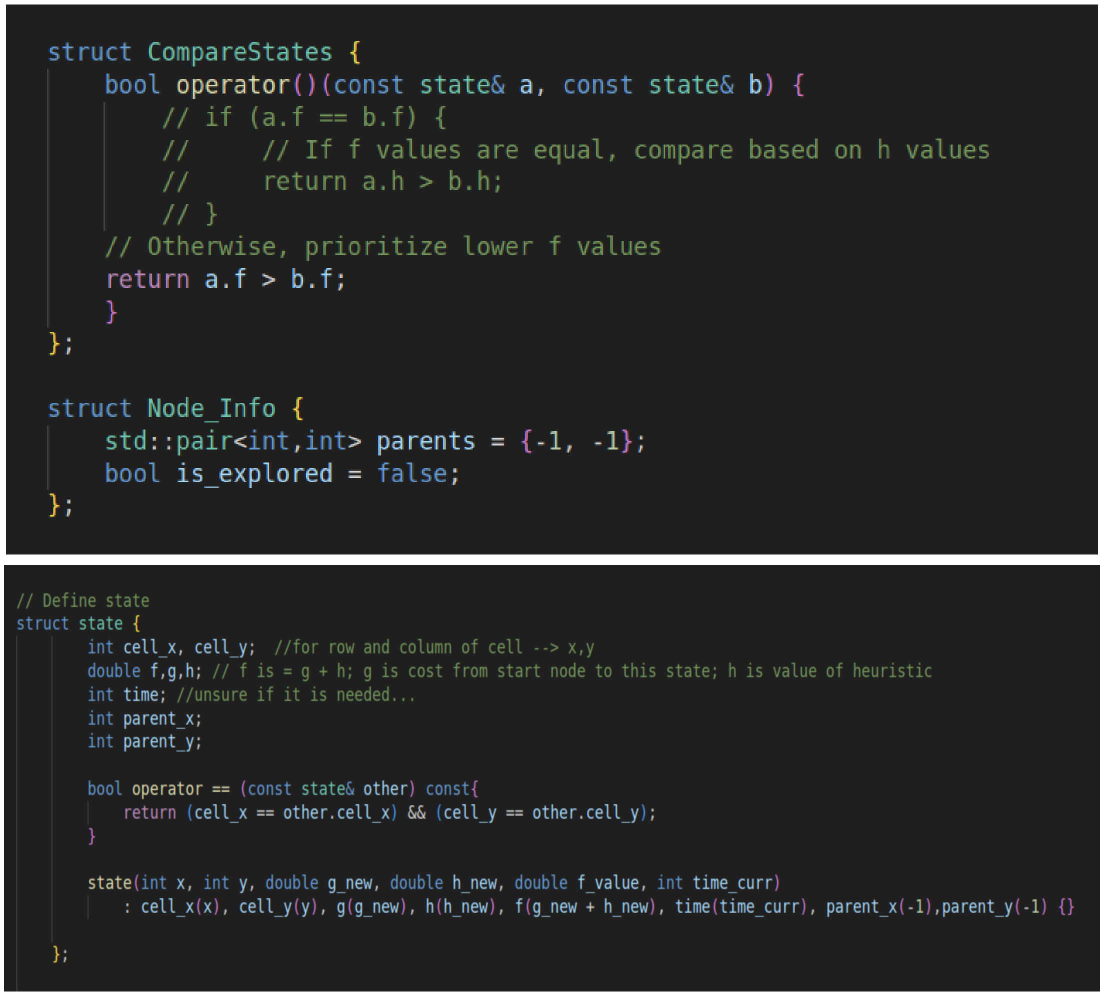
1 Summary of Approach

1.1 Data Structures Used

The algorithm that I implemented was an A* approach to find a path from a start state to a goal state in the provided 8-connected grid environment. This approach combines the advantages of Dijkstra's algorithm and also takes advantage of the greedy planner that was first implemented in the sample code to estimate the cost from the current state to the goal state. In the code itself I used the following data structures: **structs, vectors, unordered maps, pairs, adjacencyLists and priority queues**. The structs were used to **create a structure for the state** and represented a default state within the search space. I implemented it such that the values included were the cell coordinates (x,y), g-value (cost of traversing from one node to its successor node) h-value (heuristic estimate), f-value (g + h), time, and the parent state coordinates. In addition to this, I also added a bool operator to allow for comparison between states, this was useful in later sections of the code when trying to evaluate whether the start or goal location had been reached and the final section of the *'struct state'* included a declarator for the state itself. The second use for the struct was in the '*struct CompareStates'* which **initialized a comparator for the priority queue** (the open list) used in the