1) PRM (Probabilistic RoadMap):

This planner is able to compute collision-free paths for robots of almost any type moving among fixed obstacles .But where PRM is very useful is for robots with many degree of freedom. A general view of method is :

- *Learning Phase*
  - *Construction Step*
  - *Expansion Step*
- *Query Phase*

To create a roadmap using PRM, what we need to mention beforehand the number of nodes/vertices. Our roadmap is then created consisting of these many node, all in obstacle free regions.

Learning phase: A roadmap of the nodes which are not in obstacles and the edges joining them, also not passing through obstacles is made. This collision-free path is built repeating two steps : Pick a random configuration of the robot, check it for obstacle (This is achieved by a Boolean collision checking process.) and repeat this step until the slected configuration is obstacle-free. With a fast local planner, an attempt is made to connect the former configuration to the roadmap. The local planner stores local paths in roadmap. A slow local planner takes fewer nodes but takes more time to form a path. Whereas a fast local planner needs more nodes, but less time to form a path. To find a path between an initial and goal configuration, this step attempts to connect these configurations to the roadmap and searches in the roadmap for a sequence of local paths linking them.

Query Phase : We need to find path from the start and goal configurations to two nodes of the roadmap*.* We use an algorithm like Djikstra's to find the optimal path.

All these processes make the algorithm quite time consuming to run, which is why better path planning techniques like RRT were researched and obtained. Making a decision about RRT or PRM will depend on some external constrains like size of obstacles and free space. Time is also an important factor, although *PRM gives shorter paths than RRT.*

2) RRT (Rapidly-exploring Random Trees) :

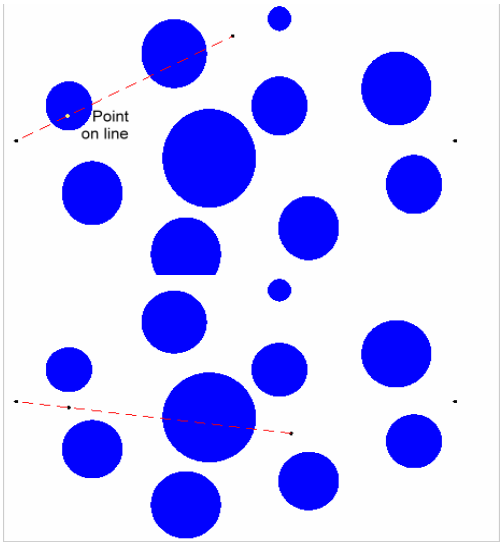
The RRT algorithm is quite straight forward. There is a starting node, from where the robot starts; and an ending node or the goal of the robot. Generally, a goal region is defined, which is a sphere or a circle of radius equal to the tolerance or limit. Once our path is connected to a node inside the limit, the path is successfully planned. RRT constructs a tree using random sampling in search space. In each iteration, a vertex/node is randomly generated and connected to the closest available vertex/node. Each time a vertex is generated, a check must be made that it lies outside of an obstacle. This is achieved by a Boolean collision checking process. The method of determining a random position is a design decision. Simple

methods, such as using built in random number generators can be used. For basic applications, such approaches suffice. Random number generators are not truly random and do contain a degree of bias. Generally, a configuration region (C-Space) is predefined in which the random vertex is generated. C-Space is divided into two parts- C_{free} (Region free from obstacles) and C_{obs} (region containing obstacles). Obviously, our C-Space contains the starting and the ending vertex. It is important to note that the straight line joining the node to its closest neighbor must NOT pass through any obstacles (C_{obs}). Our Boolean collision checking process will return a False value if an intersection between the line joining closest node and the randomly generated new node is passing through the C_{obs}. After a specific number of iterations/time, a path is obtained which contains nodes present only in the C_{free}. The algorithm ends when a node is generated within the goal region, or a limit is hit.
While the algorithm stays true to its name and is really helpful to rapidly explore our C-Space, it has a certain bias to select nodes in a region it is yet to explore. The RRT algorithm is quite simple to program; so this quality leads to a fast analysis to find a path. The RRT algorithm can be incorporated into a lot of planning systems.
There are also a few disadvantages for a basic RRT algorithm. The algorithm is not deterministic. A found path may not be the same the next time the algorithm is run on the same start and end nodes. If there are a lot of obstacles in the space, the chance of finding a node that is *not* lying in an obstacle, and the chance of making connection to the tree is very small. A narrow passage is also difficult to pass. The found path is sharp-edged, which can't be driven easily. An important and worth noting con is that once a path is established between two nodes, even if we generate 1000 more nodes, our path won't change.

