EE211: Robotic Perception and Intelligence

Lecture 5 Sampling-based Methods

Jiankun WANG

Department of Electronic and Electrical Engineering
Southern University of Science and Technology

Undergraduate Course, Oct 2024

Outline

Outline

Complete Algorithms for Motion Planning

Get from point A to point B avoiding obstacles

Complete Algorithms for Motion Planning

Motion Planning Problem

Consider a dynamical control system defined by an ODE of the form

$$dx/dt = f(x, u), \; x(0) = x_{init},$$

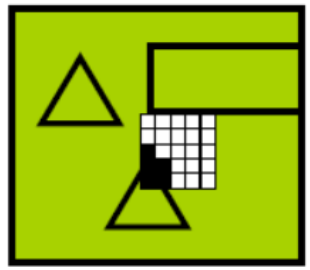
where x is the state, u is the control. Given an obstacle set $\mathcal{X}_{obs} \subset \mathbb{R}^d$, and a goal set $\mathcal{X}_{goal} \subset \mathbb{R}^d$, a complete algorithm for motion planning must find, if it exists, a control signal u such that the solution satisfies $x(t) \notin \mathcal{X}_{obs} \; \forall t \in \mathbb{R}_+$, and $x(t) \in \mathcal{X}_{goal} \; \forall t > T$ for some finite $T \geq 0$, or return failure if no such control signal exists. The algorithm must terminate in finite time.

- Basic problem in robotics (and intelligent life in general).
- Provably very hard: a basic version (the Generalized Piano Mover's problem) is known to be PSPACE-hard [Reif, '79].

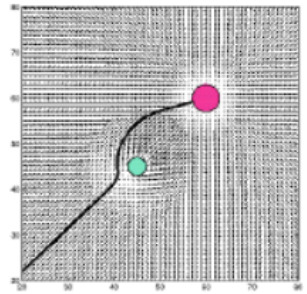
Motion Planning in Practice

Many techniques have been proposed to solve motion planning problems, e.g.,

- Algebraic planners: Explicit representation of obstacles. Use complicated algebra (visibility computations or projections) to find the path. Complete, but impractical.



- Discretization + graph search: Analytic/grid-based methods do not scale well to high dimensions. Graph search methods (A*, D*, etc.) can be sensitive to graph size. Resolution complete.



- Potential fields/navigation functions: Virtual attractive forces towards the goal, repulsive forces away from the obstacles. No completeness guarantees, unless "navigation functions" are available - very hard to compute in general.

These algorithms achieve tractability by foregoing completeness altogether, or achieving weaker forms of it, e.g., resolution completeness.

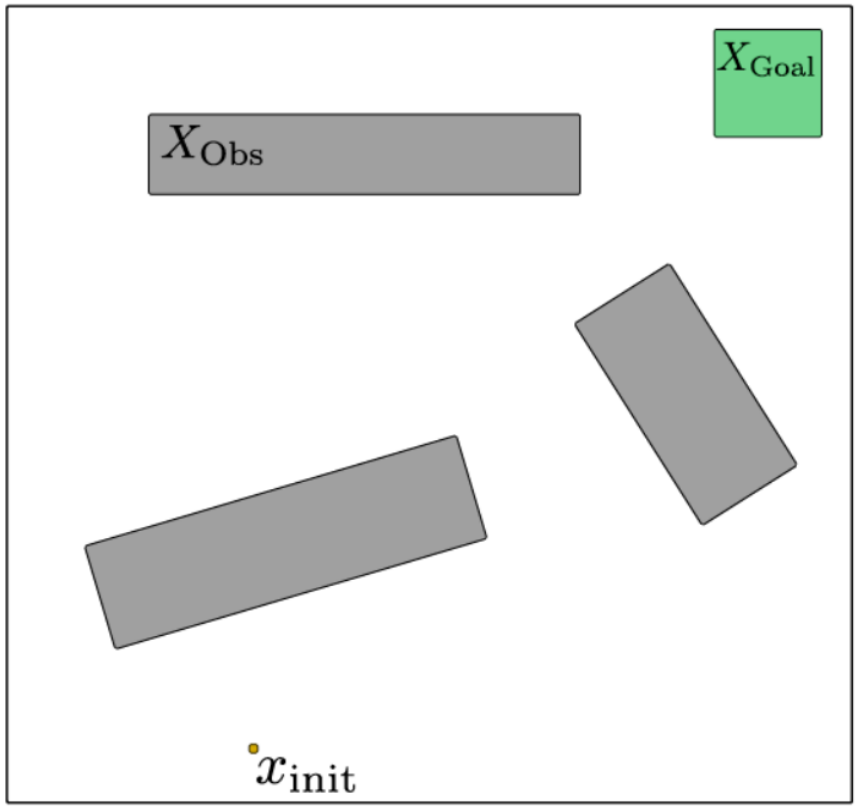
Sampling-based Algorithms

- A class of motion planning algorithms that has been very successful in practice is based on (batch or incremental) sampling methods: solutions are computed based on samples drawn from some dense sequence (stochastic or deterministic).
- Sampling-based algorithms retain a weaker form of completeness, e.g., probabilistic or resolution completeness.
- Probabilistic RoadMaps (PRM) [Kavraki & Latombe, 1994] was the first planner to demonstrate the ability to solve practical planning problems "in high dimensions" ($>$ 4-5 dimensions!).

Probabilistic RoadMaps (PRM)

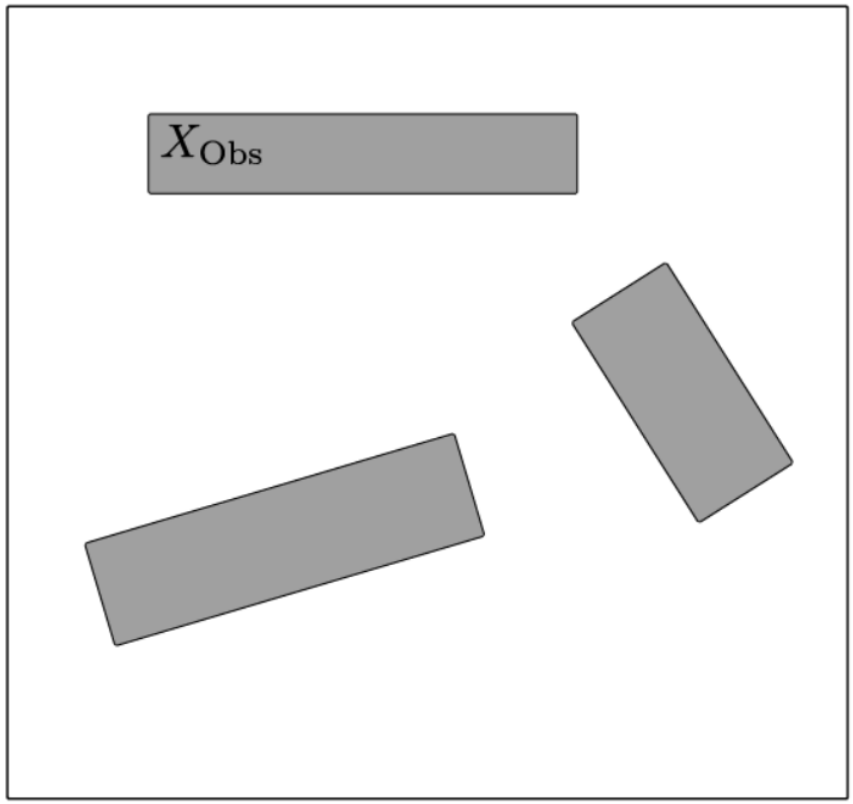
- Mainly geared towards "multi-query" motion planning problems.
- Idea: build (offline) a graph (i.e., the roadmap) representing the "connectivity" of the environment; use this roadmap to figure out paths quickly at run time.
- Offline pre-processing phase:
  - Sample n points from $\mathcal{X}_{free} = [0,1]^d \setminus \mathcal{X}_{obs}$.
  - Try to connect these points using a fast "local planner" (e.g., ignore obstacles).
  - If connection is successful, add an edge between the points.
- At run time:
  - Connect the start and end goal to the closest nodes in the roadmap.
  - Find a (shortest) path on the roadmap.

Probabilistic RoadMap (PRM) Algorithm: Step 1



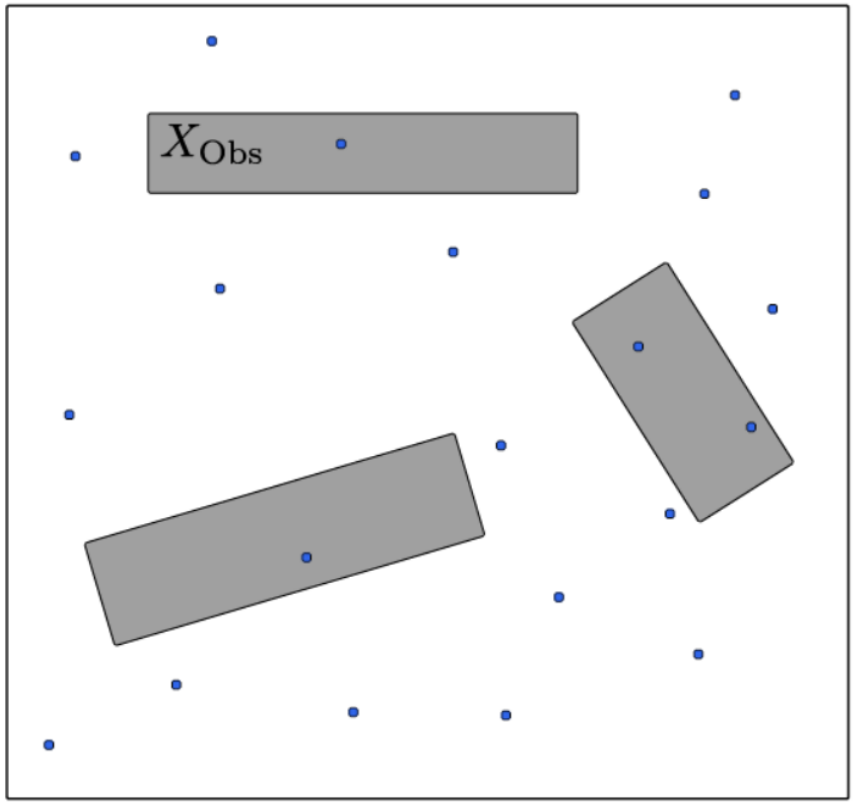
- Simplified PRM: connections attempted to all vertices within distance r.
- "Real" PRM: connections attempted in increasing order of distance, only to other connected components in the graph.

Probabilistic RoadMap (PRM) Algorithm: Step 2



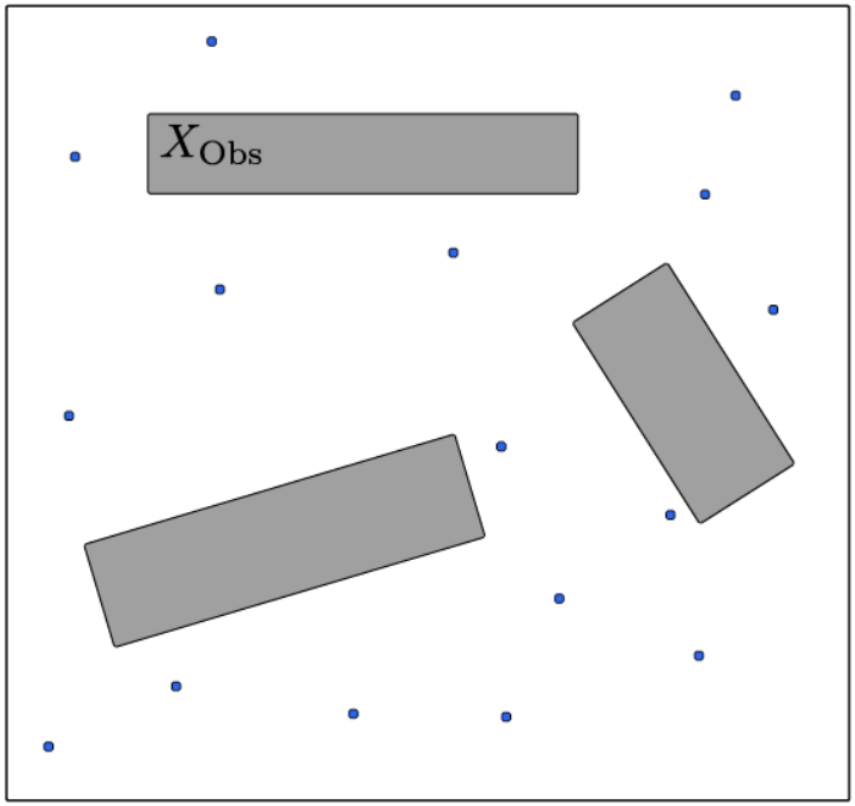
- Simplified PRM: connections attempted to all vertices within distance r.
- "Real" PRM: connections attempted in increasing order of distance, only to other connected components in the graph.

Probabilistic RoadMap (PRM) Algorithm: Step 3



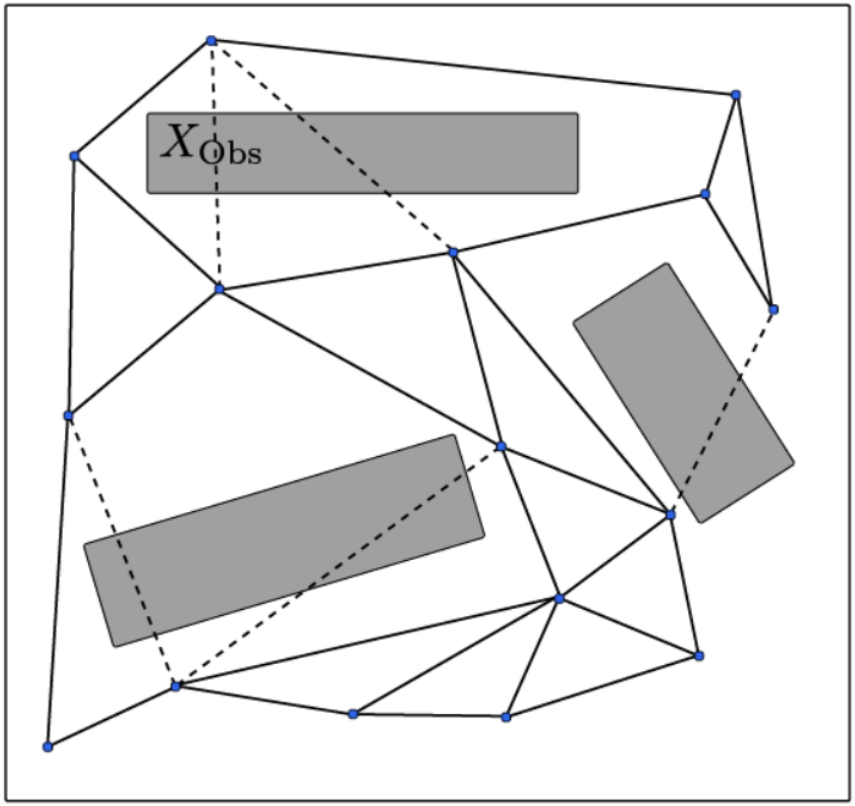
- Simplified PRM: connections attempted to all vertices within distance r.
- "Real" PRM: connections attempted in increasing order of distance, only to other connected components in the graph.

Probabilistic RoadMap (PRM) Algorithm: Step 4



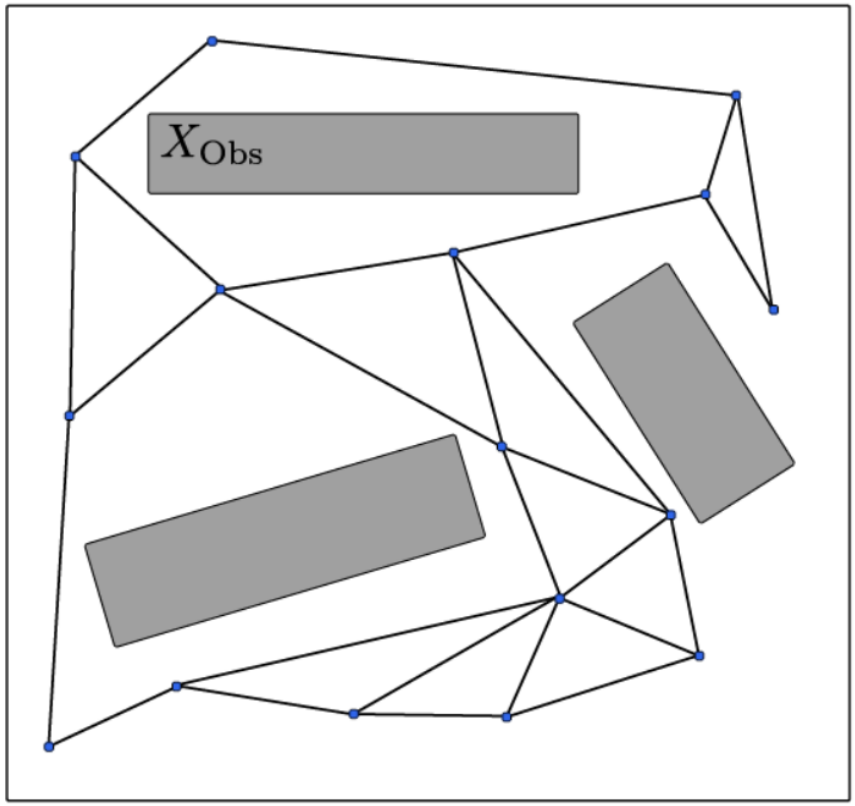
- Simplified PRM: connections attempted to all vertices within distance r.
- "Real" PRM: connections attempted in increasing order of distance, only to other connected components in the graph.

Probabilistic RoadMap (PRM) Algorithm: Step 5



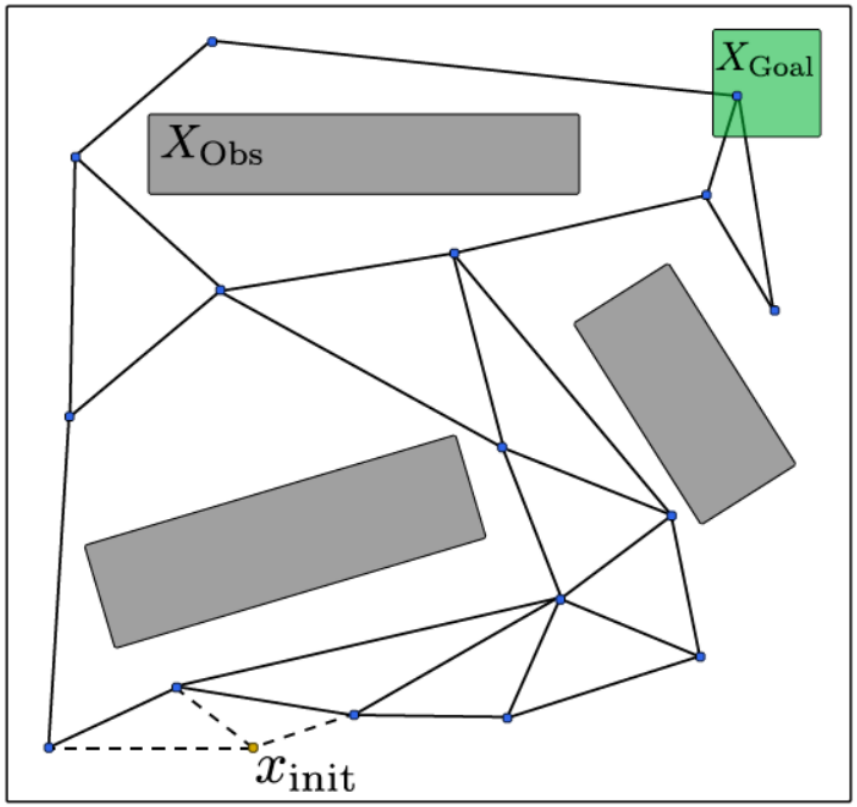
- Simplified PRM: connections attempted to all vertices within distance r.
- "Real" PRM: connections attempted in increasing order of distance, only to other connected components in the graph.

