COMP417
Introduction to Robotics and Intelligent Systems

Path Planning , part one

McGill

MRL Mobile Robotics Lab at McGill University

Drawbacks of grid-based planners

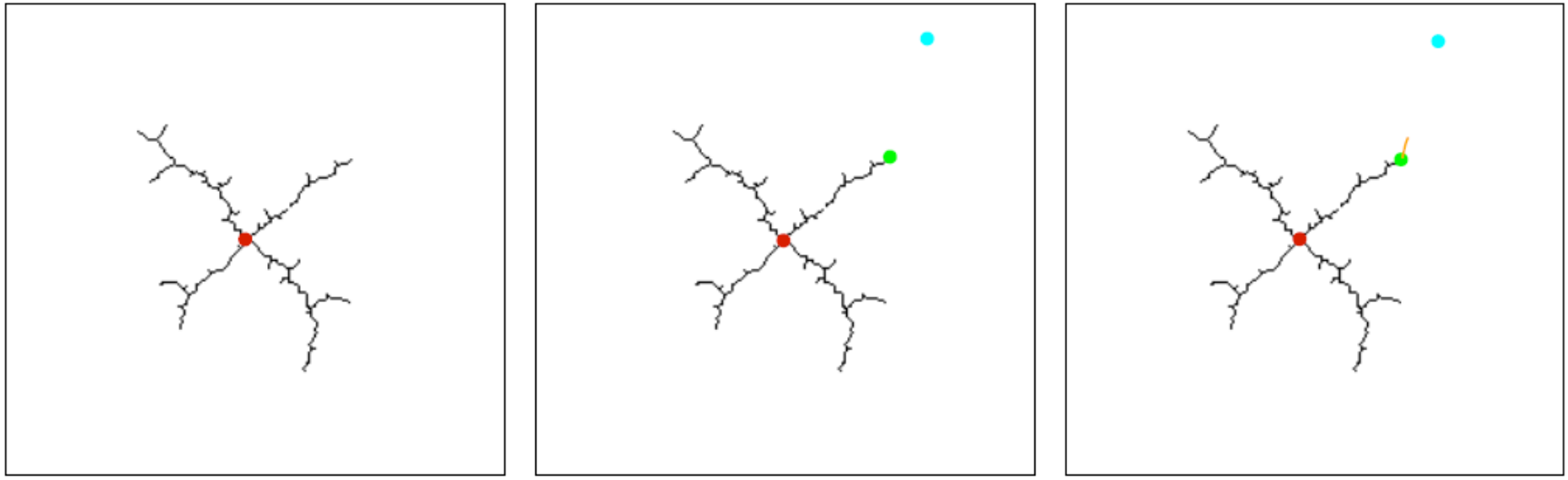
- Grid-based planning works well for grids of up to 3-4 dimensions
- State-space discretization suffers from combinatorial explosion:
- If the state is $\mathbf{x} = [x_1, ..., x_D]$ and we split each dimension into N bins then we will have N^D nodes in the graph.
- This is not practical for planning 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

RRT



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

$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
for $i = 1, \ldots, n$ **do**
  $x_{\text{rand}} \leftarrow \texttt{SampleFree}_i$;
  $x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$;
  $x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
  if $\texttt{ObtacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $V \leftarrow V \cup \{x_{\text{new}}\}$; $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$;
return $G = (V, E)$;

RRT

$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
for $i = 1, \ldots, n$ **do**
  $x_{\text{rand}} \leftarrow \texttt{SampleFree}_i$;
  $x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$;
  $x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
  if $\texttt{ObtacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $V \leftarrow V \cup \{x_{\text{new}}\}$; $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$;
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:

$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
for $i = 1, \ldots, n$ **do**
  $x_{\text{rand}} \leftarrow \texttt{SampleFree}_i$;
  $x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$;
  $x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
  if $\texttt{ObtacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $V \leftarrow V \cup \{x_{\text{new}}\}$; $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$;
return $G = (V, E)$;

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

RRT

Things to pay attention to:

$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
for $i = 1, \ldots, n$ **do**
  $x_{\text{rand}} \leftarrow \texttt{SampleFree}_i$;
  $x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$;
  $x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
  if $\texttt{ObtacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $V \leftarrow V \cup \{x_{\text{new}}\}$; $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$;
return $G = (V, E)$;

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:

$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
for $i = 1, \ldots, n$ **do**
  $x_{\text{rand}} \leftarrow \texttt{SampleFree}_i$;
  $x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$;
  $x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
  if $\texttt{ObtacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $V \leftarrow V \cup \{x_{\text{new}}\}$; $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$;
return $G = (V, E)$;

ObstacleFree() checks the path from the nearest state to the new state for collisions. How do you do collision checks?

