Photomosaic

Chad Estill, Ben Katz, Raymond Burtnick, John McClure, Weimu Song

Abstract

In this paper, the process of creating a photomosaic is discussed as well as the different results that come with changing the process. We discuss different ways of storing and retrieving the data that makes up the tiles for the photomosaic, as well as the different ways of choosing which tile is the most appropriate for the cell. After each of these are run there are considerable differences on which give better results. A best histogram accuracy of 98.28%, and 94.84% was obtained on the accuracy image set for both the Antipole Tree and KD Tree respectively.

Introduction

The techniques used to create photomosaics have other applications such as stitching together satellite images. This would be similar to the tile selection algorithms. Multiple factors determine what is the best image for each section of the original image making it difficult to find the optimal tile quickly. This is mainly due to the large amount of data to sort. Additionally, it is difficult to compare images and many ways to compute accuracy. This project focuses on comparing two different methods of sorting data and three different methods of placing tiles. The comparisons offer a way of visualizing the two different data structures by comparing the difference in tile selections.

Literature Review

The first algorithm that we implemented was the linear algorithm. This was mentioned in Generating Photomosaics: An Empirical Study. This was relatively simple and we wanted to make sure we could get any type of photomosaic before going on to harder stuff. In the paper, it talks about how to cut the photomosaic into smaller pieces and then iterate over all of the pieces to find their average color. Every time that it calculates the mean RGB value, it will find the image within the database that has the closest RGB value to that cell. Then the cell will be replaced by the image that is selected as the closest to its color. When this is finished, the photomosaic looks like a grid where each cell is a different image. Those images make up the photomosaic and because they are close to the colors of the original image, the photomosaic looks very similar to the original.

The second algorithm used to break up the photomosaic was found in Smart Ideas for Photomosaic Rendering. This paper talks about a recursive algorithm that will originally break the image into a grid that has only four cells. The RGB mean is calculated for each and checked with the tile database to see how closely the image with the closest RGB mean value is. If it is not within a certain threshold

specified by the user, then it will take that cell and divide it into four subcells, and each of those will be tested, until all of the subcells have an image that will fit. We used this approach, but when the threshold is low, (the colors of the tile have to be very similar to the original cell) then this algorithm takes a long time to run. So we decided to put a limit on the amount of times that it was allowed to recurse, and will allow the closest image even if it is not below the threshold.
We decided that there needs to be a way to sort all of the images so that it was possible to retrieve the data quickly when trying to find the closest RGB value. One of the solutions we used was mentioned in Fast Photomosaic. This is called Antipole Clustering, which is a way of sorting the data. It will take the data, and check to see if there are two points that are further to each other than a certain radius theta. If there are, then it will find the two that are the closest to that radius theta. Those will be the new centers for two sub-clusters. It will recursively do this until there are not any points that are further than the radius theta. Those points that are in the same cluster at the end of the sorting algorithm are of very similar RGB values and can be found very quickly.

Feature Extraction

To create a photomosaic, the first step is to break the original photo into subsections to extract color information. The color information from each of these subsections is then used to select an image from the database of images to replace that subsection when creating the final photomosaic. We used two methods of breaking up the original photo subsections; linear subsections and different sized subsections found by recursively finding variation in different sections of the image.

Linear Feature Extraction
The first step in breaking up the original photo into subsections using the linear method is to determine how many images will be in each column and row of the photomosaic. This can be any number smaller than the number of rows and columns in the original image, but for the true photomosaic effect, the number of images in each row and column should be less than a quarter of those in the original image. Using the number of small images to use per row and column of the original image, the size of each subsection of the original image is determined. A function then traverses all of the subsections on the original image and extracts the mean color (red, green, and blue channels) from each subsection and saves this information into a data structure. This data structure is then fed to the tile selection method to select the appropriate image that should replace each subsection.

Recursive Feature Extraction
The basic algorithm for recursively splitting the original image into subsection starts with splitting the image into 2 by 2 subsections. A metric is calculated for each of these subsections. If this metric does not meet a certain threshold, that subsection is then itself split into 2 by 2 subsections. Once all of the

subsections meet the threshold, color means are extracted for each section for tile selection. The two metrics we used for recursive splitting of the image are mean and standard deviation.
When using the mean as a metric to check whether subsections should be split again, the mean colors of the subsection and potential subsections are calculated. The mean color is from 0 to approximately 441. Any of the potential subsections that is farther than a Euclidean distance of 1.2 from the larger subsection, the potential subsection is recursed on.
When using standard deviation as a metric to split subsections, the standard deviation of each color channel in the subsection is calculated. If the standard deviation of the mean of the color channels exceeds 10, the subsection is split and its subsections are checked recursively.

Tile Selection

Tile selection is a process where we feed in the RGB color averages of a region of the original image into a nearest neighbor classifier. Two such classifiers that were used were K-D trees and Antipole k-means clustering.

K-D Tree
K-D trees will find a local minimum of euclidean distance. First a K-D tree is built using the tile bank. These are the RGB color value of all the tiles that are available. K-D trees are not self-balancing and therefore depend on have a large distribution in your dataset, which increases performance. K-D trees work similar to a binary tree, however, only one dimension is compared in each layer. Red will be compared in the first, green in the second, and blue in the third layer. This then repeats for the number of layers in the tree.

Antipole Tree
Described in *Fast Photomosaic*, the Antipole Tree is an efficient way to query a large dataset for the nearest point in n-dimensional space. Antipole clustering begins by obtaining the antipoles of the input set. The antipoles are the two points within the set that have the greatest euclidean distance from one another. The algorithm assumes that the distance from each point to all other points in the set is known. For our application, since this assumption is untrue, an efficient way for calculating the antipoles of a set was developed. The convex hull of the input set was used to reduce the number of distance calculations needed to determine the antipoles.
Once the antipoles were obtained, each point in the set was assigned to the nearest of the antipoles. This algorithm was the run recursively on each of the resulting subsets until stopping criteria was met. A visualization for this algorithm is shown in Figure 1.

