Computer Vision & 3D Graphics

Final project - Fast SIFT Matching

Enrico Polo, 1124766

Assignment

Starting from the paper:
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. pdf
implement the described FAST SIFT matching strategy using some of the datasets.
SIFT software is available at: *http://www.cs.ubc.ca/ lowe/keypoints/*
Datasets to be used are available at: *http://www.robots.ox.ac.uk/ vgg/data/data-aff.html*
Programming language: **MATLAB** or C++.

Introduction

The paper propose a fast method to find matches between features obtained running some kind of descriptor in a set of images. In fact, for large datasets, comparing each feature of an image with all the ones of the others (linear search) becomes a bottleneck in most applications.
For this reason authors propose an algorithm that performs a hierarchical decomposition of the search space by successively clustering the features to be compared and constructing a tree in which every non-leaf node contains a cluster center and the leaf nodes contain the features that must be compared.

1 Building random hierarchical trees

The first operation is to cluster the features set in which we are looking for a match. In our implementation this operation is done by the function `clustering.m` whose beahviour is described in Alg. 1.

```
input : features dataset (feat), indexes of the features to be clustered (idx),
        max features number in a leaf (leaf_size)
output: random hierarchical tree (part_tree)
1  if row number of feat ≤ leaf_size then
2      create leaf node with idx ;
3  end
4  else
5      centers ← select branch random points from idx ;
6      part_tree ← centers ;
7      $ind_i$ ← indexes of features in cluster i ;
8      for $i$ ← 1 to branch do
9          temp ← clustering(feat, branch, leaf_size, $ind_i$) ;
10         if temp ≠ ∅ then
11             part_tree ← merge part_tree and temp;
12         end
13     end
14 end
```
Algoritmo 1: `clustering.m` pseudocode.

The function has three parameters to be chosen:

1) `branch`: the branch factor that is the number of clusters to be built at each iteration;

2) `leaf_size`: the maximum number of elements in a leaf;

3) `idx`: an optional parameter that represents the indexes of the features we want to cluster, if not set our function considers all features to be clustered.

Regarding the input/output structure and the parameters values we have that:

- the input of our algorithm must be some kind of features set organized row-wise (one descriptor per row). In particular in our tests we have considered sift descriptors returned applying the provided function `sift.m` to each of considered images. So in our case we have an m by 128 `feat` matrix as input, where m is the number of features found.

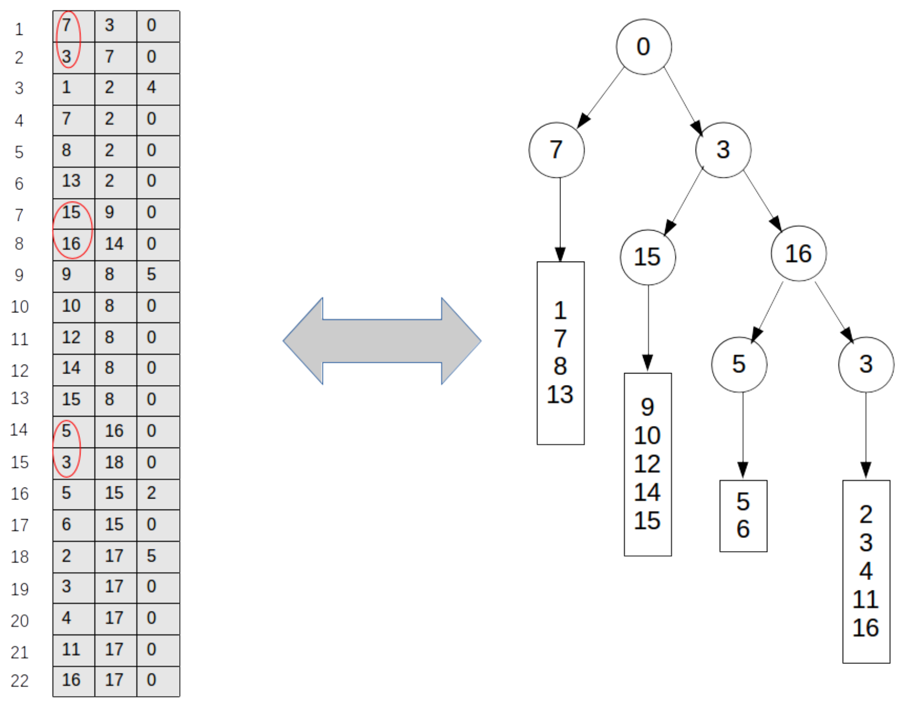
- `leaf_size` must be at least equal to `branch`. Otherwise, if we look at Alg. 1, we can see that step 5 fails if we don't have at least `branch` elements to be clustered.

