CS 188 Robotics Week 5

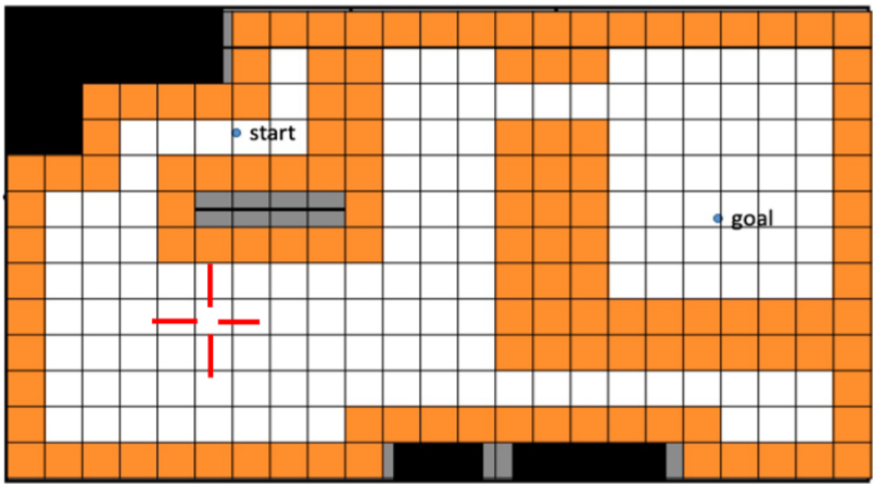
Aidan Jan

May 2, 2025

Motion Planning

Occupancy Grid, accounting for C-Space

Problem: Occupancy grids perform exhaustive search across the state space (i.e., large graph with many nodes and edges.) There are 10^6 nodes for a 6 DoF arm. What can we do differently?



Now we can build a graph:

- Each cell is a node
- Edges between adjacent cells

Task: find the shortest path from the starting cell to the ending cell

- You can use any graph search algorithms:
  - Breadth First Search (Grassfire)
  - Depth First Search
  - A-star Search

Sampling based planning

Rather than exhaustively explore ALL possibilities, randomly explore a smaller subset of possibilities while keeping track of progress. Then, search for collision-free path only on the sampling points.

There are two methods:

1. PRM: Probabilistic Road Map
2. RRT: Rapidly-Exploring Random Tree

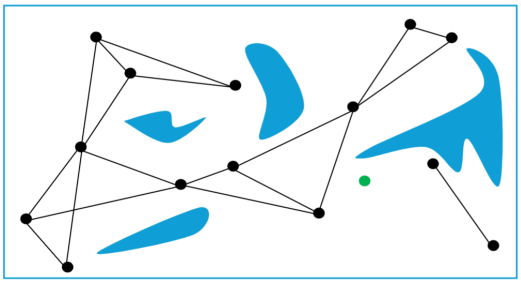
Probabilistic Road Map (PRM)

Probabilistic Roadmap methods proceed in two phases:

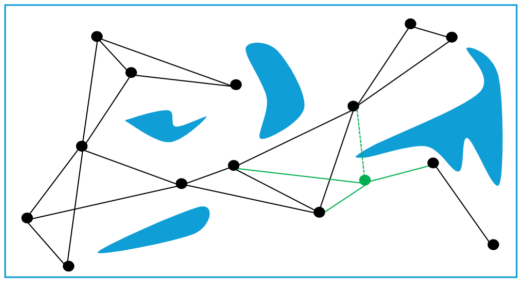
1. Preprocessing Phase - to construct the roadmap G
2. Query (Search) Phase - to search a path from q_init and q_goal

- The roadmap is an undirected graph $G = (N, E)$.
- The nodes in N are a **subset of configurations** of the robot sampled that are **collision free**.
- The edges in E correspond to feasible straight-line paths.

Build-Roadmap:
G.init() ; i = 0
While i < N
  $\alpha \leftarrow$ SAMPLE
  if $\alpha \in C_{free}$
    G.add Vertex(α); i = i+1
    for each $q \in ngd(\alpha, G)$
      if (connect(α , G))
        G.addEdge(α , q)



Build-Roadmap:
G.init() ; i = 0
While i < N
  $\alpha \leftarrow$ SAMPLE
  if $\alpha \in C_{free}$
    G.add Vertex(α); i = i+1
    for each $q \in ngd(\alpha, G)$
      if (connect(α , G))
        G.addEdge(α , q)



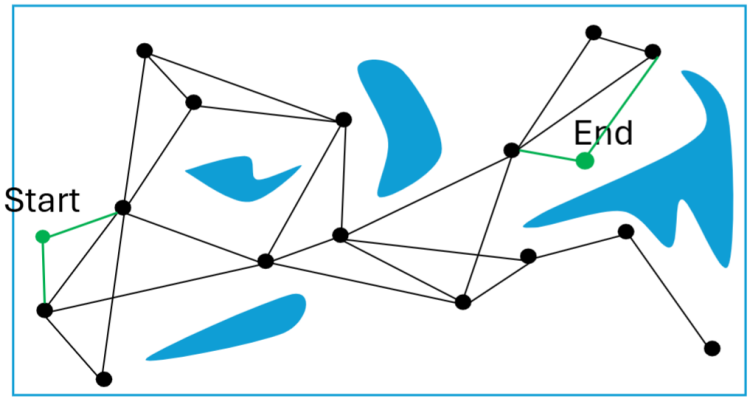
Build-Roadmap:
G.init() ; i = 0
While i < N
  $\alpha \leftarrow$ SAMPLE
  if $\alpha \in C_{free}$
    G.add Vertex(α); i = i+1
    for each $q \in ngd(\alpha, G)$
      if (connect(α , G))
        G.addEdge(α , q)

