1 Command to Run

```
mex planner.cpp
runtest("map.txt")
```

Recompile the code before every run to get correct results.

2 Overview

Initially, I thought of performing a reverse A* search in 2D (x,y) considering the all the trajectory points as goal states. Then performing a forward A* search in 3D (x,y,t) trying to meet the target at the trajectory position exactly at the same time. However, since the trajectory is so spread out the Heuristic calculated is not directing the search towards a certain region but spreads it out. This causes the search to go on for a very long time (even to minutes sometimes). In order to reduce the search time I had to come up with several ideas to prune the search and also boil down possible trajectory points from the given trajectory. This way we can focus the 3D search and allow it to terminate much faster.

The general approach of the solution is-

1. **Forward A* Search:** Perform a forward A* search in 2D till all the trajectory points are visited. This is done to identify the minimum steps to reach each point in the trajectory and the cost incurred to take those paths.
2. **Selection of Terminal States:** Based on the results of the forward A* search, remove unreachable trajectory points and select a window of reachable trajectory points.
3. **Reverse A* Search:** Now perform a reverse A* search in 2D to get the minimum cost of reaching these selected terminal trajectory points from the rest of the map.
4. **Perform 3D A* Search:** Perform a 3D A* search in (x,y,time) to find a minimum cost path to reach one of these terminal trajectory points and extract the trajectory from this search.

Overall, the third map and the fourth run almost instantaneously and the first and second map take some time to plan, due to the complexity of the map.

3 Detailed Explanation of Approach

In this section I will go through the explanation of the structure of the code and the design decisions made in order to achieve the results that I got. I will also explain the optimisations made.

3.1 Forward A* (Dijkstra's) Search

Forward A* search is performed from the goal state of the robot till we have visited all the trajectory points. Along the way we compute both the minimum number of steps and the cost incurred in taking minimum of number of steps to reach the points on the trajectory. This information can now be used to identify spots in the trajectory which can be reached.

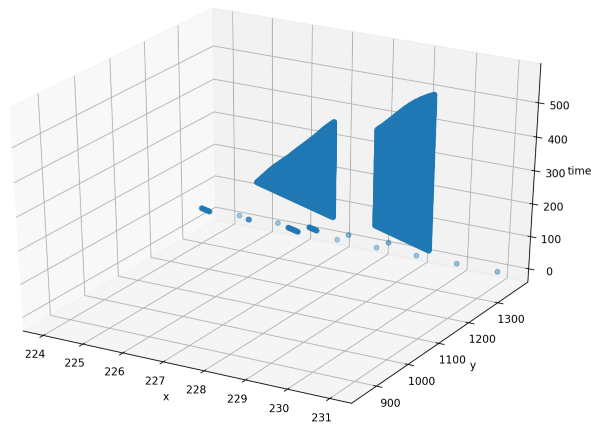
3.2 Selection of Terminal States

This is an important step because when the Reverse A* search is run taking all the goal states in the open list with g-values as 0, we get a heuristic which does not focus the search towards a particular goal. When such a heuristic is used for a 3D search the search horizon is wide and the search expands a lot of nodes before reaching a trajectory point in the 3D space. This takes a lot of time (in the order of minutes sometimes) for complex maps.

In order to solve this problem, we need to focus our search. An optimal window of trajectory points can be identified by taking the following steps-

1. Remove all points of the trajectory which cannot be reached by the robot before the trajectory.
2. Calculate a upper bound cost for the rest of the trajectory points. The upper bound cost is defined as summing up the cost to reach a trajectory point with minimum steps and the cost to stay there until the robot arrives.
3. Identify the point with lowest upper bound cost.
4. Select a threshold size. In my case I selected it to be as 1e-6 times the total number of states in the graph. This is used as a cushioning to account for planning latencies.
5. Add all terminal states within a fixed window (50 used in implementation) about the point of lowest upper bound cost which can be reached before the threshold number of time steps into a set. In case no points were found which satisfies this threshold, halve the threshold and repeat from step 2.

