Motion Planning

Sambaran Ghosal
Department of Electrical and Computer Engineering
UC San Diego
La Jolla, USA
sghosal@ucsd.edu

Nikolay Atanasov
Department of Electrical and Computer Engineering
UC San Diego
La Jolla, USA
natanasov@ucsd.edu

***Abstract*—A very important problem in the real world robotics is to find the shortest path from a starting position to a goal position without hitting any obstacles present in a known or unknown environment. This problem is known as Motion Planning. Often it is also the case that instead of a stationary goal, we may have to catch a moving target and hence we need to plan the next move of the robot within a given time constraint and then run multiple planning instances to catch the target.**

***Index Terms*—Motion Planning, Search Based Motion Planning, Sampling Based Motion Planning, A*, Agent Centered Search, Anytime Search, RRT, RRTConnect**

I. Introduction

Often in real world robotics applications, a common problem is to get the robot from a start point to a goal point in the shortest path possible along with satisfying some constraints that may be provided by the environment(obstacles) or the robot kinematics and dynamics. This is one of the crucial problems in robotics and is often termed as Planning and Control. The problem of obtaining the shortest possible path from a start position to the goal position is termed as **Deterministic Shortest Path(DSP)** problem. The most common method to solve these kind of DSP problems is to apply a **Dynamic Programming(DP)** algorithm. There are many variants to the basic Dynamic Programming algorithm such as Label Correction, Backwards Dynamic programming, forward dynamic programming, Dijkstra's algorithm, A*, Rapidly Exploring Random Tree(RRT) and variants of these.

Motion Planning is a special type of Deterministic Shortest Path problem defined over continuous state space and control space in presence of obstacles in a known environment. Motion Planning problems require us to find a feasible and cost minimal path from an initial state to a goal region without hitting any obstacles. The cost function for the problem may be defined as distance travelled, time taken, energy consumption etc.

In this project, we want to solve a motion planning problem where there is a robot starting from a given position, and the objective is to catch a moving target. We implement variants of the **A*** algorithms such as **Anytime Search, Agent Centered Search** in Search based motion planning and **RRT, RRTConnect** algorithms in Sampling Based motion planning.

II. Problem Formulation

The **Motion Planning** problem is almost the same as the **Deterministic Shortest Path(DSP)** problem where we have

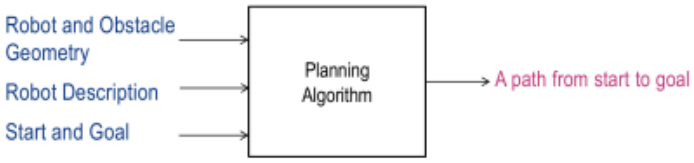

Fig. 1: The Motion Planning Problem

to find the shortest path starting from a initial position to the goal position. Although the underlying state space and control space in Motion Planning is generally continuous, solving motion planning in continuous space assumption using **Exact Algorithms** is often computationally expensive and unsuitable for high-dimensional spaces. Hence the state space and control space is again discretized into either regular grids or irregular states using sampling methods. In both cases, the motion planning problem can be formulated as follows : Given a graph with set of vertices $\mathcal{V}$ which represent the states the robot can have, edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ which represents possible transitions from states to states, and edge weights $\mathcal{C} = \{c_{ij} \in R \cup \infty | (i,j) \in \mathcal{V}\}$, where c_{ij} represents the cost of transition to go from a state/vertex i to j, the **objective** is to obtain the shortest path from a given start state s to a goal state τ. **Path** can be defined as the sequence of vertices from s to τ as $P_{s,\tau} := \{i_{1:q} | i_k \in \mathcal{V}, i_1 = s, i_q = \tau\}$. The path length associated with a path is defined as the sum of cost of transition from i to $j \in P_{s,\tau}$ as $J^{i_{1:q}} = \sum_{k=1}^{q-1} c_{i_k, i_{k+1}}$. Hence the objective can be formulated as finding the path from s to τ with the minimum path length.

$$dist(s,\tau) = \min_{i_{1:q} \in P_{s,\tau}} J^{i_{1:q}} \tag{1}$$

$$i^*_{1:q} = \underset{i_{1:q} \in P_{s,\tau}}{argmin}\, J^{i_{1:q}} \tag{2}$$

Here $i^*_{1:q}$ is the shortest path from s to tau with the minimum path length. We can also obtain the sequence of controls $u \in \mathcal{U}$, where $\mathcal{U}$ is the set of feasible controls, that can produce the shortest path sequence starting from the start s to the goal τ.

The start state s is the position in the map where the robot is situated at the beginning of each planning time. The goal state τ is the target position at that time.

We define our state space graph $\mathcal{X}$ to be a 8-connected graph which means that from a given state, we can execute 8 different motions. The vertex set is basically the set of all possible coordinates in the map. Hence the vertex set is described as

$$\mathcal{V} = \{(x, y)|x, y \in R^2, 0 \leq x < x_{max}, 0 \leq y < y_{max}\} \quad (3)$$

where x_{max}, y_{max} is described by the specific map and is the maximum possible x, y coordinates in the map.

The possible control actions in a 8-connected graph is moving up, down, left, right and diagonally up-right, up-left, down-left or down-right. We represent these actions as a vector as follows :

TABLE I: Control Space $\mathcal{U}$

u	Vector
UP	(0,1)
DOWN	(0,-1)
LEFT	(-1,0)
RIGHT	(1,0)
UP-LEFT	(-1,1)
UP-RIGHT	(1,1)
DOWN-LEFT	(-1,-1)
DOWN-RIGHT	(1,-1)

Hence, the control space $\mathcal{U}$ is the set of all these vectors.

The edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the connection between nodes. Given that we describe our state space as a 8-connected graph, $\mathcal{E}$ consists of transitions from parent node to nodes that are either left, right, up, down or diagonally up-left, up-right, down-left and down-right to the current node.

$$\mathcal{E} = \{x_i \implies x_j \, \forall x_i, x_j \in \mathcal{V} \, | \, \exists u \in \mathcal{U} \, \text{s.t.} f(x_i, u) = x_j\} \quad (4)$$

The motion model can hence be described as follows :

$$x' = f(x, u) = x + u \quad (5)$$

$\forall u \in \mathcal{U}$. The map is given as a 2D grid with 0 and 1's. A given position is a free space if the value of the map at that position is a 0, whereas the position is an obstacle if the map value at this position is a 1. If we denote the map as a matrix **envmap** $\in R^{(x_{max}-1)\times(y_{max}-1)}$, then

$$C_{free} = \{(x, y) \in \text{envmap} \mid \text{envmap[x,y] = 0}\} \quad (6)$$

and

$$C_{obs} = \{(x, y) \in \text{envmap} \mid \text{envmap[x,y] = 1}\} \quad (7)$$

where C_{free} is the freespace and C_{obs} is the obstacle space of the map. Additionally we defined a space C_{out} as

$$C_{out} = \{(x, y)|x, y \in R^2; x < 0 \cup x > x_{max}, y < 0 \cup y > y_{max}\} \quad (8)$$

which basically describes points outside the map. The corresponding cost function can be described as follows :

$$c(x_i, x_j) = l(x_i, u) := \begin{cases} ||u||_2 & x_j \in C_{free} \\ \infty & x_j \in C_{obs} \cup C_{out} \end{cases} \quad (9)$$

where $u \in \mathcal{U}$ that results in the transition from x_i to x_j i.e. $u \in \mathcal{U}$ s.t. $f(x_i, u) = x_j$, and $||u||_2$ represents the L2 norm of the control. So effectively our cost is the distance travelled by the robot if the next state transitioned to is a feasible space i.e. not an obstacle or not outside the map. Otherwise the cost is ∞ if the state transitioned to is an obstacle or outside the map.

In addition to this, our problem includes a smart target, where the target moves towards the direction that maximizes the distance between itself and the robot. If the robot takes more than 2 seconds to make its next move, the target moves multiple steps according to the **MinMax** rule to maximize the distance from the robot. Hence we need to plan the next move of the robot within 2 seconds so that the target does not move too far away.

We will implement some Search Based and Sampling Based motion planning algorithms on the given maps and compare them based on their performance.

III. Technical Approach

A. Setup

Before we can implement any of the Search Based or Sampling Based motion planning algorithms, we need to construct crucial elements of the motion planning problem which includes setting up the connectivity of the graph, which decides the states accessible from a given parent state, the corresponding cost of the transitions from one state to another.

We also require a **heuristic** function to use for A* because in high-dimensional motion planning, using normal Dijkstra's algorithm would expand a lot of nodes that are not potential members of the shortest path from the start to the goal. A heuristic function helps A* bias its search of nodes close to the goals and hence the optimal path is obtained by expanding a lot lesser number of nodes.

A heuristic function needs to be **admissible** and **consistent**. Heuristic function is the measure of distance from the current node to the goal node. A heuristic function is admissible if it is an underestimate of the shortest distance from the node to the goal i.e. $h(x, \tau) \leq dist(x, \tau)$ where $dist(x, \tau)$ is the shortest distance from node x to goal node τ.

A heuristic function is consistent if it satisfies the triangle inequality i.e. heuristic estimate of the node to goal should be less than the sum of cost of going to some other node and the heuristic estimate of this node. Mathematically, this is described as $h(x, \tau) \leq h(y, \tau) + c(x, y)$.