RRT

Things to pay attention to:

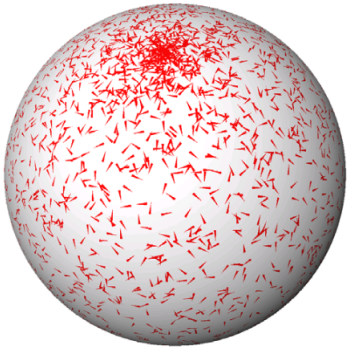
$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
for $i = 1, \ldots, n$ **do**
  $x_{\text{rand}} \leftarrow \texttt{SampleFree}_i$;
  $x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$;
  $x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
  if $\texttt{ObtacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $V \leftarrow V \cup \{x_{\text{new}}\}$; $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$;
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.

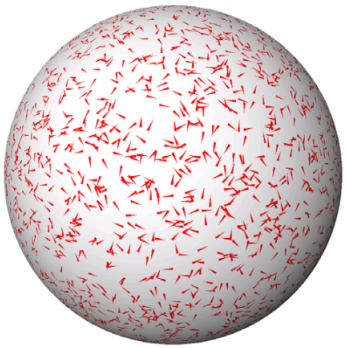
RRT: uniform sampling

- Only tricky case is when the state contains rotation components
- For example: $\mathbf{x} = [{}^W_B\mathbf{q}\ {}^W\mathbf{p}_{WB}]$
- 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



Idea #1
Not uniform after all

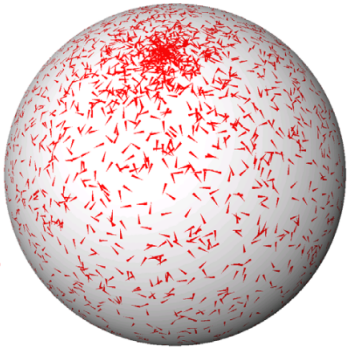


Correct, uniform

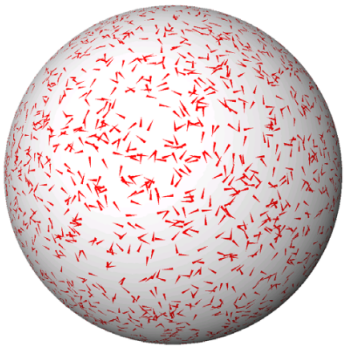
RRT: uniform sampling

- Only tricky case is when the state contains rotation components
- For example: $\mathbf{x} = [{}^W_B\mathbf{q}\ {}^W\mathbf{p}_{WB}]$
- 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)

Nonuniformity at the north pole caused by Gimbal Lock: same rotation parameterized by different Euler angles



Idea #1
Not uniform after all



Correct, uniform

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

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

- Idea #3: Uniformly sample rotation matrices.
- It's possible but we won't discuss it here.

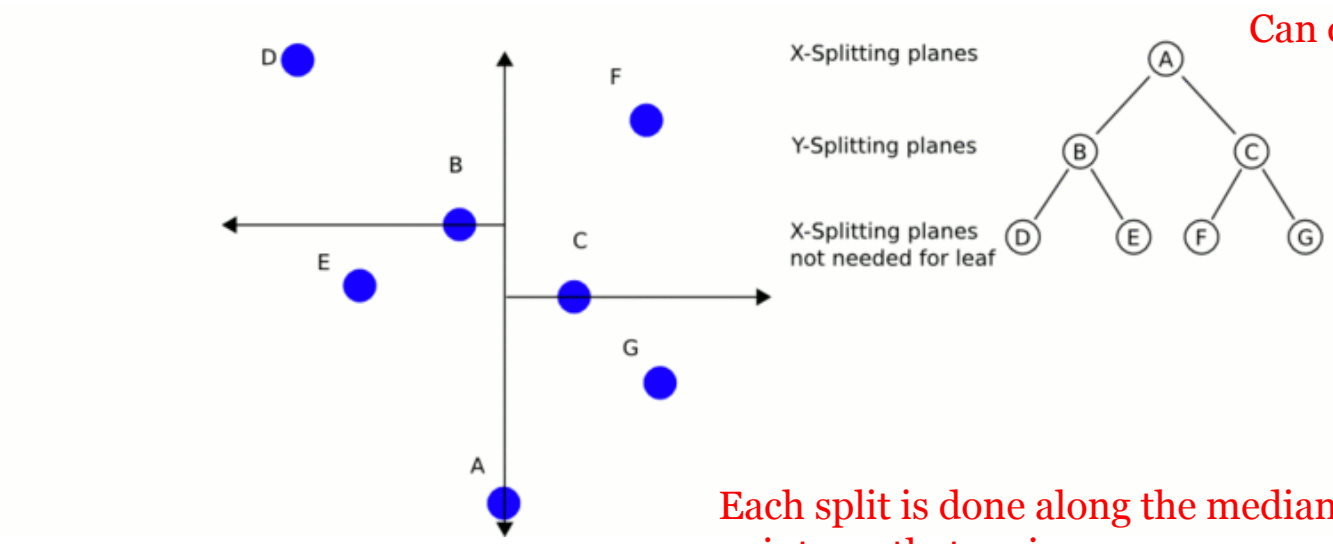
RRT: finding the nearest neighbor

- 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(logn)



