L06: Planning in continuous spaces

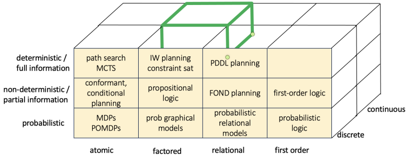
AIMA4e: Chapter 4.2; 26.5.1, 26.5.2 26.5.4
(skim traj opt part–don't panic!)

What you should know after this lecture

- Completely general planning in continuous spaces is very hard.
- Local search methods (gradient-based or “evolutionary”) are general-purpose but suffer badly from local optima.
- In geometric problems, such as robot motion planning, there are good solutions, particularly sampling-based strategies.

Continuous path-search problems

Factored, continuous states,
usually in $\mathbb{R}^n$
Smoothness of some kind in T



Problem formulation

- We will assume discrete decision epochs (continuous time important but we won't study it)
  - State set: $\mathcal{S} \subset \mathbb{R}^D$ can include angles, positions, velocities, etc. (be careful with angles!!)
  - Initial state: $s_0 \in \mathcal{S}$
  - Action set: $\mathcal{A}$
    - If discrete, then use regular forward path search
    - Generally, in some other space $\mathcal{A} \subset \mathbb{R}^A$
    - Important common special case: $\mathcal{A} = \mathcal{S}$
  - Transition model: $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$
    - Often T is smooth (differentiable)
    - A very common special case when $\mathcal{A} = \mathcal{S}$ is discontinuous: $T(s, a) = a$ except when (s, a) is "blocked" in which case $T(s, a) = s$
  - Goal set: $G \subset \mathcal{S}$
  - Cost function $C : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$
- We need to find next action to take. Sometimes the number of steps is fixed, sometimes not.
- Find plan $a_0, \ldots, a_{k-1}$ from s_0 to some state in G such that $T(s_0, a_0) = s_1, \ldots, T(s_{k-1}, a_{k-1}) = s_k$ and $s_m \in G$
- We try to minimize $\sum_i C(s_i, a_i)$ but may only approximate.

Trajectory optimization

Assume target $g \in \mathcal{S}$ and additional cost $l_f(s_k, g)$ for reaching final state s_k. Often $l_f(s_k, g) = \|s_k - g\|_2$.

Direct Shooting

- Choose $a_0, \ldots a_{k-1}$ to minimize

$$l_f(s_k, g) + \sum_{j=0}^{k-1} c(a_j)$$

  where $s_k = T(T(\ldots T(T(s_0, a_0), a_1) \ldots), a_{k-1})$
- Can be hard to optimize–gradient is weak

Direct Transcription

- Add explicit variables to optimization problem for s_j
- Choose $a_0, \ldots a_{k-1}, s_1, \ldots, s_k$ to minimize

$$l_f(s_k, g) + \sum_{j=0}^{k-1} c(a_j) + l(T(s_j, a_j), s_{j+1})$$

Optimize using gradient methods. Local optima can be bad.

Robot motion planning: abstract formulation

An important special case with algorithms that exploit its structure

- Let $\mathcal{S}$ be the set of configurations of a robot
- Assume you are given a map of obstacles in the 3D world
- You want to find a collision-free path between a starting and ending state of the robot
- Let $\mathcal{A} = \mathcal{S}$ and assume
  - $\mathsf{T}(s, s') = s'$ if there are no obstacles on a (often linear) path between s and s'
  - $\mathsf{T}(s, s') = s$ otherwise (not worth considering)

This formulation is reasonable for holonomic systems that can directly make incremental motions in all dimensions of $\mathcal{S}$. Needs to be extended to handle, e.g., cars.

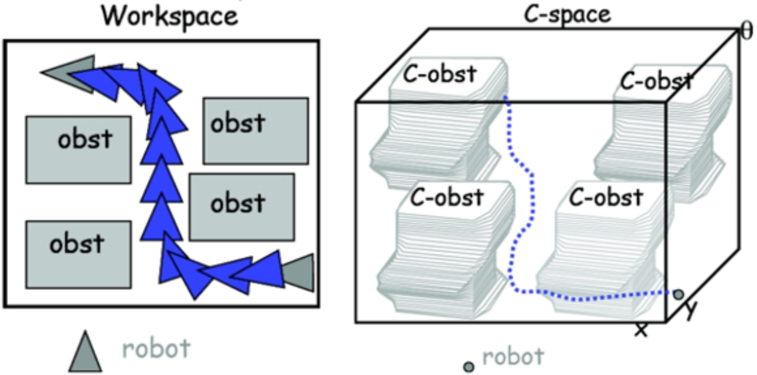
Robot motion planning, more concretely

- Consider a robot that's made up of a collection of rigid bodies with joints between them
  - Joint types: rotational, prismatic (sliding), free (mobile robot)
  - Joint limits: some rotational joints can go all the way around, but generally there are limits
  - Ignore dynamics, but we still have to think about what "motors" we have: differential drive vs omni-directional robot base
- Environment is some bounded 2D or 3D space (called the "workspace") with some immovable obstacles in it.
- configuration is a vector of positions of all the joints $q \in Q$
- motion planning problem: given two configurations q_s and q_g, is there collision-free trajectory?
  - trajectory: continuous function $f : [0, 1] \to Q$
  - $f(0) = q_s$ and $f(1) = 1_g$
  - collision-free: for all values of $f(t)$, if the robot is in that configuration it does not collide with any obstacle (or itself)

