Random sampling based algorithms

…

Content

- What are random sampling based algorithms
- Quick brief for Tree data structures
- Rapidly exploring random trees (RRT Algorithm)
- RRT* Algorithm
- Quick summary for ellipse
- Informed RRT* Algorithm

What are random sampling based algorithms?

Algorithms whose goal is to find the shortest path between two given points called start and goal points.

It is used mainly in path planning problem.

It is based on randomly creating samping points and connect between then based on the nearest distance between each point (node) and its parent (previous connected node).

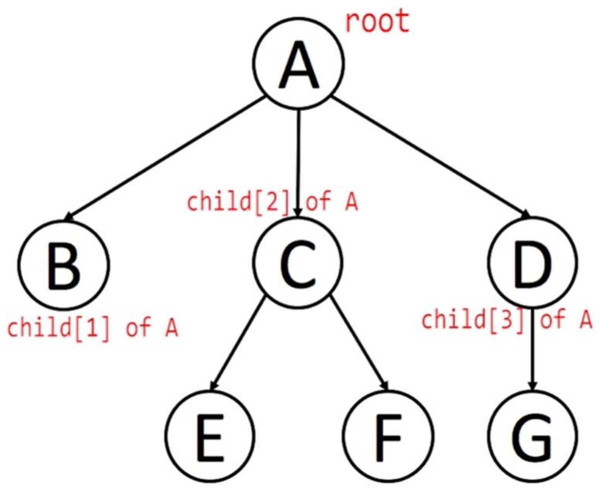
- RRT → Can find the path but not the optimal (shortest) one.
- RRT* → Can reach the shortest path but takes a very much time to reach it.
- Informed RRT* → An optimized algorithm of RRT* which aims to reduce the sampling area for each iteration to reduce the time of execution.

Quick brief for tree data structure

1. **<u>Introduction</u>**

In graph data structure, each node can have many parents and many children, but the tree can have only one parent and many children.

The tree is a simplified graph with no cycles, which means there is only one path to reach any node.

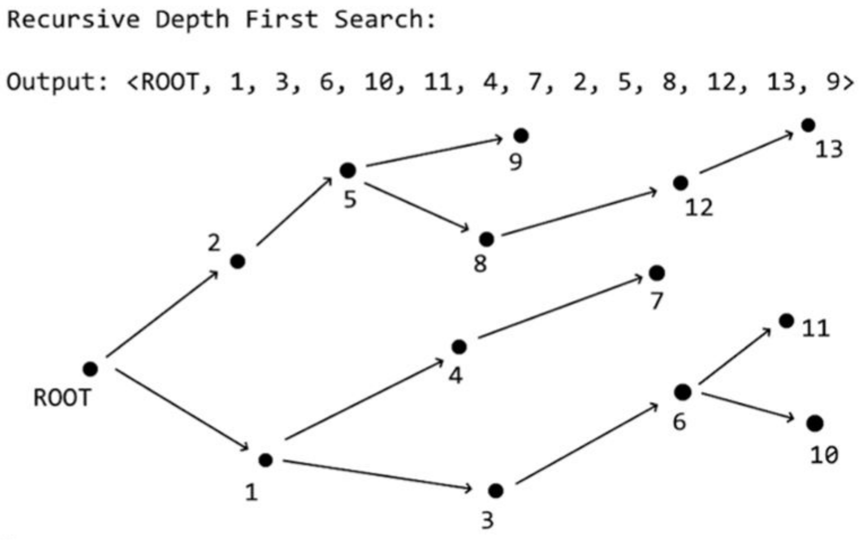


- ❑ Here, A is called the root; which is the starting of the tree. B, C, D, E, F, G are called nodes.
- ❑ Path ⇒ The way between two points (nodes).
- ❑ Parent(B) = A
- ❑ If both B and C were connected to E, this becomes a graph.

2. Tree search using DFS (Depth First Search) [Recursive way]:

```
search(root)
For child in root.children: search (root)
```

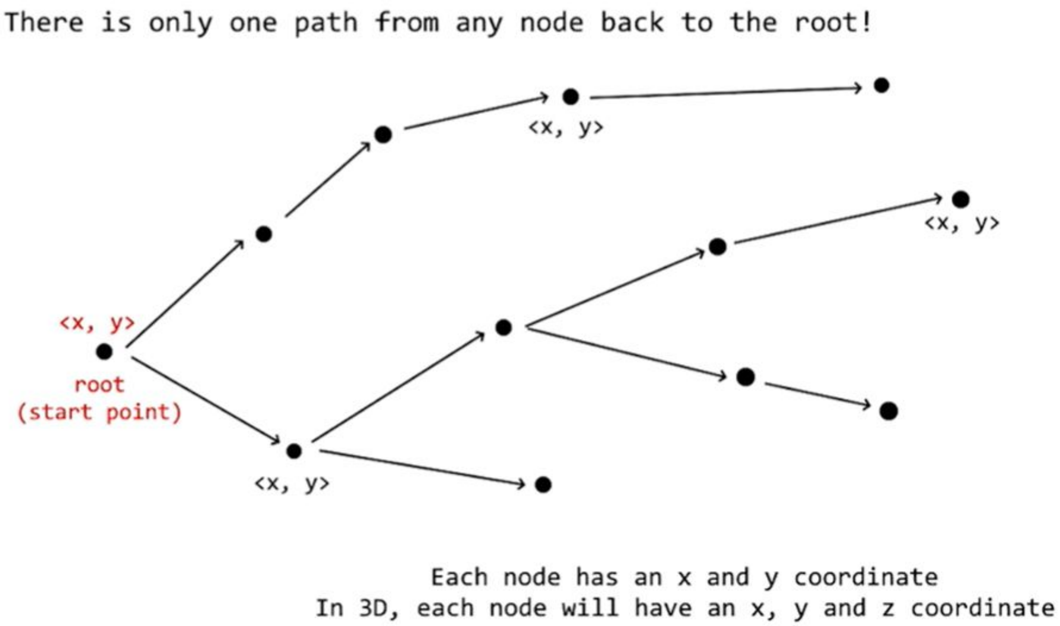
- Always start at node
- Tree DFS is needed when finding the nearest node to a randomly generated point.
- The Euclidean distance is the distance between any 2 nodes.



