# Planning & Decision-making in Robotics

Manuj Trehan
Andrew ID: mtrehan

**Homework 1**

16-782
September 30, 2022

## Planner Summary

To solve the planning problem, I implemented a *forward weighted A* multi-goal* planner with an informative heuristic. For the graph nodes, I created a struct with the following structure:

Node

1. map index (output of GETMAPINDEX)
2. time
3. g-value
4. h-value
5. f-value
6. shared_pointer to parent node

My robot state is 3-dimensional - $< x,\ y,\ t >$. The grid is actually 9-connected instead of 8. At any point, the robot has the possibility to move to 9 other states in the next time layer - 8 neighbors + staying in its current cell.

To save on memory, I only store one copy of my graph nodes, and use *shared_pointers* to the nodes for all other maps/lists/queues. Since it is a smart pointer, this also takes care of automatically deleting the memory allocated on the heap once all pointer instances are destroyed.

I used the following data-structures:

- An unordered map to map $int - shared\_pointer$ and store my graph of nodes
- A priority queue of *shared_pointers* for the open list, prioritized based on the f-values using a custom comparator
- An unordered set of *int* for the closed list
- A stack of *int* to store the computed path. Once the entire path is computed, the algorithm backtracks stores all the nodes in the stack. The planner keeps popping a node at every time-step and returns it for the next move

For maps/sets, I use a custom hash function - for every state, I append the time value to the end of the map-index. Since every state will have a unique set of (map-index, time) pair, this hash function will work. E.g. the grid-cell with a map-index 2535 at time-step 43, will be mapped to 253543 based on the hash function.

To calculate the heuristics, I used a lower dimensional (2D) planning problem, and performed a backward Dijkstra search from all possible goal cells. I used the $2^{nd}$ half of the target trajectory as goals. The heuristic cost for a grid cell in the map is basically the *Dijkstra cost to the closest goal*.

Apart from these standard data-structures, I also used the following:

- An unordered map which maps $int - int$. This is used as a multi-goal map. It stores all the possible goals, i.e. all map-index positions of the target as the key, and the time-step for each goal as the value
- An unordered map which maps $int - pair\ of\ ints$. This is used to store the heuristics from the 2D backward Dijkstra approach. It stores the heuristic cost for each valid grid-cell in the map. It stores the index of the cell as the key, and stores a pair in the value. In the pair, it stores the heuristic cost of the closest goal as the first element. In the second element, it stores the time-step of the closest goal to the grid cell.

In the main path computation loop, I add a time factor to the heuristic for each state to make it more informed. So for each state, the final heuristic is a sum of the heuristic cost calculated by the backward 2D Dijkstra approach, and the time difference between the current time, and the time of the goal which is closest the state. So, for every state being explored which is not a goal index,

$$h_s = 2D\ Dijkstra\ cost\ to\ closest\ goal + (time\ of\ closest\ goal - current\ time) \tag{1}$$

Here, $current\ time = time\ of\ the\ state + current\ execution\ time$

The current execution time of the planner had to be added since the object will skip those many time-steps in its trajectory once a path is computed. I added this factor to enable a comparison between states which had same/similar 2D heuristic costs, but were leading the robot to different goals at different times. An earlier goal might be better, since catching the target early will reduce waiting costs, if any. It helps in speeding up the search, but also makes it sub-optimal.

However, if the index of the state being explored is a goal state index, then the heuristic is,

$$h_{goalindex} = (cost\ of\ goal\ index\ in\ map) \times (time\ of\ goal - current\ time) \tag{2}$$

This is to take care of the fact that if the algorithm, while exploring, reaches a goal index in the map, but the target has not reached there yet, there is a possibility for the robot to just wait at that cell until the target arrives. In this case, the cost incurred will be the cost of that grid cell multiplied by the time remaining for the target to arrive. In this case, where a goal index is being explored, the heuristic perfectly estimates the cost to the goal.

