Harvard CS 182: Artificial Intelligence September 14, 2022

Lecture 4: Motion Planning

Lecturer: Stephanie Gil *Author: Max Guo*

This lecture on *motion planning* will be divided into a more theoretical components and more practical components. The theoretical components will cover discretizing an environment, searching for a path from start to goal states (which ties into previous lectures), and optimality considerations. The practical components will discuss what happens in higher dimensions and give examples of real world problems (including many video examples that are in the lecture recordings!). As a reference, you may consider reading Chapters 5 and 6 of "Planning Algorithms" by Steven Lavalle, which is available freely online.

1 Setting up Motion Planning

We'll start the more theoretical component of lecture by defining the problem of motion planning.

1.1 Background

Fundamentally, motion planning asks the question of

"What series of motions will get a robot to a goal?"

We're curious about:

- Whether or not there exists a solution to this problem.
- How we can find out if a solution exists.
- If a solution doesn't exist, why it doesn't exist.

More formally, we model the problem in terms of configurations, asking the question of whether or not we can find a series of *valid* configurations that move the robot from source to destination. We especially consider two criteria for validity:

1. Environment-imposed restrictions. Is the path obstacle-free?
2. Dynamic-imposed restrictions. Is the agent able to transition through these configurations?

1.2 Configuration Space

The configuration space is defined by Lavalle as the set of possible transformations that could be applied to the robot. This can be encoded in several different ways. For example:

- In a $2d$ world, we could use location (x, y) and orientation (θ).
- In a $3d$ world, we could use location (x, y, z) and orientation (in 3d space) (ρ, γ, θ).
- For a robotic arm with N degrees of freedom, this could be some other manifold of dimension N.

To solve a motion planning problem, algorithms must conduct a search in the configuration space. In the real world, it's often useful to *discretize* continuous spaces into discrete configurations so that the resulting search problems are more feasible.

1.3 Problem

In the motion planning problem, we're given the following:

- A robot with configuration space C
- A set of obstacles C_{obs}
- An initial configuration q_{init}
- A goal configuration q_{goal}.

We wish to find a path $x : [0, 1] \to C$ such that:

- $x(0) = q_{init}$ (start from the initial configuration)
- $x(1) = q_{goal}$ (reach the goal configuration)
- $x(s) \neq C_{obs}$ for all $s \in [0, 1]$ (avoid collision with obstacles).

We can turn our motion planning problem into a search problem by defining what configurations (defining the states) are achievable from other configurations (defining the actions), and then applying our search algorithms that we've already learned to solve the search problem. Similar to search problems, we care about performance guarantees, such as *completeness*. An algorithm is *complete* if it terminates in finite time and returns a solution when one exists, and returns failure otherwise.

2 Cell Decomposition

One natural way to discretize a free space is to decompose it into cells. Letting cells be nodes and letting adjacencies between cells be edges, we can then obtain a graph from this representation.

Any cell decomposition must satisfy the following properties:

1. (Accessibility) Computing a path from a point to another inside the same cell must be easy.
2. (Representation) Adjacency information for cells can be easily extracted to build a graph (also known as the roadmap).
3. (Querying) It should be efficient to determine which cell contains any one point.

For example, a condition that would imply accessibility is that each cell is *convex*, which means that any straight line segment connecting two points in the cell is fully contained in the cell.

2.1 Vertical Cell Decomposition

Let C_{obs} represent the space of obstacles, and C_{free} represent free space. Assume we're in a two dimensional environment. Given polygonal obstacles, we can draw vertical lines from each vertex until we hit a boundary or obstacle, obtaining a cell decomposition that results in convex cells and preserving the connectivity of C_{free}. The two types of resulting cells are known as 1-cells and 2-cells.

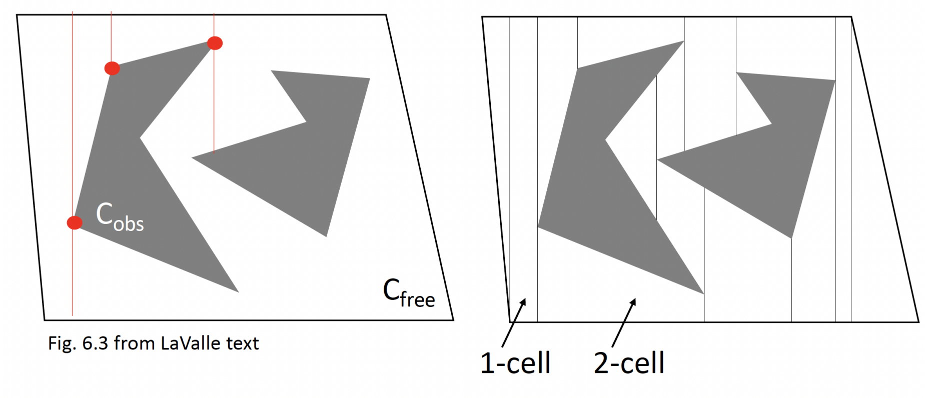


Figure 1: Vertical Cell Decomposition

Once we have the cell decomposition, we can choose a point in every cell and a point on the boundary of every neighboring pair of cells as our vertices of the graph, resulting in our roadmap.

