Shelby Wilson

ECE1895

Design Project 3 – Report

Design Overview

For my final project, I created an algorithm visualizer application. This application simulates the steps of four popular searching and sorting algorithms: breadth-first search, A* search, quicksort, and selection sort. The pathfinding component of the application allows users to upload mazes and see them get solved by the algorithm, while the sorting algorithm component allows users to watch different sized bars get sorted to form a right triangle. The user can run and compare the algorithms based on speed and number of operations or comparisons performed. Screenshots from my final application windows can be seen below in Figure 1.

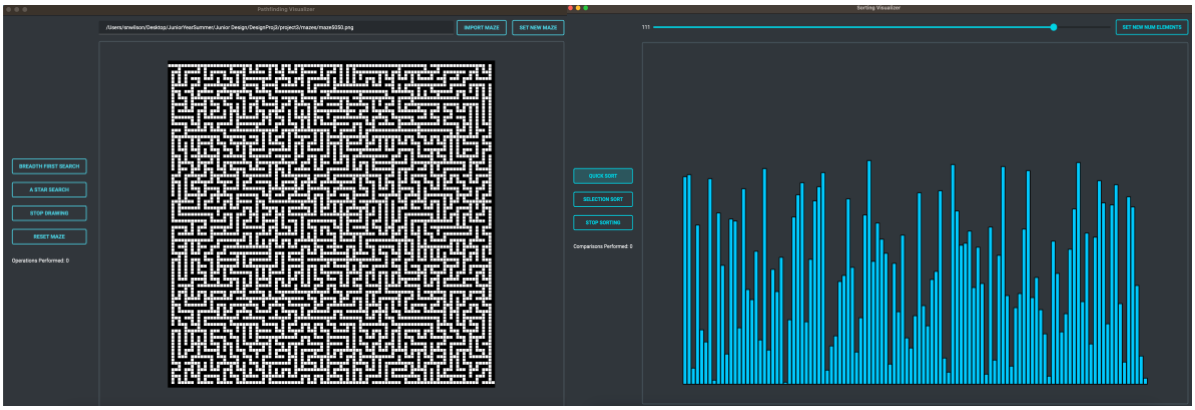

Figure 1: Main Application Windows: Pathfinding Visualizer and Sorting Visualizer. The launch window has been omitted but is shown elsewhere in the report.

Throughout the past few weeks, I had to make some modifications to my design. Originally, I wanted to use a Raspberry Pi and LED matrix to create the algorithm visualizer. I ended up switching to an Arduino UNO to save money for the project. Unfortunately, this ended up being an issue when I started coding because the Arduino UNO could not handle the amount of memory needed for all algorithms I would be simulating. Also, the shield I bought to connect

the matrix to the UNO ended up covering up some input pins I would have needed to select between algorithms in my design. The LED matrix I was going to use is seen below in Figure 2, along with the shield that goes along with it.

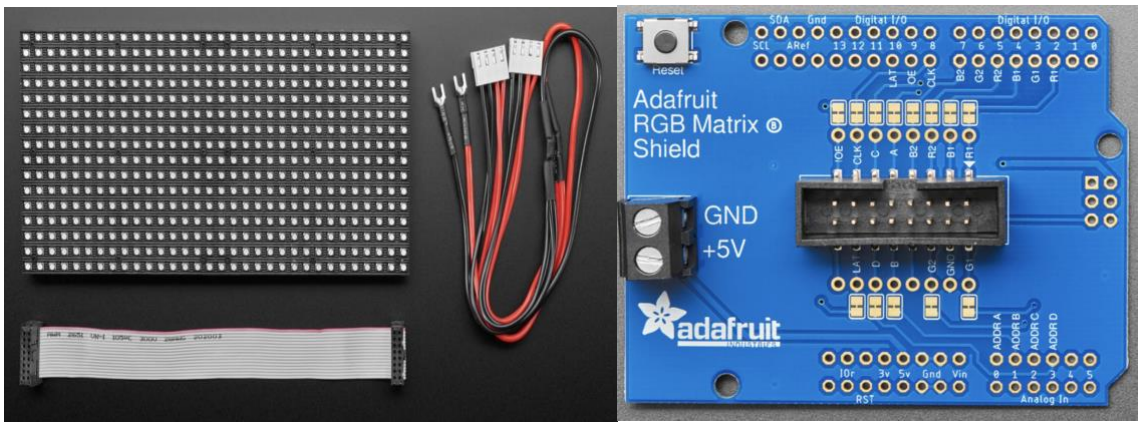

Figure 2: Adafruit 16x32 RGB LED Matrix (left) https://www.adafruit.com/product/420, Adafruit RGB Matrix Shield (right) https://www.adafruit.com/product/2601

