Mini Project 2

Austen Skopov-Normane and HyeSeung Lee

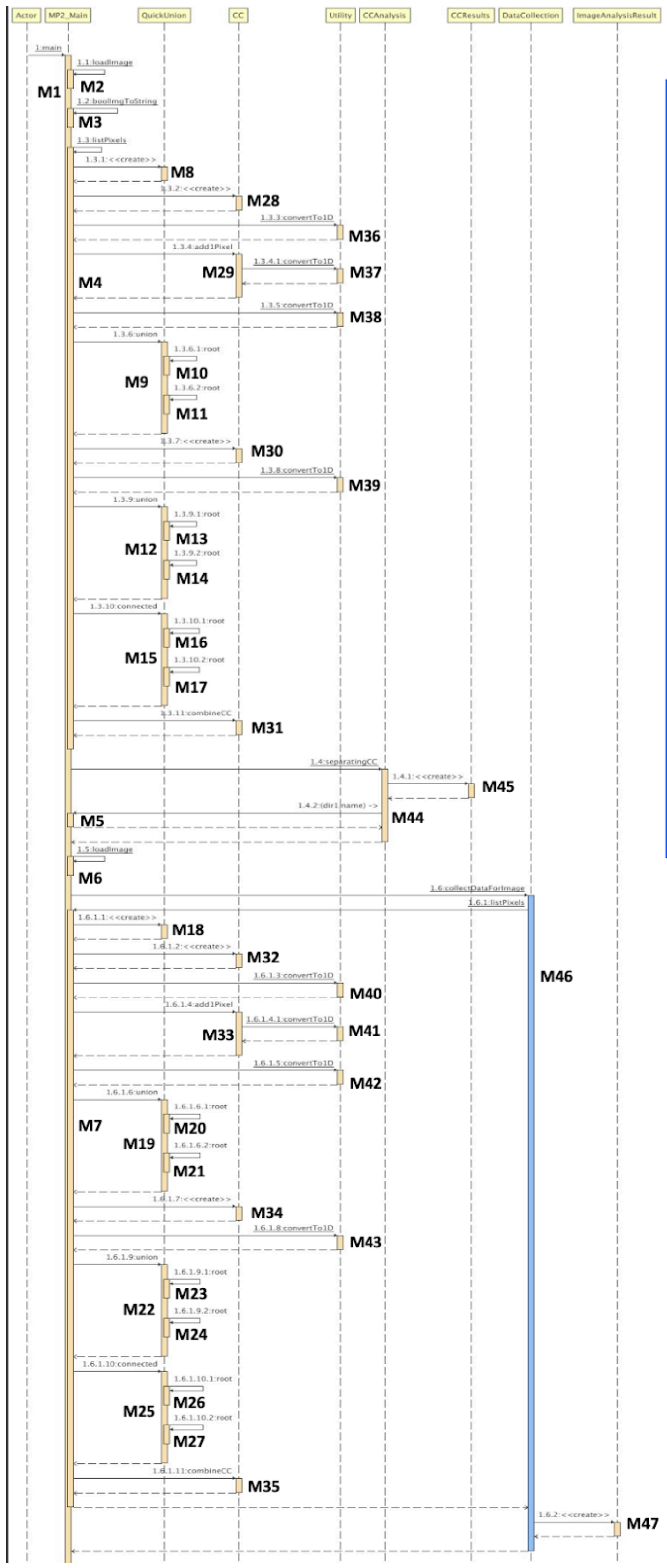
Introduction

The overarching objective of this assignment is to combine our knowledge of multiple data structures with connected component analysis to inspect binary images. More specifically, we are being tested on our understanding and implementation of Union Find and List data structures. This project also requires us to make logically sound code that is organized efficiently, as we were given a great amount of freedom in the structure and overall implementation. We are using our Java knowledge and applying it to a real-world application of analyzing connected components to figure out their shapes and the resolution of their images. In timing our code, we are also noting the efficiency of our data structures and algorithms. Through inspecting the architecture of our code, we are seeing how our classes and our methods interact.

