Computer Vision - Final Project
Tree Detection

Davide Dravindran Pistilli

22nd June 2020

1 Introduction

The purpose of this project is to create a program that is able to detect and locate all major trees in a given image. The process is organised in three main phases:

- **segmentation**: the image is segmented to identify the areas that have to be analysed;
- **classification**: the most relevant segments are analysed through a Bag of Words vocabulary in order to check the presence or absence of trees;
- **growth**: the candidate regions resulting from the previous step are enlarged in order to envelop the entire tree.

2 Training

2.1 Dataset

The Bag of Words approach requires a training phase to be performed to enable detection. This process uses as input a text file that specifies all training images with the trees they contain, if there are any (*Listing 1*).

```
1 FILE BW_1.jpg
2 1991,908,781,1606
3 FILE BW_2.jpg
4 24,325,2243,2979
5 FILE BW_3.jpg
6 FILE BW_4.jpg
```

Listing 1: `dataset2.txt`.

A rectangle specified by four parameters (x and y coordinates of the top-left corner, the width and the height) is defined for each tree. The training dataset used for this project contains 246 images, and is available on Google Drive[1]. All images and the trees chosen for training can be viewed through a python program, `DatasetVisualiser.py`.

2.2 Training Program

The training phase can be run with a command structured as specified below:

```
./Prj train ../dataset2/dataset2.txt ../dataset2/
```

where the arguments are as follows:

- **train**: special keyword that tells the program to train the Bag of Words data;

[1] `https://drive.google.com/file/d/16PJH2uVHWV3S7zXArRMKbwitqt5M1Y9y/view?usp=sharing`

- **../dataset2/dataset2.txt**: configuration file for the dataset;
- **../dataset2/**: root folder for the dataset images.

The actual training commences after the configuration file has been parsed. First of all, the program computes SIFT features from suitable areas of the image. If the image contains one or more trees, features are computed for each one in their respective rectangles. If there are no trees, however, features are computed on each node of a quad-tree that has the whole image as root (*Listing 2*).

```
157    // N-ary tree used to segment an image.
158    template<int Children, int Depth>
159    class ImagePyramid
```

Listing 2: `include/utility.hpp`

A depth of 2 provides an adequate number of views on the image while avoiding very specific details, such as individual forest trees. Note that all images are resized to a fixed resolution.
Completion of this step yields two distinct sets of features: tree features and non-tree features. These sets are then separately clustered to produce the corresponding Bag of Words vocabularies.
However, a reference histogram for each class is also required to classify an image. Hence, the final training step reloads each training image (images are loaded only when needed to reduce memory usage) and computes its BoW histograms on the same areas as before, using the appropriate vocabulary. To avoid using weak areas of the image, a minimum number of features is required for a histogram to be computed and considered in the average (*Listing 3*).

```
36        // Minimum number of features for training.
37        static constexpr int min_train_features{ 128 };
```

Listing 3: `include/TreeDetectorTrainer.hpp`

At the end of the process, the two vocabularies and the average tree and non-tree histograms are saved to a file named `trees.xml` in the dataset root folder.

3 Detection

3.1 Start-up

The detection phase can be run with a command structured as specified below:

```
./Prj detect trees.xml testdata/Figure_1.jpg
```

where the arguments are:

- **detect**: special keyword that tells the program to detect trees in an image;
- **trees.xml**: file containing the trained vocabularies and histograms;
- **testdata/Figure_1.jpg**: image on which detection is performed.

After loading the input image, the first step is to resize it to a fixed resolution (*Listing 4*), as done during training, in order to have a normalised initial condition.

```
34    // Image dimensions.
35    constexpr int img_width{ 2048 };
36    constexpr int img_height{ 2048 };
```

