Approach

1. Compute a 2D Heuristic on the map, where we start from all locations on the target's trajectory with cost 0, and compute costs to all other non-obstacle locations on the map. Store this in a HashMap that maps (x, y) locations to a cost H.
2. Perform an A* in 3D, where states are represented by (x, y, t). The cost of a state is defined as the cost of the shortest path to the state (g) plus epsilon times the 2D Heuristic computed in the previous step to the state's location. Epsilon is experimentally set to 20, which ensures a close to optimal path while speeding up compute time by finding the goal faster.
3. Starting by initializing a priority queue PQ and add the start state as the robot's current position and current time. We keep a hash set *closed* on (x, y) to keep track of closed states.
4. While PQ is not empty, and the goal position is not found:
    a. Pop the lowest cost state from PQ. If the state is on the target trajectory (as in, it's (x,y,t) is equal to the (x,y,t) of a state on the target's path), break.
    b. There are 2 reasons why we would skip this state:
        i. If the location has been visited, i.e., (x, y) is in the hash set *closed*. We don't want to re-expand the state if we've already visited the located. However, we give an exception if the previous move is in place—we should allow the robot to stay in place if it's on the target's path and the time of the state is before the time the target reaches that location, i.e., allow the robot to wait for the target to come to it.
        ii. The time is greater than target_steps time. Since at this point, we've failed to catch the target.
    c. Next we will add neighbors for this state (x, y, t) to PQ
        i. For normal states, we add 8 neighbors in the directions of 8-grid connectivity, as well as 1 additional neighbor for staying in place. We also increment the time to t+1, and compute the respective g (add the cost of the cell) and h (use 2D heuristic). We set a backpointer of new states to the current state.
        ii. If the state is inPlace, we only add staying inPlace for the next state (not all 8 neighbors). This was a huge fix I found, since before implementing this, my PQ would be millions of states long and A* would time out. The logic here is that we only want to stay put if we don't intend on moving anymore.
    d. Lastly, add this state to the closed set.
5. Given that we found the goal, we construct the path using the back pointers starting from the goal state until the first state.
6. Because the compute time for the heuristic and A* may take longer than 1 second, we actually do the search on current_time + plan_time, where plan_time is a buffering time based on the size of the map. This means the robot stays still for a few rounds while it's planning, but as soon as it computes a correct plan, it saves it and does not need to plan for the rest of the run. In case the plan_time is not enough, it will try again planning A* again with an increased plan_time (note that the Heuristic is only computed once since it is independent of time).

Results

Map 1

Compute time:

2D Heuristic: 4.395s

A*: 5.853s

Path length: 2862

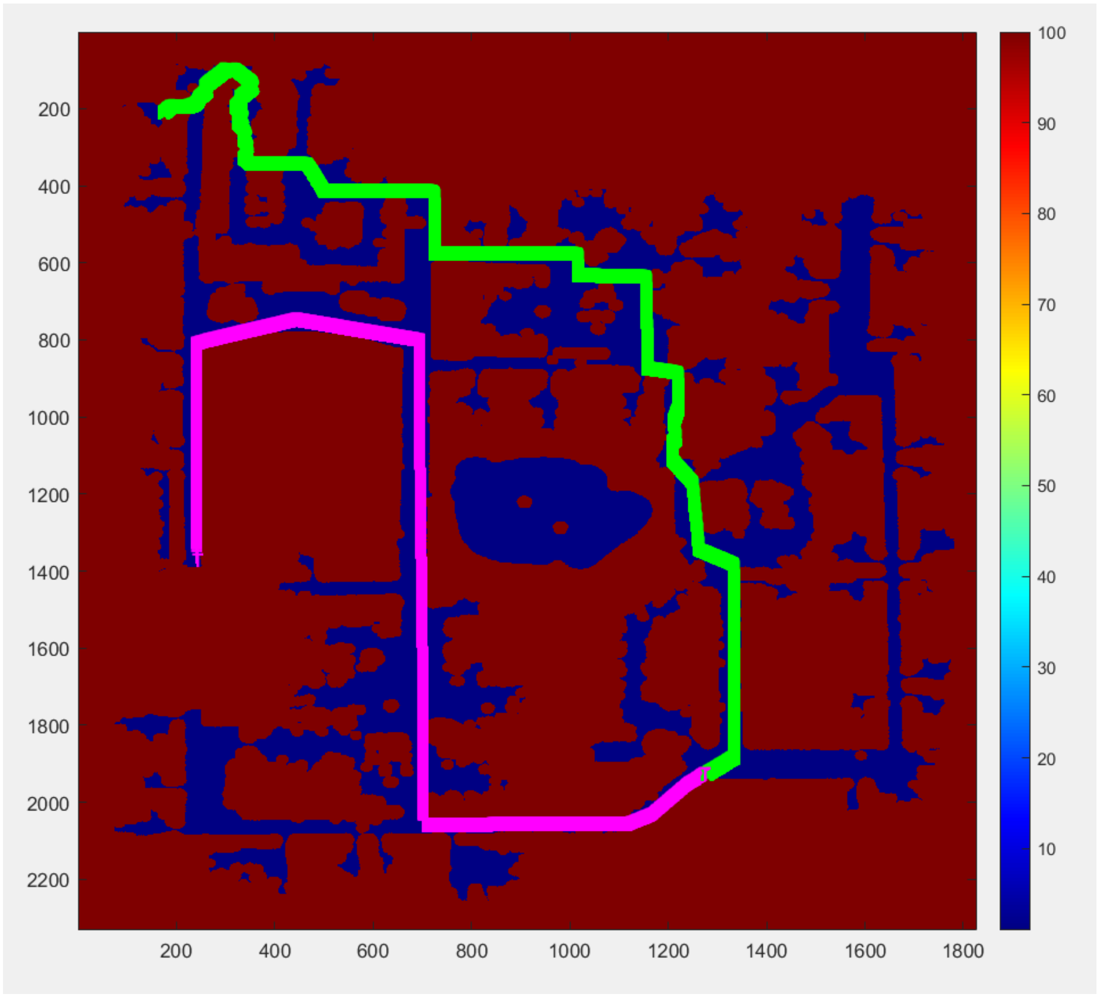
target caught = 1

time taken (s) = 2879

moves made = 2861

path cost = 2879

At the junction around (700, 600), the robot decides to turn left instead of continuing straight. While at first this may seem surprising since it isn't following the target's path, this is actually a strategy so that the robot can catch the target after it makes a turn near the bottom of the map. If the robot had stayed straight to follow the target, it wouldn't have been able to catch up since the robot speed is equal to the target speed.