Depending on the connectivity of the graph, the heuristic function may vary. For our purposes, since we have a 8-connected graph, we define the heuristic functions as the **Euclidean** distance between the node and the goal node i.e. $h(x, \tau) = ||x - \tau||_2$. This heuristic is consistent since the L2 norm is a valid norm in the Euclidean space and hence it has to satisfy the triangle inequality. In addition, it is also admissible since $h(\tau, \tau) = 0$ which in turn would satisfy the admissibility condition for all other nodes since it is not an

overestimate of the shortest distance from the node to the goal. Hence we define the heuristic as :

$$h(x, \tau) = ||x - \tau||_2 \tag{10}$$

Search based algorithms use the label of the nodes to find the nodes that are the best candidates through which to go to the goal state. This is denoted by g. Mathematically, the label of the node represents the distance of the node from the start node $g(x) = c(s, x)$ where $c(s, x)$ represents the cost of moving from start state s to x.

B. Search Based Algorithm

Search Based algorithms discretize the continuous state space of the motion planning problem to regular sized grids. We define a 8-connected grid in our case as discussed before. One of the most common Search Based algorithms used for motion planning is the A* algorithm. A* algorithm tries to find out the best path from the start state to the goal state by expanding nodes in order of their f-value which is defined as $f_i = g_i + h_i$ i.e. the sum of the label of the node and the heuristic function of the node. A* includes two steps : Label correction of a node, where we see if the current label of a node is less than or more than the cost of transitioning to the current node from a different parent node added with the label of this parent node. If yes, then it means that that we have found a better way to get at this node than the previous path found, and hence we update the label g of this node.

The A* algorithm and its variants keep track of nodes in two ways : there is a OPEN list which keeps track of the frontier of the search i.e. the nodes whose labels are not optimal yet, and a CLOSED list which includes the collection of nodes that have optimal labels assigned to them. Initially, the OPEN list is initialized with the start state and the CLOSED list is empty. We then expand the child nodes of the start state, update their g values and add them to the OPEN list. The start node is removed from the OPEN list and inserted to the CLOSED list. Now the node with minimum f value in the OPEN list is selected, removed from OPEN, added to CLOSED and then the children of this node is added to the OPEN list and their g values are corrected. This process is repeated until the goal state enters the CLOSED list which indicates that now we have found the best path to the goal state from the start state.

The A* algorithm is described in the pseudo code below :

After the A* is terminated, or in case of partial expansion of nodes, we can find the optimal path sequence by starting at the goal state or the most promising node in the open list after partial expansion and selecting the parent such that the label of the parent node summed with the cost of transitioning from parent to the goal state / most promising node in open list is minimum. This process is repeated until we reach the start state. This process is discussed in the algorithm below :

Now we have all the elements needed for solving the motion planning algorithms. Since we have a time constraint requirement for getting the next move of the robot, we will typically look at variations of the A* algorithm such as the Agent Centered Real Time Adaptive A*(RTAA*) algorithm, the Anytime Repair A*(ARA*) for Anytime Search to solve the problem. In addition we will also look at the performance of a Sampling Based Search algorithm called RRT and evaluate its performance. In the subsequent sections, we will first discuss the search based sampling algorithms and then the sampling based search algorithms.

Algorithm 1 A*

Require: : start s, goal τ, costs c_{ij}
  $OPEN \leftarrow \{\}, CLOSE \leftarrow \{\}$
  while $\tau \notin CLOSE$ **do**
    Remove node i with minimum $f_i = g_i + h_i$ in OPEN
    Add node i to CLOSE
    for j $\in Children(i)$ **do**
      IF $g_j > g_i + c_{ij}$
      $g_j \leftarrow g_i + c(x_i, x_j)$
      IF j in OPEN
      Update f value of j in OPEN
      ELSE add j to OPEN with $f_j = g_j + h_j$

Algorithm 2 Obtaining the optimal path

Require: A* terminated or Most promising node in open list

  $\tau \leftarrow$ goal or most promising open list node
  $path \leftarrow [\tau]$
  $i \leftarrow \tau$
  while i not start **do**
    $j \leftarrow \min_{j \in Parent(i)} g_j + c(x_j, x_i)$
    $path \leftarrow path \cup j$
    $i \leftarrow j$
  end while
  return path

1) Agent Centered Search: Consider the scenario where the map is huge and it is not possible to plan the full path to the goal. In that case, what we do is we place a strict limit on the amount of computation. First an assumption of freespace assumption is made which states that all unknown space is first assumed to be a free space and the cost of transition between unknown freespace cells is the same as known free cells. Next, we compute a partial short feasible path by expanding atmost N nodes around the robot current position. Once the move to most promising node around the robot is made, the map is updated from the new robot position by incorporating sensor information. Due to this, there may be a few changes to the map as to some assumed freespace cells may turn out to be obstacles. To incorporate these map changes, the heuristic estimate of the previously expanded cells is updated. The heuristic update of these expanded cells makes the h value of these cells more informed about the actual environment.

We implement the Real Time Adaptive A*(RTAA*) algorithm to implement the agent centered search for our problem. This algorithm constitutes of expanding at most N nodes around the robot, and then using the $f = g + h$ value of

the most promising node in the open list, using the new sensor information to calculate the heuristic at this promising node, and then updating the heuristic value of the previously expanded nodes i.e. nodes in the closed list using this f value at the most promising node. The steps of the algorithm are described in the pseudo code below :

Algorithm 3 Real Time Adaptive A*

Require: : current robot position , goal τ, costs c_{ij}
Expand N nodes using A*
$f^* = \min_{i \in OPEN} g_i + h_i$

for j $\in$ CLOSED **do**
$h_j = f^* - g_j$
end for

After each step of RTAA*, we take a step towards the most promising node in the open list. And then the process is repeated again until we reach the goal state.

2) Anytime A:* The idea behind Anytime A* is that instead of finding the optimal path to the goal, we can find an ϵ sub-optimal path to the goal and this will take less time than finding the optimal path. After we have a sub-optimal path, if there is still time left, we can try to find a better path by decreasing the value of ϵ, otherwise we take a few steps using the sub-optimal path and then run the algorithm again from the new position. As we proceed closer and closer to the goal, we can start getting better and better paths because there is not much time spent in node expansion and hence we can repeat the process for several of ϵ values. One of the main ideas in this is to re-use the labels of nodes in the graphs that were saved in previous searches using higher ϵ values. This saves us a lot of time. This is called the **Anytime Repair A*(ARA*)** algorithm. ARA* tracks the nodes with changing g-values using a separate v-value. If a given node is expanded again in a different epsilon value search and its g-value is found to change from the previous v value, it is added to an INCONSISTENT list. Later when ϵ is reduced for a new search, the open list is initialised as the inconsistent nodes in the search. The algorithm to keep track of the inconsistent nodes is given below :

Now that we have A* which keeps track of the inconsistent nodes, we can implement the ARA* algorithm as given in **Algorithm 5**.

After we have some path from the current position to the goal position, we can move a few steps along this path and then call a new planner once again from the new robot position and the new goal position until we reach the goal position. As we go closer and closer to the goal, we will start getting optimal paths since due to less expansion of nodes, ϵ will become 1 very quickly.

C. Sampling Based Search

Contrary to search based algorithms, sampling based algorithms do not discretize the state space into regular grids. Sampling based algorithms use sampling based discretization

Algorithm 4 A* keeping track of inconsistent nodes

$OPEN \leftarrow \{s\}, CLOSED \leftarrow \{\}, \epsilon \geq 1$
$g_s = 0, g_i = \infty : \ \forall i \in \mathcal{V} - \{s\}$
$v_i = \infty : \ \forall i \in \mathcal{V}$
COMPUTEPATH()
function COMPUTEPATH()
while $f_\tau > \min_{i \in OPEN} f_i$ **do**
remove i with smallest f_i from OPEN

Insert i into CLOSE and $v_i = g_i$

for j $\in Children(i)$ **do**
$g_j \leftarrow min(g_j, g_i + c_{ij})$
If j $\notin$ CLOSE then insert j into OPEN
otherwise insert j into INCONS

Algorithm 5 Anytime Repairing A*

Set ϵ to large value
$OPEN \leftarrow \{s\}$
$g_s = 0, g_i = \infty : \ \forall i \in \mathcal{V} - \{s\}$
$v_i = \infty : \ \forall i \in \mathcal{V}$
while $\epsilon \geq 1$ **do**
$CLOSED \leftarrow \{\}, INCONS \leftarrow \{\}$
$OPEN, INCONS \leftarrow COMPUTEPATH()$
Publish current ϵ sub-optimal path
Decrease ϵ
OPEN = OPEN $\cup$ $INCONS$
If time $\leq$ constraint
Repeat

to generate trees that connect different feasible nodes to each other and try to find a connection from the start state to the goal state. We try to implement a simple vanilla **Rapidly Exploding Random Tree(RRT)** algorithm and a slightly complicated version of RRT called **RRTConnect** algorithm to find the path from the start state of the robot to the goal state. Once we get a path from the sampling based algorithms, we follow the path for some time, and then call the planner again to search for a new path from the current position of the robot to the new target position. The pseudo code for the RRT algorithm is shown below :

Algorithm 6 Rapidly Exploring Random Tree(RRT)

Require: $V \leftarrow \{x_s\}, E \leftarrow \{\}$
for i = 1,2,....n **do**
$x_{rand} \leftarrow SampleFree()$
$x_{nearest} \leftarrow Nearest((V, E), x_{rand})$
$x_{new} \leftarrow Steer(x_{nearest}, x_{rand})$
IF $CollisionFree(x_{nearest}, x_{new})$
$V \leftarrow V \cup \{x_{new}\}; E \leftarrow E \cup \{(x_{nearest}, x_{new})\}$
return $G = (V, E)$
end for

In the above algorithm, we use several functions such as SampleFree, Nearest, Steer and CollisionFree. Following is the description of each of the functions :

- **SampleFree** : This function is used to generate a point randomly from the map. The sampling is done as a uniform distribution.
- **Nearest** : This selects a point in the graph that is closest to the randomly sampled point from the above function.
- **Steer** : This function generates a new point in the line connecting the two points $x_{nearest}$ and x_{rand} within a distance ϵ from $x_{nearest}$.
- **CollisionFree** : This function checks if the line connecting the two points x_{new} and $x_{nearest}$ lies within any obstacles in the map. Otherwise if the line intersects with an obstacle, the function will create a point x_c close to the obstacle boundary as allowed by the collision detection algorithm.

