Fast matching of SIFT features

Avdullaj Mario (1184284), Mazzetto Alessio (1177943)
Department of Information Engineering, University of Padova – Via Gradenigo, 6/b, 35131 Padova, Italy
Email: {alessio.mazzetto.1, mario.avdullaj}@studenti.unipd.it
Course: 3D Augmented Reality - Report A.1

***Abstract*—Local feature matching is an important issue which has many applications in computer vision. If the dataset is small, linear search can be employed for this task. However the matching can become computationally very expensive as the number of features grows. Thus, in recent years, a wide range of approximate solutions have been proposed to speed up the matching of features in large datasets. In this report, we will discuss these methods and experimentally test their accuracy, focusing on SIFT features.**

I. Introduction

In recent years, local image features have been extensively studied and employed in many computer vision tasks, as image classification and retrieval, stereo correspondence, object recognition and 3D scene reconstruction. Simply put, features are defined as "interesting" parts of an image. The basic approach used to extract features from an image is the following. First, salient points of the image, also called *keypoints*, are detected, then for each of those points a discriminative descriptor of its surroundings is constructed. It is supposed that these descriptors belong to a metric space, so that the distance between two descriptors can be calculated. An image feature is the pair keypoint and associated descriptor. As these features are used to find correspondences between images, they need to be invariant with respect to image scaling and rotation, and partially invariant to change in illumination and 3D camera viewpoint. Many different kind of feature extraction algorithms have been proposed, as BRIEF, ORB, BRISK, SURF and SIFT. In this report, we will focus on SIFT [2] features.

The easiest way to find matches between features of two different images $\mathcal{X}$ and $\mathcal{Y}$ is to compute, for each feature descriptor of $\mathcal{X}$, the nearest feature descriptor of $\mathcal{Y}$. In other words, we are finding for each feature in $\mathcal{X}$ the nearest neighbor in $\mathcal{Y}$. Linear search can be used for this task and it is the best solution if the combined number of features of the two images is small. However, it is easy to see that the complexity of linear search can become a bottleneck in case we want to find matches between $\mathcal{X}$ and a large dataset of features $\mathcal{D}$. This situation can occur for example when we want to find a correspondence between an image and a big set of other images, for example in image retrieval tasks. The typical situation in these cases is to use approximate nearest neighbors algorithms instead of linear search. While these algorithms do not guarantee to find the optimal solution, they offer a significative speed up with respect to linear search and return features which are usually close in distance to exact neighbors. These algorithms usually precompute a hierarchical decomposition of the search space of the features of $\mathcal{D}$, which is used to only consider a smaller subset of features of $\mathcal{D}$ while looking for the nearest neighbor of a feature of $\mathcal{X}$. Example of such algorithms are the $k-d$ tree algorithm and the hierarchical clustering.

This report is structured as follows. In the next section, we will briefly present the SIFT features. In the third section, we will discuss several algorithmic approaches which can be used to find approximate nearest neighbors. In the fourth section, we will experimentally test hierarchical clustering based methods for fast matching of SIFT features on a real dataset. The last section shortly discusses the results of the experiments.

II. SIFT features

SIFT features are a particular kind of features which can be extracted from images. They are not only invariant to image scale and rotation, but also provide robust matching across a wide range of affine transformations, change in 3D viewpoint, addition of noise and changes in illumination. The Scale Invariant Feature Transform (SIFT) [2] is an algorithm which *transforms* image data to a set of scale invariant keypoints-descriptors of local features. The four major step of the SIFT algorithm are the following

1) **Scale-space extrema detection:** In this stage, it is searched over all scales and image locations. Locations of the image which are invariant to scale and orientation, also called keypoints candidates, are identified.
2) **Keypoint localization:** For each keypoint candidate, the algorithm tries to determine its location and scale fitting a model. Keypoints are selected based on measures of stability of keypoint candidates.
3) **Orientation assignment:** One or more orientations are assigned to each keypoint.
4) **Keypoint descriptor:** For each keypoint, local image gradients are computed in its surrounding region of the image. These gradients are transformed into a representation which allows the image to be robust on significant levels of change in illumination and local shape distortion: the feature descriptor.

