Search and Sampling Based Finite Time Motion Planning with a Moving Target

Kwok Hung Ho
Department of Electrical and Computer Engineering
University of California, San Diego
La Jolla, U.S.A.
khh019@ucsd.edu

Nikolay Atanasov
Department of Electrical and Computer Engineering
University of California, San Diego
La Jolla, U.S.A.
natanasov@eng.ucsd.edu

I. Introduction

With many advancements in the field of robotics, one that is crucial for autonomous mobile robots is motion/path planning. Efficient and optimal motion planning can allow robots to execute dangerous or repetitive tasks to the level of that of a human or even better, whilst minimizing cost in fuel and time. Such tasks can include applications in autonomous vehicles/planes, automated manufacturing, healthcare, search and rescue missions, space/cave/deep water exploration and more.

There are a myriad of motion path planning algorithms that can be applied in different maps, constraints and scenarios. Examples include algorithms that use use heuristics, algorithms that are search-based or algorithms that are sampling-based. Even within these broad categories of search algorithms, there are a host of search algorithms that are optimized for moving targets, dynamic environments, finite planning time, etc.

This paper presents several approaches to solve the 2-D path planning problem in challenging large 2-D map environments with a moving target using the Weighted A^* algorithm, Anytime Repairing A^* (ARA^*), and the Rapidly Exploring Random Tree (RRT) Algorithm, while comparing the speed, space and performance of these methods.

II. Problem Formulation

A. Problem Formulation: Objective and Description

Understanding that there can be a myriad of approaches to solve the problem, only the description of the problem, and associated characteristics that cannot be changed and are inherent to the problem will be presented here. The technical approach, where solutions to the problems that will be described, will be presented in the next section.

The objective is to find the shortest path from the start to the goal, and so the problem at hand can be modeled as a shortest path problem, although due to constraints, this notion can be sacrificed for speed in some cases. Given the 2-D Cartesian coordinates of the starting position of the robot $robotpos$ and the target $targetpos$, the objective is to reach the target in an optimal and efficient way. Due to certainty in the motion model, The Deterministic Shortest Path (DSP) problem is equivalent to a finite-horizon deterministic finite-state (DFS) optimal control problem.

As such, given all paths $\mathcal{P}$:

$$s \in \mathcal{V} \text{ to } \tau \in \mathcal{V} : \mathcal{P}_{s,\tau} = \{i_{1:q} | i_k \in \mathcal{V}, i_1 = s, i_q = \tau\} \quad (1)$$

Where s is the start, τ is the goal, $\mathcal{V}$ is the set of vertices, $i_{1:q}$ is a sequence of nodes $i_k \in \mathcal{V}$, The objective is to find the minimum path length from node s to node τ:

$$dist(s,\tau) = \min_{i_{1:q} \in \mathcal{P}_{s,\tau}} J^{i_{1:q}} \quad (2)$$

$$i^*_{i_{1:q}} = \arg\min_{i_{1:q} \in \mathcal{P}_{s,\tau}} J^{i_{1:q}} \quad (3)$$

Where J is the path length (sum of weights along the path):

$$J^{i_{1:q}} = \sum_{k=1}^{q-1} c_{i_k, i_{k+1}} \quad (4)$$

The 2D map environments that this paper will implement motion planning on are 2D square grid maps that either contain free space or obstacles.

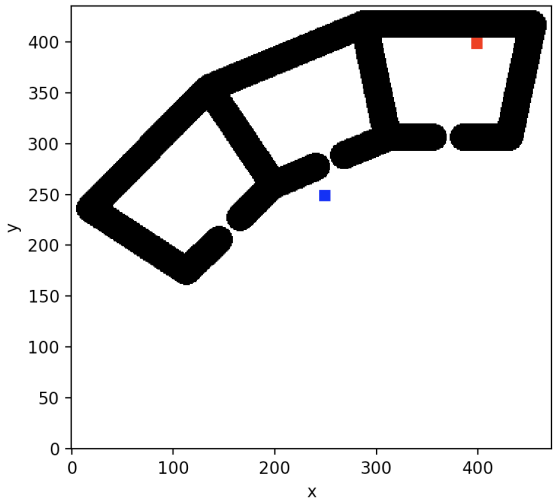

Fig. 1. Example of a map environment (Blue: robot, Red: target)

B. Problem Formulation: Problem Constraints

At any given time, there is also the robot position and target position, which are transient states. At each time step, the robot can plan and make a move, at which the target will respond by also making a move according to the mini-max decision rule.

$$\overline{v_i} = \min_{a_{-i}} \max_{a_i} v_i(a_i, a_{-i}) \tag{5}$$

Where v is the value (distance in this case), a_i is the target's action, and a_{-i} is the robot's action. The Minimax decision rule in summary is the smallest cost the robot can force the player to receive. Intuitively, marginalizing away a_i from $v_i(a_i, a_{-i})$ by maximizing over a_i, leaving only dependence on a_{-i} and then minimizing over a_{-i}. In practice, this gets the target to move away from the robot as the robot approaches the target.

The constraint with respect to the moving target is that the robot has 2 seconds to plan before making a move, and if the planning took longer than 2 seconds, the robot is able to move more than 1 step. The task is to plan and make a move towards the goal at each time step before 2 seconds, and if failing the 2 second mark, at least eventually be able to reach the target. Intuitively, the robot can never be faster than the target because as soon as the robot moves, the target moves, but if the robot takes too long to plan, the robot can move multiple times. This is quite an open-ended task, and as will be discussed, methods optimizing for speed and accuracy will be discussed.

The maps are given as txt files with ones indicating obstacles and 0 indicating free space, separated by spaces and new lines. Grids can also be assumed to have walls (the edges), and optimal paths are assumed to exist. Robot and target starting positions are also given. The given template code also has a main loop which counts how many moves it took to catch the target and also the time elapsed per planning stage. Although there were 8 maps, (map0.txt - map8.txt), there are multiple starting positions for some of them as well.

At each time step, both the robot and the target can also only move to one of its neighbor grid cells that are free. In 2D, this means they can only move to its surrounding 8 cells since we allow diagonal moves.

C. Problem Formulation: State Space and Model

Due to the problem being formulated as a DSP and DFS problem, it can be viewed as a Markov Decision Process. However due to the simple nature of the problem, the MDP can be viewed in a much simpler manner due to the naivety of the motion model and spatial constraints.

The state space $\mathcal{X}$ can be defined as the 2-D Cartesian coordinates (x, y), of the free cells in the 2D grid. The motion model $\surd^f$ as such, will be simply the deterministic control decision to go to the 8 neighboring cells if they are free. The time horizon T is finite as there are finite states in these maps and so the search will be finite given a worst case exhaustive search. The terminal cost q will be 0 at the goal and infinite every else as the goal is the only absorbing Markov state. The discount factor γ will be simply 1 due to the finite case and the stage cost l will be defined in the technical approach.

The terminal absorbing state given a complete search, is defined as being within L_1 norm of ≤ 1 between the target and the goal.

III. Technical Approach

A. Technical Approach: Introduction

To solve the problem formulation described above, this paper examines the weighted A^* algorithm, Anytime Repairing A^* (ARA^*), and the Rapidly Exploring Random Tree (RRT) Algorithm.

To better examine each algorithm, a section will be dedicated to each, which will describe them in detail, discuss their implementations

Key ideas will be presented about what enables the planner and algorithms to scale to larger maps within an allotted time constraint using search-based planning and sampling-based planning.

B. Technical Approach: A^* Intro and Heuristics

The A^* algorithm is a search-based motion planning algorithm that uses a heuristic function to strengthen the search. A heuristic can be thought of as a shortcut or something to rely on to solve a problem when there are no exact solutions for it or the time to obtain the solution is too long. Compared to naive planning algorithms such as the Dynamic Programming Algorithm and the Label Correcting Algorithm, the heuristic function uses special knowledge about the problem to obtain a faster search.

In the case of the A^* algorithm, the heuristic function must be admissible and consistent to work properly and are defined as follows respectively:

$$h_i \leq dist(i, \tau), \forall i \in \mathcal{V} \tag{6}$$

$$h_\tau = 0 \text{ and } h_i \leq c_{ij} + h_j, \forall i \neq \tau \text{ and } j \in \text{Children}(i) \tag{7}$$

Where h is the heuristic function, i is the current node, τ is the goal, $\mathcal{V}$ is the set of vertices/nodes, $dist$ is the distance/cost function and c_{ij} is the cost of transitioning from node i to node j.