Listing 4: `include/utility.hpp`

3.2 Pre-processing

To get meaningful results from the BoW classification, relevant areas of the image that will later be analysed must be identified. This is achieved by segmenting the image using the watershed algorithm. All resulting segments pass on to the analysis step.
Two filters are applied before segmenting the image: a bilateral and a Gaussian filter. The bilateral filter smoothens the image while preserving its edges. The Gaussian filter, then, smoothens the whole image uniformly, further reducing the effect of small high contrast areas (such as leaves).
After these filters have been applied, the actual segmentation can be performed (*Listing 5*).

```
310 void Image::segment(const SegmentationParams& params)
311 {
312     Image edgeMap{ mat_ };
313     edgeMap.canny(params.cannyTh1, params.cannyTh2);
314     if constexpr (debug) edgeMap.display();
315     edgeMap.negative();
316     edgeMap.distanceTransform();
317     edgeMap.log();
318     edgeMap.normalise(0.0, 1.0, cv::NORM_MINMAX);
319     if constexpr (debug) edgeMap.display();
320     edgeMap.threshold(params.distTh, 1.0, cv::THRESH_BINARY);
321     if constexpr (debug) edgeMap.display();
322     edgeMap.connectedComponents();
323 
324     labels_ = edgeMap.labels();
325     setColourSpace(ColourSpace::bgr);
326     cv::watershed(mat_, labels_);
327 }
```

Listing 5: `src/Image.cpp`

The watershed algorithm requires a set of starting seeds from which to grow the regions. These should be the farthest points from the edges of the image, so the 'edge map' and then the 'distance transform' are computed. To obtain only the farthest points from the edges, all values are normalised and a threshold is applied. Then, each connected component is labelled differently from the others so that watershed considers each of them a separate segment.
Another approach was also attempted: instead of segmenting the image through an algorithm like watershed, fixed segments were used. These were obtained through the same quad-tree structure used for training, just with a higher depth. The overall results were comparable to the current solution and sometimes even better, but the execution time was up to three-fold longer in some cases. Due to this and some difficulties with the growing phase, the quad-tree approach was replaced with watershed.

3.3 Analysis

Depending on the image, watershed could output a huge amount of segments. The first analysis step is to choose only those, whose score is high enough to be considered part of a tree. The score for each segment is computed as shown in *Listing 6* based on its distances from the two classes: tree and non-tree.

```
194 double TreeDetector::computeScore_(double treeDist, double nonTreeDist)
195 {
196     if (treeDist < 0) return treeDist;
197 
198     return nonTreeDist - treeDist;
199 }
```

Listing 6: `src/TreeDetector.cpp`

These are the distances between the histogram of the segment (computed with the appropriate vocabulary) and the average histogram of the class. A negative distance means that the segment does not have a sufficient number of features.
The candidate segments are then sorted by decreasing score, and go through a fusion round where overlapping segments are fused together. All resulting candidates move to the growth phase.

3.4 Growth

Candidates usually do not envelop the entirety of a tree due to over-segmentation. This is solved by growing each one of them. For each candidate, all four sides of the rectangle are examined at each iteration, and the best one is chosen for expansion. If no areas are suitable, the growth is stopped, a fusion round is performed and the next candidate is processed. The results of this phase are the final trees that will be displayed.
This is the most demanding part of the program, since SIFT features are continuously computed and matched against two BoW vocabularies. Without it, however, many trees are detected correctly but become over-segmented with non-overlapping regions.

4 Results

4.1 Benchmark Dataset

The quality of the results is quite mixed: since trees can appear in many different scenes and illumination conditions, there is no parameter combination that is effective for all images. The chosen parameters aim at achieving acceptable results across as many settings as possible.
Figure 1 shows how clear and isolated trees can be detected, provided that they have enough features. The tree on the left, in fact, had too few features to be considered. Lowering the thresholds further resulted in many false positives and lower quality regions overall.

Figure 1: Figure_1.jpg

Figure 3 shows some false positives that survived the analysis phase due to their high scores (trees from 2 to 5). Tree number 6, however, expanded toward the wrong direction during the growth phase.