Now we will briefly explain each of these steps. That is, we will show how to transform an image $I(x,y)$ to a set $F_I = \{f_1, f_2, ..., f_l\}$ of SIFT features, where $f_i = (k_i, d_i)$, with k_i and d_i respectively the keypoint and the descriptor associated to feature f_i.

Scale-space extrema detection: In this step, keypoint candidates are identified. Let the two-dimension gaussian function be

$$G(x,y,\sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Let $L(x, y, k\sigma)$ be the convolution of the original image $I(x, y)$ with the Gaussian blur $G(x, y, k\sigma)$ at scale $k\sigma$, that is $L(x, y, k\sigma) = G(x, y, k\sigma) * I(x, y)$. The DoG image between scales $k_i\sigma$ and $k_j\sigma$ is $L(x, y, k_i\sigma) - L(x, y, k_j\sigma)$. In SIFT, the original image is convoluted with Gaussian blur at different scales. The convolved images L are grouped by octave (i.e. doubling the value of σ), and the values of k are selected in order to obtain a fixed number of convolved images per octave. The DoG images are computed as difference from adjacent Gaussian-blurred image for each octave. Maximum and minimum points of DoG images are detected by comparing every pixel with its 8 neighbors pixel in the current scale, and with the 9 pixels at the adjacent scales (for a total of $8 + 2 \cdot 9 = 26$ pixel comparisons). If the pixel value is the maximum or the minimum among all its neighbors, then it is selected as a keypoint candidate. The original paper [2] gives some empirical data on the best choice of parameters in practise: 4 octaves, 5 images for octave, $\sigma = 1.6$ and $k = \sqrt{2}$.

Keypoint localization: The list of keypoint candidates obtained in the previous step is refined. We want to remove the keypoints which do not have enough contrast or lie along an edge. First of all, we want to obtain better locations for the candidate keypoints. In fact, the candidate keypoints obtained in the step before have a pixel unit precision. However, it is possible to obtain a subpixel precision, thus a better precision on the location of the maximum/minimum of the DoG image, by interpolation of nearby data. This is done by using a second-order Taylor-series around a keypoint:

$$D(\mathbf{x}) = D + \frac{\partial D^T}{\partial \mathbf{x}}\mathbf{x} + \frac{1}{2}\mathbf{x}^T\frac{\partial^2 D}{\partial^2 \mathbf{x}}\mathbf{x}$$

The location of the extremum $\hat{\mathbf{x}}$ is found by setting the derivative of the function above to 0. On solving, we'll get a subpixel precision on the location of the keypoint. The function value at $D(\hat{\mathbf{x}})$ is used for rejecting keypoints with low contrast. A keypoint is discarded if $|D(\hat{\mathbf{x}})| < 0.03$.

Moreover, we would also like to remove keypoints which lie along an edge. In fact, DoG images will have strong response around an edge, even if the location is poorly determined. The approach is the following. We compute a Hessian matrix H at the location and scale of the keypoint.

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}$$

The Hessian eigenvalues are proportional to the principal curvatures, thus the matrix characterizes the edge. A keypoint is discarded if the ratio between the square of the trace and the determinant of H is above a certain threshold th.

$$\frac{\mathtt{Tr}(H)^2}{\mathtt{Det}(H)} > th$$

The experimentally determined [2] value of th is 10.

Orientation assignment The scope of this step is to consistently assign a orientation to keypoints based on local image properties. Then, the keypoint descriptor can be represented relative to this orientation and thus achieve invariance with respect to image rotation. An orientation histogram is formed from the gradients orientation of sample points (using pixel differences) within a region around keypoints. An histogram with 36 bins covering 360 degrees of possible orientations is created. The highest peak in the histogram is detected and used to create a keypoint with that orientation. Moreover any local peak on the histogram that is within 80% of the highest peak is used to create other keypoints at the same location but with different orientation.

The result of the previous three steps assign to every keypoint an image location, scale and orientation. These parameters are invariant to rotation and change of scale of the images. A keypoint k can be represented as

$$k = (x, y, s, \mathbf{o})$$

where x and y are the coordinates of the keypoint in the image, s is the scale and $\mathbf{o}$ is the orientation of the keypoint.

