AA 274
Principles of Robotic Autonomy

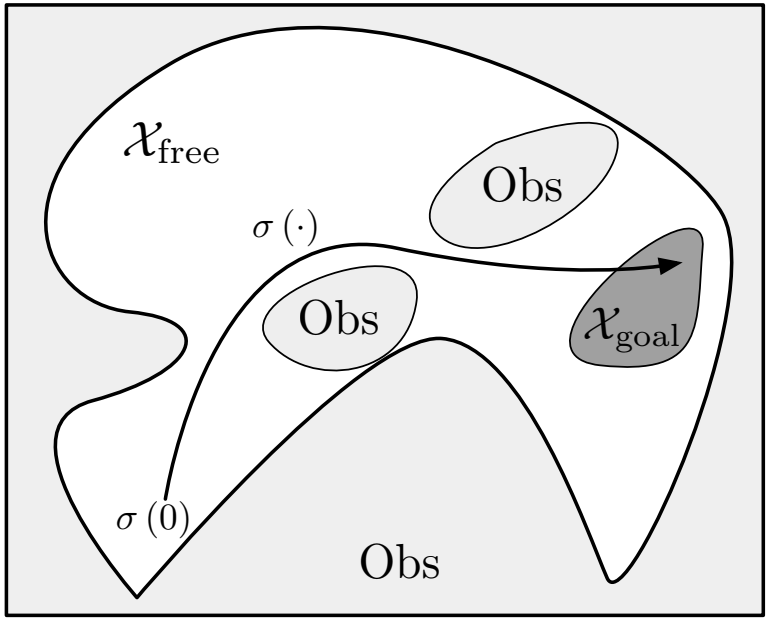
Sampling-based motion planning

Today's lecture

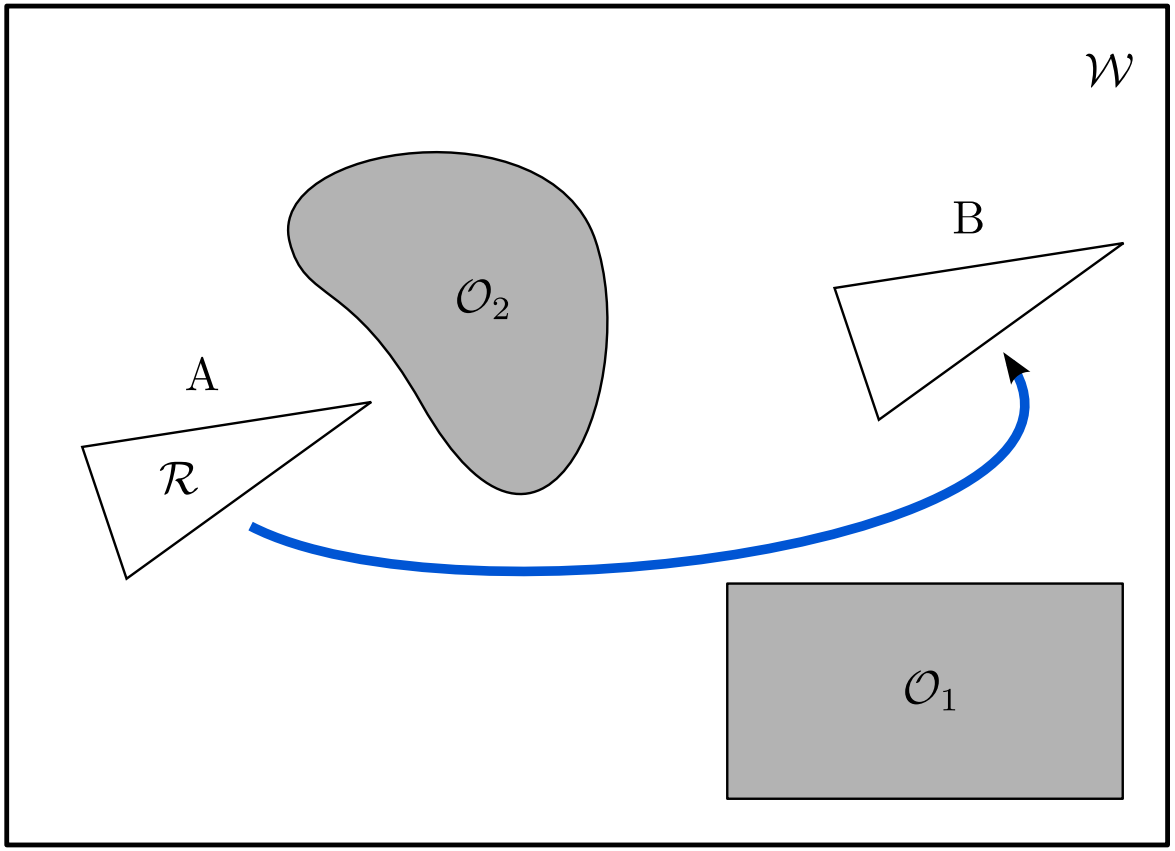
- Aim:
  - Learn about sampling-based motion planning algorithms
- Readings:
  - LaValle, Planning Algorithms, Chapter 5

The Motion Planning Problem

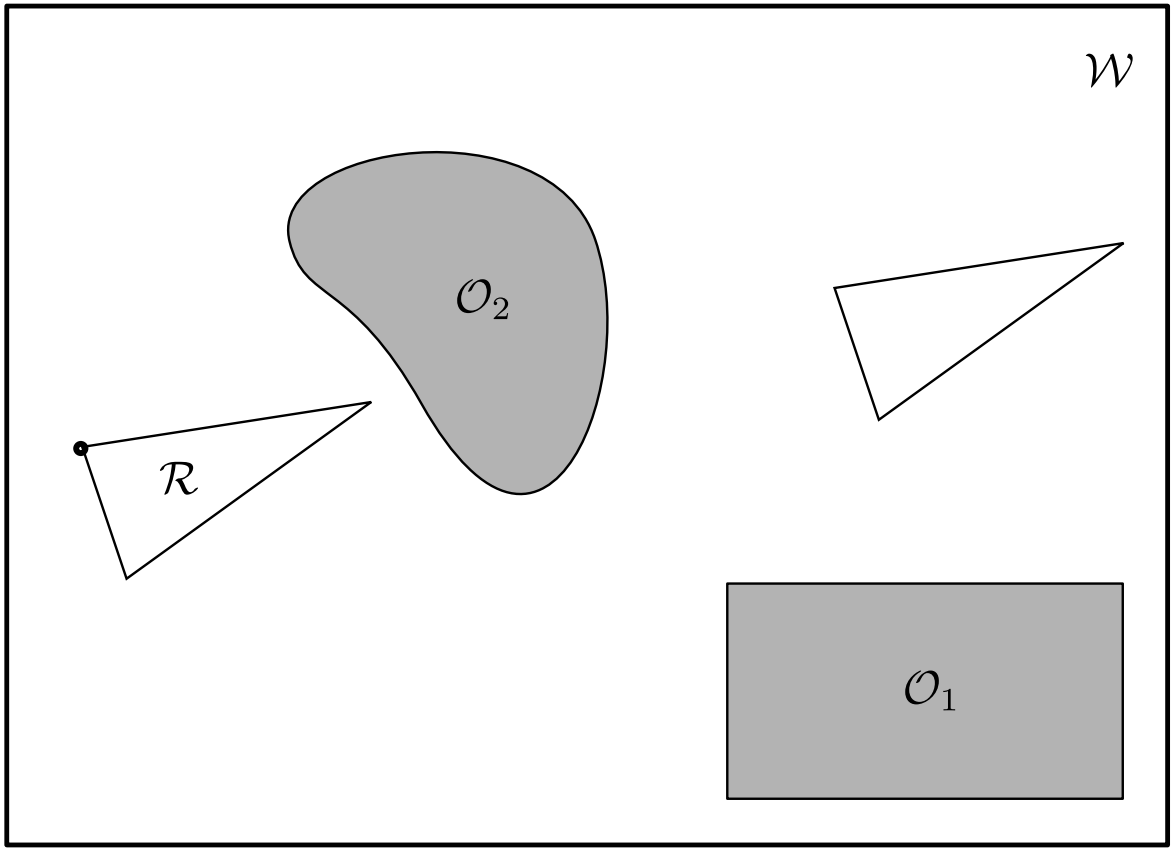
Find an action trajectory $\mathbf{u}(t)$ yielding a feasible path $\mathbf{x}(t) \in X_{\text{free}}$ over time horizon $t \in [0,\ T]$, which satisfies $\mathbf{x}(T) \in X_{\text{goal}}$ and minimizes $J = \int_0^T g(\mathbf{x}, \mathbf{u})\ dt$



