Samuel Hodges

CSC 584 HW3

The purpose of this homework was to demonstrate graph path-search algorithms on datasets as well as to implement a pathfinding agent for a maze-like indoor environment which utilizes Craig Reynolds path following algorithm. I have completed these goals and provided a README.md file which details how to test and verify my code. This writeup will cover the specified discussion points from the homework description and provide details on implementation decisions.

The **First Steps** section for this homework required that I construct weighted digraphs to be later utilized by the graph path-search algorithm. The first required graph must be "something meaningful in the world". To this end, I have constructed a graph which details the straight line walking distance between HarrisTeeter Grocery Stores in the Triangle Area. Additionally, I have connected Centennial campus (as represented by node 0 in the diagram) to the graph such that students and faculty may use the graph path finding algorithm to navigate to and from stores in an optimal fashion. This project will no doubt provide a valuable service to many. For my graph representation, I have chosen to represent weighted edges in an adjacency list (found in graphFiles/*_edges.csv) and the individual graph vertices in lists (found in graphFiles/*_nodes.csv). Later in the project, I will use the manhattan and euclidean heuristics for my A* graph search. As such, the node lists include x,y coordinates for each node. In the case of the HarrisTeeter Grocery Store graph (found in graphFiles/custom_edges.csv and graphFiles/custom_nodes.csv), I used google maps in order to get x,y coordinates as well as straight line measurements between stores. For the HarrisTeeter graph, I gathered data on 20 stores and Centennial campus for a total of 21 nodes. Additionally, I connected nearby stores such that the graph has 80 edges (bidirectional edges count twice). From a technical standpoint, I chose the HarrisTeeter graph because it used real world location data meaning it is a valid subject for the Euclidean heuristic. For my second graph (the big graph), I have devised 2 methodologies for generating graphs. The first methodology is to generate some number of nodes, assign random edges between nodes, and to assign random coordinates and edge weights between the nodes. This first methodology (found in scripts/randomDigraph.py) is capable of rapidly generating a graph but lacks any meaningful way of improving search performance via a heuristic given that the data is random. For this type of graph, I have devised the "Many-Paths Heuristic" which I will elaborate on in the **Heuristics** section of this paper. My second methodology for graph generation is to create a geometric graph of nodes with nodes and edges that correspond to coordinate information. This second methodology works by first generating nodes with random coordinate locations and then for each node, creating an edge (of weight distance) to each other node within a maximum radius. This second methodology for generating nodes and edges is extremely resource intensive for large numbers of nodes since each node must calculate its distance from every other node in the graph. For speed improvements, I have utilized the python multiprocessing library. When generating a graph of 10,000 nodes, the script utilizes upwards of 24 Gigabytes of Memory. You can see this script in action by choosing "s" when prompted by the Main executable.

The **Dijkstra's Algorithm and A*** section had us implement both pathfinding algorithms for our graph data. Although A* is often described as an extension of Dijkstra's algorithm, I found it helpful to think of Dijkstra's as being an implementation of A* in which the heuristic always returns 0. In this way, I have devised a search function titled shortestPath which accepts a graph as input as well as any heuristic object. In my code, Heuristic is an abstract class which is implemented by all of my unique heuristic schemes. Also, in my code, when I describe my A* heuristic I am referring to the euclidean distance even though A* is independent of the heuristic. For my performance comparisons, I timed the entire search process using the built in chrono package. For consistency purposes, I used the same start and finish nodes for each test. After gathering timing data, I then re-ran the code with Valgrind in order to measure how many bytes of memory were allocated by the program. I have compiled my measurements for all heuristics in Table 1 and Table 2. For the Table 1 data, A* only provides a marginal search speed improvement. In both Dijkstra's and A*, the number of nodes visited is the same which indicates that A* has not improved the search for this graph. Considering that the graph was hand generated, this may indicate a possible error in my graph files. Although the runtime for A* is faster than Dijkstra's, both values are within a small enough range such that the difference may be ignored. For more expected test results, we turn to Table 2. Since the data for Table 2 is algorithmically generated, we can be slightly more

confident in our graph's correctness. For runtime, we see that A* roughly halves the run time when compared to Dijkstra's algorithm. Additionally, the number of nodes visited in A* is roughly a fourth the number of nodes visited in Dijkstra's. This combination of results follows everything we know about A*. Although A* requires that the Euclidean distance be estimated for each node visited, far less nodes are visited overall. The net result is therefore a speedup. As for the memory usage, less nodes visited also results in less memory needing to be allocated overal. Although this difference is mostly negligible when compared to the large amount of memory required to load the csv of edges and nodes. Lastly, both algorithms find the optimal solution of path length 1460.

