Motion Planning

Vaibhav Bishi
Department of Electrical and Computer Engineering
University of California, San Diego
La Jolla, USA
vbishi@ucsd.edu

***Abstract*—This project presents an approach for implementing a search-based motion planning algorithm, specifically a variant of A* algorithm, on a robot for interception of a moving target. The results show that the robot successfully intercepts the target in various environments.**

***Keywords*—search-based motion planning, A* algorithm, autonomous navigation, moving target interception, heuristics**

I. Introduction

Motion planning is one of the most crucial problems in autonomous robotics. The problem is to find the best possible way to reach the goal from the starting position while minimizing the costs of executing the steps. This is basically an optimization problem, although we are aware of the environment here, i.e., the environment is fixed. This kind of shortest path problems can be solved extremely efficiently using **Dynamic Programming (DP)** algorithms such as Label Correcting (LC), Dijkstra's algorithm and A*, rather than simple brute-force methods. We basically solve the problem by constructing an optimal control policy for the robot from start to end. However, when we have a moving target, the knowledge of the distance from the target position in real-time is valuable for developing a planning strategy. The **A* algorithm** is a search-based motion planning algorithm that takes this into account in the form of heuristic values and thus, performs better than general DP algorithms like Dijkstra's algorithm. In this project, we implement the vanilla A* algorithm and its variants to solve the motion planning problem for the moving target. Real-time motion planning for moving targets has potential applications in gaming, simulations, autonomous vehicles and missile technology where the agent has to reach its goal which is mobile in the shortest time possible.

II. Problem Statement

The objective is to develop a search-based motion planning algorithm to intercept the moving target in the environment. We have 3 major elements that are essential in describing the problem at hand, namely, the environment, robot and target.

Environment: The environment is a 2D map in the form of an $m \times n$ grid consisting of free spaces and obstacles. The environment $envmap$ can be represented as a matrix of zeros and ones such that

$$envmap(x, y) = \begin{cases} 0, & free \\ 1, & obstacle \end{cases}$$

where (x, y) refers to the cartesian coordinates of a point in the grid. Any environment provided in the problem is known to us and remains static throughout.

Robot: The robot is the pursuing agent that has to intercept the moving target in the environment. It is treated as a point object and occupies a single cell in the $envmap$ at any given instant. The position $robotpos$ refers to the current position of the robot in the environment in terms of (x, y) coordinates. The robot is allowed to move in any of the 8 possible directions, i.e., left/right, up/down and diagonally, with respect to the current cell. It is, however, constrained to move by only one step to any of its possible neighbors at a time, and it can neither move over to any obstacles in the $envmap$ nor go beyond the map boundaries.

Target: The target is the mobile terminal goal in the environment which the robot has to intercept. It is also treated as a point object and occupies a single cell in the $envmap$ at any given instant. The position $targetpos$ refers to the current position of the target in the environment in terms of (x, y) coordinates. The target is allowed to move strictly in 4 directions, i.e., left/right and up/down, with respect to the current cell. The target makes a move every 2 seconds and the target planner tries to maximize the distance the robot can achieve using a minimax decision rule.

To achieve our objective, we need to design the robot planner in such a way that it returns the next best position for the robot within 2 seconds, given the current $robotpos$ and $targetpos$ in the $envmap$ at any given instant.

The navigation problem at hand can be represented as a shortest path problem at each instant. Since this is a noiseless motion case, it can be formulated as a **Deterministic Shortest Path (DSP)** problem. The elements of the DSP are described explicitly as follows.

We consider our 2D $m \times n$ $envmap$ as a graph with a finite **vertex set** $\mathcal{V}$, where each vertex corresponds to a grid coordinate (x, y) in the map, such that $|\mathcal{V}| = mn$.

The **edge set** $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ represents the connections between all the cells in the $envmap$. The **edge weights** $C := \{c_{ij} \in \mathbb{R} \cup \{\infty\} | (i, j) \in \mathcal{E}\}$, where c_{ij} denotes the arc length or cost for moving from vertex i to j.