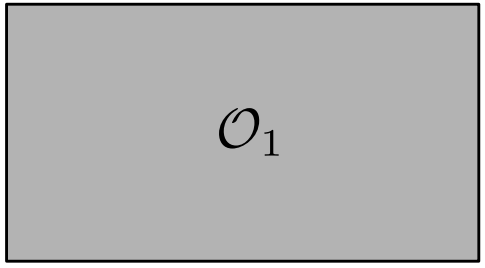
Configuration space



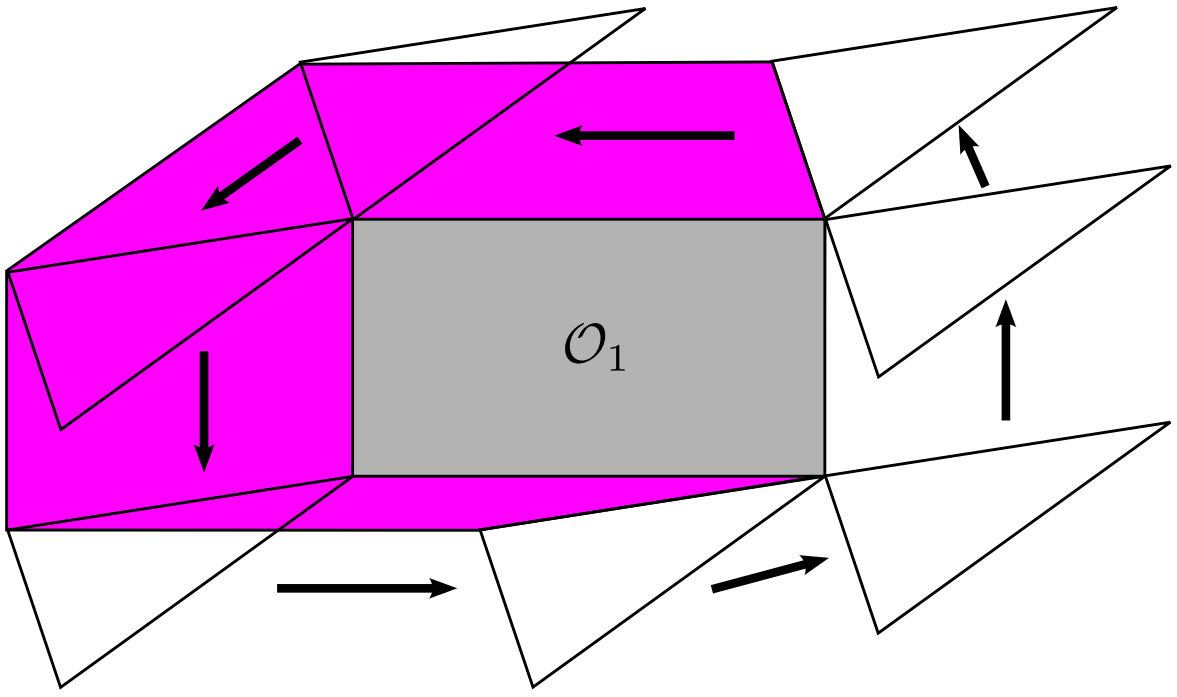
Configuration space



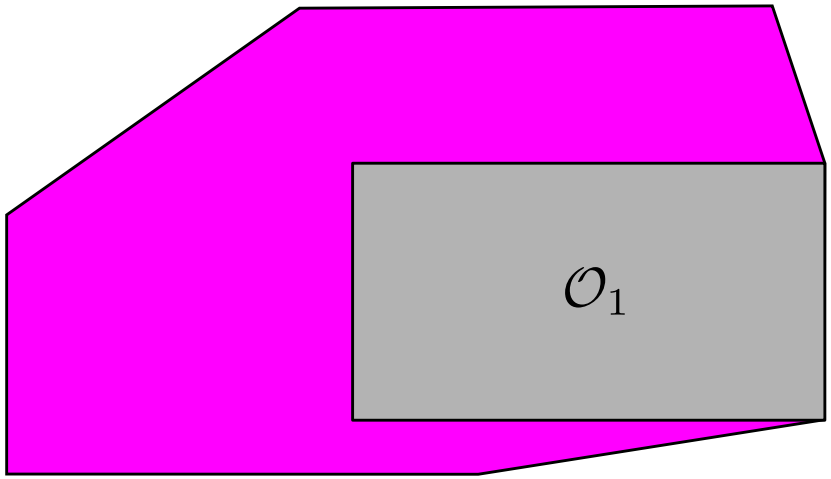
Configuration space



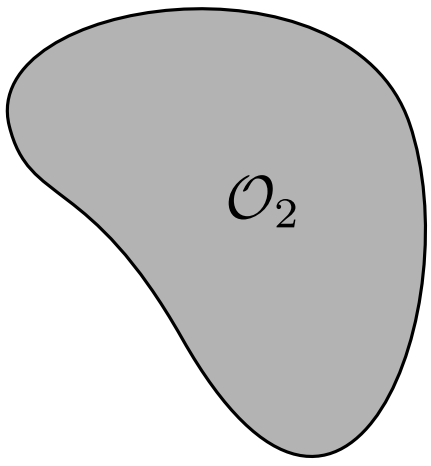
Configuration space



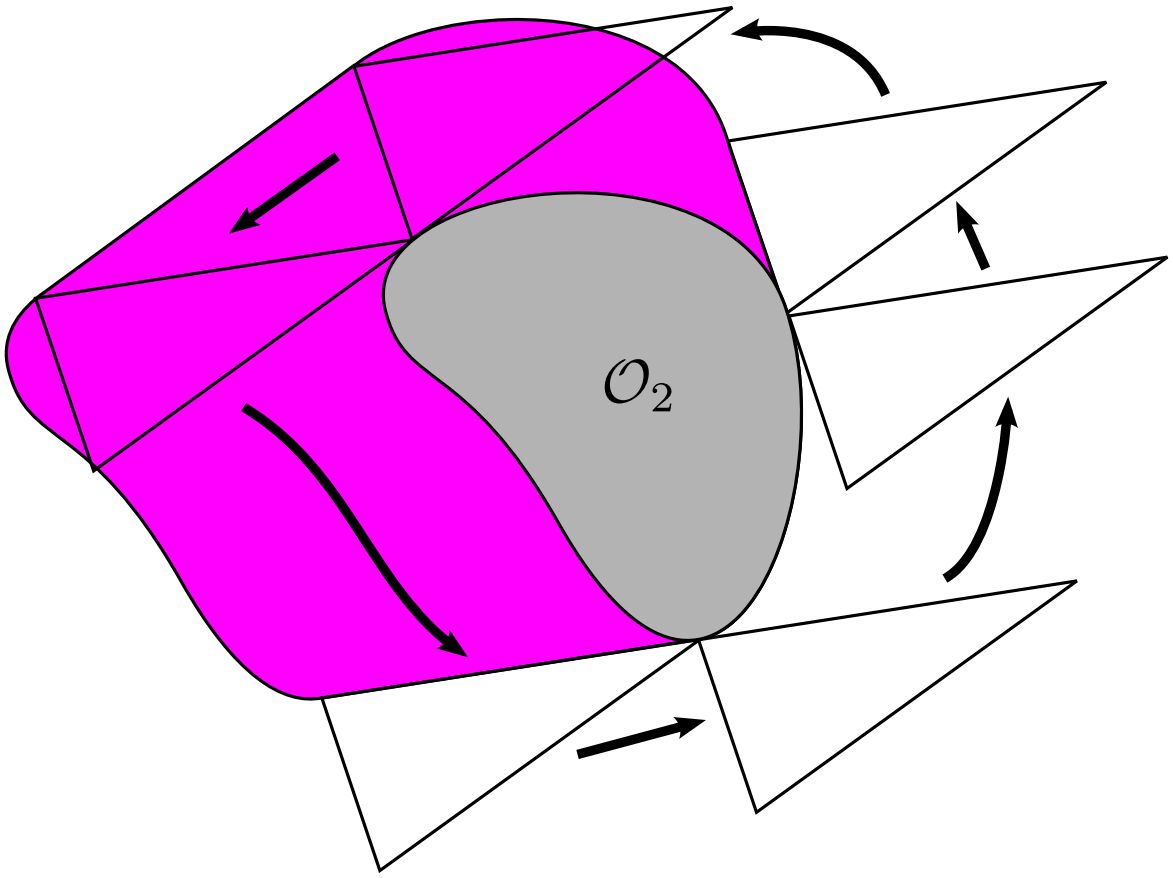
Configuration space



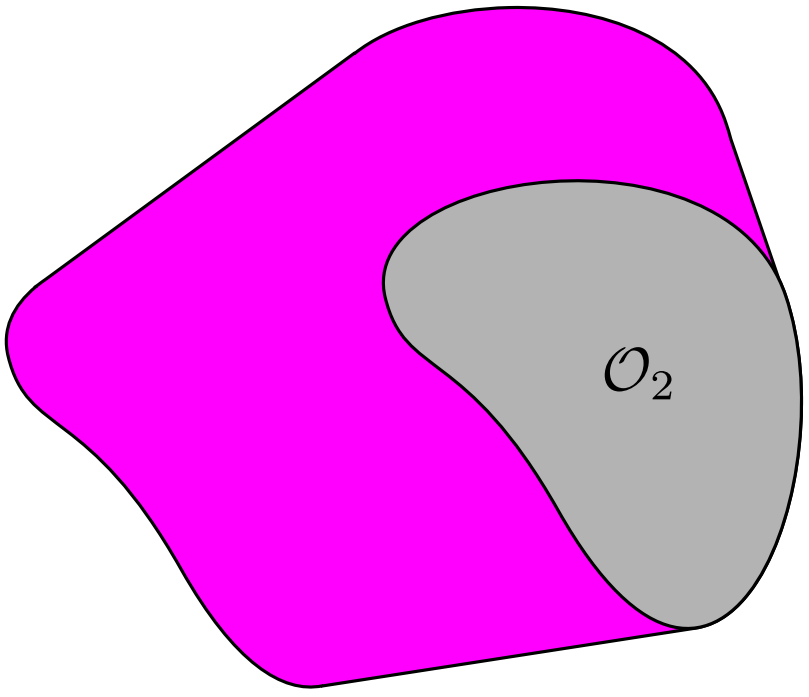
Configuration space



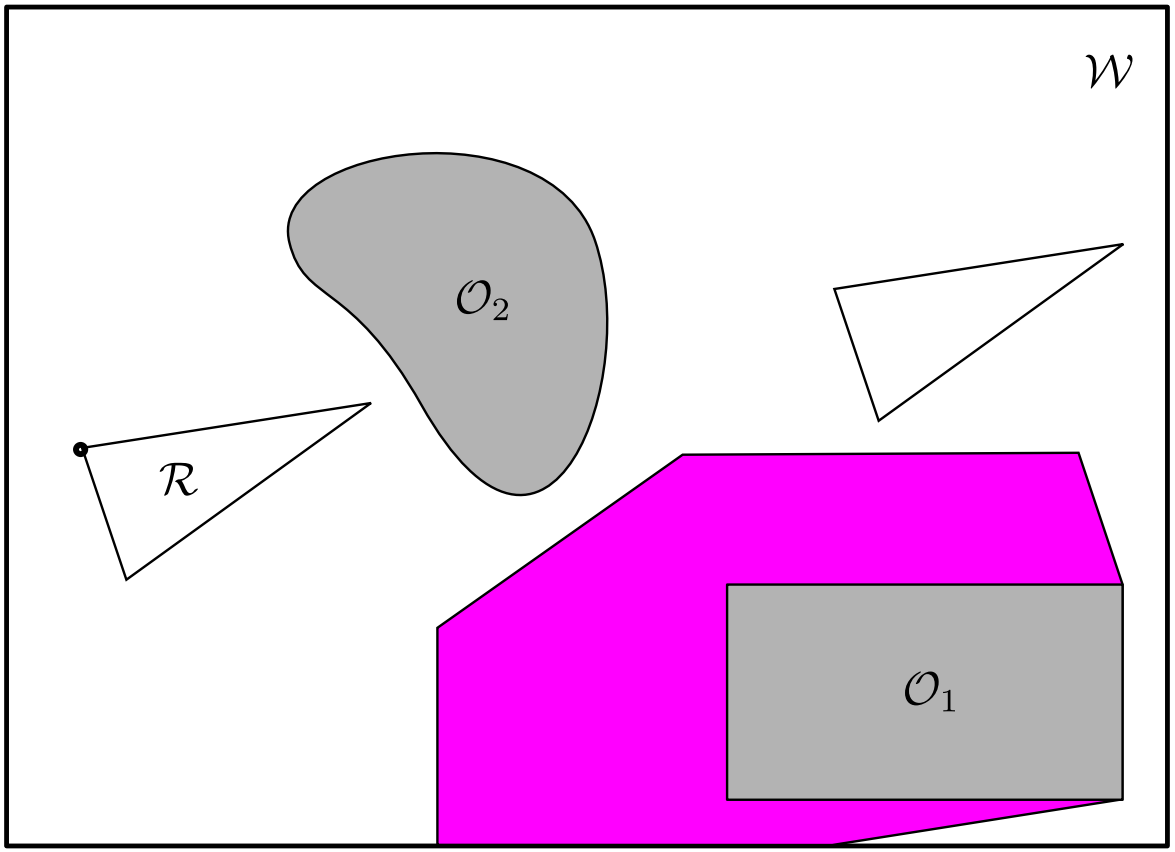
Configuration space



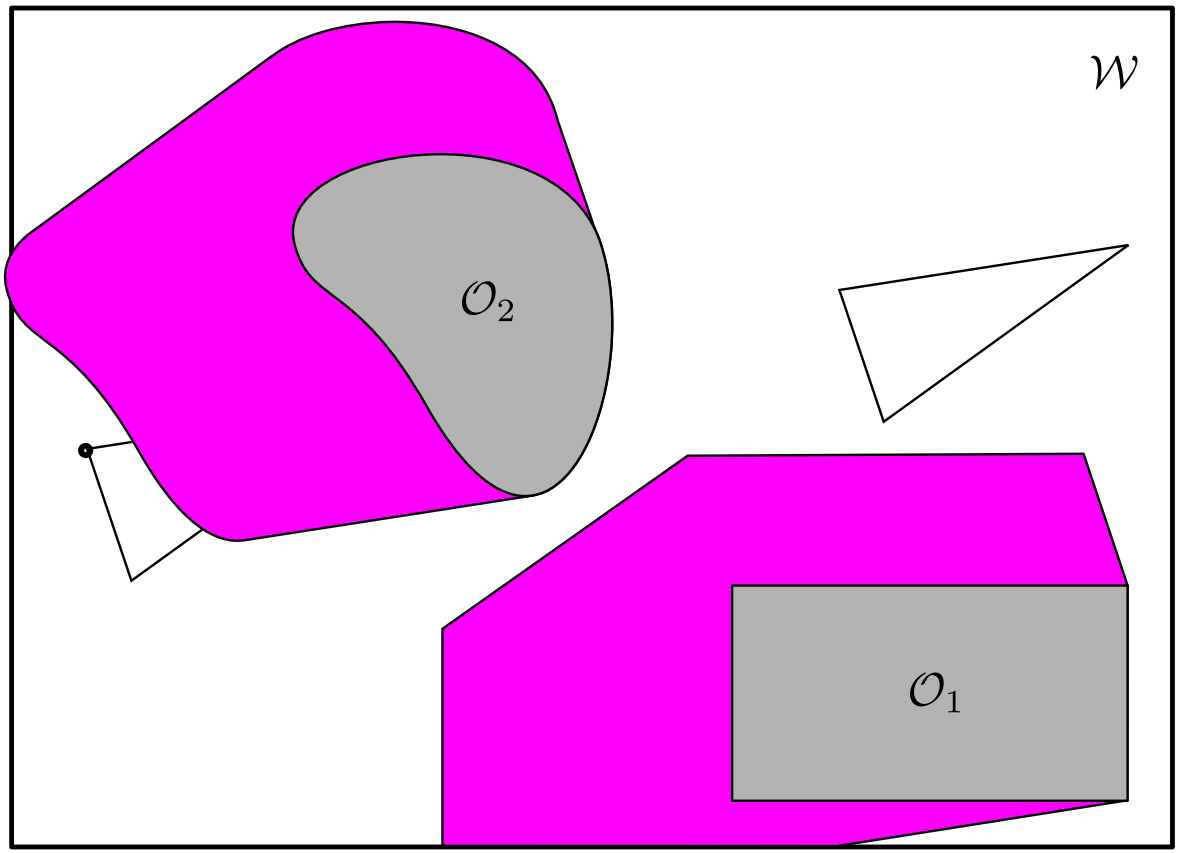
Configuration space



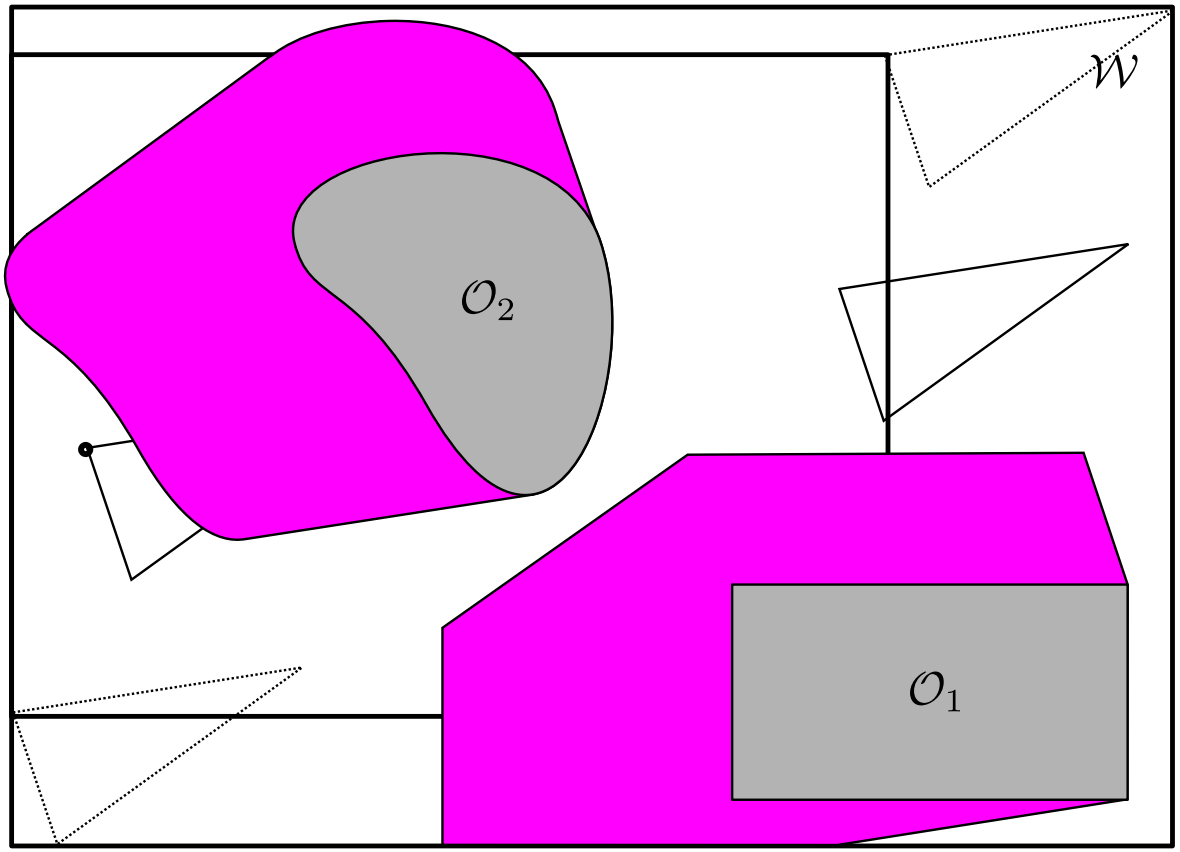
Configuration space



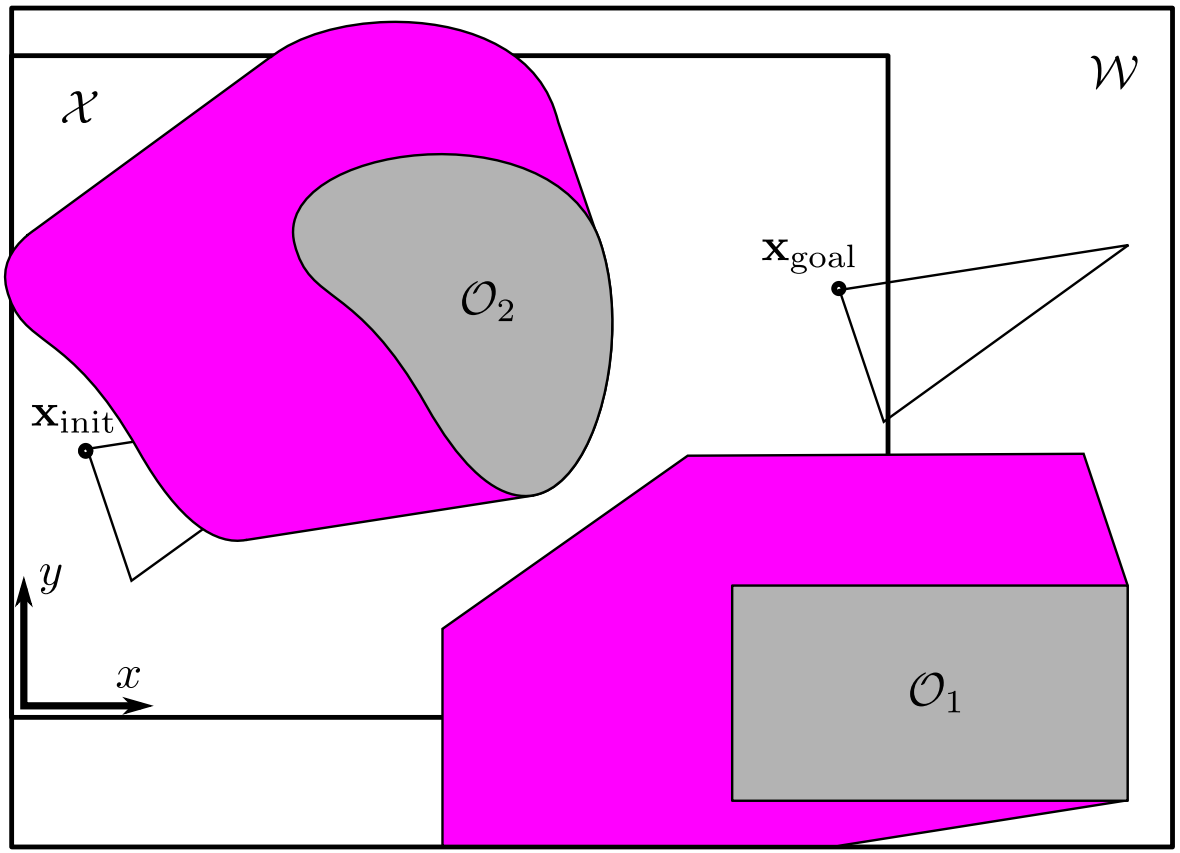
Configuration space



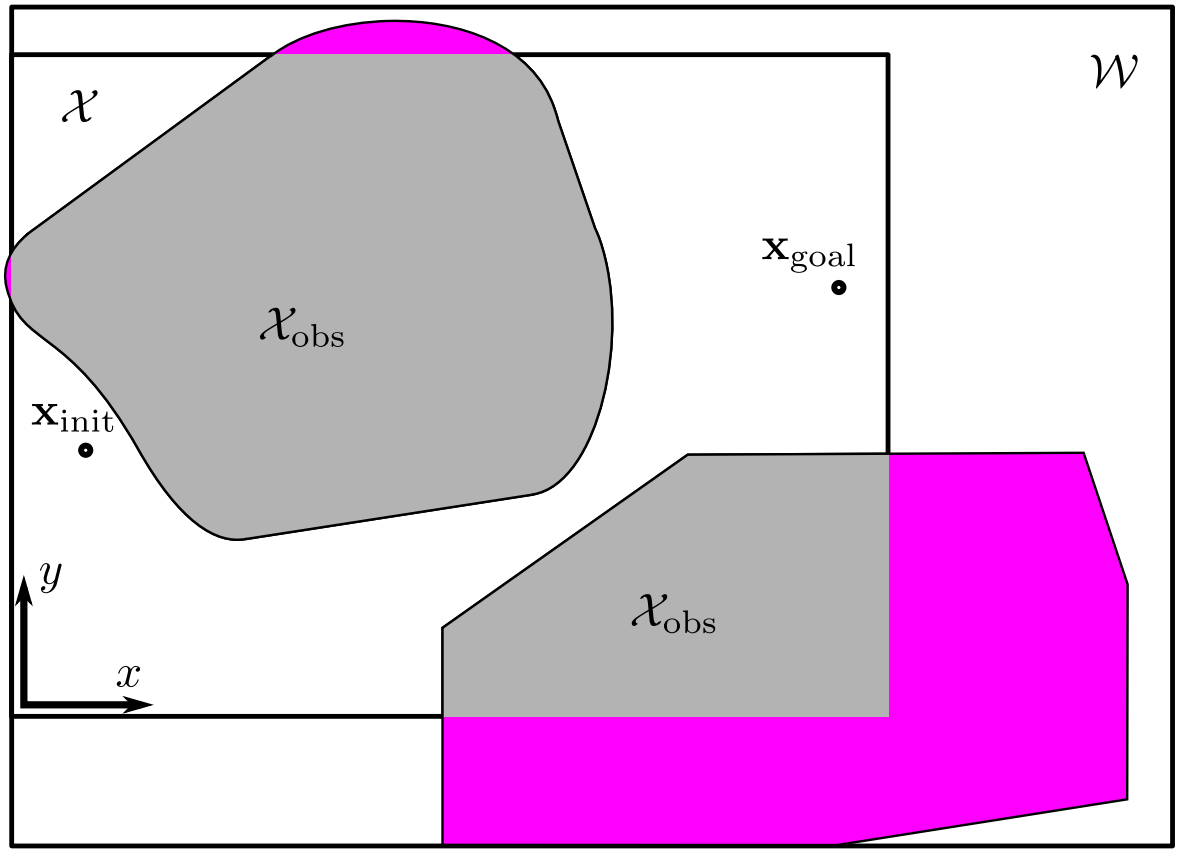
Configuration space



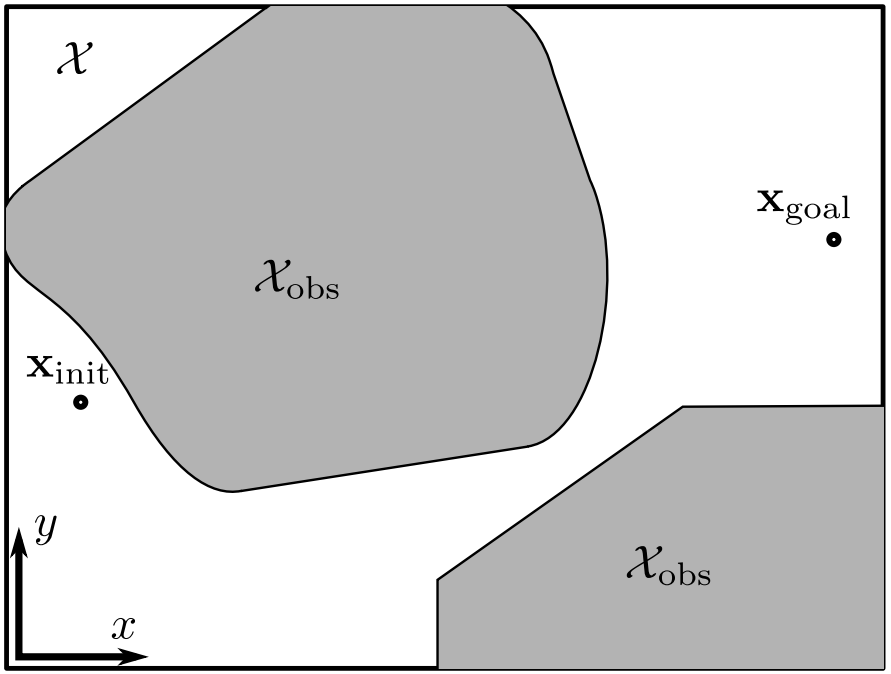
Configuration space



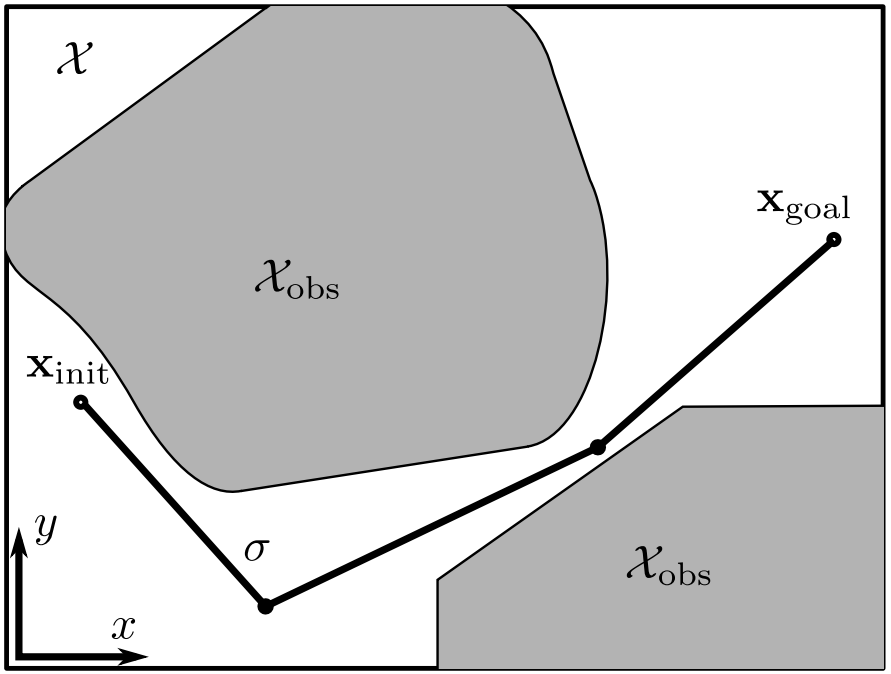
Configuration space



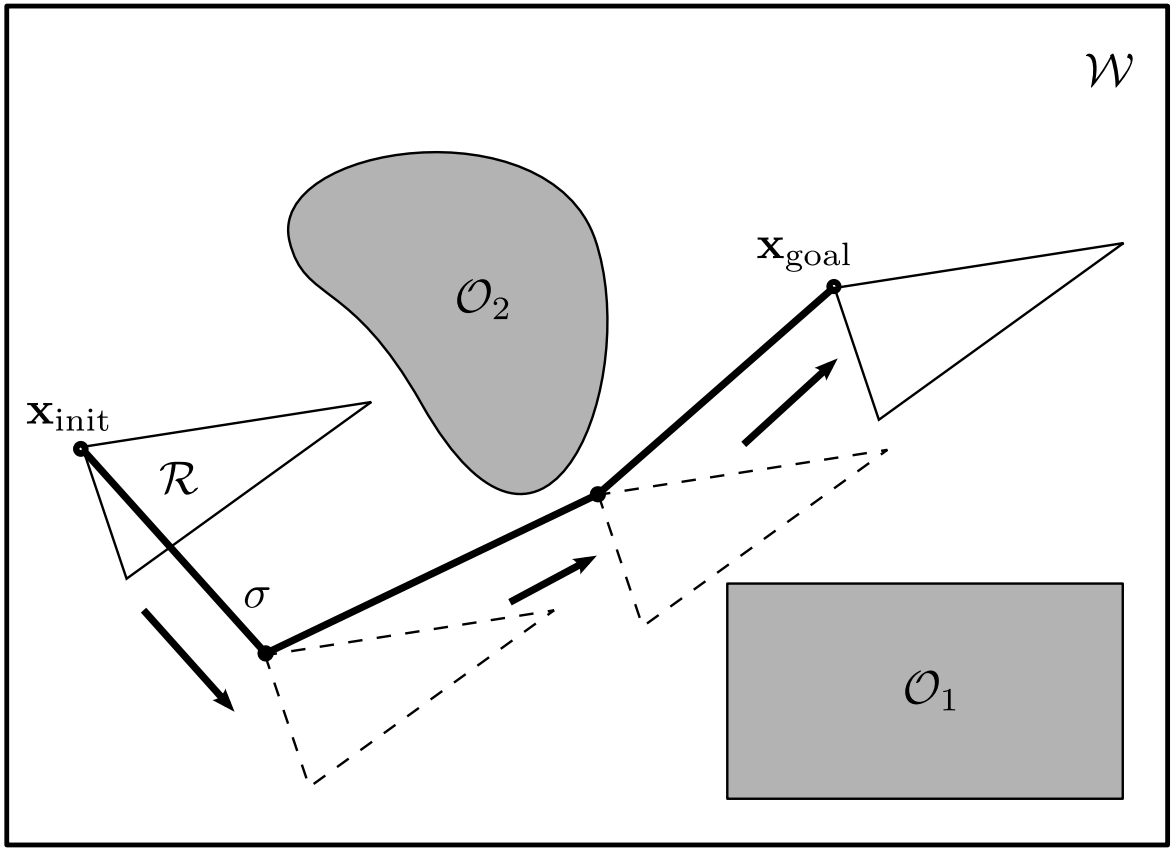
Configuration space



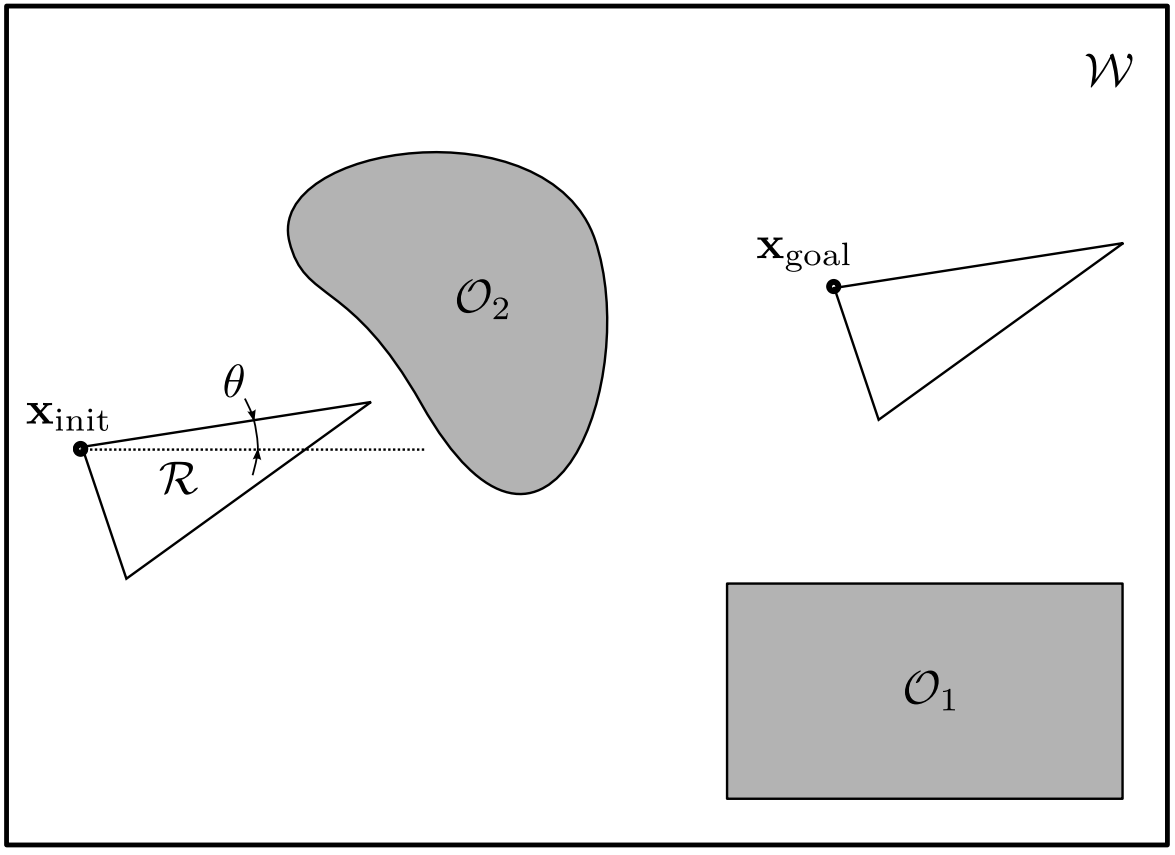
Configuration space



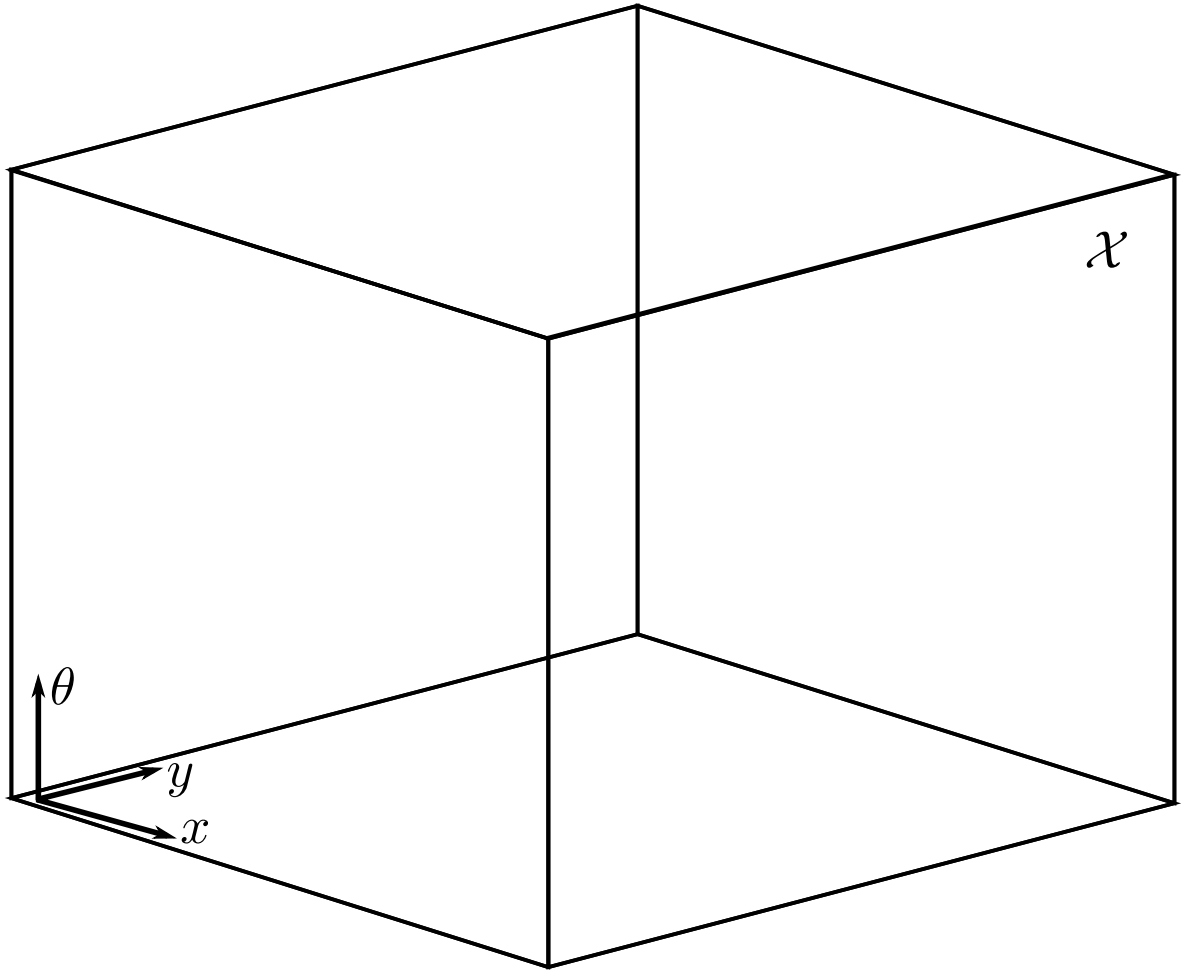
Configuration space



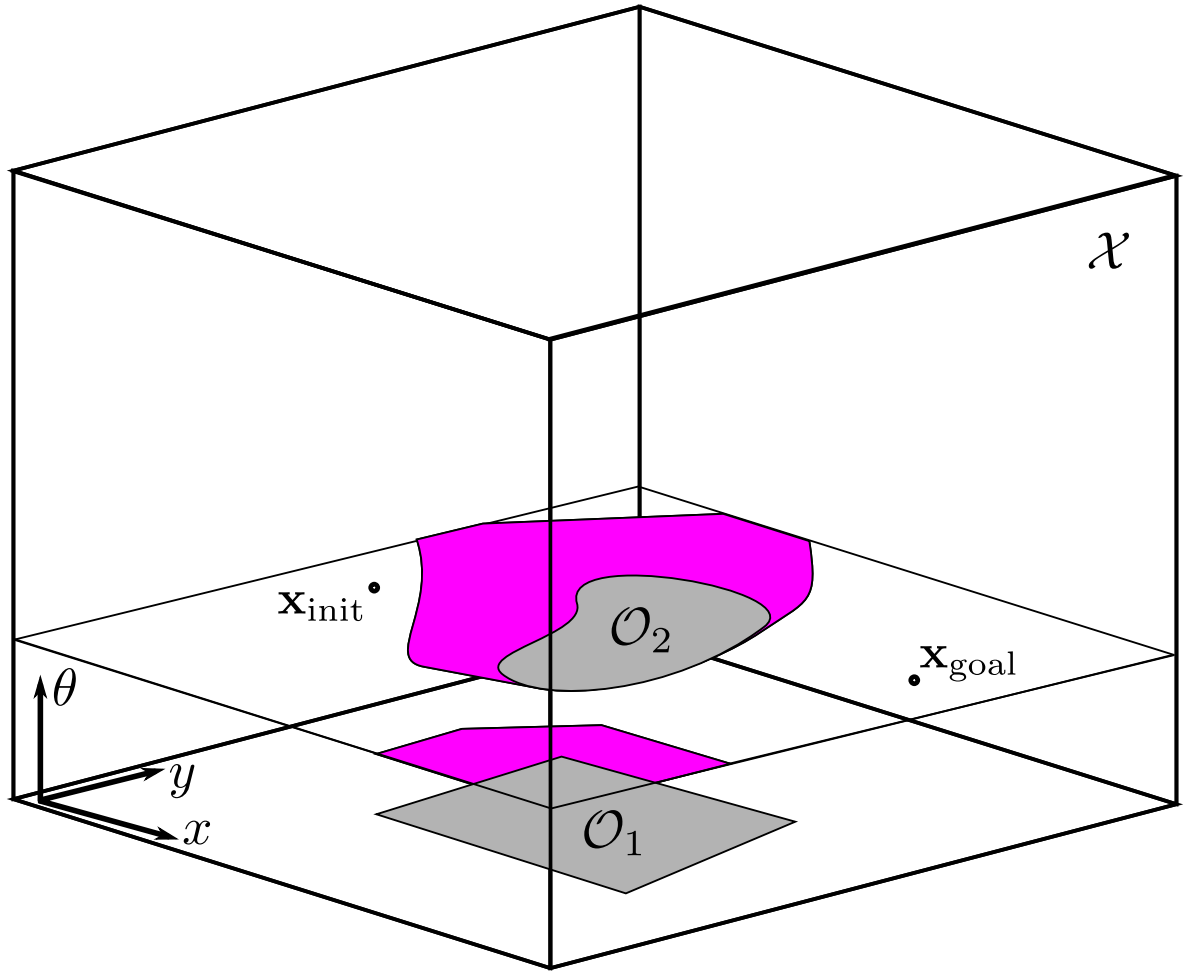
Configuration space



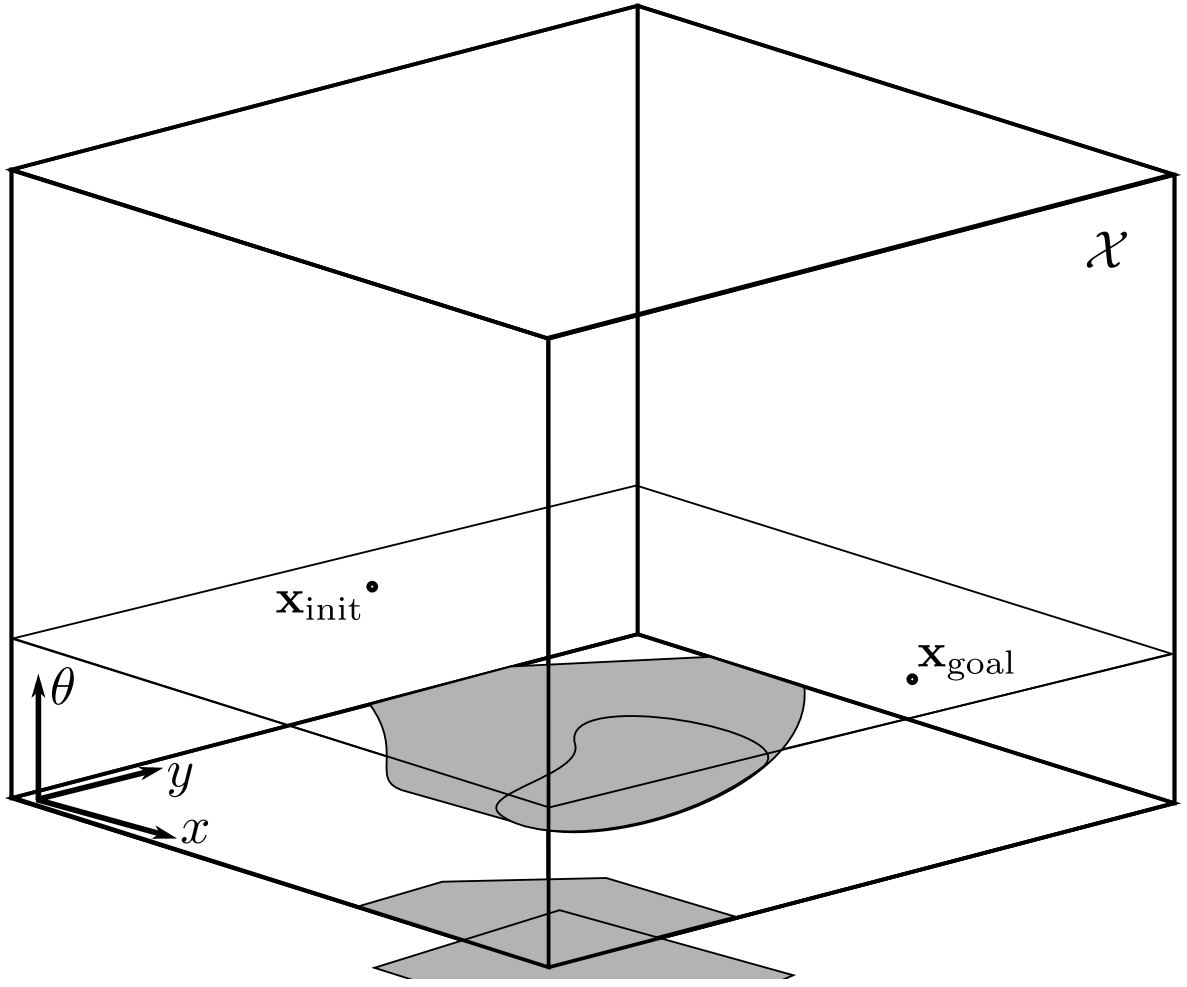
Configuration space



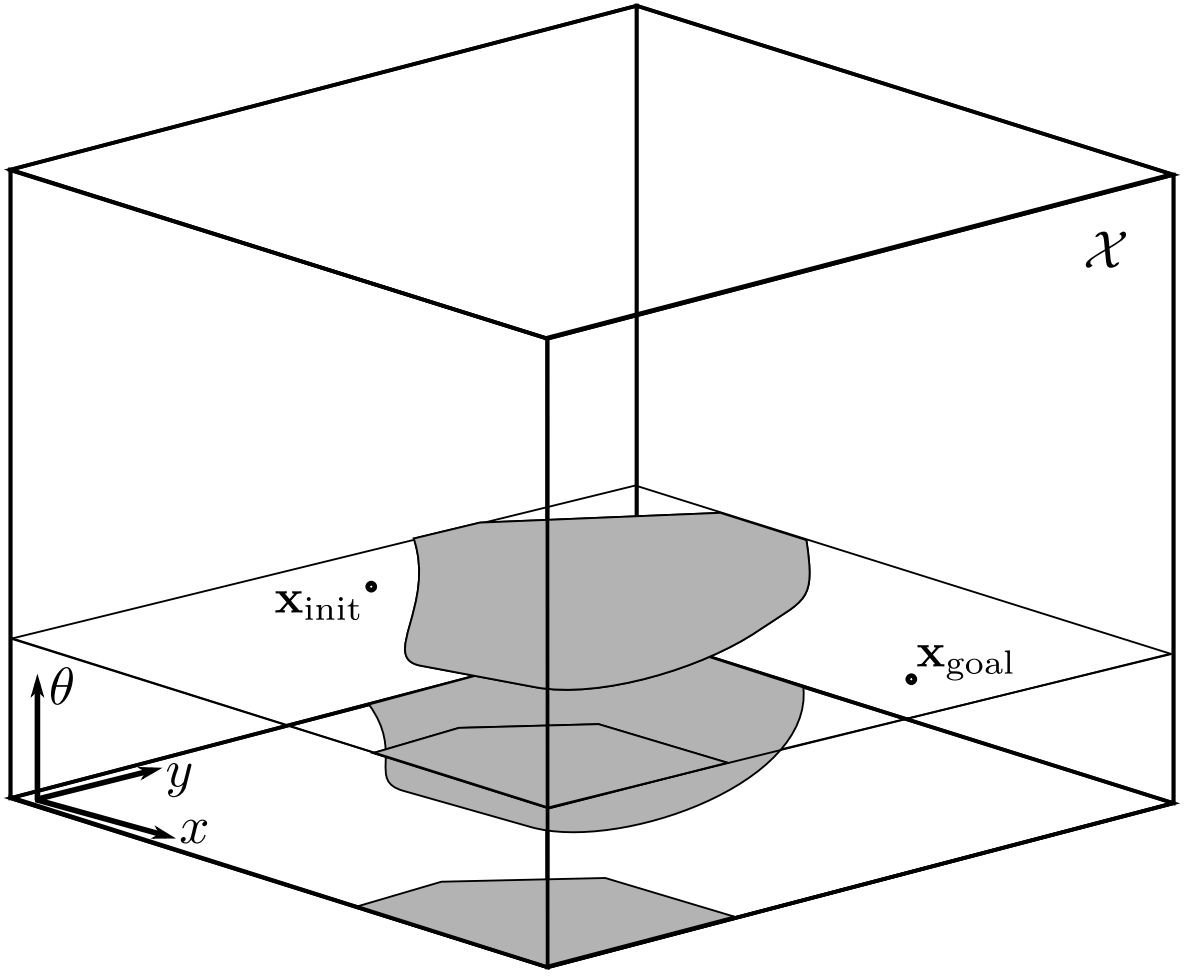
Configuration space



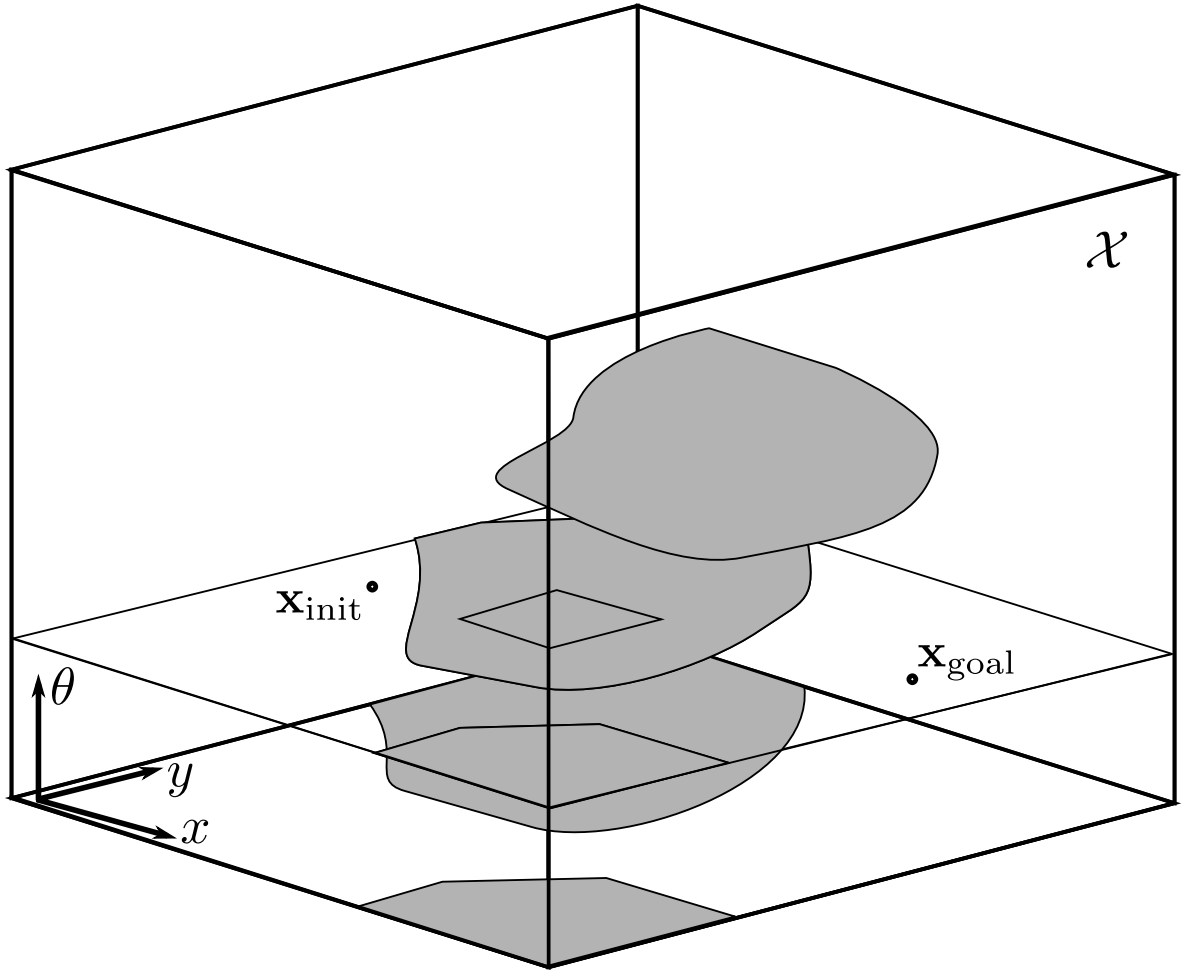
Configuration space



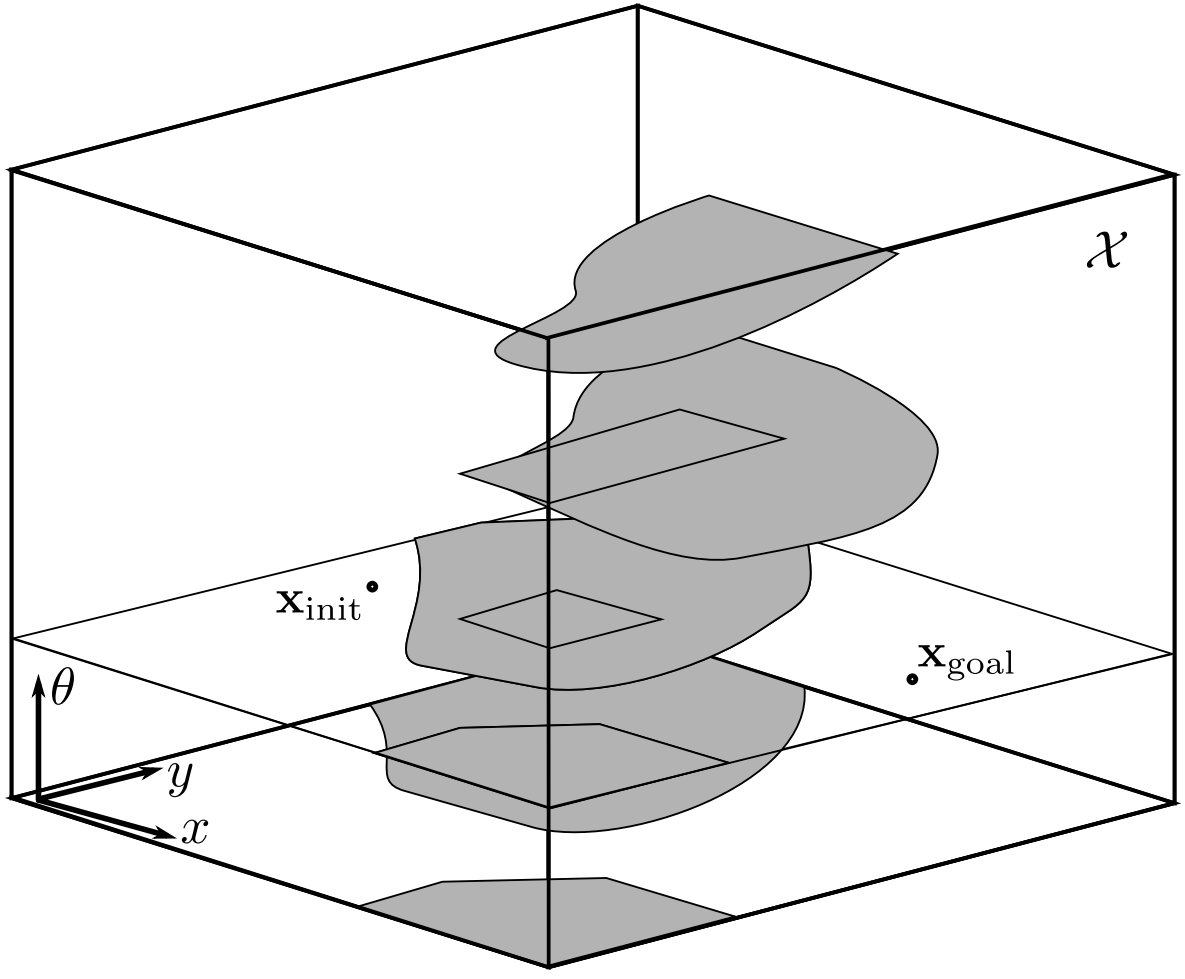
Configuration space



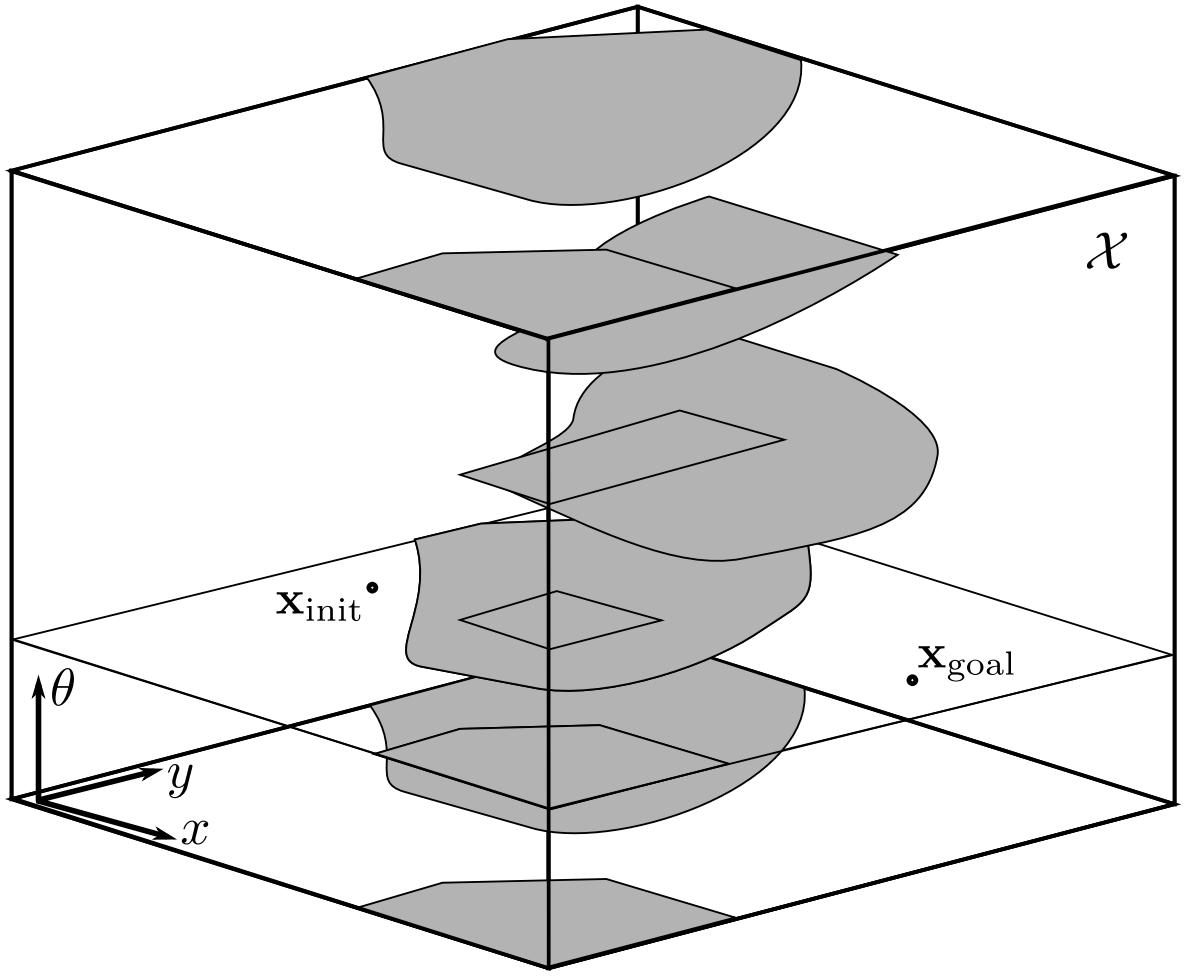
Configuration space



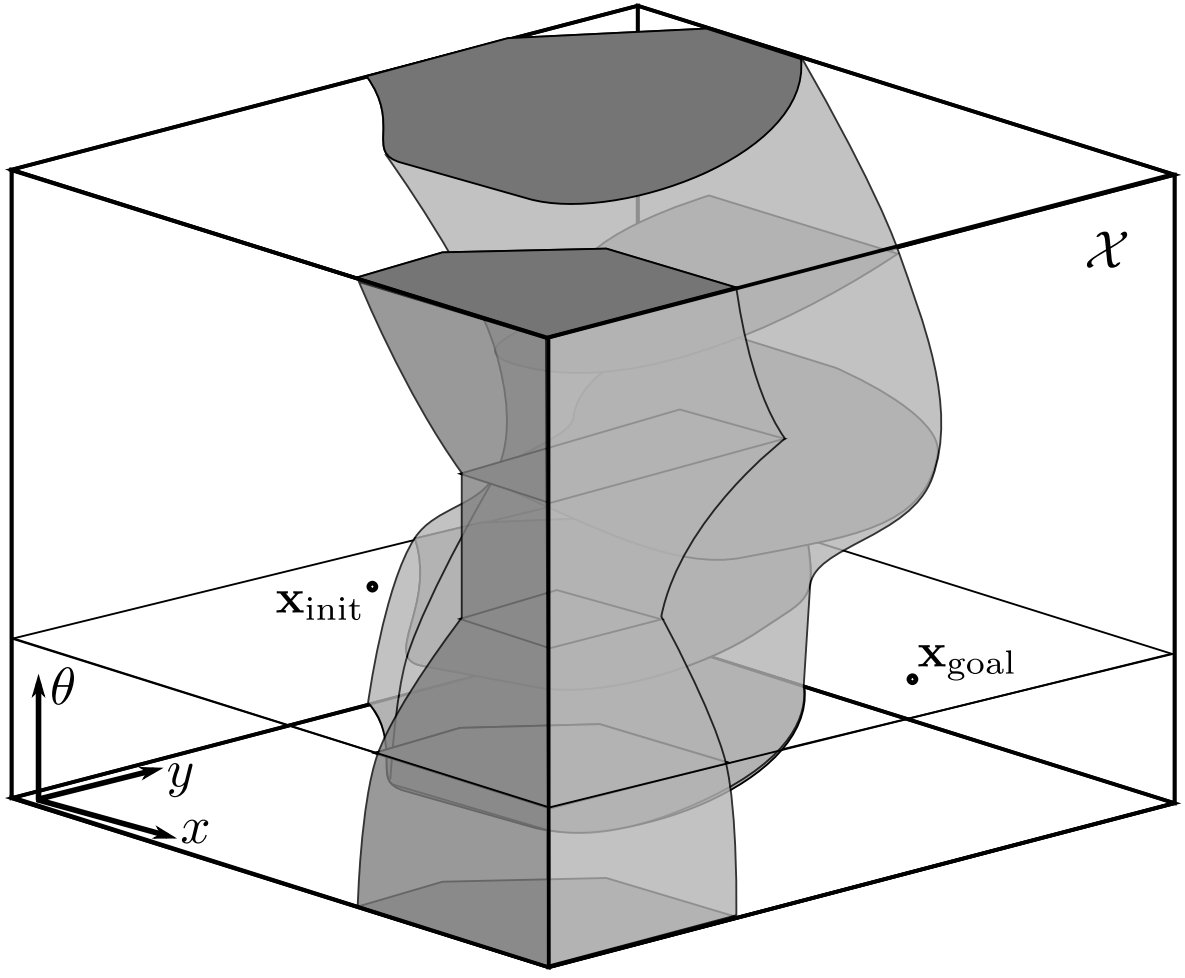
Configuration space



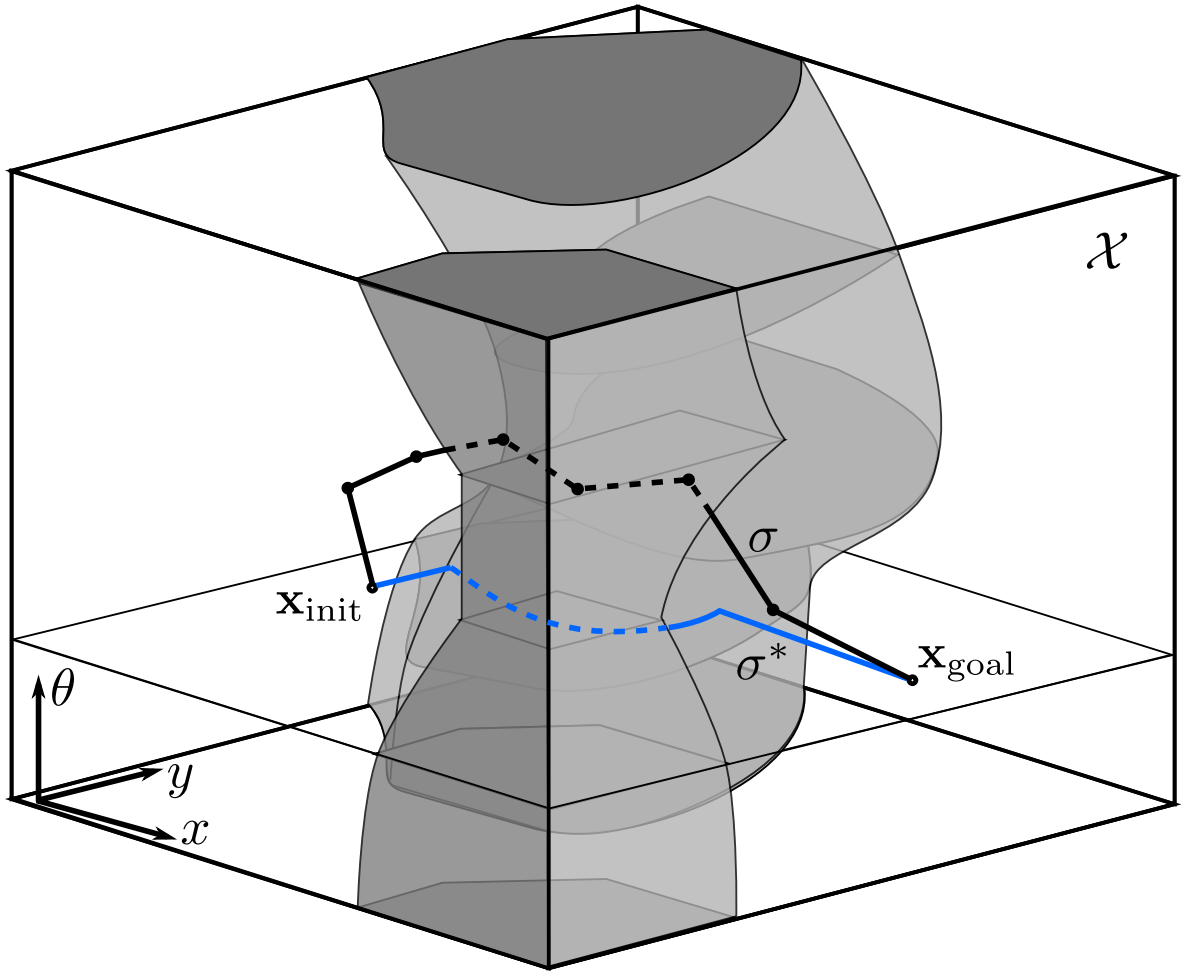
Configuration space



Configuration space



Configuration space



Configuration space: additional example

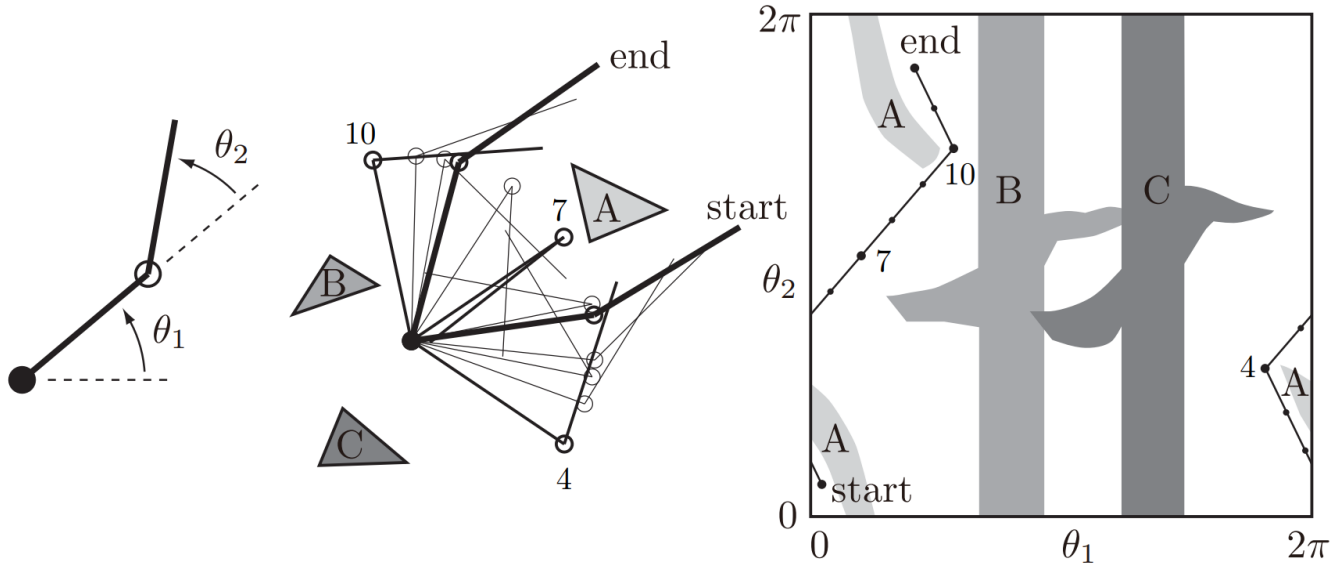


Figure 10.2: (Left) The joint angles of a 2R robot arm. (Middle) The arm navigating among obstacles A, B, and C. (Right) The same motion in C-space. Three intermediate points, 4, 7, and 10, along the path are labeled.

* Taken from "Modern robotics: mechanics, planning, and control", by Lynch and Park.

Motion Planning Algorithms

Key point: motion planning problem described in the real-world, but it really lives in an another space – the configuration (C-) space!

