<u>16-782: Planning and Decision-making in Robotics, CMU Fall 2020</u>

Homework 1 – Catching a Moving Object

Deadline: September 30, 2020, 11:59PM

(v1)

<u>Problem 1</u> - 100 points

For this homework, you will program a planner that allows a point robot to catch a moving object. During execution, the planner will be given an 8-connected 2D gridworld (that is, the robot can only move by at most one unit along the X and/or Y axis). Each cell in the gridworld will be associated with the cost of moving through it. This cost will be a positive integer. For each map, there is an associated collision threshold specified for the planner. Any cell with cost greater than this threshold is to be considered an obstacle that the robot cannot traverse. The gridworld will be of size M by N, with the bigger of two gridworlds in this homework around 2,000 x 2,000 units.

The planner will also be given the start position of the robot, and the trajectory of the moving object as a sequence of positions (for example: (2,3), (2,4), (3,4)). The object will also be moving on the 8-connected grid. The object will move at the speed of one step per second.

All this information is specified in text files named ***map*.txt***. Specifically, the format of the text file is:

1. The letter ***N*** followed by two comma separated integers on the next line (say N1 and N2 written as *N1,N2*). This is the size of the map.
2. The letter ***C*** followed by one integer on the next line. This is the collision threshold for the map.
3. The letter ***R*** followed by two comma separated integers on the next line. This is the starting position of the robot in the map.
4. The letter ***T*** followed by a sequence of two comma separated integers on each line. This is the trajectory of the moving object.
5. The letter ***M*** followed by N1 lines of N2 comma separated floating point values per line. This is the map.

Images of these maps are at the end of this document.

The file ***readproblem.m*** included in the homework packet parses the text files and returns all of the data. It is called inside the ***runtest.m*** script once at the beginning.

The task for the planner is to generate a path for the robot that will allow it to catch the object with the *least cost incurred.* The cost of a path is calculated as the sum of the costs of cells visited by the robot, each multiplied by the number of seconds the robot spends in that cell. For example, if the robot spends 3 seconds in cell (2,3), then that contributes 3 times the cost of cell (2,3) to the cost of the path.

You will run the ***runtest.m*** script for all four problem maps. It only takes the name of the text file as input. It returns four values – a boolean specifying whether the object was caught, and three integers specifying the time taken to run the test, number of moves made by the robot, and the cost of the path traversed by the robot. It also prints this information out before returning.

While running ***runtest.m***, your planner will be called once per simulation step. Your planner should only return the next action to take, i.e. it should return a 2D vector. We will calculate the time it takes for your planner to return a solution (to the closest integer second, rounding up, say K). We will then move the object by K steps along its trajectory. The robot will only take the action returned by the planner, i.e. the robot will only move by one legal step (legal in terms of the 8-connectedness of the grid).

Note: After the last cell on its trajectory, the object disappears. So, if the given object's trajectory is of length 40, then at timestep = 41 (assuming timesteps begin with 1) the object disappears and the robot can no longer catch it.

This means for a moving object trajectory that is T steps long, your planner has at most T seconds to find a full solution.

There are two other files included. ***robotplanner.m*** returns the next action by executing the planner (either from the compiled MEX binary or from within MATLAB). The choice between using MATLAB and C/C++ is left to you, but we strongly suggest using C/C++ since it is highly unlikely something written in MATLAB will be fast enough to solve the problem.

planner.cpp is where you will implement your planner in C/C++. The planner function in this file is the only one you should change. Feel free to add any additional header and source files as you desire. Currently, the planner greedily moves towards the last position on the moving object's trajectory. If you run it as is, the planner only succeeds on map4.

