CSC420H5 Final Project

Lang Qin, Student ID: 1005348982

Introduction

For this final project, I chose to implement Project 4 from the projects handout, which is the DVD cover recognition application. Given a test input image contains some DVD cover, the program will return the best-matched DVD cover from the database. And this program is generally in the same area as other objects recognition or retrieving applications, e.g. searching photos on the web for particular places. From lectures 8 to 10 we have learned that to match images we need to compute feature descriptors and find matching images with most similar descriptors. Normally, to perform matching, we need to match all reference descriptors to all descriptors in each database image. This method is not efficient and causes slow run-time for query lookup. To improve this, Nister and Stewenius come up with an efficient retrieval approach by implementing a vocabulary tree [1]. And this project is mainly to implement the approach introduced in their paper.

Vocabulary Tree

The vocabulary tree defines a hierarchical quantization that is built by hierarchical k-means clustering [1]. That is, at each level of the hierarchical tree, we define k different cluster centres on training data and split the data (descriptors) into k different groups. Each group forms a new node and each new node will also compute another k clusters. In this way, we can build up the vocabulary tree recursively to contain all the descriptors for each database image. Once the quantization is defined, we wish to determine the relevance of a database image to the query image based on how similar the paths down the vocabulary tree are for the descriptors from the database image and the query image [1]. By locating the image with best similarity, we can retrieve the best-matched image. Thus, instead of working on all the images from database, we can just work on some best-matched images. And that improves our query performance a lot. Although building a vocabulary tree requires some computational cost (depends on the size of the tree), the program can directly reload it once it is built. Also, it's easy and fast to accommodate new database images.

Tasks and Implementation

In this project I implemented 4 tasks which listed in the handout:

1. **Build vocabulary tree**
   The first step is to build the vocabulary tree. The tree is built based on two parameters, the branch factor k which specifies how many children nodes does each node have in the tree. And the depth limit L which specifies the maximum depth of the tree. Thus, a vocabulary tree

will have $\frac{k^{L+1}-k}{k-1}$ nodes. When performing clustering at each tree node, I used `Kmeans from sklearn.cluster` which will separate the descriptors into k clusters and each cluster will contain descriptors closest to it. I implemented the tree class to have hierarchical structure so that each node will have k child nodes and each child will also have children.
One important feature for nodes in the tree is the indexing scheme. That is, for each node, it has some descriptors. The node will store the indices of all the images which contain any of those descriptors.
Also, after the tree is built, the program will save it to disk. So when performing other recognition queries, the program can directly load it.

2. **Given test image, retrieve top 10 matches**
When performing a recognition query, we need to compare the similarity between query descriptors and database descriptors. I implemented the similarity as score introduced in the paper [1], which defines as $s(q,d) = ||\frac{q}{||q||} - \frac{d}{||d||}||$. Here q and d are query and database vectors respectively. q and d are defined as $q_i = n_i w_i$, $d_i = m_i w_i$ where n_i and m_i are the number of descriptor vectors of the query and database image, respectively. w_i is the weight factor and it is defined as $\log \frac{N}{N_i}$ where N is the total number of database images and N_i is the number of images with at least one descriptor vector path through node i.
For database vectors, we can pre-compute them as we build the tree. For query vector, we can compute it by traversing through the tree and get a path. All the nodes along that path will contribute to the vector. When creating nodes, we store critical attributes such as weights w_i so that it's easy for us to compute $q_i = n_i w_i$.
Now, we have database vectors which has dimension $N \times M$ where N is total number of database images and M is the total number of nodes. We also have the query vector which has dimension $1 \times M$. Thus, we can compute scores for each database image. The smaller score is, the better the match is. We retrieve top 10 matches by selecting 10 images with smallest scores.

3. **Perform homography estimation with RANSAC**
After we retrieve top 10 match images, we need to perform homography estimation with RANSAC on those images and the test image. Each estimation will return a homography transformation and corresponding number of inliers. Since I already did homography estimation with RANSAC in Assignment 4, I imported the code from it in this part. To be specific, I use 3000 iterations of RANSAC for each pair and distance threshold as 3.

4. **Retrieve the best DVD cover and visualize**
Among those homographies produced from 3), we choose the one with the highest number of inliers. And that is the best DVD cover image. In the end, we return the best DVD cover image and visualize the optimal homography transformation on test image. By visualizing, that is to draw a bounding box around the DVD cover located in the test image.

