Homework 3 write up Tdhall6@ncsu.edu

For homework 3 the objective was to add path finding to our sprites from homework 2, using pathfinding algorithms For part 1-3 I decided to use python to program and test the pathfinding algorithms for my graphs, and convert them to C++ for part 4.

Part1:

I decided to create a Jupyter notebook that randomly creates and tests graphs and edges. The notebook allows for me to view the graphs inline of the code and be able run sections of code without re-running all of it. I created a random graph generator that had parameters, like N the number of vertices, edge likelihood and number of edges. The program also has the ability to output to an excel file if needed. I limited edges for the higher vertices count to save on runtime. The graph output is visible using the graphs library for python, and is created when the cell for plot is run.

Part 2:

Dijkstra's Algorithm and A* are both graph search algorithms used for finding the shortest path between nodes. prioritize exploration, in different ways.

Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path from a starting node to all other nodes in a graph with non-negative edge weights.

How It Works:

Assign a tentative distance of 0 to the starting node and infinity to all others.

Use a priority queue (min-heap) to always expand the node with the smallest known distance.

For each visited node, update the distances of its neighbors if a shorter path is found.

Mark the node as visited and continue until all reachable nodes are processed or the target is reached.

A* is an optimized version of Dijkstra's algorithm that uses heuristics to guide the search towards the goal more efficiently.

How It Works:

Similar to Dijkstra, but instead of just using the current known distance (g(n)), A* adds a heuristic function (h(n)) that estimates the cost from the current node to the goal.

Nodes are expanded in order of the sum of both values:

$f(n) = g(n)+h(n)$

The heuristic helps prioritize nodes closer to the goal, reducing unnecessary exploration, making it faster for longer mor complicated graphs.

Part 3:

For the two Huristics I chose Manhattan and Euclidean distance

Euclidean Distance (Straight-Line Distance)

Euclidean distance measures the shortest direct (as-the-crow-flies) distance between two points. It is calculated using the Pythagorean theorem: h(n)=(x2−x1^)2+(y2−y1)^2

Manhattan Distance (Grid-Based Distance)

Manhattan distance measures the cost when movement is restricted to horizontal and vertical directions (like moving through city blocks). It is calculated as:

$h(n)=|x2-x1|+|y2-y1|$

I made a mistake with my programing in that none of the vertices have coordinates so the relevancy is not that good for times. The a* algorithms iterated through the entire graph, so it was likely a worst case scenario for all of them. Dijkstras was always the fastest followed by Manhattan. The 15000 vertex graph took 10 minutes to create and was visually unreadable, it looked like a black dot.

Part4:

To enable the sprite to navigate the maze in SFML, I first structured the environment as a graph, where each node represented a valid position in the maze, and edges connected navigable paths. I then implemented Dijkstra's Algorithm and A Search Algorithm* as separate classes in cpp and hpp file to compute the shortest path from the sprite's current position to the target.

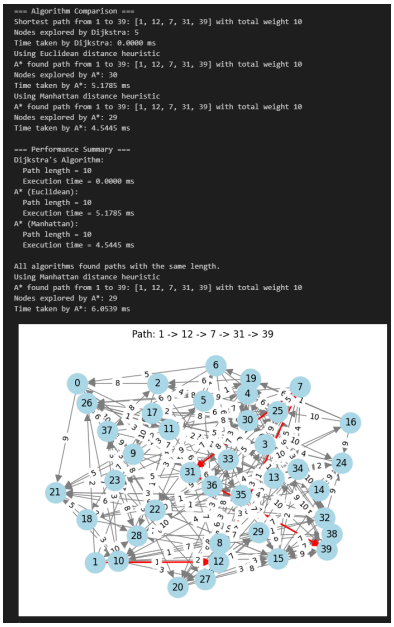
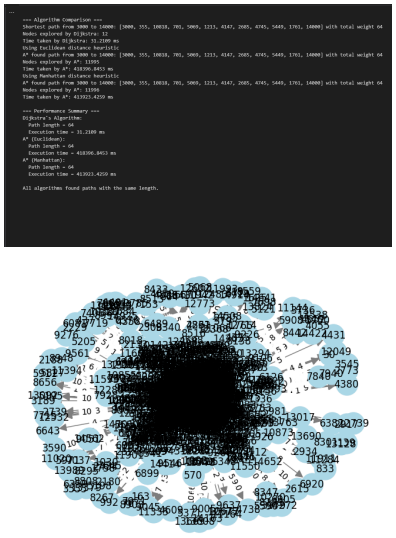
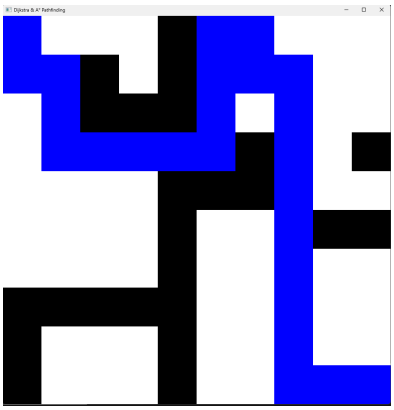
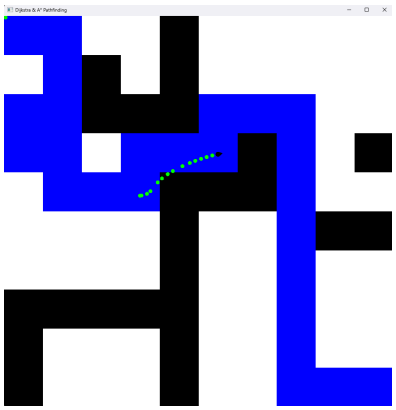
Once the path was computed, we stored the sequence of waypoints that the sprite needed to follow. During each update cycle, the sprite checked its position relative to the next waypoint. Using Arrive and Align steering behaviors, the sprite adjusted its velocity and orientation smoothly toward each waypoint rather than making abrupt directional changes. If an obstacle blocked a path dynamically, the algorithm recalculated a new route in real time. Through this approach, we created a responsive navigation system that allowed the sprite to traverse the maze efficiently while maintaining realistic movement..