Motion Planning Algorithms

Key point: motion planning problem described in the real-world, but it really lives in an another space – the configuration (C-) space!

Two main approaches to motion planning

- *Combinatorial planning*: constructs structures in the C-space that discretely and completely capture all information needed to perform planning
- *Sampling-based planning*: uses collision detection algorithms to probe and incrementally search the C-space for a solution, rather than completely characterizing all of the C_{free} structure

Motion Planning Algorithms

Key point: motion planning problem described in the real-world, but it really lives in an another space – the configuration (C-) space!

Two main approaches to motion planning

- *Combinatorial planning*: constructs structures in the C-space that discretely and completely capture all information needed to perform planning
- *Sampling-based planning*: uses collision detection algorithms to probe and incrementally search the C-space for a solution, rather than completely characterizing all of the C_{free} structure

Bottom line:

- Combinatorial planning algorithms construct a *discrete* representation of the problem that *exactly* captures the solution (they correctly determine in finite time whether or not a solution exists)

Sampling-Based Motion Planning

Limitations of combinatorial approaches stimulated the development of sampling-based approaches

- Abandon the idea of explicitly characterizing C_{free} and C_{obs}
- Instead, capture the structure of C by **random sampling**
- Use a black-box component (collision checker) to determine whether which random configurations lie in C_{free}
- Use such probing scheme to incrementally build a roadmap and then plan a path

Sampling-based algorithms are by far the most common choice for industrial-grade problems.

References:

- LaValle, S. M. "Motion planning." Robotics & Automation Magazine, IEEE 18.1 (2011): 79-89.
- LaValle, S. M.. Planning algorithms. Cambridge university press, 2006 (chapters 1-5).