The **path** is defined as a sequence $i_{1:q} := (i_1, i_2, \ldots, i_q)$ of nodes $i_k \in \mathcal{V}$. All the paths from $s \in \mathcal{V}$ to $\tau \in \mathcal{V}$ can be defined as $\mathcal{P}_{s,\tau} := \{i_{1:q} | i_k \in \mathcal{V},\ i_1 = s,\ i_q = \tau\}$. The **path length** is the sum of edge weights along the path and be defined as $J^{i_{1:q}} = \sum_{k=1}^{q-1} c_{i_k, i_{k+1}}$.

The objective is to find the **shortest path** from a **start node** $\boldsymbol{s}$ to an **end node** $\boldsymbol{\tau}$. In other words, we need to find a path that has the minimum length from node s to node τ, such that

$$dist(s, \tau) = \min_{i_{1:q} \in \mathcal{P}_{s,\tau}} J^{i_{1:q}}$$

and

$$i^*_{1:q} = \underset{i_{1:q} \in \mathcal{P}_{s,\tau}}{\operatorname{argmin}} J^{i_{1:q}}$$

This DSP problem is equivalent to a finite-horizon **Deterministic Finite-State (DFS)** problem as the motion is noiseless. We can formulate this as a DFS with the following parameters.

Time horizon: $T \coloneqq |\mathcal{V}| - 1$ stages

State space: $\boldsymbol{\mathcal{X}} := \mathcal{V}$

Control space: $\boldsymbol{\mathcal{U}} = \{N, S, E, W, NE, NW, SE, SW\}$

where,

$$N = North = (0,1)$$

$$S = South = (0,-1)$$

$$E = East = (1,0)$$

$$W = West = (-1,0)$$

$$NE = NorthEast = (1,1)$$

$$NW = NorthWest = (-1,1)$$

$$SE = SouthEast = (1,-1)$$

$$SW = SouthWest = (-1,-1)$$

Motion model: $x_{t+1} = f(x_t, u_t) \coloneqq \begin{cases} x_t, & x_t = \tau \\ x_t + u_t, & otherwise \end{cases}$

Since the robot can move by one step to its valid neighbors at an instant, we can define the cost for adjacent vertex transition from node i to node j with control u as follows

$$c_{ij,u} = \begin{cases} 0, & i = j \\ 1, & u \in \{N, S, E, W\} \\ \sqrt{2}, & u \in \{NE, NW, SE, SW\} \\ \infty, & obstacle/out\ of\ bounds \end{cases}$$

Stage cost: : $\ell(x, u) \coloneqq \begin{cases} 0, & x = \tau \\ c_{x,u}, & otherwise \end{cases}$

Terminal cost: $\mathfrak{q}(x) \coloneqq \begin{cases} 0, & x = \tau \\ \infty, & otherwise \end{cases}$

The objective is to get the **optimal control sequence** $u_{0:T-1}$ in order to get the best possible path from start to end such that

$$\min_{u_{0:T-1}} \mathfrak{q}(x_T) + \sum_{t=0}^{T-1} \ell(x_t, u_t)$$

Since the target is moving in our case, we need to implement this DSP approach multiple times whenever the target moves, in order to get the next best possible robot state at that instant. The algorithm starts again with the new $robotpos$ and $targetpos$, and terminates when the robot eventually reaches a neighboring cell of the target, such that

