity of partitioning the data points corresponding to a single node v with n_v associated data points. The computation of the mean and variance for each dimension costs $O(d\,n_v)$ where d is the number of dimensions. The complexity of partitioning the set of points according to the splitting dimension and splitting value is $O(n_v)$, so the total complexity of partitioning the points is $O(d\,n_v)$. If we assume a balanced tree, we'll have a total of n points on each level, resulting in a complexity of $O(d\,n)$ for partitioning the nodes on a level and $O(\log n)$ levels, for a total tree construction complexity of $O(d\,n\log)$. For m parallel trees this becomes $O(m\,d\,n\log n)$.

Search Time Complexity

To perform an approximate search, a top-down traversal of the trees is performed as many times as the number of leafs L we want to visit. During each descent the algorithm needs to check O($\log n$) internal nodes and one leaf node.

For each internal node the algorithm needs to decide which side of the decision boundary to explore next, which is a O(1) operation, and to insert the unexplored branch into the priority queue. We are using a binary heap, which has a O($\log n_Q$) insertion complexity however the size of the priority queue n_Q is limited by the number of leafs visited. A binomial heap could be used for $O(1)$ amortized insertion cost. For each leaf, the distance between the query and the leaf point needs to be computed, an $O(d)$ operation, where d is the data dimensionality. The overall search time complexity when examining L leaf nodes is $O(L\,d + L\log n)$.

Storage Cost

The storage cost for the randomized k-d forest is low as each inner node only needs to store the split dimension, split value and two pointers to the left and right nodes. Each leaf node only needs to store a pointer to the data point it contains. The total storage cost for storing m trees is $O(mn)$ and if we include the cost of storing the data points: $O(nd + mn)$.

3.2 The Priority Search K-Means Tree Algorithm

We have found the randomized k-d forest to be very effective in many situations, however on other datasets a different algorithm, the *priority search k-means tree*, has been more effective at finding approximate nearest neighbours, especially when a high precision is required.

The priority search k-means tree tries to better exploit the natural structure existing in the data, by clustering the data points using the full distance across all dimensions, in contrast to the (randomized) k-d tree algorithm which only partitions the data based on one dimension at a time. Therefore, it can be expected for the priority search k-means tree algorithm to be able to better discover the natural clustering structure present in the data and to perform better on some datasets.

3.2.1 Algorithm description

The priority search k-means tree is constructed by partitioning the data points at each level into K distinct regions using k-means clustering, and then applying the same method recursively to the points in each region. The recursion is stopped when the number of points in a region is smaller than K (see Algorithm 3).

Nearest-neighbour algorithms that use hierarchical partitioning schemes based on clustering the data points have been previously proposed in the literature [Brin, 1995; Fukunaga and Narendra, 1975; Nister and Stewenius, 2006]. All these algorithm differ in the way they construct the partitioning tree (whether using k-means, agglomerative or some other form of clustering) and especially in the strategies used for exploring the hierarchical tree.

The algorithm we propose in this section explores the k-means tree using a *best-bin-first* strategy, by analogy to what has been found to significantly improve the performance of the approximate kd-tree searches.

The tree is searched by initially traversing the tree from the root to the closest leaf, following at each inner node the branch with the closest cluster centre to the query point, and adding all unexplored branches along the path to a priority queue (see Algorithm 4). The priority queue is sorted in increasing distance from the query point to the boundary of the branch being added to the queue. After the initial tree traversal, the algorithm resumes traversing the tree, but always starting with

Algorithm 3 Building the priority search k-means tree

Input: features dataset D, branching factor K, maximum iterations I_{max}, centre selection algorithm to use C_{alg}
Output: k-means tree

if $|D| < K$ **then**
  create leaf node with the points in D
else
  $P \leftarrow$ *select K points from D using the* C_{alg} *algorithm*
  converged $\leftarrow$ **false**
  $iterations \leftarrow 0$
  while not *converged* **and** $iterations < I_{max}$ **do**
    $C \leftarrow$ *cluster the points in D around nearest centres P*
    $P_{new} \leftarrow$ *means of clusters in C*
    if P=P_{new} **then**
      converged $\leftarrow$ **true**
    end if
    $P \leftarrow P_{new}$
    $iterations \leftarrow iterations + 1$
  end while
  for *each cluster* $C_i \in C$ **do**
    create non-leaf node with center P_i
    recursively apply the algorithm to the points in C_i
  end for
end if

the top branch in the queue, the branch with the closest centre to the query point. In each traversal the algorithm keeps adding to the priority queue the unexplored branches along the path.

The degree of approximation is controlled in the same way as for the randomized kd-trees, by stopping the search early after a predetermined number of dataset points have been examined.

Algorithm parameters

The number of clusters K to use when partitioning the data at each node is a parameter of the algorithm, called the *branching factor* and choosing K is important for obtaining a good search performance. In section 3.4 we propose an algorithm

Algorithm 4 Searching the priority search k-means tree

Input: k-means tree T, query point Q, maximum number of points to examine L
Output: K nearest approximate neighbours of query point

procedure SEARCHKMEANSTREE(T,Q,L)
1: *count* $\leftarrow 0$
2: $PQ \leftarrow$ *empty priority queue*
3: $R \leftarrow$ *empty priority queue*
4: *call* TRAVERSETREE*(T,PQ,R)*
5: **while** PQ not empty **and** *count* $< L$ **do**
6: $N \leftarrow$ *top of* PQ
7: *call* TRAVERSETREE*(N,PQ,R)*
8: **end while**
9: **return** *K top points from R*

procedure TRAVERSEKMEANSTREE(N,PQ,R)
1: **if** *node N is a leaf node* **then**
2: *search all the points in N and add them to R*
3: *count* $\leftarrow$ *count* $+ |N|$
4: **else**
5: $C \leftarrow$ *child nodes of N*
6: $C_q \leftarrow$ *closest node of C to query Q*
7: $C_p \leftarrow C \backslash C_q$
8: *add all nodes in* C_p *to PQ*
9: *call* TRAVERSETREE*(*C_q*,PQ,R)*
10: **end if**

for finding the optimum algorithm parameters, including the optimum branching factor. Figure 3.4 contains a visualisation of several hierarchical k-means decompositions with different branching factors computed for the same dataset.

Another parameter of the priority search k-means tree is I_{max}, the *maximum number of iterations* to perform in the k-means clustering loop. One disadvantage of the priority search k-means tree over the randomized k-d forest is that it has a higher tree-build time. The build time can be significantly reduced by doing a small number of iterations in the k-means clustering stage instead of running it until convergence. This results in a slightly un-optimal clustering (if we consider the sum of squared errors from the points to the cluster centres as the measure of optimality), however we have observed that even after a small number of iterations

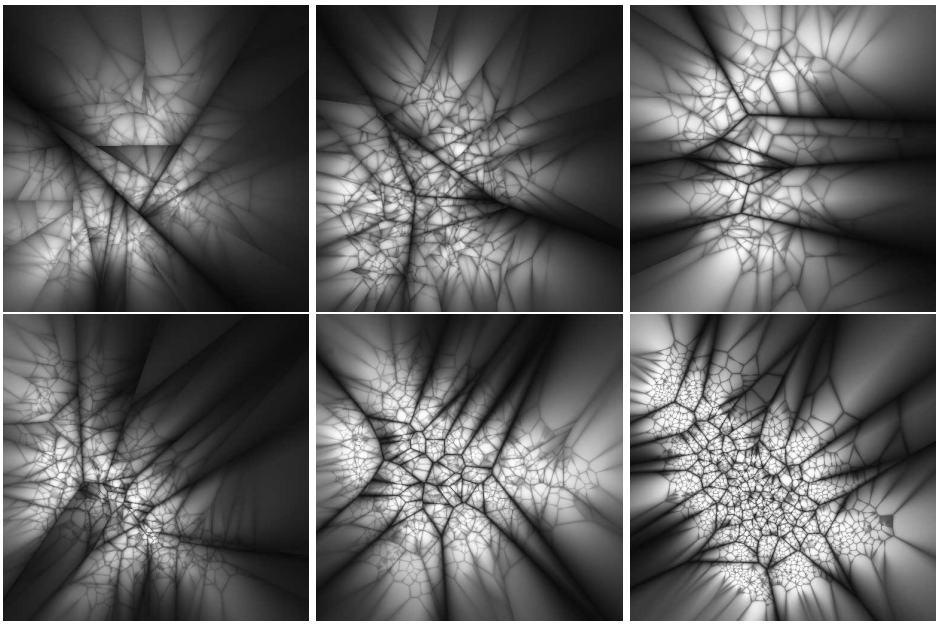

Figure 3.4: Projections of priority search k-means trees constructed using different branching factors: 2, 4, 8, 16, 32, 128. The projections are constructed using the same technique as in [Schindler et al., 2007]. The grey values indicate the ratio between the distances to the nearest and the second-nearest cluster centre at each tree level, so that the darkest values (ratio$\approx$1) fall near the boundaries between k-means regions.

the nearest neighbour search performance is similar to that of the tree constructed by running the clustering until convergence.

The above is illustrated in Figure 3.5 which shows the performance of the tree constructed using a limited number of iterations in the k-means clustering step relative to the performance of the tree when the k-means clustering is run until convergence. It can be seen that using as few as 7 iterations we get more than 90% of the nearest-neighbour performance of the tree constructed using full convergence, but requiring less than 10% of the build time.

When using zero iterations in the k-means clustering we obtain the more general GNAT tree of [Brin, 1995], which assumes that the data lives in a generic metric space, not in a vector space. However Figure 3.5 shows that the search performance of this tree is worse than that of the priority search k-means tree (by

factor of 5).

The algorithm to use when picking the initial centres in the k-means clustering can be controlled by the C_{alg} parameter. In our experiments (and in the FLANN library) we have used the following algorithms: random selection, Gonzales' algorithm (selecting the centres to be spaced apart from each other) and KMeans++ algorithm [Arthur and Vassilvitskii, 2007]. We have discovered that the initial cluster selection made only a small difference in terms of the overall search efficiency in most cases and that the random initial cluster selection is usually a good choice for the priority search k-means tree.

We have also experimented with constructing multiple randomized priority search k-means trees and have found that using multiple trees did not bring significant performance improvements. We suspect the reason for this to be the fact that the k-means clustering process forced the partitioning domains of the multiple randomized trees to overlap, eliminating the benefit of the randomization.

3.2.2 Analysis

We analyse the complexity of the priority search k-means tree algorithm with respect to the data size and dimensionality, build and search parameters. As before we consider the tree construction time, search time and the memory requirements for storing the tree.

Construction Time Complexity

During the construction of the k-means tree, a k-means clustering operation has to be performed for each inner node. Considering a node v with n_v associated data points, and assuming a maximum number of iterations I in the k-means clustering loop, the complexity of the clustering operation is $O(n_v dKI)$. Taking into account all the inner nodes on a level, we have $\sum n_v = n$, so the complexity of constructing a level in the tree is $O(ndKI)$. Assuming a balanced tree, the height if the tree will be $(\log n / \log K)$, resulting in a total tree construction cost of $O(ndKI(\log n / \log K))$.

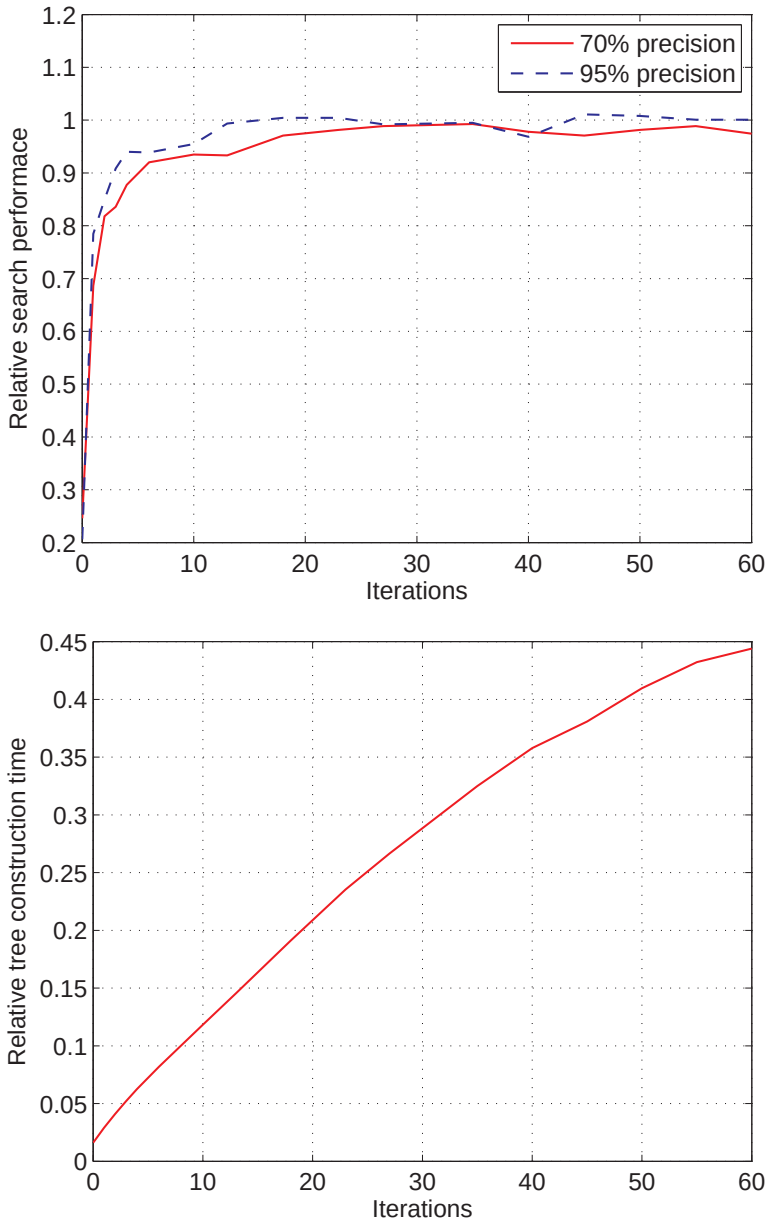

Figure 3.5: The influence that the number of k-means iterations has on the search time efficiency of the k-means tree (top) and on the tree construction time (bottom) (100K SIFT features dataset). Both figures show the relative search/build time compared to the case of using full convergence for the k-means algorithm.

Search Time Complexity

In case of the time constrained approximate nearest neighbour search, the algorithm stops after examining L data points. Considering a complete priority search k-means tree with branching factor K, the number of top down tree traversals required is L/K. During each top-down traversal, the algorithm needs to check $O(\log n/\log K)$ inner nodes and one leaf node.

