STRUCTURAL INDEXING USING LOCAL IMAGE FEATURES

GERTRUDA GROLINGER

A THESIS SUBMITTED TO THE SCHOOL OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF SCIENCE IN COMPUTER SCIENCE

GRADUATE PROGRAM IN DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TORONTO
TORONTO, ONTARIO
FEBRUARY 2009

STRUCTURAL INDEXING USING LOCAL IMAGE FEATURES

by **Gertruda Grolinger**

Supervisor:

1. Professor Sven Dickinson

Examination Committee Members:

1. Professor Sven Dickinson
2. Professor Allan D. Jepson

Acknowledgements

I would like to thank my supervisor, Professor Sven Dickinson, for his guidance and very useful comments and input. I appreciate very much the amount of time that he spent on discussing ideas because these discussions allowed me to evaluate the implemented ideas, to summarize, explain and understand the obtained results and plan the further path of the project.

I would also like to give many thanks to Professor Allan Jepson for making parts of his code available to us and for his help and very useful comments.

It is difficult to overstate my gratitude to Mike Jamieson for his mentorship. With his enthusiasm, his inspiration, and his great efforts to explain things clearly and simply, he helped to make research fun for me.

I would like to thank the many people who have provided us with resources needed for competition of this thesis: Marius Muja and Professor David Lowe for FLANN library, Professor David Lowe for his code for extracting SIFT features and David Nistér and Henrik Stewénius for their image dataset.

Further, I would like to thank my parents, Karmen and Eduard, for their support and understanding during the time I spent far from home, and my partner, Marcel Birkner, for staying with me and supporting me morally during the last few years of my education.

Last, but not least, I would like to thank my friends, especially, Tom, Nassim, and Sabbir, for their continuous support and encouragement.

Table of Contents

List of Figures

1 Abstract

Local interest point features are very popular in today's object recognition systems due to their distinctiveness and invariance to minor changes in lighting and viewpoint. However, as an object database grows to contain millions of images and when, due to occlusion or clutter, the number of object features in a query image represents a small fraction of the total number of image features, even reasonably distinctive single image features can yield a daunting number of candidate matches for expensive spatial verification. One way to reduce querying ambiguity and make the index richer and more specific may be to build some spatial information into the index using pairs of local features. In this report, we present a method for combining two local interest features into a compound feature. We show that while our compound features are much less ambiguous than singleton features (even at equivalent dimensionality), they are also less likely to survive intact in different images of an object. Studying the trade-offs of querying using compound features will give us insight into the possibilities of improving object appearance databases.

2 Introduction

The problem of object recognition is one of the most challenging and important problems in computer vision. Many of today's recognition systems extract a sparse set of local features, each of which describes a small, distinctive patch of image data along with its position, scale and orientation. The most popular such feature, called a SIFT (Scale Invariant Feature Transform) [8], characterizes the local image data in a manner that is invariant to minor changes in lighting and viewpoint. For each such feature in an input image, we can query a well-structured object database to learn which objects contain highly similar features. Objects that share many features with the input image (i.e., that recieve a large number of "feature votes") are considered more likely to be present in the input image. However, an object and input image can share many features in common without sharing a common spatial configuration. Therefore, recognition systems often apply a relatively expensive spatial consistency check to candidate objects. As the database grows to contain millions of object images, each individual feature tends to appear in a larger number of objects. Depending on the number and distribution of features in the input image, this can lead to a dauntingly large number of candidates to be spatially verified. Overall ambiguity is further compounded when the number of features on the target object represents a small fraction of the total number of image features due to occlusion or clutter. Instead of invoking geometric constraints only at verification time, this report explores ways to incorporate these constraints at

query time, with the aim of creating more distinctive features and thereby reducing the number of candidates to be verified.

One way to improve distinctiveness is to encode more information in each feature, such as using a larger patch size and higher dimensionality. However, by encoding more information, we would increase the memory footprint of each feature and therefore the size of the database as a whole. Instead, we propose a method for combining two local interest features into a single compound feature. In addition to retaining the encoding of each patch, we also have the option of encoding the spatial relationship between the features. Next, we reduce the dimensionality of the compound feature down to that of the original interest feature so that they have an equivalent memory footprint. Our results indicate that, even after this dimensionality reduction, invidual compound features are less ambiguous than singletons. On the other hand, the process of combining the features into meaningful groups of two comes at a certain cost in computational complexity and feature survivability. Pairs are more distinctive but also more fragile and slightly more expensive than singletons. Studying this trade-off will help us develop insight into the balance between bottom-up feature extraction and top-down model verification as a function of database size.

There are four main steps in our process: feature extraction, database construction, querying and scoring. Chapter 3 gives an overview of the related work. Chapter 4 compares the design of a singleton vs. a pair-based local feature. Chapter 5 talks about the index structure and query and voting processes, while Chapter 6 analyzes the advantages and disadvantages of a pair-based retrieval system. The experiments and obtained results are presented in Chapter 7, while conclusions and future work are outlined in Chapters 8 and 9, respectively.

3 Related work

The idea of grouping image features is not new; feature grouping was an essential part of most recognition systems of the 1970's and 80's. Researchers in the past were obliged to grouped features into geometric and relational structures in order to improve the distinctiveness of the relatively generic, ambiguous features (e.g., lines and corners) in use at the time. In this chapter, we briefly compare earlier and more recent recognition system in terms of the types of information they encode and how this information is organized and retrieved from the object index. We also sketch how the challenge of dealing with clutter and increasing database sizes may renew interest in the methods for grouping features before indexing.

Many earlier systems use features describing an object's shape or physical edges. These features are more generic and individually much less distinctive than local features that describe texture or local surface structure. For instance, Stein and Medioni [13], Huet and Hancock [6] and Beis and Lowe [2] use extracted edges. Costa and Shapiro [5] also include arcs and ellipses, and Lamdan *et al.* [7] record points of sharp convexities and deep concavities. Nister and Stewenius [11] use SIFT features ([8]) to encode image patches anchored at MSERS (Maximally Stable Extremal Regions) [9].

Systems that use individually ambiguous features use grouping to make their features richer and more specific. Lamdan *et al.* [7] use ordered point triples to define an affine basis of a plane while Stein and Medioni group consecutive line

segments into larger and more distinct entities. Beis and Lowe [2] form perceptually-significant edge groupings based on parallelism, cotermination, and symmetry.

Spatial relationships often play a vital role whenever compound features are formed by grouping. Costa and Shapiro [5] construct graphs where vertices represent features, and edges represent qualitative relationships (such as enclosing or coaxial). Huet and Hancock [6] include in their descriptor the relative angle and relative position of line segments. Beis and Lowe [2] incorporate angle and edge-length relationships between edge segments into their multidimensional feature vectors.

While more recent systems do not tend to use grouping or relationships between features, more sophisticated indexing structures and voting schemes are often used to make full use of the increased information in a local interest feature such as SIFT. Earlier systems, such as [5, 7, 13], often store extracted features in a hash table. Features are considered to match if they fall into the same hash bin. Beis and Lowe [2], though they use grouping to construct their features, also organize their features in a relatively sophisticated kd-tree. Bins neighboring a query are retrieved in order of proximity according to the Best Bin First (BBF) heuristic. This approach is used again in Lowe's SIFT paper [8], where it is paired with a voting scheme that discards query features that have ambiguous matches. We adopt the kd-tree structure and BBF method ourselves since it deals reasonably well with features in our chosen dimensionality range. Finally, Nister and Stewenius [11] also use a tree structure, hierarchical k-means, to help retrieve nearest-neighbors. They evaluate the match between a query image and an object image using a hierarchical voting scheme that takes into account the number of matching features and the implied ambiguity at each node in the tree.

While the object databases used to evaluate earlier systems tended to be fairly small, the deployment of ever larger databases demands highly distinctive features.

While systems such as [2, 3, 6] might use anywhere from a few dozen up to a thousand object images, today there is a need for databases that contain hundreds of millions of images. Local features that are considered highly distinctive for small databases can become ambiguous at this scale. For instance, Nister and Stewenius [11] automatically dismiss a larger and larger portion of their feature tree as ambiguous as the size of the indexed image set increases.

Also driving the search for more distinctive features are the challenges of cluttered images and highly occluded objects. These two factors increase retrieval difficulty by reducing the portion of input features that describe the object. While grouping can aid retrieval by improving the distinctiveness of a feature, it also introduces the problem of improper grouping across object boundaries. Stein and Medioni [13] and Costa and Shapiro [5] allow for such errors by introducing a degree of flexibility into the group matching process. Huet and Hancock [6] attempt to account for occlusion and clutter by adding relational information to their feature groups, making them more specific and distinct. Beis and Lowe [2] use various combinations of primitive features to develop a rich set of redundant groupings as a counter to occlusion and clutter.

Considering the challenges of increasing database size and dealing with clutter and occlusion, the need to reduce feature ambiguity is clear. Therefore it is interesting to revisit the grouping techniques and relational information used in earlier work to boost the distinctiveness of indexed features. The following chapter details our method of constructing such compound local interest features.

