16350: Planning Techniques for Robotics

[redacted]
Carnegie Mellon University
[redacted]

1. File Structure and Execution

The folder contains several files and the instructions for running them are as follows:

1. mex planner.cpp
2. runtest("map(x).txt") where x is the number of the map

It uses a large portion of static memory. So if MATLAB fails due to a low level graphics error, change the default graphics option in MATLAB by using **opengl('save','software')**. Also for the larger maps, allocate larger memory to MATLAB or open new instances of MATLAB to run them.

2. Assumption

The basic assumption is that we are made aware of the target's entire trajectory at the very start of the the experiment. Also the target takes at the most 1 second to make a move in any of the 9 directions (including staying in the same position).

3. Methodology

3.1. Initial approach: 3D A* Search with 2D Dijkstra heuristic at every time instant

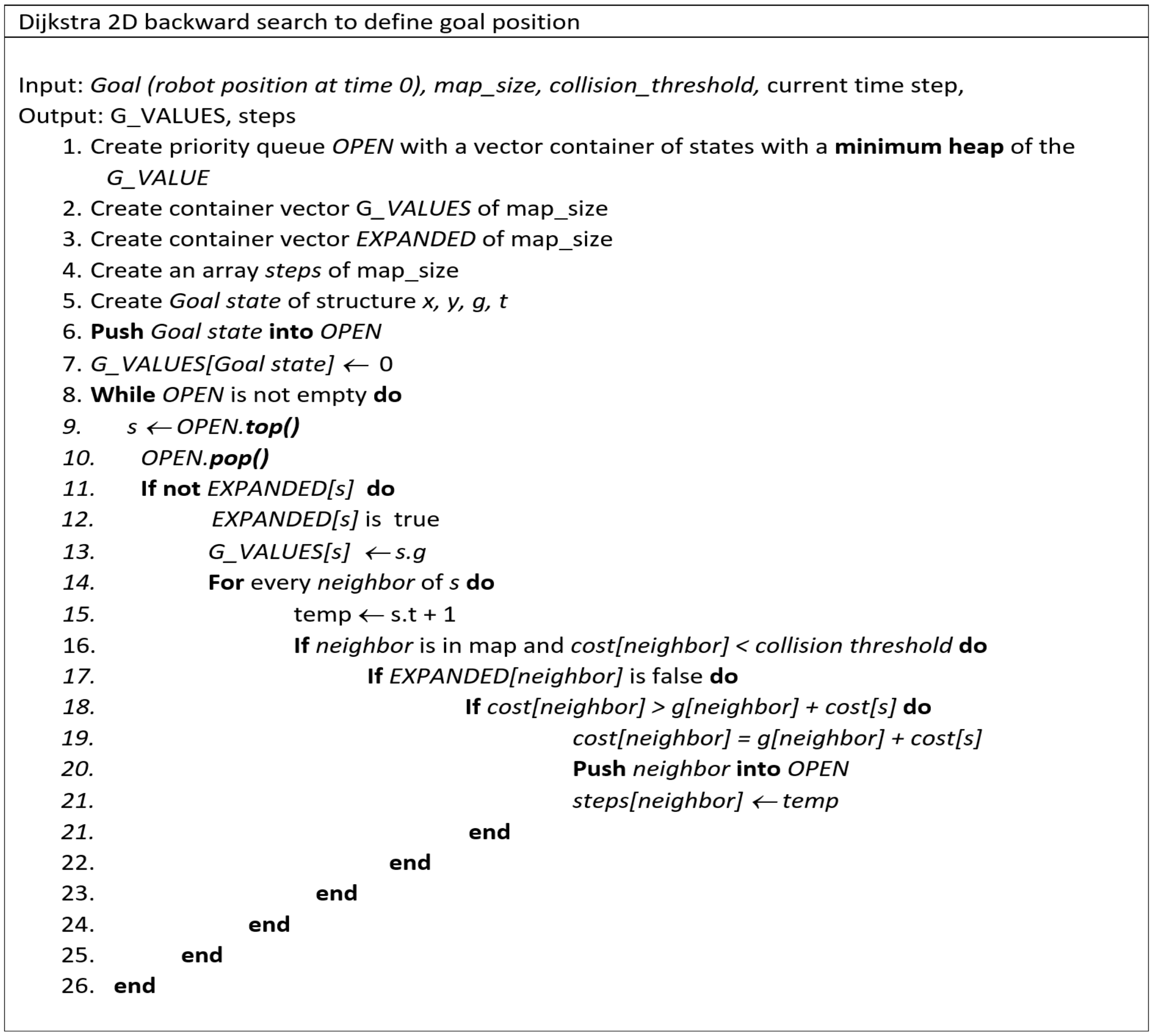
The goal in this case at every time instant is a point on the target's trajectory exactly 20 seconds ahead of the current time instant. If the current time was less than 20 seconds from the final time, the goal was then just the target's current position. The initial approach involved calculating a 2D Dijkstra Heuristic at every time instant to the goal followed by a 3D A* star search. However, the time complexity of this search is immense and it failed to compute an intended action within one second. Also this approach failed to guarantee completeness i.e. there was no guarantee that the robot would catch the target within the given amount of time especially if the target's trajectory was too short.