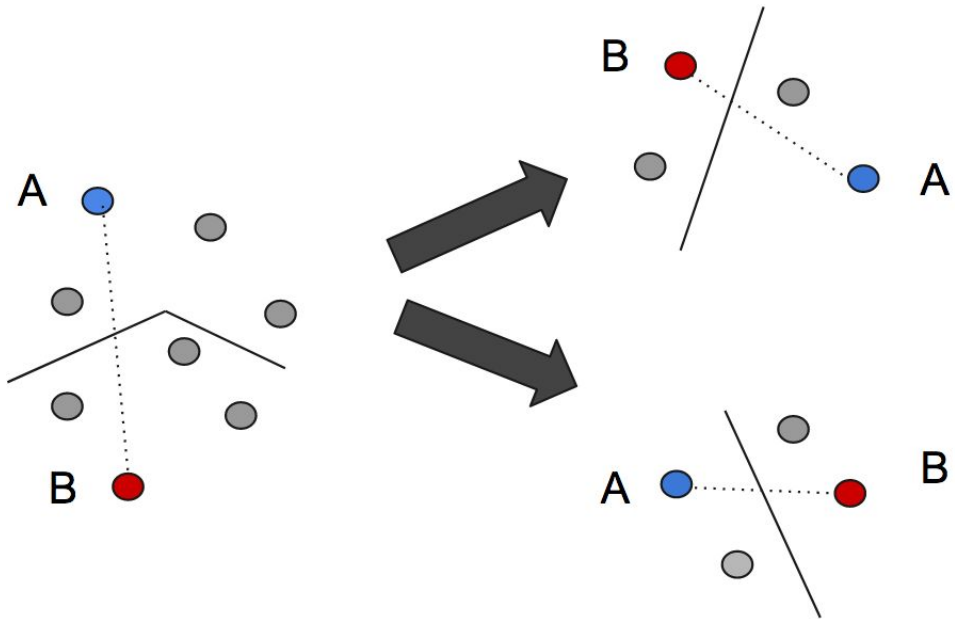


Figure 1 - Antipole Tree

Results

In Figures 3 and 4 the original image is split up into fixed size sections. These dimensions can be specified. To make all images and runtimes comparable, all images will be 2048x2048 pixels. This means that the image can be split up a max of 128x128 times with the smallest tiles being 16x16 pixels.

Figure 2 - Original image

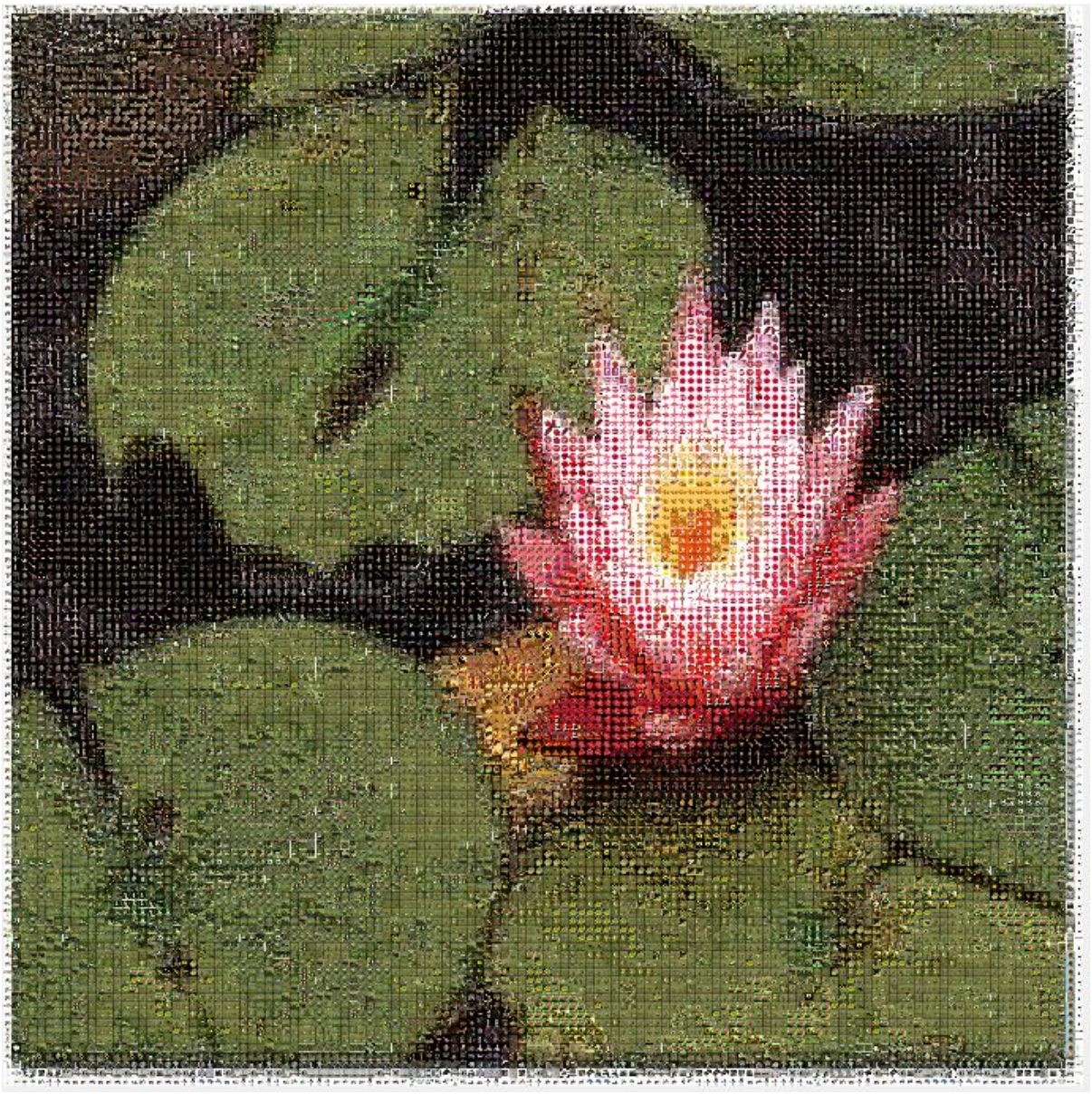


Figure 3 - Photomosaic broken up into 128x128 tiles of size 16x16 pixels. KD tree used to select tiles. It takes 322 seconds to run.

