Fast SIFT Matching

3D - Augmented Reality

A.Y. 2018-2019

Padoan Alberto - ID 1179704
Ravagnani Giuseppe - ID 1176391

INTRODUCTION

In computer vision matching problems, it is more and more frequent to use binary features found with algorithms as BRIEF, ORB and BRISK. Binary features are compact to store and the match can be performed quite fast using Hamming distance between pairs of features. This operation can be performed with a XOR and a bit count on the result: manipulating compact number of bits is nowadays an easy challenge. The problem rises when the dataset is large and so the linear search is efficient no more. For this reason Muja e Rowe developed a method based on priority search of multiple hierarchical clustering trees [1]. Starting from this method, our purpose is to build a speed-up algorithm to be applied for the matching of vector-based features found with sophisticated algorithms such as SIFT or SURF; in particular we are going to focus on SIFT features. In fact vector-based features, differently from binary features, are more difficult to manipulate and less compact. The distance between two pairs of vector-based features is computed with the Euclidian distance with a consequent increase of the computational cost. It follows that, with large datasets, the computational speed is drastically decreased. So in this report we are going to analyse the performances of a linear search algorithm and a tree search algorithm similar to the one presented in [1] with proper refinements.

For our tests we used Matlab platform: the dataset of features is provided by the Oxford University [2] and for computing descriptors we used the David Lowe's SIFT keypoint detector [3]. The data structure useful to create the tree is written by us and to implement the search three algorithm we used a priority queue provided in [4]. For both tree creation and tree search methods we used the algorithms presented in [1]. The main difference is that we do not perform a parallel search over multiple trees. In fact, since it could be possible to use the parallel search with the linear search too, we decided to implement single thread algorithms because we are interested in the pure comparison between the tree search and the linear search algorithms.

SIFT

SIFT (Scale-Invariant Feature Transform) [5] is a computer vision algorithm used to extract vector-based descriptors by an image in order to perform tasks as object detection, image stitching, match moving, video tracking etc… The descriptor provides a vector representation of a local patch of the image and the match is done between features of two or more images but, it may happen, that one or more images have undergone a transformation. SIFT, differently from simple corner detection algorithms (Harris, Fast, etc.), is invariant at the same time to scale, rotation, illumination and view point, making this algorithm one of the most used for computer vision problems. These peculiar properties comes from the built of the SIFT descriptor that is performed in the following steps.

1) Scale space

Fist of all we need to build a scale space which aim is to replicate the multi-scale nature of an object. In fact we may be interested in to seeing a leaf of a tree or the entire tree; in the latter we do not need some details (leaves, twigs, etc.). So, starting from the original image, we progressively blur it. Then the image is resized to half size and again progressively blurred. By blurring an image, it is intended the convolution between a gaussian operator and the image, as follows:

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y).$$

Where x, y are the pixel coordinates and

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

is the Gaussian operator that depends by image coordinates and σ, that defines how much the image is blurred and usually is augmented by a factor $\sqrt{2}$ from the previous one.
This process is usually repeated more or less 4 times in order to obtain 4 octaves with about 5 blur levels as it is represented in figure 1.

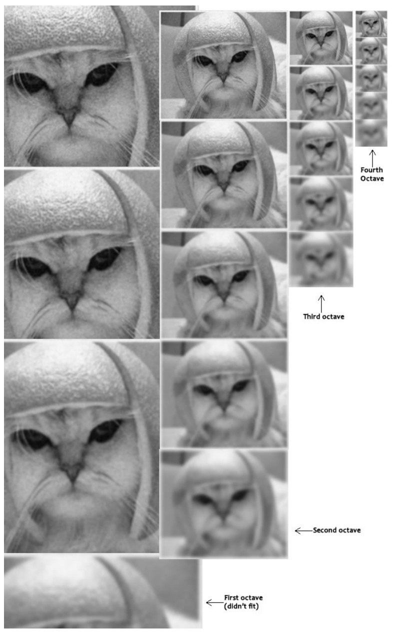

Figure 1: scale space of an image.

2) Difference of Gaussians

Corners and edges are a good source of keypoints, so we need to extract them. Usually it is used the Laplacian of Gaussian (LoG) that, in practice, is the second order derivative. But it is quite expensive in therms of computational cost, so in SIFT it is used an approximation of the LoG, that is the Difference of Gaussian (DoG). The DoG is obtained by the difference of Gaussian blurring of an image with two different σ, let it be σ and $k\sigma$ as it is represented in figure 2.