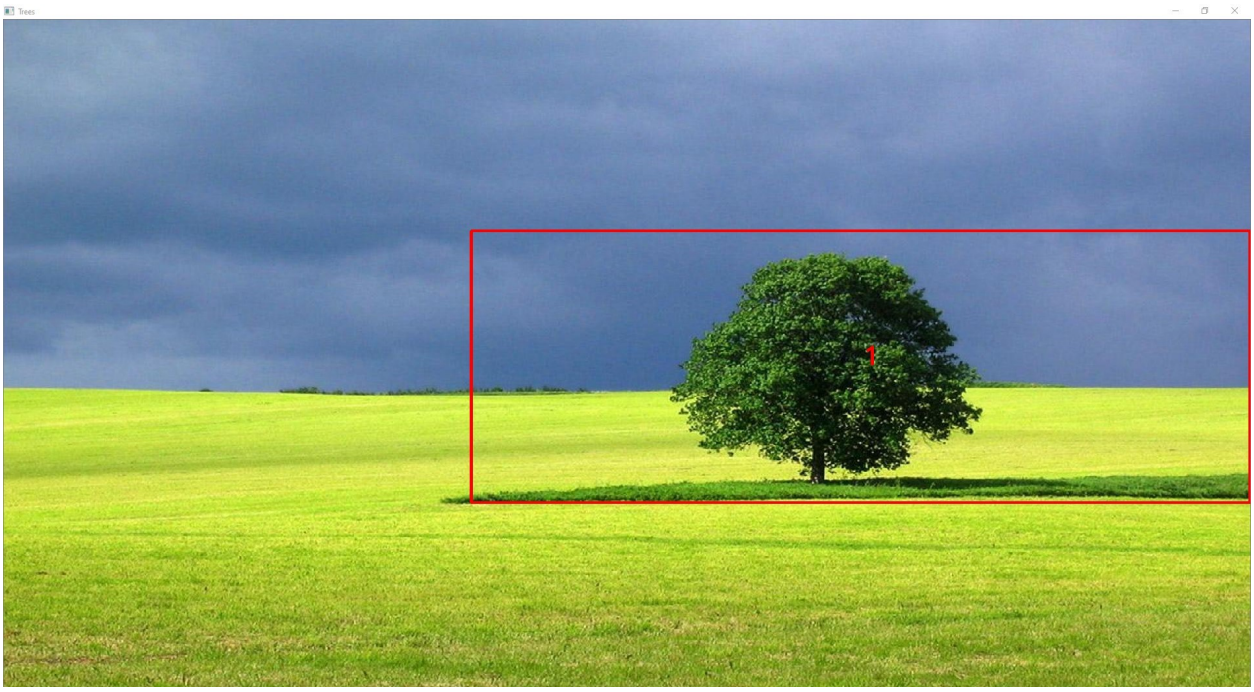

Figure 2: Figure_2.jpg

Figure 5 illustrates a situation where the growth phase is unable to compensate for the over-segmentation resulting from watershed: growing any of those two regions would have caused an excessive reduction in their score, so the algorithm ended.
Figure 6 is an example where the segmentation phase failed to provide small enough regions, so the entire image was analysed. As with the previous images, parameters could be tuned to enable detection at the expense of overall accuracy in the entire dataset.
There were no false positives in figures 7, 8 and 9, even implementing significant changes to the parameters. However, this is not always true.

4.2 Training Dataset

Some tests were performed on the training set, since the training phase did not cover the entire detection algorithm but just the creation of the BoW data required for classification.
Figures 10 and 11 show a couple of false positives, both related to rocky features. In some conditions, in fact, the program interprets them as part of a tree. Nevertheless, other rocky images are just fine (like *Figure 12*).
Figures 13, 14 and 15 show other results.