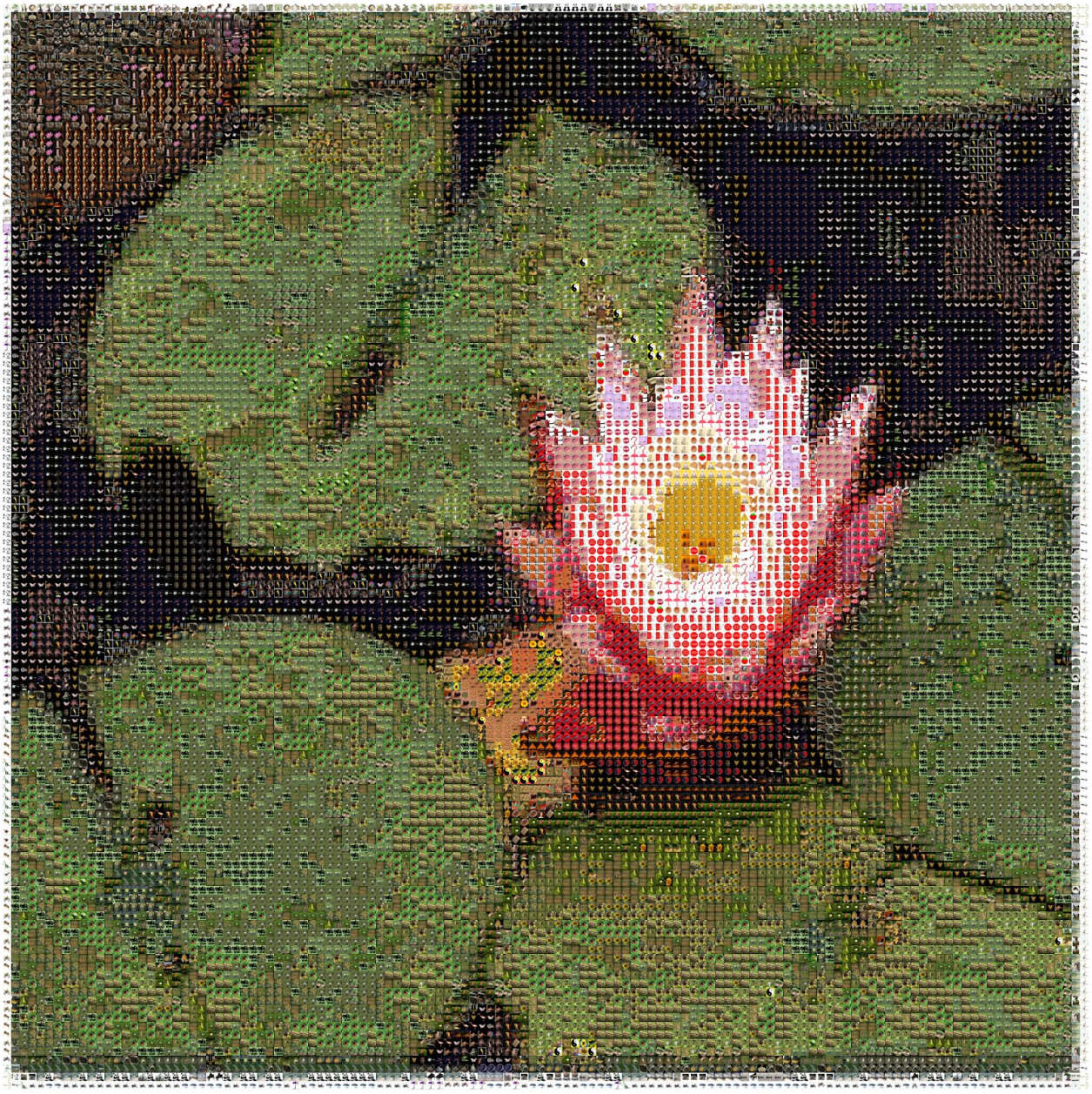


Figure 4 - Photomosaic broken up into 128x128 tiles of size 16x16 pixels. Antipole clustering used to select tiles. It takes 264 seconds to run.