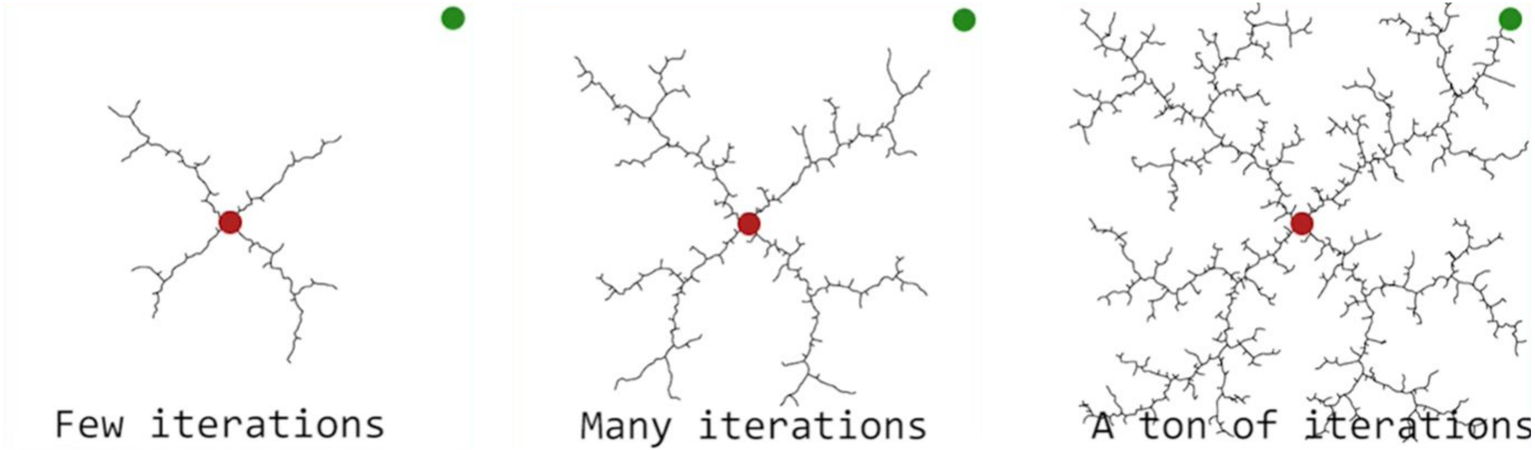
3. Rapidly Exploring Random Tree:

Each node:

- Has an X location.
- Has a Y location.
- Has 1 parent.
- Has many children.



Rapidly Exploring Random Trees (RRT Algorithm)

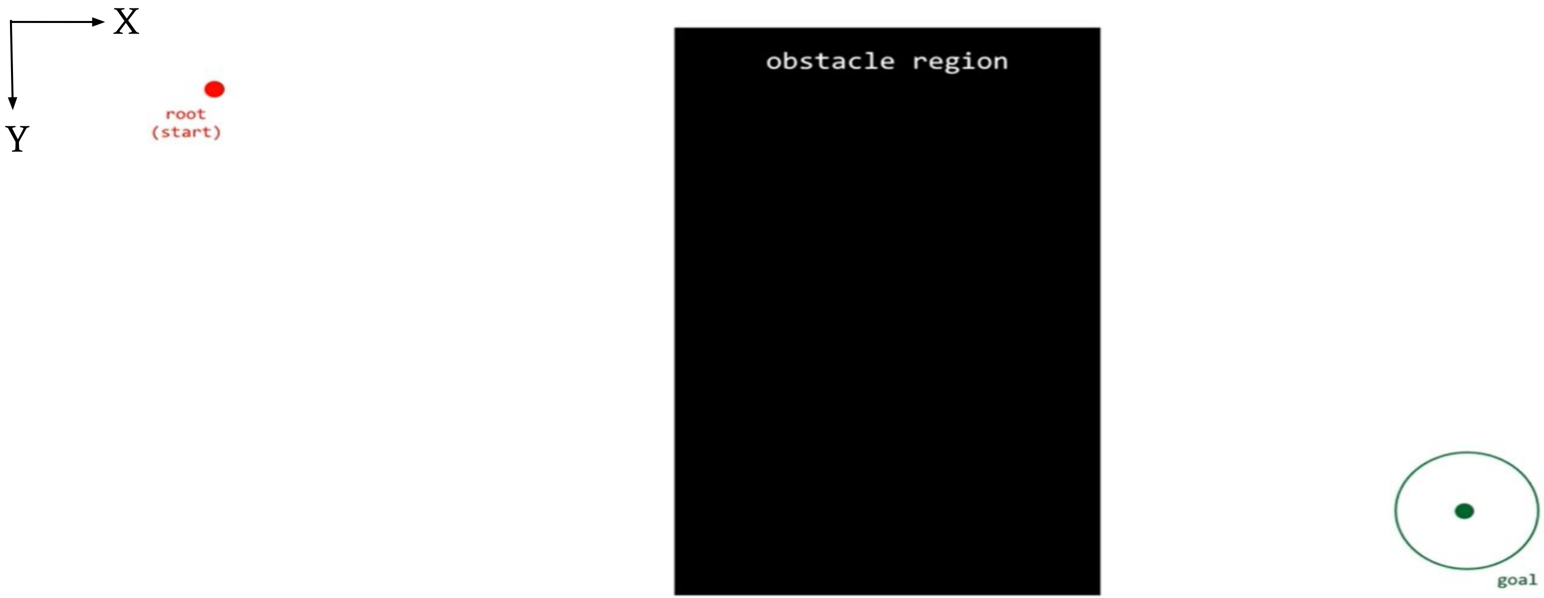


Generated purely by <u>sampling</u>.

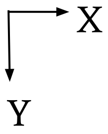
This is how the tree expands outward in all directions, filling up the space. This is in case of no obstacles exist.

