Homework 1 – Catching a Moving Object

Akshit Mishra (akshitm)

September 2020

Abstract

For this homework, the objective was to program a planner that allows a point robot to catch a moving object. The planner was given an 8-connected 2D gridworld. The task at hand was to generate a path for the robot that will allow it to catch the object with while incurring least cost.

1 Planner Implementation

To catch the given moving target with minimum cost incurred by the robot, I implemented various approaches to see which approach gave the most auspicious results. The next few paragraphs explain all these approaches in detail:

1.1 Approach 1: Forward A* Search with backward Dijsktra

The first method I tried to implement was to plan in 3D space with (x, y, t) as the dimensions. For this approach, a heuristic map was generated by running a backward Dijsktra search from all the **goal states**. For this case, the values of the heuristic was the number of steps it took to reach from goal state to any state on the map. In this manner, all the states on the map were expanded and assigned a heuristic value. The data structures used for this backward search were **minimum priority queue** with heuristic value as the sorting criterion. Moreover, two **hash maps** were also used for keeping track which states have been expanded fully and the ones that have been visited but not closed yet.
After completing the backward search and populating the heuristic map, the program ran a forward A* search to find the optimal goal state to intercept the cell. For this part, all the planning was done at the first time step and planning time in seconds was calculated. Using the planning time along with the time (t) dimension of a popped node from the priority queue, the program could check if a goal state could be intercepted at that state. An action map was then populated with the respective actions to take for the robot in that case.
This approach provided an optimal solution for the smaller maps, map 3 and map 4. However, the approach could only find the solution in the required time for map 1 if a **weighted A*** approach was implemented with a weight of at least 7. Although this approach worked well for maps 1, 3, and 4, it crashed every time it was ran for map2. MATLAB exited with a messaged reading **"Killed"** even though forward A star found a goal to intercept the target. The possible reasons for this fault could be:

- Memory leak: It is possible that all the memory from the heap was not deallocated properly.
- Too much RAM space claimed by the program for bigger maps when planning in 3D.

1.2 Approach 2: Forward Dijsktra in 3D

Since approach I did not yield favorable results for all the maps, for the final solution I implemented a simple Dijsktra search in 3D. The data structures used for this were a **priority queue** with a custom class for comparing the g values of the cells, **hash maps** for storing the visited and closed cells. I also used **hash maps** for storing the action maps for the robot. As shows in figure 1, in this algorithm three lists are maintained: open list, closed list, and visited list. Once a cell is removed from the priority and added to the closed list, its 8 successors are explored. If any of the successors have already been fully expanded and added to the closed list, the algorithm moves to the next successor. If a successor has not been visited, a cell is created and added to the open and visited list and if the cell has been visited then the g value is checked for optimality.
Once the open list is exhausted, the algorithm find a suitable cell to intercept by looping over the target's trajectory. From there on, simple backtracking is performed to populate the map of actions to be taken by the robot. All of these steps are performed at the first time step and the following time steps just use the already populated action map.
For efficient memory management of all the cells that were dynamically allocated on the heap, the cell structure class had a destructor to delete the parent of the cell. This insured that the memory on the heap was cleared after use. Moreover, the priority queue and all the hash maps were also cleared and reset after use. Each cell on the map is represented as a custom structure called **SearchCell** with the following representation:

Parameter type	Name	Description
int	x_{pos}	x coordinate of the cell
int	y_{pos}	y coordinate of the cell
int	$cost_{val}$	map cost of the cell
int	t_{val}	steps required to reach the cell
double	g_{val}	optimal cost to reach a cell
*SearchCell**	parent	parent of the cell

Table 1: Structure of a map cell.