I believe my final design is still within scope for a final project. I feel it is justifiable for the summer semester and current situation. I had to change my design a few times throughout this process, which led to extended research and pushed back the coding process. I feel I had no other choice but to switch to a PySide6 application because I knew I could get it done within the shortened deadline.

Preliminary Design Verification

As explained above, I had some issues while testing my initial design. After writing some of the code the breadth-first search algorithm, I quickly realized I was already using about 50% of the UNO's memory. I realized that even if I was able to simplify my algorithms, a final design using the Arduino UNO runs the risk of being choppy and hard to follow. This is when I asked Dr. Dallal to switch to a software application design because I did not feel I had the time to

pursue debugging and limiting my algorithms. A screenshot from the memory usage with just breadth-first search added to the original Arduino program can be seen below in Figure 3:

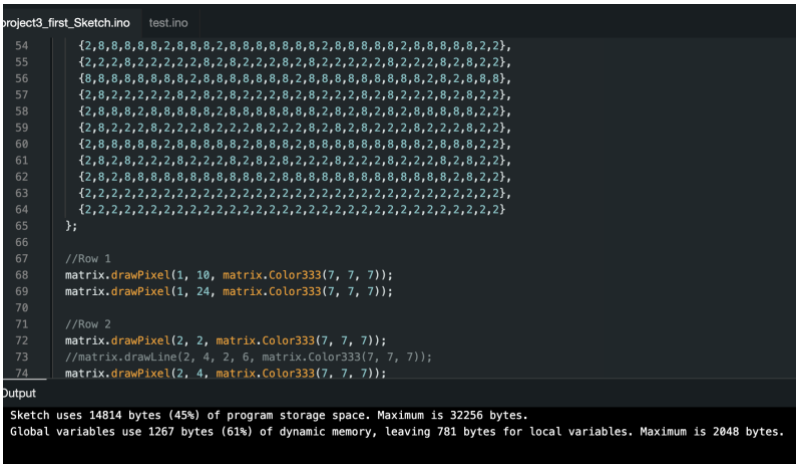

Figure 3: Overwhelming memory usage from implementing breadth-first search on Arduino UNO.

After I decided was making a software application instead, I started creating a very basic application to simulate breadth-first search on the same maze I was trying to simulate on the Arduino UNO. I made this simple application to make sure my breadth-first search algorithm worked and to learn basic concepts of the QGraphicsScene class. This also helped me verify that the project was feasible to complete within the limited time frame and would require enough work to be within scope. A screenshot of this application is shown, and the demonstration video is linked below in Figure 4:

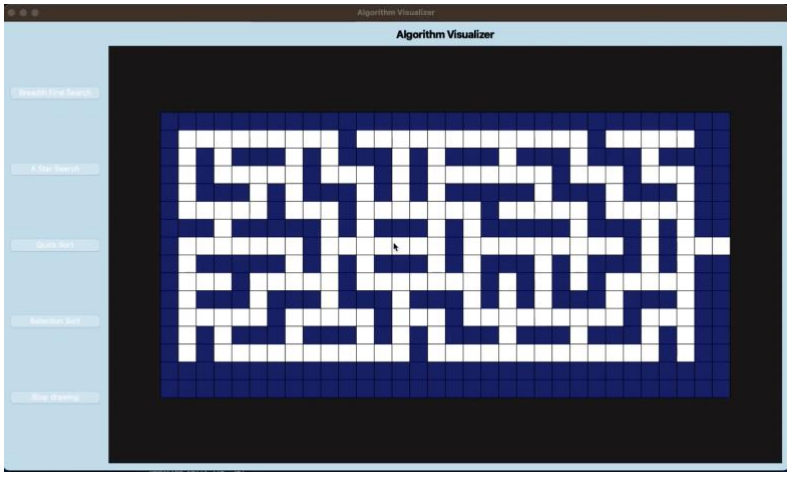

Figure 4: Test application. Link: https://youtu.be/FMfEblQiR8c

After completing this basic breadth-first search visualizer, I started working on my final design for the pathfinding component. The testing and design process for each major feature I added are described in the following sections and in the Final Presentation.