For the **Heuristics** section, we were tasked with creating **two heuristics** for A* (obviously not including dijkstra's). For this section, I have created **three heuristics.** The **first heuristic** is the euclidean distance. The euclidean distance is simply a measure of the straight line distance between two points in a coordinate system. In the two dimensional case, this is simply the distance formula $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ where the current node's coordinate is (x_1, y_1) and the goal node's coordinate is (x_2, y_2). For a coordinate system based graph, the euclidean distance is both admissible and consistent. As long as the weights between nodes are never less than the distance between those nodes, this heuristic will never overestimate which makes it admissible. My **second heuristic** is the Manhattan distance. The Manhattan distance is equal to the formula $|x_2 - x_1| + |y_2 - y_1|$. In other words, the Manhattan distance is the absolute change in x plus the absolute change in y. In the case of my coordinate graphs, the Manhattan distance is not admissible since it is capable of over estimating the path length to the target. If my graph were a coordinate grid in which edges were only vertical or horizontal, then the Manhattan distance would be admissible since it would never overestimate (such as for diagonal edges). From Table 2 we can see that the Manhattan distance does a really good job of finding a path quickly. The only problem is that the path found is a bit longer than the optimal path found by dijkstra's and A*. Interestingly, the Manhattan-based search only needs to visit 70 nodes before it finds an answer. I suspect this may just be a coincidence, but it is interesting. My **third heuristic** is the Many-Paths heuristic which is something I made up. My goal was to devise a heuristic which might perform well on completely random graphs but did not require a bunch of upfront preprocessing (such as with the cluster heuristic). To this goal, I devised a heuristic which favors nodes with more edges. Before the search begins (but after the timer is started) the Many-Paths heuristic first adds up each edge weight and then divides by the total number of edges so as to learn the average edge cost. The $h()$ function is then equal to the $average\_edge\_cost\ /\ (1\ +\ number\_node\_edges)$. Since the maximum $h()$ value is simply the average edge cost, this heuristic should theoretically very rarely overestimate (it's more likely to overestimate close to the goal node). As one can see from Table 2 the Many-Paths heuristic performs very poorly. Although the table only shows geometric data, in every test I ever performed (including non-geometric totally random data), Many-Paths always tended to perform worse (runtime) than Dijkstra's algorithm and always needed to visit every node anyways. My suspicion/explanation for this is that since Many-Paths only ever contributes an edge length or less to the $f(n)$ calculation, the algorithm pretty much ends up just doing Dijkstra's search but with added computations to slow it down.

For the **Putting it All Together** section, I have devised a python script for generating random indoor environments of rooms and doors. The python script starts with an empty room as represented by a 2D array with "#" marks representing the 4 outermost walls. The script then recursively divides the rooms into smaller rooms and somewhat randomly places doors. When a recursion reaches its maximum depth, it fills in the room with an ID number. The entire 2D array is then saved to a CSV file to be read by the c++ program. As of a week ago, I introduced a bug into the python script which occasionally (maybe one in 3 times) causes the main loop to segfault upon reading in the CSV. The solution is to simply regenerate the indoor map until the program succeeds. When the c++ graph builder reads in the CSV file, it creates a graph node for each door that it reads in. For each door, it creates edges between any other doors which share that node's room. By having the python script fill in the rooms with an ID number, it makes it easier (but not easy) to identify which doors belong to which rooms. This process of creating the graph from the

csv file was a big pain, but ultimately paid off since the agent always goes in an optimal straight line path in rooms. While running, the user may hit the space-bar in order to overlay the navigation graph as well as some debug information such as room and node IDs (see **Screenshots** section of **Appendix**). Had I gone with a 2d grid based implementation (wherein the map is a 2d grid of node tiles), the agent would only have been able to travel in 8 different directions (vertically, horizontally and diagonally). What this would mean is that the agent would be incapable of taking the optimal path if that path were any non-standard angle. Since I do not use a grid approach, my paths are always as short as possible. Unfortunately, since the agent always travels in optimal straight lines, its sprite can clip into walls if the optimal path has it moving close to a wall. In robotics, a costmap is used to guide the robot's navigation away from objects which it may accidentally collide with. In future work, I may implement a costmap which discourages the agent from getting too close to the wall. Alternatively, I could implement a raycasting system in which the agent is made aware of its proximity to walls. Since either of these options is beyond the scope of this homework, I have instead left the code as is.

