Motion Planning using A* - Moving target capture

Rishabh Bhattacharya
Department of Mechanical & Aerospace Engineering
University of California, San Diego
ribhattacharya@ucsd.edu

***Index Terms*—A*, weighted A*, motion planning, heuristic, target capture, DSP, deterministic shortest path**

I. Introduction

Path planning is a major area of emphasis for numerous robots, due to its importance in the mobility of any autonomous system. A robot must be able to plan its path in the wake of obstacles in its way, either by offline pre-computation or by online learning. As is obvious, online learning is computationally intensive, and it also requires the computations to be done efficiently to minimize delay between perception and control action.

Even with its high computational requirements, online learning is required in instances where the goal/target is not stationary. This makes the problem challenging enough, to the point where pre-computed policies cannot be used for target capture. We have to recompute the optimal policies and paths with the updated goal and start position at every iteration, or at best use the previously computed data and adapt it to the new (but small) changes.

The forward Dynamic Programming algorithm (used in PR1) computes the shortest paths from the start s to all nodes (including goal τ). Usually, most nodes are not part of the shortest path from s to τ, which is a waste of computational resources and CPU runtime. This makes it undesirable for real-time implementations.

Label correcting (LC) algorithms do **not** necessarily visit every node of the graph. LC algorithms prioritize the visited nodes i using the cost-to-arrive values $V_t^F(i)$, which make them a better candidate for realtime search-based motion planning algorithms.

For this project, we use the weighted A* algorithm (which is a special type of LC algorithm) for motion planning. Our environments have an agent, walls/obstacles, and a moving target. The objective is to catch the goal in a reasonable amount of time, while the target itself is moving away from our agent. Every path computation is required to be within 2 secs, else the target will move by multiple steps. We have 11 environments of varying sizes, obstacles and initial start positions to test out algorithm and discuss the results.

We begin by setting up the problem as a Deterministic Shortest Path (DSP) problem. We then define our graph, nodes, edges and costs. We then use A* motion planning algorithm to compute our optimal path for the environments and discuss the results.

II. Problem formulation

The overall problem consists of a series of single moves made by the robot and target consecutively, until the robot catches the target. Every single iteration is a static environment with robot position and target position fixed. This particular instance can be formulated as a Deterministic Shortest Path (DSP) problem (II-A), with the objective to find the shortest path from robot to target.

Given:

1) An environment with initial robot and target positions
2) Target is evading the robot using a *minimax* decision rule given in *targetplanner.py*. This information should **not** be made available to the path planning algorithm.
3) DSP$_i$ (II-B) for the i^{th} iteration, which needs to be solved to get the shortest/optimal path

Assumptions:

1) Model is deterministic
2) Target can only move in straight lines (North, South, East, West), while the robot can move diagonally as well (North, South, East, West, North-east, North-west, South-east, South-west)
3) All transition costs are positive ($c_i \geq 0$)

Objective: Given the target location, compute the closed loop optimal move for the robot (amongst 8 adjacent cells) within 2 seconds for every iteration, until the robot intercepts the target.

A. Deterministic Shortest Path (DSP) problem

Every single iteration is a new DSP problem with start (s) and goal (τ) locations changed.

Given: Consider a graph defined by the tuple $(\mathcal{V}, \mathcal{E}, \mathcal{C}, p_f, s, \tau)$ where,

$\mathcal{V}$ is a discrete and finite set of states

$\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$

p_f is the motion model

$\mathcal{C} = \{c_{ij} \in \mathbb{R} \cup \{\infty\} \mid (i,j) \in \mathcal{E}\}$

s is the start node

τ is the end node

$\mathcal{E}$ is the set of edges connecting any two adjacent vertices (parent-child pair). $\mathcal{C}$ is the set of edge costs/weights for the edges in $\mathcal{E}$. c_{ij} denotes weight/cost from parent i to child j.

Define:
Path: a sequence $i_{1:q} := (i_1, i_2, \ldots, i_q)$ of nodes $i_k \in \mathcal{V}$

All paths from (start) $s \in \mathcal{V}$ to (goal) $\tau \in \mathcal{V}$ are given by:

$$\mathcal{P}_{s,\tau} := \{i_{1:q} \mid i_k \in \mathcal{V}, i_1 = s, i_q = \tau\}$$

Path length: sum of edge weights/costs along the path:

$$J^{i_{1:q}} = \sum_{k=1}^{q-1} c_{i_k, i_{k+1}}$$

Assumption: There are no negative cycles in the graph, i.e., $J^{i_{1:q}} \geq 0$, for all $i_{1:q} \in \mathcal{P}_{i,i}$ and all $i \in \mathcal{V}$. Optimal path need not have more than $|\mathcal{V}|$ elements.

Objective: find a path that has the minimum cost from node s to node τ :

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

B. DSP parameters for our environment

Vertices $\boldsymbol{\mathcal{V}}$: set of all *free* cells in the environment of the form $(x,y) \in \mathbb{Z}^2$ where $x \in \{0,1,\cdots m-1\}$, $y \in \{0,1,\cdots n-1\}$ and $m \times n$ are the map dimensions.

Edges $\boldsymbol{\mathcal{E}}$: set of all edges between a parent i and child j. For every $\mathcal{V}_i$, we have a maximum of 8 children $\mathcal{V}_j$, thus 8 edges. If $\mathcal{V}_i = (x,y)$ then,

$$\mathcal{V}_j = \{(x+a, y+b) \mid (a,b) \in (-1,0,1)^2 \backslash (0,0)\}$$

where $0 \leq x+a < m, 0 \leq y+b < n$ (children are within map bounds) and $(x+a, y+b)$ is not an obstacle. Number of edges are reduced if an obstacle exists in the adjacent cells (Fig 1).