"neighborhood"

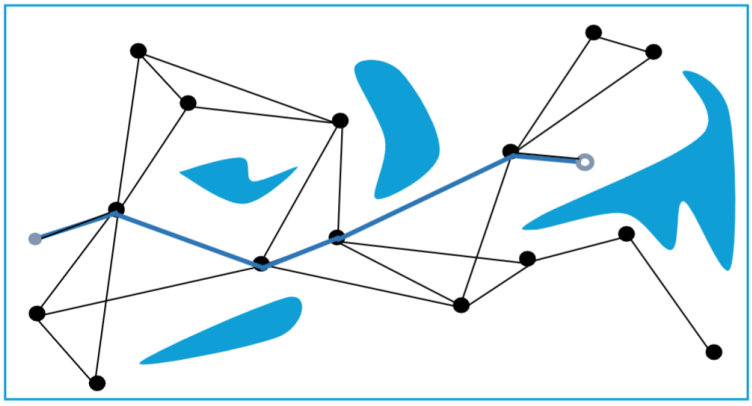
PRM - Phase 2

Query phase: search a path given q_init and q_goal with the constructed graph

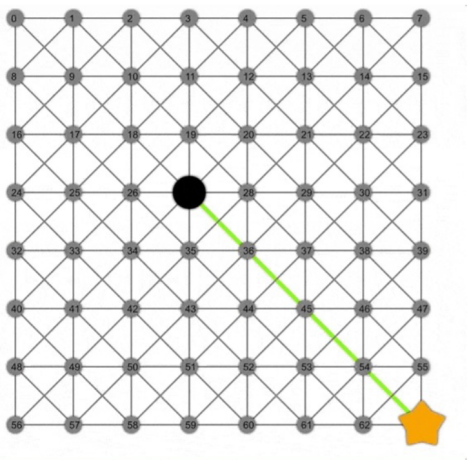
- First, add these two nodes into the graph
- Then, connect the possible edges to existing nodes



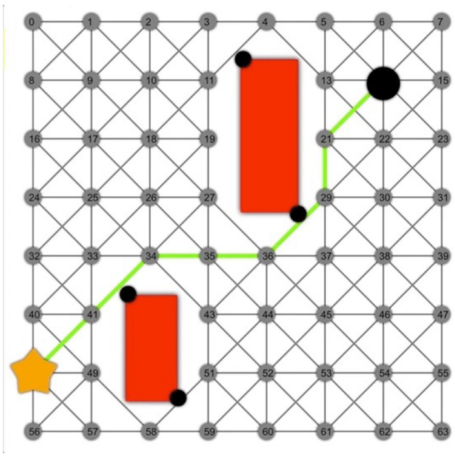
- Finally, perform a graph search



Graph Search



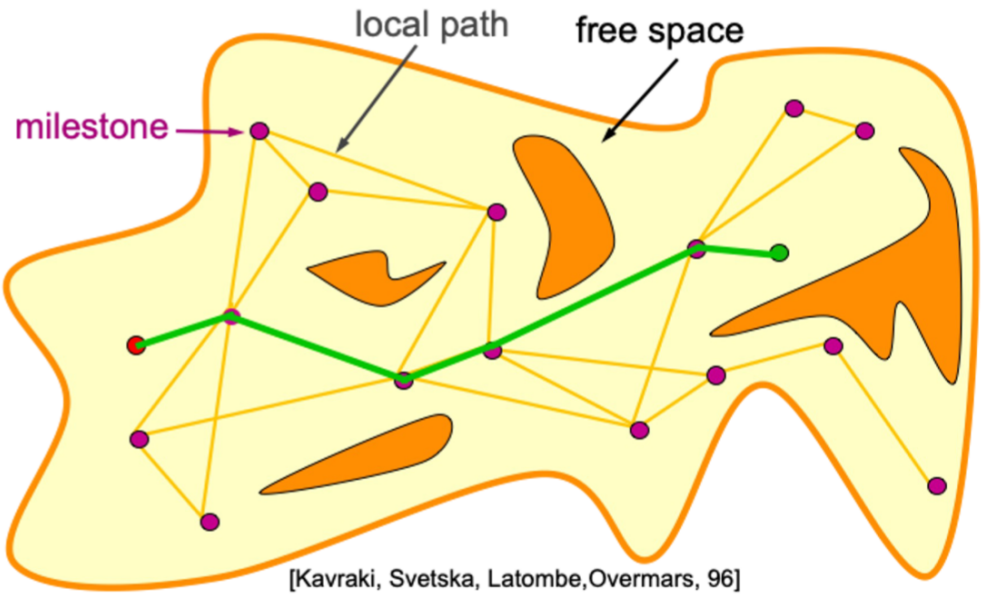
Dijkstra



A*

Aside: Some Terminology

- The graph G is called a **probabilistic roadmap**
- The nodes in G are called **milestones**
- Edge between two milestones called **local path (or roadways)**



A few design decisions:

- How do we sample a new point?
- Randomly or using other "clever" methods? The biggest advantage for random sampling is that there is no bias.
- How to choose the distance of neighbor R? L2 distance? How far?