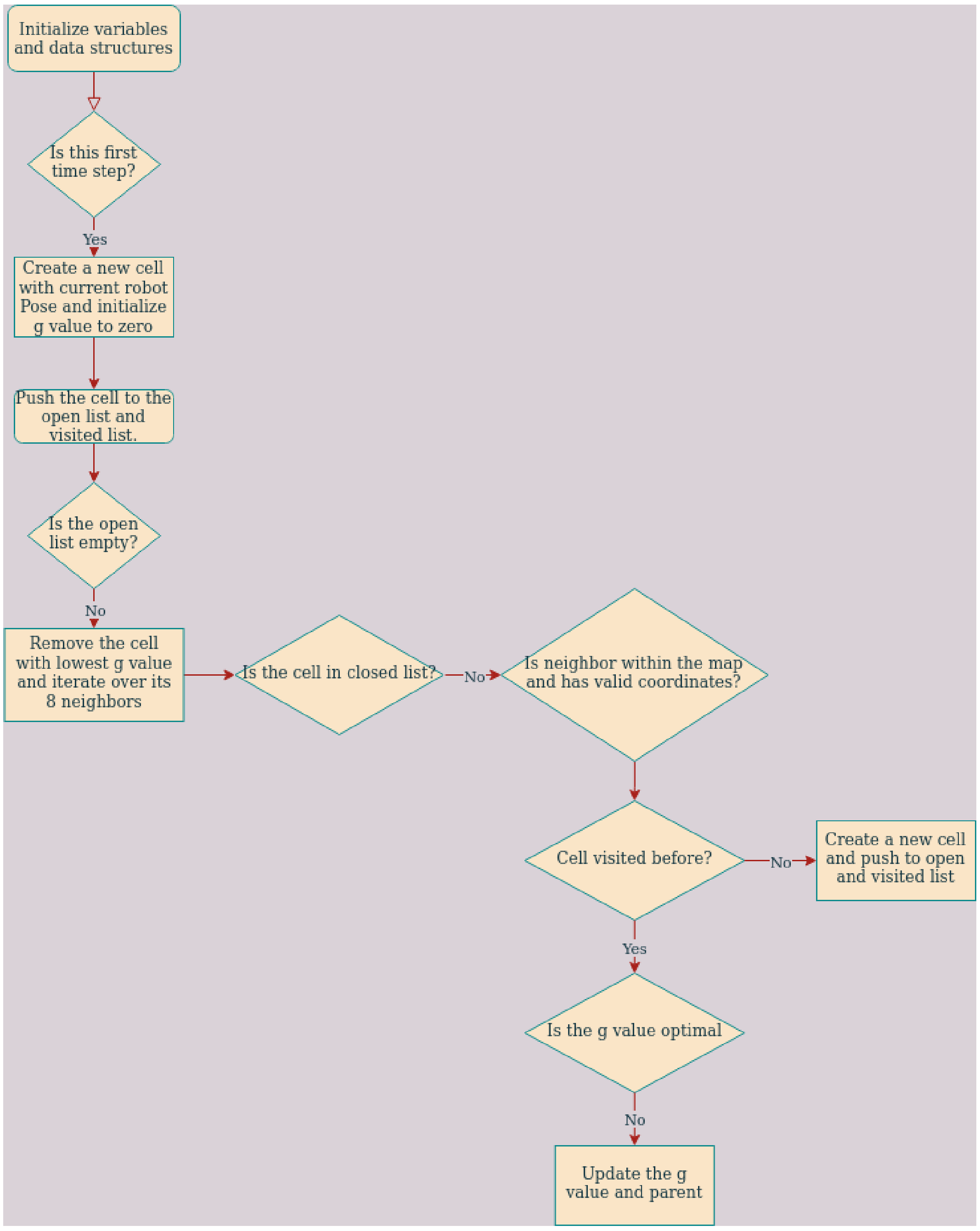
The cell where the robot should intercept the target is found by looping through all the points on the target's trajectory and finding the step with minimum number of required steps by the robot (t_{val} in table 1). The final action sequence for the robot is generated by backtracking from the intercept cell using the *parent* member variable from the intercept cell. The flowchart in figure 1 shows the logic behind the forward Dijsktra search.