Motion model $\boldsymbol{p_f}$: If the parent node is given by $\mathbf{x}_i = (x,y)$, then the motion model ($\mathbf{x}_j = f(\mathbf{x}_i, \mathbf{u})$) is (Fig 1),

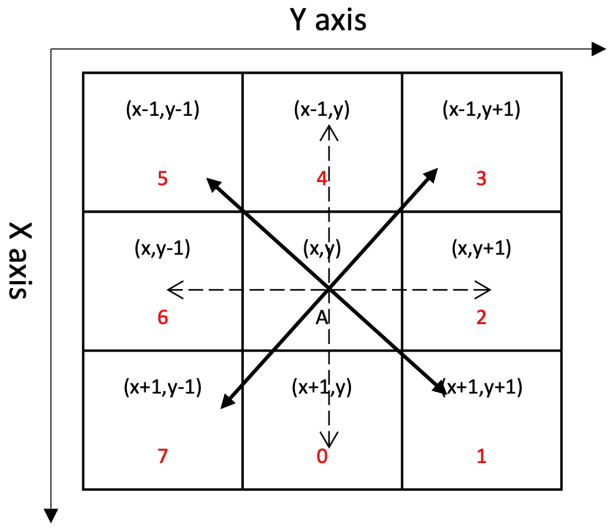

Fig. 1: **Possible moves** around any random node *A*. Notice that since the environments have Y axis extending towards the east and X axis is extending south, moving east and south increases the y & x coordinates respectively. The map is transposed in order to be consistent due to the first array index corresponding to the 'row'. Bold lines denote the diagonal moves ($c_{ij} = \sqrt{2}$), while dashed lines have ($c_{ij} = 1$).

$$\mathbf{x}_j = \begin{cases} (x+1, y) & , \mathbf{u} = \text{ South} \\ (x+1, y+1) & , \mathbf{u} = \text{ South-east} \\ (x, y+1) & , \mathbf{u} = \text{ East} \\ (x-1, y+1) & , \mathbf{u} = \text{ North-east} \\ (x-1, y) & , \mathbf{u} = \text{ North} \\ (x-1, y-1) & , \mathbf{u} = \text{ North-west} \\ (x, y-1) & , \mathbf{u} = \text{ West} \\ (x+1, y-1) & , \mathbf{u} = \text{ South-west} \end{cases}$$

Since no nodes have been created for the obstacles, we do not need to consider collisions.

Edge weights $\boldsymbol{\mathcal{C}}$: $= \left\{c_{ij} \in \{1, \sqrt{2}\} \mid (i,j) \in \mathcal{E}\right\}$ where i is a parent node while j is a child node. c_{ij} is given by Alg. 0, while a visualization is present in Fig. 1.

Algorithm 1 Stage cost c_{ij}

$(x,y) = \mathbf{x}_i$ ▷ $i \rightarrow parent, j \rightarrow child$
if $\mathbf{x}_j \in \{(x \pm 1, y \pm 1)\}$ **then** ▷ diagonal moves (NE, NW, SE, SW)
    $c = \sqrt{2}$
else ▷ Straight line moves (N,S,E,W)
    $c = 1$

Start node $\boldsymbol{s} \in \mathcal{V}$: Robot initial position s, which is known.

Terminal node $\boldsymbol{\tau} \in \mathcal{V}$: Target position τ, which is known.

III. Technical Approach

Our problem consists of a moving target, which has to be intercepted by our robot. For every move that our robot makes, the target takes another step. At this instance, we need to provide the next optimal move to our robot.

We compute this by considering the robot (s) and target (τ) as stationary, and solving a DSP problem for the same, with the objective being to *compute the shortest path* from $s \to \tau$. We will use the A* algorithm for computing the shortest path. Once we obtain this path, the optimal move for the robot is basically to move to the *next node* on the computed path. The target then takes another step accordingly, and we solve a new DSP yet again. This iterative process continues until we successfully intercept the target.

Algorithm 2 Overall problem approach

while robot position $\neq$ target position **do**
    s = robot position, τ = target position
    Solve DSP for $(\mathcal{V}, \mathcal{E}, \mathcal{C}, s, \tau)$ using Label Correcting methods (A* in this case)
    Recover the shortest path from $s \to \tau$
    robot position = next node from the path

In PR1, we solved the DSP problem using Dynamic programming (DP). The DP algorithm computes the shortest paths from *all* nodes to the goal (τ), while the forward DPA computes the shortest path from the start to *all* nodes. As discussed in the I, this ends up being wasteful and detrimental for our project as we have a decision bound of ≤ 2 seconds. This is why Label correcting methods show great potential, since they visit only promising nodes, thereby greatly reducing CPU runtime. Some key ideas of LC methods are:

- Label $\boldsymbol{g_i}$: stores the lowest cost discovered so far from s to each visited node $i \in \mathcal{V}$
- Node expansion: each time g_i is reduced, the labels g_j of the children of i can be corrected: $g_j = g_i + c_{ij}$
- **OPEN** list : set of nodes that can potentially be part of the shortest path to τ

Algorithm 3 Label Correcting Algorithm

OPEN $\leftarrow \{s\}, g_s = 0, g_i = \infty$ for all $i \in \mathcal{V} \backslash \{s\}$
while OPEN is not empty **do**
    Remove i from OPEN
    for $j \in$ Children (i) **do**
        if $(g_i + c_{ij}) < g_j$ and $(g_i + c_{ij}) < g_\tau$ **then** ▷ Only when $c_{ij} \geq 0 \; \forall \; i, j \in \mathcal{V}$
            $g_j = g_i + c_{ij}$
            Parent$(j) = i$
            if $j \neq \tau$ **then**
                OPEN $\leftarrow$ OPEN $\cup \{j\}$
=0

If there exists at least one finite cost path from s to τ, then the Label Correcting (LC) algorithm terminates with $g_\tau = \text{dist}(s, \tau)$, the shortest path length from s to τ. Otherwise, the LC algorithm terminates with $g_\tau = \infty$.

