

Motion Planing Algo2

Nadav Bluger, Yaniv Kaniel

November 2025

2

2.1 Warmup

- 1.c 20%
- 2.d 60%

2.2 Exercise

2.2.1 Ploting

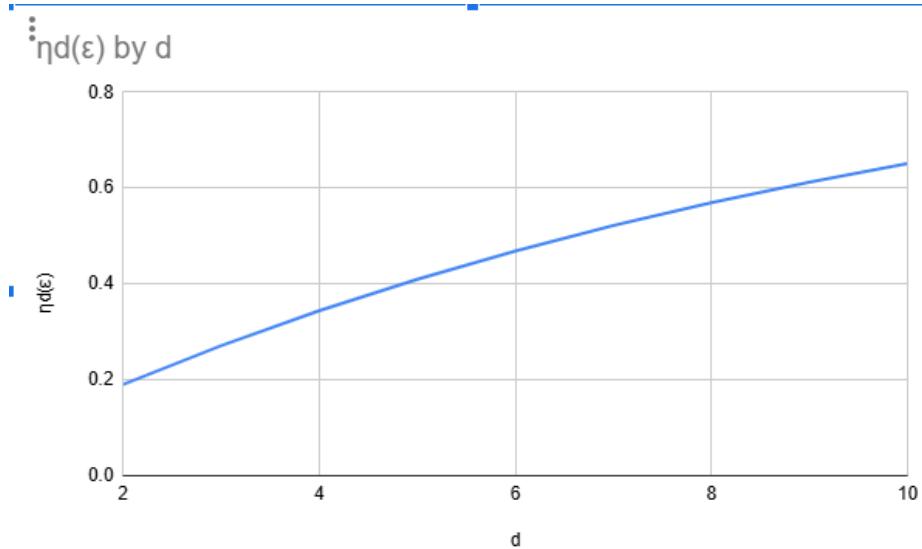


Figure 1: $\delta = 0.1$

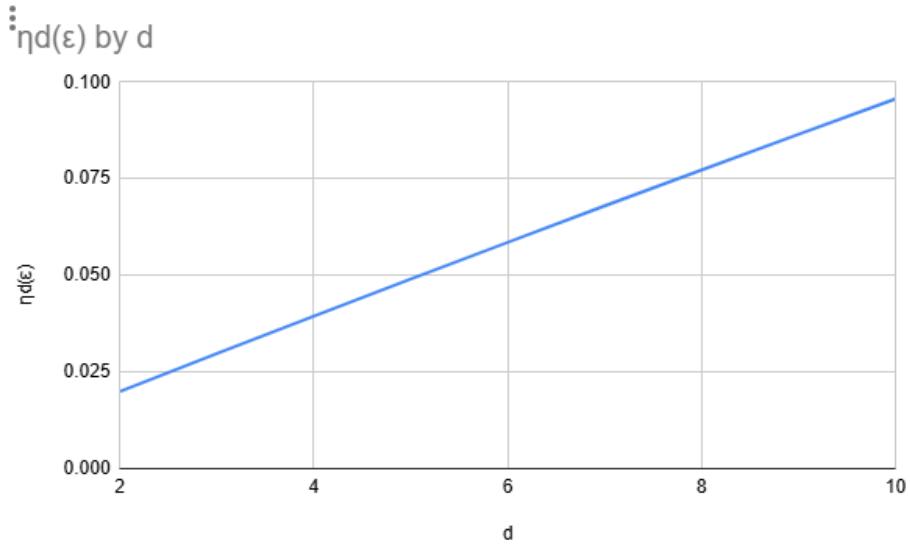


Figure 2: $\delta = 0.01$

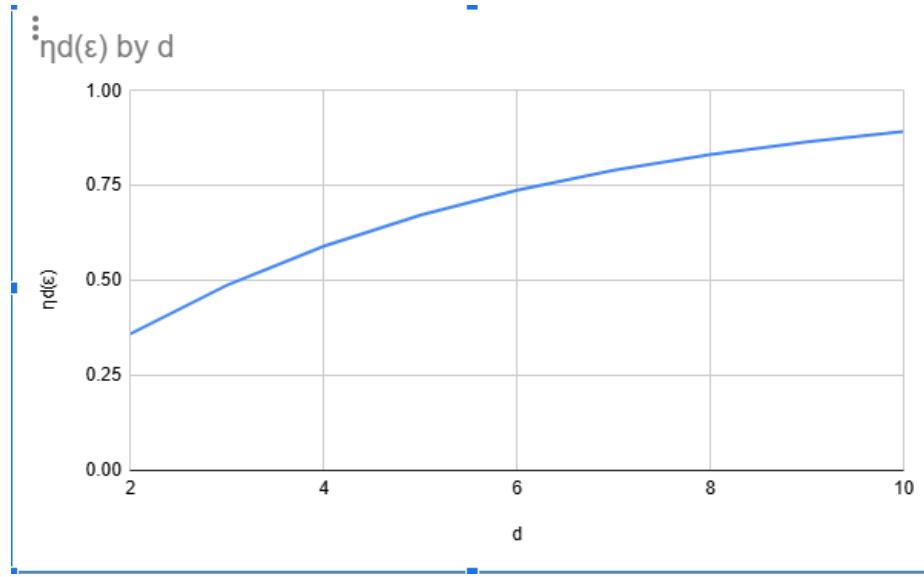


Figure 3: $\delta = 0.2$

2.2.2 Discussion

When plotting our D dimensional balls and by looking at the equation of a D dimensional ball it is easy to see that as D grows most of the volume of the

ball is in the thin shell near the outer surface. When we shrink the radius from r to r -epsilon we are removing the outer shell, which is removing most of the volume of the ball. Therefore in higher dimensions, a small decrease in connection radius eliminates a huge fraction of possible neighbors.

3 Tethered Robots

3.1

The paths structure is a sequence of configurations (q_i, h_i) , in which q represents the physical location of the robot and h represents the tether description in that location, where the first configuration is the start and the last is the target. The path consists of straight line segments between 2 points and valid transitions between the tether states. The tether placement is defined by the point in which it meets an obstacle vertex along the way which anchors it in place.

3.2

We suggest an efficient way to encode the tethers description: For each obstacle we will create an imaginary infinite ray that goes outward for the obstacle, as shown in class. When the robot passes the ray from the obstacle from right to left we will add "a", considering a is the obstacles ray it has passed. When the robot passes the ray from left to right we will add " a^{-1} ". This way if we were to go around an obstacle for example we would get aa, but if we were to go back and forth and not change the homotopy class (because the line can be warped if we don't go around the full obstacle), we would get aa^{-1} which would cancel out and the homotopy class would be equivalent to one that did not pass the ray.