This project also requires our knowledge of data collection and data output, as we are tasked with implementing an automated data collection system that runs tests in our code for us; this makes for an easier time down the road when analyzing our data. The goal of data collection is not only to find the correlations between two different variables from our program and the average run times but also to acknowledge the use of the scientific methods that are typically used for analysis.

The user can view data in two modes: image analysis mode or data collection mode. In image analysis mode, the user can view relevant results about one singular image. In data collection mode, all seventy images are analyzed together to compute numerous statistics. As this is an academic project, the main goal is not to create the most efficient program. The goal is to combine connected component analysis with data structures to ensure our understanding of Java code implementation, as well as learn about how to understand the correlation of changing variables in a program based on data collected from multiple test runs.

Code Architecture



Sequence Diagram

Method Key by Class

<u>MP2_Main</u>
M1: main method
M2: loadImage
M3: boolImgToString
M4: listPixels
M5: filter
M6: loadImage
M7: listPixels

<u>QuickUnion</u>
M8: QuickUnion
M9: union
M10: root
M11: root
M12: union
M13: root
M14: root
M15: connected
M16: root
M17: root
M18: QuickUnion
M19: union
M20: root
M21: root
M22: union
M23: root
M24: root
M25: connected
M26: root
M27: root

<u>CC</u>
M28: CC constructor
M29: add1Pixel
M30: CC constructor
M31: combineCC
M32: CC constructor
M33: add1Pixel
M34: CC constructor
M35: combineCC

<u>Utility</u>
M36: convertTo1D
M37: convertTo1D
M38: convertTo1D
M39: convertTo1D
M40: convertTo1D
M41: convertTo1D
M42: convertTo1D
M43: convertTo1D

<u>CCAnalysis</u>
M44: separatingCC

<u>CCResults</u>
M45: CCResults constructor

<u>DataCollection</u>
M46: collectDataForImage

<u>ImageAnalysisResult</u>
M47: ImageAnalysisResult constructor

Our coding solution works through the implementation of seven classes: MP2_Main, Utility, CCAnalysis, CCResults, QuickUnion, CC, and DataCollection. First, MP2_Main contains a couple of methods already encoded into the skeleton of the project to load and read the digital images. Then, Part 2 of the project is implemented here, where the connected components (foreground pixels) are identified and grouped together. In the main method, the user is given menu options for what mode they want (image analysis or data collection).

Next, the Utility class has a very straightforward purpose. This class converts a connected component into either 1D or 2D (i.e., a singular integer location/number or a point with a row and column). An inner class, Pair, can be found inside Utility as well. This class serves the purpose of creating a pair of integer points (to represent a row and column).

Then, the analysis class contains one method for Part 3 of the project. The method separates connected components by shape and returns a list of triangles and rectangles. These lists are held in the next class, CCResults. The sole purpose of CCResults is to contain two lists of connected components: rectangles and triangles.

Furthermore, the QuickUnion class contains all relevant union methods. Two optimizations have been added to QuickUnion: weighted quick union and path compression. As the name suggests, this class unions the connected components together.

Additionally, the CC class represents a connected component and completes Part 4 of the project. It also contains methods for adding one pixel to a shape (in essence) to calculate the bounding box, combining multiple connected components for the bounding box, figuring out the size range of shapes, a toString method, and comparing the shapes based on their size ranges, horizontal positions, and vertical positions to generate an ordered list of the shapes. Lastly, the DataCollection class is simply for the project's data collection. It contains one method that generates the results for Part 1 of data collection.

As depicted above, a sequence diagram of our project was generated to show how the classes and their methods interact with each other in the project. M1 starts the flow of the code, which then leads to loading the digital images through M2. Then, M3 and M4 are vital in setting up the digital images for further analysis in subsequent steps. M5 is used to filter the digital images to ensure the ones uploaded are in the right format. Pixels are then listed and loaded in M6 and M7. The QuickUnion class is initialized in M8 and M18, where it finds the roots of the connected components (M10, M11, M13, M14, M16, M17, M20, M21, M23, M24, M26, M27), unions them (M9, M12, M19, M22), and ensures connections exists (M15 and M25).