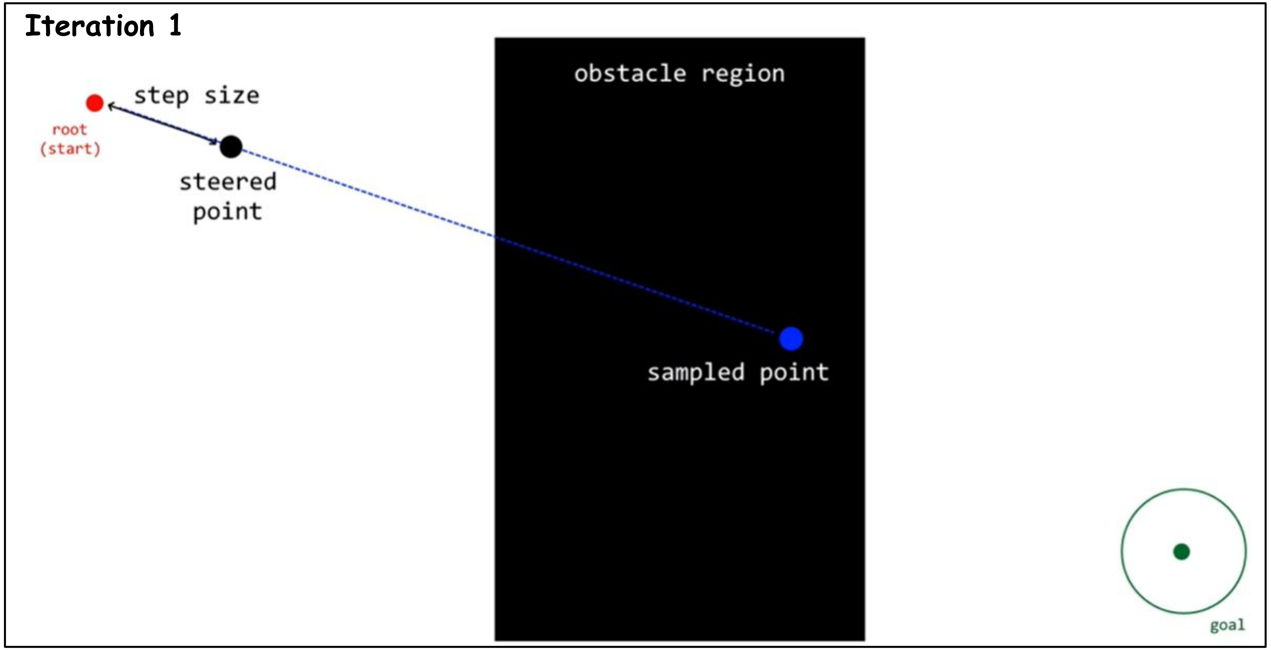
RRT Logic

<u>Given two points (start and goal) with one obstacle between them:</u>

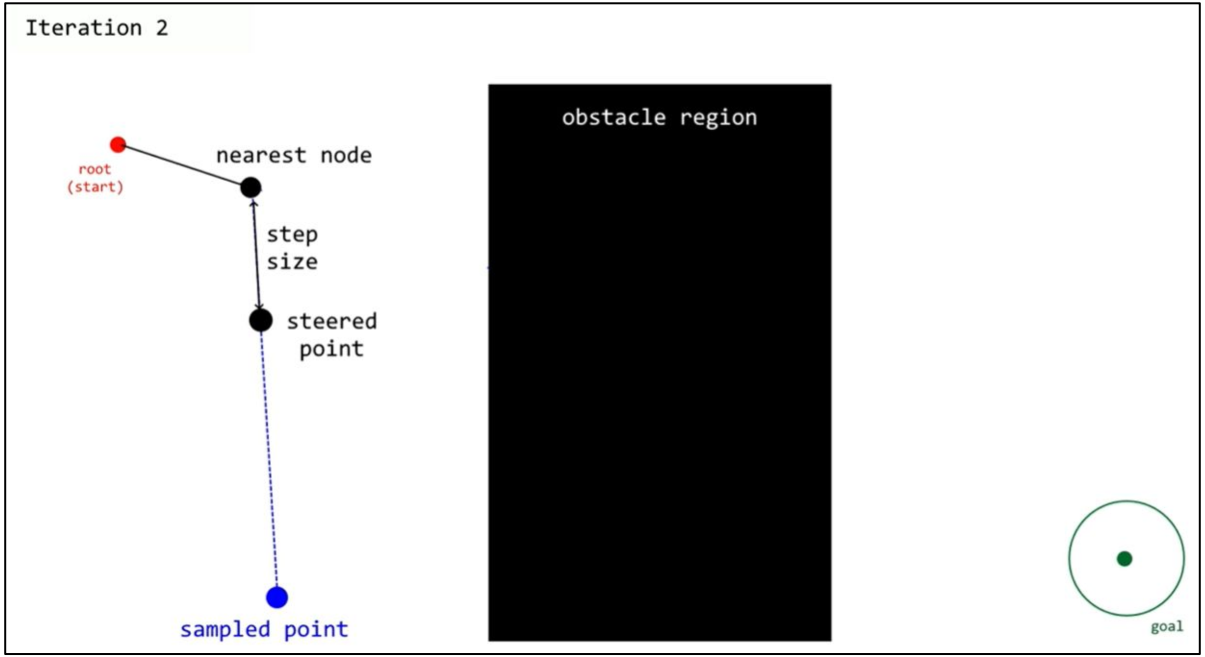


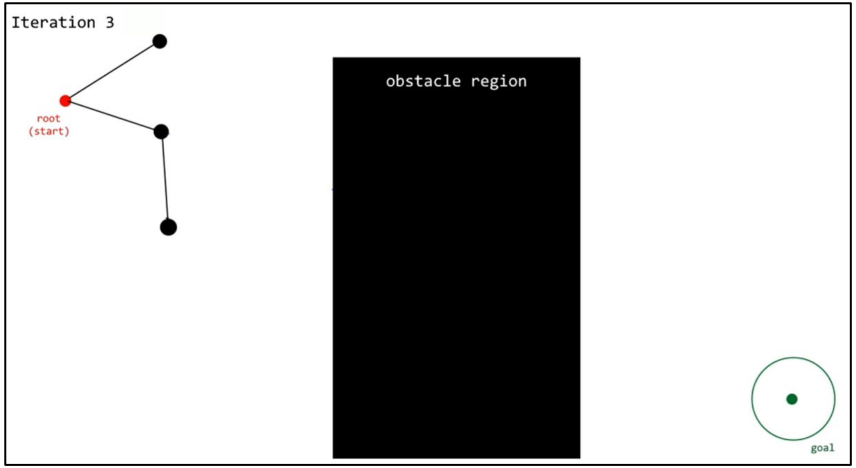
1. We sample a point, steer to a point with the step size offset from the start.
2. We check if there is no obstacle between the start and the steer points, if there is not one, we add this to the points list.