3.3

A node is a tuple of a physical location (vertex of the original graph), and the tether description that we suggested in 3.2. We will use a Dijkstra like algorithm to compute the graph. At each step, we pop the node with the shortest current rope length from our queue. Look at the neighbors in the original graph and change the homotopy class accordingly if it passes the ray of an obstacle. Then we check that the change in the homotopy class is valid, meaning that the tether in the class of our original node can be extended along the edge of our neighbour without intersecting an obstacle. Then calculate the total length of the new tether state (v, h') , meaning the geometric length of the path defined by h' ending in v . If the new length is smaller than L then this a valid transition and we add it as a child of our vertex and push to our queue (assuming we haven't visited (v,h') yet or found a shorter way to get there). Unlike dijkstra where the algorithm terminates when the destination is reached here the algorithm will only terminate when the priority queue is empty. The priority queue will empty because our tether is finite. So eventually our graph will not be able to

add new nodes because the tether length will be greater than L , and then our extension step will reject the move, and the branch will die.

3.4

We need to add the starting vertex (ps, ws) . And we need to add the end vertex pt as the goal location. We need to add to our graph the edges of all the possible scenarios where we can move from point (ps, ws) to another vertex, meaning connecting to nodes that are visible from ps and the tether remains at a length smaller than L after the update. Now that we have our graph we can use a Dijkstra algorithm, that traverses the augmented graph and the algorithm will terminate when we pop a node where the physical location is pt . This guarantees we found the shortest valid tether path to the goal.