The set with these selected states will be used to calculate our heuristic and we will attempt to form an optimal cost trajectory to these points in the 3D A* search. However, the assumption that an optimal path to catch the trajectory would lie in a window around the point with the lowest upper bound cost, may not be true. This assumption is required to prune the 3D search to at least ensure of completeness of my implementation.



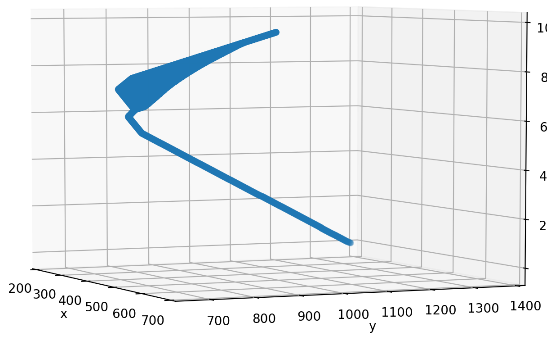


Figure 1: (a) 3D A* search when all trajectory points are used to calculate the heuristic. (b) 3D A* search when a window of trajectory points is used to calculate the heuristic

Here we can see that by giving fewer terminal states for the heuristic the search becomes much more focused.

3.3 Reverse A* Search

All the selected terminal states are pushed into an open list and the minimum cost to get to all reachable points in the rest of the map from these states is calculated. This can be done with a reverse A* search in 2D(x,y). This will act as informed heuristic for the next stage of planning. This search takes about a second in the most complex map. It takes a few hundred milliseconds in the smaller maps.

3.4 3D A* Search

A 3D search is performed in x,y and time dimensions to get a trajectory from current robot position to one of the selected terminal states. Weighted A* search is implemented with weight of 50. Additionally certain measures are taken to not waste time expanding unnecessary nodes. Some of the measures taken were-

1. Do not add neighbors with cost greater than the collision threshold
2. If the minimum number of steps needed (assuming obstacle free), to reach the farthest selected terminal node from the current node position, is more time than the steps

needed for the target to reach that terminal node then, there is no point expanding further from this point on wards. This condition can be expressed in code as-

```
/* max_time_state is the vector <x,y> coordinate of the furthest
terminal state and curr is the vector <x,y,t> representing the current
expanded state in the 3D* search and max_time is the time taken to
reach the farthest state by the target */

int min_time_req = max(abs(max_time_state[0]-curr[0]),
                       abs(max_time_state[1]-curr[1]));
int time_diff = max_time-curr[2]-thresh;
bool condition = time_diff<min_time_req;
```

4 Code Structure

Initially the unordered_map STL container was used to store the structs of states with a vector of ⟨ x,y ⟩ or ⟨ x,y,t ⟩ as the key to perform A* search. However these implementations turned out to be inefficient. A trade-off between performance and memory had to be made. If an unordered_map is used performance reduces but only those states which were discovered can be stored and hence are memory efficcient. Whereas using 2D arrays means it is memory inefficient but the speed increases. Therefore 2D arrays were used for performing 2D A* searches and 3D A* search was implemented with unordered_maps.

4.1 Data structures used in 2D search

This is the data structure used in 2D searches to store the visited and g values.

```
struct forward_heuristic_map{
    vector<bool> visited;
    vector<int> count;
    vector<double> cost;
    forward_heuristic_map(){
        visited.resize((x_len+1)*(y_len+1), false);
        count.resize((x_len+1)*(y_len+1), INT32_MAX);
        cost.resize((x_len+1)*(y_len+1), __DBL_MAX__);
    }
};
```

Additionally, a min-heap implemented from a priority_queue STL container is used for the open list.

```
class Comparison{
    public:
    bool operator () (pair<double,vector<int>>& a, pair<double, vector<int>>& b){
        return a.first > b.first;
    }
};
priority_queue<pair<double,vector<int>>,vector<pair<double,vector<int>>>,Comparison>
                                                                      open_list;
```