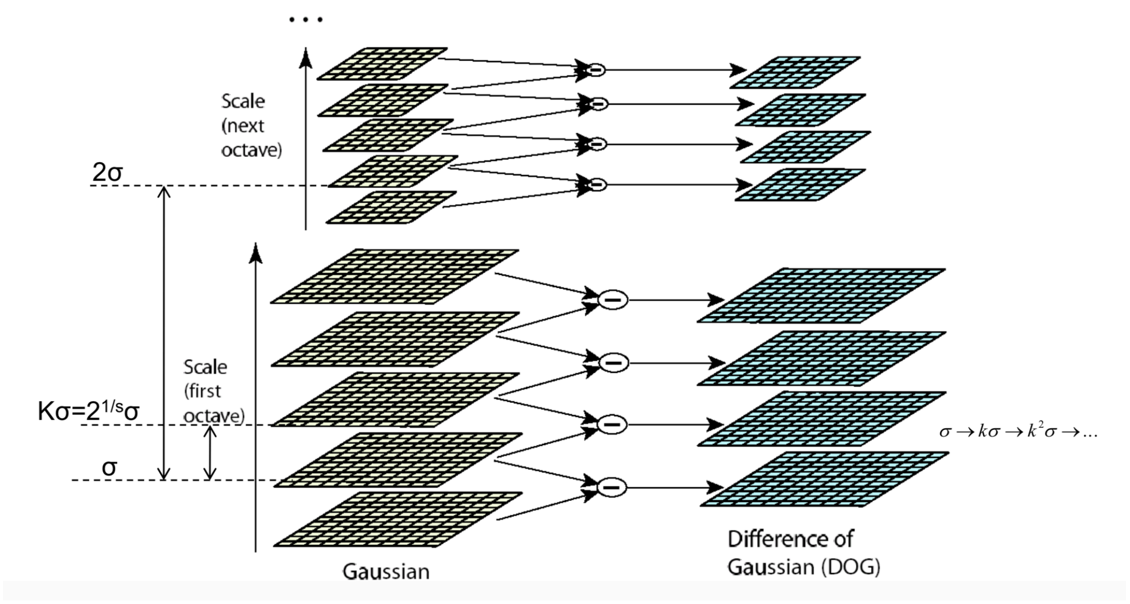

Figure 2: DoG starting from a scale space.

3) Maxima and minima detection

In this phase each pixel is compared with its 8 neighbours as well as 9 pixels in next scale and 9 pixels in previous scale (Figure 3): a pixel is a potential keypoint if it is the local maxima or minima.

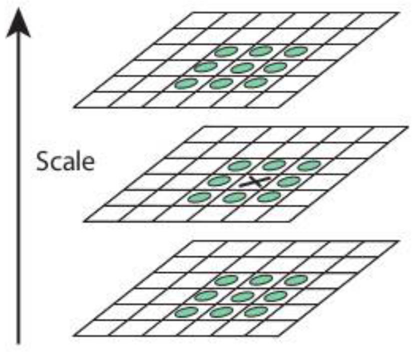

Figure 3: local maxima and minima search.

4) Keypoint localisation

In order to obtain stronger and accurate keypoints, it is computed the second order Taylor expansion of the scale space and, if its modulo value it is less than a certain threshold (usually 0.03), the referring keypoint is rejected. Then, since corners contain stronger keypoints than edges, we are interested only in keypoints that lies on them. In order to do that it is used a 2x2 Hessian matrix, and all the potential keypoints which do not have large eigenvalues are discarded (Figure 4). So now we have only strong keypoints and those which have a low contrast or refer to an edge, are eliminated.

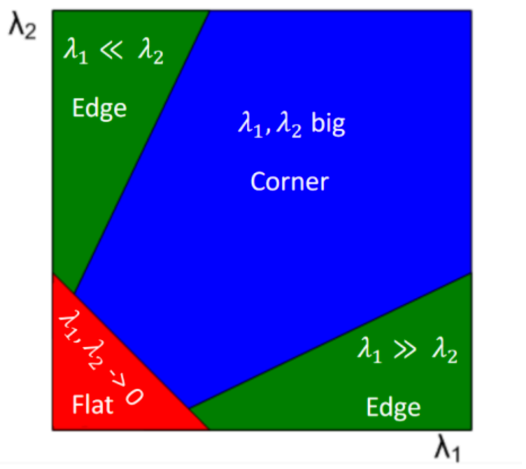
λ_2
$\lambda_1 \ll \lambda_2$
Edge
λ_1, λ_2 big
Corner
$\lambda_1, \lambda_2 \to 0$
Flat
$\lambda_1 \gg \lambda_2$
Edge
λ_1

Figure 4: relations of eigenvalues of the Hessian matrix

5) Orientation assignment

Once invariance to scaling is achieved, we need to achieve invariance to rotation too and so to assign an orientation to each keypoint. The idea is to compute gradient directions and magnitude around a keypoint. So it is created a histogram with 36 bins of 10° and we consider gradients which have the maximum peak. If there are peaks higher than the 80% of the maximum one, they generates new keypoints with same location and scale, but different orientation.

6) Feature vector

Around each keypoint it is considered a 16x16 window sub-divided into 16 4x4 windows. For each-sub window, orientations are calculated again and it is built a histogram with 8 bins of 45° that is weighted with a gaussian function centred at the keypoints (it gives lower value to orientation far from the keypoint). Once they are normalised to 1, we have a descriptor with 128 bins. To obtain rotation independence, the keypoint's orientation is subtracted from each orientation. Finally, to obtain illumination independence, every bin higher than 0.2 (bins associated to illumination artefacts) is downscaled to 0.2 and in conclusion the vector is normalised again.

Tree Creation

First of all we built the tree in which is based the speed-up search algorithm. The process starts with the selection of K random points from the whole dataset. K, named branching factor, is the input parameter of the algorithm. Those K points are the cluster centres and each point of the dataset is assigned to the nearest centre. This process is repeated recursively until the number of points in each cluster is under a certain threshold S_L (maximum leaf size) and so those nodes become leaf nodes. The input parameters of the algorithm are:

- D: the dataset;
- K: branching factor;
- S_L: maximum leaf size.