Appendix:

Image 1: Small Graph

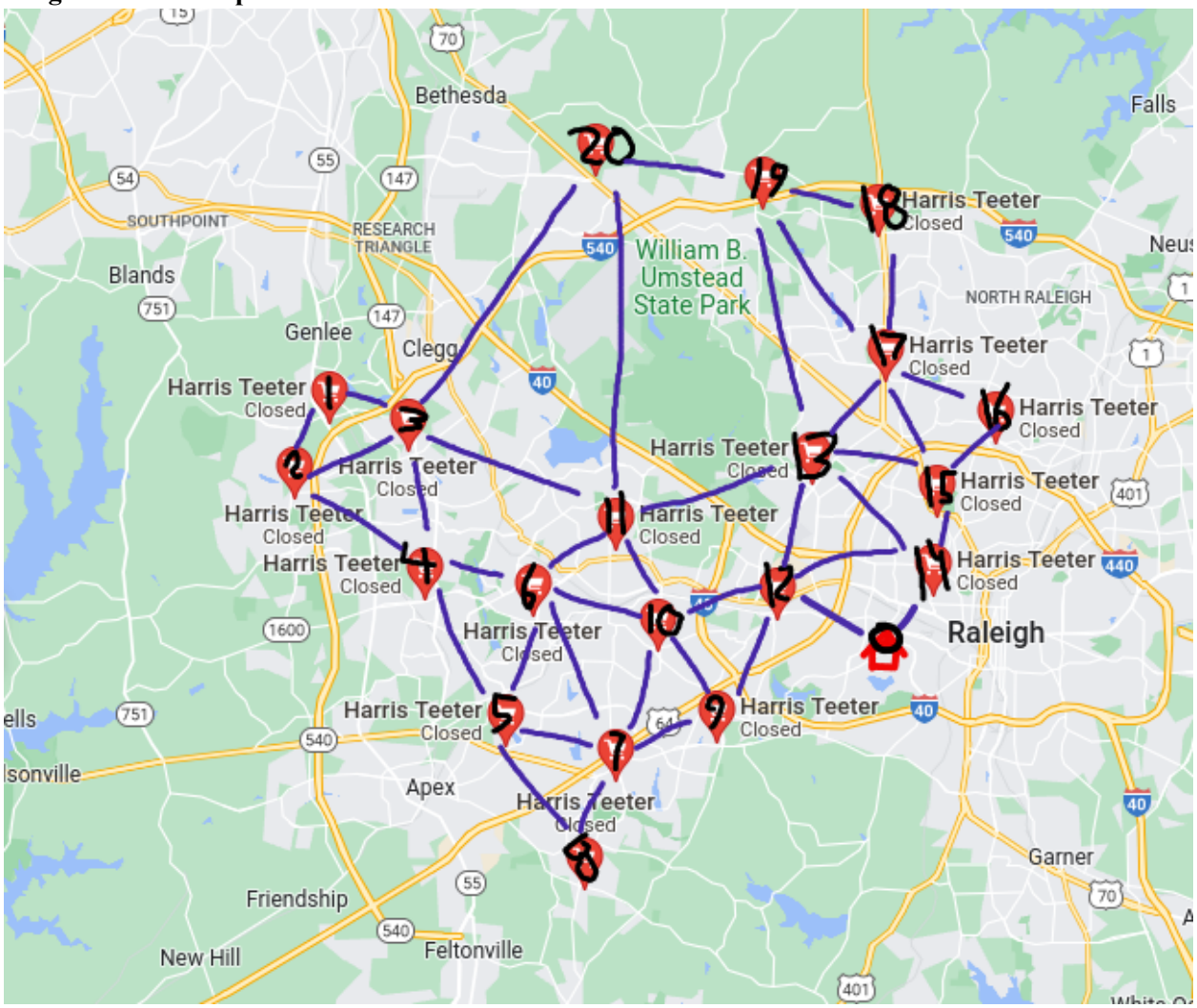


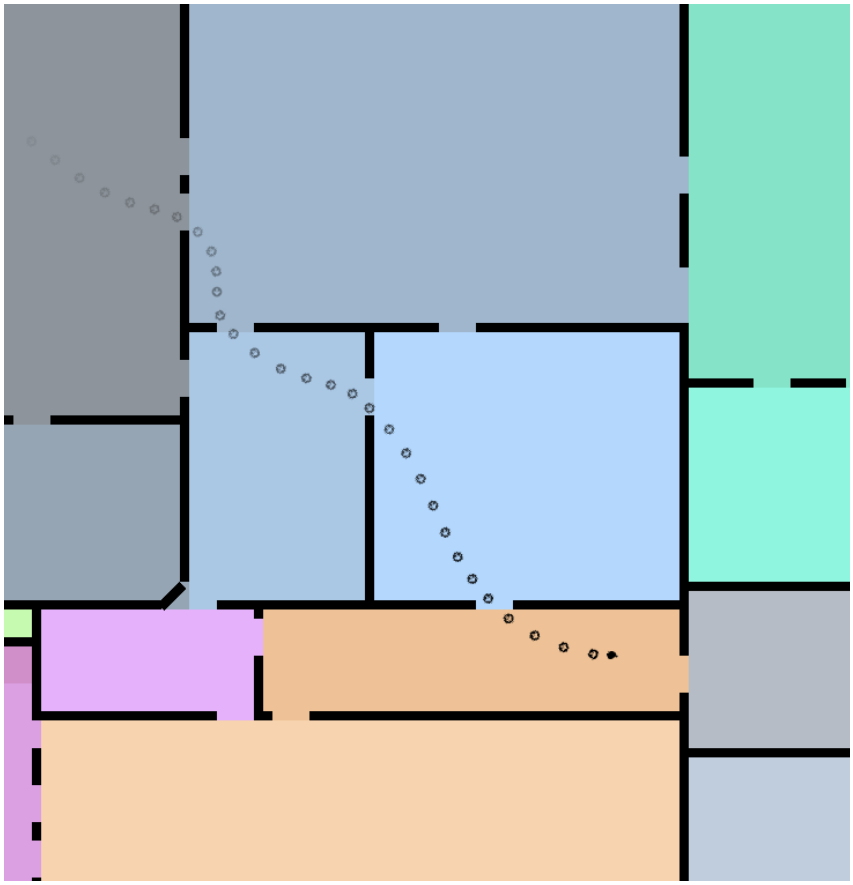
Table 1: Performance on small graph (21 nodes, 80 edges)

