AA274A: Principles of Robot Autonomy I
Course Notes

Oct 8, 2019

6 Motion planning I: Graph Search Methods

In past lectures, we developed controllers for tracking a desired trajectory. Starting this week, we look at how the desired trajectory can be obtained, which is the problem of motion planning.[1] We will first start with a definition of motion planning and give a broad overview common approaches. Afterwards, we will focus on a survey two major approaches in motion planning: discrete (grid-based) planning and combinatorial planning.

6.1 Introduction

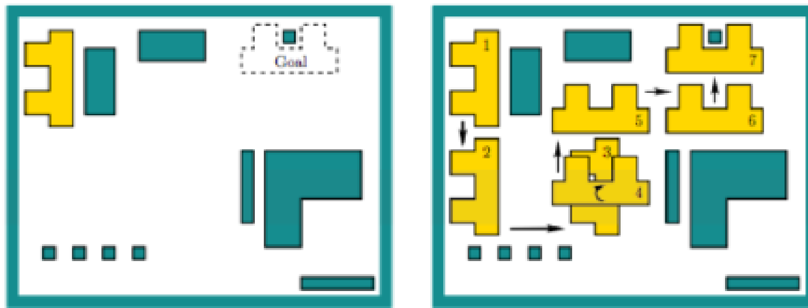

Figure 1: Simplest setup: 2D Workspace

A robot should reach its goal position as efficiently as possible given a map of its environment. This task gives rise to the navigation or motion planning problem. For example, consider a *workspace* defined in 2D, $\mathcal{W} \subseteq \mathbb{R}^2$. Within this workspace, we define the robot as a rigid polygon, and we introduce the set of obstacles $\mathcal{O} \subset \mathcal{W}$ as a subset of the workspace shown in Fig 1. As can be seen in the figure, reaching the goal without collision requires a series of translation and rotation maneuevers. Now we're ready to define a motion planning problem:

[1] Much of this lecture note is a direct excerpt from [LaV06].

Definition 6.1 (Motion planning problem)**.** *Compute a sequence of actions to go from an initial condition to a terminal condition while respecting constraints and possibly optimizing a cost function.*

Motion planning can be solved in a discrete space or a continuous space. In continuous space, motion planning can be done exactly or approximately (e.g. using sampling-based methods). In this lecture, we start surveying popular algorithms to solve motion planning as a discrete search problem. Afterwards, we will look into combinatorial methods to solve motion planning in continuous space. Next lecture will cover sampling-based methods that solves motion planning problem stochastically in continuous space.

6.2 Discrete planning

The planning problems considered here are the simplest to describe because the state space will be finite in most cases. When they are not finite, they will at least be countably infinite (i.e., a unique integer may be assigned to every state). Therefore, no geometric models or differential equations will be needed to characterize the discrete planning problems. Furthermore, no forms of uncertainty will be considered, which avoids complications such as probability theory. All models are completely known and predictable.

6.2.1 Discrete planning: Problem Formulation

The discrete feasible planning model finds a path in state-space models, or a graph. The basic idea is that each distinct situation for the world is called a *state*, denoted by x, and the set of all possible states is called a state space, X. For discrete planning, it will be important that this set is countable, and in most cases it will be finite. Each action u, when applied from the current state x, produces a new state, x', as specified by a state transition function f. It is convenient to use f to express a state transition equation,

$$x' = f(x, u). \tag{1}$$

Let $U(x)$ denote the action space for each state x, which represents the set of all actions that could be applied from x. For distinct x, $x' \in X$, $U(x)$ and $U(x')$ are not necessarily disjoint; the same action may be applicable in multiple states. Therefore, it is convenient to define the set U of all possible actions over all states:

$$U = \cup_{x \in X} U(x) \tag{2}$$

As part of the planning problem, a set $X_G \in X$ of goal states is defined. The task of a planning algorithm is to find a finite sequence of actions that when applied, transforms the initial state x_{init} to some state in X_G. The model is summarized as:

Formulation 2.1 (Discrete Feasible Planning)

1. A nonempty state space X, which is a finite or countably infinite set of states.

2. For each state $x \in X$, a finite action space $U(x)$.

3. A state transition function f that produces a state $f(x, u) \in X$ for every $x \in X$ and $u \in U(x)$. The state transition equation is derived from f as $x' = f(x, u)$.