Sampling-Based Motion Planning

- Pros:
  - Conceptually simple
  - Relatively-easy to implement
  - Flexible:
    - One algorithm/implementation applies to a variety of robots and problems
  - Beyond the geometric case:
    - Can cope with complex differential constraints, uncertainty, etc.
    - Uncertainty
- (Mild) Cons:
  - Unclear how many samples should be generated to retrieve a solutions
  - Incapable of determining that a solution doesn't exist

Outline

Outline

Review of Sampling-Based Methods

Traditionally, two major approaches:

- Probabilistic Roadmap (PRM): graph-based
  - **Multi-query** planner, i.e., designed to solve multiple path queries on the same scenario
  - Original version: [Kavraki et al., TRA '96]
  - "Lazy" version: [Bohlin & Kavraki, ICRA '00]
  - Dynamic version: [Jaillet & T. Siméon, ICRA '04]
  - Asymptotically optimal version: [Karaman & Frazzoli, IJRR '11]
- Rapidly-exploring Random Trees (RRT): tree-based
  - **Single-query** planner
  - Original version: [LaValle & Kuffner, IJRR '01]
  - RDT: [LaValle, Cambridge Press '06]
  - SRT: [Plaku et al., TR '05]
  - Asymptotically optimal version [Karaman & Frazzoli, IJRR '11]

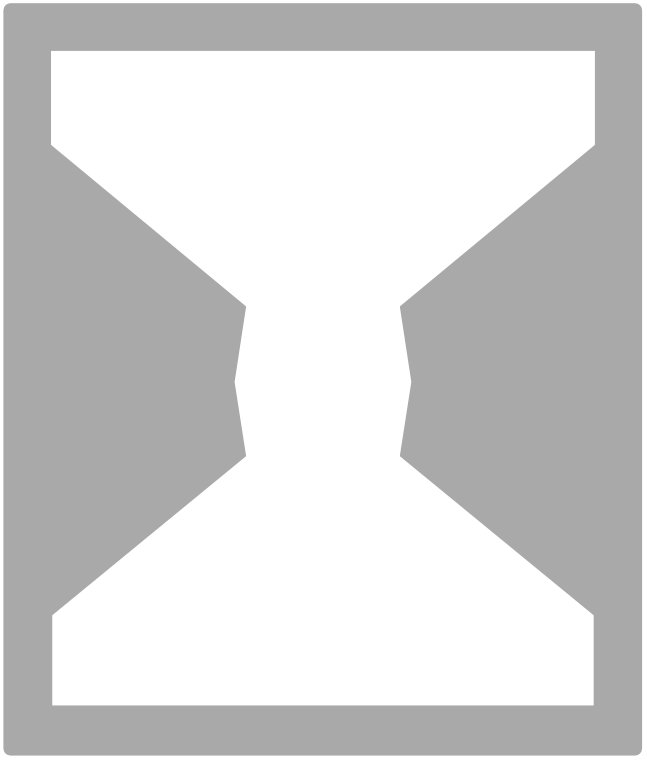
Probabilistic roadmaps (PRM)

A **multi**-query planner by Kavraki, Svestka, Latombe, and Overmars; 1995: generates a roadmap (graph) $\mathcal{G}$, that is embedded in the free space.

Preprocessing step:

1. Sample a collection of n configurations $\mathcal{X}_n$; discard forbidden configurations
2. Draw edge between every two samples $x, x' \in \mathcal{X}_n$ with $\|x - x'\| \leq r$ **and** straight-line path between x and x' is collision free

Given a query S, T, add these two vertices to $\mathcal{G}$ and find a roadmap path.



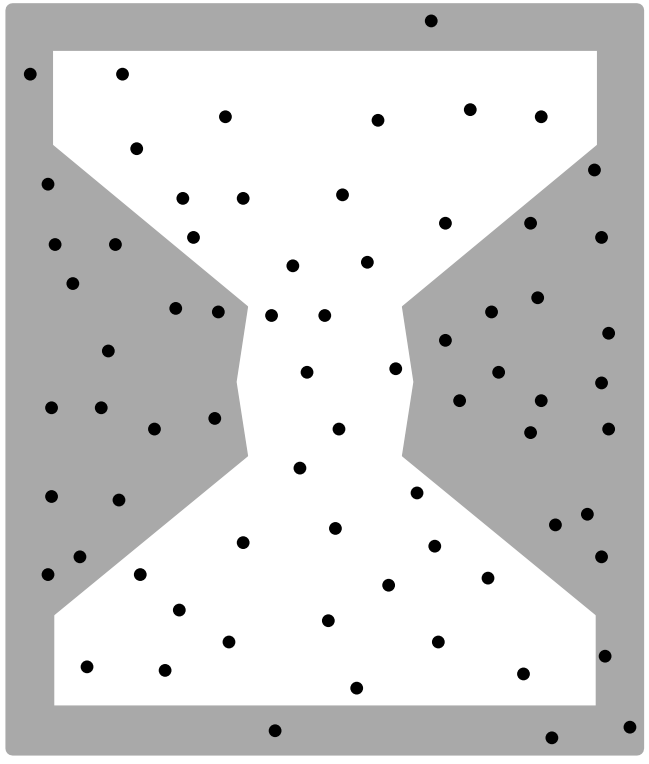
Probabilistic roadmaps (PRM)

A **multi**-query planner by Kavraki, Svestka, Latombe, and Overmars; 1995: generates a roadmap (graph) $\mathcal{G}$, that is embedded in the free space.

Preprocessing step:

1. Sample a collection of n configurations $\mathcal{X}_n$; discard forbidden configurations
2. Draw edge between every two samples $x, x' \in \mathcal{X}_n$ with $\|x - x'\| \leq r$ **and** straight-line path between x and x' is collision free

Given a query S, T, add these two vertices to $\mathcal{G}$ and find a roadmap path.



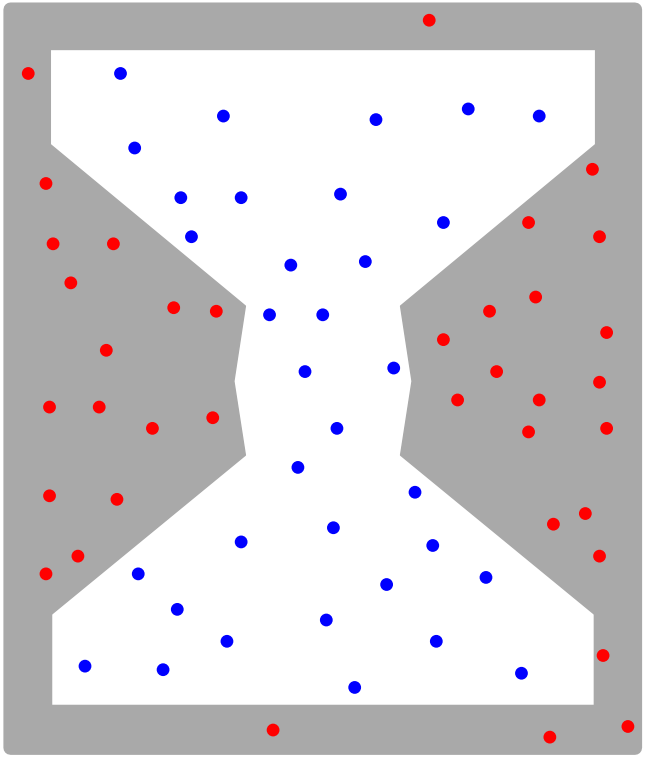
Probabilistic roadmaps (PRM)

A **multi**-query planner by Kavraki, Svestka, Latombe, and Overmars; 1995: generates a roadmap (graph) $\mathcal{G}$, that is embedded in the free space.

Preprocessing step:

1. Sample a collection of n configurations $\mathcal{X}_n$; discard forbidden configurations
2. Draw edge between every two samples $x, x' \in \mathcal{X}_n$ with $\|x - x'\| \leq r$ **and** straight-line path between x and x' is collision free

Given a query S, T, add these two vertices to $\mathcal{G}$ and find a roadmap path.



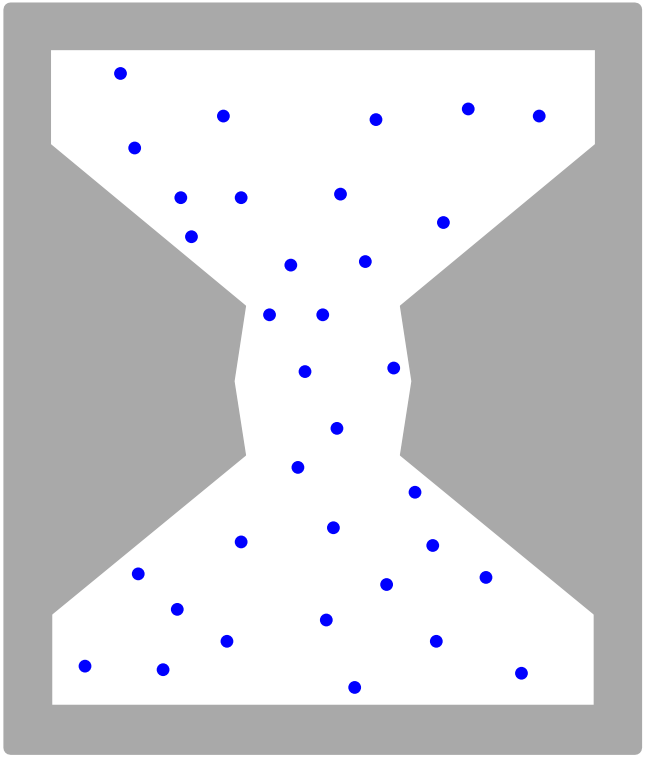
Probabilistic roadmaps (PRM)

A **multi**-query planner by Kavraki, Svestka, Latombe, and Overmars; 1995: generates a roadmap (graph) $\mathcal{G}$, that is embedded in the free space.

Preprocessing step:

1. Sample a collection of n configurations $\mathcal{X}_n$; discard forbidden configurations
2. Draw edge between every two samples $x, x' \in \mathcal{X}_n$ with $\|x - x'\| \leq r$ **and** straight-line path between x and x' is collision free

Given a query S, T, add these two vertices to $\mathcal{G}$ and find a roadmap path.



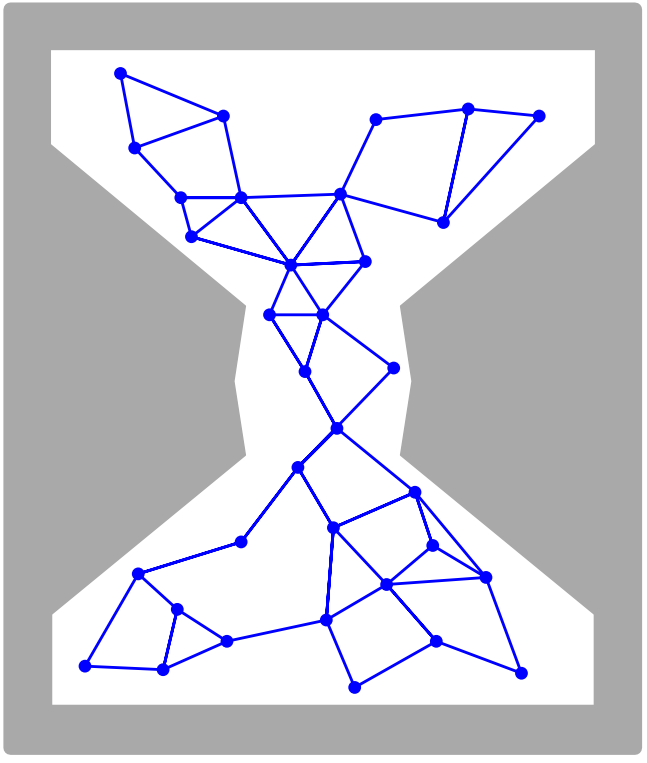
Probabilistic roadmaps (PRM)

A **multi**-query planner by Kavraki, Svestka, Latombe, and Overmars; 1995: generates a roadmap (graph) $\mathcal{G}$, that is embedded in the free space.

Preprocessing step:

1. Sample a collection of n configurations $\mathcal{X}_n$; discard forbidden configurations
2. Draw edge between every two samples $x, x' \in \mathcal{X}_n$ with $\|x - x'\| \leq r$ **and** straight-line path between x and x' is collision free

Given a query S, T, add these two vertices to $\mathcal{G}$ and find a roadmap path.



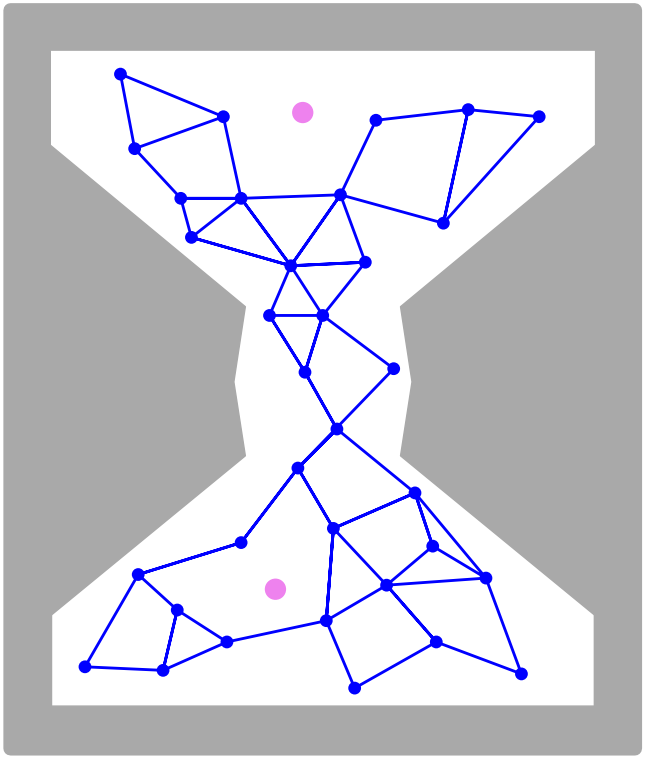
Probabilistic roadmaps (PRM)

A **multi**-query planner by Kavraki, Svestka, Latombe, and Overmars; 1995: generates a roadmap (graph) $\mathcal{G}$, that is embedded in the free space.

Preprocessing step:

1. Sample a collection of n configurations $\mathcal{X}_n$; discard forbidden configurations
2. Draw edge between every two samples $x, x' \in \mathcal{X}_n$ with $\|x - x'\| \leq r$ **and** straight-line path between x and x' is collision free

Given a query S, T, add these two vertices to $\mathcal{G}$ and find a roadmap path.



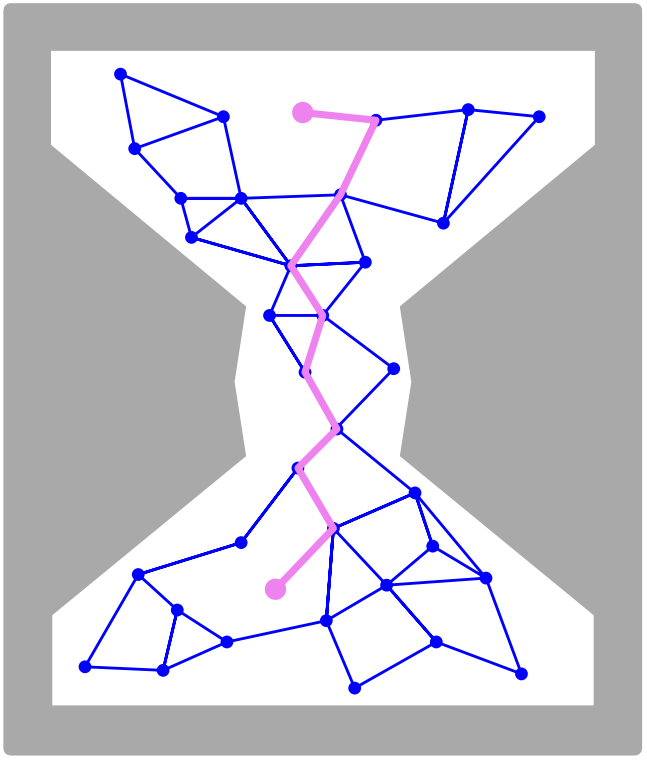
Probabilistic roadmaps (PRM)

A **multi**-query planner by Kavraki, Svestka, Latombe, and Overmars; 1995: generates a roadmap (graph) $\mathcal{G}$, that is embedded in the free space.

Preprocessing step:

1. Sample a collection of n configurations $\mathcal{X}_n$; discard forbidden configurations
2. Draw edge between every two samples $x, x' \in \mathcal{X}_n$ with $\|x - x'\| \leq r$ **and** straight-line path between x and x' is collision free

Given a query S, T, add these two vertices to $\mathcal{G}$ and find a roadmap path.

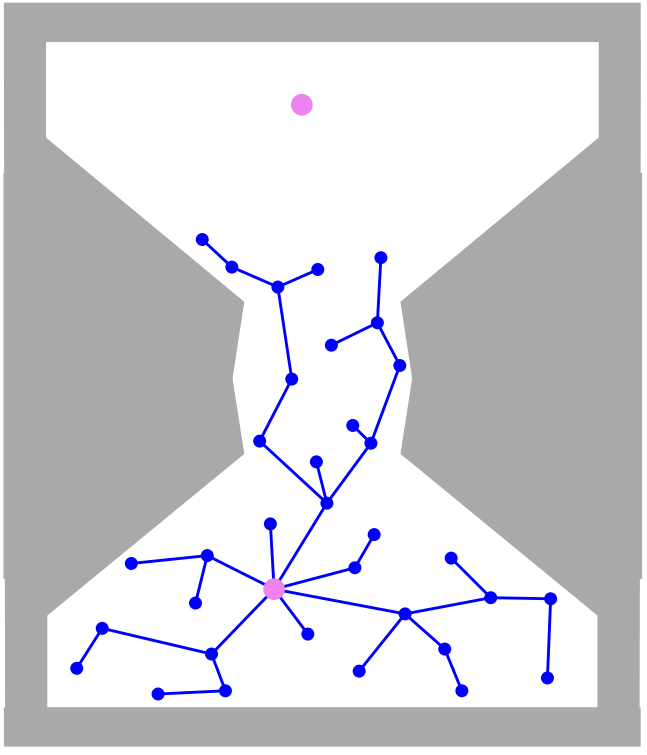


Rapidly-exploring random trees (RRT)

A **single**-query planner by LaValle; 1998: grows a tree $\mathcal{T}$, rooted in the start configuration S, that is embedded in the free space.

Algorithm works in n iterations. One iteration:

1. Sample a configuration x_{rnd}
2. Find the nearest vertex x_{near} of $\mathcal{T}$ to x_{rnd}
3. Generate a configuration x_{new} in the direction of x_{rnd} from x_{near}, such that $\overline{x_{\text{near}} x_{\text{new}}} \subset \mathcal{F}$
4. Update tree: $\mathcal{T} = \mathcal{T} \cup \{x_{\text{new}}, (x_{\text{near}}, x_{\text{new}})\}$



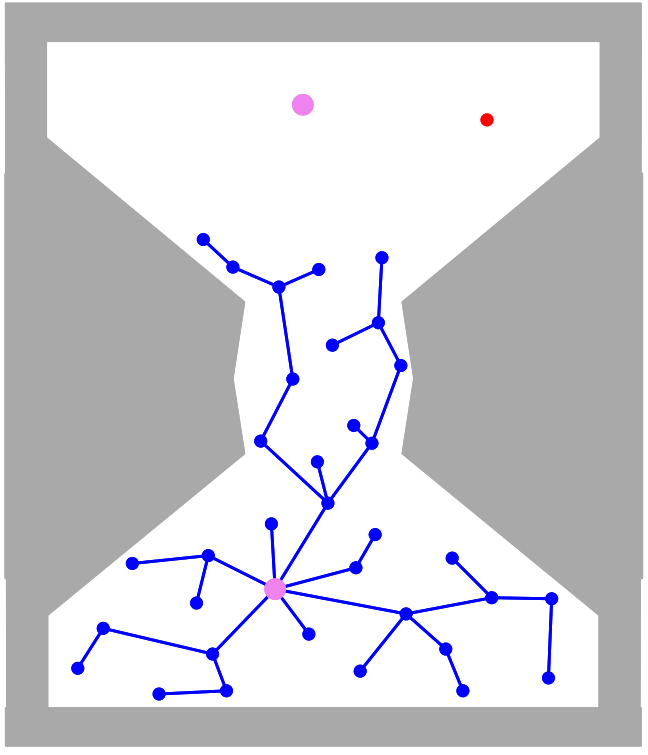
Every once in a while, set x_{rnd} to be the target vertex T; terminate if $x_{\text{new}} = T$.

Rapidly-exploring random trees (RRT)

A **single**-query planner by LaValle; 1998: grows a tree $\mathcal{T}$, rooted in the start configuration S, that is embedded in the free space.

Algorithm works in n iterations. One iteration:

1. Sample a configuration x_{rnd}
2. Find the nearest vertex x_{near} of $\mathcal{T}$ to x_{rnd}
3. Generate a configuration x_{new} in the direction of x_{rnd} from x_{near}, such that $\overline{x_{\text{near}} x_{\text{new}}} \subset \mathcal{F}$
4. Update tree: $\mathcal{T} = \mathcal{T} \cup \{x_{\text{new}}, (x_{\text{near}}, x_{\text{new}})\}$



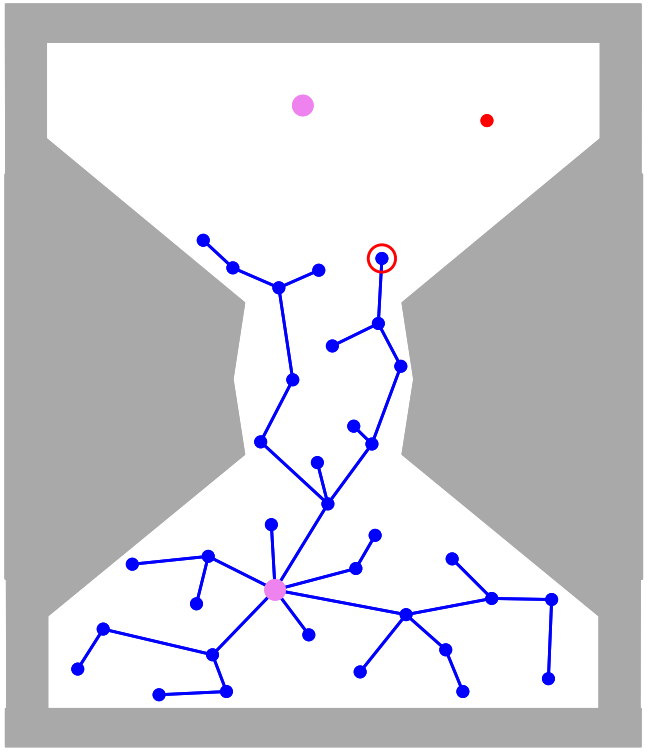
Every once in a while, set x_{rnd} to be the target vertex T; terminate if $x_{\text{new}} = T$.

Rapidly-exploring random trees (RRT)

A **single**-query planner by LaValle; 1998: grows a tree $\mathcal{T}$, rooted in the start configuration S, that is embedded in the free space.

Algorithm works in n iterations. One iteration:

1. Sample a configuration x_{rnd}
2. Find the nearest vertex x_{near} of $\mathcal{T}$ to x_{rnd}
3. Generate a configuration x_{new} in the direction of x_{rnd} from x_{near}, such that $\overline{x_{\text{near}} x_{\text{new}}} \subset \mathcal{F}$
4. Update tree: $\mathcal{T} = \mathcal{T} \cup \{x_{\text{new}}, (x_{\text{near}}, x_{\text{new}})\}$



Every once in a while, set x_{rnd} to be the target vertex T; terminate if $x_{\text{new}} = T$.

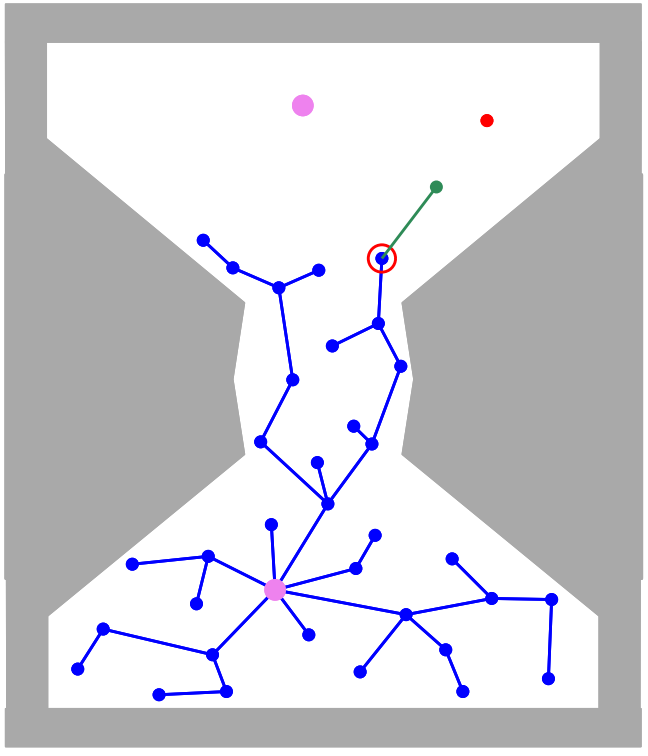
Rapidly-exploring random trees (RRT)

A **single**-query planner by LaValle; 1998: grows a tree $\mathcal{T}$, rooted in the start configuration S, that is embedded in the free space.

Algorithm works in n iterations. One iteration:

1. Sample a configuration x_{rnd}
2. Find the nearest vertex x_{near} of $\mathcal{T}$ to x_{rnd}
3. Generate a configuration x_{new} in the direction of x_{rnd} from x_{near}, such that $\overline{x_{\text{near}} x_{\text{new}}} \subset \mathcal{F}$
4. Update tree: $\mathcal{T} = \mathcal{T} \cup \{x_{\text{new}}, (x_{\text{near}}, x_{\text{new}})\}$

Every once in a while, set x_{rnd} to be the target vertex T; terminate if $x_{\text{new}} = T$.

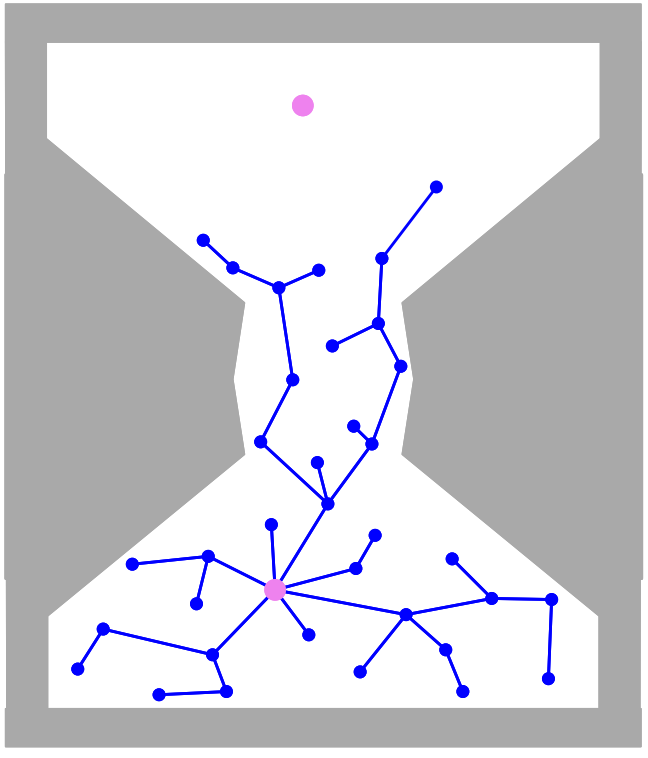


Rapidly-exploring random trees (RRT)

A **single**-query planner by LaValle; 1998: grows a tree $\mathcal{T}$, rooted in the start configuration S, that is embedded in the free space.