4 Computing index features

The choice of features in a particular computer vision system is highly dependent on the specific problem. The design of a feature can have a huge impact on the system performance. In this chapter, we look into a common type of feature used, and present our method for extending it to form a more distinctive image feature. We do that by combining two features into a single compound feature with the goal of making the index richer and more specific. In addition to retaining the encoding of the patch that is contained in each feature, we consider encoding the relations between the features that we combine. In the end, to make the computation tractable and for fair comparison, we reduce the dimensionality of the compound feature down to that of a singleton.

4.1 Singletons

Given a set of images, we extract the SIFT features from them using the original implementation of the SIFT extractor [8] that generates 128-dimensional feature vectors. Further, we apply PCA to those features to reduce their dimensionality to a lower dimension in order to make the calculations and storage tractable. We use a range of dimensions in order to be able to compare the effect of dimensionality on a feature's distinctiveness.

4.2 Dimensionality reduction

Our choice of dimensionality is motivated in several ways. The kd-tree that we use as one of the index structures poses a limit to the effective dimensionality of the index features. The kd-tree is believed to be less effective with increasing dimensionality. Further, the dimensionality reduction allows features to be stored in memory when the database is on the order of millions of index features. Calculations with image features are also more tractable if the dimensionality of the features is lower.

The SIFT extractor [8] generates 128-dimensional features. We reduce the dimensionality of singletons and pairs of features to $nDim$ by applying PCA dimensionality reduction. In Chapter 7, we present tests for several different values of $nDim$ in order to be able to see the impact of dimension on the feature's distinctiveness. By keeping the dimensionality of singletons and pairs the same, we can compare single features and feature-pairs while factoring out the variations in performance of the index structure due to varying feature dimensionality.

4.3 Pairs

Our goal is to build feature-pairs based on singletons such that the pairs have the same dimensionality as the singletons. There are many ways in which we can create feature-pairs, and many different pairs that we can form. In this section, we explain how we create feature-pairs.

We avoid pairing each feature with each other (this would produce n^2 pairs for n singletons) since we want to keep the feature database the same size for singletons and for pairs. For this reason, we choose one pair per feature point. First, for each feature in an image, we take k (for example, 30) spatially closest points and pair it

with each of them, producing k feature-pairs. Next, we score these pairs according to spatial and feature-based metrics explained in Section 4.3.2, and choose the pair with the best score. Finally, we reduce the dimensionality of the composite pair feature vector to that of a singleton.

In the pair building process, we choose the k spatially closest points for each feature in the image because the points that are closer in space are more likely to belong to the same object. We want to avoid the situation where two elements of a feature-pair span two objects since that would decrease feature survivability across different images of the same object. We characterize the relationship between the pair elements by using spatial relations. An additional option is to encode the relationships between the features, such as relative scale, orientation, and proximity. We present these spatial relations in more detail in Section 4.3.1.

Next, each of these k feature-pairs is scored in order to choose the ones that will survive across different images of the same object. In addition to survivability, we want our feature-pairs to be distinctive. Balancing these two factors is a hard problem. The k chosen feature-pairs are scored according to three scoring criteria to produce the following scores: **relative scale score** λ_s, **spatial distance score** λ_d and **feature distance score** λ_f. These three scoring functions are explained in more detail in Section 4.3.2. The overall score λ_{pair} for each feature-pair is given by the product of the three scores. The feature-pair with the best overall score is chosen to form the final feature-pair that we keep.

Once the pairs are formed, their dimensionality is reduced, using PCA dimensionality reduction, to the same dimensionality as a singleton, and they are treated the same as singleton index features in our system.

4.3.1 Spatial relations

The spatial relations between points can be used in two ways: as a basis for pairing, and as additional information to make the feature-pair more distinctive. The spatial relations that we use are a modified version of the spatial relations used by Carneiro and Jepson in [4]. In this section, we present our method for incorporating the relational information between the elements of a feature-pair into the encoding of that pair. The relational information that we consider adding to a feature-pair includes relative scale, relative distance, and heading in both directions.

Relative scale between the two points A and B is defined as follows:

$$r_s = \frac{min(s_A, s_B)}{max(s_A, s_B)}, \tag{4.1}$$

where r_S is the relative scale between point A and point B, and s_A and s_B are the scales of A and B, respectively.

Relative distance between the points A and B is defined as follows:

$$r_d = \frac{|A, B|}{min(s_A, s_B)}, \tag{4.2}$$

where r_d is the relative distance between A and B, and $|A, B|$ is the Euclidean distance between A and B.

The heading between points A and B is defined in both directions, i.e., we have the "heading AB" and the "heading BA". It represents the difference between the orientation of point A and the angle to point B, and is defined as follows:

$$h_{AB} = \Delta_\Theta(tan^{-1}(y_A - y_B, x_A - x_B) - o_A), \tag{4.3}$$

where $\Delta_\Theta(.) \in [-\Pi, +\Pi]$ denotes the principle angle, h_{AB} stands for "heading AB", o_A is the orientation of A, and y_A, y_B, x_A, x_B are the coordinates of points A and

B. h_{BA} is defined in a similar way.

If we want to add these spatial relations to feature-pairs, we have to consider the scale of this information since otherwise it could end up representing an insignificant part of the feature vector and get lost in the dimensionality reduction. Before the values for the spatial relations between the two elements of a pair are added to the feature-pair's descriptor vector, they are scaled to an appropriate size. We scale the spatial relation values so that the range of all four values together represents approximately 30% of the feature-pair range. After the spatial relations are added to a full dimensional ($128 + 128 = 256$) feature-pair vector, we reduce its dimensionality by applying PCA dimensionality reduction and bring it down to the same dimension as that of a singleton feature.

4.3.2 Pairing criteria

As described earlier, we can encode the spatial relations as an addition to a feature-pair to make it more distinctive. In addition, we also use some of the spatial relations as a basis for feature pairing. In this section, we present the three functions that we use to score feature-pairs in the process of choosing appropriate ones.

The scoring functions for feature-pairs are as follows:

- **scale score** λ_s is a function based on the scale relationship r_s (equation 4.1) between the pair elements.
- **spatial score** λ_d is a function based on the relative spatial distance r_d (equation 4.2) between the pair elements.
- **feature score** λ_f is a function based on the distance in feature space f_d (equation 4.5) between the pair elements.

Two of these scoring functions are tightly related to the spatial relations presented in Section 4.3.1. Each scoring function produces a score between 0 and 1, and the overall score for a pair λ_{pair} is the product of the three scores:

$$\lambda_{pair} = \lambda_s * \lambda_d * \lambda_f \tag{4.4}$$

It is clear that we want the two elements of a pair to be close in scale. If the points' scales are far apart, there is a chance that in another image of the same object but viewed from further away, the finer-scale point may be lost. Similarly, if we have a different image of the same object viewed from a closer distance, the larger-scale point may be lost. To prevent this and to increase the probability that both of the points survive, we score higher those pairs whose elements are closer in scale. The function for scoring the **scale relation** between the two elements of a pair is based on their relative scale r_s (equation 4.1). It is defined as shown in Figure 4.1, part (a). In our experiments, we set $r_{s1} = 0.1$ and $r_{s2} = 0.8$.

We want to avoid combining two points that are located at the opposite ends of an image since the probability that they belong to the same object in a cluttered scene is very low. On the other hand, we do not want these two points to be too close or completely overlapping either. If the encoded areas of the two points are partially or completely overlapping, then they encode the same part of the object and the distinctiveness of the feature-pair would be lessened. For the distance calculation, we use the Euclidean distance between the locations of the two points, and normalize it by the scale of the smaller point. The function for scoring the **spatial distance** between the two pair elements is based on their relative distance r_d (equation 4.2). It is defined as shown in Figure 4.1, part (b). In our experiments, we set $r_{d1} = 12$, $r_{d2} = 14$ and $r_{d3} = 30$.

The third criteria in scoring pairs is the distance in feature space between the

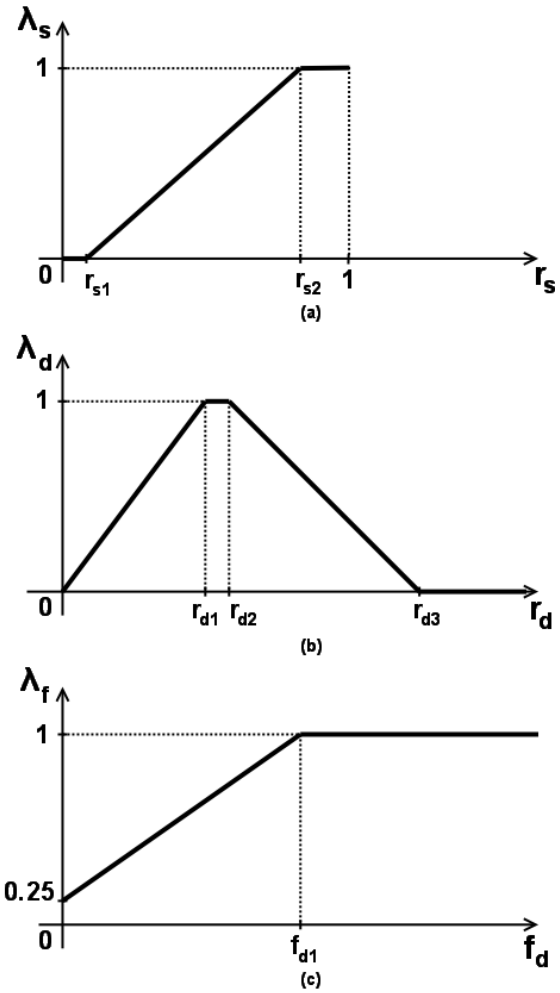


Figure 4.1: In this figure, we can see the pairing functions for scoring feature-pairs. The function for scoring the scale relation between the two elements of a pair (part (a)) is based on their relative scale r_s (equation 4.1). The function for scoring the spatial distance between the two pair elements (part (b)) is based on their relative distance r_d (equation 4.2). The function for scoring the feature distance between the two pair elements (part (c)) is based on Euclidean distance between the elements of the pair f_d (equation 4.5).