3.2. Final approach: One shot planning, 3D A* Search with 2D Dijkstra heuristic and a predefined goal

This approach was divided into several parts. The first part involved computing a 2D backward Dijkstra search with the initial position of the robot being the goal position. This helped me get an idea of which point on the target's trajectory could be the goal for computation of an actual heuristic through a forward 2D Dijkstra search. The afore mentioned method is performed exactly once during execution at the first time instant.This is followed by a 3D A* search at every time instant which guarantees optimality and completeness of the solution.

3.2.1 Performing a 2D backward Dijkstra search to decide a static goal

The position of the robot at time instant 0 is taken as the goal and a backward Dijkstra search is computed. This also indirectly tracks the amount of time the robot takes to reach every point on the target's trajectory. The algorithm is shown in the figure along with the containers deployed for the same in the corresponding C++ program. Vectors are chosen over sets because of their insertion and deletion computational superiority. The function is the 'compute Heuristic' function in the code



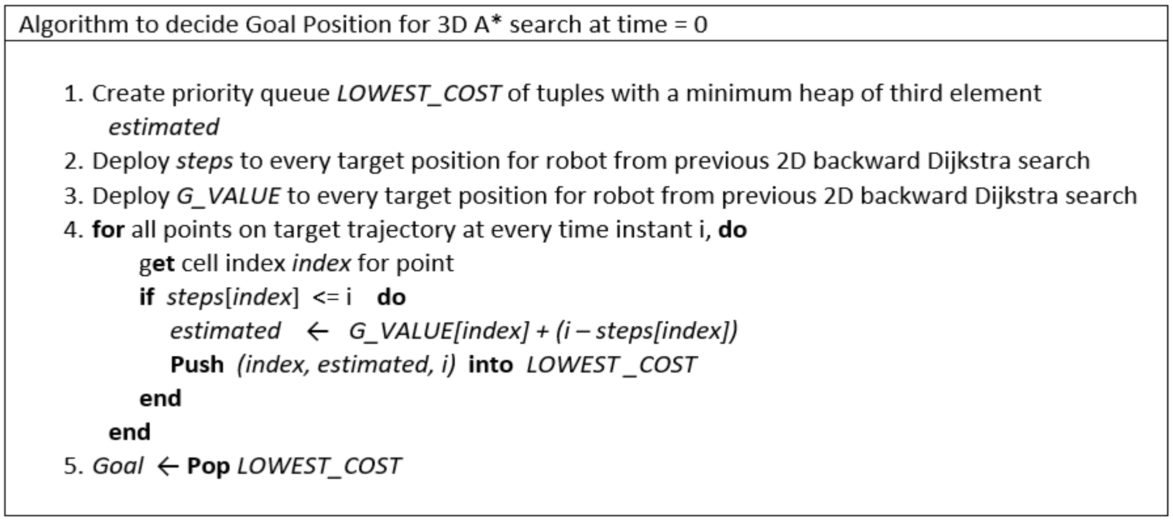
```
Dijkstra 2D backward search to define goal position

Input: Goal (robot position at time 0), map_size, collision_threshold, current time step,
Output: G_VALUES, steps
   1. Create priority queue OPEN with a vector container of states with a minimum heap of the
      G_VALUE
   2. Create container vector G_VALUES of map_size
   3. Create container vector EXPANDED of map_size
   4. Create an array steps of map_size
   5. Create Goal state of structure x, y, g, t
   6. Push Goal state into OPEN
   7. G_VALUES[Goal state] ← 0
   8. While OPEN is not empty do
   9.     s ← OPEN.top()
   10.     OPEN.pop()
   11.     If not EXPANDED[s] do
   12.         EXPANDED[s] is true
   13.         G_VALUES[s] ← s.g
   14.         For every neighbor of s do