$$|robotpos(x) - targetpos(x)| \leq 1$$

and

$$|robotpos(y) - targetpos(y)| \leq 1$$

where,

$robotpos(x)$ refers to robot's x-coordinate

$robotpos(y)$ refers to robot's y-coordinate

$targetpos(x)$ refers to target's x-coordinate

$targetpos(y)$ refers to target's y-coordinate

III. TECHNICAL APPROACH

In order to solve the DSP problem, we can implement various DP algorithms such as LC, Dijkstra's algorithm, A*, etc. However, in our project, we have a mobile target where the knowledge of the distance of our robot from the target at any given instant of time can prove extremely valuable for developing a planning strategy. We base our approach on the popular A* algorithm, a search-based motion planning algorithm that takes the distance from target into account in the form of heuristic values, performing much better than general DP algorithms like LC and Dijkstra's algorithm. Since the target is moving, we break the entire planning problem into multiple independent DSP problems, by estimating the best next robot move at each instant, and then feeding the new $robotpos$ and $targetpos$ into our algorithm. In this project, we implement the vanilla A* algorithm and certain variants of A* to solve the path planning problem.

Firstly, we describe the basic A* algorithm, before delving into its implementation in the context of our problem.

A* algorithm

The A* algorithm uses a combination of cost-to-arrive values and heuristic searching to obtain the shortest path. The cost-to-arrive values for a node are represented as **labels** g_i, which is defined as the lowest cost discovered so far from start node s to each visited node $i \in \mathcal{V}$. The $OPEN$ set decribes the set of nodes that can potentially be part of the shortest path to τ. A* is essentially a modification to LC and Dijkstra's algorithm in which the admissibility requirement into the OPEN set is strengthened as follows

$$\boldsymbol{from} \quad g_i + c_{ij} < g_\tau \quad \boldsymbol{to} \quad g_i + c_{ij} + h_j < g_\tau$$

where, h_j is a positive lower bound on the optimal cost from node j to node τ, known as a **heuristic function**, such that

$$0 \leq h_j \leq dist(j, \tau)$$

The heuristic function must follow certain criteria for the accuracy and efficiency of the algorithm. It must be **admissible** for it to work correctly and **consistent** for higher efficiency.

Admissibility:

$$h_i \leq dist(i, \tau)\ \forall i \in \mathcal{V}$$

Consistency:

$$h_\tau = 0 \quad and \quad h_i \leq c_{ij} + h_j \quad \forall i \neq \tau, j \in Children(i)$$

$\boldsymbol{\epsilon}$-Consistency:

$$h_\tau = 0 \quad and \quad h_i \leq \boldsymbol{\epsilon} c_{ij} + h_j \quad \forall i \neq \tau, j \in Children(i)$$

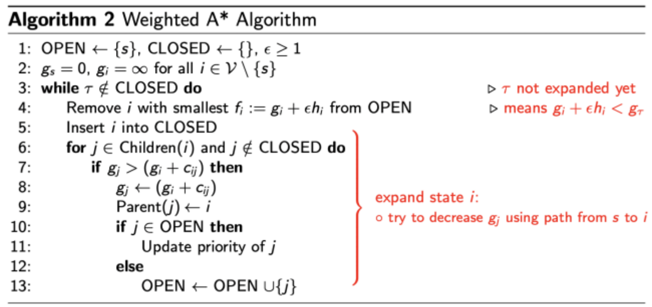
The heuristic function takes the form of a distance metric with common examples like Euclidean distance, Manhattan distance, etc. The more accurately it estimates the actual minimum distance, the more efficient A* becomes.

A* acts as a best-first algorithm since it evaluates each cell's priority in the space with respect to the f-value such that

$$f = g + h$$

It performs with much greater efficiency than the regular LC algorithms as it not only considers the cost of moving to the next cell from the start node, but also the cost of moving from said node to the end node.