implementation to compare the f-value of states. The last use was for the *'struct Node_Info'* this allowed me to store information about states that had been explored (*bool is_explored*) and the parent state coordinates (*pair of integers*), see Figure 1 for the structs defined in the code.



```cpp
struct CompareStates {
    bool operator()(const state& a, const state& b) {
        // if (a.f == b.f) {
        //     // If f values are equal, compare based on h values
        //     return a.h > b.h;
        // }
    // Otherwise, prioritize lower f values
    return a.f > b.f;
    }
};

struct Node_Info {
    std::pair<int,int> parents = {-1, -1};
    bool is_explored = false;
};
```

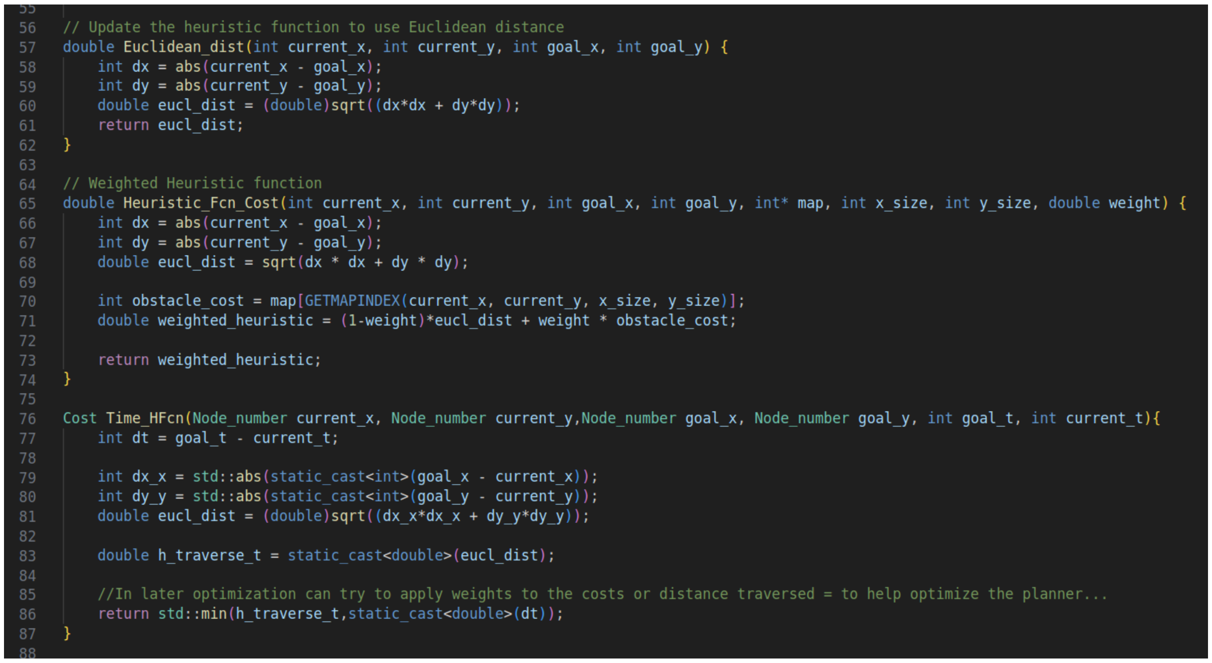
```cpp
// Define state
struct state {
        int cell_x, cell_y;  //for row and column of cell --> x,y
        double f,g,h; // f is = g + h; g is cost from start node to this state; h is value of heuristic
        int time; //unsure if it is needed...
        int parent_x;
        int parent_y;

        bool operator == (const state& other) const{
            return (cell_x == other.cell_x) && (cell_y == other.cell_y);
        }

        state(int x, int y, double g_new, double h_new, double f_value, int time_curr)
            : cell_x(x), cell_y(y), g(g_new), h(h_new), f(g_new + h_new), time(time_curr), parent_x(-1),parent_y(-1) {}

    };
```