two pair elements. Since we want to avoid repetitive textures, we penalize features that are too close in feature space by giving them a lower score. In this case, the score is never zero since the pair containing two very similar feature points can also be useful. The feature-pair is scored with a higher score if the feature vectors of the two points are not similar. The function for scoring the **feature distance** between the two pair elements is based on the Euclidean distance between the elements of the pair:

$$f_d = |f_A - f_B|_2, \tag{4.5}$$

where f_d is the distance in feature space, and f_A and f_B are feature vectors of the two pair elements, respectively. It is defined as shown in Figure 4.1, part (c). In our experiments, we set $f_{d1} = 50$.

5 Querying and voting

The primary goal of indexing is rapid runtime recovery of the correct hypothesis. As can be seen in Figure 5.1, we query a complete image database with a query image to obtain nearest neighbours which, in return, vote for images that contain the most similar features to those in the query image. The images that receive the most votes are more likely to be the correct hypothesis.

Since a linear search for nearest neighbours is not possible in our case (due to the high dimensionality and number of features), we use an approximation to nearest-neighbour search, in which, for a given feature vector, a number of closest points according to the index structure are retrieved. A variety of index structures can be used for obtaining approximate nearest neighbours. We focus primarily on the kd-tree data structure, which is one of the standard choices for obtaining nearest neighbour approximation. We also conduct experiments with other structures, such as hierarchical k-means [11], [10], and a forest of different numbers of kd-trees, where multiple randomized kd-trees are created [12], [10]). We will discuss index structures further in Chapter 7, where we present our experiments and results. Throughout this chapter, we will use the kd-tree as a detailed example of an index structure.

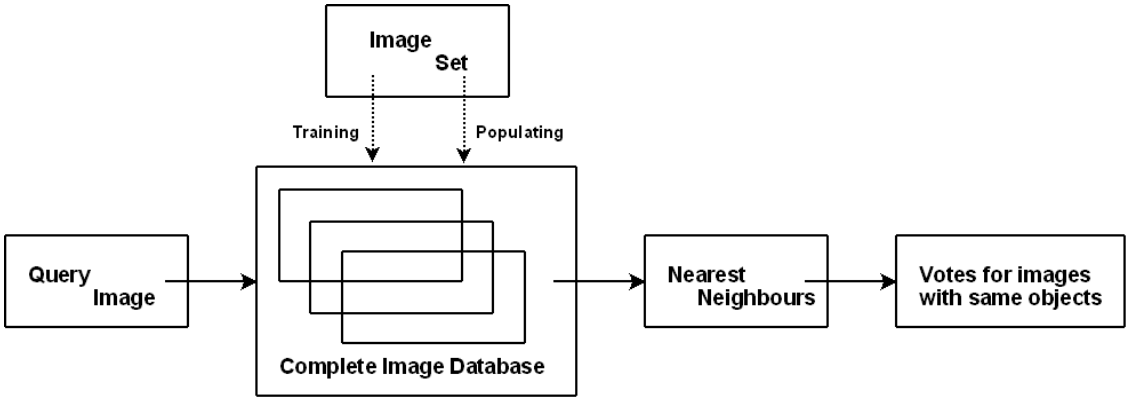


Figure 5.1: We query a complete image database with all the features contained in a query image to obtain nearest neighbours, which vote for images that contain the most similar features to those in the query image.

5.1 Building the index structure

Building the index structure in this context means to train a data structure using a set of N-dimensional points. By doing this, we make the data structure closely represent our specific point feature space. The goal of building the index structure is to allow for easier retrieval of closest points (in feature space) to the query point.

A kd-tree is a space-partitioning data structure that is used to organize points in k dimensions. It is built as follows: Starting with a complete set of N high-dimensional points, the space is split on the dimension d in which the data exhibits the greatest range (the difference between the largest and the smallest coordinate for this dimension is the largest). Then a split point p is chosen along the split dimension according to one of the following two rules: the cut is made at the mean value of the data in that dimension, or the cut is made with a sliding-split point rule, where the split is at the center unless that would leave all the points on one side; in that case, the split is shifted so that there is at least one point on each side. This

has the effect of generating trees with fewer long, thin bins. We have implemented and experimented with both methods and found that the latter produces better results. Next, an internal node is created to store d and p (the split dimension and the split value) and the process iterates with both halves of the data until a bin contains fewer than a small, fixed number of points. The leaves of the kd-tree form a complete partitioning of the data space, with the property that bins are smaller in higher-density regions and larger in lower density regions. This means that the nearest neighbour to any query should lie, with high probability, in the bin where the query falls or in an adjacent bin [2]. Unfortunately, the number of adjacent bins grows rapidly with dimensionality.

5.2 Populating the index structure

Populating the index structure in this context means to organize a complete set of N-dimensional points using the trained index structure. In such a way, organized points can be queried, and the closest points to the query point can be retrieved more efficiently.

The kd-tree stores the features from all images in the dataset. Populating the tree is done as follows: For each feature, we insert the feature at the end of the path down the tree based on the split information in the inner nodes. At the end of the populating process, we have the leaves of the tree containing the feature vectors of all the data set images. The leaves represent the bins that completely partition the high-dimensional data space.

5.3 Query

At runtime, the index is queried with features from query images in order to retrieve the correct hypothesis. After the features are extracted from the query image, they are used to query the database. Recall that the retrieval of true nearest neighbours from a database with millions of features in a high-dimensional space is very hard because we cannot use linear search. Recall, we use a Nearest Neighbour index structure such as a kd-tree. However, there are limits to Nearest Neighbour index structures in high dimensional spaces due to the phenomenon referred to as "the curse of dimensionality".

There are different index structures and methods that are appropriate for finding the approximate nearest neighbours. One of the more popular methods is to use a kd-tree with Best-Bin-First (BBF) [1] retrieval. The BBF search algorithm provides fast access to the closest stored neighbours (that are the most similar in feature space). The main idea behind the BBF search is to search first the bins closest in Euclidean distance to the query point rather than those closest in the kd-tree branch structure (which is the method used in the standard kd-tree search).

5.4 Voting

The goal of voting is to find which images from the database contain the most similar features to the features in the query image. After finding the nearest neighbours to the query point with one of the approximation methods, votes are cast by each obtained neighbour feature for the image where that feature belongs. The images with the highest numbers of votes are the ones hypothesized to contain the same objects as the query image.

We examined and implemented several different voting methods. Each query

feature can vote for all images that contain the features retrieved by nearest neighbour search. Alternatively, we can disallow an image to receive multiple votes from a single query point. The final option we looked at is to weight the votes inversely to the Euclidean distance between the query feature and the features obtained by the nearest neighbour search, or according to the rank of the features obtained by the nearest neighbour search. To account for dense areas of the tree (which are less important since the points in these areas are more ambiguous due to their similarity), we might normalize the weight by the density of the tree space considered by the nearest neighbour search. We conducted some experiments with these methods, but in our results in Chapter 7, we use a very simple method in which each query point votes for each image containing the nearest neighbours for that query point. Given the voting scheme, we calculate the score for the query image. Our scoring method is further discussed in Chapter 7.

6 Factors in system performance

In this chapter, we discuss the three main factors that influence the recognition performance of singleton and pair-based systems: survivability, distinctiveness and the distribution of noise matches. Survivability tells us if a feature will appear in a different image of the same object and distinctiveness tells us if we can tell a feature apart from all the other features in the database. Survivability and distinctiveness are extremely important, even to the point that if a feature fails on either one of them, then it will not be useful. If we want pairs to perform better than singletons then they will need to win in both distinctiveness and survivability or have one of them so much better that it compensates for the other one. Survivability and distinctiveness are properties of a single feature that tell us something about one feature at a time. However, we need to consider how features interact when they come together in the voting process. That is where our third factor, the noise distribution, can have a significant impact. We discuss these three factors further in Sections 6.1, 6.2 and 6.4.

6.1 Survivability

Survivability of a feature is its property that tells us what is the probability that that feature will also exist in a different image of the same object. Survivability of features across two images that contain the same object is extremely important

since if the feature does not appear in a different image of the same object, we cannot use it for recognition. Having one feature in an image, for us to find a match in the other image it is absolutely crucial to have a surviving feature in that other image. Since the pairs of features are built from two singletons, there are more ways in which their survivability can fail. This is illustrated in Figure 6.1. Naturally, we expect the pairs to have lower survivability because to detect the same pair in two images, both elements of that pair have to appear in both images and we have to choose consistent pairs (i.e., pair the same singletons) in the process of pair creation.