Occasionally, the goal configuration is added to the graph and a check is done to see if it can be connected to any of the vertices currently in the graph. If yes then we have successfully found a path connecting the start and the goal.

We won't discuss the exact details for the implementation of RRTConnnect. But the main idea is that in RRTConnect, a tree is constructed from both the start point and the goal point, and at every iteration an attempt is made to connect the two trees at a given random position. The advantage of RRTConnect over simple RRT is since we are working with bi-directional trees, there is slightly better chance of finding feasible paths from the start to the goal in complex maze like maps where simple RRT would fail to find even a single path.

D. Implementation Details

Before we present the results, we discuss a few implementation details.

- Often in motion planning problems, we need to create a graph. The complexity of graphs increases as the maps become bigger, number of obstacles increases and so on. So if we try to create the full graph at once in the beginning, it will take up a huge amount of time. Instead, what we do is create the graph on the go. If a given node is expanded for the first time, it is initialised on the graph as a dictionary containing the node position in the map, the current label g of the node and h value of the node. The graph is implemented as a dictionary in python or generally a Hash Table data structure. The key of the hash table is defined as an integer ranging from 0 to the total number of nodes in the map which is equal to $x_{max} * y_{max} - 1$. Hence in summary, each coordinate in the map is defined in the following way :

$$id : \{pos : (x, y), g : g(x, y), h : dist((x, y), \tau)\} \quad (11)$$

- Given the test maps, scenarios 1b and 7 are the most challenging in terms of map complexity. Using the original map, neither the Agent Centered Search or the Anytime Search were able to find the path to the goal in the desired time constraint. So we tried out the motion planning on a compressed versions of the map. We used the open-cv resize function to compress the maps. This is not a very good idea because resizing can lead to loss of information such as some thin walls may vanish. But for this project, because of limited time we did not try out other methods for lossless compression of maps and hence these are left for future work.
- For the search based A* algorithms, the OPEN list was implemented as a Priority Queue, containing the key as the id of the node, and the value as the f-value of the node. CLOSED list is implemented as a simple list.

In the next section, we see the results for the search based and sampling based motion planning algorithms on the provided test cases and will try to compare the performance in terms of whether the robot is able to catch the target and if so, in how many moves.

IV. Results

In this section, we present the results of our motion planning algorithms on the provided test cases. First we present the results for the 2 search based algorithms that we implemented namely the Agent-Centered Search and the Anytime Search. Later we present the results for the sampling based RRT algorithm. We try to evaluate the performance of the algorithms based on if the robot was able to catch the target and if so, how many moves was needed.

A. Search Based Motion Planning

1) Agent Centered Search: In this approach, we only expanded a small number of nodes in the A* algorithm. Once the number of specified nodes are expanded, we make a single move towards the most promising node in the current OPEN list using the heuristic function. Contrary to Real Time Adaptive A*, where we update the heuristic value of a node after expansion of the specified number of nodes, we do not update the heuristic function of the nodes. We then call the planner again to get the path from the new robot position to the new target position. We ensure that the next move of the robot is produced in the specified 2 seconds time constraint by controlling the number of nodes expanded. Results are shown in Fig(2) - Fig(12). For map7 i.e. Fig(12), we were not able to make the Agent Centered Search work and hence we show the result only for Anytime Search for this map.

2) Anytime Search: In this approach, we compute and ϵ sub-optimal path from the current robot position to the current target position. If a path is obtained within the time constraint, we decrease ϵ and try to find a better path, otherwise we make a few steps along the sub-optimal path published. After taking a few steps, we call the planner again with the new robot position and the new target position. Results are shown in Fig(2) - Fig(12). For map7, the result is shown only for the Anytime Search since we were not able to make the Agent Centered Search work.

B. Sampling Based Motion Planning

1) RRT: In this section, we present the results for the motion planning implemented as a sampling based RRT algorithm. We use the simple vanilla RRT algorithm for the maps. We get some paths using RRT for maps 0,1,2,4,5 and 6. This is because these scenarios are not so complex and RRT performs well in problems where the map is almost sparse or the goal is directly connectable to the start position. RRT fails to produce paths for maps 1b, 3, 3b, 3c because these are complex and involve maze like structures. Since sampling based algorithms are only probabilistically sub-optimal, the performance is not well for these kind of scenarios where the map-complexity is significant because of maze like structures inside the map. The results for the RRT algorithm for maps 0,1,2,4,5,6 are shown in Fig(13) - Fig(14).

2) RRTConnect: Contrary to the vanilla RRT algorithm, we observe that the RRTConnect algorithm was able to give us feasible paths even in some complicated maps like 1b, 3 and 3b. This is credited to the fact that in RRTConnect, bi-directional trees are constructed from both the start position to the goal position. Results are shown in Fig(15) - Fig(16)

C. Discussion

TABLE II: Number of moves before target was caught

Map	Agent-Centered	Anytime	RRT	RRTConnect
0	5	4	6	7
1	1279	1279	1151	1301
1b	615	535	NA	695
2	13	12	14	20
3	224	223	NA	1028
3b	470	441	NA	1220
3c	774	762	NA	NA
4	9	8	18	11
5	102	86	116	398
6	39	40	56	346
7	NA	673	NA	NA

From Table II, we can evaluate the performance of search based motion planning algorithms vs sampling based motion planning algorithms. Following are the comparisons :

- We observe that Search Based motion planning methods are able to find the path in all of the graphs except in original MAP7. It works on a resized version of MAP7 but since there was significant resizing, we lose many of the information from the original map (the thin walls), and hence this performance is not very fair. Among search based algorithms, both Agent-Centered and Anytime Search have a similar performance except for resized Map7 where we could not make the Agent Centered Search. This may be due to the fact that there are a lot of obstacles and hence with partial expansion of nodes and not updating the heuristic, the robot gets stuck in a local minima.
- Vanilla RRT was able to find the path only in maps 0,1,2,4,5 and 6. This is because there is a significant maze like structure in maps 1b,3,3b,3c and 7.
- RRTConnect was able to find the path in maps 1b,3 and 3b in addition to maps where RRT worked. But the paths obtained are highly fluctuating and hence as observed from Table II, the number of moves made by the target before being caught is a lot more in some maps compared to vanilla RRT and Search Based motion planning. This is probably due to the fact that since in RRTConnect the tree is bi-directional, as the target keeps moving in our case, the structure of the tree keeps on changing very frequently and hence the path is fluctuating.
- Since Sampling based motion planning uses sampling to discretize the grid, there is a randomness present in the path that we obtain. For different runs the path may change or it is also possible that we do not get a path at all.
- Search based algorithms are guaranteed to return an ϵ sub-optimal path to the goal, whereas Sampling based algorithms like RRT and RRTConnect can only find a feasible path with no guarantee of optimality.

V. Conclusion and Future Work

In this project, we implemented search based and sampling based motion planning algorithms to solve the problem of a robot trying to catch a moving target. To satisfy the time constraint to produce the next move, we considered variants of the A* algorithm including the Agent-Centered Search and Anytime Search. We constructed the graph for the search based motion planning on the go instead of constructing the full graph at the beginning since it would turn out to be very time consuming.

For future work, one of the most promising ideas is to try to represent complicated maps using more effective data structures such as **Quadtree, Octree** which would in turn make the search based algorithms much quicker. Another idea is to implement slightly complex versions of RRT such as RRT*, RRT* with bidirectional heuristic rewiring that again uses the concept of node labels and heuristics as seen in Search Based motion planning, and try to rewire the trees to try to find an optimal path from the start to the goal even in Sampling based algorithms. We could also implement Path Smoothing to smoothen the paths obtained from RRT, RRTConnect.

VI. Acknowledgement

I would like to thank Professor Nikolay Atanasov for his help and suggestions throughout this project. His assistance helped me overcome some crucial problems encountered during the project.

