HW2: Sampling-Based Motion Planning

Alex Furman (941180150)

Tom Agami (302485628)

January 2, 2023

1 Introduction

In this homework, we explore:

1. How the number of dimensions may affect sample-based search algorithms performance.
2. The idea of a 'tethered robot', and how to succinctly represent paths the robot may take, using certain data-structures.
3. Implementing a weighted version of A* and seeing how different weights affect the resulting path found.
4. Implementing sampling-based search algorithms (RRT, and RRT*) and seeing how different parameters affect the resulting path found.

2 Distribution of points in high-dimensional spaces

2.1 Warmup

2.1.1 In a 2-dimensional unit ball (namely, a disk), how much of the volume is located at most 0.1 units from the surface?

Using only intuition, and no calculations, we would guess roughly 10% of the volume is located in this ring

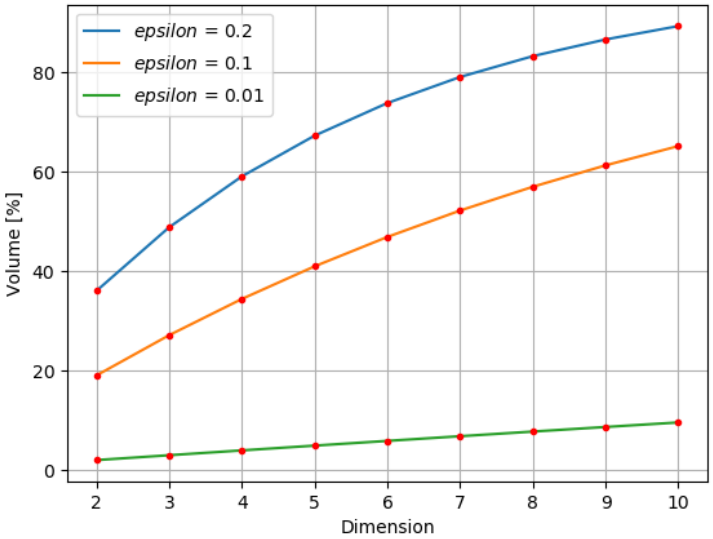
2.1.2 In a 9-dimensional unit ball, how much of the volume is located at most 0.1 units from the surface?

Again using only intuition, we imagine that the volume contained by this ring decreases as the number of dimensions increase, and therefore would guess roughly 1% of the volume of the 9-D ball is located in the 9-D ring

2.2 Exercise

2.2.1 Plot $\eta_d(\varepsilon)$ as a function of d for $\varepsilon = 0.2, 0.1, 0.01$ for $d = 2...10$.

Figure 1: Percent of volume over the dimension number, for different values of ε



2.2.2 Discuss the implications to reducing the connection radius required for a sampling-based algorithm to maintain asymptotic optimality.

The plots revealed results which contradict to our prior guesses. We learn that as the dimension number increases, the percentage of volume taken up by the slice of ε thickness increases - and tends to 100%. This is important in terms of choosing an appropriate connection radius for a sampling-based planner, because the connection radius will determine the volume of our C-Space that will be 'skipped over'. In our case, later in this homework when dealing with RRT or RRT^*, if the connection radius is too large, we essentially ignore a large area where there may be collisions, making expansion of the

tree less likely per sample.

When studying the proof of asymptotic optimality of a sampling algorithm, given an environment with obstacles, where there is some minimal clearance - large emphasis is put on the connection radius we choose, in order to ensure asymptotic optimality.

Therefore, we would like to choose a connection radius that is on the scale of the minimal clearance, which helps keep our search algorithm somewhat near asymptotically optimal.

3 Tethered robots

We consider a 2D point robot translating amidst polygonal obstacles while being anchored by a tether to a given base point p_b. The robot may drive over the cable, which is a flexible and stretchable elastic band remaining taut at all times. We study the problem of constructing a data structure that allows to efficiently compute the shortest path of the robot between any two given points p_s, p_t while satisfying the constraint that the tether can extend to length at most L from the base.