The snapshot of the pseudo-code for the regular A* algorithm[1] is given as follows



Algorithm 2 Weighted A* Algorithm

```
1: OPEN $\leftarrow \{s\}$, CLOSED $\leftarrow \{\}$, $\epsilon \geq 1$
2: $g_s = 0$, $g_i = \infty$ for all $i \in \mathcal{V} \setminus \{s\}$
3: while $\tau \notin$ CLOSED do                                ▷ $\tau$ not expanded yet
4:     Remove $i$ with smallest $f_i := g_i + \epsilon h_i$ from OPEN    ▷ means $g_i + \epsilon h_i < g_\tau$
5:     Insert $i$ into CLOSED
6:     for $j \in$ Children($i$) and $j \notin$ CLOSED do
7:         if $g_j > (g_i + c_{ij})$ then
8:             $g_j \leftarrow (g_i + c_{ij})$
9:             Parent($j$) $\leftarrow i$
10:            if $j \in$ OPEN then
11:                Update priority of $j$
12:            else
13:                OPEN $\leftarrow$ OPEN $\cup \{j\}$
```

Firstly, we initialize the $OPEN$ set with the start node s. It refers to the set of candidates for expansion. We initialize the $CLOSED$ set as empty, which refers to the set of states that have already been expanded. We initialize the g-values of s as 0 and rest of the nodes as ∞. As long as we have not expanded the end node τ, we pick the best next node for the robot by calculating the f-values for all the nodes in $OPEN$ and we put that node with the minimum value inside $CLOSED$. We determine all possible children of that node and as long as a child is not in $CLOSED$, each time its g-value is reduced, we correct that value by adding the stage cost. If the child is in $OPEN$, we simply update its priority if needed, otherwise we add it into $OPEN$. We keep track of the parents whenever a child is added to $OPEN$ in order to determine the optimal shortest path given by the set of nodes with the least costs.

Now, we can see how A* can applied in the context of our moving target interception problem. Since the target makes a move every 2 seconds, we need to readjust our plan for the next robot move with respect to the new $targetpos$. Thus, instead of planning just once with respect to a static goal and moving along that optimal path, as is done in A*, we now plan our optimal path to the target every time the target moves by running A* and we move along said path to the next cell with respect to the current $robotpos$. The new $robotpos$ and $targetpos$ become then become the current $robotpos$ and $targetpos$ and we run A* again. This cycle continues until the robot reaches a neighboring cell of the target.

The heuristic function $getHeuristics()$ that has been implemented in our code takes the Euclidean distance which can be defined as follows

$$h_i \coloneqq \|\boldsymbol{x}_\tau - \boldsymbol{x}_i\|_2$$
$$= (x - targetpos(x))^2 + (y - targetpos(y))^2$$

The $getChildren()$ function checks all the adjacent 8 cells with respect to the current $robotpos$ and determines whether a cell is inside the map-bounds and if it is obstacle-free, thus obtaining all the valid children of that node.

The $pathfinder()$ function generates the optimal path from $robotpos$ to $targetpos$ at any given instant using the $parent$ dictionary via backtracking.

Now, it is quite evident that the vanilla A* implementation as discussed would be very time-consuming, especially for larger maps, as it has to replan the optimal trajectory from the $targetpos$ every single time the target moves. We introduce certain variations in the A* algorithm to deal with the larger maps.

A* with node expansion

In this A* variant, we make a major change to the vanilla A* algorithm implemented already by limiting the number of nodes that can be expanded during a single A* run. So, instead of estimating the shortest path from the target, which requires complete expansion of the grid, we now determine the best possible node in $OPEN$ using the f-values after expanding only N nodes in $CLOSED$. We then trace the optimum path to that node from the current $robotpos$ and get the next cell the robot must move to in this A* run. We also notice that in larger maps, when the target is very far from the robot, the slight change in the generalized location of the target in the map does not impact the optimal direction that the robot must take to get closer to the robot. Therefore, after one run of A* with N nodes expansion, instead of taking the next best state adjacent to the current $robotpos$ and reimplementing A* again, we store the optimal path till N nodes and let the robot move on that path until it reaches the best node in $OPEN$ at that time. This helps reduce the time taken by the robot to move as it does not have to replan all the time, and the robot is able to get closer to the target much faster.

Now, even though we compute the fresh heuristics with respect to $targetpos$ every time the target moves, we do not update the heuristic values after every node expansion, even if it encounters an obstacle. Thus, it is highly likely that the robot may get stuck in a local minimum, i.e., get stuck around an obstacle or a corner. We introduce another A* variant to deal with this problem.