Each split is done along the median of the points on that region

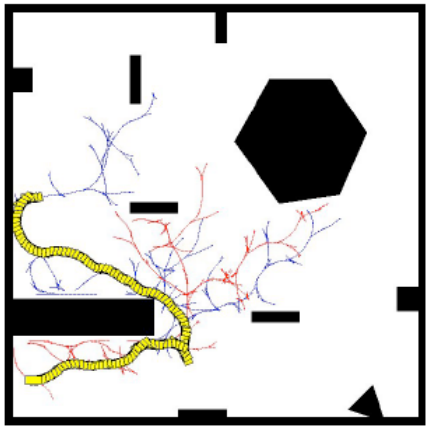
RRT: finding the nearest neighbor

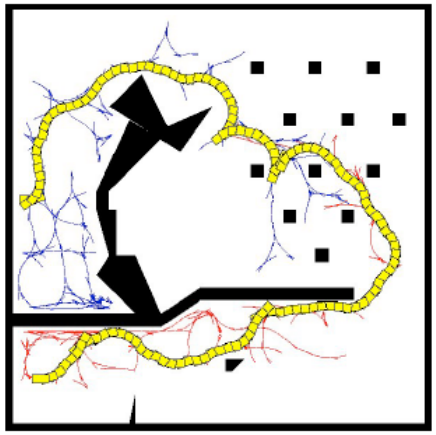
- Any alternatives to linear (brute force) search?
- Idea #1: space partitioning, e.g. kd-trees
- Idea #2: locality-sensitive hashing
  - Maintains buckets
  - Similar points are placed on the same bucket
  - When searching consider only points that map to the same bucket

RRT: steering to a given state

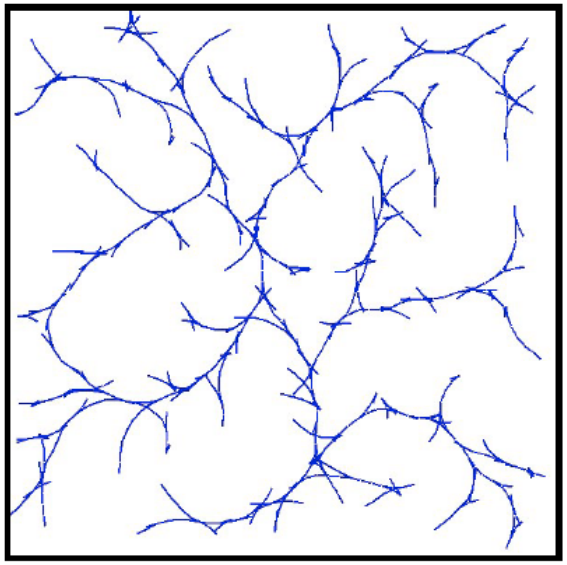
- This is an optimal control problem, but without a specified time constraint
- For omnidirectional systems we can connect states by a straight line.
- For more complicated systems you could use LQR.
- You could also use a large set of predefined controls, one of which could be able to take the system close to the given state