   15.             temp ← s.t + 1
   16.             If neighbor is in map and cost[neighbor] < collision threshold do
   17.                 If EXPANDED[neighbor] is false do
   18.                     If cost[neighbor] > g[neighbor] + cost[s] do
   19.                         cost[neighbor] = g[neighbor] + cost[s]
   20.                         Push neighbor into OPEN
   21.                         steps[neighbor] ← temp
   21.                     end
   22.                 end
   23.             end
   24.         end
   25.     end
   26. end
```

Figure 1: 2D backward Dijkstra search

3.2.2 Defining a goal position

After conducting a backward 2D Dijkstra search, we evaluate each point on the target trajectory to decide on a goal position that will guarantee completeness of the solution when calculating the heuristic and performing A* search. The algorithm is as shown below. This goal decided at time instant 0 is set as a static goal through every execution of the map.



```
Algorithm to decide Goal Position for 3D A* search at time = 0

1. Create priority queue LOWEST_COST of tuples with a minimum heap of third element
   estimated
2. Deploy steps to every target position for robot from previous 2D backward Dijkstra search
3. Deploy G_VALUE to every target position for robot from previous 2D backward Dijkstra search
4. for all points on target trajectory at every time instant i, do
       get cell index index for point
       if steps[index] <= i  do
           estimated  ← G_VALUE[index] + (i – steps[index])
           Push (index, estimated, i) into LOWEST_COST
       end
   end
5. Goal ← Pop LOWEST_COST
```

Figure 2: Defining the goal

3.2.3 Computing a one time 2D Dijkstra Heuristic

Heuristic Computation is done exactly once at time step 0 using 2D Dijkstra search. It is stored as a static vector container HVALUES for the duration of the code execution. The same function and algorithm is used as in subsection 1 except the goal is now the predefined goal instead of the robot's start position

