Robot Planning Algorithms

CSE276C
November 12, 2024

1

"Planning" is a very broad term

"Planning" can mean a lot of different things to different people

Generally, planning is figuring out what to do in order to achieve a goal, generally ahead of time.

Control theory - Find inputs to a nonlinear dynamical system that drive it from an initial state to a specified goal state.

Motion/Path Planning - Convert high-level specifications of tasks from humans into low-level instructions about how to move.

(Classical) AI - Finding a set of discrete actions that can be applied sequentially to transform an initial state into a goal state.

2

Some different types of planning problems

Discrete - State space is finite or countably infinite. Action space is finite.
Continuous - State or configuration space is uncountably infinite.

- Sometimes this can refer if a plan is a discrete sequence of actions or a continuous function of time.

3

Some different types of planning problems

Discrete - State space is finite or countably infinite. Action space is finite.
Continuous - State or configuration space is uncountably infinite.

- Sometimes this can refer if a plan is a discrete sequence of actions or a continuous function of time.

Deterministic - The next state is completely specified by the current state and action.
Stochastic - Some information about the next state cannot be predicted with certainty.

4

Some different types of planning problems

Discrete - State space is finite or countably infinite. Action space is finite.
Continuous - State or configuration space is uncountably infinite.

- Sometimes this can refer if a plan is a discrete sequence of actions or a continuous function of time.

Deterministic - The next state is completely specified by the current state and action.
Stochastic - Some information about the next state cannot be predicted with certainty.

Static World - The world is static - no state changes will occur except in response to actions (no exogenous events).
Dynamic World - State changes may occur independent of robot actions.

5

Some different types of planning problems

Discrete - State space is finite or countably infinite. Action space is finite.
Continuous - State or configuration space is uncountably infinite.

- Sometimes this can refer if a plan is a discrete sequence of actions or a continuous function of time.

Deterministic - The next state is completely specified by the current state and action.
Stochastic - Some information about the next state cannot be predicted with certainty.

Static World - The world is static - no state changes will occur except in response to actions (no exogenous events).
Dynamic World - State changes may occur independent of robot actions.

Fully Observable - The robot has complete information about the current state.
Partial Observability - The robot has incomplete information about the current state.

6

Some different types of planning problems

Discrete - State space is finite or countably infinite. Action space is finite.
Continuous - State or configuration space is uncountably infinite.
- Sometimes this can refer if a plan is a discrete sequence of actions or a continuous function of time.

Deterministic - The next state is completely specified by the current state and action.
Stochastic - Some information about the next state cannot be predicted with certainty.

Static World - The world is static - no state changes will occur except in response to actions (no exogenous events).
Dynamic World - State changes may occur independent of robot actions.

Fully Observable - The robot has complete information about the current state.
Partial Observability - The robot has incomplete information about the current state.

Offline - Planning occurs before execution using state transition models in lieu of observations.
Online - Planning occurs in parallel with observing and acting. Planner integrates and reacts to new information in real-time.

Some different types of planning problems

Discrete - State space is finite or countably infinite. Action space is finite.
Continuous - State or configuration space is uncountably infinite.
- Sometimes this can refer if a plan is a discrete sequence of actions or a continuous function of time.

Deterministic - The next state is completely specified by the current state and action.
Stochastic - Some information about the next state cannot be predicted with certainty.

Static World - The world is static - no state changes will occur except in response to actions (no exogenous events).
Dynamic World - State changes may occur independent of robot actions.

Fully Observable - The robot has complete information about the current state.
Partial Observability - The robot has incomplete information about the current state.

Offline - Planning occurs before execution using state transition models in lieu of observations.
Online - Planning occurs in parallel with observing and acting. Planner integrates and reacts to new information in real-time.

for motion planning only…
Kinematic Constraints - Robot is able to follow any trajectory that respects continuity, joint limits, and avoids obstacles.
Differential Constraints - Robot motion must respect additional complex constraints, e.g. kinodynamic and nonholonomic constraints.

Some different types of planning problems

Discrete - State space is finite or countably infinite. Action space is finite.
Continuous - State or configuration space is uncountably infinite.
- Sometimes this can refer if a plan is a discrete sequence of actions or a continuous function of time.

Deterministic - The next state is completely specified by the current state and action.
Stochastic - Some information about the next state cannot be predicted with certainty.

Static World - The world is static - no state changes will occur except in response to actions (no exogenous events).
Dynamic World - State changes may occur independent of robot actions.

Fully Observable - The robot has complete information about the current state.
Partial Observability - The robot has incomplete information about the current state.

Offline - Planning occurs before execution using state transition models in lieu of observations.
Online - Planning occurs in parallel with observing and acting. Planner integrates and reacts to new information in real-time.

for motion planning only…
Kinematic Constraints - Robot is able to follow any trajectory that respects continuity, joint limits, and avoids obstacles.
Differential Constraints - Robot motion must respect additional complex constraints, e.g. kinodynamic and nonholonomic constraints.

9

Shared concepts

State - a representation of the parts of the world that are relevant to the planning problem; sometimes called *configuration space* when representing internal state.

Actions - how the robot influences the world; sometimes called *operators* or *inputs* or *control signals*.

Order - states and actions occur in a set, monotonic order; a sequence of timesteps, continuous time intervals, etc.

Goal - description of what constitutes a solution to the planning problem; often a set of states or constraints that must hold.

Criterion - specification of whether a feasible/satisficing solution is acceptable or if the optimal solution is required; in the latter case, some loss function must be provided to optimize, e.g. shortest path vs. maximum clearance.

10

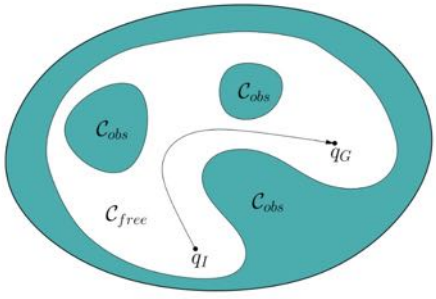
Motion Planning

11

Problem Formulation: Simple Motion Planning

Given

- configuration space C,
- a set of obstacles O,
- an initial configuration $q_I \in C$, and
- a goal configuration $q_G \in C$



Compute

A continuous path $\tau : [0, 1] \rightarrow C_{free}$
with $\tau(0) = q_I$ and $\tau(1) = q_G$.

Typically expressed as a sequence of line segments that can be postprocessed into smoothness

12

Assumptions

Point robot (e.g. if obstacle C-space reps already account for robot geom)

- Implies translations only (pt has no orientation)

No differential constraints

General approach:

Find a way to discretize the space to be able to plan over discrete domain.

13

Roadmaps

Way to represent continuous configuration space as a **connectivity graph** such that motion planning is reduced to a **graph search problem.**

A **roadmap** is a graph in which

- Each node is a point in C_{free}
- Each edge is a "simple" path through C_{free}

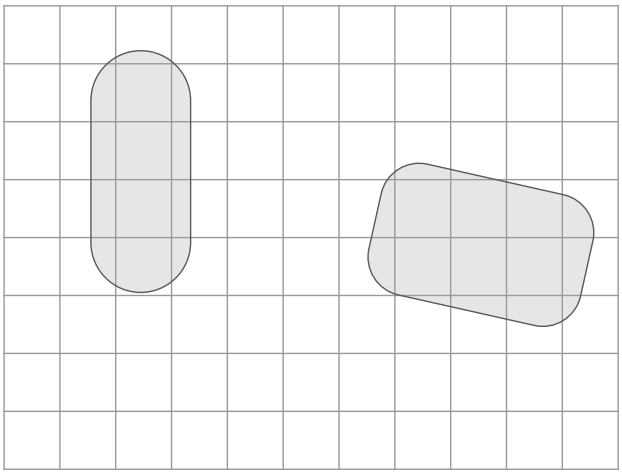
and that exhibits the following two properties:

1. **Accessibility:** It is simple to reach a point on the roadmap from any $q \in C_{free}$ while trivially avoiding collisions;
2. **Connectivity-preserving:** For any pair q_1, q_2 of points that is connected to the roadmap, a path exists between them in the roadmap *if and only if* there is a path between q_1 and q_2 in C_{free}.

Note that the roadmap is built **agnostic of a particular choice of q_I and q_G**. It can therefore be reused for other MP queries in the same domain.

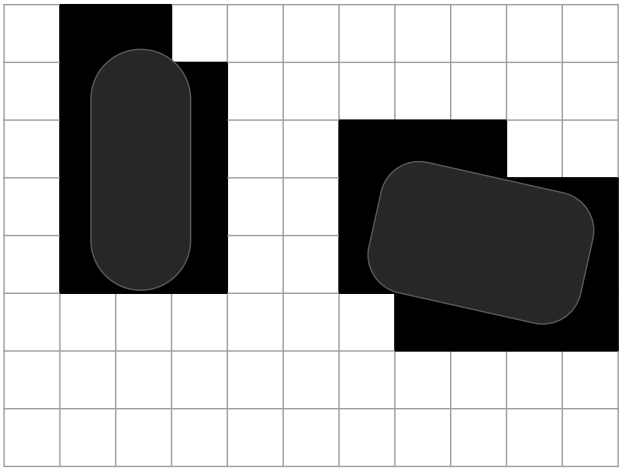
14

Binary Occupancy Grid



15

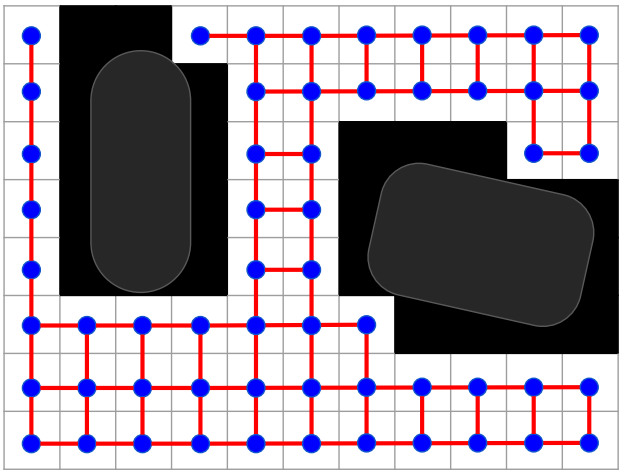
Simplest Roadmap: Binary Occupancy Grid



16

Simplest Roadmap: Binary Occupancy Grid

Define Adjacency Graph

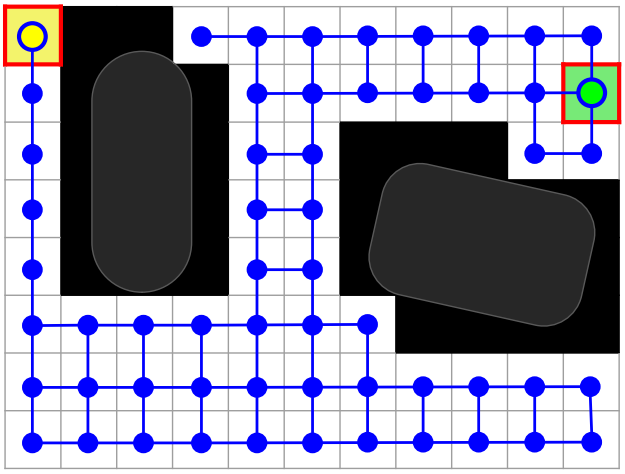


17

Simplest Roadmap: Binary Occupancy Grid

Define Adjacency Graph

Given a Motion Planning Query (q_I,q_G)