1.3 Use Case

"mex planner.cpp"
"runtest mapXX.txt"

Files required to run: <planner.cpp> <SearchCell.hpp>

1.4 Dijsktra Forward Search Algorithm Flowchart



Initialize variables and data structures
Is this first time step?
Yes
Create a new cell with current robot Pose and initialize g value to zero
Push the cell to the open list and visited list.
Is the open list empty?
No
Remove the cell with lowest g value and iterate over its 8 neighbors
Is the cell in closed list?
No
Is neighbor within the map and has valid coordinates?
Cell visited before?
No
Create a new cell and push to open and visited list
Yes
Is the g value optimal
No
Update the g value and parent

Figure 1: Algorithm Flowchart for Dijsktra forward search

2 Results

The results from the four provided maps using Dijsktra search are provided below:

2.1 Map 1

```
RESULT:
            target caught = 1
            time taken (s) = 5345
            moves made = 421
            path cost = 5345
```

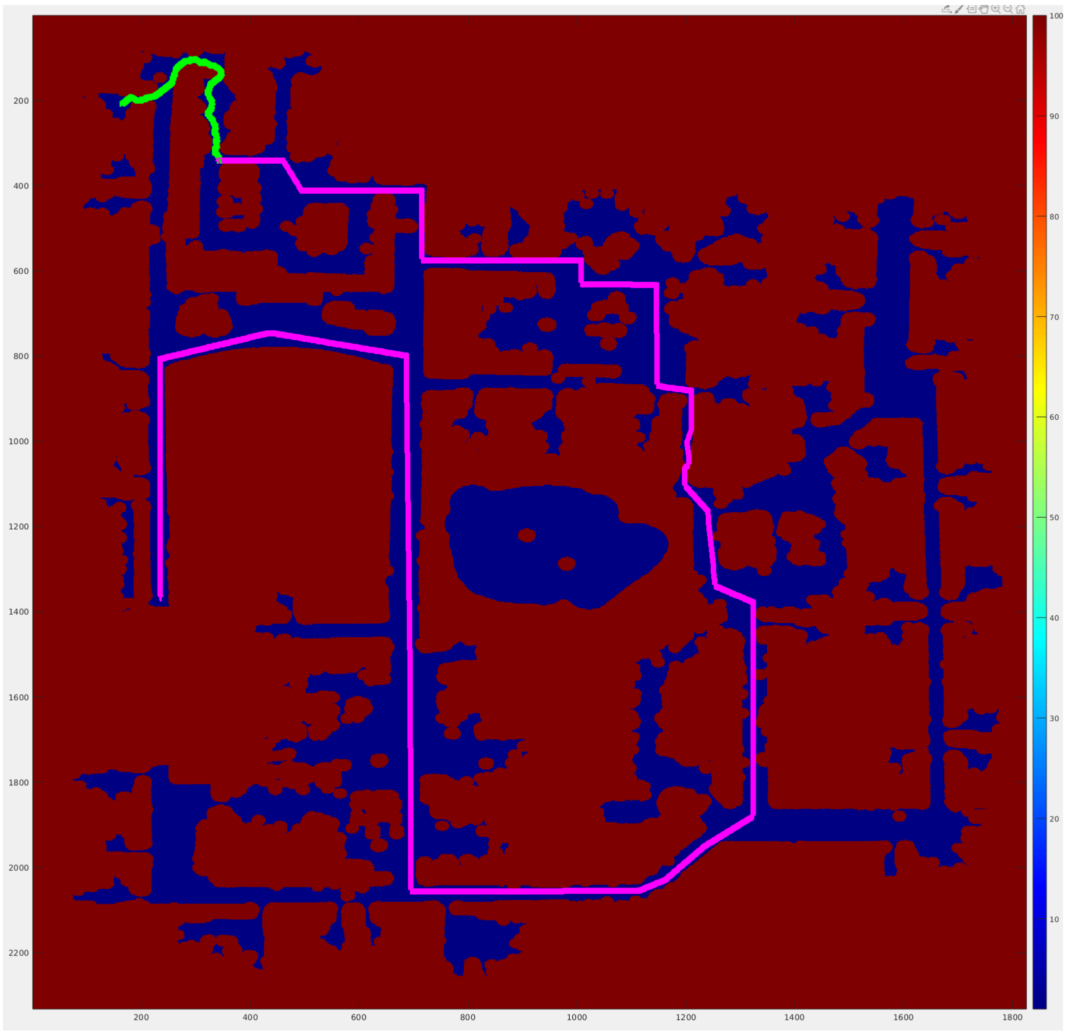


Figure 2: Map 1 executed trajectories

2.2 Map 2

RESULT:

```
    target caught = 1
    time taken (s) = 5245
    moves made = 1728
    path cost = 9665268
```

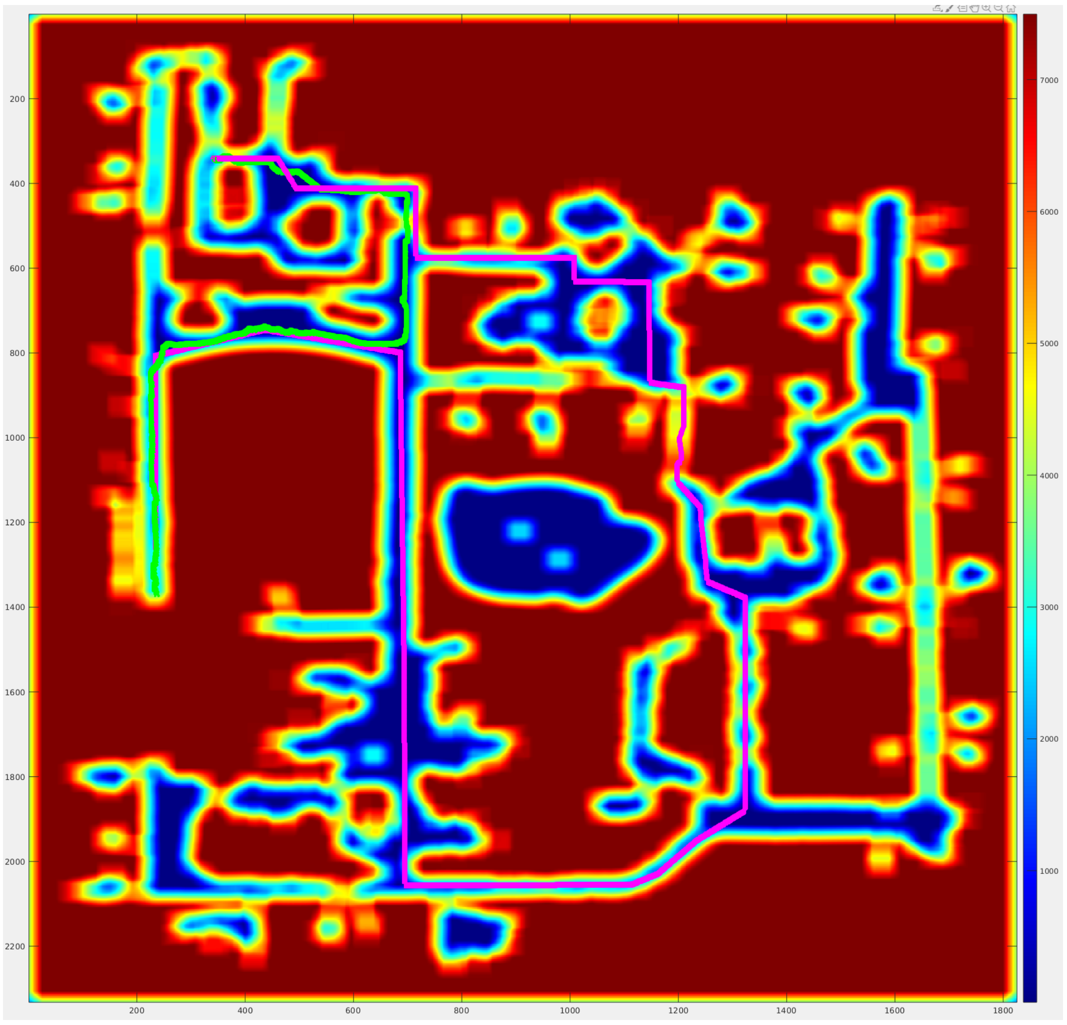


Figure 3: Map 2 executed trajectories

2.3 Map 3

RESULT:

```
        target caught = 1
        time taken (s) = 792
        moves made = 283
        path cost = 792
```

