Principles of Robot Autonomy I

Motion planning I: graph search methods
Motion planning

Compute sequence of actions that drives a robot from an initial condition to a terminal condition while avoiding obstacles, respecting motion constraints, and possibly optimizing a cost function

• Aim
  • Introduction to motion planning
  • Learn about search-based methods for motion planning

• Readings:
  • D. Bertsekas. Dynamic Programming and Optimal Control, Vol I. Section 2.3.
The see-think-act cycle

Knowledge

- Localization
  - Map Building
  - environmental model
  - local map

- Information extraction
  - raw data
  - Sensing

- Real world environment

Position

- global map

Decision making

- Motion planning
  - trajectory

Trajectory execution

- actuator commands
  - Actuation

Mission goals

See-think-act
Examples from:
https://ompl.kavrakilab.org/gallery.html
More examples of motion planning

- Steering autonomous vehicles
- Controlling humanoid robot
- Surgery planning
- Protein folding
- …
Some history

• Formally defined in the 1970s
• Development of exact, combinatorial solutions in the 1980s
• Development of sampling-based methods in the 1990s
• Deployment on real-time systems in the 2000s
• Current research: inclusion of differential and logical constraints, planning under uncertainty, parallel implementation, feedback plans and more
Simplest setup

• Assume 2D workspace: $\mathcal{W} \subseteq \mathbb{R}^2$
• $\mathcal{O} \subseteq \mathcal{W}$ is the obstacle region with polygonal boundary
• Robot is a rigid polygon
• Problem: given initial placement of robot, compute how to gradually move it into a desired goal placement so that it never touches the obstacle region
Popular approaches

- **Potential fields** [Rimon, Koditschek, '92]: create forces on the robot that pull it toward the goal and push it away from obstacles

- **Grid-based planning** [Stentz, '94]: discretizes problem into grid and runs a graph-search algorithm (Dijkstra, A*, …)

- **Combinatorial planning** [LaValle, '06]: constructs structures in the configuration (C-) space that completely capture all information needed for planning

- **Sampling-based planning** [Kavraki et al, '96; LaValle, Kuffner, '06, etc.]: uses collision detection algorithms to probe and incrementally search the C-space for a solution, rather than completely characterizing all of the $C_{free}$ structure
Grid-based approaches

• Discretize the continuous world into a grid
  • Each grid cell is either free or forbidden
  • Robot moves between adjacent free cells
  • **Goal**: find sequence of free cells from start to goal

• Mathematically, this corresponds to pathfinding in a discrete graph $G = (V, E)$
  • Each vertex $v \in V$ represents a free cell
  • Edges $(v, u) \in E$ connect adjacent grid cells
Graph search algorithms

- Having determined decomposition, how to find “best” path?
- **Label-Correcting Algorithms**: $C(q)$: cost-of-arrival from $q_I$ to $q$

```plaintext
FRONTIER/ALIVE/PRIORITY QUEUE

Node $q$

Nodes $q' \in Succ(q)$

$q' \neq q_G$?

Yes $\Rightarrow C(q') := C(q) + C(q, q')$

$C(q) + C(q, q') \leq \min(C(q'), \text{UPPER})$?
```

* [https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)
Label correcting algorithm

**Step 1.** Remove a node $q$ from frontier queue and for each child $q'$ of $q$, execute step 2

**Step 2.** If $C(q) + C(q, q') \leq \min(C(q'), \text{UPPER})$, set $C(q') := C(q) + C(q, q')$ and set $q$ to be the parent of $q'$. In addition, if $q' \neq q_G$, place $q'$ in the frontier queue if it is not already there, while if $q' = q_G$, set UPPER to the new value $C(q) + C(q, q_G)$

**Step 3.** If the frontier queue is empty, terminate, else go to step 1

**Initialization:** set the labels of all nodes to $\infty$, except for the label of the origin node, which is set to 0
GetNext()?

Depth-First-Search (DFS): Maintain \( Q \) as a **stack** – Last in/first out
- Lower memory requirement (only need to store part of graph)

Breadth-First-Search (BFS, Bellman-Ford): Maintain \( Q \) as a **list** – First in/first first out
- Update cost for all edges up to current depth before proceeding to greater depth
- Can deal with negative edge (transition) costs

Best-First (BF, Dijkstra): Greedily select next \( q \): \( q = \text{argmin}_{q \in Q} C(q) \)
- Node will enter the frontier queue at most *once*
- Requires costs to be non-negative
Correctness and improvements

**Theorem**

If a feasible path exists from $q_I$ to $q_G$, then algorithm terminates in finite time with $C(q_G)$ equal to the optimal cost of traversal, $C^*(q_G)$. 
A*: Improving Dijkstra

• Dijkstra orders by optimal “cost-to-arrival”
• Faster results if order by “cost-to-arrival”+ (approximate) “cost-to-go”
• That is, strengthen test
  \[ C(q) + C(q, q') \leq \text{UPPER} \]
  to
  \[ C(q) + C(q, q') + h(q') \leq \text{UPPER} \]
  where \( h(q) \) is a heuristic for optimal cost-to-go (specifically, a positive underestimate)
• In this way, fewer nodes will be placed in the frontier queue
• This modification still guarantees that the algorithm will terminate with a shortest path
Grid-based approaches: summary