The CC class constructs and manipulates the connected components. The constructor of this class is used in M28, M30, M32, and M34; pixels are added to the components in M29 and M33. Connected components are combined in M31 and M35, and utility methods (M36 and M42) help transform these components into 1D. Connected components are separated in M44 to view them as distinct shapes, and data is held and analyzed with the help of M45 and M46. Lastly, M47 is used solely for data collection.

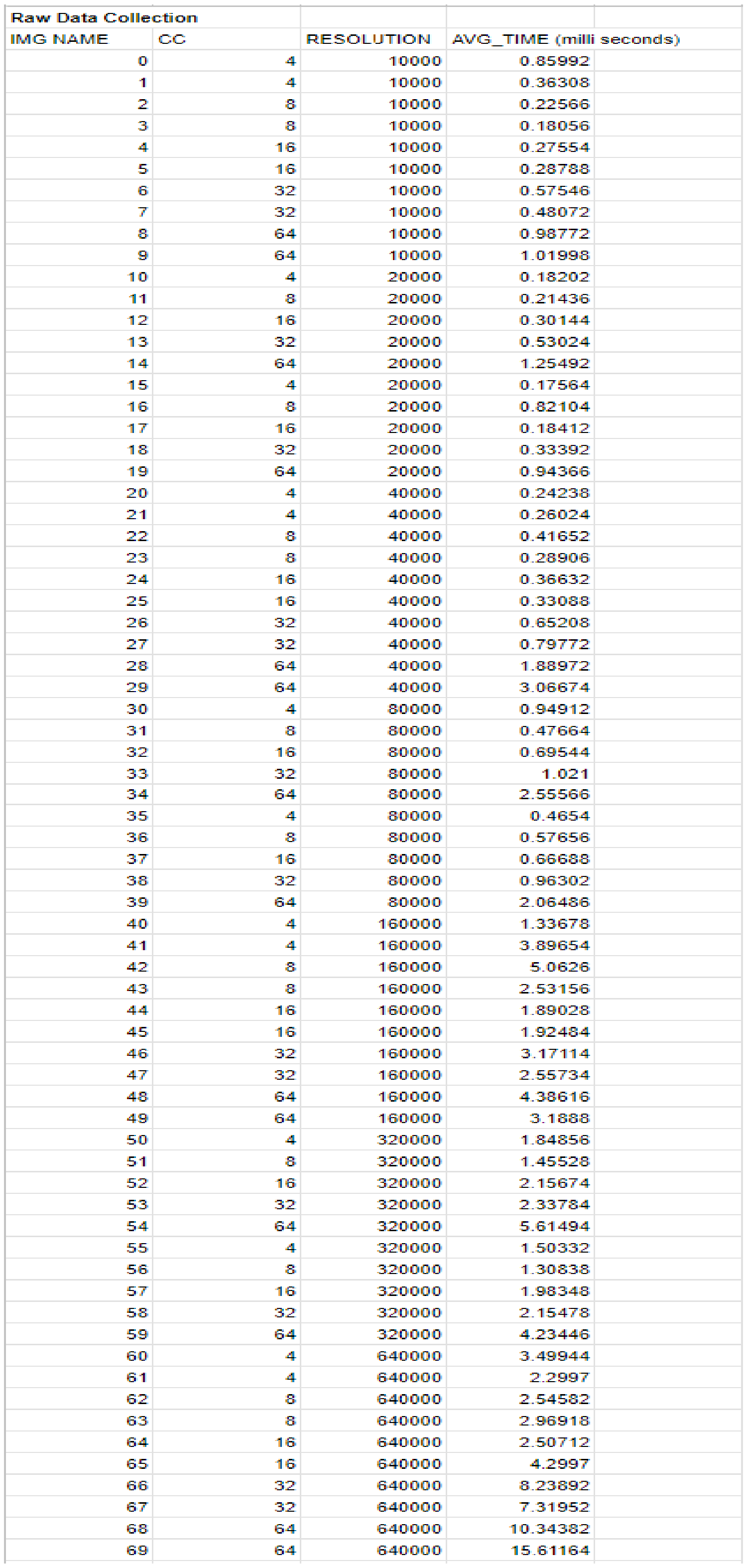
Computer Specs

For this experiment, the data collection was done on one laptop. The processor is an Intel(R) Core(™) i7-8565U CPU@ 1.80GHz 1.99 GHz. The installed RAM is 16.0GB(15.7 usable). The system type is a 64-bit operating system with an x64-based processor. The edition is a Windows 11 Home, and the manufacturer is HP. The SSD is KXG60ZNV256G by Toshiba, and the GPU is Intel(R) UHD Graphics 620 with a total available graphics memory of 8171MB. Lastly, the Java version used is version 17.0.9.

As a part of our discovery, the impact of computer specs on program run times in particular was very interesting to us. The Windows(C:) hardware on our laptop is the KXG60ZNV256G by Toshiba, which is an SSD (solid state drive). These drives use NVMe(non-volatile memory express) technology over the PCIe interface, which allows for much faster data transfer than HHDs.

While the GPU does not directly affect the compilation speed of JAVA programs, its impact is more about ensuring the system remains responsive and usable during these tasks, rather than speeding up the compilation process itself. Since the GPU on this laptop uses shared system memory, the available RAM for both the GPU and the Java applications is from the same pool. With 8171 MB available to the GPU, this is beneficial for running memory-intensive Java applications or multiple applications simultaneously in IntelliJ.