RRT: steering to a given state





nonholonomic constraints

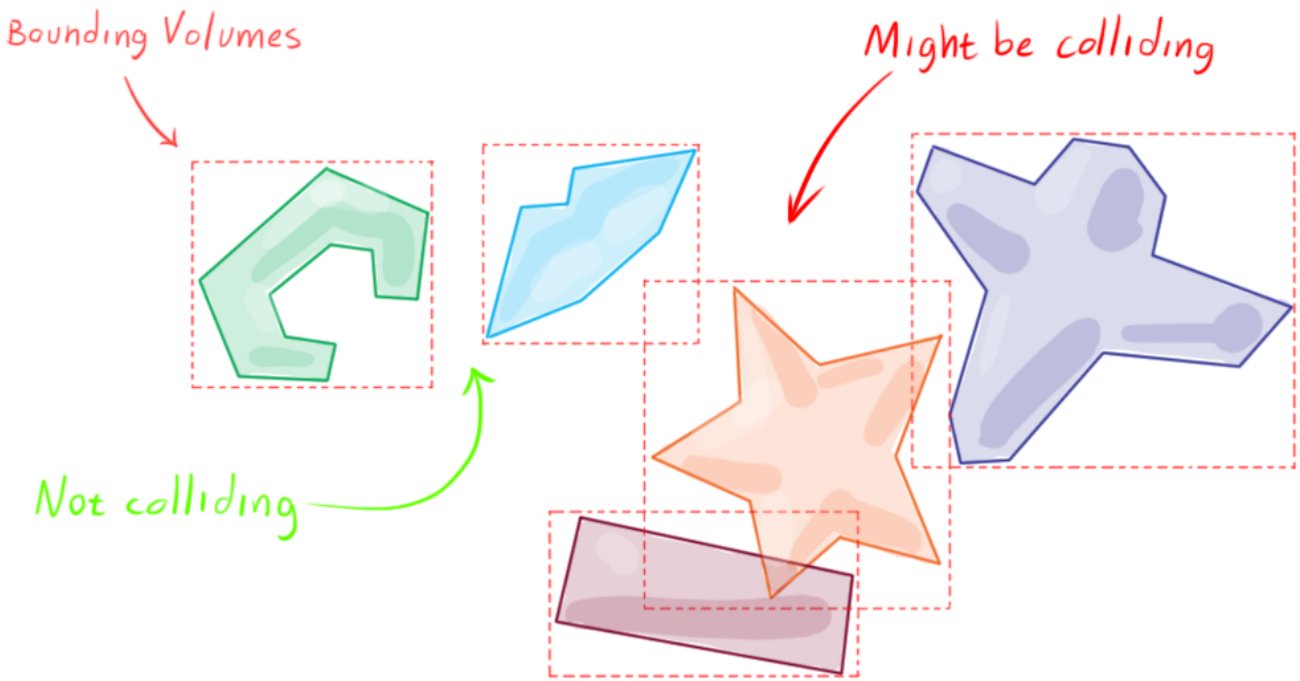


RRT for a robot with car-like kinematics

can the control problem get more difficult?

RRT: collision detection

- Main idea: bounding volume collision detection



Source: https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects

RRT example: moving a piano

RRT: properties of the planning algorithm

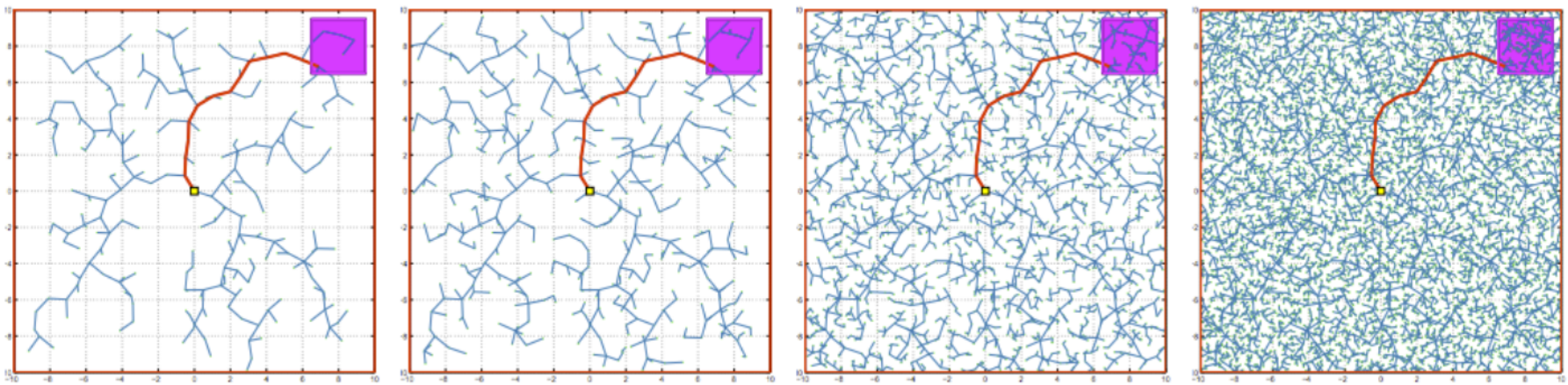
#1: The RRT will eventually cover the space, i.e. it is a space-filling tree

