1. **Forward RRT**
In forward RRT, configuration space is divided into points and only one tree is expanded from ‘start’ and expanded till the tree doesn’t reach the ‘goal’.

1.1. **Pseudo Code:**
T ← \{q_s\}
while not contains (T,q_g)
    q ← SampleConfig()
    m ← NearestNeighbour(q)
    m’ ← Extend(m,q)

While extending the tree using the function “Extend(m,q)”, we can use two planners: B_s (Simple Planner) and Coll(q) (Collision Checker).

However we need to bias the tree to grow towards the goal. Hence we can modify the way we draw samples by modifying the function SampleConfig().

SampleConfig:
    p ← uniform[0,1]
    if p<p_sample
        return q_g
    else
        return uniform(sample)

In summary, we drop the points randomly on grid and if the point reached is q_g then we stop and backtrack our path.

**Figure 1** shows forward RRT.
2. **Bidirectional RRT**

2.1. **Option 1:**

There are two trees simultaneously expanded from ‘start’ and ‘goal’. These trees are expanded without being dependent on the other. During their expansion, if these trees encounter obstacles, they will find other path and go ahead. When these trees meet they will stop expanding and final path from start to goal is traced.

2.1.1 **Pseudo Code:**

\[
\begin{align*}
T_s & \leftarrow \{q_s\} \\
T_g & \leftarrow \{q_g\}
\end{align*}
\]

while not connected:

\[ q \leftarrow \text{SampleConfig}() \]

for \( T \) in \( \{T_s, T_g\} \):

\[ m \leftarrow \text{NearestNeighbor}(T) \]

\[ m' \leftarrow \text{Extend}(q, m) \]

\[ \text{connected} \leftarrow \text{CheckConnection}(T_g, T_s) \]

*Figure 2* shows option 1 for Bidirectional RRT
2.2. Option 2:

The path traced in Option 1 can be very long. To avoid it, when a random point is selected during the expansion of trees from ‘start’ and it hits the obstacle, a point nearest to this point is selected from ‘goal’ such that it doesn’t collide the obstacle. The tree from ‘start’ is then expanded and targeted towards this point from ‘goal’.

2.2.1 Pseudo Code:

```
T_s ← {q_s}
T_g ← {q_g}
while not connected:
    T_1, T_2 ← SortTrees(forward)
    q ← SampleConfig()
    m_1 ← NearestNeighbour(T_1)
    m_1' ← Extend(q, m)
    m_2 ← NearestNeighbour(T_2)
    m_2' ← Extend(m_1', m_2)
    Connected ← CheckConnection(T_g, T_s)
    forward = !forward
```
2.3. Path Shortening

It is obvious that result path of RRT consists of a large amount of short lines connected by turns. We can use some path shortening algorithms to reduce unnecessary movements to significantly shorten the final path. How can we deal with this problem?

A simple approach is to check if consecutive segments could be merge into one segment without causing any collision.

Algorithm:

**Input:** Path

**Output:** Shortened Path

For idx = 1 : length(Path) – 2
n1 = Path[idx – 1]
n2 = Path[idx + 1]
if Bs(n1, n2)
    path.remove(idx)

Figure 4 describes this algorithm.
To optimize more globally, we could iterate the previous algorithm until there is no more change.

Algorithm:
While (at least one path changed)
   For idx = 1 : length(Path) – 2
      n1 = Path[idx – 1]
      n2 = Path[idx + 1]
      if Bs(n1, n2)
         Path.remove(idx)
         Mark Path changed

Figure 5 and 6 describe path update to two consecutive optimization steps of this new algorithm. After three iterations, it found the shortest path between start point and goal point which is simply a straight line.
There is another approach which randomly picks two nodes in the tree and tries to connect them.

**Algorithm:**
**Input:** Path
**Output:** Shortened Path

While (some threshold)

- idx1 = random(0, length(Path) – 1)
- idx2 = random(idx1 + 1, length(Path) – 1)
- if Bs(Path[idx1], Path[idx2])
How we decide the threshold for the above algorithm?
There are several ways, for example:

1. Time limit
2. Path reduction amount

2.4. Bidirectional Randomized Physics-based Recognition Planning with Exact Pre-images

In robotics and motion planning, kino-dynamic planning also incorporates dynamic constraints, such as velocity, acceleration, force and torque. In the case of this physics-based planning problem, here we define the environment with the following four elements, the manipulator, the movable objects, the goal object and the goal region, as shown in Figure 7.

![Figure 7](image)

The goal of the task is to use the manipulator to move the goal object form the initial position to the goal region, given the pre-images of all the movable objects. The configuration space of this problem is defined as below,

\[ X = C^R*C^G*C^1*C^2*...*C^M, \]

where
- \( C^R \): configuration of manipulator
- \( C^G \): configuration of goal object
- \( C^1 \) ... \( C^M \): configuration of movable objects

The pseudo code of the Kinodynamic RRT is shown as below,

**Input:** The start node \( X_0 \)

**Output:** The path

\[ T \leftarrow \text{InitTree}(X_0) \]

\[ \text{path} \leftarrow \emptyset \]
while not ContainsGoal(T) do
    Xrand ← SampleConfiguration
    Xnear ← NearestNeighbor(T, Xrand)
    Xnew ← Extend(T, Xnear, Xrand)
    T ← UpdateTree(T, Xnew)
path ← ExtractPath(T)

A toy example is shown in figure 8.

Figure 8. The left picture shows the initial configuration. The right picture shows a random configuration that can connect to the goal configuration

Considering all the dynamic constraints, the results are shown in figure 9.

Figure 9. The intermediate steps to change the configuration from the initial configuration to the random configuration, considering all the relative dynamics.

2.5. Backward search:

2.5.1. Challenge-1:
In case of a bidirectional RRT in an environment containing movable objects, there is an additional problem in the backward search. While computing the arm trajectory the position of the movable object are not paid attention to (where they move after they are disturbed), so the forward and backward tree will not be exactly the same as marked by a box in the Figure 9. The three figures in the first row represent the position of the goal object (yellow) and the
obstacles (blue) during the forward search. The bottom three figures is for the backward search. as it is evident from the last figure in the two rows (represented by a dotted rectangle), the position of the obstacles are different.

Figure 9 Forward (top) and backward(bottom) search for a given goal region.

When the hand is moved forward, the obstacles are disturbed and moved in some way. When the hand is moved backward to the same point, the obstacles may not be disturbed in the same way. This produces a difference between the two trees. So though both the tree may have the same node, because of the difference in obstacle positions computer may not be able to combine both the trees.
Since the obstacles are not taken into account, the Goal is **underspecified**.

**Resolution:**
So to combat this problem, the reverse tree is grown in a lower dimensional space i.e., the moving obstacles are removed from the tree and only the goal object is considered (as shown in the bottom three figures in figure 10). In this way the backward tree is constructed. Once the both the forward and backward tree has the same node, the obstacles are placed in the backward by the referring their positions in the forward tree. In this way both the forward and the backward tree will have the same structure and can be matched. The matched forward and backward configurations are shown in the figure 11.
Figure 10 shows underspecified backward search

Figure 11 shows backward search with obstacles placed in correct position

2.5.2. Challenge-2:
In backward search, problem exists in backward simulation of the goal object as well. There is an one-to-many mapping between the start and the goal states. This might lead to the same problem as before.

Resolution:
This challenge is combated by computing the pre-image of the object as shown in the figure 12.