Figure 1: Code with struct initializations

The next data structure used was the vectors which it's main purpose was to create iterable lists for the open and closed lists of the algorithm. However, I found very quickly that this was an inefficient way of implementing the open and closed lists as this increased the computation time and made the algorithm run a lot slower resulting in less targets being captured. As such I opted to create the closed_list as an unordered_map 'std::unordered_map<std::string, state*> closed_list_map', this mapped state identifiers to state pointers to allow for efficient checking of explored states. So, instead of storing state objects themselves inside of the closed list, we can provide a state's coordinates as a key identifier and a pointer to the state object as the value. This eliminates the need to search through an entire list trying to find a single state. Alternatively, for the open list I initialized it as a priority queue to explore the states more efficiently. Essentially what the priority queue does is it compares the f-values of the states (a.f > b.f) and places the states with lower f-values at the top of the queue while the states with larger f-values will be pushed to the bottom. Then once the value is popped from the priority queue it is subsequently added to the closed list (map). The vectors were used to store neighbors (after a state was expanded and before they were added to the priority queue) and to store the found path. The variable defined as an adjacency list was "merged" with the struct defaultInfo in that the adjacency list values were stored with the default structure of Node_Info. Though it is more accurate to say that instead of an adjacency list, a grid of x_size, y_size was initialized to store data. This provided an efficient way to ensure that all the parents were kept track of, so that later when the path is reconstructed (from the goal to the start location) there are no errors. Finally, the path_found was stored as a pair of integers from the start state to the goal state and passed to the action_ptr's to execute the actions to reach the goal state.

1.2 Heuristics

The heuristic function was perhaps one of the most difficult parts of the implementation for me. I initially started with using the euclidean distance as the heuristic function, just to estimate the euclidean distance from a state to the goal state. This method proved to be quite efficient but upon inspection was not working for all the maps. Though this is an admissible heuristic function, in maps 5 and 7 in particular I was consistently unable to get my code to work. This prompted me to delve further into other heuristic functions (see Figure 2 below).



```cpp
// Update the heuristic function to use Euclidean distance
double Euclidean_dist(int current_x, int current_y, int goal_x, int goal_y) {
    int dx = abs(current_x - goal_x);
    int dy = abs(current_y - goal_y);
    double eucl_dist = (double)sqrt((dx*dx + dy*dy));
    return eucl_dist;
}

// Weighted Heuristic function
double Heuristic_Fcn_Cost(int current_x, int current_y, int goal_x, int goal_y, int* map, int x_size, int y_size, double weight) {
    int dx = abs(current_x - goal_x);
    int dy = abs(current_y - goal_y);
    double eucl_dist = sqrt(dx * dx + dy * dy);

    int obstacle_cost = map[GETMAPINDEX(current_x, current_y, x_size, y_size)];
    double weighted_heuristic = (1-weight)*eucl_dist + weight * obstacle_cost;

    return weighted_heuristic;
}

Cost Time_HFcn(Node_number current_x, Node_number current_y,Node_number goal_x, Node_number goal_y, int goal_t, int current_t){
    int dt = goal_t - current_t;

    int dx_x = std::abs(static_cast<int>(goal_x - current_x));
    int dy_y = std::abs(static_cast<int>(goal_y - current_y));
    double eucl_dist = (double)sqrt((dx_x*dx_x + dy_y*dy_y));

    double h_traverse_t = static_cast<double>(eucl_dist);

    //In later optimization can try to apply weights to the costs or distance traversed = to help optimize the planner...
    return std::min(h_traverse_t,static_cast<double>(dt));
}
```

Figure 2: Heuristic Function testing

I experimented with two other heuristic functions, one taking into consideration the time it would take to traverse certain paths and the other considering the obstacles along the path in a weighted heuristic approach. The weighted heuristic combines the euclidean distance with a weighted obstacle cost, and the goal here was to try a more informative approach in that I thought a better estimate of the cost to reach the goal could be leveraged by considering both distance and obstacle factors. The time-based heuristic function was used to take into account the time remaining to reach the goal, this was an admissible heuristic as well since the cost of traversing a unit of time is uniform throughout and outline a relationship between the time component and the spatial distance, so I was able to treat the time the same as other dimensions. With this in mind, I tested all the heuristic functions and found that I was able to catch the target most reliably when using the **Time-based** heuristic function as one of the main factors of the problem is being able to catch the target within the time range. For cases like map 5 and 7 this was key in improving the success rates though ultimately in the end I was still not able to catch the target on map 7. In summary, the heuristic function chosen was highly dependent on the time aspect of the problem and its role in finding an optimal solution.