A. A* algorithm

We introduce a heuristic h_i, which is an underestimate of the distance from node i to the goal τ. A heuristic is

- **Admissible** if $0 \leq h_j \leq \text{dist}(j, \tau)$
- **Consistent** if $h_\tau = 0$ and $h_i \leq c_{ij} + h_j$ for all $i \neq \tau$ and $j \in$ Children (i)
- $\boldsymbol{\epsilon}$ **Consistent** if $h_\tau = 0$ and $h_i \leq \epsilon c_{ij} + h_j$ for all $i \neq \tau, j \in$ Children (i), and $\epsilon > 1$

The A* algorithm (0) is a modification to the Label Correcting (LC) algorithm in which the requirement for admission to OPEN list is strengthened:

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

where h_j **must** be admissible, but may/may not be consistent (consistency improves efficiency).

Some theoretical properties of the A* algorithm that help us in this project approach are,

1) A* terminates in a finite number of iterations if $\mathcal{V}$ is finite or if $c_{ij} \geq \delta > 0$ for $i, j \in \mathcal{V}$ and the degree of each node $i \in \mathcal{V}$ is finite. **Thus we can rest assured that our solution will terminate (either path will be found or not found)**.
2) If $c_{ij} \geq 0$ for $i, j \in \mathcal{V}$ and A* uses a consistent heuristic, then:
    a) g_i equals the least-cost from s to i for every expanded state $i \in$ CLOSED. **This mean that nodes i in CLOSED need not be re-opened, and the shortest path from $s \to i$ has been found.**
    b) g_i is an upper bound on the least-cost from s to i for every $i \notin$ CLOSED.
3) A* performs the minimal number of state expansions to guarantee optimality. **Thus our solution will be optimal.**

A rundown of the A* algorithm approach for our problem is,

1) We initialize the OPEN list with the start node s, while the CLOSED list is empty.
2) Node s has a g value $g_s = 0$ while $g_i = \infty, i \in \mathcal{V} \backslash \{s\}$. This is because we know the shortest path from s to itself has 0 length (trivially), while shortest path to other nodes is unknown at the beginning.
3) Node i with the smallest priority $f_i := g_i + h_i$ is moved from OPEN $\to$ CLOSED. This priority is an estimate of the shortest path length that potentially goes through node i $(s \to i \to \tau)$.

$$i = \underset{k \in \text{ OPEN}}{\arg\min} f_k$$

4) Once a node i is removed from OPEN and entered into CLOSED, it shall not re-enter OPEN, since the shortest path from $s \to i$ has already been determined (theoretical property 2.a). **Thus we can terminate the algorithm as soon as τ moves from OPEN $\to$ CLOSED.** Any

remaining nodes in OPEN are not part of the shortest path from $s \to \tau$.

5) As soon as parent i is moved from OPEN $\to$ CLOSED, we need to update the g values of its children j if a shorter path is found. In other words,

$$\textbf{if } g_j > g_i + c_{ij} \textbf{ then } g_j = g_i + c_{ij}$$

This essentially means that a shorter path from $s \to j$ has been found through i ($s \to i \to j$).

6) If child j was unexplored, we add that to the OPEN list with its priority f_i. If child $j \in$ OPEN already, then we update its f_i.

The only difference between vanilla A* and weighted A* algorithm is that

$$f_i := g_i + \epsilon h_i, \begin{cases} \epsilon = 1 & \text{A*} \\ \epsilon > 1 & \text{weighted A*} \end{cases}$$

Algorithm 4 Weighted A* Algorithm

OPEN $\leftarrow \{s\}$, CLOSED $\leftarrow \{\}, \epsilon \geq 1$
$g_s = 0, g_i = \infty$ for all $i \in \mathcal{V} \backslash \{s\}$
while $\tau \notin$ CLOSED **do** ▷ loop until τ is expanded
    Remove i with smallest $f_i := g_i + \epsilon h_i$ from OPEN
    Insert i into CLOSED
    for $j \in$ Children(i) and $j \notin$ CLOSED **do** ▷ this loop expands the parent i and tries to decrease the children's g_j using the path from s to i
        if $g_j > (g_i + c_{ij})$ **then**
            $g_j \leftarrow (g_i + c_{ij})$
            Parent$(j) \leftarrow i$
            if $j \in$ OPEN **then**
                Update priority of j
            else
                OPEN $\leftarrow$ OPEN $\cup \{j\}$

h_i biases our node search in the direction of low $f_i = g_i + h_i$. A more stringent heuristic criterion can reduce the number of iterations required by the A* algorithm. The more accurately h_j estimates $\text{dist}(j, \tau)$, the more efficient it becomes.

- If A^* uses a consistent heuristic, then it is **guaranteed to return an optimal path** to τ (and, in fact, to every expanded node)
- If A^* uses an ϵ-consistent heuristic, then it is **guaranteed to return an ϵ-suboptimal path** with cost $\text{dist}(s, \tau) \leq g_\tau \leq \epsilon\, \text{dist}(s, \tau)$ for $\epsilon \geq 1$. However, it trades optimality for speed, and is orders of magnitude faster than A* in many domains.

For our problem, we have used a Euclidean 2-norm as the heuristic.

$$\boxed{h_i = ||\mathbf{x}_\tau - \mathbf{x}_i||_2}$$

This guarantees that h is admissible ($0 \leq h_i \leq \text{dist}(i, \tau)$) and consistent (using triangle inequality). For some environments and quantitative comparisons, we have also used an $\epsilon = 2$ consistent h.