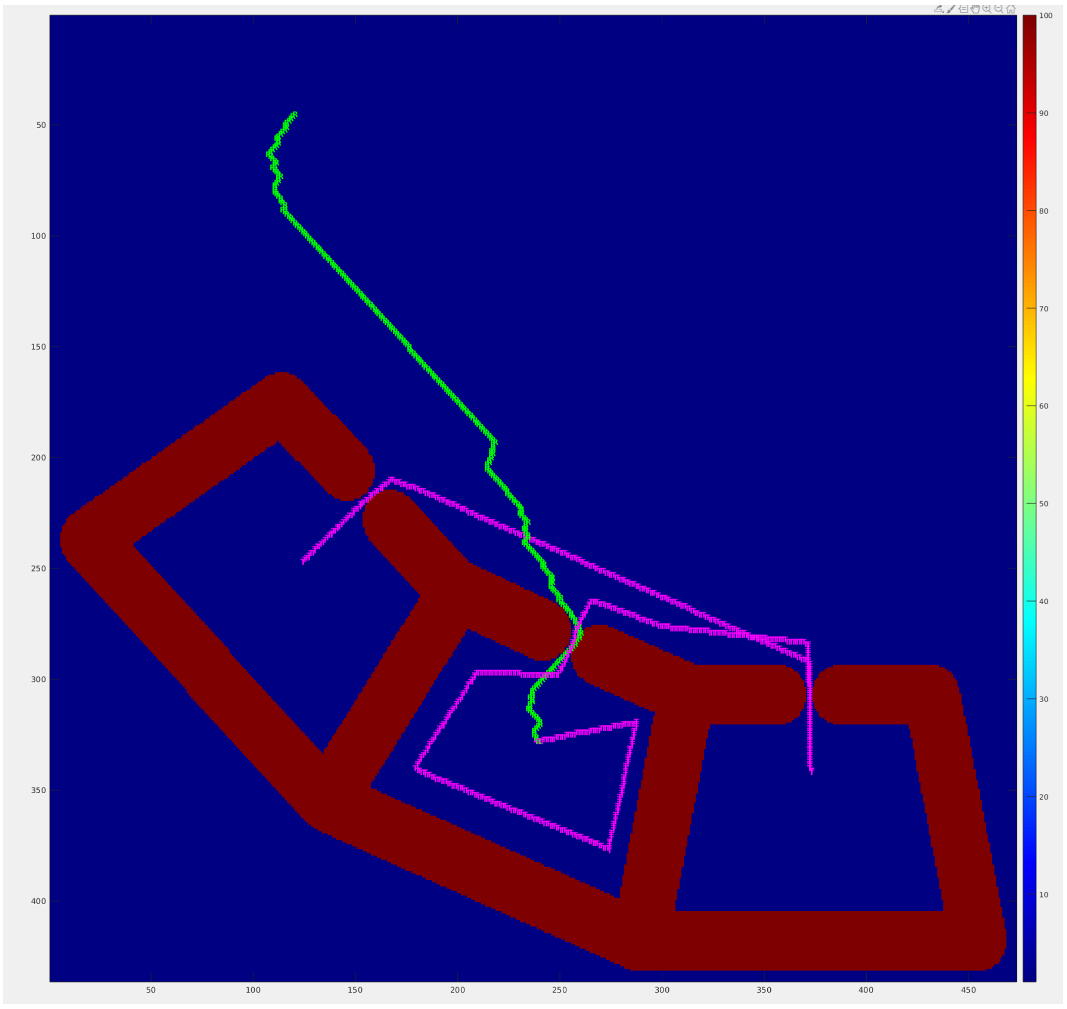


Figure 4: Map 3 executed trajectories

2.4 Map 4

RESULT:

```
        target caught = 1
        time taken (s) = 792
        moves made = 336
        path cost = 70272
```