Intuitively, this means the heuristic must be less than the true distance and the true distance to the goal and the heuristic should be always less than or equal to the estimated distance from any neighbouring vertex to the goal, plus the cost of reaching that neighbour.

A heuristic can also be ϵ consistent which is defined as:

$$\mathbf{h}_\tau = 0 \text{ and } h_i \leq \epsilon c_{ij} + h_j, \forall i \neq \tau \text{ and } j \in \text{Children}(i), \epsilon \tag{8}$$

Where $\epsilon \geq 1$ with $\epsilon = 1$ meaning the heuristic is consistent or the path is optimal. Having a higher ϵ means one is placing more reliance on the heuristic, trading a faster convergence for less optimally.

With the heuristic function defined, the Weighted A^* algorithm is as follows:

Algorithm 1 Weighted A^* Algorithm

Input: vertices $\mathcal{V}$, start $s \in \mathcal{V}$, goal $\tau \in V$, and costs c_{ij} for $i, j \in \mathcal{V}$

OPEN $\leftarrow$ {s} , CLOSED $\leftarrow$ {}, $\epsilon \geq 1$
$g_s = 0, g_i = \infty, \forall i \in \mathcal{V} \neq s$
while $\tau \notin$ CLOSED **do**
    Pop 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**
        if $g_j > g_i + c_{ij}$ **then** ▷ Node Expansion
            $g_j \leftarrow g_i + c_{ij}$
            $Parent(j) \leftarrow i$
            if $j \in$ OPEN **then**
                Update priority of j
            if Otherwise **then**
                OPEN $\leftarrow$ OPEN $\bigcup j$

Where $\mathcal{V}$ is the set or vertices, s is the start node $\in \mathcal{V}$, g is defined as the label, storing the lowest cost discovered so far from s to each visited node $i \in \mathcal{V}$. (similar to the one in the Label Correcting Algorithm), h is the heuristic function, ϵ is the weight and f is just another variable for storing $g + h$.

$OPEN$ and $CLOSED$ are used for Node Expansion and checking if nodes have been expanded. Once the goal is in the $CLOSED$ set, the algorithm terminates. In the node expansion phase, the children nodes of i are simply the up to 8 free neighboring grid cells and we update their labels if they are greater than the parent node + cost of transitioning.

If the child node is already in the $OPEN$ priority queue (More on implementation later), the value gets updated in $OPEN$. Otherwise, after updating its value, it gets sent into $OPEN$, ready for its eventual expansion unless we terminate at the goal.

C. Technical Approach: A^* Implementation, Heuristic Choice

Implementation of the A^* star in an optimal way consists of using efficient data structures for the computation of the algorithm. Similar to Dijkstra's algorithm, the $OPEN$ set is modeled as a priority queue, specifically a min heap since we want to pop items with the smallest f value. However due to the possibility of updating the priority of a node if its already in $OPEN$, an indexed priority queue, or one that allows efficient update of priority is preferred. For this, Python's pqdict package is used, which is simply a priority queue that works like a dictionary. In implementation, the node is the key and the value of f is the value.

For $CLOSED$, the data structure used will simply be a Python hash set, allowing fast insertion and checking if a value exists in the set. The labels g are stored in a dictionary and are dynamically allocated in memory instead of initiated to infinity for space optimization. Therefore if a node is nonexistent in g, we assume it is infinity.

The cost c_{ij} and the heuristic h are computed when needed, and are simply chosen to be the L_2 norm or euclidean distance of the states.

The $L-2$ norm is an admissible heuristic because even in the absence of obstacles, it will be the same as the true distance, therefore h satisfies equation (6).

It is also consistent when ϵ is 1 because the L_2 distance from a node to the goal will always be less than the L_2 distance of going to a new node and then going to the goal. (In the worst case equation (7) is equality when the new node is already in the path).

In code, A^* is implemented as a function Astar(), taking inputs: map, robot position, target position, and ϵ, which returns the next valid move.

D. Technical Approach: A^* Constrained Solution

For solving the constrained problem as described in the problem formulation (2 seconds max of planning time before target moves), A^* is run at each time step. After a path $i^*_{i_1:q}$ is returned, the next step after the starting position of the path (second index) of the path is returned. In the main loop, the robot position gets updated, the target's position gets updated, and the process is repeated until the L_1 norm between the robot and target is ≤ 1.

As one will notice, this naive implementation will have some flaws, but its bottlenecks and performance will be discussed in the Results section.

In the main.py file, one can choose to call A^* by modifying the $algo$ variable to $algo = 1$. One can also choose the map by changing the function $test_map()$. In the main function's loop (where each new robot move is an iteration), a function $RobotPlanner()$ simply calls the A^* function and returns its value back to the main loop. The rest of the code checks for valid moves and also sees if the robot reaches the target.

E. Technical Approach: ARA^* Intro and Definition

Similar to the A^* algorithm, The Anytime Repairing A^*, (ARA^*) algorithm is a also a search-based algorithm, but it is also classified as an anytime search algorithm, which is defined as an algorithm that returns the best plan possible within a fixed planning time.

This particularly characteristic of the ARA^* makes it an attractive choice, since solving the constrained problem of the 2 seconds planning time is objectively similar to an anytime search algorithm. The idea is to run a series of weighted A^* searches while decreasing the weight ϵ. Though this may seem inefficient at first glance, the repairing part of the ARA^* allows the reusing of results/labels from previous searches, effectively reducing the number of expansions and increasing the speed of the search. If ϵ decreases to 1 before the finite planning time constraint, the path is already optimal and can be returned.

The way ARA^* is able to reuse labels from previous searches is by marking nodes whose g values hace changed

since the last expansion as inconsistent. A consistent node is defined as a node such that:

$$v_i = g_i \tag{9}$$

Where v is the value of the node since its last expansion and g is the current value of the node in the current expansion. Naturally, this means $v_i = \infty$ for nodes that were never expanded.

An overconsistent node is defined as a node such that:

$$v_i > g_i \tag{10}$$

And a underconsistent node is simply the reverse. However underconsistency of nodes will never exist for this problem due to only having positive costs. Using a ϵ -consistent heuristic for anytime, and keeping track of inconsistency, the ARA^* algorithm is as follows:

Algorithm 2 Anytime Repairing A^* (ARA^*) Algorithm

Input: vertices $\mathcal{V}$, start $s \in \mathcal{V}$, previous labels v

Initialize ϵ to large value
OPEN $\leftarrow \{s\}$
$g_s = 0, g_i = \infty, \forall i \in \mathcal{V} \neq s$
$v_i = \infty, \forall i \in \mathcal{V}$
while $\epsilon \geq 1$ **do**
    CLOSED $\leftarrow \{\}$, INCONS $\leftarrow \{\}$
    OPEN, INCONS $\leftarrow ComputePath()$
    Publish Current ϵ-subotimal Path
    Decrease ϵ
    OPEN = OPEN $\bigcup$ INCONS

$ComputePath()$ defined in the next algorithm.

Where Compute Path is A^* that keeps track of inconsistent nodes and is defined as follows:

Algorithm 3 ComputePath

Input: vertices $\mathcal{V}$, start $s \in \mathcal{V}$, goal $\tau \in V$, cost c_{ij} for $i, j \in \mathcal{V}$

function COMPUTEPATH
    while $f_\tau > \min_{i \in OPEN} f_i$ **do**
        Pop 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**
            if $g_j > g_i + c_{ij}$ **then**
                $g_j \leftarrow g_i + c_{ij}$
                if $j \notin CLOSED$ **then** $OPEN \leftarrow j$
                if Else **then** $INCONS \leftarrow j$

Where $INCONS$ is the inconsistent set of nodes, and are union-ed with the OPEN set after each decreasing ϵ iteration.

F. Technical Approach: ARA^* Implementation

Though very similar to the A^* algorithm in code implementation, several new variables are introduced which are $INCONS$ and v. For $INCONS$, the data structure used is also an indexed min-heap and v in implementation does not need to be used in any sense as $INCONS$ is sufficient. All other variables and functions used the same data structures as in A^*.

However, due to persistent information being stored across states ($INCONS$ and g), object oriented programming was employed to more easily implement ARA^*. In code, ARA^* is a class which contains data members as fields, and both an $ARAstar()$ function and a $ComputePath()$ function.