Programming Tips

1. You can run matlab without the GUI using *"matlab -nodesktop".*

2. Use *"mex planner.cpp"* to compile your cpp code.

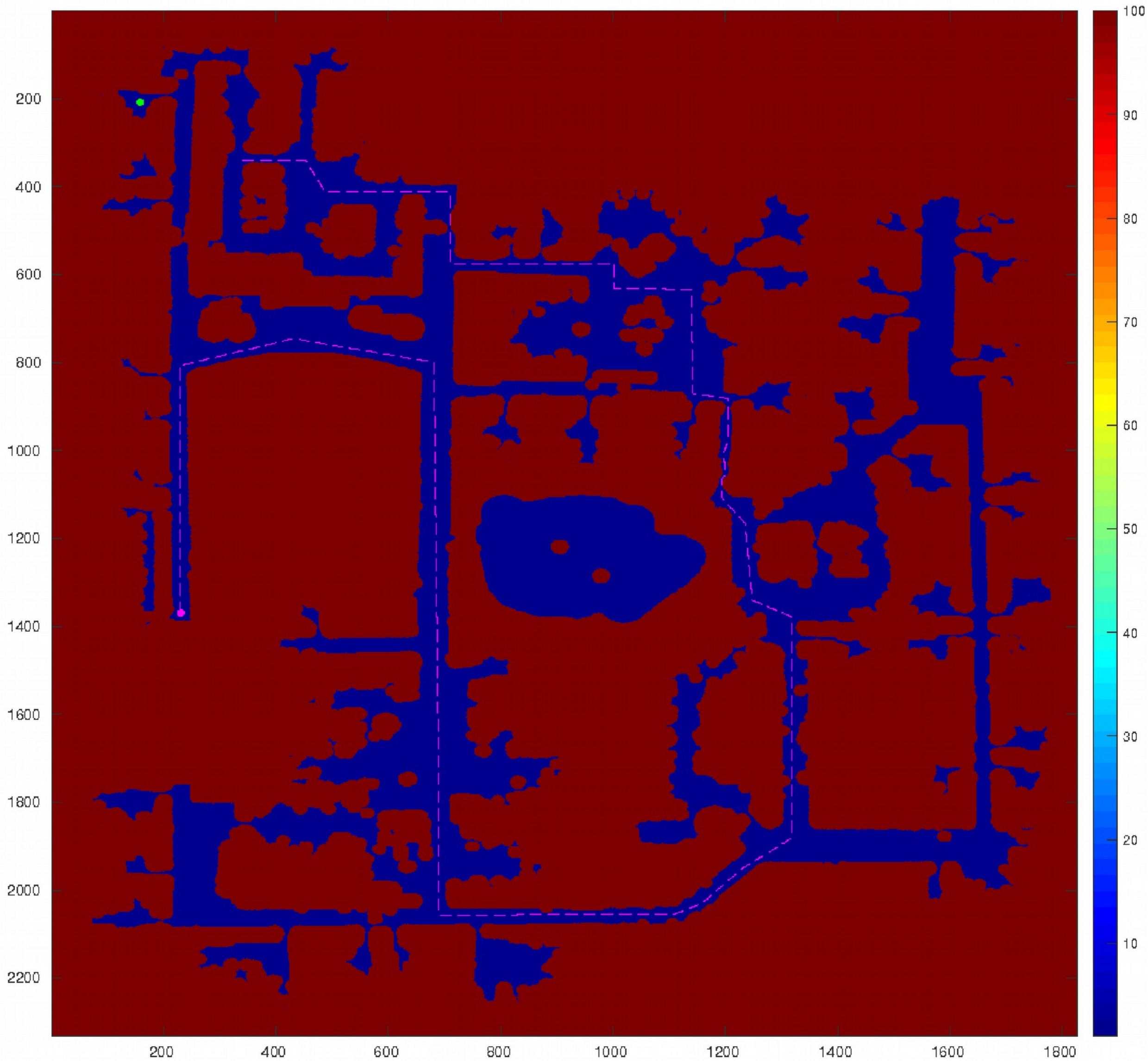

Image of information in map1.txt. The green dot is the starting position of the robot, magenta dot is the starting position of the moving object, dashed magenta line is the object's trajectory. Blue cells have cost 1, red cells have cost 100, collision threshold is 100.

