Graph Project

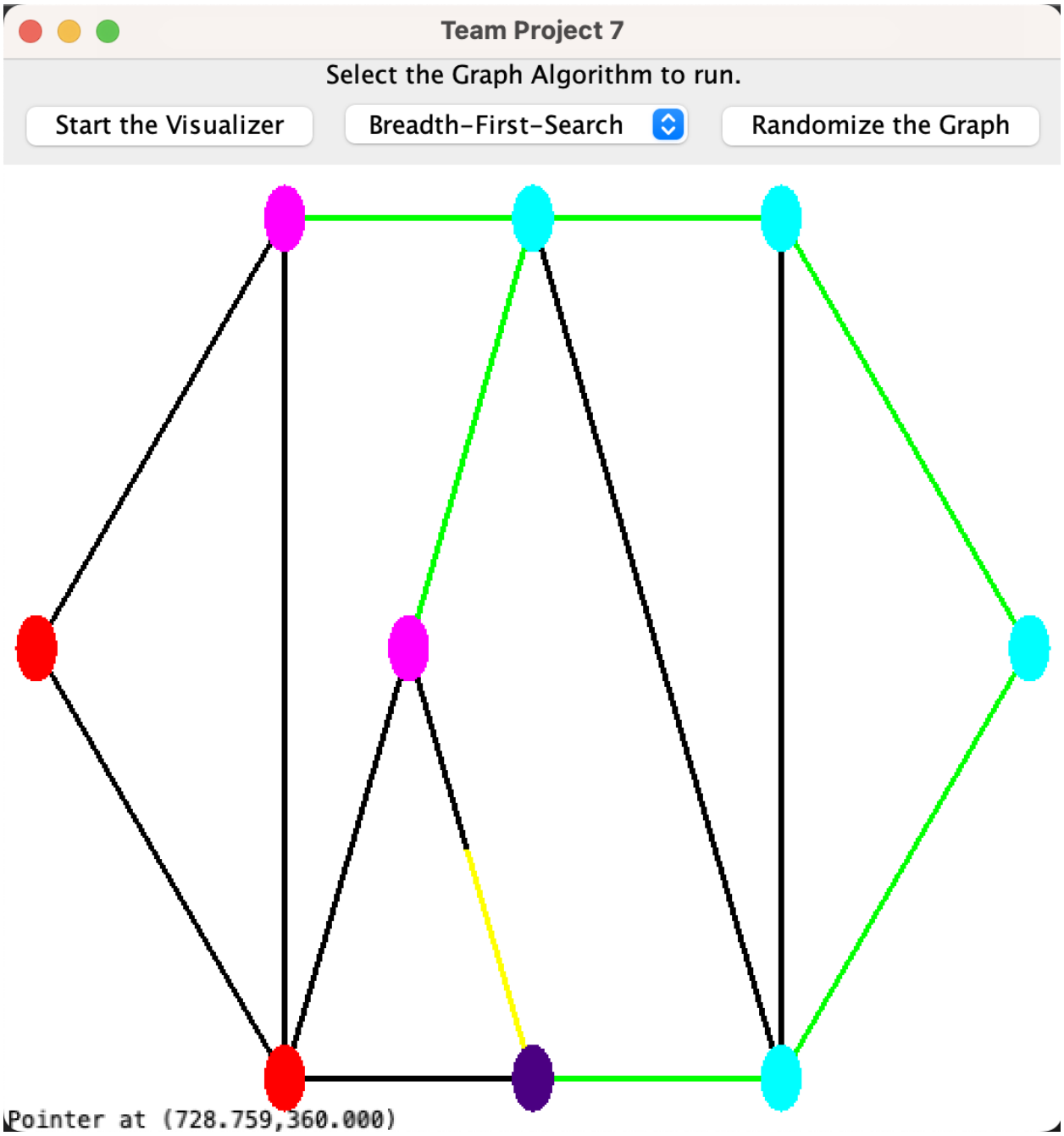
4/28/2023

Team 7

Brandon Michaud, Minh Dinh, and Trevor Scogin

During our algorithm analysis course, we identified a fundamental problem: Comprehending graph algorithms using only pencil and paper or textbook diagrams was quite difficult and required significant imaginative thinking. To address this issue, we decided to leverage the knowledge we hoped to gain from this graphics course to develop a solution. We intended to make a graph algorithm visualizer that showed, with respect to time, the algorithm physically traveling along edges and reaching nodes in breadth-first search, depth-first search, and shortest-path. The user could then randomize the graph, change the start node, and change the end node (in shortest-path) to see the effect it would have on the algorithm and better their understanding of the intricacies of the code. After the semester-long development process, our application presents the user with an interface that allows them to explore each algorithm step-by-step, making the learning process more intuitive and achieving our goals for the utility of the program. We hope that when other students take the algorithm analysis course in the future, they can use this tool to broaden their comprehension. Our solution demonstrates the merit of interactive tools in enhancing the learning experience and promoting a deeper comprehension of complex visual concepts.