References

[1] Nikolay Atanasov, "https://natanaso.github.io/ece276b/ref/ECE276B_5_DSP.pdf"
[2] Nikolay Atanasov,"https://natanaso.github.io/ece276b/ref/ECE276B_6_CSpace.pdf"
[3] Nikolay Atanasov,"https://natanaso.github.io/ece276b/ref/ECE276B_7_SearchBasedPlan
[4] Nikolay Atanasov,"https://natanaso.github.io/ece276b/ref/ECE276B_8_AnytimeIncremen
"
[5] Nikolay Atanasov,"https://natanaso.github.io/ece276b/ref/ECE276B_9_SamplingBasedPl
"
[6] "https://github.com/motion-planning/rrt-algorithms"

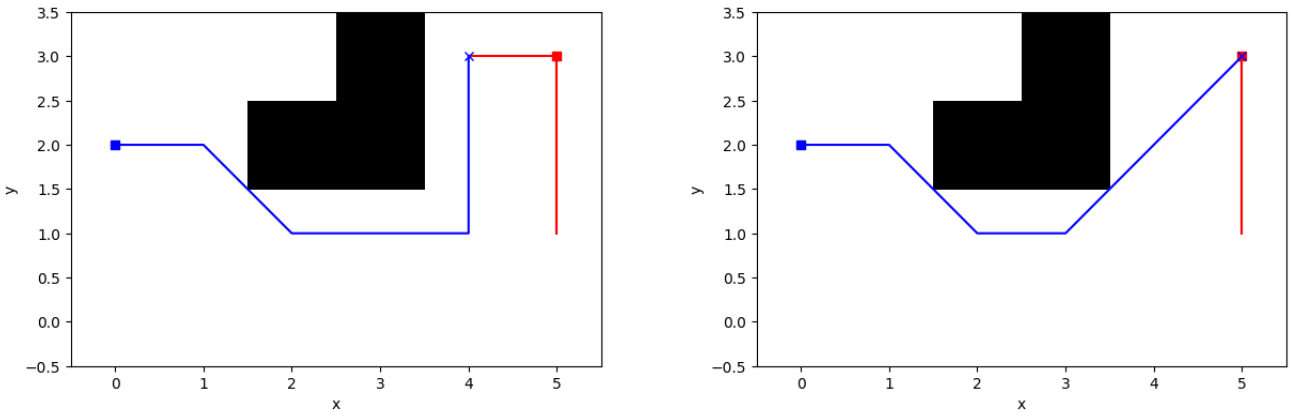

Fig. 2: Search based motion planning on map 0, on left - agent centered search; on right - anytime search

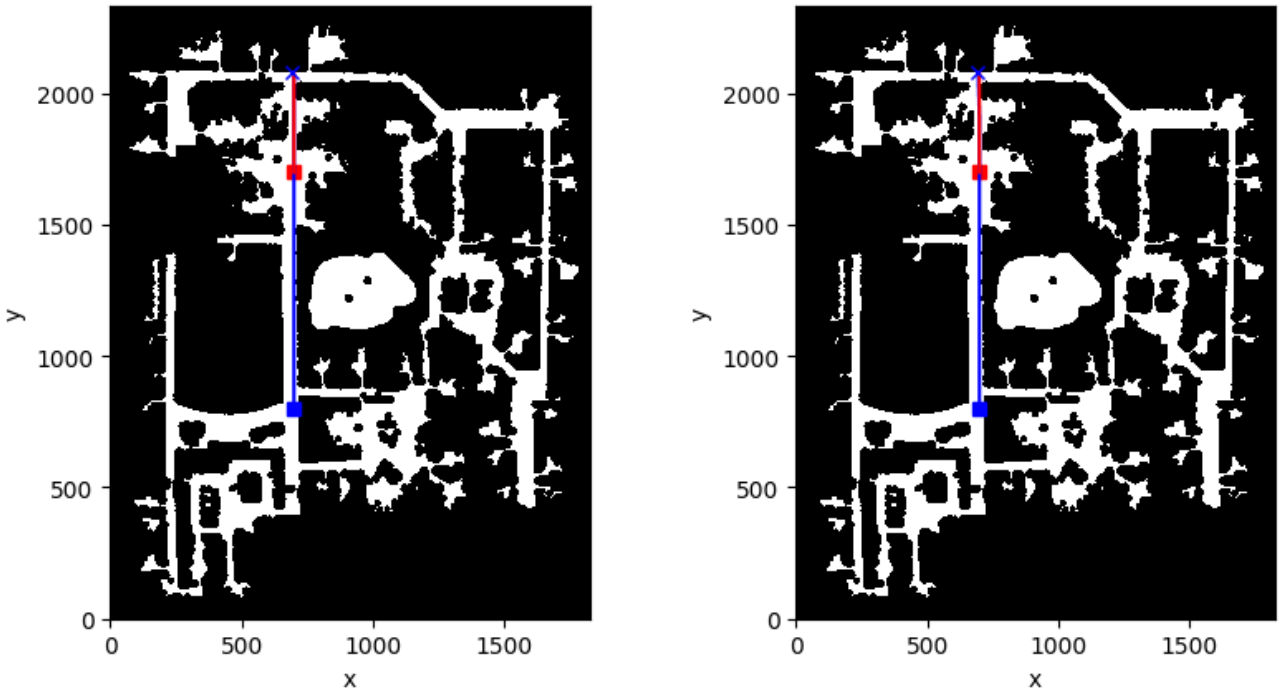

Fig. 3: Search based motion planning on map 1, on left - agent centered search; on right - anytime search

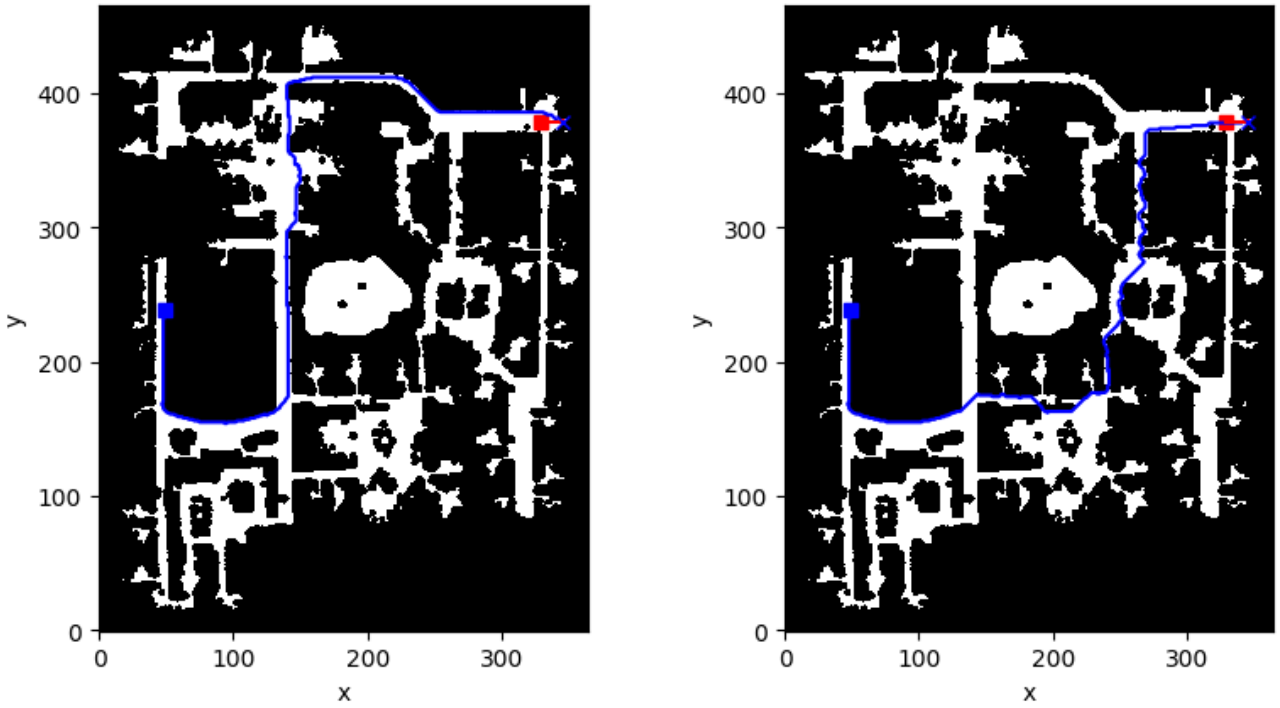

Fig. 4: Search based motion planning on map 1b, on left - agent centered search; on right - anytime search

