16-350: Planning for Robotics
Homework 1 Writeup - Undergraduate

Bill Qin
bzq@andrew.cmu.edu
March 2, 2021

The code can be compiled through the default MEX command of "mex planner.cpp".

Performance on Test Cases

- **Test Case 1** --- runtest("map3.txt",[10 10],[200 100]);
  - Moves made: 271, with the final target position at [258 173], **Target Caught**
- **Test Case 2** --- runtest("map3.txt",[250 250],[400 400]);
  - Moves made: 183, with the final target position at [364 362], **Target Caught**
- **Test Case 3** --- runtest("map1.txt",[700 800],[850 1700]);
  - Moves made: 1013, with the final target position at [762 1783], **Target Caught**

In each test case, no one move required more than 200 milliseconds, as the target only moved one space each time.

Assumptions Made

The only assumptions made for this assignment was that the cost of a diagonal movement on the 8-connected grid was also of cost 1.

The Heuristic/Method

As the prompt gave us a soft restriction on 200 milliseconds per move, the most fitting algorithm that could find as good of an answer as possible within this certain timeframe was ARA*. The heuristic I selected ended up being the "half-Euclidean", which as its name suggests is simply half of the Euclidean distance between the two points. Based on the assumption above, it may seem more fitting to do a diagonal heuristic. However, this can lead to some strange movements, such as the bot moving up-right if the goal was southeast-east of the robot. Instead, the half-Euclidean heuristic prefers to move in closer to the target in a way so that its random movements do not have high impact on the overall cost.

The half-Euclidean heuristic is admissible but not consistent, but that is a tradeoff I ultimately decided was fine, particularly as when $\epsilon \geq 2$, it ends up being "consistent".

I chose to do six levels of epsilon, from 4.0 to 3.0 to 2.5 to 2.0 to 1.5 to 1.0, stopping whenever the time from when the function is called to the iteration's completion exceeds 150 milliseconds, but guarantees that $\epsilon = 4.0$ is called. This means that worst-case ARA* ends up being weighted A* with this epsilon value, which in turn guarantees a sub-optimality of a factor of 4.

The routes are stored by having each explored node remember which node initially pushed it into the OPEN queue and updating its parent node when a better g-value is presented. This means that by starting from the goal state it is possible to backtrack using these relations to recover the suboptimal or optimal route that was found and then returns the first step in that direction.

Data Structures

The code was done with the built-in priority queue data structure found in queue.h, with a manual comparison function. Each square was represented by a struct node pointer consisting of x-y coordinates, g-value, priority ($f = g + \epsilon h$), and parents to store optimal paths, and ultimately only the pair of coordinates were passed into the priority queue. Two priority queues were used for the OPEN queue and the INCONS queue. The closed set was represented by a Boolean flag within the node_t type, which allowed for O(1) access to whether or not it was a member of the closed set.

As the board was said to be within 5000 by 5000, a 2D array of node_t pointers were used. While this may seem memory intensive (25 million by 8 bytes on a 64-bit processor is 200 million bytes), all of these pointers were initially set to NULL.