Let us assume that a user is looking to develop a better understanding of how the breadth-first search algorithm works on graphs, and that they received a suggestion to use our program to aid in their learning. When they first launch the program, they are met with a default graph (see below figure) that was specially chosen to be a good representation of the three algorithms under different conditions. To specify they wish to run breadth-first search, the user can click the drop-down menu at the top of the program and select it. They can then click on a node to select it as the start node. This will make the node turn a dark blue color. Now that they have specified breadth-first search and selected the start node, the user can press the “Start the Visualizer” button on the left side of the tool bar. This will begin the breadth-first search traversal of the graph. Beginning from the start node and continuing from each other node, the current node will turn a dark purple, and yellow edges will extend to each other connected node. If the connected node has not been seen, it will turn magenta, indicating it has been seen, but not closed. The edge connecting to it will also turn green. Once the current node has checked each connected node, it will turn cyan, indicating it has been closed. The algorithm will run until all nodes have been closed. This process can be seen in the figure below:



Our general approach to this project was to split it into four main sections: Implementing a basic representation of graphs, both structurally and visually; Implementing the algorithms themselves; Adding convenience functionalities; and Testing different visualization techniques for the algorithms. This separation of tasks allowed us to split work amongst ourselves so that each person had much to work on while also being able to work semi-independently without having to rely too much on the work of another member. This maximized our production and made it so we did not have to fully understand the structure and approach of our teammates' code, but just the basic process.

We then further split these sections into tasks. The complete list of tasks is shown in the table below, including tasks we initially planned and decided to remove (shown in red), as well as tasks we did not plan on either the project plan or status report, but we thought of after and decided they would be good to add (show in blue):

Blue: New task post-Status Report Red: Task was scrapped

Set End Date	**Actual End Date**	**Task**
3/02	2/27	Represent graph structure visually
3/02	3/08	Add functionality for selecting start and end nodes
3/09	3/08	Implement graph algorithms
3/16	3/08	Illustrate the paths of the algorithms
3/16	3/30	Implement panning and zooming
3/28	4/22	Visualize backtracking (reworked)
3/28	4/01	Add interactive animation speed (including pausing)
N/A	4/25	Add menu for selection and starting algorithm
N/A	4/20	Add weighted edges, for shortest-path
4/04	4/22	Generate random graphs
4/13	4/25	Explore different visualization techniques
3/30	N/A	Implement lighting effects (Stretch)
4/18	N/A	Allow graph customization (Stretch)
4/13	N/A	Allow users to write their own graph algorithms (Stretch)

As shown in the table above, we started off ahead of schedule. This was a result of the work being related more to the graph algorithms themselves, rather than pure graphics. It was also helped by the fact that it was early in the semester and none of our group members had much to do for other classes. Once the more detailed and intricate graphics tasks were on the table, the work became more difficult, and we moved at a pace similar to the class, implementing things as we learned them.

Unfortunately, since we were implementing things as we learned them, we did not have enough time to implement any of the lighting or three-dimensional effects we planned. This became evident to us sometime before the status report, so we adjusted our focus appropriately to things we had learned how to implement. We had also planned to allow the user to create and delete nodes and edges, but after rigorous testing and feedback on our plan, we decided that feature should not be part of our core product. Likewise, we removed the custom algorithm task we envisioned in our plan because it was not graphics-oriented.

After reviewing our status report, we witnessed our project taking shape as we refined overlooked tasks and added new ones. We modified our backtracking task concept to align more closely with the actual algorithm mechanics. Alongside speed controls, we introduced a pause button as a minor enhancement. Furthermore, we integrated two key tasks: a top menu for seamless algorithm selection and execution, as well as edge weights for shortest-path problems, ultimately elevating the educational experience.

Overall, we would have liked to implement the tasks we removed, but they were either not aligned with the course goals or we did not possess the knowledge required. The most disappointing of them all was the lighting effects because we had imagined some pretty cool things, but we did our best with what we had and the new tasks we added certainly did a lot to help our project achieve our goal of being a useful educational tool.

The fundamental features of our program are the basic visual representation of graphs, the functionality of the algorithms, and the simple user interaction. These features are what we would expect to see from any other graph algorithm visualizer. Our interactions include a couple graphical elements for algorithm selection and a program start button. These interactions are accomplished by using JButtons and JComboboxes from Java AWT. Then, they are added into JPanels and finally into our JFrame with our desired alignments and order. We also included keyboard and mouse inputs for panning and zooming, a reset panning and zooming button, start and end node selection, visualization speed controls, a pause button, a toggle weights button, and a clear algorithm button. Altogether, these features provide the skeleton for a basic, but respectable graph algorithm visualizer.