Defining Distance

- The PRM procedure relies upon a distance function, **Dist**, that can be used to measure the **distance between two points in configuration space.** This function takes as input the coordinates of the two points and returns a real number.

Common choices for distance functions include:

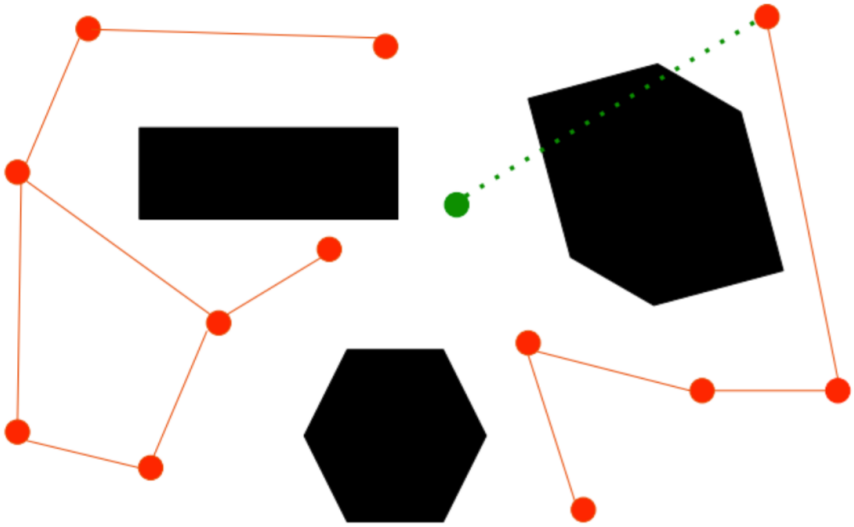
- The L1 distance: $\text{Dist}_1 = \sum_i |x_i - y_i|$
- The L2 distance: $\text{Dist}_2 = \sqrt{\sum_i (x_i - y_i)^2}$

Defining Distance with Angle

- Sometimes, configuration space correspond to angular rotations. In these situations, we want to ensure that the Dist function correctly reflects distances in the presence of wrap around, if there is no joint limit.
- For example, if θ_1 and θ_2 denote two angles between 0 and 360 degrees, the expression below can be used to capture the angular displacement between them.

$$\text{Dist}(\theta_1, \theta_2) = \min(|\theta_1 - \theta_2|, (360 - |\theta_1 - \theta_2|))$$

Check for collision along a path

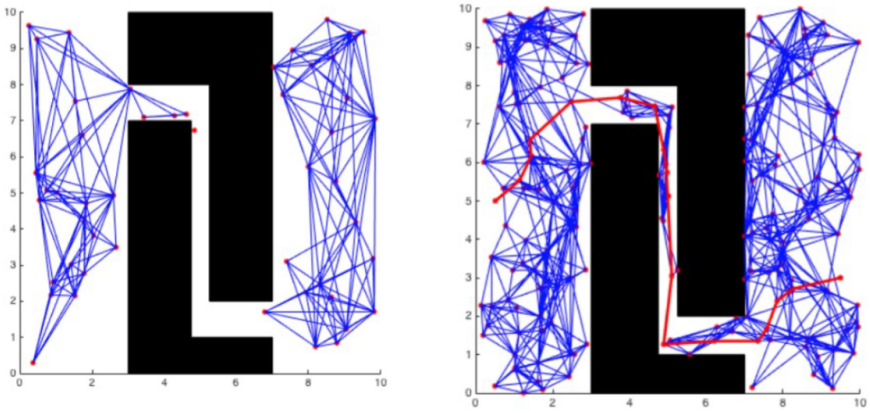


Method:

- Uniformly sample points along the path.
- Check: if all the points are in free space, we assume the path is collision free

Random Sample in PRM

It is possible to have a situation where the algorithm would fail to find a path even when one exists if the sampling procedure fails to generate an appropriate set of samples.



How do we solve this problem?

- More samples
- Sample more points close to C-space obstacles

The only thing we can say is if there is a route and the planner keeps adding random samples, it will, eventually find a solution.

- **In summary: PRM is not complete, but probabilistically complete.**

Summary: Probabilistic

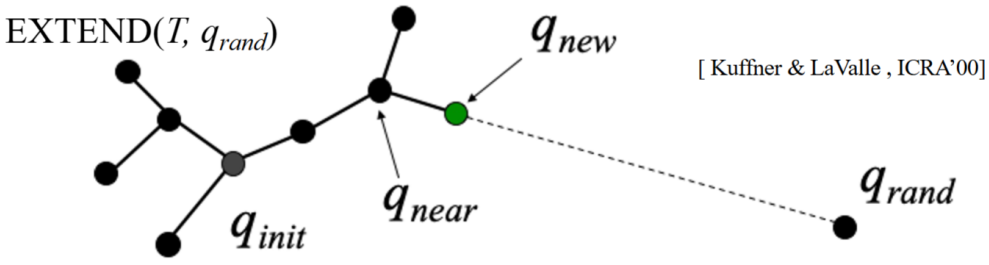
- Assumption:
  - Static obstacles
  - Many queries (start and end state) to be processed in the same environment
- Example applications
  - Navigation in static virtual environments
  - Robot manipulator arm in a static workcell
- **Advantages**: one roadmap that works for many q_init and q_goal, repeatedly
  - But in many cases, we have the start and end point in mind, can we use it to improve the sampling efficiency?
- **Disadvantage**: it spends a lot of time on preprocessing (roadmap construction), and if the environment changes after the road map is constructed, then we need to redo the process.

