Name: 杨晨翰(Yang Chenhan)
ZJU ID: 3210110301
Intl ID: chenhan.21
UIUC NetID: chenhan7

(Below are two parts of analysis, question a. b. c. d. e. f. are on page 5)

memory complexity

1.Description:

For the memory allocations being used in populating tile images, I think the original MP5 implementation has already reach the memory complexity of O(C*w*h) as required.

2. Theoretical analysis:

The process of populating tile images is implemented by the function named “mapTiles” in maptiles.cpp

```
MosaicCanvas* mapTiles(SourceImage const& theSource,
                       vector<TileImage>& theTiles)
```

The key part of this function is the code shown below, we can see two nested loops, in each loop cycle, we use the function “setTile” to put a tile into result, which is an object of mosaic. So we can now be sure that the running time is already w*h*x, where x should be decided by looking into the function “setTile”.

```
    for(int m = 0; m < n_rows; m++){
        for(int n = 0; n < n_columns; n++){
            result->setTile(m, n, get_match_at_idx(tree, tile_avg_map,
theTiles, theSource, m, n));
        }
    }
```

Below are the codes in setTile, we can see that we only pass a pointer of TileImage into myImages using the function “get_match_at_idx”, which returns a pointer of a maptile. So the memory allocated for each “setTile” should be C, which is a small constant for storing a pointer.

```
void MosaicCanvas::setTile(int row, int column, TileImage* i)
{
    if (enableOutput) {
        cerr << "\rPopulating Mosaic: setting tile ("
             << row << ", " << column
             << ")" << string(20, ' ') << "\r";
        cerr.flush();
    }
    myImages[row * columns + column] = i;
}
```

From the two functions above, we can know that the total memory allocated for populating tile images is O(C*w*h).

3. Empirical analysis:

To test running time, I use valgrind

Command: valgrind –leak-chech=full ./mp6 tests/mysource.png tiles/ 400 5 mymosaic.png

Result:

```
==17264== HEAP SUMMARY:
==17264==     in use at exit: 0 bytes in 0 blocks
==17264==   total heap usage: 158,633,727 allocs, 158,633,727 frees, 55,669,762,
992 bytes allocated
==17264==
==17264== All heap blocks were freed -- no leaks are possible
==17264==
==17264== For lists of detected and suppressed errors, rerun with: -s
==17264== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

I put another image set called “tiles” into mp6 repository with about 5400 images. I choose a source image called “mysource.png” which has 1280*1280 pixels and takes about 2.2MB in memory. And I try to create a mosaic that takes 400 tiles and each tile contains 5*5 pixels. From the output, we can see that 55,669,762,992 bytes are allocated in memory to produce the output image. Though there is no worse program to be compared to mp6, I think mp6 is good enough since the input source image and image set is large enough to take it fully.

Running time complexity

1.Description:

For the running time drawing a mosaic, I will analyze why original MP5 implementation has running time complexity of O(w∗h∗w′∗h′).

2.Theoretical analysis:

The process of drawing mosaic has three part.

The first part is to load tiles, which is implemented by function “getTiles” with running time O(n)

```
void makePhotoMosaic(const string& inFile, const string& tileDir, int
numTiles,
                int pixelsPerTile, const string& outFile)
{
   PNG inImage;
   inImage.readFromFile(inFile);
   SourceImage source(inImage, numTiles);
   vector<TileImage> tiles = getTiles(tileDir);
```

The second part is to populate mosaic, that’s to say, we need to search tiles for w*h tiles, each with lg(n) running time, so running time for second part is O(w*h*lg(n)). This is implemented by “setTile”->“findNearestNeighbor” -> “get_match_at_idx”.

The third part is to actually draw the image, which is implemented by “drawMosaic”

```
PNG MosaicCanvas::drawMosaic(int pixelsPerTile)
```

In this function there are two main parts to draw a mosaic.

The first part is the two nested loops. So the running time for this part is O(w*h).

```
for (int row = 0; row < rows; row++) {
```

```
        for (int col = 0; col < columns; col++) {
            int startX = divide(width  * col,       getColumns());
            int endX   = divide(width  * (col + 1), getColumns());
            int startY = divide(height * row,       getRows());
            int endY   = divide(height * (row + 1), getRows());
            images(row, col).paste(mosaic, startX, startY, endX - startX);
        }
```

The second part is to paste corresponding tiles in myImages into the mosaic we created above. By analyzing the function “paste”, we can see that each time we call “paste”, we use x*y, which is w’*h’ running time.