Results



Raw Data Collection			
IMG NAME	CC	RESOLUTION	AVG_TIME (milli seconds)
0	4	10000	0.85992
1	4	10000	0.36308
2	8	10000	0.22566
3	8	10000	0.18056
4	16	10000	0.27554
5	16	10000	0.28788
6	32	10000	0.57546
7	32	10000	0.48072
8	64	10000	0.98772
9	64	10000	1.01998
10	4	20000	0.18202
11	8	20000	0.21436
12	16	20000	0.30144
13	32	20000	0.53024
14	64	20000	1.25492
15	4	20000	0.17564
16	8	20000	0.82104
17	16	20000	0.18412
18	32	20000	0.33392
19	64	20000	0.94366
20	4	40000	0.24238
21	4	40000	0.26024
22	8	40000	0.41652
23	8	40000	0.28906
24	16	40000	0.36632
25	16	40000	0.33088
26	32	40000	0.65208
27	32	40000	0.79772
28	64	40000	1.88972
29	64	40000	3.06674
30	4	80000	0.94912
31	8	80000	0.47664
32	16	80000	0.69544
33	32	80000	1.021
34	64	80000	2.55566
35	4	80000	0.4654
36	8	80000	0.57656
37	16	80000	0.66688
38	32	80000	0.96302
39	64	80000	2.06486
40	4	160000	1.33678
41	4	160000	3.89654
42	8	160000	5.0626
43	8	160000	2.53156
44	16	160000	1.89028
45	16	160000	1.92484
46	32	160000	3.17114
47	32	160000	2.55734
48	64	160000	4.38616
49	64	160000	3.1888
50	4	320000	1.84856
51	8	320000	1.45528
52	16	320000	2.15674
53	32	320000	2.33784
54	64	320000	5.61494
55	4	320000	1.50332
56	8	320000	1.30838
57	16	320000	1.98348
58	32	320000	2.15478
59	64	320000	4.23446
60	4	640000	3.49944
61	4	640000	2.2997
62	8	640000	2.54582
63	8	640000	2.96918
64	16	640000	2.50712
65	16	640000	4.2997
66	32	640000	8.23892
67	32	640000	7.31952
68	64	640000	10.34382
69	64	640000	15.61164

Raw Data (milliseconds)

RESOLUTION	MIN	MAX	MEDIAN	AVERAGE
10000	0.18056	1.01998	0.4219	0.525652
20000	0.17564	1.25492	0.31768	0.494136
40000	0.24238	3.06674	0.39142	0.831166
80000	0.4654	2.55566	0.82228	1.043458
160000	1.33678	5.0626	2.86424	2.994604
320000	1.30838	5.61494	2.06913	2.459778
640000	2.2997	15.61164	3.89957	5.963486