And the pseudo-code of the algorithm is presented below.

```
create_tree(D, K, S_L)
1: if size of D < S_L then
2:     create leaf node with the points in D
3: else
4:     P ← select K points at random from D
5:     C ← cluster the points in D around nearest centres P
6:     for each cluster C_i∈C do
7:         create non-leaf node with centre P_i
8:         recursively apply the algorithm to the points in C_i
9:     end for
10: end if
OUTPUT: hierarchical clustering tree
```

Searching Algorithms

In this section we present the two searching algorithms at the centre of our comparison. First we present a priority search based on the hierarchical tree built above and then a simple linear search algorithm. The search, differently by binary features, is based on the Euclidean distance. Linear search gives the best results since it searches the point which has the minor distance with respect to the query point. But it is possible to use probabilistic algorithms, such the one we are going to present, that may reduce the precision in order to achieve better timing performances.

Searching in hierarchical clustering tree

This algorithm performs a search over the hierarchical clustering tree. The input parameters of the algorithm are:

- T: the tree in which perform the search;
- Q: the query point;
- K: number nearest descriptors returned by the algorithm;
- L_{max}: max number of points to examine.

The output of the algorithms are K points which are the nearest points to the query point. First of all the search starts with a call of the *traverse_tree* method in which the algorithm finds the closest point to Q and recursively explores it. The unexplored nodes are inserted in a priority queue. Then, until the number of searched points is less than L_{max} , the *traverse_tree* method is executed on the unexplored nodes. The pseudo code of both the searching algorithm and the *traverse_tree* method is reported below.

Searching algorithm

```
1: L ← 0        %number of points searched
2: PQ ← 0       %empty priority queue
3: R ← 0        %empty priority queue
4: call traverse_tree(T.root, PQ, R, Q)
5: while PQ not empty and L < $L_{max}$
6:      N ← top of PQ
7:      call traverse_tree(N, PQ, R, Q)
8: end while
9: return K top points from R
method traverse_tree(N, PQ, R, Q)
1: if N is a leaf node then
2:      search all the points in N and add them to R
3:      L ← L + |N|
4: else
5:      C ← child of N
6:      $C_q$ ← closest node of C to query Q
7:      $C_p$ ← C/{$C_q$}
8:      add all nodes in $C_p$ to PQ
9:      call traverse_tree($C_q$, PQ, R, Q)
10: end if
```

Linear search

The following linear search algorithm is based on the bubble sort algorithm. The input parameters are:

- D: the dataset in which perform the search;
- Q: reference descriptor;
- K: number nearest descriptors returned by the algorithm.

The output are the K descriptors nearest to Q. The algorithm runs bubble sort based on the Euclidean distance K times on the dataset and, at the end, the first K elements of the dataset are the searched points. Note that the algorithm is $\theta(NK)$. Below there is the pseudo-code of our implementation.

linear_search(D, Q, K)

```
1: for i = 1 and i <= K
2:      for j = size of D and j > i
3:           if norm(D[j],Q) < norm(D[j-1],Q)
4:                swap D[j] and D[j-1]
5:           end if
```

```
6:            j ← j-1
7:      end for
8:      i ← i+1
9: end for
OUTPUT = first K elements in D
```

Results

Here follows the results and the performances of our algorithm. First of all we are interested into analyse the computational cost of the creation of a tree, then we examine and compare the searching performances and, in the end, we evaluate the matching quality. Please note that we are not interested in the absolute value of timing result: Matlab is not optimised for this type of evaluation and the results may vary with respect to the device used and the processes that are running on it. Anyway, we are interested into the comparison of the results between different methods and parameters.

Tree Creation

The creation of a tree is the most expensive operation of our analysis. We tested it by repeating the creation of a tree with an increasing dataset size (from 100 to 5000 descriptors) and with three different branch parameters: 2, 4 and 8. The maximum leaf size parameter is set to 20. The timing results with respect to the dataset size are reported in Figure 5.

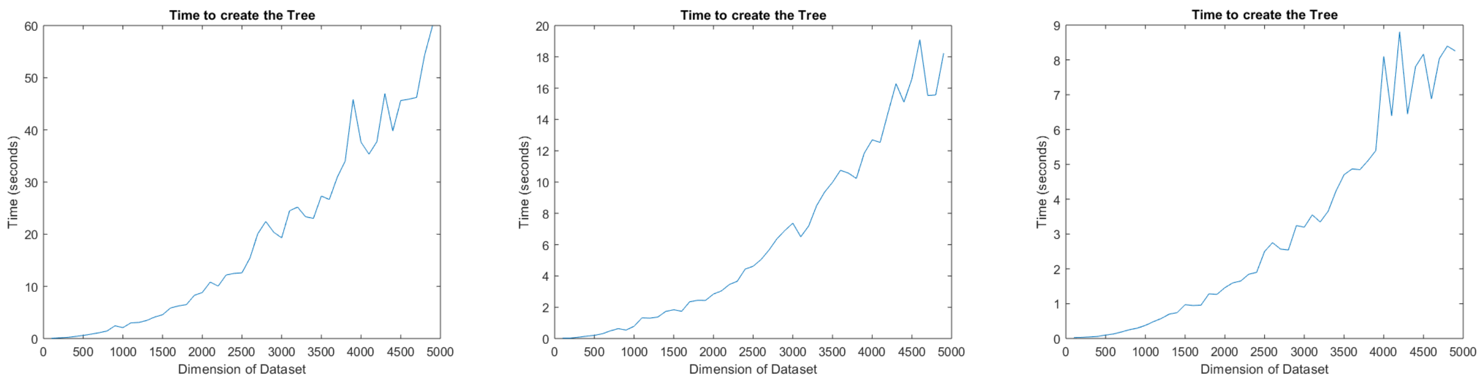

Figure 5: tree search time with branch size 2, 4, 8.