3.1 Path description

A strong way to describe the structure of a path of a tethered robot is to use a list of configurations, where we define each configuration as the position of the robot ($< x, y >$), and the homotopy class of the robot - which describes how the robot is currently wrapped around the obstacles.

3.2 Encoding of the tether description

We are given that the tether will always remain taut, which means the tether will either be a straight line, or it will be a piecewise straight line which touches some of the vertices of the obstacles.

Therefore an efficient way of encoding the tether description would be an ordered list of vertices, where each vertex is a vertex of an obstacle that the tether is currently in contact with.

Note that in order to calculate the homotopy classes, we would need access to a list of obstacles.

3.3 Homotopy-augmented graph

In order to find the homotopy-augmented graph, we run a dijkstra-like algorithm on G.

Note, that we use some kind of auxiliary function $find-homotopy$ to calculate the homotopy class of some current node, given its previous edge leading to current node + homotopy class of previous node. This algorithm can be calculated using an algorithm similar to graph search using a visibility graph, where the tether's anchor point is the start point and the end point is the current vertex we are interested in.

Further note that the current length of the tether can be easily calculated at all times, for example: using $find-homotopy$ and calculating the length of all the vertices in the homotopy class.

The algorithm works as follows:

function h-augmented:

define a queue Q

mark start node using p_s.

add some arbitrary homotopy class to the node's homotopy class list (possibly using visibility graph shortest search between tether and start node)

add start node to Q

while Q not empty:

pop the node current-node at the front of Q

for edge leading out of current-node:

get node neighbour-node from the edge

h = find-homotopy(current-node, edge, homotopy class of current-node)

if length of tether $\leq$ L, and h is not in neighbour-node's homotopy list:

add homotopy class to neighbour-node's homotopy list

if neighbour-node not in Q:

add neighbour-node to Q

The idea here is that eventually, all 'legal' homotopy classes (which ensure tether length less than L) will be visited. This is different from Dijkstra in a few ways, mainly that we do not keep a priority queue, we add nodes back into queue, and we iterate through the edges out of a current node, and not its neighbour nodes.

3.4 Efficient graph-search using homotopy-augmented graph

In order to efficiently search an h-augmented visibility graph, we need to add the points p_s and p_t (assuming the points p_s and p_t have JUST been added). First, we connect edges to all the vertices which are visible from these 2 points (like we did in the last homework). Now, we run the 'Dijkstra-like' algorithm on the current augmented graph, which will add all of the relevant homotopy classes to the nodes of p_s and p_t.

This newly constructed graph may more efficiently solve motion-planning problems by taking advantage of the fact that we know have full information over permitted homotopy classes. We could run Dijkstra or A* on the graph, which will return a path constrained by the tether's maximum length.

4 Motion Planning: Search and Sampling

4.1 Code overview

blah blah

4.2 Environment Modelling

The environment we have been provided with is a discrete 2D map, along with obstacles, a start point, and an end point.

4.3 A^* Implementation

Here we used a weighted version of A^* to find a path to the goal. The heuristic used by the algorithm is $f = g + \varepsilon \cdot h$, where g is the the total cost of the sub-path taken to reach some node n_i, and h is the Euclidian distance from some node n_i to the goal.

We run the algorithm for different values of ε. Below are the results:

Figure 2: A^* with $\varepsilon = 1$

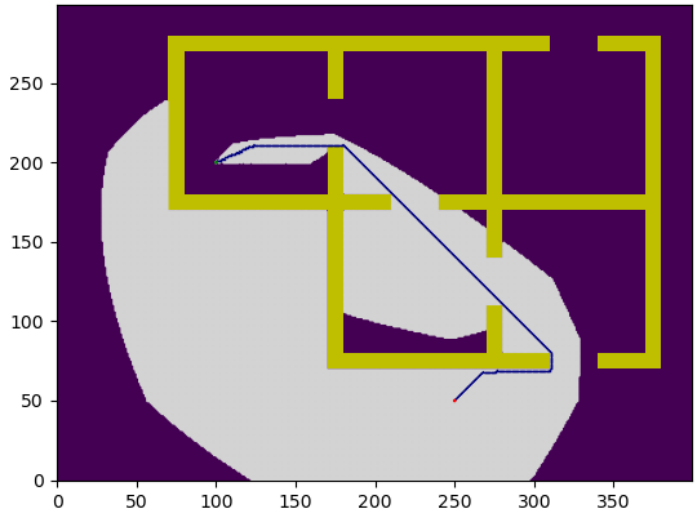


Figure 3: A^* with $\varepsilon = 10$

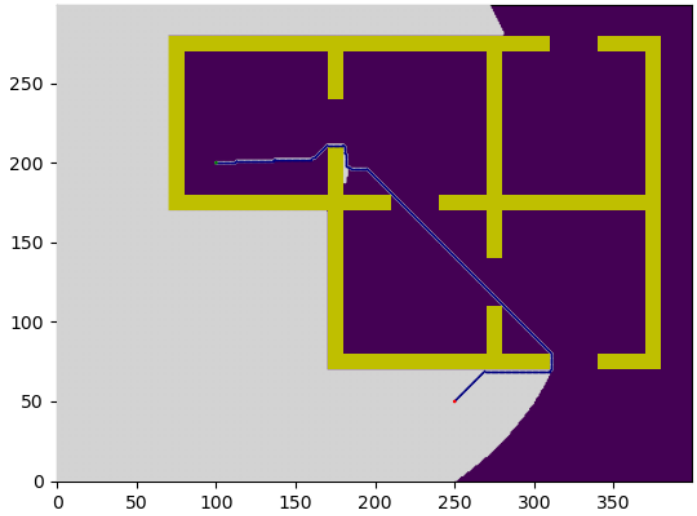
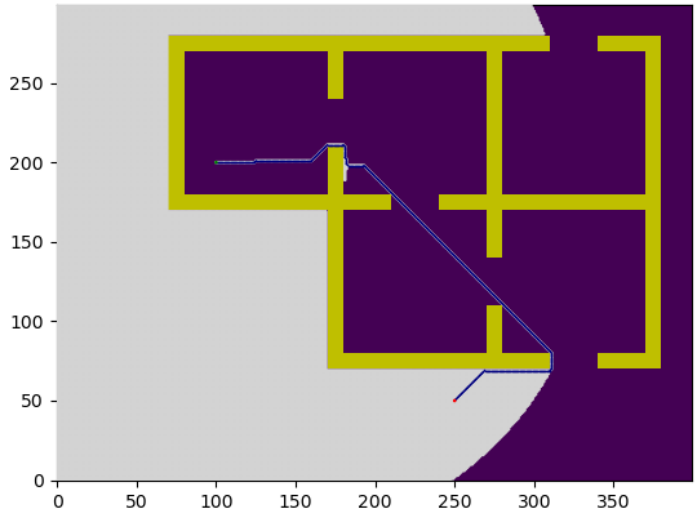


Figure 4: A^* with $\varepsilon = 20$



ε	Nodes Expanded	Total Cost
1	43779	349.10
10	50291	356.13
20	50684	355.55

We notice that as ε grows in value, the algorithm becomes more greedy, meaning that it values proximity to the goal more than the cost to the current state. This may be an effective strategy in the absence of obstacles - however, as can be seen in the figures above, this may cause many node expansions which do not lead to the goal, even though the states are nearby to the goal-state.

In terms of the path found by the algorithm, the path found using $\varepsilon = 1$ yields a solution with the lowest cost. This makes sense since the cost to the goal is valued equally to the distance from the goal when choosing node expansions.