Algorithm works in n iterations. One iteration:

1. Sample a configuration x_{rnd}
2. Find the nearest vertex x_{near} of $\mathcal{T}$ to x_{rnd}
3. Generate a configuration x_{new} in the direction of x_{rnd} from x_{near}, such that $\overline{x_{\text{near}} x_{\text{new}}} \subset \mathcal{F}$
4. Update tree: $\mathcal{T} = \mathcal{T} \cup \{x_{\text{new}}, (x_{\text{near}}, x_{\text{new}})\}$



Every once in a while, set x_{rnd} to be the target vertex T; terminate if $x_{\text{new}} = T$.

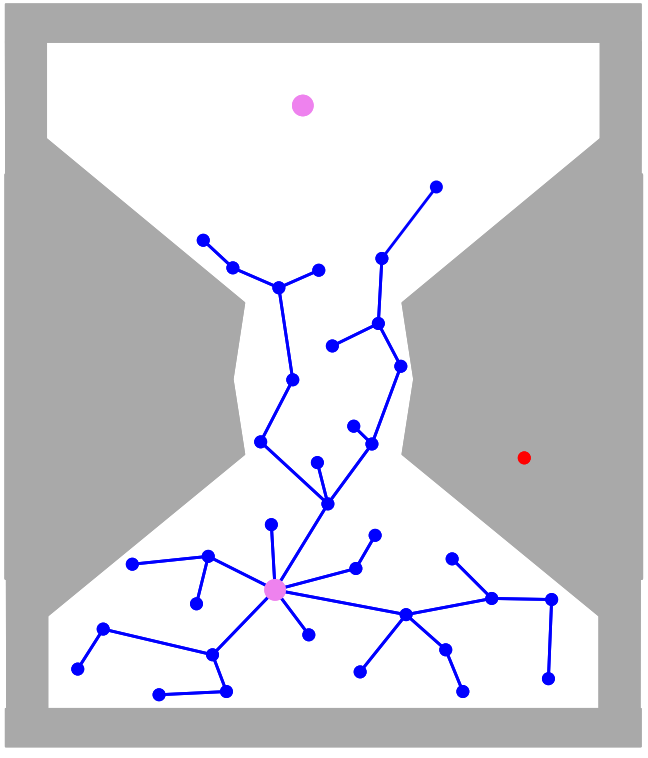
Rapidly-exploring random trees (RRT)

A **single**-query planner by LaValle; 1998: grows a tree $\mathcal{T}$, rooted in the start configuration S, that is embedded in the free space.

Algorithm works in n iterations. One iteration:

1. Sample a configuration x_{rnd}
2. Find the nearest vertex x_{near} of $\mathcal{T}$ to x_{rnd}
3. Generate a configuration x_{new} in the direction of x_{rnd} from x_{near}, such that $\overline{x_{\text{near}} x_{\text{new}}} \subset \mathcal{F}$
4. Update tree: $\mathcal{T} = \mathcal{T} \cup \{x_{\text{new}}, (x_{\text{near}}, x_{\text{new}})\}$

Every once in a while, set x_{rnd} to be the target vertex T; terminate if $x_{\text{new}} = T$.

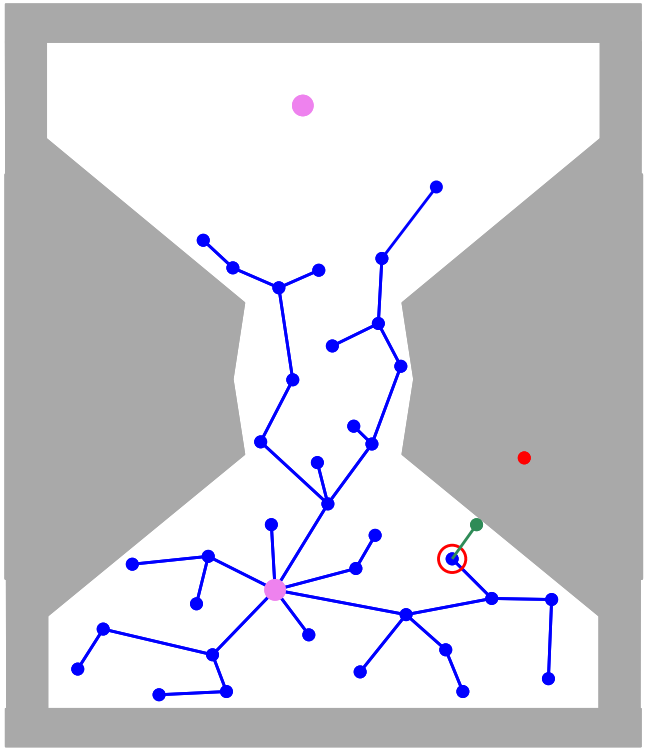


Rapidly-exploring random trees (RRT)

A **single**-query planner by LaValle; 1998: grows a tree $\mathcal{T}$, rooted in the start configuration S, that is embedded in the free space.

Algorithm works in n iterations. One iteration:

1. Sample a configuration x_{rnd}
2. Find the nearest vertex x_{near} of $\mathcal{T}$ to x_{rnd}
3. Generate a configuration x_{new} in the direction of x_{rnd} from x_{near}, such that $\overline{x_{\text{near}} x_{\text{new}}} \subset \mathcal{F}$
4. Update tree: $\mathcal{T} = \mathcal{T} \cup \{x_{\text{new}}, (x_{\text{near}}, x_{\text{new}})\}$



Every once in a while, set x_{rnd} to be the target vertex T; terminate if $x_{\text{new}} = T$.

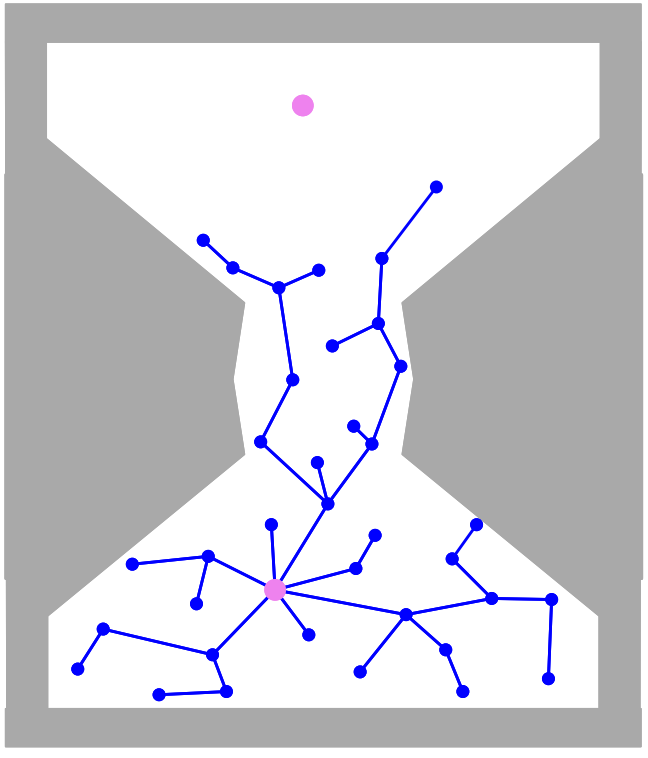
Rapidly-exploring random trees (RRT)

A **single**-query planner by LaValle; 1998: grows a tree $\mathcal{T}$, rooted in the start configuration S, that is embedded in the free space.

Algorithm works in n iterations. One iteration:

1. Sample a configuration x_{rnd}
2. Find the nearest vertex x_{near} of $\mathcal{T}$ to x_{rnd}
3. Generate a configuration x_{new} in the direction of x_{rnd} from x_{near}, such that $\overline{x_{\text{near}} x_{\text{new}}} \subset \mathcal{F}$
4. Update tree: $\mathcal{T} = \mathcal{T} \cup \{x_{\text{new}}, (x_{\text{near}}, x_{\text{new}})\}$

Every once in a while, set x_{rnd} to be the target vertex T; terminate if $x_{\text{new}} = T$.

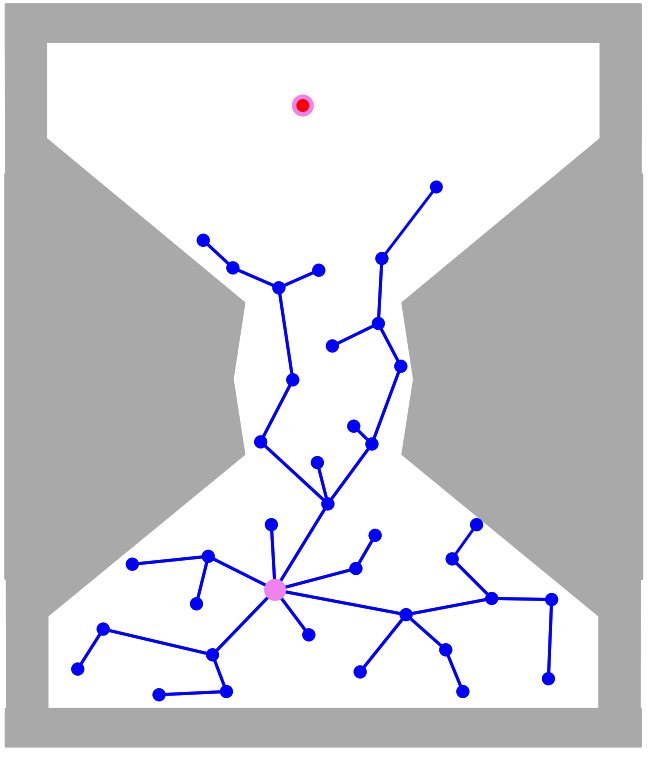


Rapidly-exploring random trees (RRT)

A **single**-query planner by LaValle; 1998: grows a tree $\mathcal{T}$, rooted in the start configuration S, that is embedded in the free space.

Algorithm works in n iterations. One iteration:

1. Sample a configuration x_{rnd}
2. Find the nearest vertex x_{near} of $\mathcal{T}$ to x_{rnd}
3. Generate a configuration x_{new} in the direction of x_{rnd} from x_{near}, such that $\overline{x_{\text{near}} x_{\text{new}}} \subset \mathcal{F}$
4. Update tree: $\mathcal{T} = \mathcal{T} \cup \{x_{\text{new}}, (x_{\text{near}}, x_{\text{new}})\}$



Every once in a while, set x_{rnd} to be the target vertex T; terminate if $x_{\text{new}} = T$.

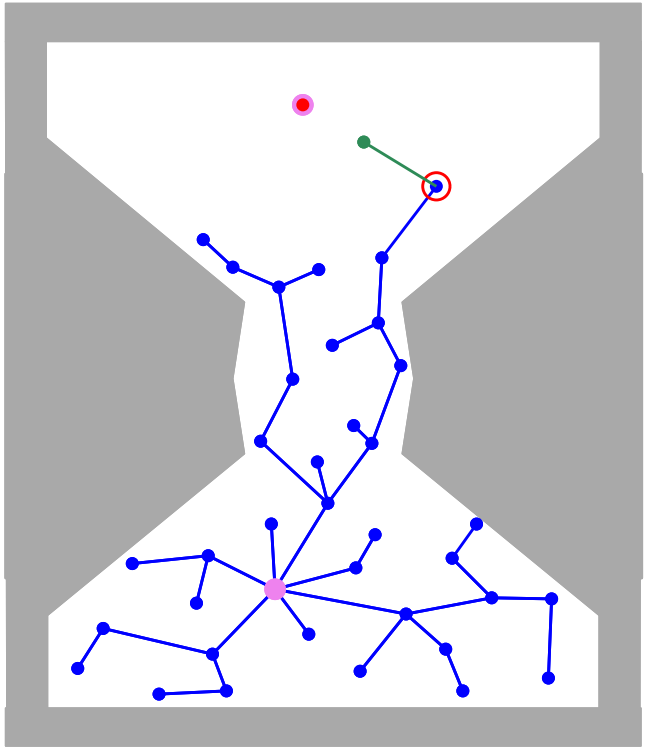
Rapidly-exploring random trees (RRT)

A **single**-query planner by LaValle; 1998: grows a tree $\mathcal{T}$, rooted in the start configuration S, that is embedded in the free space.

Algorithm works in n iterations. One iteration:

1. Sample a configuration x_{rnd}
2. Find the nearest vertex x_{near} of $\mathcal{T}$ to x_{rnd}
3. Generate a configuration x_{new} in the direction of x_{rnd} from x_{near}, such that $\overline{x_{\text{near}} x_{\text{new}}} \subset \mathcal{F}$
4. Update tree: $\mathcal{T} = \mathcal{T} \cup \{x_{\text{new}}, (x_{\text{near}}, x_{\text{new}})\}$

Every once in a while, set x_{rnd} to be the target vertex T; terminate if $x_{\text{new}} = T$.



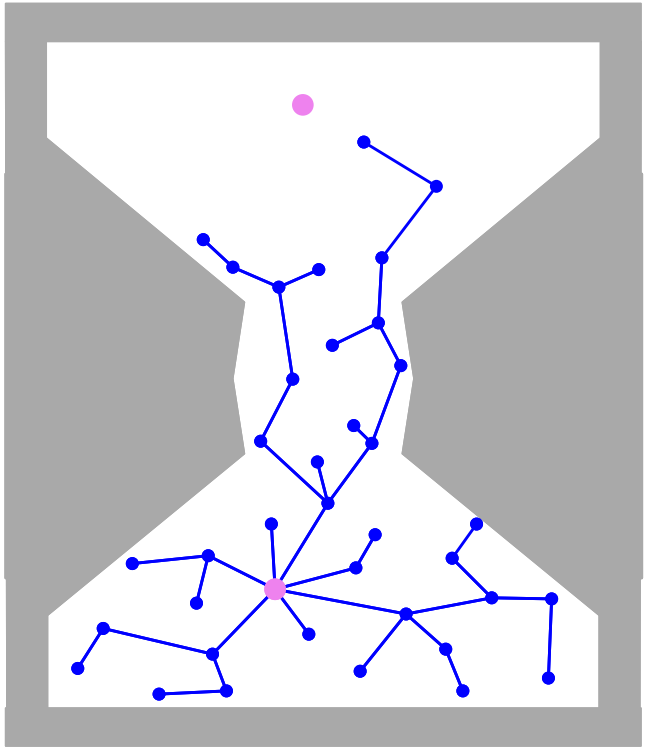
Rapidly-exploring random trees (RRT)

A **single**-query planner by LaValle; 1998: grows a tree $\mathcal{T}$, rooted in the start configuration S, that is embedded in the free space.

Algorithm works in n iterations. One iteration:

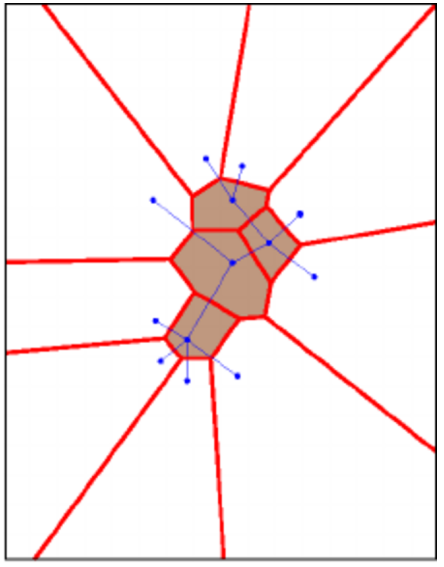
1. Sample a configuration x_{rnd}
2. Find the nearest vertex x_{near} of $\mathcal{T}$ to x_{rnd}
3. Generate a configuration x_{new} in the direction of x_{rnd} from x_{near}, such that $\overline{x_{\text{near}} x_{\text{new}}} \subset \mathcal{F}$
4. Update tree: $\mathcal{T} = \mathcal{T} \cup \{x_{\text{new}}, (x_{\text{near}}, x_{\text{new}})\}$

Every once in a while, set x_{rnd} to be the target vertex T; terminate if $x_{\text{new}} = T$.



Rapidly-exploring random trees (RRT)

- RRT is known to work quite well in practice
- Its power can be attributed to its Voronoi bias:
  - Consider a Voronoi diagram defined on the vertices of the tree
  - A Voronoi cell consists of all the points that are closer to vertex representative of the cell
  - Vertices on the frontier of the tree (closer to unexplored regions of the space) have larger Voronoi cells



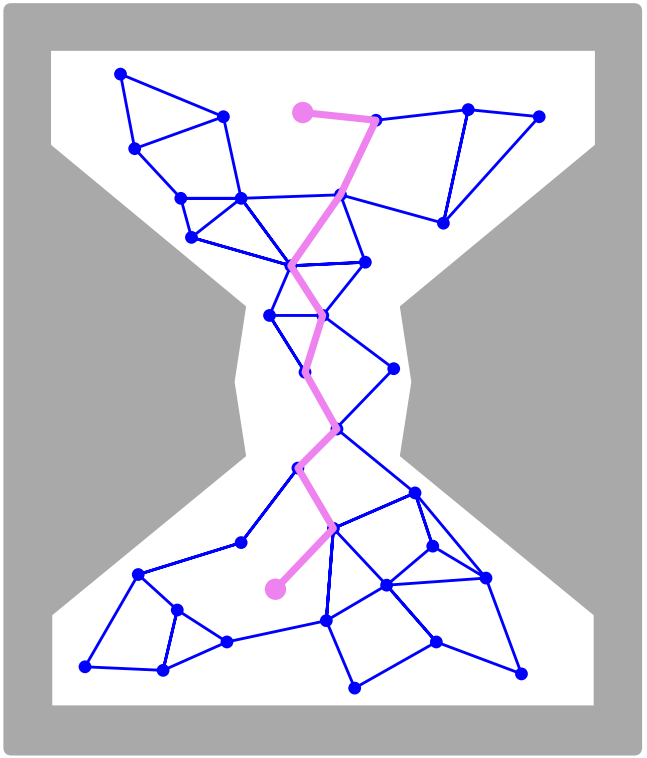
Voronoi bias of RRT.

Single-query version of PRM

- We wish to exploit the nice properties of PRM in a single-query fashion
- **Main issue:** PRM covers the entire space (even regions that are irrelevant to current query)
- **Observation:** The heaviest operation is local planning

Improvements:

- Let A* decide whether to perform local planning for given edge!
- Generate the entire graph, but collision test only A*-relevant edges
- Even better: generate the graph on the fly using NN-search

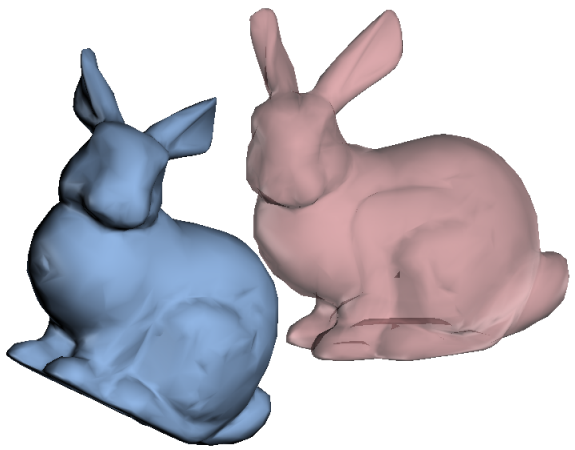


Sampling-based planning: intuition

Sampling-based planners transform the difficult global problem into a large set of **local** and **easy** problems.

A key ingredient is **collision detection**: decide whether a given configuration is in the free space.

Collision detection is relatively easy, since it can be solved in the workspace (2D or 3D).



Local planning (edge validation) is typically performed by dense sampling of path and collision detection.

Another key ingredient is **nearest-neighbor search**: given a query point find its nearest neighbor(s) within a set of points. Also well studied theoretically and practically.

Empirical and Conceptual Observations

Empirical

- PRM-like motion planning algorithms
  - For a give number of nodes n, they find "good" paths
  - ...however, require many costly collision checks
- RRT-like motion planning algorithms
  - Finds a feasible path quickly
  - ...however the quality of that path is, in general, **poor**
  - "traps" itself by disallowing new better paths to emerge [Frazzoli, ICRA '10] — RRT* (partially) offsets this behavior

Empirical and Conceptual Observations

Empirical

- PRM-like motion planning algorithms
  - For a give number of nodes n, they find "good" paths
  - ...however, require many costly collision checks
- RRT-like motion planning algorithms
  - Finds a feasible path quickly
  - ...however the quality of that path is, in general, **poor**
  - "traps" itself by disallowing new better paths to emerge [Frazzoli, ICRA '10] — RRT* (partially) offsets this behavior

Conceptual

- Asymptotic optimality studied under notion of *almost sure convergence*

Theoretical guarantees: probabilistic completeness

Question: how large should the number of samples n be?

Theoretical guarantees: probabilistic completeness

Question: how large should the number of samples n be?

We can say something about the asymptotic behavior:

Kavraki et al. 96: PRM, with $r = \text{const}$, will *eventually* (as $n \to \infty$) find a solution if one exists.

LaValle, 98: RRT will eventually find a solution if one exists.

* Unless stated otherwise, the configuration space is assumed to be the d-dimensional Euclidean unit hypercube $[0, 1]^d$, with $2 \leq d \leq \infty$.

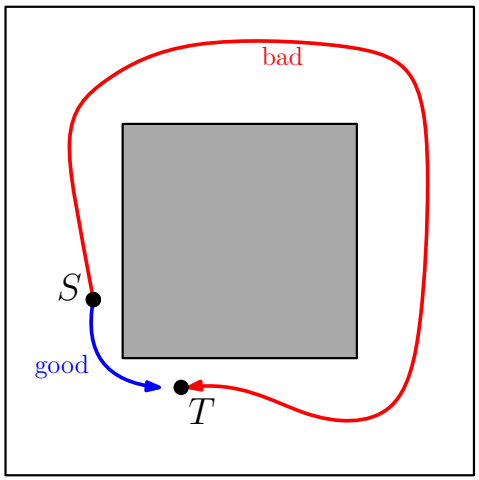
Theoretical guarantees: quality

Question: what can be said about the **quality** of the returned solution for PRM and RRT, in terms of length, energy, safety?

Several works attempted to address this question in an experimental setting (see, e.g., [Lien et al., 03], [Nieuwenhuisen and Overmars, 04], [Raveh et al., 11]).

Nechushtan et al. (2011) and Karaman and Frazzoli (2011) proved that RRT can produce arbitrarily-bad paths with non-negligible probability:

- RRT would prefer to take the long (red) way



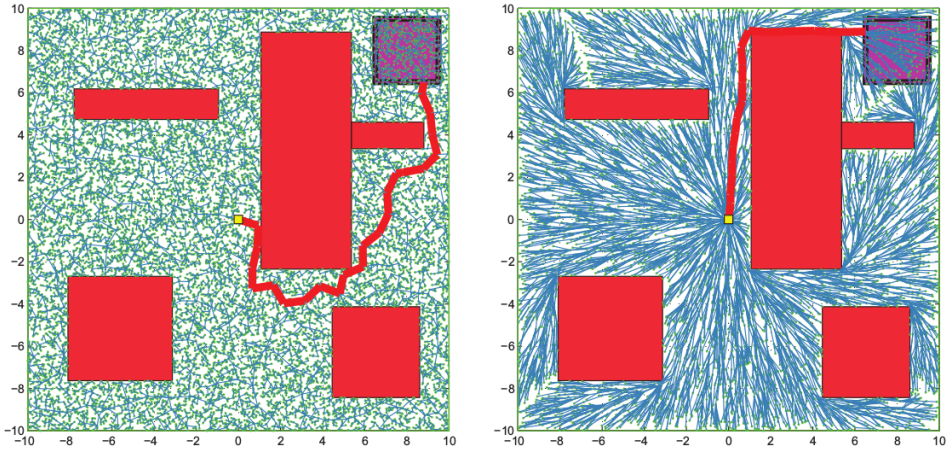
Theoretical guarantees: quality

In their breakthrough paper, Karaman and Frazzoli (2011) provide the first rigorous study of optimality in sampling-based planners.

Theorem: The **cost** of the solution returned by PRM converges, as $n \to \infty$, to the **optimum**, when $r_n = \gamma \left(\frac{\log n}{n} \right)^{1/d}$, where γ only depends on d.

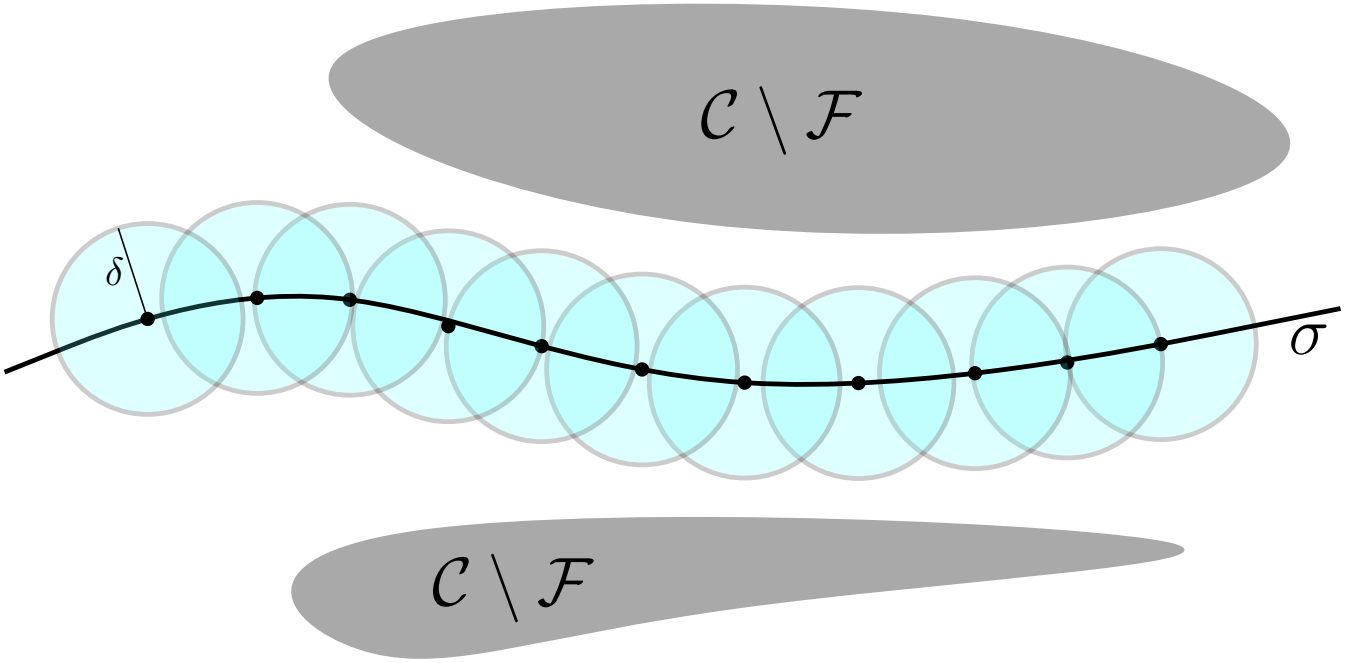
This statement was later refined and improved [Janson et al.; 15], [Solovey and Kleinbort.; 18]

KF11 also introduced an asymptotically-optimal variant of RRT called RRT* (right).