Once I had my final pathfinding application, I started working on a simple sorting visualizer component. Surprisingly, I had more issues with this than I did while developing the initial breadth-first search visualizer. I found many issues with rendering the bars after swaps were performed in the back-end algorithms. Due to my improper indexing of the elements in the bar container and not emitting enough signals, the rendering of the bars was choppy and some of them overlapped. Unfortunately, I don't have a screenshot of this error because I did not commit it to my GitHub repository.

I was able to solve this issue by thoroughly testing my application and found that re-rendering the entire list of bars after each swap is better than trying to swap each individual bar. After learning this, I changed the signals within the QuickSortRunner to emit entire lists instead of two indices to swap. This is when I felt ready to develop and test my final Sorting Visualizer application, as described in the following sections.

Design Implementation

I created my final algorithm visualizer using PySide6, a Python-compatible binding of the C++ Qt6 application framework. It contains both a front-end for user interactions and a back-end to run the algorithms and emit signals to the front-end. The link to my GitHub repository with all my application components is seen below:

Link to Public GitHub Repository: https://github.com/snw0502/designproject3

The major sub-components of my application are also described below:

- **Front-end** – main user interface for the application, includes:

- **LaunchWindow** – window that appears upon running the app; allows users to run either the pathfinding or sorting visualizers; allows user to exit all windows with a single button
- **PathfindingVisualizer –** main window for the pathfinding application; allows users to import external maze images to simulate pathfinding; contains buttons for running breadth-first search and A*, along with buttons that allow the user to stop the visualization and reset the maze; displays the number of operations performed while either algorithm is running
- **SortingVisualizer –** main window for the sorting application; allows users to select the number of elements to sort using a draggable slider; contains buttons for running quick sort, selection sort, along with a button for stopping the current algorithm; displays the number of comparisons performed while either algorithm is running
- **GridScene** – derived from QGraphicsScene; renders the maze using the PIL Python module; contains a grid of **GridCell** objects
- **GridCell** – pixel components of the image used in maze rendering
- **SortingScene** – generates a random configuration of bars of differing heights based on the user's selected number of elements

All components of the front-end can be seen in Figure 5 below and in further detail on my GitHub repository:

Figure 5: Front-end components

- **Back-end Algorithm Runners:**
    - **Algorithm Runners:** QRunnable objects used to run each respective algorithm inside a QThreadpool; contains a run() method for running the algorithm and a stop() method that sets the stop flag to true
    - **Algorithm Signal Objects:** used to contain signals for each algorithm; used by respective **Algorithm Runners** to emit signals to the UI for visualization and number of operations/comparisons

All components of the back-end of my application can be seen below in Figure 6 and in further detail on my GitHub repository.

Figure 6: Back-end components

Throughout the design process, I implemented features incrementally and tested them thoroughly before moving onto the next feature. I started with the breadth-first search algorithm implementation, then moved onto image rendering and file-reading for the mazes. After working out a few bugs with this feature, I then moved onto adding the A* algorithm and additional buttons for user control. I added text to display the number of operations performed and buttons for the user to reset the maze or stop the algorithm early to set up a new configuration. I also added a signal to display the shortest path found by A* in a dark blue color when the algorithm finds the maze exit. This shortest path visualization is seen below in Figure 7:

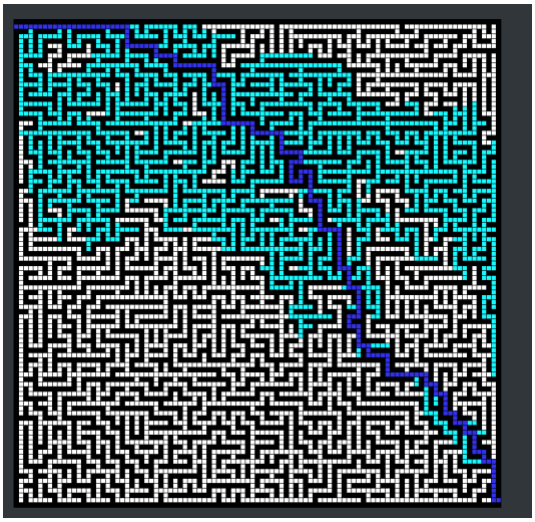

Figure 7: Shortest path found by A* on 50x50 maze

Once the pathfinding application was complete and verified to be functional, I started implementing the sorting visualizer. I started by implementing a QGraphicsScene of randomly generated bars. As explained in Design Verification, I had many issues with this feature. I solved this by doing research and debugging my program.