Our program goes beyond these basics to offer a more unique and intuitive learning experience. Instead of simply drawing the final path that the program takes, we show the algorithm actually trying each edge to see if it can be added to the path. We distinguish between nodes that have tried each adjacent edge and those that have not by coloring them differently. We also color the probing edge differently than the actual path edges. This gives the user a unique insight into the inner workings of the algorithms. The user also has the option to generate a random graph. To do this, we made use of the vector mathematics we learned in class to project each non-endpoint node onto a potential edge and determine if they intersect, making it an ineligible edge. We also check to make sure that the graph is fully connected. This random graph functionality gives the user the opportunity to explore new graphs and deepen their comprehension.

We started our design process by simply defining nodes to be represented as circles and edges to be represented as lines spanning between those circles. Once we had the basic structure, we chose to make standard nodes appear red, and standard edges appear black. This provided adequate contrast for the graph on the white background. We then decided to add a drop-down menu to select the algorithm, a button to start the algorithm, and a button to randomize the graph. This allowed them to run algorithms without knowing any of the keyboard inputs. To maximize the information provided by illustrating the algorithms, we chose to draw edges in yellow and green, and the nodes in purple and cyan. This gave us enough different colors to illustrate the intricacies of each algorithm. We did not have excessive stylistic elements so as not to distract from the goal of educating the user about graph algorithms.

For our implementation, we used OpenGL with JOGL combined with a few Java libraries like Java Abstract Window Toolkit (AWT). We decided on this approach because we were already somewhat familiar with the JOGL library through attending the course lectures and working on the homeworks, and therefore would not need to spend significant time learning new interfaces/frameworks/libraries/game engines. This did, however, severely limit us in many aspects that would have been much simpler by using game engines like Unity or Unreal engine, due to their high levels of abstraction. For instance, our affine transformations of the camera using the OpenGL methods took a long time to perfect and caused us many issues when implementing panning and zooming. As for our organization, we copied the model-view-controller structure from the homework, with files for our keyboard and mouse handlers, the view, the model, and our application. We also added files for our Node, Edge, Graph, and SearchNode classes. These classes are essentially extensions of the model that abstract much of the graph information into a single variable in our model. We developed our project in Visual Studio Code. We chose this integrated development environment due to our familiarity with it, its lightweight design, intellisense, and easy-to-use debugging.

In order to successfully execute our program, the user only needs to meet a basic set of requirements. Starting with the most obvious, they need a device with mouse and keyboard input and a graphical output capable of rendering moderately complex graphics. Most modern laptops and desktop computers should work, but full specifications can be found on OpenGL's website. Next, the user needs to install a Java 8 JDK and specify the path in their environment variables. Once they have done this, they should be able to compile and run the program following the readme, because the gradle wrapper is provided.

Now that we have finished the project, we have a concrete list of the tasks that we planned but left unimplemented. We had to scrap lighting because we did not cover it in class. We also planned to implement graph customization but ultimately did not because we could not get it to work and because of feedback we received on our plan. Some things that we did not plan, but would like to add are more graph features like islands, directional edges, and self-loops. This would increase the educational value of our program. On the graphical side, we would also like to add three-dimensional graphs using meshes. This may not provide the easiest viewing experience but would serve to further our understanding of graphics, which was a goal of ours.

Along these lines, future development could take a couple of paths: Adding more algorithms and graph-related features to enhance the informative value of the program, or adding stylistic elements such as lighting and three-dimensional effects to improve the aesthetic.

Standard graph algorithm diagrams and drawings can often be confusing and jump between discrete points in time, making them difficult to follow. Our main goal for this project was to show these same algorithms under continuous time and add functionality so that the user can explore the behaviors of the algorithms and deepen their comprehension. Our finished product provides an in-depth look at breadth-first search, depth-first search, and shortest-path that gives users a unique learning experience that we would have liked to have when we first learned graph algorithms. In addition to accomplishing our primary goal, our team also gained a deeper understanding of modern-day graphics and the techniques that are used, achieving our goal of knowledge. Through this project, we learned how to transform, scale, and rotate objects as well as how to combine primitive shapes to create more complex illustrations, and devised solutions that employed our knowledge of higher-level mathematics. We also gained valuable experience in handling user input and designing interactive interfaces. Overall, our team is proud of what we have achieved, and we believe that our tool can be a valuable resource for students and other individuals who want to improve their understanding of graph algorithms.

We broke our team into roles as follows: Brandon wrote the algorithms, Trevor worked on the toolbar interaction, and Minh worked on the keyboard and mouse interaction. We then all worked on visualizing the algorithms together because that was the main focus of the project and required a diverse set of opinions to develop the most unique and informative product. This allowed us to work on multiple things at once while also producing a creative visualizer. These roles were not absolute, however, and at some point, each group member had their fingerprints on every aspect of the project.