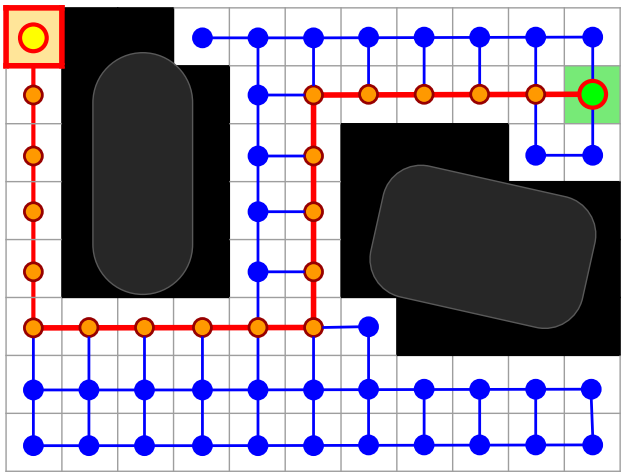
18

Simplest Roadmap: Binary Occupancy Grid

Define Adjacency Graph

Given a Motion Planning Query

Find path to goal

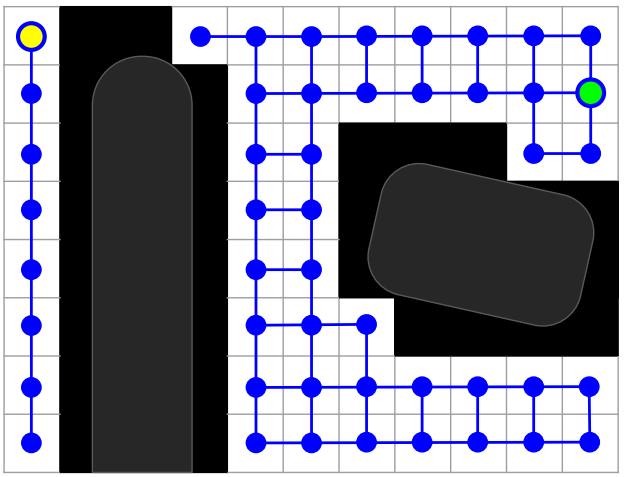


19

Simplest Roadmap: Binary Occupancy Grid

Now there's no path!

Now it's problematic that we've ruled out the upper route.



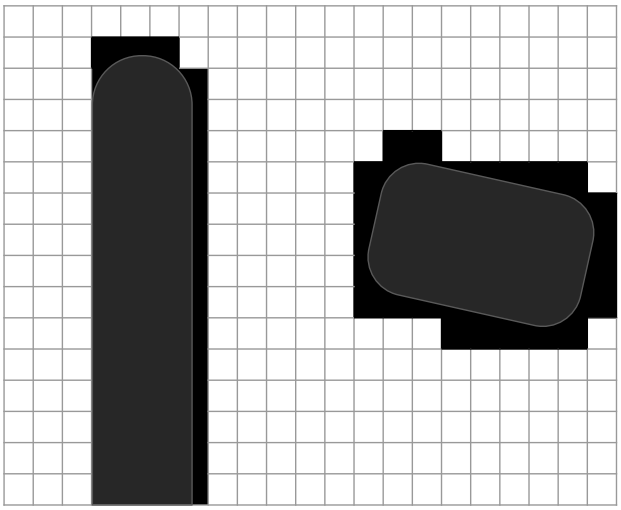
20

Binary Occupancy Grid

Higher resolution

More expensive

Now a path can be found



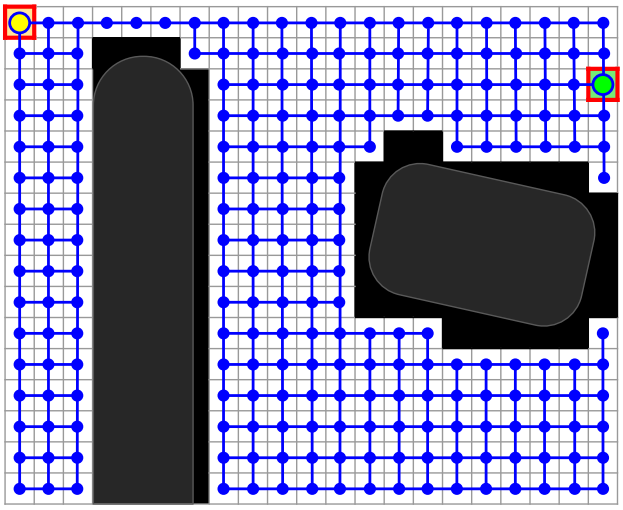
21

Binary Occupancy Grid

Higher resolution

More expensive

Now a path can be found



22

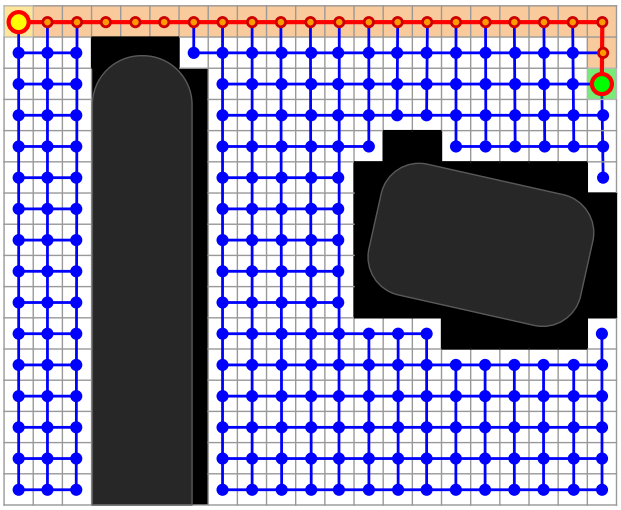
Binary Occupancy Grid

Higher resolution

More expensive

Now a path can be found

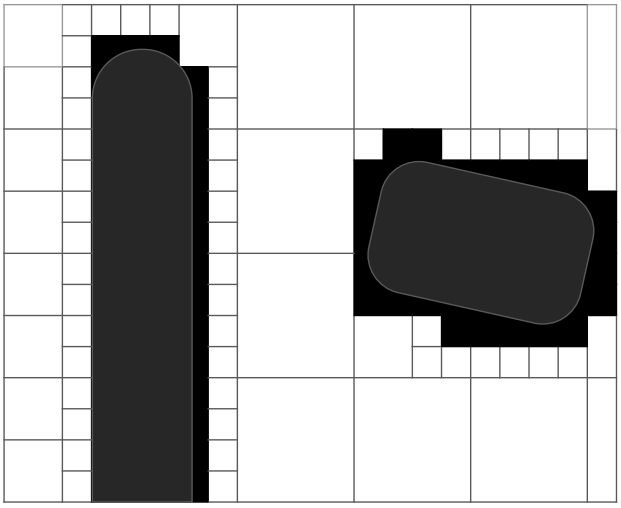
If the resolution can be increased to an arbitrarily high level, it is **resolution-complete.**



23

Adaptive Binary Occupancy Grid

Recursively "zoom in" on any grid squares that are partially occupied.



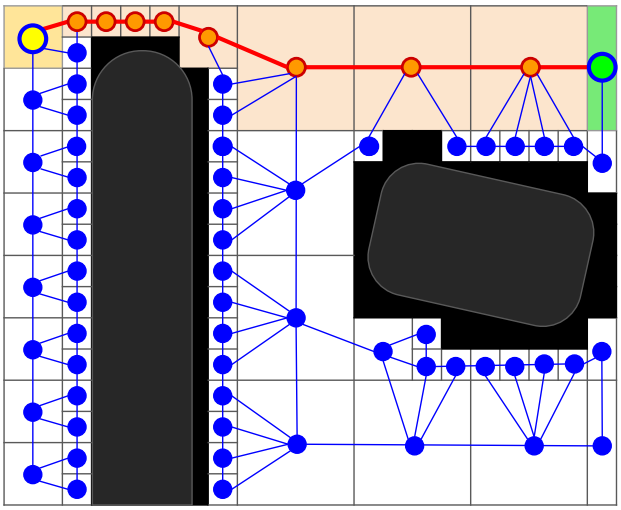
24

Adaptive Binary Occupancy Grid

Recursively "zoom in" on any grid squares that are partially occupied.

Reduce the size of the graph while maintaining detail where necessary.

Store in a quad-tree data structure.



25

Geometrical Approaches

- Assumption: exact geometry of world/obstacles/configuration space is given
- "Combinatorial" planning
- Constructs structures in C-space that **discretely** and **completely** capture all information needed for planning

General Process

- **Preprocessing phase:** Create a graphical **roadmap** representation that perfectly captures the connectedness of C_{free}
- **Query phase:** Use **graph-search algorithms** to find a path on this topological graph

26

Visibility/Shortest path roadmap

- Cuts corners as close as possible to minimize path length
- Requires relaxing the problem such that paths can go through C_{free} or the boundary of C_{obs}

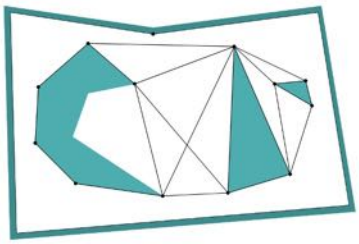


Figure 6.11: The shortest-path roadmap includes edges between consecutive reflex vertices on $\mathcal{C}_{obs}$ and also bitangent edges.

Voronoi/Maximum clearance roadmap

- Keeps as far away from C_{obs} as possible
- Can decrease likelihood of collisions even with uncertainty or environmental perturbations
- Often preferred for mobile robots

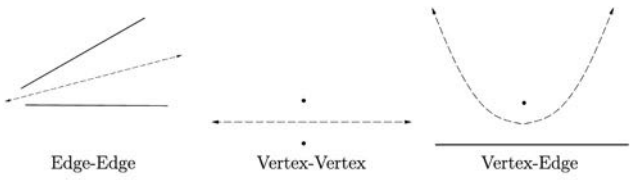


Figure 6.9: Voronoi roadmap pieces are generated in one of three possible cases. The third case leads to a quadratic curve.

Cell Decomposition

- Partition C_{free} into ***cells*** of simple geometry representing trivially path-connected regions.
  - i.e. it can be assumed that any point can be reached from any other within the same cell.
- Roadmap formed from **cell adjacency graph**:
  - Vertices represent cells and/or cell boundaries;
  - Edges connect vertices of adjacent cells and/or boundaries;
  - e.g. an interior node is connected its boundary nodes, which are then connected to interior nodes of any cells sharing that boundary.

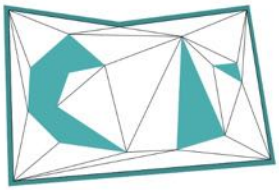


Figure 6.16: A triangulation of $\mathcal{C}_{free}$.

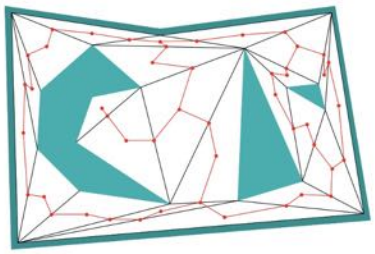


Figure 6.17: A roadmap obtained from the triangulation.

29

Cell Decomposition

An effective cell decomposition should ensure the following **computations can be performed efficiently**:

1. Calculating the cell representation of the configuration space;
2. Finding a path between any two points within a cell (trivial e.g. if every cell is convex);
3. Determining cell adjacency to construct the roadmap; and
4. Given a query (q_I, q_G), computing which cells they belong to.

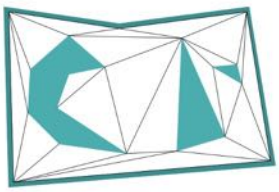


Figure 6.16: A triangulation of $\mathcal{C}_{free}$.