1.3 Efficiency Tricks and Memory Management Details

Some of the efficiency tricks have been mentioned in previous sections but to reiterate using an unordered map as the closed list's data structure helps to efficiently check if a state has already been explored, by mapping state identifiers to state pointers. Additionally, the planner has a time limit in seconds, and if this limit is exceeded the search will be terminated. The priority queue being used for the open list helps to manage the states and ensure that the entire open list does not have to be searched through to find the state with the minimum f-values thus ensuring that states with lower estimated costs are explored first. To avoid states being "re-explored" and the same neighbors being added to the open list, states are marked as explored to avoid re-exploration. Lastly, my algorithm traces the paths from the goal state back to the start state using the parent pointers stored in each state. To help manage the memory states are stored in data structures that help minimize the storage of variables and helps in updating/reusing existing state objects. My implementation also sets a '*planned_path*' flag to true once a path is successfully generated. This was used to ensure that a path is generated only once.

1.4 Areas for Improvement

While the algorithm works very reliably and is consistent there are still some areas that I feel could be improved to help improve the costs of the path. One is in the algorithm itself, my initial plan was to implement the multi-goal A* approach, to be able to catch the target at different goal points along the trajectory. However, given my limited understanding of the A* approach when starting the homework I decided to first start by implementing the A* approach and optimize it as best as I could before trying to implement the multi-goal A* approach. Another improvement would be switching to a 3D implicit graph instead of just a 2D implementation. This would have allowed for more realistic and accurate path planning in a 3D environment, and this could have been taken into account throughout the code (i.e. inside of data structures, and also within the heuristic functions as well), while it has been addressed in some areas it is not consistent throughout. Lastly, there is one bug in my code that I cannot figure out as a path is found and planning is finished but instead of breaking the loop and not re-planning, as I have defined a global static variable. The variable will be redefined to false every time the planner runs and re-enters the planning loop no matter if the planning is finished or not.

2 Results

2.1 Map 1

Table 1: Map 1

Questions	**Answers**
Robot Caught?	True
Time taken (s) to run the test?	5,344
Number of moves made	421
Path Cost	5,344

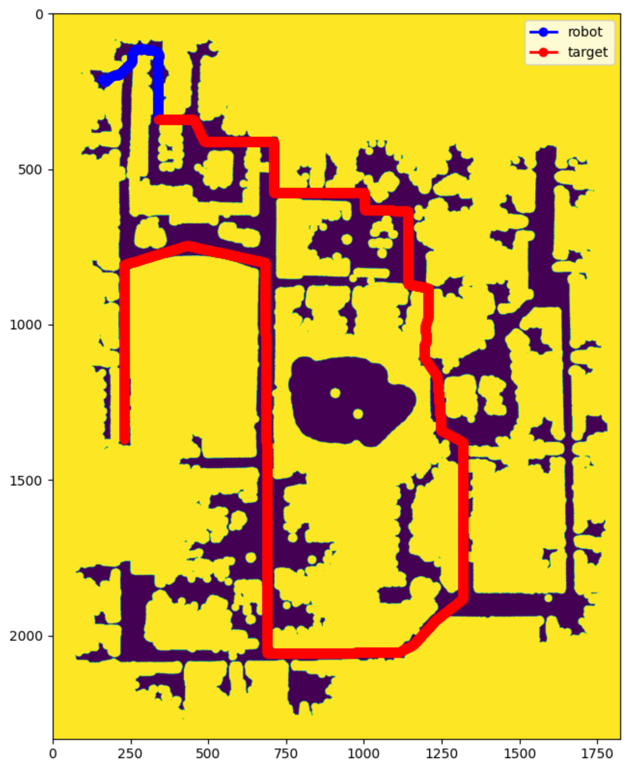


Figure 1: Plot of Map 1

2.2 Map 2

Table 2: Map 2

Questions	Answers
Robot Caught?	True
Time taken (s) to run the test?	5,344
Number of moves made	421
Path Cost	5,344