- `idx` must be a column vector and its size must be less than or equal to the features number (m).

- The output is an h by 3 matrix where:
  - the value of h depends on the parameters choice and on the (random) selection of the cluster centroids at each iteration. Of course h is greater than or equal to m.

- The first column contains the indexes of the considered feature (i.e. if we find value 4, we refer to the feature stored in the 4th row of `feat`).
- The second column contains the index (row of part_tree) in which the first child node is stored, in the case of non-leaf node; otherwise it doesn't provide any information.
- The third column is set to zero for each element except for the fist element in a leaf node, in which is stored the effective leaf size.

We can see a simple example of this structure in Fig. 1, where all non-leaf nodes are highlighted.



1	7	3	0
2	3	7	0
3	1	2	4
4	7	2	0
5	8	2	0
6	13	2	0
7	15	9	0
8	16	14	0
9	9	8	5
10	10	8	0
11	12	8	0
12	14	8	0
13	15	8	0
14	5	16	0
15	3	18	0
16	5	15	2
17	6	15	0
18	2	17	5
19	3	17	0
20	4	17	0
21	11	17	0
22	16	17	0

Figure 1: Tree example: `branch` = 2, `leaf_size` = 6, `feat` = 16x4 random values.

In a first implementation we build the tree structure using only two columns (to save storage space) but, for improving performances in traversing the tree, we prefer using this last implementation. As suggested in the paper, we may need to build more than one tree to improve the precision and the number of matches returned in our search. For this reason we implement the function `random_hierarchical_tree.m` that builds n trees using the function described above, adds the root, updates the pointers and finally returns `tree` that is a list of "n" `part_tree` structures.

2 Finding matches

In this section we want to describe how we navigate each of the trees stored in `tree` in order to find matches.
The function that implements the search algorithm is `r_search.m` and its behaviour is described in Alg. 2.

```
input  : features we want to match (Q), features in which we want to search
         (ft), trees built as in the previous section (T), minimum number of
         comparisons we want to do (Lmax), number of approximate neighbours
         we want to return (ret)
output:  ret nearest neighbours to each feature in Q (K)
initialize K;
for $m \leftarrow 1$ to row number of Q do
    for $w \leftarrow 1$ to number of trees in T do
        initialize PQ, R, L;
        TraverseTree(T,0);
    end
    if $PQ \neq \emptyset$ and $L < Lmax$ then
        sort(PQ) ;
        sz $\leftarrow$ 1;
        while $PQ \neq \emptyset$ and $L < Lmax$ do
            w, N $\leftarrow$ PQ(sz);
            TraverseTree(T(w),N);
            sz++;
        end
    end
    unique(R);
    K(m) $\leftarrow$ ret top elements of R;
    remove_wrong(K(m));
end
function TraverseTree(node,res_idx);
extract first_child_node;
if first_child_node is a leaf then
    R $\leftarrow$ linear search in ft(leaf_indexes);
end
else
    compare Q( w ) with all child_nodes;
    res_idx $\leftarrow$ nearest child_node index;
    PQ $\leftarrow$ all others child_nodes indexes;
    TraverseTree(T(w),res_idx);
end
```

Algoritmo 2: `r_search.m` pseudocode.