4. A goal set $X_G \subseteq X$.

It is often convenient to express Formulation 2.1 as a directed state transition graph. The set of vertices is the state space X. A directed edge from $x \in X$ to $x' \in X$ exists in the graph if and only if there exists an action $u \in U(x)$ such that $x' = f(x, u)$. The initial state and goal set are designated as special vertices in the graph, which completes the representation of Formulation 2.1 in graph form.

6.2.2 From grids to graphs: Searching for feasible plans

We solve discrete feasible planning problems using graph search algorithms. One distinction is that the cost of travelling at each node is revealed incrementally through the application of actions, instead of being fully specified in advance. The presentation in this section can therefore be considered as approaching graph search algorithms from a planning perspective. An important requirement for these or any search algorithms is to be *systematic*. If the graph is finite, this means that the algorithm will visit every reachable state, which enables it to correctly declare in finite time whether or not a solution exists. To be systematic, the algorithm should keep track of states already visited; otherwise, the search may run forever by cycling through the same states. Ensuring that no redundant exploration occurs is sufficient to make the search systematic.

We first present a generic search algorithm to introduce the concept of *unvisited, dead* and *alive* states. Afterwards, we willl present speicific search algorithms that implements the generic search algorithm efficiently.

Generic Forward Search

- **Unvisited**: States that have not been visited yet. Initially, this is every state except x_I.

- **Dead**: States that have been visited, and for which every possible next state has also been visited. A next state of x is a state x' for which there exists a $u \in U(x)$ such that $x' = f(x, u)$. In a sense, these states are dead because there is nothing more that they can contribute to the search; there are no new leads that could help in finding a feasible plan.

- **Alive**: States that have been encountered, but possibly have unvisited next states. These are considered alive. Initially, the only alive state is x_I.

The set of alive states is stored in a priority queue, Q, for which a priority function must be specified. The only significant difference between various search algorithms is the

```
FORWARD_SEARCH
 1  Q.Insert(x_I) and mark x_I as visited
 2  while Q not empty do
 3      x ← Q.GetFirst()
 4      if x ∈ X_G
 5          return SUCCESS
 6      forall u ∈ U(x)
 7          x' ← f(x, u)
 8          if x' not visited
 9              Mark x' as visited
10              Q.Insert(x')
11          else
12              Resolve duplicate x'
13  return FAILURE
```

Figure 2: A general template for forward search.

particular function used to sort Q. Many variations will be described later, but for the time being, it might be helpful to pick one. Therefore, assume for now that Q is a common FIFO (First-In First-Out) queue; whichever state has been waiting the longest will be chosen when `Q.GetFirst()` is called. The rest of the general search algorithm is quite simple. Initially, Q contains the initial state x_I . A while loop is then executed, which terminates only when Q is empty. This will only occur when the entire graph has been explored without finding any goal states, which results in a FAILURE (unless the reachable portion of X is infinite, in which case the algorithm should never terminate). In each while iteration, the highest ranked element, x, of Q is removed. If x lies in X_G , then it reports SUCCESS and terminates; otherwise, the algorithm tries applying every possible action, $u \in U(x)$. For each next state, $x' = f(x, u)$, it must determine whether x' is being encountered for the first time. If it is unvisited, then it is inserted into Q; otherwise, there is no need to consider it because it must be either dead or already in Q.

Particular Forward Search Methods Now we present several search algorithms, each of which constructs a search tree. Each search algorithm is a special case of the algorithm in Figure (2), obtained by defining a different sorting function for Q. Most of these are just classical graph search algorithms.

- Depth-First Search (DFS) Depth-first search expands each node up to the deepest level of the graph, until the node has no more successors. As those nodes are expanded, their branch is removed from the graph and the search backtracks by expanding the next neighboring node of the start node until its deepest level and so on. DFS stores only a

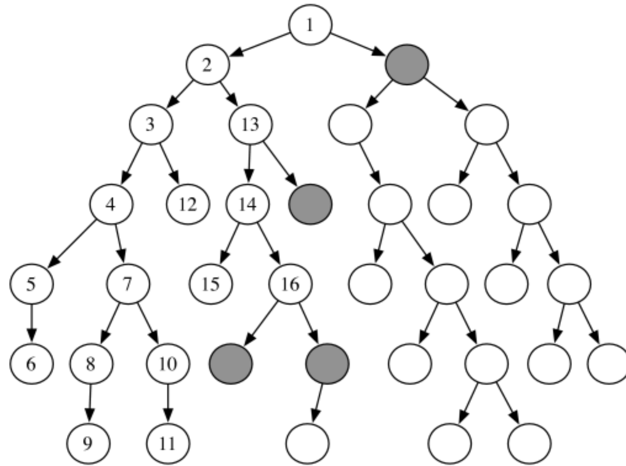

Figure 3: Depth First Search

single path from the start nodes for each node on the path, leading to a lower memory requirement. However, previously visited nodes might be visited and the algorithm might enter redundant paths.

