Motion Planning
: A search-based motion planning algorithm to intercept a moving target

Orish Jindal
Department of Electrical Computer Engineering
University of California, San Diego
ojindal@ucsd.edu

Abstract – This paper discusses a search-based planning algorithm to solve the problem of intercepting a moving target by feeding the shortest path to the robot. The maps are provided in form of 2D matrices with known initial positions of robot and target along with the obstacle locations. Results shows the success of current algorithm with plots of the paths tracked by both: robot and the target.

Keywords - Label Correcting, Dijkstra, A*, Heuristics, Search-based planning, Agent-centered search, State Space, Control Input, Planning Horizon, discrete space.

I. INTRODUCTION

Path planning is the first step for navigation of an Autonomous vehicle. This is done by first deciding the states that a robot needs to visit throughout the whole motion and then ossifying the optimal set of control inputs which leads our robot to the respective states. One of the many ways to do it is by using any of these popular dynamic programming algorithms such as Label Correcting, Dijkstra, A* etc. These algorithms only work in an ideal environment where the obstacles and goal position does not change with time and are known to the robot prior to starting the path planning. But the real world does not work that way where the locations of obstacles and goal may change or unknown to the robot before beginning the journey.

In this paper, we will tackle a problem that is somewhat similar to some real-world path planning problems. In this project, the aim is to dynamically plan a path for the robot on random environments in order to intercept a target which is changing its location with time. The environments are represented as 2D grid maps and hence the state space is discrete. As we can decipher, we must construct an algorithm that is able to plan the path and make the robot move one step before the target moves one step. We will discuss one of feasible algorithms for this problem, based on **Agent-centered Search** method which acts as one of the variations of the good old A* algorithm, to intercept the moving target in finite time.

These methods have many real-world applications such as game development, missile technology, autonomous vehicles, drones etc. The techniques used in those applications uses the advanced version of the solutions that we will discuss in this paper.

II. PROBLEM FORMULATION

In the ideal case of a fixed goal location and fixed obstacle locations that are known to the robot prior to starting the journey can be efficiently solved by using one of the many dynamic programming algorithms such as Vanilla Label Correcting Algorithm, to avoid the most basic approach of calculating the value function for all possible combinations of the states in state space.

It still doesn't solve the complete practical problem as in the large environments with changing goals, this algorithm if run repeatedly: recalculates the whole path each time after taking one step towards the goal to incorporate for changing goals. It is clearly inefficient as many unnecessary repetitions in calculations can be avoided if we use a better way to somehow reuse some portion of the already calculated path and reconsider calculating the path only in the vicinity of the goal to incorporate changes in the goal location.

In the problem that is discussed in this paper, we are provided with some random 2D grids (2D matrices) with known obstacle locations and the initial robot and target positions. The obstacles hold the value = 1 and the free space holds the value = 0 in the matrix representation of maps. The target (goal) makes one move in 't' seconds, but it can only move in four directions (no diagonal moves) whereas the robot can move in one of the of 8 directions in one step. Since this is a deterministic case

problem where we have to catch the target with our robot, we can formulate it as a Deterministic Shortest Path (DSP) problem.

Since our target is moving with time, we have to implement the DSP multiple times back-to-back until we intercept the target with our robot. One DSP will lead the robot to a certain path keeping in mind the instantaneous target position. The next DSP will pick from there and lead the robot further till the termination criteria is met (defined below).

- Current position of the robot, R = (R_x, R_y)
- Current position of the target, T = (T_x, T_y)

Termination criteria:

- $|R_x - T_x| <= 1$ and $|R_y - T_y| <= 1$.

The elements of a single DSP for this problem are as follows:

- Considering a graph with the finite vertex set ν,

  ν := 2D coordinates of each block in the environment matrix as nodes.
- Edge set $\mathcal{E} \subseteq \nu \times \nu$,
- Edge weights C := $\{c_{ij} \in \mathbb{R} \cup \{\infty\} \mid (i, j) \in \mathcal{E}\}$, where c_{ij} denotes the arc length or cost from vertex *i* to vertex *j*.

  In our case, the robot is allowed to make only one move in any direction (one of 8 directions) so the edge weights for adjacent nodes will be as follows:

$$c_{ij} = \begin{cases} 0, & if\ i = j \\ 1, & if\ Euclidean\ dist.(i,j) = 1\ and\ j \notin obstacle \\ \sqrt{2}, & if\ Euclidean\ dist.(i,j) = \sqrt{2}\ and\ j \notin obstacle \\ \infty, & \text{if j is out of bounds or represents an obstacle} \end{cases}$$

- Path: a sequence $i_{1:q} := (i_1, i_2, \ldots, i_q)$ of nodes $i_k \in \nu$.
- All paths from s ∈ ν to τ ∈ ν: $P_{s,\tau} := \{i_{1:q} \mid i_k \in \nu,\ i_1 = s,\ i_q = \tau\}$.
- Path length: sum of edge weights along the path: $J^{i_{1:q}} = \Sigma_{k=1}^{q-1} c_{i_k, i_{k+1}}$
- Assumption: There are no negative cycles in the graph, i.e., $J^{i_{1:q}} \geq 0$, for all $i_{1:q} \in P_{i,i}$ and all i ∈ ν.
- Objective: To find a path that has the min length from node s to node τ:

$$\text{dist.}(s, \tau) = \min_{i_{1,q} \in P_{s,\tau}} J^{i_{1:q}},$$
$$i^*_{1:q} = arg \min_{i_{1,q} \in P_{s,\tau}} J^{i_{1:q}}$$

In other words, our objective is to find the shortest path from a start node s to an end node τ.