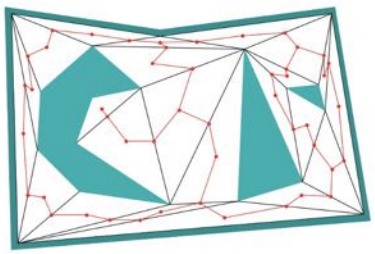
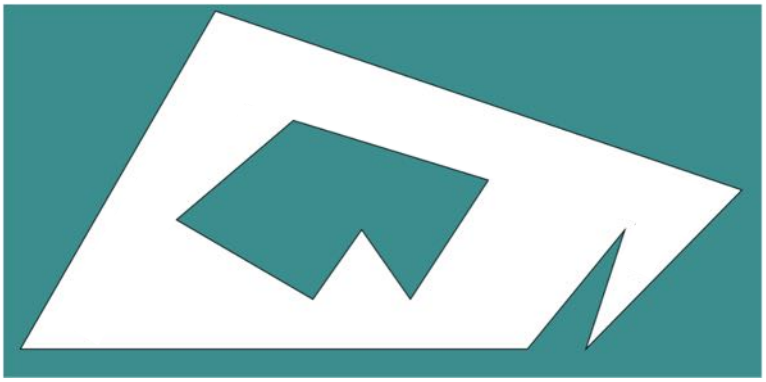


Figure 6.17: A roadmap obtained from the triangulation.

30

Example: Trapezoidal Cell Decomposition

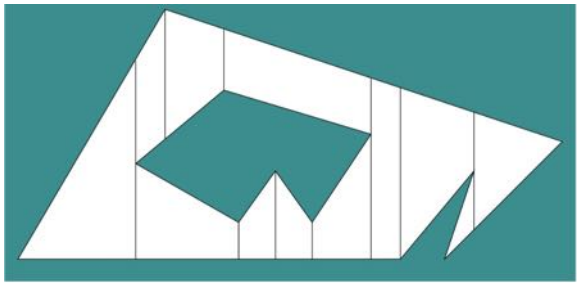
Also called vertical cell decomposition.



31

Example: Trapezoidal Cell Decomposition

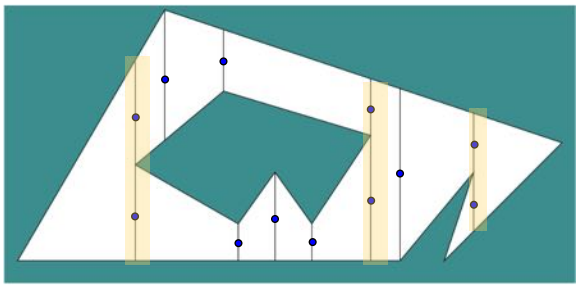
1. **Decompose C_{free} into trapezoids with vertical side segments.**
   - From each polygon vertex, shoot rays up and down until immediately blocked or it meets another part of the obstacle boundary.



32

Example: Trapezoidal Cell Decomposition

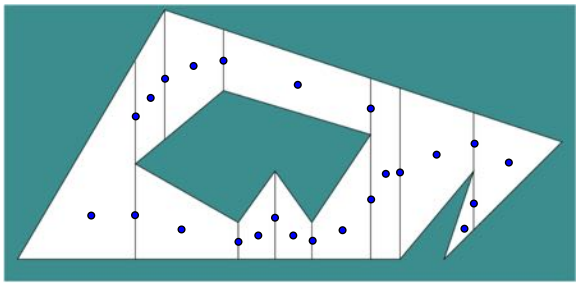
1. **Decompose C_{free} into trapezoids with vertical side segments.**
   - From each polygon vertex, shoot rays up and down until immediately blocked or it meets another part of the obstacle boundary.
1. **Place one vertex in every vertical segment.**
   - E.g. at the midpoint. Note that if two collinear segments meet, they are still treated as separate (highlighted).



33

Example: Trapezoidal Cell Decomposition

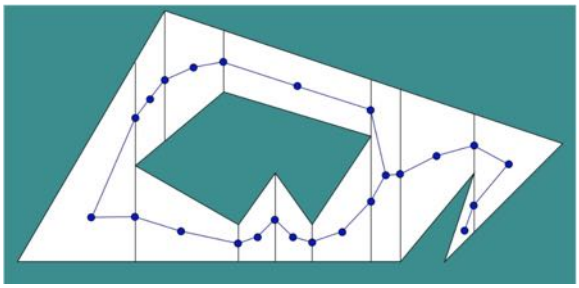
1. **Decompose C_{free} into trapezoids with vertical side segments.**
   - From each polygon vertex, shoot rays up and down until immediately blocked or it meets another part of the obstacle boundary.
1. **Place one vertex in every vertical segment.**
   - E.g. at the midpoint. Note that if two collinear segments meet, they are still treated as separate.
1. **Place one vertex in the interior of every trapezoid.**
   - E.g. at the centroid.



34

Example: Trapezoidal Cell Decomposition

1. **Decompose C_{free} into trapezoids with vertical side segments.**
   - From each polygon vertex, shoot rays up and down until immediately blocked or it meets another part of the obstacle boundary.
1. **Place one vertex in every vertical segment.**
   - E.g. at the midpoint. Note that if two collinear segments meet, they are still treated as separate.
1. **Place one vertex in the interior of every trapezoid.**
   - E.g. at the centroid.
1. **Connect each segment vertex to the two adjacent interior vertices.**
   - Each connection forms an edge in the graph and corresponds to a "simple" straight-line path.

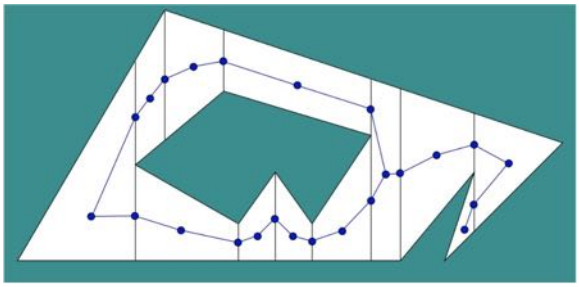


35

Querying the Roadmap

Recall: roadmap is constructed without considering the query pair, q_I and q_G

- Roadmap can be reused to solve numerous MP problems in the same space
- Invest in preprocessing step to construct the roadmap once, use many times



36

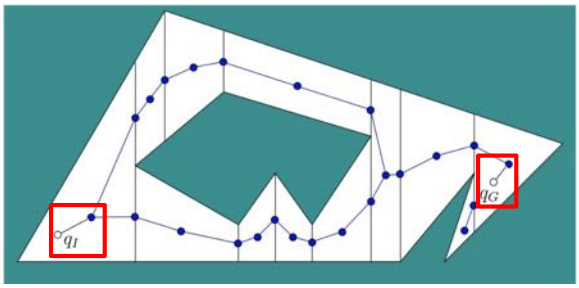
Querying the Roadmap

Recall: roadmap is constructed without considering the query pair, q_I and q_G

- Roadmap can be reused to solve numerous MP problems in the same space
- Invest in preprocessing step to construct the roadmap once, use many times

Query Handling (simple example)

1. Find the trapezoids that contain q_I and q_G.
2. Connect q_I and q_G to the vertices in their respective trapezoids.



37

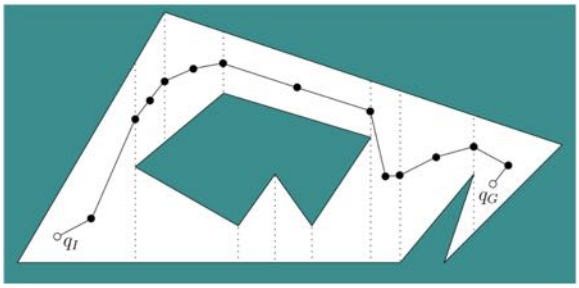
Querying the Roadmap

Recall: roadmap is constructed without considering the query pair, q_I and q_G

- Roadmap can be reused to solve numerous MP problems in the same space
- Invest in preprocessing step to construct the roadmap once, use many times

Query Handling (simple example)

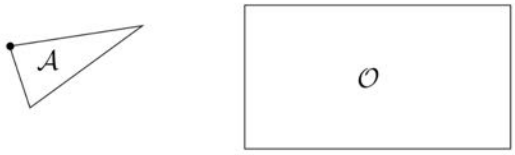
1. Find the trapezoids that contain q_I and q_G.
2. Connect q_I and q_G to the vertices in their respective trapezoids.
3. Search the graph for a path that connects q_I to q_G. *(we'll expand on this soon!)*



38

The Curse of Combinatorial Complexity

- Non-convex robots and obstacles
  - Can decompose nonconvex shapes into groups of many convex shapes
- Rotations - nonlinearity!
  - Even polynomial representations result in >70 facets for the simple case shown right
- High dimensional C-space
  - Canny's algorithm can produce a roadmap in 3D space in single exponential time, but is almost never implemented due to numerical issues and combinatorial explosion
- Differential Constraints - more nonlinearity!
- Multi-robot scenarios
- Etc…



39

Sample-Based Motion Planning

40

Sample-based Motion Planning

Abandon the idea of explicitly characterizing C_{free} and C_{obs}

Leave the planning algorithm "in the dark" when exploring C_{free}.

The only "light" is provided by querying collision detection algorithm

Collision detection algorithm is a black box to planner, computes if a given configuration is in C_{free}

Lazy Incremental Search

- Planners **incrementally** sample points in C_{free}, gradually revealing more and more of it in concert with the collision detector.
- Note that obstacles and robot geometries are still known
- However, prohibitive to explicitly represent.
- Sampling-based approaches attempt to find a solution quickly while cheating their way out of building a full "map" of C_{free}.
- **Don't compute more than you have to!**

Two Dominating Approaches

Interleaved Sampling and Searching for Single Queries

Choose me when there are **few advantages to precomputation**, e.g. if…

- The current configuration space is not likely to be relevant for future motion planning queries.
- You have limited memory or preparation time before a query must be solved.
- Regions of the configuration space cannot be sampled beforehand and must be probed on-the-fly.

Roadmap Methods for Multiple Queries

Choose me when there are **advantages to precomputation**, e.g. if…

- There will be multiple motion planning queries (i.e. choices for initial and goal configurations) will be chosen for the same robot and obstacles
- You have time and memory to front-load effort into building a roadmap beforehand.
- New queries must be solved very quickly online.

Rapidly-Exploring Random Trees (RRT)

Poster-child of "Interleaved Sampling and Searching for Single Queries"

Overview:

- Aggressively probe and explore C-space by expanding incrementally from a seed configuration q_0.
- Each iteration extends a graph tree rooted at q_o by generating a new vertex and connecting it to the rest of the tree with a new edge.
- Each edge is a collision-free path between two configurations.

```
RRT(q_0)
1  G.init(q_0);
2  repeat
3      q_rand → RANDOM_CONFIG(C)
4      q_near ← NEAREST(G, q_rand);
5      G.add_edge(q_near, q_rand);
```

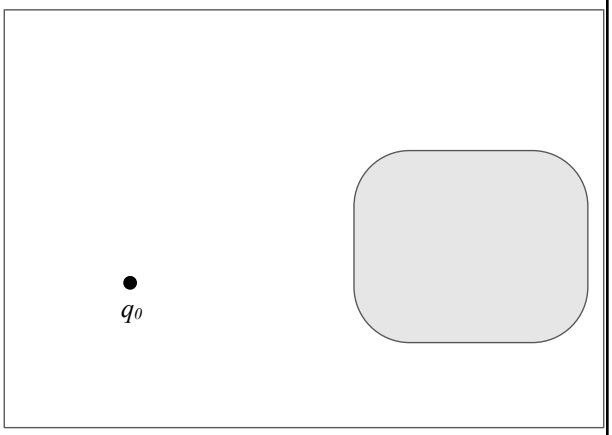
45 iterations

2345 iterations

RRT: Walkthrough

1. **Initialize Tree:** Declare tree with one vertex, q_0.