- Breadth-First Search (BFS) Breadth-first search begins with the start node and explores all of its neighboring nodes. Then for each of these nodes, it explores all their unexplored neighbors and so on. This search always returns the path with the fewest number of edges between the start and goal node.

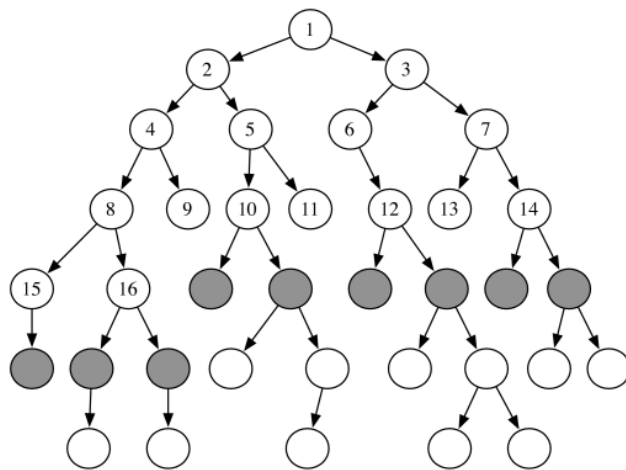

Figure 4: Breadth First Search

- Dijkstra's Algorithm (Shortest Path First) Dijkstra's algorithm is similar to Breadth-first search, except that edge costs may assume any positive value and the search still

guarantees solution optimality. Dijkstra's algorithm uses the concept of the heap, a specialized tree-based data structure and selects the next q as:

$$q = \arg\min_{q \in Q} C(q)$$

```
FORWARD_LABEL_CORRECTING(x_G)
1   Set C(x) = ∞ for all x ≠ x_I, and set C(x_I) = 0
2   Q.Insert(x_I)
3   while Q not empty do
4       x ← Q.GetFirst()
5       forall u ∈ U(x)
6           x' ← f(x, u)
7           if C(x) + l(x, u) < min{C(x'), C(x_G)} then
8               C(x') ← C(x) + l(x, u)
9               if x' ≠ x_G then
10                  Q.Insert(x')
```

Figure 5: A generalization of Dijkstra's algorithm, which upon termination produces an optimal plan (if one exists) for any prioritization of Q, as long as X is finite. Compare this to Figure 2

More on Dijkstra: Label Correcting Algorithm Dijkstra's algorithm belongs to a broader family of label-correcting algorithms, which all produce optimal plans by making small modifications to the general forward-search algorithm in Figure 2. Figure 5 shows the resulting algorithm.

The main difference is to allow states to become alive again if a better cost-to-come is found. This enables other cost-to-come values to be improved accordingly. This is not important for Dijkstra's algorithm and A^* search because they only need to visit each state once. Thus, the algorithms in Figures 2 and 5 are essentially the same in this case. However, the label-correcting algorithm produces optimal solutions for any sorting of Q, including FIFO (breadth first) and LIFO (depth first), as long as X is finite. If X is not finite, then the issue of systematic search dominates because one must guarantee that states are revisited sufficiently many times to guarantee that optimal solutions will eventually be found.

Another important difference between label-correcting algorithms and the standard forward-search model is that the label-correcting approach uses the cost at the goal state to prune away many candidate paths; this is shown in line 7. Thus, it is only formulated to work for a single goal state; it can be adapted to work for multiple goal states, but performance degrades. The motivation for including $C(x_G)$ in line 7 is that there is no need to worry about improving costs at some state, x', if its new cost-to-come would be higher than $C(x_G)$;

there is no way it could be along a path that improves the cost to go to x_G. Similarly, x_G is not inserted in line 10 because there is no need to consider plans that have x_G as an intermediate state.