Due to the deterministic nature, this DSP problem is equivalent to a finite-horizon deterministic finite-state (DFS) optimal control problem with the following elements:

- T:= |ν| − 1 stages:
- State space X = ν
- Now for the location of the Robot at time = t, $x_t := (R_x, R_y)$,
  Control space:

$$U(x_t) = \begin{cases} (R_x, R_y + 1): \text{Up} \\ (R_x, R_y - 1): \text{Down} \\ (R_x + 1, R_y): \text{Right} \\ (R_x - 1, R_y): \text{Left} \\ (R_x + 1, R_y + 1): \text{Top} - \text{right} \\ (R_x - 1, R_y + 1): \text{Top} - \text{left} \\ (R_x - 1, R_y - 1): \text{Bottom} - \text{left} \\ (R_x + 1, R_y - 1): \text{Bottom} - \text{right} \end{cases}$$

- Motion model,
  For $u_t \in U$,
  $x_{t+1} = f(x_t, u_t) = \begin{cases} x_t, & if\ x_t = \tau \\ u_t, & otherwise \end{cases}$
- Stage cost,
  $l(x, u) = \begin{cases} 0, & if\ x = \tau \\ c_{x,u}, & otherwise \end{cases}$
- Terminal cost,
  $q(x) = \begin{cases} 0, & if\ x = \tau \\ \infty, & otherwise \end{cases}$
- Objective: $\min_{u_{0:T-1}} q(x_T) + \sum_{t=0}^{T-1} l(x_t, u_t)$
  Such that,
  $x_t + 1 = f(x_t, u_t)$, t = 0, . . . , T – 1
  $x_t \in X$,
  $u_t \in U(x_t)$,

III. TECHNICAL APPROACH

Since we have a moving target, we cannot catch the target with our robot with a states sequence obtained from a single go of planning with variations of Vanilla Label Correcting Algorithms like: Dijkstra, A* etc. We have to integrate multiple such sequences to account for the moving target.

The solution for the problem discussed in this paper is based on multiple runs of the popular A* algorithm. Since we already discussed that applying it in the elementary form will lead to an unwanted inefficiency as this algorithm if run repeatedly: recalculates the whole path each time after taking one step towards the goal to incorporate for changing goals. To eliminate this inefficiency to some extent, we will only expand 'n' (n is a variable that needs to be tuned) number of nodes in the label correcting instead of computing the shortest path for the whole map with respect to a target which will not even be there till the time the robot reaches this position.

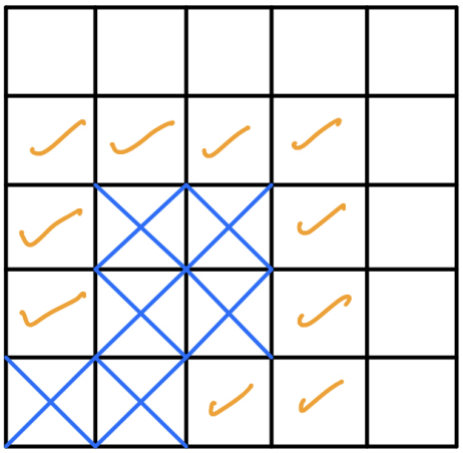
The number of nodes is chosen based on the time it takes to calculate the shortest path for reaching the most promising node in that small portion of map (the portion resulted from expansion of 'n' nodes.). Since the target is moving every 2 seconds, the 'n' should be such that it gives us the next state of the robot in strictly less than 2 seconds in order to intercept the robot in finite time with multiple runs of the variation of A* that is elaborated below:

Modified A* algorithm for 'n' nodes expansion:

```
OPEN ←{s}, EXPANDED ←{}, OUTER←{}
g_s = 0, g_i = ∞ for all i ∈ v \ {s}
while OPEN is not empty do
    if length of OPEN >= n
        Full = True
    Remove i with the smallest f_i = g_i+h_i
    Insert i into EXPANDED
        for j ∈ Children(i) and j ∈/ CLOSED do
            if g_j > (g_i + c_ij ) then
                g_j←(g_i + c_ij )
                Parent(j) ← i
                    if j = τ then
                        Update priority of j
                        OUTER← OUTER ∪{j}
                        break
                    else
                        if Full is True
                            Update priority of j
                            OUTER← OUTER ∪{j}
                        else
                            Update priority of j
                            OPEN ← OPEN ∪{j}
```

Some terms related to the algorithm written above:

- h_t is a positive lower bound on the optimal cost from node j to τ known as a heuristic function
  $$0 \leq h_t \leq \text{dist.}(j, \tau)$$
  In this project, h_t is taken as the Euclidean distance of the robot to the goal. This will be admissible as it ignores all the obstacles to calculate the straightforward distance whereas the real path may be longer because of moving along the obstacles.
- OPEN:
  It is initiated as an empty list which is of the form of a priority stack with node of the least f_i value is at the top of the stack in order to pop that node easily while the algorithm runs.
- EXPANDED:
  It is the list of the nodes that has been explored by the algorithm. Length of this list will be at max. 'n'.
- OUTER:
  It is the list that made of immediate children (that are not obstacles) that are in the outer layer of nodes of EXPANDED list.