For each internal node, the algorithm has to find the branch closest to the query point, so it needs to compute the distances to all the cluster centres of the child nodes, an $O(Kd)$ operation. The unexplored branches are added to a priority queue, which can be accomplished in $O(K)$ amortized cost when using binomial heaps. For the leaf node the distance between the query and all the points in the leaf needs to be computed which takes $O(Kd)$ time. In summary the overall search cost is $O(Ld + Ld(\log n/\log K))$.

Storage Complexity

Storing the k-means tree requires memory for $O(n/K^2)$ inner nodes and $O(n/K)$ leaf nodes. Each inner node stores the K centres of the child clusters and K pointers to the associated child nodes, while the leafs store up to K pointers to the data points. In conclusion the total storage requirements for the entire tree is $O(nd/K + n)$.

3.3 Experiments

In this section we present several experiments we have conducted in order to analyse the performance of the two algorithms described in this chapter.

For these experiments we used a selection of datasets with a wide range of sizes and data dimensionality. Among the datasets used are the Winder/Brown patch dataset [Winder and Brown, 2007], datasets of randomly sampled data of different dimensionality, datasets of SIFT features of different sizes obtained by sampling from the CD cover dataset of [Nister and Stewenius, 2006] as well as a dataset of 100K SIFT features extracted from the overlapping images forming a panorama. For each experiment, we randomly sample a query set of points from each of these datasets.

We analyse the performance of the algorithms presented above by looking at the speedup over linear search of each algorithm for different search precisions. Typically, there is a trade-off between the accuracy of the search and the search performance, trade-off illustrated by Figure 3.6.

For each experiment we measure only the processor time used by the process running the experiment, to not be influenced by the other processes executing on the machine at the same time. In order to eliminate noisy variations that can occur during an experiment, we execute each experiment multiple times (between 5 and 10 times) and present the average run times between all executions. We generally omit the error bars on the graphs in the paper because the measurement errors are typically small, as indicated by Figure 3.6, and the conclusions we draw from these experiments are based on much larger differences between the different algorithms we compare.

3.3.1 Data dimensionality.

Data dimensionality is one of the factors that has a great impact on the nearest neighbour matching performance. The top of Figure 3.7 shows how the search performance degrades as the dimensionality increases in the case of random vectors. The datasets in this case each contain 10^5 vectors whose values are randomly sampled from the same uniform distribution. These random datasets are one of the most difficult problems for nearest neighbour search, as no value gives any predictive information about any other value.

As can be seen in the top part of Figure 3.7, the nearest-neighbour searches have a low efficiency for higher dimensional data (for 68% precision the approximate search speed is no better than linear search when the number of dimensions is greater than 800). However real world datasets are normally much easier due to correlations between dimensions.

The performance is markedly different for many real-world datasets. The bottom part of Figure 3.7 shows the speedup as a function of dimensionality for the Winder/Brown image patches[1] resampled to achieve varying dimensionality. In this case however, the speedup does not decrease with dimensionality, it actually

[1] http://phototour.cs.washington.edu/patches/default.htm

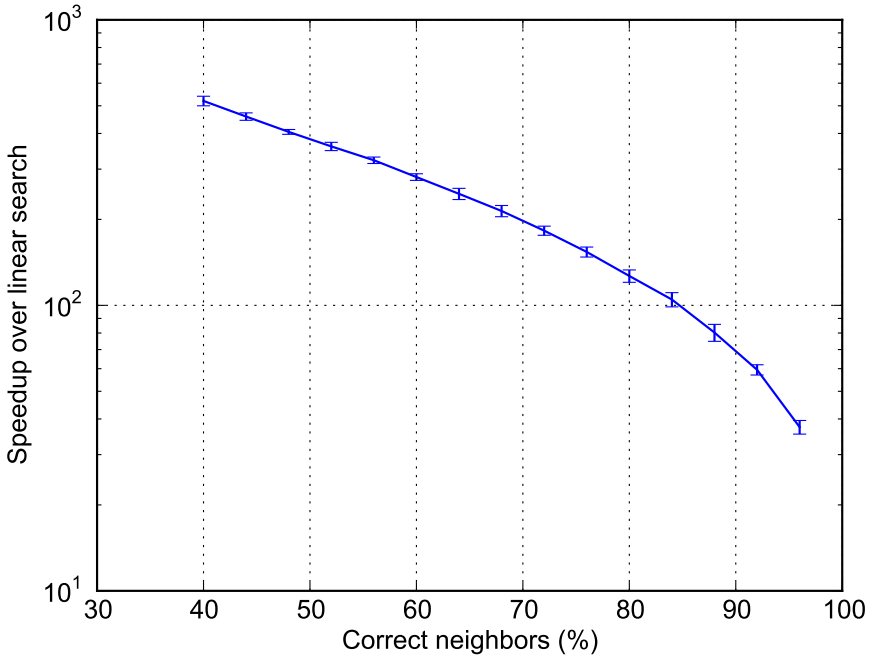

Figure 3.6: Search speedup for different precisions (the priority search k-means tree algorithm on a dataset of 100K SIFT features extracted from the Winder/Brown patches). The error bars indicate the standard deviation of the run time measurements.

increases for some precision levels. This can be explained by the fact that there exists a strong correlation between the dimensions, so that even for 64x64 patches (4096 dimensions), the similarity between only a few dimensions provides strong evidence for overall patch similarity.

In the experiments of Figure 3.7, we used the randomized kd-trees and the priority k-means tree algorithms and each data point was obtained using the algorithm performing best for that particular combination of data dimensionality and search precision.

In Figure 3.8 we present, for illustrative purposes, four examples of nearest neighbour search on the Trevi dataset of patches, using patches with varying dimensionality. For each query, we present the first five approximate nearest neighbour returned by the search. The patches marked with an "X" are patches that represent objects different than the one in the query patch.

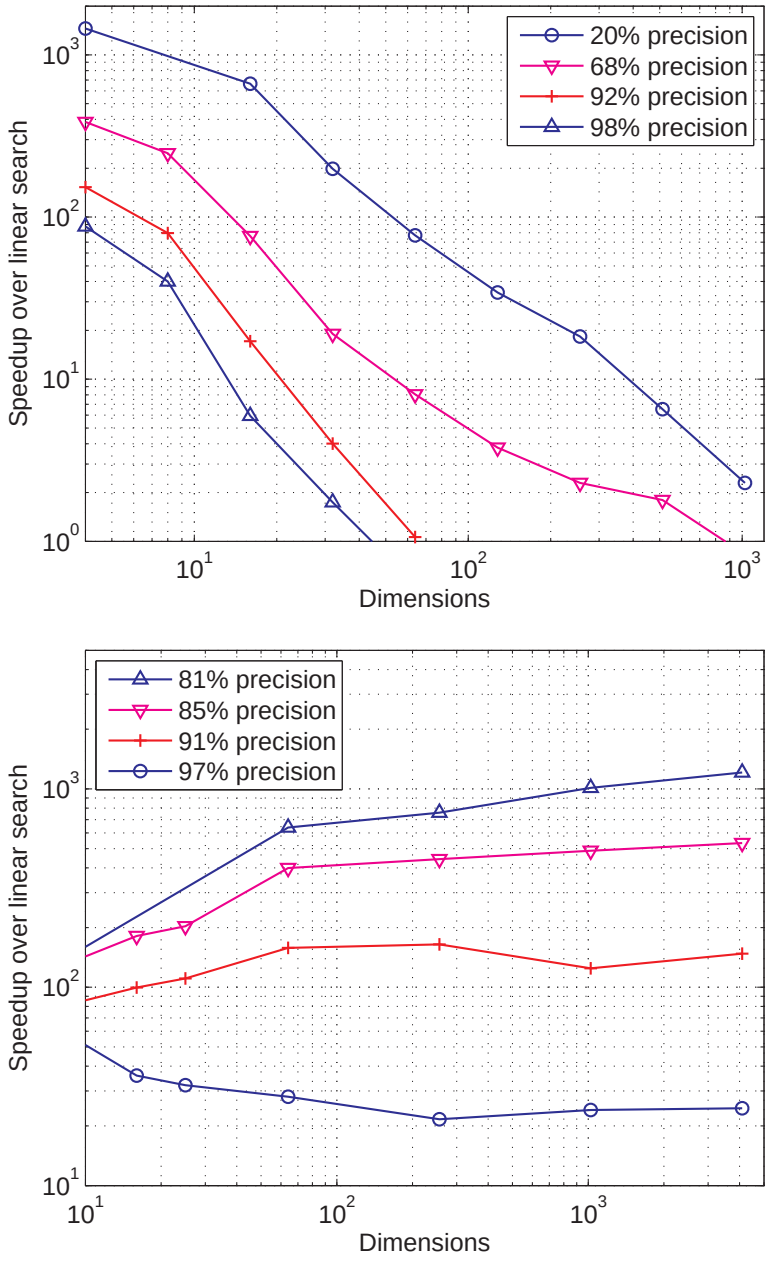

Figure 3.7: Search efficiency for data of varying dimensionality. We experimented on both random vector and image patches. The random vectors (top figure) represent the hardest case in which dimensions have no correlations, while most real-world problems behave more like the image patches (bottom figure).

Figure 3.8: Example of nearest neighbour queries with different patch sizes. The Trevi Fountain patch dataset was queried using different patch sizes. The query patch is on the left of each panel, while the following 5 patches are the nearest neighbours from a set of 100,000 patches. Incorrect matches are shown with an X.

3.3.2 Search precision.

The desired search precision determines the degree of speedup that can be obtained with any approximate algorithm. Looking at Figure 3.10 (the SIFT 1M dataset) we see that if we are willing to accept a precision as low as 60%, meaning that 40% of the neighbours returned are not the exact nearest neighbours, but just approximations, we can achieve a speedup of three orders of magnitude over linear search (using the multiple randomized kd-trees). However, if we require a precision greater then 90% the speedup is smaller, less than 2 orders of magnitude (using the priority search k-means tree).

We use several datasets of different dimensions for the experiments in Fig-

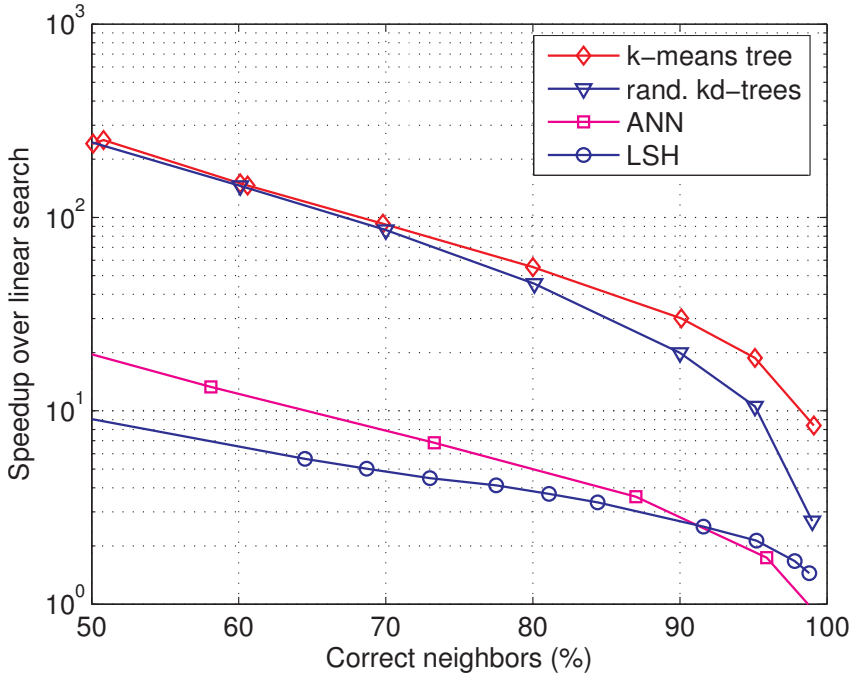

Figure 3.9: Comparison of the search efficiency for several nearest neighbour algorithms (100K SIFT features dataset).

ure 3.10. We construct a 100K and 1 million SIFT features dataset by randomly sampling a dataset of over 5 million SIFT features extracted from a collection of CD cover images [Nister and Stewenius, 2006][2]. These two datasets obtained by random sampling have a relatively high degree of difficulty in terms of nearest neighbour matching, because the features they contain usually do not have "true" matches between them. We also use the entire 31 million feature dataset from the same source for the experiment in Figure 3.10. Additionally we use the patches datasets described in subsection 3.3.1 and another 100K SIFT features dataset obtained from a set of images forming a panorama.

We compare the two algorithms we found to be the best at finding fast approximate nearest neighbours (the multiple randomized kd-trees and the priority search k-means tree) with existing approaches, the ANN [Arya et al., 1998] and

[2] http://www.vis.uky.edu/~stewe/ukbench/data/

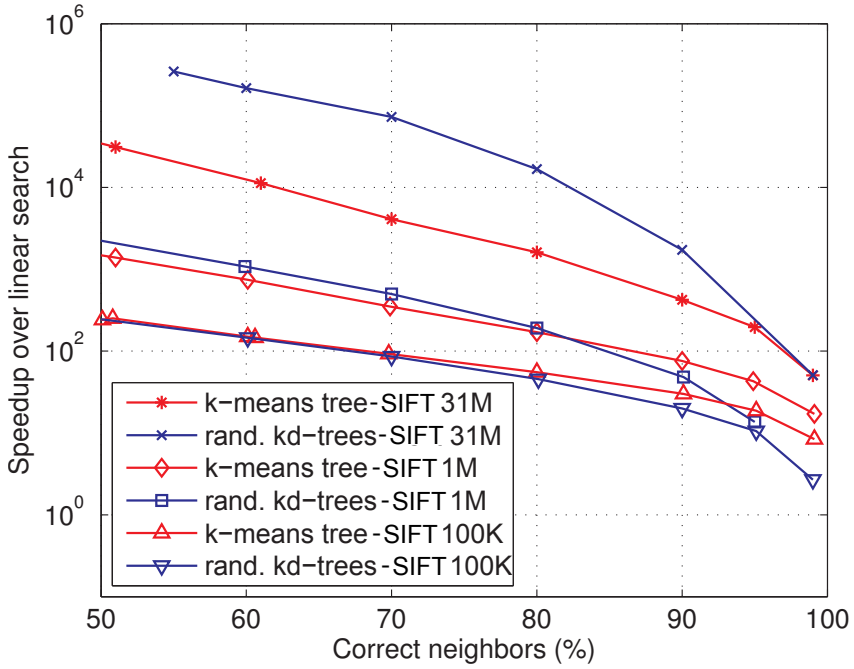

Figure 3.10: Search speedup for different dataset sizes.