Probabilistic RoadMap (PRM) Algorithm: Step 6



- Simplified PRM: connections attempted to all vertices within distance r.
- "Real" PRM: connections attempted in increasing order of distance, only to other connected components in the graph.

Probabilistic RoadMap (PRM) Algorithm: Step 7

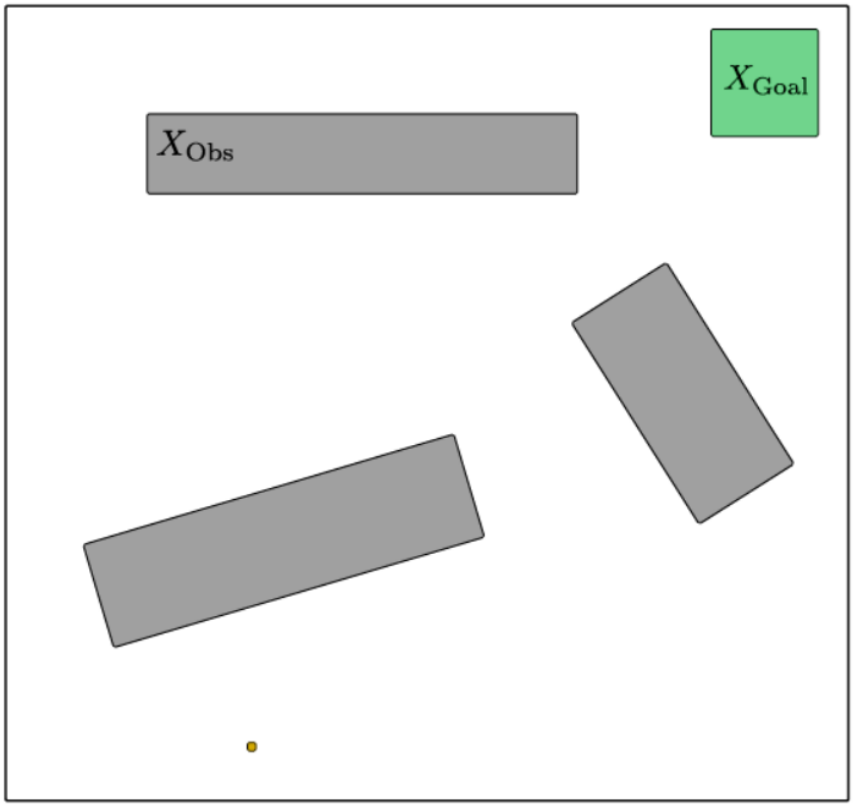


- Simplified PRM: connections attempted to all vertices within distance r.
- "Real" PRM: connections attempted in increasing order of distance, only to other connected components in the graph.

Rapidly-exploring Random Trees

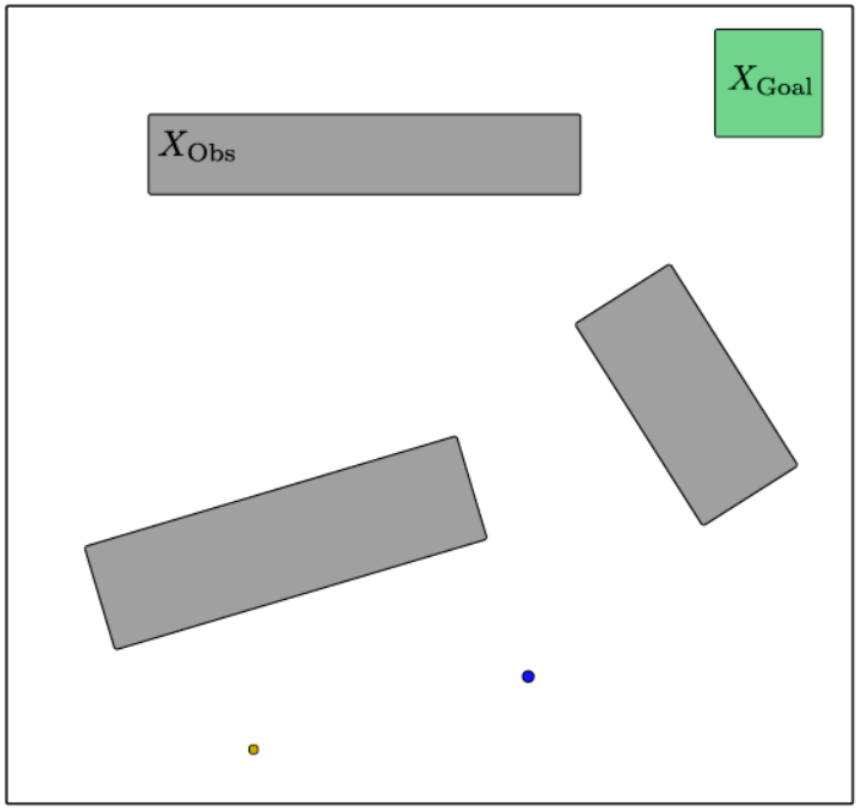
- Introduced by LaValle and Kuffner in 1998.
- Appropriate for single-query planning problems.
- Idea: build (online) a tree, exploring the region of the state space that can be reached from the initial condition.
- At each step: sample one point from $\mathcal{X}_{free}$, and try to connect it to the closest vertex in the tree.
- Very effective in practice, "Voronoi bias".

Rapidly-exploring Random Tree (RRT) Algorithm: Step 1



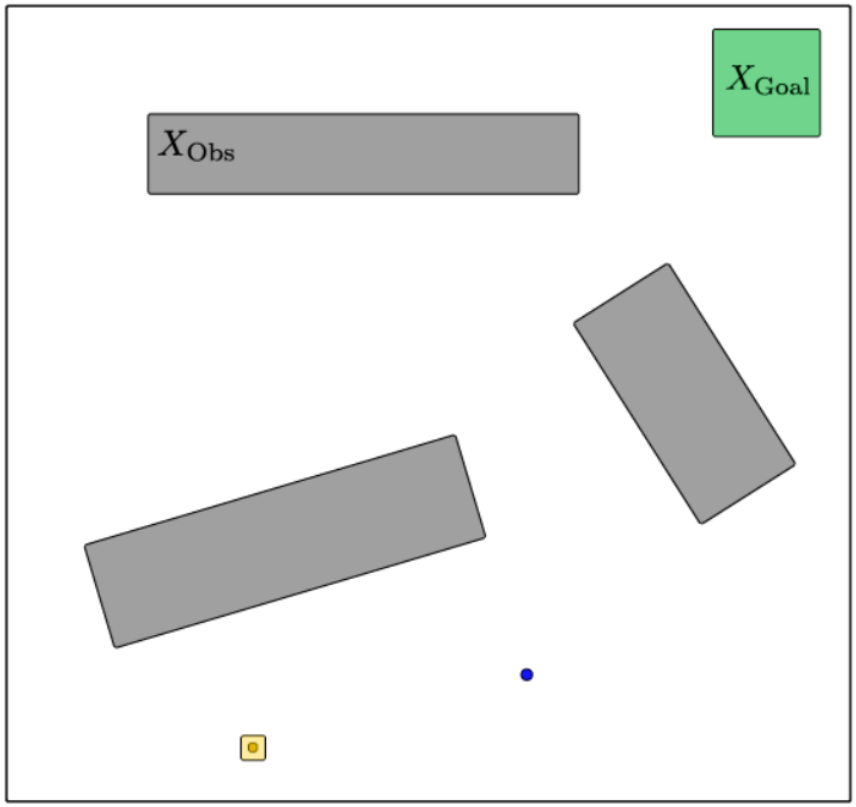
- Connections attempted to the nearest neighbor only.

Rapidly-exploring Random Tree (RRT) Algorithm: Step 2



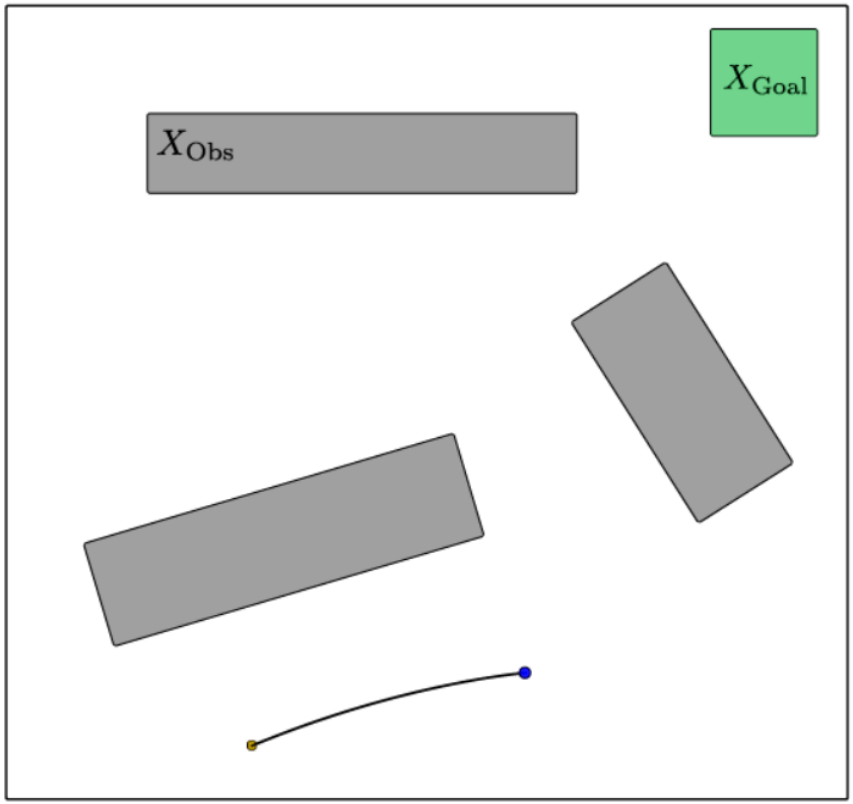
- Connections attempted to the nearest neighbor only.

Rapidly-exploring Random Tree (RRT) Algorithm: Step 3



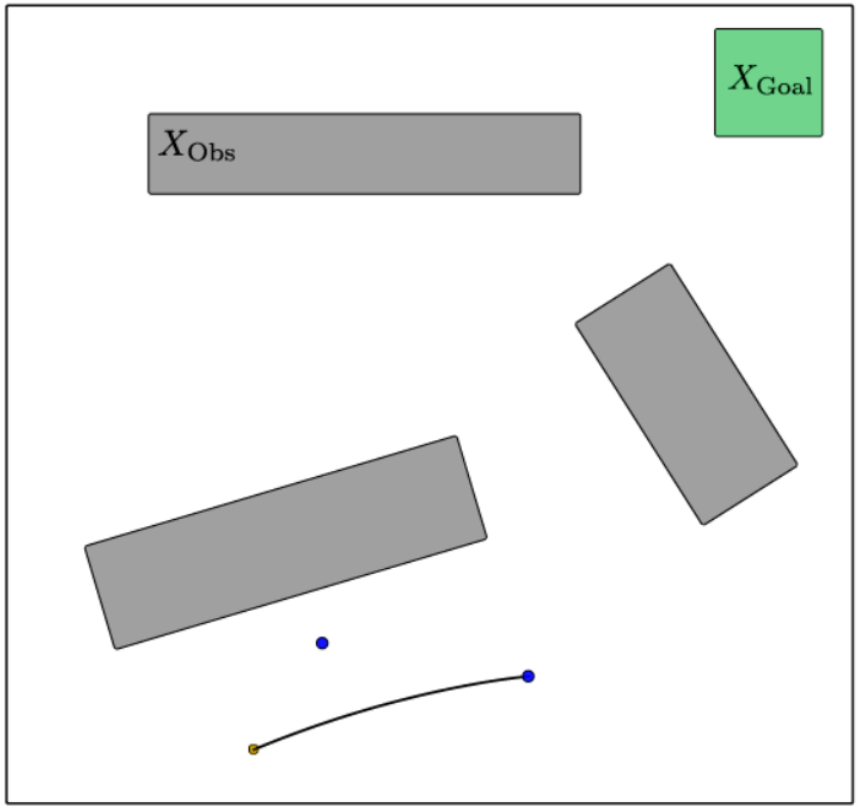
- Connections attempted to the nearest neighbor only.

Rapidly-exploring Random Tree (RRT) Algorithm: Step 4



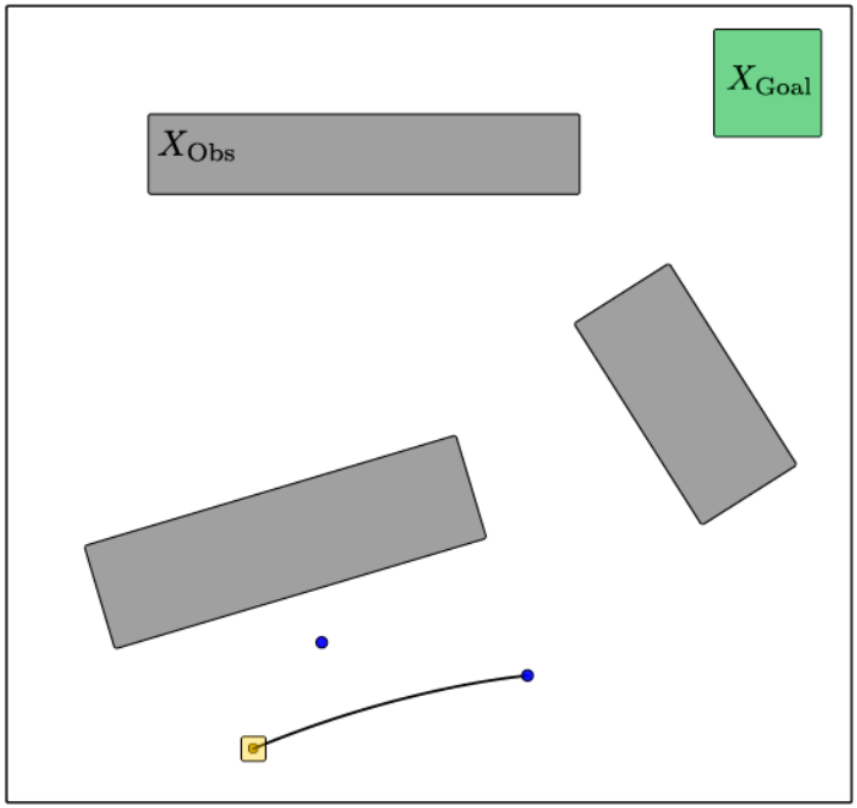
- Connections attempted to the nearest neighbor only.

Rapidly-exploring Random Tree (RRT) Algorithm: Step 5



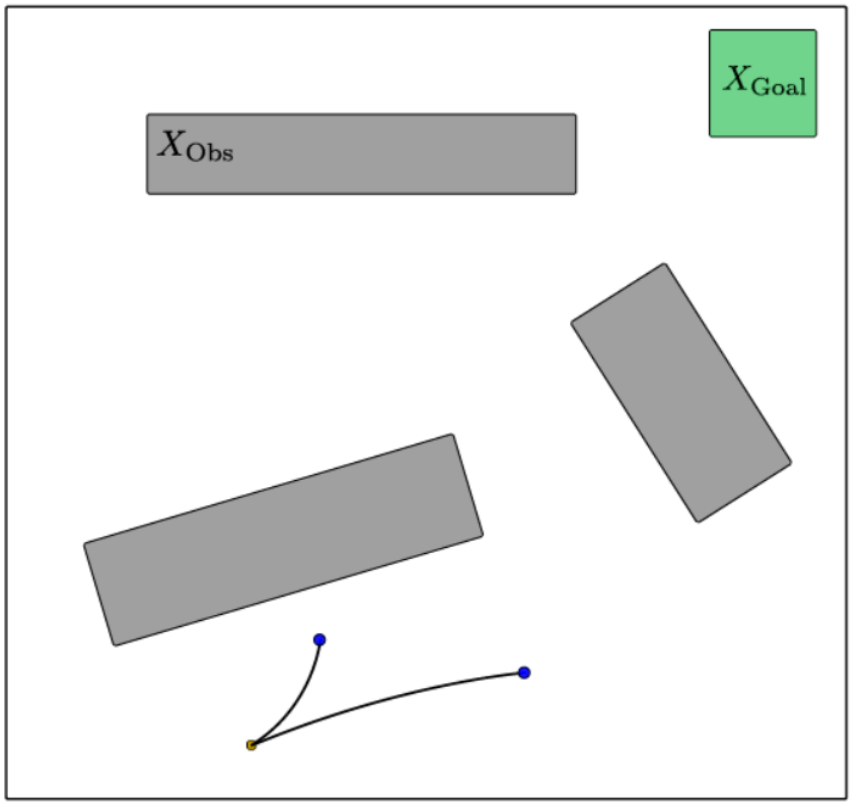
- Connections attempted to the nearest neighbor only.

Rapidly-exploring Random Tree (RRT) Algorithm: Step 6



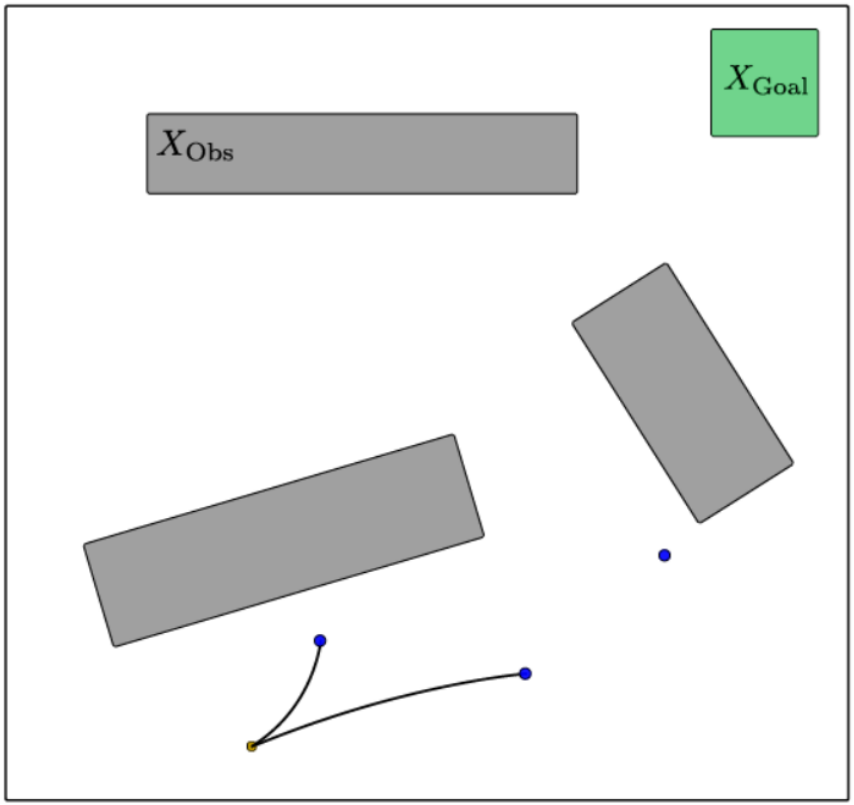
- Connections attempted to the nearest neighbor only.