   $G = (V=\{q_0\}, E=\emptyset)$



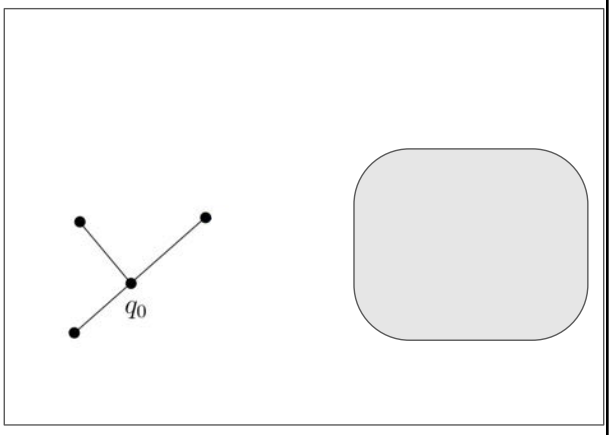
45

RRT: Walkthrough

1. **Initialize Tree:** Declare tree with one vertex, q_0.

   $G = (V=\{q_0\}, E=\emptyset)$

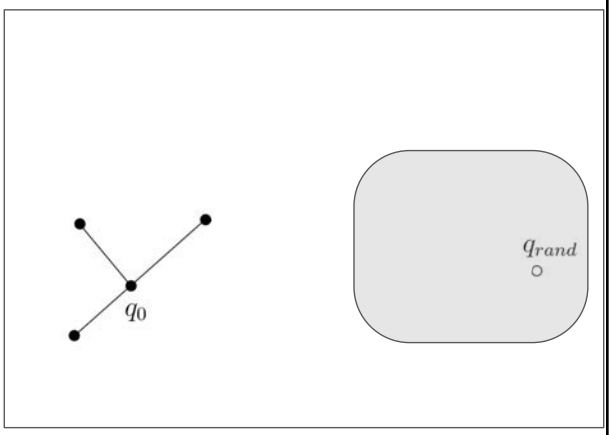
(let's skip ahead a few iterations so we have a tree to discuss)



46

RRT: Walkthrough

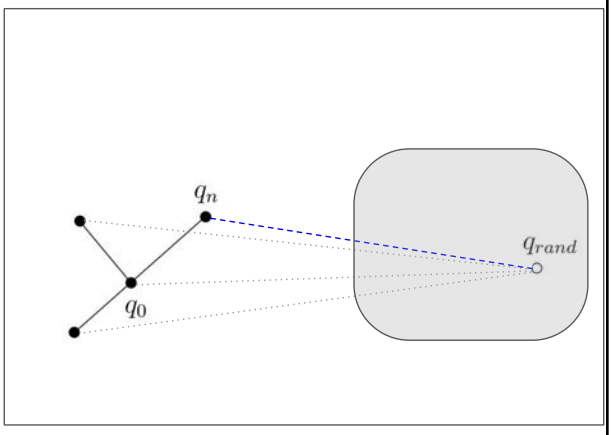
1. **Initialize Tree:** Declare tree with one vertex, q_0.
   $G = (V=\{q_0\}, E=\emptyset)$
2. **Random Sample:** Generate a random configuration $q_{rand} \in C$.



47

RRT: Walkthrough

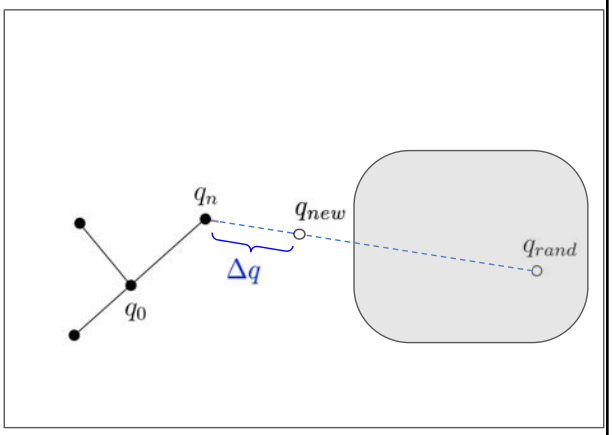
1. **Initialize Tree:** Declare tree with one vertex, q_0.
   $G = (V=\{q_0\}, E=\emptyset)$
2. **Random Sample:** Generate a random configuration $q_{rand} \in C$.
3. **Identify Nearest Vertex:** Compute the nearest tree vertex $q_{near} \in V$ to q_{rand}.



48

RRT: Walkthrough

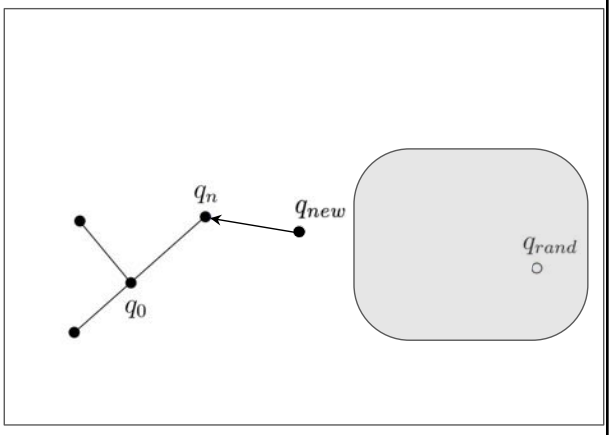
1. **Initialize Tree:** Declare tree with one vertex, q_0.
   $G = (V=\{q_0\}, E=\emptyset)$
2. **Random Sample:** Generate a random configuration $q_{rand} \in C$.
3. **Identify Nearest Vertex:** Compute the nearest tree vertex $q_{near} \in V$ to q_{rand}.
4. **Steer:** Create a new configuration q_{new} located Δq units from q_{near} toward q_{rand}.



49

RRT: Walkthrough

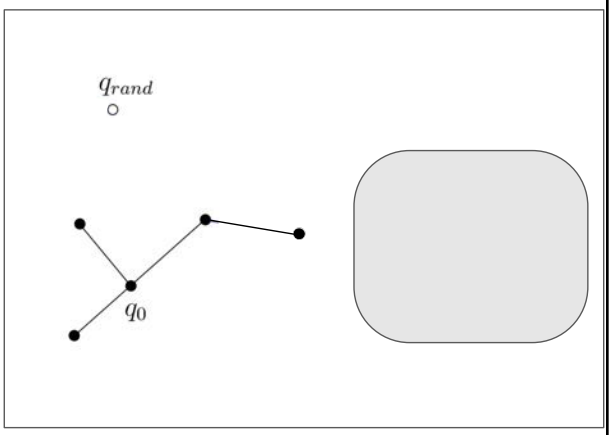
1. **Initialize Tree:** Declare tree with one vertex, q_0.
   $G = (V=\{q_0\}, E=\emptyset)$
2. **Random Sample:** Generate a random configuration $q_{rand} \in C$.
3. **Identify Nearest Vertex:** Compute the nearest tree vertex $q_{near} \in V$ to q_{rand}.
4. **Steer:** Create a new configuration q_{new} located Δq units from q_{near} toward q_{rand}.
5. **Connect:** If no collisions at or along path to q_{new}, add q_{new} to V and an edge from (q_{new}, q_{near}) to E.



50

RRT: Walkthrough

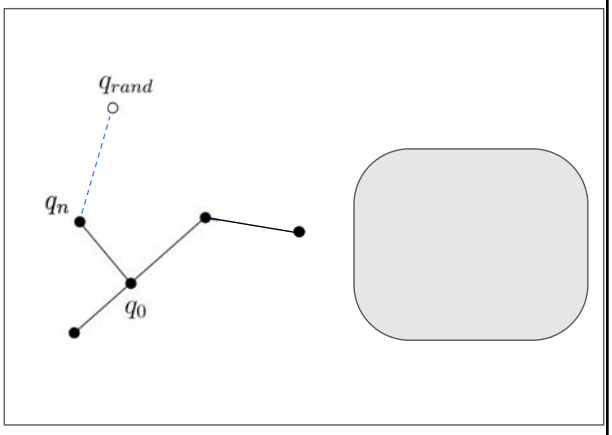
1. **Initialize Tree:** Declare tree with one vertex, q_0.
   $G = (V=\{q_0\}, E=\emptyset)$
2. **Random Sample:** Generate a random configuration $q_{rand} \in C$.
3. **Identify Nearest Vertex:** Compute the nearest tree vertex $q_{near} \in V$ to q_{rand}.
4. **Steer:** Create a new configuration q_{new} located Δq units from q_{near} toward q_{rand}.
5. **Connect:** If no collisions at or along path to q_{new}, add q_{new} to V and an edge from (q_{new}, q_{near}) to E.
6. **Iterate:** Return to Step 2 and repeat.



51

RRT: Walkthrough

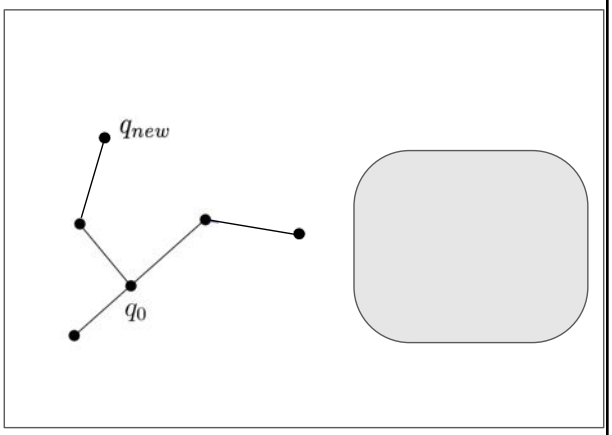
1. **Initialize Tree:** Declare tree with one vertex, q_0.
   $G = (V=\{q_0\}, E=\emptyset)$
2. **Random Sample:** Generate a random configuration $q_{rand} \in C$.
3. **Identify Nearest Vertex:** Compute the nearest tree vertex $q_{near} \in V$ to q_{rand}.
4. **Steer:** Create a new configuration q_{new} located Δq units from q_{near} toward q_{rand}.
5. **Connect:** If no collisions at or along path to q_{new}, add q_{new} to V and an edge from (q_{new}, q_{near}) to E.
6. **Iterate:** Return to Step 2 and repeat.



52

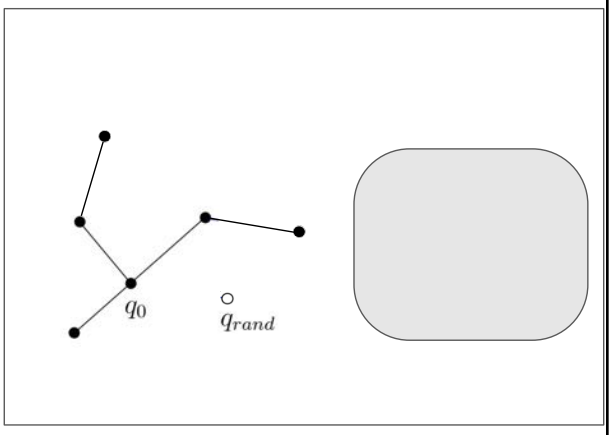
RRT: Walkthrough

1. **Initialize Tree:** Declare tree with one vertex, q_0.
   $G = (V=\{q_0\}, E=\emptyset)$
2. **Random Sample:** Generate a random configuration $q_{rand} \in C$.
3. **Identify Nearest Vertex:** Compute the nearest tree vertex $q_{near} \in V$ to q_{rand}.
4. **Steer:** Create a new configuration q_{new} located Δq units from q_{near} toward q_{rand}.
5. **Connect:** If no collisions at or along path to q_{new}, add q_{new} to V and an edge from (q_{new}, q_{near}) to E.
6. **Iterate:** Return to Step 2 and repeat.