Keypoint descriptor: In this step, for each keypoint a descriptor is created. These descriptors need to be both highly distinctive and as invariant as possible to remaining variations, as change of illumination or 3D viewpoint. The procedure to compute a SIFT keypoint descriptor is the following. A 16x16 neighbourhood region around the keypoint is taken and for each sample point in this region it is computed the gradient magnitude and orientation. These gradients are accumulated into orientation histogram of 8 bins summarizing the contents over 4x4 subregions. As in a 16x16 region there are 16 4x4 sub-blocks, and for each sub-block the number of bins is 8, the total number of features is $16 \cdot 8 = 128$. These 128 values form a vector which is used as descriptor d. To enhance the quality of the descriptor, further manipulations (as for example normalization) are performed to obtain the final descriptor. To wrap it up, a SIFT keypoint descriptor is a 128-dimensional vector.

III. Approximate nearest neighbors

In the previous section, we have briefly discussed how to obtain from an input image $\mathcal{X}$ a list of SIFT features $F_{\mathcal{X}} = \{f_1, ..., f_l\}$, where $f_i = (k_i, d_i)$. Given two images $\mathcal{X}$ and $\mathcal{Y}$, a very important task is to match the features of the two images. The best candidate match for a feature $f = (k, d) \in F_{\mathcal{X}}$ is found by identifying its nearest neighbor in $F_{\mathcal{Y}}$. The nearest neighbor of f is defined as the keypoint with the minumum euclidean distance of its descriptor to d.

However many features from an image will not have any correct match to the other image, as for example they might be the result of background clutter. It is possible to heuristically drop features that do not have a good match with the Lowe ratio test. In this test, we compare the value of the distance of the nearest neighbor with the value of the second nearest-neighbor. Ideally, correct match (distance of nearest neighbor) will have a way lower distance than uncorrect matches (distance of the second nearest-neighbor). Lowe [2] suggests to drop all features for which the ratio between these two values is greater than 0.7, as they may lead to uncorrect matches.

For object recognition, cluster of at least three features matches which agree on an object and its pose are first identified. Then each cluster is checked by performing a detailed geometric fit to the model, and the result is used to accept

or reject the interpretation. More advanced techniques, for example based on the Hough transform, can also be employed [2].

We have seen that the matching of features revolves around the algorithmic problem of finding the nearest neighbors. That is, we want to solve the k-nearest neighbor problem (k-NN). The problem is formulated as follows: given set of points P from a metric space $\mathcal{M}$, a positive integer k and a point $q \in \mathcal{M}$, we want to identify the set $\texttt{NN}(q)$ of the k nearest point to q in P. As an example on how the k-nearest neighbor problem applies to feature matching, consider the same case of above, where we want to match the SIFT features of two images $\mathcal{X}$ and $\mathcal{Y}$. For each feature descriptor $\hat{d}$ of image $\mathcal{X}$, we solve the k-nearest neighbor problem where $P = \{d : \exists(k', d') \in F_{\mathcal{Y}}, d' = d\}$, $q = \hat{d}$ and $k = 1$. In case we also want to perform the Lowe ratio test, we set $k = 2$.

Assuming that the distance computation is $O(1)$ (for SIFT descriptors, the distance computation is linear in the vector length, that is 128), the k-NN problem can be solved using linear search with time complexity $O(|P| \log k)$. The algorithm is pretty simple and uses an heap that can contain at most $k+1$ elements at any given time. The heap stores distances as keys.

Algorithm 1 LinearSearch(P, q, k)

```
PQ <- empty max-heap
for all p ∈ P do
    compute dist(q, p)
    insert the couple key-value (dist(q, p), p) into PQ.
    if size(PQ) > k, remove the top element of PQ.
end for
```

It's easy to see that the algorithm returns the correct solution. In fact a point p is removed from the priority queue only if there are at least k points which have a smaller distance to q.

As any point is inserted or removed only once in the priority queue, and the size of the priority queue is $O(k)$, the time complexity of the algorithm is $O(|P| \log k)$. Thus, if we want to find the nearest neighbor ($k = 1$) of all features in $\mathcal{X}$ to the features of image $\mathcal{Y}$, the total time complexity is $O(|F_{\mathcal{X}}||F_{\mathcal{Y}}|)$. Hence linear search is feasible only if the number of features of the two images is small. However in many applications, we want to find the matches between the features of an image $\mathcal{X}$ and a dataset of features $\mathcal{D}$. The dataset $\mathcal{D}$ can be seen for example as the union of many features coming from different images. This situation happens in image classification or retrieval tasks, where we want to find the correspondences between a query (or input) image and a dataset of images. If the size of $\mathcal{D}$ is large, linear search is not feasible due to its time complexity.