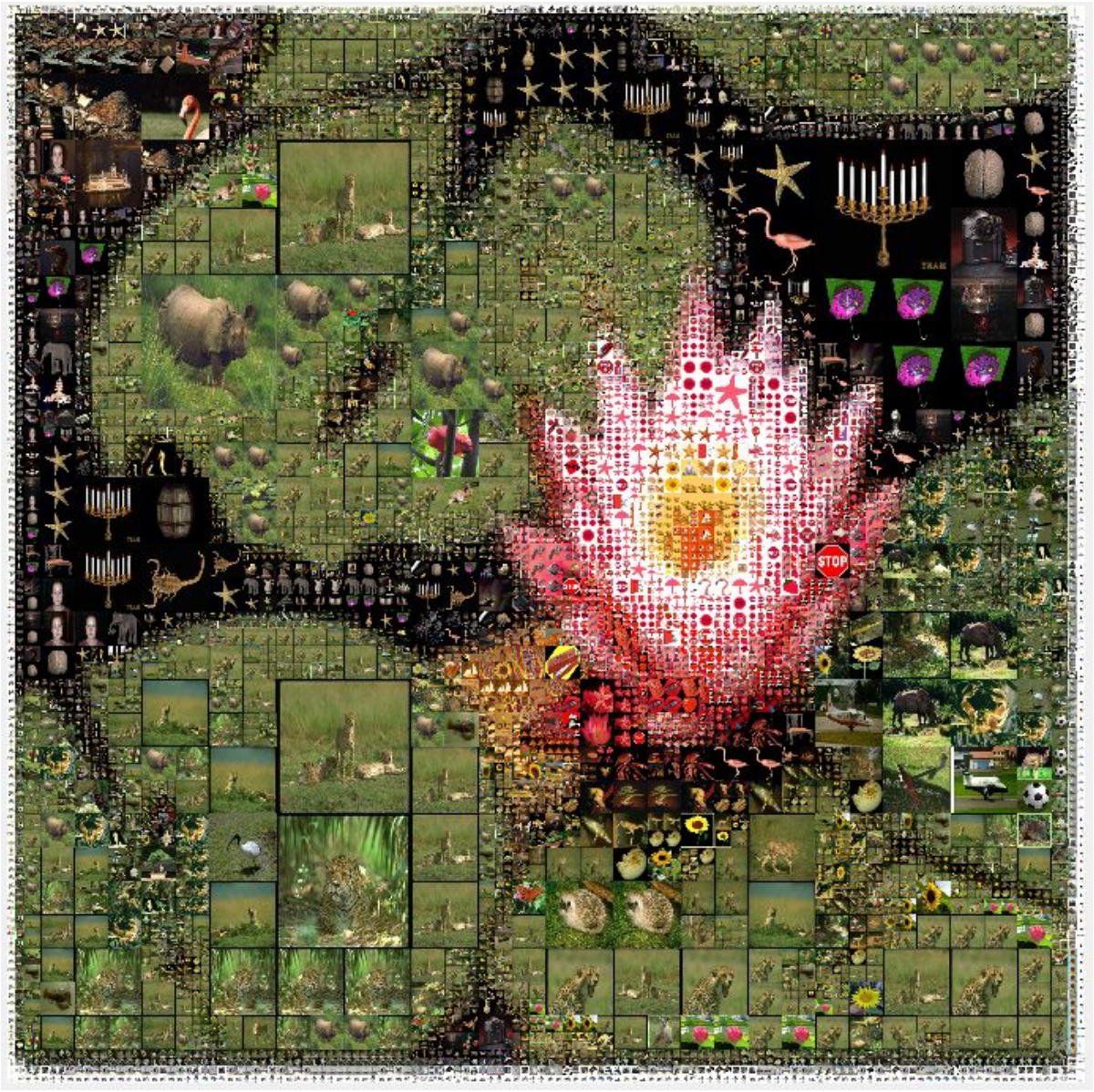


Figure 5 - Photomosaic recursively broken up if a region's standard deviation is greater than ten. KD tree used to select tiles. It takes 139 seconds to run.

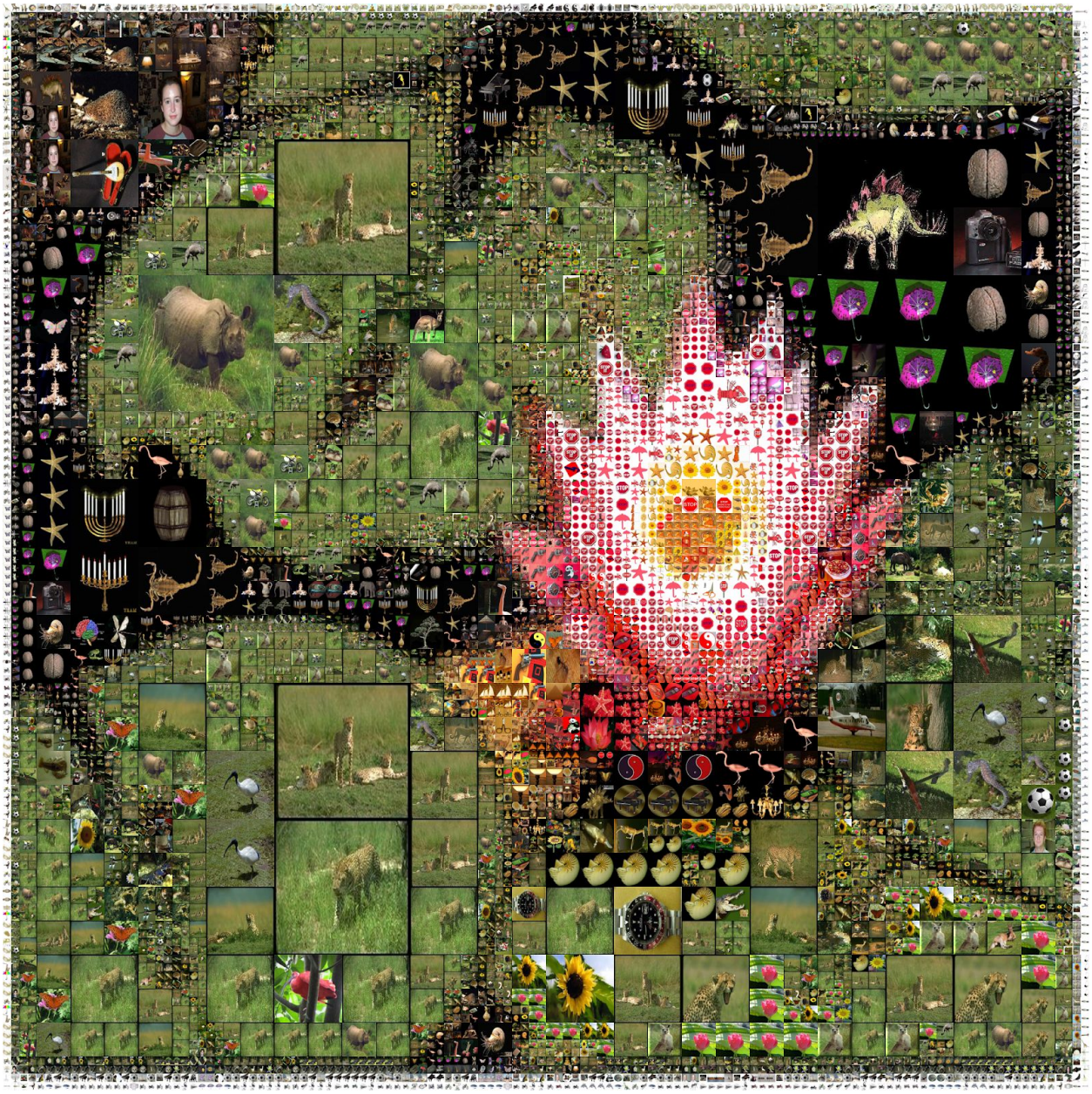


Figure 6- Photomosaic recursively broken up if a region's standard deviation is greater than ten. Antipole clustering used to select tiles. It takes 122 seconds to run.

