Planner Summary:

I implemented a forward planning weighted A* multi-goal planner with a backward Dijkstra's as a 2D heuristic. The A* algorithm I implemented considers the robot state to be 3 dimensional in [x, y, t], where x and y are spatial coordinates and t is the time. As the robot moves to the next time layer, it can either stay in its current cell or move to one of its 8 neighbors.

For the graph nodes, I defined a struct containing:

- **mapIndex**: Output of GETMAPINDEX.
- **time**: The time at which the node is visited.
- **g-value**: Cost from the start to the current node.
- **h-value**: Heuristic cost from the current node to the closest goal.
- **f-value**: The sum of the g-value and h-value.
- **parent**: A pointer to the parent node to allow backtracking.

Memory and Data Structures:

To optimize memory usage, I only store one copy of each graph node and use pointers for all other references, which simplifies memory management and prevents memory leaks. Specifically, the following data structures were used:

- Priority Queue (**openQueue)** which is used for the A* open list, prioritizing the order of nodes based on the f-value
- Unordered_map (**nodes)** which maps the node's unique index, of type *int*, to the node information
- Unordered_set (**closed**) serves as the closed list, storing all the visited nodes
- Stack (**actionStack**) stores the computed path by backtracking from the goal to the start once the path is computed
- Unordered_map (**heuristics)** which maps the node's map position, *int*, to a pair of <*int, int*> where the first element is the h-value and the second stores the time-step of the closest goal to the grid cell
- Unordered_map (**goals**) stores every position of the target in its trajectory which is used as a multi-goal map with the key being the target positions' map index and the value is the timesteps for each goal position

Planner & Heuristic:

I used a lower-dimensional (2D) planning problem to calculate the heuristics, followed by a backward Dijkstra search. I used the second half of the target trajectory to determine the goals. It is the Dijkstra cost to the nearest goal that determines the heuristic cost for a grid cell on the map.

In the A* search, I employ a weighting factor of ϵ=1.2 to expedite the search process by prioritizing nodes with lower heuristic costs, although this may lead to suboptimal solutions. During the pathfinding loop, the planner assesses whether the current state corresponds to a goal state. Upon reaching a goal, it backtracks from the goal to the start, adding each node along the path to the action stack.

Within the backtrack, I have a conditional check which verifies if the current state is a goal state and if the robot has arrived at the goal at the anticipated time. If both conditions are met, the planner backtracks to record the path by pushing each state's map index onto the action stack, and then removes the start node from the stack to finalize the recorded path.

The planner explores neighboring nodes by expanding in 8 possible directions. For each neighboring state, it verifies that the new position is valid (i.e., within the grid boundaries and collision-free) and checks if it yields a lower cost than any previously evaluated state. For each state, the final heuristic is a sum of the backward Dijkstra's heuristic cost and the time difference between the current time and time of the closest state, the goal.

$$h_s = \text{Dijkstra heuristic cost to the goal} + (\text{time of the goal} - \text{current time})$$

When the state being expanded is identified as a goal index and the robot arrives before the target, the robot can pause at the goal until the target reaches the same position. In these scenarios, the heuristic is computed as:

$$h_{goal} = (\text{cost of the goal cell}) \times (\text{time of the goal} - \text{current time})$$

which accurately reflects the necessity for the robot to wait until the target arrives. This consideration ensures that the cost incurred by waiting is factored into the decision-making process, optimizing the overall pathfinding strategy.

This algorithm is able to perform the best out of the approaches I have tried, allowing for low cost and optimization on time. Although, I am unable to get it working for map7. I suspect it might be because I am giving more importance to cost optimization than time optimization as my implementation scales well to map1 and map2.

Approaches tried:

Before going ahead with my final planner and heuristic, I had first implemented a basic A* with euclidean distance as the heuristic. In that version of the planner I was pre-computing the cost map to traverse and finding the most optimal path for each call of the planner. This was not memory efficient at all, nor was it planning "on-the-fly". This implementation was useful to test my understanding of A* and look at areas to iterate.