The search starts traversing the tree selecting at each step the closest node to the

query feature. Then it saves all the others child nodes (in the same level) in `PQ` and recursively explores the subtree that corresponds to the selected node. When a leaf node is reached all elements in the leaf are linear compared to our query and inserted in `R`. After each tree has being traversed once, if the procedure has not checked at least `Lmax` features, the algorithm proceeds sorting the elements in `PQ` in ascending order, from the closest unexplored node to the furthest. Then the search restarts from the first element in `PQ` until `Lmax` comparisons has been done or the `PQ` empties. Finally the `ret` best matches in `R` are returned.
Regarding our specific implementation we want to explain more in detail some aspects:

- to traverse the tree (built as in the previous section) we consider the `first_child_node` (the value stored in the second column of the tree array) of the node we are considering and:
  - if its third component (`lgth`) is different from zero, then its a leaf and we need to linear search for a match from `first_child_node` to the next `lgth` -1 elements in the array (proceeding row-wise), that are all the elements within the reached leaf;
  - if it's zero we are in a non-leaf node so we compare the query with the next `branch` -1 nodes and `first_child_node` itself, select the best one and restart the traversing from there (reiterating the traversing procedure starting from the best match index). Other indexes are stored in `PQ`.
- Instead of implementing a priority queue, as suggested in the paper for `PQ` and `R`, we prefer to store our data (indexes and distance from the query and the index of the considered tree in case of `PQ`) in two arrays and to sort them, as described before, after the trees has been explored once. In particular we sort `PQ` only if needed (L<`Lmax`) and we sort `R` before selecting the best `ret` matches, removing duplicates using the Matlab function `unique.m`. In this way we simulate the behaviour of the described priority queues with an implementation that is less time demanding using Matlab. In fact we tried to use a Matlab library for priority queues but it was very slow to execute. We try also to use a C++ library [1] that implements that structure in order to have a more efficient implementation but we find out that, for how it's implemented, to store large indexes we need to allocate a large amount of memory wasting a lot of resources and time in case of large datasets. Although for the smaller ones we reach good results also using this library.
- To improve performances in computing the euclidean distances between query and each cluster center (in case on non-leaf nodes) we use a C function (`sqdist2.c`) from the library *Utilities for MEX files*[2];
- To be faster in computing the minimum euclidean distance (in case on non-leaf nodes) we use another C function (`minkmex.c` from *MinMaxselection*[3] library)

[1]https://it.mathworks.com/matlabcentral/fileexchange/24238-priority-queue–mex-c++-
[2]https://it.mathworks.com/matlabcentral/fileexchange/26825
[3]https://it.mathworks.com/matlabcentral/fileexchange/23576-min-max-selection

because it appears to be faster than the built-in Matlab function and it's called a lot of times during the search;

- To perform the linear search within the leaf nodes we use the same method implemented in the `match.m` function provided with the sift software in order to compare the performances between the two approaches in which the linear search is implemented in the same way. In particular we implement also the check to remove some wrong matches in `K`. We check the second best match returned by our procedure using the same `distRatio` parameter (`distRatio` = 0.6) of the provided matching function.

3 Tests and results

In this section we evaluate the performances of the proposed method against the linear matching strategy, in therms of search precision and execution time. Moreover we want to evaluate how different parameters (`Lmax`, `branch`, `leaf_size` , `n`) affect matching performances.
Before starting to discuss our results we want to briefly describe the setup we used and that we provided in the folder *working_directory*:

- the folder *datasets* contains all the datasets assigned for the tests, one per subfolder (*bark*, *bikes*, *boat*, *graf*, *leuven*, *trees*, *ubc*, *wall*);
- the folder *descriptors* contains the data for each dataset (features descriptors, descriptors locations and images) obtained running the script `sift_extract.m`;
- the folder *MatlabPriorityQueue-master* contains the Matlab priority queue library that can be used in place of our final implementation uncommenting the lines marked with "%mat_pq" in the file `r_search.m` and commenting the equivalent sections in the final version (i.e. initialization, restart traversing...);
- the folder *myqueue_1.1* contains the C++ priority queue that can be used in place of our final implementation as described in the previous point uncommenting lines marked as "%cpp_pq" (before running it there's the need to compile the function using the `mex` Matlab compiler);
- the folder *siftDemoV4* contains sift extraction functionalities and the linear matching implementation we use in our tests (`match.m`). All functions that involve running `sift.m` must be launched from this folder;
- the folder *utilities* contains the other two C (or C++) functions (`minkmex` and `sqdist2`) that needs to be compiled with `mex` before running the code;
- the main folder contains all the functions described before and the functions we used to do our tests: `test.m` and `match_mine.m`;

- finally there's also `print_match.m`, a function to visualize matches between two images of the same dataset.

