Photomosaic Report

1. Analysis of the Problem

Photomosaic is a picture (usually a photograph) that has been divided into (usually equal sized) tiled sections, each of which is replaced with another photograph that matches the target photo.

1.1 Restatement of the Problem

- Provide a target image and create a photomosaic of the image. As a result, the picture is divided into several parts and each part is replaced with another similar picture.
- The mosaic image should be as similar as possible to the target image.

1.2 Establishment of ideas

1.2.1 Acquisition of mosaic images

It is necessary to create a large number of mosaic images, called tiles_images, in order to make the mosaic image as close as possible to the target image. Therefore, I used 245424 training pictures from the ImageNet dataset. Since pictures are divided into multiple folders, they will be merged into one folder, which is achieved by using the process_folder function. The schematic diagram of this function is shown in Figure 1.

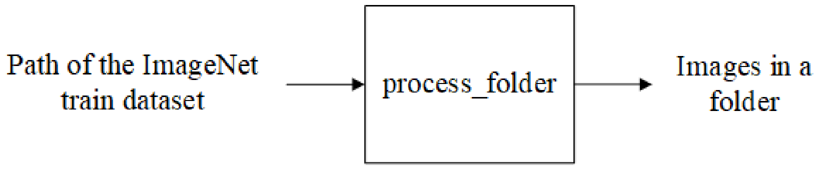


Figure 1: the I/O of process_folder function

1.2.2 Similarity measurement method

Distance between the average values of the three channels of the image in Euclidean space.

1.2.3 The creation of the mosaic image

In order to create the final mosaic image, a small area must be divided from the target image. The size of this area depends on the input parameter tile_ size. In other words, the number of pixels in every area to be compared is equal to tile_ size * tile_ size. A square area of this tile_ size can be used to compare the areas and tiles of the target image. Then, from top to bottom and left to right, traverse every region of the target image to find the most similar tile_image. Here is how the algorithm works:

Algorithm 1: The process of creating photomosaic

```
Input: target_image, tile_size
Output: mosaic_image
1. set mosaic_image equal to target_image
2. for every tile in target_image do:
     a) calculate the mean color of the tile and assign it to tile_mean
     b) set best_tile to -1 and min_distance to INT_MAX
     c) for i in range of 0 to the number of tiles in the image do:
        a)  calculate the Euclidean distance between the color of tile i and
            tile_mean and assign it to distance
        b)  If distance is less than min_distance, set best_tile to i and min_dis-
            tance to distance
    d)  resize the image at best_tile to tile_size
    e)  insert the image at best_tile into mosaic_image at the position of the cur-
        rent tile
3.  return mosaic_image
```

1.3 Directions for improvement

- It is possible to measure color similarity by using more robust methods, such as comparing color histograms between images, and other methods such as cosine distance and Manhattan distance.
- It is too slow to read tile_image in sequence. It may be possible to store information about each image in a hash table or tree. Then color matching will be more efficient.
- The reading folder contains pictures that can be read in multiple threads simultaneously.

2 Improvement methods

2.1 Save images with the red-black tree

The red-black tree is a binary balanced tree, which is able to efficiently complete the node search process. The following is a description of how the red-black tree works. In algorithm 2, the most similar node is found. Input parameter value is a weighted combination of three channel colors. In order to compensate for the fact that the human eye is least sensitive to red and most sensitive to blue, the red weight is set to 0.6, the blue weight to 0.1, and the yellow weight to 0.3. The algorithm for finding the nearest node in 1.3 is similar to that used for creating a mosaic.

Algorithm 2: The process of findClosest node using red-black tree

```
Input: value
Output: closest_node
1.  create a variable "closest_node" and initialize it to null
2.  create a variable "closest_distance" and initialize it to INT_MAX
3.  set the current node to the root of the tree
4.  while the current node is not null do:
    c)  calculate the distance between the current node's value and the input value
    d)  if the distance is less than closest_distance, update closest_node and clos-
        est_distance
    e)  if the input value is less than the current node's value, set the current node
        to the left child
    f)  else, set the current node to the right child
    end
5.  return closest_node
```

2.2 Save images with the kd tree

Since RGB color space is three-dimensional, creating a kd-tree is a helpful way to speed up the search for the most appropriate matching image. The DFS algorithm is also used to locate the nodes with the greatest compatibility. And the process of building the photomosaic image is as algorithm 4.