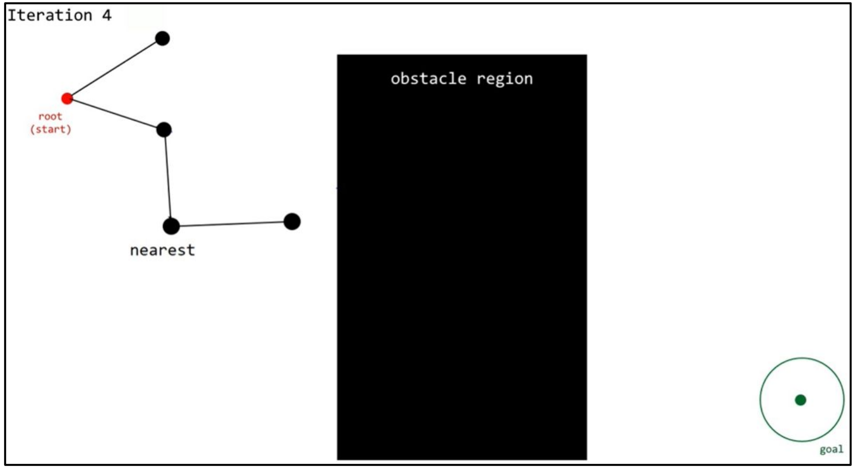


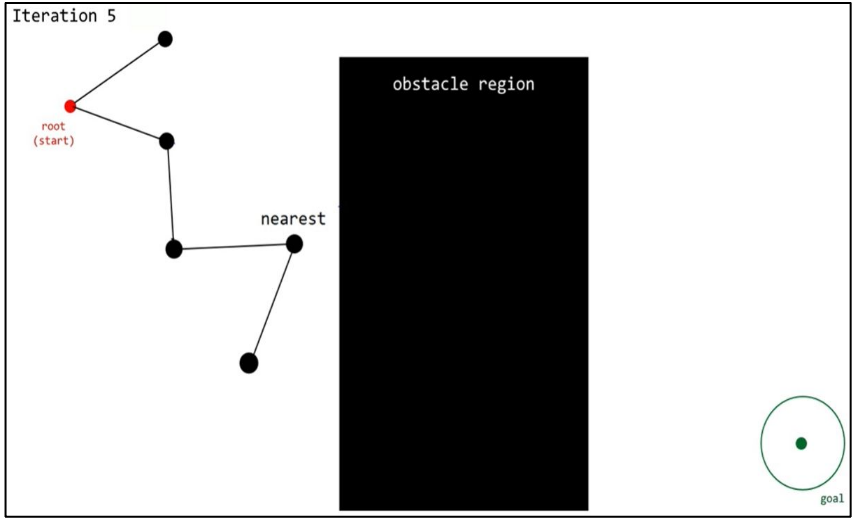


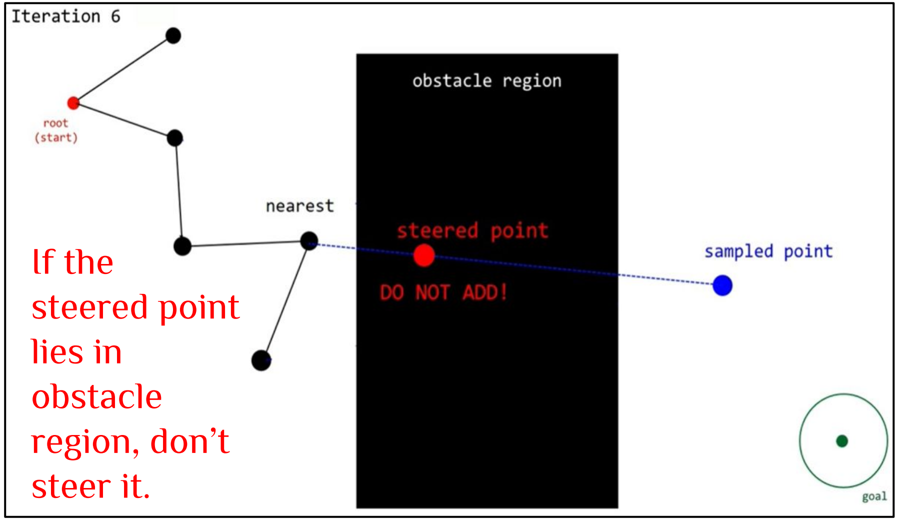
3. For the 2nd iteration, we sample another point, we then find the nearest node, which is not the start here but the second one, we then steer from the nearest node to the sample point. Since there is no obstacles between the nearest node and the sample point, we can add the steered point.



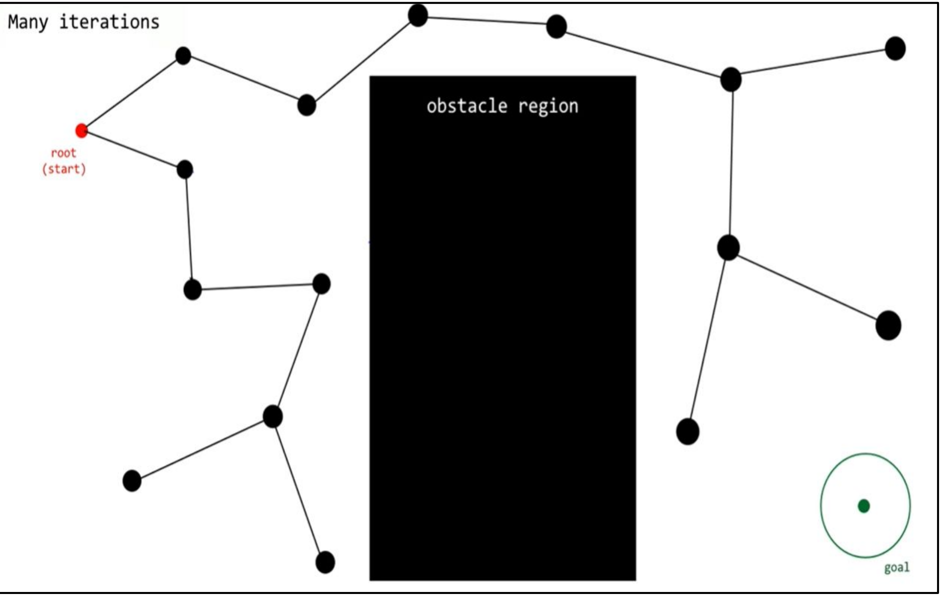


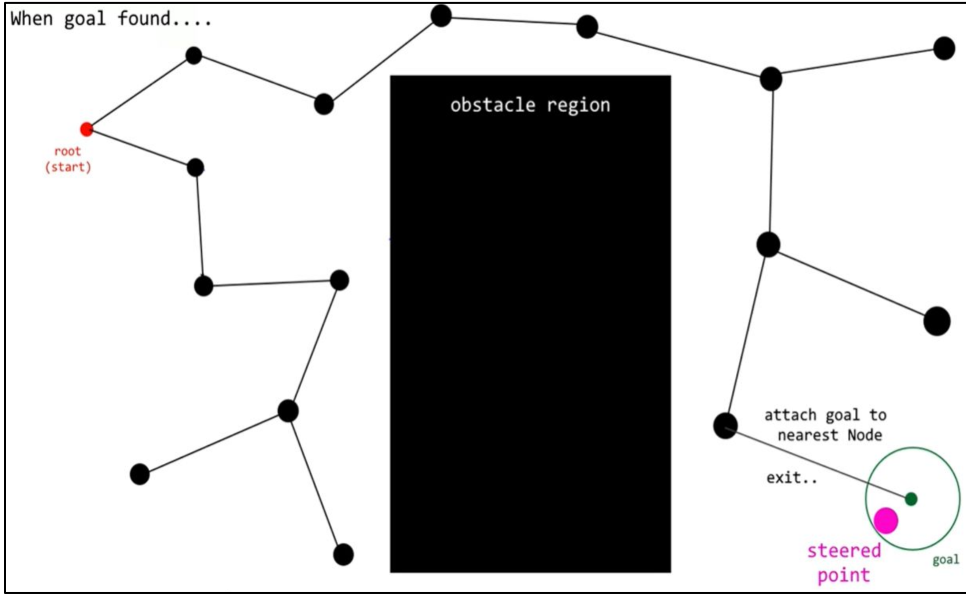






If the steered point lies in obstacle region, don't steer it.

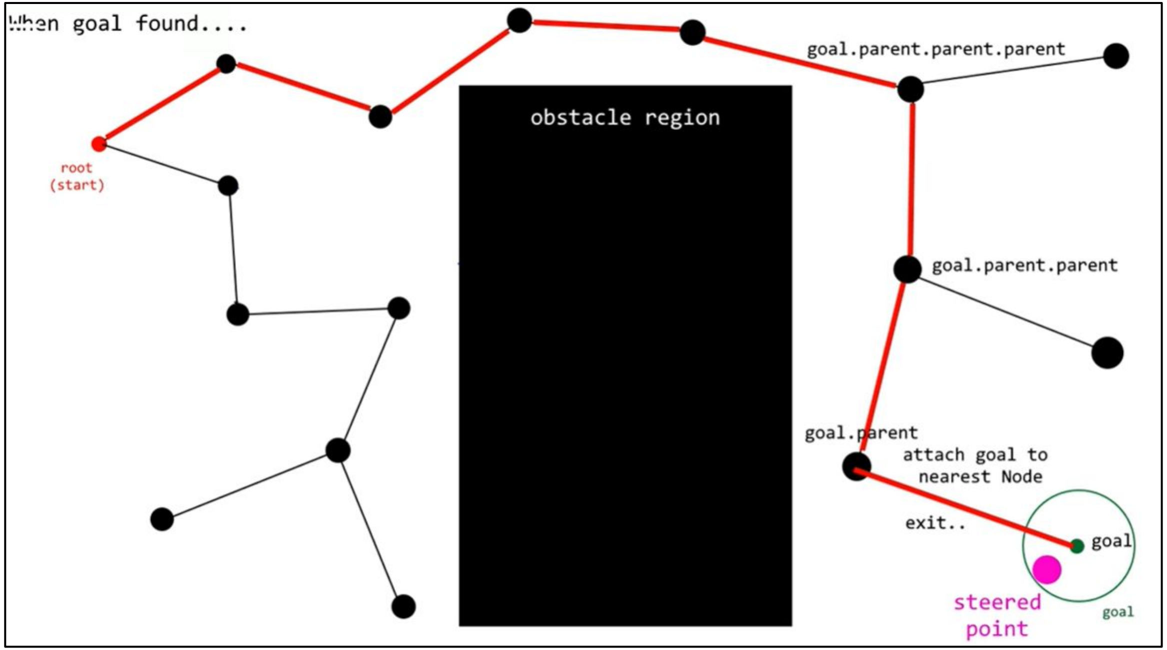




4. If the goal region is reached, we check if the steered point lies within the goal region, so we attach the goal to the nearest point.