For my second take at the planner, I went ahead with a 2D forward multi goal A* using a backup greedy planner with a weighted average distance heuristic. This approach definitely worked much better than the previous as I was using a heuristic that was weighting the target positions based on how far along they are in the target's trajectory. This allowed for the final point of the target's trajectory to have the highest weight. In this planner, I calculated the number of steps to a point which tells us how much time the target takes to reach that point which allowed me to figure out if the robot can reach that point in time. If the point can be reached, then the planner chooses that as the goal and backpacks to send the robot there. The backup greedy planner, using just euclidean distance as the cost, worked well to catch targets that need to be caught in

a lesser amount of time (map5 & map7). This approach worked well on all maps but the cost was not optimized enough.

This led me to the final version of my planner, which I have given a brief about above.

Code Directory Structure:

```
16782-planning/
└── 16782-HW/
    ├── build/
    ├── code/
    │   ├── src/
    │   │   ├── planner.cpp
    │   │   └── obsolete/
    │   │       └── <<older versions of planner.cpp>>
    └── maps/
        └── mapN.txt   # N → map number for each map file used in tests
```

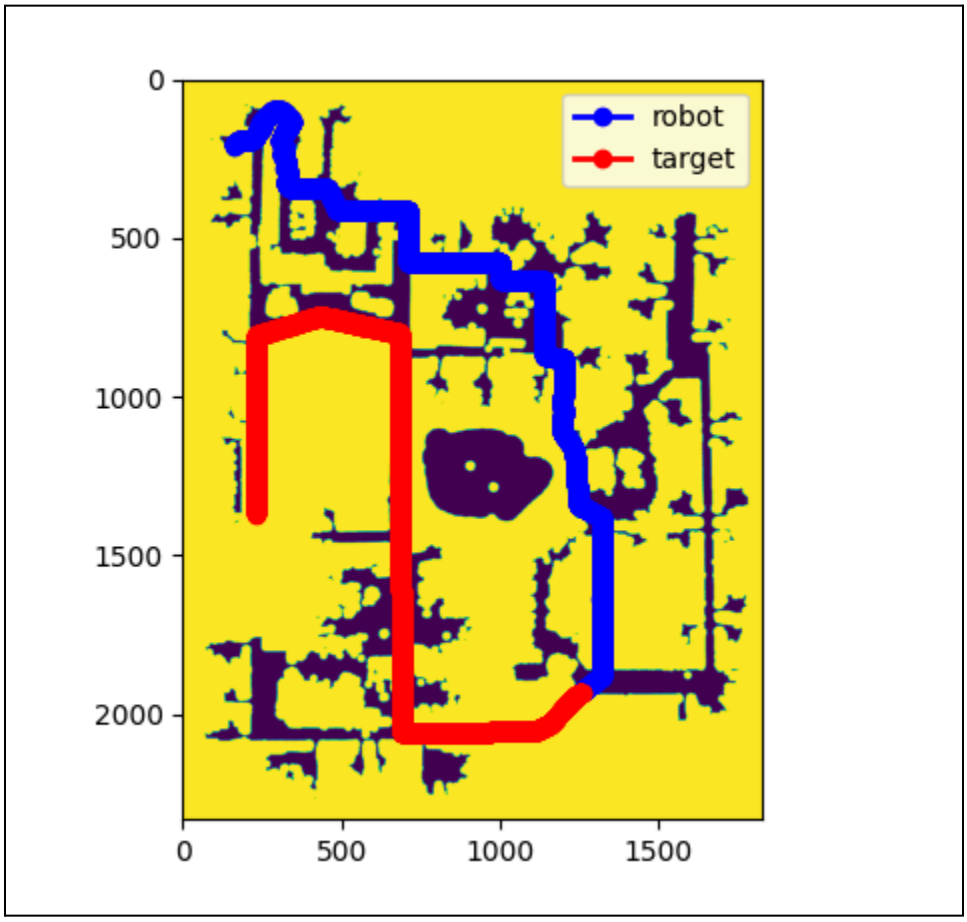
Compiling and Running the code:

```
cd build
cmake ../code && make -j8
./run_test ../maps/mapN.txt
```

Results:

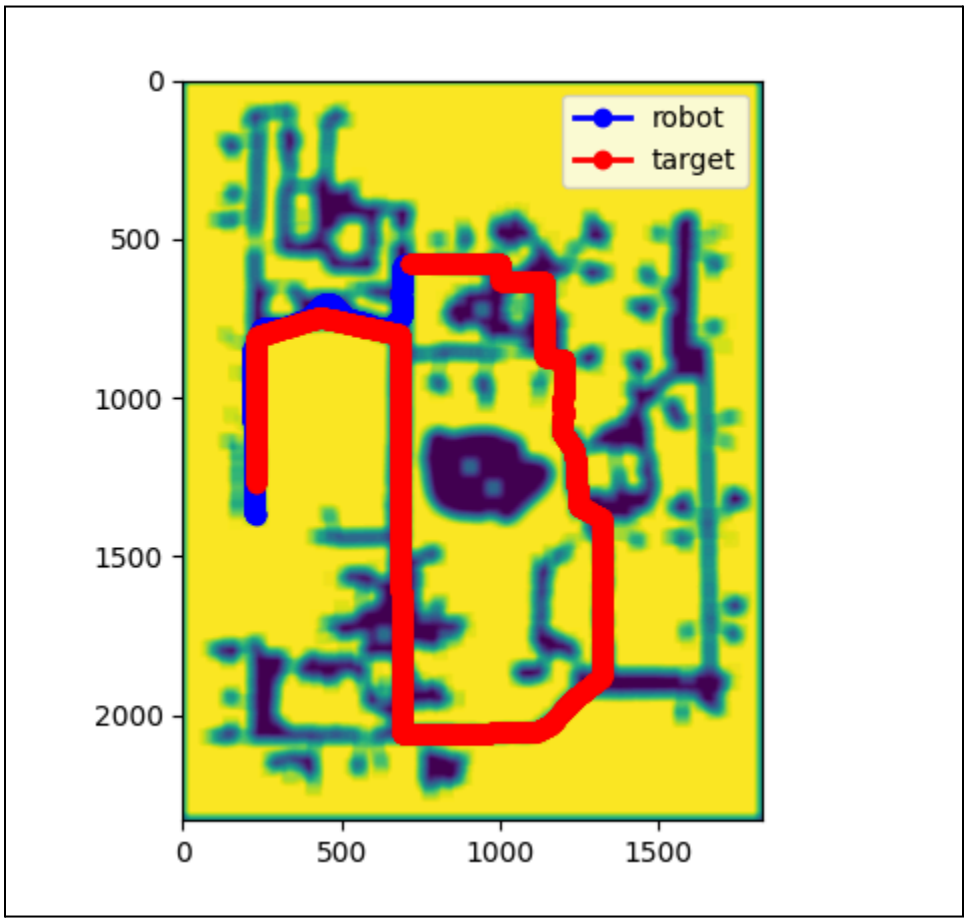
Map1.txt

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



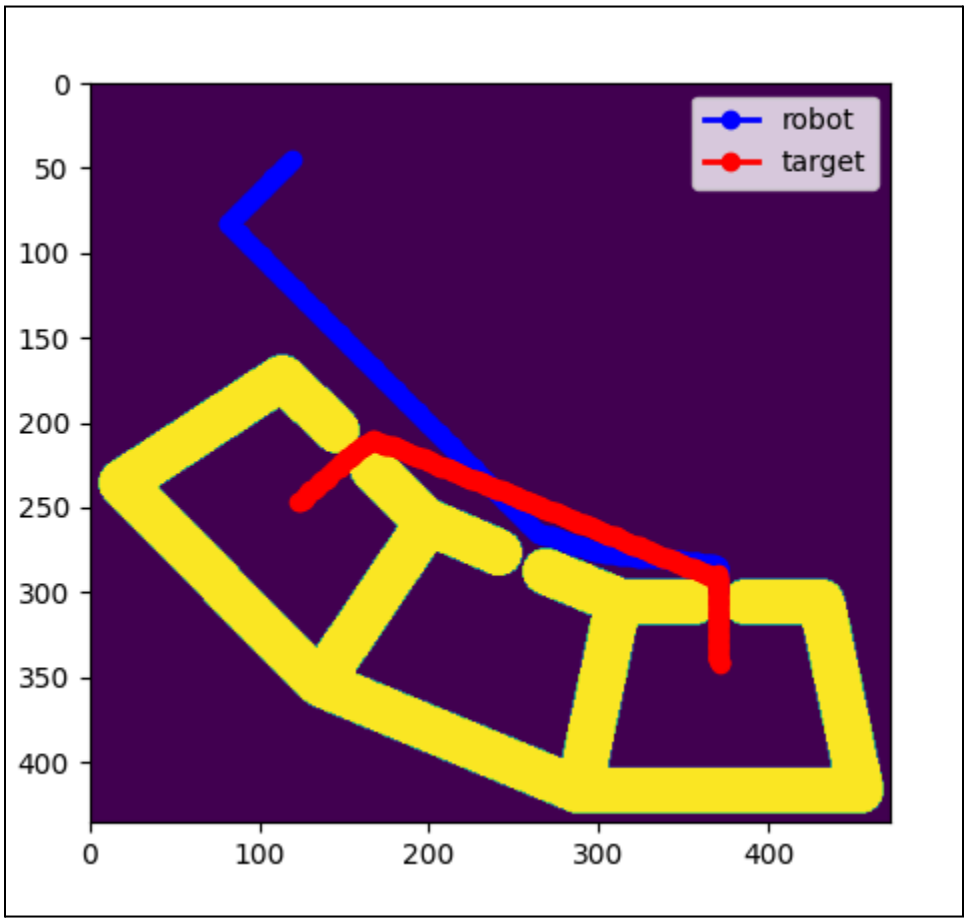
Map2.txt

target caught = 1
time taken (s) = 4680
