CSC 584/484 Spring 23 Homework 3
Kuangyao Mai – kmai

Part1:

For this part of the assignment, I first create a graph that is relatively small. The graph I created represents a map from the video game "Monster Hunter World" called "Elder Recess." The graph consists of 26 vertices and 73 edges, most of which are bidirectional. To create this graph, I first found a picture of the map and identified the locations that would be represented as vertices. Then, I used Microsoft Paint to create a grid on top of the map picture to help me identify the exact locations of the vertices. After determining the vertices, I manually drew the edges and determined the weights between them. The weights were based on the distance and terrain. For instance, some parts of the map are closer to lava or monster habitats, which makes them more challenging to traverse and have higher weights. Conversely, some areas are less hostile and easier to navigate, resulting in lower weights. To determine which locations should be represented as vertices in the graph, I considered their significance within the game world and their connectivity. Some locations include resource points, campsites, and areas that serve as choke points or necessary traversal points that make sense within the map layout. I also avoid impassable places as they would not contribute to the navigability of the graph. In addition, there are also locations where it can perform fast travel, which has a weight of 1 and can travel across the map. I chose this map to represent the graph because of its overall layout and structure. I think the map has many distinguishable areas that are connected logically and intriguingly. Furthermore, I think this map is suitable for Path-finding analysis as it presents a real-world scenario where pathfinding is commonly seen in video games. Just like in Monster Hunter World, when the player selects a location on a map, it will provide a path that guides the player. Therefore, I think this graph will be good practice for this assignment.

To represent this map as a graph, I created a Graph class. The implementation was inspired by the Graph data structure presented on redblobgames.com (https://www.redblobgames.com/pathfinding/grids/algorithms.html). The Graph class contains a map that stores the vertex id as the key and a map of edges as the value. The edges map uses a string as the key to represent the adjacent vertex and a float as the value to represent the weight of the edge between them. This allows for easy insertion of edges between vertices, as I can simply write "edges[src][dest] = weight;." Since I drew a grid on the picture to create the graph, each location has an assigned coordinate to support the use of a heuristic function in the A* algorithm. These coordinates are stored as a pair in a map within the Graph class, where the key represents the vertex, and the value represents the pair of coordinates. Then, I also created a function called printGraph which helps me ensure that what I am making has a valid result. The tedious parts of creating this small graph are probably the fact that I have to manually input the coordinates and insert the edges to the map, and physically draw them on paint. For the large graph consisting of tens or hundreds of thousands of vertices and four or five times that number of edges, I created a function that randomly generates one. Due to difficulties importing libraries into the system and uncertainty about the grading environment, I took inspiration from this website https://github.com/edervishaj/strongly-connected-components. This function takes in four parameters: numVertices, weightMin, weightMax, and edgeProb. This function utilizes the Erdos–Renyi model to randomly generate edges between vertices with a given probability (edgeProb). In addition to generating the edges, the function assigns random coordinates to each vertex within a square region. The graph generator function has a fixed seed of 1, which is used for testing purposes. If we want a completely random graph, we can use the current time by using the time().

Part 2:

For the next part of the assignment, I implemented two pathfinding algorithms, which were A* and Dijkstra's. Just like in the previous part, I followed the algorithm pseudocode provided in the textbook and on redblobgames.com. The two algorithms are essentially identical except for one line of code, where A* considers the heuristic function when calculating the cost. Then, to measure the performance differences between the two algorithms, I utilized the built-in SFML library to calculate their runtime.