From these graphs we can see that the time grows with the dimension of the dataset. There is a significant difference with respect to the branch size (i.e. the K parameter) too. In fact, although a branch size equal to 2 gives finer results, it requires about 6 times more than the creation of the tree with branch size equal to 8 and about 3 times more than the one created with branch size equal to 4.

Searching Performances

In this section we compare the searching performances between the linear search and the tree search. The tree search is performed on three different trees created with maximum leaf size equal to 20 and branch factors of 2, 4, and 8. As regards the search parameters, K is equal to 3 for both linear and tree search and the L_{max} is one twentieth of the size of the dataset. In Figure 6 we can appreciate the results with an increasing size of the dataset correlated with the time in logarithmic scale.

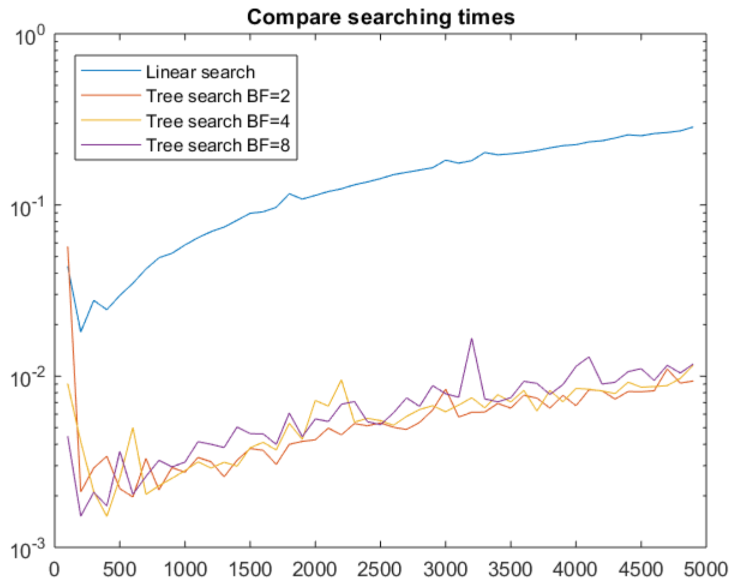

Figure 6: searching times comparison.

The linear search is slower than then tree search but, in order to have a clearer and more precise view between different branching parameters, we plot the same results but comparing multiple searches (Figure 7).

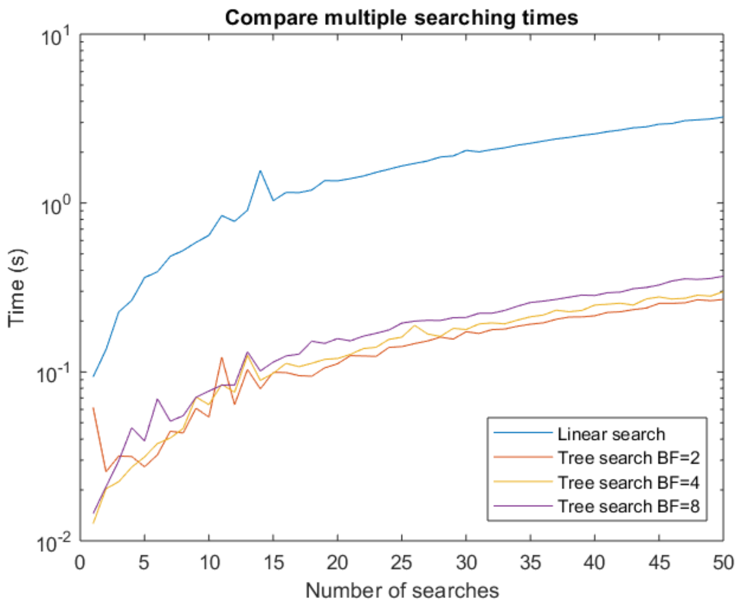

Figure 7: multiple searching times.

The results are very similar but now we can appreciate the differences between different types of tree search. They are very similar but asymptotically it is possible to notice that a higher branch factor gives slightly less searching time.
Since the creation of the tree is very expensive, we are interested into evaluating when the whole tree creation + tree search process is faster than the linear search. So we added to the tree searching time the time required to create the tree and we plot the results in Figure 8.