Rapidly-Exploring Random Tree (RRT)

- Searches for a path from the initial configuration to the goal configuration by expanding a search tree - (single root node, start), no longer a general graph
- For each step,
  - The algorithm samples a **new** configuration and expands the tree towards it.
  - The new sample can either be a **random configuration** or the **goal configuration** itself.

The Basic Idea: Iteratively expand the tree

```
BUILD_RRT (q_init) {
  T.init(q_init);
  for k = 1 to K do
    q_rand = RANDOM_CONFIG();
    EXTEND(T, q_rand)
}
```



RRT requires the following functions

- x_{rand} = RandomSample()
  - Uniform random sampling of free configuration space
  - This can easily be done by sampling randomly in $\mathbb{R}^n$ (or whatever your configuration space is) and then discarding any points which fail the collision check
- $x_{nearest}$ = Nearest(G, x_{rand})
  - Given point in Cspace, find vertex on tree that is closest to that point
- x_{new} = Steer($x_{nearest}, x_{rand}$)
  - Find x_{new} take 'one step' towards x_{rand} from $x_{nearest}$

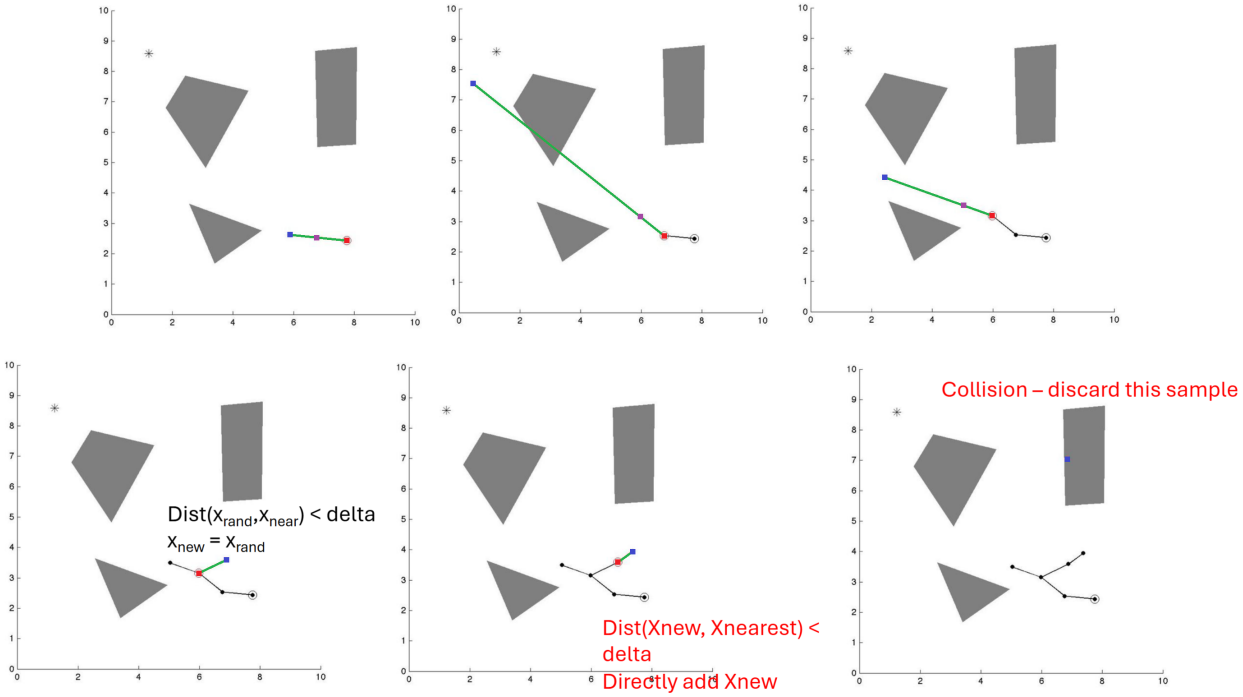
```
if (dist(x_nearest, x_rand) <= delta):
    x_new = x_rand
else:
    compute a new configuration x_new, \
        that is along the path from x_nearest to x_rand, \
        such that dist(x_nearest, x_new) = delta
```

- ObstacleFree(x, y)
  - Check if the path from x to y is collision free
  - Oftentimes, we assume delta is small enough, and we only need to check obstacleFree on the new point x_{new}

RRT Algorithm

Input: Initial configuration x_{init}, number of vertices in tree N, incremental distance delta. Note: end configuration is not part of the input for this version of RRT.

