COMP417
Introduction to Robotics and Intelligent Systems
Path Planning, part one
Drawbacks of grid-based planners

• Grid-based planning works well for grids of up to 3-4 dimensions

• State-space discretization suffers from combinatorial explosion:

• If the state is \( x = [x_1, \ldots, x_D] \) and we split each dimension into \( N \) bins then we will have \( N^D \) nodes in the graph.

• This is not practical for planning paths for robot arms with multiple joints, or other high-dimensional systems.
(Sub)Sampling the state-space

• Need to find ways to reduce the continuous domain into a sparse representation: graphs, trees etc.

• Today:
  • Rapidly-exploring Random Tree (RRT),
  • Probabilistic RoadMap (PRM)
  • Visibility Planning
  • Smoothing Planned Paths
Main idea: maintain a tree of reachable configurations from the root

Main steps:

- Sample random state
- Find the closest state (node) already in the tree
- Steer the closest node towards the random state
RRT

1. \( V \leftarrow \{ x_{\text{init}} \}; \ E \leftarrow \emptyset; \)
2. \textbf{for} \( i = 1, \ldots, n \) \textbf{do}
3. \quad \quad \quad \quad x_{\text{rand}} \leftarrow \text{SampleFree}_i;
4. \quad \quad \quad \quad x_{\text{nearest}} \leftarrow \text{Nearest}(G = (V, E), x_{\text{rand}});
5. \quad \quad \quad \quad x_{\text{new}} \leftarrow \text{Steer}(x_{\text{nearest}}, x_{\text{rand}});
6. \quad \quad \textbf{if} \ \text{ObstacleFree}(x_{\text{nearest}}, x_{\text{new}}) \ \textbf{then}
7. \quad \quad \quad \quad V \leftarrow V \cup \{ x_{\text{new}} \}; \ E \leftarrow E \cup \{ (x_{\text{nearest}}, x_{\text{new}}) \};
8. \quad \textbf{return} \ G = (V, E);
RRT

1. $V \leftarrow \{x_{\text{init}}\}; E \leftarrow \emptyset$;
2. for $i = 1, \ldots, n$ do
   3. $x_{\text{rand}} \leftarrow \text{SampleFree}_i$;
   4. $x_{\text{nearest}} \leftarrow \text{Nearest}(G = (V, E), x_{\text{rand}})$;
   5. $x_{\text{new}} \leftarrow \text{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
   6. if $\text{ObstacleFree}(x_{\text{nearest}}, x_{\text{new}})$ then
      7. $V \leftarrow V \cup \{x_{\text{new}}\}; E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$;
3. return $G = (V, E)$;

**Things to pay attention to:**
SampleFree() needs to sample a random state from the uniform distribution. How do you sample rotations uniformly?
RRT

Things to pay attention to:

Nearest() searches for the nearest neighbor of a given vector. Brute force search examines |V| nodes (increasing). Is there a more efficient method?

1. \( V \leftarrow \{x_{\text{init}}\}; E \leftarrow \emptyset; \)
2. \( \text{for } i = 1, \ldots, n \text{ do} \)
3. \( x_{\text{rand}} \leftarrow \text{SampleFree}_i; \)
4. \( x_{\text{nearest}} \leftarrow \text{Nearest}(G = (V, E), x_{\text{rand}}); \)
5. \( x_{\text{new}} \leftarrow \text{Steer}(x_{\text{nearest}}, x_{\text{rand}}); \)
6. \( \text{if } \text{ObstacleFree}(x_{\text{nearest}}, x_{\text{new}}) \text{ then} \)
7. \( \quad V \leftarrow V \cup \{x_{\text{new}}\}; E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}; \)
8. \( \text{return } G = (V, E); \)
Things to pay attention to:

Steer() finds the controls that take the nearest state to the new state. Easy for omnidirectional robots. What about non-holonomic systems?
RRT

Things to pay attention to:

1. $V \leftarrow \{x_{\text{init}}\}; E \leftarrow \emptyset$;
2. for $i = 1, \ldots, n$ do
   3. $x_{\text{rand}} \leftarrow \text{SampleFree}_i$;
   4. $x_{\text{nearest}} \leftarrow \text{Nearest}(G = (V, E), x_{\text{rand}})$;
   5. $x_{\text{new}} \leftarrow \text{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
   6. if \text{ObstacleFree}(x_{\text{nearest}}, x_{\text{new}}) then
      7. $V \leftarrow V \cup \{x_{\text{new}}\}; E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$;
8. return $G = (V, E)$;

\text{ObstacleFree()}$ checks the path from the nearest state to the new state for collisions. How do you do collision checks?
RRT

```plaintext
1  V ← \{x_{init}\};   E ← \emptyset;
2  for \ i = 1, \ldots, \ n \ do
3     x_{rand} ← \text{SampleFree}_i;
4     x_{nearest} ← \text{Nearest}(G = (V, E), x_{rand});
5     x_{new} ← \text{Steer}(x_{nearest}, x_{rand});
6     if \ \text{ObstacleFree}(x_{nearest}, x_{new}) \ then
7        V ← V \cup \{x_{new}\}; E ← E \cup \{(x_{nearest}, x_{new})\};
8  return \ G = (V, E);
```