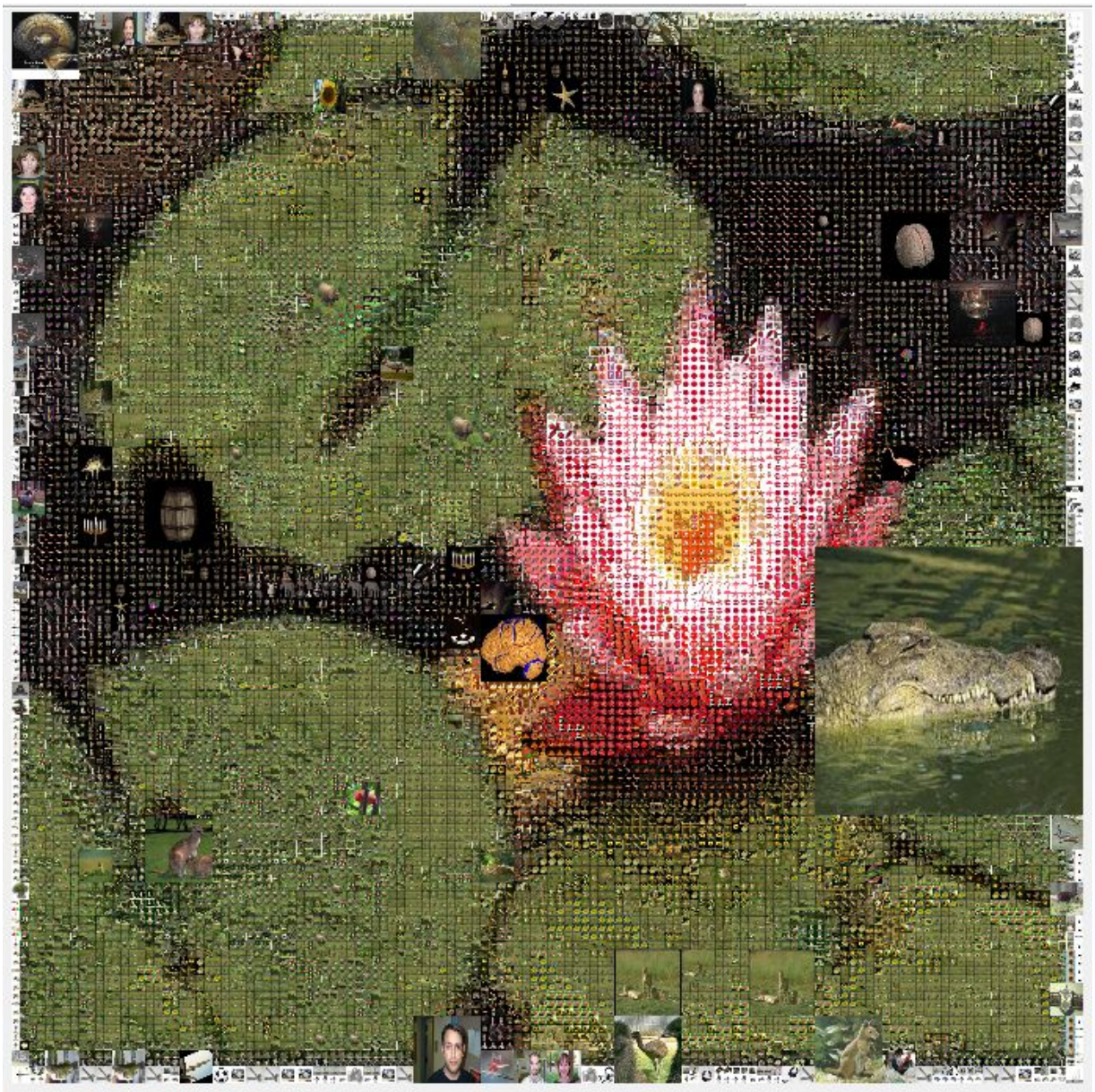


Figure 7- Photomosaic recursively broken up if the tile selected's average was greater than 1.2 away from the region of the original image. KD tree used to select tiles. It takes 216 seconds to run.

During the algorithm, the antipoles are stored in nodes belonging to a tree-like data structure allowing for traversal. By changing the stopping criteria of the algorithm, the tree could be configured to have leaves containing any number of images. This could be useful in the case of an extensive dataset of images, where features other than rgb means could be used to select the image for a given subregion. To test the effect of cluster size on the Antipole Tree, images were generated for trees of different maximum cluster sizes. For each case where the maximum cluster size was not 1, a random image from the nearest cluster to the sub-image was chosen. The results of this test are shown in Figure x.