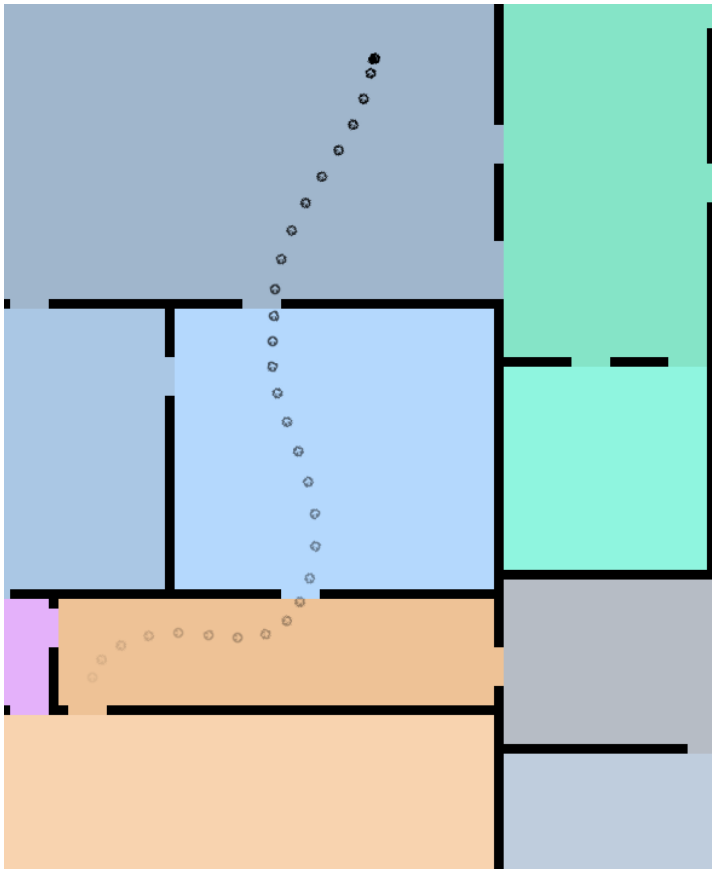
	Run Time (s)	Bytes Allocated	Nodes Visited	Path Length (cost)
Dijkstra's	0.000115147	105,830	21	11
A* (euclidean)	0.000112734	105,830	21	11