3.2.4 3D A* Search

A regular 3D A star search was done using the 2D Dijkstra heuristic calculated at the initial time instant. For the 3D search, the number of moves the robot could perform was 9. The difference being the ninth move allowed the robot to stay in the same position. This allowed for the robot to reach the goal computed in section 2 before the target did. It continued to wait in the low cost goal state until it caught the target. Both completeness and optimality are guaranteed with this solution. States are expanded based on the sum of their g and h values. A closed state is also defined in the form of a priority queue that stores states that have been expanded. Backtracking is then performed on the goal state to get the required action at the given time instant. The function Astar3D computes this.

```
3D A* search

Input: Goal (pre-defined goal), map_size, collision_threshold, current time step,
Output: robot trajectory
  1. Create priority queue OPEN with a vector container of states with a minimum heap of the
     G_VALUE
  2. Create priority queue CLOSED with a vector container of states with a minimum heap of the
     F_VALUE   (g+h)
  3. Create container vector G_VALUES of map_size
  4. Create container vector EXPANDED of map_size
  5. Create Goal state of x, y, t, g, f, a
  6. Push Goal state into OPEN
  7. G_VALUES[Goal state] ← 0
  8. While OPEN is not empty do
  9.     s ← OPEN.top()
  10.    OPEN.pop()
  11.    CLOSED.push(s)
  12.    If not EXPANDED[s]  do
  12.        EXPANDED[s] is  true
  13.        G_VALUES[s]  ← s.g
  14.        For every neighbor including current cell of s do
  15.            If neighbor is in map and cost[neighbor] < collision threshold do
  16.                If EXPANDED[neighbor] is false do
  17.                    If cost[neighbor] > g[neighbor] + cost[s] do
  19.                        cost[neighbor] = g[neighbor] + cost[s]
  20.                        Push neighbor into OPEN with new f value
  21.                    end
  22.                end
  23.            end
  24.        end
  25.        If state s == goal do
  26.            backtrack(CLOSED)
  27.        end
  28.    end
```

Figure 3: 3D A* search

3.3. Fine tuning

3.3.1 Time for initial step

Since both forward and backward Dijkstra were computed during the initial time instant, some concerns are raised with regard to the robot's ability to spit out its first action during the very second. So a clock was deployed which had sensitivity in the order of milliseconds. Small maps like 3 and 4 barely took 50 milliseconds or so for the initial computation. Larger maps like 1 and 2 took around 600 milliseconds. This being said all the maps obeyed the 1 second time constraint for action computation even at the initial time instant. So no headstarts on behalf the target had to be accounted for.

	Map 1	Map 2	Map 3	Map 4
Target caught	1	1	1	1
Time Taken	2641	4673	244	380
Moves made	2639	1235	242	266
Path cost	2641	4452615	244	380

Figure 4: 3D A* search

3.3.2 Weighted A* search

Experiments were performed with weighted A star search. They resulted in nearly the same cost and number of steps taken but optimality could not be guaranteed since weighted A* is suboptimal.

4. Evaluation of Results

4.1. Evaluation of cost and time taken

The table below highlights the cost and time taken for each map. Images of the proposed trajectory have also been shown.