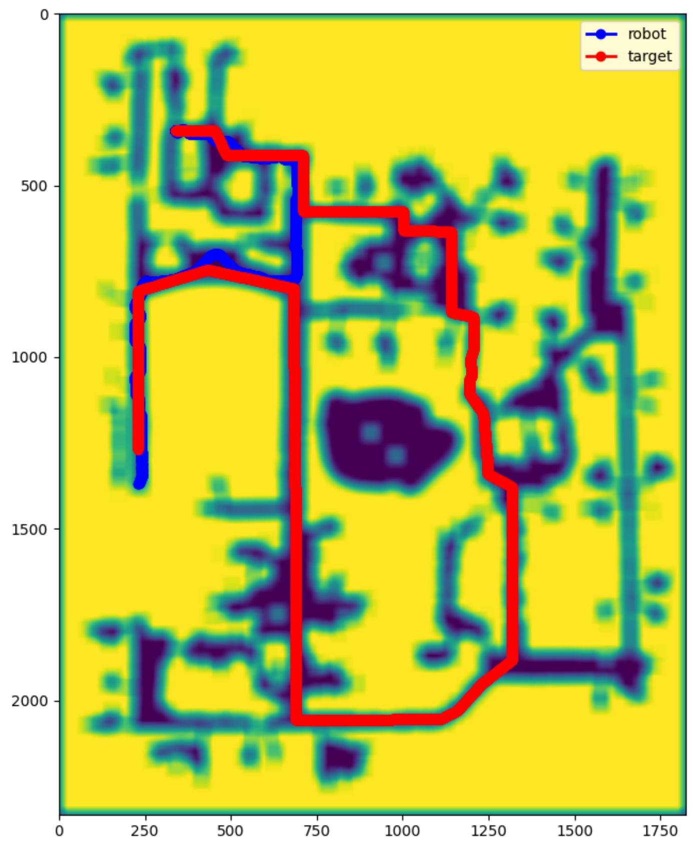


Figure 2: Plot of Map 2

2.3 Map 3

Table 3: Map 3

Questions	**Answers**
Robot Caught?	True
Time taken (s) to run the test?	791
Number of moves made	283
Path Cost	791

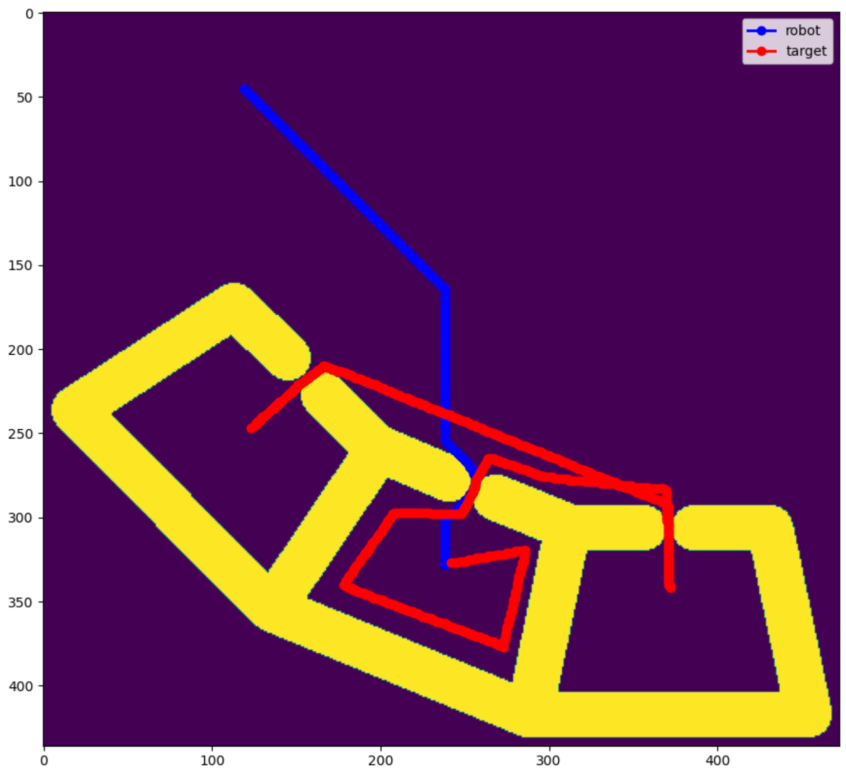


Figure 3: Plot of Map 3

2.4 Map 4

Table 4: Map 4

Questions	**Answers**
Robot Caught?	True
Time taken (s) to run the test?	791
Number of moves made	336
Path Cost	70, 271