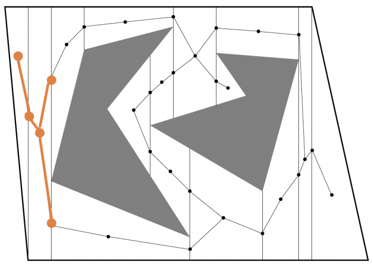
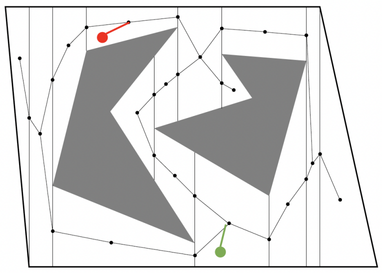


Figure 2: Defining a roadmap from the cells

Now that we have the roadmap, we can apply our search algorithms so find a path from some start state to some goal state.

Example 1 *Consider the following graph, and try solving for a path between the two query points q_1 and q_G. Use 1) DFS expanding the rightmost descendant first and 2) A^* search using Euclidean distance as edge cost.*



DFS with the rightmost descendant expansion rule results in the path that visits all the nodes on the right side of the right polygon. A^* results in the following path:

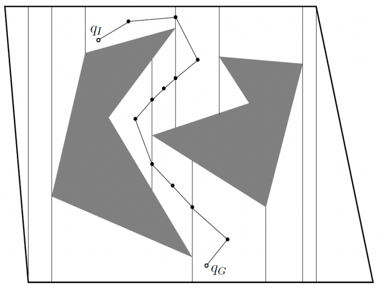


Figure 3: A^* path in example problem

2.2 Visibility Graphs

Visibility graphs define another type of discretization which results in an optimal shortest path roadmap. To define the visibility graph, we define the vertices and edges as follows:

- Vertices: Any polygon vertex such that the interior angle in C_{free} is larger than π. These are known as *reflex vertices.*
- Edges:
  - Any edge of an polygonal obstacle whose endpoints are reflex vertices.
  - Any edge connecting two reflex vertices that isn't blocked by obstacles (and are thus mutually visible from each other). These are known as *bitangent lines.*

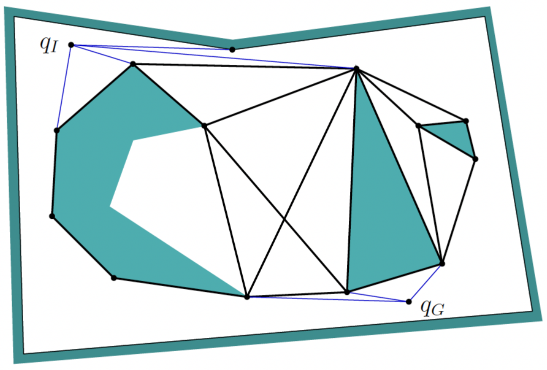


Fig. 6.12 from LaValle text

Figure 4: Example of Visibility Graph

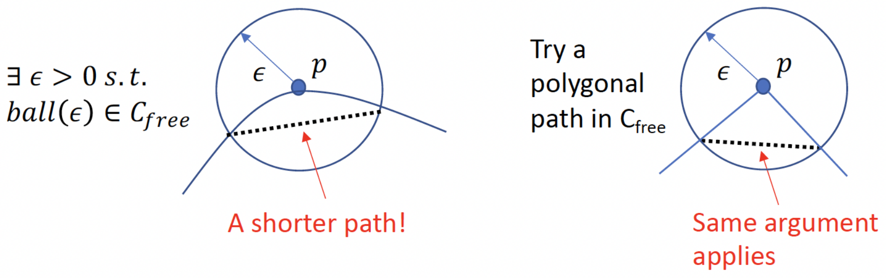
Once we have this visibility graph, the resulting search problem can give us our optimal path! The computation of the shortest-path roadmap has complexity $O(n \log n + m)$, where n is the number of vertices in C_{obs} and m is the total number of edges in the roadmap. The derivation of this is in Lavelle.

Note. Something important to note is that the optimal path problem may not be solvable (i.e. there may not exist an optimal path) if C_{free} is not closed, so we can consider the closure of C_{free}, denoted $cl(C_{free})$, to ensure our problem has a defined solution. This means that our agent is allowed to travel on the edge of the obstacles.