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



Source: Karaman, Frazzoli, 2010

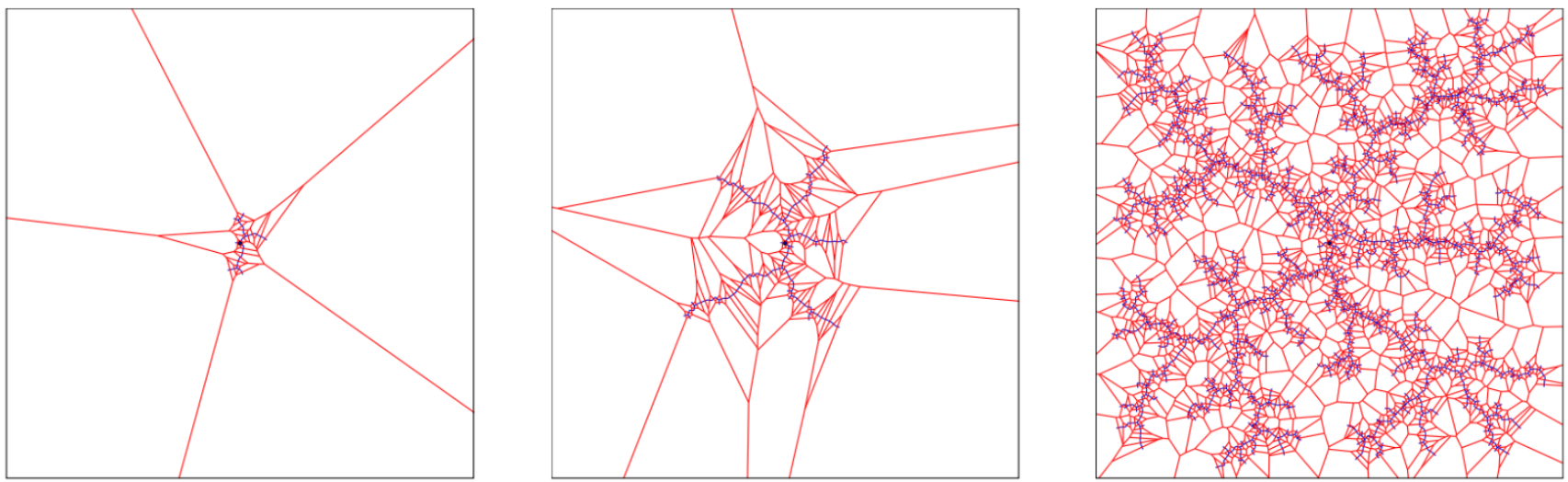
This problem has been addressed in recent years by RRT*, BIT*, Fast-Marching Trees

RRT: properties of the planning algorithm

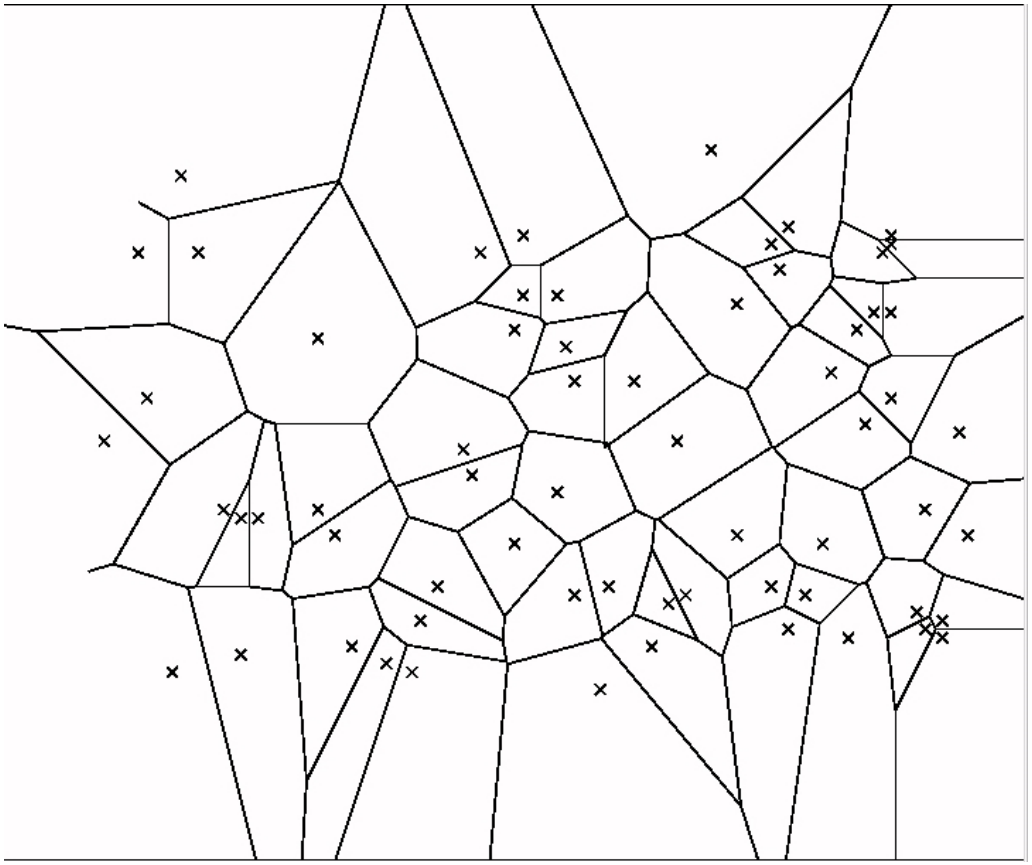
#1: The RRT will eventually cover the space, i.e. it is a space-filling tree