Algorithm 3: The process of findClosest node using kd-tree

```
Input: data_img, node, depth, dim
Output: closest_node
1. if node is null, return
2. calculate distance between data_img and node's data
3. set min_distance to distance and set closest_node to node
4. if node has left child and distance is less than min_distance, recursively call findClosest
   on left child with updated min_distance and closest_node
5. if node has right child and distance is less than min_distance, recursively call findClosest
   on right child with updated min_distance and closest_node
6. return closest_node
```

Algorithm 4: The process of creating photomosaic using Kd-tree

```
Input: target_image, tree, tile_size, dim
Output: mosaic_image
1. set closest_node to null and set min_distance to infinity
2. set current_node to root of tree
3. while current_node is not null do
4. calculate distance between tile_mean and current_node's data
5. if distance < min_distance, update min_distance and closest_node
6. if current_node has a left child and distance is less than min_distance, set current_node
   to left child
7. else if current_node has a right child and distance is less than min_distance, set cur-
   rent_node to right child
8. else break loop
9. return closest_node
```

2.3 Multi-threaded image processing

The code here uses “#pragma omp parallel for” code directly from the C++ standard library. This code can be executed in multiple threads and is output on the for loop.

2.4 Other improvement

To make the code more readable, I have defined a ini file that stores parameters that users can modify.

3 Comparison of results

This is the input image and output image of different algorithm.

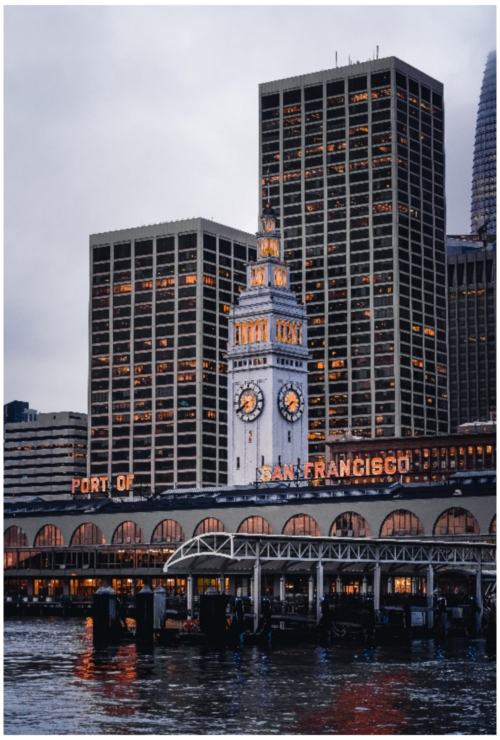
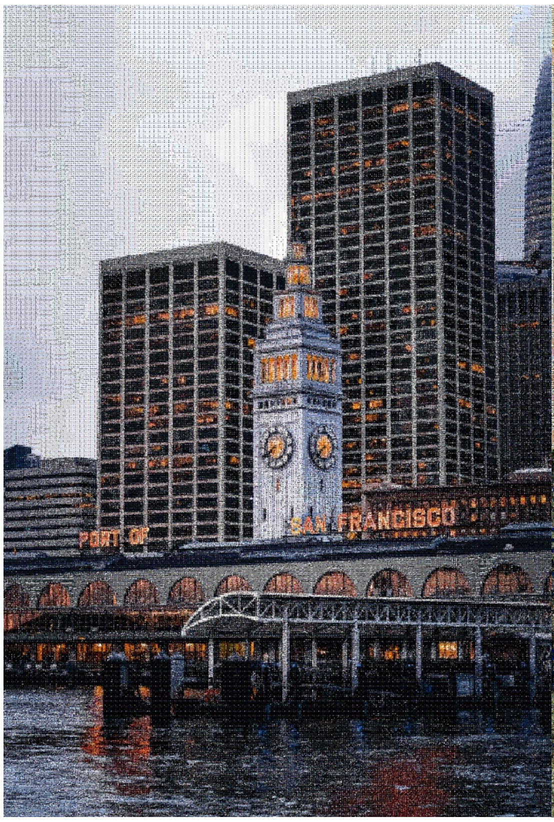