The actual map itself is stored as a 2d vector, 0 represents open nodes that the sprite can travel upon while 1 represents closed nodes that the sprite cannot access. After the map is created the positions of and the boxes that the nodes fill is dependent upon the number of vertices and the size of the window. More vertices = smaller squares. The blocked nodes(walls) are shown as black squares. The path that the sprite takes is shown in blue. For my testing I use100 vertices and set the start point as 0,0 the first vertex and the finish as 9,9 the last vertex. The way I set up the path finding is, first a search algorithm is chosen (Dijkstra's Algorithm , A*(Manhattan), or A*Euclidean), then it creates the best possible path to complete the maze. For the pathing of the sprite itself, I used the code for following mouse clicks similar to what we did for home work 2 as a base. An array of all of the vertices in the calculated path is made, then when the sprite starts the next vertex in the array. When the sprite approaches the array the target switches to the next vertex in the array until the end is reached.

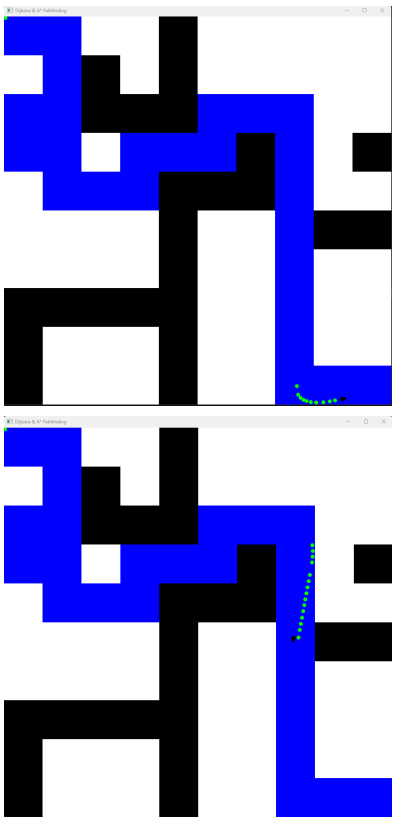
Conclusion:

Through this project, I explored the implementation and comparison of shortest path algorithms in both a Jupyter Notebook environment and an SFML-based maze navigation system. By generating and visualizing random graphs, I was able to test Dijkstra's Algorithm and A* Search using different heuristics—Manhattan and Euclidean distance. However, a limitation arose from the fact that my graphs lacked explicit coordinates, which affected the effectiveness of A*heuristics. As a result, A* often performed worse than Dijkstra's algorithm. With the SFML implementation, I successfully structured the maze as a graph and integrated both Dijkstra's Algorithm and A* for pathfinding. The sprite navigated the environment dynamically, adjusting its movement using Arrive and Align steering behaviors. The system also handled real-time path recalculations in response to obstacles, ensuring flexible and efficient navigation. Overall, this project provided valuable insights into pathfinding in different contexts—random graphs and structured mazes. Future improvements could include optimizing A* by incorporating more relevant heuristics and refining the sprite's movement to enhance smoothness and responsiveness.

Homework 3 write up Tdhall6@ncsu.edu

40 Vertex path	15000 vertexpath
	
Maze	Maze with sprite pathfiding
	

Homework 3 write up Tdhall6@ncsu.edu



Sources:
A* Search Algorithm GeeksForGeeks
https://www.geeksforgeeks.org/a-search-algorithm/
Dijkstra's shortest path algorithm in Python GeeksForGeeks
https://www.geeksforgeeks.org/python-program-for-dijkstras-shortest-path-algorithm-greedy-algo-7/
Claude AI
Chat GPT