The function `match_mine.m` takes as inputs the descriptors of two photos (as usual organized row-wise) and all the parameters to be set in `r_search.m` function. It performs the search using this function and returns the number of matches, the match vector, the execution time and the CPU time. More precisely it returns execution and CPU times divided by the number of considered trees to simulate the behaviour (and performances) of a parallel implementation. It performs also the classical linear search and returns the same values also for this approach.
Instead of using precomputed descriptors the function can accept as inputs two image files and, starting from these, it can compute sift descriptors and their locations. In this case there is also the option to visualize directly the results obtained using our matching strategy uncommenting the last lines in `match_mine.m`. This option generates `ret` images in which are represented respectively the first best matches, the second ones and so on.
The script `test.m` simply executes the matching function on an entire dataset, where each image is compared to the next one.
The function `print_match.m` takes as input two images, their descriptor locations and a match vector computed with `match_mine.m` that can be either the one obtained with our approach or with the linear one and visualizes the results of the considered matching strategy.
For example in Fig. 2, we can see the correspondences obtained with random hierarchical search between the first and the third photo of *bikes* dataset.

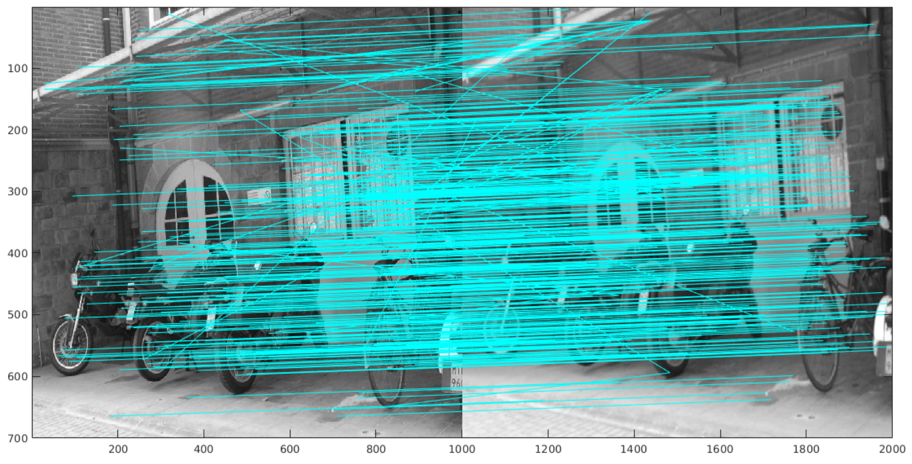


Figure 2: Matching example: bike dataset, `n` = 1, `Lmax` = 2, `branch` = 16, `leaf_size` = 20.

We can see that there are some wrong matches but the most are correct. Of course, we can improve the performances setting correctly `r_search.m` parameters instead of choosing them randomly like in this example. This will be the focus of our next section.

3.1 Accuracy

To evaluate the accuracy of our algorithm with a specific configuration we use three main metrics returned by our `text.m` function:

$$\texttt{perc\_matches(i)} = \frac{|M_{hier}(i)|}{|M_{lin}(i)|} \tag{1}$$

where $|M_{hier}|$ is the number of matches found with our approach, $|M_{lin}|$ is the same quantity for the linear one and i is represent the iteration number of our procedure. For example $i = 1$ represents the matches found between 1^{st} and 2^{nd} photo of a particular dataset, $i = 2$ between 2^{nd} and 3^{rd} and so on.
The second quantity is:

$$\texttt{match(i)} = \frac{|M_{hier}(i) \cap M_{lin}(i)|}{|M_{hier}(i)|} \tag{2}$$

that represent the percentage of correct matches found (assuming all linear matches are correct) with respect to all matches returned by our procedure.
We evaluate also:

$$\texttt{perc\_right\_matches(i)} = \frac{|M_{hier}(i) \cap M_{lin}(i)|}{|M_{lin}(i)|} \tag{3}$$

that is the percentage of correct matches with respect to the total returned with linear search.
The function returns also the overall performances for each dataset by computing the mean value for each of these quantities.
In this section we don't test each dataset but we use *bikes* and *trees* that are respectively a medium sized dataset (from $\simeq 10^3$ features per image to $\simeq 10^2$) and a large one (from $\simeq 10^4$ features per image to $\simeq 10^3$).