Theoretical guarantees: robustness

A crucial assumption for those theoretical guarantees to hold is *robustness*: there exits a solution with positive clearance:



Another limitation of sampling-based planners: cannot determine that a solution DOESN'T exists

Outline

Kinodynamic Planning

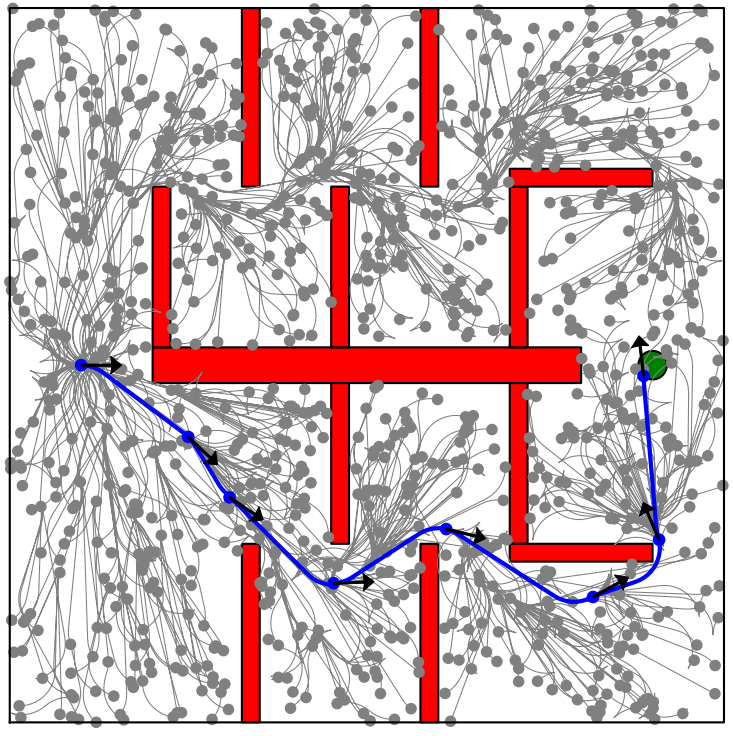
Kinodynamic motion planning problem: in addition to obstacle avoidance, feasible paths are subject to differential constraints

([LaValle & Kuffner, IJRR '01], [LaValle, '06], [Webb and van den Berg, ICRA'13], [Karaman and Frazzoli, ICRA '13], [Littlefield et al., WAFR '14])

- The robot operates in the state space $\mathcal{X}$
- To move the robot we apply controls $u \in \mathcal{U}$
- Motion needs to satisfy system's constraints:

$$\dot{x} = f(x, u), \quad \text{for } x \in \mathcal{X}, u \in \mathcal{U}$$

DFMT* Tree, N = 1000 Samples



Kinodynamic Planning

- We may extend the aforementioned algorithms (PRM, RRT, etc.) to the kinodynamic setting:
  1. Draw samples from $\mathcal{X}$
  2. Connect nearby samples $x, x' \in \mathcal{X}$ using an optimal path
- Step **2** corresponds to the **boundary value problem** (BVP): find a control function taking the robot from x to **precisely** x'.
- If BVP solver is available, then the theoretical guarantees extend to this case
  - some conditions may apply [Schmerling et al.; 2015]
- In general, BVP solvers (when available) are computationally heavy in practice

Kinodynamic Planning with Forward Propagation

RRT can be extended to work without a BVP solver:

1. Draw a random state and find its nearest neighbor x_{near}
2. Sample a random control $u \in \mathcal{U}$ and random duration t
3. **forward propagate** the control u for t time from x_{near}

Algorithm 2 RRT $(x_{\text{init}}, \mathcal{X}_{\text{goal}}, k, T_{\text{prop}}, \mathbb{U})$

$\mathcal{T}$.init(x_{init})
for $i = 1$ to k **do**
  $x_{\text{rand}} \leftarrow$ RANDOM_STATE()
  $x_{\text{near}} \leftarrow$ NEAREST_NEIGHBOR$(x_{\text{rand}}, \mathcal{T})$
  $t \leftarrow$ SAMPLE_DURATION$(0, T_{\text{prop}})$
  $u \leftarrow$ SAMPLE_CONTROL_INPUT$(\mathbb{U})$
  $x_{\text{new}} \leftarrow$ PROPAGATE(x_{near}, u, t)
  if COLLISION_FREE$(x_{\text{near}}, x_{\text{new}})$ **then**
    $\mathcal{T}$.add_vertex(x_{new})
    $\mathcal{T}$.add_edge$(x_{\text{near}}, x_{\text{new}})$
return $\mathcal{T}$

Kleinbort et al.: "Probabilistic completeness of RRT for geometric and kinodynamic planning with forward propagation." 2018.

Outline

Should Probabilistic Planners be Probabilistic?

Key question: would theoretical guarantees and practical performance still hold if these algorithms were to be derandomized, i.e., run on deterministic samples?

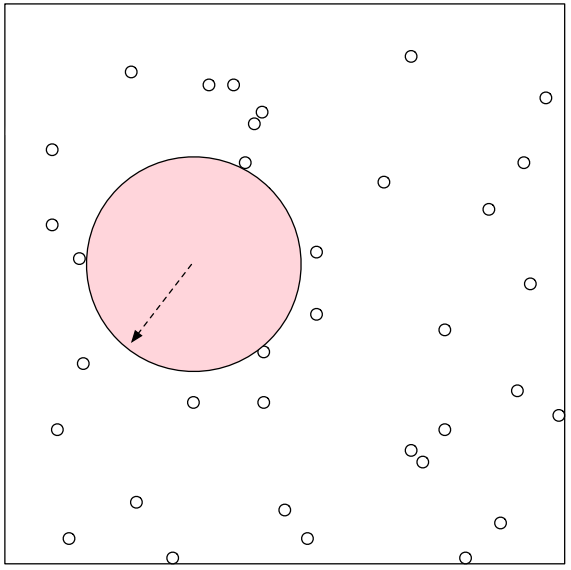
Important question as derandomization would:

- Ease certification process
- Ease use of offline computation
- Potentially simplify a number of operations (e.g., NN search)

Designing "Good" Sequences

ℓ_2-**dispersion**: For a finite set S of points contained in $\mathcal{X} \in \mathbb{R}^d$, ℓ_2-dispersion $D(S)$ defined as

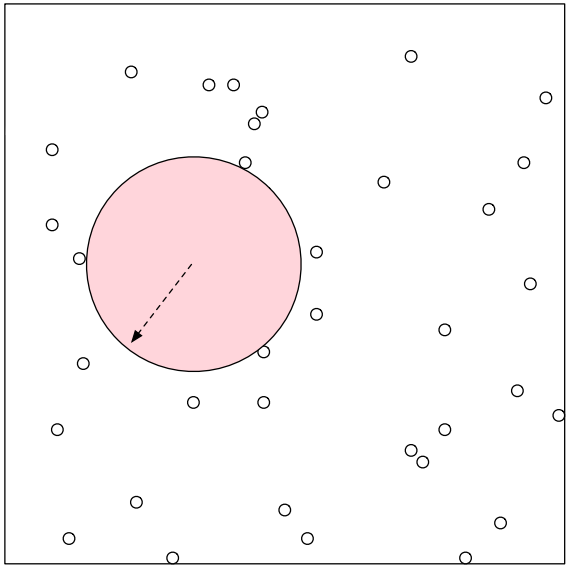
$$D(S) := \sup_{x \in \mathcal{X}} \min_{s \in S} \|s - x\|_2$$



Designing "Good" Sequences

ℓ_2-**dispersion**: For a finite set S of points contained in $\mathcal{X} \in \mathbb{R}^d$, ℓ_2-dispersion $D(S)$ defined as

$$D(S) := \sup_{x \in \mathcal{X}} \min_{s \in S} \|s - x\|_2$$



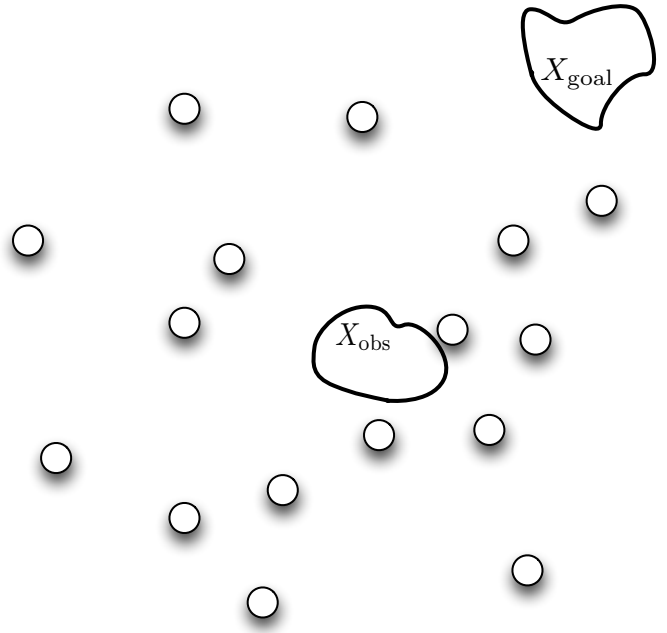
Key facts:

- There exist deterministic sequences with $D(S)$ of order $O(n^{-1/d})$, referred to as low-dispersion sequences
- Sequences minimizing ℓ_2-dispersion only known for $d = 2$

Optimality of Deterministic Planning

Algorithm gPRM

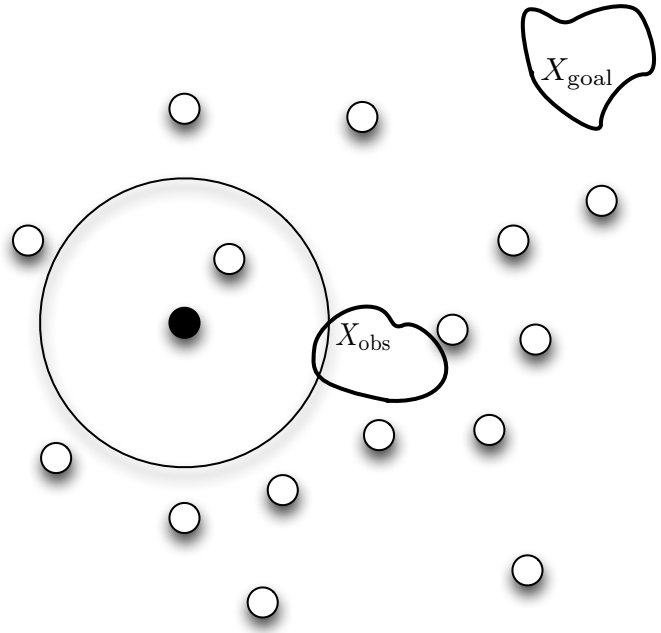
```
1  V ← {x_init} ∪ SampleFree(n); E ← ∅
2  for all v ∈ V do
3      X_near ← Near(V\{v}, v, r_n)
4      for x ∈ X_near do
5          if CollisionFree(v, x) then
6              E ← E ∪ {(v, x)} ∪ {(x, v)}
7          end if
8      end for
9  end for
10 return ShortestPath(x_init, V, E)
```



Optimality of Deterministic Planning

Algorithm gPRM

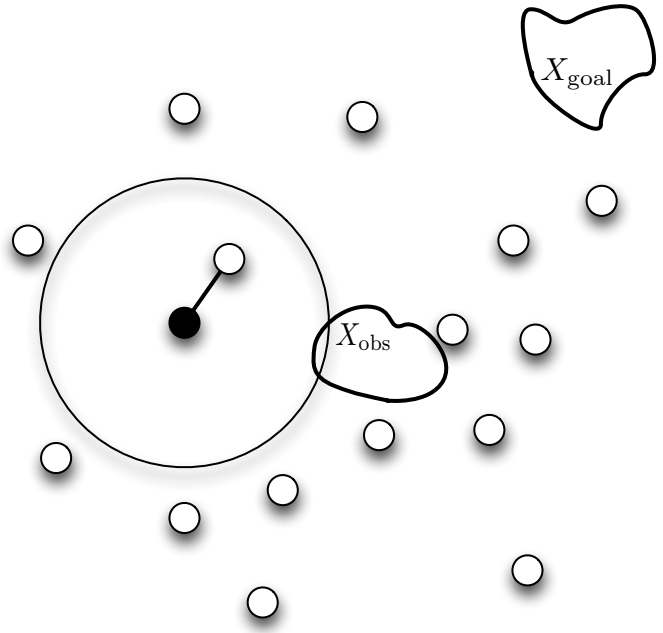
```
1  V ← {x_init} ∪ SampleFree(n); E ← ∅
2  for all v ∈ V do
3      X_near ← Near(V\{v}, v, r_n)
4      for x ∈ X_near do
5          if CollisionFree(v, x) then
6              E ← E ∪ {(v, x)} ∪ {(x, v)}
7          end if
8      end for
9  end for
10 return ShortestPath(x_init, V, E)
```



Optimality of Deterministic Planning

Algorithm gPRM

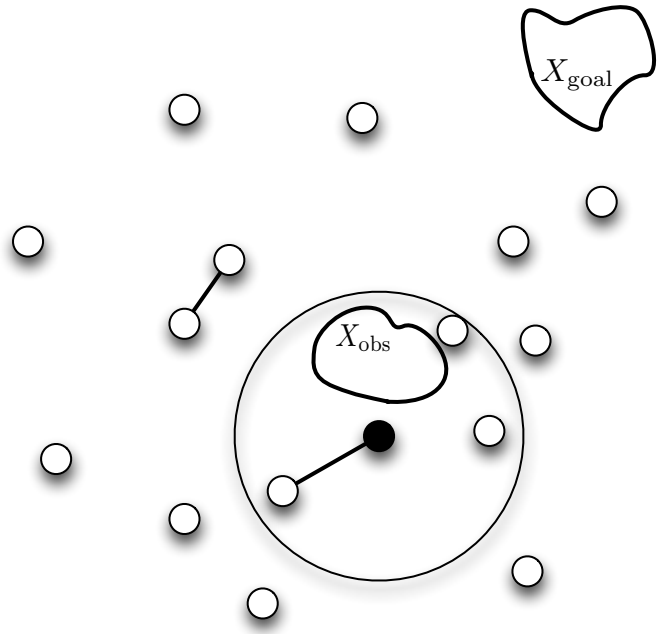
```
1  V ← {x_init} ∪ SampleFree(n); E ← ∅
2  for all v ∈ V do
3      X_near ← Near(V\{v}, v, r_n)
4      for x ∈ X_near do
5          if CollisionFree(v, x) then
6              E ← E ∪ {(v, x)} ∪ {(x, v)}
7          end if
8      end for
9  end for
10 return ShortestPath(x_init, V, E)
```



Optimality of Deterministic Planning

Algorithm gPRM

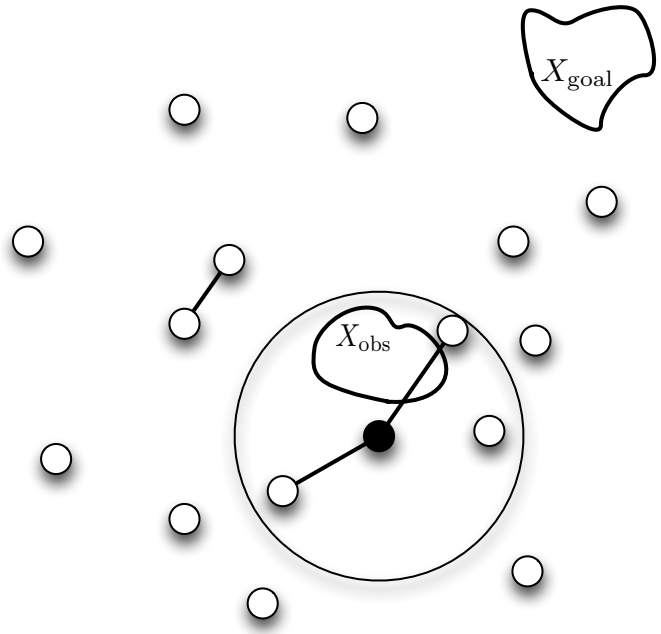
```
1  V ← {x_init} ∪ SampleFree(n); E ← ∅
2  for all v ∈ V do
3      X_near ← Near(V\{v}, v, r_n)
4      for x ∈ X_near do
5          if CollisionFree(v, x) then
6              E ← E ∪ {(v, x)} ∪ {(x, v)}
7          end if
8      end for
9  end for
10 return ShortestPath(x_init, V, E)
```



Optimality of Deterministic Planning

Algorithm gPRM

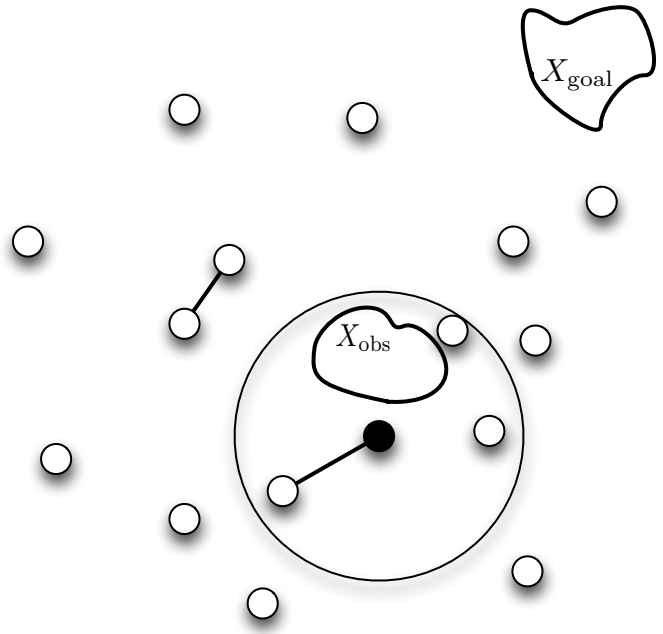
```
1  V ← {x_init} ∪ SampleFree(n); E ← ∅
2  for all v ∈ V do
3      X_near ← Near(V\{v}, v, r_n)
4      for x ∈ X_near do
5          if CollisionFree(v, x) then
6              E ← E ∪ {(v, x)} ∪ {(x, v)}
7          end if
8      end for
9  end for
10 return ShortestPath(x_init, V, E)
```



Optimality of Deterministic Planning

Algorithm gPRM

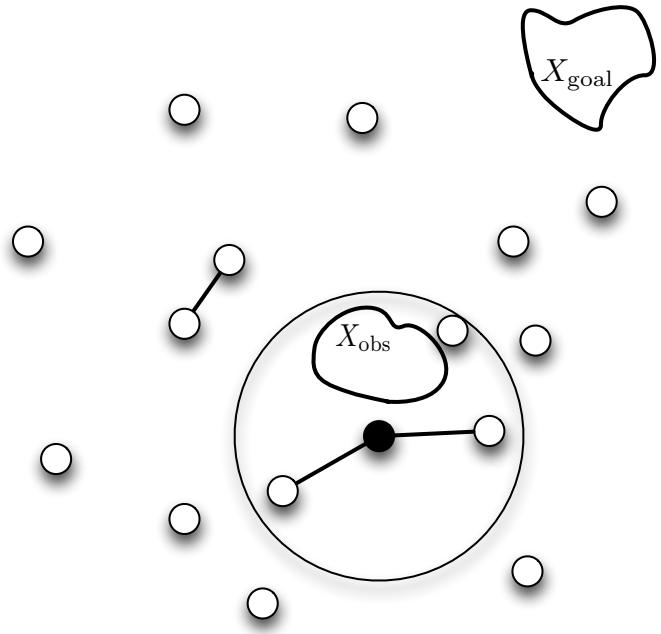
```
1  V ← {x_init} ∪ SampleFree(n); E ← ∅
2  for all v ∈ V do
3      X_near ← Near(V\{v}, v, r_n)
4      for x ∈ X_near do
5          if CollisionFree(v, x) then
6              E ← E ∪ {(v, x)} ∪ {(x, v)}
7          end if
8      end for
9  end for
10 return ShortestPath(x_init, V, E)
```



Optimality of Deterministic Planning

Algorithm gPRM

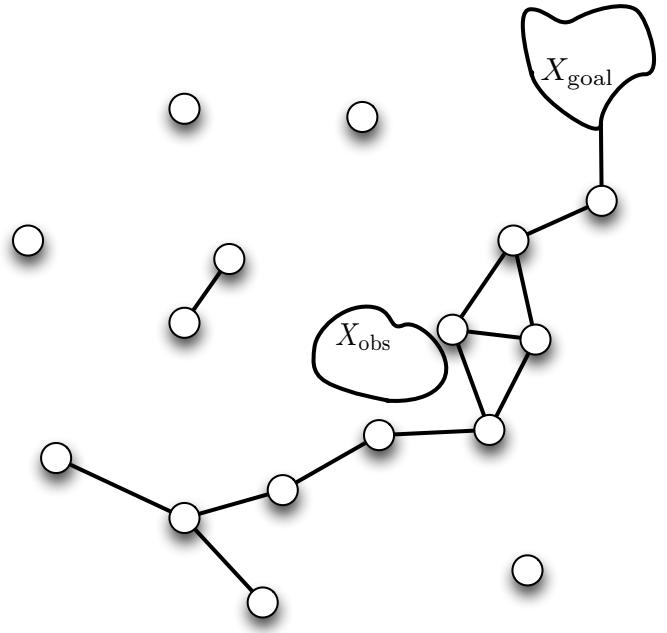
```
1  V ← {x_init} ∪ SampleFree(n); E ← ∅
2  for all v ∈ V do
3      X_near ← Near(V\{v}, v, r_n)
4      for x ∈ X_near do
5          if CollisionFree(v, x) then
6              E ← E ∪ {(v, x)} ∪ {(x, v)}
7          end if
8      end for
9  end for
10 return ShortestPath(x_init, V, E)
```



Optimality of Deterministic Planning

Algorithm gPRM

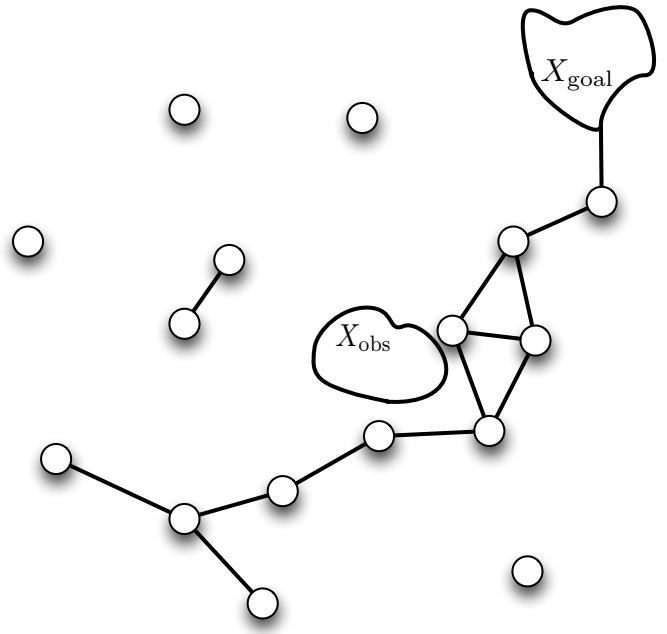
```
1  V ← {x_init} ∪ SampleFree(n); E ← ∅
2  for all v ∈ V do
3      X_near ← Near(V\{v}, v, r_n)
4      for x ∈ X_near do
5          if CollisionFree(v, x) then
6              E ← E ∪ {(v, x)} ∪ {(x, v)}
7          end if
8      end for
9  end for
10 return ShortestPath(x_init, V, E)
```



Optimality of Deterministic Planning

Algorithm gPRM

1 $V \leftarrow \{x_{\text{init}}\} \cup \texttt{SampleFree}(n)$; $E \leftarrow \emptyset$
2 **for all** $v \in V$ **do**
3 $\quad X_{\text{near}} \leftarrow \texttt{Near}(V \backslash \{v\}, v, r_n)$
4 $\quad$ **for** $x \in X_{\text{near}}$ **do**
5 $\quad\quad$ **if** $\texttt{CollisionFree}(v, x)$ **then**
6 $\quad\quad\quad E \leftarrow E \cup \{(v, x)\} \cup \{(x, v)\}$
7 $\quad\quad$ **end if**
8 $\quad$ **end for**
9 **end for**
10 **return** $\texttt{ShortestPath}(x_{\text{init}}, V, E)$



Optimality: Let c_n denote the arc length of the path returned by gPRM with n samples. Then, if

1. sample set S has dispersion $D(S) \leq \gamma\, n^{-1/d}$ for some $\gamma > 0$,
2. $n^{1/d}\, r_n \to \infty$,

then $\lim_{n\to\infty} c_n = c^*$, where c^* is the cost of an optimal path.

Insights

Remarks:

- Deterministic convergence guarantee
- $r_n \in \omega\left((1/n)^{1/d}\right)$ vs. $r_n \in \Omega\left((\log(n)/n)^{1/d}\right)$ neighborhood size
- If gPRM reports failure, then (i) no solution exists or (ii) solution goes through "narrow" (w.r.t. dispersion) corridors

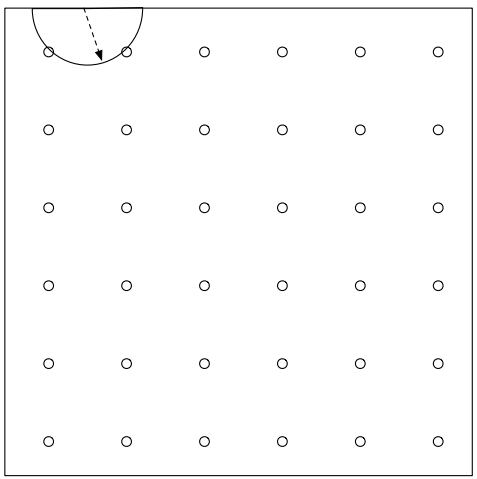
Insights

Remarks:

- Deterministic convergence guarantee
- $r_n \in \omega\big((1/n)^{1/d}\big)$ vs. $r_n \in \Omega\big((\log(n)/n)^{1/d}\big)$ neighborhood size
- If gPRM reports failure, then (i) no solution exists or (ii) solution goes through "narrow" (w.r.t. dispersion) corridors

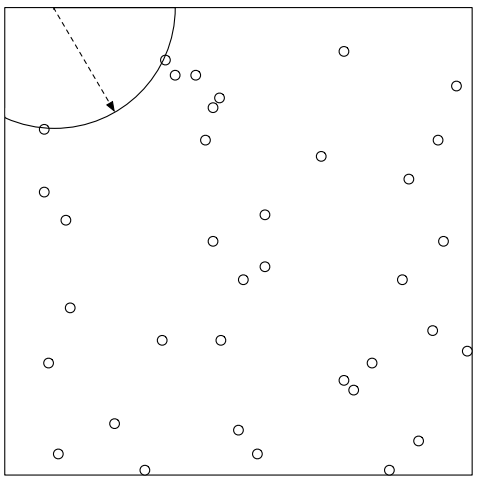
Key insight: low-dispersion, deterministic sets enjoy an improvement by a factor $\log(n)^{1/d}$ in terms of ℓ_2-dispersion!

low-dispersion point set



dispersion $\sim O((1/n)^{1/d})$

random i.i.d. point set



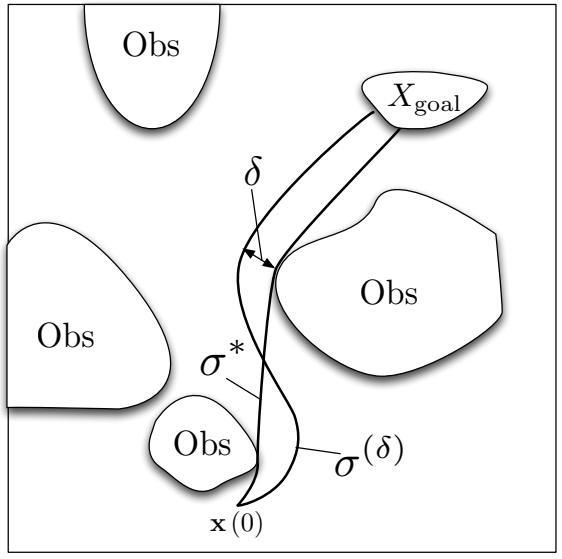
dispersion $\sim O((\log(n)/n)^{1/d})$

Convergence Rate and Complexity

Convergence rate: Assuming $r_n > 2D(S)$

$$c_n \leq \left(1 + \frac{2D(S)}{r_n - 2D(S)}\right) c^{(\delta)},$$

where $c^{(\delta)}$ cost of shortest path with strong δ-clearance.