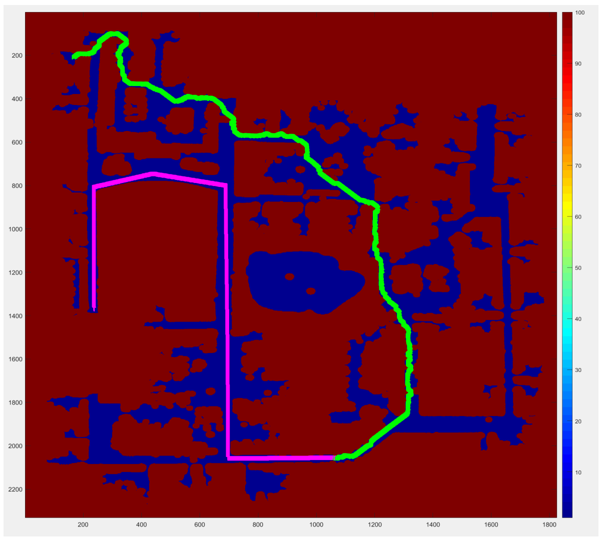


Figure 5: Map 1

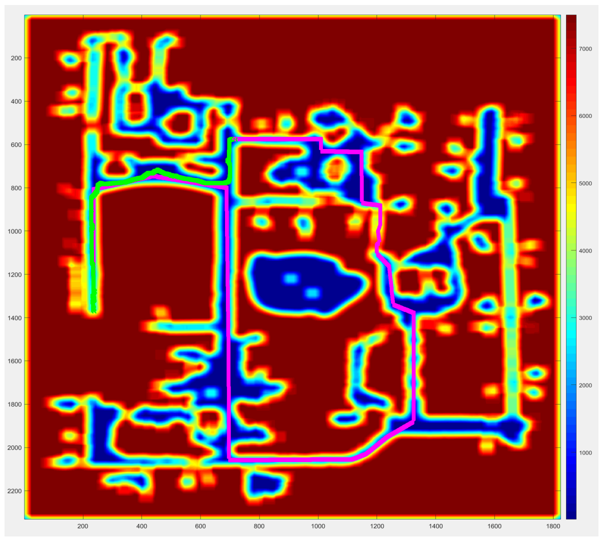


Figure 6: Map 2

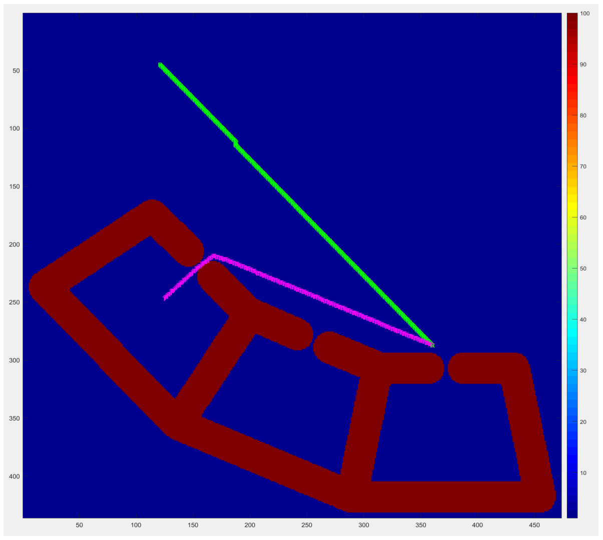


Figure 7: Map 3

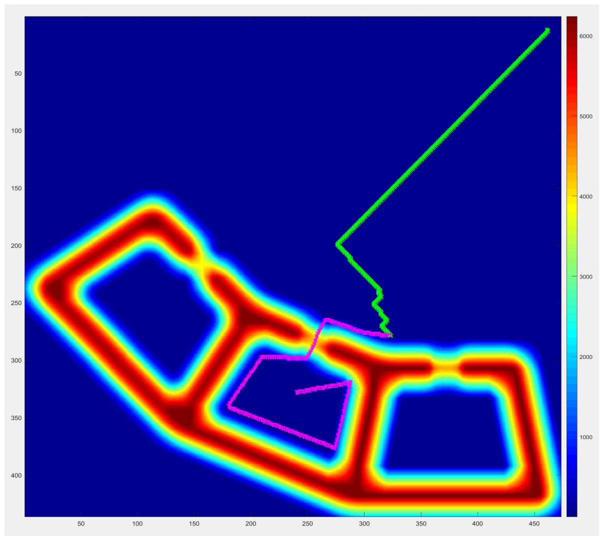


Figure 8: Map 4

4.2. Evaluating completeness of the solution

This approach guarantees completeness of the search. In earlier trials, I proposed a 2D A*star search with a 2D Dijkstra Heuristic. In its pursuit of a lower cost, there was no way that it could guarantee completeness since it would often take more time to reach the pre-defined goal than the target through the A* search. In small maps, where the target's trajectory lasts for a few seconds, this approach was bound to fail. So in my final approach with a 3D A*star search **completeness is guaranteed** with a minor tradeoff in terms of cost especially for targets with trajectories of a shorter duration. I have tested it on smaller maps and where time is of essence, and it catches the target with a success rate of a 100 percent.

4.3. Evaluating optimality of the solution

The **heuristic in this case is 2D Dijkstra. It is both admissible and consistent** since its H value is always 0. It will never overestimate the cost to the goal. A* search guarantees optimality as long as the heuristic is admissible, meaning the path found in this experiment is always optimal.