The same heuristic as A^* was chosen for ARA^* which is the euclidean distance for convenience and for the reasons discussed in the A^* section.

In the main call of ARA^*, an object of the ARA^* is initialized and passed along with $algo$ into $RobotPlaner$, which calls the search. The next valid move is appropriately returned.

G. Technical Approach: ARA^* Constrained Solution

To solve the constrained problem, ARA^* effectively returns the most optimal path given a finite planning time. The paths that are returned in theory should be more optimal than a naive A^* approach as ARA^* also provides a bound on the sub-optimality.

Due to two seconds being a very long planning time, (especially for large maps), one implementation detail that speeds up ARA^* significantly, is reusing labels even across the main loop. This idea however deviates from what the ARA^* algorithm is supposed to do as it does not hold any robustness to assumptions around dynamic systems such as a moving target or changing costs.

Nonetheless, the idea was tested and yielded interesting results that will be discussed in the results section. This idea is also made possible due to the object oriented implementation as the object and its data can be stored in memory rather than being scoped inside a function.

H. Technical Approach: RRT, Sampling VS Search-based

Last but not least, we have the Rapidly Exploring Random Tree Algorithm (RRT). RRT is an algorithm for sampling based motion planning, which in nature is very different compared to the search-based notion of the first two algorithms introduced.

Sampling-based approaches to motion planning generally build a sparse sample-based graph instead of searching the entire state space right from the start. Graph construction and searching can be done interweavingly as needed, and it provides asymptotic suboptimality bounds on the solution.

Advantages to search-based planning include speed and space efficiency for high dimensional domains. In the case for some challenge maps in the problem, a sampling based approach could outperform search-based.

For search-based algorithms, several new definitions are needed. A configuration space C is simply all the states in the state space $\mathcal{X}$, defined for a MDP but it also includes the obstacles $C_{obs} \subset C$ and $C_{free} \subset C$ is the valid state space. A feasible path:

$$\rho : [0, 1] \rightarrow C_{free} \tag{11}$$

is a continuous function such that $\rho(0) = x_s$ and $\rho(1) = x_\tau$ where x is the state, s is the start and τ is the goal. Sampling based algorithms also uses several primitive procedures including: SAMPLE(), which returns independent and identically distributed samples from C, SAMPLEFREE() the same for C_{obs}, NEAREST() which given a graph and a point, returns the closest node in the graph, NEAR() which returns all nodes in a graph within r radius of a point, STEER() which given two points, adds a new point to the graph along their trajectory, and COLLISIONFREE(), which checks for obstacles along a path given 2 points.

All these functions will be used in the implementation of RRT.

I. Technical Approach: RRT, Intro and Definitions

not to be confused with RRT^*, RRT does not search the graph it is creating natively. Instead, RRT constructs a tree/graph from random samples starting at root x_s and grows until it is able to reach the goal x_τ.

RRT is a good algorithm to use for this problem because the tree is sparse and therefore much smaller than the entire search space. This allows for various search algorithms to perform on a much smaller state space and hence converge to the target much faster.

Additionally, the saved tree can also be reused even when the goal moves, and if the goal moves a little bit, the tree is usually able to still connect the goal to the tree, and if not just do more sampling.

The RRT algorithm is defined as follows:

Algorithm 4 Rapidly Exploring Random Tree (RRT)

```
V ← {x_s}; E ← ∅
for i = 1, ··· n do
    x_rand ← SAMPLEFREE()
    x_nearest ← NEAREST((V, E), x_rand)
    x_rand ← SAMPLEFREE()
    if COLLISIONFREE(x_nearest, x_new) then
        V ← V ⋃ {x_new}
        E ⋃ {(x_nearest, x_new)}
return G = (V,E)
```

Where V represents a vertex and E represents an edge in the graph G respectively. As one can see, this algorithm only returns the graph/tree and not an optimal path.

For an optimal path, the vanilla A^* algorithm will be used but instead of searching in C_{free}, A^* will search in the graph.

J. Technical Approach: RRT, Implementation Details

Similar to ARA^*, due to persistent data that can be reused in the main loop, object oriented programming is used at a high level, where RRT is a class. For the algorithm itself, the graph/tree is implemented as an adjacency list, where for simplicity is a dictionary of sets of tuples. The reason a dictionary was used is so each node as a tuple can be easily accessed via its state as the key. The value is a set instead of a list to prevent duplicates and redundant computation. As such, each edge/node is essentially a vertex/leaf.

C, C_{free} are implemented as hash sets for quick insertion and checking of states, but a copy of them as lists are also present in the class for ease of random sampling (cannot get random item from sets).

For the primitive procedures, the sampling methods SAMPLE and SAMPLEFREE make use of the Random library and sample from the list twin of C_{free}. NEAREST and NEAR both just iterate through all nodes in the graph until the nearest is found or near enough are found respectively. This is linear in time and can be sub-optimal but since a new node can be closer to any vertex, there is no better conceivable way to do this.

For STEER, the Bresenham 2D ray-tracing algorithm is used to find all points in the graph between two points. After having all the points, STEER iterates through the points until it is $\geq \epsilon$ distance away from the start and returns that point. As such, ϵ is a parameter for RRT and represents how far to steer into a valid trajectory and add a node to the graph.

As such ϵ in RRT can be tuned by the user for different sizes of maps and environments. Though there are ways to optimize this parameter, it will simply be the pre-selected for now. Another parameter for RRT is the frequency in which to check for the goal. By default, RRT is set to check whether the goal is COLLISIONFREE() from any node in the graph every 100 samples. This again can be fine-tuned to a lower value for speed or a higher value for more sampling.

For actual searching of the graph, A^* is used and is slightly modified to search an adjacency list rather than a 2D grid. As such the resulting path returned will be discontinuous but collision free. To actually return the next move, Bresenham is run between the start node and the next vertex, and the actual node returned is the node after the start node in the output of Bresenham.

K. Technical Approach: RRT, Constrained Solution

For the constrained solution, RRT utilizes the persistence of the graph to enable fast searches. Though the first time the goal is seen from the graph's nodes can take a long planning time due to random sampling in big or complex maps, due to the goal only moving to one of its 8 neighbors, subsequent searches are generally very quick.

Once again the RRT algorithm is selected in main by setting $algo = 3$ and is initialized as ab object before the main loop before being called in $RobotPlanner()$. A^* is being run on the graph every time the goal is within ϵ distance from the closest node.

For speed optimization, one variant and technique used is to always go to the node returned from A^* rather than sampling more and running Bresenham more. However long the edge, the robot will just make its way to that node. This path is stored in a data member list called $stored_path$, which pops items as the robot moves through it. Until $stored_path$ is empty, the robot will continue on that path before sampling all over again.

This allows for a much faster algorithm although the paths will be less optimal and will be discussed in results.

IV. Results

Below are images of all maps and configurations from all three algorithms discussed.

A. Results: Interesting Details of Implementation

To collect the results, all algorithms were run on all maps.

For A^*, the algorithm was simply run on each map, and no extra parameters were tuned. ϵ was kept to 1 for all maps, meaning the path returned is the most optimal. This however meant significant trade-offs in speed. In fact, for map 1b, A^* took 4 seconds per planning time and it also completely hung for map 7. Towards the end of the goal

For ARA^*, the initial ϵ was set to 32 and divided by 2 each repetition. 32 was chosen because it is a power of 2, enabling epsilon to reach 1 exactly. Additionally, as briefly mentioned in the Technical Approach, ARA^* 2 variations of ARA^* were implemented. Both reuse labels within one planning time, but the special one also reuses labels across planning times, even after the goal moves. The second approach is faster, but it can sometimes lead to cases where the path is much more sub-optimal. This is again likely due to ARA^* not being an incremental algorithm like D^*.

For RTT, its standard version was implemented with all nodes in the path from bresenham being added to the graph, but a second variation that sacrifices optimality of the path for optimality in search speed was also implemented. As mentioned in the technical approach, the second variation does not recompute A^* or sample again until all points returned by Bresenham for the optimal path in the graph returned from A^* are traversed. This second variation as expected performed faster but the paths were less optimal.

In addition to RRT, I initially gravitated more towards RRT^* since it could also search while building the tree. To test RRT^*, I modified the (github.com/motion-planning/rrt-algorithms) repository and tested it on several maps. An issue I faced with this repository's implementation however, was that obstacles had to be defined as blocks rather than points. As such I simply expanded them to be one by one squares rather than points which changes the problem slightly. This wasn't a big issue however since most maps did not have a one point wide path.