There are two important observations to be made. First of all, in the case we want to find the matches with a dataset of features $\mathcal{D}$, we usually set a value $k > 1$ in the k-NN problem. In fact a feature of the image $\mathcal{X}$ can be correctly matched to more features of $\mathcal{D}$. Imagine as an example a dataset that contains multiple images of the same scene. Secondly, the Lowe ratio test needs to be adapted for the same reasons just mentioned. In fact, if there are multiple images of the same object, then we define the second-closest neighbor as being the closest neighbor that is known to come from a different object than the first in the Lowe ratio test.

Therefore, for matching features with a large dataset, alternative solutions to linear search needs to be adopted. In these situations, linear search is replaced with approximate nearest neighbors algorithm that can offer speedups of several orders of magnitude over linear search, at the cost of giving up on obtaining the exact nearest neighbors. Many approximate algorithms for the k-NN problem are based on the construction of a data structure over the set P, which is used during search. This data structure allows to answer to the query "what are the k nearest neighbors to q in P" by only considering a fraction of the points of P. Typically, this data structure perform a hierarchical decomposition of the search space and can be represented as a tree. It's important to point out that if $\mathcal{D}$ is fixed, the data structure needs to be constructed only once and then can be used to answer to an arbitrary amount of queries. Thus the time cost of constructing the data structured is amortized over the number of queries. In this section, we will see two of these data structures: $k-d$ trees and hierarchical clustering trees.

k-d tree. The k-d tree (short for k-dimensional tree) is an important data structure which is widely used for approximate nearest neighbor searches. The drawback of this data structure is its low performance in high-dimensional spaces. As SIFT feature descriptors are 128-dimensional vectors, k-d tree is impratical for SIFT feature matching. However, k-d trees are very effective for approximate nearest neighbors in low dimensions, for example with smaller feature descriptors, and for this reason they are shortly presented in this section.

The k-d tree is recursively constructed over a set $P \subseteq \mathbb{R}^d$. Beginning with the full set P, the dimension i with the biggest variance is found. A cut is made at the median value m of the data in that dimension, so that the two halves induced by the cut are equally sized. An internal node is created to store i, m and the point with the median value m on i-th dimension. The process is recursively performed on both halves of the data. The tree obtained in this process is a balanced binary tree with depth $\lceil \log_2 |P| \rceil$. The leaves of the tree, also called bins, form a complete partition of P.

This data structure can be used to find the exact nearest neighbor of a given point q, using a backtracking branch-and-bound search. First, the bin containing q is found traversing the tree. This step requires only $O(\log |P|)$ iterations. The point in the bin is a good approximate of the nearest neighbor of q. Then, in the backtracking stage, whole branches of the trees can be pruned if the region of the space they represent is further from the query point than the current distance from q to the nearest neighbor yet seen. Search terminates when all branches have been either explored or pruned. The search is exhaustive, thus the correct solution is found. In low dimensional space, this search is very fast and large portions of the tree are pruned. However in high dimensional space, the pruning is not as effective and the performance degrades quickly.

The search described above can be slightly modified in order to be made faster, at cost of obtaining an approximate solution. This strategy is called BBF (Best Bin First) [3]. With this

method, a min-priority queue is used: the idea is to look for bins in order of increasing distance from the query point. During the search described above, when a decision is made at the internal node to branch in one direction, the other is added to the priority queue. The key of this entry is the distance of the query point to the node. After a leaf node has been examined, the top entry of the priority is removed and used to continue the search. In order to speed up the method, we can put an upper limit on the number of leaves we can visit. This limit regulates the accuracy of the approximation (if it is removed, the method obtains the optimal solution).