A* with node expansion and heuristic update

In this A* variant, we implement almost the same algorithm as A* with N node expansion, but we now update the heuristics of all the nodes in $CLOSED$ after expansion as follows

$$h_i = f_{j^*} - g_i$$

where

$$j^* = \operatorname*{argmin}_{j \in OPEN} g_j + h_j$$

Then once we have the fresh $robotpos$ and $targetpos$, we check if the $targetpos$ has changed and we recompute the heuristics of all the nodes in the $envmap$ as follows

$$h(\boldsymbol{x}) := \max(H(\boldsymbol{x}, target_{new}), h(\boldsymbol{x}) - h(target_{new}))$$

where, $h(\boldsymbol{x})$ is the heuristic value of node $\boldsymbol{x}$ and $H(\boldsymbol{x}, \boldsymbol{x}')$ is the Euclidean distance between nodes $\boldsymbol{x}$ and $\boldsymbol{x}'$.

This ensures that the new heuristic values are consistent with respect to the new $targetpos$. This algorithm has been implemented with inspiration taken from the **Moving-Target Adaptive A* (MT-A*) algorithm**[2]. This enables the robot to avoid getting stuck around local minima in the map and get to the target eventually. The only catch here is that we need to recompute heuristic values every time we run A* which has a considerable impact on the algorithm runtime.

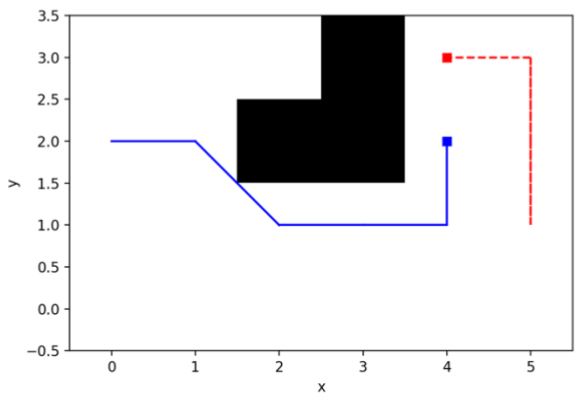
The class $environment()$ is created in our code to initiate the environment and store different parameters such as $envmap$, h-values, controls, etc for repeated use. This class also takes an argument called $algo$ which decides the algorithm that will be implemented.

IV. Results

In this project, we have implemented the A* algorithm and its variants to intercept a moving target in multiple known environments of different sizes with different initial $robotpos$ and $targetpos$. We compute the number of moves taken by the robot to intercept the target and visualize the entire trajectory of the robot (marked in blue) and the target (marked in dashed-red) over time.

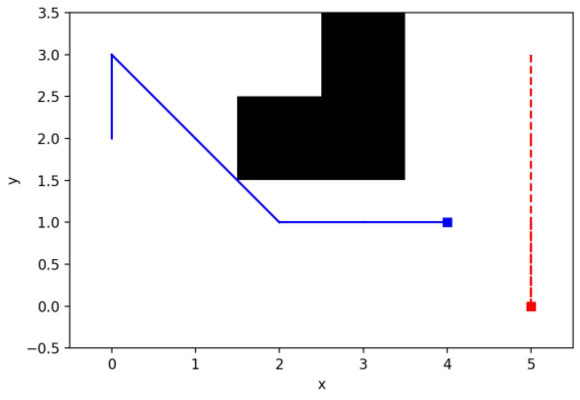
Testmap0

Initial $robotpos$ (0,2), Initial $targetpos$ (5,3)