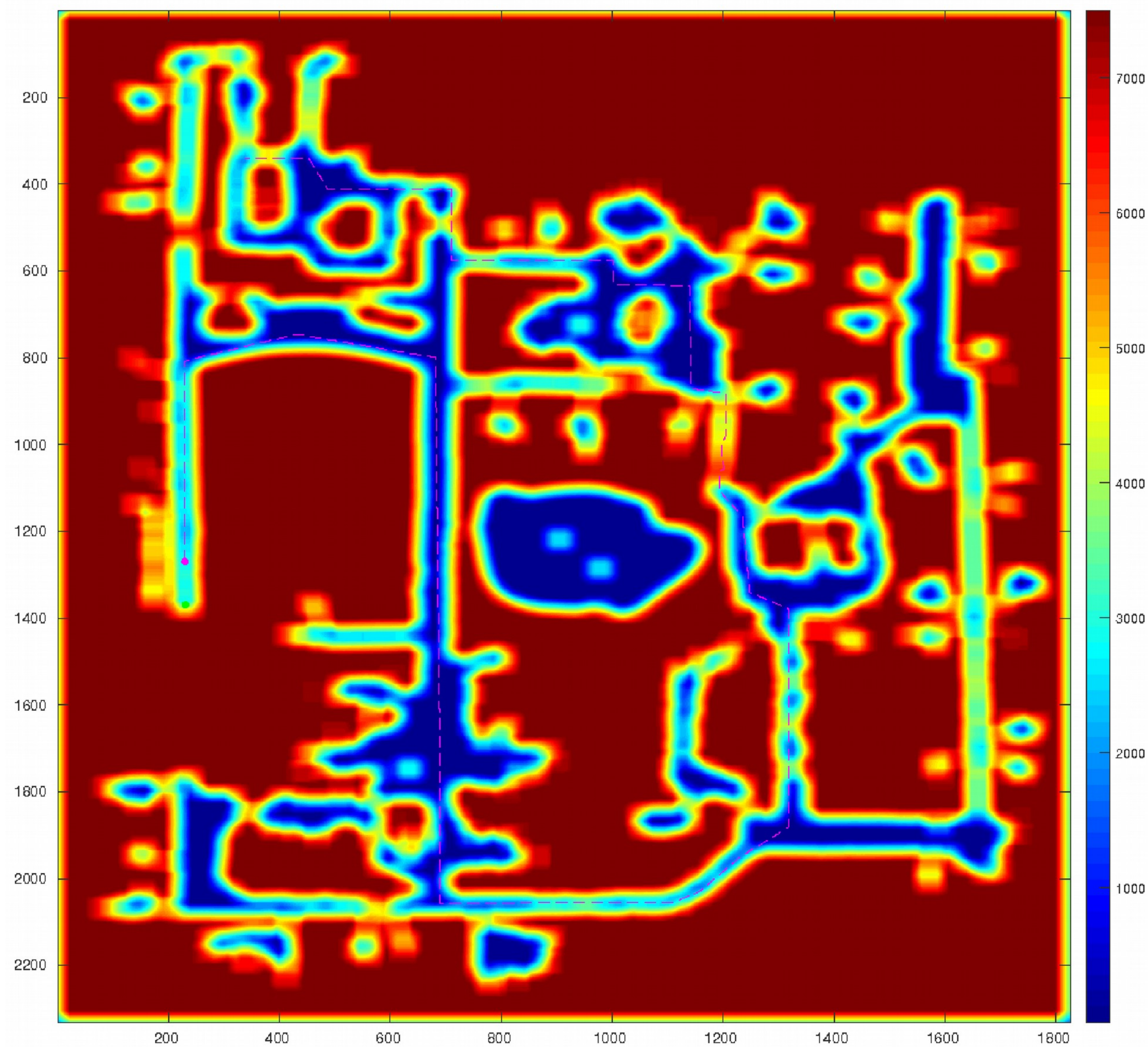

Image of information in map2.txt. The green dot is the starting position of the robot (directly below the object dot), magenta dot is the starting position of the moving object, dashed magenta line is the object's trajectory. Cells have cost between 1 and 7497, inclusive. Collision threshold is 6500.