Rapidly-exploring Random Tree (RRT) Algorithm: Step 7



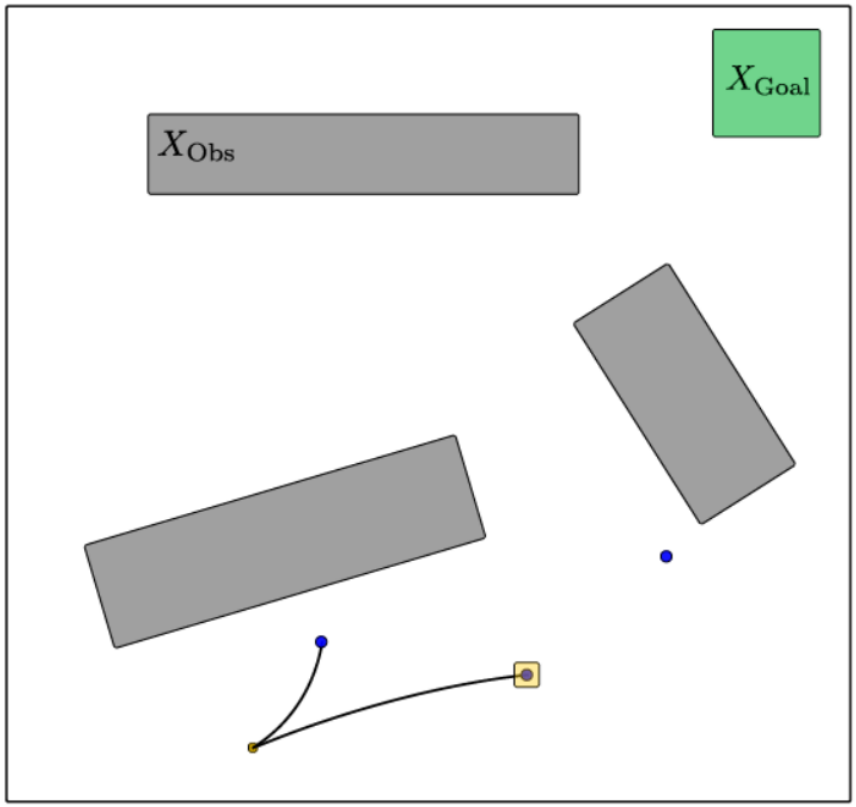
- Connections attempted to the nearest neighbor only.

Rapidly-exploring Random Tree (RRT) Algorithm: Step 8



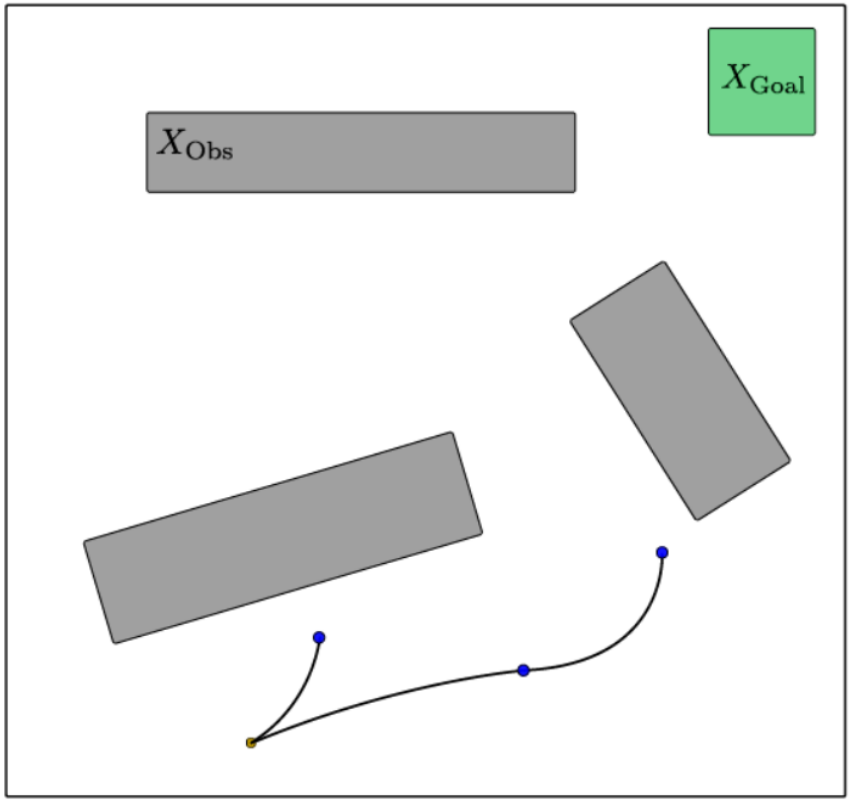
- Connections attempted to the nearest neighbor only.

Rapidly-exploring Random Tree (RRT) Algorithm: Step 9



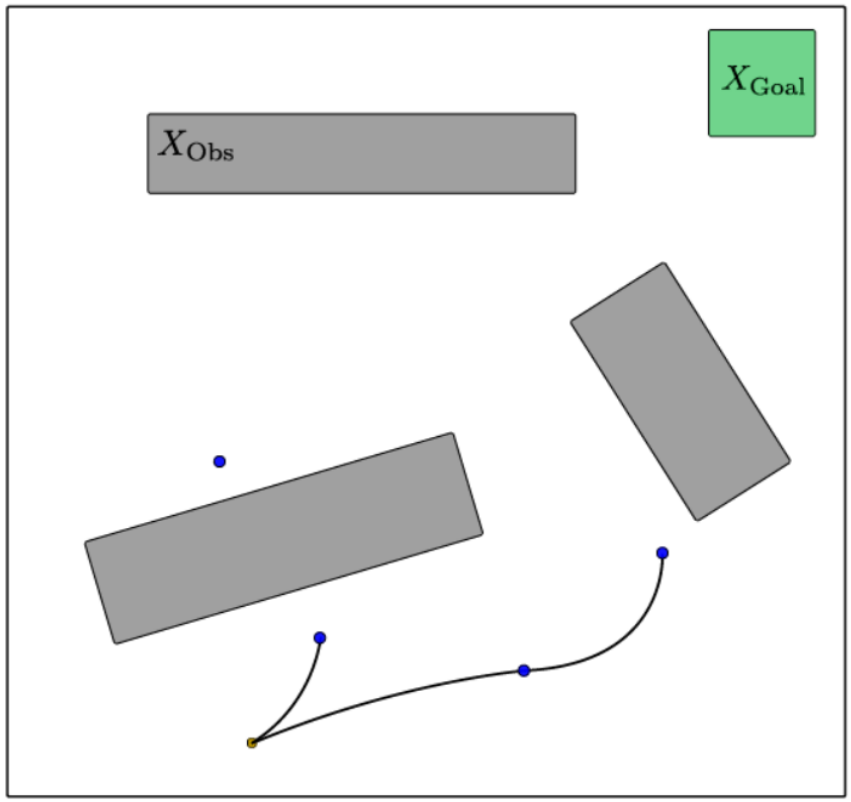
- Connections attempted to the nearest neighbor only.

Rapidly-exploring Random Tree (RRT) Algorithm: Step 10



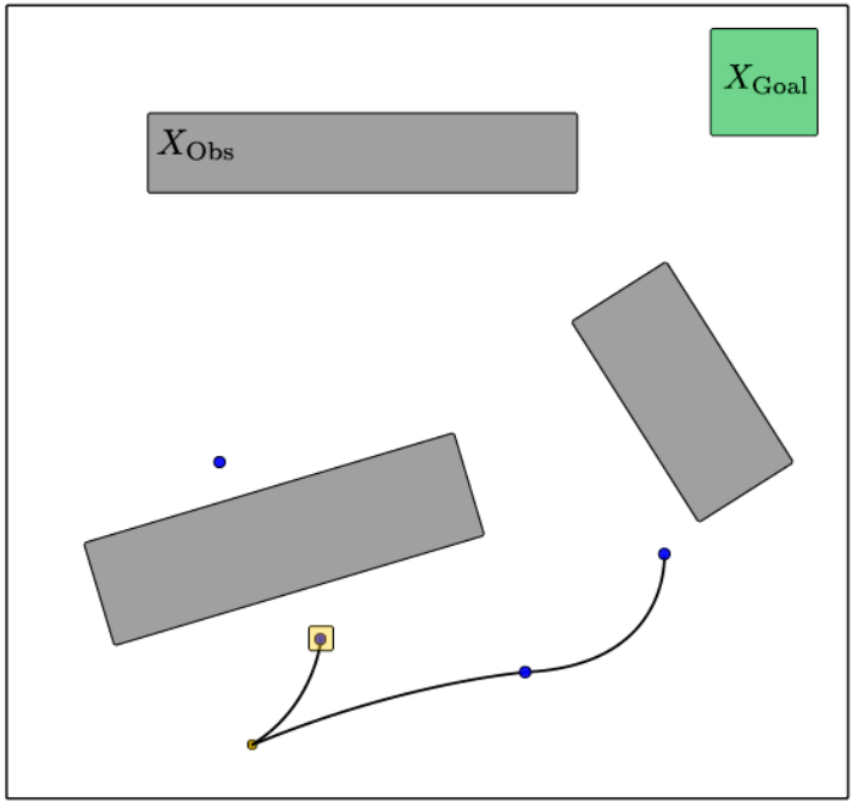
- Connections attempted to the nearest neighbor only.

Rapidly-exploring Random Tree (RRT) Algorithm: Step 11



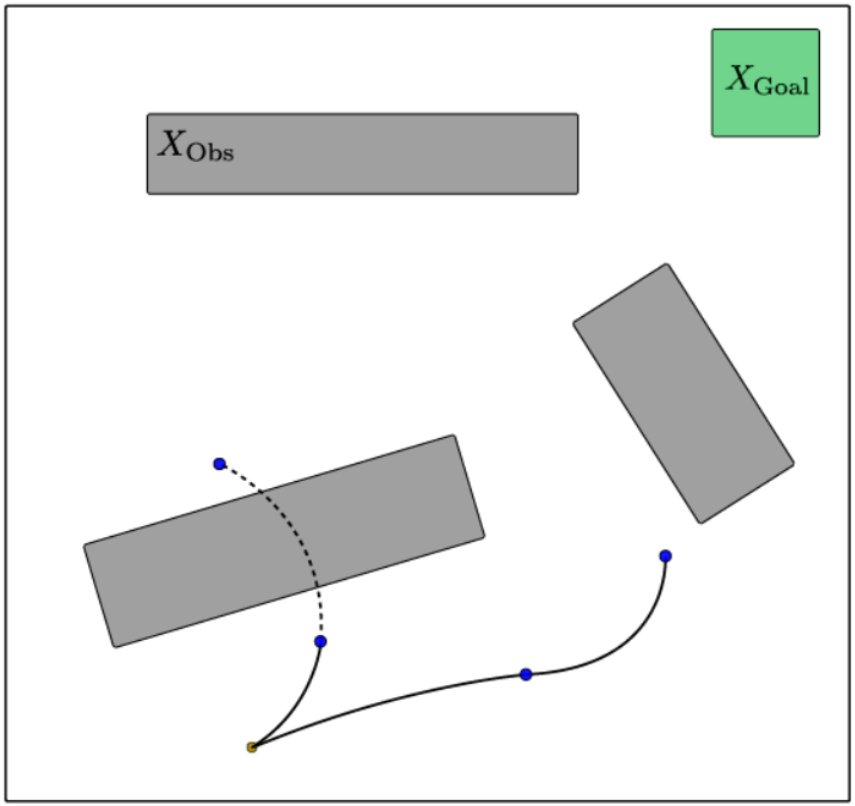
- Connections attempted to the nearest neighbor only.

Rapidly-exploring Random Tree (RRT) Algorithm: Step 12



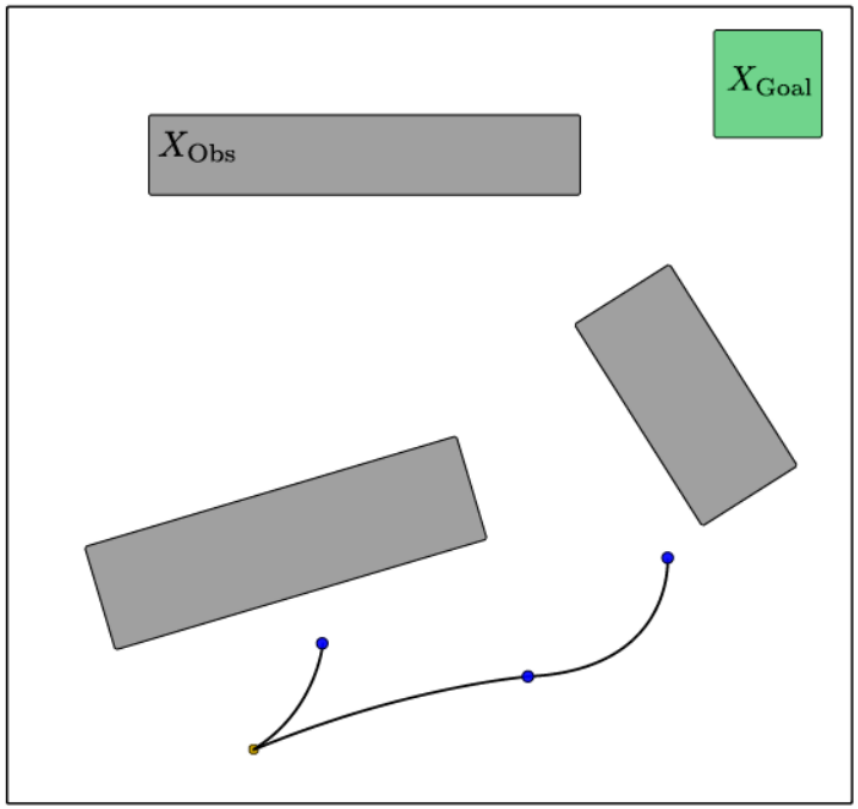
- Connections attempted to the nearest neighbor only.

Rapidly-exploring Random Tree (RRT) Algorithm: Step 13



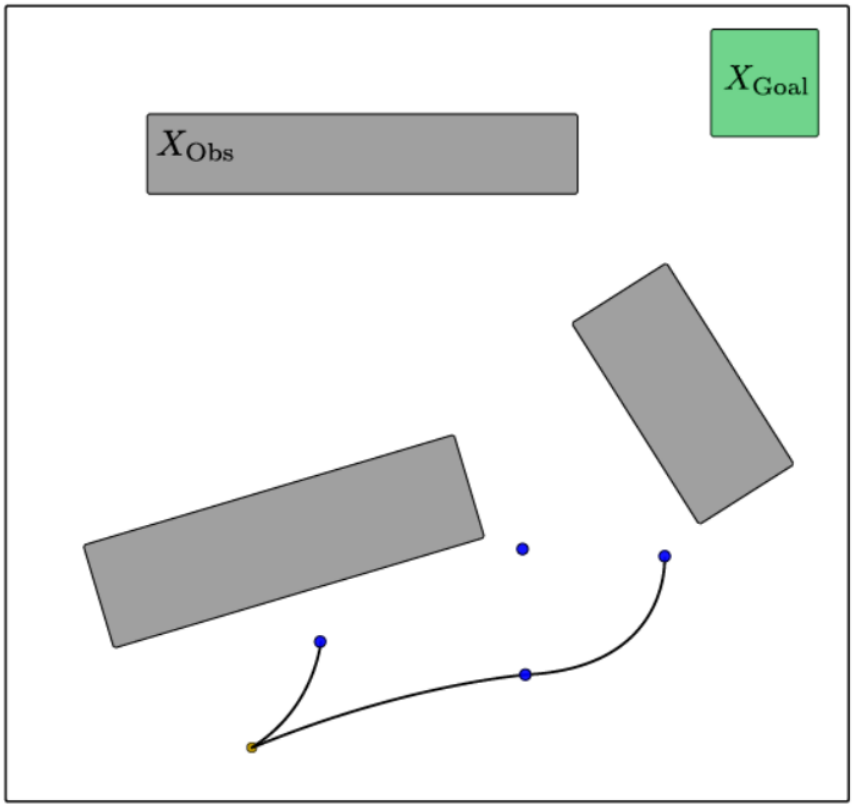
- Connections attempted to the nearest neighbor only.

Rapidly-exploring Random Tree (RRT) Algorithm: Step 14



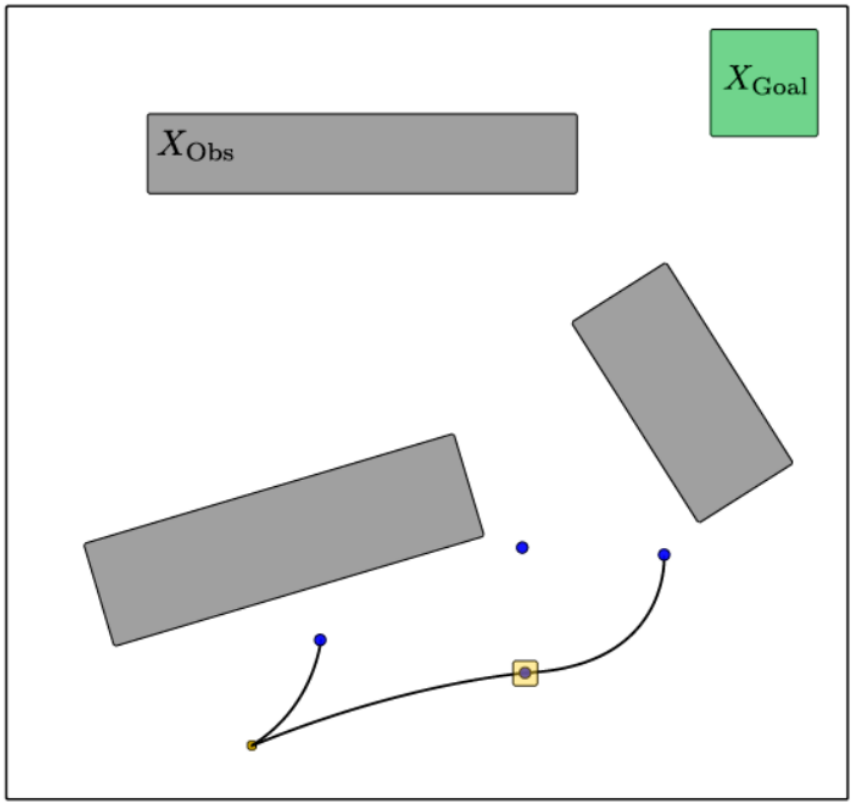
- Connections attempted to the nearest neighbor only.

Rapidly-exploring Random Tree (RRT) Algorithm: Step 15



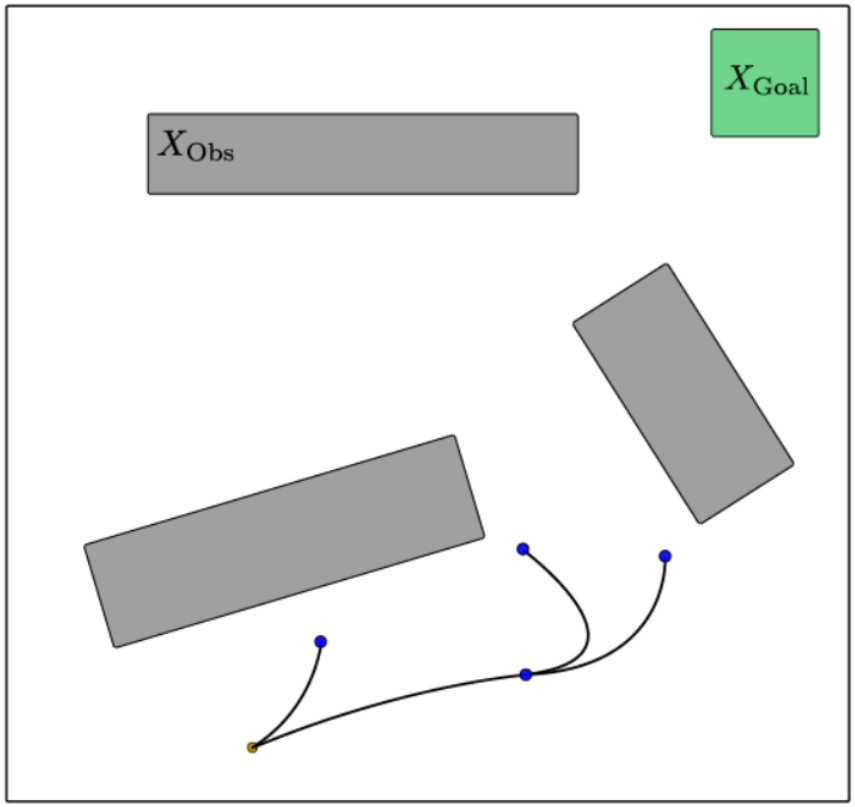
- Connections attempted to the nearest neighbor only.