In the above figure, the blue cross marks represent nodes in the EXPANDED list at the end of the while loop in the algorithm (empty OPEN list). Hence, n = 6 in this setup. The nodes with orange ticks represents the nodes in the OUTER list of which, the node with the smallest f_i is chosen as the most promising intermediate destination.

- ‘Children()’ function:
  It returns the immediate children of the node that is expanded. The children that constitute for the obstacle in the map are simply ignored.
  - So, if there are no obstacles in the vicinity of the expanded node, the function will return a list of 8 immediate children explained in U(x_t).
  - If p (<8) number of those child nodes constitutes for the obstacles, then the function will return ‘8-p’ nodes that are not obstacles (hence free space).
- Parent:
  It is a dictionary that will store the best parent of a node. This will be used to backtrack the shortest path.

The line **4 and 5** in the algorithm accounts for the fact that the algorithm will stop appending the nodes in the OPEN list from the moment its length touches the value ‘n’. After that, the nodes will only be taken out from the OPEN list and their children will be appended to a new list, i.e., OUTER instead of expanding the OPEN list further. These nodes will be the outer nodes (children of the expanded nodes) which will then be evaluated based on their f_i value, the node with the minimum f_i value will be the most promising intermediate node in greater goal to intercept the node where the target is situated. Then a shortest path will be stored starting from node ‘s’ to this most promising node by backtracking the path with the help of the Parent dictionary that we constructed in our algorithm.

This stored path is fed to the ‘runtest’ function node by node as it only allows input of the immediate child node (adjacent node) from the current robot position and moves the robot to that node. When the list storing the sequence of nodes (constituting the shortest path) becomes empty (as we are removing the nodes which are being fed to the function ‘runtest’), the modified A* algorithm re-runs again and repeats the process till the target is intercepted. This repetitive process is automatic when the function ‘runtest’ is run for only one time on any of the random environment maps that are provided.

The ‘n’ value tradeoff:

- If we chose the value of n be very small, then our function will give us the next move in very less time, but it may fail to give us the desired result. This is because the smaller n constitutes to the short-sighted approach due to which the algorithm may oscillate between a set of nodes forever in the case of complex obstacle orientations.
- If the ‘n’ value is very large, then then our algorithm may fail to give us the next move in less than 2 seconds (time in which the target changes the position).

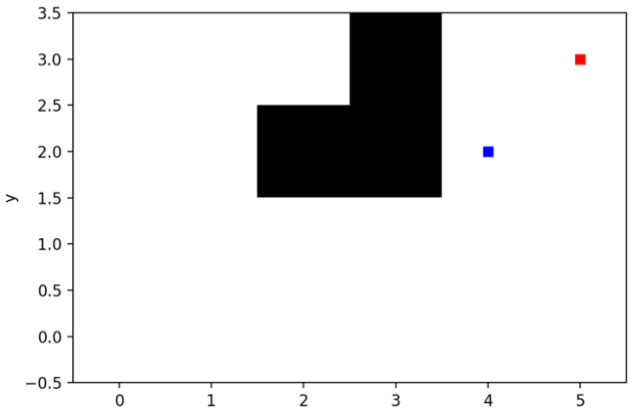
After some tuning in the n value, I observed that the small maps do not need a limited node expansion technique for fast results. So, the n can be large enough to account for the far-sighted approach in the larger maps as long as it is yielding the next move within 2 seconds. So, the any value of n from 200 to 500 will do the work. These values will be obsolete for the smaller maps (maps with lesser number of nodes than n). These values are obtained by tuning the n-value on multiple runs of the algorithm.

IV. Results and Discussion

So, the above written technical approach yielded the following results when tried it on different maps (2D matrices with 0’s representing free space and 1’s representing obstacles). Same color convention is followed in every map as stated in map 0.

Map 0:

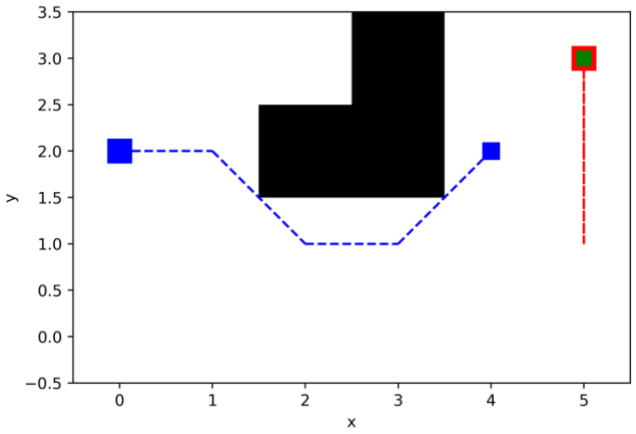
The image below represents the final map where the blue and red marks represent the robot and target position respectively, when the catching condition is met.