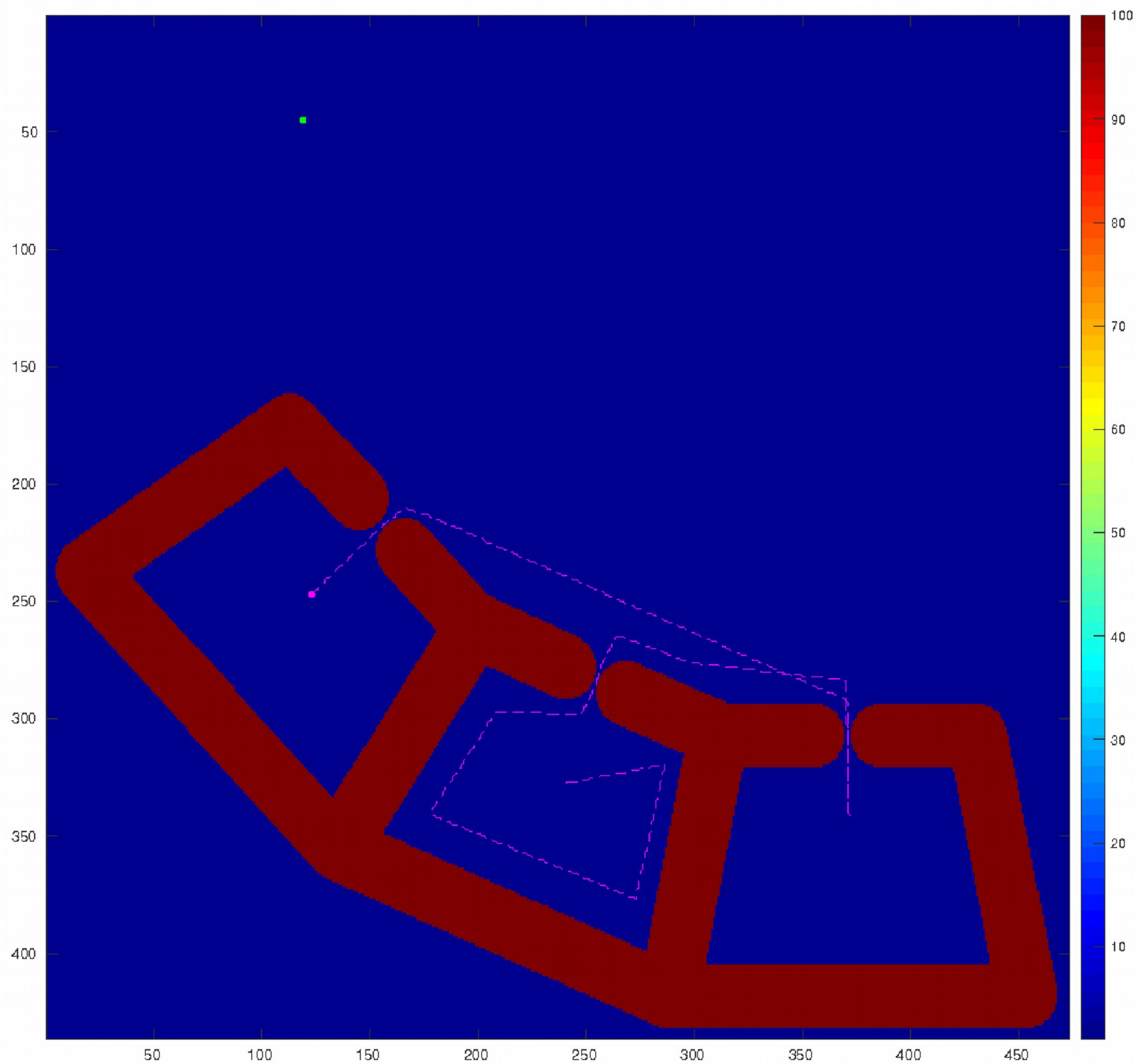

Image of information in map3.txt. The green dot is the starting position of the robot, magenta dot is the starting position of the moving object, dashed magenta line is the object's trajectory. Blue cells have cost 1, red cells have cost 100, collision threshold is 100.