6.2.3 Correctness and Improvements

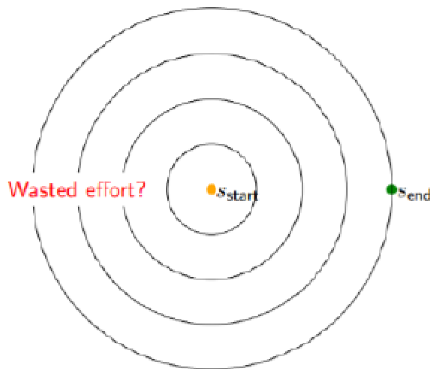

Figure 6: Problem with uniform graph search exploration strategies

Label correcting algorithms are guaranteed to find the shortest feasible path from an initial state to a goal state, given one exists. This attribute gives rise to the Correctness Theorem.

Theorem 6.2. *If a feasible path exists from initial state q_I to goal state q_G, then the algorithm terminates in finite time with path cost $C(q_G)$ equal to the optimal cost of traversal, $C^*(q_G)$.*

However, computing this path can be computationally intensive, especially in high-dimensional problems. This challenge arises from an explosion in the number of states expanded during the search procedure.

Consider Fig 6, Depth-First Search and Breadth-First Search algorithms do not greedily select states from a frontier queue for directed expansion. Hence, these algorithms search the configuration space without direction/relatively uniformly without considering the next state's cost or location relative to the goal. Dijkstra's algorithm improves on Depth-First Search and Breadth-First Search by greedily selecting states from the frontier queue based on cost and by not revisiting previously visited states. However, Dijkstra's algorithm does not take the goal state location into account, potentially leading to wasted effort. Concentrating the search in regions closer to the goal state may reduce the number of states expanded during the search, speeding up the motion planning process.

6.2.4 A*: Improving Dijkstra

Dijkstra's algorithm orders nodes added to the frontier queue (priority queue) by "cost-to-arrival" $C(q)$. The A* algorithm (given below) improves on Dijkstra by adding a heuristic

function $h(q)$ that models cost to go to the goal state from state q, i.e. A* orders by "cost-to-arrival" + (approximate) "cost-to-go." This heuristic function, for example distance to goal, guides exploration towards the region of the configuration space close to the goal, leading to faster computations as less states are expanded. Thus, the test condition changes from

$$C(q) + C(q, q') \leq \text{UPPER} \qquad \text{to} \qquad C(q) + C(q, q') + h(q') \leq \text{UPPER}.$$

The heuristic must be a *positive underestimate* of the true cost to the goal. This modification of the test condition reduces the number of nodes placed in the frontier queue and still guarantees that an optimal path will be returned. In practice, it leads to much faster runtimes.

Algorithm 1 A* Algorithm

Data: $q_{\text{init}}, q_{\text{goal}}$
Result: path
$C(q) = \infty, \;\; f(q) = \infty \;\; \forall q$
  $C(q_{\text{init}}) = 0, \; f(q_{\text{init}}) = h(q_{\text{goal}})$
  `// Init frontier queue and explored vertices`
$\text{OPEN} = \{q_{\text{init}}\}$, $\text{CLOSED} = \{\}$
while *OPEN is not empty* **do**
  $q = \arg\min_{q' \in \text{OPEN}} f(q')$ **if** $q == q_{goal}$ **then**
    return *path*
  end
  OPEN.remove(q); CLOSE.add(q)
  for q' *in* $\{q'$ — (q, q') *in G*, q' *not in CLOSED*$\}$ **do** `// Exploration`
    OPEN.add(q') **if** $C(q') < C(q) + C(q, q')$ **then**
      continue;
    end
    q'.parent = q; $C(q') = C(q) + C(q, q')$
     $f(q') = C(q') + h(q')$
  end
end
return *failure*

6.2.5 Benefits and Drawbacks of Graph-based Approaches

The graph-based approaches are simple, easy to use, and fast for some problems. However, this approach presents two drawbacks.

- These algorithms are resolution dependent, and this resolution is decided a priori. We might find that this resolution is not fine enough to uncover the best path or any path at all for the *original* motion planning problem (for example, if a narrow passage is missed by the discretization).

- These approaches don't scale well with robot degrees of freedom. Usually, we need to construct a grid with number of dimensions equal to the number of degrees of freedom of our robot. Thus, the number of gird cells increase exponentially, which also increases the number of neighbors of each vertex exponentially in the number of degrees of freedom. Thus, the graph-based approaches become less effective for high dimensional problems (more than 3 DOFs).