B. Path recovery

Once we run the A* algorithm, we need to recover the shortest path from the g values. Using g_τ, we backtrack such that

$$\begin{aligned} g_{\tau-1} &= g_\tau - c_{\tau-1,\tau} \\ g_{\tau-2} &= g_{\tau-1} - c_{\tau-2,\tau-1} \\ &\vdots \\ g_s &= g_{\tau-k} - c_{s,\tau-k} \end{aligned}$$

Thus we obtain the shortest path as

$$\tau \to \tau - 1 \to \tau - 2 \cdots \to \tau - k \to s$$

We need to flip it to get

$$s \to \tau - k \to \cdots \tau - 2 \to \tau - 1 \to \tau$$

We finally return node $\tau - k$ as the next optimal move for the robot.

For our project, we can save the parent position for each node (except s) as a field in its object. Thus all we have to do is iteratively retrieve all parent positions starting from $\tau \to s$.

Algorithm 5 Path recovery

Given s.parent = [None, None] ▷ No parent node for s
$i = \tau$, path = [ ] ▷ starting from τ, backtrack towards s
while $i \neq$ [None, None] **do**
    path.append(i)
    $i \leftarrow i$.parent
path = path.flip ▷ path is now $\tau \to s$, flip to $s \to \tau$

C. Skip formulation

In order to deal with larger maps, computing A* and returning only 1 move seemed to be detrimental for our algorithm and computational resources. The runtime would be > 2 seconds, and the target would move away by multiple steps. Thus in order to save on computational time, we used a *skip* based method. Essentially we used a predefined number of steps (say 50) for which our robot would continue to move on the precomputed A* optimal path, instead of just 1 step. Thus we immediately reduce the total A* computations by a factor of 50. As we can see in Table I, we managed to reduce the total computation time by significant margins on most maps.

IV. Results

The vanilla A* algorithm (without weights) was able to catch the target successfully in all maps except map #7 and #1b. For #1b (23), we used $\epsilon = 2$ and skip = 50, which led to target capture in a reasonable amount of time. However, #7 was still not successful since i) the map is too large, thus A* solver takes a lot of time finding even a single optimal path,

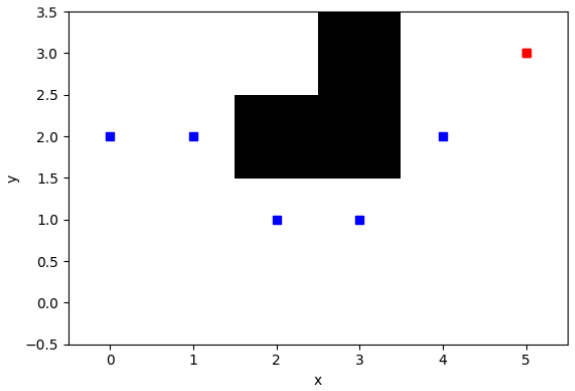

Fig. 2: **Map #0**: Target capture

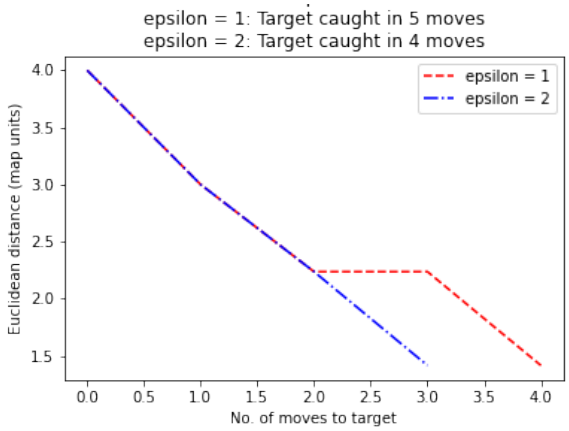

Fig. 3: **Map #0**: As we can see, we saved 1 move with weighted A*. Although that does not have much significance since the map is very small.

ii) the robot and target are placed far apart initially and iii) $\epsilon > 2$ has diminishing returns on the runtime.

TABLE I: Runtime (in seconds) comparison

Map #	$\epsilon = 1$	$\epsilon = 2$	$\epsilon = 2$ w 50 skip
0	0.42	0.19	-
1	239.71	130.99	71.80
2	0.45	0.60	-
3	34.80	11.82	10.33
4	0.37	0.56	-
5	6.26	4.93	-
6	1.94	1.98	-
1b	-	-	251.97
3b	168.19	42.64	24.00
3c	623.75	262.37	92.28

We can see from Table I that increasing ϵ and introducing *skip* helps a lot with total computational runtime. This effect is predominantly visible in larger maps, with little to negligible (sometimes worse) performance in small maps, where vanilla A* is fast enough. Maybe $1 < \epsilon < 2$ would be a sweet spot for the smaller maps that show an increase in runtime with increasing ϵ.

V. Conclusions

We have successfully implemented A* and weighted A* path finding algorithms to catch an evasive target that moves according to the minimax rule. We realised the significance of ϵ heuristic and its significant role in reducing the computational runtime. We also implemented a *skip* based algorithm which proved to be quite faster, and was in fact the only way to compute map #1b (23) realistically in a short time. We would like to explore other strategies to compute the paths even faster, using variations of A* and sampling based search algorithms. A combination of those might help render the solution to map #7, which remains unsolved.