B. Results: Performance Comparison

Though briefly mentioned in the Interesting Details section, performance of the three algorithms will be further compared in this subsection against different map environments, and with respect to the problem constraint.

For A^* running on a weight of $\epsilon = 1$, the speed is very acceptable for the small and even mid-sized maps in individual searches. For the problem constraint. It's lack of reusing information across planning causes it to be quite slow for the mid and large sized maps. However if the algorithm does terminate, the path is definitely the most optimal path in terms of cost as shown in all the images.

For ARA^*, similar to A^*, its default version shares the weakness of having to lose information across planning horizons. However the default version's main advantage, and why it was chosen to be implemented was because of the problem constraint. Given a 2 second timing horizon, ARA^* almost always returns a path. This means the robot is at least always going to move towards the target albeit ϵ-suboptimal. For the common case where ARA^* finishes before the 2 second mark, its speed is comparable to A^*, so it can be viewed as a direct upgrade to the naive A^*, (Atleast for this problem).

For the fast variation, even more suboptimality in the path is introduced in exchange for much greater speed for this problem, though for individual searches, is directly equivalent to the default implementation.

For RRT^*, the algorithm is incredibly fast in finding paths in small maps and mid-sized maps, but for larger and giant maps, the initial search for the goal and building the tree takes slightly longer. However once the first tree is built, subsequent samples often do not take long to find the goal and running A^* on the adjacency list is very quick, and with the second variation, is even quicker albeit slightly sub-optimal. Once major concern of the second variation is that sometimes the robot will backtrack along the tree, sometimes leading to a cat and mouse game with the target until new samples allow it to be eventually caught.

With respect to the 2 second planning time constraint before the goal moves, often times, the first sample will take less than 2 seconds for the small and mid-sized maps. For large maps like 1 and 7, it can take much longer, enough for the target to make quite a lot of moves before the robot even makes its first.

Graphs below show the speed optimality performance of each algorithm running on a M1 Mac. For RRT* the paths weren't deterministic so an average of 5 runs were taken. For map 3, RRT sometimes backtracked which lead to the second variation actually taking even more time than the default.

	Runtime (Default)	Runtime (Variation)
A^*	36.8170 s	N/A
ARA^*	16.9948 s	17.0649 s
$RTT + A^*$	20.0585 s	25.9373 s

Fig. 2. Comparison of speeds for map 3 (To catch the goal)

	# Moves (Default)	# Moves (Variation)
A*	0.949422	N/A
ARA*	288	288
RTT + A* 340	545	

Fig. 3. Comparison of moves made for map 3 (To catch the goal)

C. Results: What Worked and What Did Not

All implementations worked for the most part, but some bugs included backtracking for RRT. This is probably due to improper definition of the heuristic. For better results, the heuristic should be distance to the goal in terms of the edges rather than the 2D grid space. This was overlooked because the problem at hand uses a 2-D grid but the individual path search uses the tree. Mapping the problem back and forth was pretty difficult and up for improvement.

Also, no algorithms were able to successfully search map 7, not even for an individual search. For RRT even after 10 million samples, no path was found. Same story for the github modified RRT*. For the search based ones, the program just hangs. This is probably due to map 7 having 25,000,000 states in its configuration C. The snake path and many tiny obstacles also present a challenge.

D. Results: Images RRT

Below are images of running A^* on the graph returned from my own implementation of RRT on all maps. There is also one image of RRT^* run on map 3 using the recommended GitHub repository (github.com/motion-planning/rrt-algorithms).

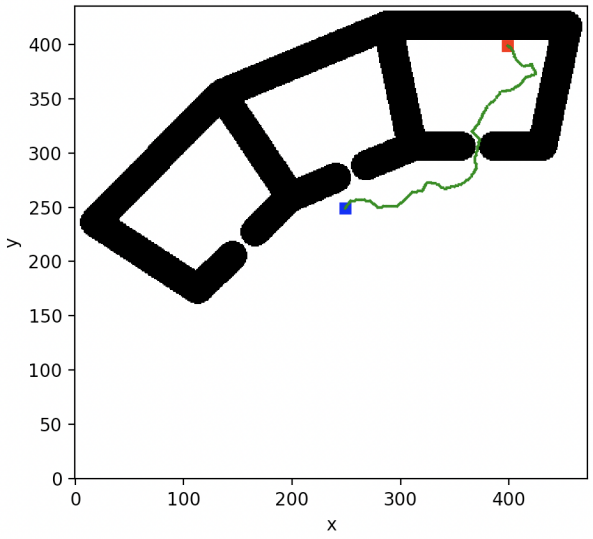

Fig. 4. RRT on map 3

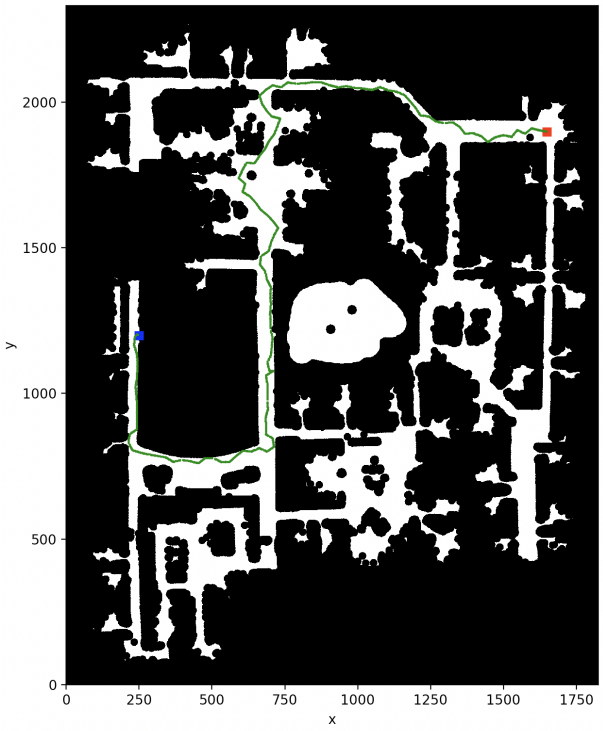

Fig. 5. RRT on map 1b (Challenge Map)

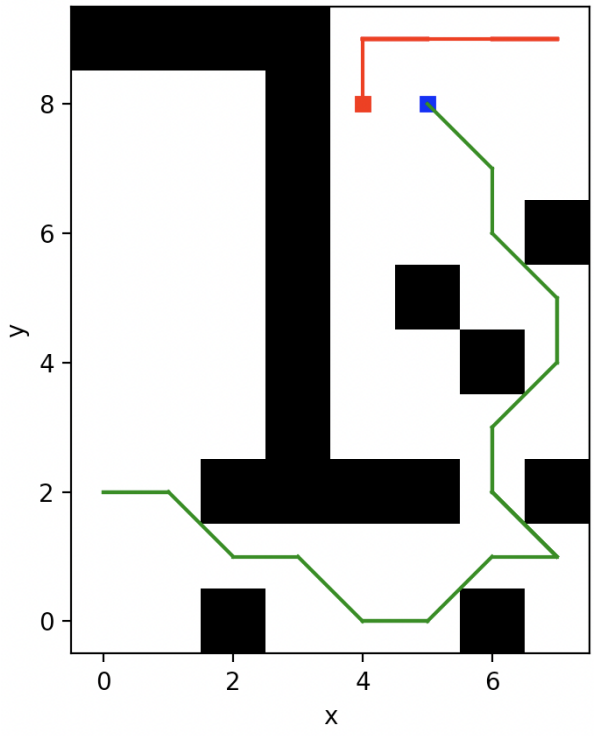

Fig. 6. RRT on map 2

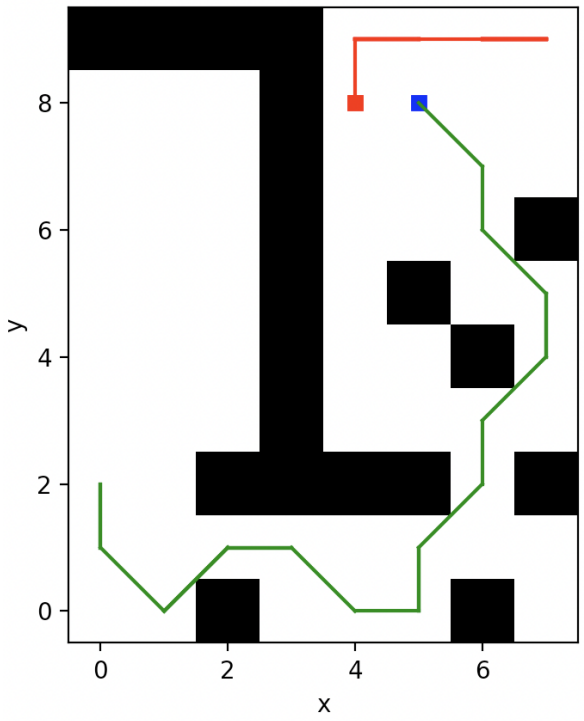

Fig. 7. RRT on map 2 (Different sample)