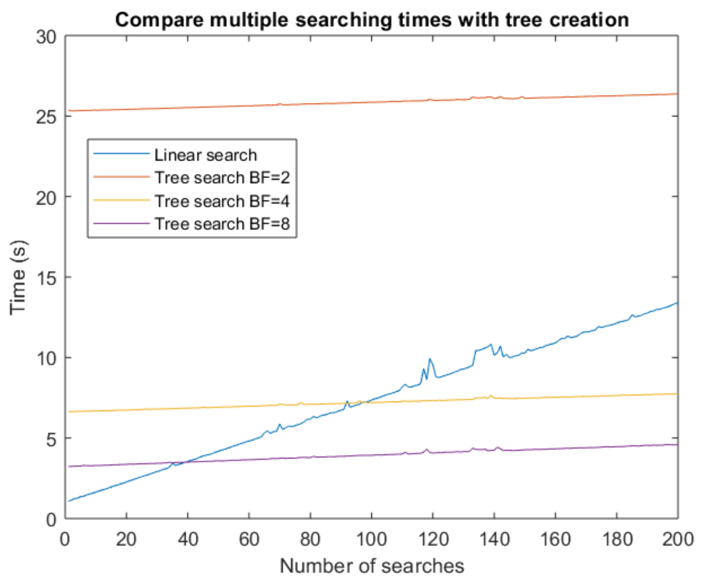

Figure 8: comparison by adding tree creation time.

As we expected, the linear search it is less convenient by augmenting the number of searches. In particular, since the creation of a tree with branch factor equal to 2 is very expensive, it is worse than the linear search and it may require more searches to be faster. Instead, the trees created with branch factor equal to 4 and 8, are much better than the linear search when the number of searches grows. So it is possible to state that the higher is the branch factor the slightly worse is the searching time with respect to minor branch factors but, considering the tree creation time, the performances are significantly improved.
Then we tested our algorithms on two images in order to have visual and practical results. We run the searches on descriptors extracted by two images, one is high resolution and the other is low resolution. In the tables below there are the results of a search tested with different parameters.

First Image - High res	
Number of features	10178
Linear search time	0,65 s
Tree creation BF = 2, $S_L = 10$	332,6 s
Tree search time BF = 2	0,15 s
Tree creation BF = 8, $S_L = 10$	39,16 s
Tree search time BF = 8	0,064 s
Tree creation BF = 16, $S_L = 40$	23,92 s
Tree search time BF = 16	0,073 s
Tree creation BF = 32, $S_L = 80$	11,24 s
Tree search time BF = 32	0,07 s
Tree creation BF = 64, $S_L = 100$	12,12 s
Tree search time BF = 64	0,099 s
Tree creation BF = 128, $S_L = 150$	8,15 s
Tree search time BF = 128	0,061 s

Second Image - Low res	
Number of features	3791
Linear search time	0,24 s
Tree creation BF = 2, $S_L = 10$	39,22 s
Tree search time BF = 2	0,064 s
Tree creation BF = 8, $S_L = 10$	5,67 s
Tree search time BF = 8	0,088 s
Tree creation BF = 16, $S_L = 40$	3,05 s
Tree search time BF = 16	0,066 s
Tree creation BF = 32, $S_L = 80$	2,02 s
Tree search time BF = 32	0,067 s
Tree creation BF = 64, $S_L = 100$	1,98 s
Tree search time BF = 64	0,07 s
Tree creation BF = 128, $S_L = 150$	1,91 s
Tree search time BF = 128	0,073 s

As we said before, we are not interested in the absolute value of those timing results. We are rather interested in the comparison between the obtained measurements. Since the higher is the resolution and the more descriptors are found, we have higher times for the operations in the first image. In addition, in our tests, the matches found by the tree search are always the same for both the images regardless the branch factor and the tree creation parameters (Figure 9 and 10): the main difference is that the time required to found those matches decreases by augmenting the branch factor.

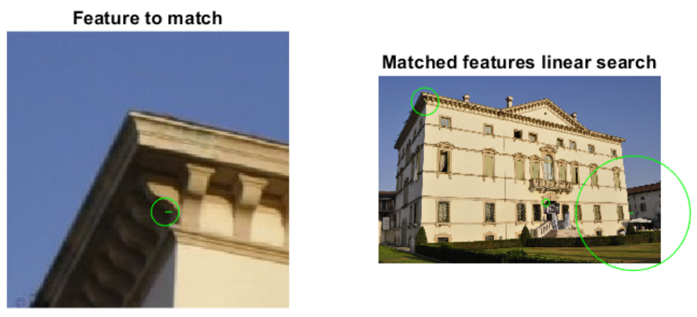

Figure 9: matches in the first image - high res.

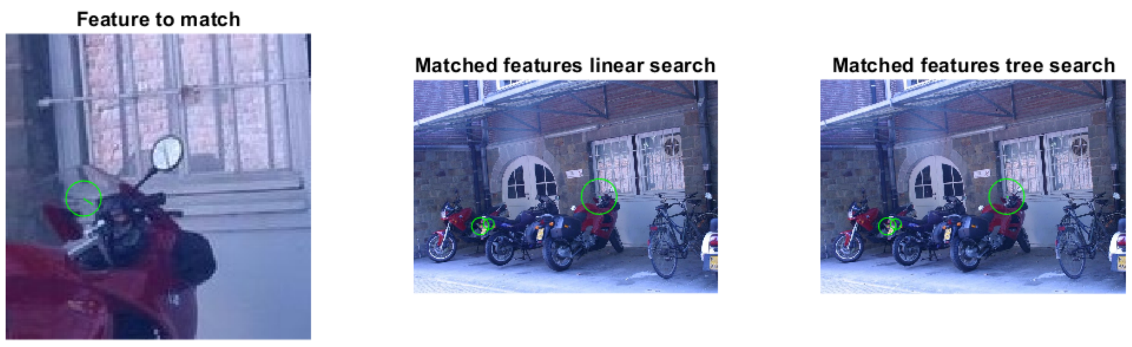

Figure 10: matches in the second image - low res.