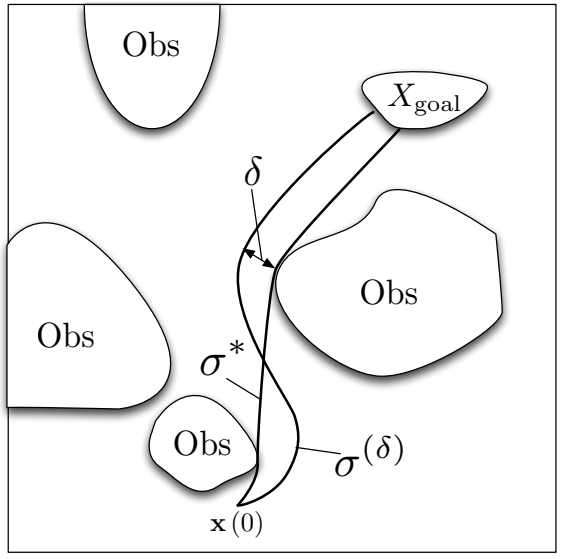


Convergence Rate and Complexity

Convergence rate: Assuming $r_n > 2D(S)$

$$c_n \leq \left(1 + \frac{2D(S)}{r_n - 2D(S)}\right) c^{(\delta)},$$

where $c^{(\delta)}$ cost of shortest path with strong δ-clearance.

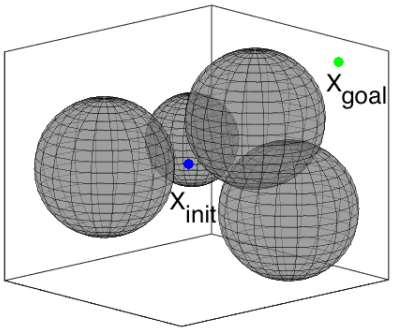


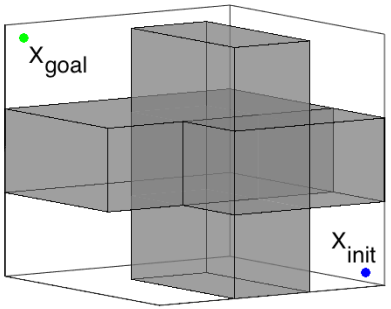
Complexity: gPRM, when run on a low-dispersion lattice, has computational and space complexity $\omega(n)$.

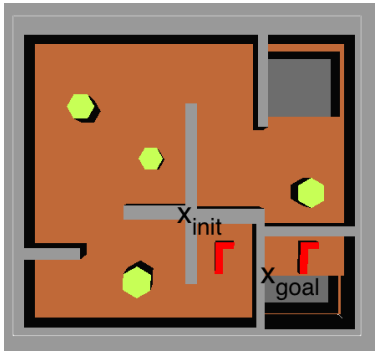
Notes:

- $O(n)$ is the theoretical lower bound
- Complexity with random sampling is $O(n \log(n))$
- Proof relies on revised implementation of Dijkstra's algorithm
- Result extends to other batch algorithms, e.g., FMT*

Numerical Results







		Halton			Lattice		
Dim	Obstacles	90% Success	Medium	High	90% Success	Medium	High
2	Rectangular	38%	118%	80%	15%	56%	80%
3	Rectangular	36%	88%	94%	19%	80%	87%
2	Rect Maze	13%	98%	99%	13%	100%	99%
2	Sphere	16%	93%	99%	7%	93%	99%
3	Sphere	36%	97%	100%	8%	97%	99%
4	Sphere	100%	97%	97%	100%	97%	100%
2	Recursive Maze	33%	100%	100%	18%	100%	100%
3	Recursive Maze	22%	95%	99%	22%	96%	98%
4	Recursive Maze	56%	95%	98%	56%	100%	100%
5	Recursive Maze	45%	97%	96%	60%	95%	96%
6	Recursive Maze	56%	95%	97%	75%	94%	96%
8	Recursive Maze	56%	98%	99%	75%	99%	99%
8	Chain	67%	112%	91%	7%	76%	87%
3	SE(2)	81%	96%	100%	81%	101%	101%
6	SE(3)	32%	96%	93%	42%	94%	95%

http://arxiv.org/abs/1505.00023

Summary

1. Asymptotic optimality can be achieved with deterministic sequences and with a smaller connection radius
2. Deterministic convergence rates: instrumental to the certification of sampling-based planners
3. Computational and space complexity: under some assumptions, arbitrarily close to theoretical lower bound
4. Deterministic sequences appear to provide superior performance

Summary

1. Asymptotic optimality can be achieved with deterministic sequences and with a smaller connection radius
2. Deterministic convergence rates: instrumental to the certification of sampling-based planners
3. Computational and space complexity: under some assumptions, arbitrarily close to theoretical lower bound
4. Deterministic sequences appear to provide superior performance

Do not randomize!

Summary

1. Asymptotic optimality can be achieved with deterministic sequences and with a smaller connection radius
2. Deterministic convergence rates: instrumental to the certification of sampling-based planners
3. Computational and space complexity: under some assumptions, arbitrarily close to theoretical lower bound
4. Deterministic sequences appear to provide superior performance

Do not randomize!

Well, not quite:

- Same $O(n^{-1/d})$ can be achieved with *non-i.i.d.* random sequences
- Such sequences enjoy same *deterministic* performance guarantees
- Might hold some benefits (e.g., avoiding axis-alignment issues)

Summary

1. Asymptotic optimality can be achieved with deterministic sequences and with a smaller connection radius
2. Deterministic convergence rates: instrumental to the certification of sampling-based planners
3. Computational and space complexity: under some assumptions, arbitrarily close to theoretical lower bound
4. Deterministic sequences appear to provide superior performance

Do not randomize!

Well, not quite:

- Same $O(n^{-1/d})$ can be achieved with *non-i.i.d.* random sequences
- Such sequences enjoy same *deterministic* performance guarantees
- Might hold some benefits (e.g., avoiding axis-alignment issues)

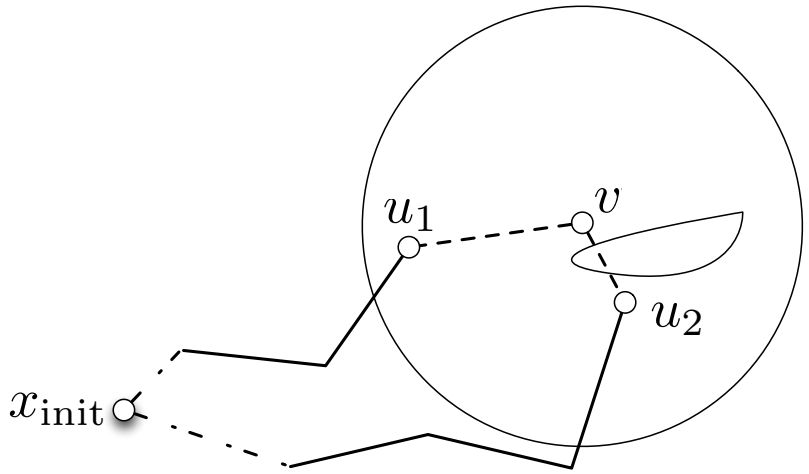
Deterministic guarantees are really a matter of i.i.d. sampling versus non-i.i.d., low-dispersion sampling

Fast Marching Tree Algorithm (FMT*)

Key idea: run dynamic programming on sampled nodes, skipping any step in which the attempted connection causes a collision

- tailored to disk-connected graphs
- graph construction and graph search performed concurrently
- lazy DP operator:

$$c(v) = \min_{u:\|u-v\|<r_n} \underbrace{\texttt{Cost}(u,v)}_{\text{“assumes” no collisions}} + c(u)$$

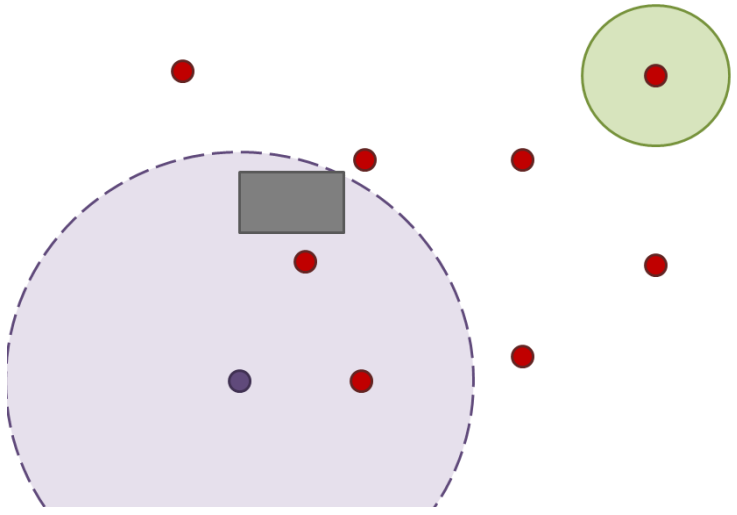


L. Janson, E. Schmerling, A. Clark and M. Pavone. Fast Marching Trees: a Fast Marching Sampling-Based Method for Optimal Motion Planning in Many Dimensions.

Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

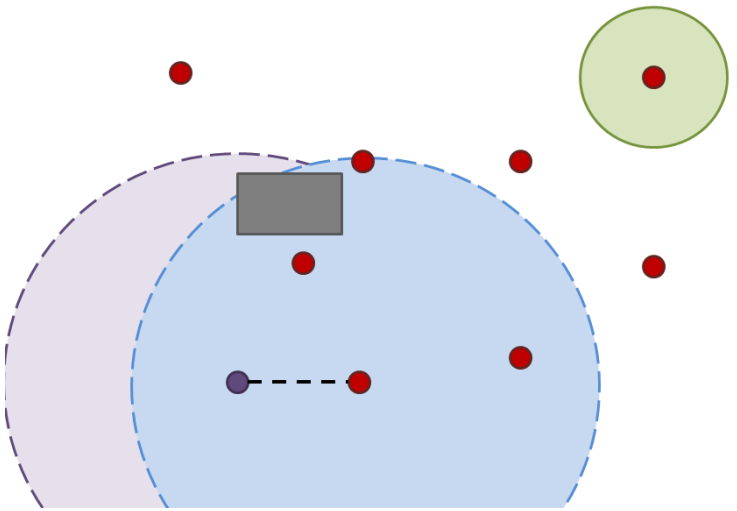
1: Sample n nodes
2: Find lowest-cost "branch" node in the tree, z
3: For each of its unexplored neighbors, x:
4: Find optimal local connection of x to tree
5: If that connection is collision-free:
6: Add edge to tree
7: Remove x from unexplored set
8: Add x to "branch" set
9: Remove z from "branch" set
10: Find new lowest-cost "branch" node and repeat until either:
11: "Branch" set is empty: failure
12: Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

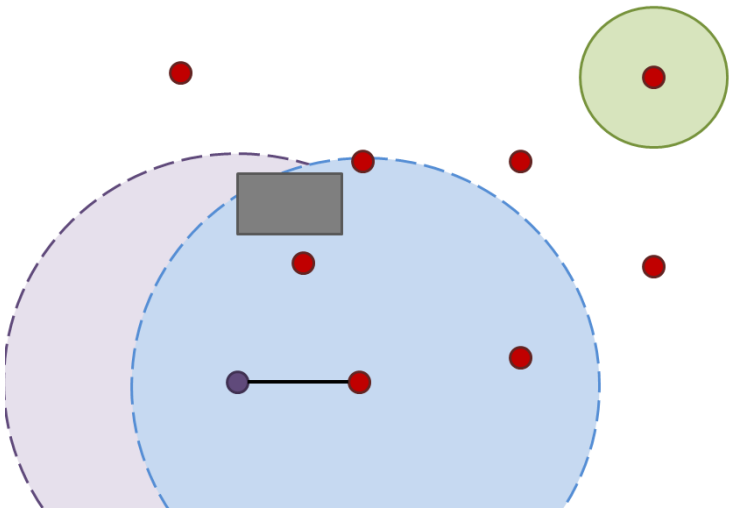
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

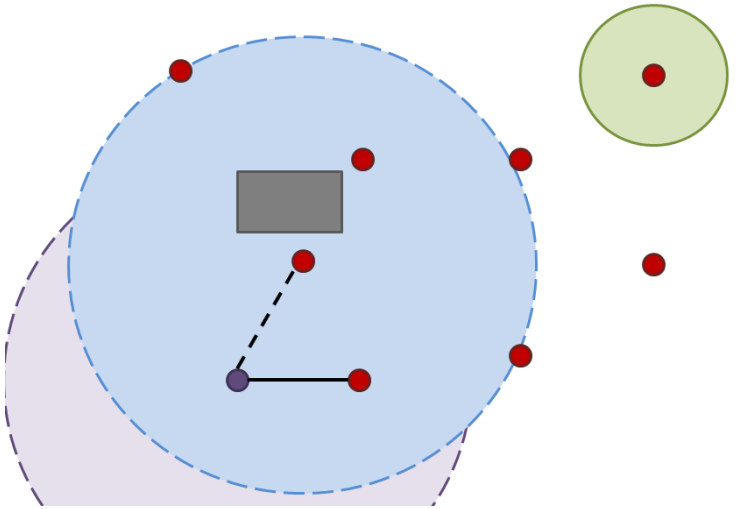
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

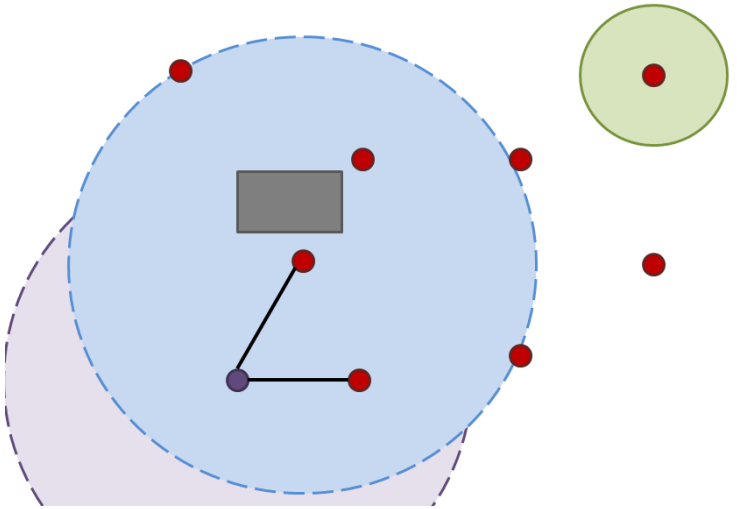
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

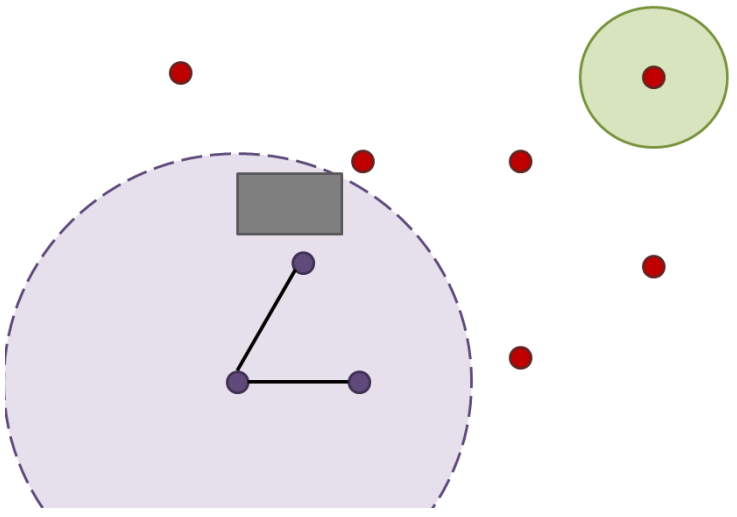
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

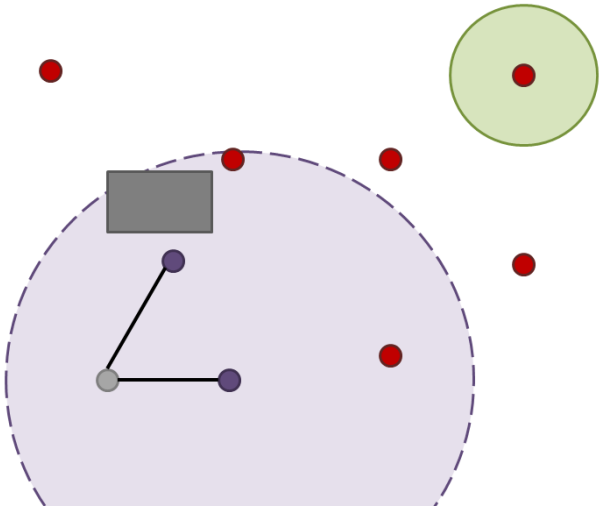
1: Sample n nodes
2: Find lowest-cost "branch" node in the tree, z
3: For each of its unexplored neighbors, x:
4: Find optimal local connection of x to tree
5: If that connection is collision-free:
6: Add edge to tree
7: Remove x from unexplored set
8: Add x to "branch" set
9: Remove z from "branch" set
10: Find new lowest-cost "branch" node and repeat until either:
11: "Branch" set is empty: failure
12: Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

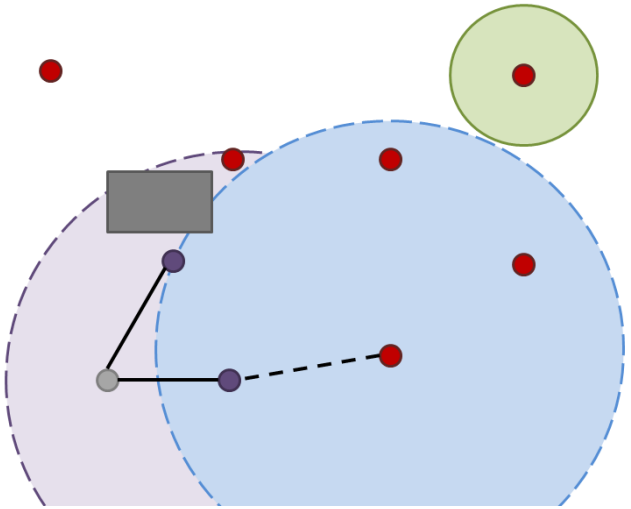
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

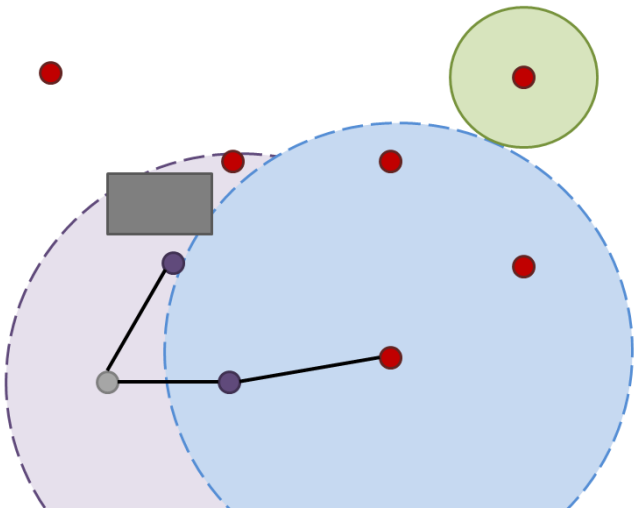
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

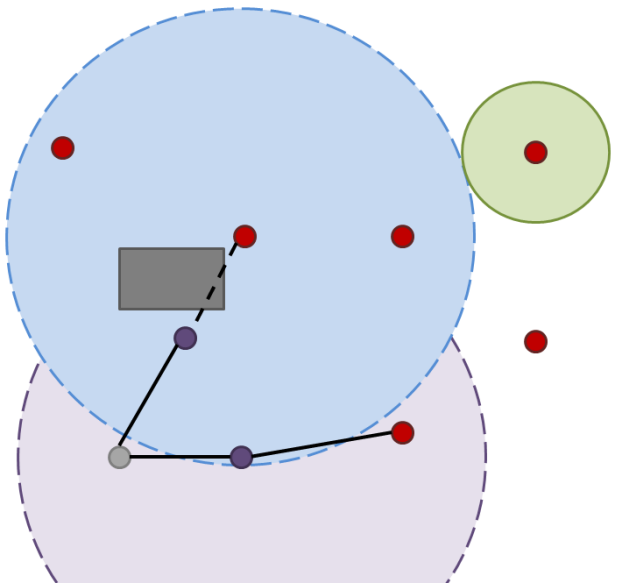
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

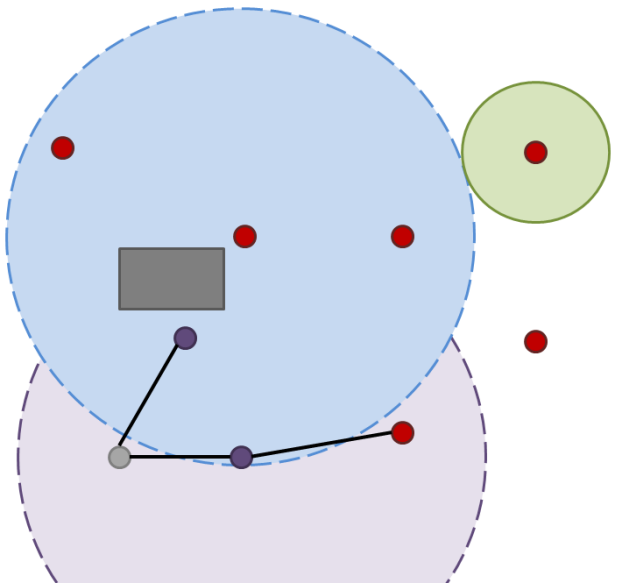
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

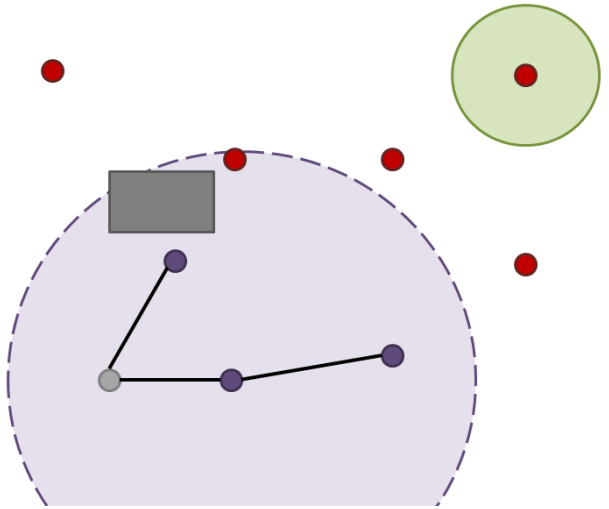
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

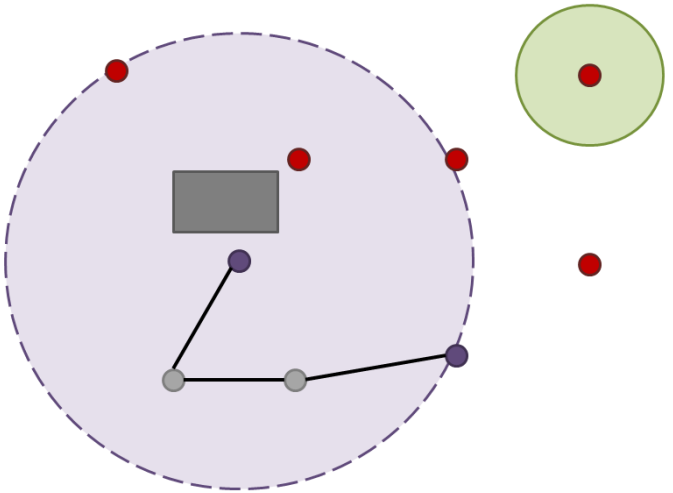
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

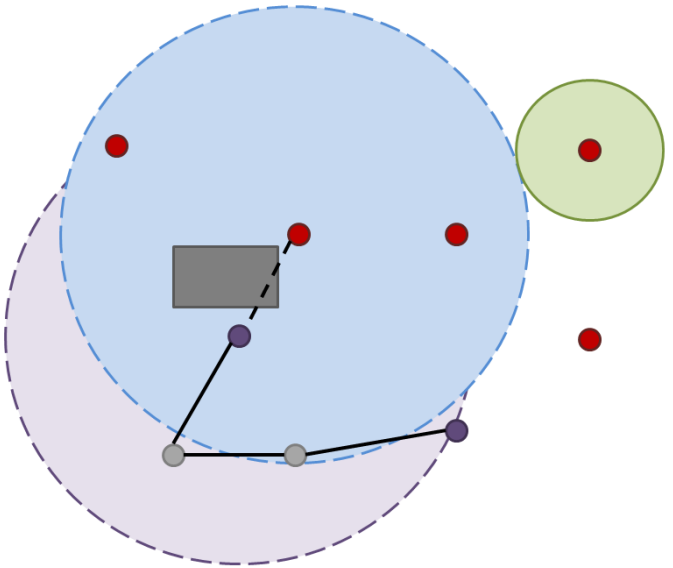
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

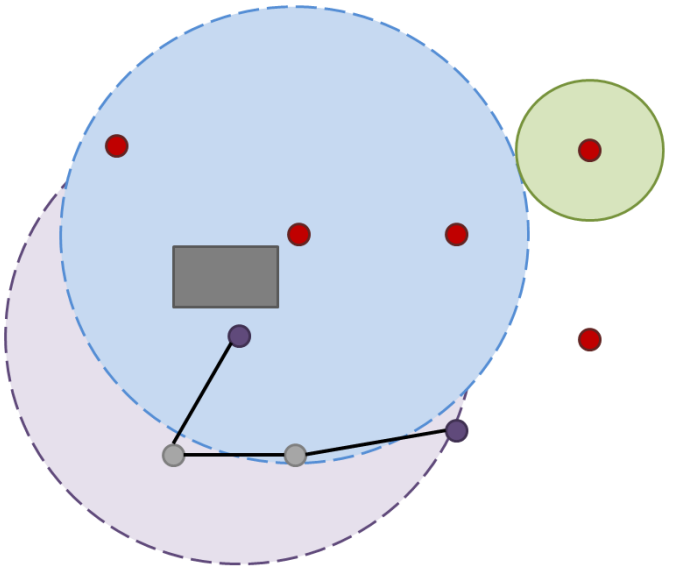
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

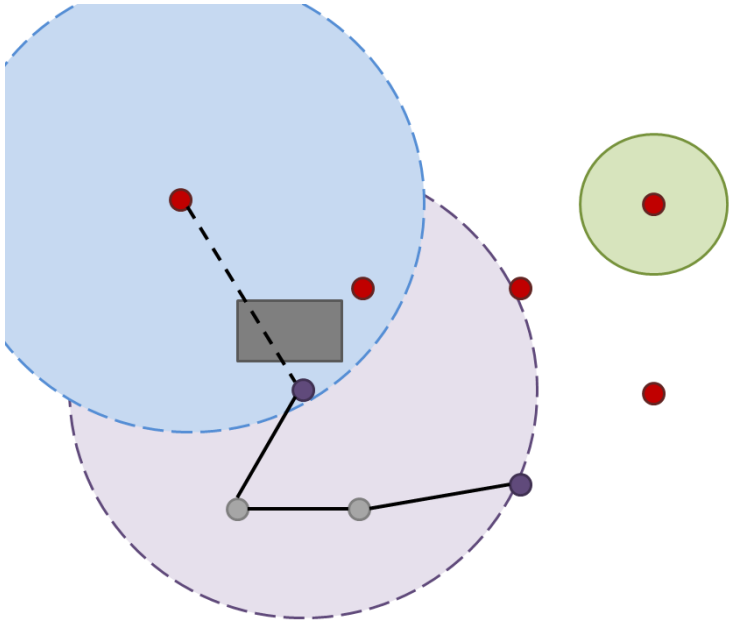
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