Once I changed my approach to re-rendering the bars, I added text to display the number of comparisons performed during algorithm execution. I also added a button for the user to stop

the algorithm during execution if they want to choose a different algorithm or number of elements without waiting for the current visualization to complete.

Finally, I decided to create a Launch Window to separate the two windows and provide a better user experience. With this control window, the user can either start the pathfinding visualizer, the sorting visualizer, or both. The control window also allows the user to exit all application windows that the user currently has open. To avoid delving into UI design, I also opted to using a pre-made UI theme (resource linked in my GitHub repository). After setting up the design theme and adding this Launch Window, as seen in Figure 8, I felt my design was complete.

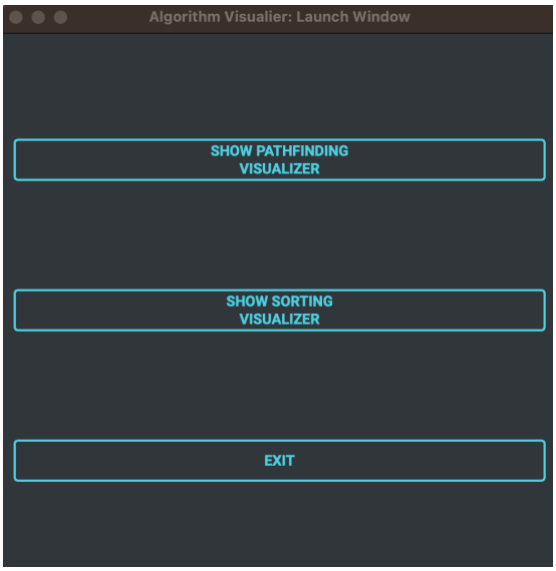

Figure 8: Launch Window

Design Testing

Once my design was complete, I ran many visual tests to ensure all components were working together correctly. For the pathfinding application, I tried many different mazes with

both algorithms to make sure they could solve any maze provided. Screenshots from these successful tests can be seen below in Figure 9:

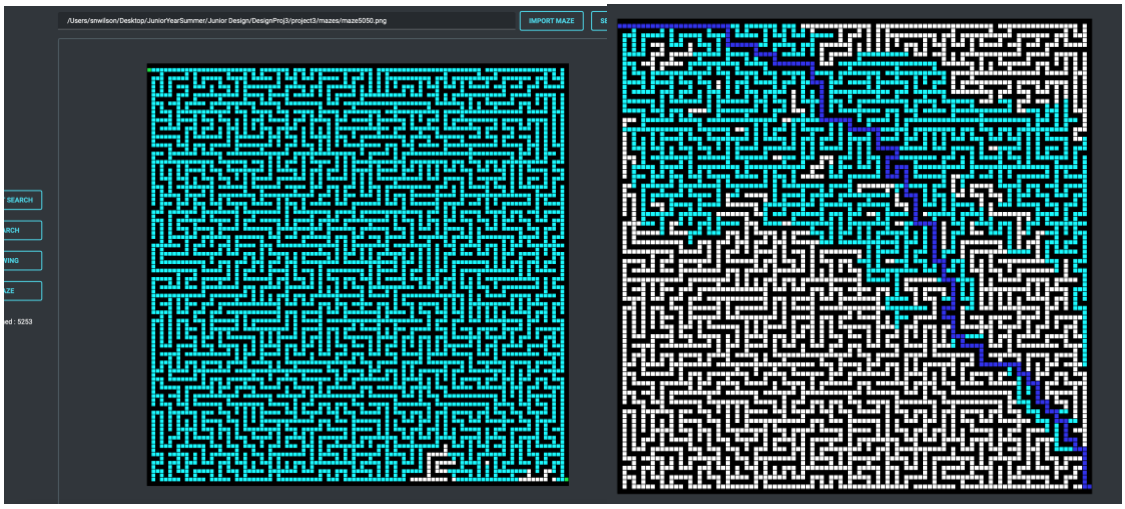

Figure 9: Successful breadth-first and A* testing

I also tested the resetting of the image, importing mazes, and the stopping of algorithms during execution, which is shown in my Final Demonstration. Unfortunately, I did not have the time to add a pausing feature, so the stopping feature only stops the search/sort at a specific moment. The user must restart the algorithm after stopping instead of being able to resume where the algorithm left off. This is due to the implementation of my Algorithm Runners and I did not have time to fix it. However, this is minor functionality that does not affect the main goal of the application. As this is hard to include in screenshots, I discuss this in my Final Presentation.