4

.1

4.5

4.5.1 Compute Distance

In a 2d environment the euclidean distance between two points is the root of the squares of the distances on each axis. Therefore we do a elementwise subtraction of each dimension, and then use the distance formula for them (add all of them squared and squareroot at the end).

4.5.2 Compute Forward Kinematics

Here we will use a geometrical rule in kinematics that each joint angle rotates everything after it, and each link extends in the direction of the cumulative rotation. So we keep a sum of our angles while multiplying them by our link length each time to add a new position.

4.6 validate robot

first we represent each robot link by its two ends and then check if any two links intersect.

4.7 PRM discussion and plots

We will show the visualization of our path with a few pictures from the gif, the gif is also added in our submission:

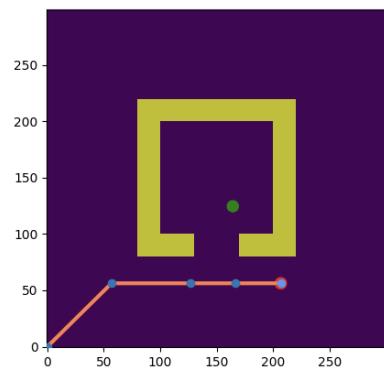


Figure 4: Starting configuration

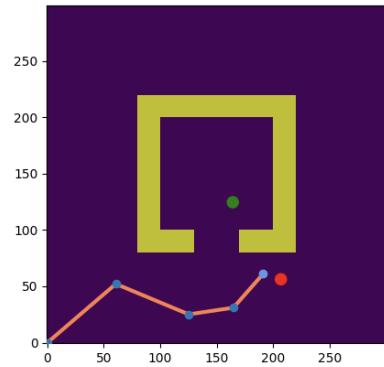


Figure 5: Moving

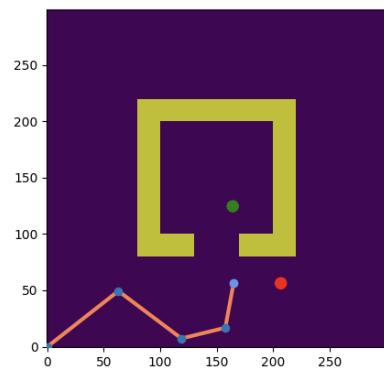


Figure 6: Moving

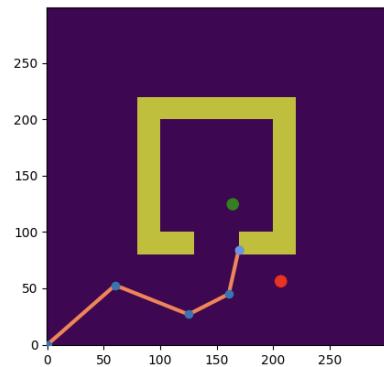


Figure 7: Moving on obstacle

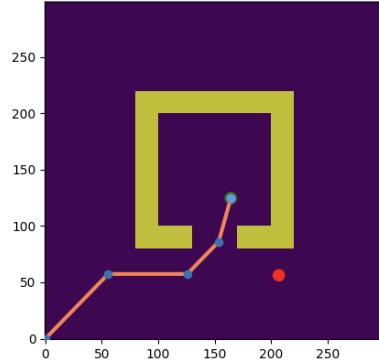


Figure 8: Finish

As we can see we start at the start configuration, bend backwards, move along the wall and end at the target configuration.

We will now discuss the plots below: Plot 1:

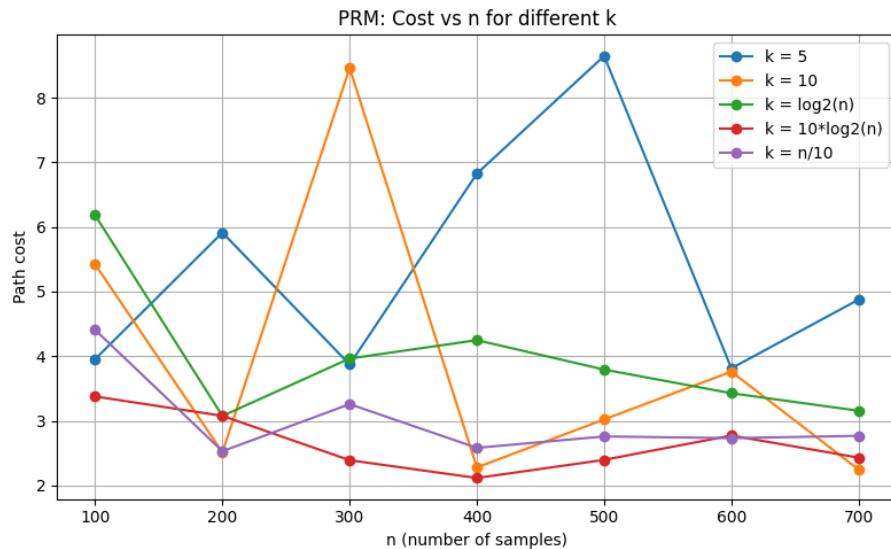


Figure 9: Cost Vs N

Here we notice a few interesting results. First we see how the results are probabilistic, there is not a specific rule that is followed that is dependent on n for the cost. But we can notice that in most scenarios (other than $k=5$ and a few outliers) the more nodes we sample and as we increase k , the cost

becomes smaller. This makes sense, when sampling more nodes and increasing our nearest neighbour factor we have a stronger graph (more nodes and edges) so there is a higher likelihood that we find a better path.

Plot 2:

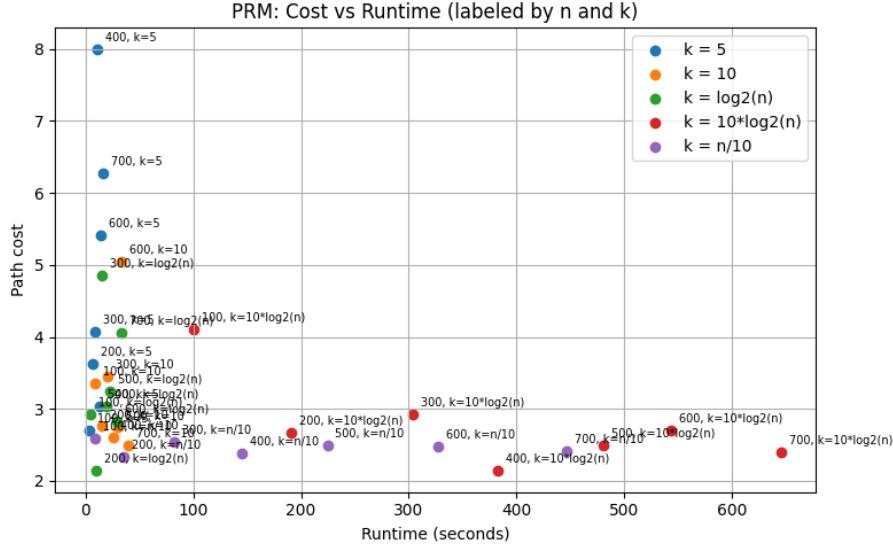


Figure 10: Cost vs Runtime

In the scatter plot above we noticed that, reasonably, there is a strong correlation between a large n and a long runtime. However we noticed that the runtime took a long time not because of the time it took to create the nodes but because it took a long time to create the graph/the edges of the nearest neighbours. Therefore you can see that for large k 's the runtime is also longer... We assume this is because of the edge validity function because as seen in the lectures the computation time to predict if an edge is intersecting an obstacle is very large.

5

5.2

In order to completely avoid false negative we need the spheres to completely contain the link. Then if any other sphere does not intersect with the link spheres we can know for sure that it does not intersect with the link itself. In order to minimize the amount of false positive we need to minimize the amount of space contained in the sphere that is not part of the link because any

intersection with a link sphere in that area will be counted as a collision though it is not.