LSH algorithms [Andoni and Indyk, 2008][3] on the first dataset of 100,000 SIFT features. Since the LSH implementation (the E^{2}LSH package) solves the R-near neighbour problem (finds the neighbours within a radius R of the query point, not the nearest neighbours), to find the nearest neighbours we have used the approach suggested in the E^{2}LSH's user manual: we compute the R-near neighbours for increasing values of R. The parameters for the LSH algorithm were chosen using the parameter estimation tool included in the E^{2}LSH package. For each case we have computed the precision achieved as the percentage of the query points for which the nearest neighbours were correctly found. Figure 3.9 shows that the priority search k-means algorithm outperforms both the ANN and LSH algorithms by about an order of magnitude. The results for ANN are consistent with the experiment in Figure 3.3, as ANN uses only a single kd-tree and does not benefit from the speedup due to using multiple randomized trees.

[3]We have used the publicly available implementations of ANN (http://www.cs.umd.edu/~mount/ANN/) and LSH (http://www.mit.edu/~andoni/LSH/)

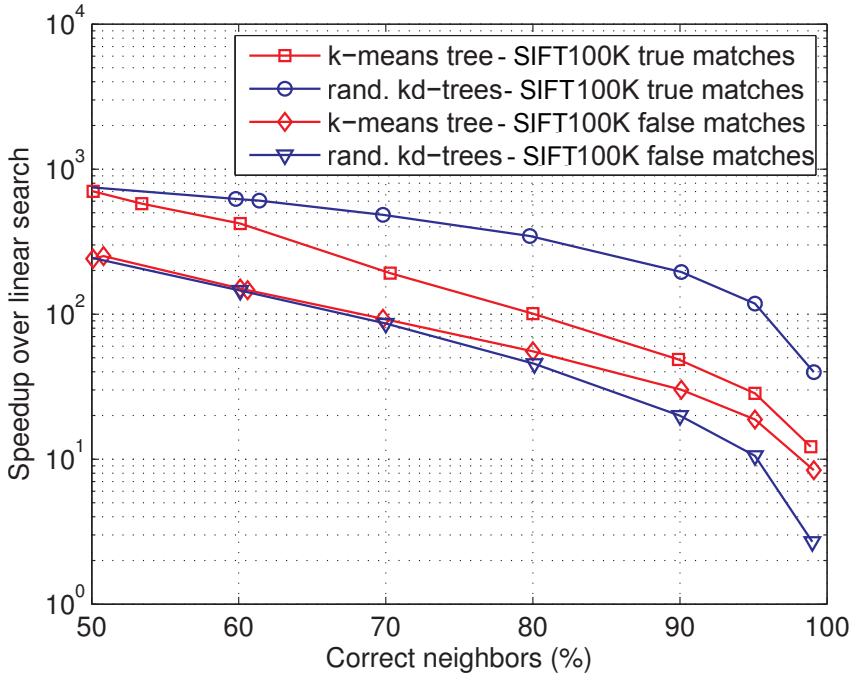

Figure 3.11: Search speedup when the query points don't have "true" matches in the dataset vs the case when they have.

Figure 3.10 shows the performance of the randomized kd-trees and the priority search k-means on datasets of different sizes. The figure shows that the two algorithms scale well with the increase in the dataset size, having the speedup over linear search increase with the dataset size.

Figure 3.11 compares the performance of nearest neighbour matching when the dataset contains true matches for each feature in the test set to the case when it contains false matches. A true match is a match in which the query and the nearest neighbour point represent the same entity, for example in case of SIFT feature, they represent image patches of the same object. In this experiment we used the two 100K SIFT features datasets described above. The first is randomly sampled from a 5 million SIFT features dataset and it contains false matches for each feature in the test set. The second contains SIFT features extracted from a set of images forming a panorama. These features were extracted from the overlapping regions of the images, and we use only those that have a true match in the dataset.

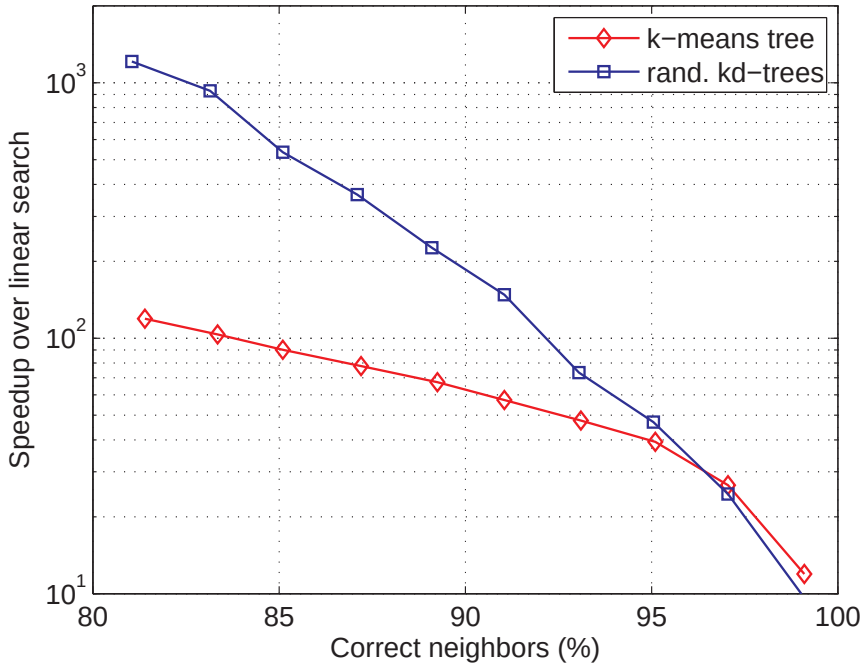

Figure 3.12: Search speedup for the Trevi Fountain patches dataset.

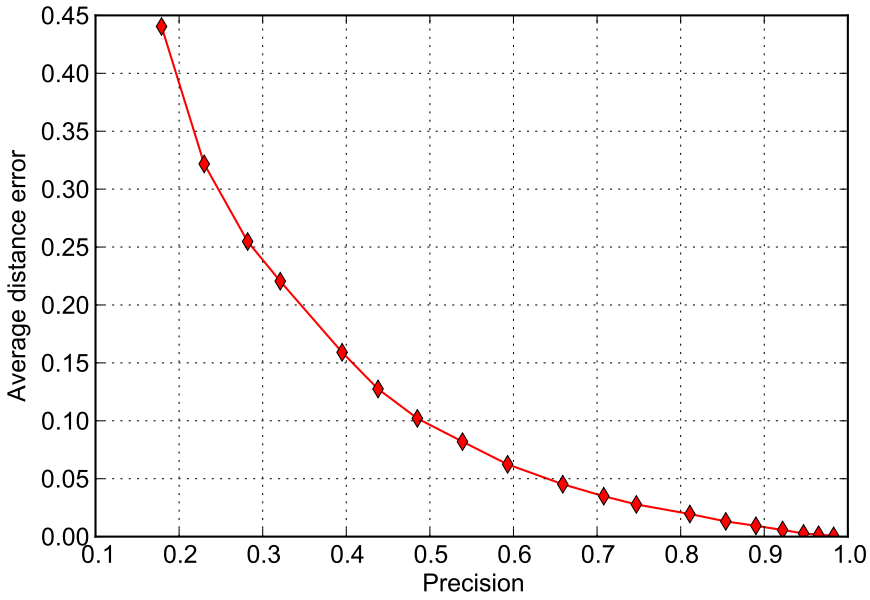

Figure 3.13: Average distance error for different precisions (SIFT features extracted from the Trevi fountain patches dataset).

Our experiments showed that the randomized kd-trees have a significantly better performance for true matches, when the query features are likely to be significantly closer than other neighbours. Similar results were reported in [Mikolajczyk and Matas, 2007].

Figure 3.12 illustrates the difference in performance between the randomized kd-trees and the priority search k-means tree for one of the Winder/Brown patches datasets. In this case, the randomized kd-trees algorithm clearly outperforms the priority search k-means algorithm everywhere except for cases where precision is very close to 100%. It appears that the kd-tree works much better in cases when the intrinsic dimensionality of the data is much lower than the actual dimensionality, presumably because it can better exploit the correlations among dimensions. However, Figure 3.10 shows that the k-means tree can perform better for other datasets (especially for high search precisions). This result demonstrates the importance of performing algorithm selection on each dataset.

Figure 3.13 shows the average relative distance error between the neighbours returned by an approximate search and the true neighbours for different search precisions, using a dataset of SIFT features computed on the patches of one of the Winder/Brown patches datasets. The relative distance error is computed as $(d - d^*)/d^*$, where d is the distance between a query point and its nearest neighbour returned by the approximate search and d^* is the distance between the same query point and its true nearest neighbour. It can be seen that the distance error quickly decreases as the precision increases, such that for 50% search precision (only half the neighbours returned are the exact neighbours) the distance error is less than 10% and for 70% search precision the distance error is less than 5%. This shows that even when the points returned by the approximate search are not the exact neighbours, they are points close to the exact neighbours.

3.4 Automatic Selection of the Optimal Algorithm

Our experiments have revealed that the optimal algorithm for approximate nearest neighbour search is highly dependent on several factors such as the data dimensionality, size and structure of the dataset (whether there is any correlation between the features in the dataset) and the desired search precision. Additionally,

each algorithm has a set of parameters that have significant influence on the search performance. These parameters include the number of randomized trees to use in the case of kd-trees or the branching factor and the number of iterations in the case of the priority search k-means tree.

Although there are hundreds of papers published on the topic of nearest neighbour search, surprisingly little has been done in the area of systematically choosing the optimum nearest neighbour algorithm and tuning its parameters. In practice and in most of the nearest neighbour literature, setting the algorithm parameters is usually a manual process carried out by using various heuristics and rarely more systematic approaches.

The manual tuning of a nearest neighbour algorithm is a process typically started with some parameter configuration that is believed to give good results in general, and which is iteratively improved by changing one parameter at a time and accepting a new parameter configuration when it results in improved performance. The process is typically stopped when the performance obtained cannot be improved through successive parameter changes or is adequate for the task at hand.

Having the possibility to automatically select the optimum nearest neighbour algorithm and tune its parameters it important for several reasons:

- The variability in performance of different nearest neighbour algorithms relative to their parameters prevents them for easily being compared on an equal footing as it's not obvious if one algorithm has better performance or just that its authors managed to find better parameters for it. Having an automatic and objective way of selecting optimum parameters would allow comparisons of different nearest neighbour algorithms to be more meaningful.

- When used in practice the performance of most nearest neighbour algorithms depends on having the optimum parameters for the situation. Using the manual parameter tuning process described above is tedious and in most cases the users of different nearest neighbour algorithms don't try to optimize the parameters, using the default ones instead. As we have showed in this chapter, any default parameters, even if they work well for a large range of problems, are bound to be sub-optimal or even perform badly for other problems.

- Users can be intimidated by the many parameters an algorithm might have

and choose to use other algorithms not well suited for a particular problem just because they have fewer or no parameters to adjust. An automated way of choosing the optimum parameters would allow users to use the appropriate algorithm for each situation.

By considering the algorithm itself as a parameter of a generic nearest neighbour search routine A, the problem is reduced to determining the parameters $\theta \in \Theta$ that give the best solution, where Θ is also known as the *parameter configuration space*. This can be formulated as an optimization problem in the parameter configuration space:

$$\min_{\theta \in \Theta} c(\theta)$$

with $c : \Theta \to \mathbb{R}$ being a cost function indicating how well the search algorithm A, configured with the parameters θ, performs on the given input data.

We define the cost as a combination of the search time, tree build time, and tree memory overhead. Depending on the application, each of these three factors can have a different importance: in some cases we don't care much about the tree build time (if we will build the tree only once and use it for a large number of queries), while in other cases both the tree build time and search time must be small (if the tree is built on-line and searched a small number of times). There are also situations when we wish to limit the memory overhead if we work in memory constrained environments. We define the cost function as follows:

$$c(\theta) = \frac{s(\theta) + w_b b(\theta)}{\min_{\theta \in \Theta}(s(\theta) + w_b b(\theta))} + w_m m(\theta) \tag{3.1}$$

where $s(\theta)$, $b(\theta)$ and $m(\theta)$ represent the search time, tree build time and memory overhead for the tree(s) constructed and queried with parameters θ. The memory overhead represents the ratio of the memory used by the tree(s) and the memory used by the data: $m(\theta) = m_t(\theta)/m_d$.

The weights w_b and w_m are used to control the relative importance of the build time and memory overhead in the overall cost. The build-time weight (w_b) controls the importance of the tree build time relative to the search time. Setting $w_b = 0$ means that we want the fastest search time and don't care about the tree build time, while setting larger values for w_b gives the tree build time increased impor-

tance. Similarly, the memory weight (w_m) controls the importance of the memory overhead compared to the time overhead. The time overhead is computed relative to the optimum time cost $\min_{\theta\in\Theta}(s(\theta)+w_b b(\theta))$, which is defined as the optimal search and build time if memory usage were not a factor. Therefore, setting $w_m = 0$ will choose the algorithm and parameters that result in the fastest search and build time without any regard to memory overhead, while setting $w_m = 1$ will give equal weight to a given percentage increase in memory use as to the same percentage increase in search and build time.

We perform the above optimization in two steps: a global exploration of the parameter space using grid search, followed by a local optimization starting with the best solution found in the first step.

Because the number of parameters of various nearest neighbour search algorithms is relatively low, a grid search approach is a feasible approach. In the case of the randomized k-d trees and the priority search k-means tree we consider the following parameter values for the grid search: $\{1,4,8,16,32\}$ as the number of random kd-trees, $\{16,32,64,128,256\}$ as the branching factor for the k-means tree and $\{1,5,10,15\}$ as the number of k-means iterations.

In the second step we use the Nelder-Mead downhill simplex method [Nelder and Mead, 1965] to further locally explore the parameter space and fine-tune the best solution obtained in the first step. Although this does not guarantee a global minimum, our experiments have shown that the parameter values obtained are close to optimum.

We use random sub-sampling cross-validation to generate the data and the query points when we run the optimization. In FLANN the optimization can be run on the full dataset for the most accurate results or using just on a fraction of the dataset to have a faster auto-tuning process. The parameter selection needs only be performed once for each type of dataset, and the optimum parameter values can be saved and applied to all future datasets of the same type.