Small Graph: 1 to 23, Big Graph: 10000 & 30000 vertices, 1-20 weight, 0.1 edgeProb

Based on my output, the A* algorithm took 0.000039 seconds to complete for the small graph, while Dijkstra's algorithm took 0.000061 seconds. Although they give the same path, A* is almost 1.5 times faster than Dijstra in this case. In this case, A* is faster because the heuristic function allows A* to prioritize the path that has a higher probability of leading to the goal. In contrast, Dijkstra's algorithm considers all possible paths from the start node to every other node, making it more computationally expensive. However, when I ran the test on the randomly generated large graph with 10,000 vertices, a probability of 0.1 for edge creation, and weights ranging from 1 to 20, the results were slightly different. The runtime for A* was 0.008131 seconds, while Dijkstra's algorithm took 0.447942 seconds. But in this case, Dijkstra provides a more optimal path with a cost of 3, while A* provides a path with a cost of 9. This also happened when I generated a graph with 30000 vertices (same weight chance and edgeProb). A* took a shorter time but did not provide a better path than Dijkstra. In a large graph like this, A* didn't have to explore as many nodes as Dijkstra's algorithm since A* uses heuristics to prioritize exploration. However, this can result in A* skipping some vertices and potentially finding a less optimal path but with faster performance.

For implementing heuristics functions, I created two, one of them being admissible and one of them being inadmissible. This is because the graph was created on a grid, so the weight between two points is measured step by step with extra weights based on the terrain and environment it is in. Therefore, the straight-line distance will always be lower than the actual weights. Then, I created an inadmissible function that modifies the Manhattan distance. I first calculated the absolute differences between the x and y coordinates. Additionally, I also made other factors that are added to the estimate. The first one is the number of unvisited nodes. This means that if there are more unvisited nodes, it is more likely that the path will have to go through them, and thus the estimate will be higher. The second factor is the average number of neighbors of the unvisited nodes. This means that if there are more neighbors, it is more likely that there will be more potential paths, and thus the estimate will be higher.

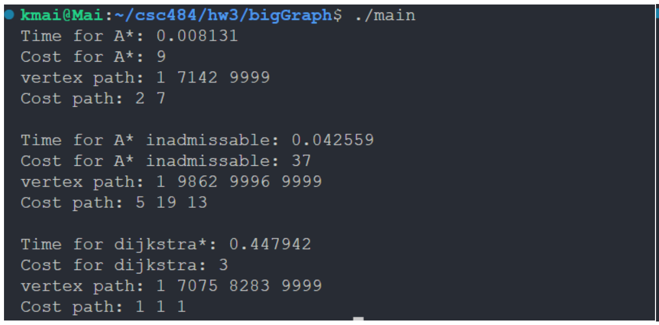
Based on the performance output, the admissible heuristic performed better than the inadmissible one, as expected. Admissible has a runtime of 0.000039 seconds, compared to

0.000228 seconds. The inadmissible heuristic was designed to overestimate more often because of the factors it includes. This makes it always less optimal in terms of performance and is more evident when running on a large graph. However, since the coordinates for the large graph were randomly generated, the results for this heuristic on the large graph have some uncertainty, and the data may not be as reliable. Nonetheless, the performance difference between the admissible and inadmissible heuristics was drastic and even more noticeable in a bigger graph.

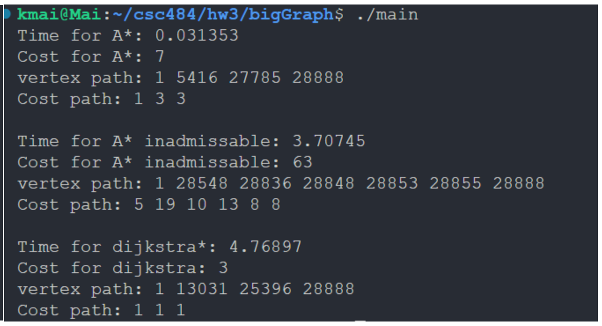
Part 3:

The next step of the assignment involves combining pathfinding and steering behavior to demonstrate path following. To get started for this part, I created an environment that defines the map. Given that the window size is 640 x 480, I decided to divide it into tiles, each of size 32 x 32 pixels, representing them as vertices. To visualize them on the screen, I used rectangles in SFML. Once the grid was set up, I used Paint again to color code the rooms and obstacles before actually setting the tiles' color. The next step was to create the actual graph that represents this map. Every tile, except for the "wall," was added as a vertex, and their id was set to the same as their index in the list of tiles. Then, every tile will be connected to its neighbors with a weight of 1, and diagonal connections will only be made if the tile does not have a wall next to them, preventing the boid from going through walls. The coordinates of the vertex will also just be the tile's position. Then, I used SFML Mouse to determine which tile the mouse was currently on. I implemented the function getClosestVertex, which takes in the graph, a vector of tiles, and the mouse position. The function will iterate through the vector of tiles and check if the mouse position is inside the tile's boundaries. Since the index for the tile list is the same as the vertex id, I can just easily return the index and consider that as the vertex id. A similar approach was also used to determine what tile the boid was on. With the tile that the boid is currently on as the start and the tile that the mouse clicked set as the target, I then pass these two vertex ids as arguments into the pathfinding algorithm, which returns a list of vertex ids representing the path. Then I can just use the coordinates map from Graph to find out the actual position of the tile and delegate to the Arrive steering behavior. After the boid has arrived at the first vertex, an index counter called pathIdx will increase, indicating that it should move on to the next vertex in the list. Overall, I believe the results worked out the way I intended. The boid was able to follow the path without going through the obstacles, but its movement was not as smooth as I had hoped. Because the vertices are located right next to each other, the boid is constantly using Arrive, resulting in a slight stopping effect when it reaches the next vertex. I made attempts to improve the boid's movement by adjusting its velocity and radius of satisfaction. However, increasing these values too much resulted in erratic behavior where the boid flew around. Thus, further adjustments are required to achieve smooth movement for the path following. One potential solution could be implementing a predictive version, which involves predicting the future location and selecting the next vertex accordingly.

Big Graph:

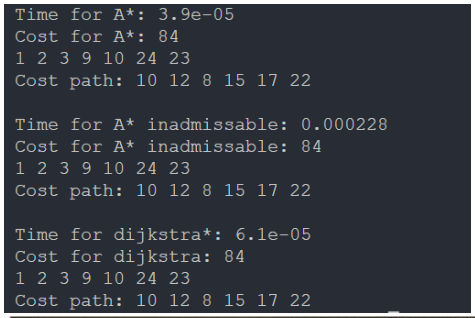


```
kmai@Mai:~/csc484/hw3/bigGraph$ ./main
Time for A*: 0.008131
Cost for A*: 9
vertex path: 1 7142 9999
Cost path: 2 7

Time for A* inadmissable: 0.042559
Cost for A* inadmissable: 37
vertex path: 1 9862 9996 9999
Cost path: 5 19 13

Time for dijkstra*: 0.447942
Cost for dijkstra: 3
vertex path: 1 7075 8283 9999
Cost path: 1 1 1
```