6.3 Combinatorial motion planning

Combinatorial approaches to motion planning find paths through the continuous configuration space without resorting to approximations. Due to this property, they are alternatively referred to as exact algorithms. All of the algorithms presented today are complete, which means that for any problem instance (over the space of problems for which the algorithm is des2igned), the algorithm will either find a solution or will correctly report that no solution exists. By contrast, in the case of sampling-based planning algorithms, weaker notions of completeness are tolerated: resolution completeness and probabilistic completeness.

6.3.1 Configuration Space

Rather than diving into the most general forms of combinatorial motion planning, it is helpful to first see several methods explained for a case that is easy to visualize. Several elegant, straightforward algorithms exist for the case in which $\mathcal{C} = \mathbb{R}^2$ and $\mathcal{C}_{obs}$ is polygonal. Most of these cannot be directly extended to higher dimensions; however, some of the general principles remain the same. Therefore, it is very instructive to see how combinatorial motion planning approaches work in two dimensions, as shown in Figure 7.

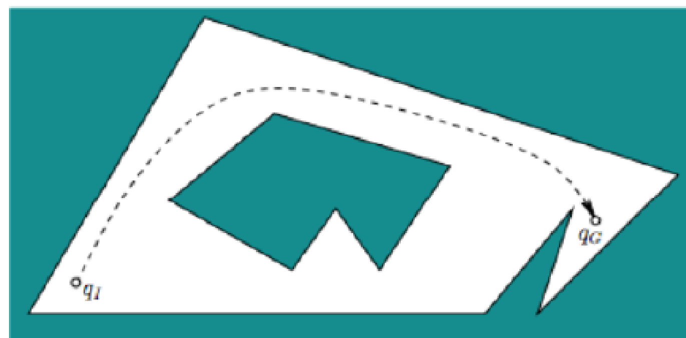

Figure 7: Combinatorial Planning

First we start by defining the environment where all motion planning problems need to be solved.

Definition 6.3 (Configuration ($\mathcal{C}$-)space)**.** *If the robot has n degrees of freedom, the set of transformations is usually a manifold of dimension n. This manifold is called the configuration space of the robot, and its name is often shortened to $\mathcal{C}$-space.*

By developing algorithms directly for this purpose, they apply to a wide variety of different kinds of robots and transformations. In Figure. 7, our $\mathcal{C}$-space contains the translational degrees of freedom and the rotational degrees of freedom of the robot, totaling $d = 3$ degrees of freedom. In this way, the 2D path planning problem in physical space is cast as a 3D path planning problem in the configuration space, with two dimensions describing the translational modes of the trajectory and one degree of freedom capturing the rotational mode of the trajectory in the planar workspace.

6.3.2 $\mathcal{C}$-space example

Formally, the degrees of freedom for our rigid polygon robot, $\mathcal{R} \subset \mathbb{R}^2$, is the set of generalized coordinates, $q = (x_t, y_t, \theta)$, where $(x_t, y_t) \subset \mathbb{R}^2$ are the translational coordinates, and θ is the rotational coordinate. Accordingly, the configuration space consists of every combination of q which yields a unique robot placement (in this case a subset of $\mathbb{R}^3$). An important distinction must be made for the generalized coordinate describing the rotational degree of freedom; namely, θ and $\theta \pm 2\pi k, k \in \mathbb{Z}$ describe equivalent attitudes. The configuration space in this case is actually in $\mathbb{R}^2 \times \mathcal{S}^1$, where $\mathcal{S}^1$ is the manifold which contains the angular displacement variable and includes the aforementioned "equivalence" between angles that differ by integer multiples of 2π. If we relax the planar restriction in our working example, the angular orientation coordinates of the robot would instead be in the manifold $\mathcal{S}^3$.

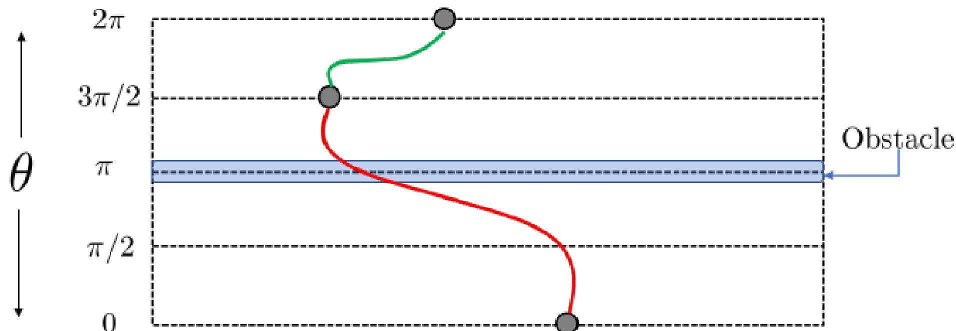

Figure 8: Example trajectory planning where the rotational degree of freedom is not described on $\mathcal{S}^1$ (red) and where it is described on $\mathcal{S}^1$ (green)