moves made = 4448
path cost = 2393998



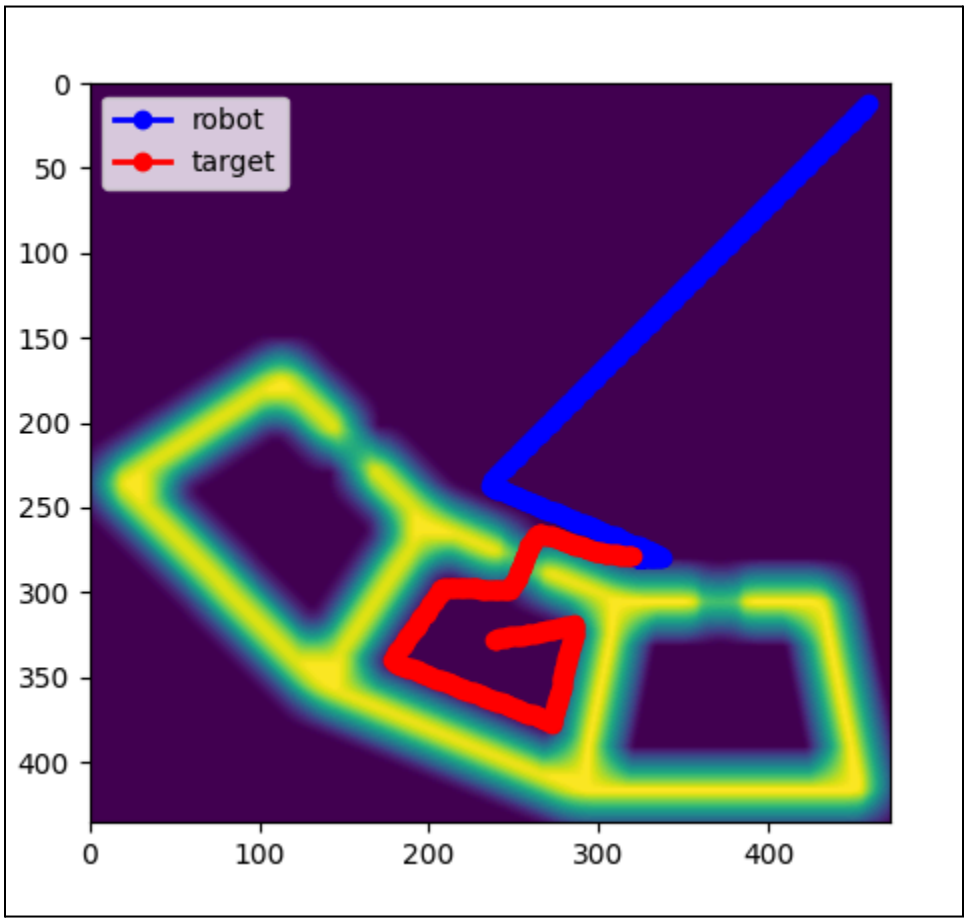
Map3.txt

target caught = 1
time taken (s) = 362
moves made = 359
path cost = 362



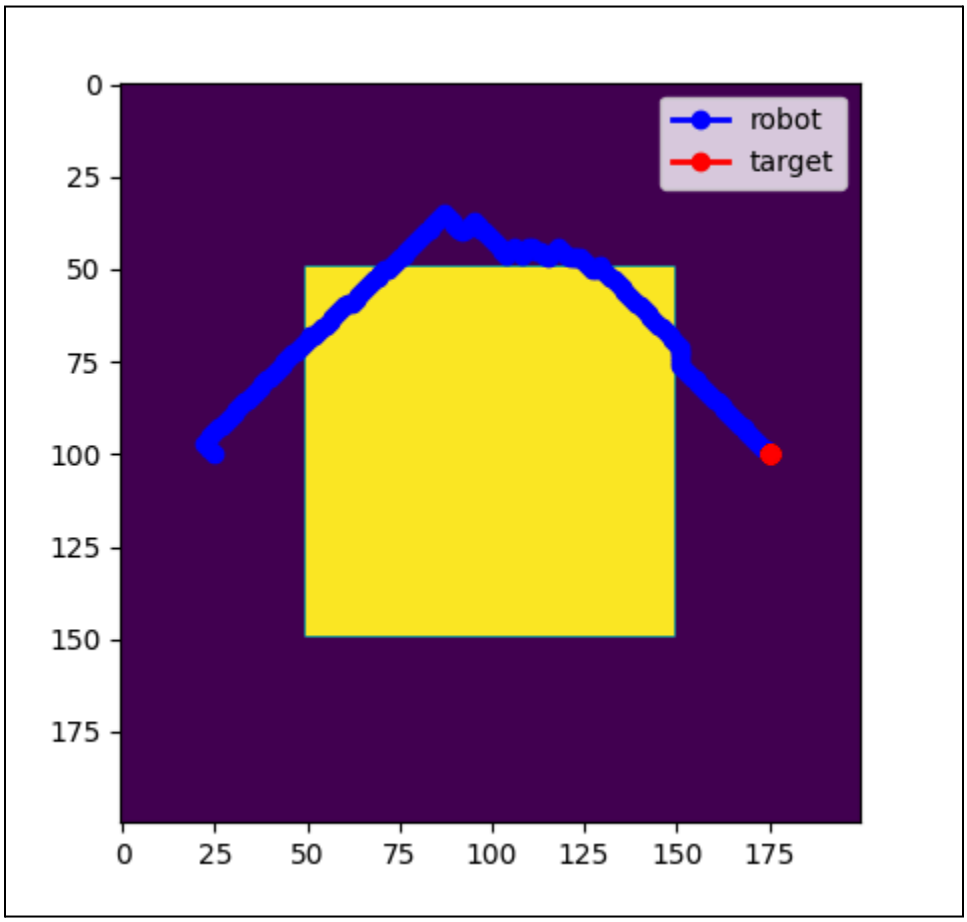
Map4.txt