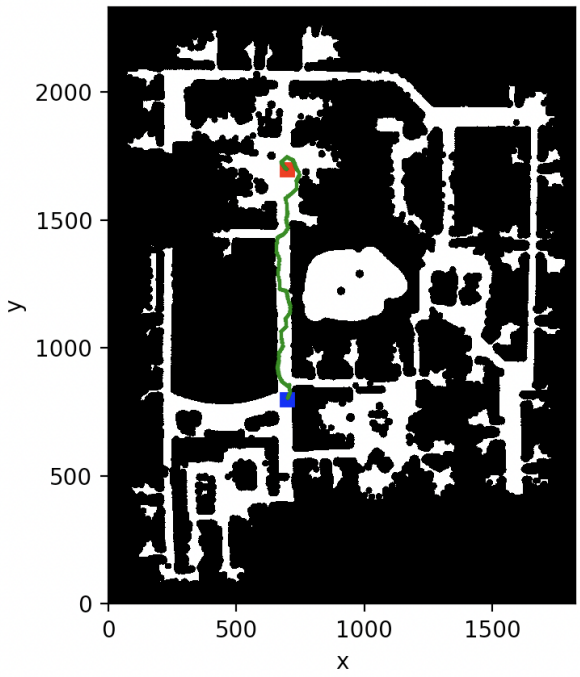

Fig. 8. RRT on map 1

Fig. 9. RRT on map 3b

Fig. 10. RRT on map 3c

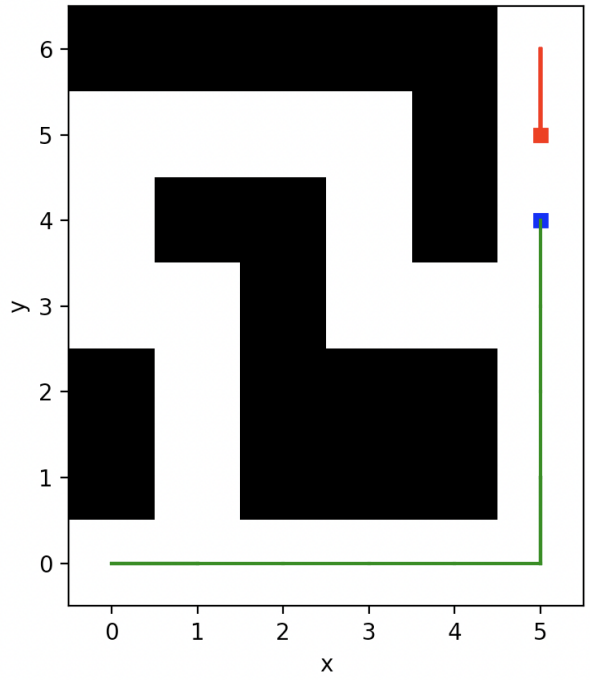

Fig. 11. RRT on map 4

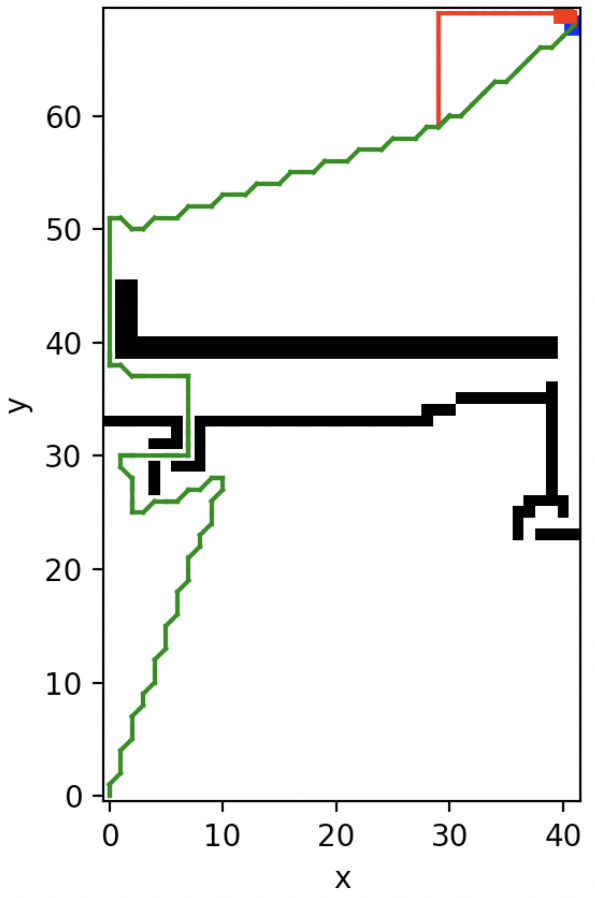

Fig. 12. RRT on map 5

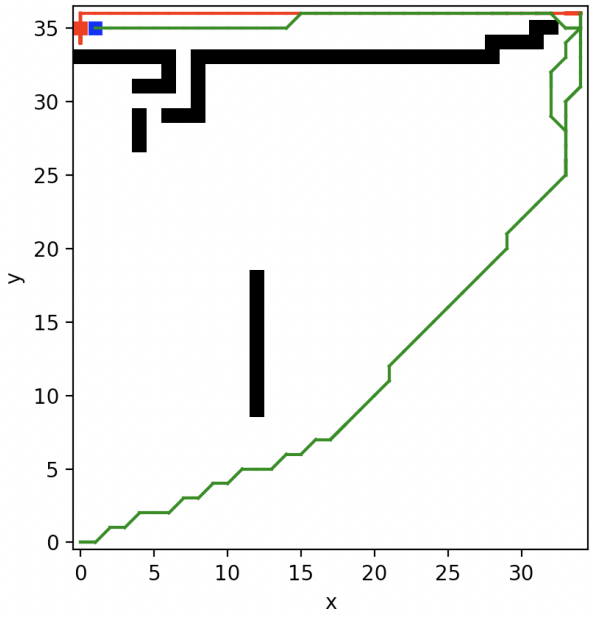

Fig. 13. RRT on map 6

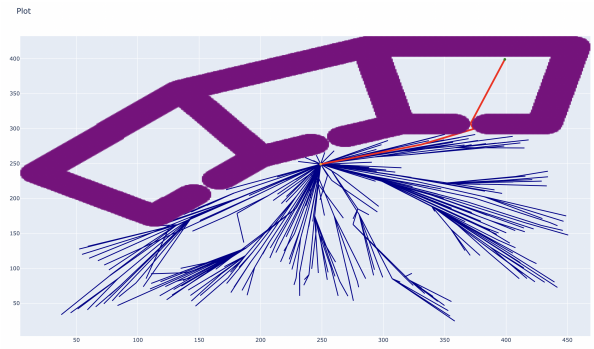

Fig. 14. RRT star (github.com/motion-planning/rrt-algorithms)

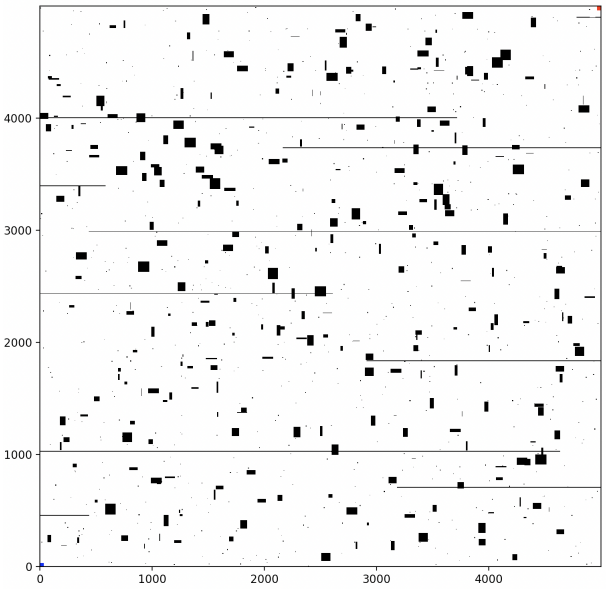

Fig. 15. Challenge Map 7 (Unsuccessful)