$V \leftarrow \{x_{init}\}$; $E \leftarrow \emptyset$
`for` $i = 1$ `to` N
  $G \leftarrow (V, E)$
  $x_{rand} \leftarrow RandomSample()$
  $x_{nearest} \leftarrow Nearest(G, x_{rand})$
  $x_{new} \leftarrow Steer(x_{nearest}, x_{rand})$
  `if` $ObstacleFree(x_{nearest}, x_{new})$
    $V \leftarrow V \cup \{x_{new}\}$
    $E \leftarrow E \cup \{(x_{nearest}, x_{new})\}$



Basic algorithm is simply exploring space - but we want it to actually head towards the goal! Therefore, we add a bias:

$V \leftarrow \{x_{init}\}; \quad E \leftarrow \emptyset$
for $i = 1$ to N
  $G \leftarrow (V, E)$
  with probability p
    $x_{rand} \leftarrow RandomSample()$
  otherwise
    $x_{rand} \leftarrow x_{goal}$
  $x_{nearest} \leftarrow Nearest(G, x_{rand})$
  $x_{new} \leftarrow Steer(x_{nearest}, x_{rand})$
  if $ObstacleFree(x_{nearest}, x_{new})$
    $V \leftarrow V \cup \{x_{new}\}$
    $E \leftarrow E \cup \{(x_{nearest}, x_{new})\}$

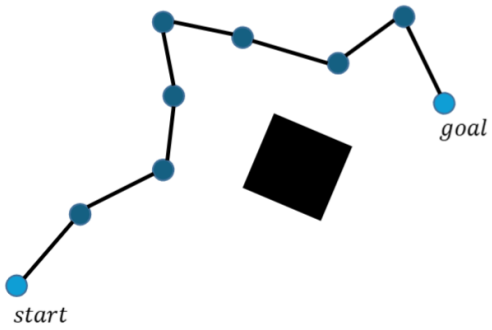
When generating a random sample, with some probability pick the goal instead of a random node when expanding

If Dist(x_{new}, x_{goal}) < delta
    path <- calculate the path from x_{init}, x_{goal}
    return path;

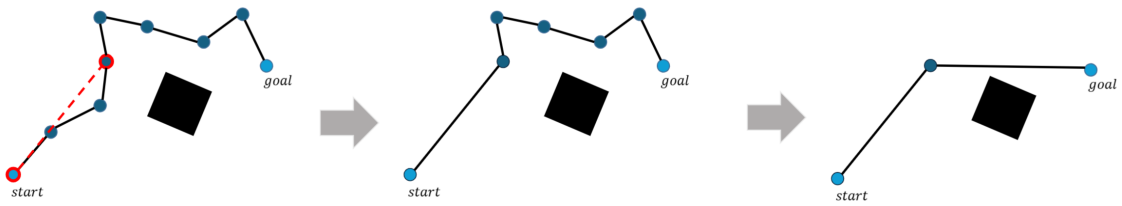
Return when reaches the goal

Optimizing the path

- Milestone-based paths are far from optimal and require additional refinement before they are usable
- A typical solution can look like this:



- A simple way to improve the path (after we find a solution) is to random pick two nodes, and check whether they can be connected by a straight line with collision. If so, use the line to shorten the path.
  - Repeat for N iterations or until no further improvements are being made
  - The result is not an optimal path, but shorter and more efficient than the original



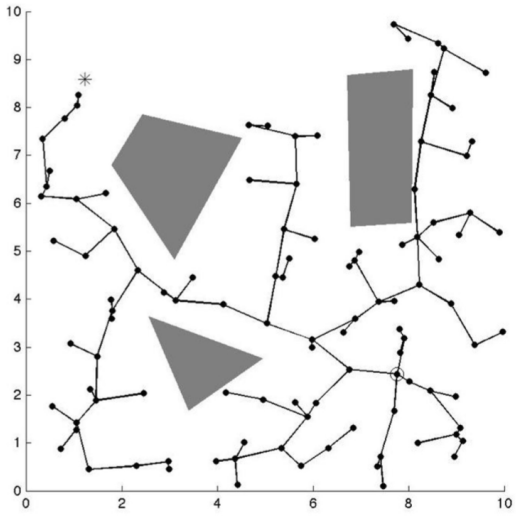
Bidirectional RRT:

- Build two tree A from Start, B from End
- While not done

- Extend Tree A by adding a new node (sample + steer)
- Find the closest node in Tree B to x : y
- If (ObstacleFree(x, y)) - check if you can bridge the two trees
  - Add edge between x and y. This completes a route between the root of Tree A and the root of Tree B.
  - Return this route
- Else
  - Swap Tree A and Tree B (in the next iteration, we will expand tree B)

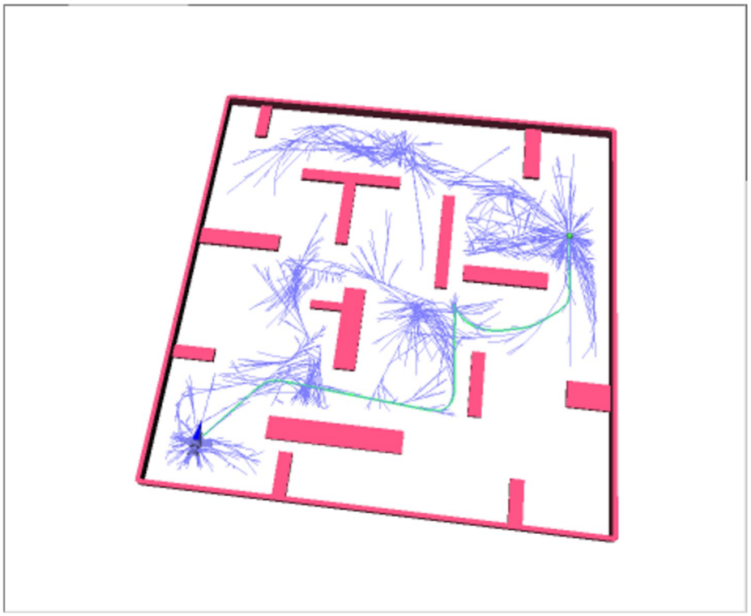
How to change the algorithm to handle high DoF robots?