Finally, I also weighted my A* search with $\epsilon = 1.8$

The algorithm is able to perform relatively well. It was able to find paths for all 6 given maps, with a very low computation time, although some costs were sub-optimal.

## How to compile and run code

*mex planner.cpp*
*runtest('map*.txt')*

## Results

### Map 1

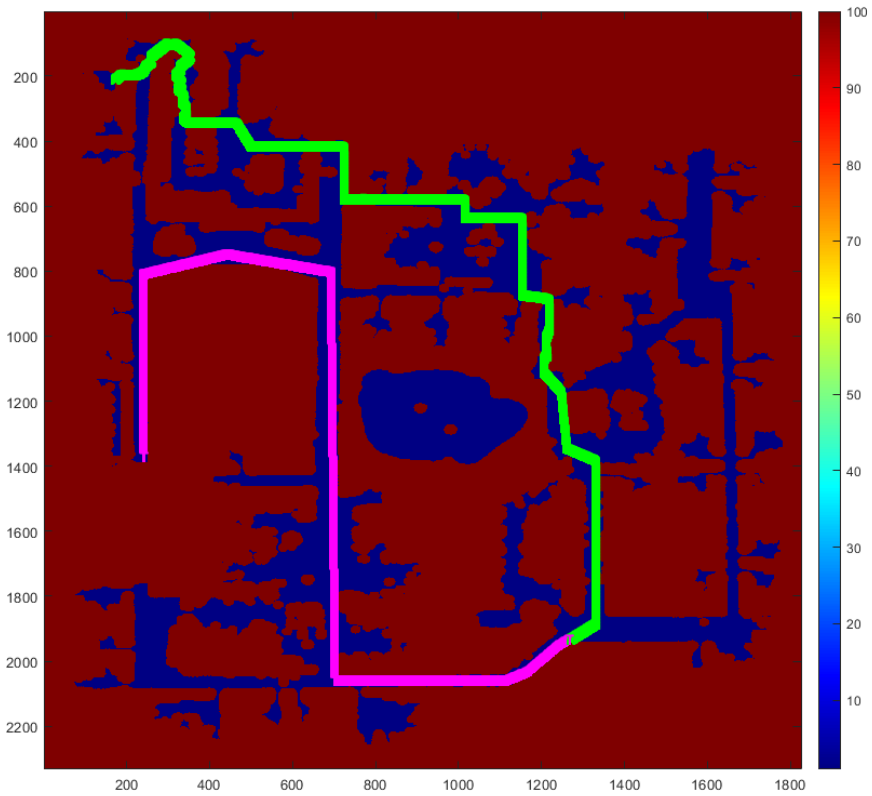


Figure 1: Map 1 Trajectory

```
RESULT:
      target caught = 1
      time taken (s) = 2871
      moves made = 2868
      path cost = 2871
```