The importance of this distinction is highlighted with a simple example: consider a robot that has a current heading of $\theta = 3\pi/2$ rads, and we want to reconfigure the robot to heading $\theta_g = 0$ subject to the constraint of avoiding a $\mathcal{C}$-space obstacle at $\theta = \pi$. The geometry of this problem is highlighted in Figure 8. Now, if the equivalence between the angles 0 and 2π

is not established in the definition of the configuration space, the robot would not be able to traverse a collision-free path to the desired heading in the configuration space (see red trajectory). Instead, since the configuration space is defined with respect to $\mathcal{S}^1$, the robot is able to achieve the desired heading (see green trajectory).

Note that for our simple planar workspace example, the dimensions of the physical space and the configuration space are quite similar. Instead, it is not uncommon for complicated dynamical systems (e.g., a robotic arm mounted on a spacecraft) to have configuration spaces of much higher dimension than the physical space counterpart. In these cases, mapping from physical space to configuration space is extremely vital to the motion planning problem as it allows all constraints of the complex kinematic system to be accounted for with higher-dimensional information (i.e., more coordinates).

6.3.3 Planning in $\mathcal{C}$-Space

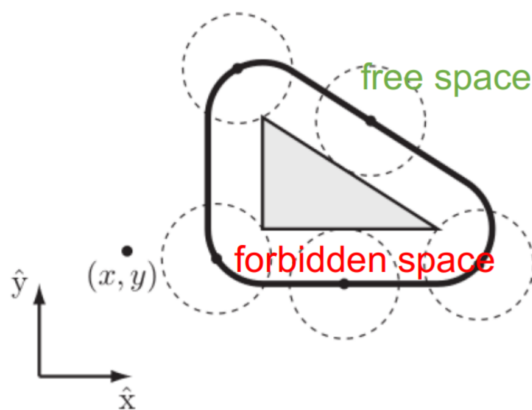

Figure 9: Forbidden space of simple robot and obstacle.

Now that we have introduced the necessary nomenclature and addressed the important distinction between motion planning in the real (physical) space and the configuration space, we can return to our prior problem statement and re-write it in rigorous mathematical detail. Let the set of points in the workspace that are occupied by the robot in configuration q be denoted by $\mathcal{R}(q) \subset \mathcal{W}$. Furthermore, we can make the intuitive definition of collision-free motion as that set of all robot configurations which do not intersect the obstacle regions: $\mathcal{R}(q) \cap \mathcal{O} = \emptyset$. Then the formal problem statement can be re-written as:

Definition 6.4 (Motion Planning Problem in Mathematical Notation)**.** *Compute a continuous path:* $\tau : [0, 1] \Rightarrow \{q \in \mathcal{C} | \mathcal{R}(q) \cap \mathcal{O} = \emptyset\}$ *with* $\tau(0) = q_1$ *and* $\tau(1) = q_G$*.*

The set $\{q \in \mathcal{C} | \mathcal{R}(q) \cap \mathcal{O} = \emptyset\}$ as $\mathcal{C}_{free}$ represents all possible combinations of the robot's degrees of freedom which are collision-free. For a simple 2 degree of freedom circle robot and triangle obstacle, as seen in Figure 9, the forbidden space can be constructed by drawing a line through the center every robot configuration that just touches the obstacle. It can be

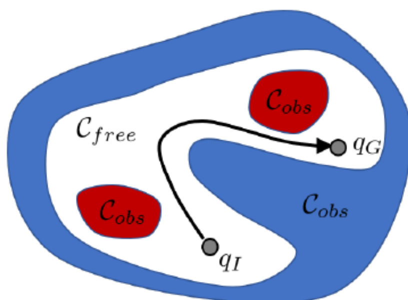

Figure 10: Robotic motion planning as a point path planning problem in $\mathcal{C}$-space.

seen that a forbidden robot state will be any state in which the center of the robot is inside the forbidden space. Combinatorial motion planning approaches construct a roadmap, a graph G which should preserve:

1. **Accessibility**: It is always possible to connect some q to the roadmap. For instance, q_I can be connected to s_1 and q_G can be connected to s_2 on the roadmap.