Final map

This image below shows the path travelled by the target (Red) and the path travelled by the robot (Blue). Bigger

blue mark represents the initial position of the robot and the smaller one represents the final position after the catching condition is met. Initial position of target is shown in green, and the final position is shown by red mark as before.

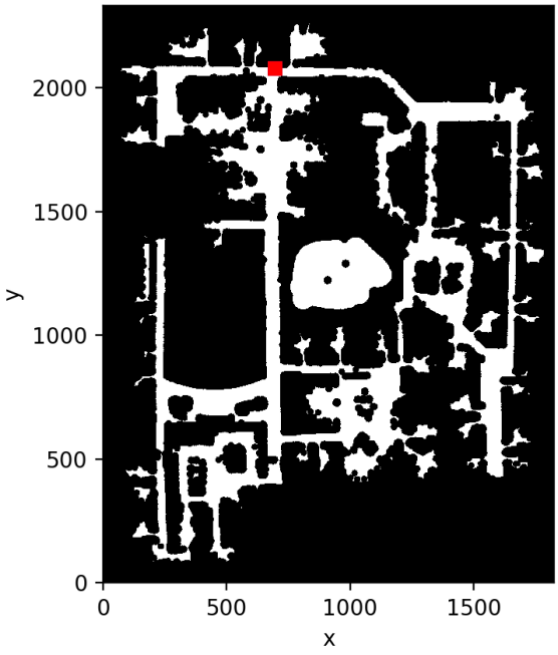


Overall path

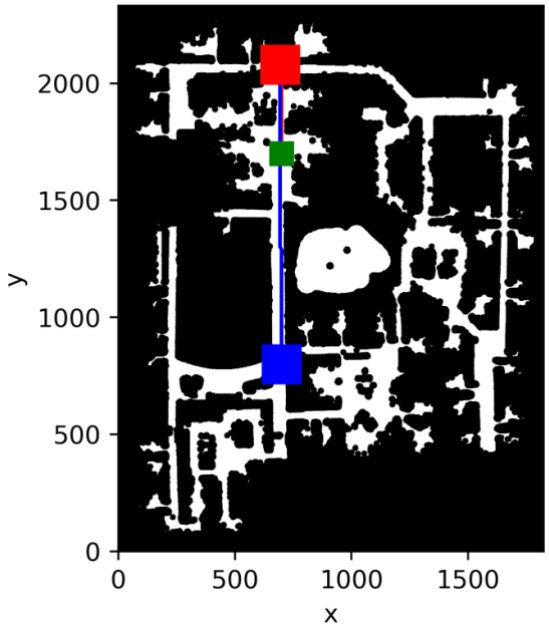
Steps to catch the target: 4

The initial position of the Robot and the Target is (0,2) and (5,3) respectively. The final position of the Robot and the Target is (4,2) and (5,3) respectively.

Map 1:



Final map (color convention is same as map 0)

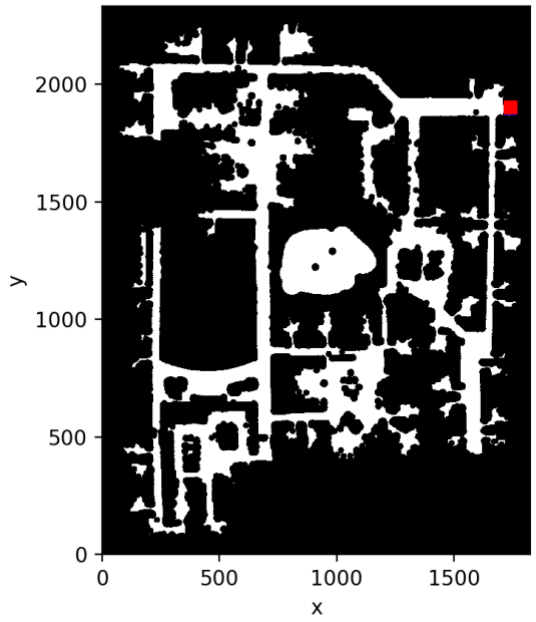


Overall path (color convention is same as map 0)