```
void TileImage::paste(PNG& canvas, int startX, int startY, int resolution) {
    // check if not resized
    if (resized_.width() == 0) {
        generateResizedImage(startX, startY, resolution);
    }

    for (int x = 0; x < resolution; x++) {
        for (int y = 0; y < resolution; y++) {
            canvas.getPixel(startX + x, startY + y) = resized_.getPixel(x, y);
        }
    }
}
```

Adding together the first part and second part, we can find that the total running time for drawing mosaic is O(w*h*w’*h’).

3.Empirical analysis:

To test running time, I use man time

Command: /usr/bin/time -l -h -p ./mp6 tests/mysource.png tiles/ 400 5 mymosaic.png

Result:

```
real 130.83
user 120.73
sys 7.35
         12097241088  maximum resident set size
                   0  average shared memory size
                   0  average unshared data size
                   0  average unshared stack size
             1821671  page reclaims
                   2  page faults
                   0  swaps
                   0  block input operations
                   0  block output operations
                   0  messages sent
                   0  messages received
                   0  signals received
               11630  voluntary context switches
               37901  involuntary context switches
       1595892894702  instructions retired
        412063012696  cycles elapsed
         13080280000  peak memory footprint
```

I put another image set called “tiles” into mp6 repository with about 5400 images. I choose a source image called “mysource.png” which has 1280*1280 pixels and takes about 2.2MB in memory. And I try to create a mosaic that takes 800 tiles and each tile contains 30*30 pixels. From the output, we can see it takes about 2 minutes to produce the output image. Though there is no worse program to be compared to mp6, I think mp6 is good enough since the input source image and image set is large enough to take it fully.

a. analysis for how much meory is used in original MP5:
this question is answered in the empirical analysis in **memory complexity** part.

b. c:
Just as I have analyzed in the theoretical analysis in **memory complexity** part. I think the use of memory in my original MP5 is already good enough. The memory complexity is already O(C*w*h). So I will skip this part.

d. Analysis of why drawing an image in your original MP5 implementation has an O(w'*h'*w*h) running time.:
This question is answered in "theoretical analysis" in **running time complexity**.

e. Describe what you have done to reduce running time in drawing images:
I find a simple way to reduce the running time. That is: in "get_match_at_idx", we pass the tile_avg_map by value. We can make modification by passing it by reference. This could save a lot of time from copying the map.

```
TileImage* get_match_at_idx(const KDTree<3>& tree,
                            map<Point<3>, int> tile_avg_map,
                            vector<TileImage>& theTiles,
                            const SourceImage& theSource, int row,
                            int col)
```

(before)

```
TileImage* get_match_at_idx(const KDTree<3>& tree,
                            map<Point<3>, int> &tile_avg_map,
                            vector<TileImage>& theTiles,
                            const SourceImage& theSource, int row,
                            int col)
```

(after)

f. Empirical analysis on your successful reduction of running time:
I use the same command before and after the modification:
/usr/bin/time -l -h -p ./mp6 tests/source.png ../mp5/mp5_pngs/ 400 5 mosaic.png
And we can see that the running time is improved from 87 seconds to 8seconds, which is a very abvious progress.

```
(base) ych@ychdeMacBook-Air mp6 % /usr/bin/time -l -h -p ./mp6 tests/source.png ../mp5/mp5_pngs/ 400 5 mosaic.png
Loading Tile Images... (4730/4730)... 4479 unique images loaded
Populating Mosaic: setting tile (399, 532)
Drawing Mosaic: resizing tiles (213200/213200)
Saving Output Image... Done
real 87.72
user 84.69
sys 1.75
        1647067136  maximum resident set size
                 0  average shared memory size
                 0  average unshared data size
                 0  average unshared stack size
            115093  page reclaims
                 1  page faults
                 0  swaps
                 0  block input operations
                 0  block output operations
                 0  messages sent
                 0  messages received
                 0  signals received
                 0  voluntary context switches
             22884  involuntary context switches
      1244326605856  instructions retired
       272576149487  cycles elapsed
         1155273216  peak memory footprint
```

(before)

```
(base) ych@ychdeMacBook-Air mp6 % /usr/bin/time -l -h -p ./mp6 tests/source.png ../mp5/mp5_pngs/ 400 5 mosaic
.png
Loading Tile Images... (4730/4730)... 4479 unique images loaded
Populating Mosaic: setting tile (399, 532)
Drawing Mosaic: resizing tiles (213200/213200)
Saving Output Image... Done
real 8.34
user 3.96
sys 1.04
         1539227648  maximum resident set size
                  0  average shared memory size
                  0  average unshared data size
                  0  average unshared stack size
             108745  page reclaims
                  2  page faults
                  0  swaps
                  0  block input operations
                  0  block output operations
                  0  messages sent
                  0  messages received
                  0  signals received
               6746  voluntary context switches
               1403  involuntary context switches
        54416443531  instructions retired
        16031746431  cycles elapsed
         1140281344  peak memory footprint
```

(after)