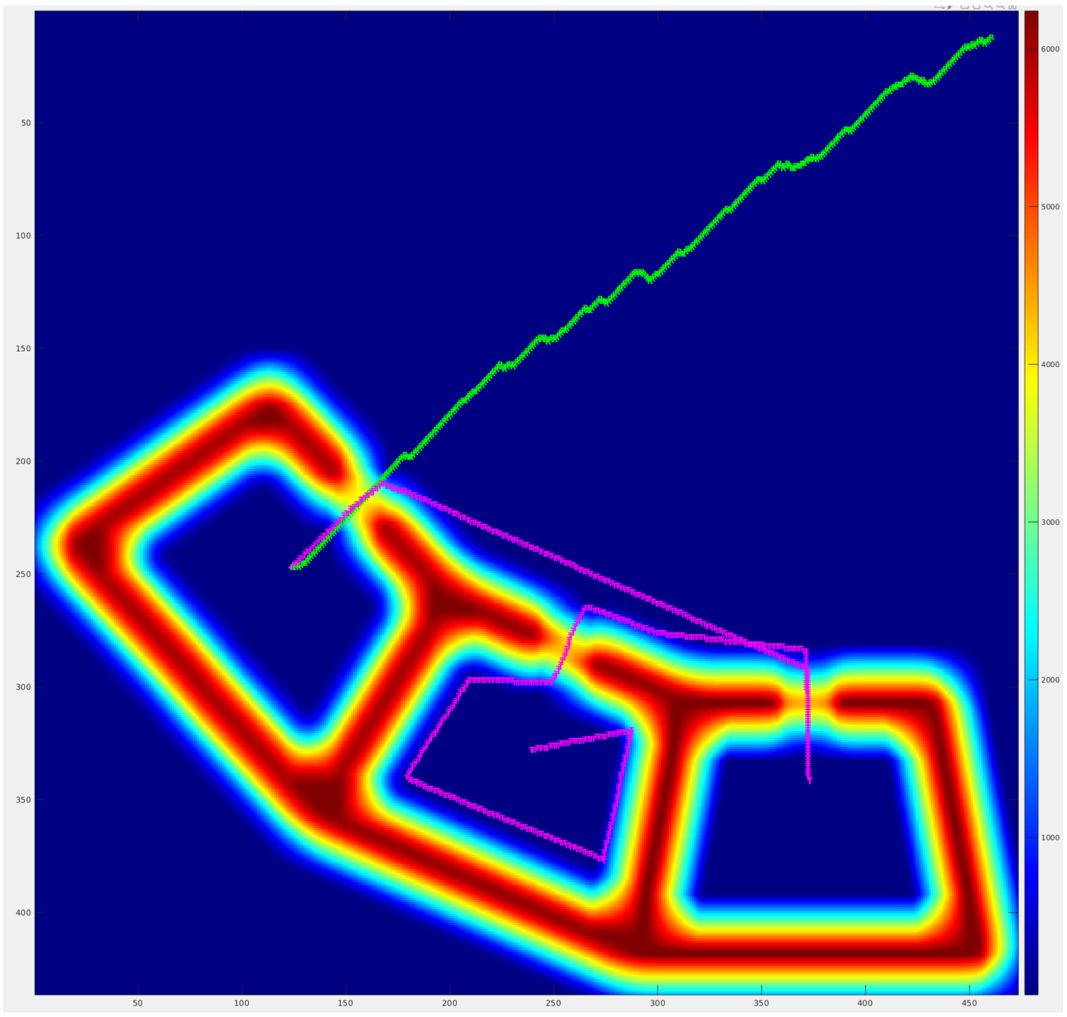
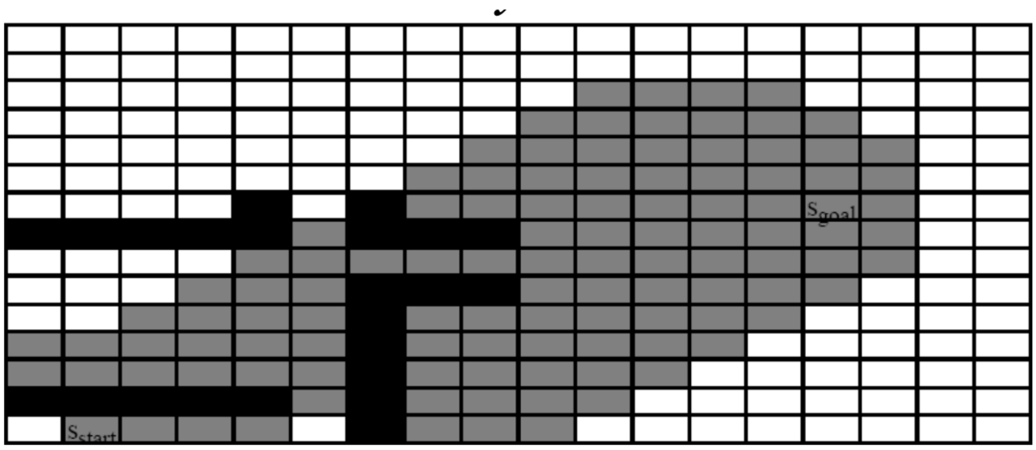


