2360901 - Algorithmic motion planning

HW2

Submitters:

Paulo Khayat – 212747018

Salih Hassan – 212148894

2. Distribution of points in high-dimensional spaces (10 points)

<u>**Warmup (0) points:**</u>

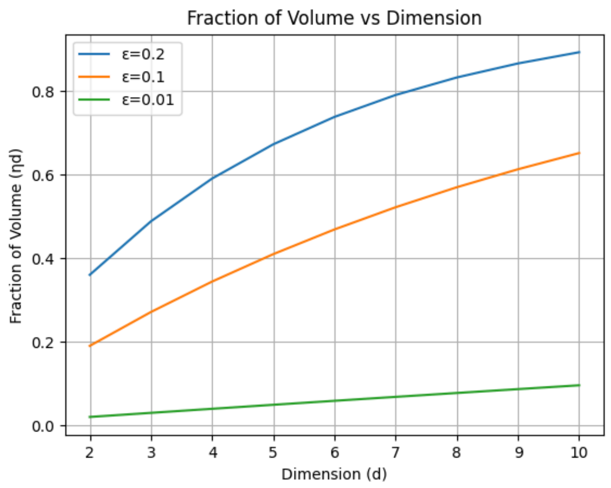
1. In a 2-dimensional unit ball (a disk), around 20% of the volume is located at most units from the surface.

2. In a 9-dimensional unit ball, around 60% of the volume is located at most units from the surface.

In both of the previous cases we received those answers because the Volume of a d-dimensional ball can be described as:$V_d = k_d * R^d$. Where R is the radius of the ball, k_d is some constant.

<u>**Exercise (10) points:**</u>

We calculated and received the following <u>plot</u> for



As we can see in the plot, as the dimension- grows, the fraction of the volume for a given value grows as well.

Discussion: As we saw in class, asymptotically, the nearest neighbor computation dominates the running time of sampling-based algorithms. In addition, from a set of sampled points the number of expected neighbors is . Therefore, as the connection radius decreases the nearest neighbor computation time decreases because the expected number of neighbors decreases as well as apparent from the relationship above, and in turn decreases the overall computation time. Moreover, with a smaller connection radius could more accurately capture the connectivity of the configuration space and potentially lead to a better approximation of the optimal path, especially in cases with narrow passages.

3. Tethered robots (20 points)

1. Given a start and target configuration p_s, p_t, the structure of the path connecting them is a list of nodes. Each node represents a step taken from the previous configuration, starting with p_s as the initial configuration. Each node stores the h-signature of the tether and the current location of the robot, represented by (x, y) coordinates. This way, we can easily check if the robot can reach his target by checking the (x,y) coordinates of the node and whether the tether satisfies the constraints on it as well by looking at its h-signature.

2. To encode the tether's description we'll compute its h-signature:

For each obstacle $\{o_i\}_{i=1}^{n}$, we'll extend a ray towards $y = \infty$ from its center of mass, and if the tether crosses the ray from left to right then we'll append "t_i" to the h-signature, and if it crosses it from right to left then we'll append "$\overline{t_i}$" to the h-signature. This ensures that all possible configurations of the tether, including circling around obstacles multiple times, are captured. Since the tether remains taut at all times, slack does not need to be accounted for. By encoding the h-signature in this manner, we differentiate between configurations where the robot is in the same location but with distinct tether descriptions.

3. Let's describe an algorithm that given a graph $G = (V, E)$, returns a graph G_h, which stores for each node the homotopy classes that can reach it.

We'll define the set of vertices in this new augmented graph to be
$V_h = \left\{\left(v,\ w_v\right):\ v\ \in V,\ w_v\ is\ the\ corresponding\ h\ -\ signature\ of\ the\ tether\ for\ vertex\ v\ in\ G\right\}$

And the set of edges to be:

$E_h = \left\{ \left(\left(v_1, w_1\right), \left(v_2,\ w_2\right) \right) :\ \left(v_1,\ v_2\right)\ \in V,\ h\left(w_1\ \cdot s(v_1, v_2)\ \right) = h(w_2) \right\}$ such that $h(\bullet)$ is its reduced signature, $s(v_1, v_2)$ is the signature of the trajectory associated with connecting $\left(v_1, v_2\right)$, calculated by iterating the obstacles and seeing which ray the line v_1 −> v_2 crosses, and from which direction.

In order to build this graph, we will create a visibility graph out of the obstacles and the rope origin: $H = visibility(OBST \cup \{p_b\})$.

Now, we will run a dijkstra-like algorithm on a new graph G_w as we build it.

Init:

$G_w = \{(p_b, w_b = \epsilon), \{\}\}$

OPEN (heap) = [$\{(p_b, epsilon), 0\}$] //stores $(node \in G_w, distance)$

CLOSE (set/hash) = []

Loop: while OPEN isn't empty and OPEN[0].distance <= L

- Pop the element (v_{new}, w_{new}), with distance d_{new} of the OPEN with the smallest distance, place the node (v_{new}, w_{new}) into the CLOSE list and expand the node as follows:
- For every neighbor v' of v_{new} in the visibility graph H, we calculate $w' = h(w_{new} \cdot s(v_{new}, v'))$ and:
    - If $\{(v', w'), d'\} \in OPEN$, and $d' > d_{new} + euclidian(v_{new}, v')$ update the distance of (v', w') in the OPEN to be $\{(v', w'),\ d_{new} + euclidian(v_{new}, v')\}$
    - If $(v', w') \notin OPEN$ and $(v', w') \notin CLOSE$, add to the OPEN: $\{(v', w'), d_{new} + euclidian(v_{new}, v')\}$, and $V_w = V_w \cup \{(v', w')\}, E_w = E_w \cup \{\left(\left(v_{new}, w_{new}\right),\ (v',\ w')\right)\}$

This graph G_w denotes for every obstacle vertex, the rope h-invariant classes reachable to it.

Now, for each $v \in V$, we list all the obstacle vertices v' visible to it. Find all the possible nodes $(v', w') \in V_w$. for each, add $V_h = V_h \cup \{(v, h(w' \cdot s(v, v')) \mid d' + euclidian(v, v') \leq L\}$ (the taut rope will be a direct line to one of these obstacle vertices, so this if statement is enough)

Then, $E_h = \{((v_1, w_1), (v_2, w_2)) \mid v_1, v_2 \in V_h\ ;\ w_2 = h(w_1 \cdot s(v_1, v_2))\}$. In words, an edge will exist in the new path if moving from (v_1, w_1) to v_2 would result in a h-class of w_2.

Return $G_h = \{V_h, E_h\}$

Note 1: In short, The nodes in G_w are each homotopy class a vertex can have for each obstacle vertex. The nodes are extended by finding the shortest distance node, and checking the visibility graph of the obstacles, And the algorithm terminates once every homotopy class for each node has been expanded, until the only ones left in the open are of best-case distance greater than L. This is monotonically increasing and non negative, so we will find all the vertices with less than L length taut ropes/ h-classes.

Note 2: Once we have the nodes for the obstacles/origin, we just need to draw a line to the original graph, from each h-class node, and create a vertex for it. We implicitly assume we saved d_{new} here, we could have added it to the close/graph but the notation is already as complicated as it is.