Figure 2: Map 1 Result

**Map 2**

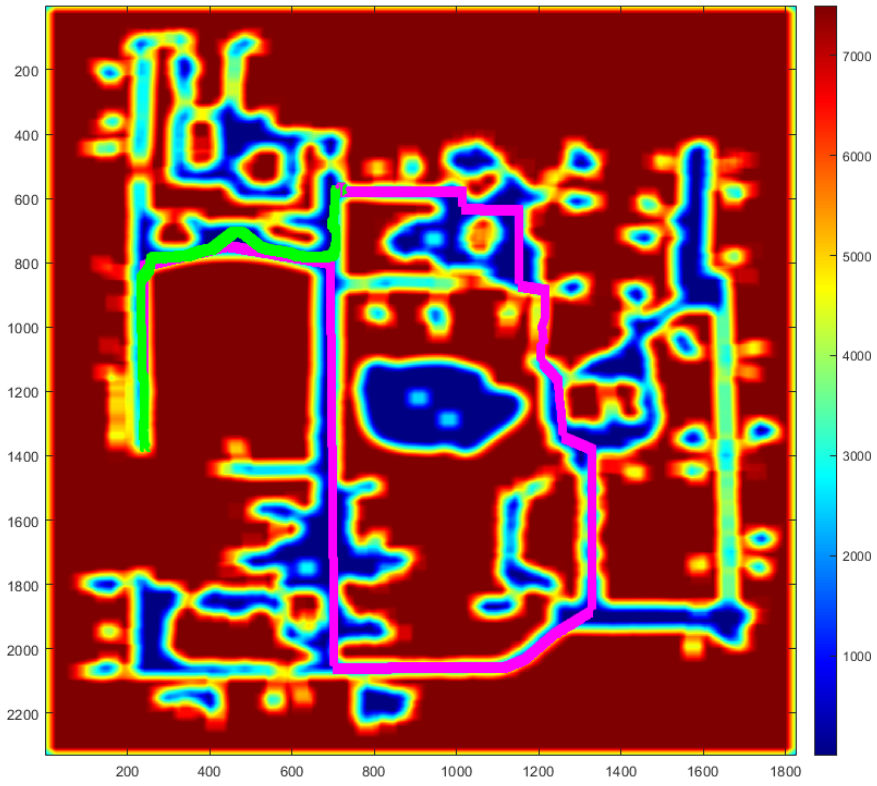


Figure 3: Map 2 Trajectory

```
RESULT:
      target caught = 1
      time taken (s) = 4685
      moves made = 4465
      path cost = 1609082
```

Figure 4: Map 2 Result

**Map 3**

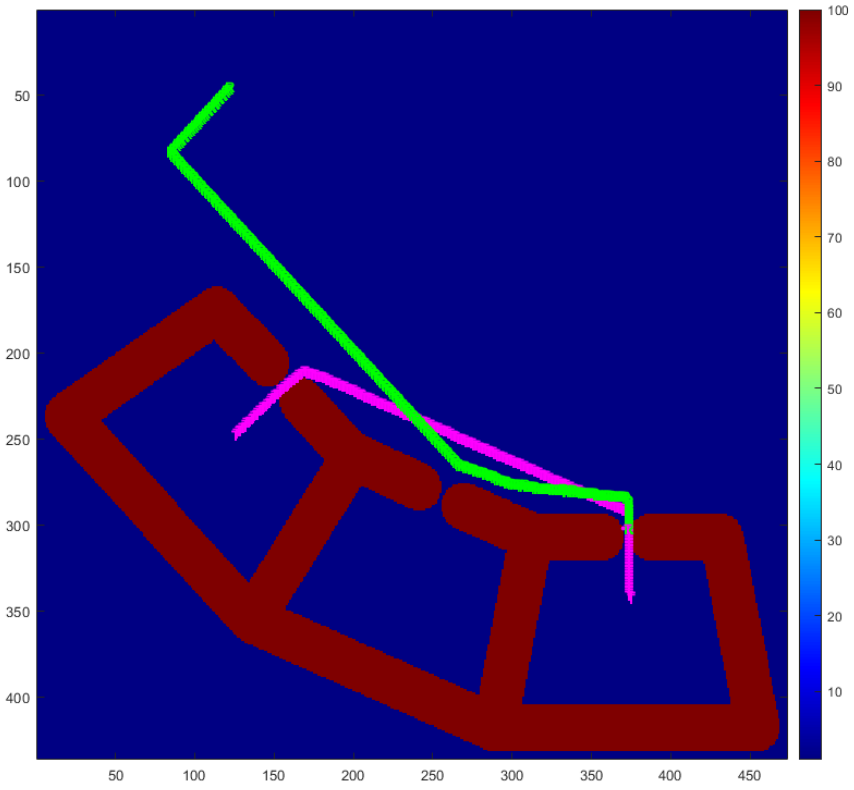


Figure 5: Map 3 Trajectory

```
RESULT:
      target caught = 1
      time taken (s) = 346
      moves made = 343
      path cost = 346
```