Hierarchical clustering tree Hierarchical clustering trees are obtained by recursively clustering the input points and constructing a tree in which every non-leaf node contains a cluster center and the leaf nodes contain the input points. The general pseudocode that builds a hierarchical clustering tree of a set of points $P \subseteq \mathcal{M}$ and with maximum leaf size S_L is presented below.

Algorithm 2 HierarchicalClusteringTree(P, S_L)

if $|P| \leq S_L$ **then**
  create leaf node with the points in P
else
  $Q \leftarrow$ subset of points from space $\mathcal{M}$
  $C \leftarrow$ cluster the points in P around nearest centers Q
  for all $C_i \in C$ **do**
    create non-leaf node with center Q_i
    call HierarchicalClusteringTree(C_i,S_L)
  end for
end if

There are many different strategies that can be used in order to select the points Q in the recursive step. If the cardinality of Q is fixed, $|Q| = K$, then we say that the hierarchical clustering tree has a branching factor K. For example a possible strategy is to use k-means to find a set Q of K points. In this report, we will focus on this choice: select Q as K random points from P. This can be done very fast. The authors of [1] note that other approaches for the choice of Q, as trying to minimize the squared error with the centers selection (k-mean), do not bring improvements if used for nearest neighbor search. Moreover, if a random approach is used to select the points in Q, multiple trees constructed from the same set P are significantly different. Therefore, it is possible to construct multiple trees and explore them in parallel to enhance the search. During our experiments, we will also the test to use the farthest-first traversal algorithm to select a fixed number K of centers Q. The farthest-first traversal algorithm iteratively picks centers that are farthest to the previously selected centers. The time complexity to find K centers in a set of $|P|$ points using the farthest first traversal algorithm is $O(k|P|)$ (way higher than the time required to select K random centers).

The algorithm used to search in parallel hierarchical clustering trees is presented in Algorithm 3. The strategy adopted is very similar to the one described for the BBF of the $k-d$ tree. The idea is to use a min-priority queue to guide the search. The algorithm receives in input a set of hierarchical clustering trees $\mathbf{T} = \{T_1, T_2, ...\}$ (T_i is also seen as the root node of the tree), a query point q, a parameter L (analog to the upper limit of the approximate BBF search), and an integer k, and outputs a set of k approximate neighbors of the query point. As observed in linear search, we can bound the maximum size of the priority queue R to $k+1$ elements. The parameter L specifies the degree of approximation desired from the algorithm. We define search precision as the intersection of the true set of k-nearest neighbors and the set of points returned by the execution of the knnSearch algorithm divided by k. The search precision is 1 if the k points returned by the approximation algorithm are also the optimal solution for the k-NN problem. As we increase L, the time required by the search and the precision both increases. Thus we can tune L to obtain precision-time tradeoffs.

Algorithm 3 knnSearch($\mathbf{T}, q, L, k$)

$l \leftarrow 0$
$PQ \leftarrow$ empty priority queue (min)
$R \leftarrow$ empty priority queue (min)
for all tree $T_i \in \mathbf{T}$ **do**
  call TraverseTree(T_i,PQ,R)
end for
while PQ not empty and $l < L$ **do**
  $N \leftarrow$ top of PQ
  call TraverseTree(N,PQ,R)
end while
return K top points from R

procedure TRAVERSETREE(N, PQ, R)
  if node N is a leaf node **then**
    search all the points in N and add them to R
    $l \leftarrow l + |N|$
  else
    $C \leftarrow$ child nodes of N
    $C_q \leftarrow$ closest node of C to query q
    add all nodes in $C - C_q$ to PQ.
    call TraverseTree(C_q,PQ,R)
  end if
end procedure

IV. EXPERIMENTS

We conducted some experiments to compare the performance between the linear search and approximate search using hierarchical clustering trees for the k-NN problem, applied to matching of SIFT features. The experiment framework is the following: the dataset chosen is the Notredame dataset (from Winder/Brown patch dataset), consisting of 406 patches of images, each containing 256 images in a 64x64 gray-scale pixel format. To build our dataset, we used 10'000 images extracted from the 406 patches, for a total of 367'751 SIFT features. The OpenCV library has been used for interfacing to the SIFT feature extraction method. To build the hierarchical clustering trees, and to perform k-NN search on them, we used the *cv::FLANN* library [4]. The implementation of the experiment (https://github.com/marioavdullaj/SIFT) is composed by the following parts:

Feature extrapolation. A C++ program has been developed which receives in input the dataset, crops and extracts all the images, and detects SIFT features using the OpenCV SIFT library. The resulting SIFT features are saved in an external YAML file, since their computation is time consuming.