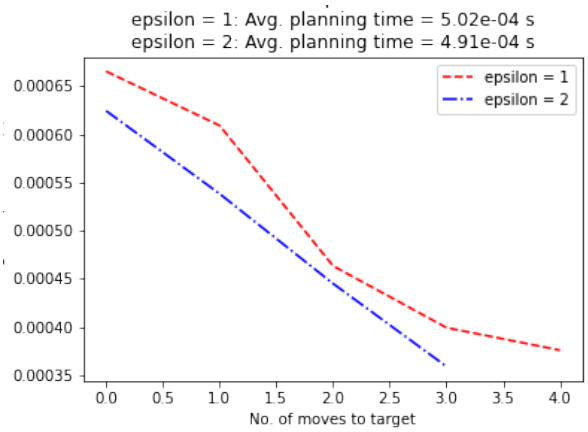

Fig. 4: **Map #0**: Computation time per move has been consistently lower for $\epsilon = 2$.

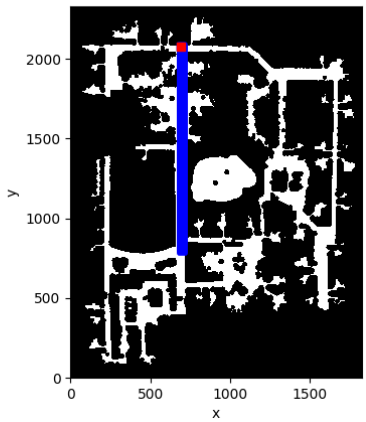

Fig. 5: **Map #1**: Target capture

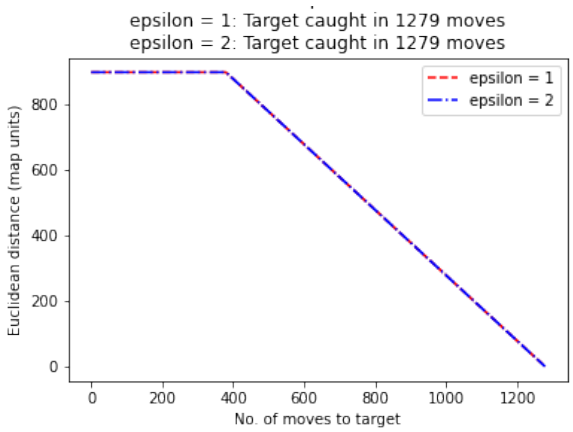

Fig. 6: **Map #1**: The target and goal were almost in a straight line initially. Thus moving with an ϵ heuristic still meant we had to make same amount of moves.

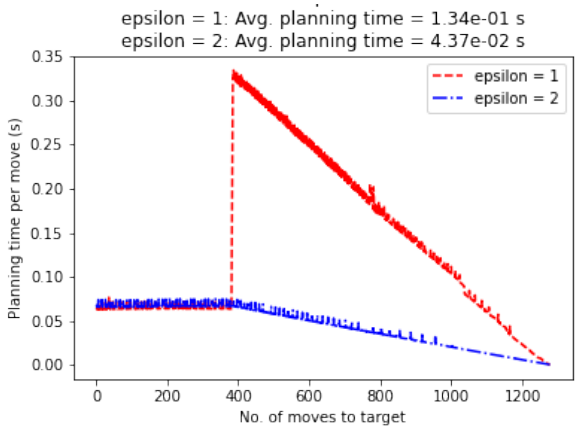

Fig. 7: **Map #1**: Computation time per move has been consistently lower for $\epsilon = 2$. This is the reason total runtime is also significantly lower with the ϵ heuristic. (Table I). The sudden spike in computation time is probably when the target got stuck at the far end and the distance started decreasing b/w it and our robot.

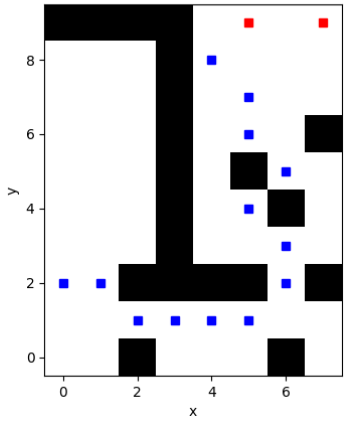

Fig. 8: **Map #2**: Target capture. Target near to the path is the final target position. Target far away from the path is the initial target position.

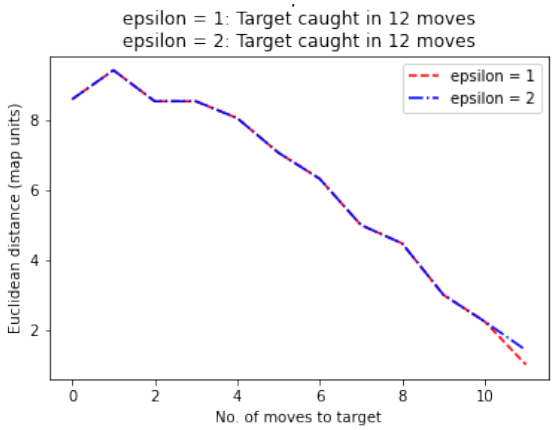

Fig. 9: **Map #2**: Distance vs moves follow closely.

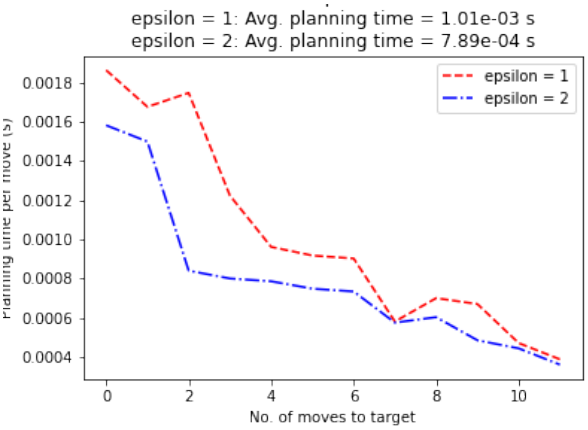

Fig. 10: **Map #2**: Not much difference b/w runtimes.