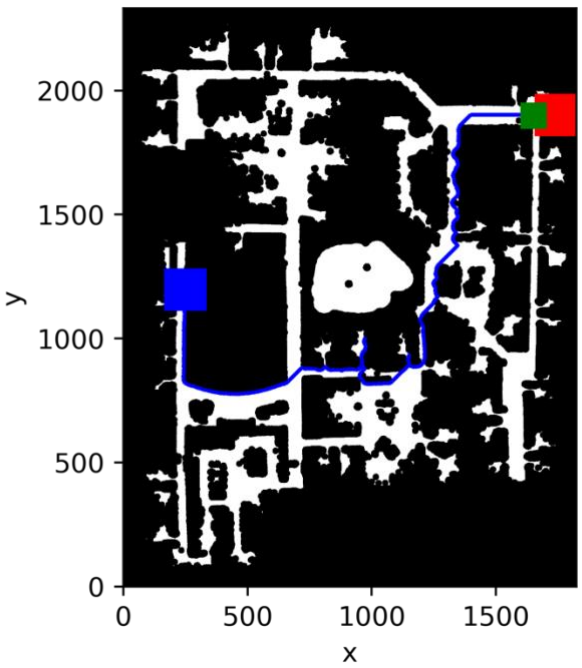
Steps to catch the target: 1280

The initial position of the Robot and the Target is (699,799) and (699,1699) respectively. The final position of the Robot and the Target is (694,2079) and (694,2079) respectively.

Map 1b:



Final map

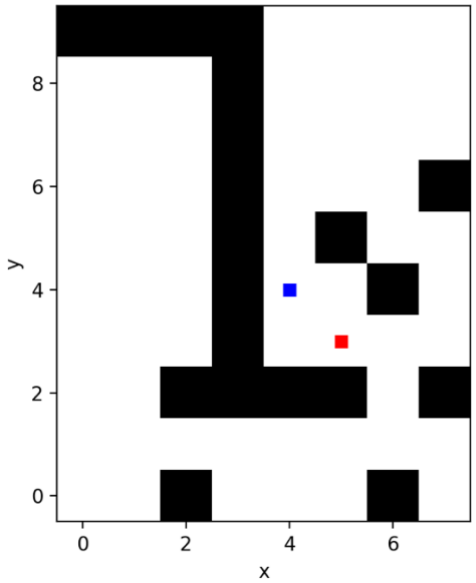


Overall path

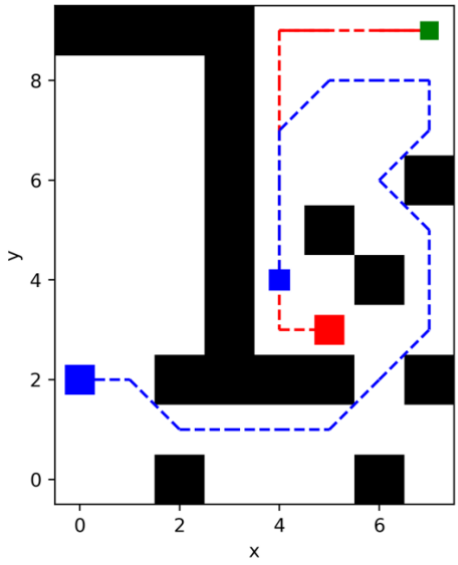
Steps to catch the target: 3051

The initial position of the Robot and the Target is (249,1199) and (1649,1899) respectively. The final position of the Robot and the Target is (1739,1901) and (1739,1902) respectively.

Map 2:



Final map

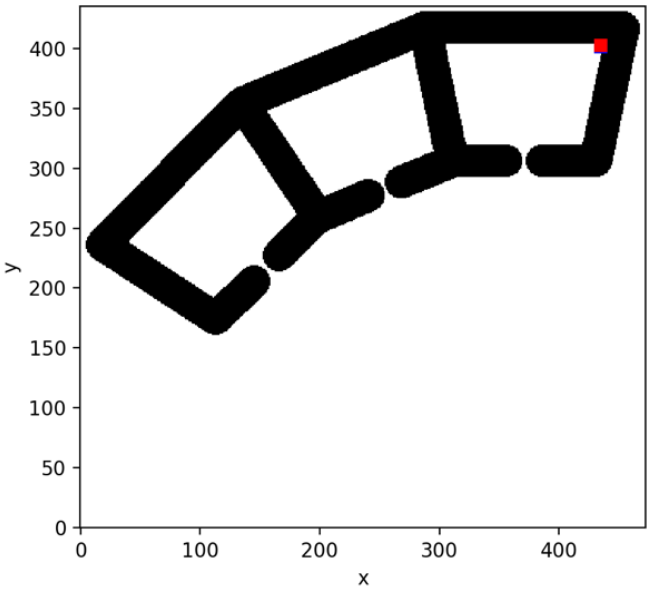


Overall path

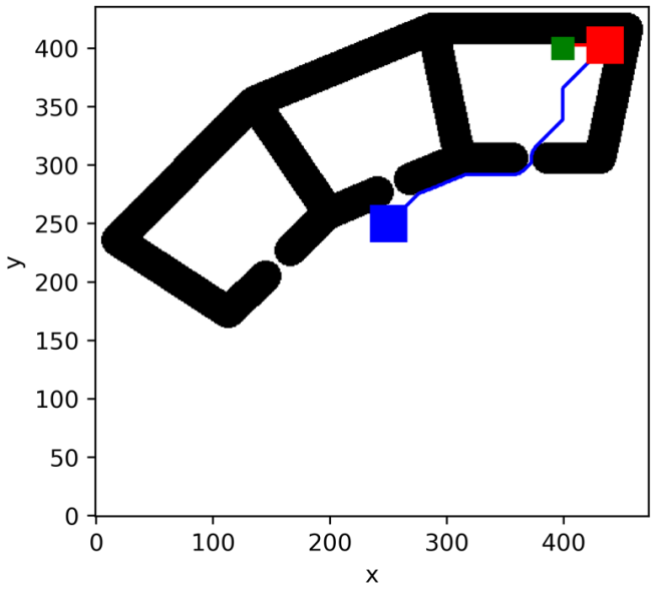
Steps to catch the target: 18

The initial position of the Robot and the Target is (0,2) and (7, 9) respectively. The final position of the Robot and the Target is (4,4) and (5,3) respectively.