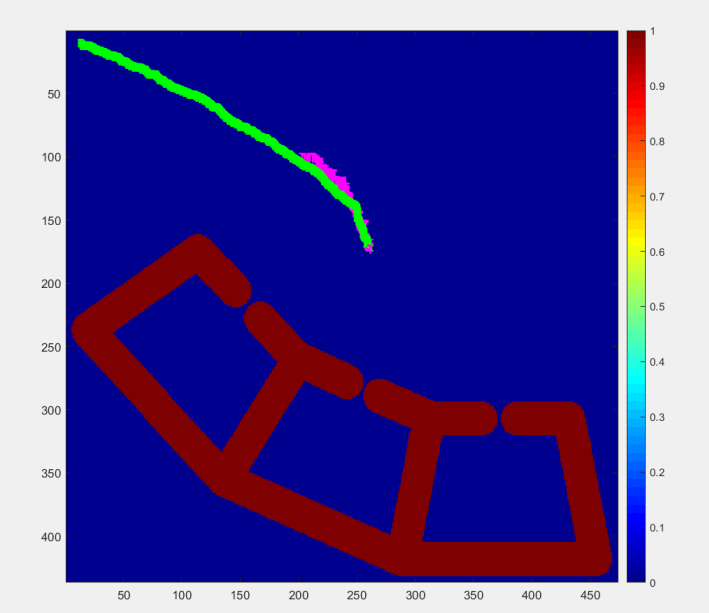
Memory Efficiency

To alleviate the issue of a big 2D array of pointers, they are initially set to NULL, which can provide an upfront time cost. The idea of this is that they are reset to NULL everytime, and when they are considered during the ARA* process, they are then initialized and instantiated (with g-values as infinity, for example) and have its coordinates pushed onto a standard queue, to which after the iteration of planner is complete they can be freed from memory.

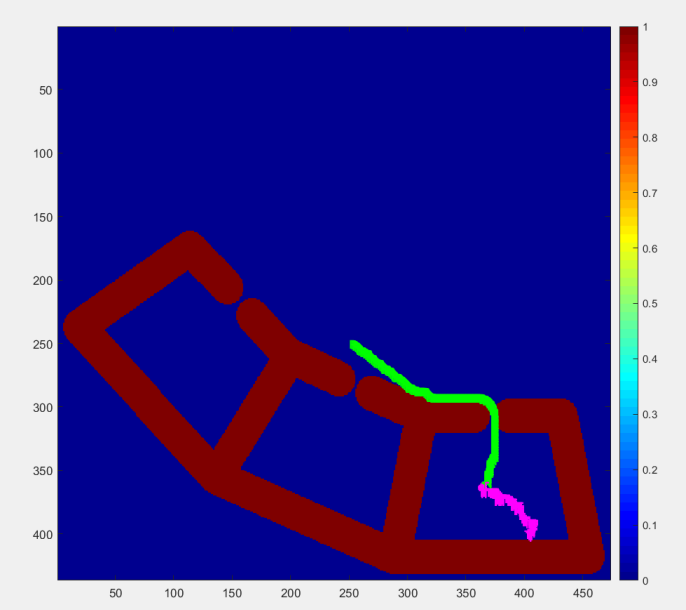
This likely would have worked with maps or other similar data structures, but as I do a lot of direct access to these nodes' information, an O(1) indexing seemed fitting.

Test Case Path Outputs

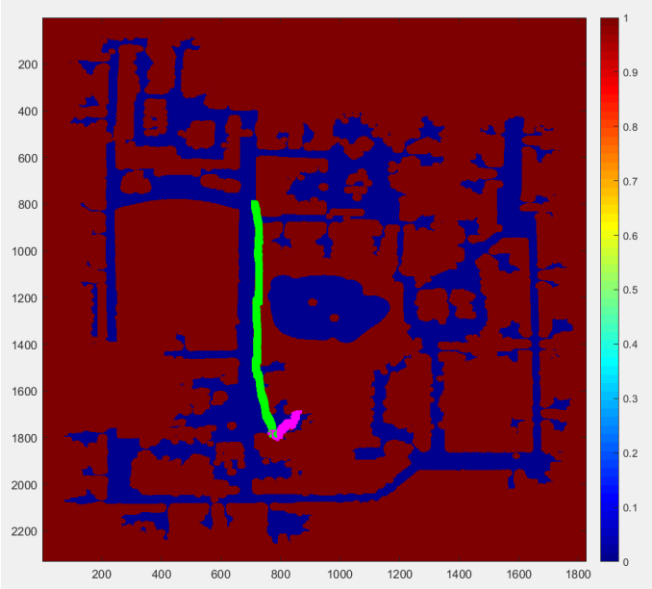
Test Case 1



Test Case 2



Test Case 3


200
400
600
800
1000
1200
1400
1600
1800
2000
2200
200
400
600
800
1000
1200
1400
1600
1800
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0