First of all we have seen in our tests that `branch` parameter doesn't affect our matching accuracy. To be precise, remembering that it's a lower bound for the minimum leaf size, it influences indirectly the search precision. Nevertheless, if we choose a `leaf_size` large enough to allows us changing the branching factor as we like, we can see that there is no difference between a small or a large one.

Another observation is about the metric (2). Thanks to how we implemented the random search, checking the correctness of a match returned comparing it with the second one (as in linear approach), we reach always a value $\geq 90\%$ for this metric, regardless of the used setup. For this reason (1) or (3) metrics give us quite the same informations about the algorithm behaviour. However we report both quantities in our analysis.

The first parameter we want to analyze is `Lmax`. In order to do this we use a single tree (`n` = 1), fixed `branch` value equal to 16 and fixed `leaf_size` = 20.

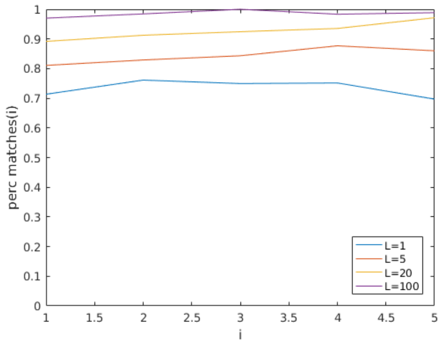


Figure 3: Dataset: bikes.

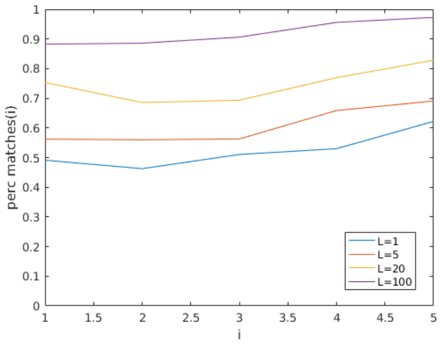


Figure 4: Dataset: trees.

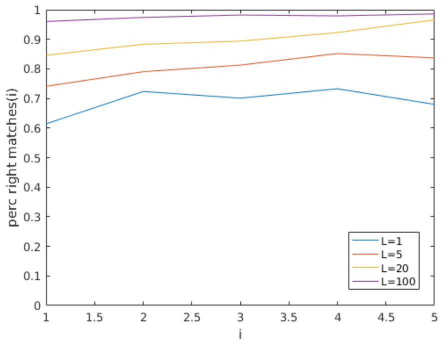


Figure 5: Dataset: bikes.

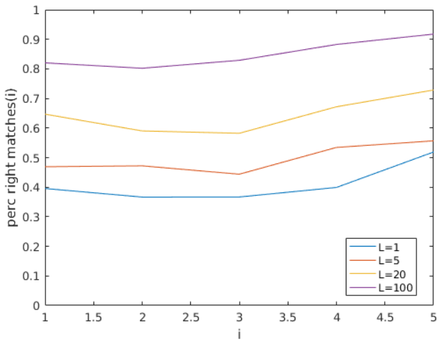


Figure 6: Dataset: trees.

We can notice that `Lmax` highly influences the search precision. As we expected, higher the value of `Lmax`, higher is the number of matches returned by our algorithm. Furthermore we see that the value of `Lmax` depends on the size of the considered dataset. In fact, for example, to reach the same results obtained in *bikes* with `Lmax` = 1, we need to set `Lmax` = 20 in *trees*.

We want now to see how `n` influences our procedure. To do that we set `Lmax` = 1, `branch` = 16, `leaf_size` = 20, obtaining the following results.

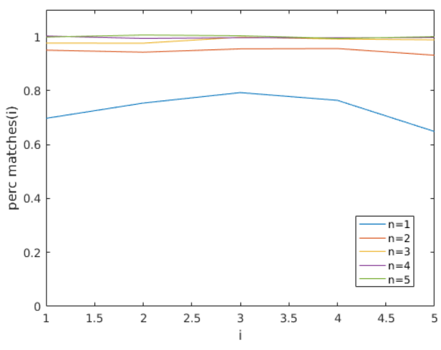


Figure 7: Dataset: bikes.