Figure 6: Map 3 Result

**Map 4**

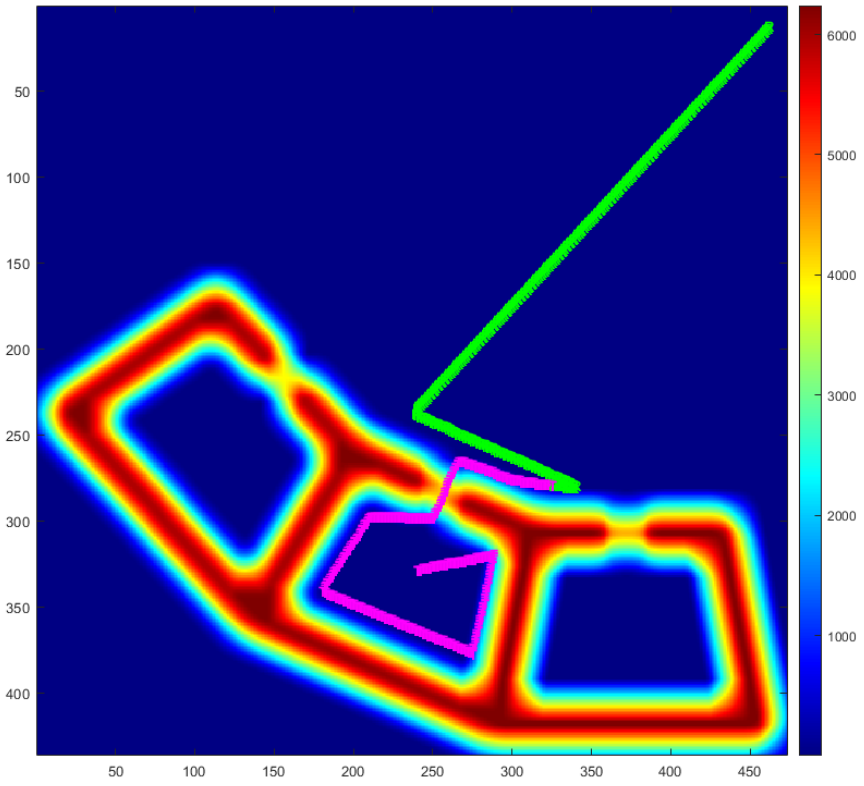


Figure 7: Map 4 Trajectory

```
RESULT:
      target caught = 1
      time taken (s) = 382
      moves made = 340
      path cost = 382
```