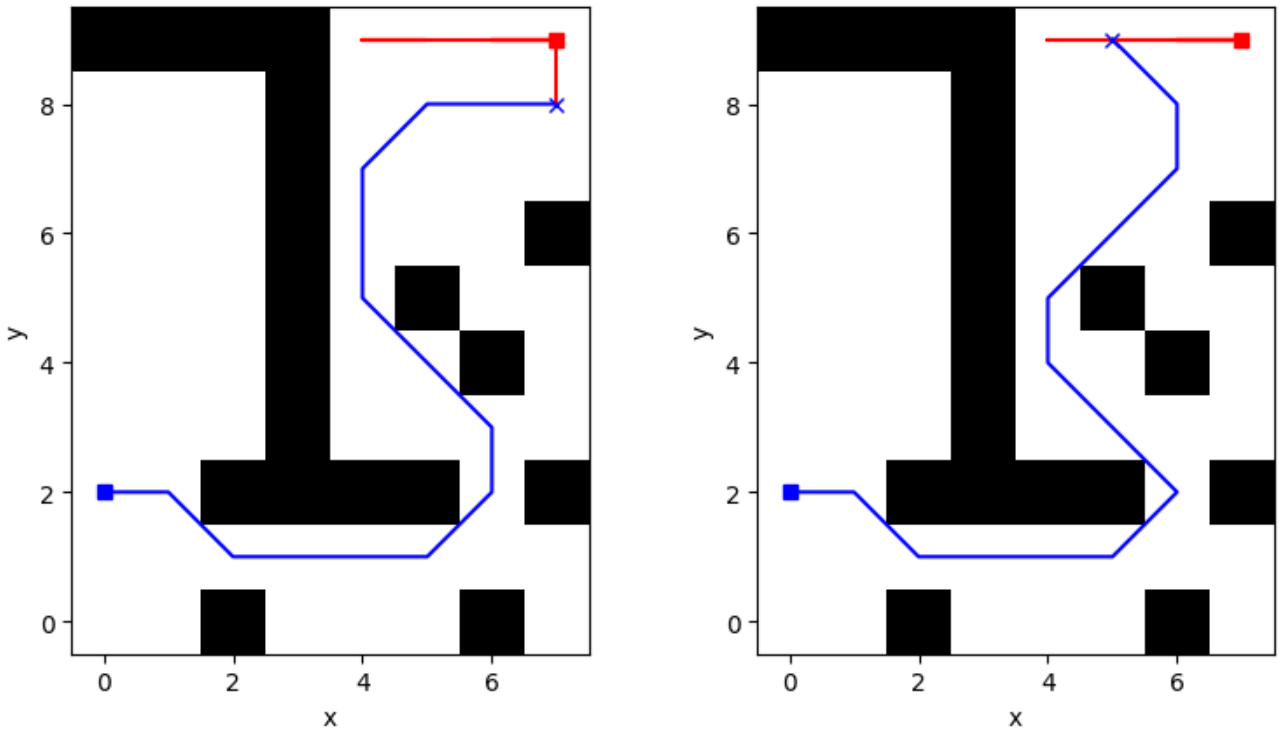

Fig. 5: Search based motion planning on map 2, on left - agent centered search; on right - anytime search

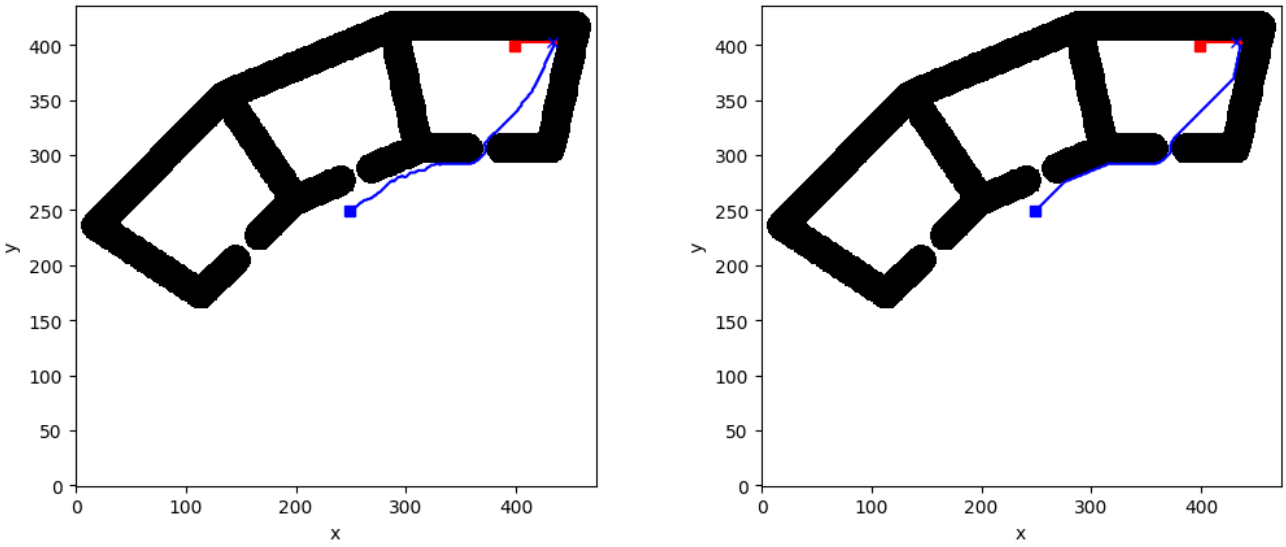

Fig. 6: Search based motion planning on map 3, on left - agent centered search; on right - anytime search

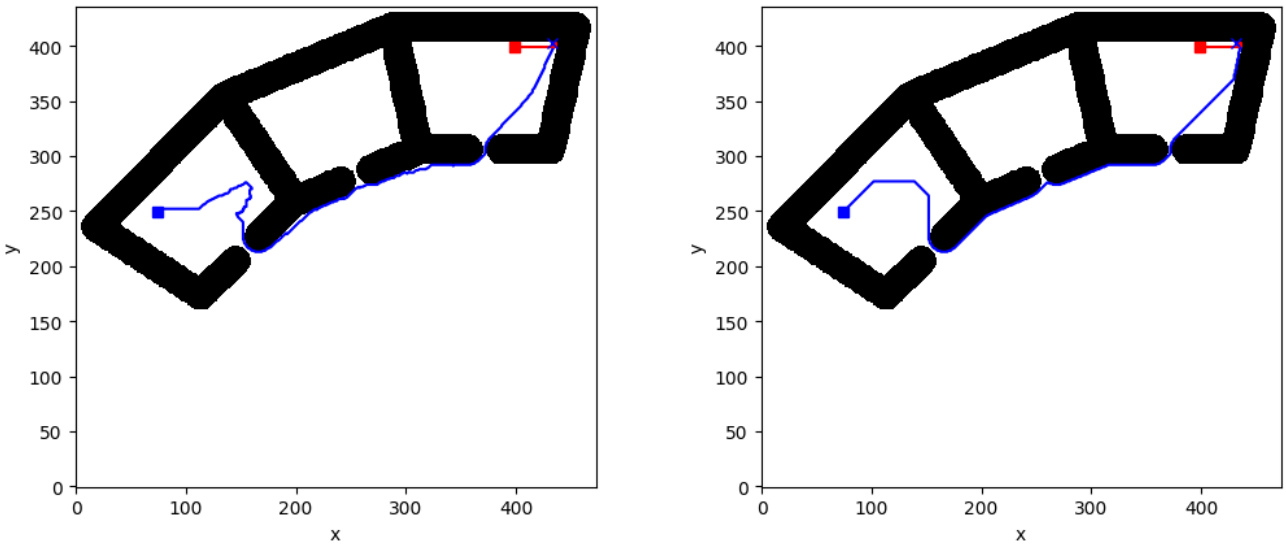

Fig. 7: Search based motion planning on map 3b, on left - agent centered search; on right - anytime search

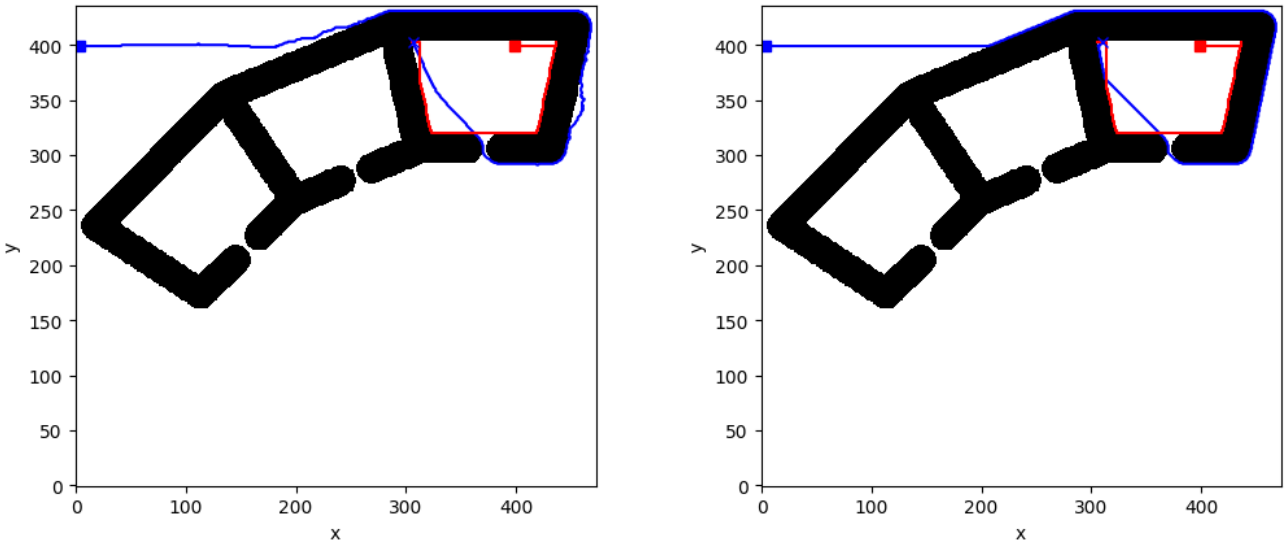

Fig. 8: Search based motion planning on map 3c, on left - agent centered search; on right - anytime search