*vanilla A**

Final $robotpos$ (4,2), Final $targetpos$ (4,3)
Number of moves made: 5

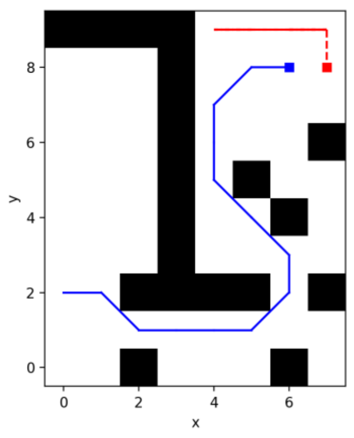


A with 500 node expansion*

Final $robotpos$ (4,1), Final $targetpos$ (5,0)
Number of moves made: 5

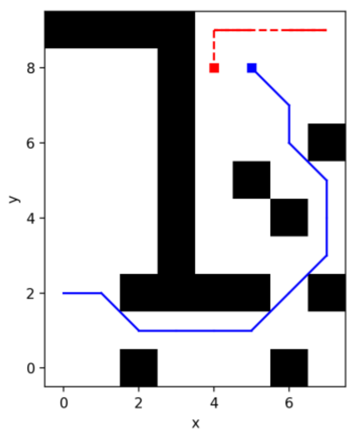
Testmap2

Initial $robotpos$ (0,2), Initial $targetpos$ (7,9)



*vanilla A**

Final $robotpos$ (6,8), Final $targetpos$ (7,8)
Number of moves made: 13

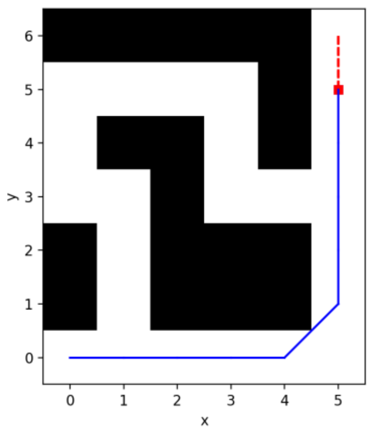


A with 500 node expansion*

Final $robotpos$ (5,8), Final $targetpos$ (4,8)
Number of moves made: 12

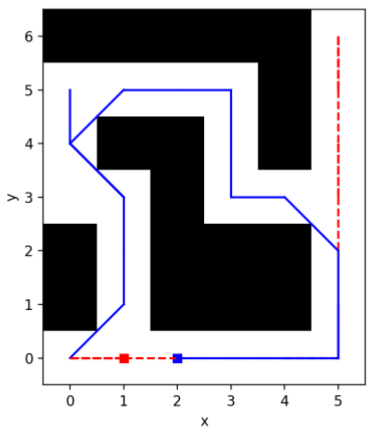
Testmap4

Initial $robotpos$ (0,0), Initial $targetpos$ (5,6)



*vanilla A**

Final $robotpos$ (5,5), Final $targetpos$ (5,5)
Number of moves made: 9

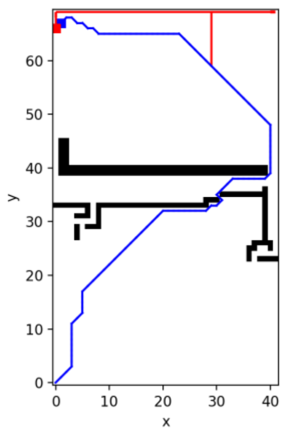


A with 500 node expansion*

Final $robotpos$ (2,0), Final $targetpos$ (1,0)
Number of moves made: 22

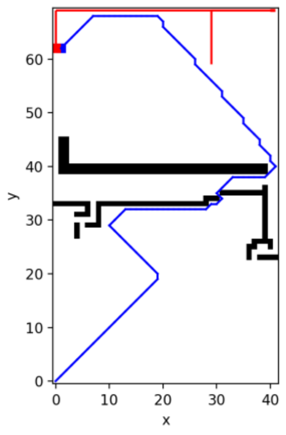
Testmap5

Initial $robotpos$ (0,0), Initial $targetpos$ (29,59)



