Sampling-Based Motion Planning

Pieter Abbeel
UC Berkeley EECS

Many images from Lavalle, Planning Algorithms
Motion Planning

- **Problem**
  - Given start state $x_S$, goal state $x_G$
  - Asked for: a sequence of control inputs that leads from start to goal

- **Why tricky?**
  - Need to avoid obstacles
  - For systems with underactuated dynamics: can’t simply move along any coordinate at will
    - E.g., car, helicopter, airplane, but also robot manipulator hitting joint limits
Solve by Nonlinear Optimization for Control?

- Could try by, for example, following formulation:

\[
\min_{u,x} \quad (x_T - x_G)^\top (x_T - x_G) \\
\text{s.t.} \quad x_{t+1} = f(x_t, u_t) \quad \forall t \\
u_t \in \mathcal{U}_t \\
x_t \in \mathcal{X}_t \\
x_0 = x_S
\]

\(\mathcal{X}_t\) can encode obstacles

- Or, with constraints, (which would require using an infeasible method):

\[
\min_{u,x} \quad \|u\| \\
\text{s.t.} \quad x_{t+1} = f(x_t, u_t) \quad \forall t \\
u_t \in \mathcal{U}_t \\
x_t \in \mathcal{X}_t \\
x_0 = x_S \\
X_T = x_G
\]

- Can work surprisingly well, but for more complicated problems with longer horizons, often get stuck in local maxima that don’t reach the goal
Examples

- Helicopter path planning
- Swinging up cart-pole
- Acrobot
Examples
Examples
Examples
Motion Planning: Outline

- Configuration Space
- Probabilistic Roadmap
  - Boundary Value Problem
  - Sampling
  - Collision checking
- Rapidly-exploring Random Trees (RRTs)
- Smoothing
Configuration Space (C-Space)

\[ \{ x \mid x \text{ is a pose of the robot} \} \]

- obstacles \( \rightarrow \) configuration space obstacles

**Workspace**

**Configuration Space**

(2 DOF: translation only, no rotation)
Motion planning
Probabilistic Roadmap (PRM)

Space $\mathbb{R}^n$ 
forbidden space

Free/feasible space
Probabilistic Roadmap (PRM)

Configurations are sampled by picking coordinates at random
Probabilistic Roadmap (PRM)

Configurations are sampled by picking coordinates at random
Probabilistic Roadmap (PRM)

Sampled configurations are tested for collision
Probabilistic Roadmap (PRM)

The collision-free configurations are retained as milestones
Probabilistic Roadmap (PRM)

Each milestone is linked by straight paths to its nearest neighbors
Probabilistic Roadmap (PRM)

Each milestone is linked by straight paths to its nearest neighbors
Probabilistic Roadmap (PRM)

The collision-free links are retained as local paths to form the PRM
Probabilistic Roadmap (PRM)

The start and goal configurations are included as milestones.
Probabilistic Roadmap (PRM)

The PRM is searched for a path from $s$ to $g$
Probabilistic Roadmap

- Initialize set of points with $x_S$ and $x_G$
- Randomly sample points in configuration space
- Connect nearby points if they can be reached from each other
- Find path from $x_S$ to $x_G$ in the graph

- Alternatively: keep track of connected components incrementally, and declare success when $x_S$ and $x_G$ are in same connected component
PRM example
PRM example 2
Sampling

- How to sample uniformly at random from $[0,1]^n$?
  - Sample uniformly at random from $[0,1]$ for each coordinate

- How to sample uniformly at random from the surface of the n-D unit sphere?
  - Sample from n-D Gaussian $\rightarrow$ isotropic; then just normalize

- How to sample uniformly at random for orientations in 3-D?
PRM: Challenges

1. Connecting neighboring points: Only easy for holonomic systems (i.e., for which you can move each degree of freedom at will at any time). Generally requires solving a Boundary Value Problem

\[
\min_{u,x} \quad \|u\|
\text{ s.t. } \quad x_{t+1} = f(x_t, u_t) \quad \forall t \\
 u_t \in U_t \\
 x_t \in X_t \\
 x_0 = x_S \\
 X_T = x_G
\]

Typically solved without collision checking; later verified if valid by collision checking

2. Collision checking:

Often takes majority of time in applications (see Lavalle)
PRM’s Pros and Cons

- **Pro:**
  - Probabilistically complete: i.e., with probability one, if run for long enough the graph will contain a solution path if one exists.