Rapidly-exploring Random Tree (RRT) Algorithm: Step 16



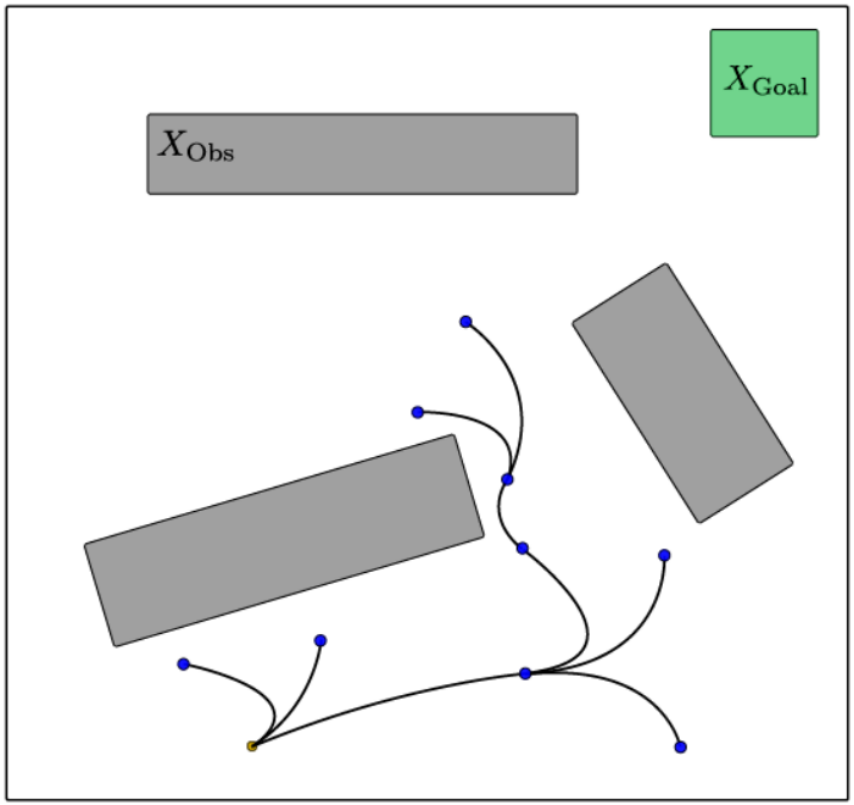
- Connections attempted to the nearest neighbor only.

Rapidly-exploring Random Tree (RRT) Algorithm: Step 17



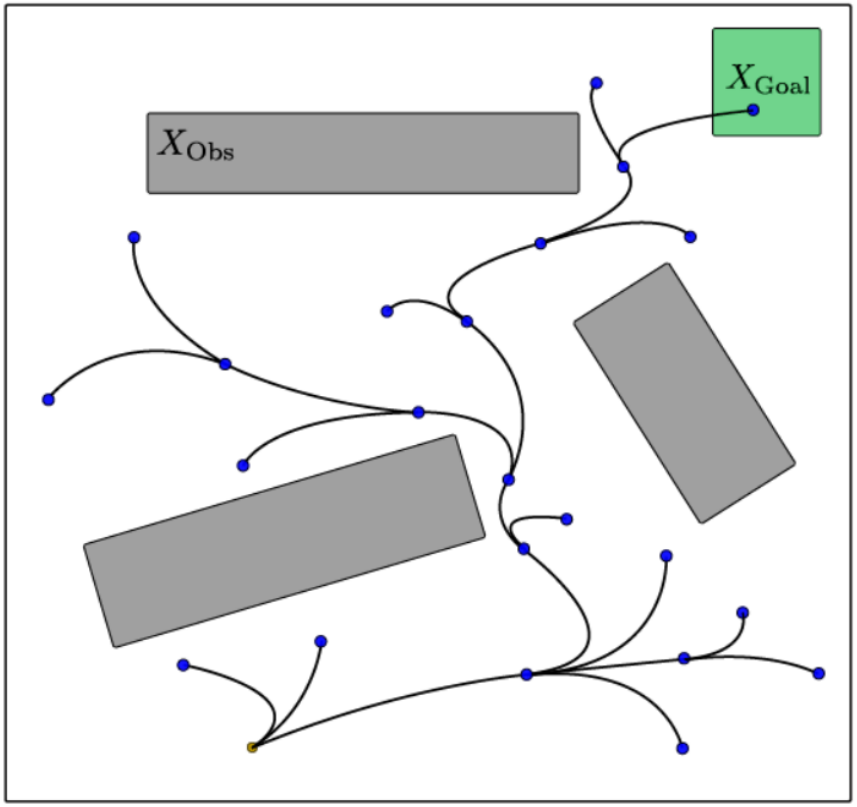
- Connections attempted to the nearest neighbor only.

Rapidly-exploring Random Tree (RRT) Algorithm: Step 18



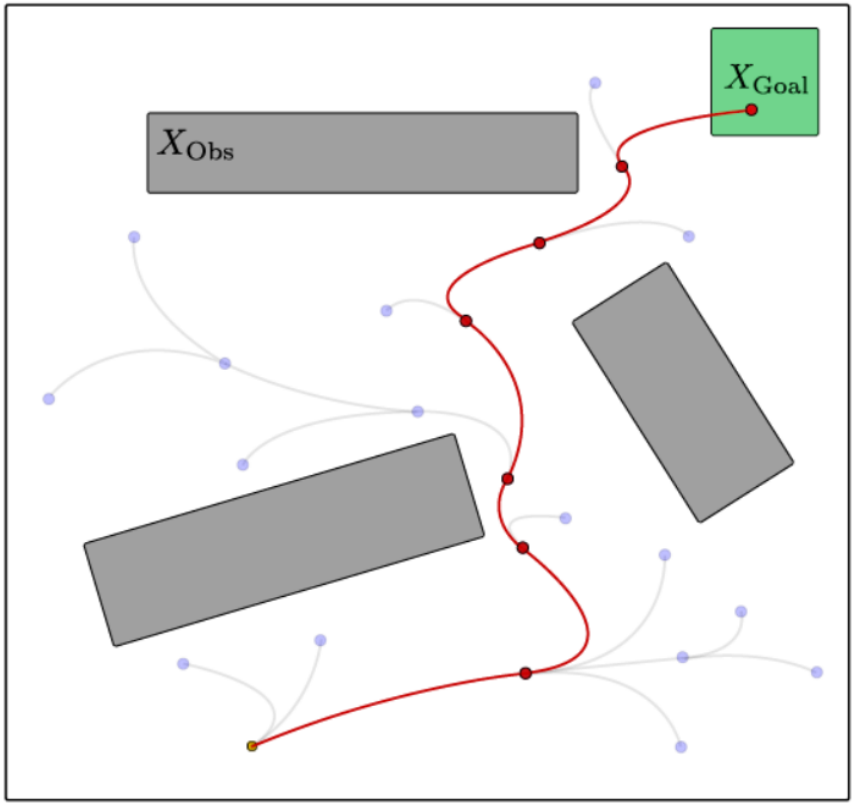
- Connections attempted to the nearest neighbor only.

Rapidly-exploring Random Tree (RRT) Algorithm: Step 19



- Connections attempted to the nearest neighbor only.

Rapidly-exploring Random Tree (RRT) Algorithm: Step 20



- Connections attempted to the nearest neighbor only.

Under The Hood, Part I: The Components

- Necessary components to implement PRM/RRT and similar algorithms:
  - A generator of dense point sequences on $\mathcal{X}$.
  - A measure of distance between two points on $\mathcal{X}$.
  - An algorithm to find nearest neighbors in a point set.
  - A "collision" checker.
  - A "local planner". [Steering function, ignoring obstacles].
  - A shortest path algorithm on graphs [Dijkstra, A*].

Distance Functions and Metrics

- Conditions for attempting the connecting between samples often rely on a notion of "distance" between points. A standard notion of "distance" is that of metric.
- A metric ρ on a space $\mathcal{X}$ is a function $\rho : \mathcal{X} \times \mathcal{X} \to \mathbb{R}_{\geq 0}, (a, b) \to \rho(a, b)$ that satisfies the following properties:
  - $\rho(a, b) = 0$ if and only if $a = b$.
  - $\rho(a, b) \geq 0$.
  - Symmetry: $\rho(a, b) = \rho(b, a)$, for any $a, b \in \mathcal{X}$.
  - Triangle inequality: $\rho(a, b) + \rho(b, c) \geq \rho(a, c)$, for any $a, b, c \in \mathcal{X}$.
- A space $\mathcal{X}$ with a metric is called a metric space.

L_p Metrics

- A commonly used family of metrics on $\mathbb{R}^n$ is given by the L_p metrics,

$$\rho(x, x') = \Big(\sum_{i=1}^{n} |x_i - x_i'|^p\Big)^{1/p}.$$

- Standard choice include:
  - Manhattan Metric L_1: in the plane, this reminds of the distance traveled by a car driving along axis-aligned city blocks.
  - Euclidean Metric L_2: this is the usual notion of "distance" in $\mathbb{R}^n$.
  - Infinity Metric L_∞: $\rho(x, x') = \max_{i=1}^{n}\{|x_i - x_i'|\}$.

More General Distance Functions

- While metrics are a standard notion of functions, there are other functions that may be more appropriate for motion planning.
- For example, in many cases of interest "distance" functions in robotics are not symmetric (these are some times called "quasimetrics"):
  - The "distance" between two points on a mountain will feel different between moving uphill vs. downhill! Same for the distance between two points in a wind field/river current/etc.
  - The driving distance between two points in Manhattan may depend on directional constraints like one-way streets.
  - The length of the shortest path between two points is not symmetric for a Dubins vehicle (which can only move forwards); it is symmetric for a Reeds-Shepp vehicle (which can also reverse).

Distance Functions for Sampling-based Motion Planning

- It turns out that the main requirement for a "distance function" to be useful for sampling-based motion planning is one of monotonicity along paths.
- This is an assumption that is typically satisfied by functions measuring some "effort" to move between two states (with or without obstacles), e.g.,
  - optimal cost-to-go functions
  - navigation functions, etc.
- Choosing a distance function appropriately is particularly important when dealing with systems with differential constraints.

Dense Sequences

- A key assumption is that the sequence of sampled points be dense in $\mathcal{X}$.
- A point sequence s on $\mathcal{X}$ is a mapping $s : \mathbb{N} \to \mathcal{X}, i \to s_i$. For any $n \in \mathbb{N}$, it generates a point set $\mathcal{S}_n = \{s_i :, i = 1, ..., n\}$.
- The sequence s is dense on $\mathcal{X}$ if, for any open set $\mathcal{O} \subset \mathcal{X}$, there exists some $\hat{n}$ such that $\mathcal{S}_n$ contains at least one point in $\mathcal{O}$, for all $n > \hat{n}$.
- In the case in which the sequence is stochastic (i.e., each s_i is a random variable), then the sequence s is dense on $\mathcal{X}$ if for any open set $\mathcal{O}$,

$$\lim_{n \to \infty} Pr[\mathcal{O} \cap \mathcal{S}_n = \emptyset] = 0.$$

- Another way to say that s is dense on $\mathcal{X}$ is that $cl(\mathcal{S}_\infty) = \mathcal{X}$.

Uniform Random Numbers on $[0, 1)$

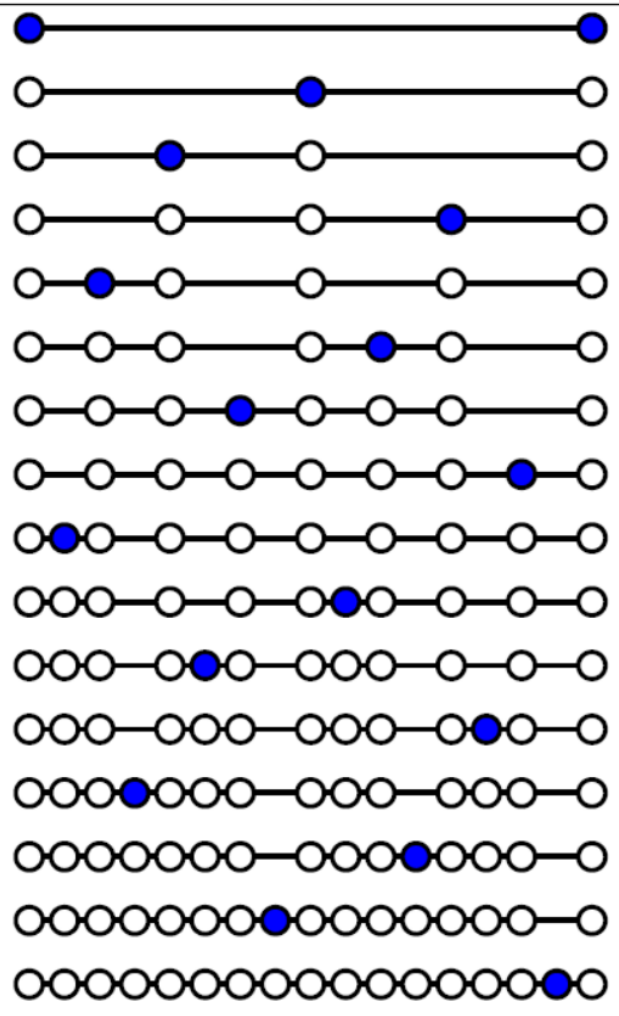
- The sequence obtained by setting $s_i = rand(1)$, i.e., a sequence of (pseudo-)random number uniformly and independently sampled from $[0, 1)$ (as in Matlab), is dense on $[0, 1]$.
- To see this, take any open interval $\mathcal{O} = (a, b) \subset [0, 1)$, and let $\epsilon = b - a$. The probability that s_i is not in $\mathcal{O}$ is $(1 - \epsilon)$. The probability that none of the n points in $\mathcal{S}_n$ is in $\mathcal{O}$ is $(1 - \epsilon)^n$, which goes to 0 as $n \to \infty$.
- The same argument extends to higher dimensions.
- Note: In sampling-based algorithms, "randomness" is most often just a convenient way to get dense sequences. Some beautiful/elegant theoretical results are also available to prove certain facts. But sample sequences do not typically need to be "random"!

The van der Corput Sequence

- We want to generate a sequence of, 16 points that is "as dense as possible" on $\mathcal{X} = [0, 1] / \sim$(i.e., the interval $[0, 1]$ where we identify 0 and 1).
- An approach to "spread" points on $\mathcal{X}$ would be to recursively bisect (one of) the largest "empty" intervals.
- There is a very simple way to do that:
  - Write the numbers $0, 1/16, ..., 15/16$ in binary notation.
  - Flip the binary figures left/right.
  - The sequence thus obtained achieves the desired objective!
- The same sequence can be extended to any desired number of samples!

The van der Corput Sequence

i	Naive Sequence	Binary	Reverse Binary	Van der Corput	Points in $[0,1]/\sim$
1	0	.0000	.0000	0	
2	1/16	.0001	.1000	1/2	
3	1/8	.0010	.0100	1/4	
4	3/16	.0011	.1100	3/4	
5	1/4	.0100	.0010	1/8	
6	5/16	.0101	.1010	5/8	
7	3/8	.0110	.0110	3/8	
8	7/16	.0111	.1110	7/8	
9	1/2	.1000	.0001	1/16	
10	9/16	.1001	.1001	9/16	
11	5/8	.1010	.0101	5/16	
12	11/16	.1011	.1101	13/16	
13	3/4	.1100	.0011	3/16	
14	13/16	.1101	.1011	11/16	
15	7/8	.1110	.0111	7/16	
16	15/16	.1111	.1111	15/16	

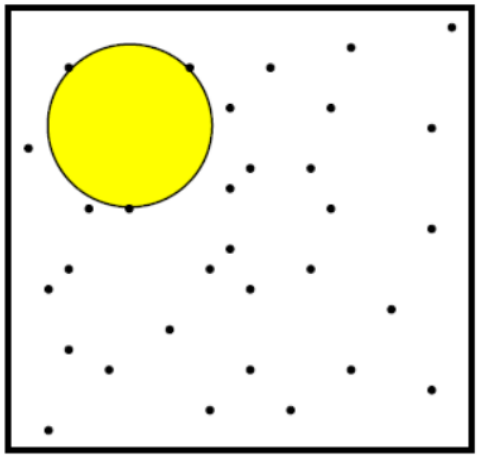


Low-Dispersion Sampling

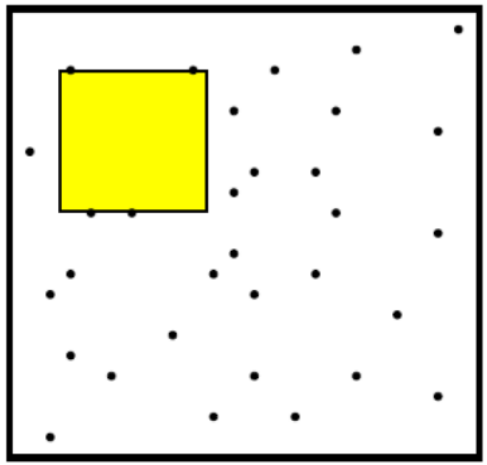
- How do we extend the notion of "uniformity" to deterministic sequences?
- Idea: measure the biggest empty "ball" in $\mathcal{X}$ and try to make it as small as possible.
- The dispersion δ of a pointset $\mathcal{P} \in \mathcal{X}$ induced by a distance function ρ is defined as

$$\delta_\rho(\mathcal{P}) = \sup_{x \in \mathcal{X}} \left\{ \min_{p \in \mathcal{P}} \{\rho(x, p)\} \right\}.$$

- Lower dispersion means that the points are nicely dispersed. Thus, more dispersion is bad, which is counterintuitive.
- Reducing the dispersion means reducing the radius of the largest empty ball.



(a) L_2 dispersion



(b) L_∞ dispersion

The Halton Sequence

- The Halton sequence can be seen as a generalization of the van der Corput sequence, and is useful to generate low discrepancy sequences in higher dimensions.
- Let $d_k(n), k \in \mathbb{N}$ be the coefficients of the expression of i in base $b \in \mathbb{N}$, i.e.,

$$i = \sum_{k=0}^{\infty} d_k(n) b^k.$$

- The generalized van der Corput sequence is obtained by reversing the order of the digits, and moving the decimal point, i.e.,

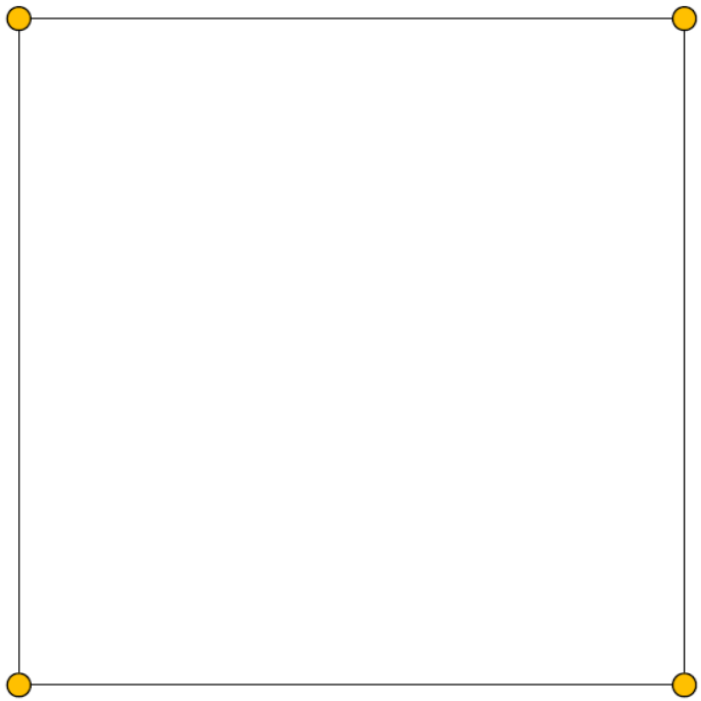
$$s_i^{(b)} = \sum_{k=0}^{\infty} d_k(n) b^{-k-1}.$$

- The Halton sequence on $[0,1]^d$ is obtained by choosing d co-prime numbers (e.g., the first d primes) as bases, and setting

$$h_i = (s_i^{(b_1)}, s_i^{(b_2)}, ..., s_i^{(b_d)}).$$

Halton Sequence Example: Step 0

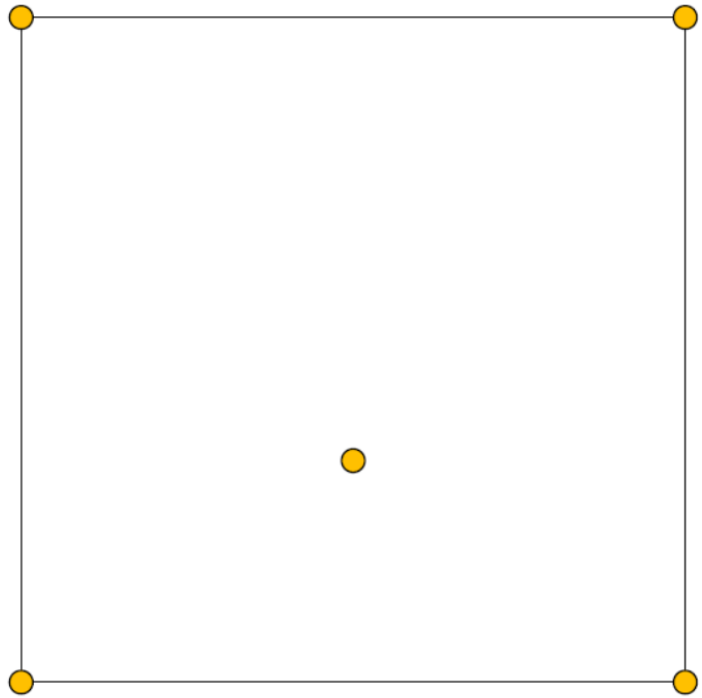
i	b=2	$s_i^{(2)}, b = 2$	b=3	$s_i^{(3)}, b = 3$
0	000	.000	000	.000
1	001	.100	001	.100
2	010	.010	002	.200
3	011	.110	010	.010
4	100	.001	011	.110
5	101	.101	012	.210
6	110	.011	020	.020
7	111	.111	021	.120
8	1000	.0001	022	.220
9	1001	.1001	100	.001
...	...	...	...	...