In Table 3.1, we show the results from running the parameter selection procedure described above on a dataset containing 100K randomly sampled SIFT features. We used two different search precisions (60% and 90%) and several combinations of the trade-off factors w_b and w_m. For the build time weight, w_b, we used three different possible values: 0 representing the case where we don't care about

Table 3.1: The algorithms chosen by our automatic algorithm and parameter selection procedure (SIFT 100K dataset). The "Algorithm" column shows the algorithm chosen and its optimum parameters (number of random trees in case of the kd-tree; branching factor and number of iterations for the k-means tree), the "Dist. Error" column shows the mean distance error compared to the exact nearest neighbours, the "Search Speedup" shows the search speedup compared to linear search, the "Memory Used" shows the memory used by the tree(s) as a fraction of the memory used by the dataset and the "Build Time" column shows the tree build time as a fraction of the linear search time for the test set.

$Pr.(\%)$	w_b	w_m	Algorithm Configuration	Dist. Error	Search Speedup	Memory Used	Build Time
60%	0	0	k-means, 16, 15	0.096	181.10	0.51	0.58
	0	1	k-means, 32, 10	0.058	180.9	0.37	0.56
	0.01	0	k-means, 16, 5	0.077	163.25	0.50	0.26
	0.01	1	kd-tree, 4	0.041	109.50	0.26	0.12
	1	0	kd-tree,1	0.044	56.87	0.07	0.03
	*	∞	kd-tree,1	0.044	56.87	0.07	0.03
90%	0	0	k-means, 128, 10	0.008	31.67	0.18	1.82
	0	1	k-means, 128, 15	0.007	30.53	0.18	2.32
	0.01	0	k-means, 32, 5	0.011	29.47	0.36	0.35
	1	0	k-means, 16, 1	0.016	21.59	0.48	0.10
	1	1	kd-tree,1	0.005	5.05	0.07	0.03
	*	∞	kd-tree,1	0.005	5.05	0.07	0.03

the tree build time, 1 for the case where the tree build time and search time have the same importance and 0.01 representing the case where we care mainly about the search time but we also want to avoid a large build time. Similarly, the memory weight was chosen to be 0 for the case where the memory usage is not a concern, ∞ representing the case where the memory use is the dominant concern and 1 as a middle ground between the two cases.

Examining Table 3.1 we can see that for the cases when the build time or the memory overhead had the highest weight, the algorithm chosen was the kd-tree with a single tree because it is both the most memory efficient and the fastest to build. When no importance was given to the tree build time and the memory

overhead the algorithm chosen was k-means, as confirmed by the plots in Figure 3.10. The branching factors and the number of iterations chosen for the k-means algorithm depend on the search precision and the tree build time weight: higher branching factors proved to have better performance for higher precisions and the tree build time increases when the branching factor or the number of iterations increase.

3.5 Chapter Summary

In this chapter we discuss the problem of fast approximate nearest neighbour search in high dimensional spaces.

We present the two algorithms we have found to work best in this domain: the randomized k-d forest and the priority search k-means tree. We describe the two algorithms, analyse their behaviour with respect to construction and search time as well as memory requirements and discuss the effect that the algorithm parameters have on their performance.

We also present a series of experiments comparing the two algorithms on several criteria, such as data dimensionality, dataset size as well as with other approximate nearest neighbour algorithms. Our experiments show that the optimum algorithm and algorithm parameters are dependent on the dataset used and the desired accuracy which emphasize the importance of automatic algorithm selection for nearest neighbour search applications.

Finally, we propose a method for automatic selection of the best algorithm and its optimal parameters for a specific dataset. This has been implemented in FLANN and provides a simple interface for selecting the optimum algorithm and its parameters.

Chapter 4

Binary Feature Matching

A number of binary visual descriptors have been recently proposed in the literature, including BRIEF [Calonder et al., 2010], ORB [Rublee et al., 2011], and BRISK [Leutenegger et al., 2011]. These have several advantages over the more established vector-based descriptors such as SIFT [Lowe, 2004] and SURF [Bay et al., 2006], as they are cheaper to compute, more compact to store, and faster to compare with each other.

Binary features are typically compared using the Hamming distance, which for binary data can be computed by performing a bitwise XOR operation followed by a bit count on the result. This involves only bit manipulation operations which can be performed quickly on many bits at a time, especially on modern computers where there is hardware support for counting the number of bits that are set in a word (the POPCNT instruction[1]).

Even though computing the distance between pairs of binary features can be done efficiently, using linear search for matching can be practical only for smaller datasets. For large datasets, linear matching becomes a bottleneck in most applications. The typical solution in such situations is to replace the linear search with an approximate matching algorithm that can offer speedups of several orders of magnitude over linear search, at the cost that a fraction of the nearest neighbours returned are approximate neighbours (but usually close in distance to the exact

[1] We are referring to the x86_64 architecture, other architectures have similar instructions for counting the number of bits set.

neighbours).

Many algorithms typically used for approximate matching in the case of vector features, which perform a hierarchical decomposition of the search space, are not readily suitable for matching binary features because they assume the features exist in a vector space where each dimension of the features can be continuously averaged. Examples of such algorithms are the kd-tree algorithm, the vocabulary tree and the priority search k-means tree.

For matching binary features, the approximate nearest neighbour search algorithms used in the literature are mostly based on various hashing techniques such as locality sensitive hashing [Rublee et al., 2011], semantic hashing [Salakhutdinov and Hinton, 2009] or min-hash [Zitnick, 2010].

In this chapter we introduce a new algorithm for matching binary features, based on hierarchical decomposition of the search space. We compare the performance of this algorithm to the approximate nearest neighbour algorithms implemented by the FLANN library for both binary and vector-based features. We show that when compared with an LSH implementation, which is what is typically used in the literature for matching binary features, our algorithm performs comparatively or better and scales favourably for larger datasets.

4.1 Matching Binary Features

We introduce a new data structure and algorithm which we have found to be very effective at matching binary features. The algorithm performs a hierarchical decomposition of the search space by successively clustering the input dataset and constructing a tree in which every non-leaf node contains a cluster centre and the leaf nodes contain the input points that are to be matched.

4.1.1 Building the tree

The tree building process (presented in Algorithm 5) starts with all the points in the dataset and divides them into K clusters, where K is a parameter of the algorithm, called the branching factor. The clusters are formed by first selecting K points at random as the cluster centres followed by assigning to each centre the points closer to that centre than to any of the other centres. The algorithm is repeated recursively

for each of the resulting clusters until the number of points in each cluster is below a certain threshold (the maximum leaf size), in which case that node becomes a leaf node.

The decomposition described above is similar to that of k-medoids clustering (k-medoids is an adaptation of the k-means algorithm in which each cluster centre is chosen as one of the input data points instead of being the mean of the cluster elements). However, we are not trying to minimize the squared error (the distance between the cluster centres and their elements) as in the case of k-medoids. Instead we are simply selecting the cluster centres randomly from the input points, which results in a much simpler and more efficient algorithm for building the tree and gives improved independence when using multiple trees as described below. We have experimented with minimizing the squared error and with choosing the cluster centres using the greedy approach used in the GNAT tree [Brin, 1995] and neither brought improvements for the nearest neighbour search performance.

Algorithm 5 Building one hierarchical clustering tree

Input: features dataset D
Output: hierarchical clustering tree
Parameters: branching factor K, maximum leaf size S_L

if *size of* $D < S_L$ **then**
  create leaf node with the points in D
else
  $P \leftarrow$ *select K points at random from D*
  $C \leftarrow$ *cluster the points in D around nearest centers P*
  for *each cluster* $C_i \in C$ **do**
    create non-leaf node with center P_i
    recursively apply the algorithm to the points in C_i
  end for
end if

We found that building multiple trees and using them in parallel during the search results in significant improvements for the search performance. The approach of using multiple randomized trees is known to work well for randomized kd-trees [Silpa-Anan and Hartley, 2008], however in chapter 3 we mentioned that it is not effective for multiple priority search k-means trees. The fact that using multiple randomized trees is effective for the algorithm we propose is likely due to

the fact that we are choosing the cluster centres randomly and perform no further iterations to obtain a better clustering (as in the case of the priority search k-means tree). When using hierarchical space decompositions for nearest neighbour search, the hard cases occur when the closest neighbour to the query point lies just across a boundary from the domain explored, and the search needs to backtrack to reach the correct domain. The benefit of using multiple randomized trees comes from the fact that they are different enough such that, when exploring them in parallel, the closest neighbour probably lies in different domains in different trees, increasing the likelihood of the search reaching a correct domain quickly.

4.1.2 Searching for nearest neighbours using parallel hierarchical clustering trees

The process of searching multiple hierarchical clustering trees in parallel is presented in Algorithm 6. The search starts with a single traverse of each of the trees, during which the algorithm always picks the node closest to the query point and recursively explores it, while adding the unexplored nodes to a priority queue. When reaching the leaf node all the points contained within are linearly searched. After each of the trees has been explored once, the search is continued by extracting from the priority queue the closest node to the query point and resuming the tree traversal from there.

The search ends when the number of points examined exceeds a maximum limit given as a parameter to the search algorithm. This limit specifies the degree of approximation desired from the algorithm. The higher the limit the more exact neighbours are found, but the search is more expensive. In practice it is often useful to express the degree of approximation as search precision, the percentage of exact neighbours found in the total neighbours returned. The relation between the search precision and the maximum point limit parameter can be experimentally determined for each dataset using a cross-validation approach.

4.2 Evaluation

We use the Winder/Brown patch dataset [Winder and Brown, 2007] to evaluate the performance of the algorithm described in section 4.1. These patches were ob-

Algorithm 6 Searching parallel hierarchical clustering trees

Input: hierarchical clustering trees T_i, query point Q
Output: K nearest approximate neighbours of query point
Parameters: max number of points to examine L_{max}

```
1: L ← 0 {L = number of points searched}
2: PQ ← empty priority queue
3: R ← empty priority queue
4: for each tree T_i do
5:    call TRAVERSETREE(T_i,PQ,R)
6: end for
7: while PQ not empty and L < L_max do
8:    N ← top of PQ
9:    call TRAVERSETREE(N,PQ,R)
10: end while
11: return K top points from R
```

procedure TRAVERSETREE(N,PQ,R)

```
1: if node N is a leaf node then
2:    search all the points in N and add them to R
3:    L ← L + |N|
4: else
5:    C ← child nodes of N
6:    C_q ← closest node of C to query Q
7:    C_p ← C\C_q
8:    add all nodes in C_p to PQ
9:    call TRAVERSETREE(C_q,PQ,R)
10: end if
```

tained from consumer images of known landmarks. We first investigate how the different parameters influence the performance of the algorithm and then compare it to other approximate nearest neighbour algorithms described in [Muja and Lowe, 2009] and implemented by the FLANN library [Muja and Lowe]. We also implemented the hierarchical clustering algorithm on top of the open source FLANN library in order to take advantage of the parameter auto-tuning framework provided by the library. To show the scalability of the algorithm we use the 80 million tiny images dataset of [Torralba et al., 2008a].

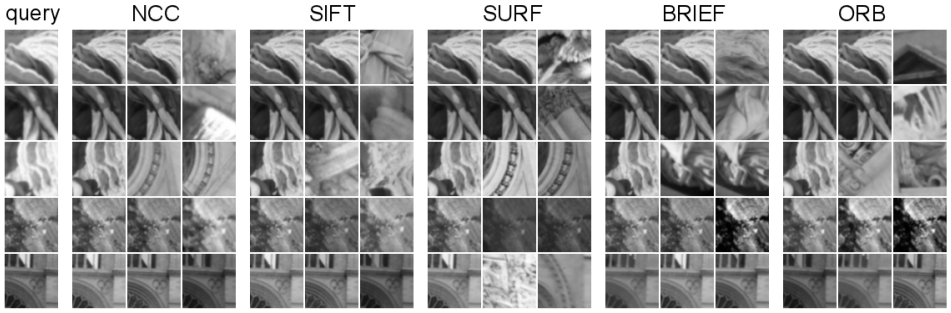

Figure 4.1: Random sample of query patches and the first three nearest neighbours returned when using different feature types.

4.2.1 Algorithm parameters

The behaviour of the algorithm presented in section 4.1 is influenced by three parameters: the number of parallel trees built, the branching factor and the maximum size of the leaf nodes for each tree. In this section we evaluate the impact of each of these parameters on the performance of the algorithm by analysing the speedup over linear search with respect to the search precision for different values of the parameters. Speedup over linear search is used both because it is an intuitive measure of the algorithm's performance and because it's relatively independent of the hardware on which the algorithm is run.

For the experiments in this section we constructed a dataset of approximately 310,000 BRIEF features extracted from all the patch datasets of [Winder and Brown, 2007] combined (the Trevi, Halfdome and Notredame datasets).

Figure 4.2 shows how the algorithm's performance depends on the number of parallel trees used. It can be seen that there's a significant benefit in using more than a single tree and that the performance generally increases as more trees are used. The optimum number of trees depends on the desired precision, for example, for search precisions below 70% there is no benefit in using more than two parallel trees, however for precisions above 85% a higher number (between 4 and 8 trees) gives better results. Obviously more trees implies more memory and a longer tree build time, so in practice the optimum number of trees depends on multiple factors such as the desired search precision, the available memory and constraints on the

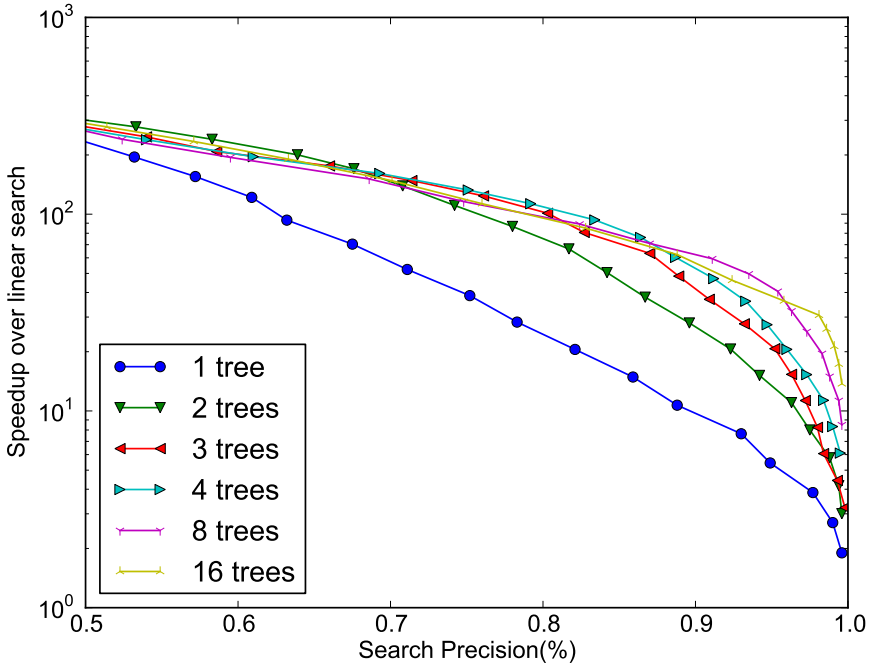

Figure 4.2: Speedup over linear search for different number of parallel randomized trees used by the index (branching factor = 16, max leaf size = 150).

tree build time.