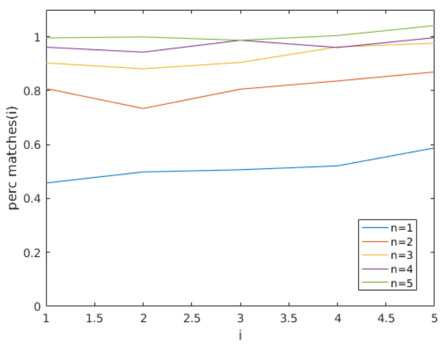


Figure 8: Dataset: trees.

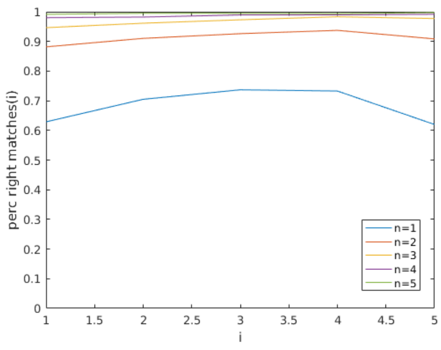


Figure 9: Dataset: bikes.

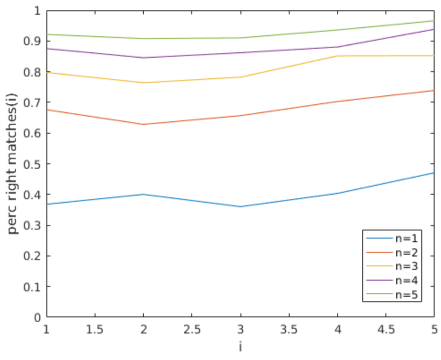


Figure 10: Dataset: trees.

Also in this case as `n` increases the precision increases and the results depends on the dataset size. We can see that for the smaller dataset 2/3 trees are enough to reach very high precisions (defined as above). Adding more trees in our computation doesn't increase significantly accuracy performances. Conversely, for the larger one, we see that adding more than three trees helps us to reach better results. Also in this case we can see that improvements obtained adding a tree are significant for `n` = 1,2 or 3; then the accuracy increases but our algorithm reach a sort of "saturation" and there's no significant benefit in using more and more trees.

Similar observations can be done evaluating the impact of `leaf_size` by setting `Lmax` = 1, `n` = 1, `branch` = 16 obtaining the results reported in the following images.

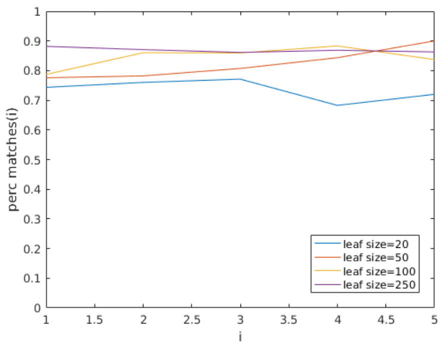


Figure 11: Dataset: bikes.

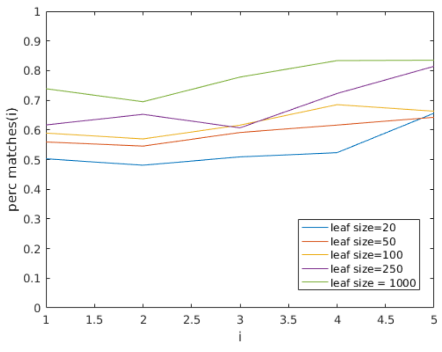


Figure 12: Dataset: trees.

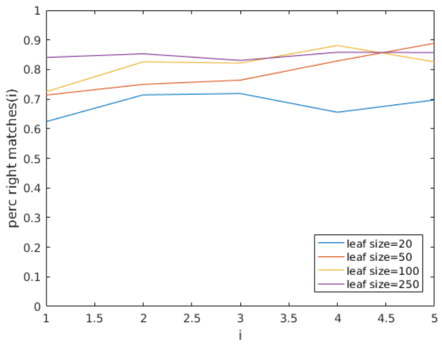


Figure 13: Dataset: bikes.

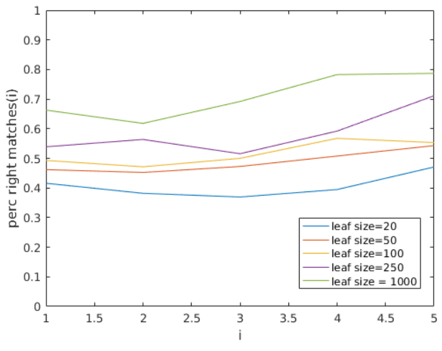


Figure 14: Dataset: trees.