6.2 Distinctiveness

A distinctive feature is a feature for which we can tell the matching one apart from almost all other features in the database. The distinctiveness of a features is a property that makes it possible to find its good matches without getting too many noise matches. For distinctiveness to be high, we want both high precision (so that good matches are not outnumbered by noise matches) and good enough recall so that we get enough good matches to build a spike in voting for the matching image.

Another source of distinctiveness is the clustering of the feature space. Namely, there is evidence that pairs are more spread in feature space than single features. In other words, pairs can have data that is reasonably spread-out in all dimensions. Singletons might have 30 dimensions but in the last (high-order) 10 there might be much less variance. That is, in some directions, the points are squeezed closer together than they are in pairs.

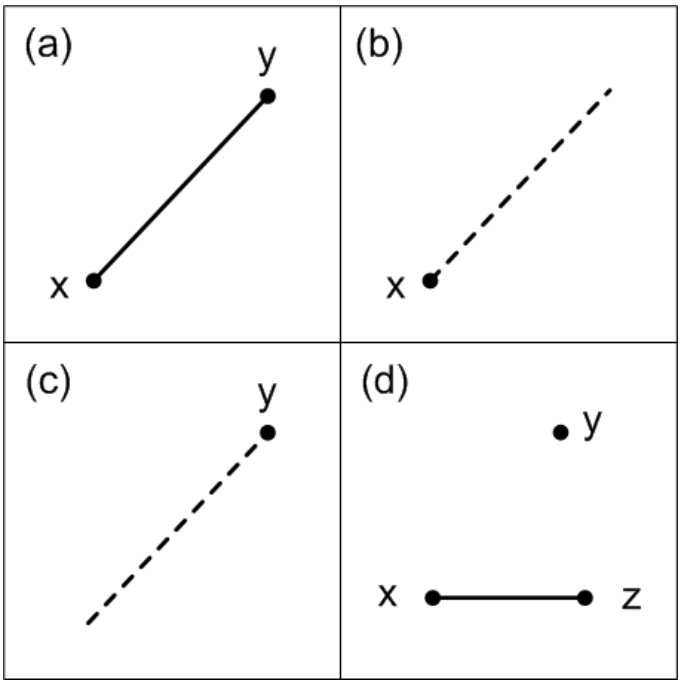


Figure 6.1: Pair survivability: A pair, as seen in part (a), can fail to survive in another image in three different ways: either one of its endpoints can fail to survive (parts (b) and (c)) or both its endpoints can survive but they may not be paired with each other (part (d)).

6.3 Effective precision

Effective precision of feature points is their precision (as an indicator of distinctiveness on a point-by-point basis) combined with their survivability. It is defined as follows:

$$ep = p * s, \tag{6.1}$$

where ep is effective precision, p is precision and s survivability. Effective precision, that is based on the whole image, takes into account the noise introduced by query features that have no match (we considered so far only the ones that have a match). Since distinctiveness and survivability act together in the retrieval process, we have to consider them together when evaluating our features in order to be able to obtain a prediction on the retrieval trends for singletons and pairs. Effective precision gives us insight into the expected relative performance of singletons and pair-features.

6.4 Distribution of noise matches

Distinctiveness and survivability do not account for the possible correlation between features, i.e., how the votes from all the query features interact when they act together on all database features in the voting process. This is where the noise distribution can have a significant impact. We did not look deeply into the distribution of noise matches but we will try to explain what role it can play in the retrieval process.

If we have one query point that retrieves a number of nearest neighbours in our retrieval process, we want to know which of the neighbours is a correct match for the query point (i.e., signal). However, we can also ask where do votes cast by remaining retrieved points go - in the voting process, this is considered to be

noise. We know that these noise matches get scattered in some way. If they are spread uniformly across images, this generally would not harm significantly the retrieval results. However, we expect these votes to be clustered because of repetitive textures in some images and that can increase the number of votes for non-matching images.

7 Experiments and results

In this chapter, we describe the setup for our experiments, and present and explain the obtained results. We present results in groups according to which factor in system performance they explain.

7.1 System set-up

Our system retrieves singletons and pair-features in exactly the same way. We use the same system and the same set of parameters for these two different types of features so that we can fairly compare the results.

The image set that we are using comes from the Center of Visualization and Virtual Environment of the University of Kentucky [1]. It consists of 10,200 images grouped into sets of 4 images of the same object taken from different points of view, from different distances or with different lighting. An example of four images that belong to the same group can be seen in Figure 7.1.

We have done experiments with several different index structures: we use a kd-tree, hierarchical k-means [11], and a forest of kd-trees composed of different numbers of randomized kd-trees [12]. We use the implementations of these index structures from the FLANN library [10]. We tested kd-tree forests, composed of 5 and 10 trees. For a single kd-tree, as well as for the 5− and 10-tree forests, the

[1] Available at http://vis.uky.edu/~stewe/ukbench/

Figure 7.1: Group of 4 images from the image set.

number of features per leaf is set to 1. For the hierarchical k-means index structure, the branching factor is set to 10.

7.2 Ground truth

We collect ground truth information in order to better understand the details of all factors involved in image retrieval, to be able to analyze distinctiveness and survivability of image features, and to measure these factors. Our ground truth image set consists of 25 image-pairs: each pair contains two images of the same object taken from a different point of view and with different lighting. For each point (singleton or pair) in each image-pair in our ground truth set, we find the one-to-one correspondences between the matching points. In other words, for each image-pair, we know which points in the first image appear in the second image and what the corresponding points are, if they exist. To achieve this, we use epipolar geometry. We manually locate a subset of matching points to fix the epipolar geometry, and then we use epipolar lines to constrain the search for point correspondences between the two images. The points that we consider to match have to fit the epipolar line in both directions (i.e., they must be close to the line), and they have to be reasonably close in feature space. The correspondences that we find in this way build our ground truth feature set.

7.3 Results

In this section, we present our results in more detail. For survivability and distinctiveness tests, we use the 25 ground truth images for querying the index, while the index itself contains the points for all $10,200$ images from our image set, as is presented in Figure 7.2. We retrieve enough $(100,000)$ nearest neighbours to

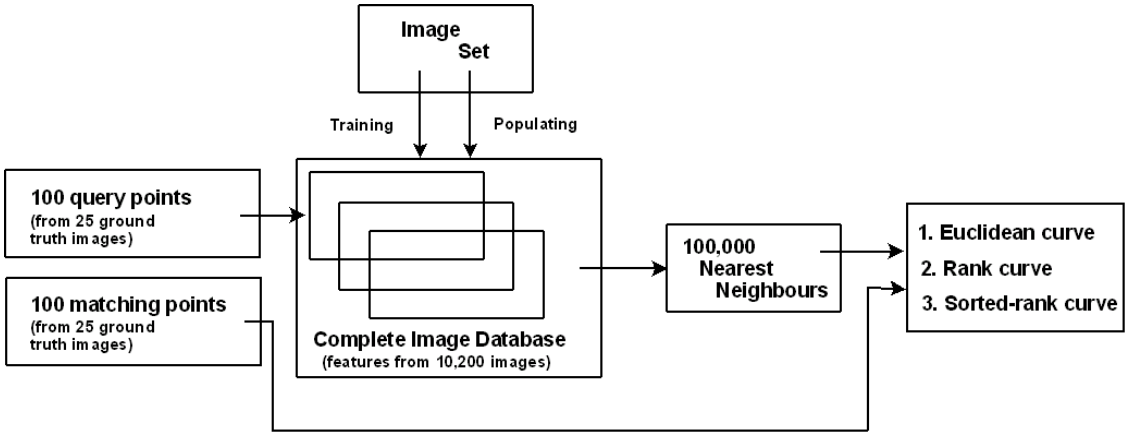


Figure 7.2: In survivability and distinctiveness tests, we query the database with 100 random points (from our 25 ground truth images) for which we know correspondences, while the database itself contains features from all $10,200$ images from the original image set. We retrieve $100,000$ nearest neighbours so that we get all the closest points in feature space regardless of index structure in order to create accurate precision-recall curves.

be confident that we get all the closest points in feature space in order to create accurate precision-recall curves.

7.3.1 Survivability results

Recall that the survivability of a feature is the probability that the feature will also exist on a different image of the same object. As one might expect, our experiments show that singletons tend to have better survivability. In our ground truth set, the average fraction of singletons that survive across our image pairs is 0.53. For a singleton feature to survive across two images, it is simply required to find that same feature in both images. In contrast, to detect the same feature-pair in two images, both singleton elements of that pair have to appear in both images and

they have to be paired the same in these images. When we take into consideration only the constraint that both singleton elements of a pair have to appear in both images, we find that in our ground truth set, the average fraction of pairs that survive is 0.33. However, when we also require that the two surviving elements of a pair are paired the same in both images, we find that (in our ground truth set) the average fraction of pairs that survive is only 0.10. We can see that the survivability of pairs is hurt more by the consistent pairing condition than by the survivability of its elements.