Figure 7 - Original Image

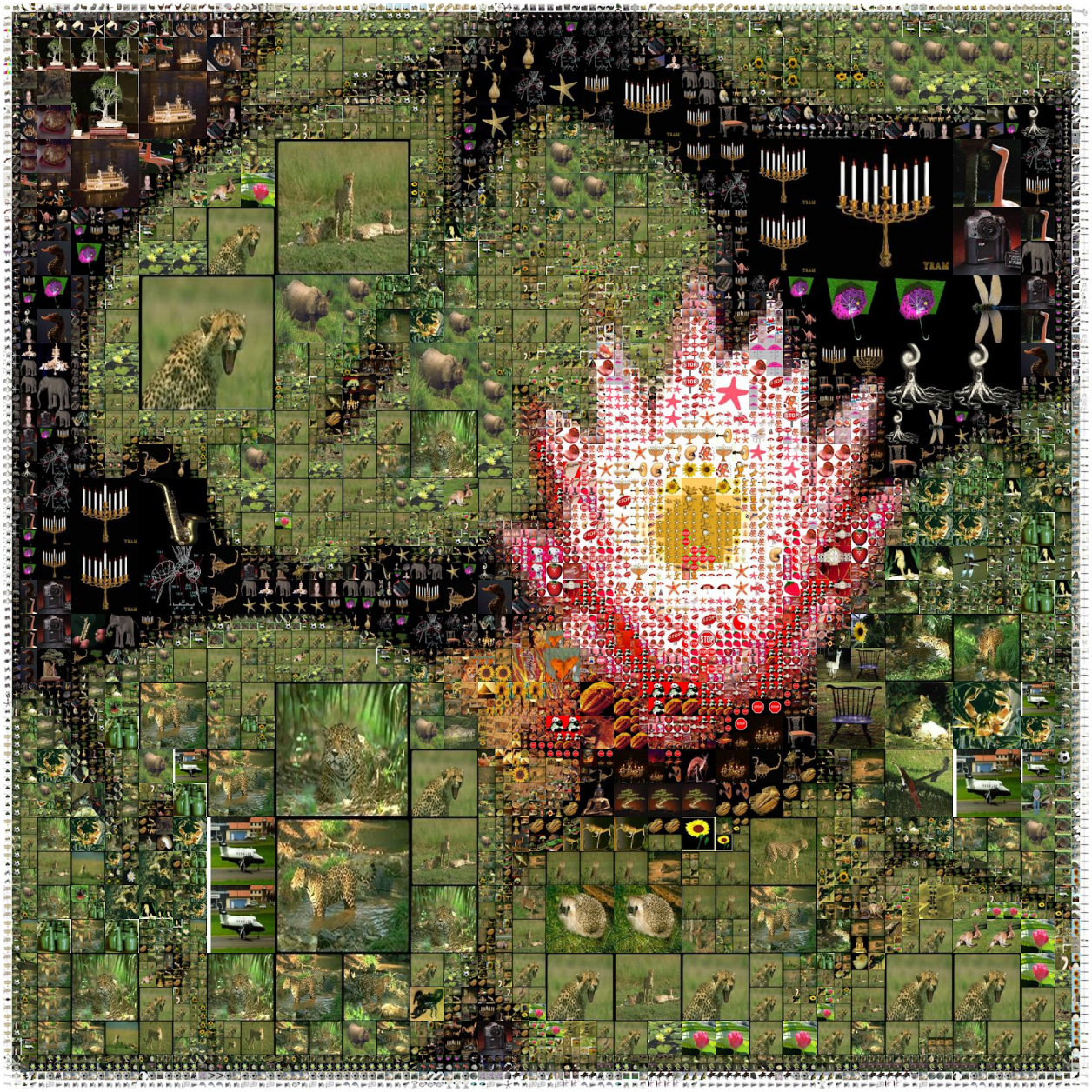


Figure 7 - Cluster Size = 1

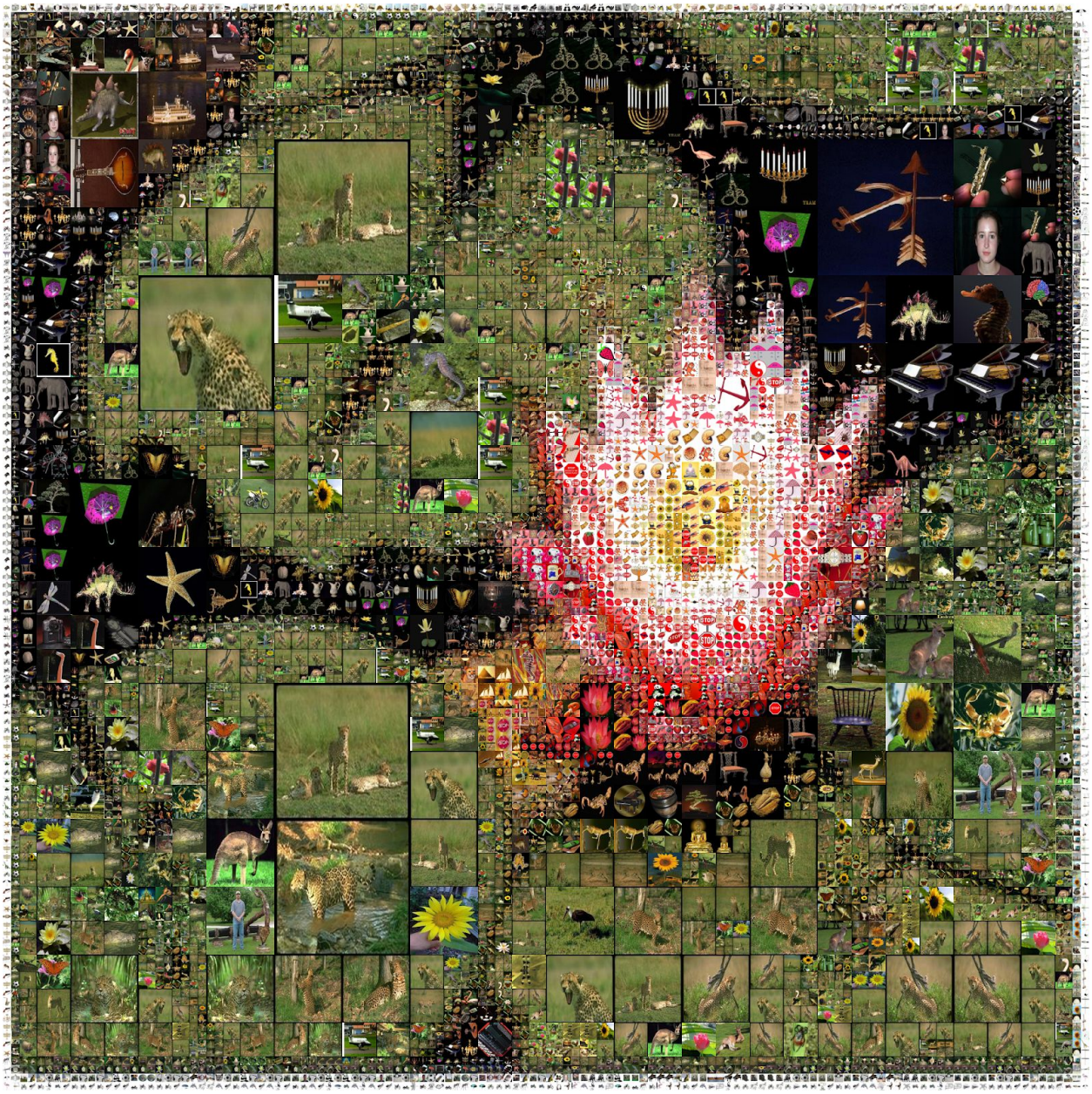


Figure 8 - Cluster Size = 5

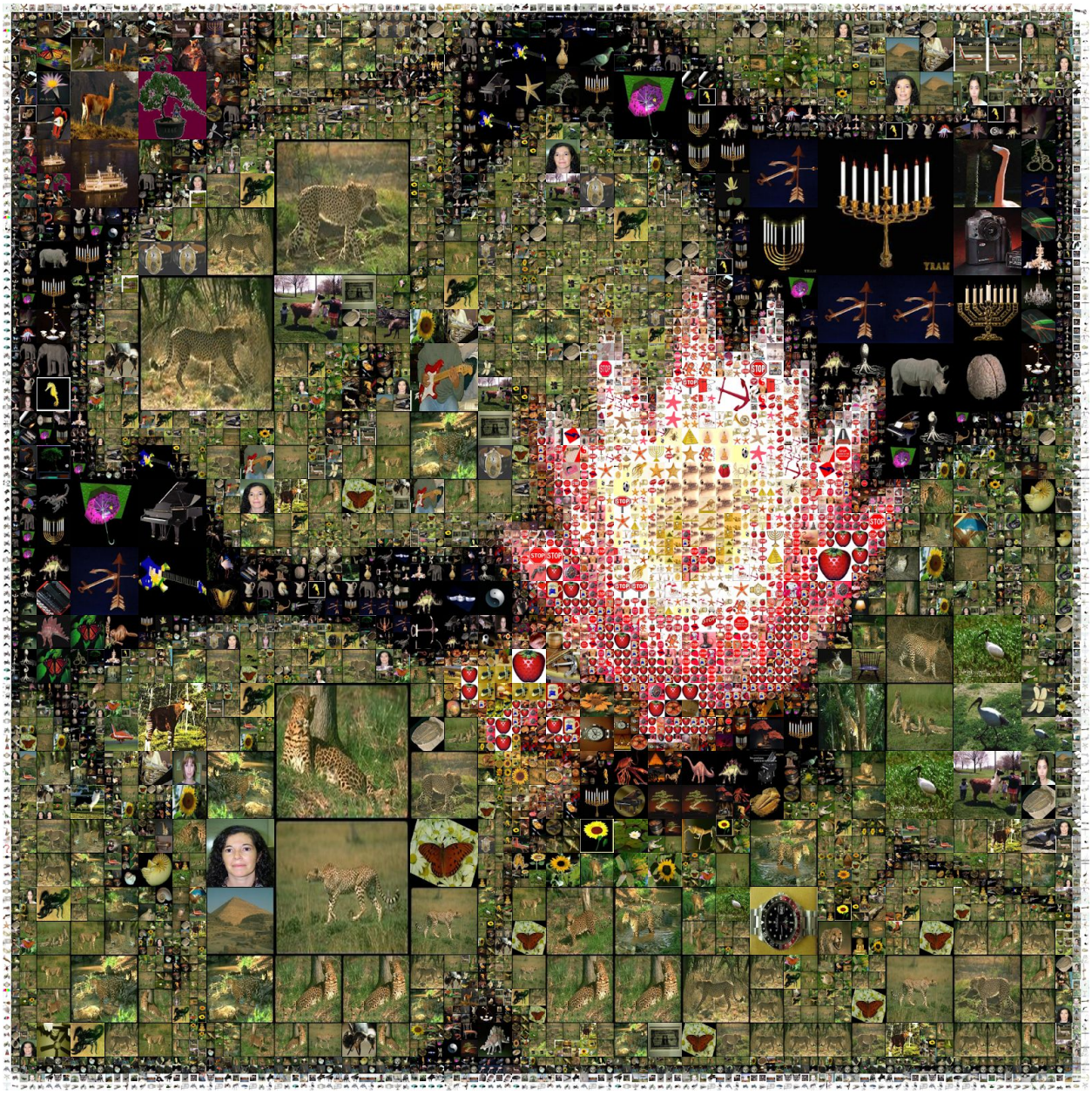


Figure 9 - Cluster Size = 20

It can be seen in the above images that an increase in cluster size of the antipole tree will increase the diversity of images present in the resulting image. For cases of an extensive dataset and desired randomness in the resulting image, larger cluster sizes could result in more visually interesting images while still accurately representing the original image. Another