For sake of completeness we report the patches found by the linear search algorithm and the tree search in both images (Figure 11, 12).

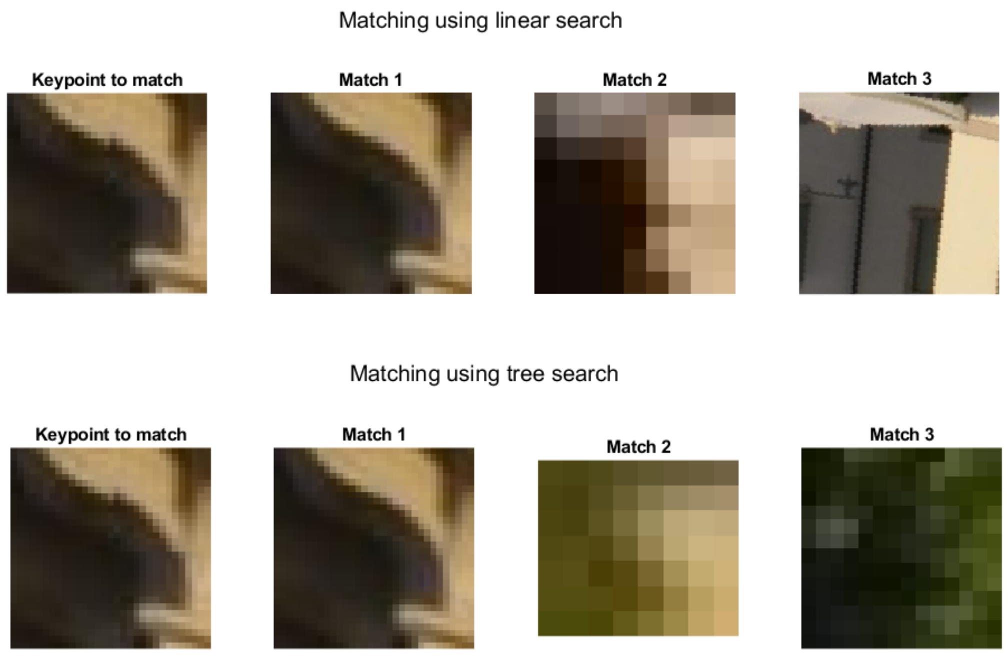

Figure 11: patches found by linear search and tree search for high res image.

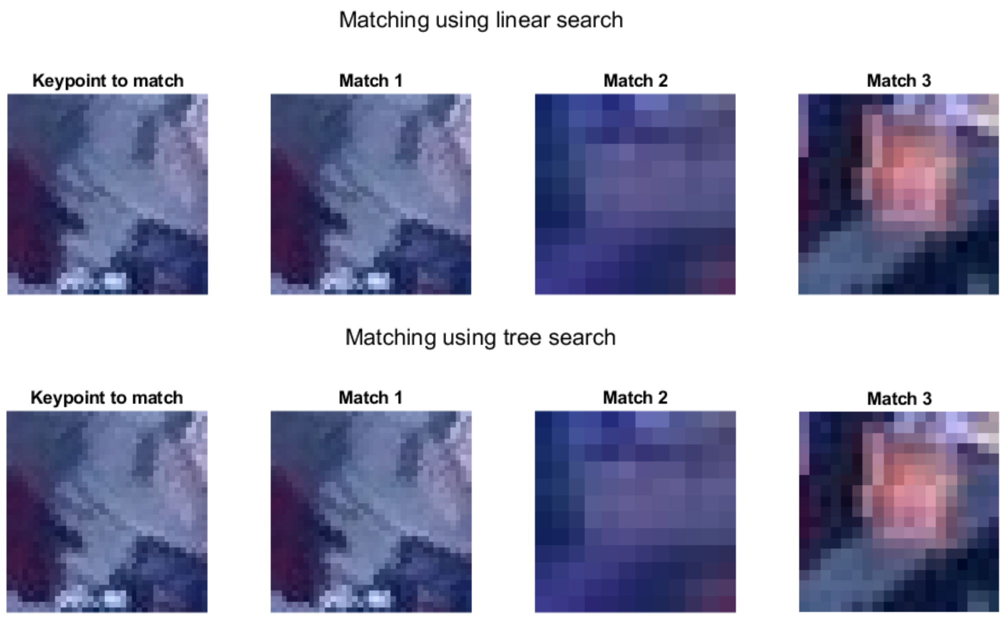

Figure 12: patches found by linear search and tree search for low res image.

Matching Quality

Now we are interested in evaluating the matching quality. We keep the linear search as our landmark because it offers the best searching result with respect to the Euclidian distance. We run SIFT algorithm on an image and the visual representation of the descriptors is provided in Figure 13.