*vanilla A**

Final $robotpos$ (1,67), Final $targetpos$ (0,66)
Number of moves made: 102

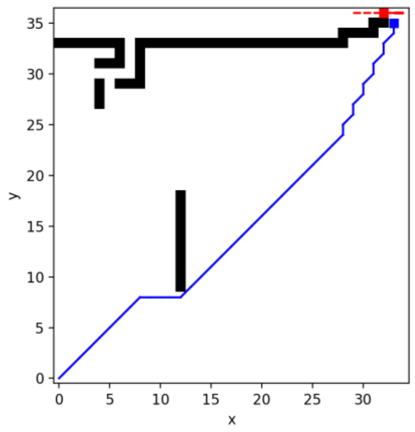


A with 500 node expansion*

Final $robotpos$ (1,62), Final $targetpos$ (0,62)
Number of moves made: 108

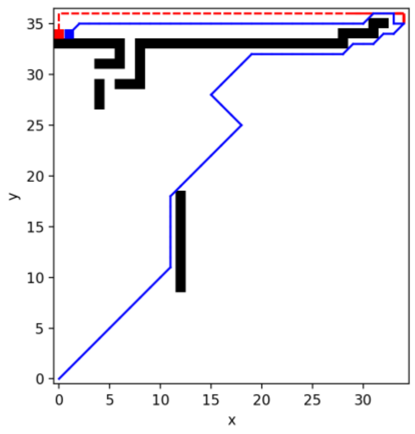
Testmap6

Initial $robotpos$ (0,0), Initial $targetpos$ (29,36)



*vanilla A**

Final $robotpos$ (33,35), Final $targetpos$ (32,36)
Number of moves made: 39

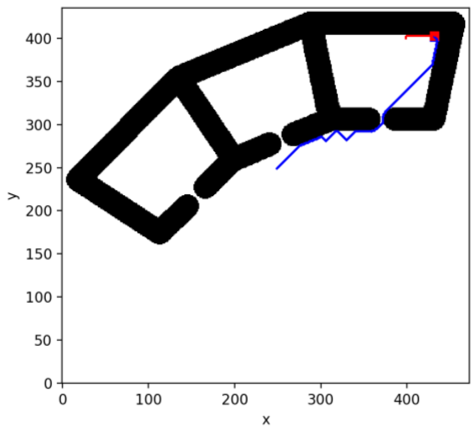


A with 500 node expansion*

Final $robotpos$ (1,34), Final $targetpos$ (0,34)
Number of moves made: 81

Testmap3

Initial $robotpos$ (249,249), Initial $targetpos$ (399,399)

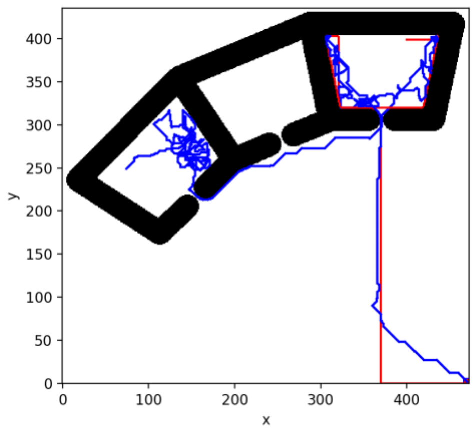


A with 500 node expansion*

Final $robotpos$ (432,402), Final $targetpos$ (432,403)
Number of moves made: 223

Testmap3b

Initial $robotpos$ (74,249), Initial $targetpos$ (399,399)