4.2 Data structures used in 3D search

Since the number of states explode in 3D search, it is inefficient to explicitly create a 3D array to store visited, g-values, backpointers etc for each state. Therefore this in search an unordered_map was used to store the information of each state expanded.

```
class Hash{
    public:
        size_t operator()(const vector<int>& p) const
        {
            return p[1]*x_len*total_time+p[0]*total_time+p[2];
        }
};
struct Node{
    public:
        double g,h;
        bool visited;
        vector<int> parent;
        Node() : g(MAX_VAL), h(-1), parent({-1,-1}), visited(false){}
};
unordered_map<vector<int>,Node, Hash> Graph;
priority_queue<pair<double,vector<int>>,vector<pair<double,vector<int>>>,Comparison>
                                                                      open_list;
```

5 Results

Maps	Time(ms) taken to perform forward A*	Time(ms) taken to perform reverse A*	Total time(ms) taken
map1.txt	660	633	1390
map2.txt	1187	1345	2704
map3.txt	122	143	271
map4.txt	107	153	269

5.1 Map-1

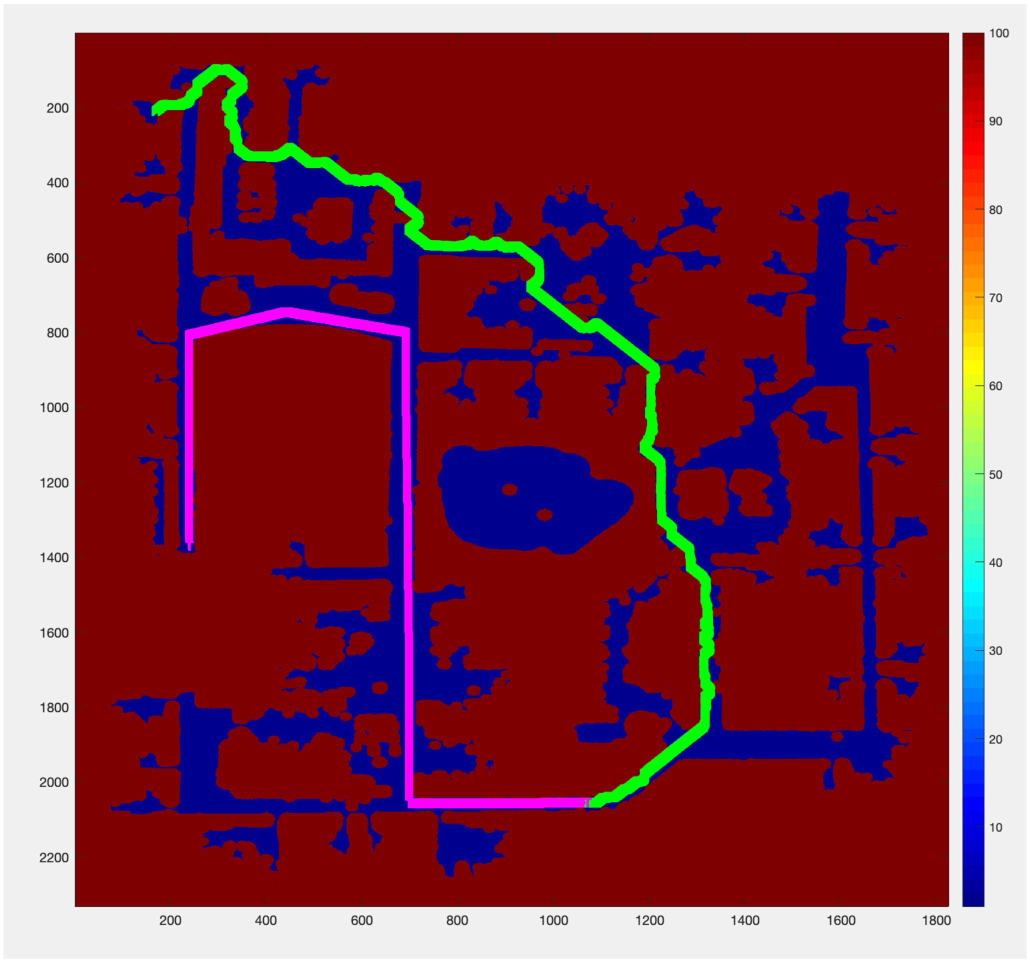


Figure 2: map1.txt

Target caught	1
Time taken(s)	2649
Moves made	2631
Path cost	2649

5.2 Map-2

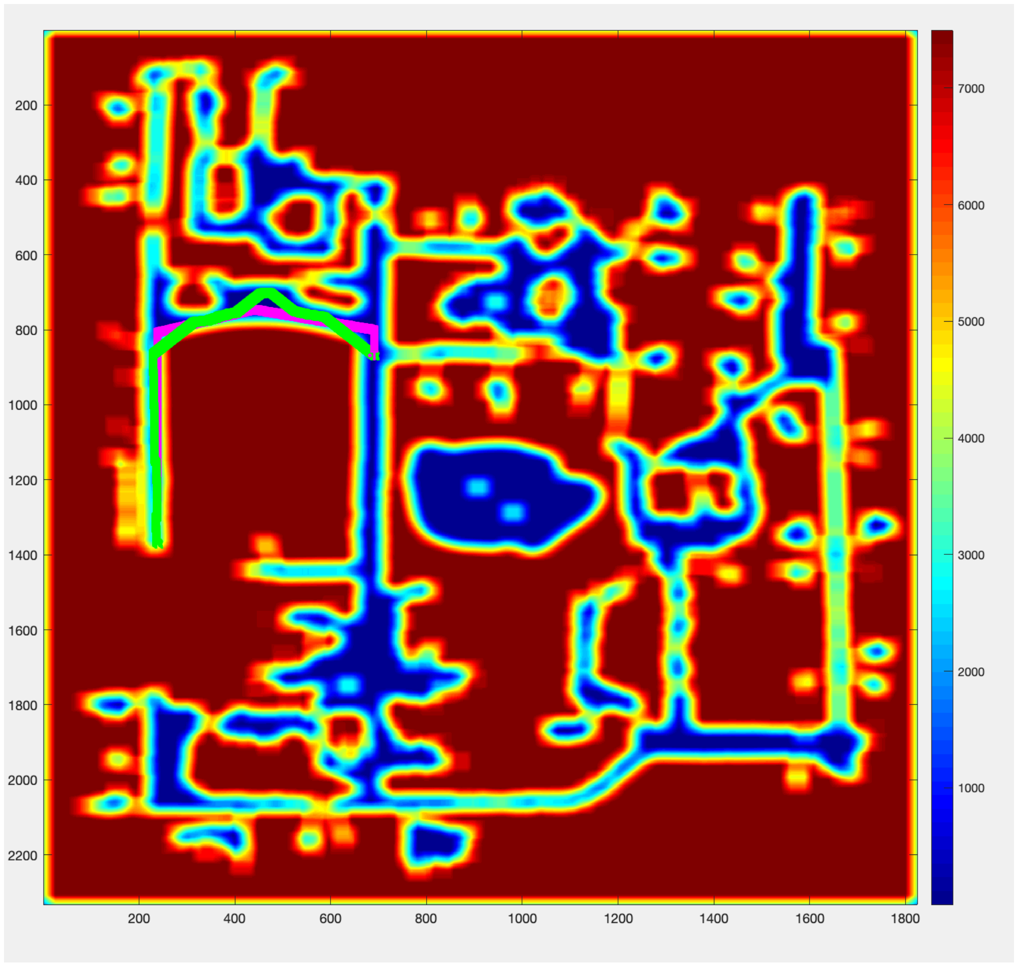


Figure 3: map2.txt

Target caught	1
Time taken(s)	987
Moves made	969
Path cost	1999900

5.3 Map-3

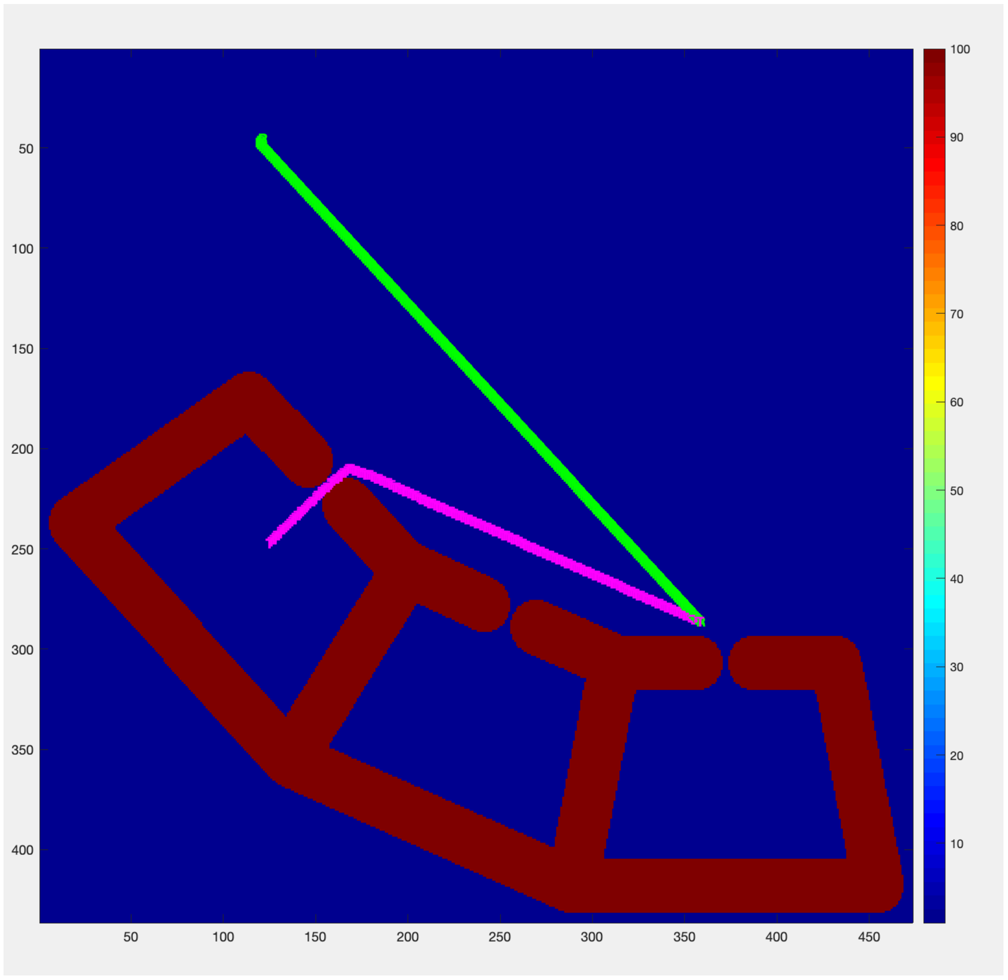