Consistent pairing is directly related to our scoring function since the scoring function is what tells us which singletons to pair and which one of those pairs to consider the best. Even though we tried to account for changes in scale, illumination and viewpoint with the design of our scoring function, inconsistent pair selection remains a serious challenge. A possible improvement to survivability of feature-pairs that we investigated is to keep more than the best pair. To ensure that we keep the database the same size, we do this only for the query images. Figure 7.3 shows the improvement in survivability that is achieved by keeping a larger number of pairs after scoring them. The horizontal line indicates the best possible survivability (0.3348, when we consider only the constraint that both singleton elements of the pair have to survive). The bars in the figure approach this ideal as the number of kept pairs grows.

7.3.2 Distinctiveness results

Recall that if q is the query feature and m is its matching feature in the database, then a distinctive feature is a feature for which, given q, we can tell m apart from almost all other features in the database. The distinctiveness is about getting a lot of good matches without getting too many noise matches. Our results demonstrate

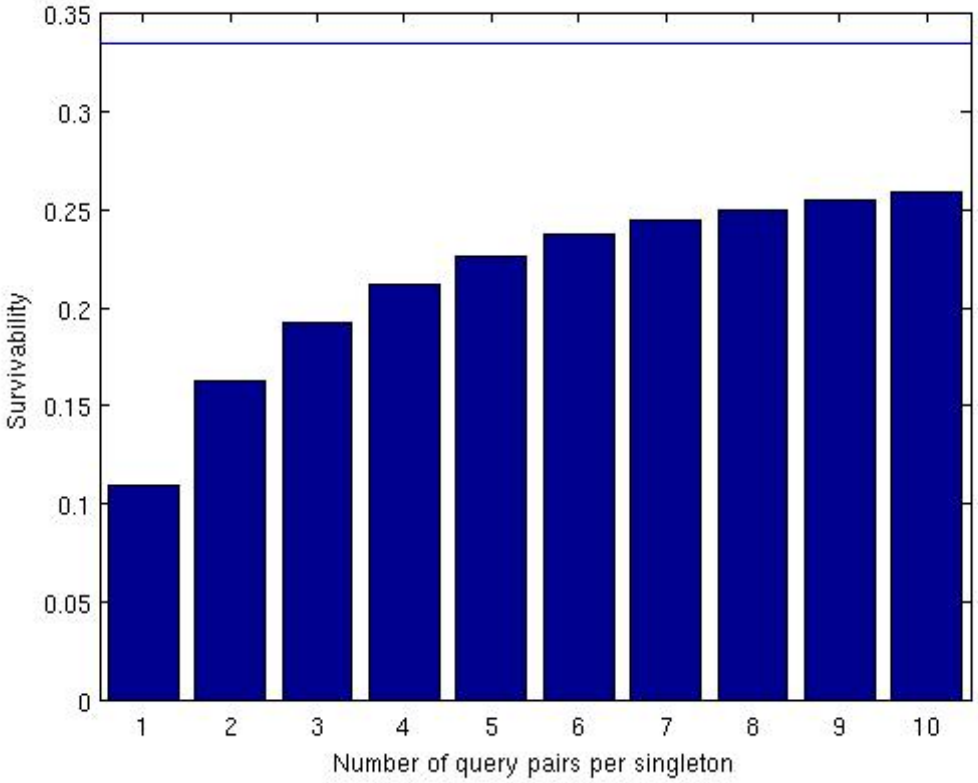


Figure 7.3: The survivability of pairs increases by increasing the number of pairs that we keep after we score them. The horizontal line in the figure indicates the best possible survivability (0.3348, when we consider only the constraint that both singleton elements of the pair have to survive). The bars in the figure approach this ideal as the number of kept pairs grows.

that if m exists, the probability of finding it in the retrieval process is much larger for pairs than for singletons. Considering the distinctiveness of feature-pairs, their potential benefit in large databases and in the presence of the occlusion and clutter in terms of retrieval is clear. While many similar singleton features could appear in many different images and mislead the search, this is less likely to happen with feature-pairs.

To measure the distinctiveness of image features, we use precision-recall curves for our ground truth image set. Precision gives us the fraction of the number of good matches retrieved to the total number of matches retrieved. Recall tells us what fraction of all possible correct matches do we manage to retrieve, i.e., the total number of relevant points retrieved divided by the total number of relevant points that could be retrieved. To produce precision-recall curves, we randomly choose 100 query points from our ground truth image pairs, and make sure to choose points for which correspondence was found. We query the index structure with these points and retrieve a set of enough closest neighbours (we use $100,000$) to be sure to get all the closest ones and to counteract possible differences in nearest neighbour structures so that we can show accurate precision-recall curves. The precision-recall curves are drawn for a range of Euclidean distances and for all ranks for the retrieved neighbours.

In the following set of results, we use the three types of precision-recall curves that we define. These curves reflect different retrieval strategies:

- **Euclidean curve** is a precision-recall curve that we obtain if we retrieve all the points within some specific Euclidean distance. For this curve, the neighbours are ordered by the Euclidean distance from the query point.

- **Rank curve** is a precision-recall curve that we get if we retrieve the first N neighbours from the index structure. For this curve, the neighbours are

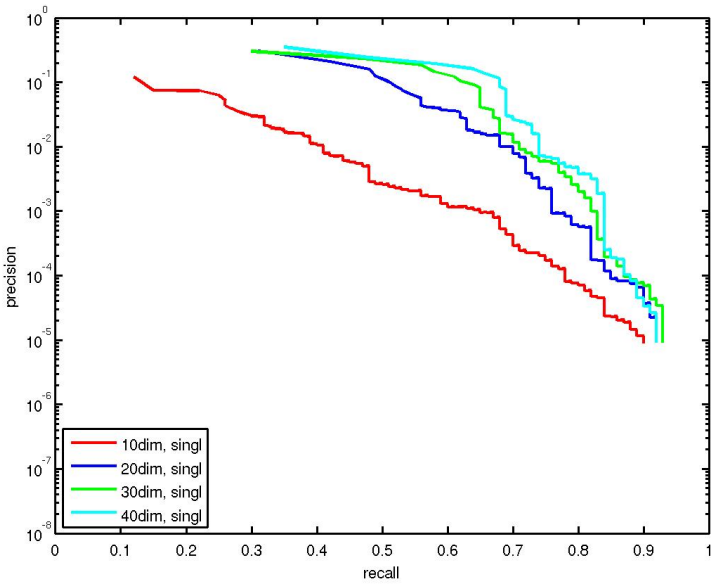
ordered as they were retrieved by the index structure.

- **Sorted-rank curve** is a precision-recall curve that we get if we retrieve the first N neighbours in terms of Euclidean distance. The neighbours for this curve are ordered by first retrieving the points by the index structure and then sorting them by Euclidean distance from the query point.

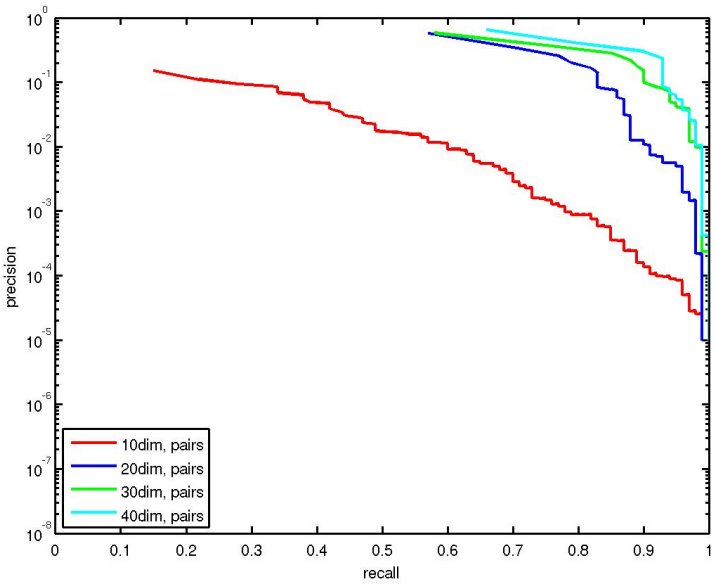
The sorted-rank curve reflects the best strategy for image retrieval and therefore we use this curve to show most of our results.

First we present the set of results where we explore the distinctiveness of singletons and pairs across different feature dimensionalities $nDim$. We use the following dimensions for the feature vectors: 10, 20, 30 and 40. In these graphs, we use the sorted-rank precision-recall curve. We adopt this specific curve because we use sorted-rank to retrieve the points for actual image retrieval. Figure 7.4 shows the precision-recall curves for the above range of dimensionalities for singletons, part (a), and for pairs, part (b). From these plots, we can see that both singleton and pair-features of 30 dimensions are the most appropriate in terms of results. The 10-dimensional features do not contain enough of the necessary information for retrieval, while the improvement of 40-dimensional features over 30-dimensional features is too negligible to justify the dimensionality increase. Therefore, we use 30-dimensional features in the remaining plots in this report.