We now prove the following lemma which illustrates (without going into all the details) optimality of searching along the visibility graph:

Lemma 1 *Consider a set S of disjoint polygonal obstacles, and start and goal positions q_1 and q_G, respectively. Any shortest path between q_1 and q_G is a polygonal path where the inner vertices are vertices of S.*

Proof: Suppose this is not the case, and that the shortest path goes through a point p in C_{free}. If it is not polygonal, we obtain the situation in the left image of the below figure, where we can draw a ball of radius $\epsilon > 0$ around p such that it is fully contained in C_{free}. By joining the points where the path intersects the ball, we obtain a shorter path. Similarly, if we have a vertex located in C_{free}, we can apply the same argument, as shown in the right image, to obtain a shorter path.


$\exists \epsilon > 0 \; s.t.$
$ball(\epsilon) \in C_{free}$
ϵ
p
A shorter path!
Try a polygonal path in C_{free}
ϵ
p
Same argument applies

Now the last case is when the vertex p of our path lies in the interior of the edge of an obstacle in C_{obs} (i.e. the path does not follow the entirety of the obstacle's edge). Suppose for contradiction that a shortest path contains such a vertex. Then there exists a ball of radius ϵ centered on the vertex such that at least one of the points of intersection between the path and the circle is not on the boundary of the obstacle. Connecting these points (dashed line below) results in a shorter path that does not intersect the obstacle.

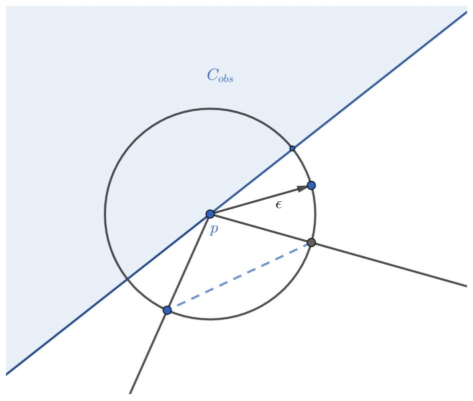


Figure 5: Last case: vertex lies in interior of edge.

This shows that all inner vertices must be vertices of S. (Check your understanding here. Why does the argument in the last case not hold when p lies on the vertex of a polygon?)
■

Now, it turns out that we cannot scale this method effectively to three dimensions or higher! Visibility graphs only have optimality in two dimensions. In higher dimensions, optimal paths might go through edges as opposed to just vertices, and the problem is NP-hard.

3 Sampling Methods

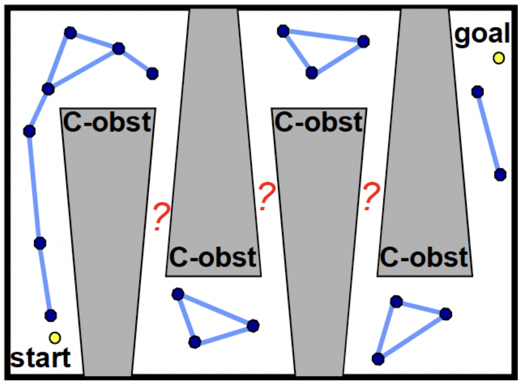
Now we turn to a different method of motion planning. The key idea behind sampling methods is to sample the configuration space and then connect samples with trajectors to infer the connectivity of the free space. This general approach has been used in autonomous vehicles, service robots, robot manipulators (e.g. robotic arms), and other cool applications. We will discuss Probabilistic RoadMap (PRM) and Rapidly-exploring Random Trees (RRT), which are two widely used variants of the sampling approach. With these algorithms, we also care about the best sampling approach and the optimality and completeness guarantees of the algorithms. These algorithms, unlike before, can be applied to higher dimensions.

3.1 Probabilistic RoadMap (PRM)

The PRM algorithm is as follows:

1. Sample N points uniformly at random from C
2. Connect each pair with a straight trajectory
3. Delete all vertices and edges that lie in the obstacle set C_{obs}.
4. Return the remaining roadmap $G = (V, E)$.

This method is not complete, but it is *probabilistically complete*. In other words, the probability of returning a solution approaches 1 as the number of samples increases. However, the performance can be environment-dependent. For example, consider how the following space with many large obstacles might require many samples to find a valid path.



[Image credit: Sertac Karaman *RSS*]

Figure 6: Potentially difficult space for PRM

However, if we don't have a good sampling method, we may not have a good bias of our samples towards the goal, resulting in a very long time before reaching the goal.

3.2 Rapidly-Exploring Random Trees (RRT)

With RRT, we randomly sample vertices and grow a "random tree" in the free space. In addition, you can bias the search if you have additional information. This algorithm is probabilistically complete, but it is not optimal. This motivated the development of a variant, RRT*, which selects the best parent node at each step that minimizes cost from the root, and then re-wires the tree to keep track of lower cost paths. This allows for optimal paths and "smoother" graphs than vanilla RRT.