Configuration space

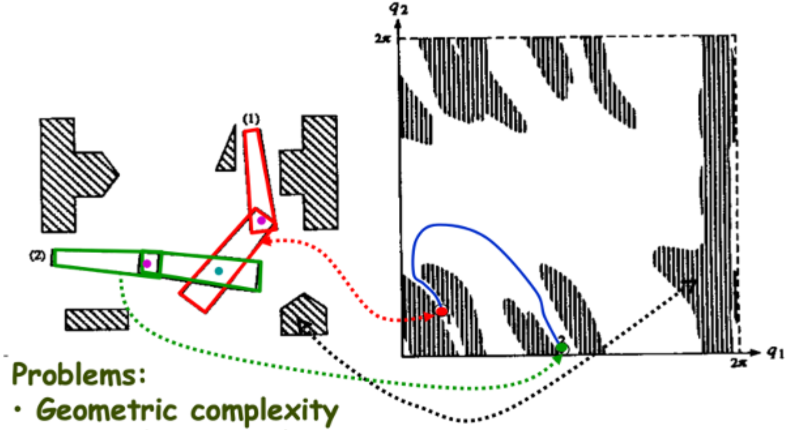
- It's hard to think about and formalize a whole robot moving around in the 2D or 3D workspace.
- Instead, think about a <u>point</u> moving around in the space of possible robot configurations (<u>cspace</u>).
- Let $\mathcal{C}_{\text{free}}$ be the set of robot configurations $q \in \mathcal{Q}$ such that if the robot is in that configuration it does not collide with the environment or itself.
- Our problem, then, is to find a trajectory for a point that goes between q_s and q_g and stays entirely in $\mathcal{C}_{\text{free}}$.
- Unfortunately, it can be hard to explicitly characterize the shape of $\mathcal{C}_{\text{free}}$, which depends both on the obstacles in the environment and the kinematics (shapes and joints) of the robot.

Workspace vs Configuration Space



[fig from Jyh-Ming Lien]

Two-joint robot arm C-space



Problems:
- Geometric complexity
- Space dimensionality

24

Searching configuration space

We focus on piecewise linear paths in configuration space. Now instead of finding a whole continuous function, we just have to find some set of points that we can connect up without collisions.

Three strategies:

- Exact: Construct an exact decomposition of $\mathcal{C}_{\text{free}}$ into traversible regions and find a path that makes linear moves between them.
  - Complete algorithms exist.
  - Drawbacks: Exponential in d, the number of *degrees of freedom* of the robot. Difficult to implement.
- Grid-based: Min-cost path search in a grid.
  - Action space: small fixed displacements of each joint within $\mathcal{C}_{\text{free}}$
  - Goal set: configurations that are close to q_g (cannot hit it exactly!)
  - Heuristic! Distance in configuration space. Can be tricky.
  - Drawbacks: Grid size is exponential in degrees of freedom. Needs fine discretization if gaps between cspace obstacles are small (increases running time).
- Sample-based: Most widely used.
  - Probabilistically complete.
  - Drawback: Narrow-passage problem.

Sample-based method: Probabilistic Road Map (PRM)

- Randomly sample configurations
- Discard samples that are in collision
- Connect near neighbors via straight-line segments
- Discard segments that are in collision
- Connect start and goal to resulting graph and search

```
BUILD-PRM(q_s, q_g, K, δ)
V = {q_s, q_g}; E = {}
for k = 1..K
    q = SAMPLE-CONF()
    if IS-COLLISION-FREE(q): V.ADD(q)
for (q_a, q_b) ∈ V × V:
    path = GENERATE-LINEAR-PATH(q_a, q_b)
    if IS-COLLISION-FREE(path): E.ADD(path)
return GRAPH-SEARCH(V, E, q_s, q_g)
```