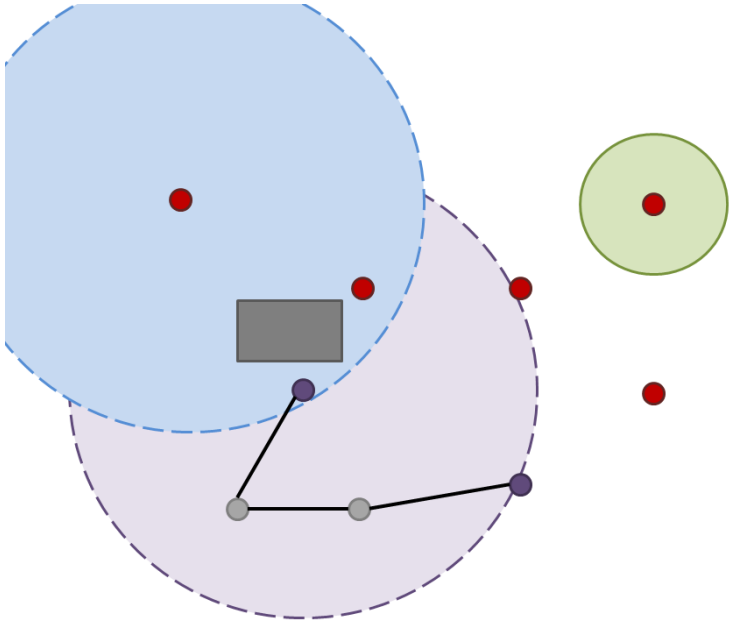
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

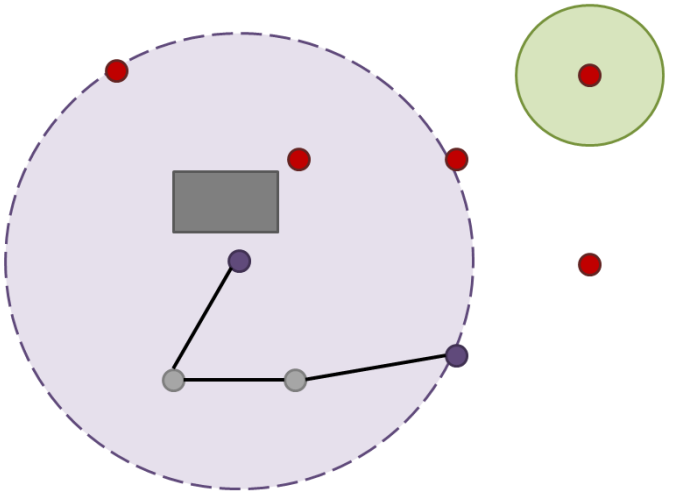
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

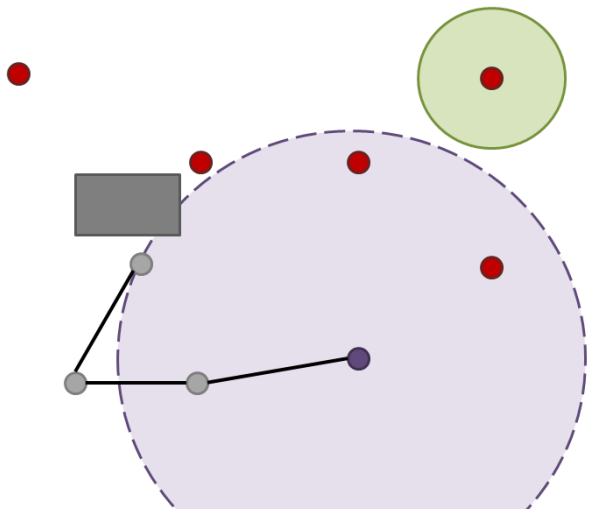
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

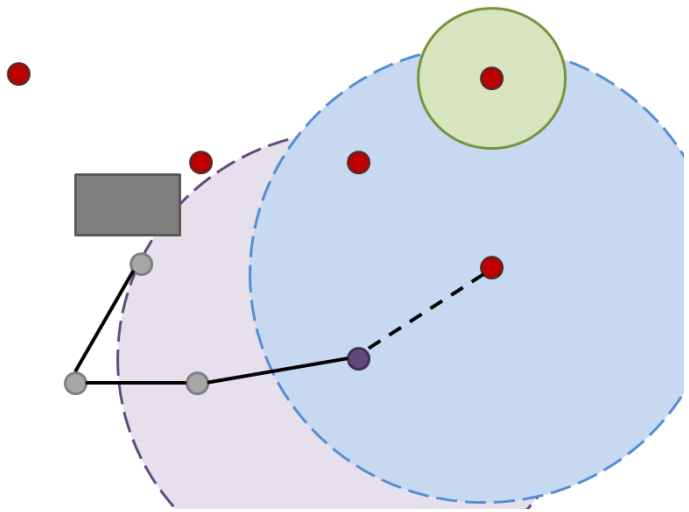
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

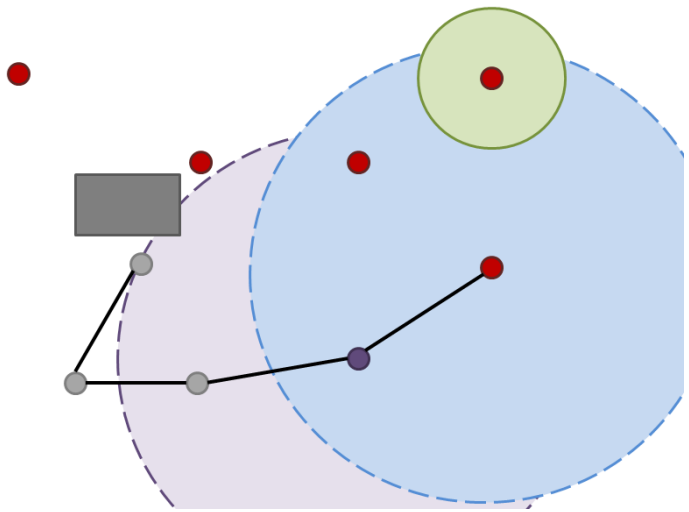
1: Sample n nodes
2: Find lowest-cost "branch" node in the tree, z
3: For each of its unexplored neighbors, x:
4: Find optimal local connection of x to tree
5: If that connection is collision-free:
6: Add edge to tree
7: Remove x from unexplored set
8: Add x to "branch" set
9: Remove z from "branch" set
10: Find new lowest-cost "branch" node and repeat until either:
11: "Branch" set is empty: failure
12: Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

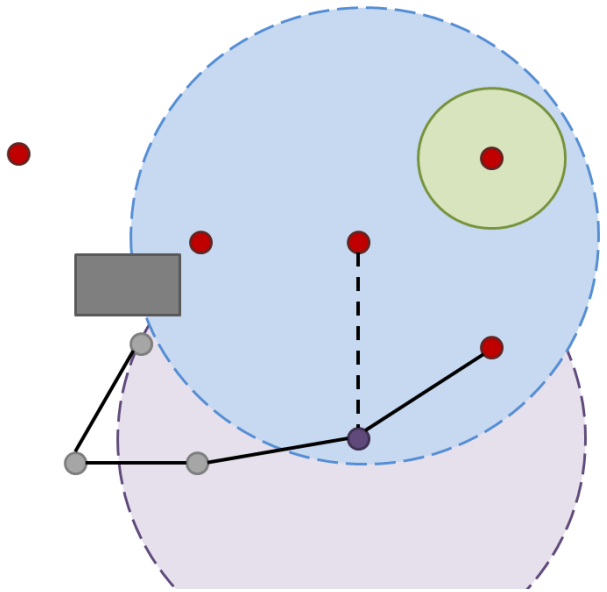
1: Sample n nodes
2: Find lowest-cost "branch" node in the tree, z
3: For each of its unexplored neighbors, x:
4: Find optimal local connection of x to tree
5: If that connection is collision-free:
6: Add edge to tree
7: Remove x from unexplored set
8: Add x to "branch" set
9: Remove z from "branch" set
10: Find new lowest-cost "branch" node and repeat until either:
11: "Branch" set is empty: failure
12: Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

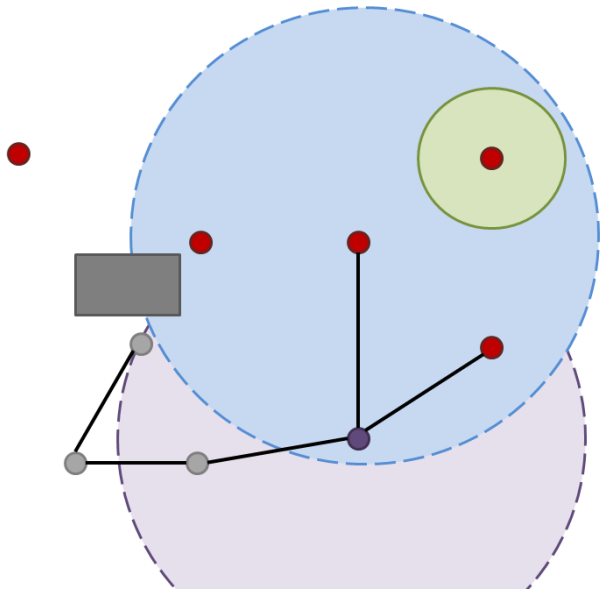
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

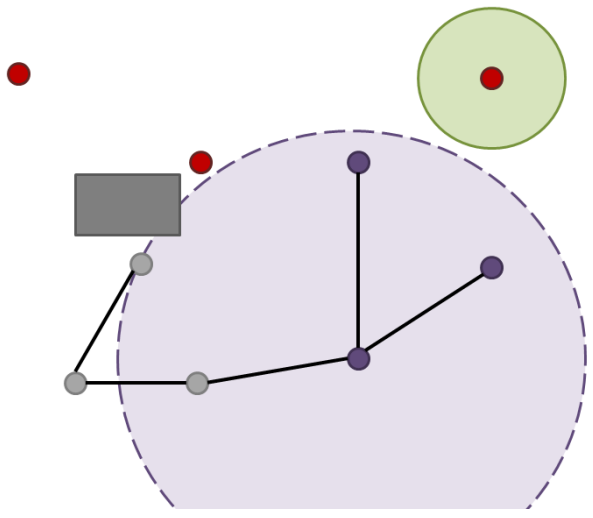
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

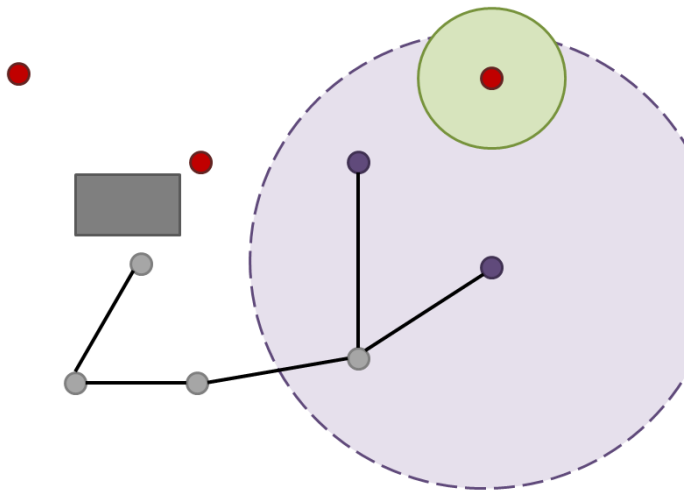
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

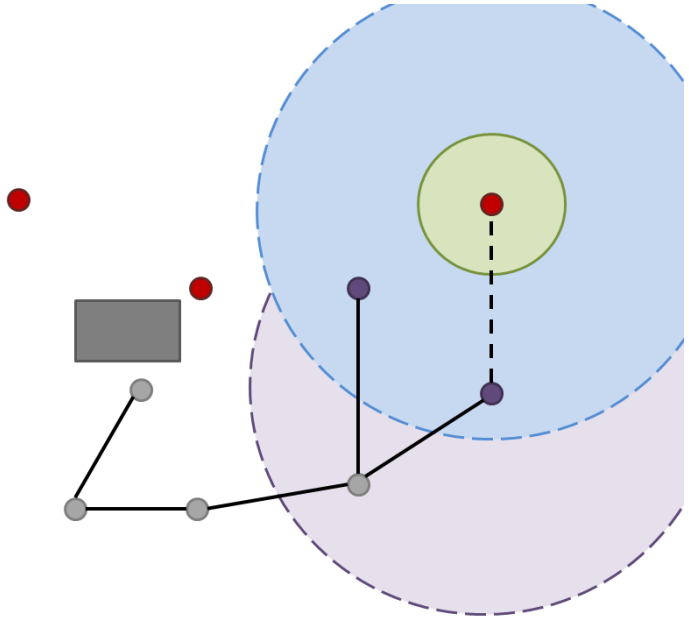
1: Sample n nodes
2: Find lowest-cost "branch" node in the tree, z
3: For each of its unexplored neighbors, x:
4: Find optimal local connection of x to tree
5: If that connection is collision-free:
6: Add edge to tree
7: Remove x from unexplored set
8: Add x to "branch" set
9: Remove z from "branch" set
10: Find new lowest-cost "branch" node and repeat until either:
11: "Branch" set is empty: failure
12: Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

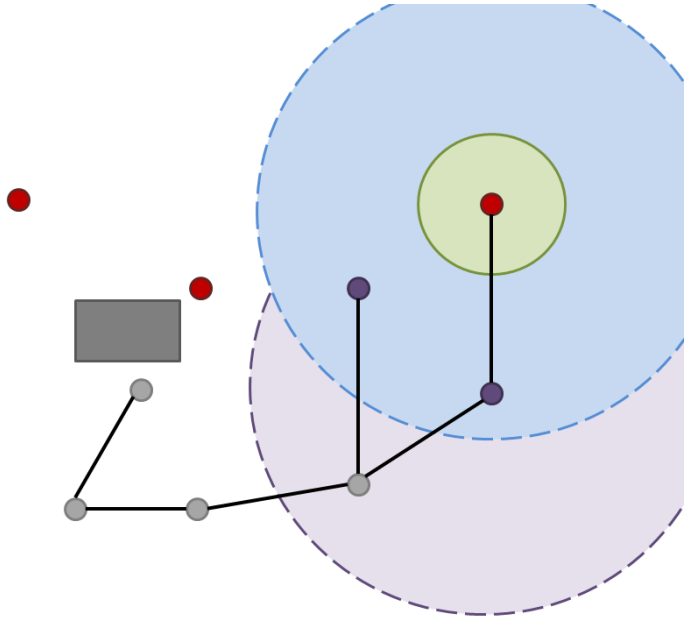
1: Sample n nodes
2: Find lowest-cost "branch" node in the tree, z
3: For each of its unexplored neighbors, x:
4: Find optimal local connection of x to tree
5: If that connection is collision-free:
6: Add edge to tree
7: Remove x from unexplored set
8: Add x to "branch" set
9: Remove z from "branch" set
10: Find new lowest-cost "branch" node and repeat until either:
11: "Branch" set is empty: failure
12: Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

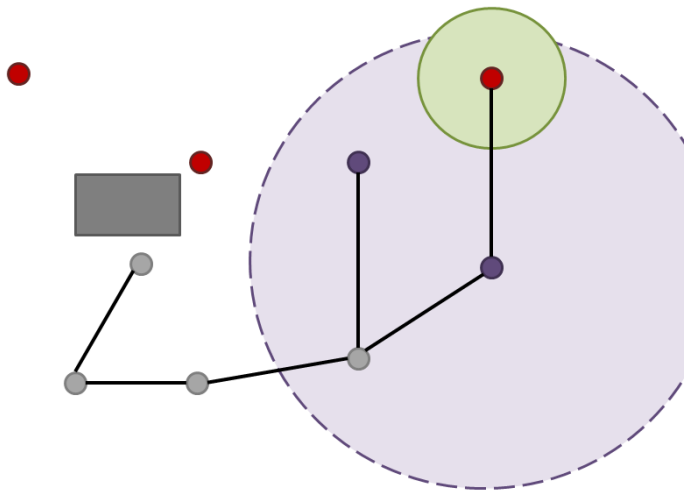
1: Sample n nodes
2: Find lowest-cost "branch" node in the tree, z
3: For each of its unexplored neighbors, x:
4: Find optimal local connection of x to tree
5: If that connection is collision-free:
6: Add edge to tree
7: Remove x from unexplored set
8: Add x to "branch" set
9: Remove z from "branch" set
10: Find new lowest-cost "branch" node and repeat until either:
11: "Branch" set is empty: failure
12: Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

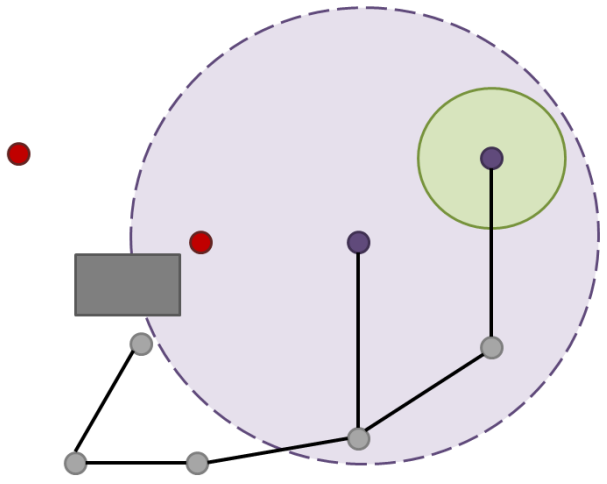
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

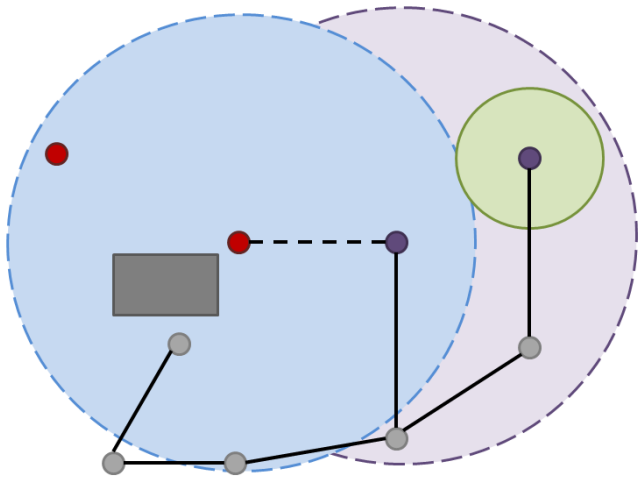
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

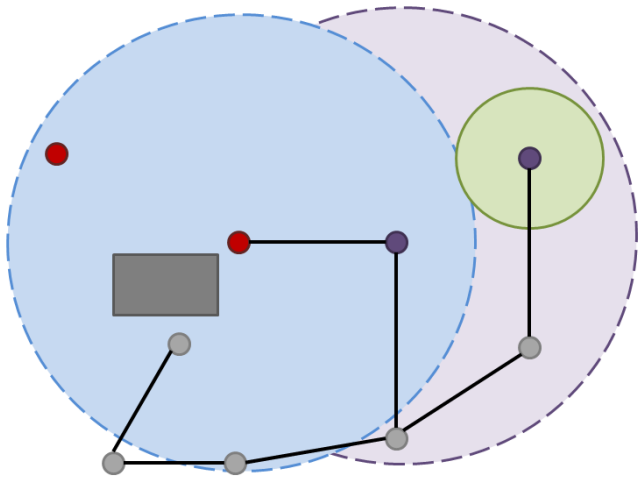
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

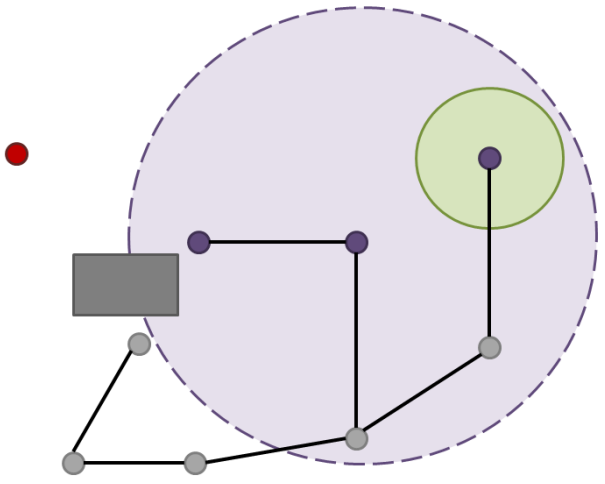
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

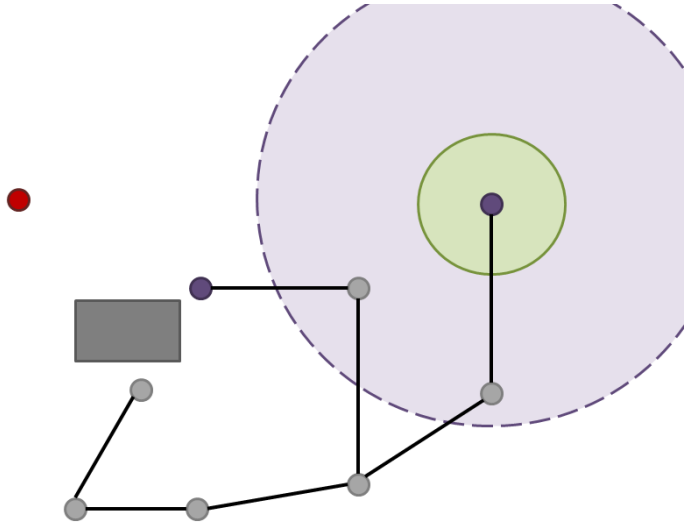
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

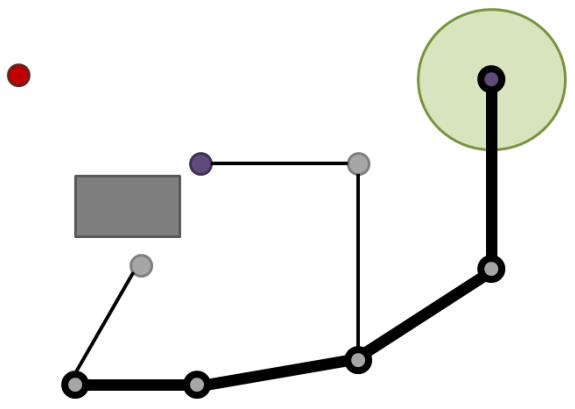
Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Fast Marching Tree Algorithm (FMT*)

Fast Marching Tree Algorithm (FMT*)

Sample n nodes
Find lowest-cost "branch" node in the tree, z
For each of its unexplored neighbors, x:
    Find optimal local connection of x to tree
    If that connection is collision-free:
        Add edge to tree
        Remove x from unexplored set
        Add x to "branch" set
Remove z from "branch" set
Find new lowest-cost "branch" node and repeat until either:
    "Branch" set is empty: failure
    Lowest-cost "branch" node is in goal



Theoretical and Experimental Results

Theory

- Laziness introduces "suboptimal" connections, but such connections are vanishingly rare and FMT* is asymptotically optimal
- Ratio of # of collision-checks for FMT* versus PRM* goes to zero
- Convergence rate bound of order $O(n^{-1/d+\epsilon})$

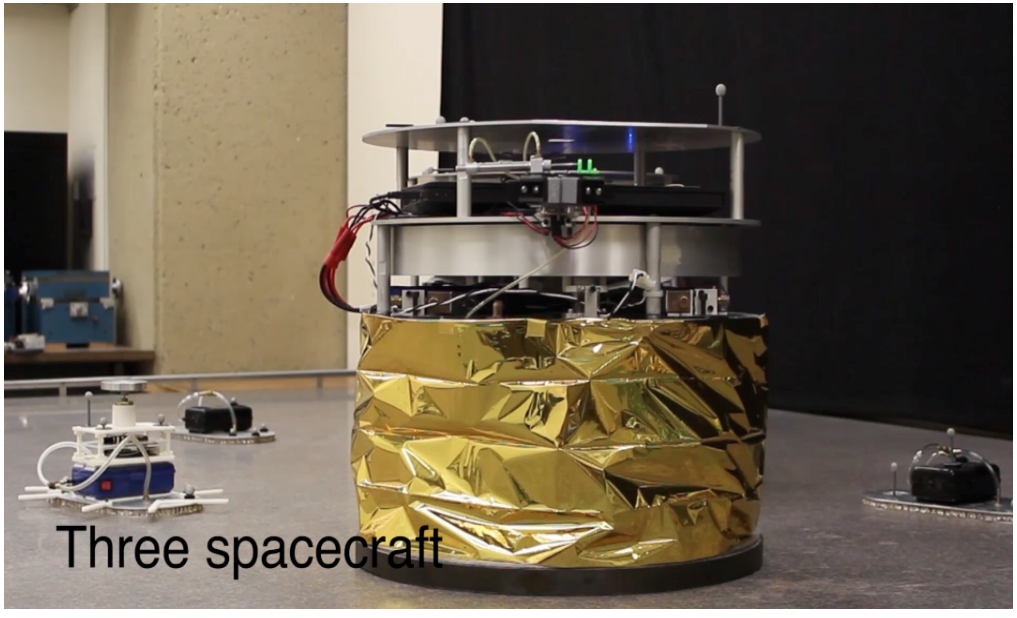
Theoretical and Experimental Results

Theory

- Laziness introduces "suboptimal" connections, but such connections are vanishingly rare and FMT* is asymptotically optimal
- Ratio of # of collision-checks for FMT* versus PRM* goes to zero
- Convergence rate bound of order $O(n^{-1/d+\epsilon})$

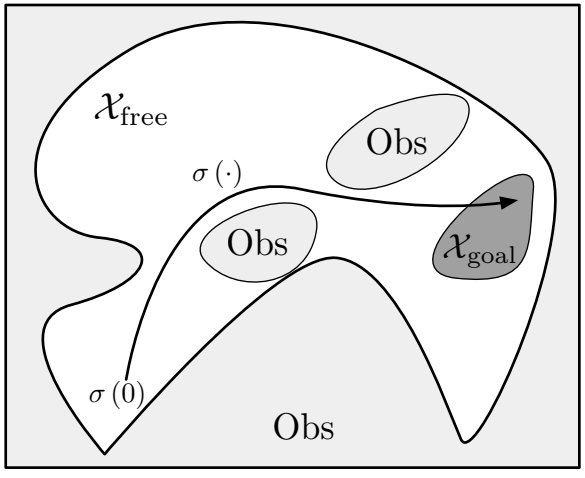
Experiments

- Numerical experiments show *significant* speed-ups



Outline

Summary about Motion Planning



Popular methods:

- Geometric algorithms (visibility graphs, cell decomposition) [LaValle, '06]
- Potential functions [Rimon, Koditschek, '92]
- Grid-based search (A*, D*) [Stentz, '94]
- Sampling-based [Kavraki et al, '96; LaValle, Kuffner, '06]

Several directions for future research:

- Smoothing via convexification
- Adaptation via learning
- Real-time computation via massive parallelization
- Seamless shared control
- Interplay with higher-level decision making via formal methods

Next time