Map 3:



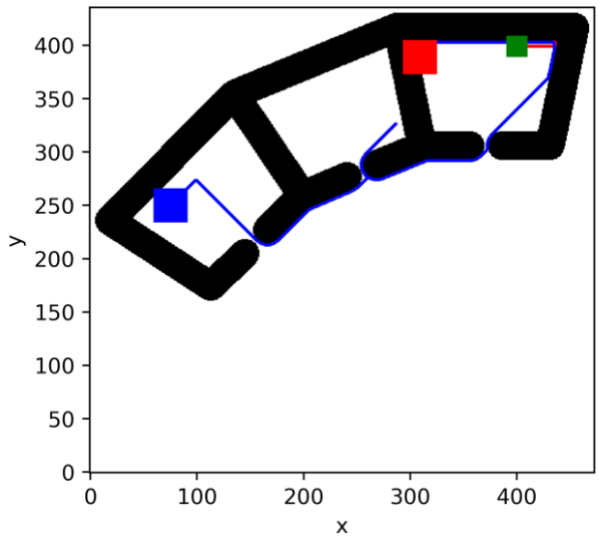
Final map



Overall path

Steps to catch the target: 223

The initial position of the Robot and the Target is (249,249) and (399, 399) respectively. The final position of the Robot and the Target is (435,402) and (435,403) respectively.

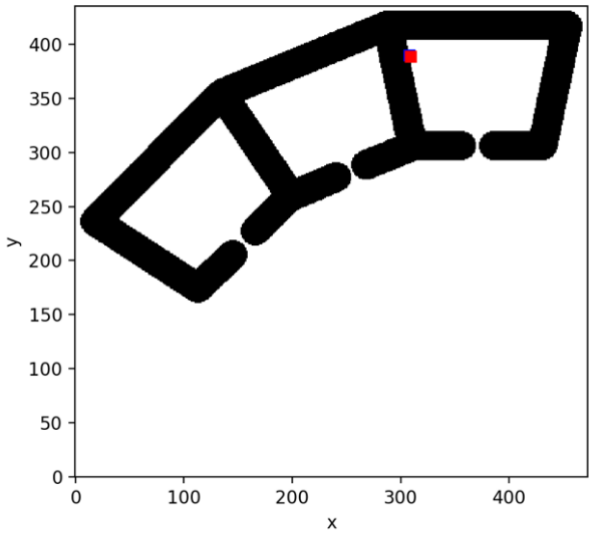


Overall path

Steps to catch the target: 645

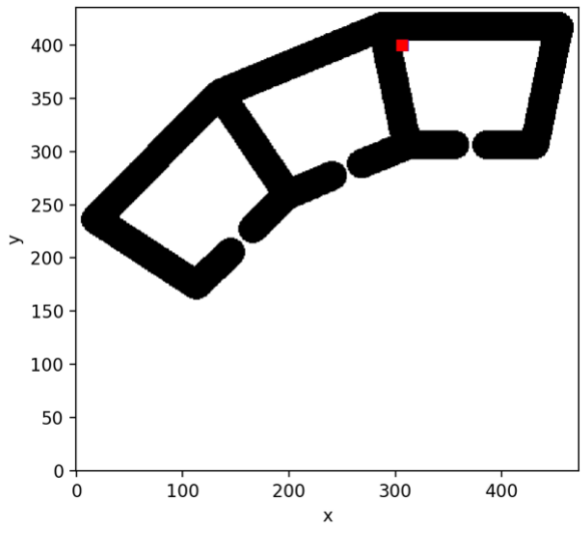
The initial position of the Robot and the Target is (74,249) and (399, 399) respectively. The final position of the Robot and the Target is (309,385) and (310,384) respectively.

Map 3b:

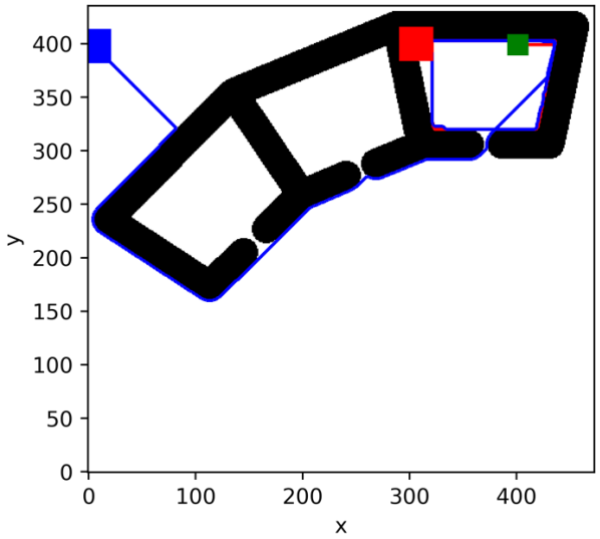


Final map

Map 3c:



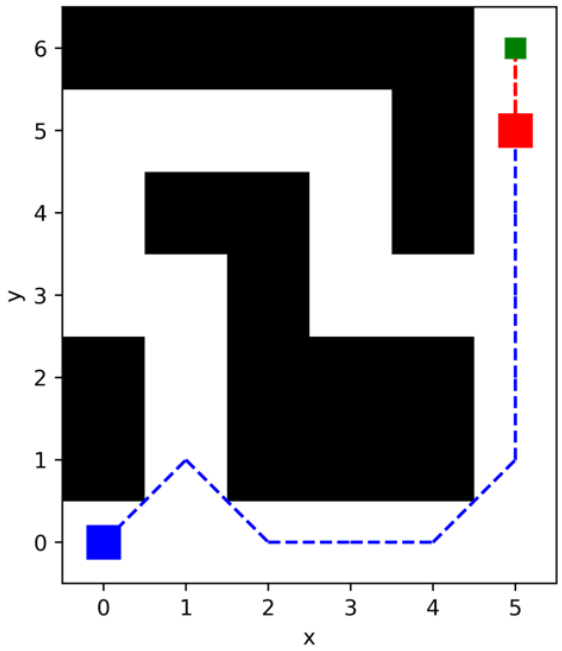
Final map



Overall path

Steps to catch the target: 773

The initial position of the Robot and the Target is (4,399) and (399, 399) respectively. The final position of the Robot and the Target is (307,395) and (308,394) respectively.

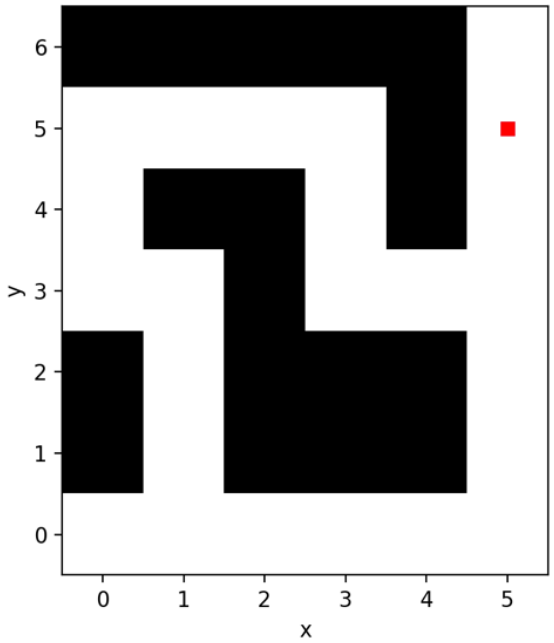


Overall path

Steps to catch the target: 9

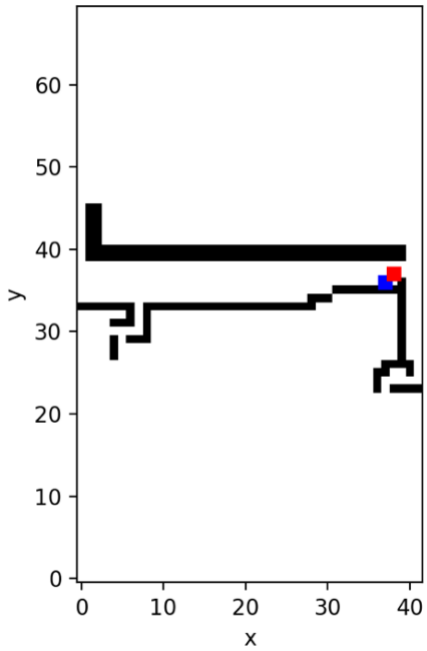
The initial position of the Robot and the Target is (0,0) and (5,6) respectively. The final position of the Robot and the Target is (5,5) and (5,5) respectively.

Map 4:

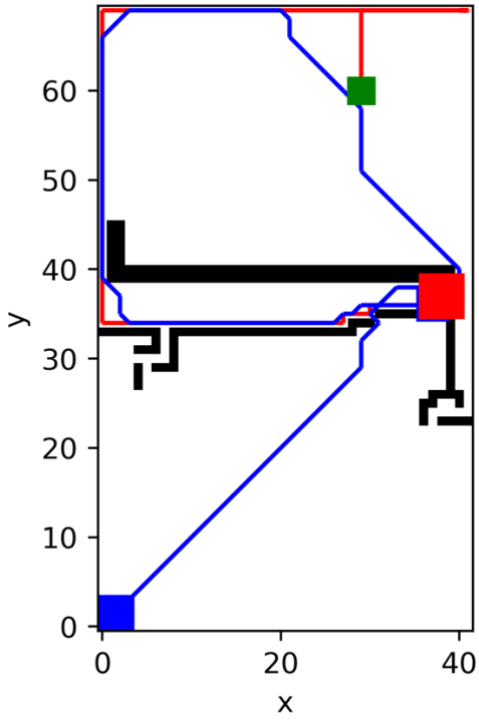


Final map

Map 5:



Final map

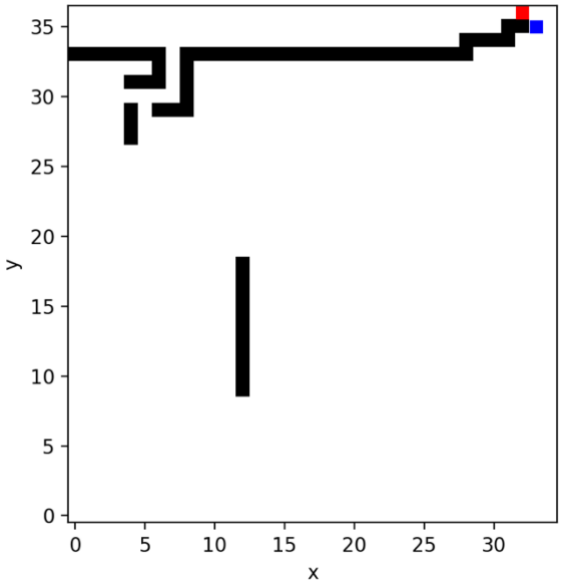


Overall path

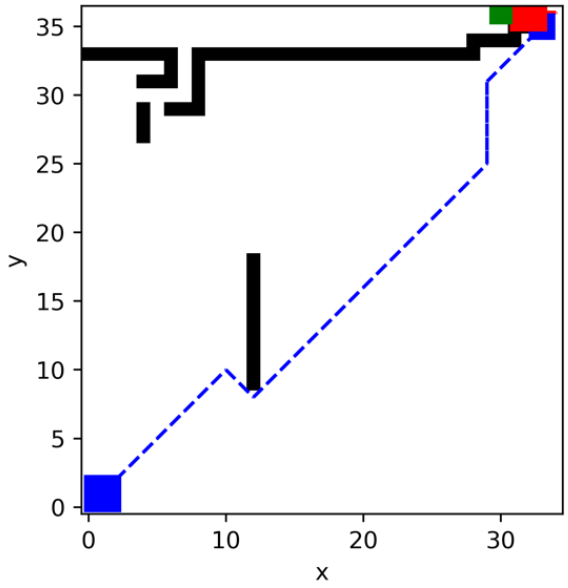
Steps to catch the target: 161

The initial position of the Robot and the Target is (0,0) and (29,59) respectively. The final position of the Robot and the Target is (37,36) and (38,37) respectively.

Map 6:



Final path



Overall path

Steps to catch the target: 39

The initial position of the Robot and the Target is (0,0) and (29,36) respectively. The final position of the Robot and the Target is (33,35) and (32,36) respectively.

Results discussion:

- In the smaller maps like **map0, map2, map4** and **map6**, the path followed by the robot is fairly optimal as we are our 'n' is much higher compared to the map size. This results in the expansion of all the nodes that are present in the map in one run of the algorithm. This accounts for the farthest-sighted approach and hence yields the best possible path for optimality.
- In larger and medium sized maps like **1b, 3b, 3c**, the robot seems to have taken some portions of the overall paths that are unnecessary. Visually, we can say there exists some path which is even shorter than the paths shown in this map. This is because our algorithm forces us to compute path expanding only n number of nodes which is resulting in a short-sighted approach and hence giving us the larger path, which could have been obtained by expanding all the nodes. This is necessary in order to get the next state of the robot within 2 seconds.
- Despite being medium sized and large maps, plotted paths of **map3** and **map1** shows a simple and optimal path. This is because the robot starts from a position where it doesn't need to pass through a lot of obstacles hence the heuristics are

much closer to the actual distance. This results in the optimal path.

- Despite being a small map, plotted path of **map5** shows a sub-optimal path. This is because of the very narrow free space available between the way of robot and the target, the heuristics doesn't approximate the actual distance very well. Hence, resulting the path shown in the plot.

Future approaches:

- Algorithms like RTAA* can be applied to solve this problem. In this, we update the heuristics of the explored nodes and hence eliminates the possibility of cycles in the path which allows us to use even smaller value of 'n'.
- Map de-sizing can be explored further in the project to yield faster results for the larger maps. Its principal is to rescale the map by compressing a m*m grid to an equivalent 1*1 cell of some p*q map. This will result in a shorter map and hence faster results. Also, this technique may result into a path that is different from the shorter possible path as we are losing some intricacies of information of the map while rescaling it. But that is one of the tradeoffs which can be decided with respect to the priorities at hand.

V. ACKOWLEDGEMENTS

I would like to thank my professor of the course ECE 276B: Prof. Nikolay Atanasov and Teaching Assistant: Hanwen Cao for providing the knowledge that is used to successfully complete this project.

VI. REFERANCES

- Lecture notes of ECE 276B provided by the professor: Nikolay Atanasov for the year 2022 at UC San Diego.