4.4 *RRT* and *RRT** Implementation

4.4.1 *RRT*

We begin by implementing *RRT*, with 2 hyper-parameters: method of extension, and goal sampling bias.

The goal sampling bias tells the algorithm how frequently to use the goal itself as a sample state.

The method of extension (either E1 or E2) changes informs the algorithm on how to extend. E1 causes the algorithm to extend an edge all the way to the sampled point, while E2 causes the algorithm to extend by a set step-size, chosen by us. We decided to use the step-size $min(L_{extend}, L_{sample})$, where L_{sample} is the length to the sampled point from the nearest neighbour node already in the tree, and $L_{extend} = 10$.

We decided to consider L_{sample}, so that the algorithm would not extend past some sample state. This is specifically important when sampline the goal itself - we would prefer to reach the goal directly, instead of overshooting.

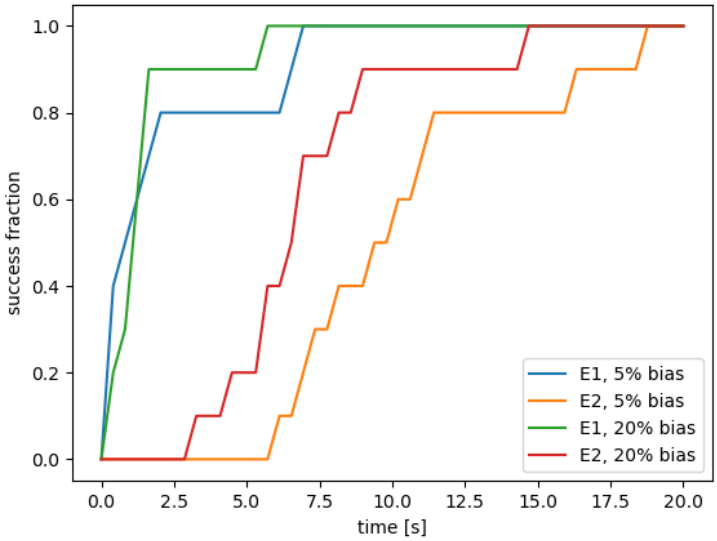
We chose $L_{extend} = 10$ based on the environment itself. We see that the obstacles have minimal clearances of just over 20. We would like to ensure that we can usually sample near these clearances, without necessarily colliding with the obstacles. Therefore, we use $L_{extend} = 10$, which just allows for sampling to the left and right, in the case we are currently at the centre of a clearance.

Since *RRT* is non-deterministic, we ran the algorithm 10 times to find average run-times and costs, when the algorithm is run with certain hyper-parameters. The results are as follows:

Extension Method	Bias	Average Cost of Path	Average Runtime
E1	5%	586.81	1.69[s]
E2	5%	492.69	10.01[s]
E1	20%	606.10	2.24[s]
E2	20%	523.52	6.91[s]

We find the following cumulative distribution function, showing what fraction of runs of the algorithm have found a solution in a given time-frame:

Figure 5: Cumulative Distribution Function



We include sample solutions found by the algorithm with different hyper-parameters:

Figure 6: Solution found using *RRT*, E1, 5% bias

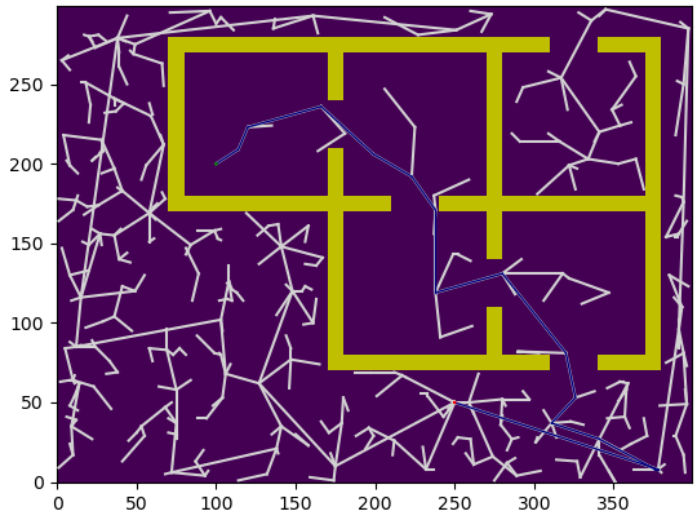


Figure 7: Solution found using RRT, E2, 5% bias

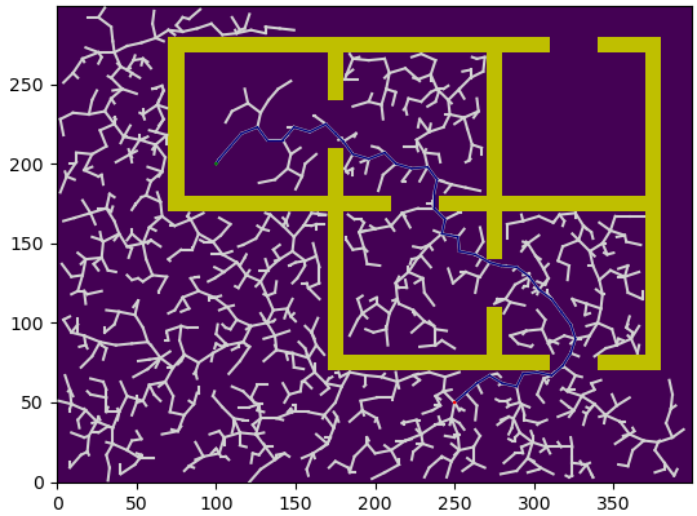


Figure 8: Solution found using RRT, E1, 20% bias

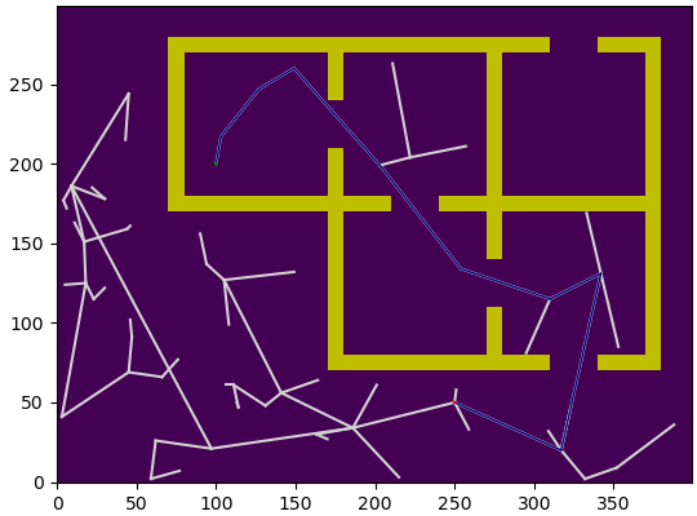
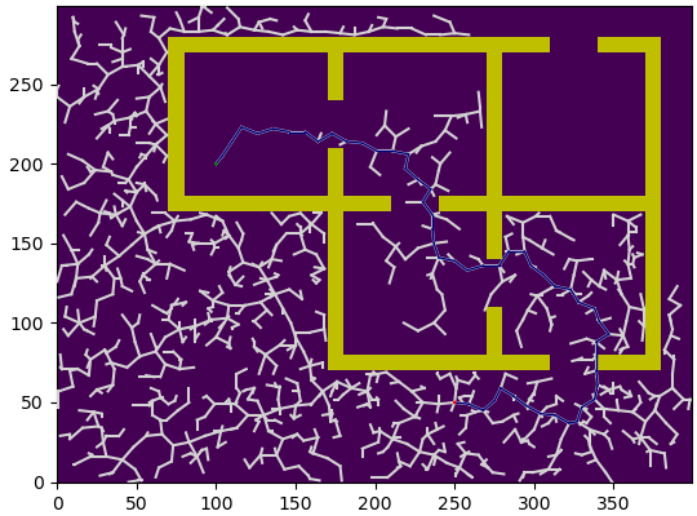


Figure 9: Solution found using RRT, E2, 20% bias



From the data, we find that E1 finds solutions faster that E2. However the downside is that the path found is more erratic, leading to a higher cost on average than using E2.