Feature matching. The dataset of image features is taken in input by a second program developed in C++. The program uses k-NN search algorithms to match a random query of 10 images (not used to generate the feature dataset) with the feature dataset. We used the value $k = 8$ for the k-NN problem throughout all the experiments. The program first uses the linear k-NN search. The computed data are the set of the exact k nearest features for every query image feature, and the time required to perform the search over the dataset. Then, the program builds several hierarchical clustering trees, using random centers, with a different combination of K (branching factor), t (number of parallel trees) and S_L (leaf size). The values of the parameters are chosen over the sets $K = \{2, 4, 8, 16, 32, 64\}$, $t = \{1, 2, 3, 4, 8, 16\}$ and $S_L = \{6, 150, 500\}$. For every hierarchical tree constructed, we perform multiple calls to the approximate method $kNNSearch$ (see previous section), varying the value of L. The time to perform the query is stored and also, for each run, the precision of the returned set of neighbors is calculated (see ending of previous section for details). Finally, the program outputs all the data collected in a YAML file.

Performance comparision. The final step consists on comparing the performances of the k-NN search algorithms. We provide an implementation of a *Python* script which takes in input the results of the feature matching program and plots the speed up as function of the search precision.

The building times of the hierarchical clustering trees are not reported here (even if they are computed by our data-collecting program). The building cost is amortized over the number of queries used and provides little information. Using a random selection of centers to build the hierarchical clustering trees, the maximum time required to build the trees is, for the most time-consuming combinations of parameters (e.g. $t = 16$), comparable to the time needed by the k-NN linear search over 10 image queries in our dataset. We also tested the construction of hierarchical clustering trees which use the farthest-first traversal algorithm for the selection of the centers. The build phase of the trees in this case takes noticeably more time due to the time complexity of the farthest-first traversal algorithm, however the results are very similar to the one obtained using a random selection of centers and there is no improvement in precision/speedup. This fact is in agreement with the statement in [1], that heuristics to select the centers do not bring significative improvements with respect to a random selection. The data and graphs obtained using the farthest-first traversal algorithm can be found on the GitHub repository. All the graphs in this report are obtained using a random selection of centers.

Figure 1 shows how the branching factor has an evident impact on the precision of the algorithm. Higher branching values obtain higher precision values. It is interesting to notice that for lower value of precision (under 60%), lower branching values achieve better speed up results, requiring also less building time.

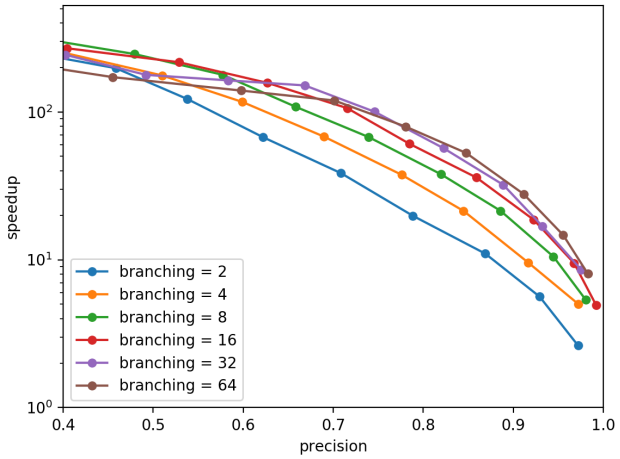

Figure 1: Speed up w.r.t. linear search for different branching values.

The number of parallel trees used to perform the search plays a fundamental role, as Figure 2 shows. Generally, an higher number of trees can give better results in terms of speed up, especially if we want to achieve high percentage of precision. In the experiment, $t = 16$ parallel trees, and tuning the L parameter to achieve more than 90% of precision, there is a 100x speed up factor with respect to linear search. However, it must be pointed out that using an higher number of parallel trees leads to more time-consuming building time. In time-constraints situations, choosing a smaller number of trees (such as $t = 4$) can achieve good speed up results at the same precision levels.