target caught = 1
time taken (s) = 381
moves made = 378
path cost = 381



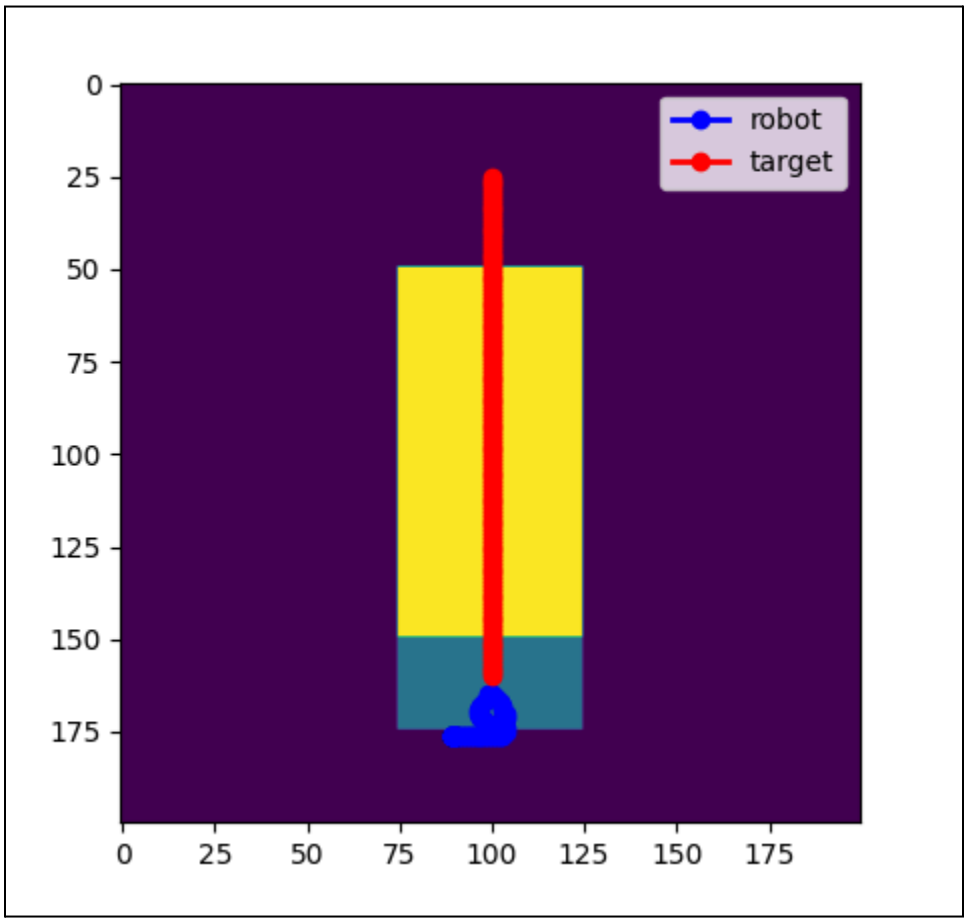
Map5.txt

target caught = 1
time taken (s) = 179
moves made = 161
path cost = 2139