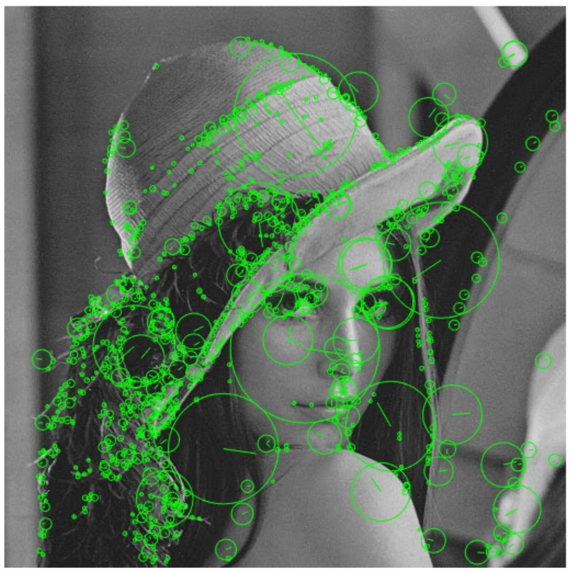

Figure 13: visual representation of descriptors.

Then, SIFT is run in another image that is a portion of the first one. By selecting one descriptor from the second image, we want to find its match on the first image.

First of all we run the linear search algorithm with the following parameters:
- D_to = descriptors of the first image, i.e. descriptors in which find the match;
- d = query descriptor chosen from the second image;
- K = 3, number of neighbours to find.

The neighbours found are represented in Figure 14.

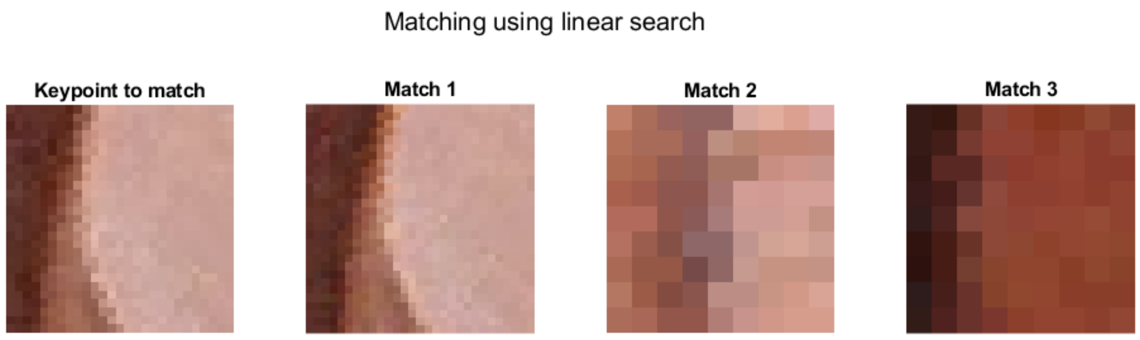

Figure 14: matching using linear search.

As it is possible to notice, the perfect visual match is the one that has the lowest distance from the query descriptor. Anyway, it may happen that the perfect visual match is not necessary the one with the minor distance. In our analysis, we are interested into evaluating the results of the two algorithms with respect to the quality of the Euclidian distance.

Then we built a tree with the descriptors of the first image. The tree is built with our *create_tree* algorithm and has the following input parameters:
- D_to = descriptors of the first image, i.e. descriptors in which find the match;
- branch_factor = 2, number of clustering centres;
- max_leaf_size = 10, threshold for the iteration.

Consequently the search over the tree is performed by the *tree_search* algorithm with the following input parameters:
- T = tree to be examined;
- d = query descriptor chosen from the second image;
- K = 3, number of neighbours to find;
- L_{max} = 150, maximum number of examined nodes.

The neighbours found by the algorithm are visualised in Figure 15.

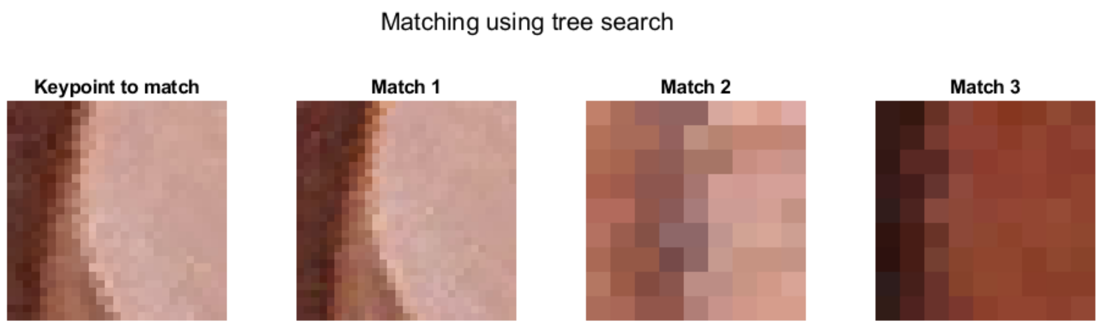

Figure 15: matching using tree search.

In order to have a more comprehensive view of the results, in Figure 16 we represent the descriptors matches: the feature to match is compared to the features found by the two algorithms.