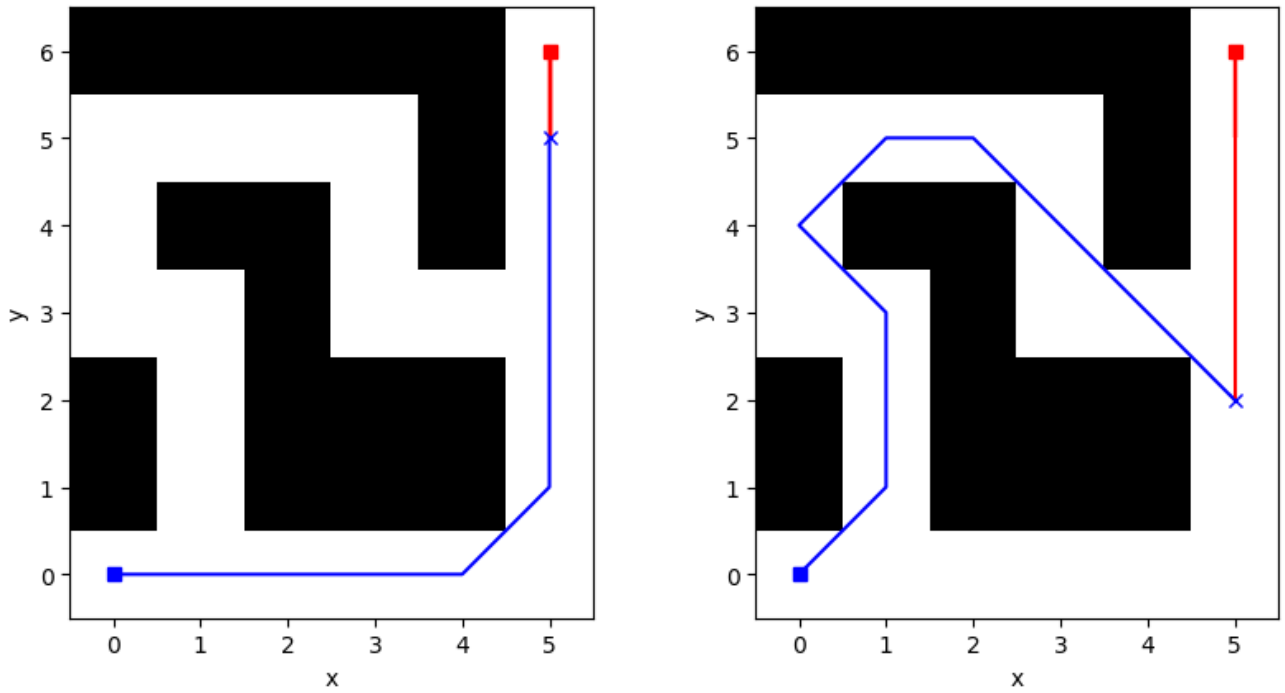

Fig. 9: Search based motion planning on map 4, on left - agent centered search; on right - anytime search

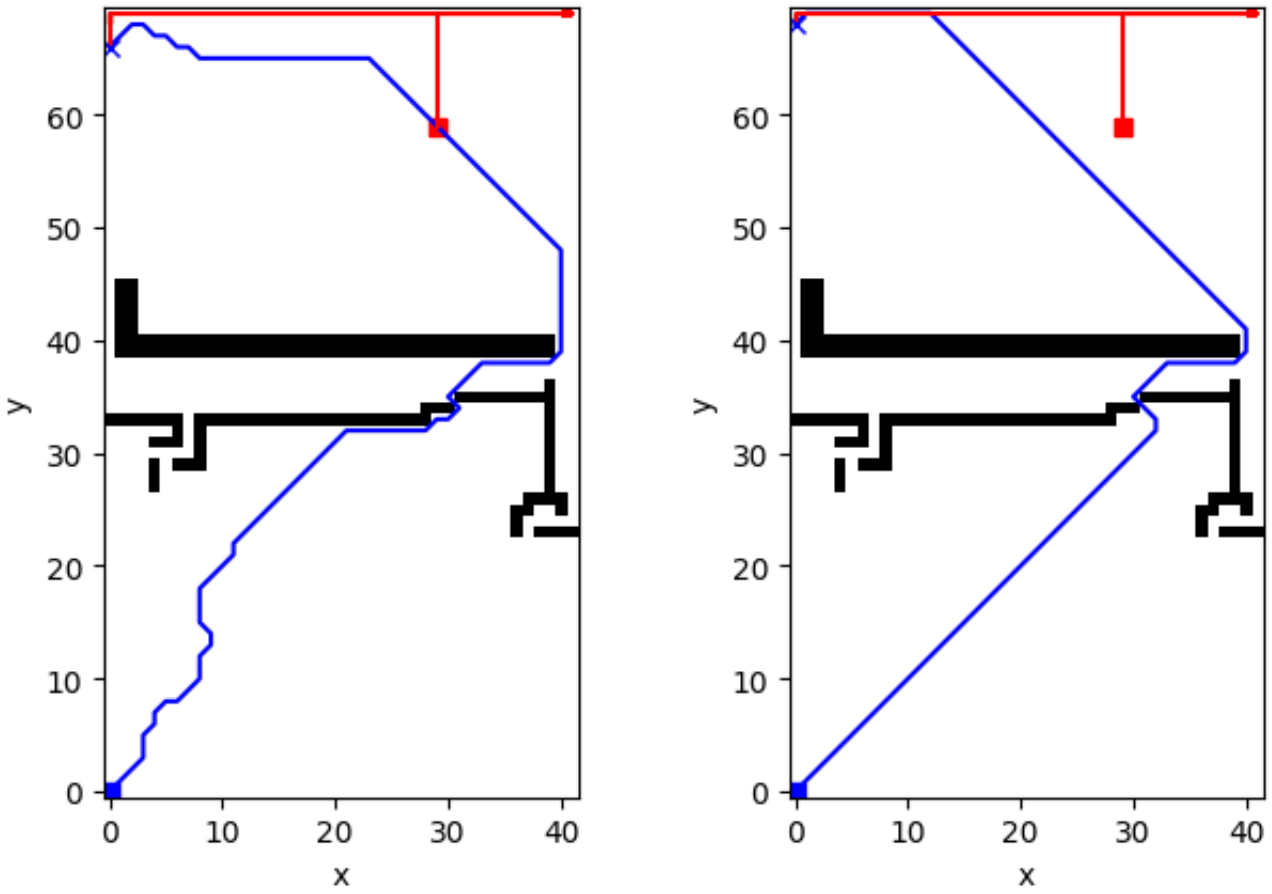

Fig. 10: Search based motion planning on map 5, on left - agent centered search; on right - anytime search

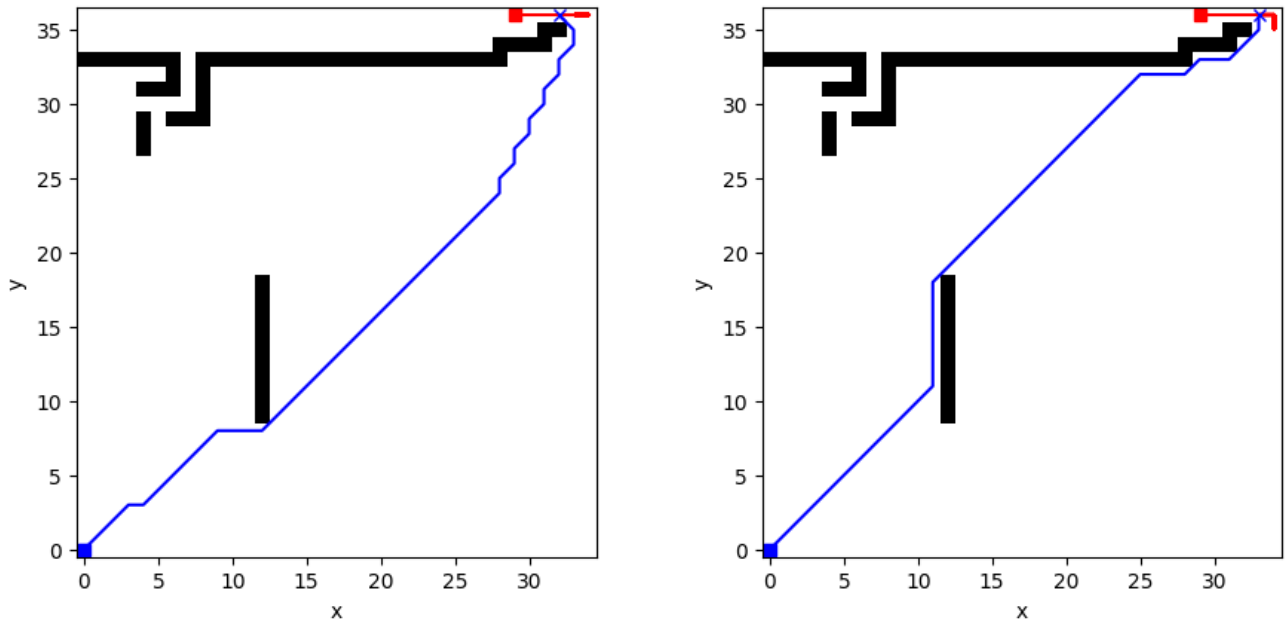

Fig. 11: Search based motion planning on map 6, on left - agent centered search; on right - anytime search