Program Structure & How to Run

The program has following structure:

```
- DVDCovers/     # Database folder contains all the DVD cover images.
    - dvd_01_name.jpg
```

```
    - dvd_02_name.jpg
    ...
    ...
- test/          # Test folder contains all the test images.
    - image_01.jpeg
    ...
    ...
- dvdCover.py    # The main pipeline program.
- homography.py  # All the functions relate to homography.
- vocabTree.py   # The implementation for vocabulary tree.
- script.sh      # Running script to perform recognition on all the test images.
```

To run the program,

```
python dvdCover.py [-r] [-k BRANCH_FACTOR] [-l DEPTH] FILE
```

A sample run would be:

```
python dvdCover.py -k 5 -l 5 test/image_01.jpeg
```

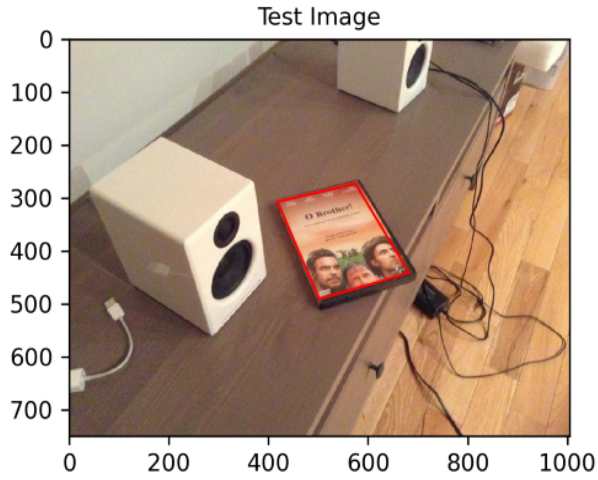
This will create a vocabulary tree with branch factor k = 5 and depth L = 5, and perform DVD cover recognition on `image_01.jpeg`. It will save the matching results as `image_01_dvd_cover.png` in current directory automatically. It will also save the vocabulary tree into disk as `vocab_tree.pickle`. So when we perform another recognition, we can just provide the test image name and the program will load the tree directly.
If we intend to reconstruct the tree, we can use `-r` flag.
For the script, it will first create a vocabulary tree with $k = 8$ and $L = 5$, then perform DVD cover recognition on all the test images in `test` folder.

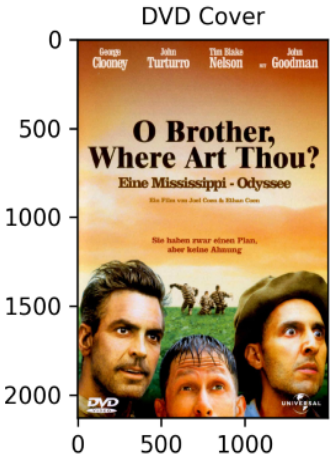
Tests and Results

There are 7 test images in the `test` folder. 6 of them can be found in `DVDCovers` folder and one of them cannot. Trying with different configurations, I found that building the tree with $k = 8$ and $l = 5$ will pass all the tests, which will create 37448 nodes in total. And one of the test results on `image_02.jpeg` would be:

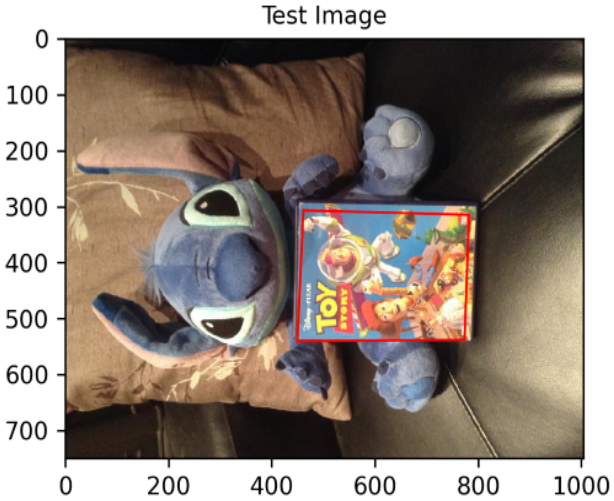
```
* Vocabulary tree file exists. Trying to load it...
* Performing DVD cover recognition on test image test/image_02.jpeg...
* The best DVD cover matched is DVDcovers\o_brother_where_art_thou.jpg
* Visualizing results...
* Saving results as image_02_dvd_cover.png
* DVD cover recognition complete.
```