Halton Sequence Example: Step 1

$$(\frac{1}{2}, \frac{1}{3})$$

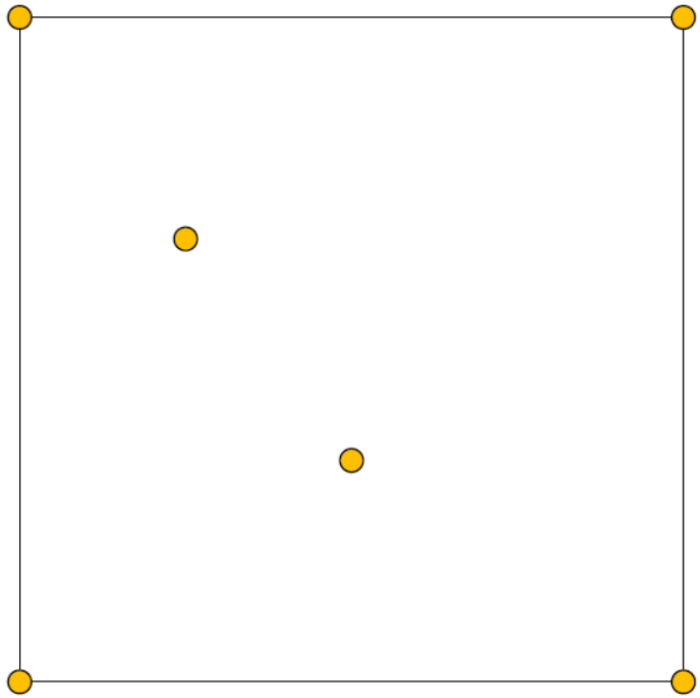
i	b=2	$s_i^{(2)}, b = 2$	b=3	$s_i^{(3)}, b = 3$
0	000	.000	000	.000
1	001	.100	001	.100
2	010	.010	002	.200
3	011	.110	010	.010
4	100	.001	011	.110
5	101	.101	012	.210
6	110	.011	020	.020
7	111	.111	021	.120
8	1000	.0001	022	.220
9	1001	.1001	100	.001
...	...	...	...	...



Halton Sequence Example: Step 2

$$(\frac{1}{4}, \frac{2}{3})$$

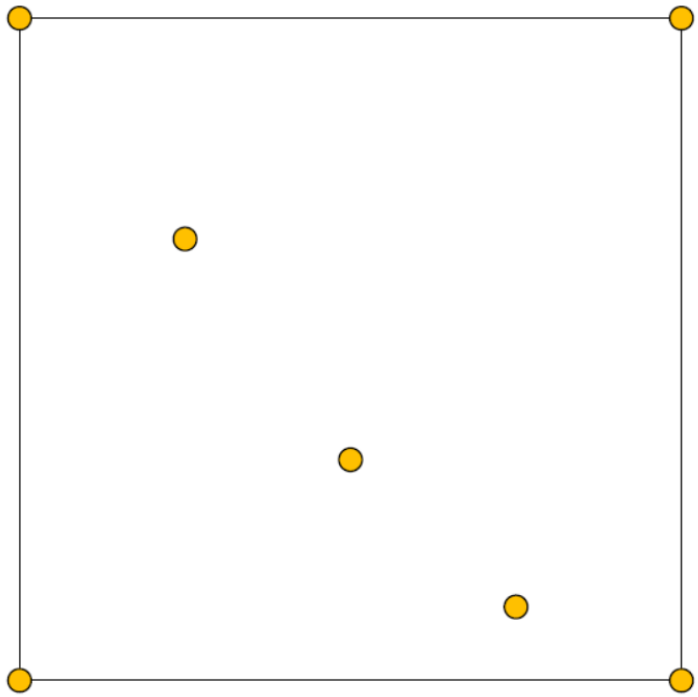
i	b=2	$s_i^{(2)}, b = 2$	b=3	$s_i^{(3)}, b = 3$
0	000	.000	000	.000
1	001	.100	001	.100
2	010	.010	002	.200
3	011	.110	010	.010
4	100	.001	011	.110
5	101	.101	012	.210
6	110	.011	020	.020
7	111	.111	021	.120
8	1000	.0001	022	.220
9	1001	.1001	100	.001
...	...	...	...	...



Halton Sequence Example: Step 3

$$\left(\frac{3}{4}, \frac{1}{9}\right)$$

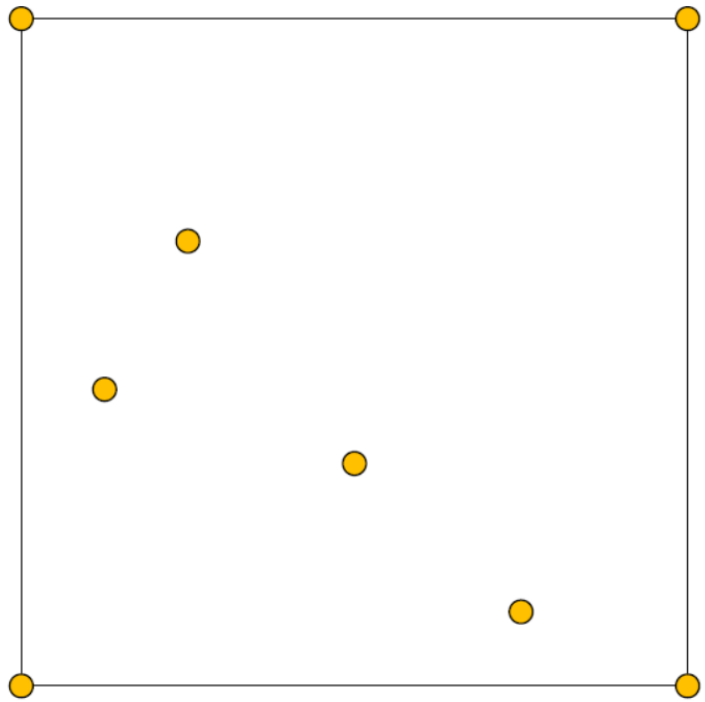
i	b=2	$s_i^{(2)}, b=2$	b=3	$s_i^{(3)}, b=3$
0	000	.000	000	.000
1	001	.100	001	.100
2	010	.010	002	.200
3	011	.110	010	.010
4	100	.001	011	.110
5	101	.101	012	.210
6	110	.011	020	.020
7	111	.111	021	.120
8	1000	.0001	022	.220
9	1001	.1001	100	.001
...	...	...	...	...



Halton Sequence Example: Step 4

$$(\frac{1}{8}, \frac{4}{9})$$

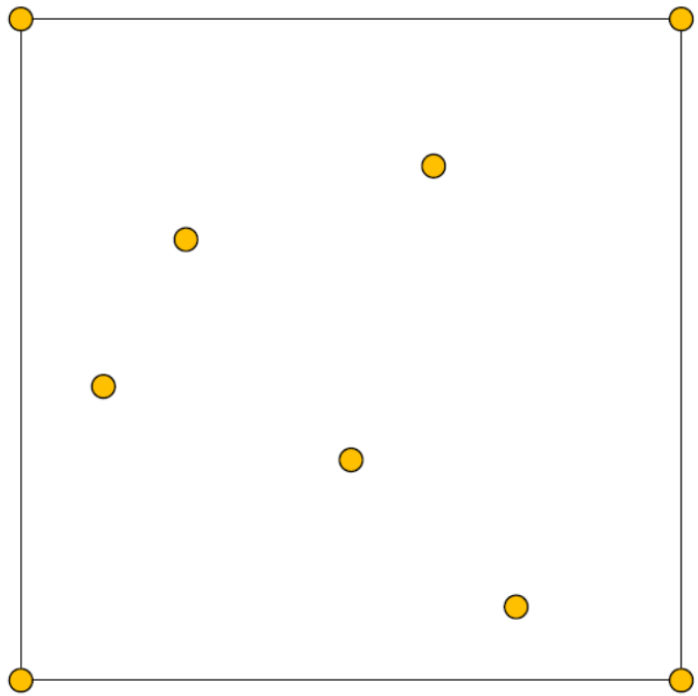
i	b=2	$s_i^{(2)}, b=2$	b=3	$s_i^{(3)}, b=3$
0	000	.000	000	.000
1	001	.100	001	.100
2	010	.010	002	.200
3	011	.110	010	.010
4	100	.001	011	.110
5	101	.101	012	.210
6	110	.011	020	.020
7	111	.111	021	.120
8	1000	.0001	022	.220
9	1001	.1001	100	.001
...	...	...	...	...



Halton Sequence Example: Step 5

$$(\frac{5}{8}, \frac{7}{9})$$

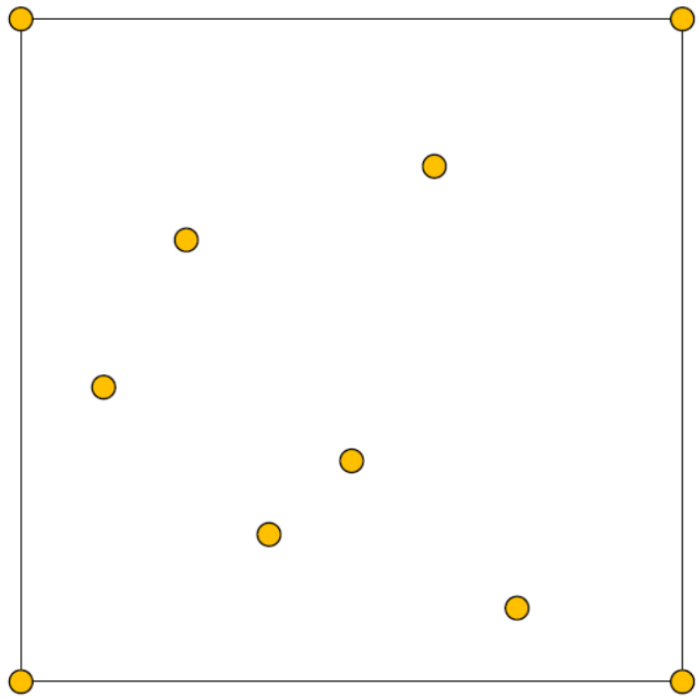
i	b=2	$s_i^{(2)}, b = 2$	b=3	$s_i^{(3)}, b = 3$
0	000	.000	000	.000
1	001	.100	001	.100
2	010	.010	002	.200
3	011	.110	010	.010
4	100	.001	011	.110
5	101	.101	012	.210
6	110	.011	020	.020
7	111	.111	021	.120
8	1000	.0001	022	.220
9	1001	.1001	100	.001
...	...	...	...	...



Halton Sequence Example: Step 6

$$(\frac{3}{8}, \frac{2}{9})$$

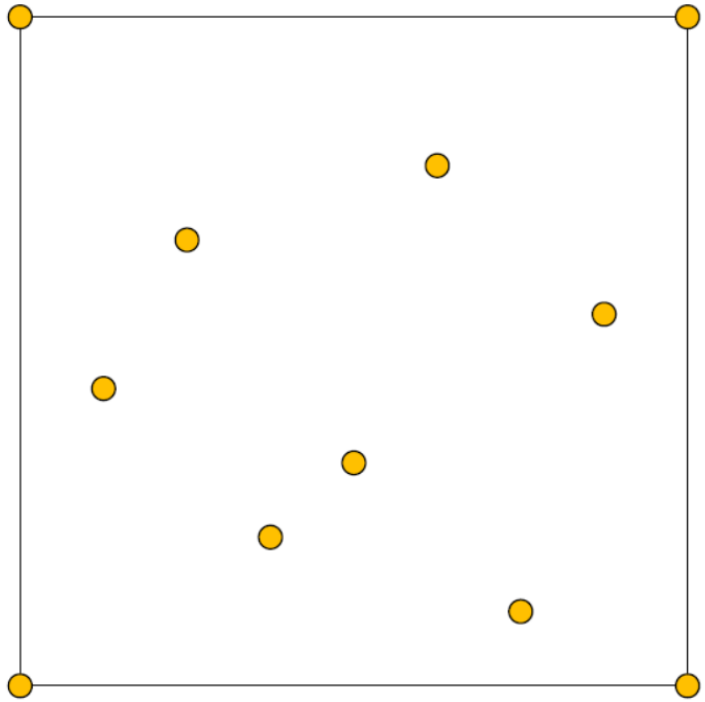
i	b=2	$s_i^{(2)}, b=2$	b=3	$s_i^{(3)}, b=3$
0	000	.000	000	.000
1	001	.100	001	.100
2	010	.010	002	.200
3	011	.110	010	.010
4	100	.001	011	.110
5	101	.101	012	.210
6	110	.011	020	.020
7	111	.111	021	.120
8	1000	.0001	022	.220
9	1001	.1001	100	.001
...	...	...	...	...



Halton Sequence Example: Step 7

$$(\frac{7}{8}, \frac{5}{9})$$

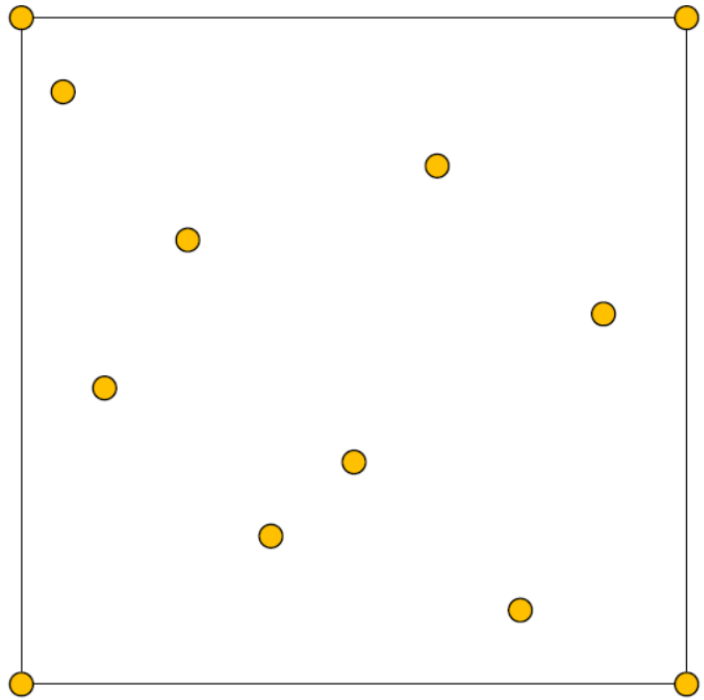
i	b=2	$s_i^{(2)}, b=2$	b=3	$s_i^{(3)}, b=3$
0	000	.000	000	.000
1	001	.100	001	.100
2	010	.010	002	.200
3	011	.110	010	.010
4	100	.001	011	.110
5	101	.101	012	.210
6	110	.011	020	.020
7	111	.111	021	.120
8	1000	.0001	022	.220
9	1001	.1001	100	.001
...	...	...	...	...



Halton Sequence Example: Step 8

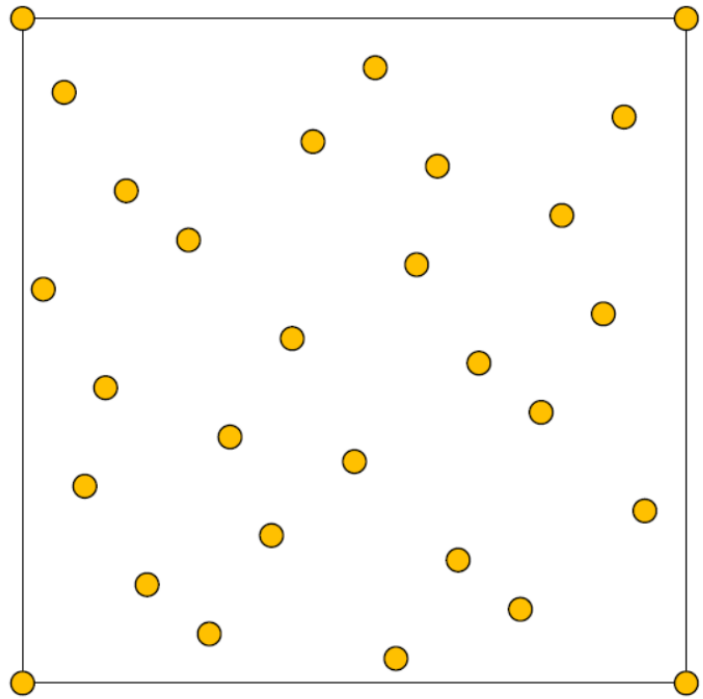
$$(\frac{1}{16}, \frac{8}{9})$$

i	b=2	$s_i^{(2)}, b=2$	b=3	$s_i^{(3)}, b=3$
0	000	.000	000	.000
1	001	.100	001	.100
2	010	.010	002	.200
3	011	.110	010	.010
4	100	.001	011	.110
5	101	.101	012	.210
6	110	.011	020	.020
7	111	.111	021	.120
8	1000	.0001	022	.220
9	1001	.1001	100	.001
...	...	...	...	...



Halton Sequence Example: Step N

i	b=2	$s_i^{(2)}, b=2$	b=3	$s_i^{(3)}, b=3$
0	000	.000	000	.000
1	001	.100	001	.100
2	010	.010	002	.200
3	011	.110	010	.010
4	100	.001	011	.110
5	101	.101	012	.210
6	110	.011	020	.020
7	111	.111	021	.120
8	1000	.0001	022	.220
9	1001	.1001	100	.001
...	...	...	...	...



Under The Hood, Part I: The Components

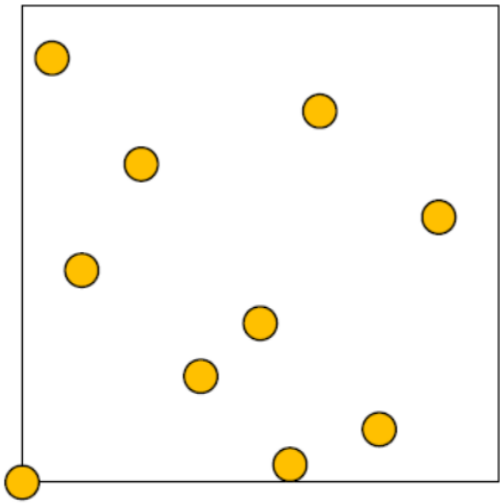
- Necessary components to implement PRM/RRT and similar algorithms:
  - A generator of dense point sequences on $\mathcal{X}$.
  - A measure of distance between two points on $\mathcal{X}$.
  - An algorithm to find nearest neighbors in a point set.
  - A "collision" checker.
  - A "local planner". [Steering function, ignoring obstacles].
  - A shortest path algorithm on graphs [Dijkstra, A*].