It's now clear how parameters influence the matching accuracy: increasing each of them the accuracy improves. In the next section we want to find a good overall configuration in order to reach a good accuracy but taking care of time performances, trying to find the best trade-off between the two aspects.

3.2 Execution time

To evaluate the performances in therms of execution time we consider the quantity:

$$\texttt{speedup(i)} = \frac{t_{lin}(i)}{t_{hier}(i)} \tag{4}$$

where $t_{lin}(i)$ and $t_{hier}(i)$ are the linear and hierarchical searching speeds measured by tic-toc functions in Matlab as described before (in case of hierarchical search the quantity is divided by the number of trees used in our search). This quantity tells us how faster

(or eventually slower) is our algorithm with respect to the linear search. Also in this case we can consider the overall performances considering $\sum_i t_{lin}(i)$ at the numerator and $\sum_i t_{hier}(i)$ at the denominator.
Before reporting and commenting the results in the considered datasets, we want to report some observations regarding the optimal configuration for each dataset:

1) a good rule to reach high accuracy and shorter execution times is to increase `n` instead of `Lmax`. In fact we can build and navigate trees in parallel while, increasing `Lmax` to reach the same number of comparisons, increases also the computation time.

2) Increasing `n` instead of `leaf_size` it's also a good rule improve the speed of our algorithm. In fact when our procedure reaches a leaf node it linear compares all features inside the leaf.

3) `branch` plays an important role in determining the execution time (conversely to how we have seen in accuracy evaluations).

4) We need to find a good trade-off between the building-tree time and the traversing-time and this, of course, depends on `branch` and `leaf_size`, together with the dataset size.

For this reasons in our tests we select `n` ≥ 2 and a low value for `Lmax`. In fact also with `Lmax` $= 1$ using at least 2 trees we have a very high number of matches returned (as can be seen in the previous section in Fig. 9 and 10). So we focused our attention to the correct setup for `branch` and `leaf_size` in different datasets. To better understand the situation we report in Fig. 15 the speedup values obtained in dataset *bark* ($\simeq 10^3$ features per image) for different configurations.

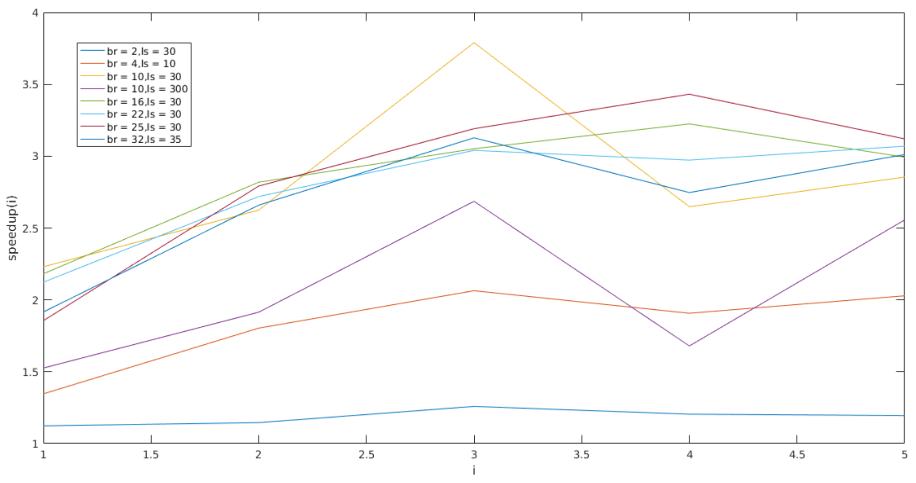


Figure 15: Dataset: bark. Speedup for different parameters.