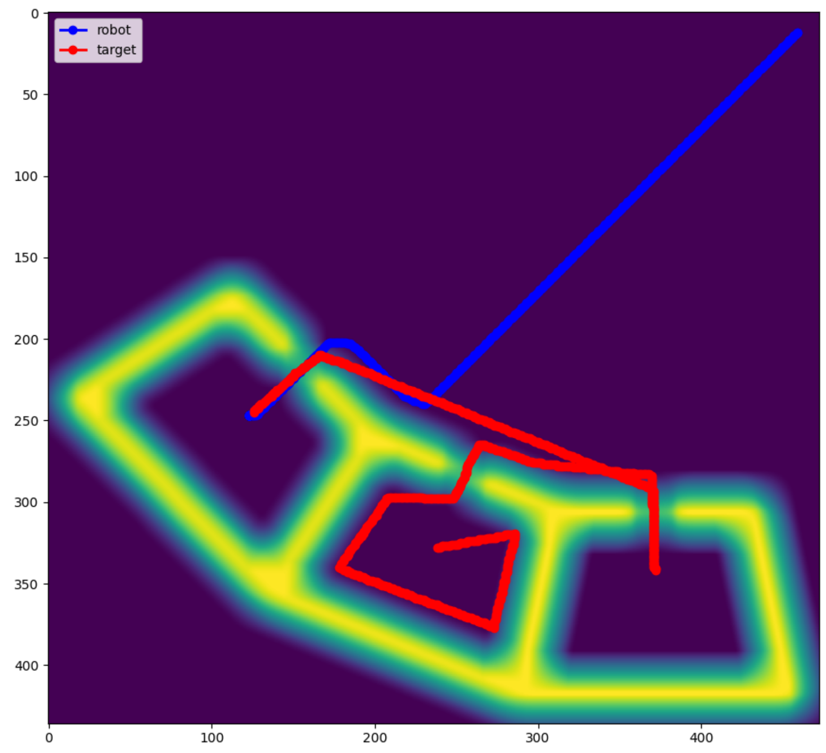


Figure 4: Plot of Map 4

2.5 Map 5

Table 5: Map 5

Questions	**Answers**
Robot Caught?	True
Time taken (s) to run the test?	150
Number of moves made	150
Path Cost	5,050

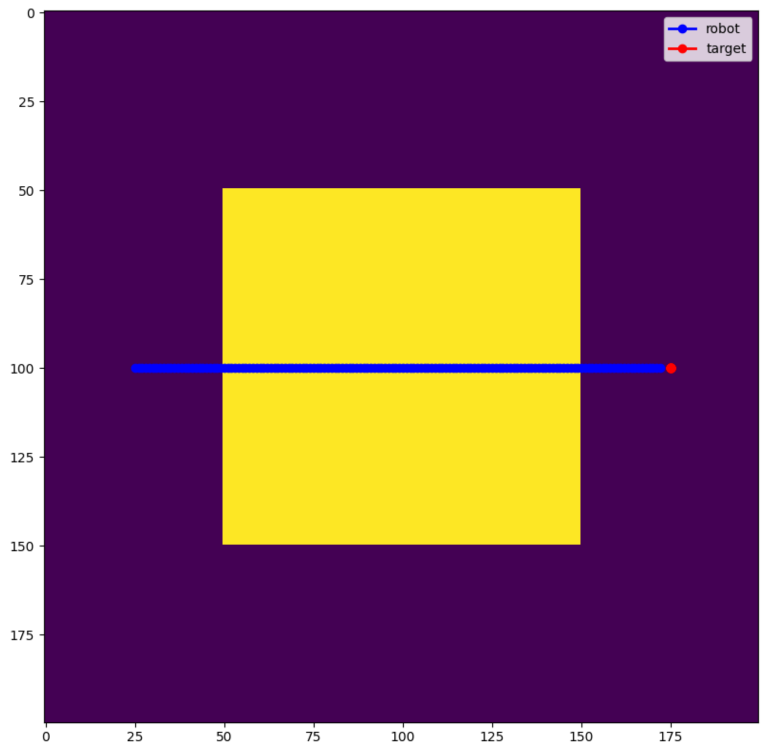


Figure 5: Plot of Map 5

2.6 Map 6

Table 6: Map 6

Questions	Answers
Robot Caught?	True
Time taken (s) to run the test?	140
Number of moves made	0
Path Cost	2,800