Figure 4: map3.txt

Target caught	1
Time taken(s)	242
Moves made	241
Path cost	242

5.4 Map-4

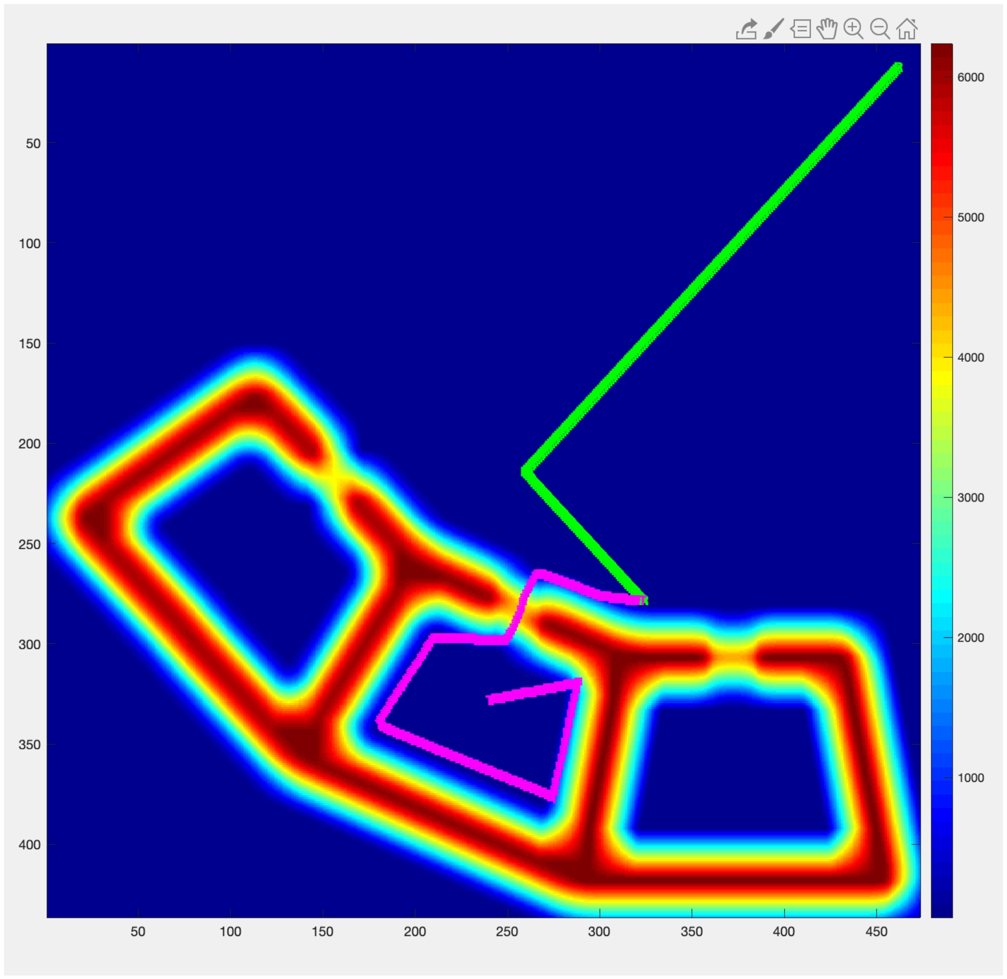


Figure 5: map4.txt

Target caught	1
Time taken(s)	380
Moves made	266
Path cost	380

6 Conclusion

There are some strong assumptions made when calculating optimal paths. Assumption that the optimal path exists in the region of the lowest upper bound as defined in 3.4 may not always hold. Additionally a threshold is defined before which the robot must reach the trajectory point, in order to account for the latency of the code in case of complex cases. This threshold may need to be recomputed if no candidate trajectory points are found.

Nevertheless, the implementation can find an optimal path in a 3D graph after selecting a narrow window of goal states. While not completely optimal it ensures that it is at least sub-optimal after selecting a narrow target window of goal states. Additionally, it also ensures completeness for all kinds of graphs. The method while works well for the given maps, and can be shown to work for a wide variety of such maps, still leaves room for improvement.

For future work, it would be great to further optimise the 2D searches and make the whole heuristic computation segment run under a second. A better provable technique to identify possible trajectory parts and focus the 3D search can greatly improve the implementation.