In the following experiment, we show the performance of the different index structures for singletons and pairs. This time we use 30-dimensional features and the rank precision-recall curve. We choose the rank curve because this curve (out of the three that we use) is the only one that depends on the choice of index structure. If we use a curve that is obtained by retrieving points ordered by Euclidean distance and taking into account that we use $100,000$ nearest neighbours, the performance

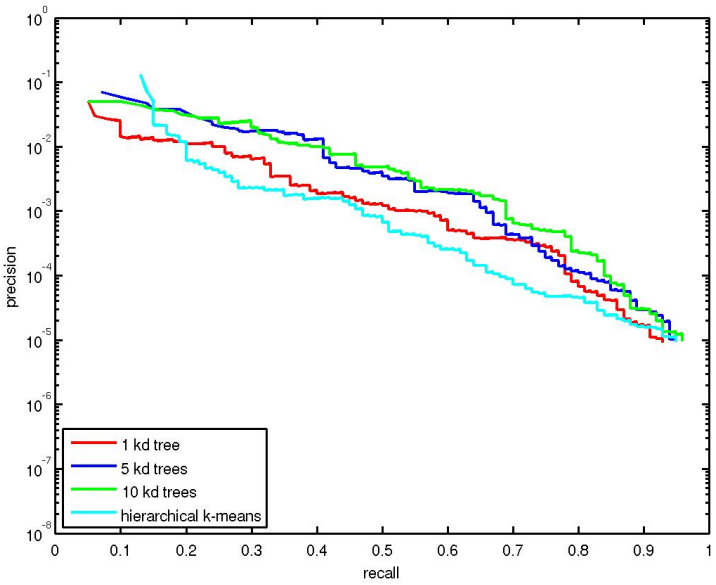


(a) Singletons

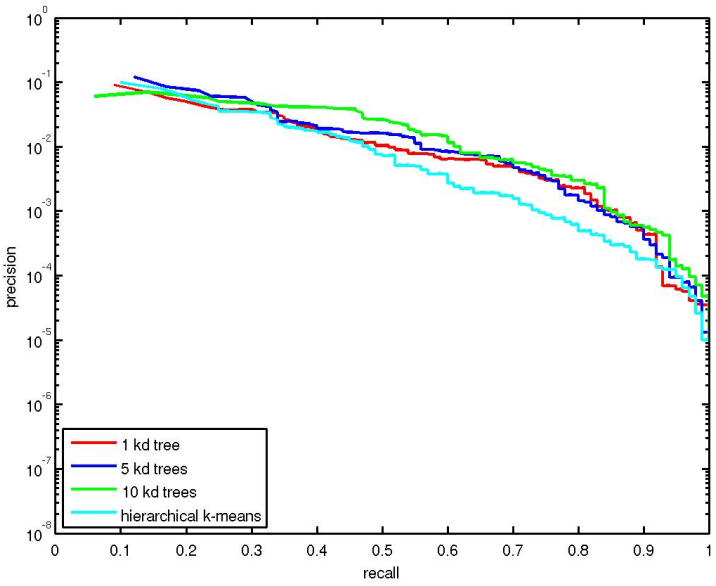


(b) Pairs

Figure 7.4: Sorted-rank curve for different feature dimensionalities for singletons (a) and pairs (b).



(a) Singletons



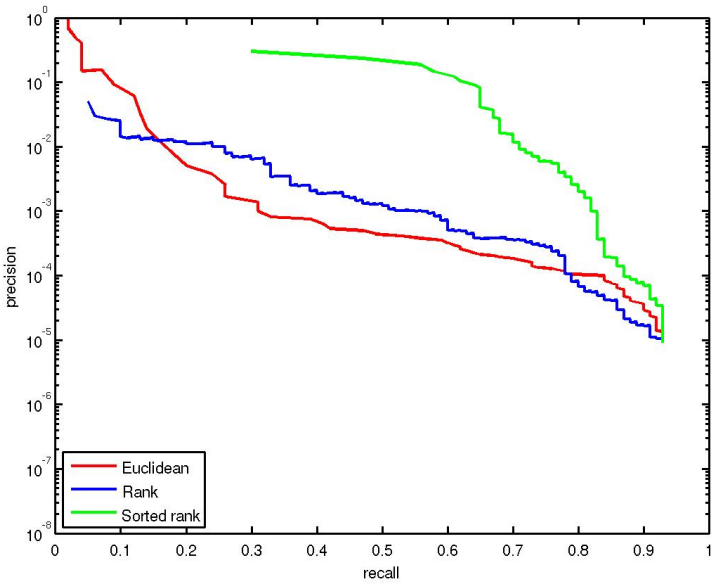
(b) Pairs

Figure 7.5: Rank curve for different index structures for 30-dimensional singletons (a) and pairs (b).

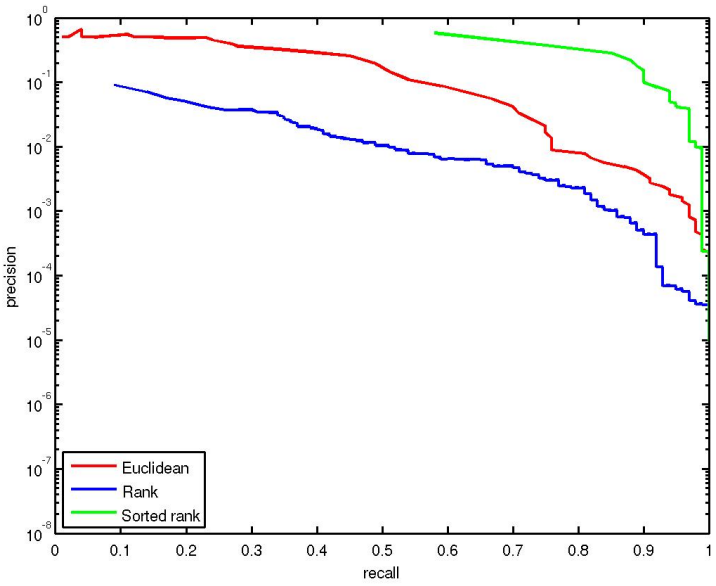
of all the index structures would look the same. Figure 7.5 [2] shows the results for singletons, part (a), and pairs, part (b), for the following index structures: the kd-tree, multiple randomized kd-trees (we use 5 and 10 kd-trees), and hierarchical k-means. We find that for both singletons and pairs, the forest of 10 kd-trees gives the best results in terms of retrieval. However, there is only a slight improvement over a single kd-tree when a forest of 10 or 5 kd-trees is used. Taking into account the overhead of having multiple kd-trees in memory, we consider the single kd-tree a more appropriate index structure. In the following plots, we show the results for a single kd-tree index structure.

Figure 7.6 shows the three types of precision-recall curves for singletons, part (a), and feature-pairs, part (b). In both plots, we use 30-dimensional features and a single kd-tree as the index structure. In these plots, we can see that the Euclidean curve is much worse for singletons than for pairs, relative to the rank and sorted-rank curves. This effect could be explained by increased clustering in the singletons' feature space (versus the pairs' feature space). When we retrieve points by the Euclidean distance rule, we get all the points within a certain diameter. For singletons, in some more dense regions, this method can retrieve many more points than in less dense regions. However, for pairs, this does not have the same effect since the pairs feature space is more evenly spread out. As additional support for this, we can look into the Euclidean and rank curves for both singletons and for pairs: for singletons, the rank curve does better than the Euclidean curve, while for pairs, the Euclidean curve is better then the rank curve. However, for both singletons and pairs, the sorted-rank curve outperforms the other two, since in this case we retrieve points by rank and then sort them by Euclidean distances from the query point.

[2]Note that Figure 7.4 shows sorted-rank curves and Figure 7.5 shows rank curve. These two types of curves are very different and should not be compared to each other in these figures.



(a) Singletons



(b) Pairs

Figure 7.6: Euclidean, rank and sorted-rank curves for 30-dimensional singletons (a) and pairs (b). Here we use a single kd-tree.

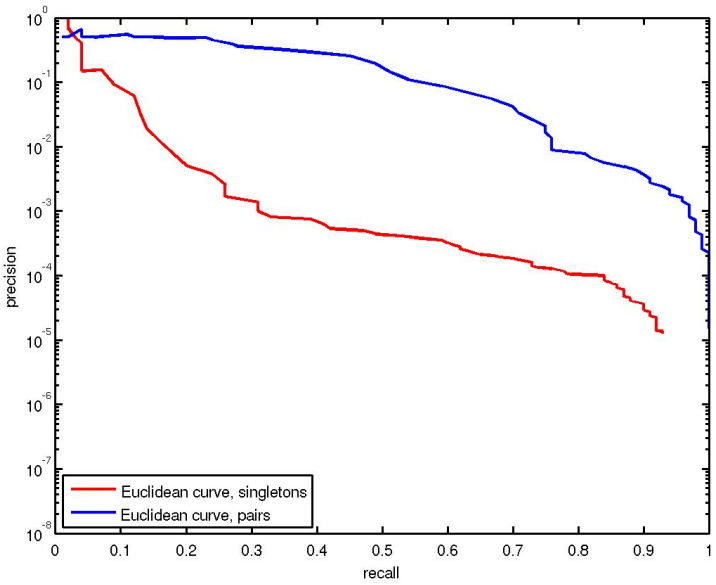


Figure 7.7: Euclidean curves for singletons and pairs (1 kd-tree, 30 dimensions).