Note 3: We put an older version of this answer in the appendix, that we changed to when we were unsure about being able to run a visibility graph on the obstacles. It was an interesting alternative solution in our opinion, however some kinks in the calculations make it suboptimal.

4. Given the graph $G = (V, E)$ build its visibility graph $G^{vis} = (V^{vis}, E^{vis})$ and then build the homotopy graph of the visibility graph $G_h^{vis} = (V_h^{vis}, E_h^{vis})$ using the algorithm we described in the previous section.

- Assuming we aren't duplicating already existing nodes in the graph $G_h^{vis}(V, E)$, we will add $\left(p_s, w_s\right)$, then, add $V_h^{vis} = V_h^{vis} \cup \{\left(p_t, h(w \cdot s(v, p_t))\right) | \, (v, w) \in V_h^{vis} \, S.T. \, (v, p_t) \, unobstructed\}$ (in words, every vertex with line of sight to Pt, add the (p_t, w) where w is the h-signature from moving from that configuration to this point)
- For every vertex $\left(v_h, w_h\right) \in V_h^{vis}$, we check if we can connect between p_s, v_h by checking if $h\left(w_s \cdot s(p_s, v_h\right) = h\left(w_h\right)$ and that the tether's length does not exceed L. if both conditions are met then we add an edge between them to the graph by inserting $\left(\left(p_s, w_s\right), \left(v_h, w_h\right)\right)$ to E_h^{vis}.

  For the target, for every vertex $\left(v_h, w_h\right) \in V_h^{vis}$, we check if we can connect between v_h, p_t by checking if $h\left(v_h \cdot s(v_h, p_t\right) = h\left(p_t\right)$ and that the tether's length does not exceed L. if both conditions are met then we add an edge between them to the graph by inserting $\left(\left(v_h, w_h\right), \left(p_t, w_t\right)\right)$ to E_h^{vis}.

After doing the required changes to the graph, we dijkstra (where the edges are euclidian distance!) on the G_h^{vis} starting at p_s and run until the algorithm stops or we've reached p_t. If the algorithm reports a finite path length, we can reconstruct the path by tracing back from the target node to the source node using the parent pointers stored during the algorithm's execution. Finally, we reverse the list of vertices to obtain the correct path and return it.

4. Motion Planning: Search and Sampling (70 points)

4.3 A* Implementation

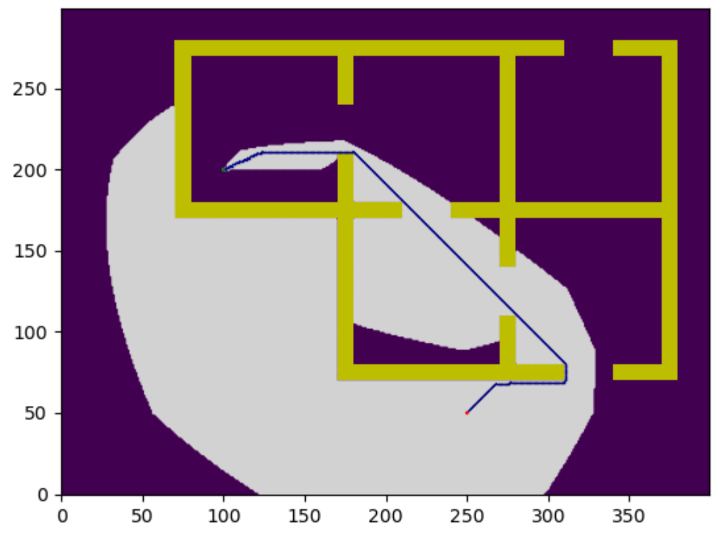
3.

Epsilon = 1:

Number of developed nodes: 43728

Number of iterations: 43728

Runtime: 83 seconds



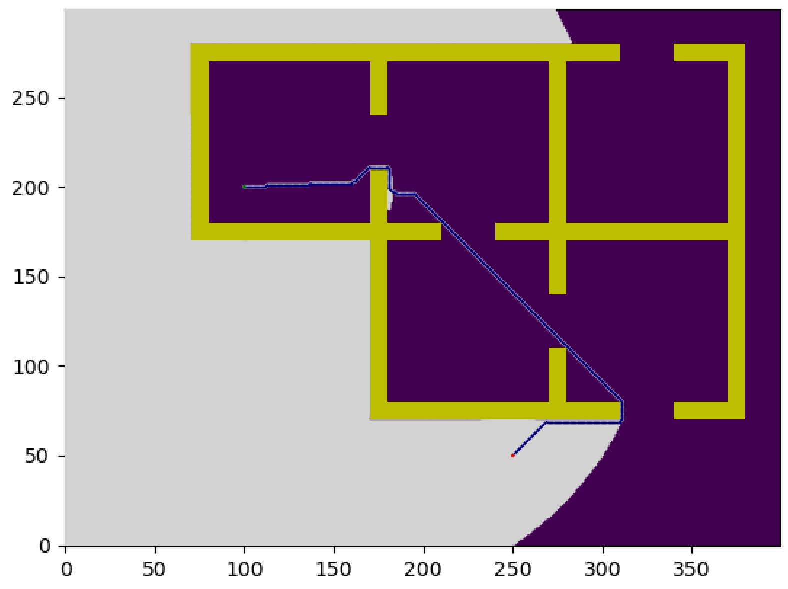
.

Epsilon = 10

Number of developed nodes: 50320

Number of iterations = 407035

Runtime: 834 seconds

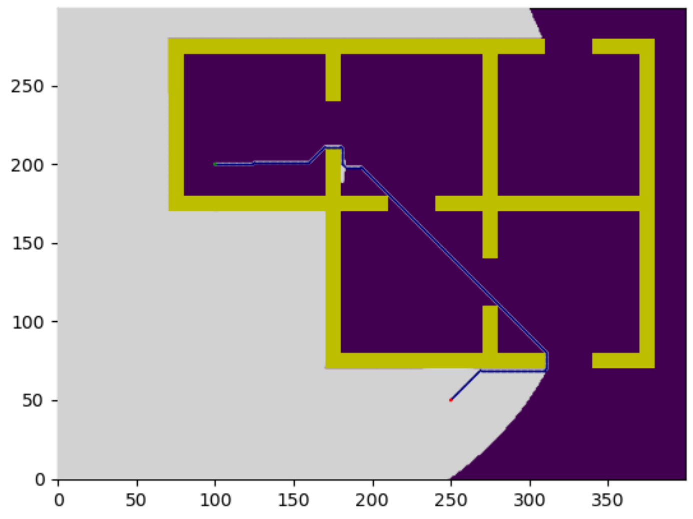


Epsilon = 20

Number of developed nodes: 50700

Number of iterations = 475899

Runtime: 963.4 seconds



4. When we raise the epsilon, the algorithm will try advancing more directly to the goal. However, this causes the algorithm to enter a local minimum (behind a wall yet near the goal). Additionally, there is a very large amount of nodes that are recalculated,

4.4 RRT and RRT* Implementation:

In our implementation we assumed that the goal is reachable from the start state. The following results were received on map 2, and for the part where we had to pick a step size for the “extend function” with a parameter of “ext_mode=E2” we picked n=19 to limit the step size.