$V \leftarrow \{x_{init}\}; \quad E \leftarrow \emptyset$
for $i = 1$ to N
  $G \leftarrow (V, E)$
  $x_{rand} \leftarrow RandomSample()$
  $x_{nearest} \leftarrow Nearest(G, x_{rand})$
  $x_{new} \leftarrow Steer(x_{nearest}, x_{rand})$
  if $ObstacleFree(x_{nearest}, x_{new})$
    $V \leftarrow V \cup \{x_{new}\}$
    $E \leftarrow E \cup \{(x_{nearest}, x_{new})\}$



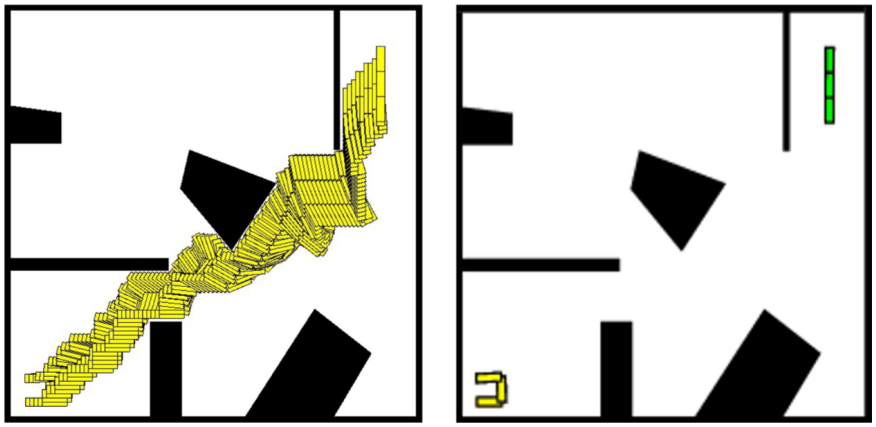
Examples: Hovercraft

What's the DoF of the robot?

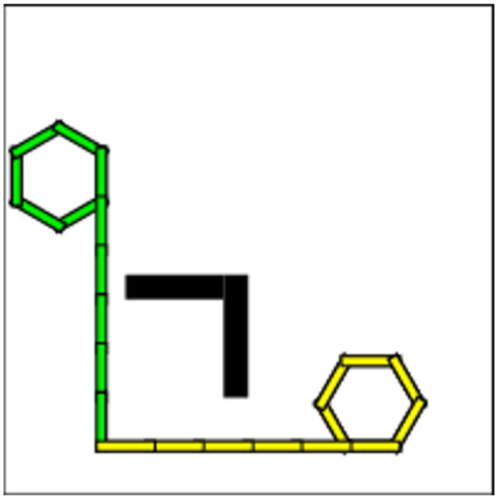


Examples: Articulated Robot

What's the DoF of the robot?



Examples: Highly Articulated Robot



Summary: Rapidly-Exploring Random Tree (RRT)

- Advantages of RRT:
  - very fast, works well for dynamic environments (scene can change between each planning episode, but not within)
- Disadvantages: Not optimal
  - In fact, it has been proven that the probability of RRT converging to an optimal solution is 0.

Markov Decision Processes and Reinforcement Learning

Decision making in deterministic systems

- State: $s_t \in \mathcal{S}$
- Action: $a_t \in \mathcal{A}(s_t)$

- Transition: $s_{t+1} = f_t(s_t, a_t)$
- Total reward:

$$J(s_0; a_0, \ldots, a_{T-1}) = r_T(s_T) + \sum_{t=0}^{T-1} r_t(s_t, a_t)$$

- Decision making problem:

$$J^*(s_0) = \max a_t \in \mathcal{A}(s_t), t = 0, 1, \ldots, T-1 J(s_0; a_0, \ldots, a_{T-1})$$

Principle of optimality

It's the key concept behind the **dynamic programming** approach.

- Suppose $A - B - C$ is the optimal path from A to C.
- First segment reward: J_{AB}
- Second segment reward: J_{BC}
- Optimal reward: $J^*_{AC} = J_{AB} + J_{BC}$

If $A - B - C$ is the optimal path from $A - C$, then $B - C$ is the optimal path from $B - C$.