*E. Results: Images A**

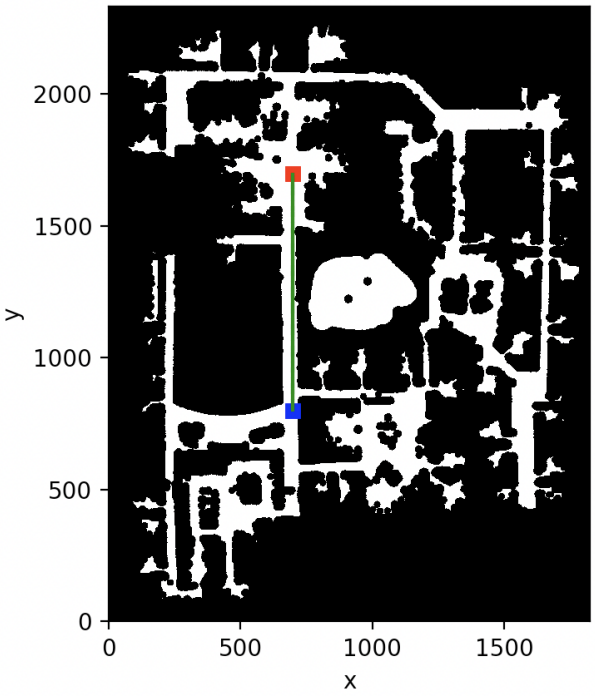

Fig. 16. A* on map 1

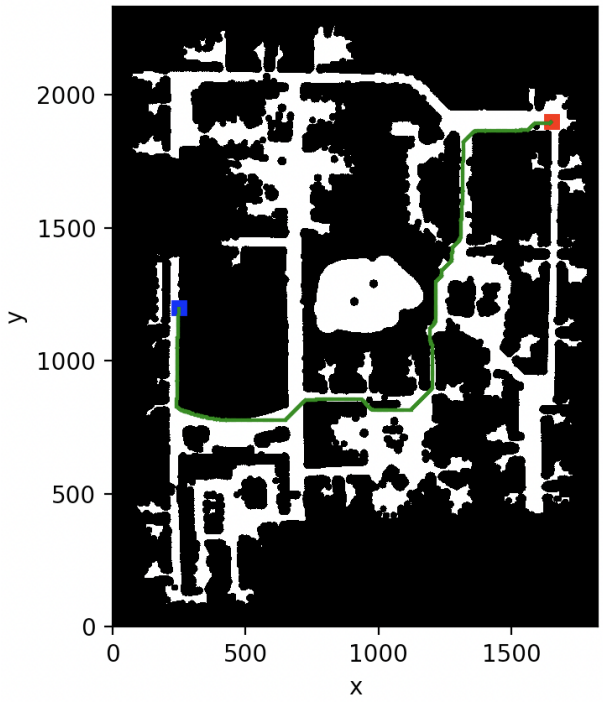

Fig. 17. A* on map 1b

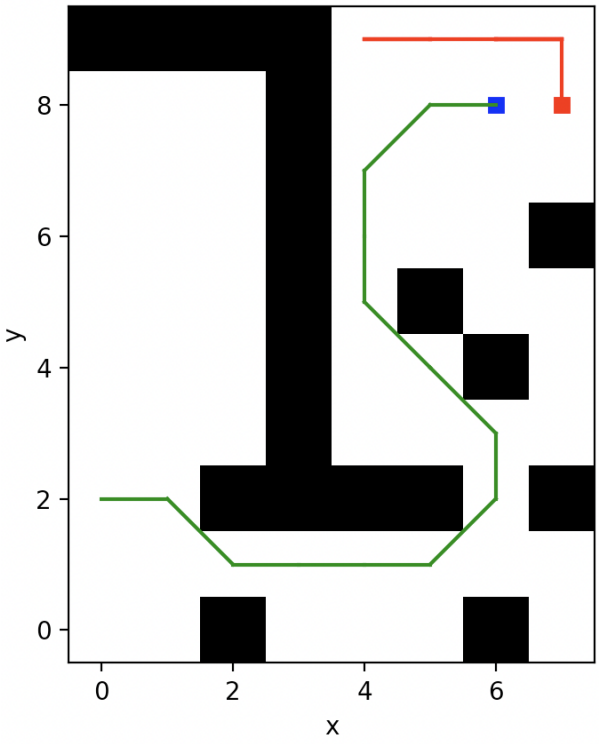

Fig. 18. A* on map 2

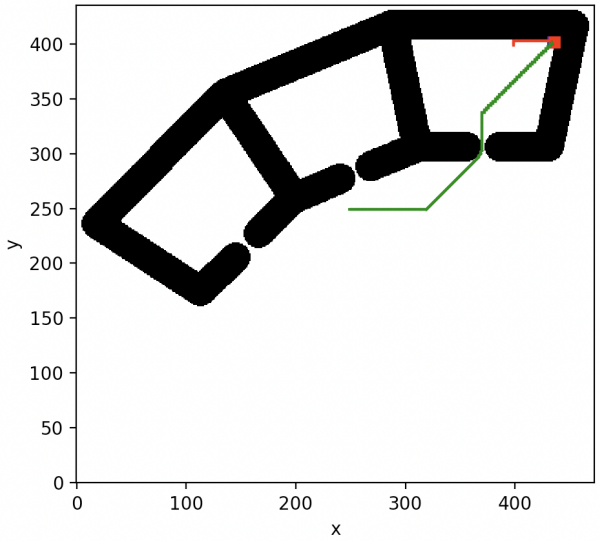

Fig. 19. A* on map 3

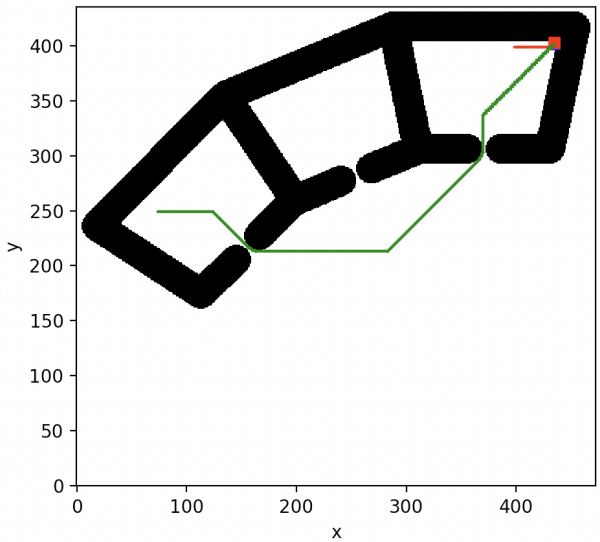

Fig. 20. A* on map 3b

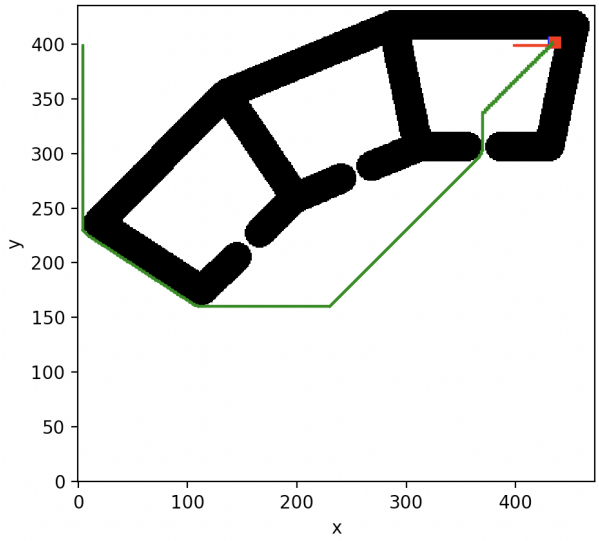

Fig. 21. A* on map 3c

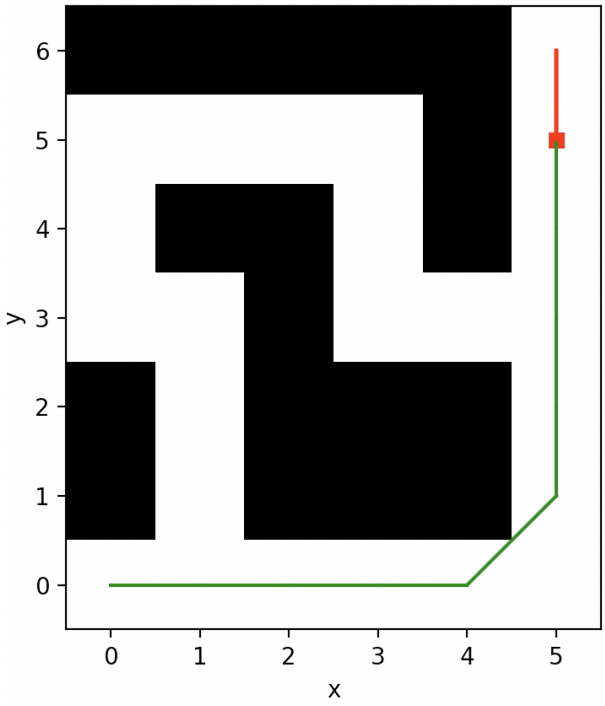