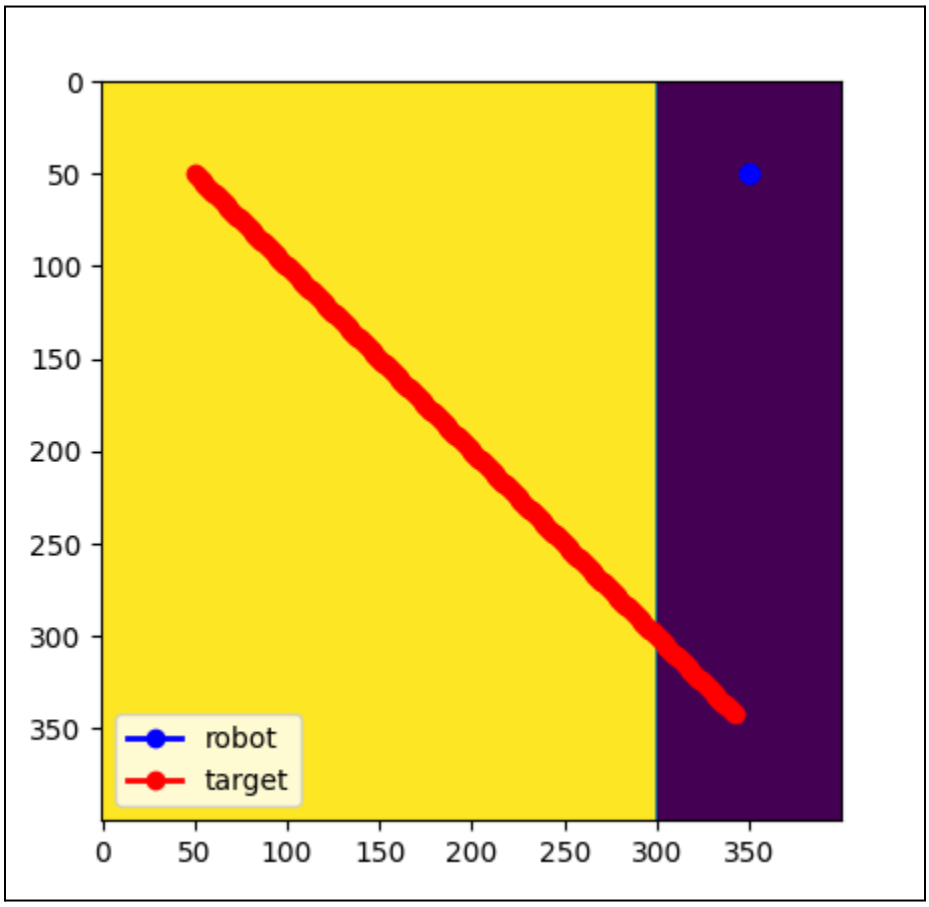
Map6.txt

target caught = 1
time taken (s) = 138
moves made = 137
path cost = 594



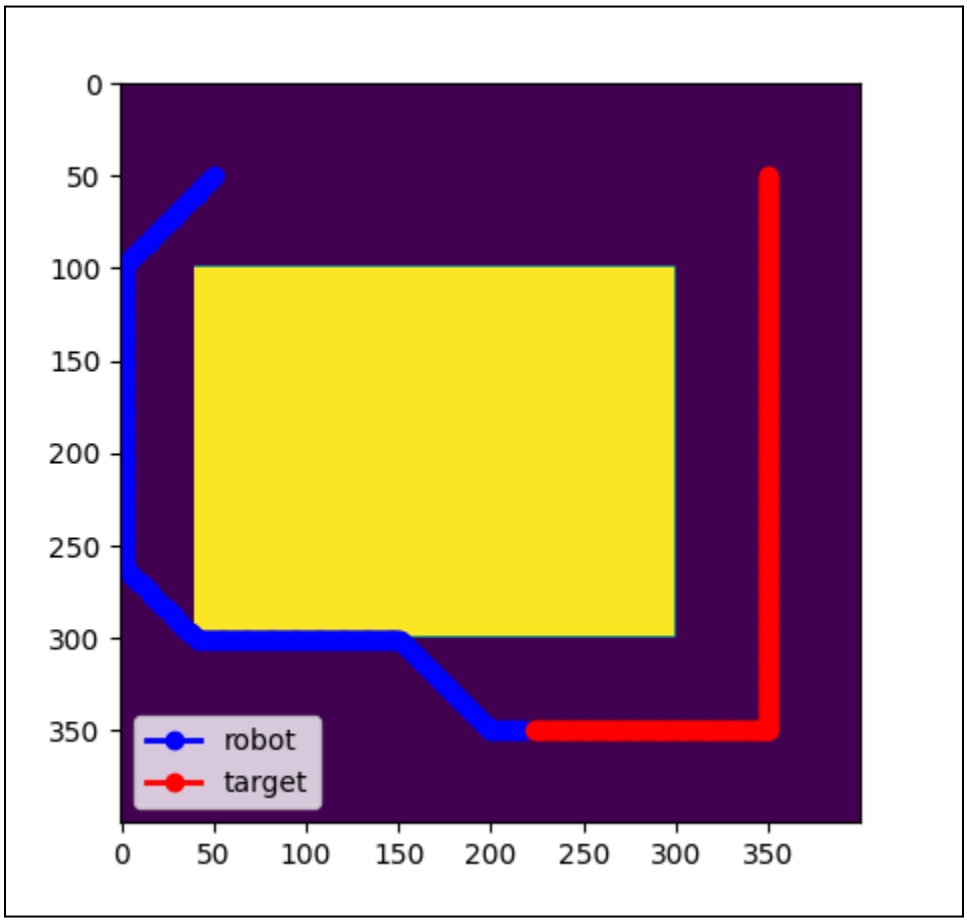
Map7.txt

target caught = 0
time taken (s) = 300
moves made = -1
path cost = 300



Map8.txt

target caught = 1
time taken (s) = 431
moves made = 430
path cost = 431



Map9.txt

target caught = 1
time taken (s) = 369
moves made = 369
path cost = 369