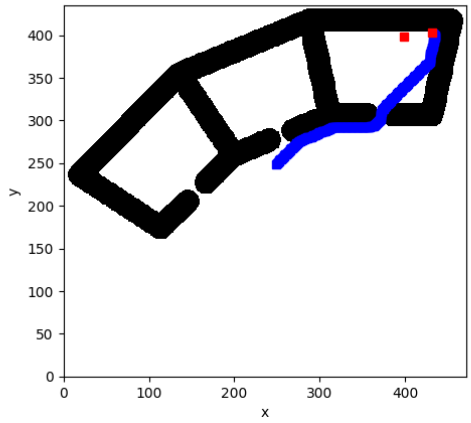

Fig. 11: **Map #3**: Target capture. Target near to the path is the final target position. Target far away from the path is the initial target position.

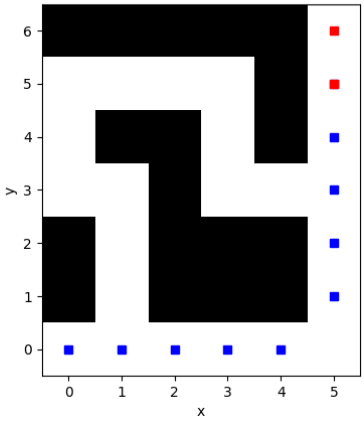

Fig. 14: **Map #4**: Target capture

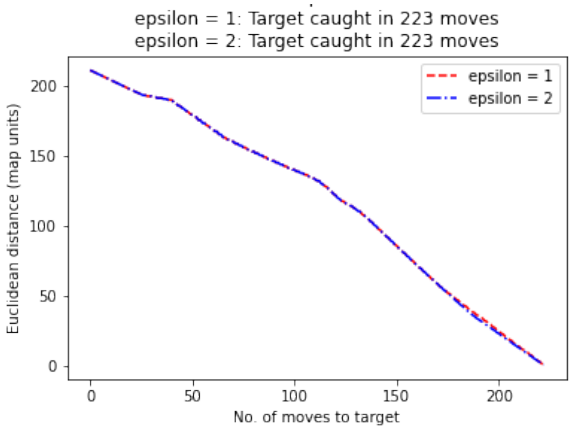

Fig. 12: **Map #3**: Distance vs moves follow closely.

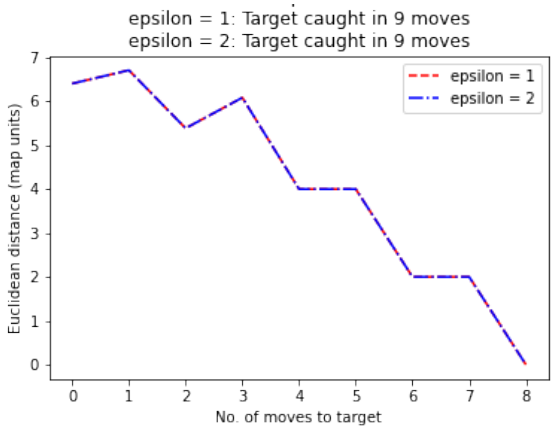

Fig. 15: **Map #4**: Distance vs moves follow closely for a small map.

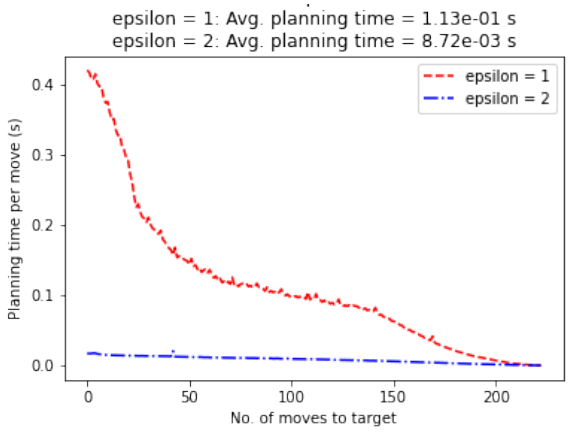

Fig. 13: **Map #3**: Computation time per move has been consistently lower for $\epsilon = 2$. This is the reason we have a third of the runtime in $\epsilon = 2$ compared to $\epsilon = 1$. (Table I)

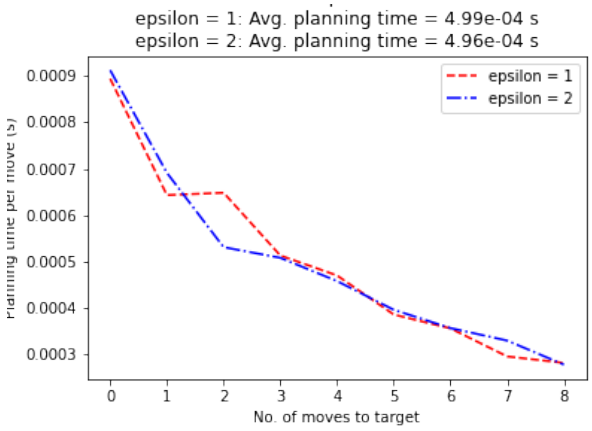

Fig. 16: **Map #4**: Computation time is not affected much due to the small size of the map.

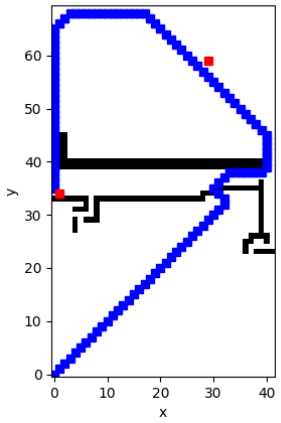

Fig. 17: **Map #5**: Target capture. Target near to the path is the final target position. Target far away from the path is the initial target position.