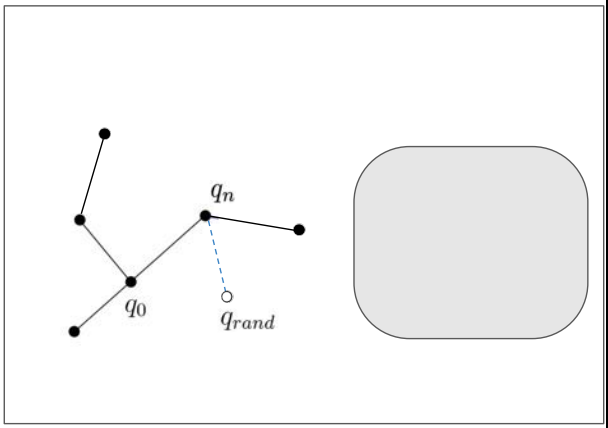
RRT: Walkthrough

1. **Initialize Tree:** Declare tree with one vertex, q_0.
   $G = (V=\{q_0\}, E=\emptyset)$
2. **Random Sample:** Generate a random configuration $q_{rand} \in C$.
3. **Identify Nearest Vertex:** Compute the nearest tree vertex $q_{near} \in V$ to q_{rand}.
4. **Steer:** Create a new configuration q_{new} located Δq units from q_{near} toward q_{rand}.
5. **Connect:** If no collisions at or along path to q_{new}, add q_{new} to V and an edge from (q_{new}, q_{near}) to E.
6. **Iterate:** Return to Step 2 and repeat.



RRT: Walkthrough

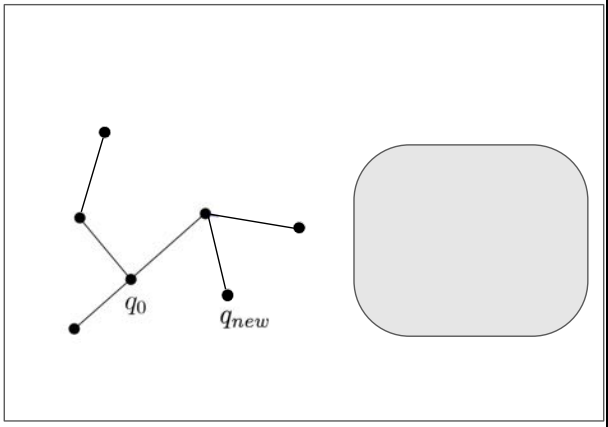
1. **Initialize Tree:** Declare tree with one vertex, q_0.
   $G = (V=\{q_0\}, E=\emptyset)$
2. **Random Sample:** Generate a random configuration $q_{rand} \in C$.
3. **Identify Nearest Vertex:** Compute the nearest tree vertex $q_{near} \in V$ to q_{rand}.
4. **Steer:** Create a new configuration q_{new} located Δq units from q_{near} toward q_{rand}.
5. **Connect:** If no collisions at or along path to q_{new}, add q_{new} to V and an edge from (q_{new}, q_{near}) to E.
6. **Iterate:** Return to Step 2 and repeat.



55

RRT: Walkthrough

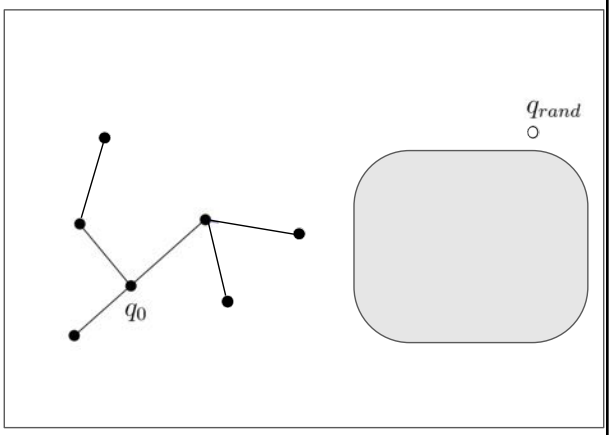
1. **Initialize Tree:** Declare tree with one vertex, q_0.
   $G = (V=\{q_0\}, E=\emptyset)$
2. **Random Sample:** Generate a random configuration $q_{rand} \in C$.
3. **Identify Nearest Vertex:** Compute the nearest tree vertex $q_{near} \in V$ to q_{rand}.
4. **Steer:** Create a new configuration q_{new} located Δq units from q_{near} toward q_{rand}.
5. **Connect:** If no collisions at or along path to q_{new}, add q_{new} to V and an edge from (q_{new}, q_{near}) to E.
6. **Iterate:** Return to Step 2 and repeat.



56

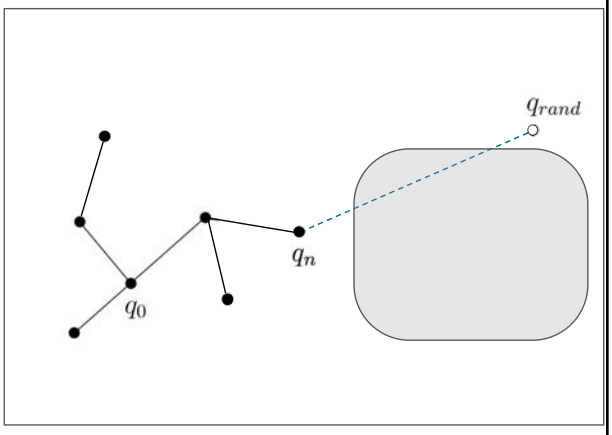
RRT: Walkthrough

1. **Initialize Tree:** Declare tree with one vertex, q_0.
   $G = (V=\{q_0\}, E=\emptyset)$
2. **Random Sample:** Generate a random configuration $q_{rand} \in C$.
3. **Identify Nearest Vertex:** Compute the nearest tree vertex $q_{near} \in V$ to q_{rand}.
4. **Steer:** Create a new configuration q_{new} located Δq units from q_{near} toward q_{rand}.
5. **Connect:** If no collisions at or along path to q_{new}, add q_{new} to V and an edge from (q_{new}, q_{near}) to E.
6. **Iterate:** Return to Step 2 and repeat.



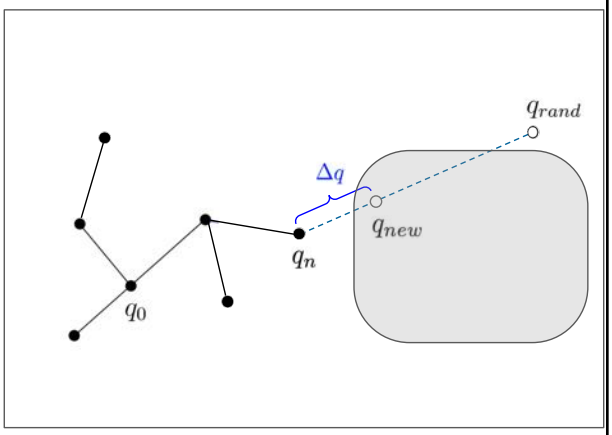
RRT: Walkthrough

1. **Initialize Tree:** Declare tree with one vertex, q_0.
   $G = (V=\{q_0\}, E=\emptyset)$
2. **Random Sample:** Generate a random configuration $q_{rand} \in C$.
3. **Identify Nearest Vertex:** Compute the nearest tree vertex $q_{near} \in V$ to q_{rand}.
4. **Steer:** Create a new configuration q_{new} located Δq units from q_{near} toward q_{rand}.
5. **Connect:** If no collisions at or along path to q_{new}, add q_{new} to V and an edge from (q_{new}, q_{near}) to E.
6. **Iterate:** Return to Step 2 and repeat.



RRT: Walkthrough

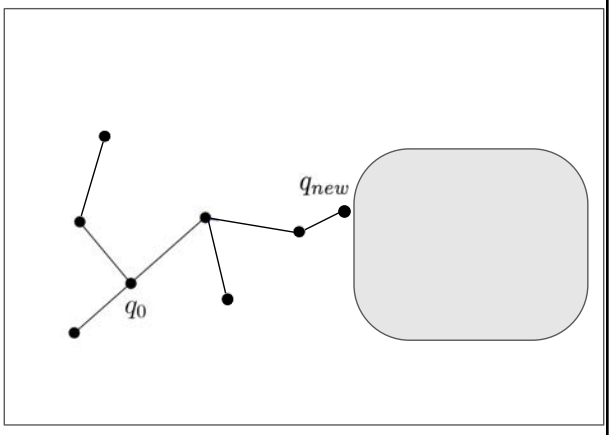
1. **Initialize Tree:** Declare tree with one vertex, q_0.
   $G = (V=\{q_0\}, E=\emptyset)$
2. **Random Sample:** Generate a random configuration $q_{rand} \in C$.
3. **Identify Nearest Vertex:** Compute the nearest tree vertex $q_{near} \in V$ to q_{rand}.
4. **Steer:** Create a new configuration q_{new} located Δq units from q_{near} toward q_{rand}.
5. **Connect:** If no collisions at or along path to q_{new}, add q_{new} to V and an edge from (q_{new}, q_{near}) to E.
6. **Iterate:** Return to Step 2 and repeat.



59

RRT: Walkthrough

1. **Initialize Tree:** Declare tree with one vertex, q_0.
   $G = (V=\{q_0\}, E=\emptyset)$
2. **Random Sample:** Generate a random configuration $q_{rand} \in C$.
3. **Identify Nearest Vertex:** Compute the nearest tree vertex $q_{near} \in V$ to q_{rand}.
4. **Steer:** Create a new configuration q_{new} located Δq units from q_{near} toward q_{rand}.
5. **Connect:** If no collisions at or along path to q_{new}, add q_{new} to V and an edge from (q_{new}, q_{near}) to E.
6. **Iterate:** Return to Step 2 and repeat.



60

Sampling Completeness

Recall: Sampling-based approaches attempt to find a solution quickly while cheating their way out of building a full "map" of C_{free}.

Can we just… do that?

Sampling theory: C is uncountably infinite. Even if we sample for eternity, we will only sample a countably infinite number of points. How can we address this mismatch?

Weaken our notion of completeness

Classical completeness: For any input, the algorithm returns a solution if one exists or reports that one does not exist in a finite amount of time.

Probabilistic completeness: As iterations tend toward infinity, the probability that a solution is found if one exists approaches 1. If no solution exists, the algorithm may run forever.

How can we be sure we achieve probabilistic completeness?

Sampling Denseness

As iterations tend to infinity, the set of sampled points becomes arbitrarily close to every point q∈C.

> Let U and V be any subsets of a topological space. The set U is said to be dense in V if $\mathrm{cl}(U) = V$, i.e. adding the boundary points to U forms V.

Examples: (0,1) is dense in [0,1]; $\mathbb{Q}$ is dense in $\mathbb{R}$

Note that random sequences can only be *probably* dense.

Generally not an issue unless number of samples is very low.

Sampling Uniformity

Every point in C is equally likely to be sampled.

Sampling method is uniform if the probability density function is consistent the Haar measure of the space.

Many configuration spaces are just the Cartesian product of all DoF, so they can be (randomly) sampled independently.

Except for…

Random Samples of 3D Rotations

Sampling each DoF independently as in Euler angles **will not** give uniform sampling over *SO(3)*.

This is relevant for generating uniformly random 3D rotations. Sampling using Euler angles will bias sampling toward the poles.

Choose three points $u_1, u_2, u_3 \in [0, 1]$ uniformly at random. Then the following expression produces a uniformly random quaternion:

$$h = (\sqrt{1-u_1}\sin 2\pi u_2,\ \sqrt{1-u_1}\cos 2\pi u_2,\ \sqrt{u_1}\sin 2\pi u_3,\ \sqrt{u_1}\cos 2\pi u_3)$$

Deterministic Sampling for *Resolution* Completeness

Resolution completeness: If a deterministic sampling process can increase the resolution arbitrarily, the planning algorithm will find a solution if one exists in finite time. If no solution exists, the algorithm may run forever.