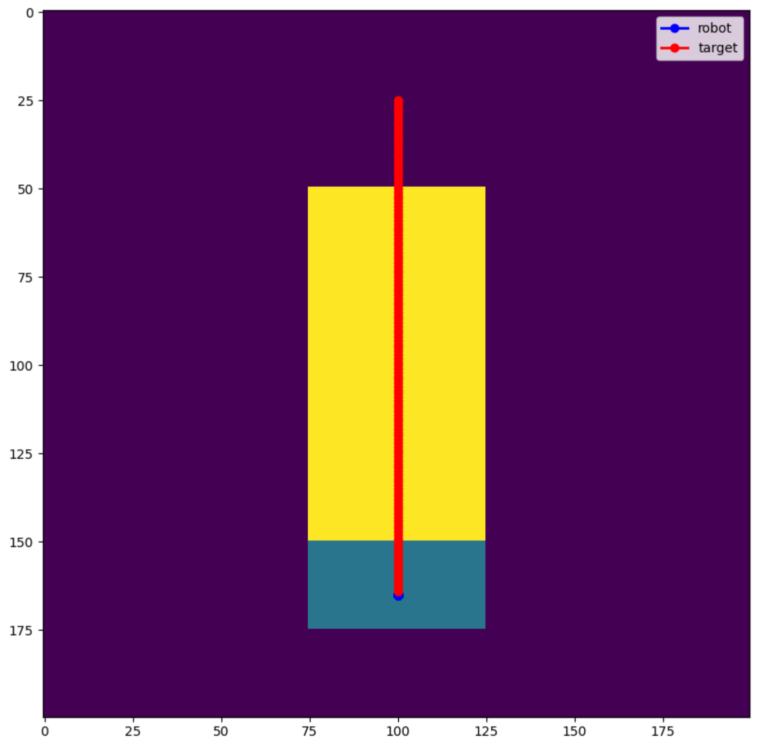


Figure 6: Plot of Map 6

2.7 Map 7

Table 7: Map 7

Questions	**Answers**
Robot Caught?	False
Time taken (s) to run the test?	300
Number of moves made	299
Path Cost	300

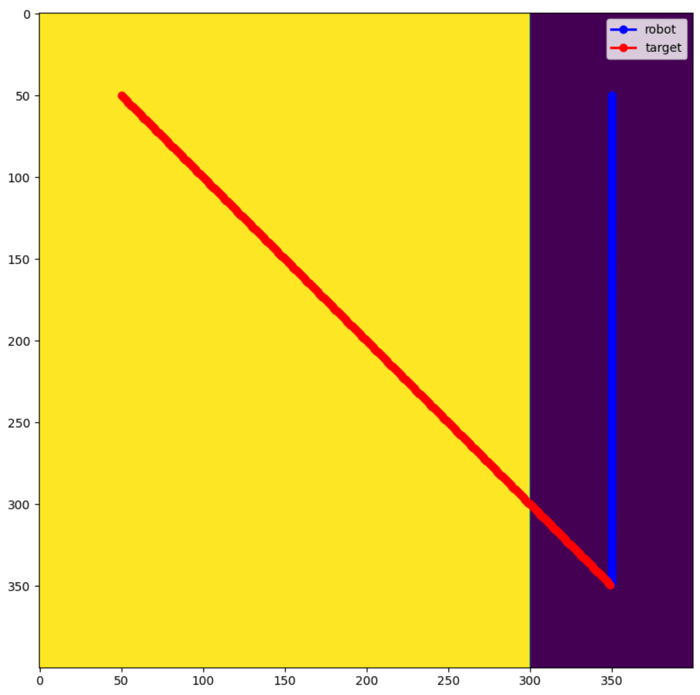


Figure 7: Plot of Map 7

This map was the only map that my planner was unable to solve, despite taking into consideration the time. The planner seems to be just one timestep behind the target, another mention here is that instead of breaking out of the end of the loop the program seems to continue running, which could be causing additional loops where it is not actually necessary.