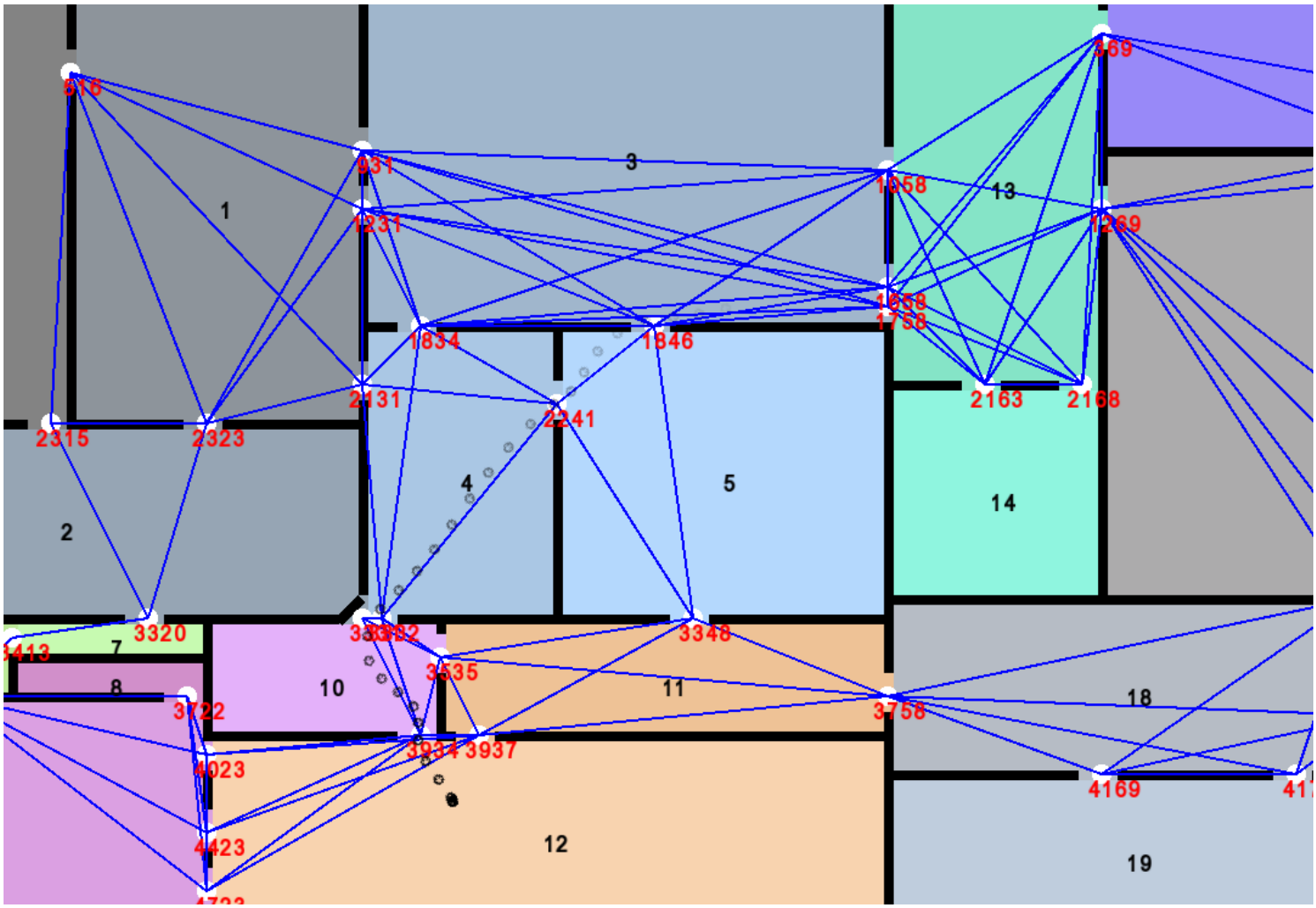
Table 2: Performance on the large graph (10000 nodes , 203241 edges)

	Run Time (s)	Bytes Allocated	Nodes Visited	Path Length (cost)
Dijkstra's	0.043285	11,916,803	10000	1460
A* (euclidean)	0.0226141	11,579,219	2775	1460
Manhattan	0.00324912	11,190,115	70	1508
ManyPaths	0.0606641	11,995,995	10000	1460

Screenshots:






1
2
3
4
5
7
8
10
11
12
13
14
18
19
2315
2323
2131
1834
1846
2241
1658
1758
2163
2168
1269
3320
3348
3535
3722
3758
4023
4423
4169