Furthermore, having a higher goal-bias seems to help find solutions faster, when looking at the CDF. Again, the downside is that the path becomes more costly, since in essence the algorithm becomes more greedy, and cares less about cost once we increase the goal-bias.

Given the above information, we have decided to use E2, since it produces a less erratic solution. E1 may also be less successful in a more cluttered environment.

Furthermore, we have decided to use a goal-bias of 5%. Although the goal-bias of 20% improves the run-time slightly, we decided that the cost of the path was more important to us.

4.4.2 RRT^*

Now we implement RRT^*. The code is very similar to that of RRT, however now we rewire our tree while the algorithm is running. When rewiring the tree, we look at the k nearest neighbours to a vertex we have just added. For each neighbour, we check if it is possible to create an edge between itself, and the new vertex - without collision. if no collision will occur, and the cost to reach the new vertex through the neighbour is less expensive than the cost of the new vertex, we destroy the old edge connecting the vertex to the tree, and add a new edge between the new vertex and the neighbour.

This rewiring process occurs twice, once where we change the edge leading to the new vertex, and again - this time where we check to change the edges of the neighbours themselves.

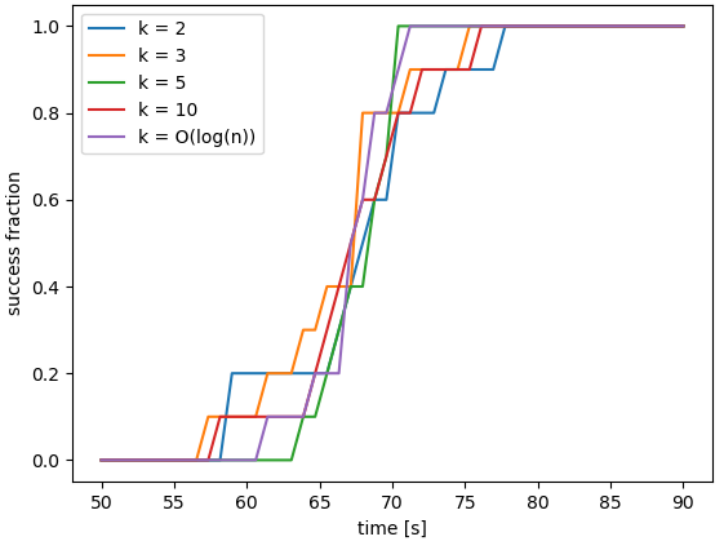
We experimented with various values of k - specifically $k = 2, 3, 5, 10, ciel(log(n))$, where n is the number of vertices currently in the tree.

Below are the results of running the algorithm when the extension method is E2, goal-bias is 5% - and with different values of k, averaged over 10 separate runs:

k	Average Cost of Path	Average Runtime
2	523.49	67.61
3	464.53	66.26
5	386.82	67.78
10	360.68	67.48
$O(log(n))$	365.37045008	67.02

We find the following cumulative distribution function, showing what fraction of runs of the algorithm have found a solution in a given time-frame:

Figure 10: Cumulative Distribution Function



We see that on average, the algorithm finds a solution 100% of the time in around 70[s].