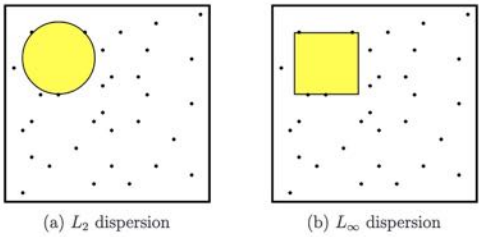
Why bother with deterministic sampling?

Low Dispersion Sampling

Intuition: place samples in a way that makes the largest uncovered area be as small as possible

Note: **low dispersion** means the points are **highly spread out**

Low-dispersion sampling schemes cause highly uniform distribution of points over C. This causes them to fail statistical tests, but the point distribution is often **better for motion planning purposes.**

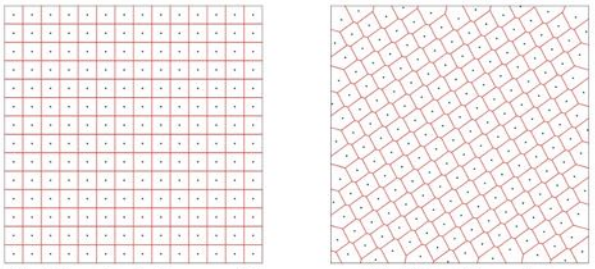


(a) L_2 dispersion (b) L_∞ dispersion

Figure 5.4: Reducing the dispersion means reducing the radius of the largest empty ball.

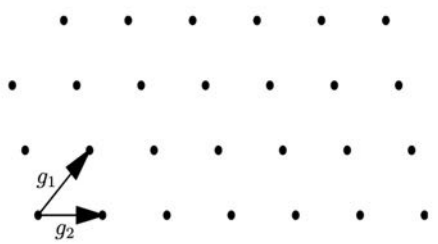
Iteratively-Refined Grids and Lattices

- Lattices: generalization of grids with nonorthogonal axes.
- Innately low dispersion.



(a) 196-point Sukharev grid (b) 196 lattice points

- **Useful property:** neighboring points can be obtained very easily be adding or subtracting axis vectors.



Low Discrepancy Sampling

Intuition: Sample points in a sequence that avoids axis alignments while distributing the points uniformly.

Methods were developed for numerical integration.

Low-discrepancy implies low-dispersion; the converse is not true.

Algorithms:
1. Halton/Hammersley sampling
2. (t,s)-sequences and (t,m,s)-nets
3. Lattices

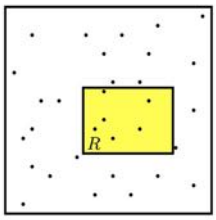
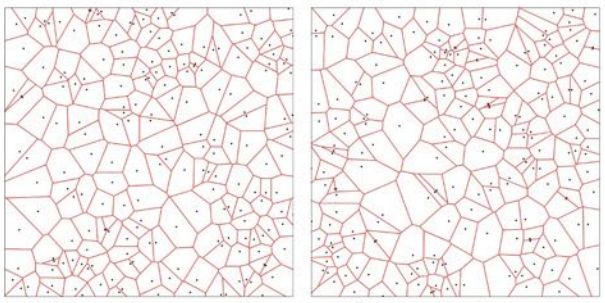


Figure 5.7: Discrepancy measures whether the right number of points fall into boxes. It is related to the chi-square test but optimizes over all possible boxes.

72

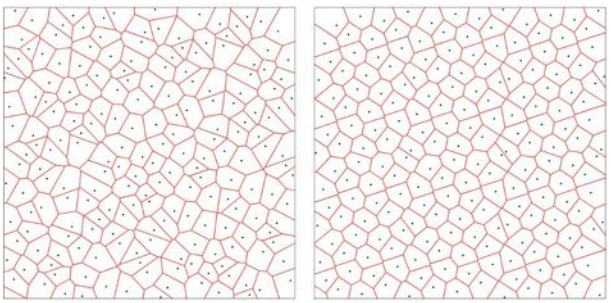
Pseudorandom vs. Halton/Hammersley Sampling

Pseudorandom number generators were made to pass statistical tests of randomness, which implies an irregularity that is not necessarily ideal for motion planning.



(a) 196 pseudorandom samples (b) 196 pseudorandom samples

Figure 5.3: Irregularity in a collection of (pseudo)random samples can be nicely observed with Voronoi diagrams.



(a) 196 Halton points (b) 196 Hammersley points

Figure 5.8: The Halton and Hammersley points are easy to construct and provide a nice alternative to random sampling that achieves more regularity. Compare the Voronoi regions to those in Figure 5.3. Beware that although these sequences produce asymptotically optimal discrepancy, their performance degrades substantially in higher dimensions (e.g., beyond 10).

73

Discrete Planning

74

Planning as Graph Search

- Classical graph search algorithms
- Search must be **systematic** – in the limit, all reachable nodes are visited at least once
  - Finite state space: cycle checking, recording if states have been visited
  - Infinite state space: Trickier, e.g. DFS not systematic; may never stop searching one direction even in limit

75

Common Graph Search Paradigms

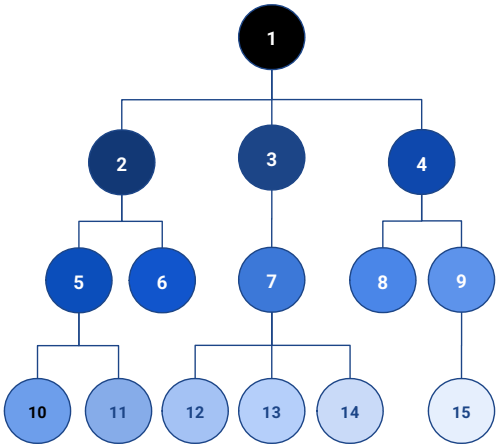
Breadth-first Search (BFS)

Queue-style search order (FIFO)

Uniform expansion of search frontier

Always systematic

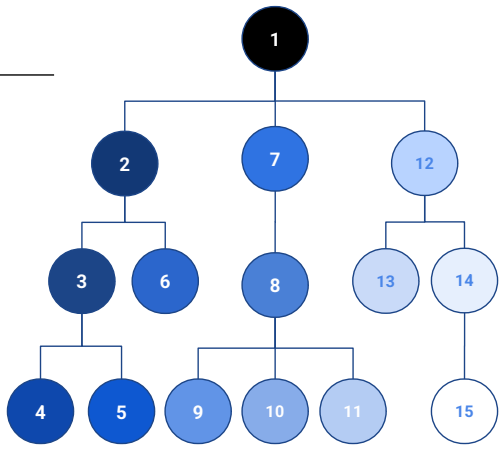
First successful path is guaranteed to be shortest path



76

Common Graph Search Paradigms

Breadth-first Search (BFS)	Depth-first Search (DFS)
Queue-style search order (FIFO)	Stack-style search order (LIFO)
Uniform expansion of search frontier	Aggressive expansion of single path
Always systematic	Systematic only for finite state spaces
First successful path is guaranteed to be shortest path	Generally less overhead – especially useful for domains with many actions per state



77

Common Graph Search Paradigms

Breadth-first Search (BFS)	Depth-first Search (DFS)	Iterative Deepening
Queue-style search order (FIFO)	Stack-style search order (LIFO)	Stack-style search order (LIFO)
Uniform expansion of search frontier	Aggressive expansion of single path	Limited aggressive expansion of single path
Always systematic	Systematic only for finite state spaces	Systematic modification of DFS
First successful path is guaranteed to be shortest path	Generally less overhead – especially useful for domains with many actions per state	Less overhead and better worst-case performance than BFS

78

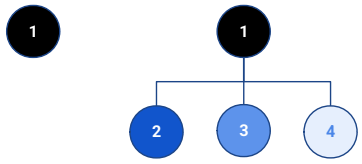
Iterative Deepening

`depth=0`

79

Iterative Deepening

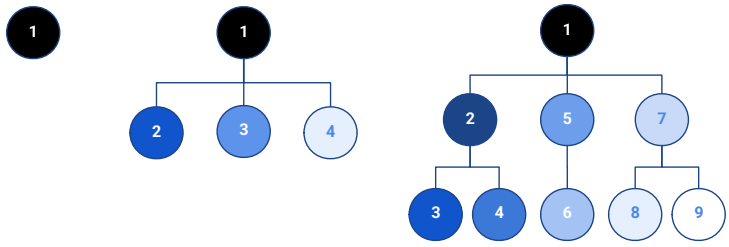
depth=0 depth=1



80

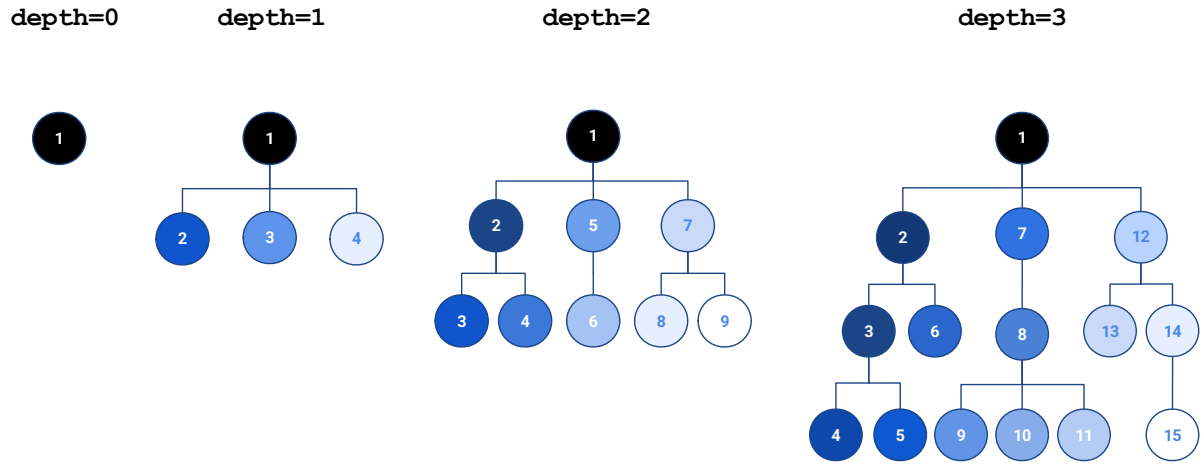
Iterative Deepening

depth=0 depth=1 depth=2



81

Iterative Deepening



82

Weighted edges as path segment costs

What if not all edges of the graph should be treated equally?

In motion plan, edge weights are often used to represent (usually an approximation of) path segment length in configuration space.

Can also be used to represent e.g. traversing difficult terrain.

83

Dijkstra (dynamic programming)

- Finds cost to *all* destinations for a given initial state
- All edges must have a *nonnegative* cost
- Cost of a path is the sum of edge costs

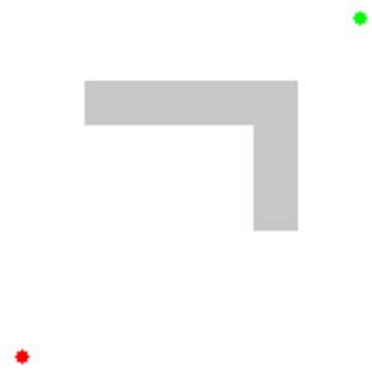
- How to choose next state to expand:
  - Create a priority queue of vertices on the search frontier ordered by current best estimate of cost-to-come.
  - "Cost-to-come of q" : cost of the minimum-cost path from q_I to q known so far

- Dijkstra will find **optimal path**.

