Algorithmic Robot Planning
Homework 2

Andrew Elashkin, Yonatan Sommer

December 2022

Q2 Distribution of points in high-dimensional spaces

Warm up

1.

(c) Roughly 20%

2.

(d) Roughly 60%

Exercise

We present figures 1, 2, 3 with the fraction of the volume as a function of d. What is evident from these results is that as d increases, the fraction of the volume that is ϵ distance from the boundary grows to significant proportions of the ball relative to ϵ. In the sampling based algorithms learned in class the bottleneck in terms of runtime and computational complexity was the collision check performed on the nearest neighbours. If the connection radius could be decreased - even by a small (epsilon) relative distance - since a significant proportion of the ball wouldn't need to be processed, we would need to perform less collision checks (since we would lose all the neighbours in the fraction of the ball not searched) and the runtime and computational complexity can improve meaningfully.

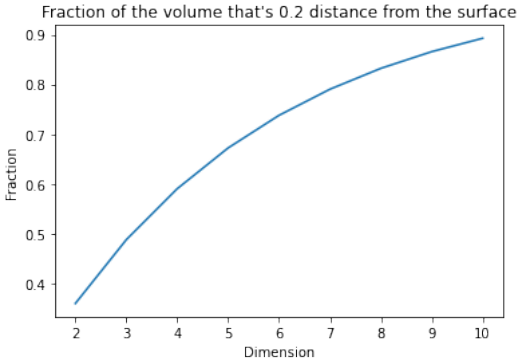

Figure 1: Epsilon = 0.2

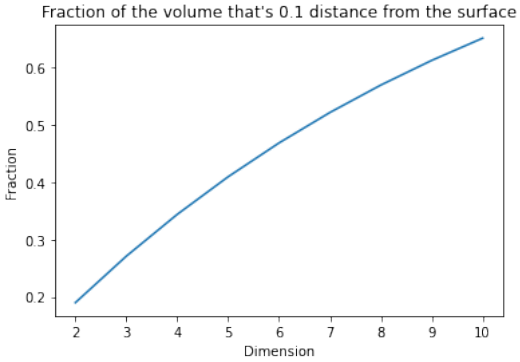

Figure 2: Epsilon = 0.1

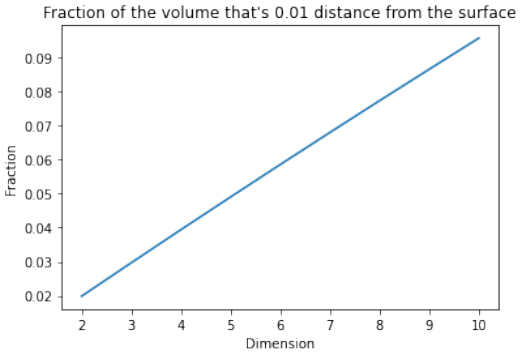

Figure 3: Epsilon = 0.01

3. Tethered robots

1. The structure of the shortest path from a start to target configuration will consist of robot starting location p_s, tether descriprton in form of h-invariant and the set of nodes that connects p_s to p_t including p_t with their h-invariants respectfully.

2. The efficient way to encode the tether's description would be the use of h-signature to represent different homotopies of the same locations. This can be done by extending a vertical ray from each obstacle towards $+\infty$ and considering the tether crossing of those rays. Each point will have a homotopy string s representing the homotopy class of the tether's configuration, and with the anchor point of the tether that would encode the tether.

3. The nodes will be (u, s) where $u \in V$ is a node in the graph G (presumably containing an (x, y) location of the robot in our case), and s is a homotopy string representing the homotopy class of the tether's configuration. We need to define two tools to use in our Dijkstra algorithm: Define $f_{hc}(u, v)$ to be a function that for each $(u, v) \in E$ returns the "homotopy-delta" when traversing the edge (u, v), meaning the string representing the homotopy class of traversing that edge (as defined in class based on the vertical lines crossed). It is important to note that though G is an undirected graph, $f_{hc}(u, v)$ is directional and $f_{hc}(u, v) = \overline{f_{hc}(v, u)}$. We can calculate this function for each $e \in E$.
   Define $len(s)$ to be a function that given a homotopy string, returns the minimum length of the tether given the configuration that matches that homotopy class. We can calculate this by calculating the poly-line placement of the tether around the obstacles as described by the homotopy class represented by the string s.
   At the start of the algorithm, define $G_h = (V_h, E_h)$ where V_h and E_h are initiated as empty sets. The extension of a node (v, s) will be as follows:
   For each neighbor u of v in G: Define a new node to be $(u, s + f_{hc}(v, u))$. The homotopy string concatenation $s + f_{hc}(v, u)$ follows the rule that if t_i is followed by $\overline{t_i}$ they are removed from the string as we saw in class. If $len(f_{hc}(v, u)) \leq L$ and $(u, s + f_{hc}(v, u))$ isn't in Open or Closed lists, we add it to Open. Also, create a node $(u, s + f_{hc}(v, u))$ and add to V_h, and add the edge $((v, s), (u, s + f_{hc}(v, u)))$ to E_h.
   Terminate when Open is empty and return G_h.