Fig. 22. A* on map 4

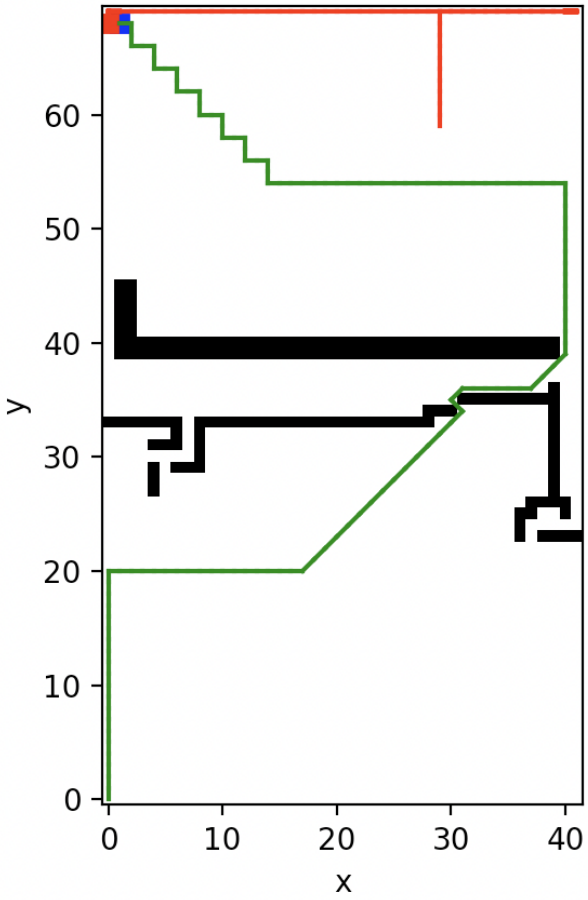

Fig. 23. A* on map 5

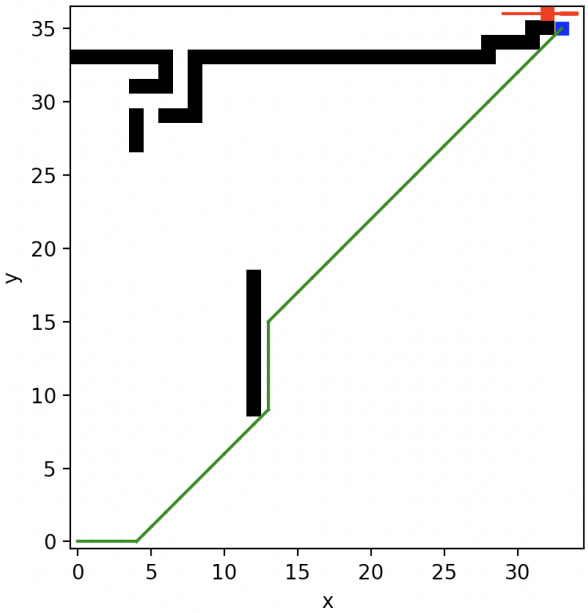

Fig. 24. A* on map 6

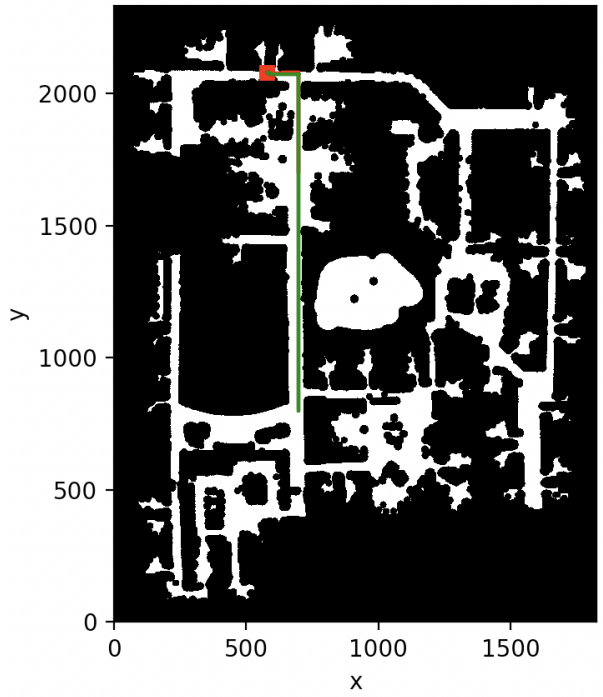

Fig. 25. ARA* on map 1

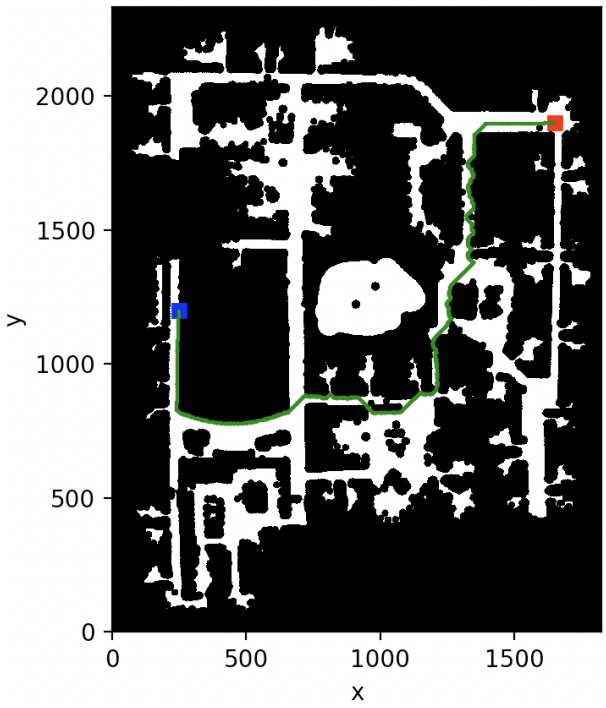

Fig. 26. ARA* on map 1b

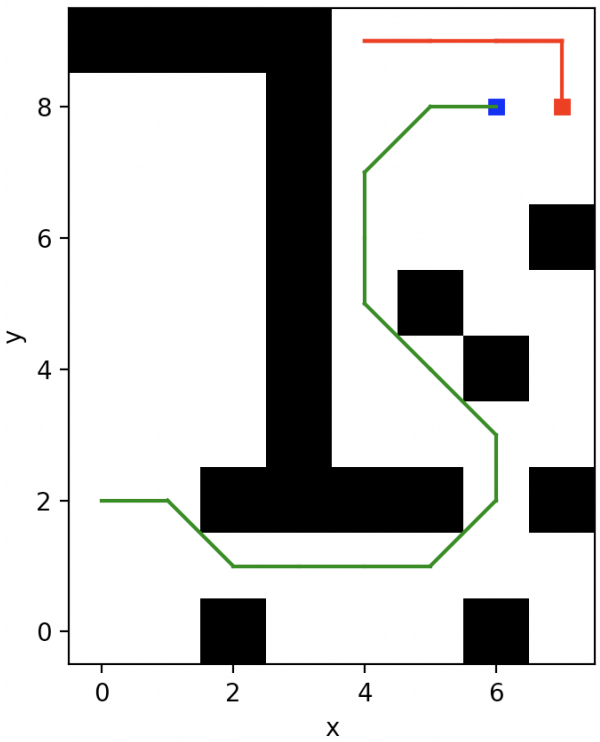

Fig. 27. ARA* on map 2

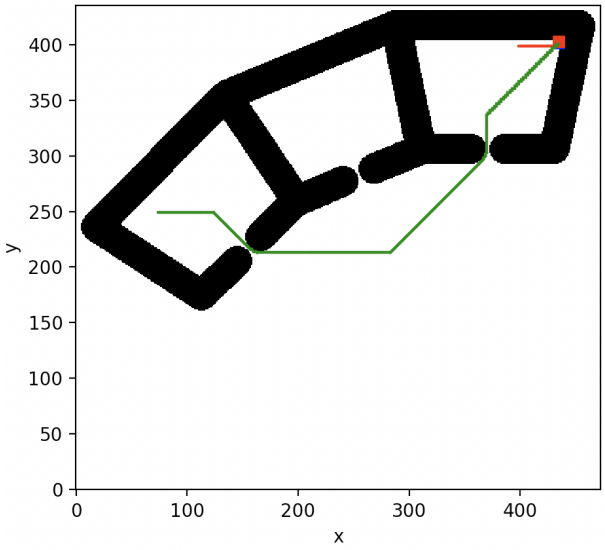

Fig. 29. ARA* on map 3b