Proof:

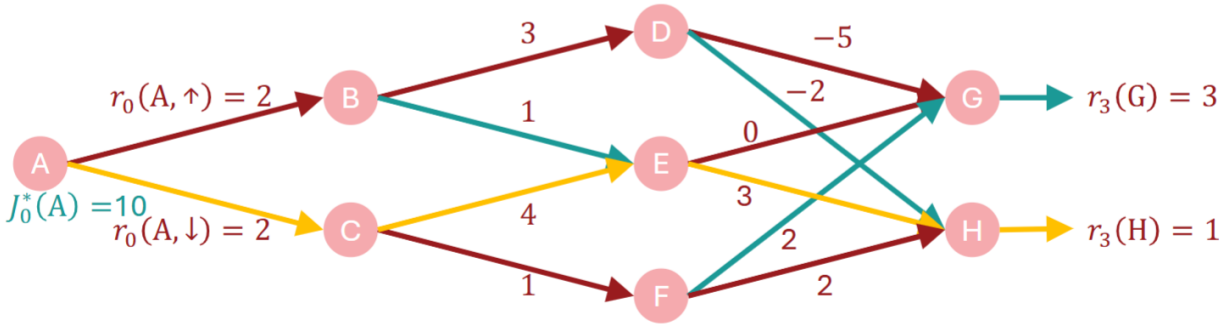
Suppose $(a_0^*, a_1^*, \ldots, a_{T-1}^*)$ is an optimal solution to the decision making problem for an initial state s_0^*, and the systems evolves as $(s_0^*, s_1^*, \ldots, s_T^*)$ for this initial state and action sequence.

Then, an optimal solution to the subproblem for moving from state s_t^* at time t until time T is $(a_t^*, a_{t+1}^*, \ldots, a_{T-1}^*)$

Tail of an optimal solution = Optimal for the tail subproblem

Dynamic Programming (deterministic)

- $J_T^*(s_T) = r_T(s_T)$, for all $s_T \in \mathcal{S}$
- for $t = T - 1$ to 0, do:
  - $J_t^*(s_t) = \max_{a_t \in \mathcal{A}(s_t)} r_t(s_t, a_t) + J_{t+1}^*(f_t(s_t, a_t))$, for all $s_t \in \mathcal{S}$
- return $J_0^*(\cdot), J_1^*(\cdot), \ldots, J_T^*(\cdot)$



Decision Making in Stochastic Systems

Similar to deterministic systems, state and action are the same.

- State: $s_t \in \mathcal{S}$
- Action: $a_t \in \mathcal{A}(s_t)$
- Transition: $s_{t+1} = f_t(s_t, a_t, w_t)$, or $s_{t+1} \sim P(\cdot|s_t, a_t)$. w_t is a random variable.
- Policies: $\pi = (\pi_0, \pi_1, \ldots, \pi_{T-1})$ where $a_t = \pi_t(s_t)$. We introduce policies since we will find an optimal closed-loop policy.
- Total reward:
$$J_\pi(s_0) = \mathbb{E}_{w_0, w_1, \ldots, w_{T-1}} \left[r_T(s_T) + \sum_{t=0}^{T-1} r_t(s_t, \pi_t(s_t), w_t) \right]$$
- Decision making problem:
$$J^*(s_0 = \max_\pi J_\pi(s_0))$$

Principle of Optimality (stochastic)

Suppose $(\pi_0^*, \pi_1^*, \ldots, \pi_{T-1}^*)$ is an optimal solution to the decision making problem and assume state s_t is reachable.

Then, an optimal solution to the subproblem for moving from state s_t at time t until time T is $(\pi_t^*, \pi_{t+1}^*, \ldots, \pi_{T-1}^*)$.

Tail of optimal policies = Optimal for the tail subproblem

Dynamic Programming (stochastic)

- $J_T(s_T) = r_T(s_T)$, for all $s_T \in \mathcal{S}$
- for $t = T - 1$ to 0 do:
  - $J_t(s_t) = \max_{a_t \in \mathcal{A}(s_t)} \mathbb{E}_{w_t} \left[r_t(s_t, a_t, w_t) + J_{t+1}(f_t(s_t, a_t, w_t)) \forall s_t \in \mathcal{S} \right]$
- return $J_0(\cdot), J_1(\cdot), \ldots, J_T(\cdot)$

Markov Decision Processes (MDP)

- DP in stochastic systems suffers from the same problems as DP in deterministic systems.
- Also, modeling transitions perfectly is not always possible.
- MDPs are useful tools to model an agent's interaction with its environment
- Reinforcement learning algorithms try to solve MDPs.
- They have advantages over DP, as they only need a reward function.

Infinite Horizon MDP's

We are looking at the infinite horizon case, since it makes stationary policies optimal.

- State: $s \in \mathcal{S}$
- Action: $a \in \mathcal{A}$. We removed the dependence on the state, although that also creates interesting research questions.
- Transition: $s_{t+1} \sim P(\cdot|s_t, a_t)$
- Reward: $r_t = R(s_t, a_t)$
- Discount: $\gamma \in [0, 1)$
- Policy: $\pi : \mathcal{S} \to \mathcal{A}$ or $\pi : \mathcal{S} \to \Delta\mathcal{A}$
- Goal: (Reinforcement learning tries to solve this problem)

$$\pi^* = \arg\max_\pi \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t))\right]$$

Example MDP

- Goal: Achieve a high score in the Atari game "Breakout"
- States: Image of current screen (?)
- Actions: Left or right keys
- Reward: Change in the score of the game

Another Example MDP

- Goal: Make an RC helicopter fly and perform some maneuvers
- States: Sensory input of the helicopter
- Actions: Control inputs
- Reward: Positive for the manuevers, negative for crashing (This is usually what we need to hand-design)
  - This is a very naive reward function. They instead learned the reward from expert demostrations.