PRM iterations

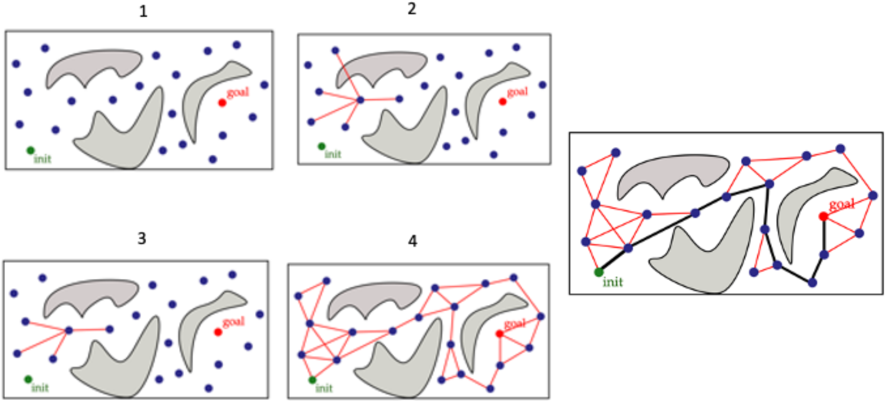


Image source: E. Plaku

Rapidly expanding random trees (RRT)

Sample-based algorithm that is easy to implement and reasonably effective

- Randomly sample configurations
- Try to connect via a linear collision-free path to closest (need a distance metric!) configuration in the tree
- Better if bi-directional!
- Not optimal—need to “shortcut” and smooth

BUILD-RRT(q_s, q_g, K, δ)

```
T = TREE(q_s)
for k = 1..K
    q_rand = RANDOM-CONF()                                // Sample q_g occasionally
    q_near = NEAREST-VERTEX(q_rand, T)
    success, path = EXTEND-PATH(q_near, q_rand, δ)
    for i = 1..len(path) − 1:
        T.add-edge(path[i], path[i + 1])
    if success: return T.path(q_s, q_g)
```

RRT iterations

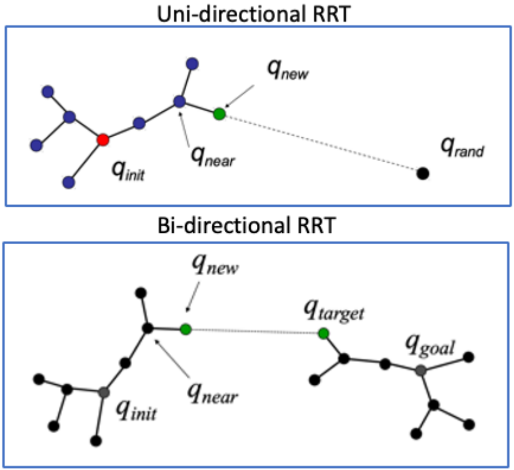
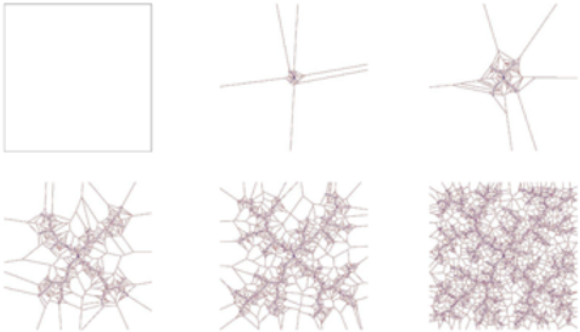


Image source: H. Choset, CMU

Voronoi Bias is key to RRT

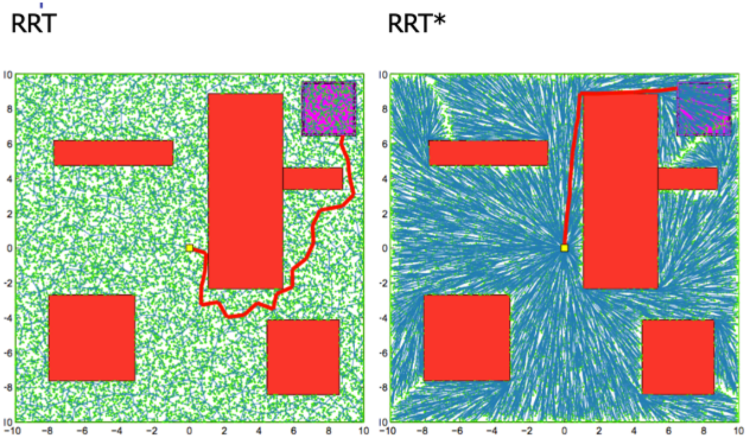
- Tree vertices near large unexplored regions are more likely to be extended.



http://msl.cs.uiuc.edu/rrt/gallery.html

RRT* - asymptotically optimal RRT

- Swap in new point as parent for nearby vertices if it leads to shorter path than the path through their curret parent



Source: Karaman and Frazzoli

Local optimization

Alternatively, we can start with a path (still defined as linear interpolation between waypoints that is not legal, and try to improve it!

- Fix K waypoints in cspace: $q_1, \ldots, q_K$
- Initialize (e.g. a straight line)
- Pick objective (cost) function

$$\sum_{k=1}^{K-1} \textit{max-penetration-depth}(q_k, q_{k+1}) + \lambda \textit{dist}(q_k, q_{k+1})$$

- Minimize using gradient-based techniques
- Can have a lot of trouble with local optima
- Less craziness in path, if it works

Next time

- Uncertainty!