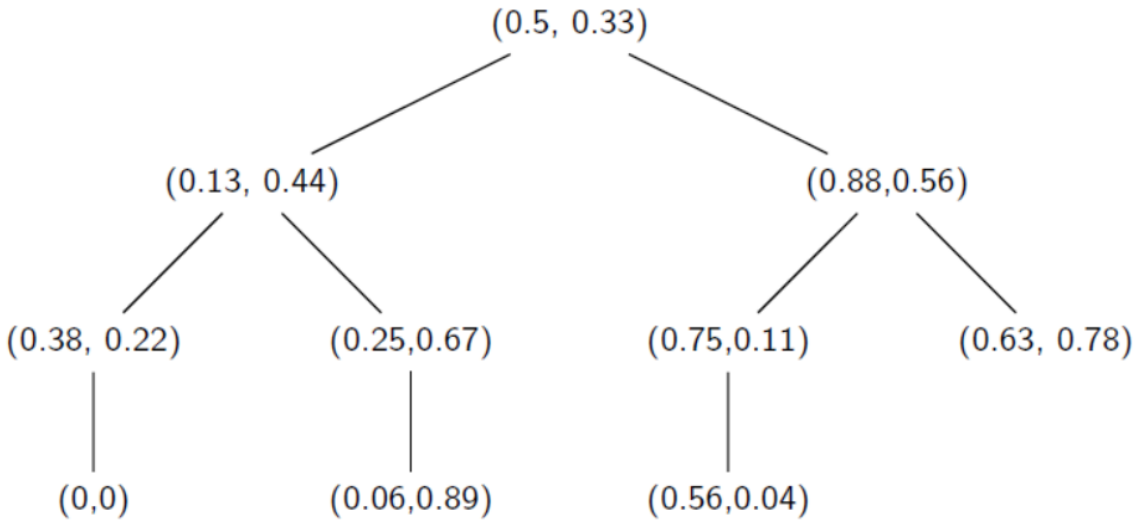
Nearest-Neighbor Search

- Sampling-based algorithms typically require looking for the nearest point (RRT), or all points within a certain radius (simple PRM).
- Computing distances to all points requires linear time in the size of the tree or roadmap - for each new point. So the complexity of building a PRM/RRT would grow as n^2. This is often too slow.
- A fast algorithm for looking for nearest neighbor is essential for good performance.
- k-d trees are a data structure that allows looking for nearest neighbors in log n time...

k-d tree Construction (batch): Step 1

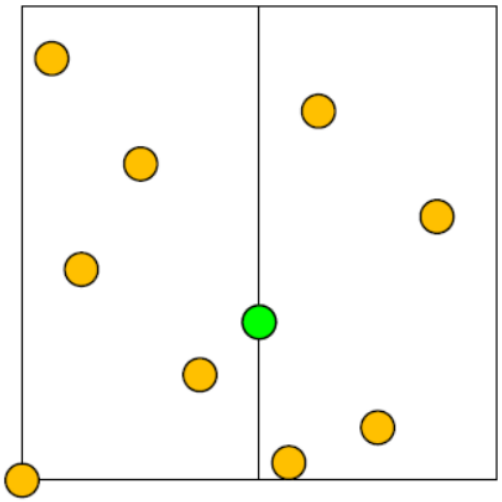
- Each node of the tree is associated to a point in $\mathbb{R}^d$, and a hyperplane splitting the remaining points into two groups.
- Points are recursively split into two, along the direction of the axes, cycling at each level of the tree.
- In order to keep the tree balanced, pick the median point in the split direction.

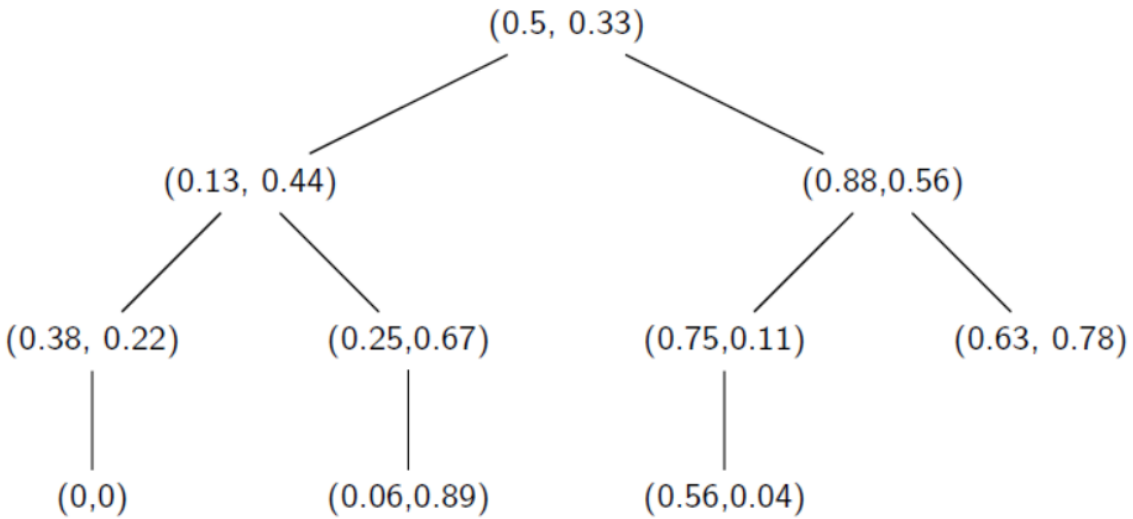




k-d tree Construction (batch): Step 2

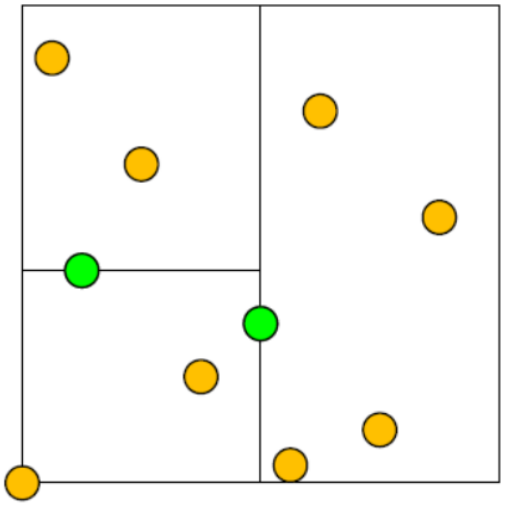
- Each node of the tree is associated to a point in $\mathbb{R}^d$, and a hyperplane splitting the remaining points into two groups.
- Points are recursively split into two, along the direction of the axes, cycling at each level of the tree.
- In order to keep the tree balanced, pick the median point in the split direction.

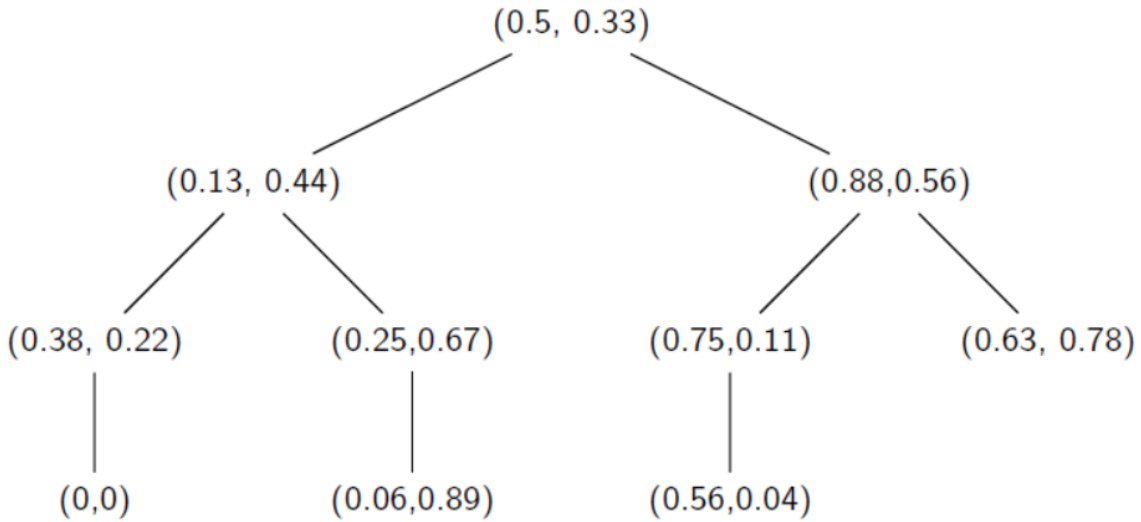




k-d tree Construction (batch): Step 3

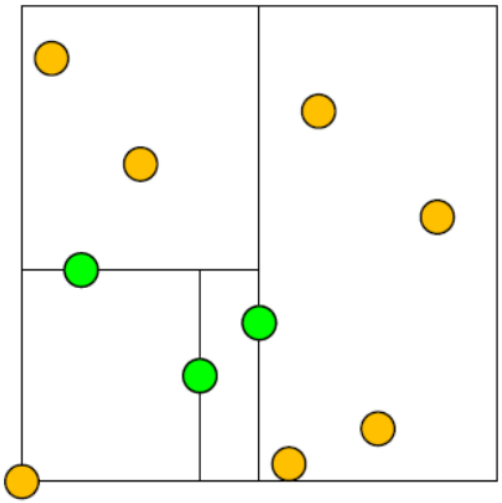
- Each node of the tree is associated to a point in $\mathbb{R}^d$, and a hyperplane splitting the remaining points into two groups.
- Points are recursively split into two, along the direction of the axes, cycling at each level of the tree.
- In order to keep the tree balanced, pick the median point in the split direction.

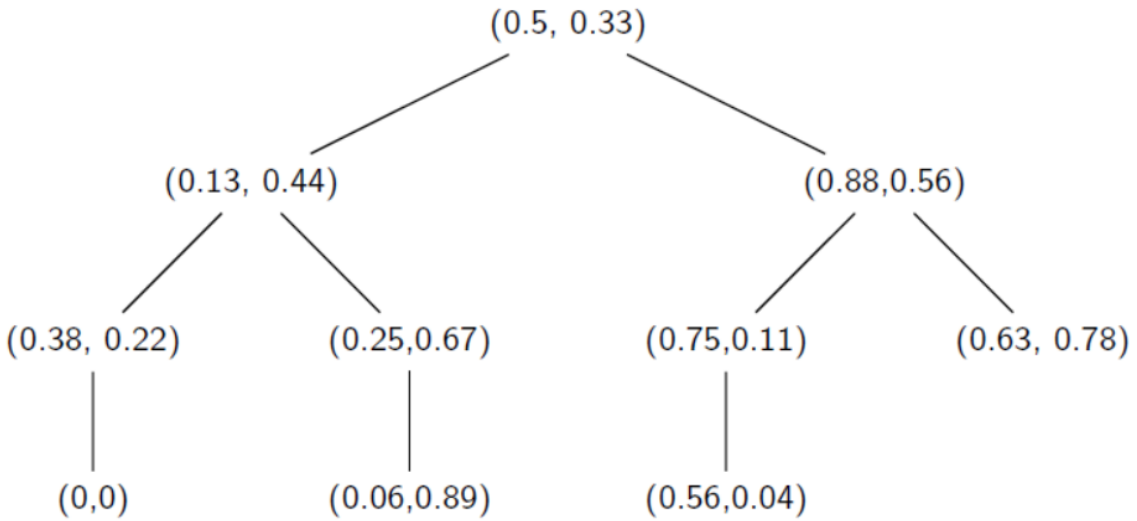




k-d tree Construction (batch): Step 4

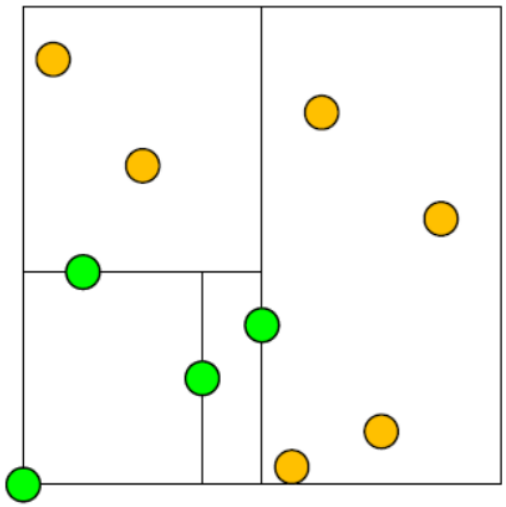
- Each node of the tree is associated to a point in $\mathbb{R}^d$, and a hyperplane splitting the remaining points into two groups.
- Points are recursively split into two, along the direction of the axes, cycling at each level of the tree.
- In order to keep the tree balanced, pick the median point in the split direction.

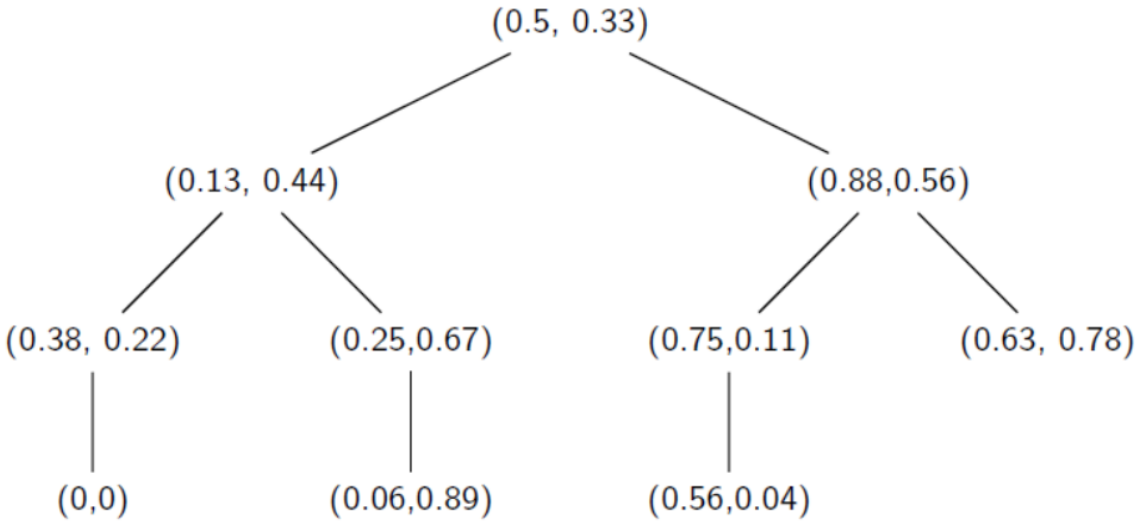




k-d tree Construction (batch): Step 5

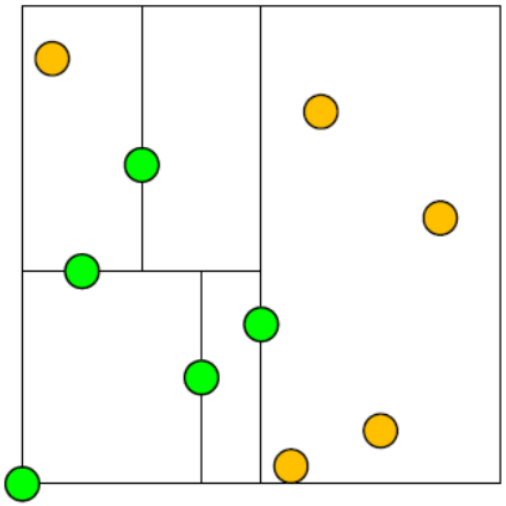
- Each node of the tree is associated to a point in $\mathbb{R}^d$, and a hyperplane splitting the remaining points into two groups.
- Points are recursively split into two, along the direction of the axes, cycling at each level of the tree.
- In order to keep the tree balanced, pick the median point in the split direction.

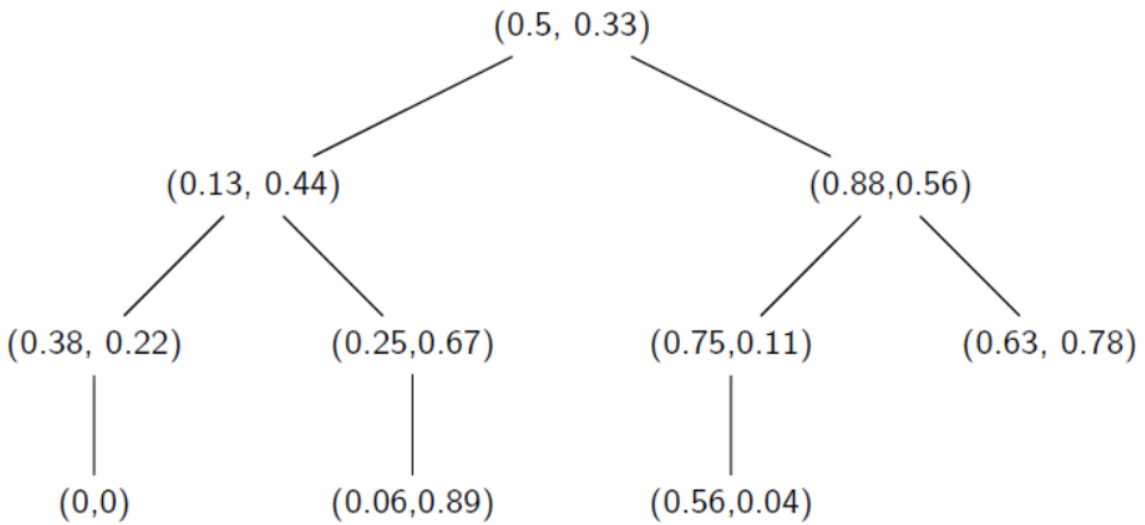




k-d tree Construction (batch): Step 6

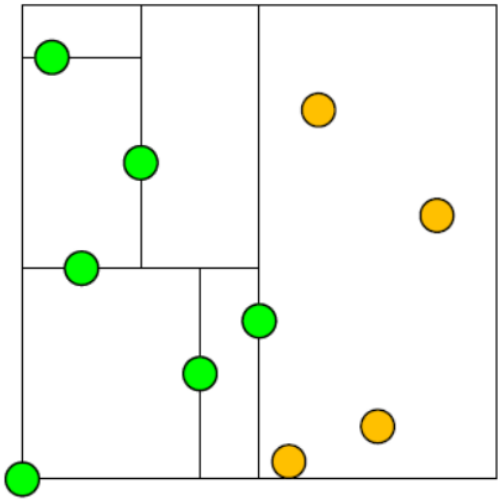
- Each node of the tree is associated to a point in $\mathbb{R}^d$, and a hyperplane splitting the remaining points into two groups.
- Points are recursively split into two, along the direction of the axes, cycling at each level of the tree.
- In order to keep the tree balanced, pick the median point in the split direction.