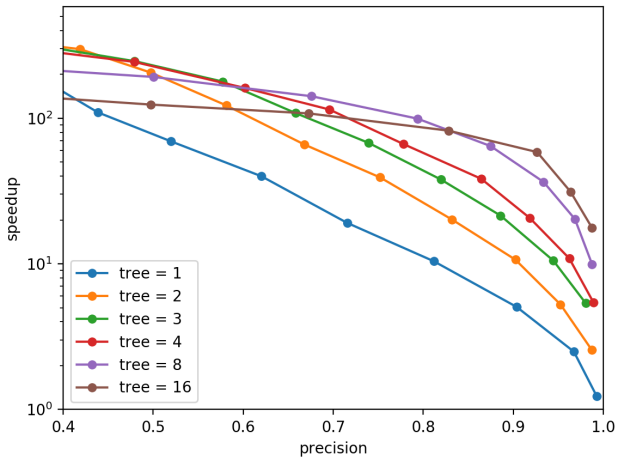

Figure 2: Speed up w.r.t. linear search for different numbers of parallel trees.

The last parameter to be examined is the leaf size. From figure 3 we can see that a maximum leaf size of 150 performs

better than a small leaf size or a large leaf size. This can be explained by the fact that small leaf sizes increase the time overhead of traversing the tree, since more leaves are explored during the search. However, if the leaf size is too big, way fewer leaves are visited and the smaller probability to find exact neighbors in the few leaves visited sensibly affects the precision.

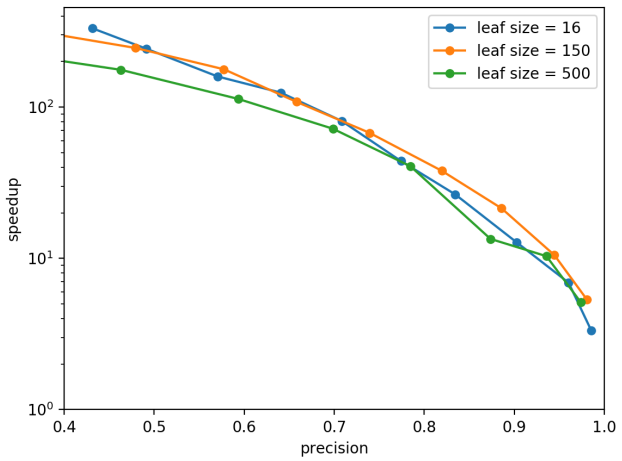

Figure 3: Speed up over linear search for different maximum leaf sizes.

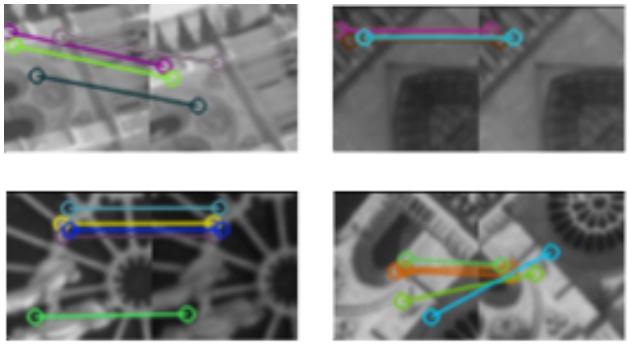

Figure 4: Feature matching of 4 queries with linear search.

In Figure 4,5 and 6 the results of possible found matches between the query image and the dataset images obtained with different k-NN search strategies are reported. The matches are obtained in the following way. For each query image feature, it is noted from what dataset image the nearest neighbor of the query feature comes from. The dataset image with more matches is reported as the best possible match. Two heuristics are applied: a match between two images is found only if at least 5 feature matches are found. If considering only the nearest neighbor, no image dataset has at least 5 feature matches, the second nearest neighbors are considered. This processes is repeated until at least 5 feature matches are found (or we don't have more neighbors to consider). If no image matches are found in this way, we select the dataset image with most feature matches. Also, a feature match is considered only if the distance of the two features is at most 2 times the distance between the closest match of a query image feature with the dataset. This heuristic allows to not consider many bad matches. We remark that in this situation, the Lowe test cannot be used as there may be multiple images of the same object in the dataset. In Figure 4, it is shown the best dataset image matches of four different queries obtained using linear search (the image on the left is the query, the image on the right is the image of the dataset). The search time of the run is 8.5 seconds in our experiment. As we can see, the first three query images have been correctly matched with images of the same object, while the last one is mismatched. The mismatch though can be reasonably justified, since the shape of the two images are indeed very similar. If we want to filter out the bad matches, we can calculate if the matches agree with the geometry of the image (for example using Hough transform).