2. **Connectivity**: If there is a path between two configurations q_I and q_G, then there exists a path on the roadmap between s_1 and s_2.

The roadmap essentially provides a discrete representation of the continuous problem without losing any of the original connectivity information needed to solve it.

The typical approach involves cell decomposition. There are requirements that the cells be easy to traverse, the decomposition be easy to compute, and the cell adjacencies be straightforward to determine.

6.3.4 Cell Decomposition

In order to pick the roadmap the environment needs to be decomposed in a way where connectivity between subspaces of the environment is captured. The most widely used method for representing an environment is cell decomposition which entails making a discrete representation of the configuration space. Cell decomposition refers to a set algorithms that partitions C_{free} into a finite set of regions called cells. Three properties should be satisfied by cell decomposition, to reduce motion planning problem to a graph search problem:

- Each cell should be easy to traverse and ideally convex.
- Decomposition should be easy to compute.
- Adjacencies between cells should be straightforward to determine, in order to build the roadmap.

2D Decomposition We first take a look at 2D vertical cell decomposition, which partitions C_{free} into a finite collection of 2-cells and 1-cells, where k-cell refers to a k-dimensional cell. Each 2-cell is either a trapezoid or a triangle. Each 1-cell is a vertical segment that serves as the border of two 2-cells. Figure 11 shows an environment represented as a configuration space, with free configuration space, C_{free}, denoted as the white regions and obstacle space, C_{obs}, denoted as the dark regions. Though the planning problem is 2-dimensional, the robot configuration space is 3-dimensional, as the heading of the robot adds another degree of freedom.

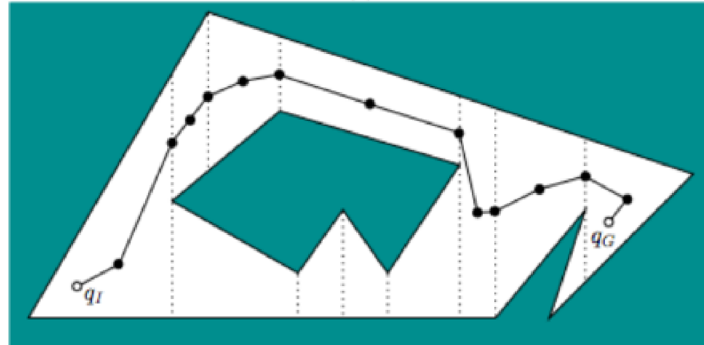

Figure 11: Example of 2D Cell Decomposition and Roadmap Extraction given q_I and q_G.

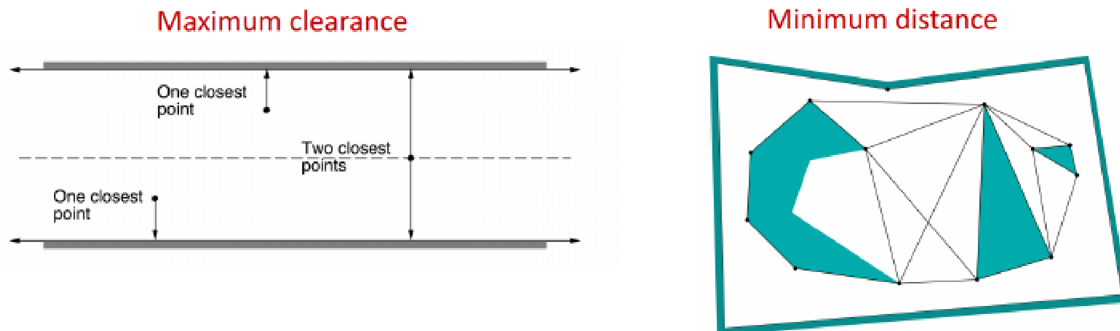

Figure 12: Other Cell Decomposition Methods Based on Requirements.

Assume the obstacle space is polygonal, then vertical cell decomposition works as follows. Let P denote the set of vertices used to define C_{obs}. For each vertex $p_i \in P$, extend rays upwards and downwards through C_{free}, until C_{obs} is reached. Every intersection forms a cell, and thus the environment becomes a set of cells. All cells are triangles or trapezoids, so they are convex. Hence, if we put a roadmap node randomly inside a cell, all other configurations

within this cell can be connected to the roadmap point without intersecting C_{obs}. It follows that connectivity of the environment is preserved through this cell decomposition.