• Pros:
  • Simple and easy to use
  • Fast (for some problems)

• Cons:
  • Resolution dependent
    • Not guaranteed to find solution if grid resolution is not small enough
  • Limited to simple robots
    • Grid size is exponential in the number of DOFs
Back to continuous motion planning

• A robot is a geometric entity operating in continuous space

• *Combinatorial techniques* for motion planning capture the structure of this continuous space
  • Particularly, the regions in which the robot is not in collision with obstacles

• Such approaches are typically complete
  • i.e., guaranteed to find a solution;
  • and sometimes even an optimal one
Simplest setup revisited

- Assume 2D workspace: $\mathcal{W} \subseteq \mathbb{R}^2$
- $\mathcal{O} \subseteq \mathcal{W}$ is the obstacle region with polygonal boundary
- Robot is a rigid polygon
- **Problem**: Given initial placement of robot, compute how to gradually move it into a desired goal placement so that it never touches the obstacle region
Simplest setup

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

- **C-space**: captures all degrees of freedom (all rigid body transformations)
- More in detail, let $\mathcal{R} \subset \mathbb{R}^2$ be a polygonal robot (e.g., a triangle)
- The robot can rotate by angle $\theta$ or translate $(x_t, y_t) \subset \mathbb{R}^2$
- Every combination $q = (x_t, y_t, \theta)$ yields a *unique* robot placement: configuration
- So C-space is a subset of $\mathbb{R}^3$
- Note: $\theta \pm 2\pi$ yields equivalent rotations $\Rightarrow$ C-space is: $\mathbb{R}^2 \times S^1$
- Concept of C-space extends naturally to higher dimensions (e.g., robot linkages)
Configuration free space

- The subset $\mathcal{F} \subseteq \mathcal{C}$ of all collision free configurations is the **free space**
Planning in C-space

- Let $R(q) \subseteq W$ denote set of points in the world occupied by robot when in configuration $q$
- Robot in collision $\iff R(q) \cap O \neq \emptyset$
- Accordingly, free space is defined as: $C_{free} = \{q \in C | R(q) \cap O = \emptyset\}$
- Path planning problem in C-space: compute a continuous path: $\tau: [0,1] \rightarrow C_{free}$, with $\tau(0) = q_I$ and $\tau(1) = q_G$
Combinatorial planning

**Key idea:** compute a roadmap, which is a graph in which each vertex is a configuration in $C_{\text{free}}$ and each edge is a path through $C_{\text{free}}$ that connects a pair of vertices
Free-space roadmaps

Given a complete representation of the free space, we compute a roadmap that captures its connectivity.

A roadmap should preserve:

1. **Accessibility**: it is always possible to connect some \( q \) to the roadmap (e.g., \( q_I \to s_1, q_G \to s_2 \))
2. **Connectivity**: if there exists a path from \( q_I \) to \( q_G \), there exists a path on the roadmap from \( s_1 \) to \( s_2 \)

**Main point**: a roadmap provides a discrete representation of the continuous motion planning problem *without losing* any of the original connectivity information needed to solve it
Cell decomposition

Typical approach: **cell decomposition**. General requirements:
- Decomposition should be easy to compute
- Each cell should be easy to traverse (ideally convex)
- Adjacencies between cells should be straightforward to determine
Computing a trapezoidal cell decomposition

For every vertex (corner) of the forbidden space:
• Extend a vertical ray until it hits the first edge from top and bottom
  • Compute intersection points with all edges, and take the closest ones
  • More efficient approaches exists
Other roadmaps

Maximum clearance (medial axis)

Minimum distance (visibility graph)

Note: No loss in optimality for a proper choice of discretization
Caveat: free-space computation

• The free space is **not known** in advance
• We need to compute this space given the ingredients
  • Robot representation, i.e., its shape (polygon, polyhedron, ...)
  • Representation of obstacles
• To achieve this we do the following:
  • Contract the robot into a point
  • In return, inflate (or stretch) obstacles by the shape of the robots
Higher dimensions

• Extensions to higher dimensions is challenging ⇒ algebraic decomposition methods
Additional resources on combinatorial planning

- Visualization of C-space for polygonal robot: https://www.youtube.com/watch?v=SBFwgR4K1Gk
- Implementation in C++: Computational Geometry Algorithms Library
Combinatorial planning: summary

• These approaches are complete and even optimal in some cases
  • Do not discretize or approximate the problem
• Have theoretical guarantees on the running time
  • I.e., computational complexity is known
• Usually limited to small number of DOFs
  • Computationally intractable for many problems
• Problem specific: each algorithm applies to a specific type of robot/problem
• Difficult to implement: require special software to reason about geometric data structures (CGAL)
Next time: sampling-based planning