We include sample solutions found by the algorithm with different values of k:

Figure 11: Solution found using RRT^*, E2, 5% bias, k = 2

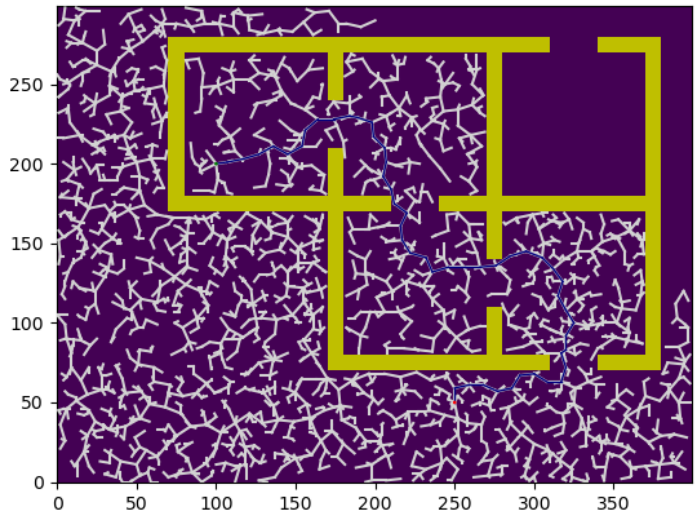


Figure 12: Solution found using RRT^*, E2, 5% bias, k = 3

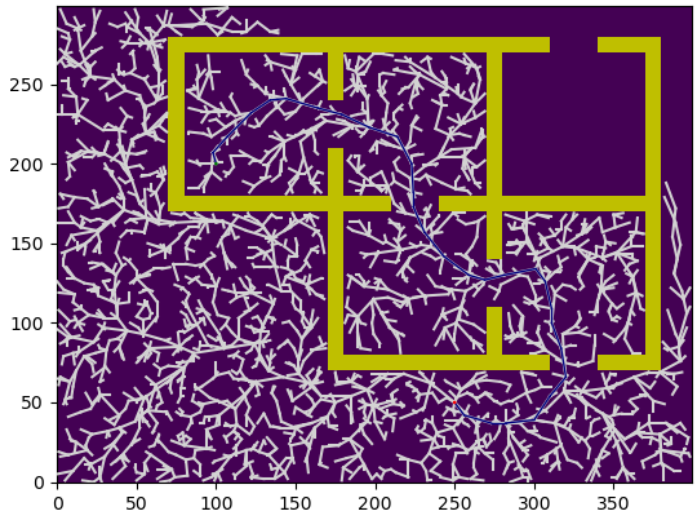


Figure 13: Solution found using RRT^*, E2, 5% bias, k = 5

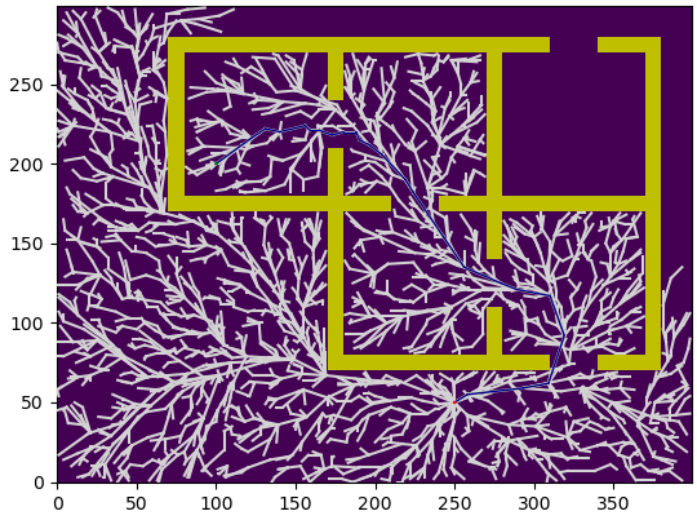


Figure 14: Solution found using RRT^*, E2, 5% bias, k = 10

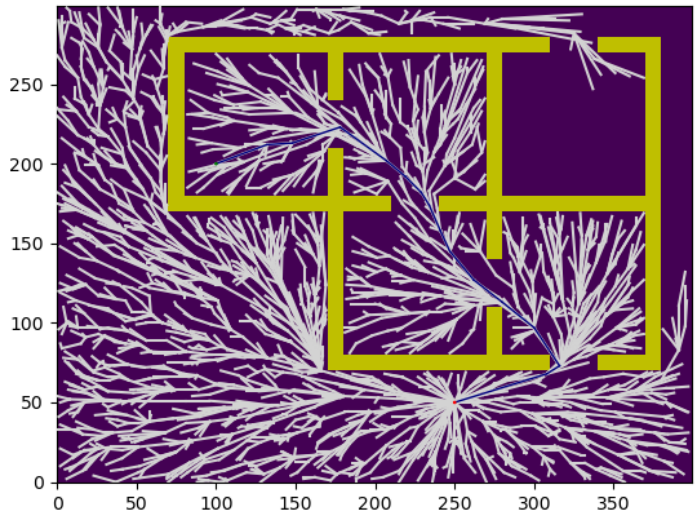
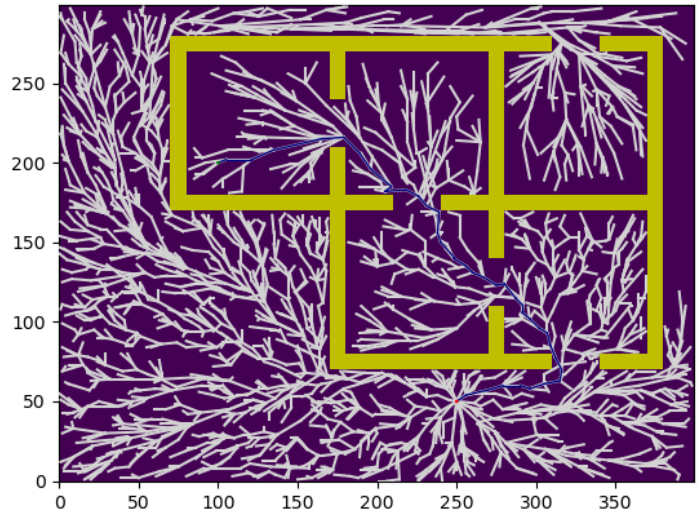
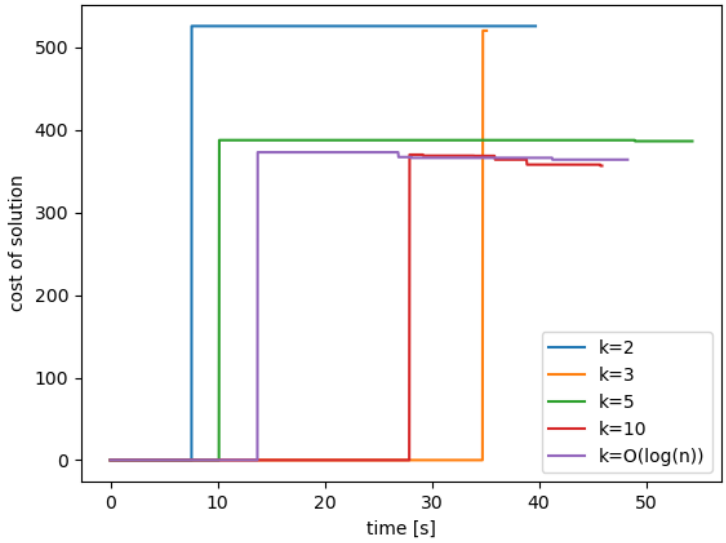


Figure 15: Solution found using RRT^*, E2, 5% bias, k = $O(log(n))$



Finally, we plot the cost of the path found by the algorithm over time, for different values of k:

Figure 16: Solution found using RRT^* over time, E2, 5% bias



We see that for higher values of k, namely $k = 10$ and $k = O(log(n))$, the cost of the path decreases more frequently. Note that we limited the number of iterations to 3000. If we had increased the number of iterations, we would see our algorithms asymptotically converge to the optimal cost.