interesting takeaway from the cluster size comparison is the degradation of accuracy in the resulting images as cluster size increases. Increasing the cluster size from 1 to 5 only slightly reduces the quality of the resulting image while greatly reducing the repetition of images. However, the increase from 5 to 20 results in a significant decrease in quality of the final image. This leads one to believe that there is a point of diminishing

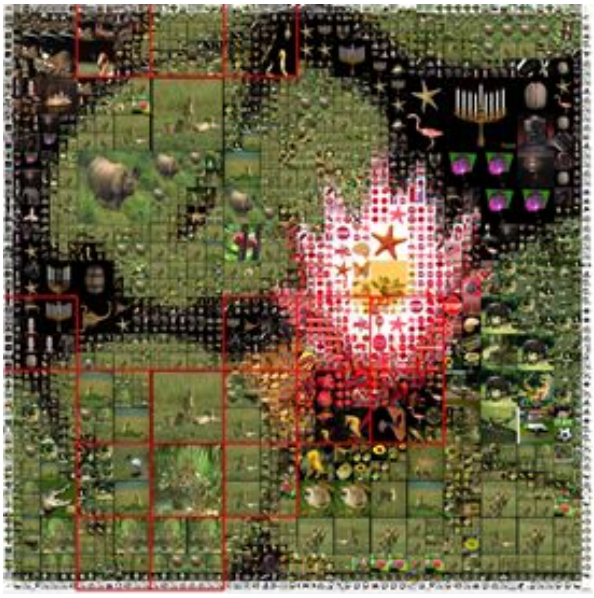
returns (most likely determined by the size of the dataset) in using cluster size to increase image randomness while maintaining accuracy.