- **Cons:**
  - Required to solve 2 point boundary value problem
  - Build graph over state space but no particular focus on generating a path
Rapidly exploring Random Trees

- Basic idea:
  - Build up a tree through generating “next states” in the tree by executing random controls
  - However: not exactly above to ensure good coverage
Rapidly-exploring Random Trees (RRT)

\[
\text{GENERATE_RRT}(x_{\text{init}}, K, \Delta t)
\]

1. \( T.\text{init}(x_{\text{init}}); \)
2. for \( k = 1 \) to \( K \) do
3. \( x_{\text{rand}} \leftarrow \text{RANDOM\_STATE}(); \)
4. \( x_{\text{near}} \leftarrow \text{NEAREST\_NEIGHBOR}(x_{\text{rand}}, T); \)
5. \( u \leftarrow \text{SELECT\_INPUT}(x_{\text{rand}}, x_{\text{near}}); \)
6. \( x_{\text{new}} \leftarrow \text{NEW\_STATE}(x_{\text{near}}, u, \Delta t); \)
7. \( T.\text{add\_vertex}(x_{\text{new}}); \)
8. \( T.\text{add\_edge}(x_{\text{near}}, x_{\text{new}}, u); \)
9. \( \text{Return } T \)

\text{RANDOM\_STATE}(): often uniformly at random over space with probability 99%, and the goal state with probability 1%, this ensures it attempts to connect to goal semi-regularly
**RRT Practicalities**

- **NEAREST_NEIGHBOR**($x_{\text{rand}}, T$): need to find (approximate) nearest neighbor efficiently
  - KD Trees data structure (upto 20-D) [e.g., FLANN]
  - Locality Sensitive Hashing

- **SELECT_INPUT**($x_{\text{rand}}, x_{\text{near}}$)
  - Two point boundary value problem
    - If too hard to solve, often just select best out of a set of control sequences. This set could be random, or some well chosen set of primitives.
RRT Extension

- No obstacles, holonomic:

- With obstacles, holonomic:

- Non-holonomic: approximately (sometimes as approximate as picking best of a few random control sequences) solve two-point boundary value problem
Growing RRT

Demo: http://en.wikipedia.org/wiki/File:Rapidly-exploring_Random_Tree_(RRT)_500x373.gif
Bi-directional RRT

- Volume swept out by unidirectional RRT:

- Volume swept out by bi-directional RRT:

- Difference becomes far more pronounced in higher dimensions
Multi-directional RRT

- Planning around obstacles or through narrow passages can often be easier in one direction than the other
Resolution-Complete RRT (RC-RRT)

- Issue: nearest points chosen for expansion are (too) often the ones stuck behind an obstacle

RC-RRT solution:

- Choose a maximum number of times, m, you are willing to try to expand each node
- For each node in the tree, keep track of its Constraint Violation Frequency (CVF)
- Initialize CVF to zero when node is added to tree
- Whenever an expansion from the node is unsuccessful (e.g., per hitting an obstacle):
  - Increase CVF of that node by 1
  - Increase CVF of its parent node by 1/m, its grandparent 1/m^2, …
- When a node is selected for expansion, skip over it with probability CVF/m
Algorithm 6: RRT*