The branching factor also has an impact on the search performance, as figure 4.3 shows. Higher branching factors perform better for high precisions (above 80%), but there is little gain for branching factors above 16 or 32. Also very large branching factors perform worse for lower precisions and have a higher tree build time.

The last parameter we examine is the maximum leaf size. From figure 4.4 we can see that a maximum leaf size of 150 performs better than a small leaf size (16, which is equal to the branching factor) or a large leaf size (500). This can be explained by the fact that computing the distance between binary features is an efficient operation, and for small leaf sizes the overhead of traversing the tree to examine more leaves is greater than the cost of comparing the query feature to all the features in a larger leaf. If the leaves are very large however, the cost of

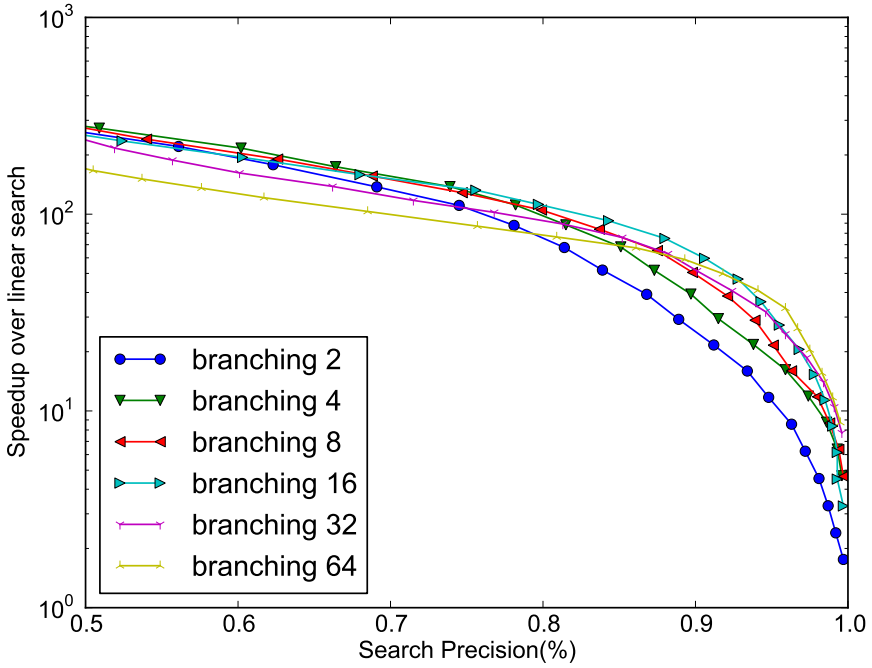

Figure 4.3: Speedup over linear search for different branching factors (trees = 8, max leaf size = 150).

linearly examining all the features in the leaves ends up being greater than the cost of traversing the tree.

The above experiments show that the best parameters to choose depend on several factors and are often a trade-off between memory usage/tree build time and search performance. A similar cross-validation approach to the one proposed in [Muja and Lowe, 2009] can be used for choosing the optimum parameters for a particular application.

4.2.2 Comparison to other approximate nearest neighbour algorithms

We compare the nearest neighbour search performance of the hierarchical clustering algorithm introduced in section 4.1 to that of the nearest neighbour search algorithms implemented by the publicly available FLANN library. For comparison we

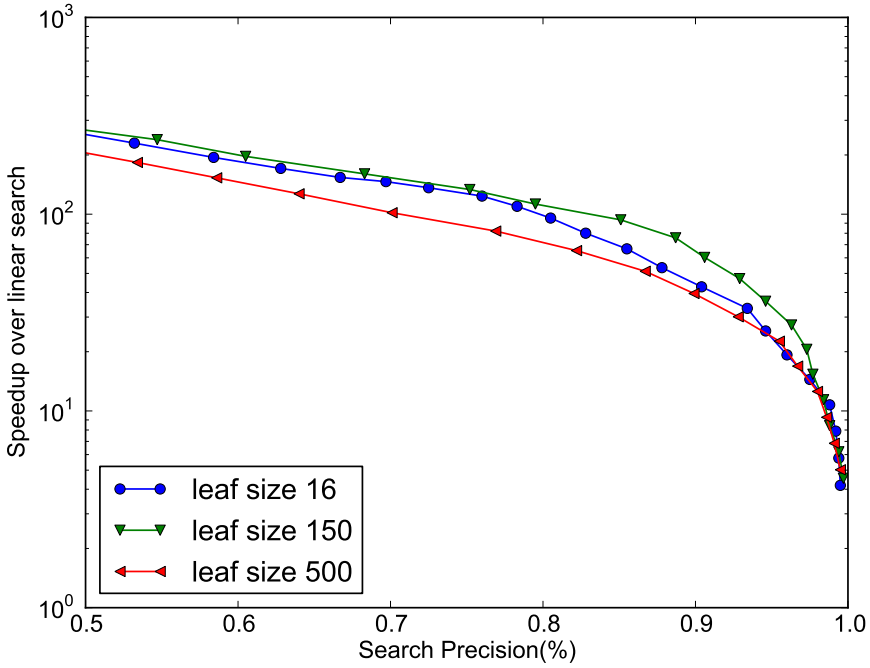

Figure 4.4: Speedup over linear search for different number leaf node sizes (branching factor = 16, trees = 8).

use a combination of different features types, both vector features such as SIFT, SURF, image patches and binary features such as BRIEF and ORB. The image patches have been downscaled to 16x16 pixels and are matched using normalized cross correlation (NCC). Figure 4.1 shows a random sample of 5 query patches and the first three neighbour patches returned when using each of the descriptors mentioned above.

Figure 4.5 shows the nearest neighbour search speedup for different feature types. Each point on the graph is taken as the best performing algorithm for that particular feature type (randomized kd-trees or priority search k-means tree for SIFT, SURF, image patches and the hierarchical clustering algorithm introduced in section 4.1 for BRIEF and ORB). In each case the optimum choice of parameters that maximizes the speedup for a given precision is used.

Examining figure 4.5 it can be seen that the hierarchical clustering tree gives

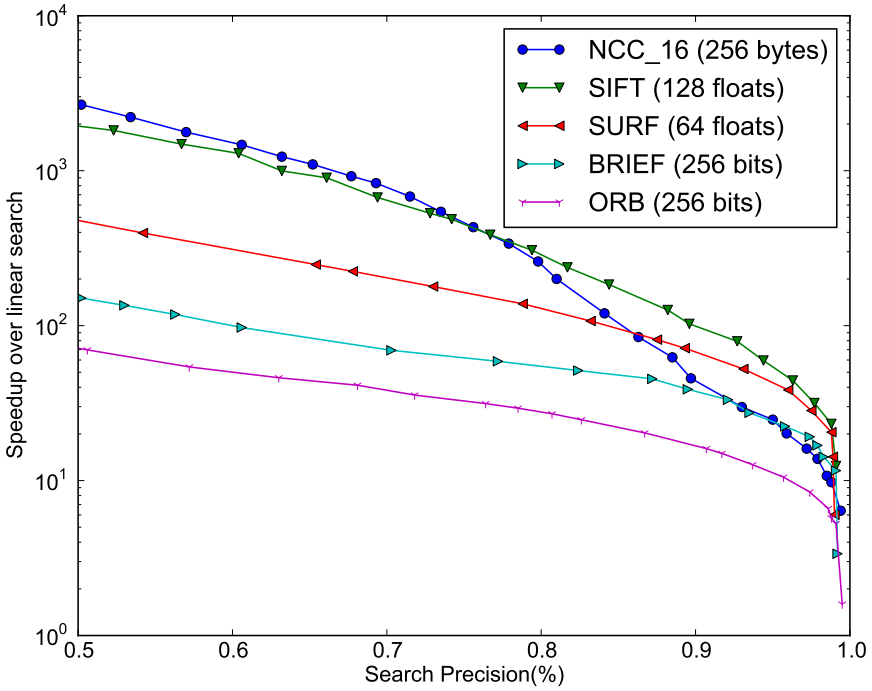

Figure 4.5: Speedup over linear for different popular features types (both binary and vector).

significant speedups over linear search ranging between one and two orders of magnitude for search precisions in the range 50-99%. This is less impressive than the search speedups obtained for the vector features, but this is because the linear search is much more efficient for binary features and all the speedups are computed relative to the linear search time for that particular feature. To better compare the efficiency of the nearest neighbour search between different features types, we show in figure 4.6 the same results, but this time using the absolute search time instead of search speedup. It can be seen that the search time for binary features (BRIEF, ORB) is similar to that of vector features (SIFT, SURF) for high search precisions.

In figure 4.7 we compare the hierarchical clustering tree with a multi-probe locality sensitive hashing implementation [Lv et al., 2007], also available in FLANN. For the comparison we used datasets of BRIEF and ORB features extracted from

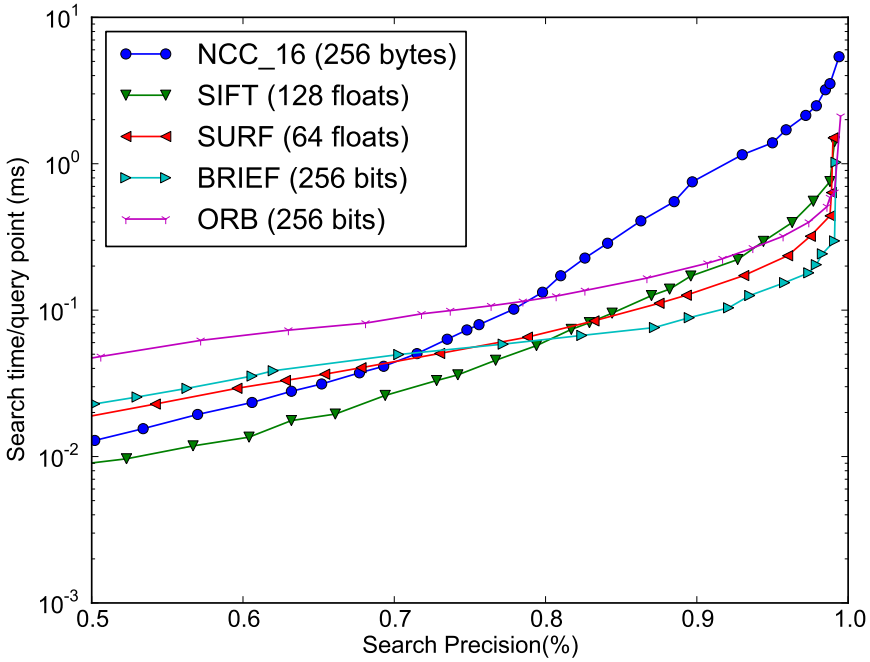

Figure 4.6: Absolute search time for different popular features types (both binary and vector).

the Winder/Brown datasets and from the recognition benchmark images dataset of [Nister and Stewenius, 2006]. The first dataset contains approximately 100,000 features, while the second contains close to 5 million features. It can be seen that for the first dataset the hierarchical clustering tree performs better for high search precisions ($> 80\%$), while the LSH implementation performs better for lower search precisions ($< 80\%$), while for the second dataset the hierarchical tree is faster than LSH for all precisions. This also shows that the algorithm proposed scales well with respect to the dataset size.

Another thing to note is that the LSH implementation requires significantly more memory compared to the hierarchical clustering trees for when high precision is required, as it needs to allocate a large number of hash tables to achieve the high search precision. In the experiments from figure 4.7 LSH required 6 times more memory than the hierarchical search in case of the larger dataset. Having both

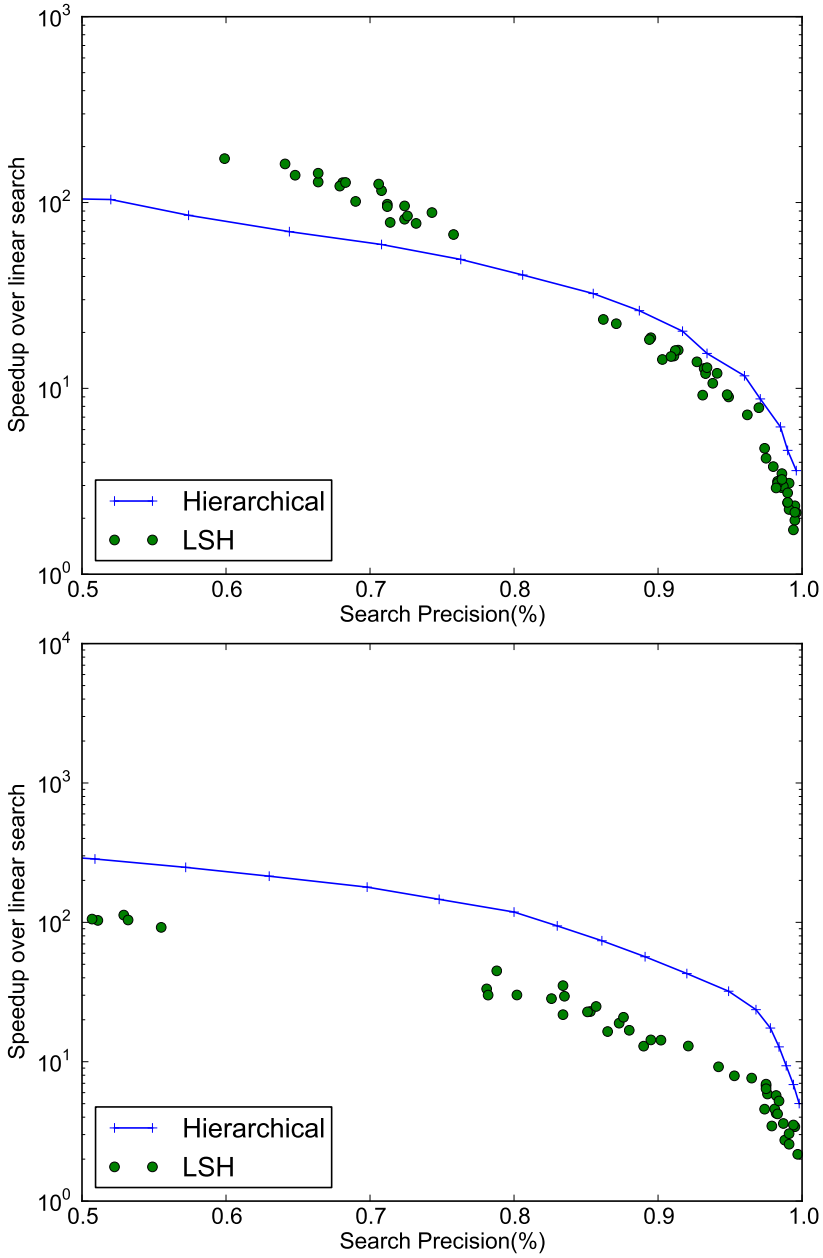

Figure 4.7: Comparison between the hierarchical clustering index and LSH for the Winder/Brown dataset of about 100,000 features (top) and the Nister/Stewenius recognition benchmark images dataset of about 5 million features (bottom).