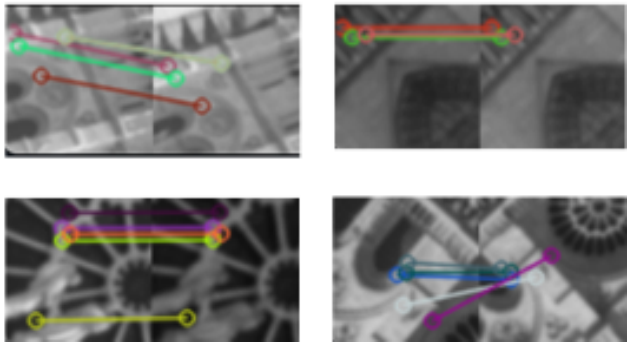

Figure 5: Feature matching of 4 queries with fast approximate search. S_L = 6400. Precision = 98%.

The same query has been used to calculate matches using approximate k-NN search. A hierarchical clustering tree is built using B = 32, t = 8 and S_L = 150. Two different runs have been made, using $L = \{200, 6400\}$. The result of the first run ($L = 6400$) is shown in Figure 5. The image matches are identical to the ones obtained from linear k-NN search. The time required to build the hierarchical clustering tree is T_B = 27 sec, while the k-NN search over the tree requires only 0.53 seconds. There is a 16x speed up factor with respect to linear search.

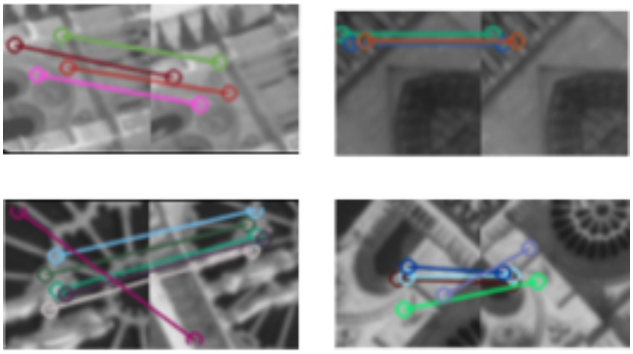

Figure 6: Feature matching of 4 queries with fast approximate search. S_L = 200. Precision = 60%.

We remark that if we increased the number of queries, the time required for the linear k-NN search would increase

linearly; however the time required to build the hierarchical clustering trees is independent on the number of queries.

The results of the second run ($L = 200$) are shown in Figure 6. The first two images are matched correctly, while the other two are mismatched. The k-NN search time required in this case is only 0.08 seconds. The higher number of mismatches is due to the lower precision obtained using $L = 200$ ($\simeq 60\%$ in our experiments for this hierarchical clustering tree).

V. Conclusions

In this report, we studied a fast approximate feature matching algorithm which uses hierarchical clustering trees to perform k-NN searches. We confirmed the experimental results of [1], that the usage of hierarchical clustering tree can considerably speed up the search of nearest neighbors, while keeping high accuracy. The method can be used to reduce the time complexity of features matching, which is a critical problem in computer vision.

References

[1] M. Muja, and D. G. Lowe, "Fast Matching of Binary Features", CRV '12 Proceedings of the 2012 Ninth Conference on Computer and Robot Vision Pages 404-410, 2012.

[2] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints", International Journal of Computer Vision November 2004, Volume 60, Issue 2, pp 91–110, 2004

[3] Jeffrey S. Beis and D. G. Lowe, Shape Indexing Using Approximate Nearest-Neighbour Search in High-Dimensional Spaces", CVPR '97, ACM, pp 1000, 1997

[4] Marius Muja and David G. Lowe, "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration", in International Conference on Computer Vision Theory and Applications (VISAPP'09), 2009.