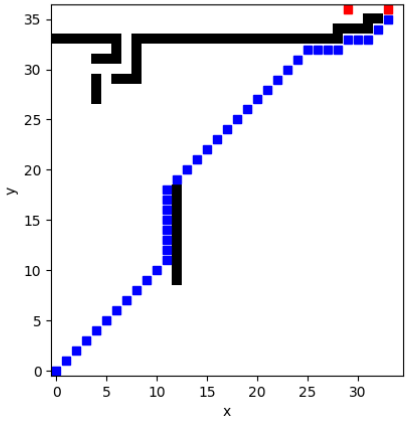

Fig. 20: **Map #6**: Target capture. Target near to the path is the final target position. Target far away from the path is the initial target position.

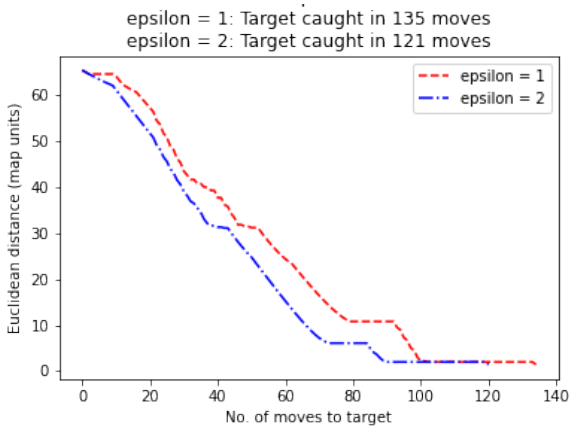

Fig. 18: **Map #5**: As we can see, we saved few moves with weighted A*. The weights start playing a role in larger maps.

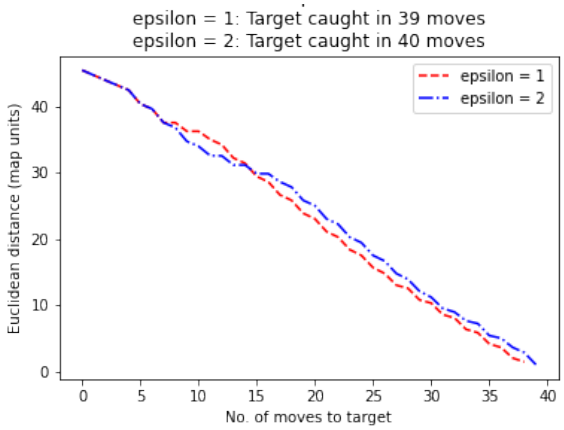

Fig. 21: **Map #6**: Weighted A* might have some difficulties in cases where there is an obstacle right near the goal. It can be illustrated in this plot, where vanilla A* was better by 1 move.

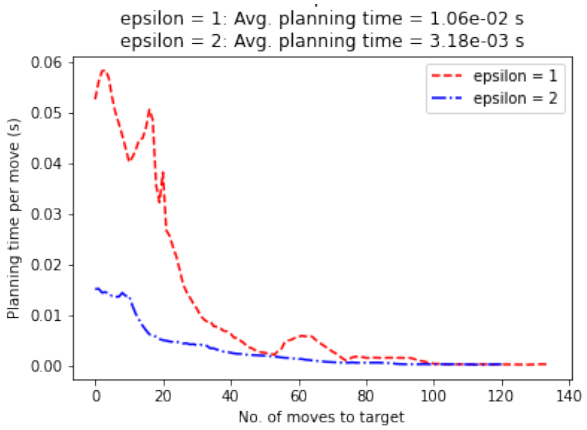

Fig. 19: **Map #5**: Computation time per move has been consistently lower for $\epsilon = 2$. This is the reason we have a reduced runtime in $\epsilon = 2$ compared to $\epsilon = 1$. (Table I)

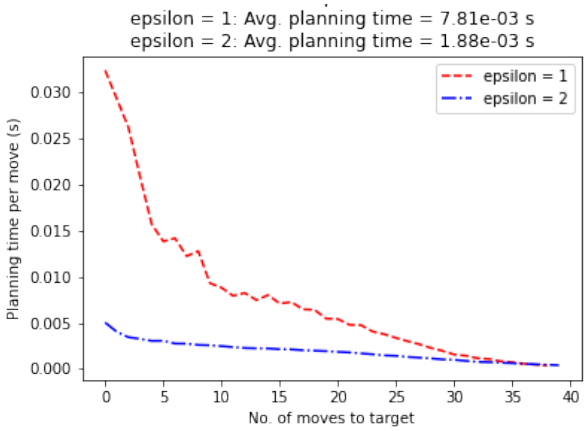

Fig. 22: **Map #6**: Computation time per move has been consistently lower for $\epsilon = 2$. Total runtime is not affected much since we are working in the order of 10^{-3} s.

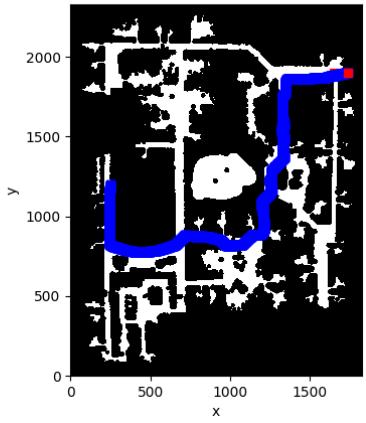

Fig. 23: **Map #1b**: Target capture

Fig. 24: **Map #1b**: Due to the sheer complexity and initial positions being far apart, this map was solved with $\epsilon = 2$ and skip ≥ 50.