4. We will use the algorithm from paragraph 3 and the homotopy-augmented graph G_h^{vis} constructed by that algorithm to solve the motion-planning problem for a point tethered robot. Since the start point and end point are not necessarily nodes of the initial visibility graph, we need to add them to G_h^{vis}.
   First, we add the starting point p_s to the graph. For every vertex $p_h \in G_h^{vis}$ we check if we can connect it to p_s by checking that the extension of the h-invatiant w_s matches the h-invariant w_h of p_h. If it is matched, we check if the length of a tether is less than L. If both of those conditions are met we add the edge between p_s and p_h.
   Second, for each vertex $p_h \in G_h^{vis}$ we add a vertex p_{ti} that represents an extension of p_h to p_t. The same rules of extension as we had in paragraph 3 apply: the new h-invariant is calculated and the tether length is checked. We call this new set P_t. After that we check for duplicates in this set (meaning the vertices that represent p_t and have the same h-invariant), those vertices are merged to one vertex with all the edges that the duplicates had. We then connect each vertex from P_t to p_t with an edge of weight 0.
   Now we use the newly-constructed graph to efficiently solve our motion planning problem by running dijkstra or any other path finding algorithm for the graph to find the shortest

path from p_s to p_t. The h-invariant of the termination point will be the h-invariant of the corresponding vertex from P_t set.

4. Motion Planning: Search and Sampling

3 Astar

The results on map2 (figures below):

Epsilon	Cost	Expanded nodes
1	349.102597	43787
10	355.546248	50320
20	354.960461	50700

The effects of increasing epsilon allows the algorithm to potentially reach the goal faster (expanding less nodes) but we lose the admissibility of the heuristic and therefor the optimality guarantee of the solution. In this case, the number of expanded nodes increases with epsilon (even though on map1 it decreases) and the cost is better for epsilon = 20 vs epsilon = 10.

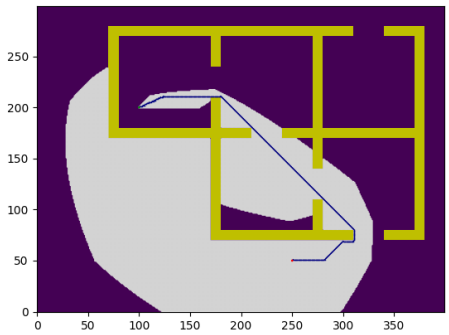

Figure 4: Epsilon = 1

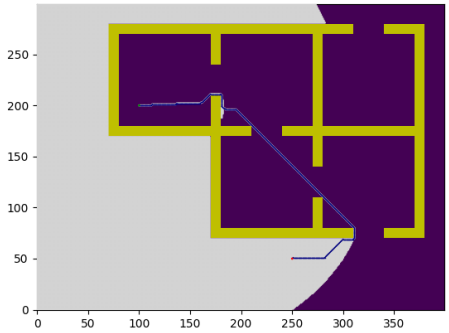

Figure 5: Epsilon = 10

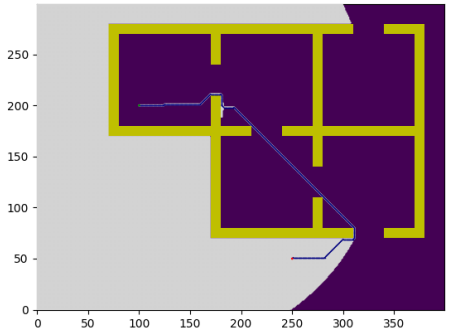

Figure 6: Epsilon = 20

4.4 RRT and RRT*

In our implementation we assumed that the goal is reachable, i.e. there exists some path from the start to the goal. If provided map requires unreachable path our implementation will stuck in infinite loop.

Table below represents the results of the RRT runs with different bias and step size on Map 2. The average was taken over 25 consecutive runs. We were asked to implement two versions of the extend function: E1 and E2. We can consider E1 as a special case of E2, where the step-size η equals ∞. That is the reason the values for E1 and E2 with step size infinity are almost identical.