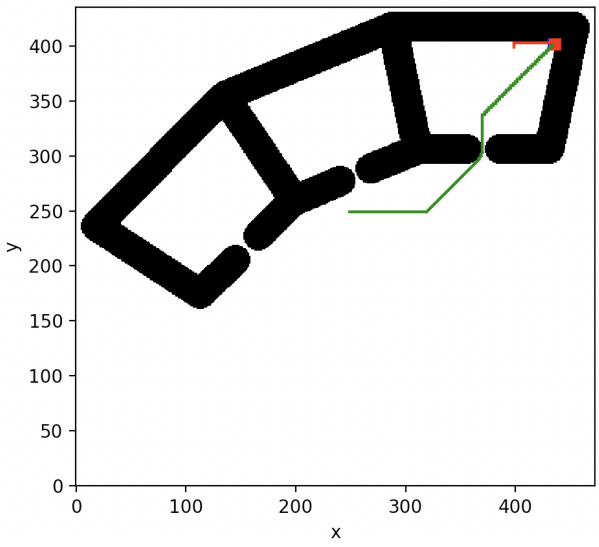

Fig. 28. ARA* on map 3

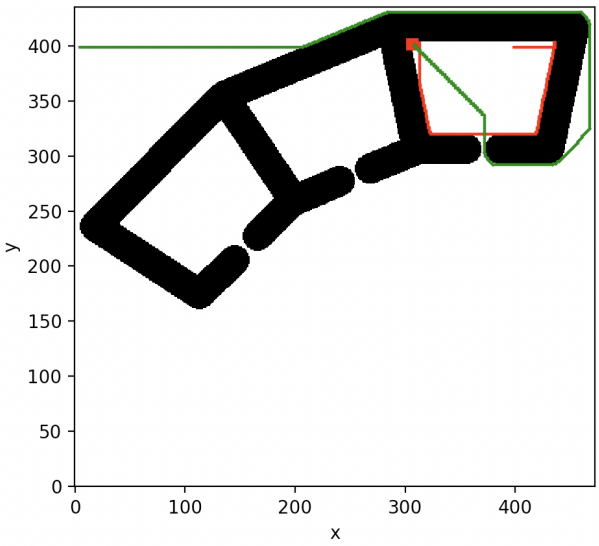

Fig. 30. ARA* on map 3c

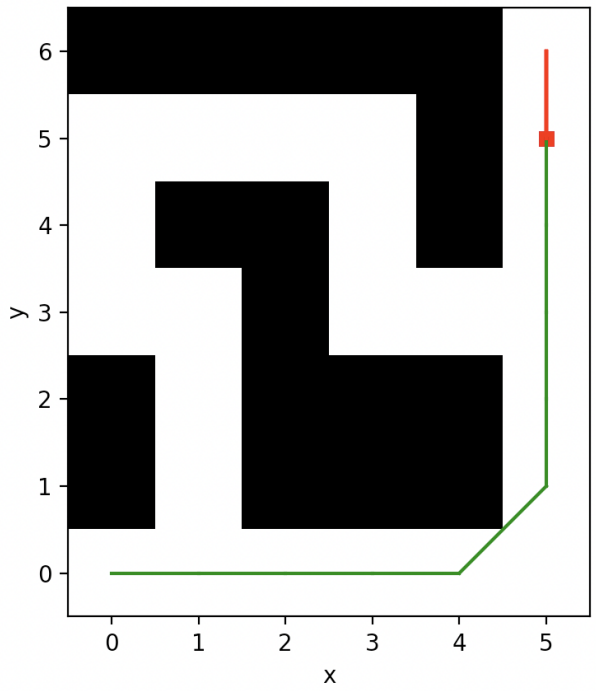

Fig. 31. ARA* on map 4

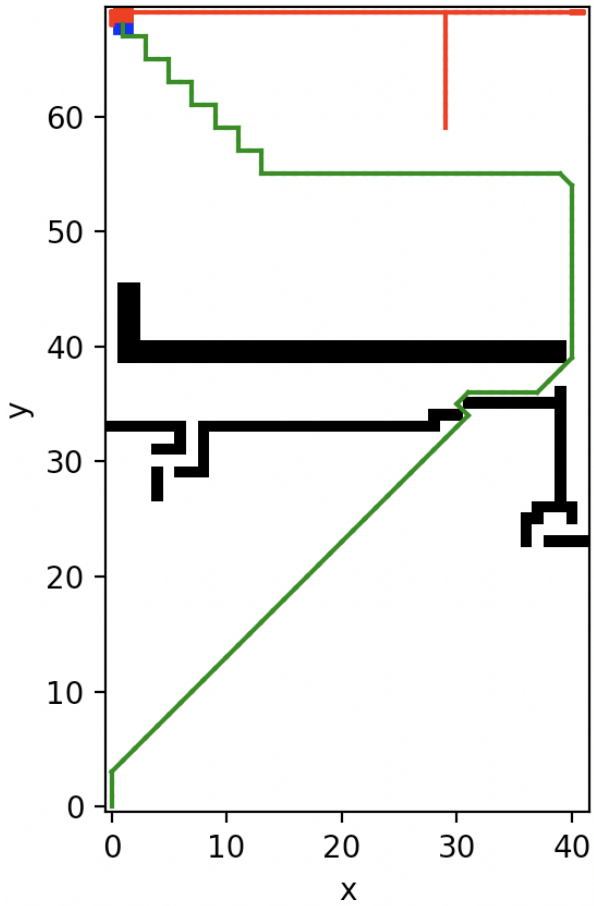

Fig. 32. ARA* on map 5

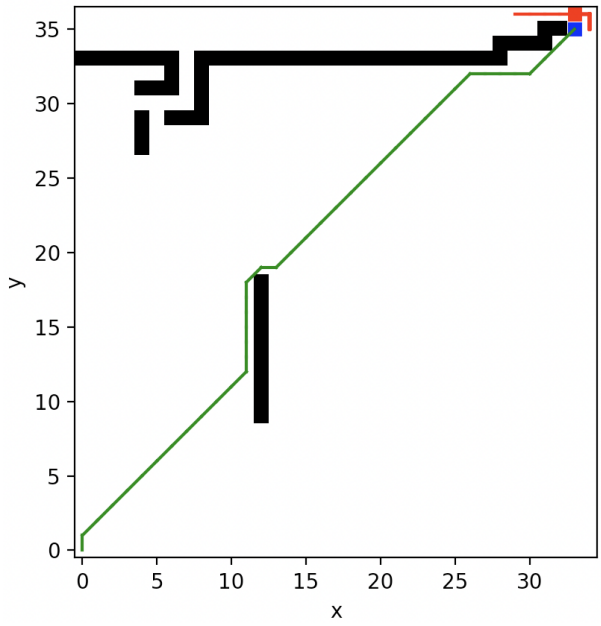

Fig. 33. ARA* on map 6

V. References

UCSD ECE276B class material and discussions with Jesus Fausto before implementation.