#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()

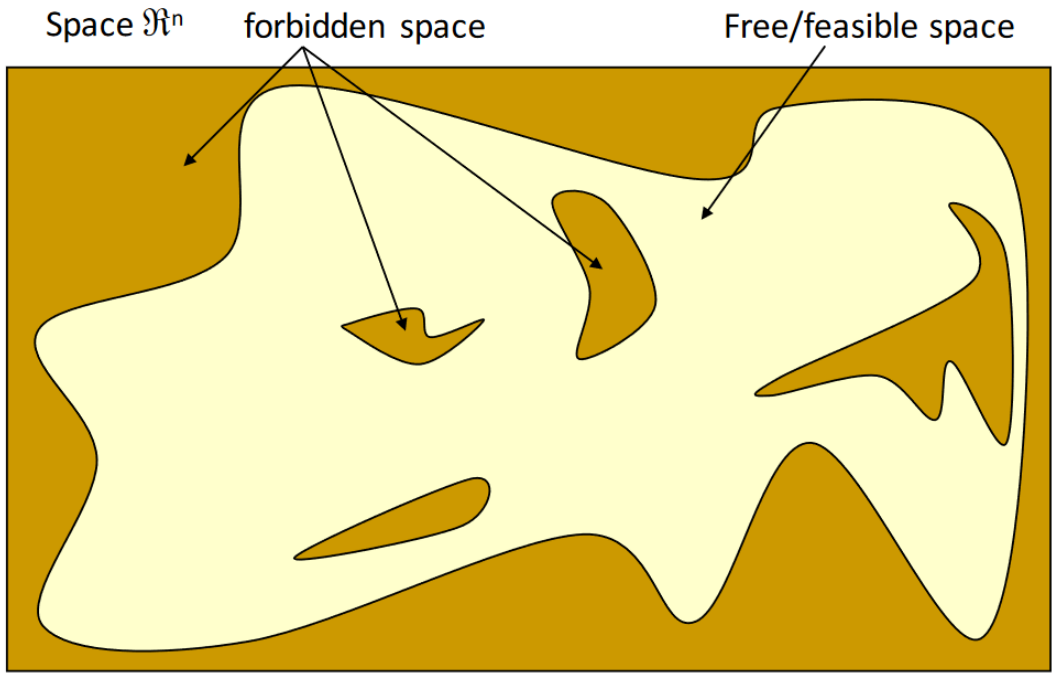
RRT variants: bidirectional search

Probabilistic RoadMaps (PRMs)

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

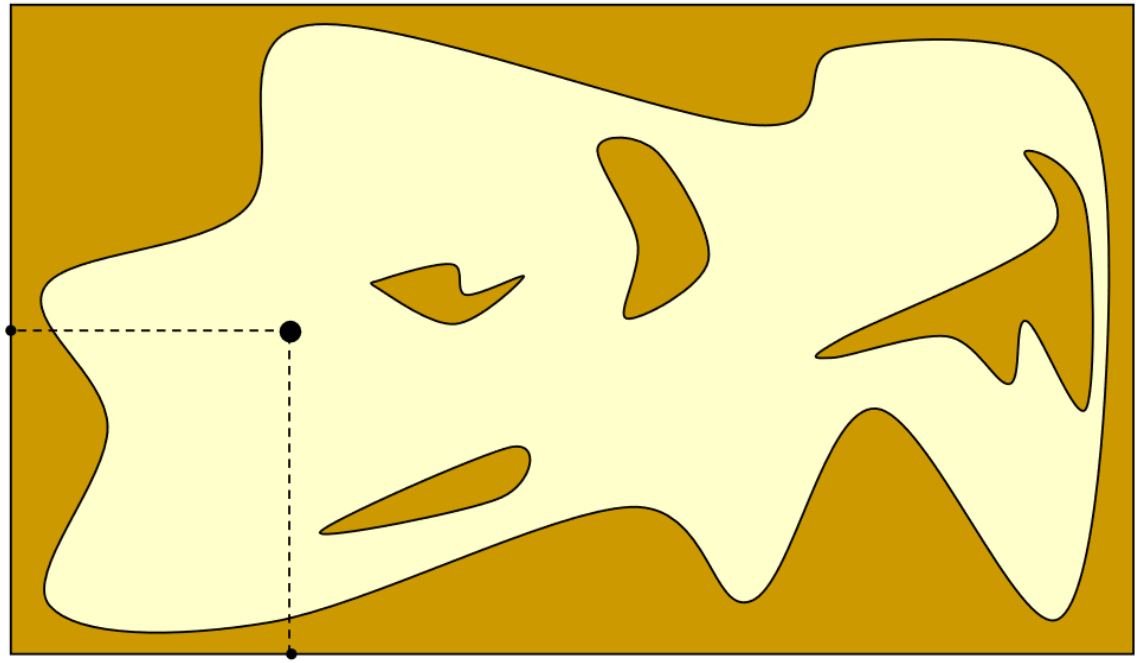
- PRM addresses this problem
- It is good for multi-query path planning