Partially Observable MDP's

- State: $s \in \mathcal{S}$
- Action: $a \in \mathcal{A}$
- Observation: $o \in \mathcal{O}$
- Observation Model: $o_t \sim \Omega(\cdot|s_t)$
- Transition: $s_{t+1} \sim P(\cdot|s_t, a_t)$
- Reward: $r_t = R(s_t, a_t)$
- Discount: $\gamma \in [0, 1)$
- Policy: $\pi : \mathcal{O} \to \mathcal{A}$ or $\pi : \mathcal{O} \to \Delta\mathcal{A}$
- Goal:

$$\pi^* = \arg\max_\pi \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(o_t))\right]$$

Value Functions

- State value function: $V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s\right]$
- State-action value function: $Q^\pi(s_a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a\right]$

$$V^\pi(s) = R(s, \pi(s)) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,\pi(s))}[V^\pi(s')]$$
$$Q^\pi(s, a) = R(s, a) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,a), a' \sim \pi(s')}[Q^\pi(s', a')]$$

For any stationary policy, these have unique solutions. Hint: Think of it as a system of linear equations.

These equations can be used to derive the **Bellman equations**.

$$V^*(s) = \max_a \left(R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V^*(s') \right) Q^*(s, a) \quad = R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) \max_{a'} Q^*(s', a')$$

Value Iteration

- Idea: Take Bellman equation and iterate until it converges. It does converge because it is a contractive mapping.
- $V_0(s) = 0 \forall s \in \mathcal{S}$
- for $k = 0, 1, \ldots$ until convergence:
  - for all $s \in \mathcal{S}$:
    - $V_{k+1}(s) = \max_a \left(R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V_k(s') \right)$
- Each iteration is $\mathrm{O}(|\mathcal{S}|^2 |\mathcal{A}|^2)$

Policy Iteration

- Initialize a random policy π_0.
- for $k = 0, 1, \ldots,$ until convergence:
  - Solve the following system for V^{π_k}:
    - $V^{\pi_k}(s) = \mathbb{E}_{a \sim \pi_k(s)} \left[R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V^{\pi_k}(s') \right]$
    - // This is called policy evaluation. It is $O(|\mathcal{S}|^2)$.
  - for all $s \in \mathcal{S}$
    - $\pi_k(s) = \arg\max_a \left(R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V^{\pi_k}(s') \right)$
    - // This is called policy improvement. It is $O(|\mathcal{S}|^2 |\mathcal{A}|)$

Value Iteration vs Policy Iteration

- Both converge.
- Policy iteration requires more complex implementation
- In practice, policy iteration usually converges faster.

Feature	**Dynamic Programming (DP)**	**Reinforcement Learning (RL)**
Model Assumption	Requires a complete and accurate model of the environment (transition probabilities and rewards).	Does **not** require a model; learns from **interaction** with the environment.
Environment Knowledge	Known and fully specified (i.e., Markov Decision Process is explicit).	Environment is **initially unknown** and must be explored.
Learning Method	Solves problems analytically (e.g., via **Bellman equations**).	Learns policies or value functions via **sampling** and **trial-and-error**.
Typical Algorithms	Value Iteration, Policy Iteration.	Q-learning, SARSA, Deep Q-Networks (DQB), Policy Gradient methods.
Computation Style	Often requires full sweeps over state space (offline, batch)	Can be **online**, updating based on individual experiences.
Use Case	Works best when the environment is small and fully known.	Scales better to **unknown or complex** environments, like games, robotics.

Evolution of Reinforcement Learning

1. Psychological Foundations (1900s - 1950s)

- Edward Thorndike (1898): Proposed the *Law of Effect* - actions followed by rewards are more likely to be repeated. A core RL idea.
- B.F. Skinner (1930s - 1950s): Developed operant conditioning - learning by interacting with the environment and receiving rewards/punishments
- These early concepts inspired the *trial and error learning* at the heart of RL.

2. Control Theory and Early AI (1950s - 1970s)

- Richard Bellman (1957): Introduced **Dynamic Programming** and the **Bellman equation**, foundational for RL. Formalized optimal decision-making in MDPs.
- Arthur Samuel (1959): Built one of the first learning programs (checkers), which used ideas akin to RL.
- DP provided a mathematical framework; AI began applying learning to games.

Temporal-Difference Learning and Modern Formulation (1980s)

- Andrew Barto, Richard Sutton, and Charles Anderson: Introduced **Temporal-Difference (TD) learning** (1983), which combined ideas from DP and supervised learning.
- Sutton's 1988 paper: Formalized **TD learning**, a core technique in RL.
- Watkins (1989): Developed **Q-learning**, a model-free RL algorithm.
- This era established RL as a distinct field with foundational algorithms.

RL Meets Machine Learning (1990s - 2000s)

- **Policy Gradient methods** introduced for continuous action spaces.
- Applications expanded to robotics, games, and scheduling.
- **TD-Gammon (1992)**: Gerald Tesauro's backgammon player used TD learning and beat top human players - a milestone for RL in practice.

Deep Reinforcement Learning (2010s - Present)

- **Deep Q-Networks (DQN)** by DeepMind (2013 - 2015): Combined Q-learning with deep neural networks, achieving human-level play in Atari games.
- **AlphaGo (2016)**: Combined deep RL with Monte Carlo Tree Search to beat world champions in Go.
- **Proximal Policy Optimization (PPO), A3C, SAC**: New policy optimization methods made training more stable and efficient.
- **OpenAI Five (2019)**: Used RL to master Dota 2 in a complex multi-agent environment.