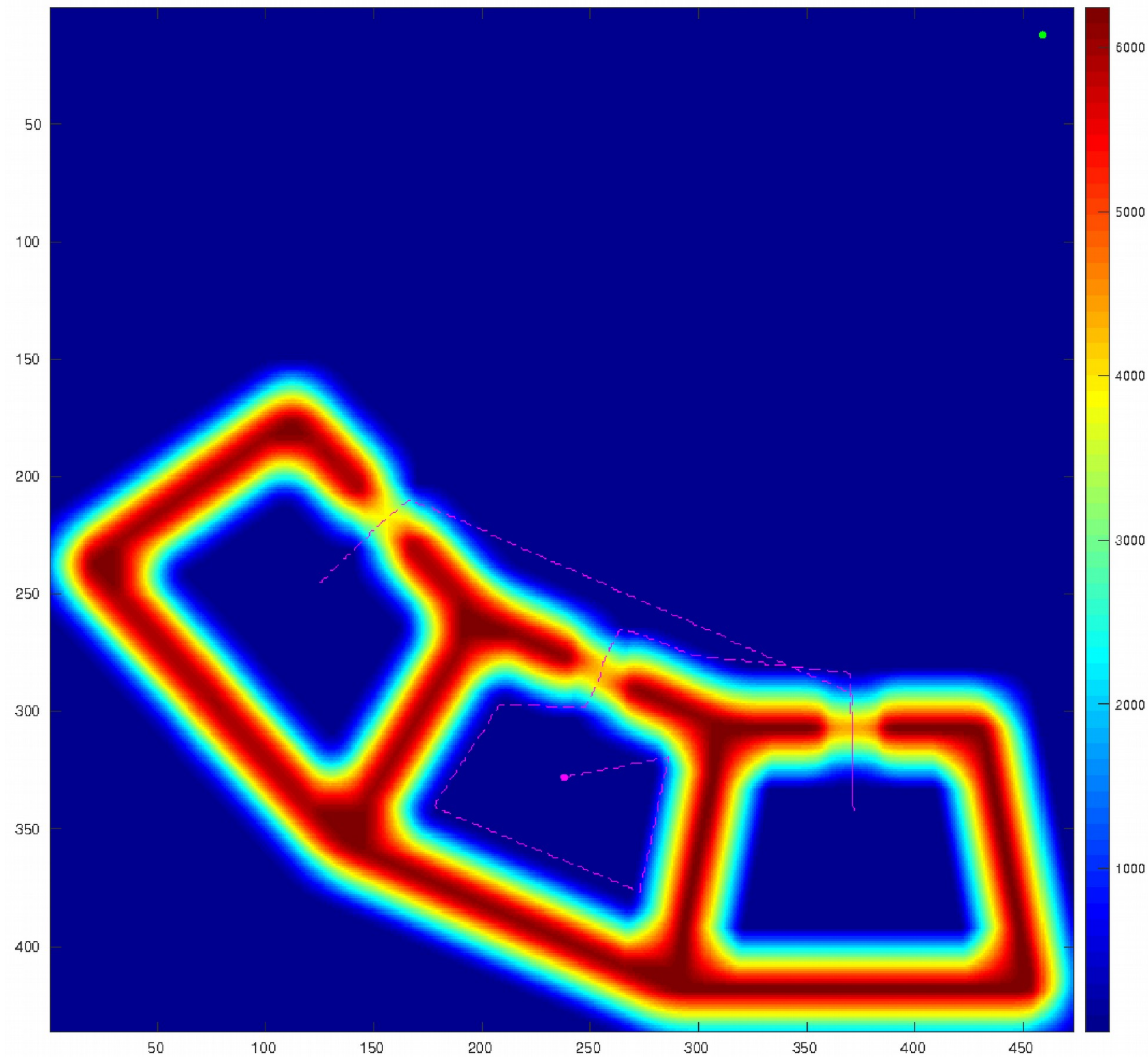

Image of information in map4.txt. The green dot is the starting position of the robot, magenta dot is the starting position of the moving object, dashed magenta line is the object’s trajectory. Cells have cost between 1 and 6240, inclusive. Collision threshold is 5000.

<u>Problem 2 (Bonus)</u> - 5 points

Weighted A* without re-expansions comes with the same sub-optimality guarantees as weighted A* with re-expansions while promising to expand every state at most once. Would you ever want to allow re-expansions in weighted A*? Is the solution computed by the former always equivalent or better than the latter? If so, prove it, else, give a counter-example.

<u>Submission Guidelines</u>

You will submit this assignment through Gradescope. You need to upload one ZIP file named *<Andrew ID>.zip*. This should contain:

1. A folder named *code* that contains all code files, including but not limited to, the ones in the homework packet. If there are subfolders, your code should handle relative paths. We will only compile your MEX code (according to instructions in the writeup) and execute the *runtest.m* script.
2. A PDF file named *<Andrew ID>.pdf*. This should contain a summary of your approach for solving this homework, the results for all four maps (whether the object was caught, the time taken to run the test, number of moves made by the robot, and the cost of the path traversed by the robot), and most importantly instructions for how to compile your code. This should be one line for us to execute in MATLAB of the form "*mex <file 1> <file 2> … <file N>*".
    - For your planner summary, we want details about the algorithm you implemented, data structures used, heuristics used, any efficiency tricks, memory management details etc. Basically any information you think would help us understand what you have done and gauge the quality of your homework submission.
    - Include plots of the maps overlaid with the object and solved robot trajectories.

Please **do not** include the map text files in your submission.

<u>Grading</u>

The grade will depend on two things:

1. How well-founded is the approach? In other words, can it guarantee completeness (finds the solution if one exists), can it provide sub-optimality or optimality guarantees on the paths it produces, can it scale to large environments?
2. How much cost the robot incurs while catching the target?