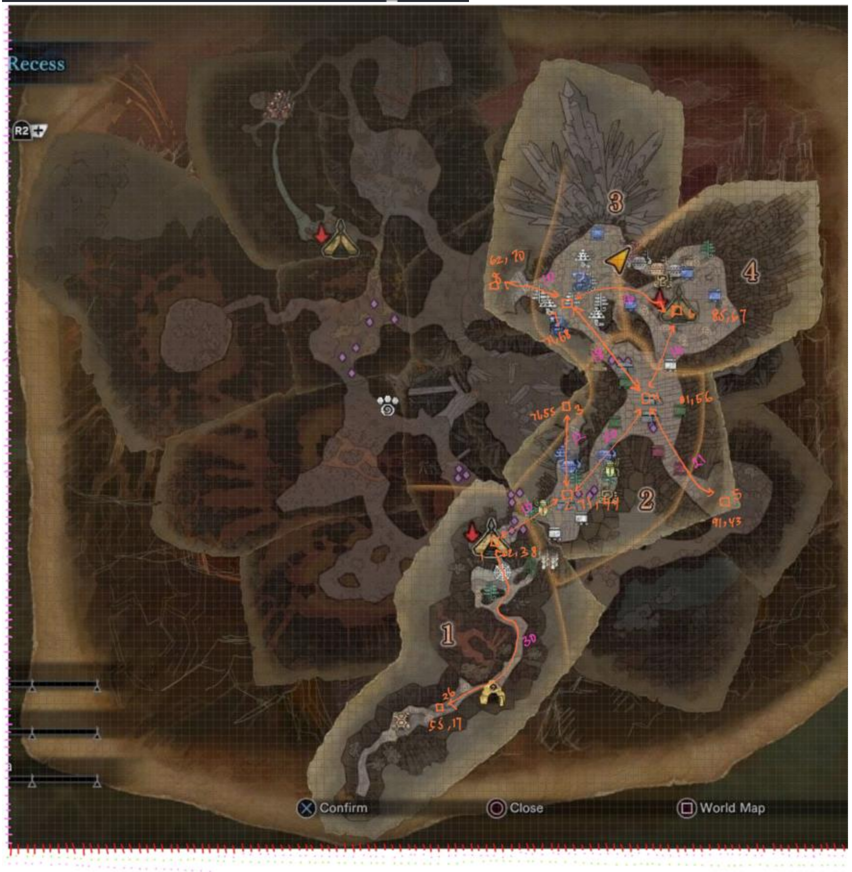


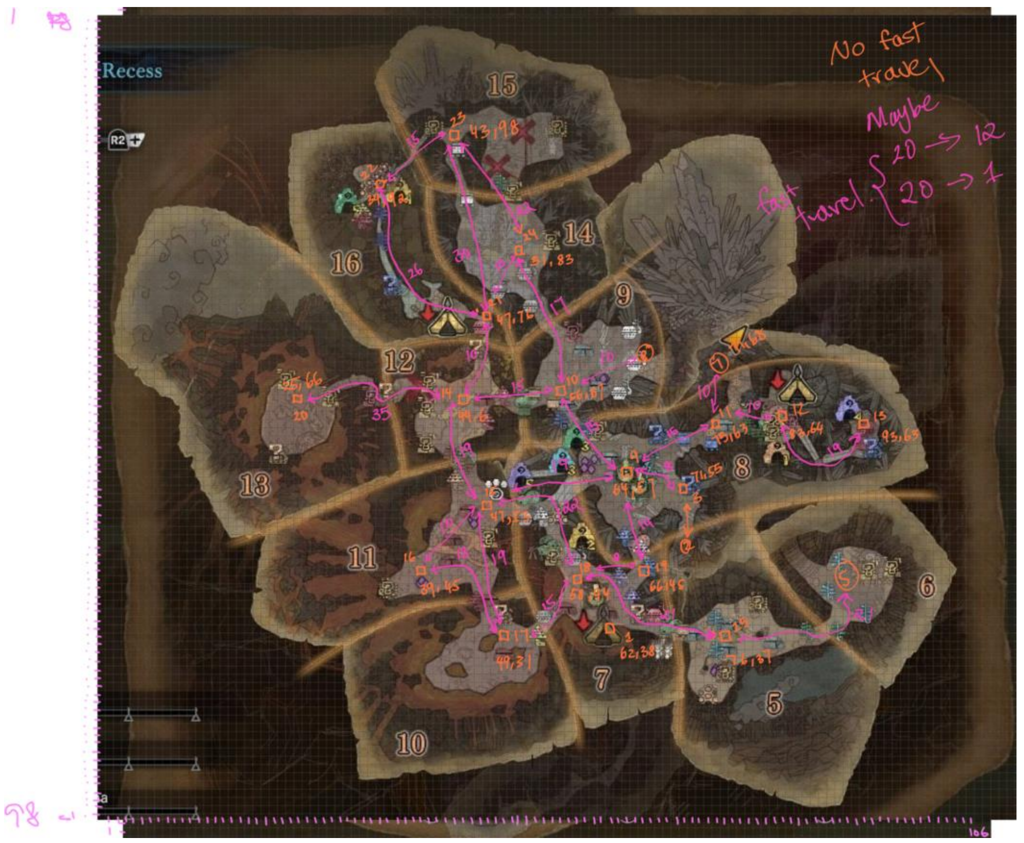
```
kmai@Mai:~/csc484/hw3/bigGraph$ ./main
Time for A*: 0.031353
Cost for A*: 7
vertex path: 1 5416 27785 28888
Cost path: 1 3 3

Time for A* inadmissable: 3.70745
Cost for A* inadmissable: 63
vertex path: 1 28548 28836 28848 28853 28855 28888
Cost path: 5 19 10 13 8 8

Time for dijkstra*: 4.76897
Cost for dijkstra: 3
vertex path: 1 13031 25396 28888
Cost path: 1 1 1
```