Map 2

Compute Time

Heuristic: 14.075s

A*: 2.636s

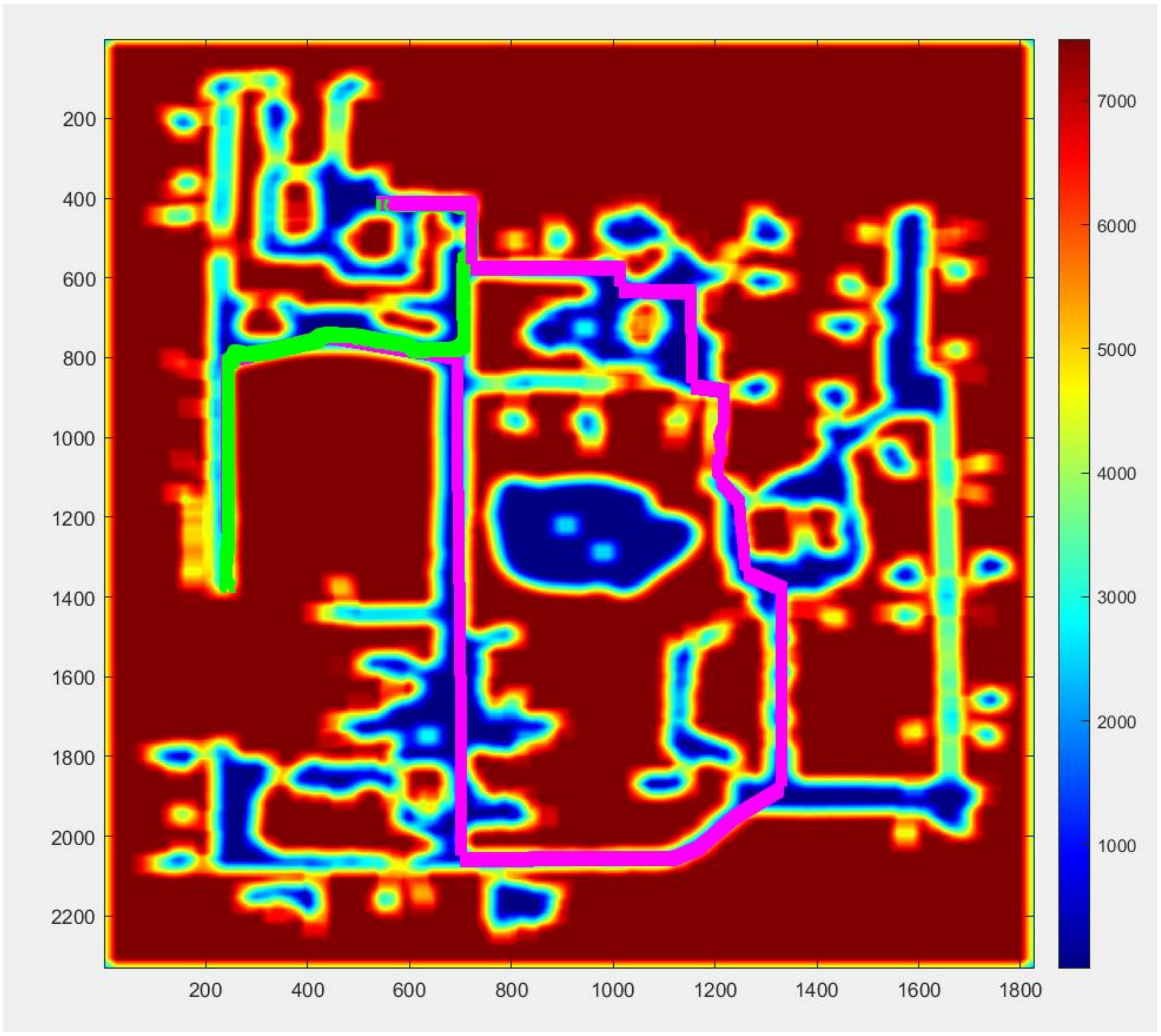
target caught = 1

time taken (s) = 5013

moves made = 1554

path cost = 2393054

This example was interesting because the robot decided to go to a very low-cost area, and then park until the target reached it. It appeared that it would be less expensive to wait in one place, where the cost of the cell is low, than to try to traverse across the high-cost cells.



Map 3

Compute Time:

Heuristic: 0.891s

A*: 0.04

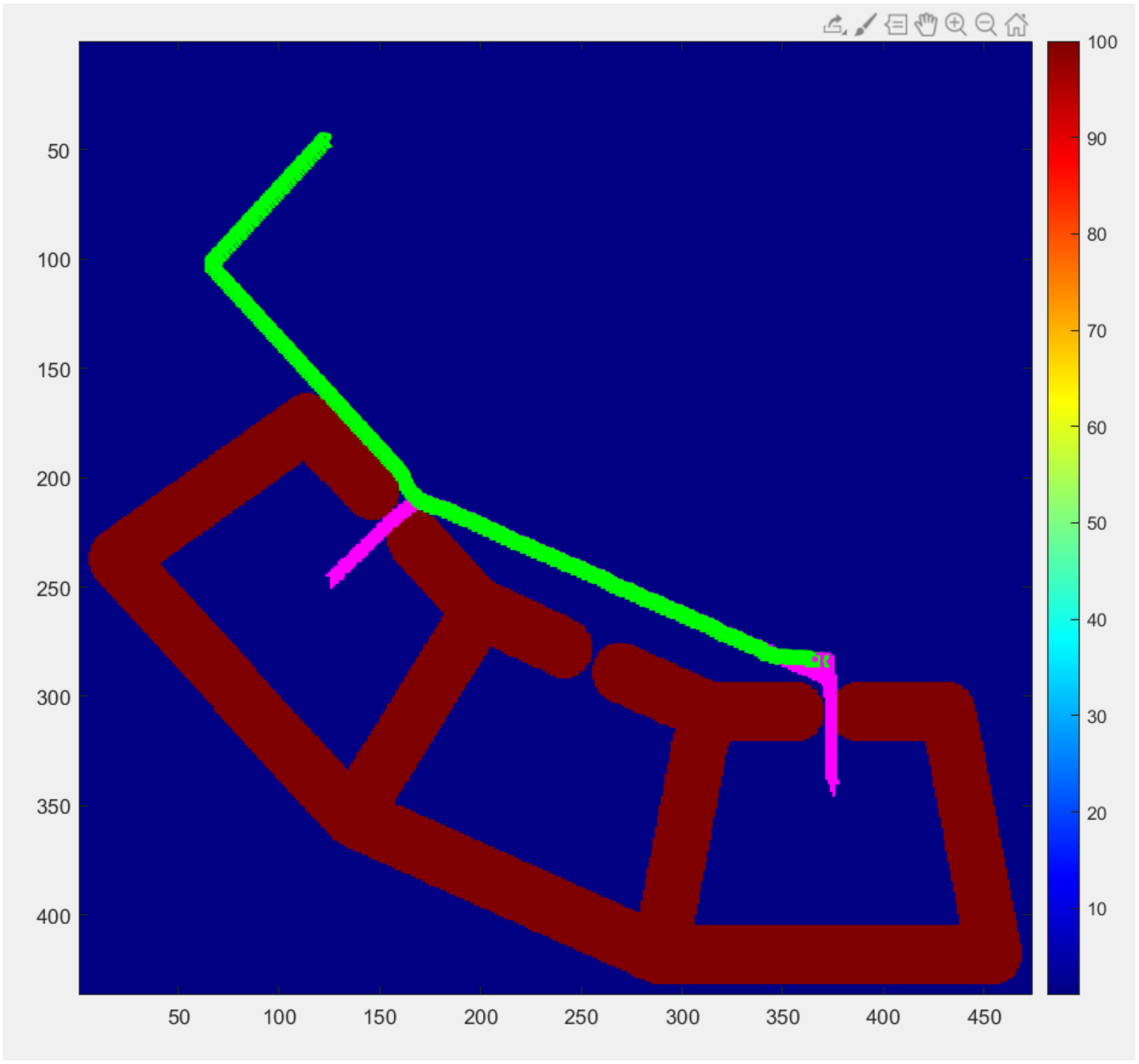
target caught = 1

time taken (s) = 368

moves made = 365

path cost = 368

Because the heuristic is lowest (= 0) when a cell is on the target's path, the robot travels to the target's path quickly, and then tracks the targets path until it gets close to reaching it. Also, because there are no costs associated to being close to the wall, the robot path hugs the wall since it's the most direct way to the target's trajectory.



Map 4

Compute time:

2D Heuristic: 0.984s

A*: 0.038s

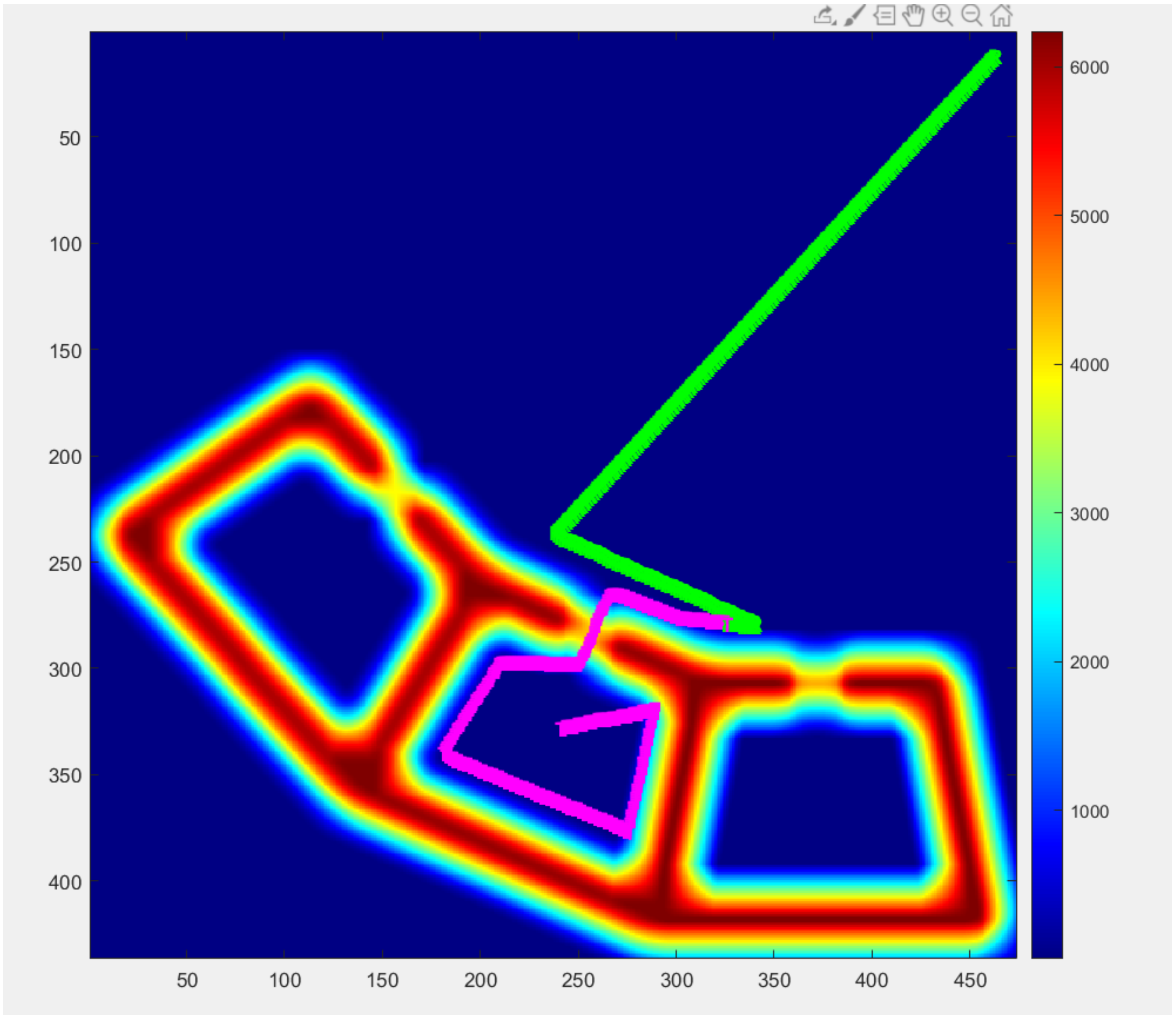
target caught = 1

time taken (s) = 382

moves made = 340

path cost = 382

The robot goes straight down to near the target trajectory until it gets near the obstacle, where it turns away to prevent it from going into the high cost areas near the wall. Instead of going into the walled off "cell", it moves sideways until it catches the target outside the walled areas so that it doesn't need to traverse across the high cost gap into the cell.



Run Instructions

```
clear all

mex planner.cpp

runtest(‘map3.txt’)
```

Please `clear all` or run `mex planner.cpp` between all runs! I use global variables and these don’t get reset otherwise.

(You shouldn’t need to do anything else)