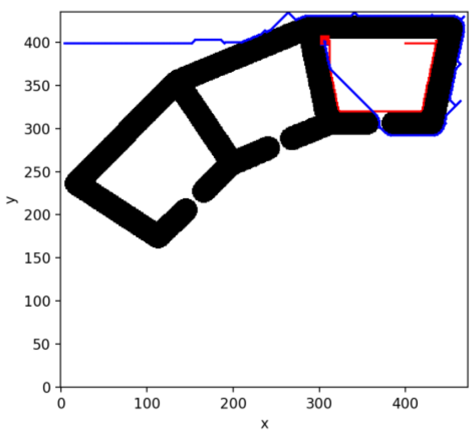


A with 500 node expansion and heuristic update*

Final $robotpos$ (471,1), Final $targetpos$ (472,0)
Number of moves made: 2310

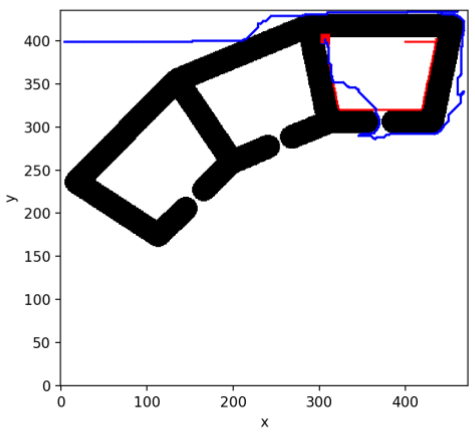
Testmap3c

Initial $robotpos$ (4,399), Initial $targetpos$ (399,399)



A with 500 node expansion*

Final $robotpos$ (306,402), Final $targetpos$ (5,0)
Number of moves made: 781

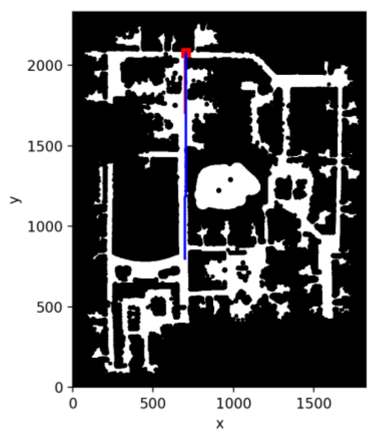


A with 500 node expansion and heuristic update*

Final $robotpos$ (307,402), Final $targetpos$ (307,403)
Number of moves made: 896

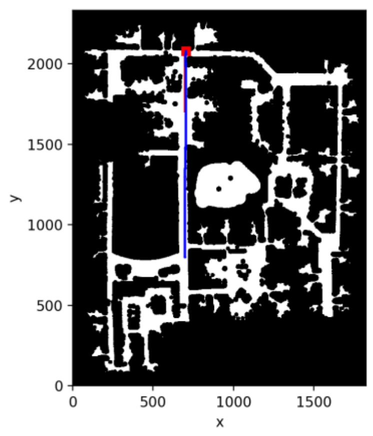
Testmap1

Initial $robotpos$ (699,799), Initial $targetpos$ (699,1699)



A with 100 node expansion*

Final $robotpos$ (703,2078), Final $targetpos$ (704,2079)
Number of moves made: 1279



A with 500 node expansion and heuristic update*

Final $robotpos$ (704,2079), Final $targetpos$ (704,2079)
Number of moves made: 1280

We can observe that the vanilla A* works perfectly fine for the smaller maps, however it takes a long time for the larger maps and thus, A* with node expansion is the better option. Also, for $Testmap3b$, the simple A* fails and thus, we implement the A* with the heuristic update which is able to circumvent the local minima issue and intercept the target.

We can try other experiments to improve the performance of the algorithm. Increasing the epsilon value ϵ can make the algorithm terminate faster if the heuristics are well-informed. Reducing the map resolution for larger maps can also help in making the algorithm more efficient. We can try other algorithms specifically made for intercepting moving targets such as **Moving-Target D* Lite**, **General Adaptive A*** and even sampling-based planning algorithms such as **RRT**.

References

[1] Nikolay Atanasov, ECE276B, Planning & Learning in Robotics, Lecture 7, Search-based Motion Planning

[2] Koenig, Sven, Maxim Likhachev, and Xiaoxun Sun. "Speeding up moving-target search." *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*. 2007.