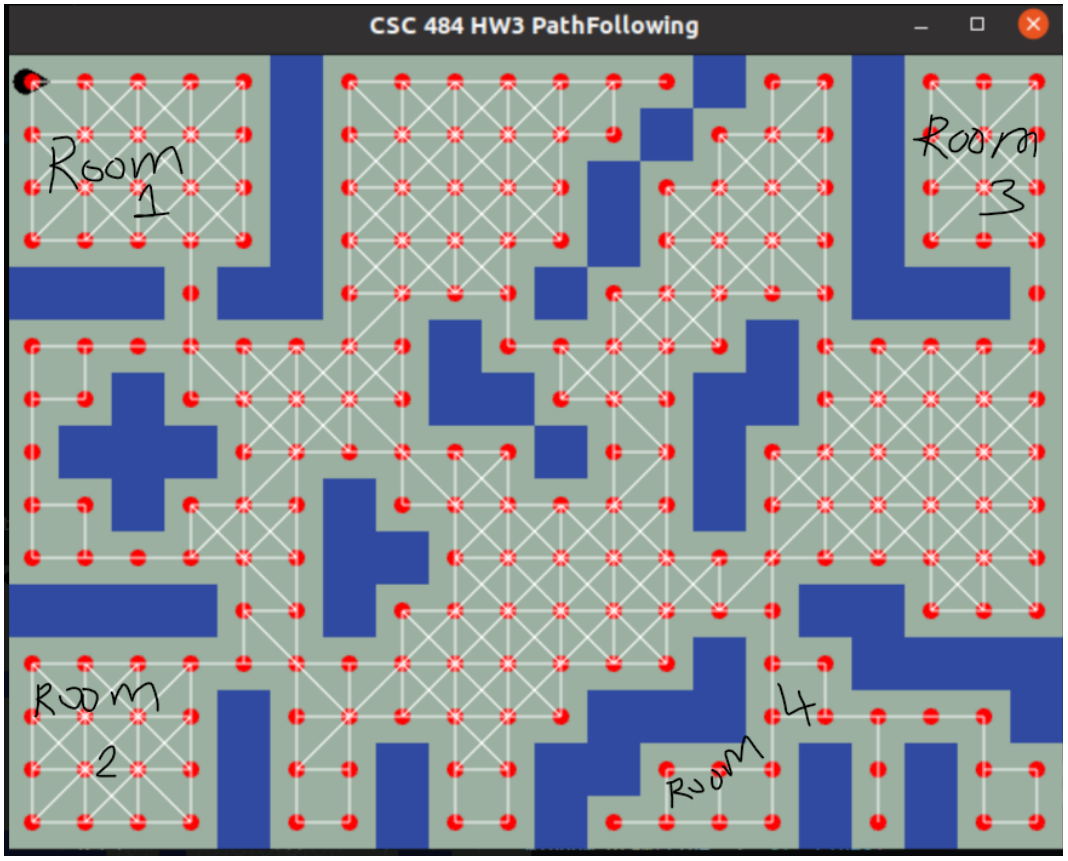
Small Graph:

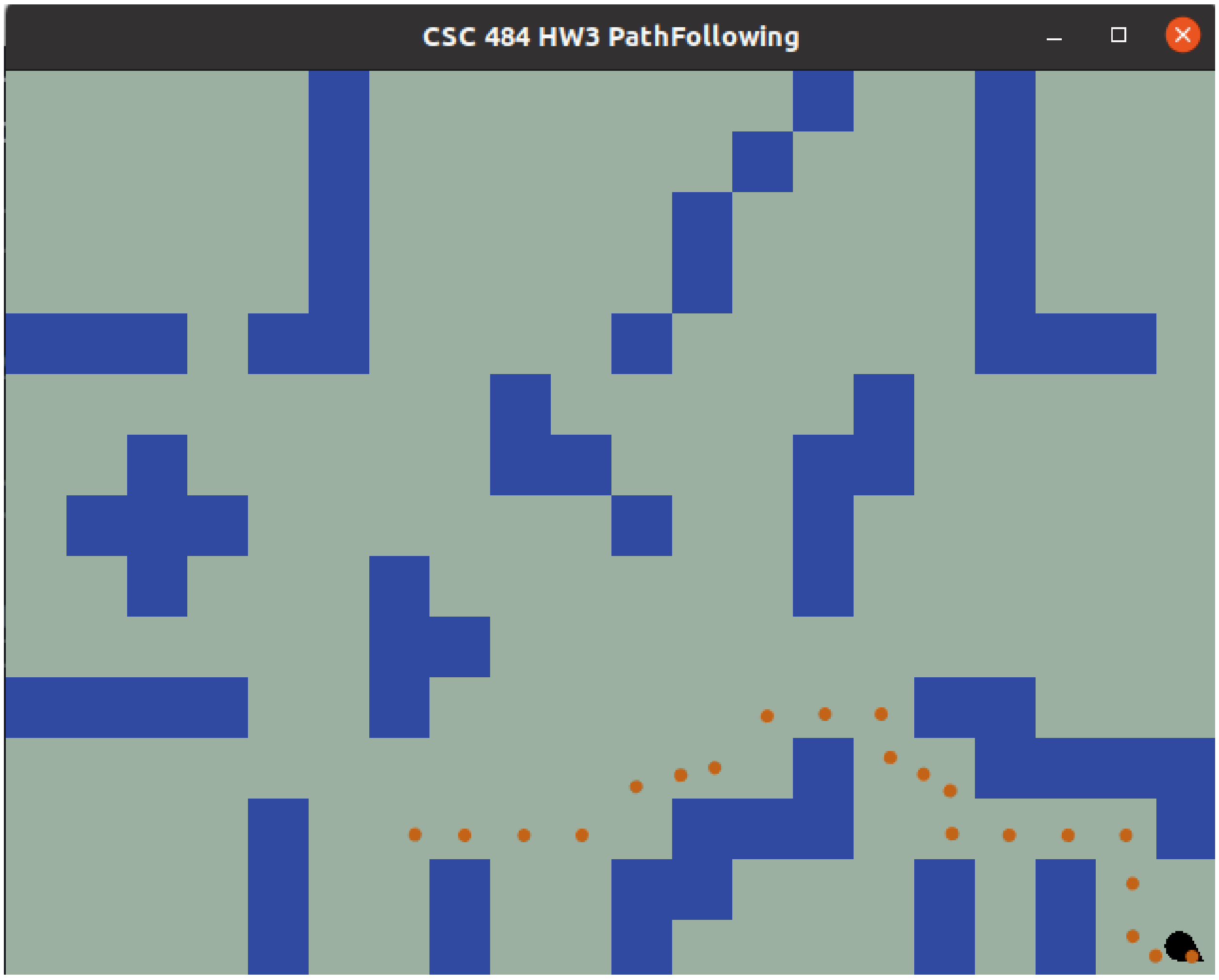


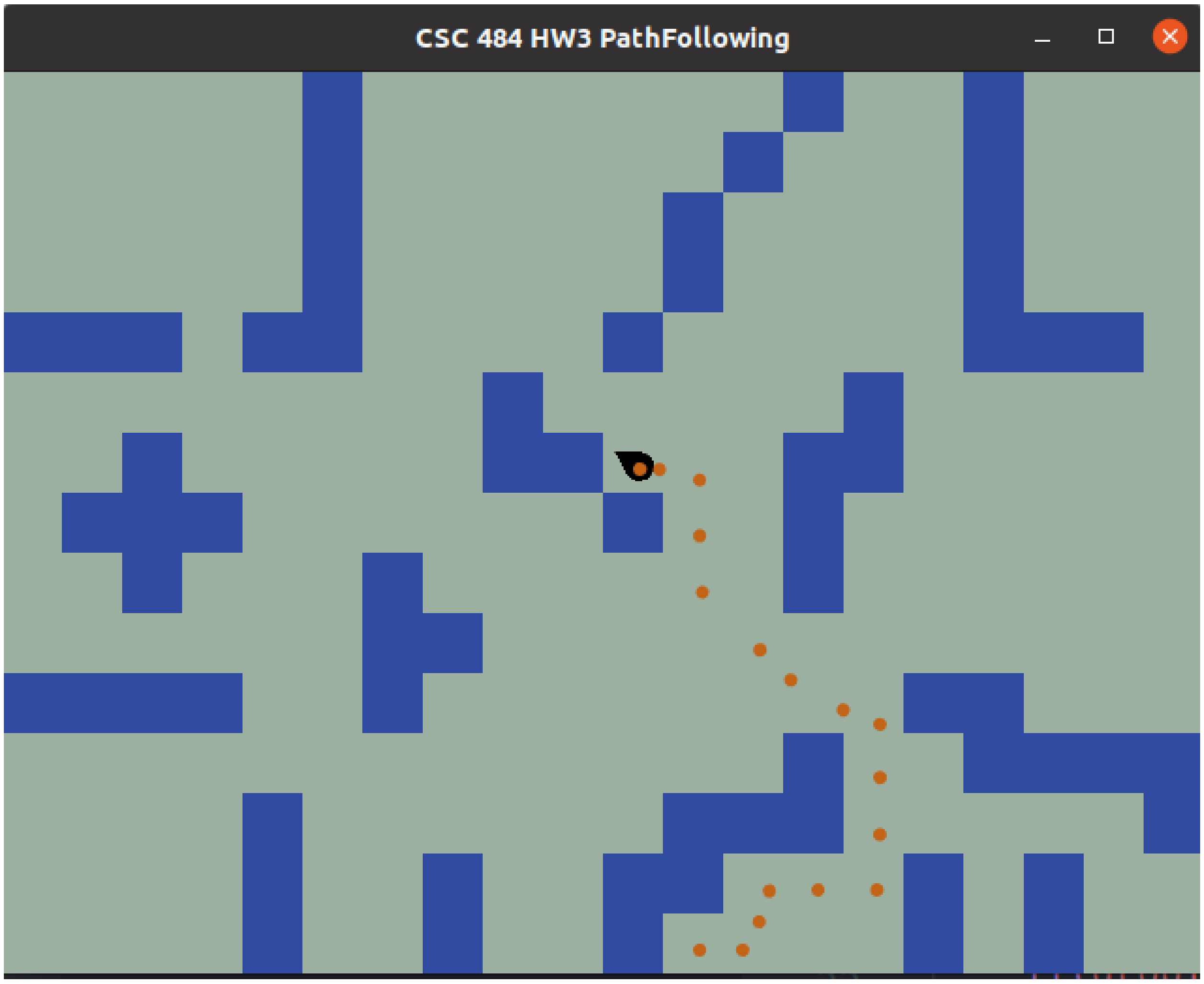
```
Time for A*: 3.9e-05
Cost for A*: 84
1 2 3 9 10 24 23
Cost path: 10 12 8 15 17 22

Time for A* inadmissable: 0.000228
Cost for A* inadmissable: 84
1 2 3 9 10 24 23
Cost path: 10 12 8 15 17 22

Time for dijkstra*: 6.1e-05
Cost for dijkstra: 84
1 2 3 9 10 24 23
Cost path: 10 12 8 15 17 22
```