In the next three figures, we show each of the three types of curves. On each plot, we display results for singletons and for pairs for comparison between these two types of features. We use 30-dimensional features and a single kd-tree as the index structure. Figure 7.7 shows the Euclidean curves for singletons and for pairs. We can see that the gap between the curve for singletons and the curve for pairs is quite large, i.e., in Euclidean space, pairs are much more distinctive than singletons. Figure 7.8 shows the rank curves for singletons and for pairs. The gap between the distinctiveness of singletons and pairs narrows because (as already mentioned) the pairs feature space is more evenly spread than the singletons feature space, and this method of retrieval helps overcome that problem with singletons. Namely, if we retrieve unevenly spread points within some Euclidean distance, we will get different number of points depending on space density, while retrieving according

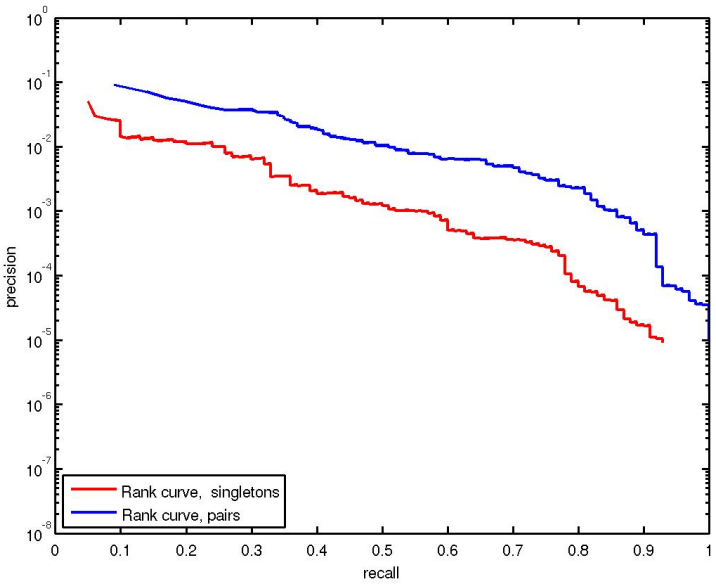


Figure 7.8: Rank curves for singletons and pairs (1 kd-tree, 30 dimensions).

to bin distance will always give us the same number of points because bins contain the same, fixed number of points. Singletons are helped more than pairs since they are affected by this problem to a larger degree. In the third figure (Figure 7.9), we see a much better performance for both singletons and pairs. It is useful to compare the portion of the curves that are likely to yield the best retrieval results (i.e., before the dramatic drop-off in precision). We do not want to compare these curves at a point where precision for singletons falls to a useless level. We can see that even though the gap is narrower between the singletons and pairs, it still exists. However, the size of the gap at this point has to compensate for the lower survivability of pairs in order for pairs to win in overall image retrieval (where survivability and distinctiveness come together). We present the image retrieval results, where survivability and distinctiveness factors come together, in Section

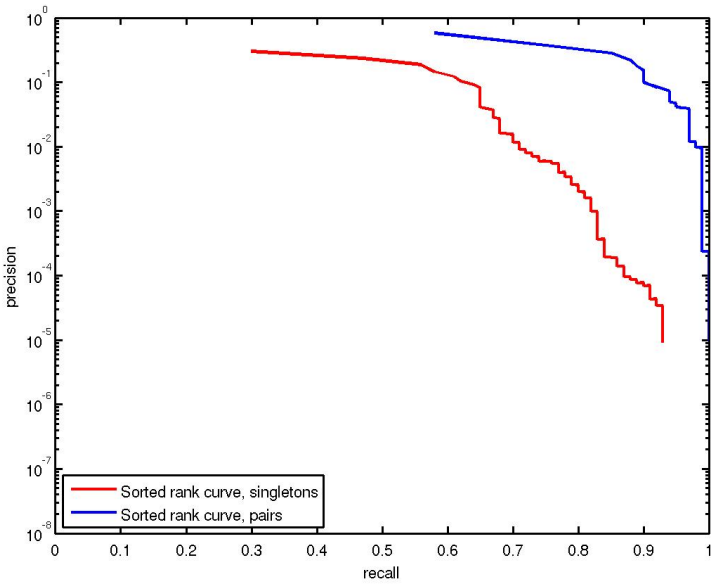
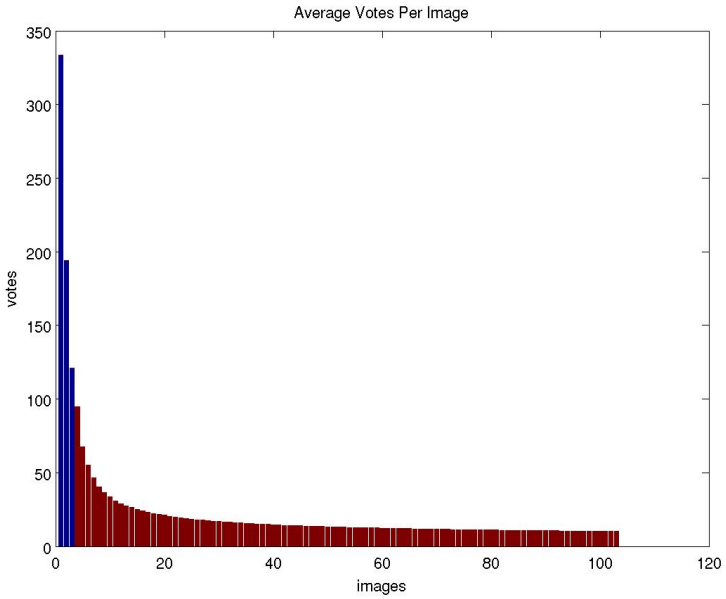


Figure 7.9: Sorted-rank curves for singletons and pairs (1 kd-tree, 30 dimensions).

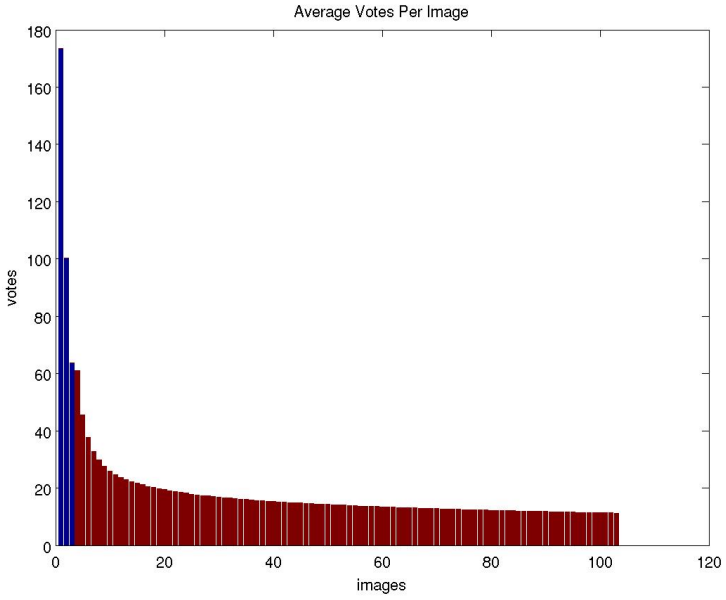
7.3.5.

7.3.3 Effective precision results

Recall that effective precision, as defined in Equation 6.1, takes into consideration precision and survivability of our features. For these results, we used the same settings as for the final image retrieval results in order to be able to compare the predicted behaviour and the obtained values for both singletons and pairs. The predicted values on a point-by-point basis that we obtain by using this method are shown in the second two rows of Table 7.1. We can see from the values in the last two rows that our prediction is that pairs will fall a bit behind singletons for all settings. From the values in the first two rows (average precision for image retrieval), we can confirm that this actually happens in image retrieval.



(a) Singletons



(b) Pairs

Figure 7.10: The distribution of noise matches for singletons (a) and pairs (b). We show the average number of votes for the correct three images in blue and the average number of votes for the first 100 incorrect images in red.

7.3.4 The distribution of noise matches results

In this section, we present the experiments that we did to better understand the distribution of noise matches. Recall that we define the distribution of noise matches in Section 6.4 in terms of signal-to-noise ratio. If we have one query point that retrieves a number of nearest neighbours in our retrieval process and one of those retrieved points is a correct match (i.e., signal), we can ask ourselves where do votes cast by the remaining retrieved points go. In the voting process, these votes are considered noise. We show our results in Figure 7.10 (for singletons (a) and pairs (b)) where we show the average number of votes for the correct three images in blue and the average number of votes for the first 100 incorrect images in red. For these experiments, we first retrieve 1000 points from the index structure (FLANN parameter *checks* - see Section 7.3.5 for further details), sort them by Euclidean distance from the query point, and keep only the 10 closest ones (FLANN parameter *nearest neighbours* - see Section 7.3.5 for further details). It is clear that the correct images usually get the most votes. However, the first few incorrect images, due to noise matches, receive almost as many votes on average as the lowest-voted correct image. Images that receive many noise matches likely contain repetitive textures that also appear in the query image.