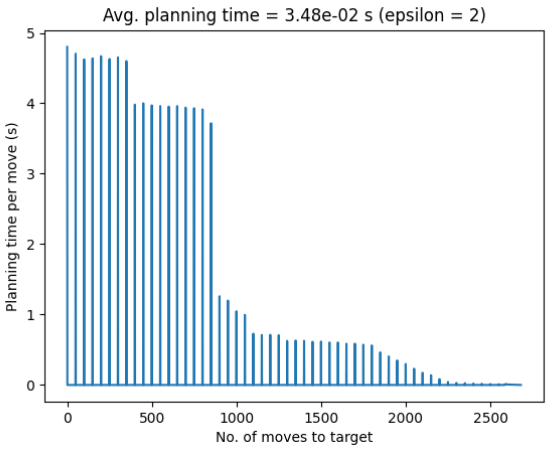

Fig. 25: **Map #1b**: Computation time has spikes every time *roboplanner* was called which is every skip $= 50$

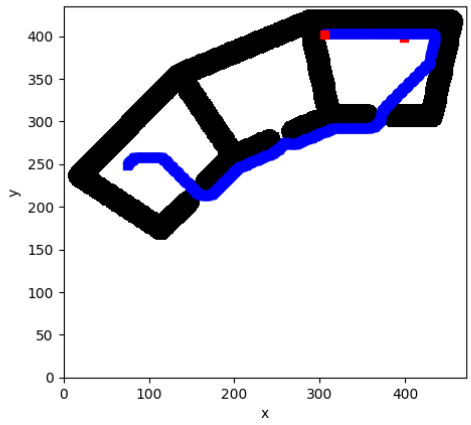

Fig. 26: **Map #3b**: Target capture

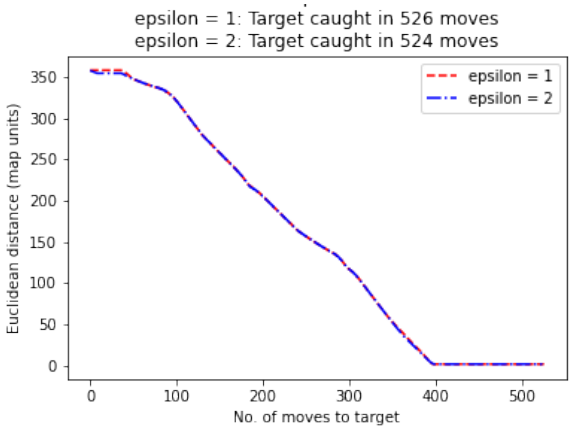

Fig. 27: **Map #3b**: Both lines follow pretty closely.

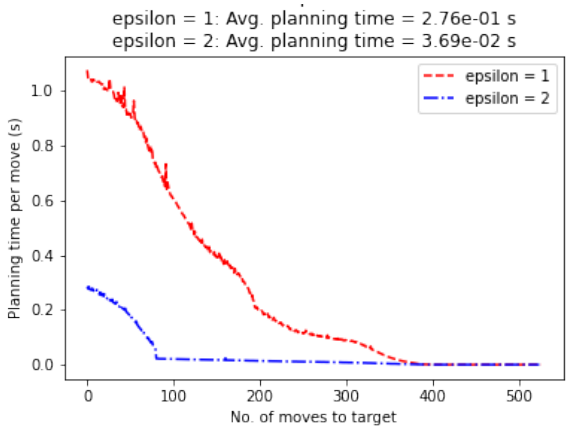

Fig. 28: **Map #3b**: Computation time per move has been an order of magnitude lower for $\epsilon = 2$. This is why total runtime improved by 75% compared to vanilla A*. (Table I)

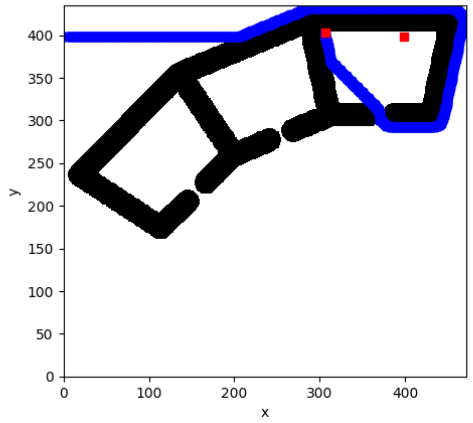

Fig. 29: **Map #3c**: Target capture

Fig. 30: **Map #3c**: We see that at move 400, weighted A* starts diverging. A further analysis at that position could help diagnose the potential issue and possible solution. Even then, the weighted A* algorithm converges significantly faster.

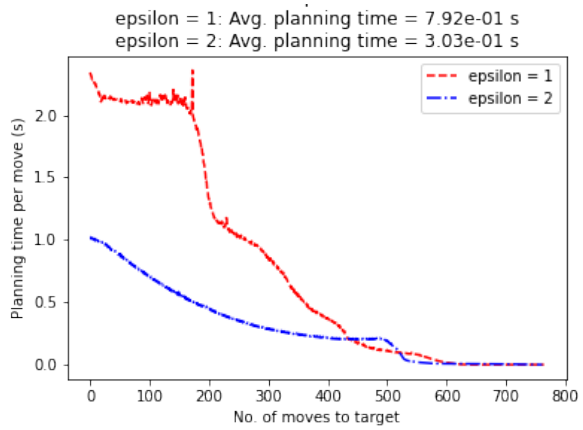

Fig. 31: **Map #3c**: Computation time per move has been consistently lower for $\epsilon = 2$. Thus we witness a significant reduction in computation time. (Table I)