Here is where the algorithm can be updated as needed. Smoothening of edges, or making the planned path more practical to travel is accomplished by RRG and RRT* algorithms.

NOTE:
Generally, the RRT algorithm is tweaked a little to make the process faster. For example, if we know that there are many obstacles present in our C-Space, we define a constant delta δ (or epsillon ε, depends on personal choice). δ is a small constant, depending on the size of our C-Space. Every time a node (node 2) is generated, we check if a node (node 3), at a distance δ from the closest possible node (node 1), lying on the line joining node 1 and node 2, lies in C_{free} or C_{obs}. If it lies in C_{free}, the node 3 is chained to node 1, and we proceed to the next iteration.

Here the node 3 lies in C_{obs} and hence is rejected. We hence move to the next iteration.



Here the node 3 lies in C_{free} and hence is accepted. After connecting the node, we move to the next iteration.

3) RRG (Rapidly-exploring Random Graph) :

In order to solve some of the cons we find in RRT, an updated approach RRG was researched upon. The RRG algorithm is similar to RRT. As RRT, it firstly attempts to chain/link the nearest node to the randomly generated node. If that connection is successful, the new node is added to the vertex structure. However, RRG is different is the following. Let us say that all the existing vertices are in a collection/set V.
Each time a new node (node 2) is added to the vertices V, then possible links are tested from all other vertices in V that are within a ball of radius r from that new node. To be precise, “The RRT algorithm links the nearest vertex towards the sample. The RRG algorithm first links the nearest vertex, and if such linkage is successful, it also links all the vertices in the ball of radius r ,producing a graph in general. In both cases, all the linkages resulting in collision-free trajectories are added to the graph as edges, and their terminal nodes as new vertices.” Hence, it could be said that *the RRT graph (a directed tree) is a subgraph of the RRG graph (an undirected graph). The two graphs share the same vertex set, and the edge set of the RRT is a subset of that of the RRG.*

4) RRT* :
The RRT* algorithm is obtained by modifying RRG, removing those edges that are redundant due to the cycles formed during the RRG algorithm. *RRT and RRT* graphs are directed trees with the same root and vertex set, and edge sets that are subsets of that of RRG, this creates the necessity to a "rewiring" of the RRT tree, in order to ensure that the nodes are reached through a minimum-cost path.* When the number of vertices/nodes generated goes larger, our optimised path goes on becoming straighter, as opposed to curved (most-probable) path in RRT.

As mentioned above, RRT* has two promising features called near neighbor search and rewiring tree operations. In Near neighbor operations, what we are doing essentially is to search for an optimal parent node in an a ball of radius *r*. Once we find such a node, we perform the rewiring operation. Rewiring operation rebuilds the tree within this radius to maintain the tree with minimal cost between tree connections. As the number of iterations increase, RRT* improves its path cost gradually due to its asymptotic quality, whereas RRT does not improves its curved and suboptimal path.
RRT* takes more execution time than RRT. This is due to the fact that RRT* uses two additional operations than RRT i.e., rewiring tree and best neighbor search. These two features improve the path cost to generate less curved and shorter path. On the other hand, these features also slow down the convergence rate and increase computational time.

***NOTE**: A ball/region of radius r was mentioned multiple times in above algorithms. This radius is calculated by a mathematical formula I choose to ignore here.*

...

Thus, while RRT* appears to be the most optimal path planning algorithm, it takes more time to plan the path than RRT. Although, both take less time in planning the path than PRM. A new algorithm named RRT*-Smart has also been researched on, which takes less time than RRT* to find the same optimal path. It still takes more time than RRT. So unless time is a very crucial factor, a hybrid model of RRT is the most preffered choice for path planning.