Image 1 - Independent Summary by Image Resolution R (milliseconds)

CC	MIN	MAX	MEDIAN	AVERAGE
4	0.17564	3.89654	0.90452	1.277295714
8	0.18056	5.0626	0.6988	1.362372857
16	0.18412	4.2997	0.68116	1.276475714
32	0.33392	8.23892	0.99201	2.223835714
64	0.94366	15.61164	2.8112	4.083077143

Image 2 - Independent Summary by Numbers of CC M (milliseconds)

AVERAGE	CC				
RESOLUTION	4	8	16	32	64
10000	0.6115	0.20311	0.28171	0.52809	1.00385
20000	0.17883	0.5177	0.24278	0.43208	1.09929
40000	0.25131	0.35279	0.3486	0.7249	2.47823
80000	0.70726	0.5266	0.68116	0.99201	2.31026
160000	2.61666	3.79708	1.90756	2.86424	3.78748
320000	1.67594	1.38183	2.07011	2.24631	4.9247
640000	2.89957	2.7575	3.40341	7.77922	12.97773

Image 3 - Average Time for Unique Combinations of R and M (milliseconds)

Analysis

Here, we will take a look at the average runtimes of data collection and analyze the numbers. For simplicity, we will be calling the number of CCs "M" and the image resolution "R". First, let's take a look at how the resolution of the input image impacts the run time. Based on the numbers from Image 1, there is a fairly consistent pattern/relationship between the two factors (R and run time).

Next, execution time increases are highly prevalent. There seems to be a trend where both the minimum and maximum execution times increase as M increases. There are a few outliers in this test run, such as the Max run time of 640k R and the Max run time of 80k R. For these abnormal numbers, we have to consider the many factors that could affect a test's run time: Java's internal operations like garbage collector, caching effects, and so on. Another factor to consider for this specific test is M. We have controlled R, but not controlled M, and since we do not know the exact effects of M, we cannot ignore the fact that the run time might be abnormal because of M. For example, the number from 80k to 160k R changed drastically -- it doubled. This could be due to the drastic increase in M jumping from 80k to 160k R, but we will look at that in the next part of the analysis. Except for the few outliers, we see a very rough ratio of approximately 1 when comparing both the average and the median run times; this usually means that there is a proportional increase in run time to the increase in R, which indicates a linear relationship O(N). However, it is hard to spot a direct and uniform relationship between the two. Based on a graph we created with this data set, the increase seems somewhat consistent in the beginning, but then it dips because of the outlier; the majority of the data appears to continue the trend.

Next, let's take a look at how the number of CCs in the image impacts the run time. Based on Image 2's data, we see a clear correlation between M and run time. As M increases, the average run time increases. We compared each of the Min, Max, and Median runtimes, and as we expected, all of them show a similar pattern when compared to M; they all grow proportionally in the range between linear and logarithmic time complexity. The relationship between runtime and Max average time is the upper bound of the time complexity (which is almost linear) and the lower bound is when compared with the Min average time (which is almost logarithmic). One of the potential outliers is the Max run time when M is 8, but it does not give a massive dip when considering the rest of the data. The approximate ratio between the increase in average run time and the increase in M is about 1, not considering the outlier in this case.

What these two independent summaries between the run time statistics and M and R tell us is that there is an increase in run time when there is an increase in either R or M (independently). Both of these two test runs' data seem noisy, excluding the outliers, there is a linear order of growth for both of them.

Finally, we will take a look at the data from Image 3. This image shows how the run time changes corresponding to both of the variables, R and M. When we look at the column (M) as a fixed variable here, and different rows (R), it is a bit hard to spot a pattern. For example, in the first and second columns (when M is 4 and 8, respectively), there isn't a proportional change in the average run time based on changing R. Although, for the rest of the tests when M is 16, 32, or 64, we can spot a pattern where the average run time increases as R increases. The number jumps significantly when R increases from 320k to 640k.