Upside of using ObstacleFree(): you don’t need to model obstacles in Steer(). For example, if Steer() computes LQR controllers you don’t need to model obstacles in the control computation.
RRT: uniform sampling

• Only tricky case is when the state contains rotation components
• For example: \( x = [W^B q^W p^W_{WB}] \)
• State involving both rotation and translation components is often called the pose of the system.
• Idea #1: Uniformly sample 3 Euler angles (roll, pitch, yaw)

3D rotation visualization:
rotation axis is a point on a sphere,
rotation angle is the direction of the red arrow
RRT: uniform sampling

• Only tricky case is when the state contains rotation components
• For example: \( x = \begin{bmatrix} W_B \mathbf{q} & W_B \mathbf{p}_{WB} \end{bmatrix} \)
• State involving both rotation and translation components is often called the pose of the system.
• Idea #1: Uniformly sample 3 Euler angles (roll, pitch, yaw)
RRT: uniform sampling

• Idea #2: Uniformly sample a quaternion
  • First, uniformly sample \( u_1, u_2, u_3 \in [0, 1] \)
  • Then output the unit quaternion

\[
q = [\sqrt{1 - u_1\sin(2\pi u_2)}, \sqrt{1 - u_1\cos(2\pi u_2)}, \sqrt{u_1\sin(2\pi u_3)}, \sqrt{u_1\cos(2\pi u_3)}]
\]

• Idea #3: Uniformly sample rotation matrices.
  • It’s possible but we won’t discuss it here.
RRT: finding the nearest neighbor

• Any alternatives to linear (brute force) search?
RRT: finding the nearest neighbor

• Any alternatives to linear (brute force) search?
• Idea #1: space partitioning, e.g. kd-trees

Balanced kd-tree: Can query in $O(\log n)$

Each split is done along the median of the points on that region
RRT: finding the nearest neighbor

• Any alternatives to linear (brute force) search?
• Idea #1: space partitioning, e.g. kd-trees
• Idea #2: locality-sensitive hashing
  • Maintains buckets
  • Similar points are placed on the same bucket
  • When searching consider only points that map to the same bucket
RRT: steering to a given state

• This is an optimal control problem, but without a specified time constraint

• For omnidirectional systems we can connect states by a straight line.

• For more complicated systems you could use LQR.

• You could also use a large set of predefined controls, one of which could be able to take the system close to the given state
RRT: steering to a given state

nonholonomic constraints

RRT for a robot with car-like kinematics

can the control problem get more difficult?
RRT: collision detection

- Main idea: bounding volume collision detection

RRT example: moving a piano
RRT: properties of the planning algorithm

#1: The RRT will eventually cover the space, i.e. it is a space-filling tree

![45 iterations](image1.png) ![2345 iterations](image2.png)

Source: Planning Algorithms, Lavalle
RRT: properties of the planning algorithm

#1: The RRT will eventually cover the space, i.e. it is a space-filling tree
#2: The RRT will NOT compute the optimal path asymptotically

This problem has been addressed in recent years by RRT*, BIT*, Fast-Marching Trees
RRT: properties of the planning algorithm

#1: The RRT will eventually cover the space, i.e. it is a space-filling tree
#2: The RRT will NOT compute the optimal path asymptotically
#3: The RRT will exhibit “Voronoi bias,” i.e. new nodes will fall in free regions of Voronoi diagram (cells consist of points that are closest to a node)
Voronoi diagram
RRT: properties of the planning algorithm

#1: The RRT will eventually cover the space, i.e. it is a space-filling tree

#2: The RRT will NOT compute the optimal path asymptotically

#3: The RRT will exhibit “Voronoi bias,” i.e. new nodes will fall in free regions of Voronoi diagram

#4: The probability of RRT finding a path increases exponentially in the number of iterations
RRT: properties of the planning algorithm

#1: The RRT will eventually cover the space, i.e. it is a space-filling tree

#2: The RRT will NOT compute the optimal path asymptotically

#3: The RRT will exhibit “Voronoi bias,” i.e. new nodes will fall in free regions of Voronoi diagram

#4: The probability of RRT finding a path increases exponentially in the number of iterations

#5: The distribution of RRT’s nodes is the same as the distribution used in SampleFree()
RRT variants: bidirectional search
Probabilistic RoadMaps (PRMs)

• RRTs were good for single-query path planning
• You need to re-plan from scratch for every query A → B

• PRM addresses this problem
• It is good for multi-query path planning
PRM

Space $\mathbb{R}^n$  forbidden space  Free/feasible space
Configurations are sampled by picking coordinates at random.
Each node is connected to its neighbors (e.g. within a radius)
PRM
To perform a query (A->B) we need to connect A and B to the PRM. We can do this by nearest neighbor search (kd-trees, hashing etc.)

```
1 V ← \{x_{init}\} \cup \{SampleFree_i\}_{i=1,...,n}; E ← \emptyset;
2 foreach v ∈ V do
3      U ← Near(G = (V, E), v, r) \ {v};
4      foreach u ∈ U do
5          if CollisionFree(v, u) then E ← E \cup \{(v, u), (u, v)\}
6  return G = (V, E);
```
To perform a query (A->B) we need to connect A and B to the PRM. We can do this by nearest neighbor search (kd-trees, hashing etc.)