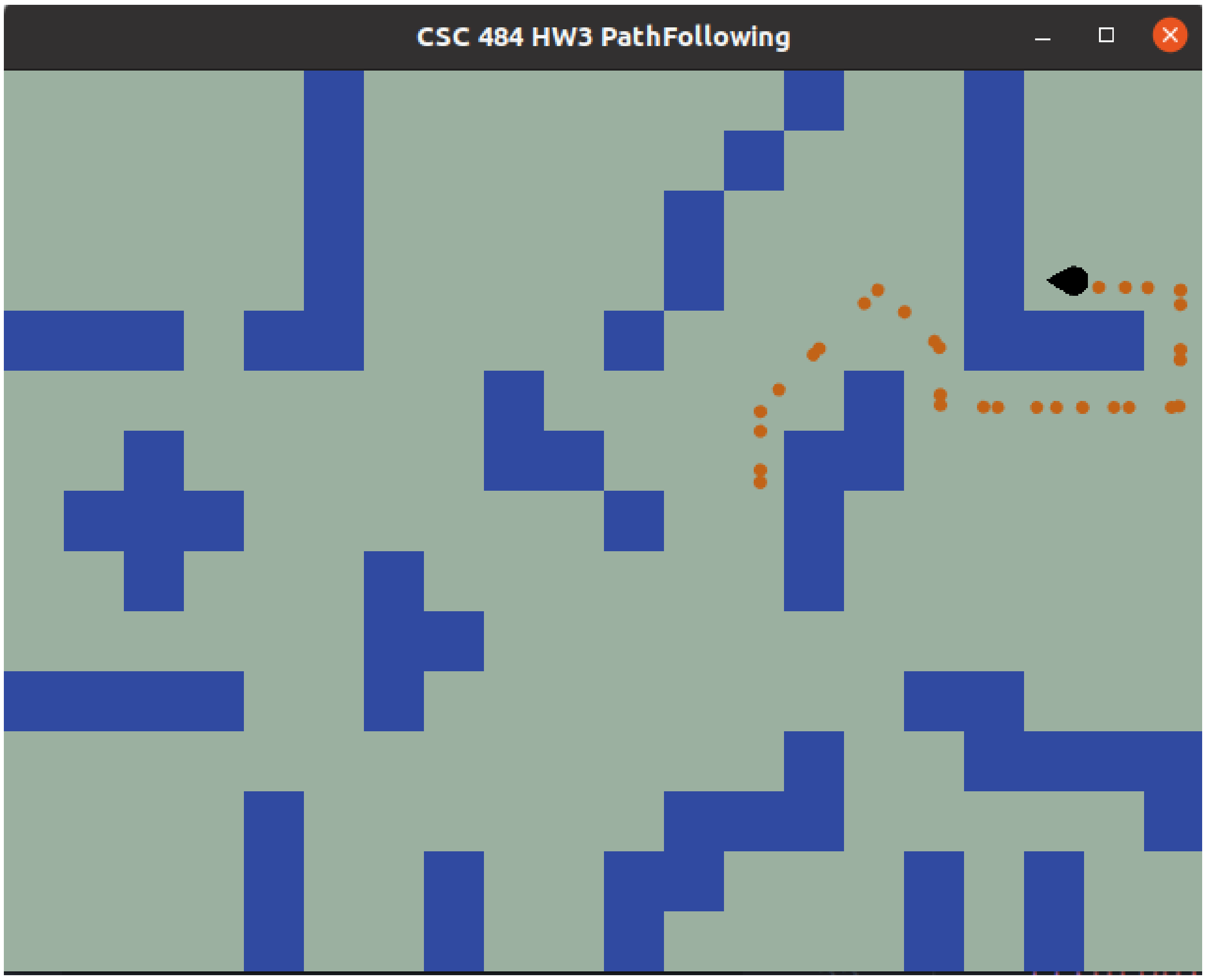



No fast travel
Maybe
fast travel: 20 → 12
20 → 1
Recess
15
14
16
9
12
13
8
11
6
7
5
10
1
98
106


CSC 484 HW3 PathFollowing
Room 1
Room 3
Room 2
Room 4


CSC 484 HW3 PathFollowing


CSC 484 HW3 PathFollowing


CSC 484 HW3 PathFollowing