LSH and the hierarchical clustering trees implementations in FLANN makes it possible to use the automatic algorithm configuration from FLANN, which takes into account both memory constraints as well as computation time to select the optimal algorithm for a specific dataset.

4.3 Chapter Summary

In this chapter we introduced a new algorithm for fast approximate matching of binary features using parallel search of randomized hierarchical trees.

We have shown that the proposed algorithm, although simple to implement and efficient to run, is very effective at finding nearest neighbours of binary features. We have also shown that the performance of the algorithm is in many cases better than that of multi-probe LSH, the algorithm most often used at present for binary feature matching, and that it scales well for large datasets.

Chapter 5

Scaling Nearest Neighbour Matching

As previously discussed in chapter 1, many papers have shown that using simple non-parametric methods in conjunction with large scale datasets can lead to very good recognition performance. Torralba et al. [2008a] use a large dataset of nearly 80 million tiny 32x32 pixel images to perform several recognition experiments using simple nearest neighbour techniques and obtain very good results comparable to leading class specific detectors for the richly represented classes. Hays and Efros [2007] demonstrate an automatic scene completion algorithm that uses nearest neighbour searches in a large scale dataset to find semantically similar images that can be used to complete missing parts of a query image. Hays and Efros [2008] propose a system that, using nearest neighbour lookups in a very large database of images, can guess the location where a picture was taken. Similar results have been shown in linguistics, where machine translation and speech recognition using large n-grams models outperform more elaborate models using grammars and phrase structure [Halevy et al., 2009].

All the above papers mention that when using small datasets the result obtained were not impressive, but increasing the size of the dataset by orders of magnitude leads to quantifiable leaps in performance.

Scaling to large datasets such as those used in the above papers is a difficult task. One of the main challenges is the impossibility of loading the data into the

main memory of a single machine. For example the size of the raw tiny images dataset of [Torralba et al., 2008a] is about 240 GB, which is greater than what can be found on most computers at present. For this reason, the authors of [Torralba et al., 2008a] only used the first 19 principal components to perform the nearest neighbour search, in order to be able to keep the data in memory. When they used a k-d tree to speedup the search, they had to limit the principal components to 17 due to the k-d tree memory overhead.

Fitting the data in memory is even more problematic for datasets of the size of those used in [Deng et al., 2009; Hays and Efros, 2007, 2008]. When dealing with such large amount of data there are several possible solutions:

- Perform some data dimensionality reduction and transform the data points into a space of lower dimensionality so that the data fits into memory. This approach is used in several papers in the literature [Jégou et al., 2010; Torralba et al., 2008a,b; Weiss et al., 2008]. Although it has the advantage of significantly reducing the storage requirements, this solution has some disadvantages as well: projecting the data into a lower dimensionality space typically involves a distortion of the relations between the points, introducing further approximations. There is a limit on how much the dimensionality of the new space can be reduced before the performance is significantly degraded. Even with the dimensionality reduction, when the size of the data increases past some point, the problem of fitting all the data in memory still remains.

- Another option is to only load parts of the data into the main memory at a time. This approach involves storing the data on a secondary memory, such as a hard drive, and only loading into main memory the parts of the data that are needed. The main issue with this approach is that currently there is a performance gap of several orders of magnitude between the primary and secondary memory, which significantly impacts the efficiency of most algorithms. Additionally, nearest neighbour methods usually require a large number of random accesses to points distributed throughout the dataset. Due to the fact that reading from the secondary memory is done in blocks of typically much higher granularity than the size of a data point, a large amount

of data has to be exchanged between the main and the secondary memory and many random accesses in the secondary memory are required, causing a significant decrease in performance for the nearest neighbour search operations.

- A third approach is to distribute the computation to more than one machine. This is the most common approach for scaling to large amounts of data at the moment, used in many places in the industry and well as academia. A typical example from the industry is Google who has popularised the Map-Reduce algorithm [Dean and Ghemawat, 2008] and successfully used it to process large amounts of data spread across a very large numbers of servers. This is also the approach we take in FLANN, and present in this chapter.

In addition to the memory limitation that needs to be overcome, scaling to very large datasets also involves increased computational burdens. From this point of view as well the approach of spreading the computation across a compute cluster is the most appealing as it decreases the amount of computation required for one machine, increasing the overall efficiency.

5.1 Scaling Nearest Neighbour Matching on a Compute Cluster

As mentioned above, in order to scale to very large datasets, we use the approach of distributing the data to multiple machines in a compute cluster and perform the nearest neighbour search using all the machines in parallel. The data is distributed equally between the machines, such that for a cluster of N machines each of them will only have to index and search $1/N$ of the whole dataset (although the ratios can be changed to have more data on some machines than others). The final result of the nearest neighbour search is obtained by merging the partial results from all the machines in the cluster once they have completed the search.

Since each machine indexes and searches only a fraction of the data, each machine needs to do less work to obtain the same overall performance compared to the non-distributed case. We perform several experiments that examine how the number of machines (and number of processes running on each machine) influence the

overall performance.

In order to distribute the nearest neighbour matching in FLANN on a compute cluster we implemented a Map-Reduce like algorithm using the Message Passing Interface (MPI) specification.

MPI is a standardized and portable message passing system design for efficient communication between a group for parallel processes. MPI was designed with high-performance distributed computing in mind and it allows for a rich range of capabilities such as synchronous and asynchronous point-to-point operations (send, receive), collective operations (broadcast, scatter, reduce, gather), node synchronization (barrier operation) as well as obtaining network topology related information such as total number of processes running in a compute cluster and the rank of each process.

There are several implementations of the MPI specification and FLANN can use any of them. In our experiments we have used OpenMPI, a popular open source MPI implementation

Since we are using a "Map-Reduce like" algorithm for distributing nearest neighbour searches across a compute cluster, an obvious question would be why not use one of the Map-Reduce frameworks available, such as Apache Hadoop, an open source Map-Reduce implementation. The main reason for choosing MPI over a Map-Reduce framework is the better latency MPI offers compared to most Map-Reduce frameworks, allowing for better overall performance. In addition, MPI offers a richer set of primitives for better control over the operations executed by the parallel process executing a particular algorithm. However, MPI is currently lacking the fault-tolerance functionality present in map-reduce implementations such as Hadoop, adding fault tolerance to MPI is being discussed for future versions of the specification.

Algorithm 7 describes the procedure for building a distributed nearest neighbour matching index. Each process in the cluster executes in parallel and reads from a distributed filesystem a fraction of the dataset. The fraction of the dataset assigned to each process is determined by using the rank of the process in the MPI cluster topology (each process in the MPI cluster has an unique rank between 0 and the number of processes). The allocation of the data to processes can be randomized in order to avoid the case when the dataset assigned to one process contains

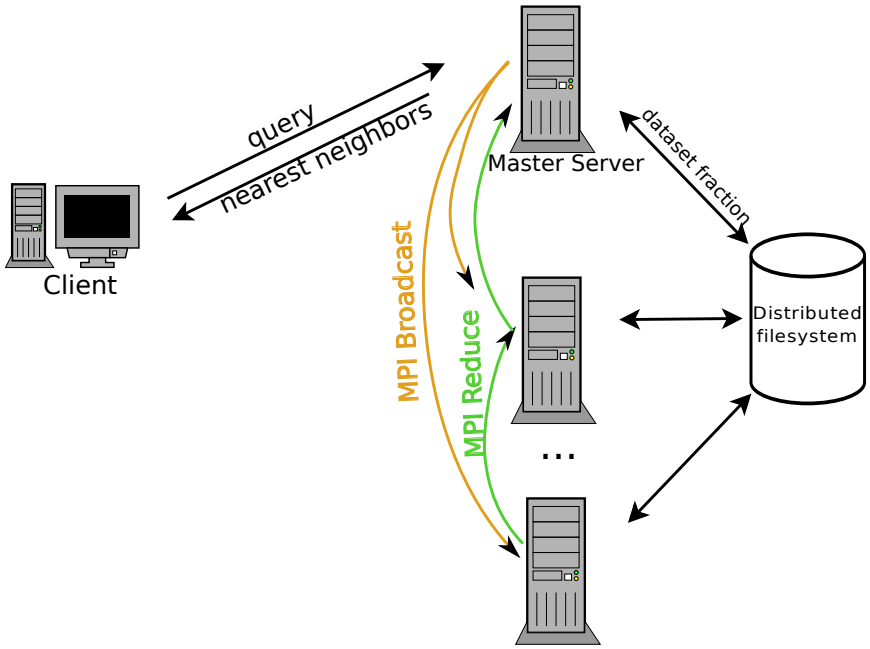

Figure 5.1: Scaling nearest neighbour search on a computer cluster using Message Passing Interface (MPI) standard.

most of the close neighbours of a query while the rest contain far neighbours.

After a fraction of the dataset is assigned to each process, all processes build in parallel their own nearest neighbour indexes using the corresponding dataset fractions. The index built by each process can be any of the algorithms supported in FLANN, such as the randomized k-d tree or the priority search k-means tree. At the end all the processes are synchronized to make sure the index construction has completed on all of them and they are ready to receive queries.

In order to search the distributed index the query is sent from a client to one of the computers in the MPI cluster, which we call the master server (see figure 5.1). By convention the master server is the process with rank 0 in the MPI cluster, however any process in the MPI cluster can play the role of master server.

The master server broadcasts the query to the all the processes in the cluster and then each process can run the nearest neighbour matching in parallel on its own fraction of the data. When the search is complete an MPI reduce operation is

Algorithm 7 Constructing a distributed index on a compute cluster

Input: dataset D, index parameters P
procedure BUILDINDEX(D,P)
1: $i \leftarrow MPI\_rank()$
2: $D_i \leftarrow$ read portion of the dataset D corresponding to rank i using the distributed filesystem
3: build index in parallel on each process with dataset D_i and parameters P
4: $MPI\_barrier()$ // synchronize all processes

used to merge the results back to the master process and the final result is returned to the client.

The master server is not a bottleneck when merging the results. The MPI reduce operation is also distributed, the partial results are merged two by two in a hierarchical fashion from the servers in the cluster to the master server. Additionally, the merge operation is very efficient, the distances between the query and the neighbours don't have to be re-computed as they are returned by the nearest neighbour search operations on each server.

Algorithm 8 Searching a distributed index on a compute cluster

Input: query Q, search parameters P
procedure SEARCHINDEX(Q,P)
1: $MPI\_broadcast(Q,P)$ // broadcast query and parameters to all processes
2: $NN_i \leftarrow$ run nearest neighbour search with query Q and parameters P on each process i
3: $NN \leftarrow MPI\_reduce(NN_i)$ // merge results using a MPI reduce operation
4: **return** NN

When distributing a large dataset for the purpose of nearest neighbour search we chose to partition the data into multiple disjoint subsets and construct independent indexes for each of those subsets. During search the query is broadcast to all the indexes and each of them performs the nearest neighbour search within its associated data. Aly et al. [2011] introduce a distributed k-d tree implementation where they propose an alternative method of partitioning the data and querying the distributed trees. They place a root k-d tree on top of all the other trees (leaf trees) with the role of selecting a subset of trees to be searched and only send the query

to those trees. They show the distributed k-d tree has higher throughput compared to using independent trees, due to the fact that only a portion of the trees need to be searched by each query.

The partitioning of the dataset into independent subsets, as described above and implemented in FLANN, has the advantage that it doesn't depend on the type of index used (randomized kd-trees, priority search k-means tree, hierarchical clustering, LSH) and can be applied to any current or future nearest neighbour algorithm in FLANN. In the distributed k-d tree implementation of [Aly et al., 2011] the search does not backtrack in the root node, so it is possible that subsets of the data containing near points are not searched at all if the root k-d tree doesn't select the corresponding leaf k-d trees at the beginning. However, we consider it an approach worth investigating as a possible future extension to the distributed nearest neighbour matching framework in FLANN.

5.2 Experiments

In this section we present several experiments that demonstrate the effectiveness of the distributed nearest neighbour matching framework in FLANN. For these experiments we have used the 80 million patch dataset of Torralba et al. [2008a].