How do we find the path?

This step is called tree tracing.



Each node can have many children, but it has only one parent. Since the goal parent is the nearest node, we keep tracing the parent back until reaching the goal.

Here, we cannot go from start to goal because each node can have many children so we may end up with a wrong path. So we have to start from the goal itself because there is only one path.

GENERATE_RRT($x_{init}, K, \Delta t$)

$\mathcal{T}$.init(x_{init});
for $k = 1$ **to** K **do**
    $x_{rand} \leftarrow$ RANDOM_STATE();
    $x_{near} \leftarrow$ NEAREST_NEIGHBOR($x_{rand}, \mathcal{T}$);
    $u \leftarrow$ SELECT_INPUT(x_{rand}, x_{near});
    $x_{new} \leftarrow$ NEW_STATE($x_{near}, u, \Delta t$);
    $\mathcal{T}$.add_vertex(x_{new});
    $\mathcal{T}$.add_edge(x_{near}, x_{new}, u);
Return $\mathcal{T}$

LaValle, Steven M. "Rapidly-exploring random trees: A new tool for path planning." (1998): 98-11.

For summary, we find the nearest neighbour, and then you steer to new point and then we add the vertex if there is no obstacle, and then we add the edge.

- The edge is the step size distance
- K is the no.of iterations.
- X is the initial point.

For code

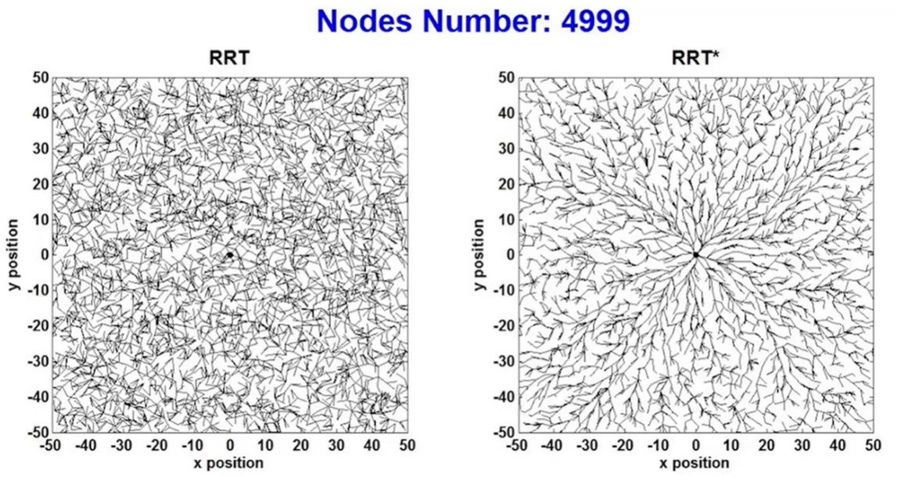
RRT* Algorithm

RRT algo. Can find a path between two nodes, but not the shortest path.

RRT* aims to find a shorter spot by optimization. It generates a lot more straighter paths in an obstacle free space.

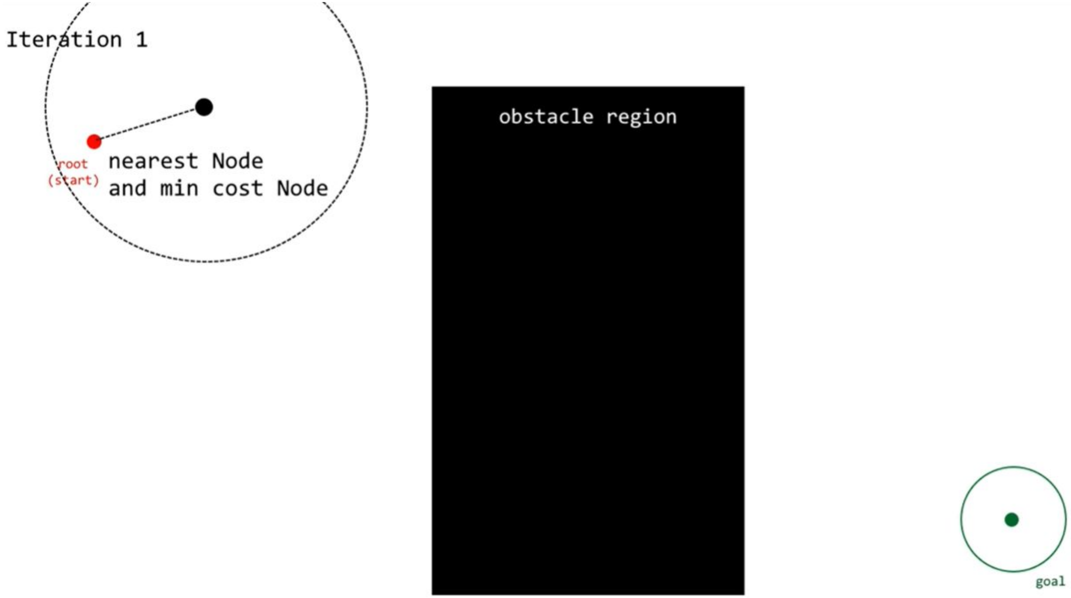
The key difference is that at each iteration, we have to search within a neighbouring region and we have to find all nodes that are in that region from the nearest point.

Nearest point is found the same way as RRT algo.