We can notice that if `branch` is too small we don't have improvements with respect to linear search. In fact we need a lot of time for building the tree and also for navigate within the tree (it's deep). Moreover if we use a too large `leaf_size` value we obtain the same results. In fact we build the tree and navigate it very faster but on the other side we need to search in very large leaf nodes. In conclusion we need to reach the best possible results we need to find a branch factor large enough (with respect to the dataset size) to guarantee fast traversing time, but not too large, otherwise leafs nodes may become too large (and we loose the benefit of building a tree). On the other side we want leaf nodes to be small enough to take advantage from linear searching between all features, but not too small, otherwise we loose a lot of time building the tree (and cannot increase the value of `branch`). From the figure above we can see that for *barks* a good choice is $\simeq 30$ for `leaf_size` and from 10 to 25 for `branch`. Now we report the results obtained for all the considered datasets, considering the overall search speedup factors.

Dataset	n	branch	leaf_size	Lmax	perc_match	perc_right_matches	speedup
bark	4	25	30	2	0.9813	0.9645	2.8779
bikes	4	25	30	2	1.0033	0.9907	1.1645
boat	4	20	30	2	1.0017	0.9447	3.9768
graf	4	15	30	2	0.9874	0.9308	2.5890
leuven	4	24	30	2	0.9993	0.9888	1.1518
trees	4	24	30	2	0.9860	0.8966	6.1632
ubc	4	22	30	2	1.0147	0.9625	3.8212
wall	4	28	40	2	0.9622	0.9216	6.6506

From our tests and looking at the best results reported in the tab, we can make the following observations:

- for matching between features sets with $10^3 \div 10^4$ features per image, the optimal leaf size value seems to be around 30;
- for the same kind of datasets the best value of `branch` seems to depend on the specific data but in the interval [15,32] (otherwise we can notice a slowdown in our computation due to mainly to the clustering operation as we will see better in the following);
- the improvements with respect to the linear highly depends on the number of the features we want to compare: higher the number of features, higher the speedup;
- for small feature set ($\simeq 10^2$) we reach the same (or sometimes lower) execution time of the linear search;
- for the larger ones we have improvements that could reach an order of magnitude (in dataset *trees* between 2^{nd} and 3^{rd} images we reach a speedup factor of 8.4294);

- for datasets with variable feature sizes, it's a good idea to optimize the search for the larger ones, to get the best overall results, even if we are slower in comparing the smaller images (with respect to the linear search);

Finally we try to test our algorithm in a larger dataset in order to find a certain number of features within a huge set, like was done in the paper. To do that we create a matrix that contains all the descriptors of all our datasets.
In this way we built a $285015x128$ descriptor matrix in which we try to find different features subsets.
In the following table we report the results obtained: we have set `n` $= 4$, `Lmax` $= 2$ and have consider both the building tree time and the overall time of the hierarchical search (measured in seconds).

# searched features	br	ls	perc_matches	building time	hier. time	linear time
17354	20	150	0.9982	3.3182	9.6282	794.8604
11746	24	150	0.9977	3.3207	7.5607	535.9329
1397	16	500	1.0000	2.2373	3.0403	63.8551
100	10	500	1.0000	2.3052	2.3712	4.8968

We can see that the speedup factor in this case highly depends on how many features we want to search: we have a speedup factor of about 80 for the largest subset considered and a speedup factor of about two for the smallest one. Also in this case we have to find a trade-off between time to build and traverse the trees, in particular:

- for large features numbers it's better to spend more time for building the trees, obtaining small leafs, in order to be faster in linear searching within them;
- conversely, for small features numbers, it's better to save time in building the trees rather than traversing them;
- analyzing more in detail how our algorithm works (using the Matlab `profiler` function) we see that building the trees for such features number (285015 descriptors in total) it's the bottleneck of our implementation. This is evident for small number of features searched because as we can see our algorithm spends almost all the time in building our structures. In particular the most of the time is spent when features are compared with centroids to be clustered (using the Matlab function `pdist2`). In larger datasets the effect is less evident but if we try to increase the branch number in order to navigate faster the tree and improve the performances, we find that the building process becomes a bottleneck also in this situation.

In conclusion we can say that proposed approach it's faster than the linear one, especially for large datasets. Results can be improved (in therms of speedup) optimizing the implementation (i.e. finding a faster method than `pdist2` to compare features with centroids, eventually accepting some degradations in therms of searching precision as in FLANN), or trying to compare a lower number of features (i.e. without searching in all the leaf elements once reached, but only in the first ”K” features or in ”K” random features within it). Another idea should be to use a metric faster to compute (i.e. correlation) to compare features. Nevertheless we can appreciate good results also using this basic implementation.