PRM

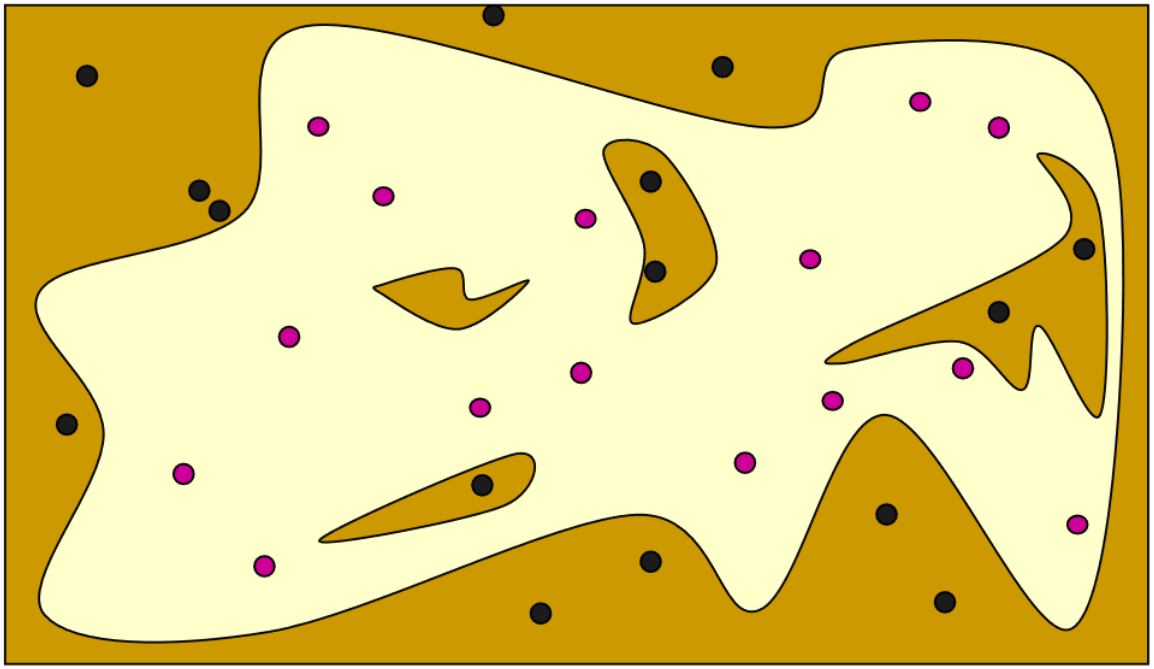


PRM

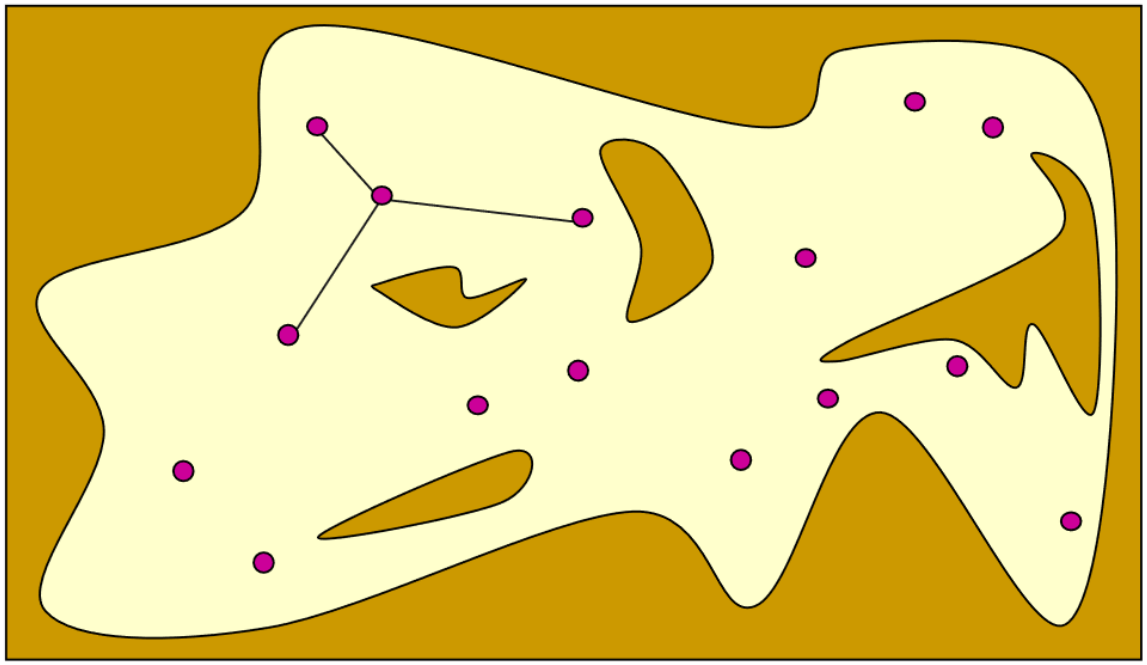
Configurations are sampled by picking coordinates at random