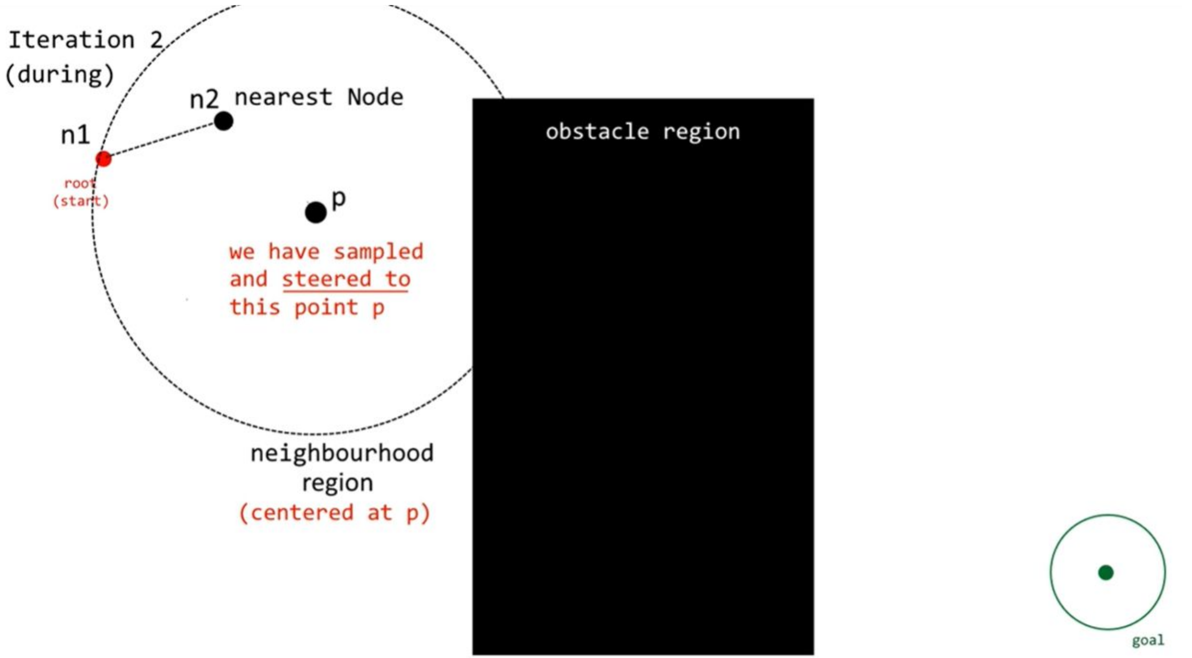
RRT* logic

1. In iteration 1: we won't have any optimization to do; because we just have a start node, we we will connect it.



2. <u>In iteration 2:</u>

a) Sample a point “p”.
b) Search a neighbourhood region centered at “p”.
c) Find that nodes are n1(start node), n2. So, the nearest point will be n2, but the path is not appended since it is obstacle free only. There are two additional steps must be done..

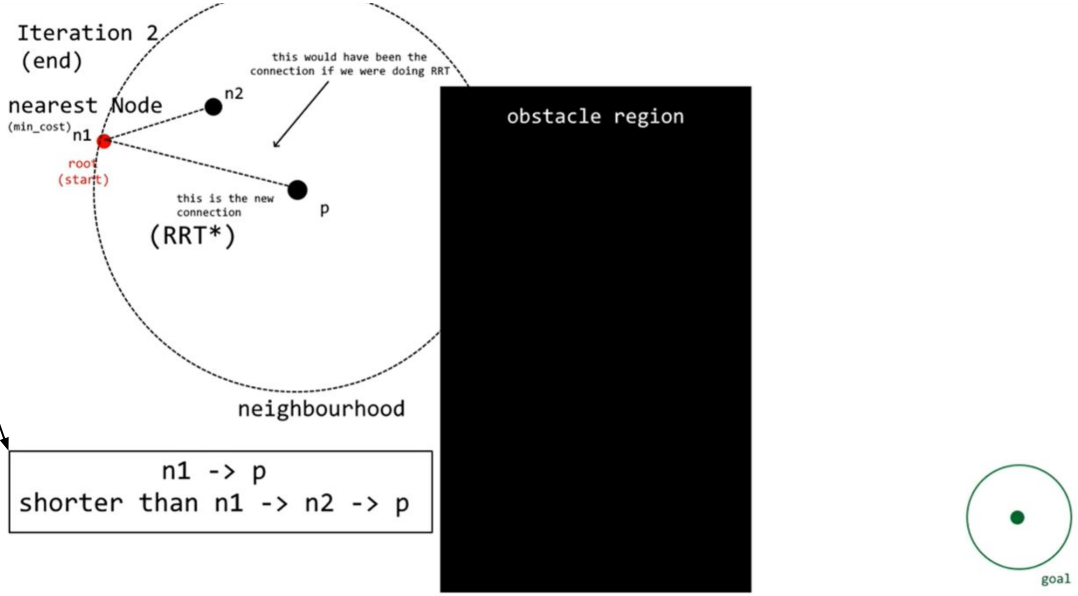


I. <u>Connect along the minimum cost path:</u>

The distance between n1, n2, and p is longer than the distance between n1 and p (directly). So if the the straight line from n1 to p is drawn, it will be shorter than going from n1 to n2 to p.

This is one of the key differences in RRT*.

If there is a node outside the region (let's say the goal), nothing will be done; because only the nodes within the neighbouring region are ones we looked at using this algorithm, any node outside this neighbouring region is an out of scope and the algorithm will not look at it.



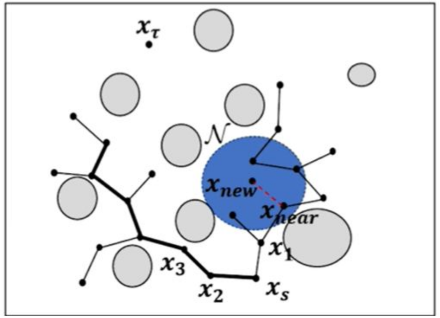
Extend step ⇒ the minimum cost path step

For summary, this what is done in this step:

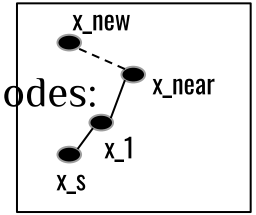
1. Find the minimum cost node to be connected. Which is done in RRT algo.
2. Find the nearest method to connect those two nodes, and then update the old nearest node with the new one (whose path has the minimum cost).

RRT*: Extend Step

- Generate a new potential node x_{new} identically to RRT
- ~~Instead of finding the closest node in the tree,~~ find all nodes within a neighborhood $\mathcal{N}$ (this is a typo), you must <u>also</u> find the closest node
- Let $x_{nearest} = \arg\min_{x_{near} \in \mathcal{N}} g_{x_{near}} + c_{x_{near}, x_{new}}$, i.e., the node in $\mathcal{N}$ that lies on the currently known shortest path from x_s to x_{new}
- Add node: $\mathcal{V} \leftarrow \mathcal{V} \cup \{x_{new}\}$
- Add edge: $\mathcal{E} \leftarrow \mathcal{E} \cup \{(x_{nearest}, x_{new})\}$
- Set the label of x_{new} to $g_{x_{new}} = g_{x_{nearest}} + c_{x_{nearest}, x_{new}}$



For this extended nodes:



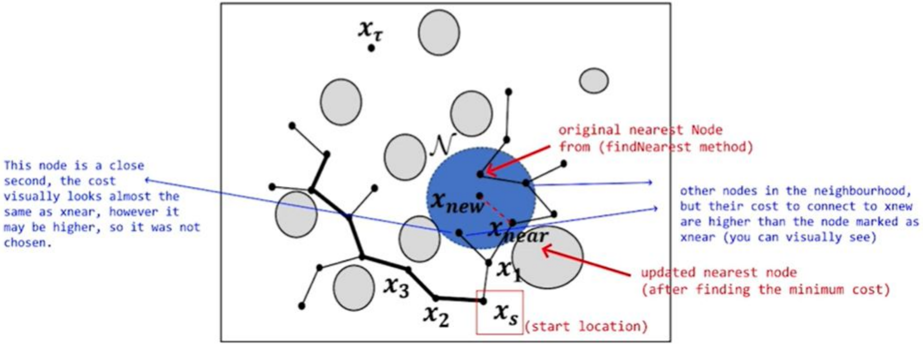
$$g_{x_{new}} = g_{x_{nearest}} + c_{x_{nearest}, x_{new}}$$

G_x_new ⇒ The cost of x_new (minimum cost to this node)
G_x_nearest ⇒ distance between x_near and x_1 + distance between x_1 and x_s
C_x_nearest, x_new ⇒ distance between x_near and x_new.