Mean run time (sec)	Result
Bias 5% E1	-
Bias 5% E2	15.26
Bias 20% E1	-
Bias 20% E2	14.96

Mean path cost	Result
Bias 5% E1	-
Bias 5% E2	520.04
Bias 20% E1	-
Bias 20% E2	523.39

Figures below are snapshots from representative runs. It is worth noting that we observed a high variance in the run times. For example, in the E2 0.2 goal bias runs, there were 4 out

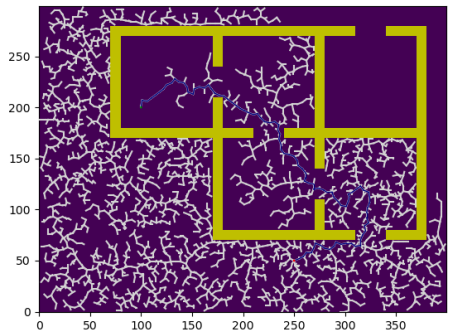

Figure 7: Bias: 5%, Step size: 5

Based on the results we achieved, for a map similar to the one we used we would choose 5% bias with the step-size 25. However we would suggest running the algorithm several times and then choose the best performing plan, since we've observed high variance on this specific task. Some of our runs found paths with less than 400 cost, which is only 10% over the optimal path.

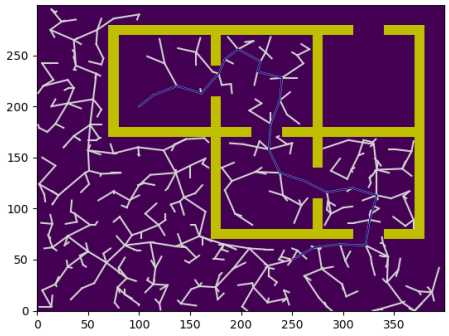

Figure 8: Bias: 5%, Step size: 20

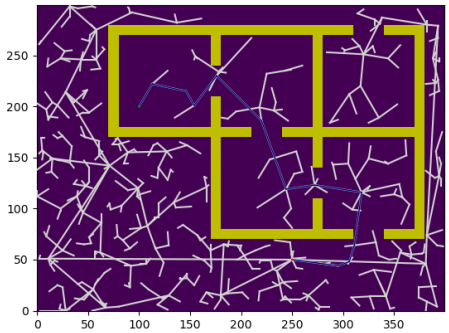

Figure 9: Bias: 5%, Step size: Inf

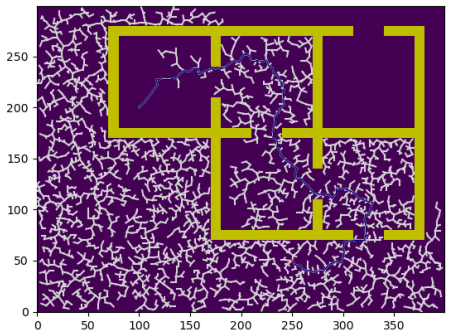

Figure 10: Bias: 20%, Step size: 5

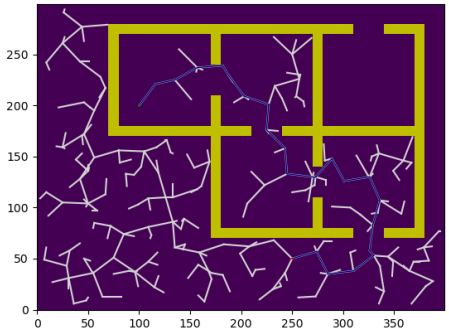

Figure 11: Bias: 20%, Step size: 20

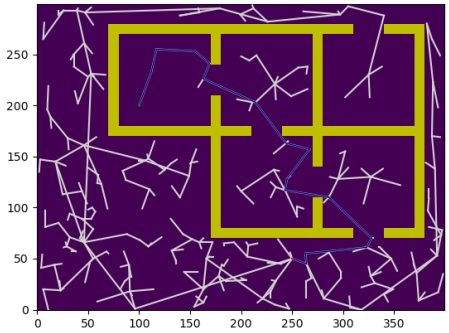

Figure 12: Bias: 20%, Step size: Inf

0.0.1 RRT*

We built our RRT* implementation based on the RRT from previous chapter. We chose to ran the RRT* with a bias to sample the goal state of 5% since it showed the best results before. We chose the step size of 35. The amount of nearest neighbors to attempt the rewiring operation we used was either constant or dynamic. The constant ones were 3, 10 50 and 5000. The dynamic one was chosen as $log(tree\_size)$, where $tree_size$ was current size or the tree. We will show one example tree and plan found for each step size and nearest neighbors choice.

Table below represents the results of the RRT* runs with different bias and step size on Map 2.