Figure 3: Figure_3.jpg

Figure 4: Figure_4.jpg

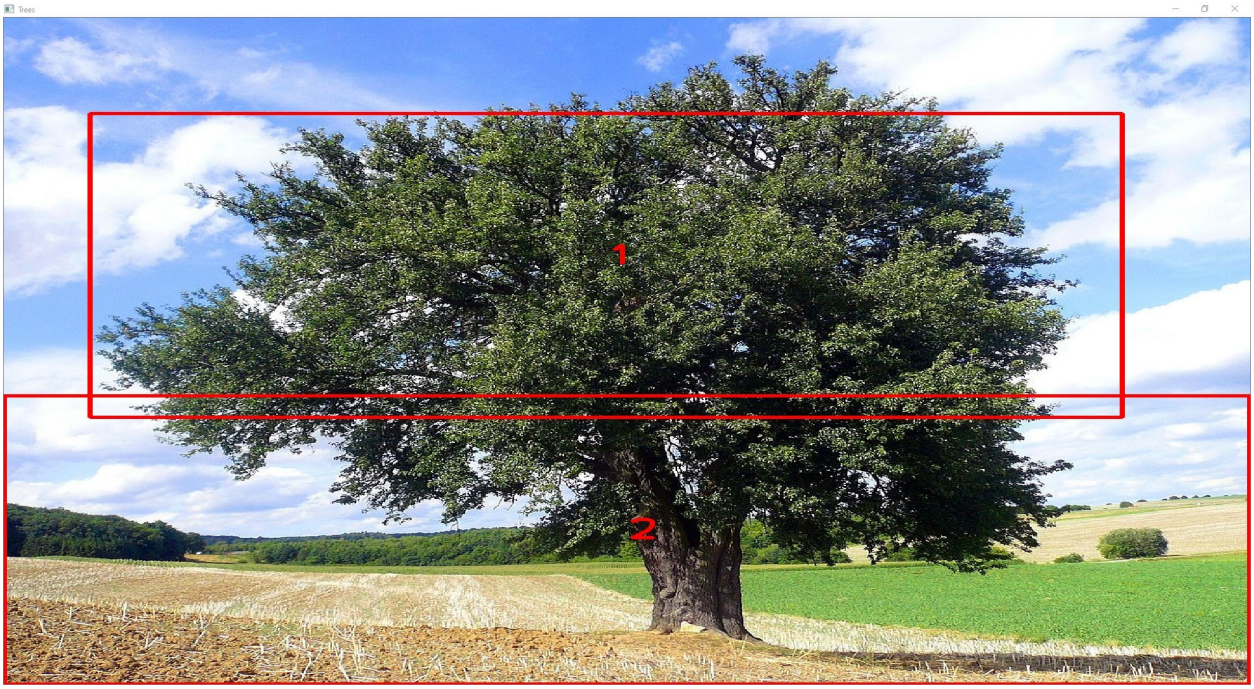

Figure 5: Figure_5.jpg

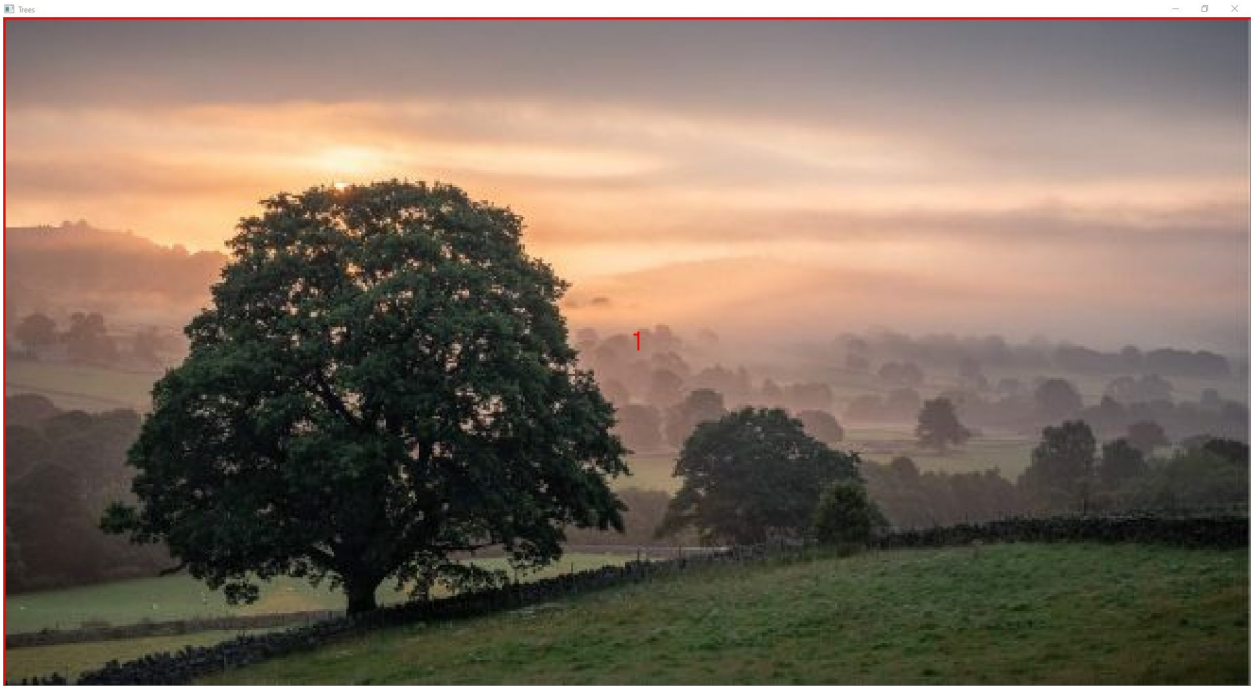

Figure 6: Figure_6.jpg

Figure 7: Figure_7.jpg