5.2.1 One Sphere

One sphere covering the entire link will cover the least space outside of it if its center is in the links center, therefore its center will be $(5r, 0, 0)$. The furthest point in the link from its center (which our one sphere must include) will be any point on the conference of the base and the top of the link. These points are at a distance of $d = \sqrt{(5r)^2 + r^2} = \sqrt{26}r$ from the center, which will be the radius of our sphere

5.2.2 Two Spheres

In order to place two sphere we will split the link into two and repeat the process of one sphere for each. So the sphere will be at $(2.5r, 0, 0), (7.5r, 0, 0)$ with a radius of $\sqrt{(2.5)^2 + r^2} = \sqrt{7.25}r$

5.2.3 Five Spheres

Similarity to two sphere we will split the link to 5 equal parts giving us spheres at $(r, 0, 0), (3r, 0, 0), (5r, 0, 0), (7r, 0, 0), (9r, 0, 0)$ with radius of $\sqrt{r^2 + r^2} = \sqrt{2}r$

5.2.4 Ten Spheres

We will now split it into 10 parts giving us spheres at $(0.5r, 0, 0), (1.5r, 0, 0), (2.5r, 0, 0), (3.5r, 0, 0), (4.5r, 0, 0), (5.5r, 0, 0)$ of radius $\sqrt{(0.5r)^2 + r^2} = \sqrt{1.25}r$

5.2.5 Trade-Off Discussion

In general the tradeoff is compute time vs accuracy. The more spheres we create the less space outside of the link is included in them and therefore the less false positives happen increasing the accuracy of our detection collision process. However this comes with a cost, the more spheres the more calculation we need to preform with each check since we would need to check the collision against each of them.

5.3

5.3.1

Here is a configuration that is collision free and one that has a self collision (link to link)

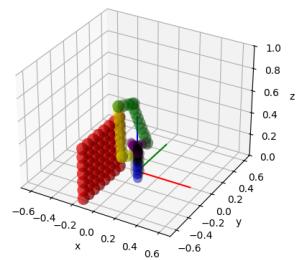


Figure 11: inflation factor = 1

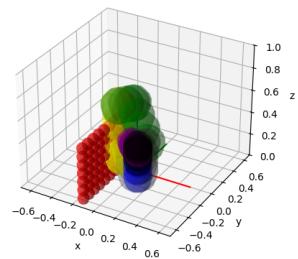


Figure 12: inflation factor = 3

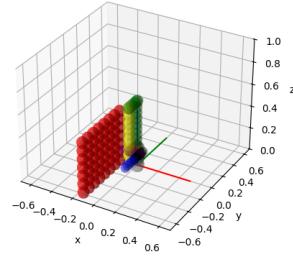


Figure 13: self collision, [0, -90, 180, 0, 0, 0][deg]

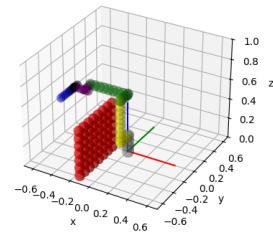


Figure 14: collision free [0, -90, 90, -90, 0, 0][deg]

5.3.2

Here is the same configuration that is collision free and one that has an obstacle collision

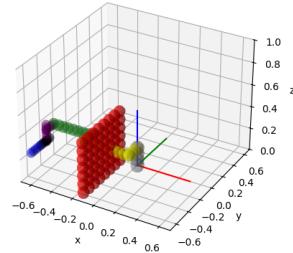


Figure 15: obstacle collision [0, -90, 180, 0, 0, 0][deg]

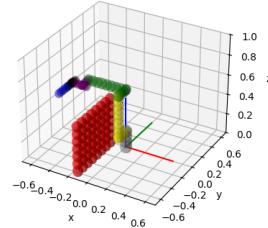


Figure 16: collision free [0, -90, 90, -90, 0, 0][deg]

5.4

For a resolution of 0.1 the checker tested two configurations before finding a collision (meaning the first intermediate configuration was invalid), because of the collision the checker returned False. For a resolution of 1 the checker tested two configurations (the minimum amount, meaning no intermediate configurations) and found them both to be collision free and therefore returned True

5.5

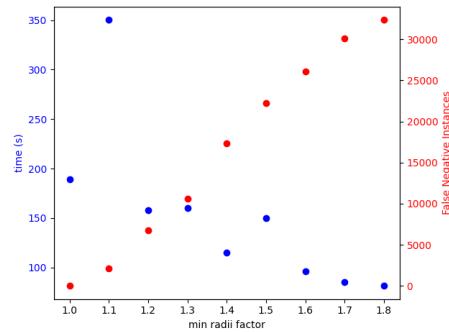


Figure 17: Enter Caption