Mean run time (sec)	k = 3	k = 10	k = 50	k = 5000	$k = log(n)$
Bias 5%, E2 Step 25	2.25	1.69	3.28	10.27	2.01

Mean path cost	k = 3	k = 10	k = 50	k = 5000	$k = log(n)$
Bias 5%, E2 Step 25	465.81	417.02	440.28	452.96	466.02

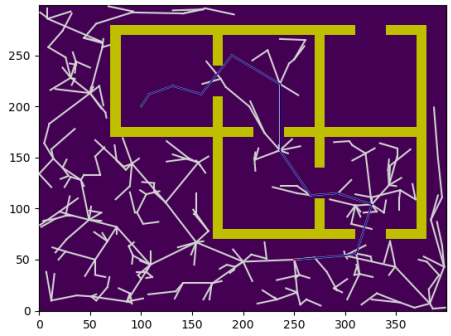

Figure 13: Bias: 5%, Step size: 25, $k = log(n)$

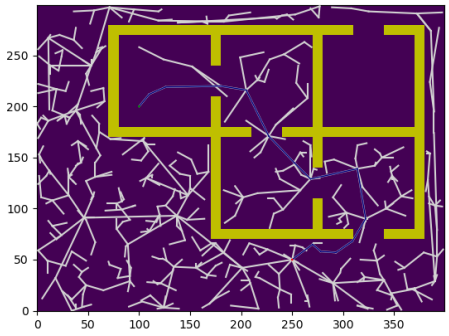

Figure 14: Bias: 5%, Step size: 25, k=3

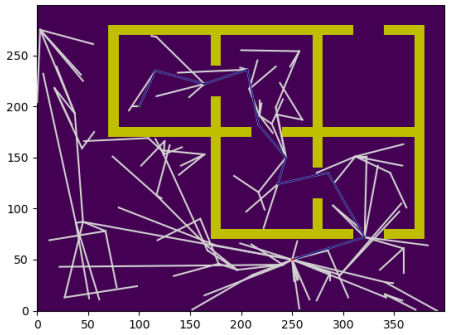

Figure 15: Bias: 5%, Step size: 25, k=10

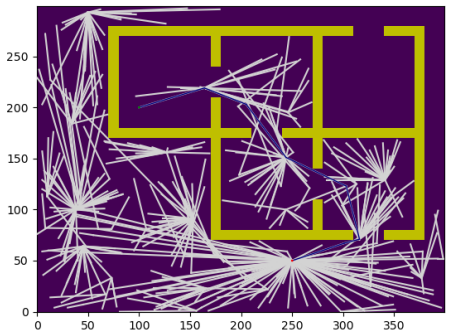

Figure 16: Bias: 5%, Step size: 25, k=50

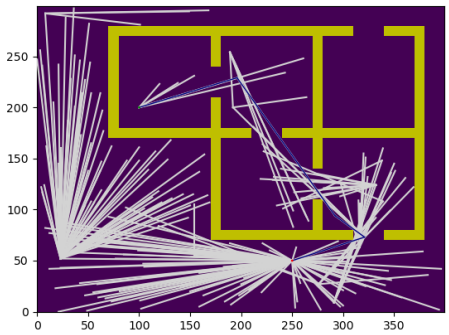

Figure 17: Bias: 5%, Step size: 25, k=5000

Based on the results we achieved for RRT* on the Map 2 with 5% bias and the step-size 35, the best k showed to be equal to 10. However, we would suggest that in general case the best k would be dynamic. During our experiments we've monitored the growth of k in dynamic cases, and we observed that most of the time it was below 10. Having dynamic k makes reduces the need to adjust k for each specific map. The graph below represents the correlation between the success rate and time for k=10.

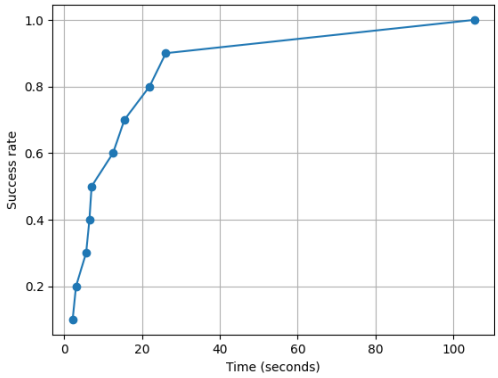

Figure 18: Success rate vs time for k=10

In general, RRT* outperforms normal RRT significantly being only twice slower in most cases. However we still would suggest running the algorithm several times and then choose the best performing plan, since we've observed high variance also on this task. Some of our runs found paths with less than 400 cost and faster than RRT, which is only 10% over the optimal path.