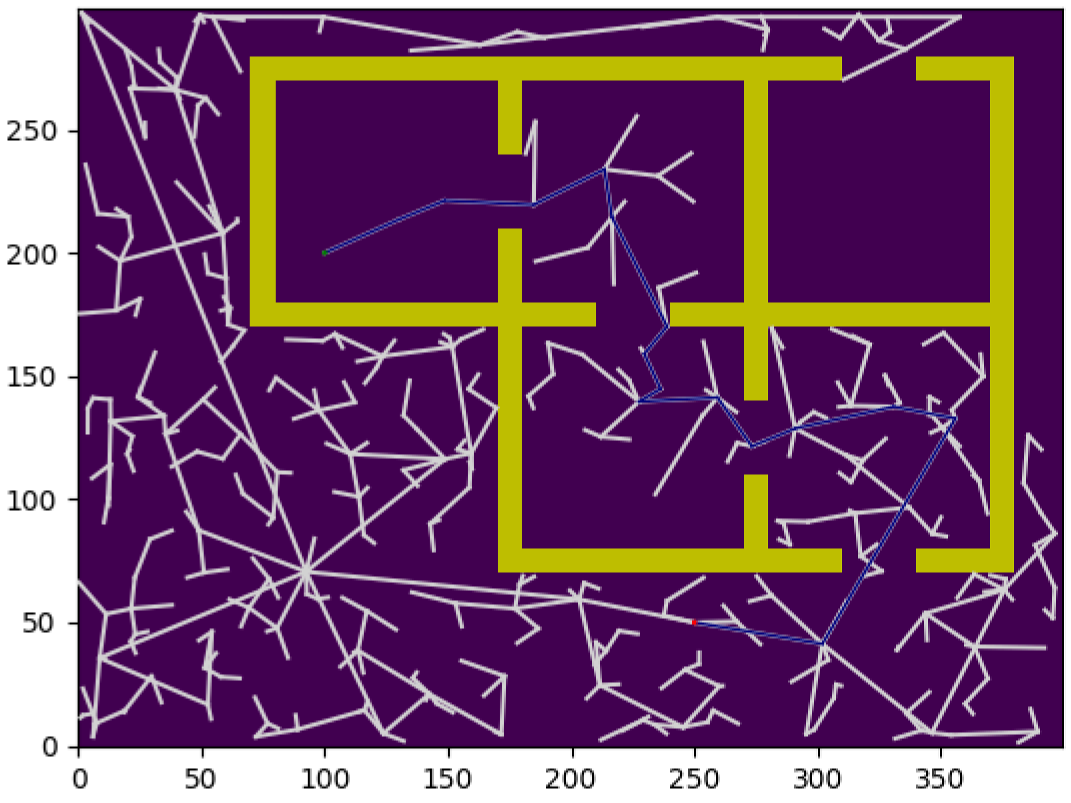
RRT:

The following is a table summarizing the average results we received over 10 runs when running the RRT algorithm to find the solution with the corresponding bias to pick the goal:

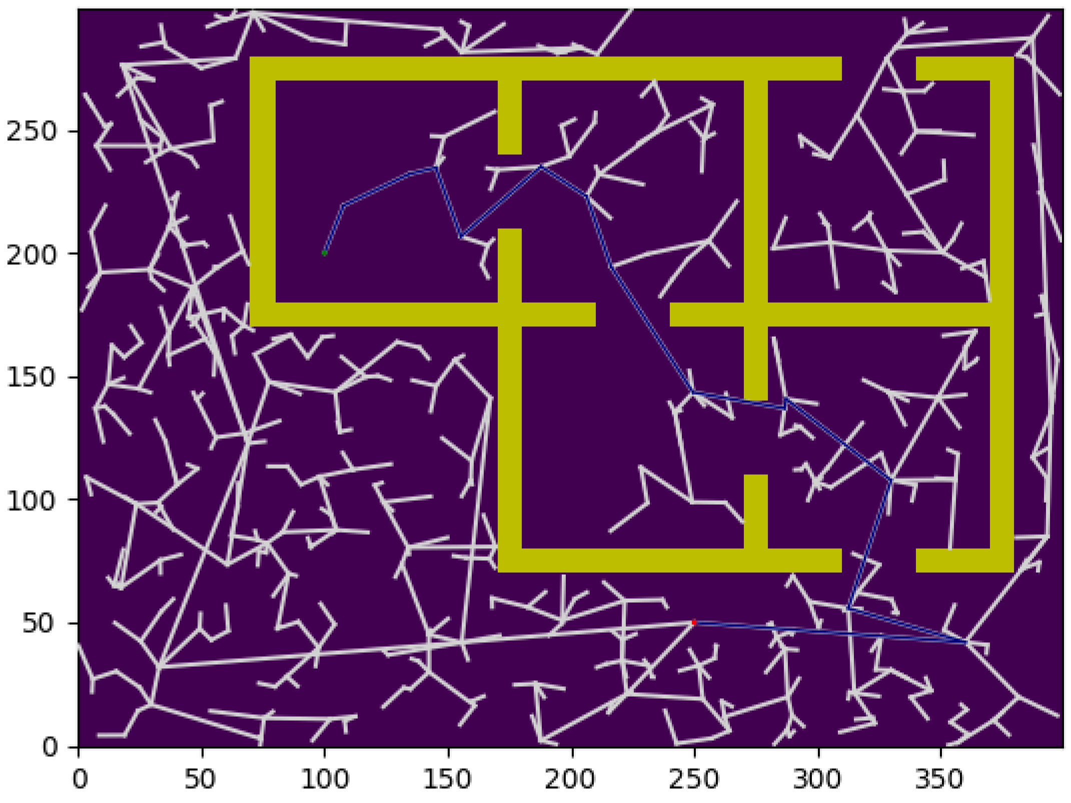
Bias to pick the goal	Performance (cost)	Performance (time) [seconds]
5%	520.67	27.79
20%	542.67	12.61

And the following figures shows the final state of the tree for both values:

For bias = 0.05 we received the following tree:



For bias = 0.2 we received the following tree:

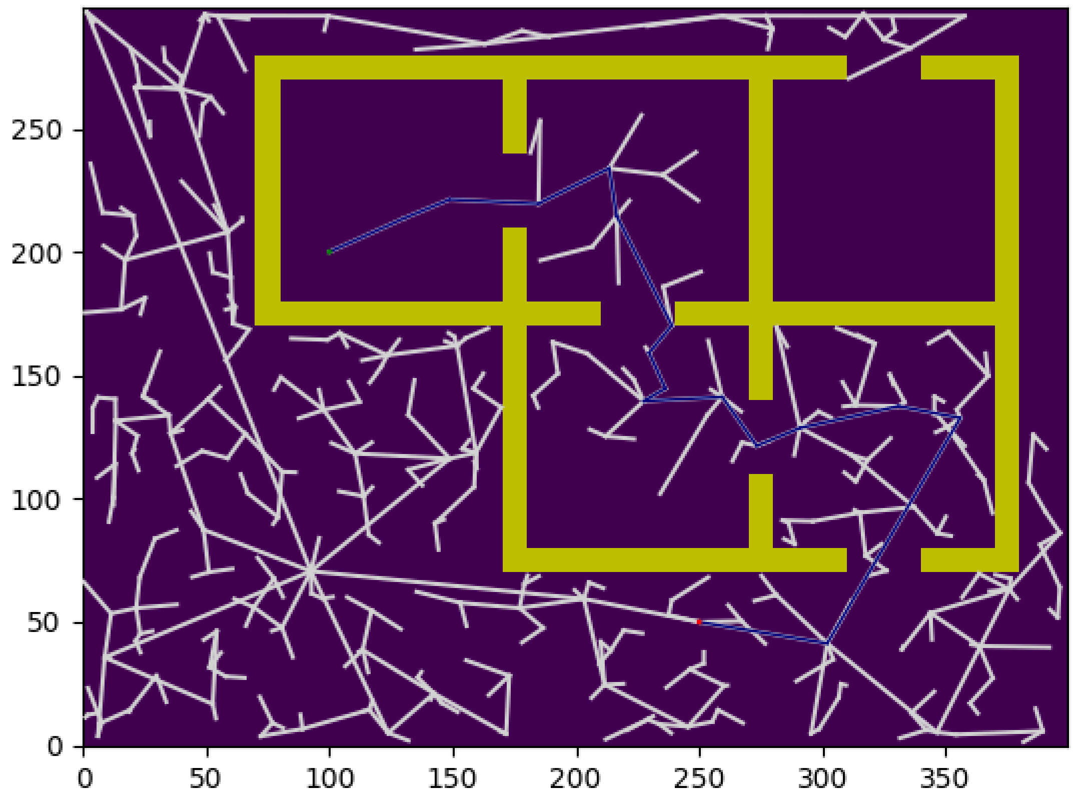


The following is a table summarizing the average of the results we received for the different versions of the extend function, and biasing values over 10 runs:

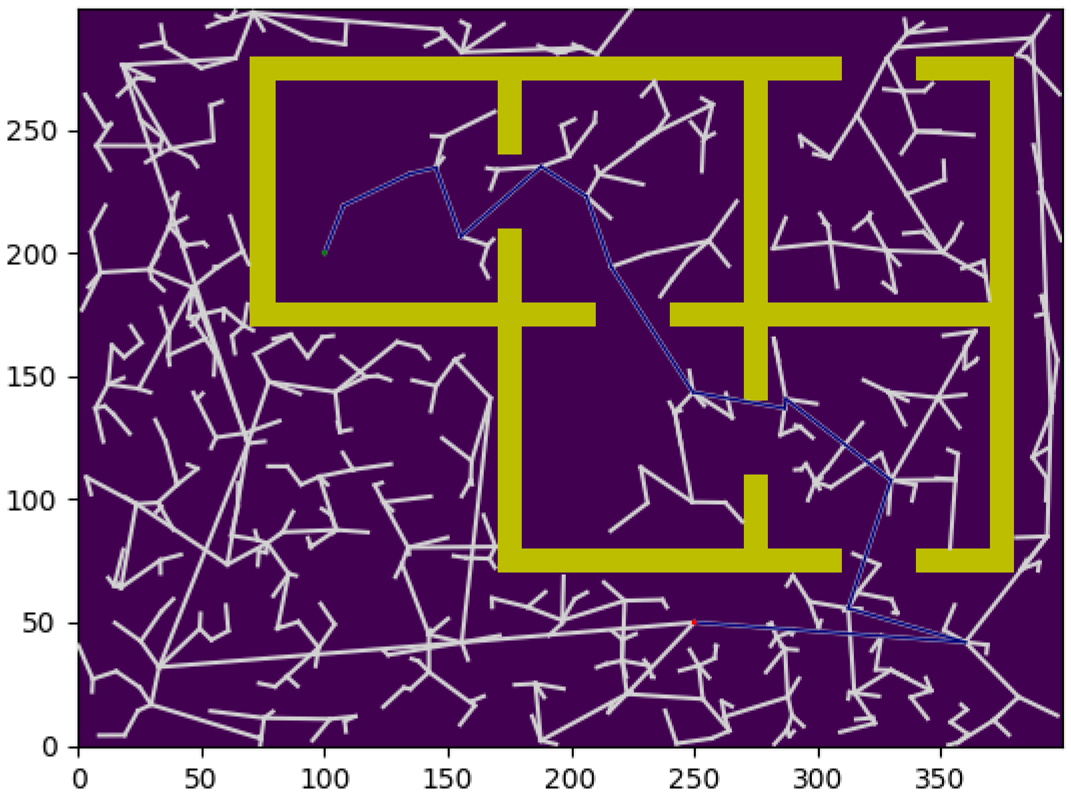
Bias to pick the goal	Extension Policy	Performance (cost)	Performance (time) [seconds]
5%	E1	520.67	27.79
5%	E2	619.68	30.10
20%	E1	542.67	12.61
20%	E2	619.82	10.37

The following figures show a representative picture of the tree for with the given biasing parameter and extension policy:

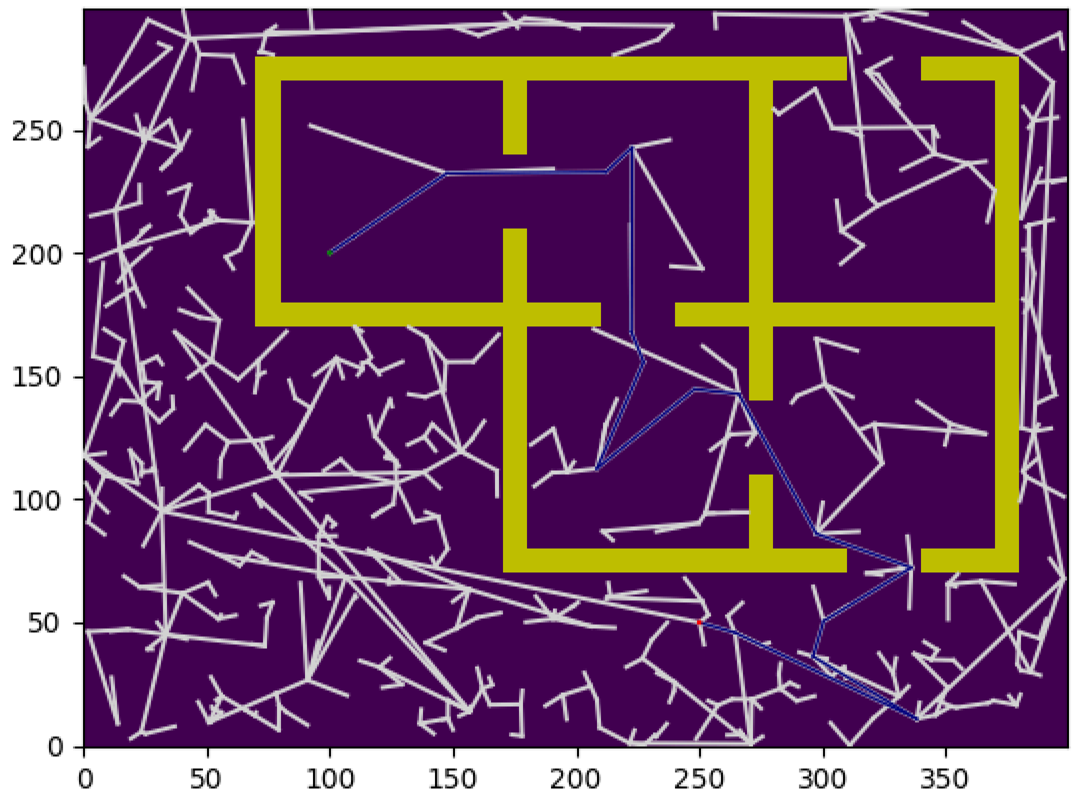
For bias = 0.05, extension policy = “E1”:



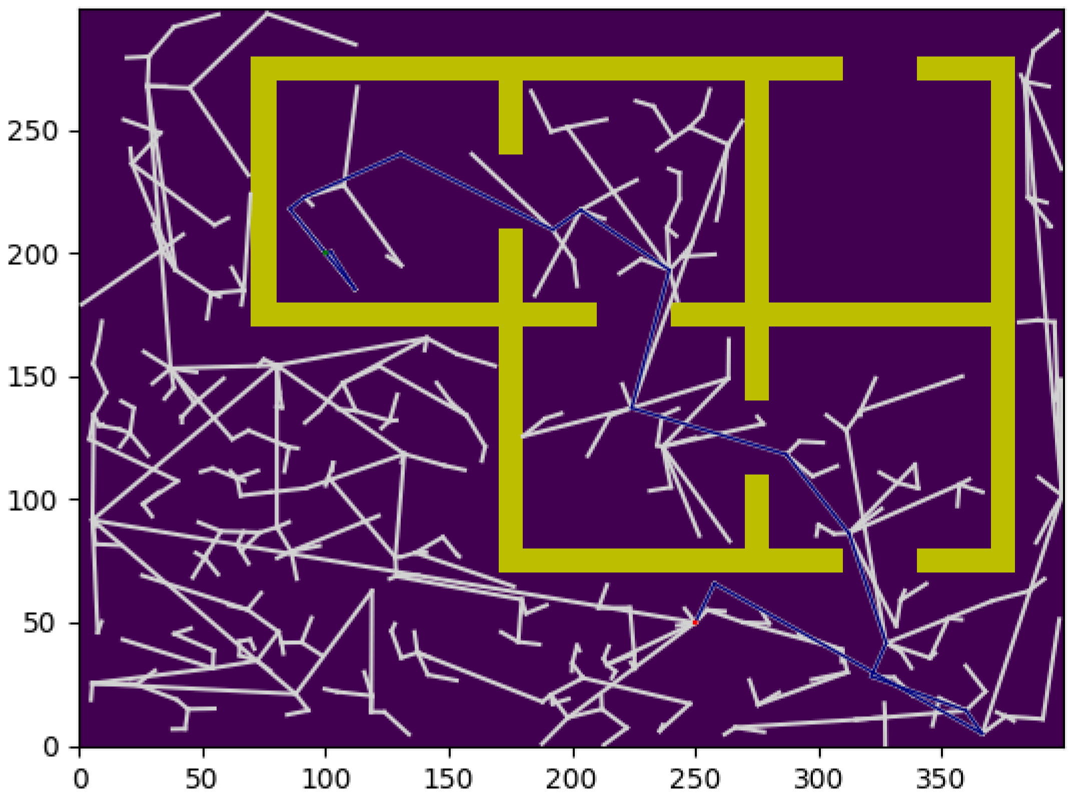
For bias = 0.2, extension policy = “E1”:



For bias = 0.05, extension policy = “E2”:



For bias = 0.2, extension policy = “E2”:



In practice, we opt for the "E1" extension policy and set the biasing factor to "0.2". This choice strikes a balance between the computation time required to find a solution and the quality of the resulting path. By doing so, we aim to achieve satisfactory performance in terms of both computation time and solution quality compared to alternative biasing factors and extension policies that were tested. However, it's important to note that our analysis is based on a limited number of runs(10). To obtain a more reliable assessment of the effectiveness of different parameters, it would be advisable to conduct experiments with a larger number of runs and average the results accordingly.

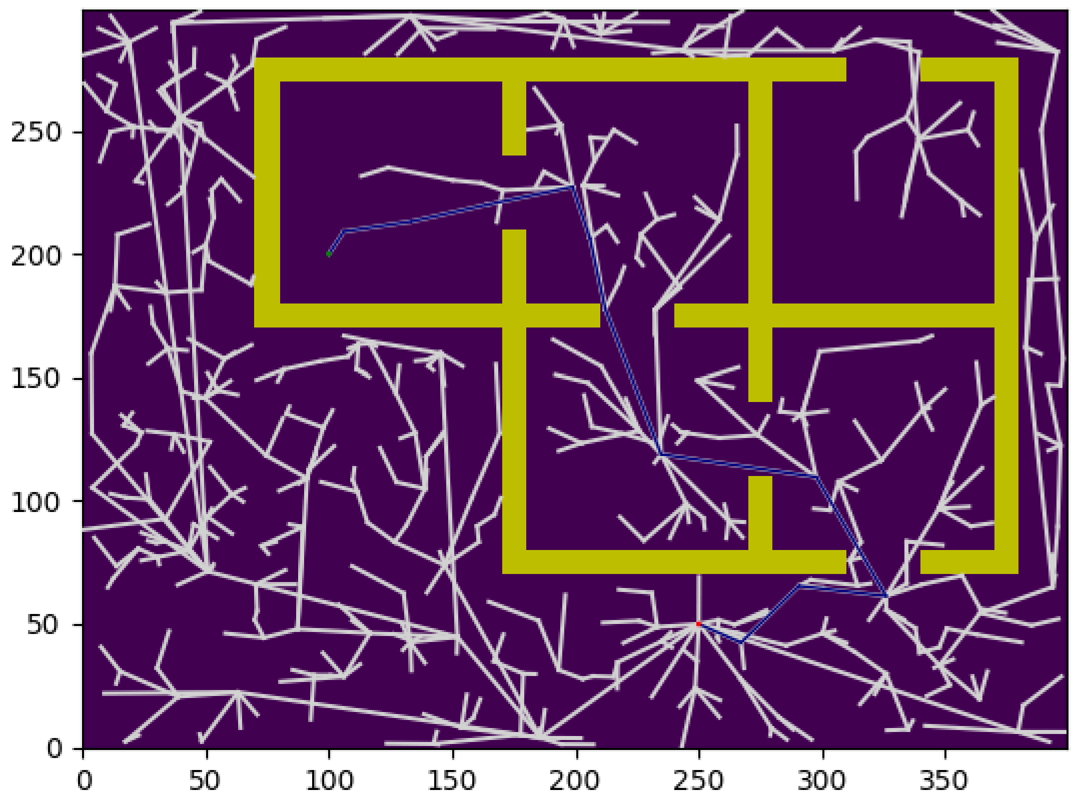
RRT*:

In this section, we'll showcase the results we received for RRT* with the different parameters of "k", we ran our algorithm with extension policy = "E1" and goal biasing = "0.2" as they showed to be most promising in the previous section, and received the following results on average:

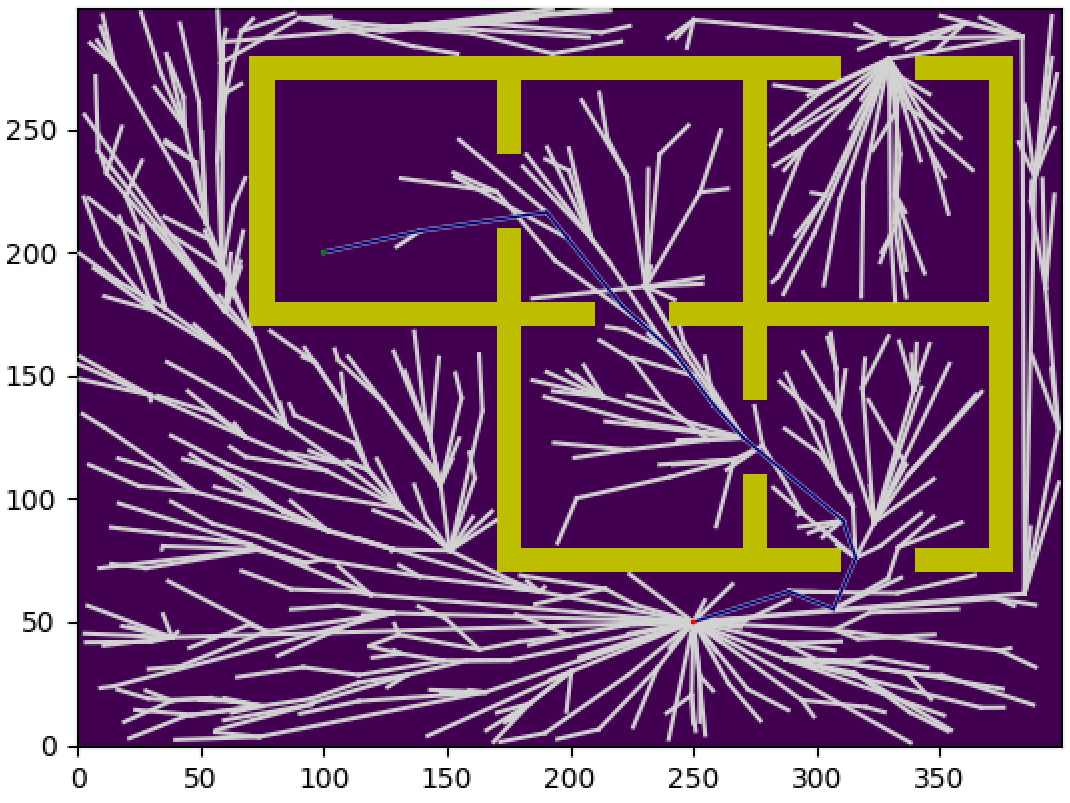
	k=3	k=11	k=47	k=73	k=log(n)
Performance (cost)	437.05	374.16	368.35	363.37	425.59
Performance (time) [seconds]	17.45	16.80	15.41	30.16	10.92

The following figures showcase a representative picture of the trees we received as results of running the RRT* algorithm:

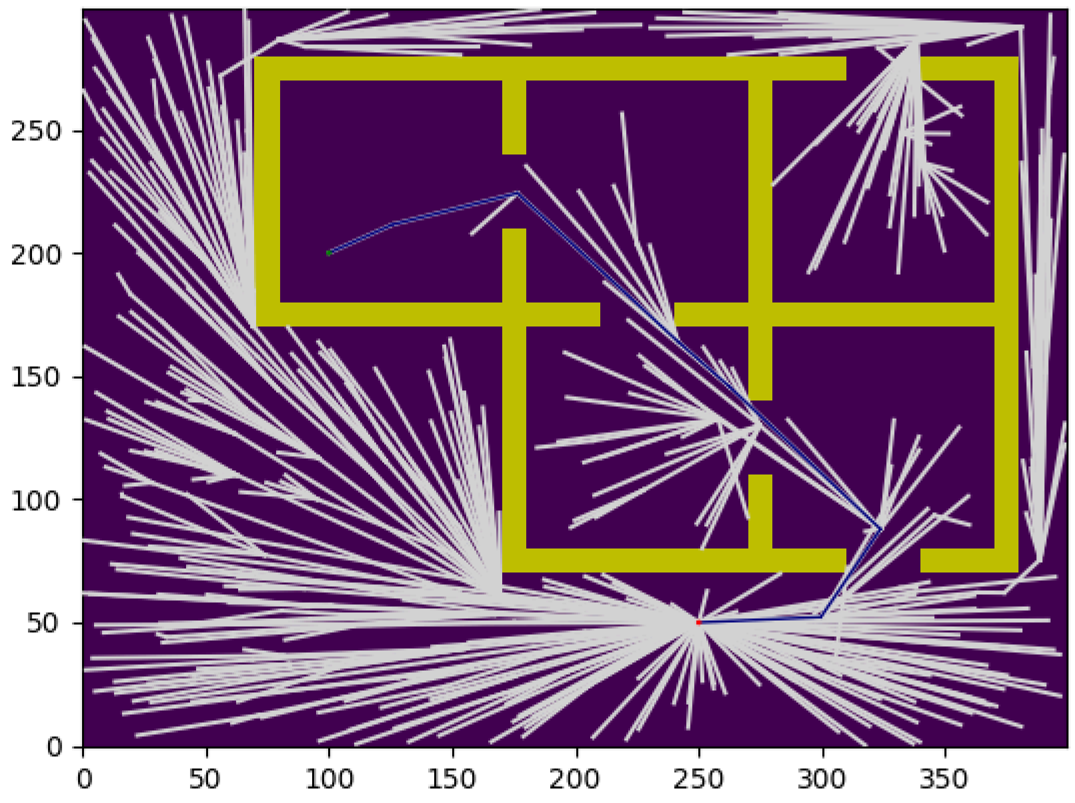
I. K = 3:



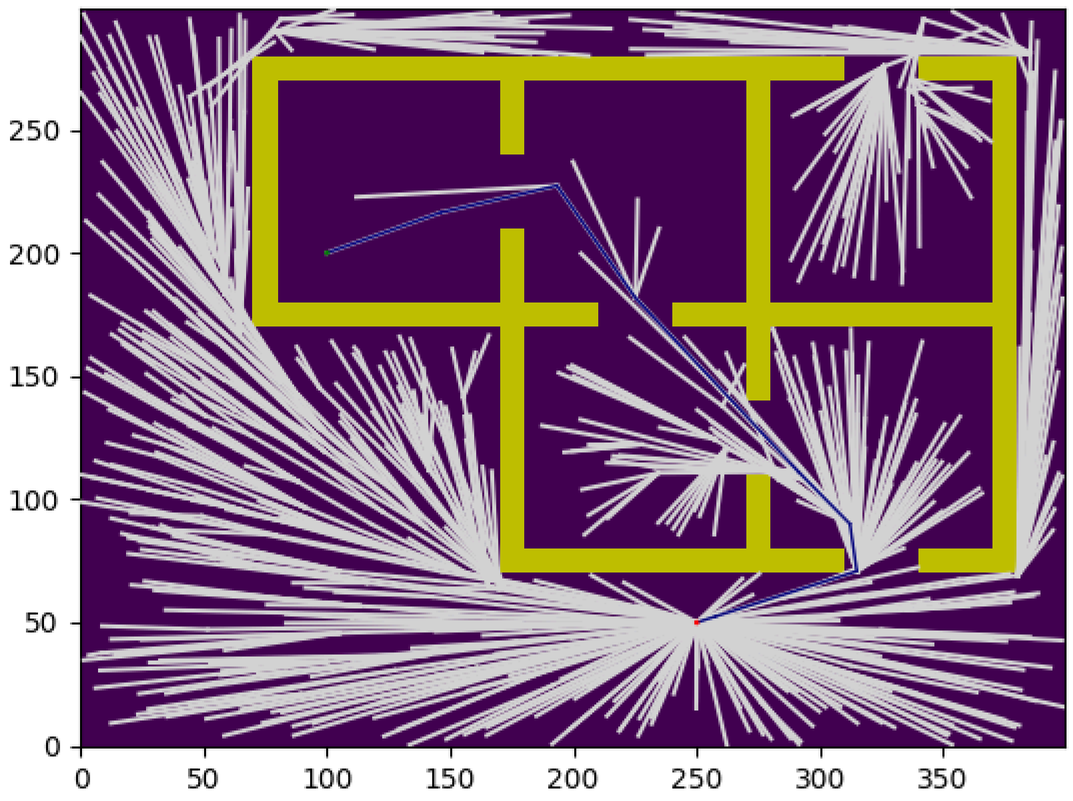
II. K = 11:



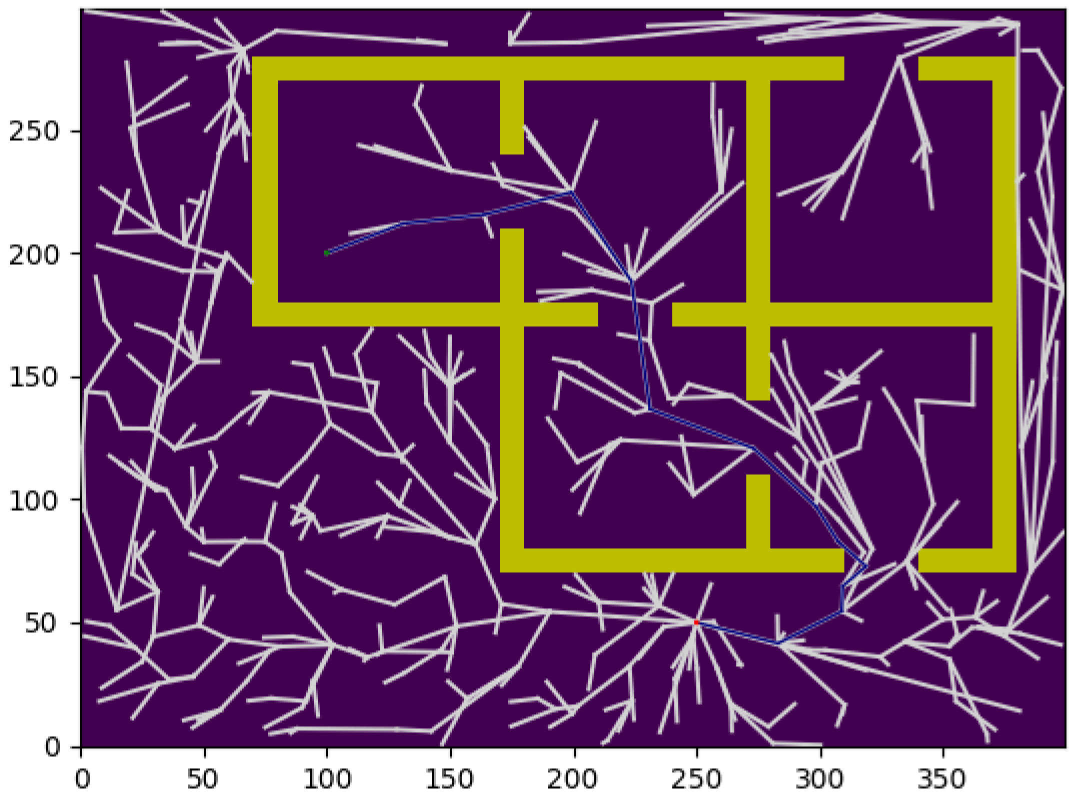
III. K = 47:



IV. K = 73:

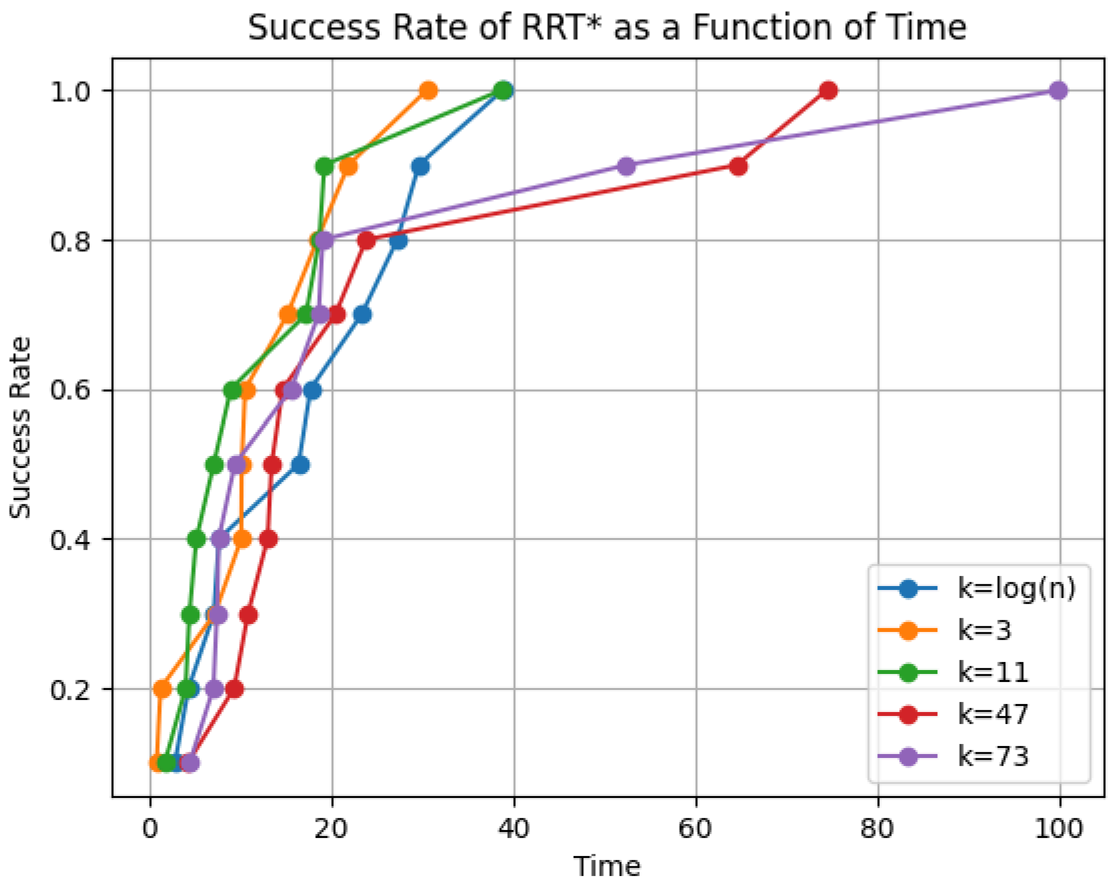


V. K = log(n) where n is the number of vertices in the tree:



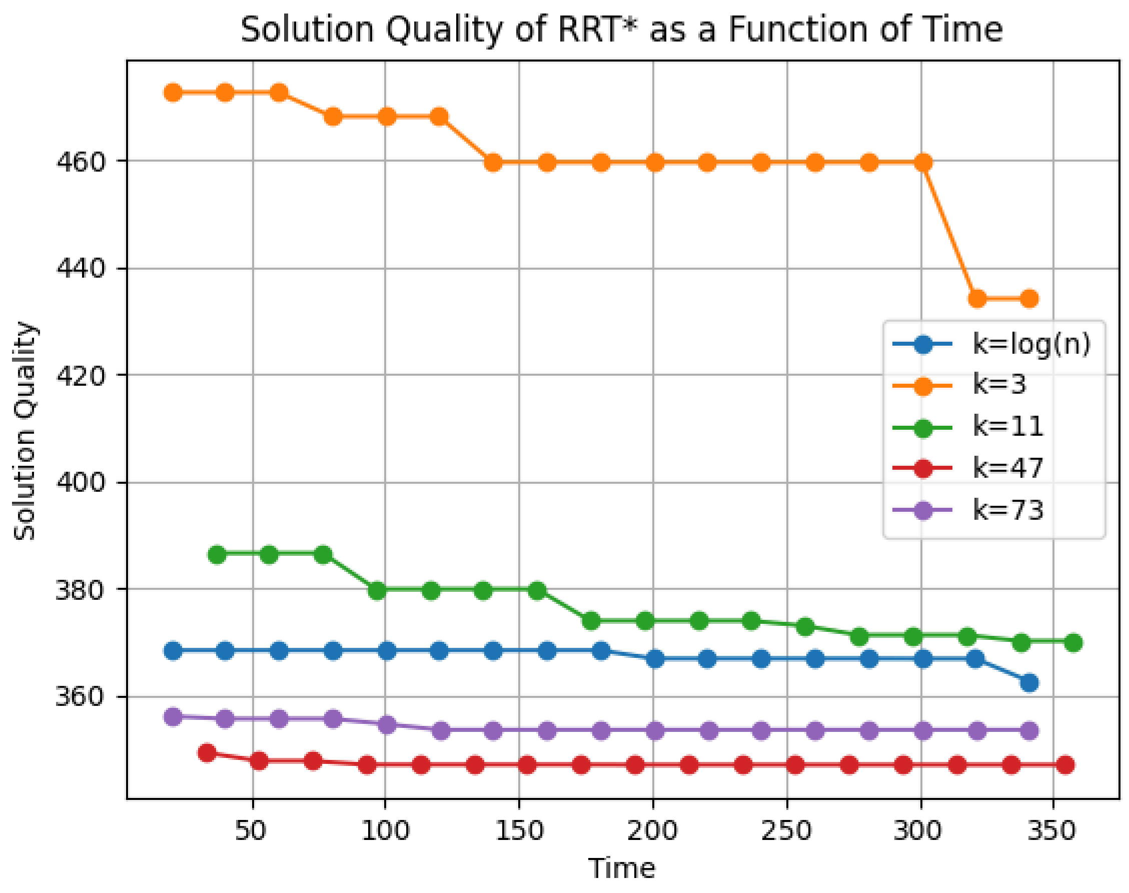
In practice, we would pick k = log(n), where n is the number of vertices in the tree, because it provides a more dynamic approach. With this approach, k adapts to the environment of the map, whereas with constant k values, its effectiveness may vary depending on the map itself.

And the following is a plot of the success rate as a function of time for the different k values we mentioned above:



As we can see in the plot, all lines succeed with probability 1 at some point because RRT* is probabilistically complete.

And the following is a plot of the solution quality as a function of time for the different k values we mentioned above:



Note that the solution quality in the plot is represented by the length of the path, and the shorter the path, the better the quality.

Appendix: Q2.3 which is different but can be too expensive (not to be graded- just for show)

3. Let's describe an algorithm that given a graph $G = (V, E)$, returns a graph G_h, which stores for each node the homotopy classes that can reach it.

We'll define the set of vertices in this new augmented graph to be
$V_h = \left\{\left(v,\ w_v\right):\ v\ \in V,\ w_v\ is\ the\ corresponding\ h\ -\ signature\ of\ the\ tether\ for\ vertex\ v\ in\ G\right\}$

And the set of edges to be:

$E_h = \left\{\ \left(\left(v_1, w_1\right),\ \left(v_2,\ w_2\right)\right):\ \left(v_1,\ v_2\right)\ \in V,\ h\left(w_1\ \cdot s(v_1, v_2)\ \right) = h(w_2)\right\}$ such that $h(\bullet)$ is its reduced signature, $s(v_1, v_2)$ is the signature of the trajectory associated with connecting $\left(v_1, v_2\right)$, calculated by iterating the obstacles and seeing which ray the line v_1 $-> v_2$ crosses, and from which direction.

LENGTH(start, h-class, vertex) calculates the length of the taut rope with the end being at the vertex, in a specific configuration

In order to build this graph, we will run a dijkstra-like algorithm on G_h as we build it, using the edges of G.

Init:

Find: For every vertex $v_i \in V,\ S.T.\ euclidian(v_i, p_b)\ <=\ L$, calculate the shortest rope configuration from the tether start to it. (we can do this by taking the direct line, if it collides with an obstacle, we calculate the length for if we go above, or under that obstacle, which means, adding $t_i/\overline{t_i}$ - going above or not including it- going under),

$G_h:\ V_h =\ \{(p_{init}, w_b\ =\ s(p_b, p_{init}))\};\quad E_h =\ \{\}$

OPEN (heap) = $[(v_i, w_i), LENGTH(p_b, w_i, v_i)]$ //stores $(node\ \in\ G_h, ropelength)$

CLOSE (set/hash) = []

Loop: while OPEN isn't empty and OPEN[0].length <= L

- Pop the element (v_{new}, w_{new}), with length d_{new} of the OPEN with the smallest length, place the node (v_{new}, w_{new}) into the CLOSE list and expand the node as follows:

- For every neighbor v' of v_{new} in G, we calculate h-class $w' = h(w_{new} \cdot s(v_{new}, v'))$ and:
    - If $(v', w') \notin CLOSE$, and $LENGTH(p_b, w', v') \leq L$
      $V_h = V_h \cup \{(v', w')\}$, $E_h = E_h \cup \{\left(\left(v_{new}, w_{new}\right), (v', w')\right)\}$
      And if not in the OPEN, add to it: $\{(v', w'),\ LENGTH(p_b, w', v')\}$

Return $G_h = \left(V_h, E_h\right)$

Note 1: In short, The nodes are each homotopy class a vertex can have for each vertex in G, We take the current node, find its neighbors in the original graph and calculate the new h-class of the end node.The algorithm terminates once every homotopy class for each node has been expanded, until the only ones left in the open are of best-case length greater than L.

Note 2: The initial setup of the OPEN list is necessary: since we aren't given the ps and pt, so if we try to start djiktra blind from any point, we may have vertices that are reachable, hidden behind a vertex that is too far from the origin. We may have a forest of connected graphs, that we'd like to calculate the h-invariant classes for each.

Note 3: There is no need to improve vertices in the open, because the calculation is already optimal (Length())

Note 4: To figure out what homotopy classes can be used to reach a vertex, run through the closed (or V_h) and check each node that includes the desired vertex