In an MPI distributed system it's possible to run multiple parallel processes on the same machine, the recommended approach is to run as many processes as CPU cores on each machine. Figure 5.2 presents the results of an experiment in which we run multiple MPI processes on a single machine with 8 CPU cores. It can be seen that the overall performance improves when increasing the number of processes from 1 to 4, however there is a decrease in performance when moving from 4 to 8 parallel processes. This can be explained by the fact that increasing the parallelism on the same machine also increases the number of request to the main memory (since all processes share the same main memory), and at some point the bottleneck moves from the CPU to the memory. Increasing the parallelism past this point results in decreased performance. Figure 5.2 also shows the direct search performance obtained by using FLANN directly without the MPI layer. As expected, the direct search performance is identical to the performance obtained when using the MPI layer with a single process, showing no significant overhead

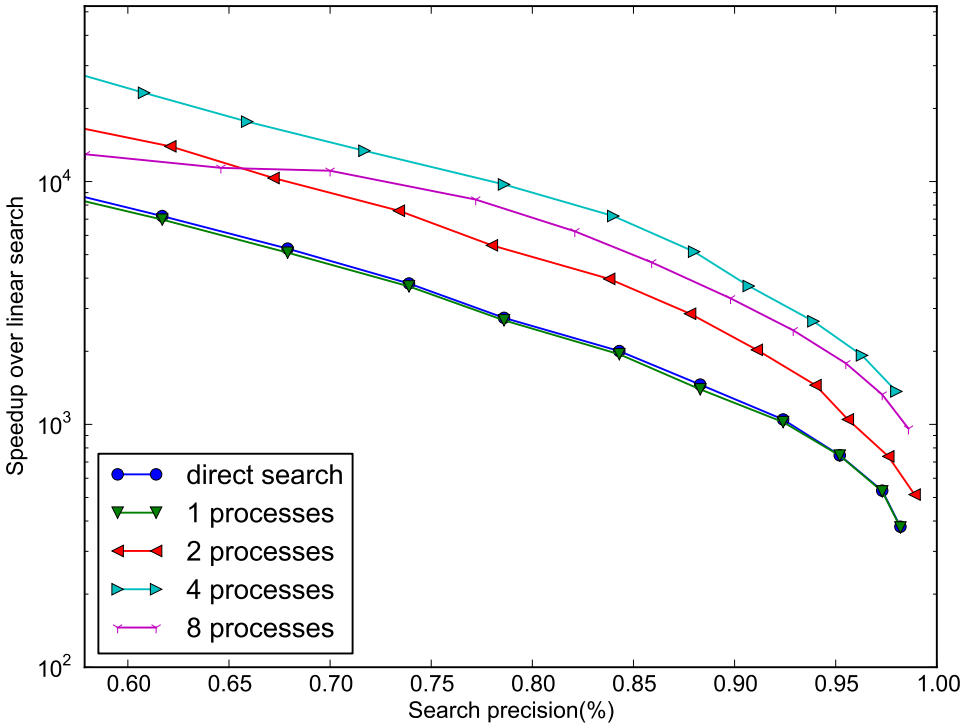

Figure 5.2: Distributing nearest neighbour search on a single multi-core machine. When the degree of parallelism increases beyond a certain point the memory access becomes a bottleneck. The "direct search" case corresponds to using the FLANN library directly, without the MPI layer.

from the MPI runtime. For this experiment and the one in figure 5.3 we used a subset of only 8 million tiny images to be able to run the experiment on a single machine.

Figure 5.3 shows the performance obtained by using 8 parallel processes on one, two or three machines. Even though the same number of parallel processes are used, it can be seen that the performance increases when those processes are distributed on more machines. This can also be explained by the memory access overhead, since when more machines are used, fewer processes are running on each machine, requiring fewer memory accesses.

We use the full 80 million tiny images dataset to demonstrate the scalability

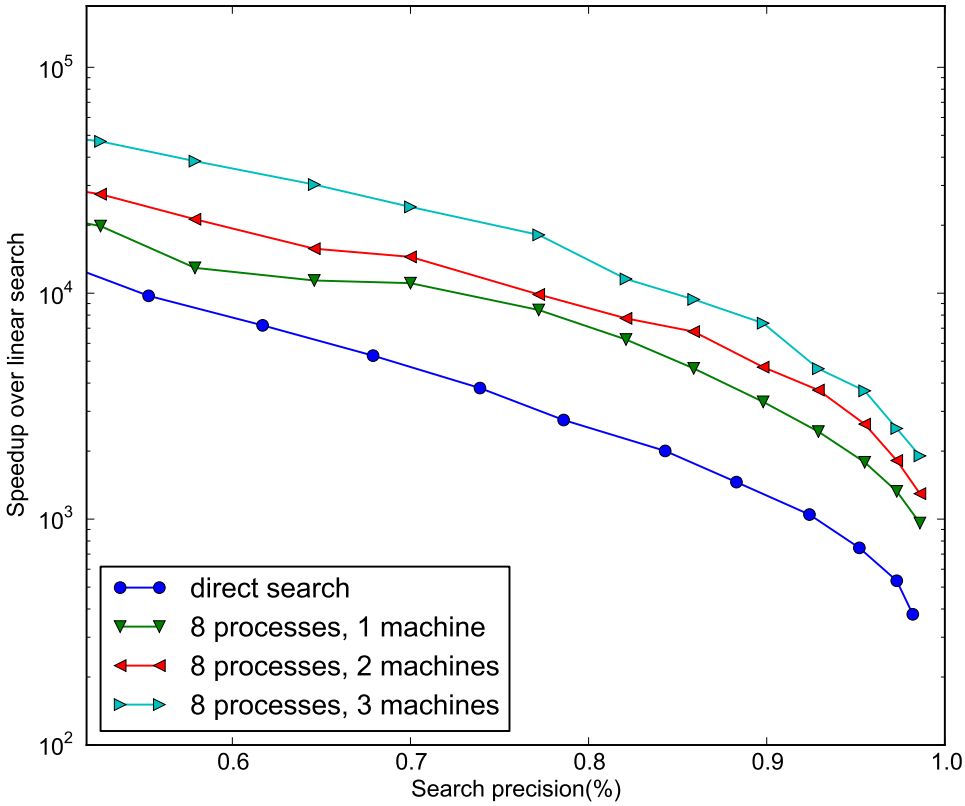

Figure 5.3: The advantage of distributing the search to multiple machines. Even when using the same number of parallel processes, distributing the computation to multiple machines still leads to an improvement in performance due to less memory access overhead. "Direct search" corresponds to using FLANN without the MPI layer and is provided as a comparison baseline.

of the distributed nearest neighbour matching framework presented in this chapter. Figure 5.4 shows the search speedup for a dataset of 80 million BRIEF features extracted from the images in the dataset. For this experiment we have used the Hierarchical Clustering Tree introduced in chapter 4. It can be seen that the search performance scales well with the dataset size and it benefits considerably from using multiple parallel processes.

In the experiment presented in figure 5.5 we match the images directly using

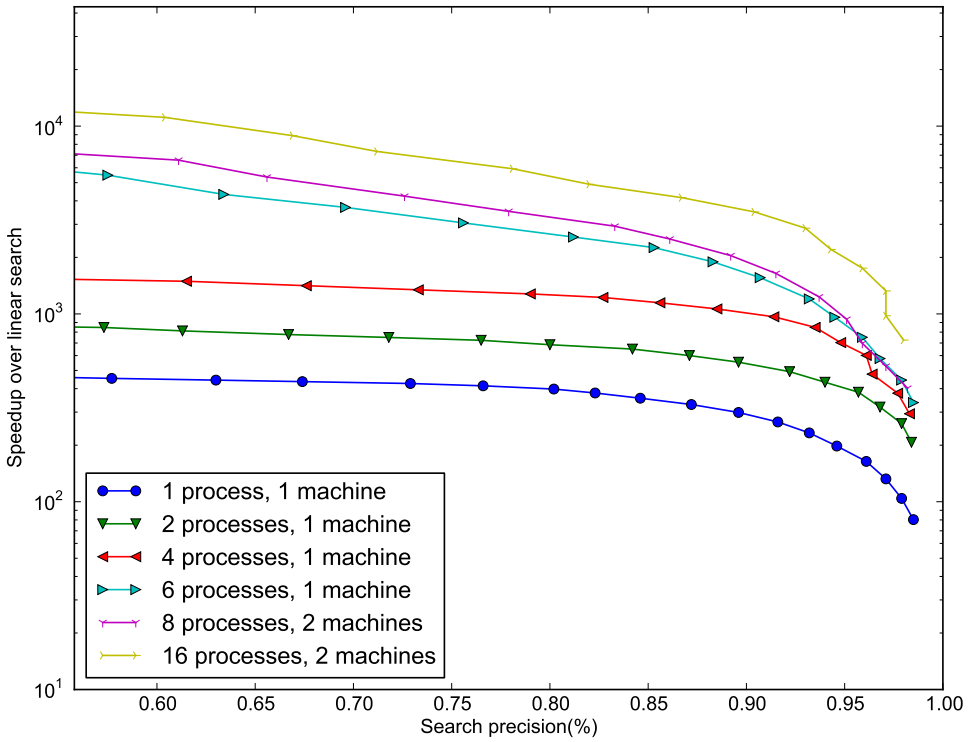

Figure 5.4: Scaling binary feature matching using distributed nearest neighbour search on a dataset of 80 million BRIEF features

the Euclidean distance (after we normalize them first), instead of extracting visual features. The algorithm used is the radomized k-d tree forest as it was determined by the auto-tuning procedure to be the most efficient in this case. In this case we obtain very high speedups over the linear search, as the multiple k-d trees can effectively exploit the correlations present between image dimensions. We still observe performance benefits from distributing the nearest neighbour matching between multiple processes/machines.

All the previous experiments have shown that distributing the nearest neighbour search to multiple machines results in an overall increase in performance. Ideally, when distributing the search to N machines the speedup would be N times higher, however in practice for approximate nearest neighbour search the speedup is smaller due to the fact that the search on each of the machines has sub-linear

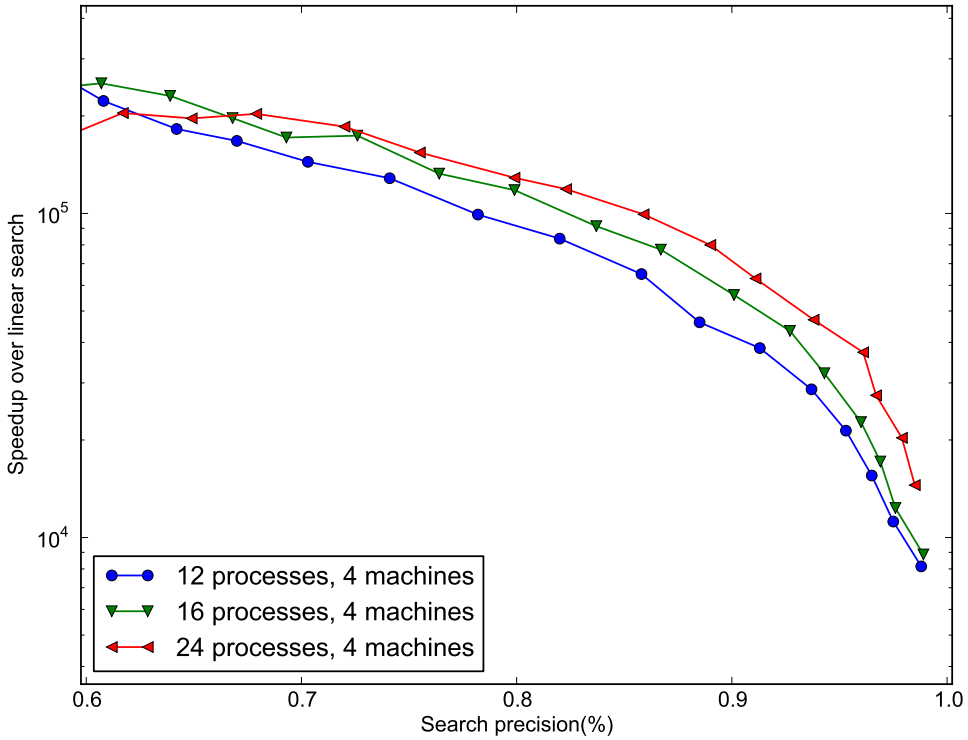

Figure 5.5: Matching 80 million tiny images directly using a compute cluster.

complexity in the size of the input dataset. Additionally, the performance of a distributed nearest neighbour search is influenced by factors such as the memory bandwidth (as shown in figures 5.2 and 5.3) and network latency.

5.3 Chapter Summary

In this chapter we presented a framework for distributing the nearest neighbour search on a compute cluster in order to be able to scale to vary large datasets, that would otherwise not fit into the main memory of a single computer. In addition to the advantage of overcoming the memory limitation, the performance is improved as well by distributing the computation across multiple machines.

We use the Message Passing Interface (MPI) standard for a high performance, portable way of exchanging data between multiple parallel processes. The solu-

tion proposed is implemented in the FLANN library and works with any nearest neighbour algorithm in the library.

We also present a set of experiments that show the scalability of the approach and the performance benefits obtained from using a compute cluster to perform distributed nearest neighbour matching.

Chapter 6

Conclusions

This thesis addresses the problem of fast nearest neighbour search in high dimensional spaces, a core problem in many computer vision and machine learning algorithms and which is often the most computationally expensive part of these algorithms. We looked at approximate nearest neighbour algorithms, which can speed up nearest neighbour search in applications where exact search is too inefficient.

This thesis presents and compares the algorithms we have found to work best at fast approximate search in high dimensional spaces and addresses the issues arising when scaling to very large size datasets. The work presented in this thesis has been included in an open source library called FLANN (Fast Library for Approximate Nearest Neighbours) which has become very popular in the computer vision community.

In chapter 3 we present the two algorithms we have found to work best for matching high dimensional features: the randomized k-d trees and the priority search k-means tree. The randomized k-d trees, previously proposed in [Silpa-Anan and Hartley, 2008], constructs a forest of k-d trees which are searched in parallel using a priority queue. The priority search k-means tree is a new algorithm that we propose and which constructs a search tree by recursive k-means clustering and then searches the tree using a best-bin-first strategy. We compare and evaluate these algorithms on a wide range of datasets and find that the best algorithm and its optimal parameters depend on dataset characteristics such as data dimensionality, dataset size and the statistics of the data stored in it (whether there are correla-

tions between the dataset features or not). We propose a method for automatically choosing the optimum algorithm and its parameters by using a grid search in the parameter space followed by a fine grained optimization using the Nelder-Mead downhill simplex method.

Binary features have recently become popular in the computer vision literature due to their compactness (requiring less storage) and the fact that they can be efficiently matched using bitwise operations and popcnt (population count) machine instructions. Many algorithms that worked well for classic vector features are not applicable or not efficient for binary features. In chapter 4 we introduced a new algorithm for fast approximate matching of binary features, which we found to perform as well or better than algorithms typically used in the literature for matching binary features.

Using very large training sets has been shown to improve the performance of many recognition algorithms. In chapter 5 we have addressed the issues arising from scaling to very large datasets that don't fit in the memory of a single machine. We introduced an algorithm for distributed nearest neighbour matching on compute clusters and presented several experiments that show its effectiveness.

6.1 Future Directions

We conclude this thesis by proposing several directions for future exploration:

- **Evaluate search performance on different datasets from other domains.** The nearest neighbour search algorithms presented in this thesis and implemented in FLANN are generic and applicable to any data from any domain (with some restrictions, for example the Randomized K-D Trees and the Hierarchical K-means require the data to be in an Euclidean space). Although we have reports of FLANN being used in other domains such as text or audio, we have evaluated FLANN mainly on computer vision specific datasets consisting of visual features and image patches. It would be interesting to perform further experiments to see how FLANN performs on datasets of various sizes and data dimensionality from other domains.