Now that we have different cells that compose the environment, the roadmap can be built by placing a sample point at the centroid of each cell and at midpoint of each boundary. Once the roadmap is constructed, given an initial and goal configuration q_I and q_G, a path can always be extracted from the roadmap. The convex properties of the cells and placement of the sample points guarantee that if a path exists it will not intersect C_{obs} (i.e. no collision will occur).

Other ways of constructing the roadmap are based on different requirements. Figure 12 shows two requirements: maximum clearance and minimum distance. Specifically, if maximum clearance from C_{obs} is required, we would like to place roadmap points along the middle of the tunnel that represents the boundary of C_{obs}. A Voronoi diagram is used to compute the roadmap by connecting q_I and q_G along edges of the constructed diagram in polygons. In the other case where the shortest path is the goal, a visibility graph can be used. A visibility graph represents the set of unblocked lines between vertices of obstacles, q_I and q_G. Usually shortest paths tend to graze the corners of C_{obs} as much as possible.

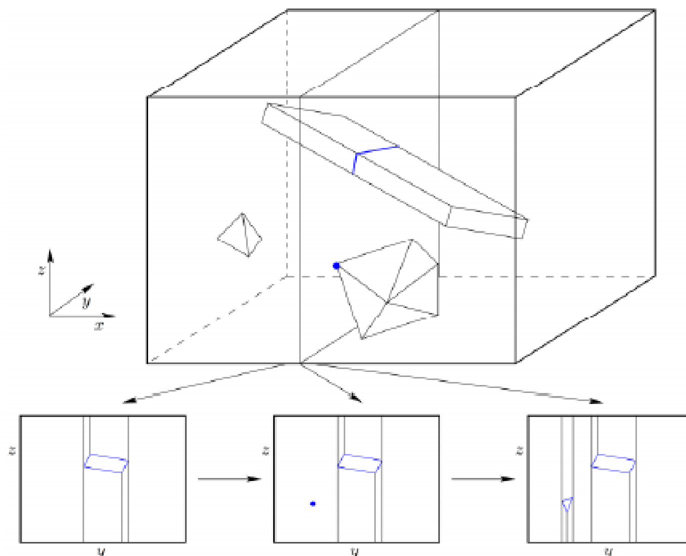

Figure 13: 3D Cell Decomposition Extended from 2D.

3D Vertical Decomposition Computing cell decomposition is much harder for higher-dimensional cases, though vertical cell decomposition is straightforward in 2D. We consider the special case where C_{obs} is piece-wise linear and polyhedral.

It turns out that we can extend the 2D vertical decomposition method by applying the idea of plane-sweeping to higher dimensions. Plane-sweeping refers to sweeping a plane

across the space, only to stop where critical change occurs in information. In 3D, assume a polyhedral robot can translate in $\mathbb{R}^3$, and the obstacles are polyhedral. Thus $C_{obs} \in \mathbb{R}^3$ is polyhedral as well. The 3D vertical decomposition algorithm is as follows (shown in Figure 13).

Let (x, y, z) denote a point in $\mathbb{R}^3$. Vertical decomposition yields 3-cell, 2-cell and 1-cell. A generic 3-cell is bounded by 6 planes, and its cross section for a fixed x yields a trapezoid or triangle in a plane parallel to the yz plane. Two sides of a 3-cell are parallel to the yz plane, and two other sides parallel to xz plane. The 3-cell is bounded above and below by two polygonal faces of C_{obs}.

The general idea is to sweep a plane perpendicular to x axis, where each fixed value of x produces a 2D polygonal slice of C_{obs}. Three example slices are shown in the bottom of Figure 13. Each slice is parallel to yz plane and simplify the 3D problem to a problem that can be solved by 2D vertical decomposition method. The middle slice shows the condition where the sweeping plane just encounters a vertex of a convex polyhedron, represented as a dot. This corresponds to an x value in interest, as critical change must occur in the slices. Hence, the 3D cell decomposition can be developed incrementally by sweeping through planes to update 2D vertical cell decomposition, in order to incorporate critical changes. To construct a roadmap, sample points are placed at center of each 3-cell and 2-cell. Edges are added to the roadmap by connecting each 3-cell to an adjacent 2-cell.

6.4 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, o.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)

References

[LaV06] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.