So, total cost = distance between x_s and x_1 + distance between x_1 and x_near + distance between x_near and x_new

RRT*: Extend Step

- Generate a new potential node x_{new} identically to RRT
- ~~Instead of finding the closest node in the tree,~~ find all nodes within a neighborhood $\mathcal{N}$ (this is a typo), you must also find the closest node
- Let $x_{nearest} = \arg\min_{x_{near} \in \mathcal{N}} g_{x_{near}} + c_{x_{near}, x_{new}}$, i.e., the node in $\mathcal{N}$ that lies on the currently known shortest path from x_s to x_{new}
- Add node: $\mathcal{V} \leftarrow \mathcal{V} \cup \{x_{new}\}$
- Add edge: $\mathcal{E} \leftarrow \mathcal{E} \cup \{(x_{nearest}, x_{new})\}$
- Set the label of x_{new} to $g_{x_{new}} = g_{x_{nearest}} + c_{x_{nearest}, x_{new}}$



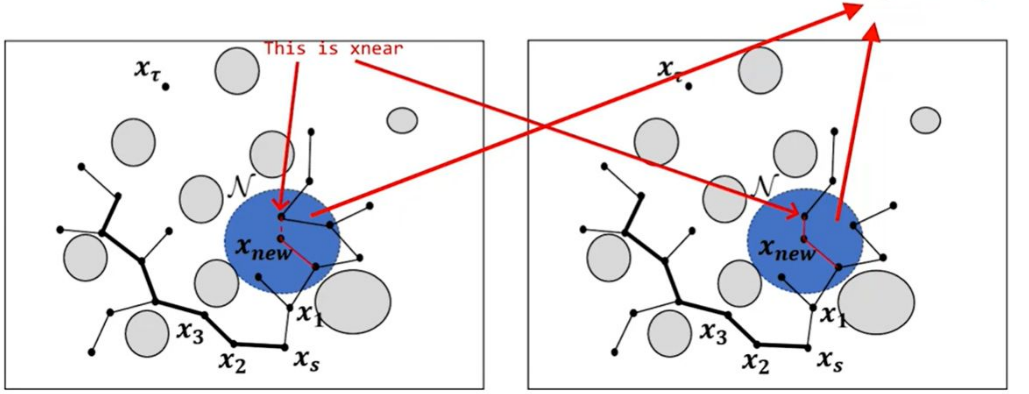
2. Rewire step:

If the cost(distance) of x_new (this is resulted from finding the minimum path cost step) + the cost(distance) of the x_new to x_near is lower than the cost (distance) between x_near and the x_s (start point), then remove the edge between the x_near and its parent, and add a new edge between x_near and x_new.

RRT*: Rewire Step

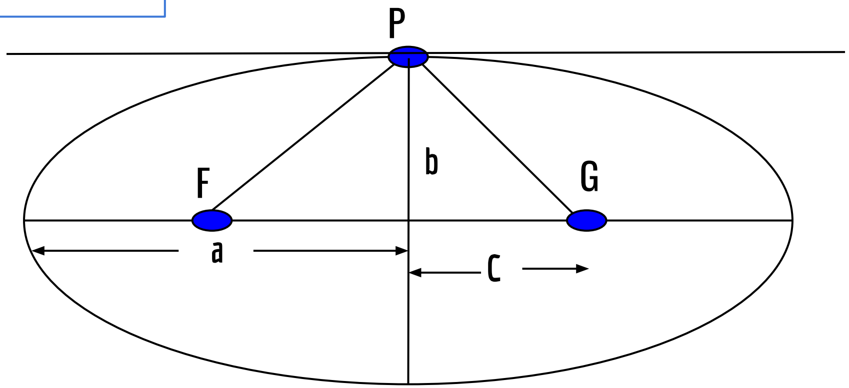
- Check all nodes $x_{near} \in \mathcal{N}$ to see if re-routing through x_{new} reduces the path length (**label correcting!**):
- If $g_{x_{new}} + c_{x_{new}, x_{near}} < g_{x_{near}}$, then remove the edge between x_{near} and its parent and add a new edge between x_{near} and x_{new}

You remove the edge by removing the parent, look carefully!



For code

Ellipse

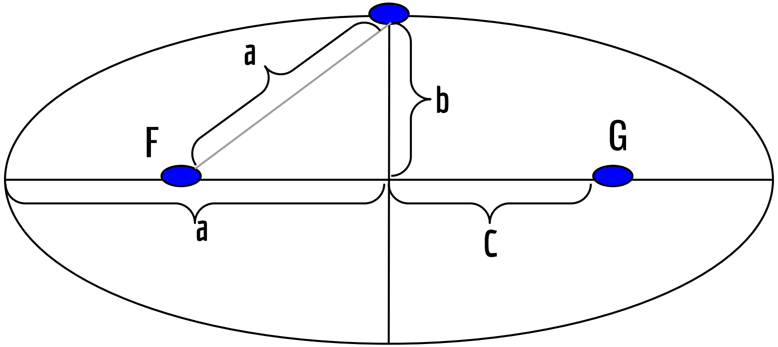


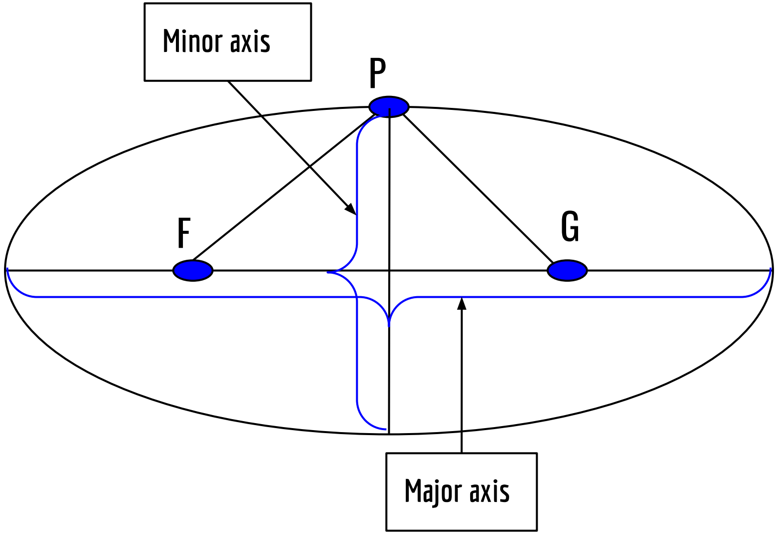
[Area = pi * a * b]where: a is semi-major axis & b is semi-minor axis.