2.8 Map 8

Table 8: Map 8

Questions	Answers
Robot Caught?	True
Time taken (s) to run the test?	451
Number of moves made	410
Path Cost	451

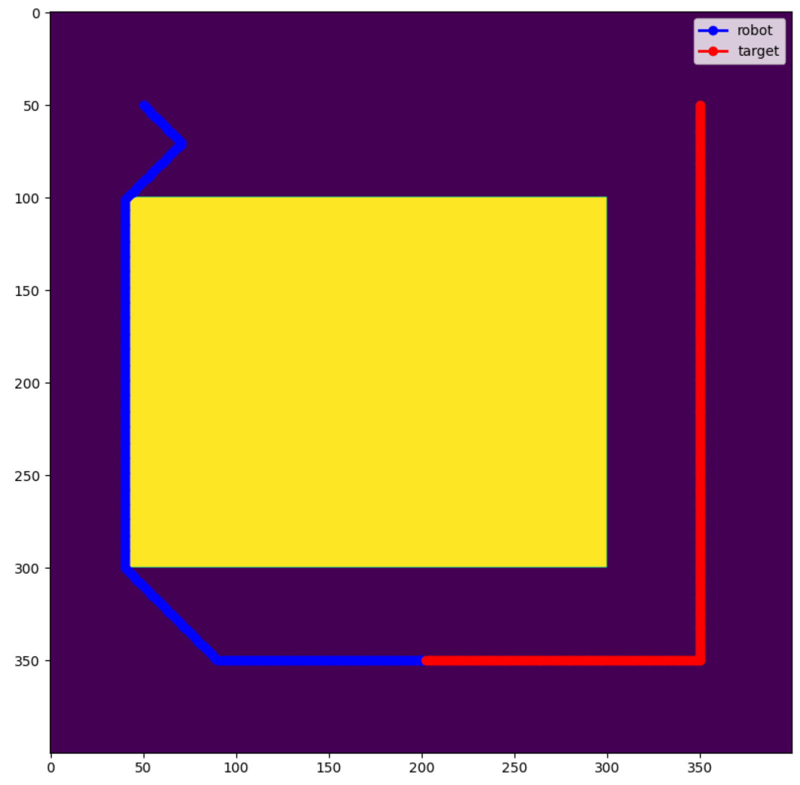


Figure 8: Plot of Map 8

2.9 Map 9

Table 9: Map 9

Questions	Answers
Robot Caught?	True
Time taken (s) to run the test?	600
Number of moves made	350
Path Cost	600

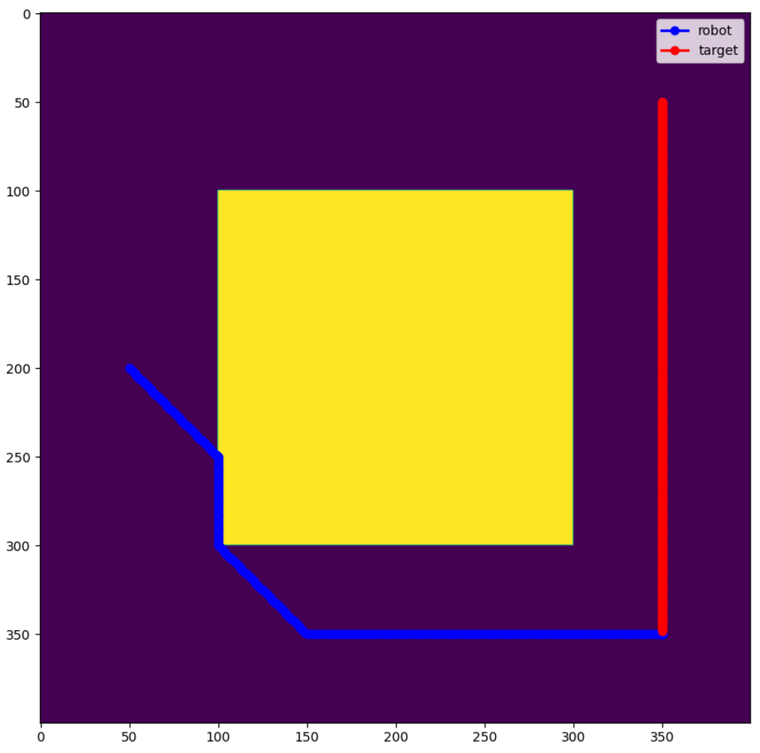


Figure 9: Plot of Map 9

3 Compiling Code

To compile the code simple run the commands provided in the homework prompt:

1. g++ runtest.cpp planner.cpp
2. ./a.out map3.txt
3. python visualizer.py map3.txt