When we consider R as the fixed variable and compare the changing M, we find it hard to spot a pattern from the data as well. The average run time does not change proportionally according to the change in M. When R is at 640k, we can see that the average run time does increase as M increases.

This inconsistency could indicate two things that are worth mentioning. Firstly, it could indicate the lack of data. We do not start seeing a consistent pattern in data until we reach a larger number, which happens fairly commonly in this practice. This indicates that we might need a bigger data set to obtain more accurate information. Second, this might indicate that there is no specific relationship between R and M. Although we do see some consistency in specific combinations in this test set, there is not enough data to prove the existence of a correlation between the variable R (image resolution) and M (number of CCs). If we were to create a proposition based on this data, it would be that the average run time increases with the increase in R and the increase of M. The order of growth of this relationship would be somewhere close to linear, and potentially quadratic.

We had an assumption about the relationship between R and M based on the two independent studies. Because we had a conclusion that both R and M have a linear runtime, if both are the factors in this case, the order of growth would be somewhere close to quadratic (N * N = N^2). Although the data did not prove us right, it did not prove us wrong either. We believe this is the nature of data collection, where multiple precise test runs are required to prove a proposition. It would be interesting to continue this study and run some more tests on a larger data set in the future.

New Application

A new application where I would use connected component analysis (CCA) is in a new mobile app designed to help detect skin cancer, specifically cancerous moles. This app would be free and help those who cannot access a doctor due to a multitude of reasons (lack of insurance, transportation, etc.). From the user standpoint, all they would have to do would be to download the app, make an account/upload personal information, and upload photos of their moles. This app would use CCA to tell the user if the mole is either potentially cancerous or completely normal. If the mole were to be identified as potentially cancerous, then the user can take it upon themselves to go ahead with the next steps, as the sole purpose of the app is an initial screening of their moles.

This app would be reliant on CCA to be effective. Cancerous moles are detectable through their overall shape. While normal moles are symmetrical (both halves are nearly mirror images of each other), cancerous moles typically are asymmetrical and are more amorphous. CCA would be able to contrast between the mole lesion and the surrounding skin by inspecting pixel color (i.e., foreground and background). Then, CCA would be able to identify the overall bounds (outer shape) of the mole, as well as the size, texture, and hue. Because CCA can differentiate between different pixels of an image, using this technique in this application would be efficient. Furthermore, cancerous moles tend to be noticed by patients because these moles have changed in appearance over time. This app can inspect multiple images of the same mole taken over time and use CCA to analyze any changes in appearance (size of moles, their outline shapes, etc.). CCA is necessary in this application because of its efficiency in analyzing and differentiating between pixels in digital images.

Conclusion

This project allowed us to put our knowledge of data structures and Java syntax to the test. We explored QuickUnion and List structures in conjunction with connected component analysis to analyze binary images. Our project consisted of seven classes (MP2_Main, Utility, CCAnalysis, CCResults, QuickUnion, CC, and DataCollection) to process these images and extract information about the connected components for further analysis.

Our code architecture was designed in a way that made for our digestible understanding, as pieces were broken up into numerous classes that each had their tasks. Our classes interacted in a manner that allowed for efficient loading and analysis of digital images, connected component separation by shape, and data collection for analysis. In the analysis, we looked at the effects of image resolution and the number of connected components on runtime. The specific steps we implemented in our program from the scientific methods include: defining the research question or hypothesis, data collection, analyzing the data, interpreting the results, reporting the findings, refining and iterating. We were able to observe relational patterns between these variables and explored potential correlations.

Some of the most important concepts and lessons we learned during the process include the following: breaking down the bigger task into smaller tasks; making sure the current part of the program compiles correctly, and debugging before moving on to the next part; making valid assumptions and propositions before and after collecting raw data, and applying those into the analysis; running the same tests a couple extra times to ensure the validity of data; and creating visual tables for easier pattern discovery as well as implementing graphs when needed. Overall, this mini project has given us valuable insights into the implementation and analysis of connected component analysis and image processing.