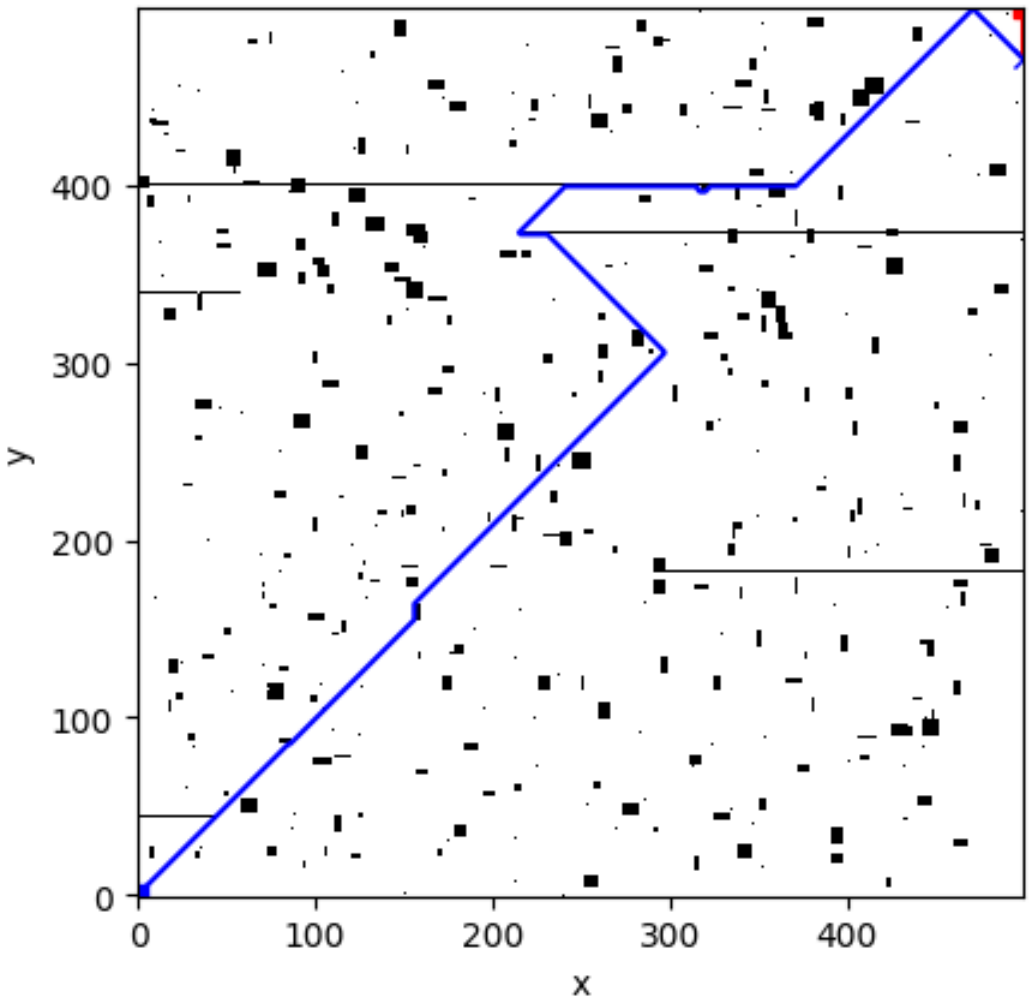

Fig. 12: Search Based Anytime Search on resized map7

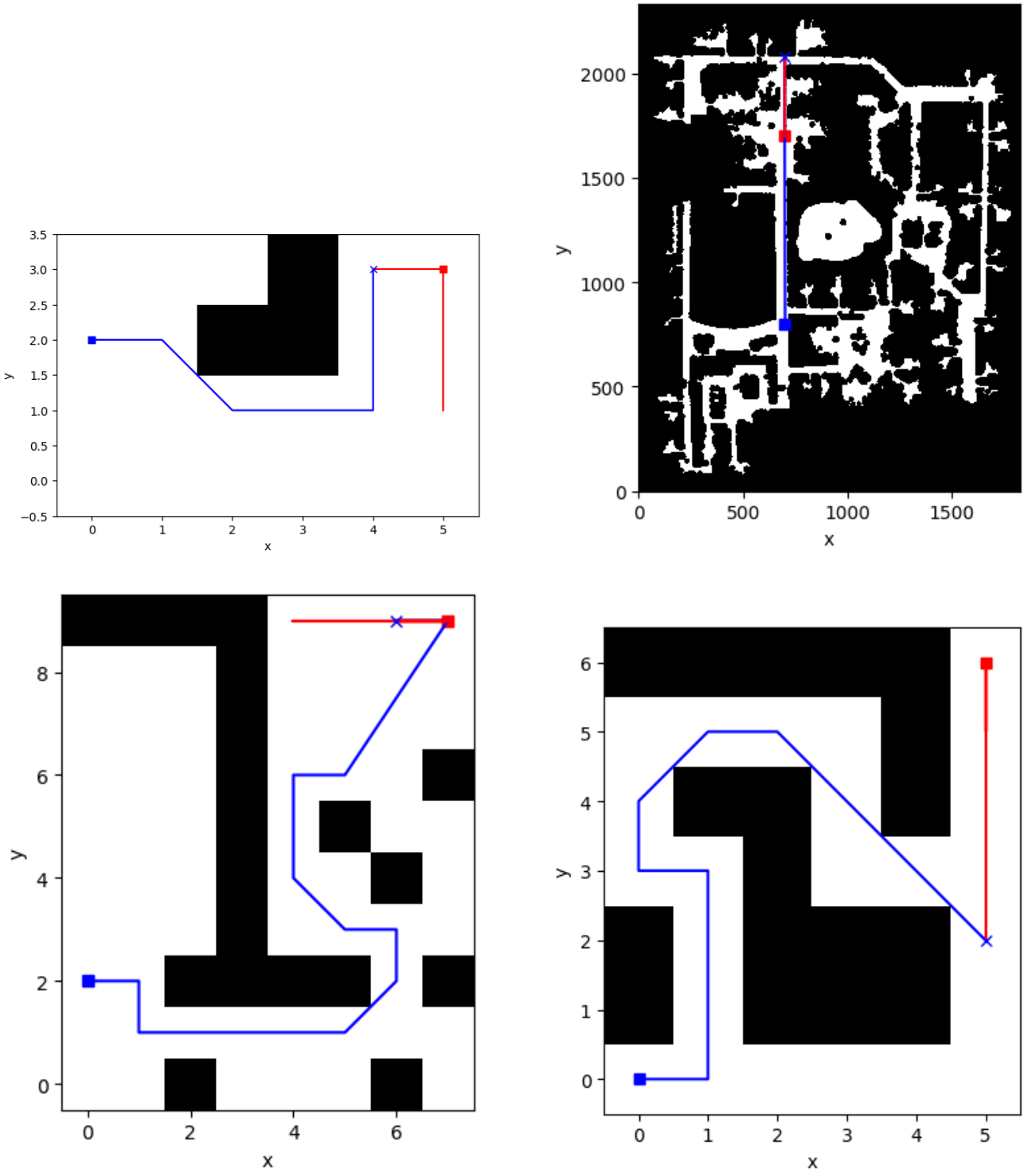

Fig. 13: Sampling based RRT motion planning on maps 0,1,2,4

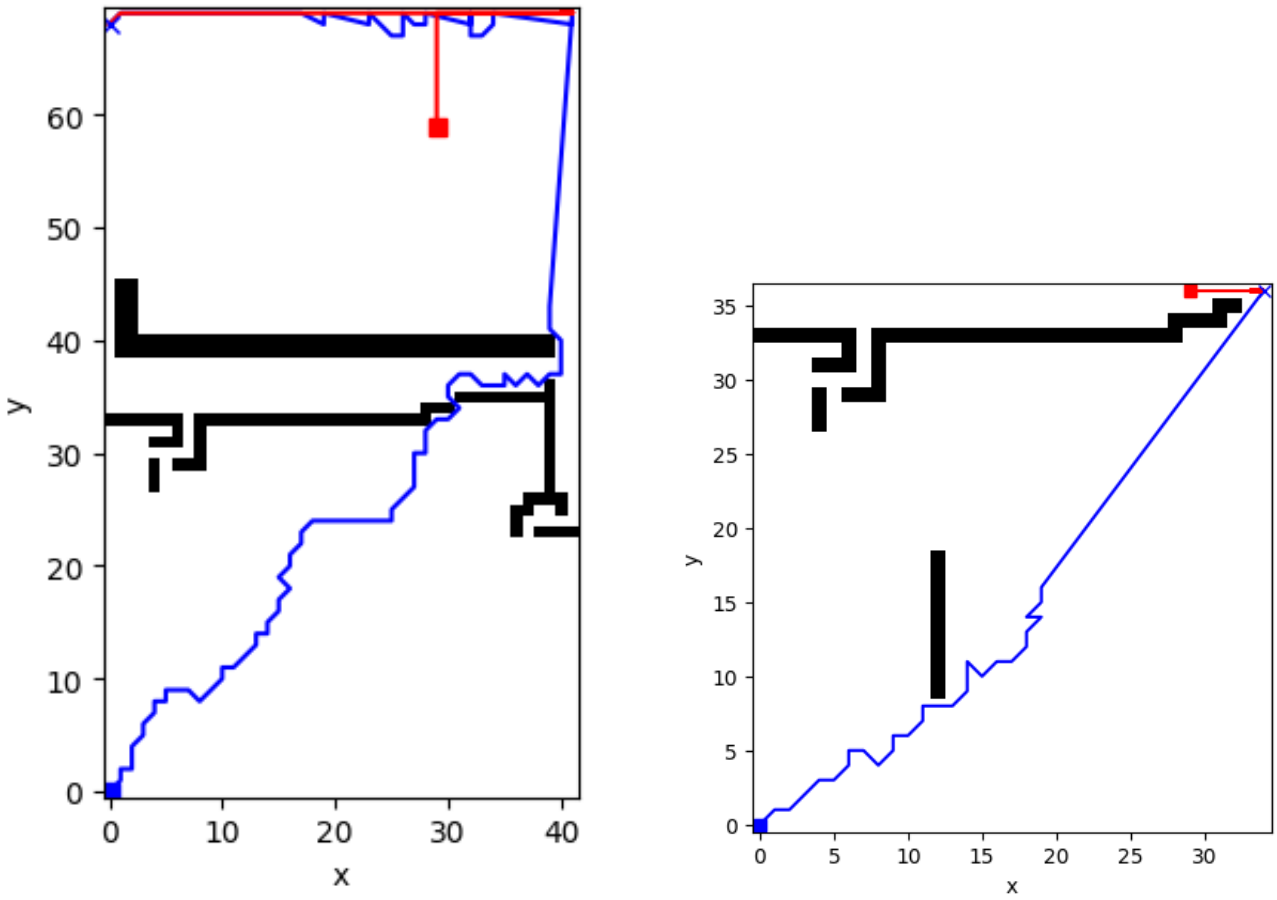

Fig. 14: Sampling based RRT motion planning on maps 5 and 6