For the sorting visualizer, I tested both algorithms with different numbers of bars. The successful test cases are shown in my Final Demonstration. I also tested to see the limit to how many bars I could show on the screen, which ended up being around 125. So, this is the limit I set for my program. Seen below in Figure 10 is a version of my app with the maximum set too high and they render outside the screen's limits:

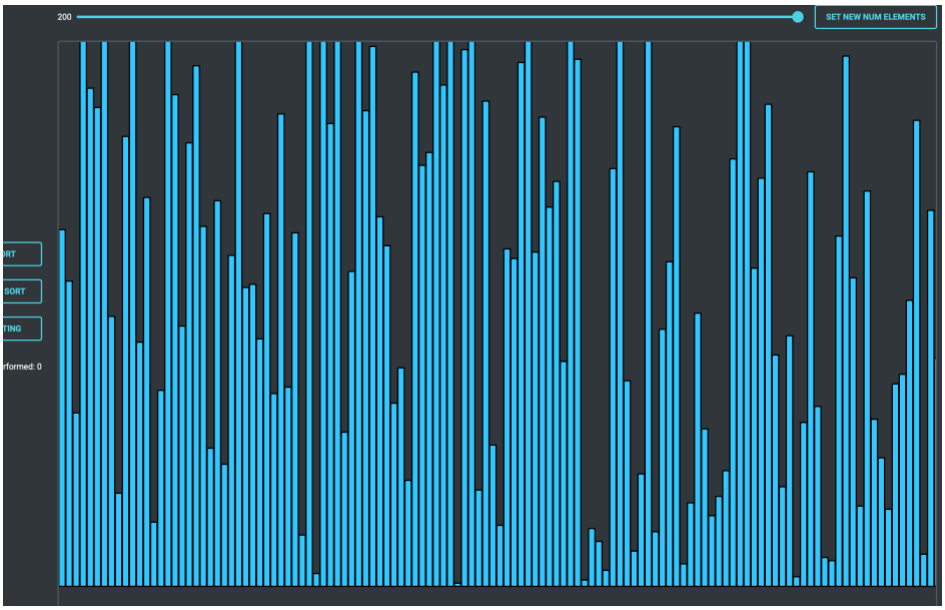

Figure 10: Bars rendering outside screen boundaries when the max is set too high.

Summary, Conclusions, and Future Work

Overall, I think this project was a success. Although I ran into some issues with my original design ideas, I think the project turned out alright. I created an interesting application that can teach others about popular pathfinding and sorting algorithms. I also learned a lot about connecting back-end algorithms to front-end visualizations, which is a very important skill to have in the field of software engineering. I had fun revisiting the algorithms I used in this project and am excited to have created my own application.

I plan on making further improvements to this design in the future. I would like my application to handle bigger mazes and more elements for sorting. As explained previously, I had to limit the size of the maze because I ran out of time to debug issues with resizing the QGraphicsScene. With more time, I am sure I could a way to make the size of the maze scale with the QGraphicsScene size.

If I had more time, I would have also liked to add more obscure algorithms to my application, so users can learn about them and compare them to the most popular algorithms. I

would also like to add more information about each algorithm within the application. It would be useful to have a timer that shows how long each algorithm takes to execute in addition to the number of comparisons/operations features. I would also probably add more visualizations or audio outputs to make the application more interesting.

Although my design could use some improvements, I think it is a successful application and meets the requirements for the scope of this project. I had fun creating it and learned a lot in the process.

Final Presentation

Link (also found on Canvas under Assignment 24): https://youtu.be/8HYPGi3eN0U

Resources

All external resources used for the final application are linked in my GitHub repository README file: https://github.com/snw0502/designproject3