The fix sized images take longest to run. This is probably because it has to read, resize, and insert the most images. The difference between using the KD tree and antipole as image classifies is small. They both have comparable times and good results.
Looking at just aesthetics, recursively selecting images based on how good of a fit they are making the photomosaic look abnormal. From a distance, the fixed size looks most like the original image, which is the point. Recursively selecting images based on standard deviation shows the edges in more detail as more smaller images since there are transitions in those places.

Discussion

One goal of the project is to generate a mosaic image as similar as possible to the original image. Our image comparison algorithm compared the euclidean distance of the two images and mark the different areas on the image.

Image comparative algorithm example: Comparing to the original image, the scissors are the different parts on the white background. For this example, the algorithm captured all of pixels which are 2.5% differ than the original pixels.

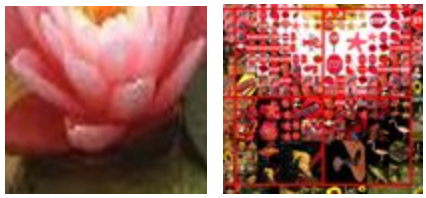


Original image Mosaic image

In the comparison between the original image and the mosaic image. We found that there are two conditions, which caused the mosaic algorithm to choose relatively low similarity images as tiles. The first condition is large area of single pure color. Like:

Since, for the low color variety area, our algorithm will use less images as tiles and the algorithm always only chooses one image as the tile in this area. Because of we do not have pure color images in the dataset, every single image it chose has a large possibility to have a low similarity compare to the original area. Like the example we showed above, even though the background of the tile is green, but the panther in the middle of the image has bright yellow, which caused this tile to have a low similarity compare to the original area.

The second condition is too many color varieties in a small area. Like:



In this example, the lily has several levels of petals and each petals' color is not pure pink but has a gradual change from pink to white; the lily also has its own shadow reflects on the leaf. These color changes all concentrated in a small area. The mosaic algorithm actually captured these color changes, and it chose some white background images and pink images, and then used some black background

images as the shadow, but the smallest tiles we have right now are not detailed enough to show these details, which causes these areas to have a relatively lower similarity.

To solve the problems, which caused by these two conditions, we can first expand our dataset, so the algorithm can have more choices on tiles when it is generating the mosaic image. The second solution is we can sacrifice some of the efficiency and use smaller tiles, so the mosaic image can present more details.

For the future works, if we can have two or three more weeks, we can expand our dataset and run the algorithm on them. This will slow down the speed of the algorithm, but can give us the mosaic image with higher similarity and less repetitive tiles. If we can have one more year to work on the project, we can build a new classifier instead of K-D tree and antipole. And for the image comparison function, instead of comparing the images in square parts, we can divide the image by edges, then compare them.

Another area for improvement also lies in the existing algorithms. When compared to precalculating all distances in the set, using the convex hull resulted consistently faster runtime. A future improvement on this algorithm would be to pass already-calculated distances between points on the convex hull. This would most likely result in a speedup because after the splitting of a set, many points that were on the convex hull in the previous set will remain on the convex hull in subsequent sets. Therefore, a large number of unnecessary distance calculations would be removed.

Finally, although we were unable to extensively test both trees' accuracy against one another, we were able to compare the two using the histogram based accuracy method which resulted in 98.24% accuracy for the Antipole Tree and 94.84% for the KD Trees. This test was performed by calculating the average accuracy of the method over the 8 image test set. The difference in accuracy is most likely due to the fact that the KD tree solves for local minima whereas the antipole tree obtains global minima.

Key challenges

Our first challenge was how we wanted to create the mosaic using the recursive algorithm. When we began, we knew that we wanted to split up the image and then check the RGB values as stated above. However, we did not know when to put the images into the mosaic. We had to decide if we wanted to cut the mosaic on the way down and then place the tiles on the way up, or place them as we go down. We decided to place them as we went down. This allowed us to have less lines of code and did not have to keep track of the sizes of the images for coming back up.

When we were resizing those images for the recursive algorithm, we had to make sure that each image was of a good resolution and were able to be resized into the cell that it was closest to. The size of the cell is different depending on which iteration of the recursive algorithm it is on. As it goes deeper, the size of the image is supposed to be smaller each time. The math behind

that was difficult for us as we had began, but once we understood how it would divide, it was not a problem. It just depended on how many squares you wanted to divide it into and then how many times you recursed.

The last problem is one that comes with group projects in any class. We had decided to split up the work for the different ways of sorting the dataset. This included KD trees and Antipole Clustering. Once we had finished this, both parties had changed the main code slightly to accommodate for their algorithm. Once both were finished, putting them together in a safe way was hard for us because we had to see where both overlapped and we wanted to make sure that the code to interchange which method we wanted to use. This was done with another variable in the method that we wanted to use, where you can decide which you want to use before running the program.

References:

Nicholas Tran. Generating Photomosaics: An Empirical Study. *Proceedings of the 1999 ACM symposium on Applied Computing*, pp. 105 -109, March 1999.
https://dl.acm.org/citation.cfm?id=298213

Gianpiero Di Blasi, Maria Petralia. Fast Photomosaic, D.M.I. - University of Catania
https://dspace5.zcu.cz/bitstream/11025/889/1/Blasi.pdf

Smart Ideas for Photomosaic Rendering Gianpiero Di Blasi, Giovanni Gallo, Maria Petralia D.M.I. - University of Catania Via A. Doria, 6 – 95125 Catania – Italy
https://pdfs.semanticscholar.org/c96f/5ac066ef9932ff6a440076dc21283b0bec6a.pdf