84

Dijkstra Description

1. **Initialize:** Define the initial cost of q_I=0, and every other node q_i=∞ . Add all nodes to the priority queue.
2. While there are unvisited nodes left in the priority queue:
   a. Select the frontier node q with the lowest cost and mark as visited.
   b. For each unvisited neighbor q' connected to q by edge e, calculate the cost of reaching q' by going through q as $c_q(q')$=cost(q)+cost(e).
   c. If $c_q(q')$ is lower than the current estimate of c(q'), update c(q')←$c_q(q')$.
   d. Mark q as the parent of q'.



https://www.redblobgames.com visualization

85

Informed Search

Can we leverage information about our domain to speed up search?

Define a **heuristic** *h*(n) that estimates the *cost-to-go*: the cost of the optimal path from the current node to the goal.

Admissible Heuristics

Admissible heuristic:

Let h*(n) be the true cost of the optimal path from n to the goal. Then a heuristic h(n) is admissible if the following holds for all n:

$$h(n) \leq h^*(n)$$

An admissible heuristic should be "optimistic," always underestimating a path's true cost.

The point is to never rule out a path if there's any chance it could be the optimal one. That could only happen by overestimating a path.

Motion Planning Heuristics

Good heuristics tend to be domain-specific, and can be difficult to formulate.

Luckily, in motion planning the workspace and configuration space manifolds have additional structure that can be leveraged for heuristics.

Common admissible heuristics in motion planning:

- Euclidean Distance
- Manhattan Distance
- Octile Distance
- Dijkstra run from the goal

88

Informed Search

Now that we have an admissible heuristic, we can use this estimated cost-to-go instead of Dijkstra cost-to-come to guide search.

redblobgames visualization

89

Informed Search

Now that we have an admissible heuristic, we can use this estimated cost-to-go instead of Dijkstra cost-to-come to guide search… right?

https://www.redblobgames.com/pathfinding/a-star/introduction.html**#greedy-best-first**

This gives **Greedy Best-First Search**. Works very quickly in some cases, but **cannot guarantee it will return the optimal path**.

90

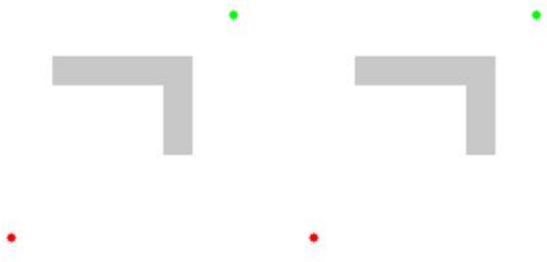
Informed **Optimal** Search - A*

Optimal Solution: need to use both exact cost-to-come (calculated exactly as in Dijkstra) *and* admissible cost-to-go heuristic to achieve optimal planning.

https://www.redblobgames.com/pathfinding/a-star/introduction.html#astar

Priority queue ordered by:

$$f(n) = g(n) + h(n)$$



Dijkstra

A*

(from Wikipedia)

91

Policy planning

Policy: A policy π is a function that maps every state to the action that should be performed if the robot finds itself in that state.

Enables the robot to recover from perturbations or finding itself in an unexpected state.

Generally done by reacting to local circumstances only to minimize computational load.

Common format: Gradient Descent

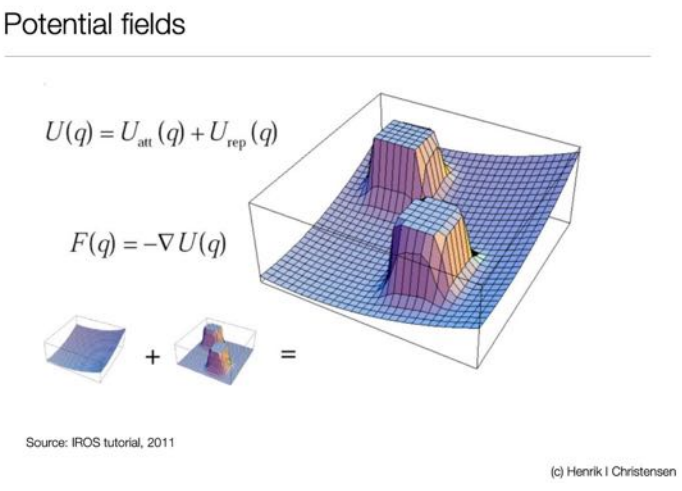
Let $U(q)$ be a scalar field over the configuration space. Then the local policy at any state can be understood as taking the action that minimizes the gradient of $U(q)$.

92

Potential Field-guided planning

Inspired by physical phenomena

- Robot acts as a charged particle in a potential field
- Goal region emits attractive field
- Obstacles emit repulsive fields
- Robot moves in the direction of the force on the charged particle, implici performing gradient descent.



93

Caveats: Potential Field-guided planning

Potential fields are infamous for getting trapped in **local minima.** This can be somewhat mitigated with random walks or simulated annealing.

As a side note, it is also **not advised to use potential functions** as heuristics for A*.

The squared distance means that *h* is on a different scale than *g* and will lead to overestimation of cost which is **not admissible**. This means that A* will tend toward greedy best-first search as distances increase.

94

Note on using Potential Functions with A*

It is also **not advised to use potential functions** as heuristics for A*.

The squared distance means that *h* is on a different scale than *g* and will lead to overestimation of cost which is **not admissible**. This means that A* will tend toward greedy best-first search as distances increase.

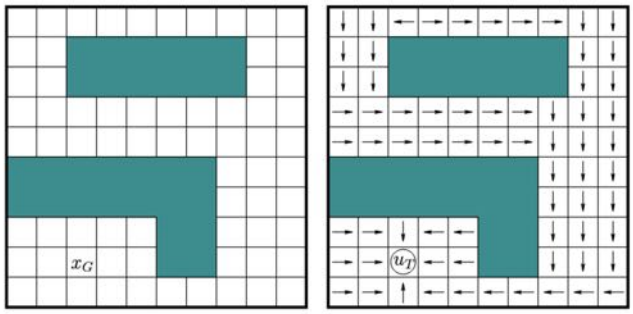
95

Navigation Functions

Navigation Function: Local decision function with the following nice properties that guarantees it always leads you to the goal:

1. $\phi(x) = 0$ for all $x \in X_G$.
2. $\phi(x) = \infty$ if and only if no point in X G is reachable from x.
3. For every reachable state, $x \in X \setminus X_G$, the local operator produces a state x ′ for which $\phi(x') < \phi(x)$.

This last property requires that $\phi(x)$ admit **no local minima**.



96

Cost-to-go as a Navigation Function

A navigation function guides the robot to the goal from **any** state.

Recall: The distance field produced by Dijkstra can produce the shortest path to any state f**rom the initial state**.

⇒ Performing Dijkstra backwards from the goal is a navigation function.

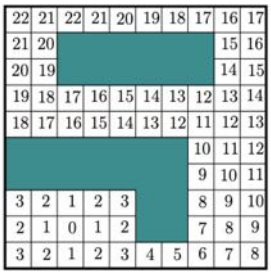


Figure 8.3: The cost-to-go values serve as a navigation function.

Note: If optimality is not important, then any backward search algorithm can be used if it can record the distance to the goal from every reached state

97

Wavefront propagation

If all edges have unit cost, useful simplification can be used to parallelize Dijkstra

Generally want higher grid resolution than just a topological graph

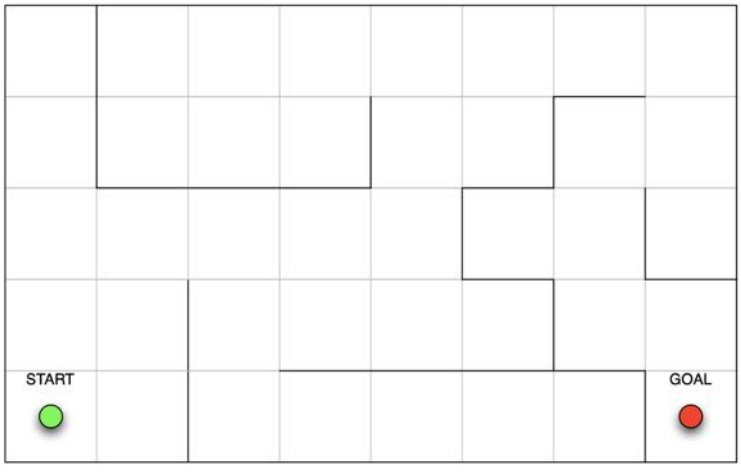
Can also use to get maximum clearance navigation function by propagating out from obstacle boundaries

WAVEFRONT PROPAGATION ALGORITHM

1. Initialize $W_0 = X_G$; $i = 0$.
2. Initialize $W_{i+1} = \emptyset$.
3. For every $x \in W_i$, assign $\phi(x) = i$ and insert all unexplored neighbors of x into W_{i+1}.
4. If $W_{i+1} = \emptyset$, then terminate; otherwise, let $i := i + 1$ and go to Step 2.

98

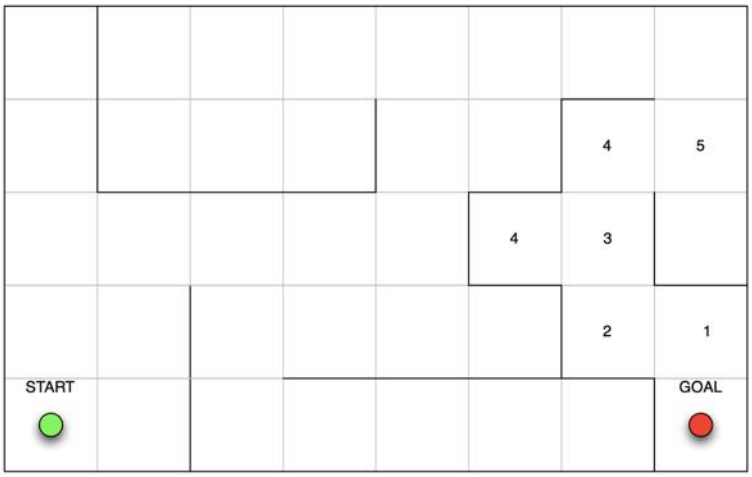
Wavefront propagation



(c) Henrik I Christensen

99

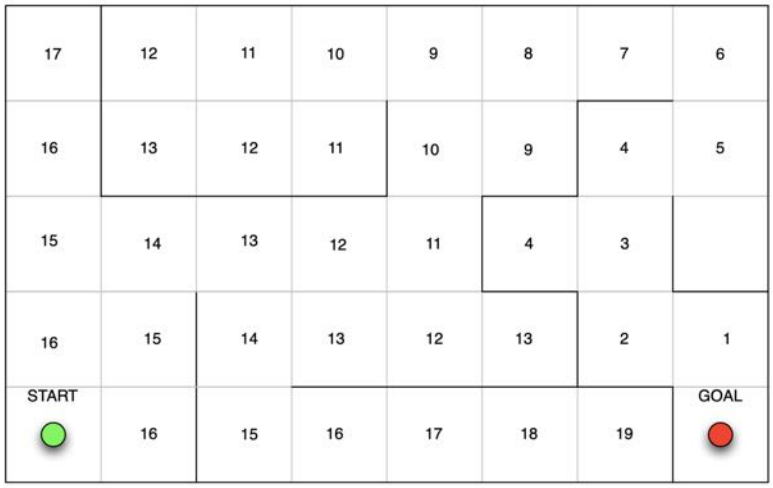
Wavefront propagation



(c) Henrik I Christensen

100

Wavefront propagation



101

Drawbacks of Navigation Functions on Grids

1. Need for very high accuracy, and a hence high-resolution grid may be necessary;
2. Dimension of the configuration space may be high; and
3. If the environment is frequently changing, replanning may take too much time for a real-time response.