1. $V \leftarrow \{x_{\text{init}}\}; E \leftarrow \emptyset$;
2. for $i = 1, \ldots, n$ do
   3. $x_{\text{rand}} \leftarrow \text{SampleFree}_i$;
   4. $x_{\text{nearest}} \leftarrow \text{Nearest}(G = (V, E), x_{\text{rand}})$;
   5. $x_{\text{new}} \leftarrow \text{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
   6. if $\text{ObstacleFree}(x_{\text{nearest}}, x_{\text{new}})$ then
      7. $X_{\text{near}} \leftarrow \text{Near}(G = (V, E), x_{\text{new}}, \min\{\gamma_{\text{RRT}}(\log(\text{card}(V))/\text{card}(V))^{1/d}, \eta\})$;
      8. $V \leftarrow V \cup \{x_{\text{new}}\}$;
      9. $x_{\text{min}} \leftarrow x_{\text{nearest}}; c_{\text{min}} \leftarrow \text{Cost}(x_{\text{nearest}}) + c(\text{Line}(x_{\text{nearest}}, x_{\text{new}}))$;
     10. foreach $x_{\text{near}} \in X_{\text{near}}$ do // Connect along a minimum-cost path
         11. if $\text{CollisionFree}(x_{\text{near}}, x_{\text{new}}) \land \text{Cost}(x_{\text{near}}) + c(\text{Line}(x_{\text{near}}, x_{\text{new}})) < c_{\text{min}}$ then
            12. $x_{\text{min}} \leftarrow x_{\text{near}}; c_{\text{min}} \leftarrow \text{Cost}(x_{\text{near}}) + c(\text{Line}(x_{\text{near}}, x_{\text{new}}))$
     13. $E \leftarrow E \cup \{(x_{\text{min}}, x_{\text{new}})\}$;
     14. foreach $x_{\text{near}} \in X_{\text{near}}$ do // Rewire the tree
         15. if $\text{CollisionFree}(x_{\text{new}}, x_{\text{near}}) \land \text{Cost}(x_{\text{new}}) + c(\text{Line}(x_{\text{new}}, x_{\text{near}})) < \text{Cost}(x_{\text{near}})$
            then $x_{\text{parent}} \leftarrow \text{Parent}(x_{\text{near}})$;
         16. $E \leftarrow (E \setminus \{(x_{\text{parent}}, x_{\text{near}})\}) \cup \{(x_{\text{new}}, x_{\text{near}})\}$
17. return $G = (V, E)$;

Source: Karaman and Frazzoli
RRT*

- Asymptotically optimal

- Main idea:
  - Swap new point in as parent for nearby vertices who can be reached along shorter path through new point than through their original (current) parent
RRT

RRT*

Source: Karaman and Frazzoli
RRT*

Source: Karaman and Frazzoli
LQR-trees (Tedrake, IJRR 2010)

- Idea: grow a randomized tree of stabilizing controllers to the goal
  - Like RRT
  - Can discard sample points in already stabilized region
Algorithm 1 LQR-tree \((f, x_G, u_G, Q, R)\)

1: \([A, B] \leftarrow \text{linearization of } f(x, u) \text{ around } (x_G, u_G)\)
2: \([K, S] \leftarrow \text{LQR}(A, B, Q, R)\)
3: \(\rho_c \leftarrow \text{level set computed as described in } \S 3.1.1\)
4: \(\text{T.init}([x_g, u_g, S, K, \rho_c, \text{NULL}])\)
5: \(\text{for } k = 1 \text{ to } K \text{ do}\)
6: \(x_{\text{rand}} \leftarrow \text{random sample}\)
7: \(\text{if } x_{\text{rand}} \in C_k \text{ then}\)
8: \(\text{continue}\)
9: \(\text{end if}\)
10: \([t, x_0(t), u_0(t)] \text{ from trajectory optimization with a} \)
\(\text{“final tree constraint”}\)
11: \(\text{if } x_0(t_f) \notin T_k \text{ then}\)
12: \(\text{continue}\)
13: \(\text{end if}\)
14: \([K(t), S(t)] \text{ from time-varying LQR}\)
15: \(\rho_c \leftarrow \text{level set computed as in } \S 3.1.1\)
16: \(i \leftarrow \text{pointer to branch in } T \text{ containing } x_0(t_f)\)
17: \(\text{T.add-branch}(x_0(t), u_0(t), S(t), K(t), \rho_c, i)\)
18: \(\text{end for}\)

C_k: stabilized region after iteration \(k\)
LQR-trees (Tedrake)

(a) Goal region

(b) One branch

(c) Six branches

(d) Thirteen branches
Randomized motion planners tend to find not so great paths for execution: very jagged, often much longer than necessary.

→ In practice: do smoothing before using the path

- Shortcutting:
  - along the found path, pick two vertices $x_{t_1}, x_{t_2}$ and try to connect them directly (skipping over all intermediate vertices)

- Nonlinear optimization for optimal control
  - Allows to specify an objective function that includes smoothness in state, control, small control inputs, etc.