When we extend the semi-minor axis evenly at the top, we get to form a triangle with sides a, b and c.

$$b^2 + c^2 = a^2$$

$$b^2 = a^2 - c^2$$





- F & G ⇒ focus
- FP + PG ⇒ Constant ⇒ foci = major axis
- Major axis ⇒ The longest diameter
- Minor axis ⇒ The shortest diameter
- Semi-major axis = ½ major axis
- Semi-minor axis = ½ minor axis

Ellipse equation:

$$\sqrt{(x+c)^2+(y-0)^2} + \sqrt{(x-c)^2+(y-0)^2} = 2\text{a}$$

$$\left(\sqrt{(x-c)^2+(y-0)^2}\right)^2 = \left(2a-\sqrt{(x+c)^2+(y-0)^2}\right)^2$$

$$(x-c)^2+y^2 = 4a^2-4a\sqrt{(x+c)^2+y^2}+(x+c)^2+\ y^2$$

$$x^2-2xc+c^2+y^2 = 4a^2-4a\sqrt{(x+c)^2+y^2}+x^2+2xc+c^2+y^2$$

$$(4a\sqrt{(x+c)^2+y^2}\,)^2 = (4a^2+4cx)^2$$

$$a^2[(x+c)^2+y^2] = (a^2+cx)^2$$

$$a^2x^2+2a^2xc+a^2c^2+a^2y^2 = (a^2+cx)^2$$

$$a^2x^2+2a^2xc+a^2c^2+a^2y^2 = a^2+2a^2cx+c^2x^2$$

$$a^2x^2-x^2c^2+a^2y^2 = a^2-a^2c^2$$

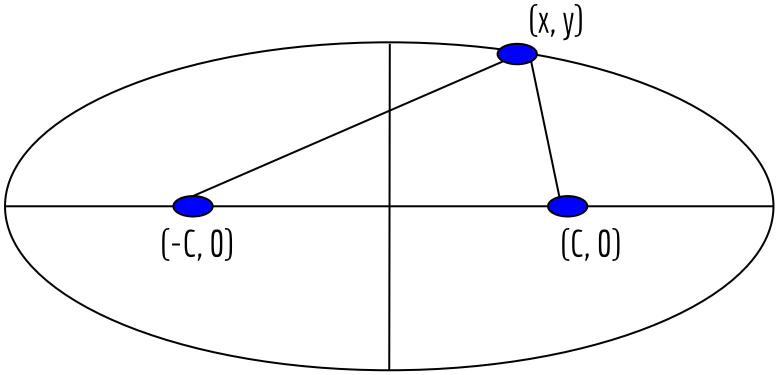
$$x^2(a^2-c^2)+a^2y^2 = a^2(a^2-c^2)$$

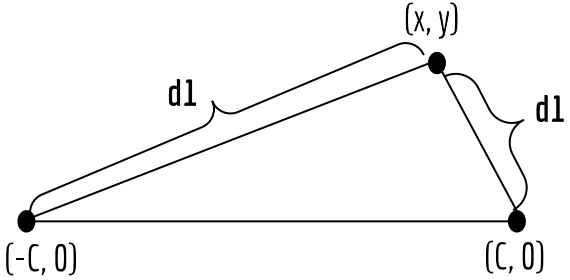
$$\frac{x^2}{a^2}+\frac{a^2y^2}{a^2(a^2-c^2)} = 1$$

$$\frac{x^2}{a^2}+\frac{y^2}{a^2-c^2} = 1$$

$$b^2 = a^2-c^2$$

$$\therefore \frac{x^2}{a^2}+\frac{y^2}{b^2} = 1$$

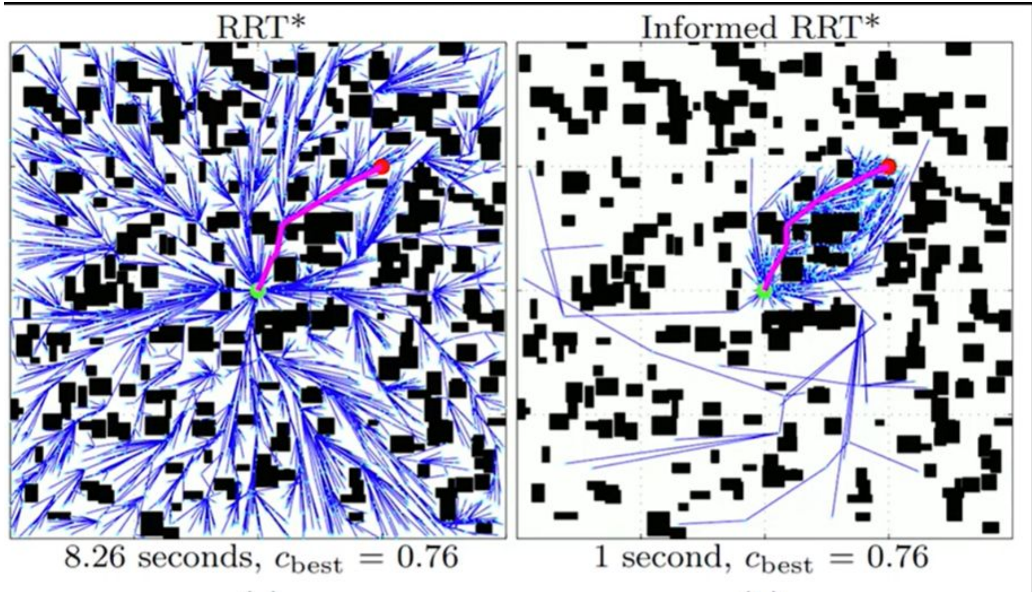




Informed RRT* algorithm

After finding the shortest path using RRT* algo, we sample within an ellipse region to reduce the time consumed.

The ellipse region covers the path and lets us sample only within there. That means that we don't need to sample anywhere else in the space, thus saving time and finding shorter paths.



The ellipse itself has to be declared by some parameters, as not any ellipse matches the optimization:

- The minimum cost of the ellipse is the euclidean distance between the start and the goal (C_min). It is not matter if there are any obstacles between them or not. The value of the C_min is always the euclidean distance.

C_min ⇒ The minimum possible cost.

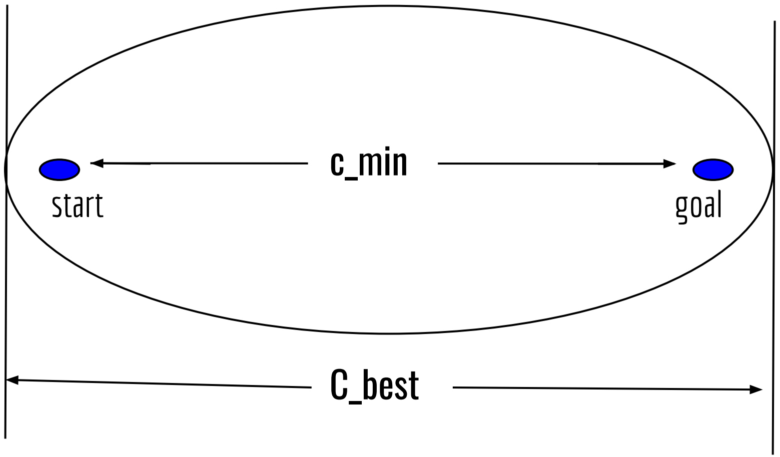
C_best ⇒ The best solution found (The cost of RRT*).

As we go along, the distance between the start and the goal (c_min) decreases, as we come close to the goal point. So, the cost will decrease (c_best) as the ellipse will get smaller and smaller.

C_best will be close to a line.

If we don't have any obstacles at all, c_best will pretty much be almost the same as c_min.

So, **Informed RRT* is RRT* algo result + sampling within an ellipse when the initial path is found using RRT* algo.**



For code