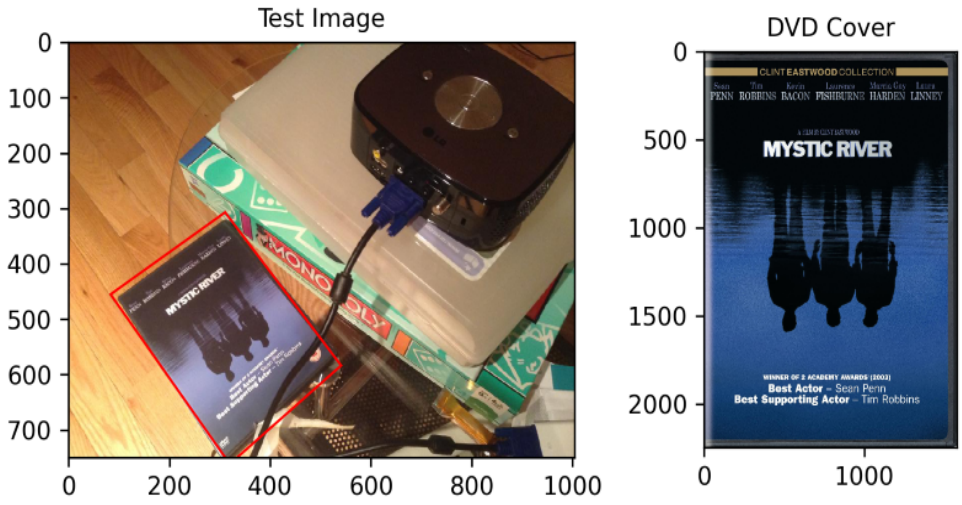
And the visualization is:





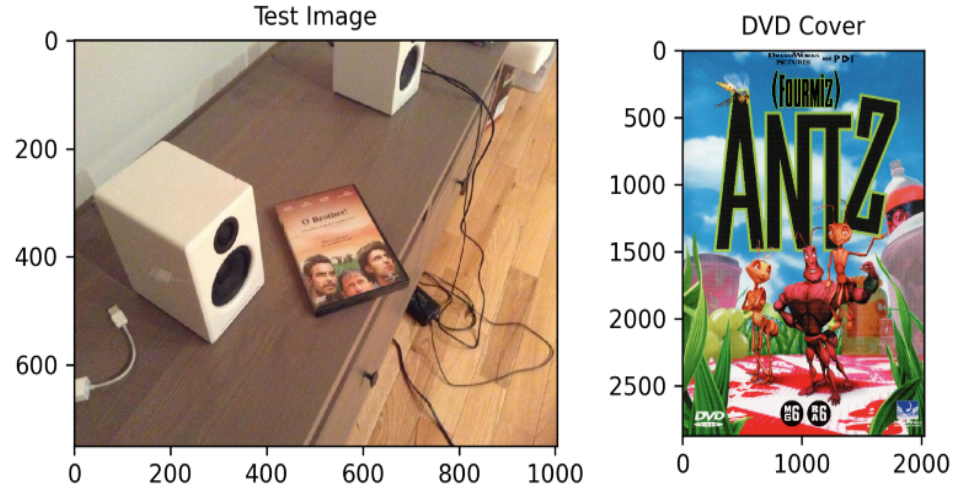
We can see that the correct DVD cover is returned and is located in the test image. And some of other test results would be:





However, if we build the tree with relatively small number of nodes such as setting $k = 5$, $l = 5$. Some of the tests will fail, for example, here is the result of running it on `image_02.jpeg`:

```
* Initializing and building vocabulary tree...
* Saving vocabulary tree...
* Performing DVD cover recognition on test image test/image_02.jpeg...
* The best DVD cover matched is DVDcovers\antz.jpg
* Visualizing results...
* Saving results as image_02_dvd_cover.png
* DVD cover recognition complete.
```



We can see that the program fails to compute the correct DVD cover. And this matches the results show in the paper [1], which is more tree nodes leads to higher overall accuracy.

Running Time

Intuitively, more nodes lead to larger computational cost when building the tree. For $k = 8$ and $L = 5$, it took me about 15 minutes to build the tree on my machine. However, once it is built, it will have a great improvement on query operations. For a single DVD cover recognition, the most time are spent on performing homography estimation with RANSAC on the top 10 images. For each image pair, it takes about few seconds to compute homography estimation. Thus, the overall run-time for a single query is less than one minute. If we use the original approach, which is to compute estimation on all the database images, it may take about 30 minutes to perform a single query!

References

[1] David Nister and Henrik Stewenius. Scalable Recognition with a Vocabulary Tree. In *CVPR, 2006*