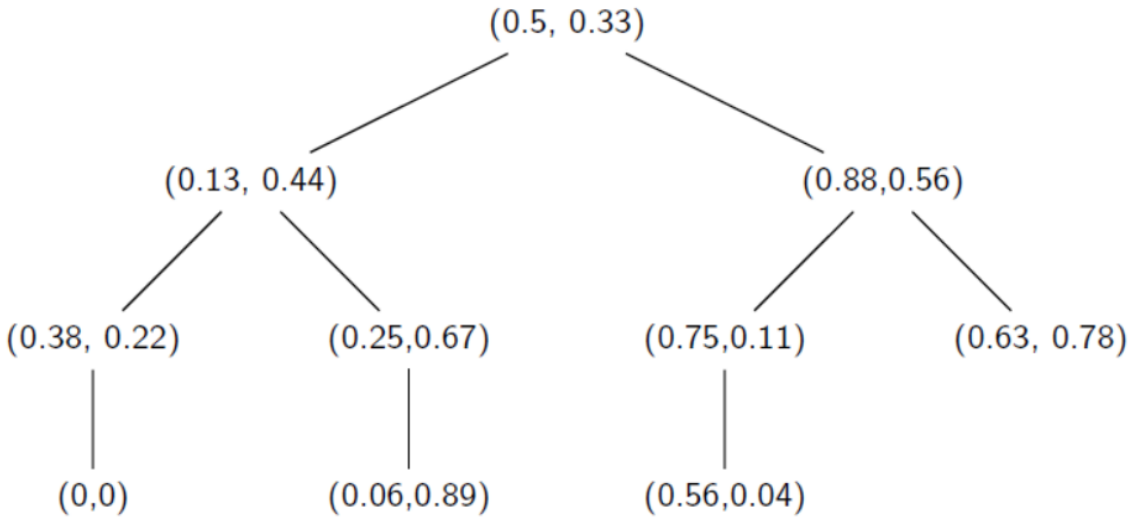




k-d tree Construction (batch): Step 7

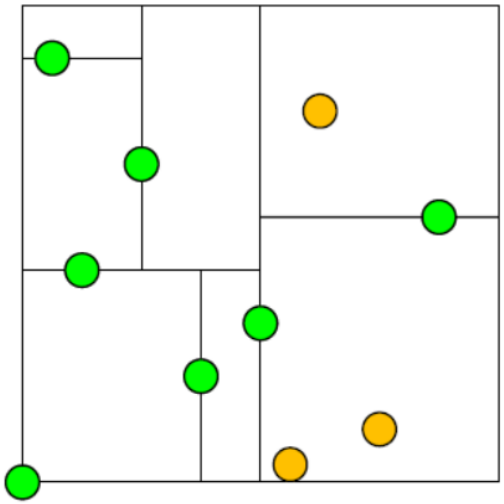
- Each node of the tree is associated to a point in $\mathbb{R}^d$, and a hyperplane splitting the remaining points into two groups.
- Points are recursively split into two, along the direction of the axes, cycling at each level of the tree.
- In order to keep the tree balanced, pick the median point in the split direction.

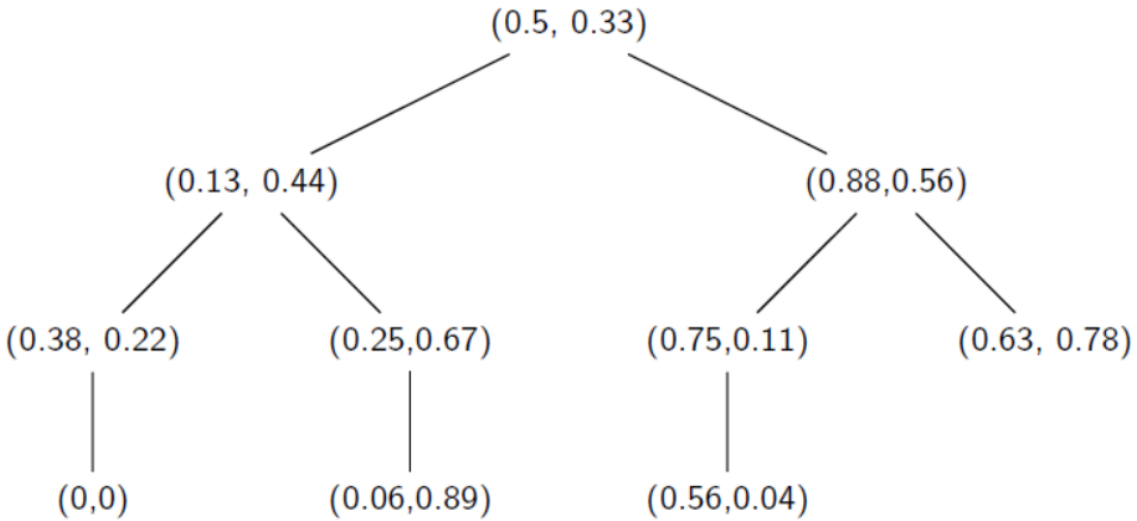




k-d tree Construction (batch): Step 8

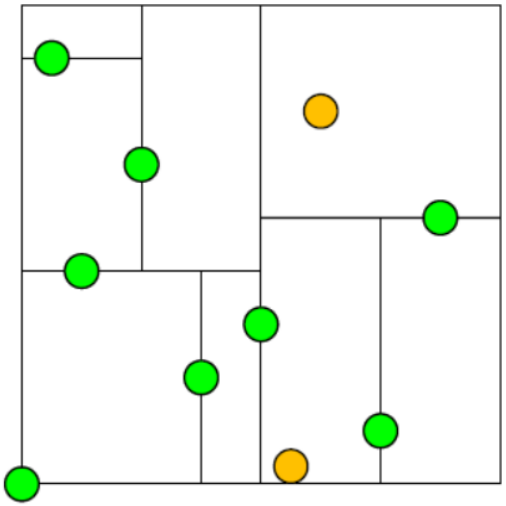
- Each node of the tree is associated to a point in $\mathbb{R}^d$, and a hyperplane splitting the remaining points into two groups.
- Points are recursively split into two, along the direction of the axes, cycling at each level of the tree.
- In order to keep the tree balanced, pick the median point in the split direction.

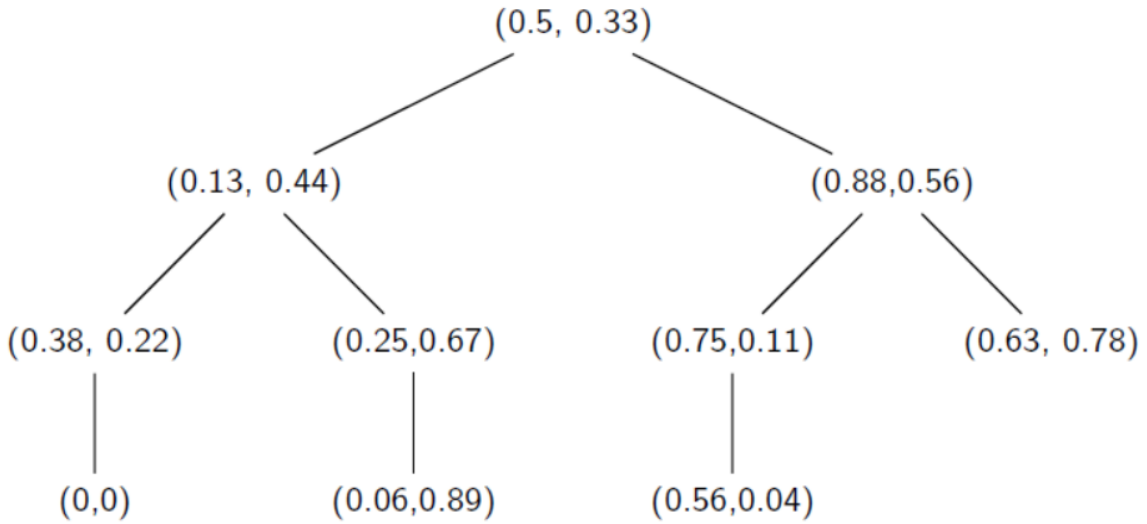




k-d tree Construction (batch): Step 9

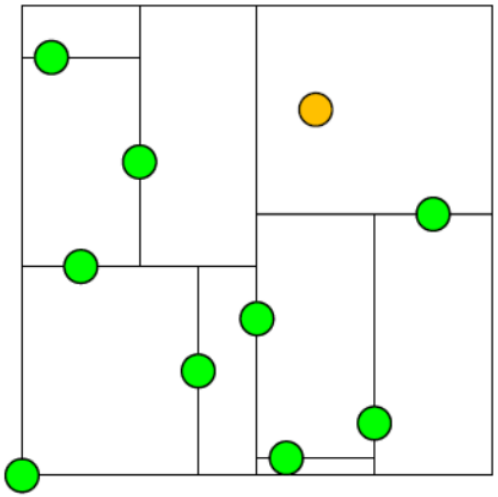
- Each node of the tree is associated to a point in $\mathbb{R}^d$, and a hyperplane splitting the remaining points into two groups.
- Points are recursively split into two, along the direction of the axes, cycling at each level of the tree.
- In order to keep the tree balanced, pick the median point in the split direction.

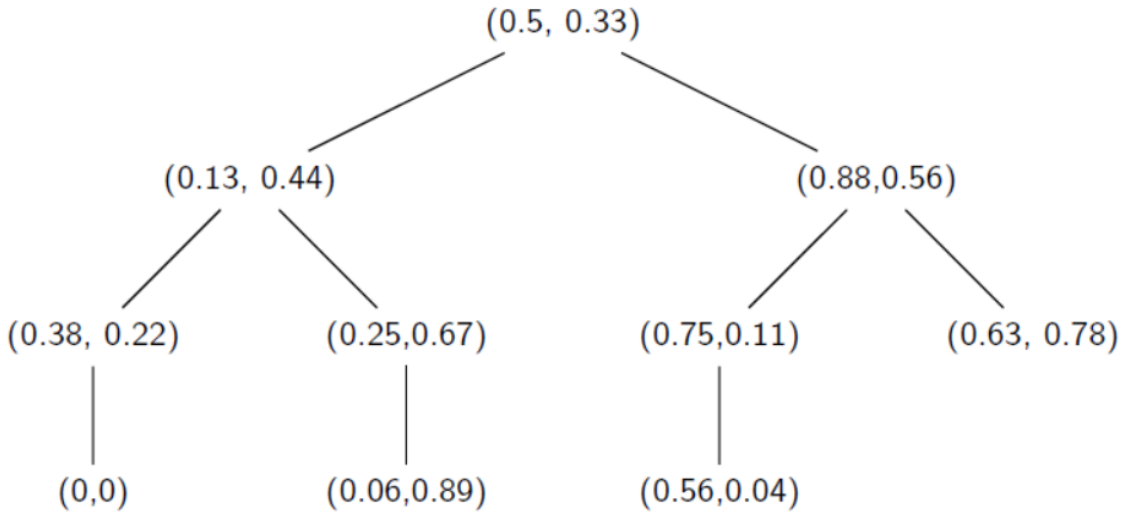




k-d tree Construction (batch): Step 10

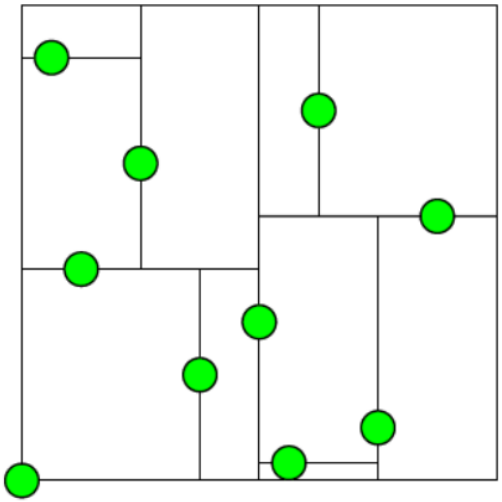
- Each node of the tree is associated to a point in $\mathbb{R}^d$, and a hyperplane splitting the remaining points into two groups.
- Points are recursively split into two, along the direction of the axes, cycling at each level of the tree.
- In order to keep the tree balanced, pick the median point in the split direction.

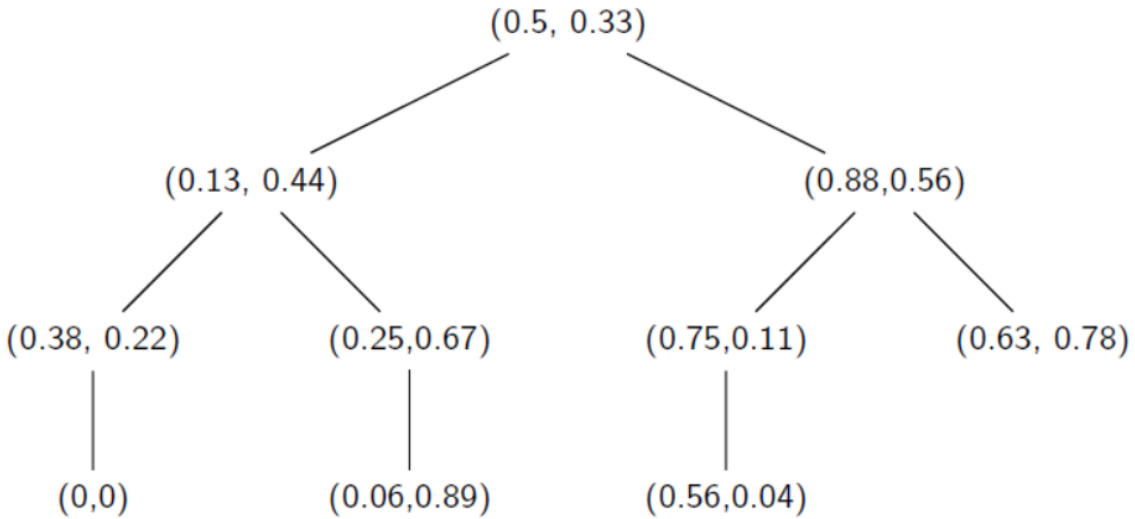




k-d tree Construction (batch): Step 11

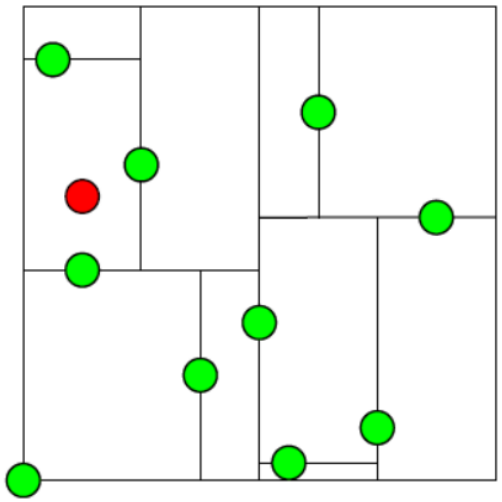
- Each node of the tree is associated to a point in $\mathbb{R}^d$, and a hyperplane splitting the remaining points into two groups.
- Points are recursively split into two, along the direction of the axes, cycling at each level of the tree.
- In order to keep the tree balanced, pick the median point in the split direction.

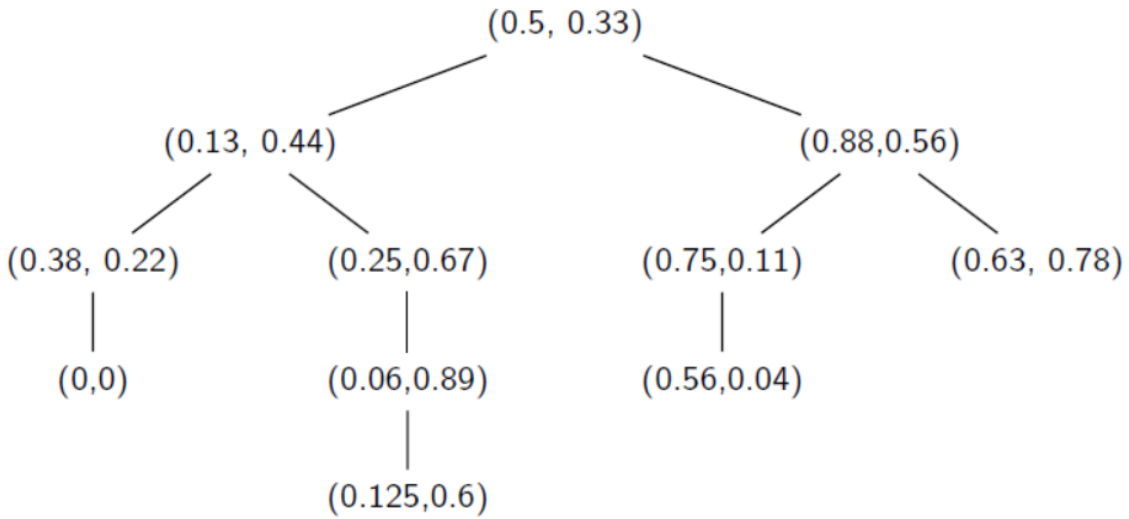




Nearest Neighbor Search on A k-d Tree

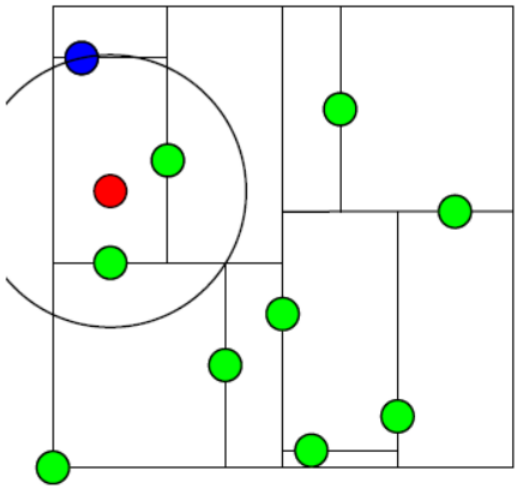
- Proceed down the tree as if x was a new point to add.
- When a leaf node is reached, the distance (squared) is computed and stored as current nearest.
- Sibling are searched if distance (squared) to the split is smaller than the current nearest.
- Otherwise the search moves up the tree, updating current nearest at parent nodes as appropriate.

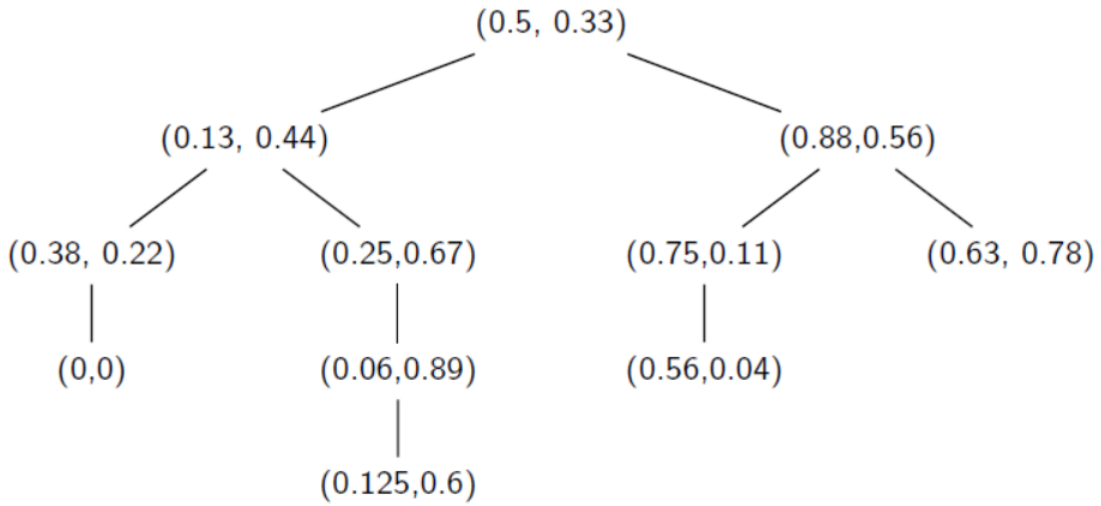




Nearest Neighbor Search on A k-d Tree

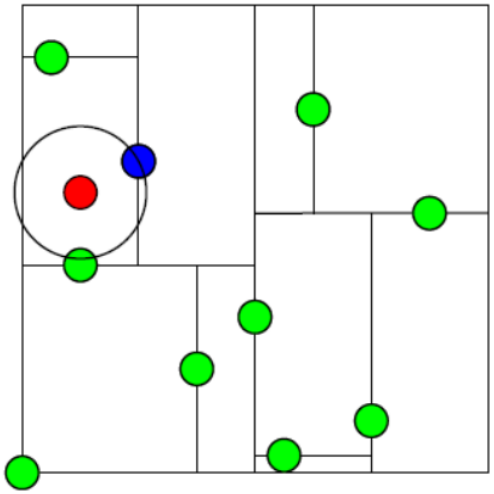
- Proceed down the tree as if x was a new point to add.
- When a leaf node is reached, the distance (squared) is computed and stored as current nearest.
- Sibling are searched if distance (squared) to the split is smaller than the current nearest.
- Otherwise the search moves up the tree, updating current nearest at parent nodes as appropriate.