- **Extend the library with new methods that prove to be effective on par-**

ticular datasets. We have shown in chapter 3 that the optimum nearest neighbour search algorithm depends on the dataset on which it is used. Although we have tested and compared the algorithms implemented in FLANN against many other algorithms proposed in the literature and found them to perform the best, it is possible that some other algorithms will be found to work better than the ones in FLANN on specific datasets.

FLANN has a flexible architecture and can easily be extended with new nearest neighbour algorithms. One possible such algorithm is the k-nearest neighbour graph algorithm presented in [Hajebi et al., 2011] and which the authors claim to compare favourably to the randomized k-d trees. It would be interesting to do more extensive comparisons and possibly include this algorithm in FLANN if it proves to work better in certain cases.

Because FLANN is an open source library other users can extend it with special purpose nearest neighbour algorithms that work better for specific applications. Any external contributions can be evaluated and merged in the library.

- **Extend/improve the automatic algorithm configuration.** The automatic algorithm configuration procedure described in chapter 3 uses a grid search approach to determine the best parameters to use for a specific dataset. Grid search is used to find the best parameters because it is known to be an effective strategy when there are relatively few parameters involved, as is the case for the nearest algorithms presented in this thesis. However, the grid search cost increases exponentially with the number of parameters, making the addition of a nearest neighbour algorithm with many adjustable parameters problematic.

  An interesting future direction is the addition of other algorithms for parameter tuning such as the random search proposed by [Bergstra and Bengio, 2012] or the ParamILS proposed by [Hutter et al., 2009].

- **Scale to larger datasets.** In chapter 5 we showed that the cluster computing algorithm implemented in FLANN scales well to large datasets bringing significant performance benefits while overcoming the memory limitations of a

single machine. It would be useful to experiment with even larger datasets using bigger compute clusters.

- **Improve the throughput of the distributed nearest neighbour search.** The distributed nearest neighbour search algorithm proposed in chapter 5 has the advantage that it is independent of the nearest neighbour algorithm implemented by each cluster node. However, during a search, each of the cluster nodes have to search their respective sub-datasets and all the results get merged at the end. Aly et al. [2011] show that using a root tree that selects which machines in the compute cluster participate in the search based on the search query can significantly increase the search throughput.

  Exploring this direction of adding an "selector" tree at the top of the machines in a distributed nearest neighbour search cluster, is an interesting direction for future research. The addition of the selector tree can hurt the accuracy of the results by excluding parts of the data from potentially being explored and further research is warranted to see how the performance is affected in practice.

- **Implement a cluster computing layer based on Map/Reduce.** The cluster computing framework currently implemented in FLANN and descried in chapter 5 is based on the Message Passing Interface (MPI) specification, due to the better latency MPI has compared to existing Map/Reduce frameworks. However, MPI currently lacks the fault tolerance guarantees that most Map/Reduce frameworks offer. For environments where fault tolerance is critical, a Map/Reduce distribution framework could be preferable and make be a good addition to the library.

Bibliography

B. Adenso-Diaz and M. Laguna. Fine-tuning of algorithms using fractional experimental designs and local search. *Operations Research*, 54(1):99–114, 2006. → pages 26

M. Aly, M. Munich, and P. Perona. Distributed kd-trees for retrieval from very large image collections. In *British Machine Vision Conference (BMVC), Dundee, UK*, 2011. → pages 77, 78, 87

A. Andoni and P. Indyk. Near-Optimal hashing algorithms for approximate nearest neighbor in high dimensions. *COMMUNICATIONS OF THE ACM*, 51 (1):117, 2008. → pages 18, 50

D. Arthur and S. Vassilvitskii. K-Means++: the advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 2007. → pages 42

S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998. → pages 14, 32, 49

M. Bawa, T. Condie, and P. Ganesan. LSH forest: self-tuning indexes for similarity search. In *Proceedings of the 14th international conference on World Wide Web*, page 651–660, 2005. → pages 18, 25

H. Bay, T. Tuytelaars, and L. Van Gool. SURF: Speeded Up Robust Features. *European Conference on Computer Vision*, page 404–417, 2006. → pages 22, 59

J. S. Beis and D. G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*, page 1000–1006, 1997. → pages 14

J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975. → pages 13, 31

A. C. Berg, T. L. Berg, and J. Malik. Shape matching and object recognition using low distortion correspondences. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, page 26–33, 2005. → pages 1

J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13:281–305, 2012. → pages 26, 86

A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd International Conference on Machine learning*, ICML '06, page 97–104, New York, NY, USA, 2006. ACM. ISBN 1-59593-383-2. → pages 16

G. Bradski and A. Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, 2008. → pages 10

S. Brin. Near neighbor search in large metric spaces. In *VLDB*, pages 574–584, 1995. ISBN 1-55860-379-4. → pages 8, 16, 38, 41, 61

M. Calonder, V. Lepetit, C. Strecha, and P. Fua. BRIEF: Binary Robust Independent Elementary Features. In *European Conference on Computer Vision*, Sept. 2010. → pages 22, 59

O. Chum, J. Philbin, and A. Zisserman. Near duplicate image detection: min-hash and tf-idf weighting. In *Proceedings of the British Machine Vision Conference*, volume 3, page 4, 2008. → pages 23

O. Chum, M. Perdoch, and J. Matas. Geometric min-hashing: Finding a (thick) needle in a haystack. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, page 17–24, 2009. → pages 23

S. P. Coy, B. L. Golden, G. C. Runger, and E. A. Wasil. Using experimental design to find effective parameter settings for heuristics. *Journal of Heuristics*, 7(1):77–97, 2001. → pages 26

M. Cummins and P. Newman. Highly scalable appearance-only SLAM-FAB-MAP 2.0. In *Proceedings of Robotics: Science and Systems (RSS)*, volume 5, 2009. → pages 9

S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, page 537–546, 2008. → pages 15

J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. → pages 74

J. Deng, W. Dong, R. Socher, L. J. Li, K. Li, and L. Fei-Fei. ImageNet: a large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, page 248–255, 2009. → pages 5, 21, 73

J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977. → pages 13, 31

K. Fukunaga and P. M. Narendra. A branch and bound algorithm for computing k-Nearest neighbors. *IEEE Trans. Comput.*, 24:750–753, 1975. → pages 16, 19, 38

A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the International Conference on Very Large Data Bases*, page 518–529, 1999. → pages 23

K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Two*, page 1312–1317, 2011. → pages 19, 86

A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *Intelligent Systems, IEEE*, 24(2):8–12, 2009. → pages 72

M. Havlena, A. Torii, M. Jancosek, and T. Pajdla. Automatic reconstruction of mars artifacts. In *European Planetary Science Congress 2009, held 14-18 September in Potsdam, Germany. http://meetings. copernicus. org/epsc2009, p. 280*, page 280, 2009a. → pages 9

M. Havlena, A. Torii, J. Knopp, and T. Pajdla. Randomized structure from motion based on atomic 3D models from camera triplets. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, page 2874–2881, 2009b. → pages 9

J. Hays and A. A. Efros. Scene completion using millions of photographs. In *ACM Transactions on Graphics (TOG)*, volume 26, page 4, 2007. → pages 3, 72, 73

J. Hays and A. A. Efros. IM2GPS: estimating geographic information from a single image. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, page 1–8, 2008. → pages 4, 72, 73

J. He, W. Liu, and S. F. Chang. Scalable similarity search with optimized kernel hashing. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, page 1129–1138, 2010. → pages 18

F. Hutter. *Automated configuration of algorithms for solving hard computational problems*. PhD thesis, University of British Columbia, 2009. → pages 26

F. Hutter, H. H. Hoos, and K. Leyton-Brown. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36: 267–306, 2009. → pages 27, 86

P. Jain, B. Kulis, and K. Grauman. Fast image search for learned metrics. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, page 1–8, 2008. → pages 18

H. Jégou, M. Douze, and C. Schmid. Hamming embedding and weak geometric consistency for large scale image search. *European Conference on Computer Vision (ECCV)*, page 304–317, 2008. → pages 21, 23

H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern*, page 1–15, 2010. ISSN 0162-8828. → pages 21, 73

Y. Jia, J. Wang, G. Zeng, H. Zha, and X. S. Hua. Optimizing kd-trees for scalable visual descriptor indexing. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, page 3392–3399, 2010. → pages 15

B. Kulis and T. Darrell. Learning to hash with binary reconstructive embeddings. *Advances in neural information processing systems*, 22:1042–1050, 2009. → pages 18

B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *Computer Vision, 2009 IEEE 12th International Conference on*, page 2130–2137, 2009. → pages 18

N. Kumar, L. Zhang, and S. Nayar. What is a good nearest neighbors algorithm for finding similar patches in images? *Computer Vision–ECCV 2008*, page 364–378, 2008. → pages 16

B. Leibe, K. Mikolajczyk, and B. Schiele. Efficient clustering and matching for object class recognition. In *Proc. BMVC*, 2006. → pages 20

S. Leutenegger, M. Chli, and R. Siegwart. BRISK: Binary Robust Invariant Scalable Keypoints. In *IEEE International Conference on Computer Vision (ICCV)*, 2011. → pages 22, 59

T. Liu, A. Moore, A. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. In *Neural Information Processing Systems*, 2004. → pages 17

D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004. → pages 1, 22, 59

Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*, page 950–961, 2007. → pages 18, 68

K. Mikolajczyk and J. Matas. Improving descriptors for fast tree matching by optimal linear projection. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–8, 2007. ISBN 1550-5499. → pages 17, 53

A. W. Moore. The anchors hierarchy: Using the triangle inequality to survive high dimensional data. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, page 397–405, 2000. → pages 16

M. Muja and D. G. Lowe. FLANN: Fast Library for Approximate Nearest Neighbors. http://people.cs.ubc.ca/˜mariusm/index.php/FLANN/FLANN. → pages 9, 63

M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application VISSAPP'09)*, page 331–340, 2009. → pages iii, 14, 25, 63, 66

M. Muja and D. G. Lowe. Fast matching of binary features. In *Computer and Robot Vision (CRV), 2012 Ninth Conference on*, page 404–410, 2012. → pages iii, 24

J. A. Nelder and R. Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965. → pages 25, 56

D. Nister and H. Stewenius. Scalable recognition with a vocabulary tree. In *CVPR*, volume 2, pages 2161–2168. Ieee, 2006. ISBN 0-7695-2597-0. → pages 8, 19, 20, 21, 38, 44, 49, 69

M. Norouzi, A. Punjani, and D. J. Fleet. Fast search in Hamming space with multi-index hashing. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, page 3108–3115, 2012. → pages 23

J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *IEEE Conference on Computer Vision and Pattern Recognition, 2007. CVPR'07*, page 1–8, 2007. → pages 1, 4, 5, 21

M. Quigley, K. Conley, B. Gerkey, J. Faust, T. B. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source robot operating system. In *International Conference on Robotics and Automation*, Open-Source Software workshop, 2009. → pages 9

M. Raginsky and S. Lazebnik. Locality-sensitive binary codes from shift-invariant kernels. *The Neural Information Processing Systems*, 22, 2009. → pages 18

E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. ORB: an efficient alternative to SIFT or SURF. In *International Conference on Computer Vision*, Barcelona, 2011. → pages 22, 23, 59, 60

R. Salakhutdinov and G. Hinton. Semantic hashing. *International Journal of Approximate Reasoning*, 50(7):969–978, 2009. → pages 22, 60

G. Schindler, M. Brown, and R. Szeliski. City-Scale location recognition. In *CVPR*, pages 1–7, 2007. → pages 21, 41

T. B. Sebastian and B. B. Kimia. Metric-based shape retrieval in large databases. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 3, page 291–296, 2002. → pages 19

G. Shakhnarovich, P. Viola, and T. Darrell. Fast pose estimation with parameter-sensitive hashing. In *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, page 750–757, 2003. → pages 1, 18

C. Silpa-Anan and R. Hartley. Optimised KD-trees for fast image descriptor matching. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, page 1–8, 2008. → pages 8, 14, 33, 61, 84

J. Sivic and A. Zisserman. Video google: A text retrieval approach to object matching in videos. In *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, page 1470–1477, 2003. → pages 1

R. F. Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6(1):579–589, 1991. → pages 14

M. M. Stark and R. F. Riesenfeld. Wordnet: An electronic lexical database. In *Proceedings of 11th Eurographics Workshop on Rendering*, 1998. → pages 21

C. Strecha, A. Bronstein, M. Bronstein, and P. Fua. LDAHash: improved matching with smaller descriptors. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (99):1–1, 2010. → pages 23

A. Torralba, K. P. Murphy, W. T. Freeman, and M. A. Rubin. Context-based vision system for place and object recognition. In *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, page 273–280, 2003. → pages 3

A. Torralba, R. Fergus, and W. T. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(11):1958–1970, 2008a. ISSN 0162-8828. → pages 1, 2, 21, 22, 63, 72, 73, 78

A. Torralba, R. Fergus, and Y. Weiss. Small codes and large image databases for recognition. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, page 1–8, 2008b. → pages 20, 21, 22, 73

T. Trzcinski, V. Lepetit, and P. Fua. Thick boundaries in binary space and their influence on Nearest-Neighbor search. *Pattern Recognition Letters*, 2012. → pages 24

P. Turcot and D. G. Lowe. Better matching with fewer features: The selection of useful features in large database recognition problems. In *Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on*, page 2109–2116, 2009. → pages 9

J. Wang, S. Kumar, and S. F. Chang. Semi-supervised hashing for scalable image retrieval. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, page 3424–3431, 2010. → pages 18

J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li. Scalable k-NN graph construction for visual descriptors. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, page 1106–1113, 2012. → pages 19

Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *Advances in neural information*. NIPS, 2008. → pages 18, 21, 22, 73

S. Winder and M. Brown. Learning local image descriptors. In *CVPR*, pages 1–8, 2007. → pages 44, 62, 64

H. Xu, J. Wang, Z. Li, G. Zeng, S. Li, and N. Yu. Complementary hashing for approximate nearest neighbor search. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, page 1631–1638, 2011. → pages 18

P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, page 311–321, 1993. → pages 16

C. Zitnick. Binary coherent edge descriptors. *European Conference on Computer Vision*, page 1–14, 2010. → pages 23, 60