16-782: Planning and Decision-making in Robotics, CMU Fall 2020
Homework 1 – Catching a Moving Object

Student- Radhika Mohanan (rmohanan)

Problem 1:

Running the Code:

The planner should always be complied before running the **runtest** command on any map by following the steps as below:

1) Go to Code Folder
2) In Matlab, run the command **mex planner.cpp** (For compiling)
3) In Matlab, run the command **runtest('map?.txt')** (? = map number)

Code Details: The planner used is Dijkstra, abstracted as C++ class in header file **Dijkstra.h**. It expands the entire map and can be easily applied to other 2D grid-based planning problems.

A) The following approach was used:

1) Expand the entire traversable map using Dijkstra.
2) For each coordinate of the moving object trajectory back-trace a path to the start position of the robot.
3) Different paths to reach the robot from the object will be obtained by back-tracing. In order to find the paths that are valid, wait time of the robot is calculated for a coordinates on the object trajectory say (p,q) as below:

$$Wait\ time\ at\ point\ (p,q) = \big(Time\ required\ for\ the\ object\ to\ reach\ (p,q)\big) - (path\ length\ from\ robot\ location\ to\ coordinates\ (p,q))$$

Only the paths with wait time greater than zero are considered as valid paths.

4) In each valid path, minimum cost coordinates *(pmin,qmin)* are selected for the robot to wait for the object. The cost is calculated during the back-tracing done in step2. This method will minimize cost.
5) Calculate the wait time (t_w) for the coordinates *(pmin,qmin)* following the equation in step3. As the robot reaches the coordinates *(pmin,qmin)*, wait there till t_w time for the object.
6) Calculate the cost of robot from start position to the desired lowest cost coordinates *(pmin,qmin).* Cost of the path to the *(pmin,qmin)* is computed during the initial expansion.

$$Cost = cost\ of\ the\ path\ traversed\ by\ robot + cost\ of\ (pmin, qmin)\ coordinates * t_w$$

7) The valid $_{paths}$ are stored in std::map with actual cost as its key value
8) We calculate the time required for steps 1 to 7 *($t_{initial}$)*, if the time is less than or equal to wait time *(t_w)* calculated in Step5 (*($t_{initial}$)≤ (t_w)*), then this is the optimal path. If not, then

continue the search until you find the time for the initial steps *($t_{initial}$)* to be lesser than or equal to wait time *(t_w).*

9) Trimming of the optimal path is done by removing $t_{initial}$ from t_w (t_w=t_w-$t_{initial}$) after the robot reaches the coordinates $(pmin, qmin)$
10) Execute the trimmed path until the object is caught at the coordinates $(pmin, qmin)$.

B) The data structures used were:

1. std::unordered_map: Unordered_maps are used to map the coordinates of object trajectory to its index.

2. std::priority_queue:Priority_queue is used to store the cells in the open-list based on the cost of the cell . The nature of a queue structure is such that, if we need to update the information (a lower cost, parent) of a cell that already exists in the open list, we do not search for it in the queue. Instead, we update the corresponding 2D vector, and then push a new element with the updated information into the open list. The older element will be automatically ignored when it is popped out from the open list later due to having higher cost.

3. 2D std::vector:Vectors are used for original map, the updated cost map, the map to store parent coordinates, visited coordinates in the open list, and closed list coordinates with the dimension same as the original map.

4. std::map:Map stores all the valid paths of the robot found by the planner and they are sorted based on the total path cost.

5. struct are used to for simplifying the code, providing better readability.

C) Efficiency Tricks:

The planner is expanded only once at the beginning and it generates the candidate paths. The robot just picks the most optimal path based on the values of t_w and $t_{initial}$ for each path. Therefore, no memory is used for replanning of the path.

D) Results:

1) Map1

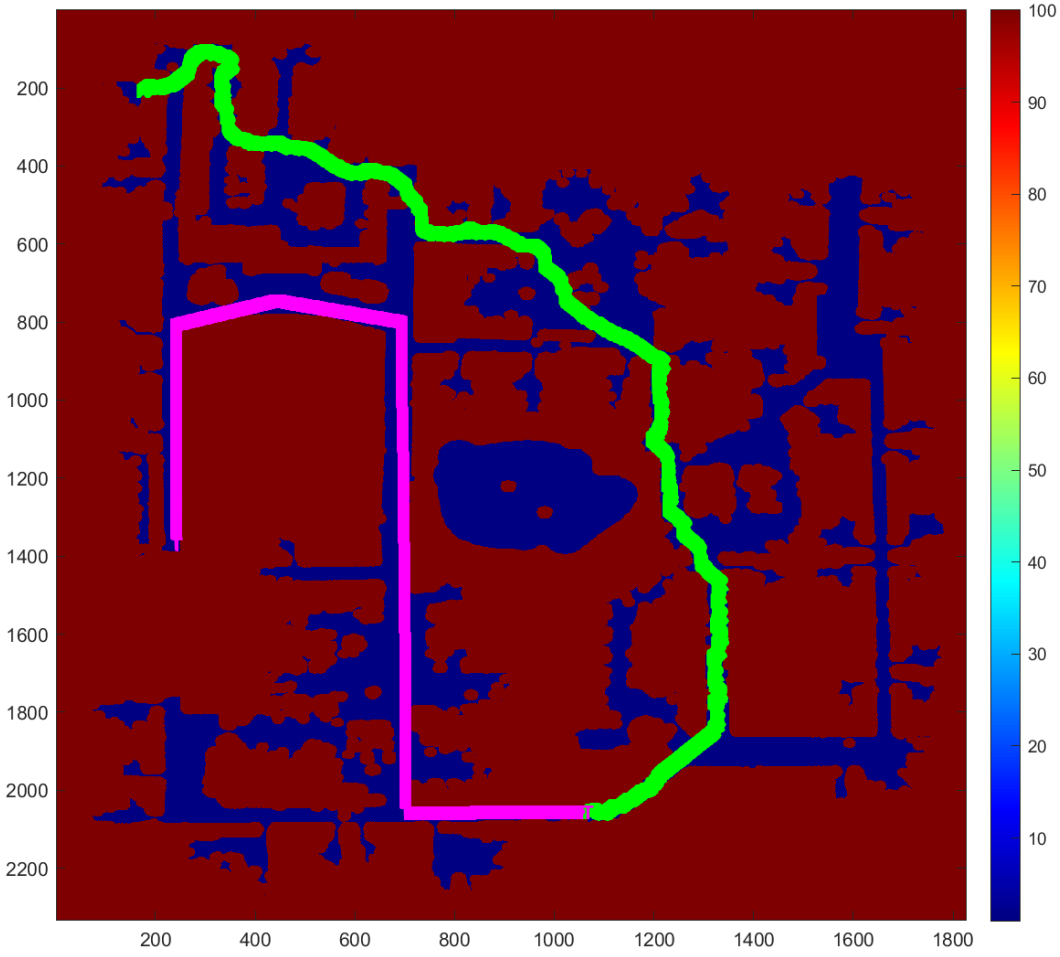

Figure 1

RESULT:

target caught = 1

time taken (s) = 2642

moves made = 2638

path cost = 2642

2) Map2

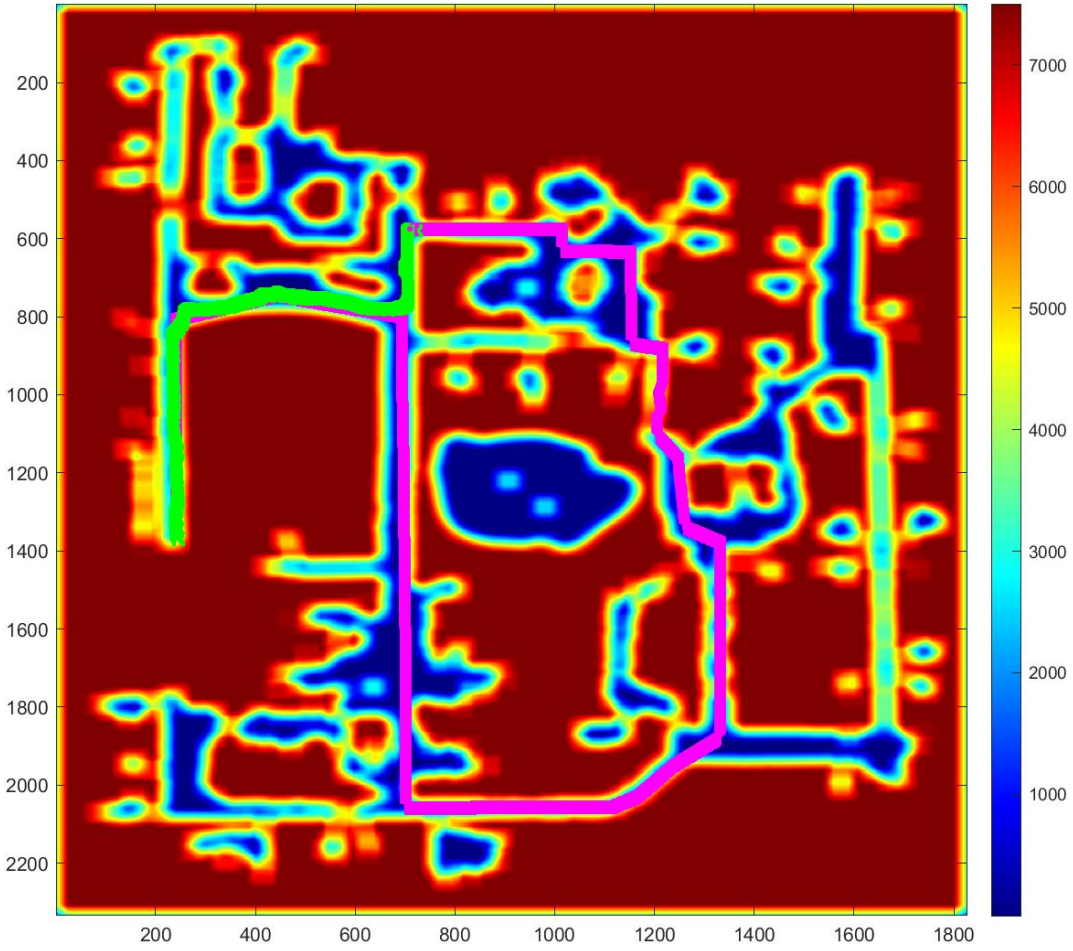

Figure 2

RESULT:

target caught = 1

time taken (s) = 4673

moves made = 1235

path cost = 1603276

3) Map3

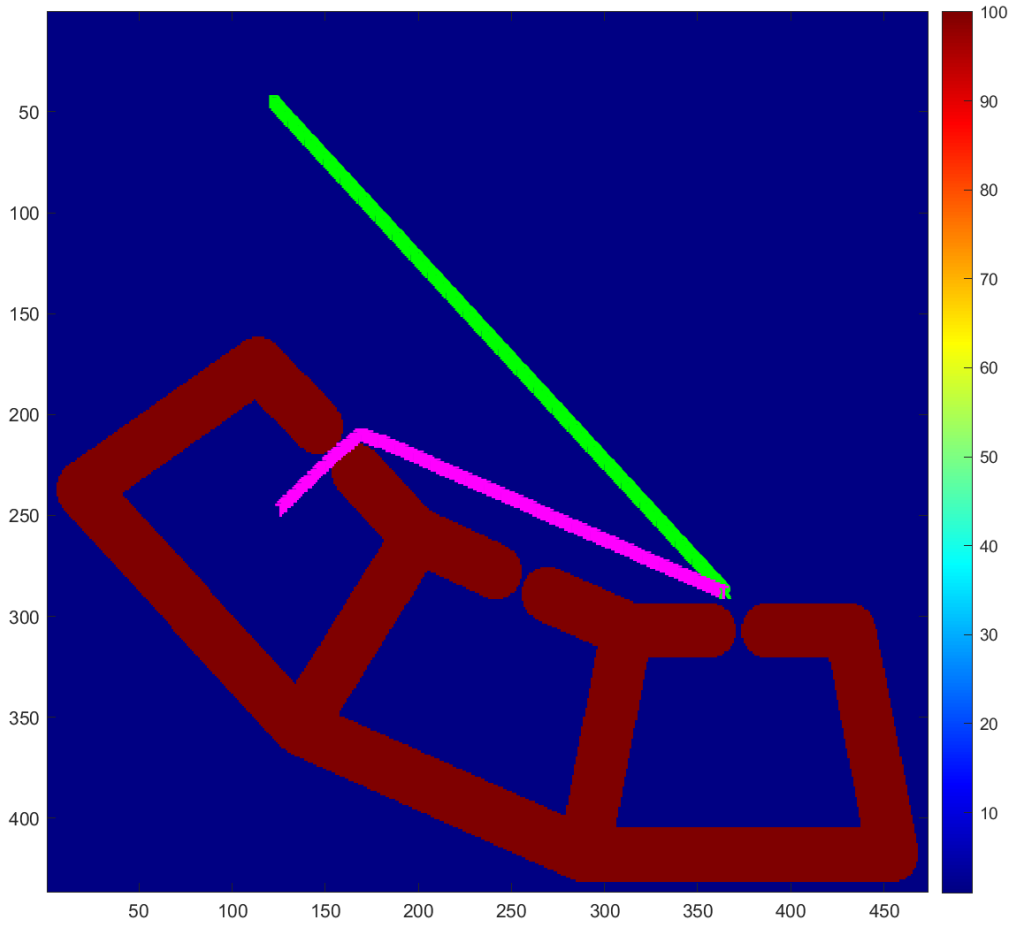

Figure 3

RESULT:

target caught = 1

time taken (s) = 247

moves made = 243

path cost = 247

4) Map4

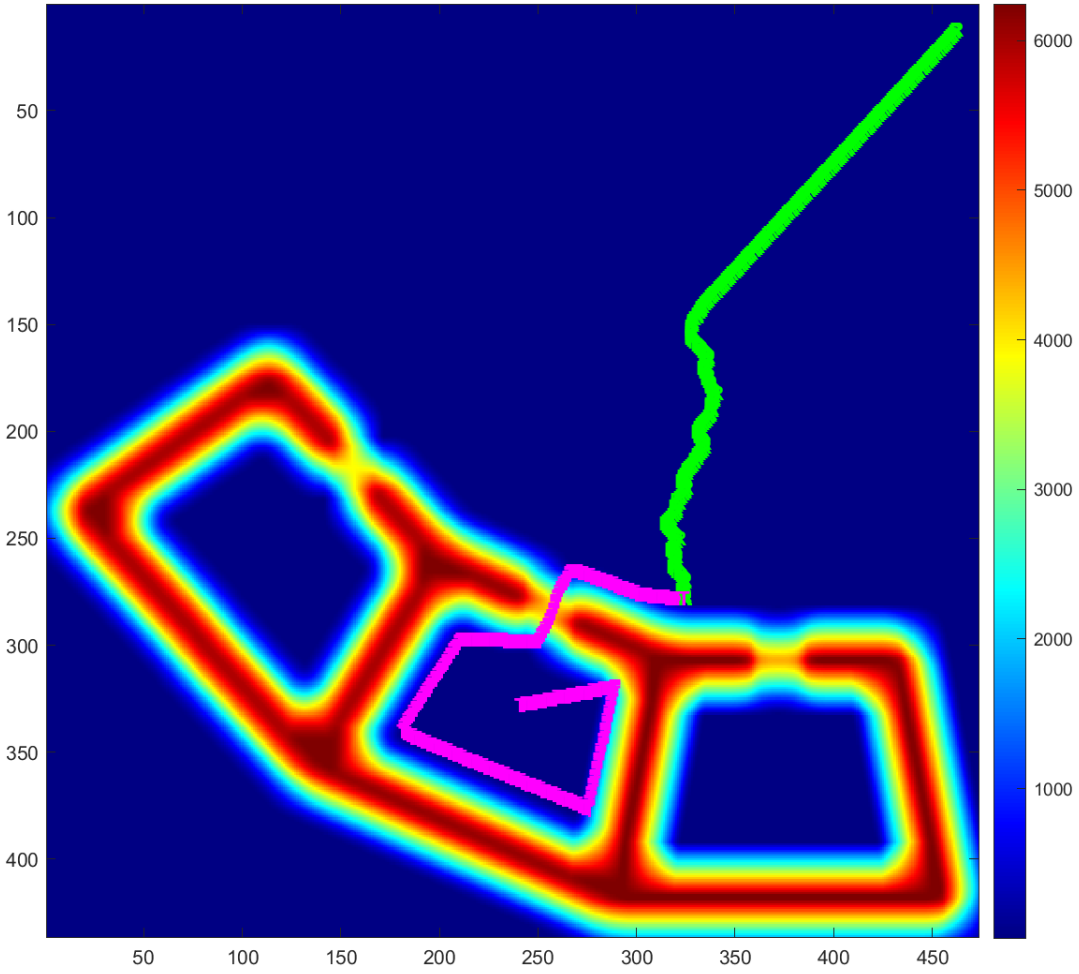

Figure 4

RESULT:

target caught = 1

time taken (s) = 380

moves made = 266

path cost = 380

Problem 2

Weighted A* without and with re-expansion while promising to expand every state at most once, have the same suboptimality guarantee as they are bounded by the same weight. But re-expansion will provide an optimal solution if performed more than once, given the required computation power is available. It can happen that while re-expanding the CPU goes out of memory and is unable to produce any results. Thus, with efficient computation the solution obtained from weighted A* with re-expansion will be better than without re-expansion as the values of the nodes in the closed lists can changed multiple times.