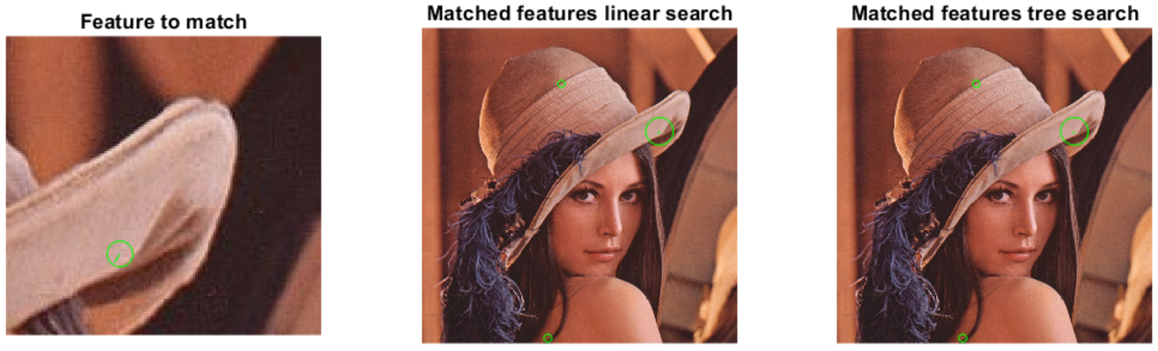

Figure 16: features found.

The most evident result is that three features of three are found by both the algorithms, and in particular one of them, that is the closest one, is the correct visual match.
It may happen that the closest descriptor to the query point is not the correct visual match, as we can notice by the result of the linear search run with a different query point (Figure 17).

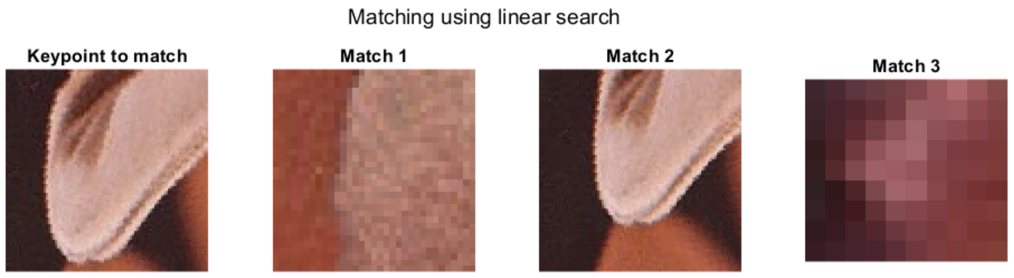

Figure 17: closest match and correct visual match.

The same could happen with tree search but, since it is a probabilistic algorithm, it may not find the closest point with respect to the Euclidean distance as we can see in Figure 18 in which the closest point found is the correct visual match though, in practice, it is the second-closest descriptor.

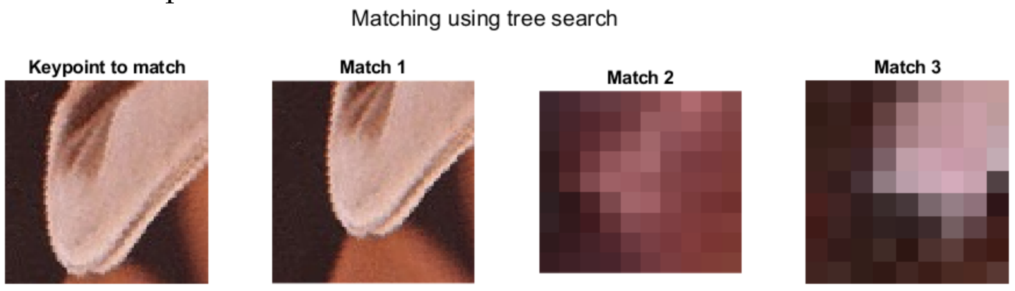

Figure 18: comparison with tree search.

In conclusion we can state that the tree search gives a good matching quality requiring less time than the linear search. The matching quality could be further improved augmenting the L_{max} parameter, so the number of examined nodes, sacrificing the speed.

Conclusion

The purpose of our project was to implement the fast SIFT matching based on priority search of hierarchical clustering tree. From our result we deduced that the time required to build a tree is not negligible so, if we need to perform few number of searches, the linear search is still the better choice. Anyway, if we deal with huge datasets and we have to perform multiple searches, the tree search method offers a faster alternative. Its performances strictly depend by the branch factor used in tree creation: higher branch factor offers faster results but it may sacrifice the precision. This behaviour is due to the fact that at each level of the tree we process the whole dataset to create the clusters. If we set an high branch factor the tree becomes larger and shorter: the less levels we have, the less we have to process the dataset and so the faster the process is.
This method offers a good alternative to the linear search SIFT matching, especially for some use. As we said before, the tree search is useful if we have to deal with large dataset and if we have to perform an high number of searches on that dataset. For this reason it does not suits well for real time application, especially if it needs to build the tree.

References

[1] : *Marius Muja and David G. Lowe. 2012. Fast Matching of Binary Features. In Proceedings of the 2012 Ninth Conference on Computer and Robot Vision (CRV '12). IEEE Computer Society, Washington, DC, USA, 404-410;*
[2] : *Oxford University Dataset: http://www.robots.ox.ac.uk/~vgg/data/data-aff.html;*
[3] : *David Lowe's SIFT keypoint detector: https://www.cs.ubc.ca/~lowe/keypoints/;*
[4] : *Priority Queue algorithm: https://github.com/guyrt/MatlabPriorityQueue*
[5] : *David G. Lowe. 2004. Distinctive Image Features from Scale-Invariant Keypoints. International Journal of Computer Vision, 2004.*