PRM

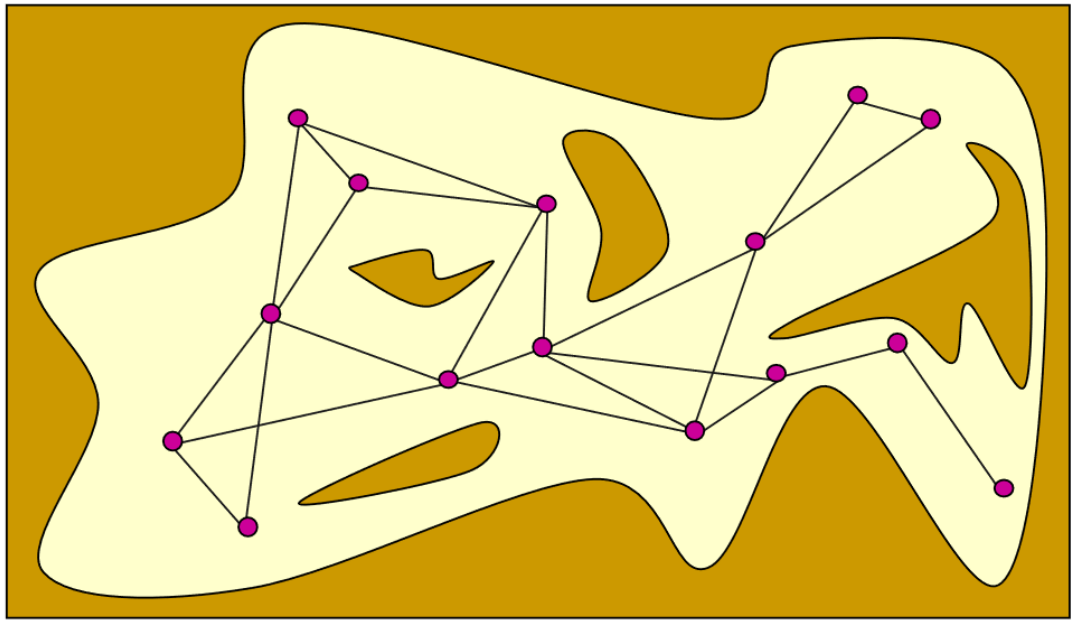


PRM



Each node is connected to its neighbors (e.g. within a radius)

PRM



PRM

$V \leftarrow \{x_{\text{init}}\} \cup \{\texttt{SampleFree}_i\}_{i=1,\ldots,n}$; $E \leftarrow \emptyset$;
foreach $v \in V$ **do**
  $U \leftarrow \texttt{Near}(G = (V, E), v, r) \setminus \{v\}$;
  foreach $u \in U$ **do**
    if $\texttt{CollisionFree}(v, u)$ **then** $E \leftarrow E \cup \{(v, u), (u, v)\}$
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.)

PRM

$V \leftarrow \{x_{\text{init}}\} \cup \{\texttt{SampleFree}_i\}_{i=1,\ldots,n}$; $E \leftarrow \emptyset$;
foreach $v \in V$ **do**
  $U \leftarrow \texttt{Near}(G = (V, E), v, r) \setminus \{v\}$;
  foreach $u \in U$ **do**
    if $\texttt{CollisionFree}(v, u)$ **then** $E \leftarrow E \cup \{(v, u), (u, v)\}$
return $G = (V, E)$;

Range search can be done efficiently using a kd-tree

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.)