Task Planning

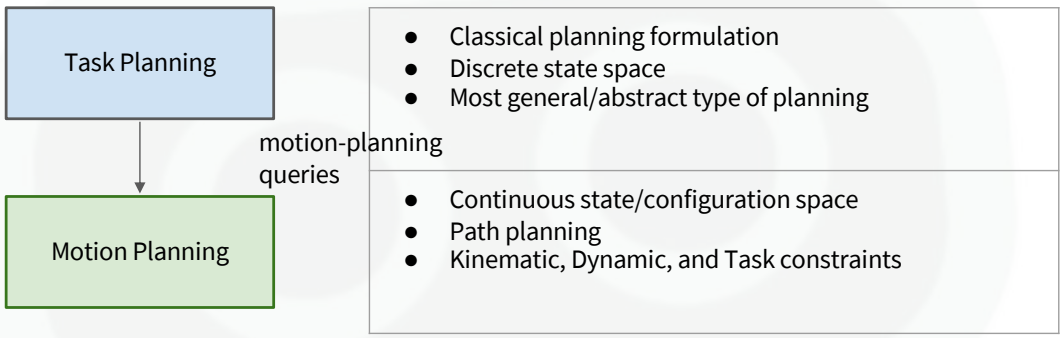
Abstract planning → "problem solving"

Able to handle more complex problems by abstracting away the complexities

Extension of discretization of C-space for motion planning, but for state spaces that don't have nice general, continuous models

105

High- and Low-level Planning



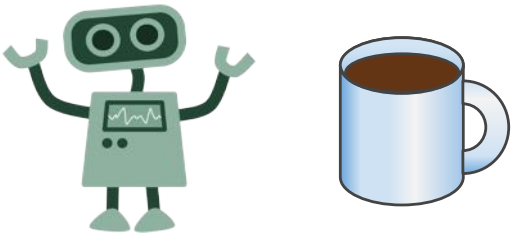
106

Motivational Example

I'm in the lab and I'm feeling a bit drowsy, so I ask my robot to go to the kitchenette and make me a coffee.

Not going to start modeling every detail…

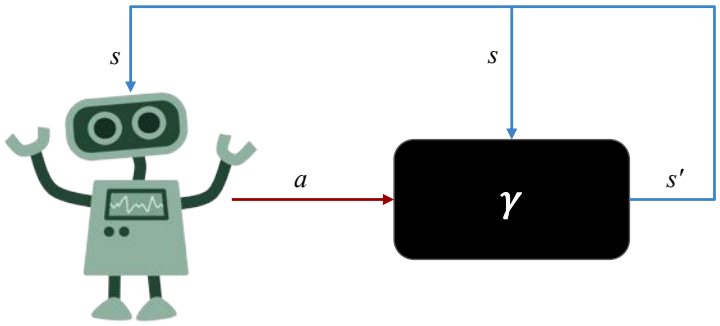
Solution: create an abstract representation that contains *only* the information needed to construct a plan skeleton, then pass that to the motion planner to compute trajectories.



Formulation: State Space Model

A discrete planning domain $\mathcal{D}$ is defined by:

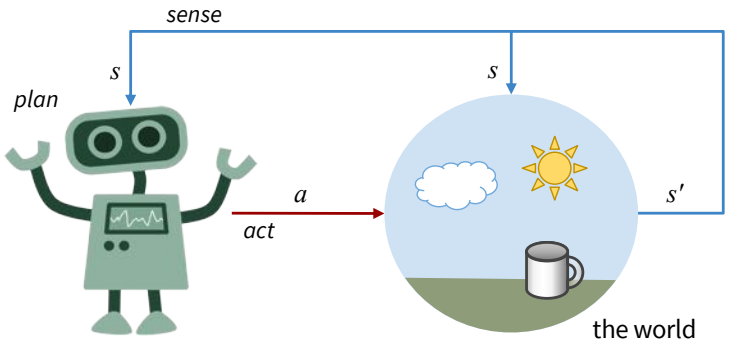
1. A nonempty state space $\mathcal{S}$, composed of a finite or countably infinite set of states s.
2. For each state $s \in \mathcal{S}$, a finite action set $\mathcal{A}(s)$ such that the total action set $\mathcal{A} = \bigcup_{s \in \mathcal{S}} \mathcal{A}(s)$ is nonempty.
3. A state transition function f that produces a state $s' = f(s, a) \in \mathcal{S}$ for every $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}(s)$.



Formulation: State Space Model

A discrete planning domain $\mathcal{D}$ is defined by:

1. A nonempty state space $\mathcal{S}$, composed of a finite or countably infinite set of states s.
2. For each state $s \in \mathcal{S}$, a finite action set $\mathcal{A}(s)$ such that the total action set $\mathcal{A} = \bigcup_{s \in \mathcal{S}} \mathcal{A}(s)$ is nonempty.
3. A state transition function f that produces a state $s' = f(s, a) \in \mathcal{S}$ for every $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}(s)$.



109

Formulation: State Space Model

A discrete planning domain $\mathcal{D}$ is defined by:

1. A nonempty state space $\mathcal{S}$, composed of a finite or countably infinite set of states s.
2. For each state $s \in \mathcal{S}$, a finite action set $\mathcal{A}(s)$ such that the total action set $\mathcal{A} = \bigcup_{s \in \mathcal{S}} \mathcal{A}(s)$ is nonempty.
3. A state transition function f that produces a state $s' = f(s, a) \in \mathcal{S}$ for every $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}(s)$.

A discrete planing problem $\mathcal{P}$ is defined by:

1. A planning domain $\mathcal{D} = (\mathcal{S}, \mathcal{A}, f)$.
2. An initial state $s_I \in \mathcal{S}$.
3. A goal set $S_G \subset \mathcal{S}, s_I \notin S_G$.

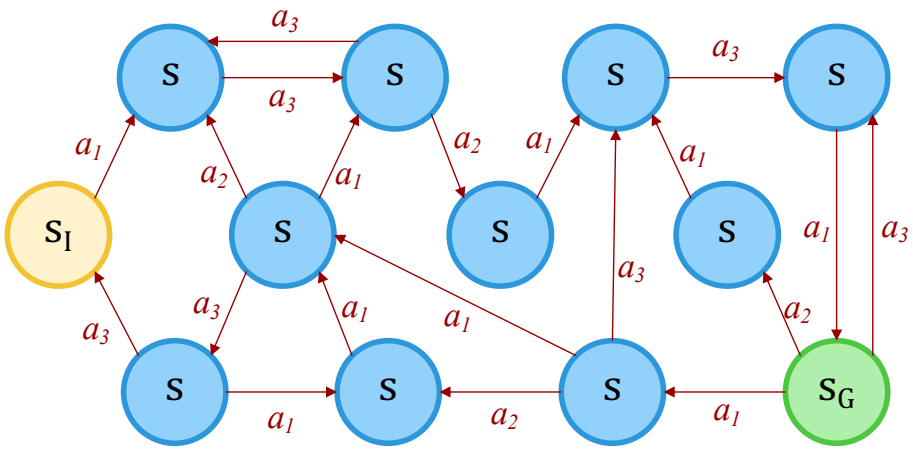
Goal: find a string of actions (the plan) that brings us from our initial state s_I to any goal state $s_G \in S_G$.

110

Graphical View

States = Nodes
- **Initial**
- **Goal**

Actions = Directed Edges



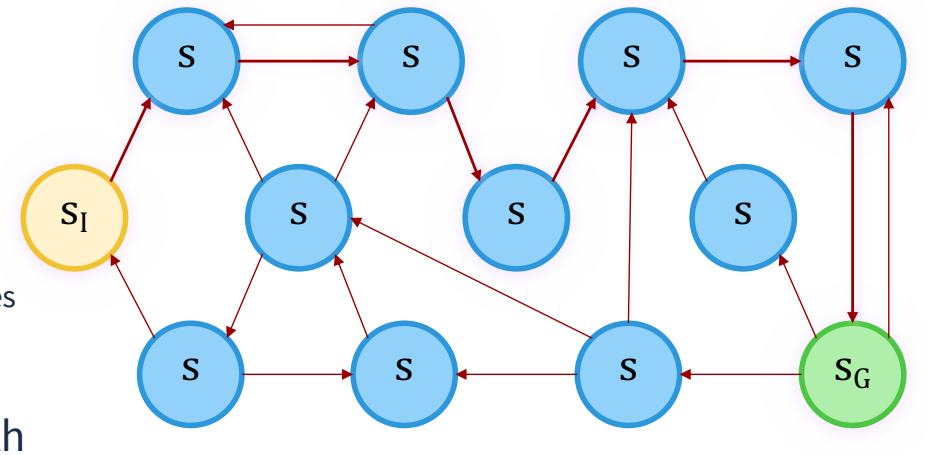
111

Graphical View

States = Nodes
- **Initial**
- **Goal**

Actions = Directed Edges

Plan = Path



112

Logic-Based Formulation

Implicitly represent state space as a collection of logical predicates

Basis of AI research in 1950s-1980s

Can represent huge state spaces extremely compactly

Produce highly interpretable plans, since predicates are written with human-readable names.

Caveat: tend to be difficult to generalize.
- predicates are usually domain-specific,
- state-space representations more

easily extend to continuous spaces, e.g. for motion planning and planning under uncertainty.

Imagine 10 propositions (i.e. ground predicates):

```
brewed(coffee)
contains(mug,coffee)
holding(Robot,mug)
holding(Andi,mug)
inKitchenette(Robot)
inKitchenette(Andi)
inLab(Robot)
inLab(Andi)
isDrowsy(Andi)
isHappy(Andi)
```

which can each be either `True` or `False`.

These 10 statements represent a state space of size **2^{10} potential states**.

STRIPS-like formulation

A STRIPS-like planning domain is represented by:

1. A set of *instances I*,
2. A set of *predicates P*,
3. A set of *operators O*, composed of
   a. *head:* a unique name/identifier,
   b. *preconditions*: positive or negative literals that must hold for the operator to be applied,
   c. *effects:* positive or negative literals that result from applying the operator.

The planning problem is specified by:

1. A STRIPS planning domain *D=(I,P,O)*;
2. An *initial state S* represented by a set of positive literals. Any literal not in *S* is assumed false.
3. A *goal set G*, represented by a set of positive and negative literals.

Instances

The complete set of distinct things that exist in the world that one might want to refer to.

In coffee example:

I = { Andi, Robot, Mug, Coffee }

Predicates

Partial functions from instances to a truth value.

Correspond to basic properties or statements that can be formed about the instances.

In coffee example:

P = { *brewed*(x), *contains*(x,y), *holding*(x,y), *inKitchenette*(x), *inLab*(x), *isDrowsy*(x), *isHappy*(x) }

Operators

Operators represent the ways in which the agent can influence the world state.

An action in a STRIPS planning domain is a ground instance of a planning operator.

Operators can be parameterized by sets of instances, making them partial functions from sets of instances to actions.

pick(obj):
- **preconditions**: ¬*occupied*(robot), *adjacent*(robot,obj)
- **effects**: *holding*(robot,obj), *occupied*(robot,obj)

place(obj,loc):
- **preconditions**: *holding*(robot, obj), *adjacent*(robot,loc), ¬*occupied*(loc)
- **effects**: ¬*holding*(robot,obj), *occupied*(loc)

carry(obj,loc):
- **preconditions**: *holding*(robot, obj)
- **effect**: *at*(robot,loc), *at*(obj,loc)

State space compression

Logical formulation **greatly** compresses the size of the state space representation:

If every predicate has *k* arguments and any instance can appear in any argument, there will be $|\boldsymbol{P}|\,|\boldsymbol{I}|^k$ complementary pairs of ground literals.

This corresponds to a state space of size $2^{|P|\,|I|^k}$!

Original STRIPS planning algorithm aimed to reduce the size of the state transition graph during planning. Unfortunately, the algorithm was shown to not be complete by the Sussman anomaly.