Figure 2: input image



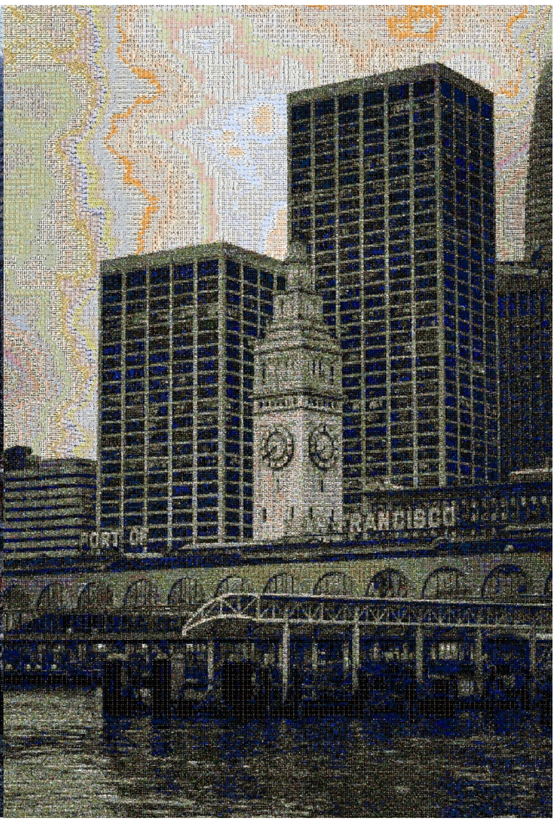


Figure 3: the result of original algorithm   Figure 4: the result of red-black tree algorithm

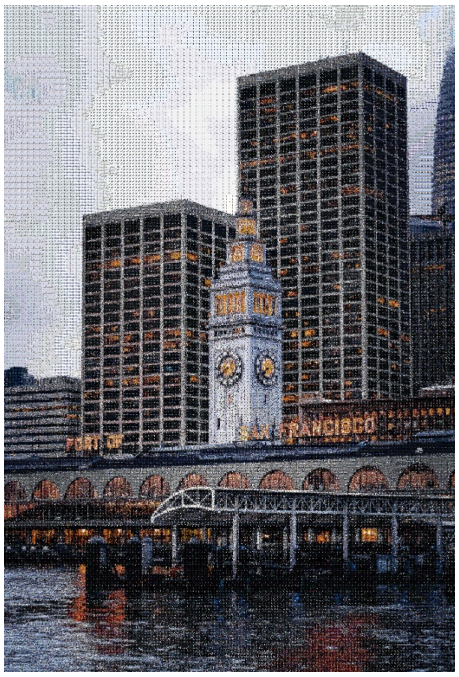


Figure 5: the result of kd tree algorithm

The number of images read for construction is 2w, and the size of the input image is 3700 * 5480, tile_ size is 10.

As a first step, analyze the results. Since the red-black tree relies directly on the RGB channel's weighted value to calculate its nearest node, the more obvious the color, the greater the difference. As a result, we were presented with a nearly black and white image. Based on the comparison between the image output from the original algorithm and the image output from the kd tree storage node, we can see that the image output from the original algorithm is superior. In a logical sense, the image produced by using kd tree should have the same result as the image produced by the original algorithm. The output image of the algorithm, however, is darker than the input image, which is possibly due to an error that is difficult to detect in the nearest neighbor search algorithm.

Take into account the running time. The running times of the three algorithms are 6.48 minutes, 25.81 seconds, and 41.65 seconds, respectively. According to the results, the initial algorithm takes too long and the red-black tree executes the fastest. However, the data structure is not suitable for this algorithm. The kd tree is much faster than the original algorithm. Of the three algorithms, kd tree is generally the most suitable.

4. Attachment description

mean.cpp: the code of original algorithm
rbtree.cpp: the code of red-black tree algorithm
kd.cpp: the code of kd tree algorithm
4.jpg: the input image
mosaic_image_mean.jpg: output of original algorithm
mosaic_image_rb.jpg: output of red-black tree algorithm
mosaic_image_kd.jpg: output of kd tree algorithm

References

Image source: https://images.unsplash.com/photo-1670382393921-516cfd9f3119?ixlib=rb-4.0.3&ixid=MnwxMjA3fDB8MHxwaG90by1wYWdlfHx8fGVufDB8fHx8&auto=format&fit=crop&w=693&q=80

Red-black tree: https://blog.csdn.net/cy973071263/article/details/122543826?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522167298916716800188571486%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fall.%2522%257D&request_id=167298916716800188571486&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~first_rank_ecpm_v1~rank_v31_ecpm-19-122543826-null-null.142^v70^control,201^v4^add_ask&utm_term=%E7%BA%A2%E9%BB%91%E6%A0%91%E7%9A%84%E8%B0%83%E6%95%B4&spm=1018.2226.3001.4187

Definition of photomosaic: https://en.wikipedia.org/wiki/Photographic_mosaic