Figure 5: Map 4 executed trajectories

Problem 2

Although weighted A* without re-expansions comes guarantees the same sub-optimiality as weighted A* with re-expansions, there might be some cases where we would want to allow re-expansions. For instance, in figure 6 the environment changes as the robot moves from its starting position. Therefore, in that case if the algorithm re-expands the states, it would be able to find the solution instead of generating a plan through the obstructed path. [Lik20]



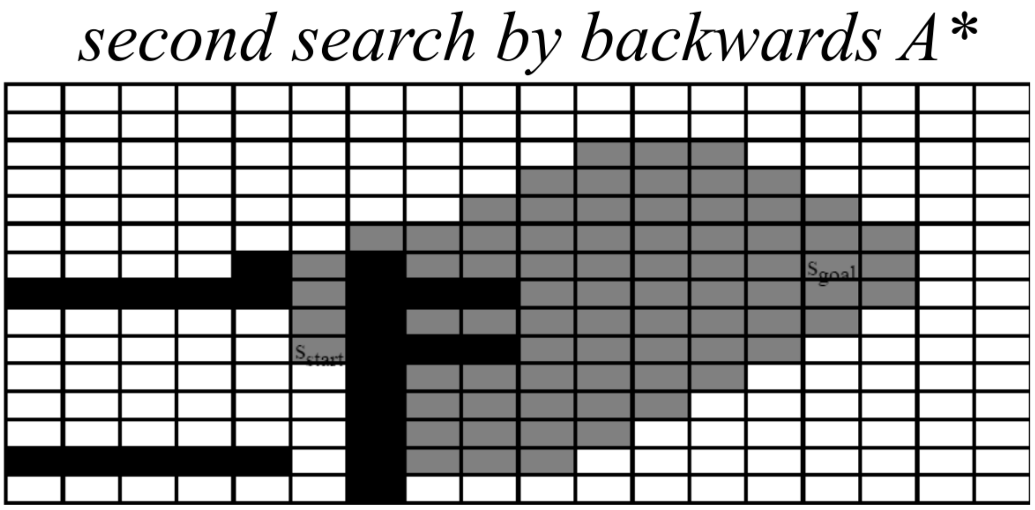


Figure 6: Reuse state values from previous searches

References

[Lik20] Maxim Likhachev. “Interleaving Planning and Execution: Anytime and Incremental A*”. In: (2020). https://www.cs.cmu.edu/~maxim/classes/robotplanning_grad/lectures/execanytimeincsearch_16782_fall20.pdf.