7.3.5 Image retrieval results

In this section, we present our image retrieval results where our three factors of system performance (survivability, retrieval, and the distribution of noise matches) come together. For these experiments, we queried using our whole image set and we query with all feature points from each query image. Recall that our image set contains images in groups of four that belong together where each of four images

in a group contains the same objects. We query with one image from each set of four images and the database contains all $10,200$ images from the original image set. The scoring method that we use is tightly related to our image set. For score calculation, we focus on the three matching images from the same group as the query image. We calculate the score by sorting the votes and by looking at the rank of the three matching images. From the rank of matching images, we obtain the precision p_{ik} of the k^{th} good match for the i^{th} query image as given by the following formula:

$$p_{ik} = \frac{k}{r_{ik}}, \tag{7.1}$$

where $k = 1, 2, 3$ stands for the first, second and third matching images and r_{ik} (i.e., r_{i1}, r_{i2}, r_{i3}) are the ranks for the first, second and third matching images for the i^{th} query image. We calculate the average precision α_k of the three images ($k = 1, 2, 3$) across all the query images $q \in Q$:

$$\alpha_k = \sum_{i=1}^{|Q|} \frac{p_{ik}}{|Q|}, \tag{7.2}$$

Finally, the overall average precision α (the average of the three: α_1, α_2 and α_3) which is used to evaluate our method is given by:

$$\alpha = \frac{\alpha_1 + \alpha_2 + \alpha_3}{3}. \tag{7.3}$$

Table 7.1 shows, in the first two rows, the average precision for singletons and pairs (α_s and α_p) obtained by using the FLANN library [10]. We use the kd-tree as the index structure (a single kd-tree) and we use 30-dimensional features for both singletons and pairs. The values that are shown in the last two rows are predicted effective precision for singletons (β_s) and pairs (β_p) on a point-by point

Table 7.1: Average precision for image retrieval for singletons α_s and pairs α_p (first two rows), and predicted effective precision on a point-by-point basis (second two rows) for both singletons β_s and pairs β_p (with varying parameters for number of checks (ch) and nearest neighbours (nn)).

	ch:1000 nn:1000	ch:1000 nn:100	ch:1000 nn:10	ch:1000 nn:1
α_s	**0.1338**	**0.5371**	**0.7044**	**0.6836**
α_p	**0.0691**	**0.3462**	**0.5926**	**0.6180**
β_s $(\times 10^{-3})$	**0.325**	**3.4**	**29.3**	**149.3**
β_p $(\times 10^{-3})$	**0.0899**	**0.899**	**8.99**	**33.9**

basis (we obtain it as a product of precision for certain settings and survivability, see Equation 6.1). The two parameters that we vary and show results for are:

- **ch** - checks - the number of points to retrieve from the index structure,
- **nn** - nearest neighbours - the number of nearest neighbours (nn) to keep after the sorting of all retrieved points.

We can see that as we keep the number of checks the same and as we decrease the number of nearest neighbours, the results improve. This happens because by having more nearest neighbours we introduce more noise and the harm from the introduced noise is larger than the improvement obtained by retrieving more features. Further, we notice that the average precision for singletons (α_s) beats the average precision for pairs (α_p) for all settings. This happens due to the fact that here we have both survivability and distinctiveness working together and they balance in such a way that the lower survivability of pairs harms pairs more than their advantage in distinctiveness can help them. We can also see that the gap

between the average precision obtained for the same parameters for singletons and for pairs gets smaller as the number of nearest neighbours is reduced. The reason is that the pairs are more distinctive and we can find the right match by looking at only a few nearest neighbours. An additional factor to keep in mind is the distribution of noise matches that we discuss in Section 7.3.4.

If we look at the last two rows of the Table 7.1, we can see how well effective precision (β_s and β_p) predicts results for average precision (α_s and α_p) for image retrieval. The prediction is consistent for singletons and for pairs, i.e., the values are increasing together, except for the last two values for α_s and β_s, where we decrease the number of nearest neighbours to 1. The prediction (β_s) here is that the average precision (α_s) will improve when we decrease the number of nearest neighbours (nn) to 1, whereas the actual value for α_s drops. This could be explained by the limitation of our effective precision calculation. Namely, for this last calculation, we cannot vote for all three matching images from the same set (since we have only 1 nearest neighbour). If we calculated the precision using all three matching images, we expect that the value for β_s would have dropped too.

8 Conclusions

To deal with increasing database sizes and with clutter and occlusion in images, inspired by recognition systems of the 1970's and 80's, we have designed a method to create pairs of local image features without increasing their dimensionality relative to singletons. It is not a surprise to us that these pairs of features are less survivable than singletons: we find that the survivability of singletons is about 53% and the survivability of pairs is about 10%. However, it is very interesting that our feature-pairs are more distinctive, even for the same dimensionality as singletons. For instance, at 0.1 precision, pairs have about 95% recall while singletons only have 65% recall (Figure 7.9). This clearly shows that there is a potential that pairs can win over singletons, even with the same dimensionality, in image retrieval.

Since the space of possibilities for running our experiments is quite large, we have not explored it completely. There are many different ways of choosing pairs, many different ways of querying and voting, and many settings that can be tweaked. Therefore, at this point we cannot conclude that a combination of parameters or a setting where pairs can win over singletons does not exist.

9 Future work

Since the space of possibilities for running our experiments is large, the future work is to explore these various options. There are small scale changes, such as parameter tweaking, but also larger possible changes. We can classify these larger options according to the following categories:

- Choosing pairs
- Querying
- Voting
- Distinctiveness

Recall that we have found that the survivability of pairs is especially harmed by the fact that we do not choose the same pairs in two different images of a same object. One of the possible solutions to this problem is to improve our functions for scoring pairs. Currently, in our method, we form a number of pairs for each point, score them and then choose the best one. One of the options is to keep the best n (where n is a number as 5 or 10) pairs in the query image to increase the possibility of having the same pair in two different images. Another possibility is to introduce a rule that would prevent pairing across the object boundaries (this could be achieved by detecting edges and using them as an indicator that two points are or are not on the same object). Improving survivability of pairs can also be

accomplished by changing the pairing rules in a way that would help make sure that when one element of a pair survives, the other element of that same pair survives as well. To achieve this, we could perhaps use part of the information returned by the feature detector.

In our query process, currently we query the database with all features from a query image. It is clear that singletons would win if there is enough of them. However, as the number of query features in a query image decreases, we expect the less ambiguous pairs to win in retrieval. Another way of improving the querying is by taking out (i.e., not voting with) points that are likely to be bad or unhelpful, assuming we can identify them. We could try to identify such points by using some of the characteristics of the feature itself.

We have already discussed some of the voting techniques. In our final results, we use a simple voting scheme where we vote with each retrieved nearest neighbour for the image where that point is coming from. Alternatively, we can disallow an image to receive multiple votes from a single query point. Another option is to weight the votes according to the Euclidean distance of the query feature to the features obtained by the nearest neighbour search or according to the rank of the features obtained by the nearest neighbour search. To account for dense areas of the tree (which are less important since the points in these areas are more ambiguous due to their similarity), we might normalize the weight by the density of the tree space considered by the nearest neighbour search.

Besides trying to improve the survivability of pairs, we can further improve the distinctiveness of pairs to the point that they overtake the performance of single features even without improving their survivability. We investigated improving distinctiveness of pairs by adding spatial relations between the elements of a pair. However, after performing some initial tests, we did not find this method promising.

Perhaps the method of adding spatial relations could be improved as well.

Bibliography

[1] Jeffrey S. Beis and David G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *CVPR '97: Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, page 1000, Washington, DC, USA, 1997. IEEE Computer Society.

[2] Jeffrey S. Beis and David G. Lowe. Indexing without invariants in 3d object recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(10):1000–1015, 1999.

[3] S. Belongie, J. Malik, and J. Puzicha. Shape matching and object recognition using shape contexts. *PAMI*, 24(4):509–522, April 2002.

[4] G. Carneiro and A. Jepson. Flexible spatial configuration of local image features. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(12):2089–2104, 2007.

[5] M.S. Costa and L.G. Shapiro. Scene analysis using appearance-based models and relational indexing. *iscv*, 00:103, 1995.

[6] Benoit Huet and Edwin R. Hancock. Relational histograms for shape indexing. In *ICCV '98: Proceedings of the Sixth International Conference on Computer Vision*, page 563, Washington, DC, USA, 1998. IEEE Computer Society.

[7] Y. Lamdan, J.T. Schwartz, and H.J. Wolfson. Affine invariant model-based object recognition. *RA*, 6:578–589, 1990.

[8] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, 2004.

[9] J. Matas, O. Chum, U. Martin, and T. Pajdla. Robust wide baseline stereo from maximally stable extremal regions. In *Proceedings of the British Machine Vision Conference*, volume 1, pages 384–393, London, 2002.

[10] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *International Conference on Computer Vision Theory and Applications*, 2009.

[11] D. Nistér and H. Stewénius. Scalable recognition with a vocabulary tree. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 2, pages 2161–2168, June 2006.

[12] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8, June 2008.

[13] Fridtjof Stein and Gérard G. Medioni. Structural indexing: Efficient 2d object recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 14(12):1198–1204, 1992.