Figure 8: Map 4 Result

**Map 5**

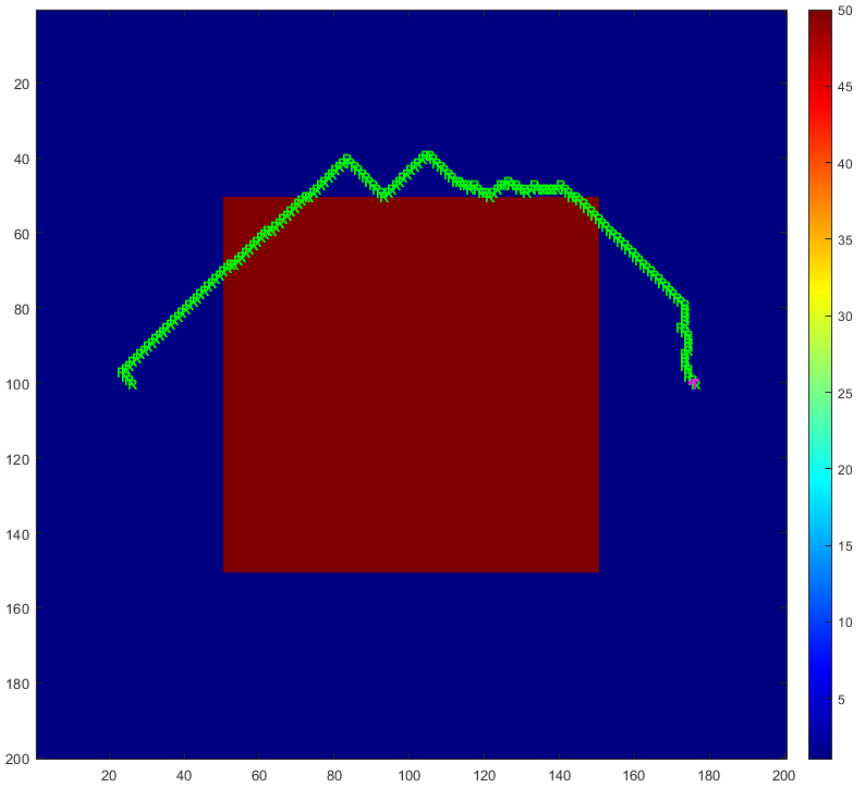


Figure 9: Map 5 Trajectory

```
RESULT:
      target caught = 1
      time taken (s) = 180
      moves made = 174
      path cost = 1503
```

Figure 10: Map 5 Result

**Map 6**

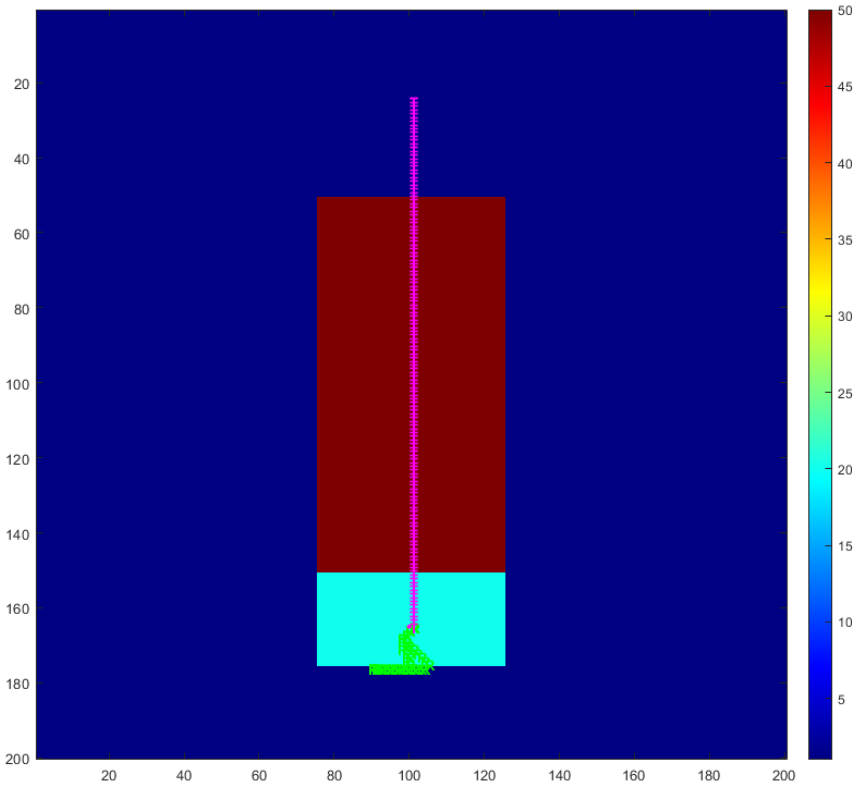


Figure 11: Map 6 Trajectory

```
RESULT:
      target caught = 1
      time taken (s) = 141
      moves made = 139
      path cost = 578
```

Figure 12: Map 6 Result

## Alternate Approach

I also tried an alternate approach which was basically the inverse of my original approach. It involved a forward 2D Dijkstra search from the robot start position as a heuristic, and a backward single-goal weighted A* approach which multiple start positions (target trajectory states).

This planner is present in the file *backward_planner.cpp* for reference

I observed that this approach worked better for maps 1, 3 and 5, where the costs were discrete values, 1 or 100. The forward approach worked better for maps 2, 4 and 6, where the cost was high and variable. Overall, I felt that the forward approach worked slightly better, since it was able to give a much better solution for map 2, while compromising only slightly on the costs of the other maps.