Figure 8: Figure_8.jpg

Figure 9: Figure_9.jpg

Figure 10: C_9.jpg

Figure 11: C_101.jpg

Figure 12: C_102.jpg

Figure 13: C_5.jpg

Figure 14: C_6.jpg

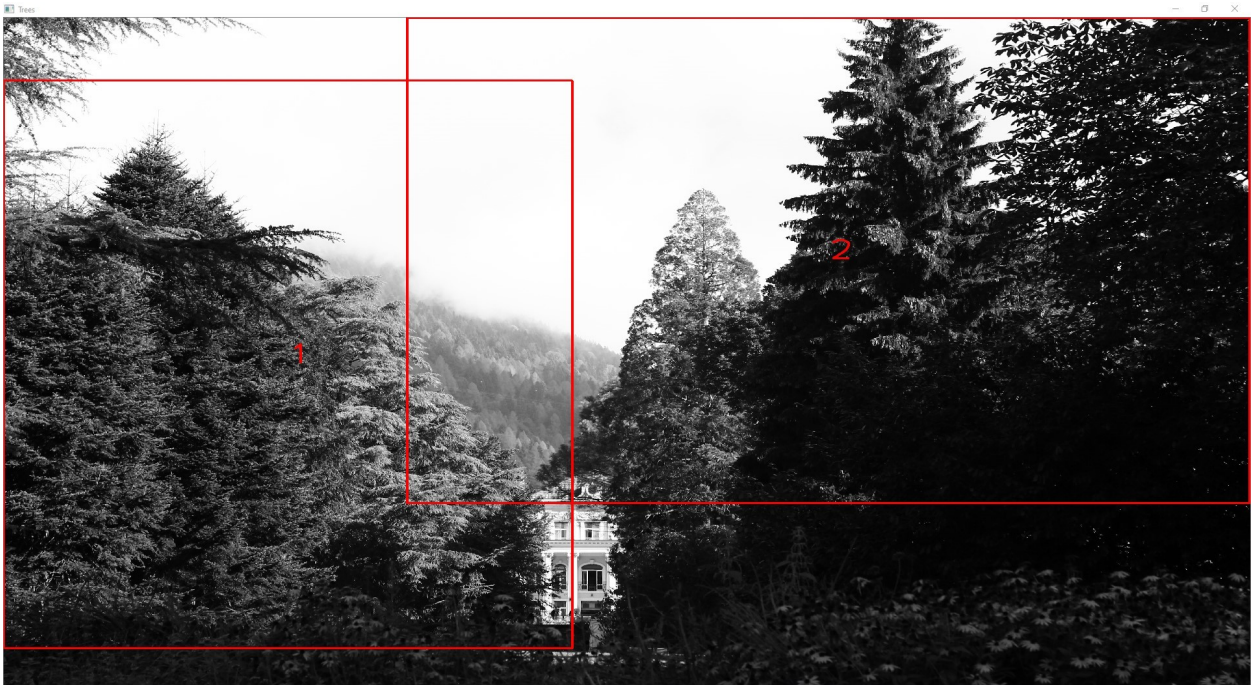

Figure 15: BW_6.jpg