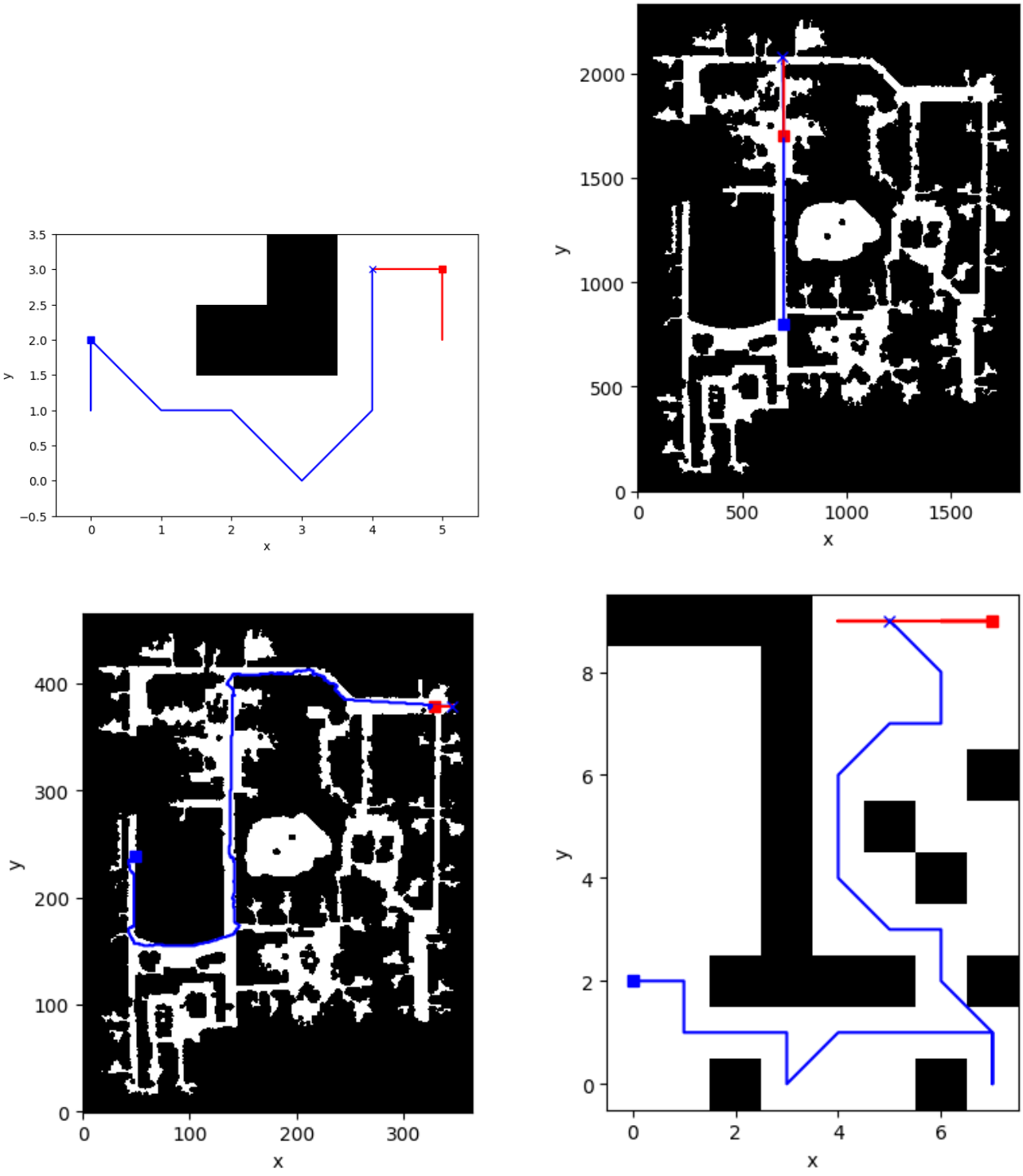

Fig. 15: Sampling based RRTConnect motion planning on maps 0(top left),1(top right),1b(bottom left) and 2(bottom right)

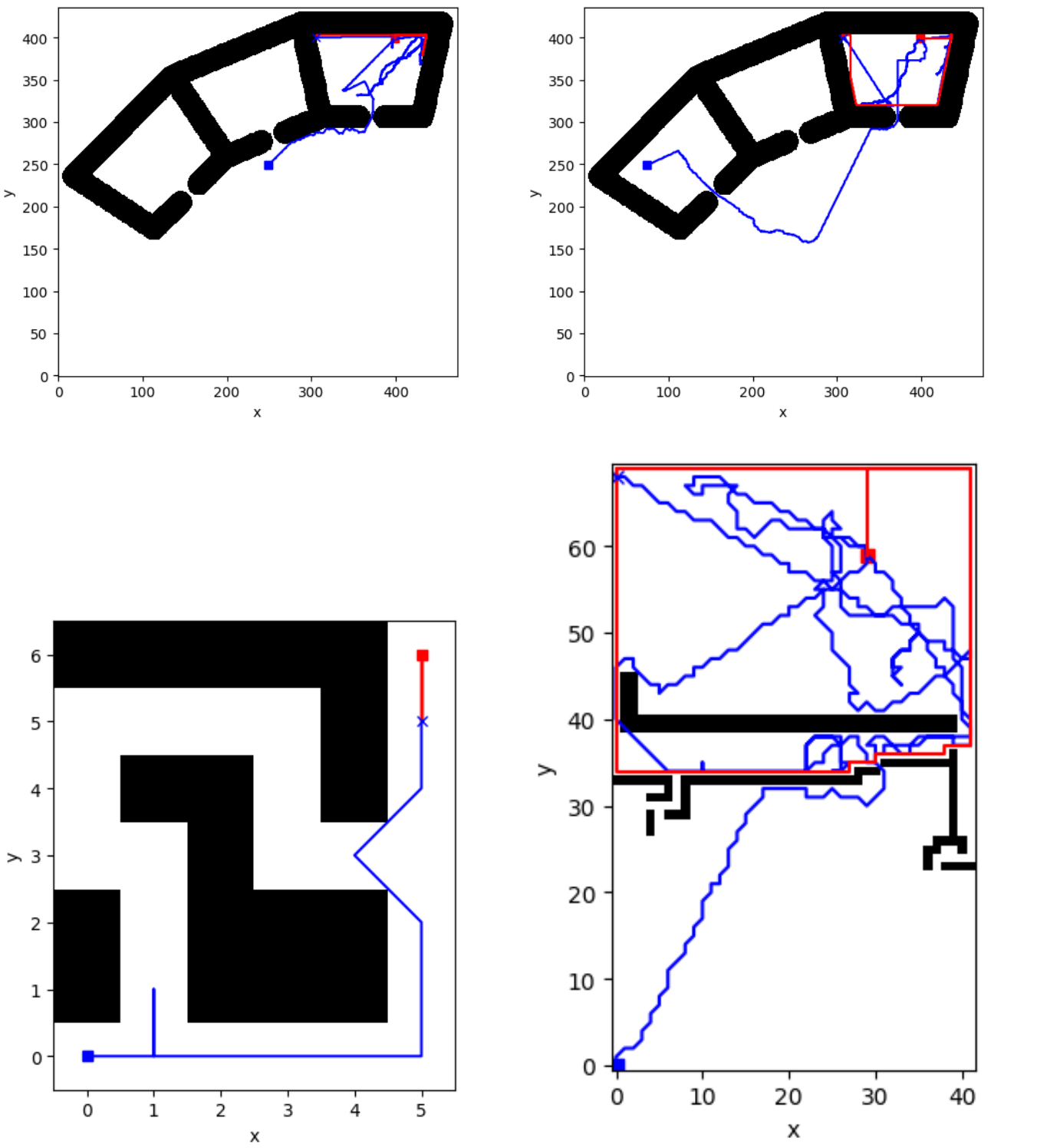

Fig. 16: Sampling based RRTConnect motion planning on maps 3(top left), 3b(top right), 4(bottom left) and 5(bottom right)