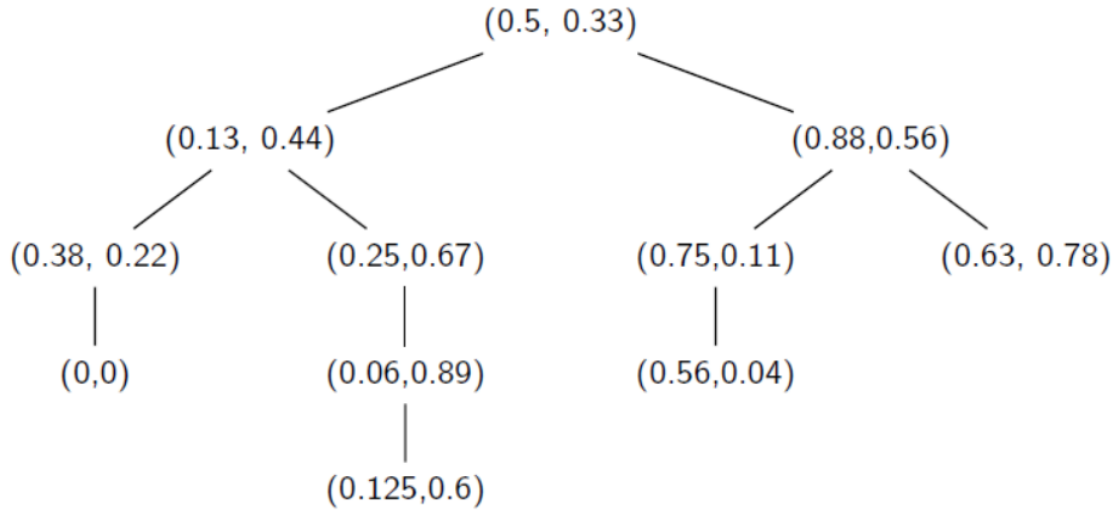




Nearest Neighbor Search on A k-d Tree

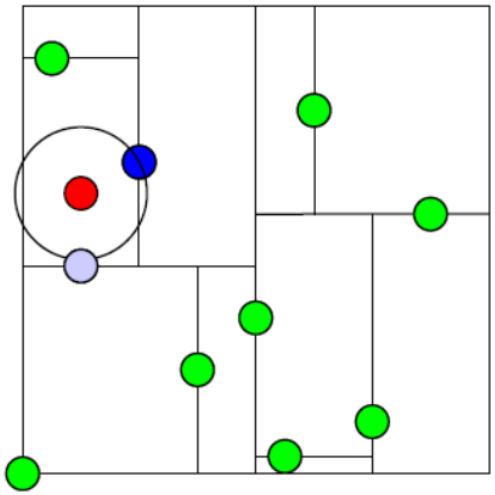
- Proceed down the tree as if x was a new point to add.
- When a leaf node is reached, the distance (squared) is computed and stored as current nearest.
- Sibling are searched if distance (squared) to the split is smaller than the current nearest.
- Otherwise the search moves up the tree, updating current nearest at parent nodes as appropriate.

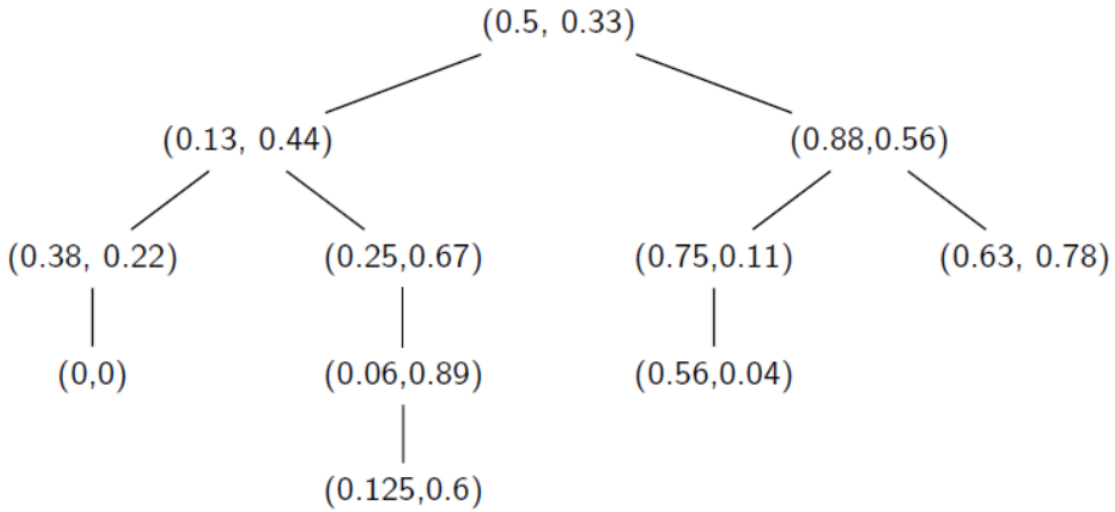




Nearest Neighbor Search on A k-d Tree

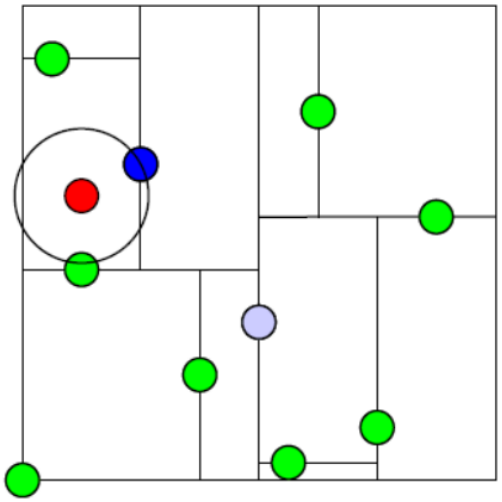
- Proceed down the tree as if x was a new point to add.
- When a leaf node is reached, the distance (squared) is computed and stored as current nearest.
- Sibling are searched if distance (squared) to the split is smaller than the current nearest.
- Otherwise the search moves up the tree, updating current nearest at parent nodes as appropriate.

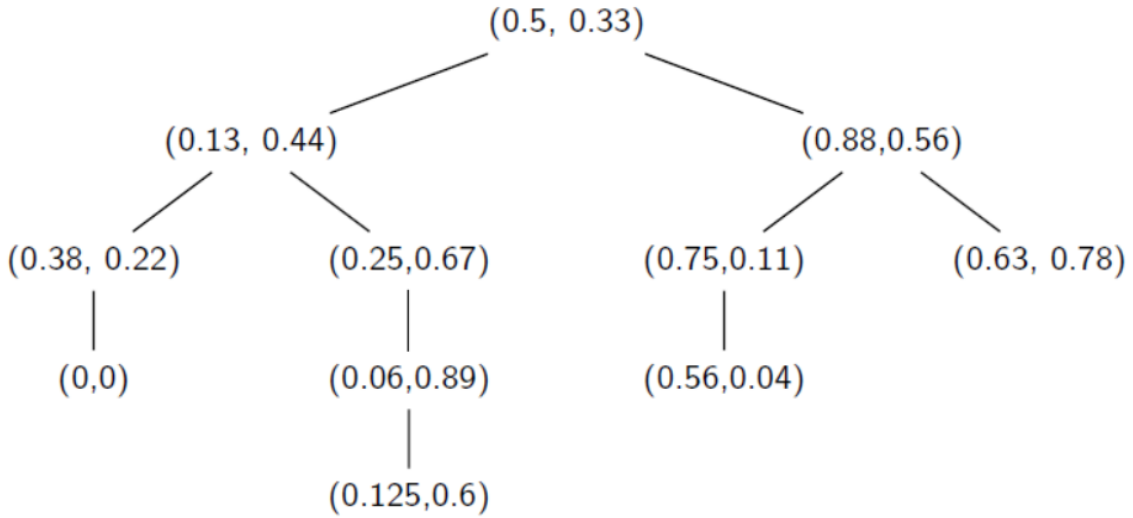




Nearest Neighbor Search on A k-d Tree

- Proceed down the tree as if x was a new point to add.
- When a leaf node is reached, the distance (squared) is computed and stored as current nearest.
- Sibling are searched if distance (squared) to the split is smaller than the current nearest.
- Otherwise the search moves up the tree, updating current nearest at parent nodes as appropriate.



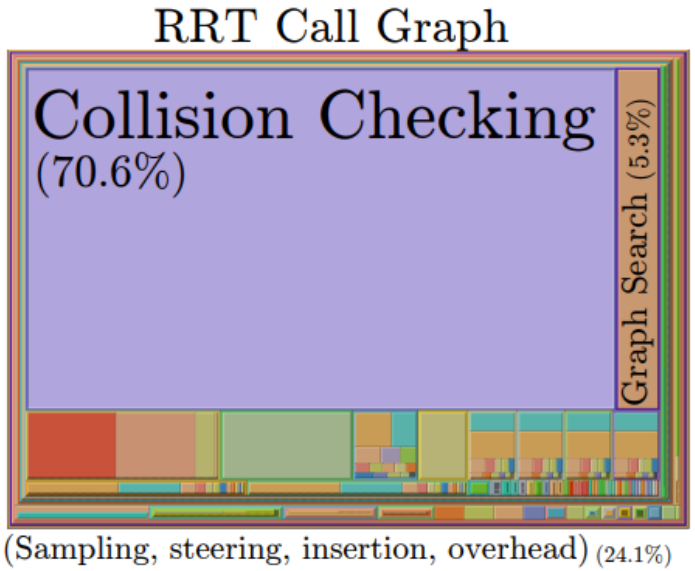


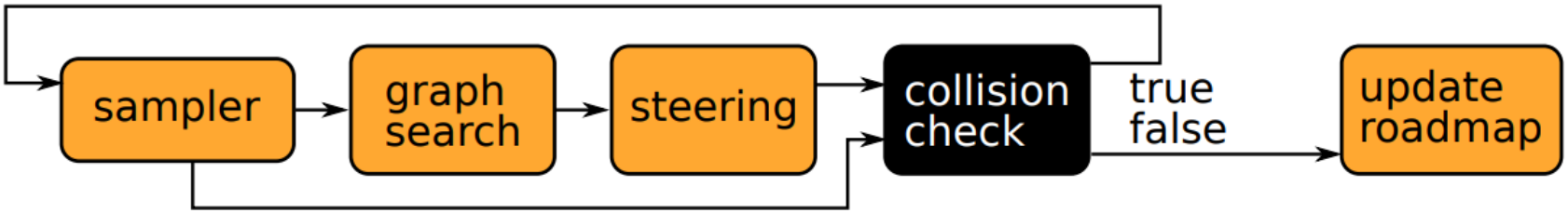
Nearest Neighbor Search Complexity

- Assuming that the k-d tree is balanced,
  - Inserting a new point takes $O(\log\ n)$ time.
  - A nearest-neighbor query takes $O(\log\ n)$ time.
- There are algorithms to maintain the tree approximately balanced with online/random insertions.
- Otherwise, it may be necessary to periodically rebalance the tree.

Collision Checking

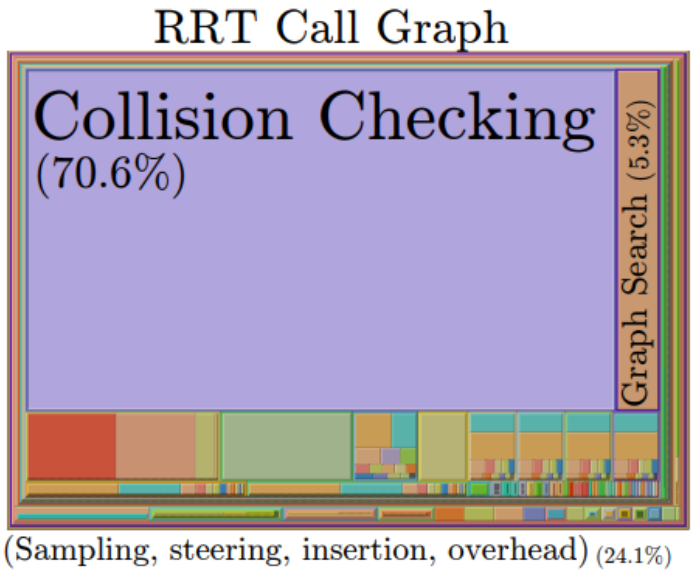
- Collision checking is typically the primary bottleneck in sampling based motion planning.
- It is also usually implemented as a black box routine.
- However, we can augment the data structure with results of the collision query — and reduce the frequency of collision checks!

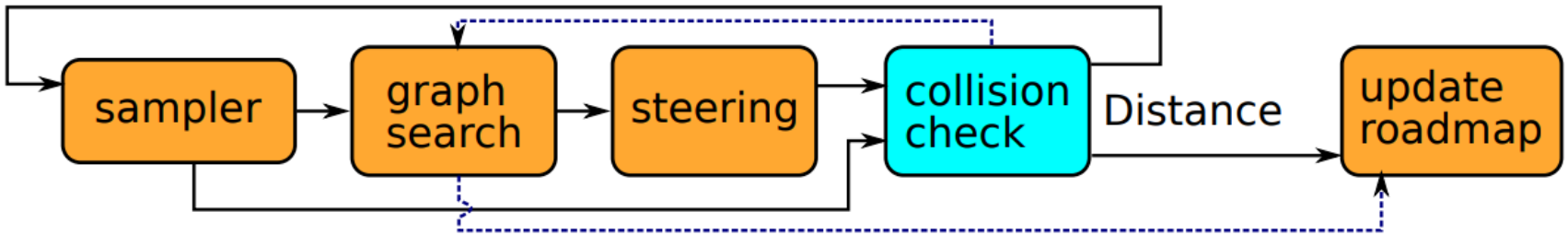




Collision Checking

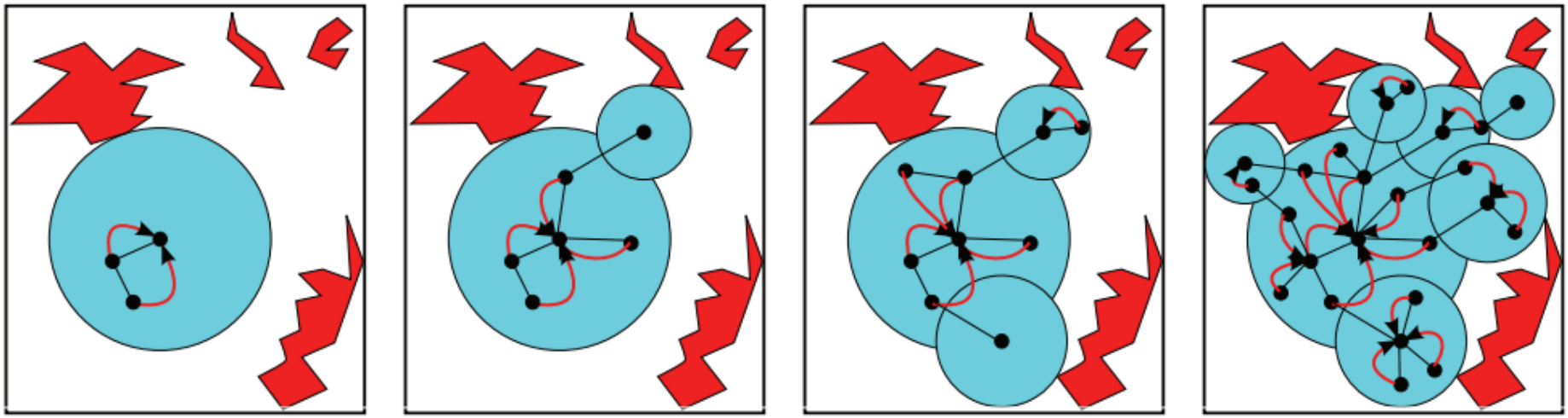
- Collision checking is typically the primary bottleneck in sampling based motion planning.
- It is also usually implemented as a black box routine.
- However, we can augment the data structure with results of the collision query — and reduce the frequency of collision checks!





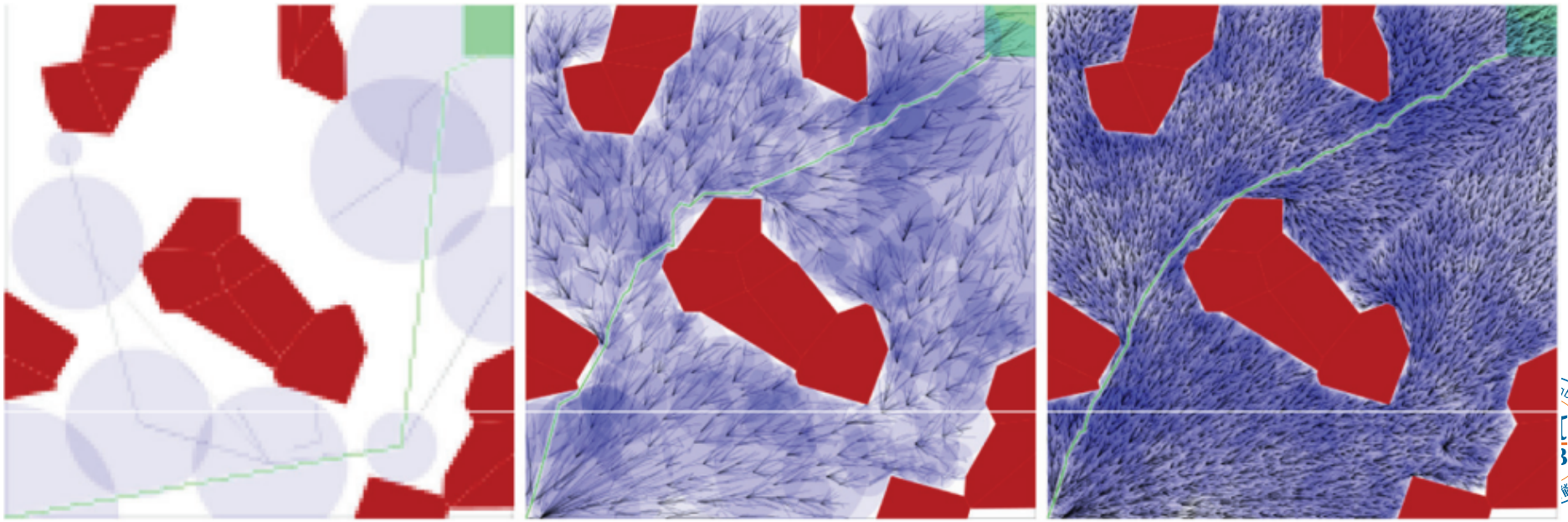
Efficient Collision Checking via Safety Certificates

- Collision-checked nodes store "safety certificates" defined by (a lower bound on) the distance to the nearest obstacle. Subsequent nodes within a certificate can forgo collision checking.
- Certificates (blue discs) asymptotically cover the space as graph size increases toward infinity. The ratio of collision checks versus graph size (nodes) approaches zero in the limit as graph size approaches infinity.
- Asymptotic complexity is driven by nearest-neighbor searches — NOT collision checks!



Efficient Collision Checking via Safety Certificates

- Collision-checked nodes store "safety certificates" defined by (a lower bound on) the distance to the nearest obstacle. Subsequent nodes within a certificate can forgo collision checking.
- Certificates (blue discs) asymptotically cover the space as graph size increases toward infinity. The ratio of collision checks versus graph size (nodes) approaches zero in the limit as graph size approaches infinity.
- Asymptotic complexity is driven by nearest-neighbor searches — NOT collision checks!



Summary and Key Learning Objectives

- Introduced two of the main sampling-based algorithms, PRM (multiple-query) and RRT (single-query).
- Discussed the main components necessary to implement a sampling-based motion planning algorithms, namely:
  - Notions of distance: metrics, quasimetrics, cost-to-go functions.
  - Dense point sequences: uniform random sequences, van der Corput and Halton sequences.
  - Nearest neighbor search algorithms and data structures (k-d tree). Please do not underestimate this.
  - Discussed safety certificates to reduce the burden of collision checking.

Q & A

1. Sampling-based Methods