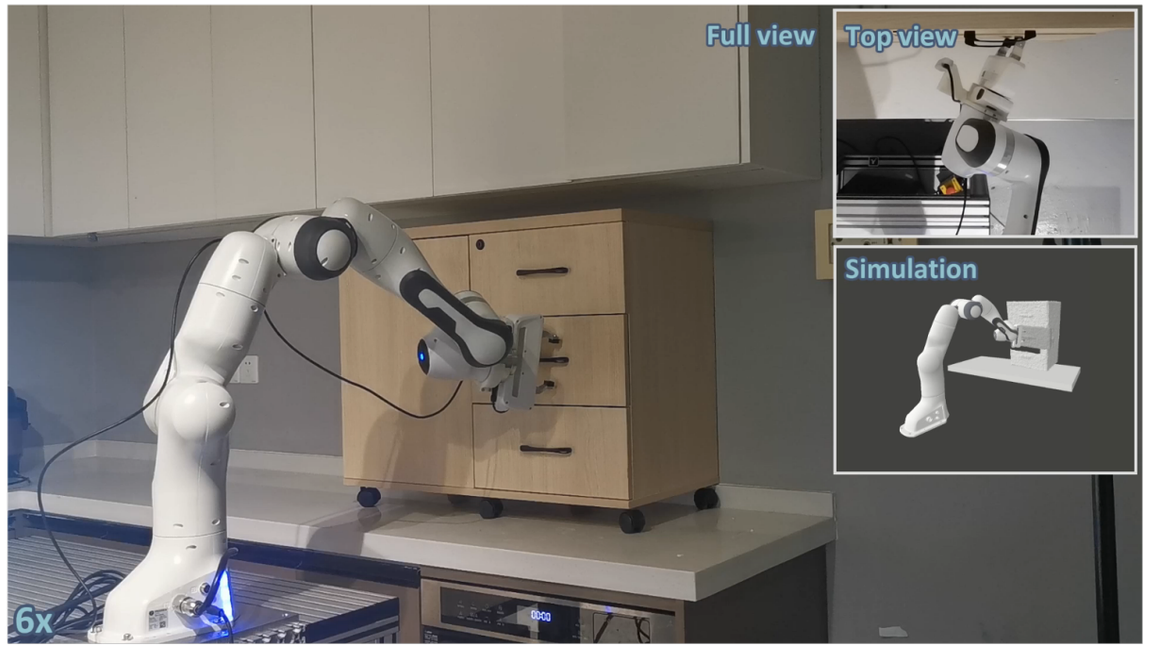
CS4278/CS5478 Intelligent Robots: Algorithms and Systems

Lin Shao

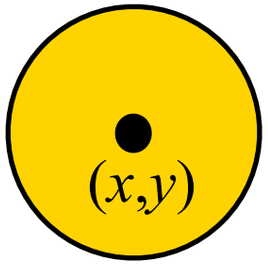
NUS

Motion Planning

Planning enables the robots to choose a sequence of actions and reach a specified goal

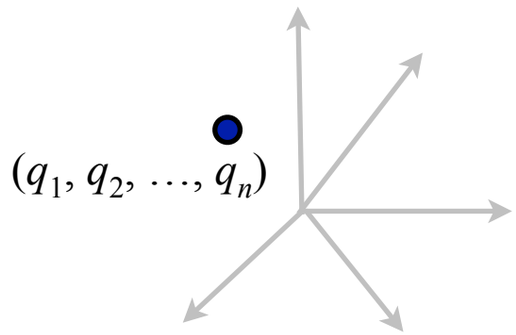


Configuration space. The configuration of a moving object is a set of parameters $(q_1, q_2, \ldots, q_n)$ that completely specifies of the position of every point on the object.

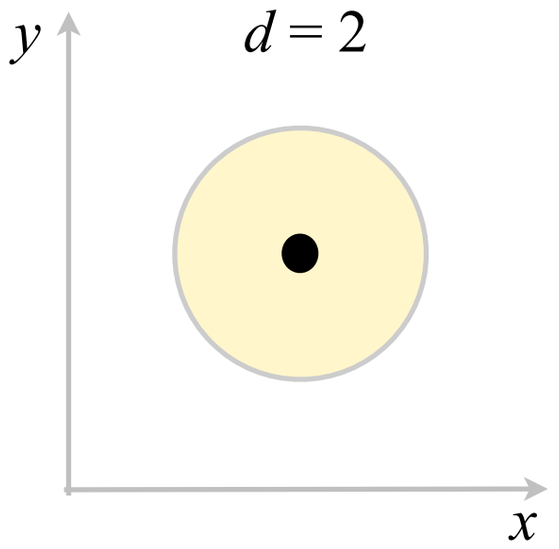


The coordinates for the disc center specify the configuration of a translation-only disc.

The configuration space (C-space) C is the set of all possible configurations. A configuration is a point in C.



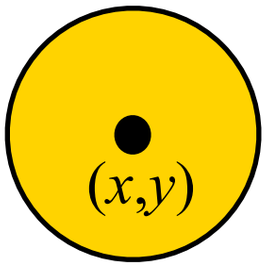
The dimension d of a configuration space is the minimum number of parameters required to specify the configuration of an object. It is also called the number of degrees of freedom (dofs) of a moving object.

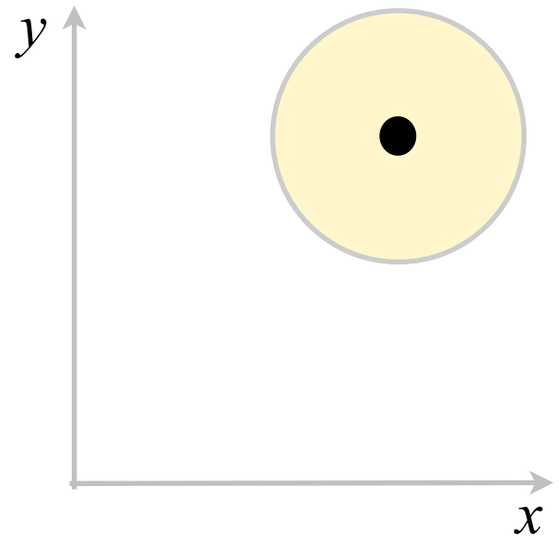


C-space topology.

Topology is the geometry of the space subject to continuous deformations. Intuitively, it is rubber-band geometry.

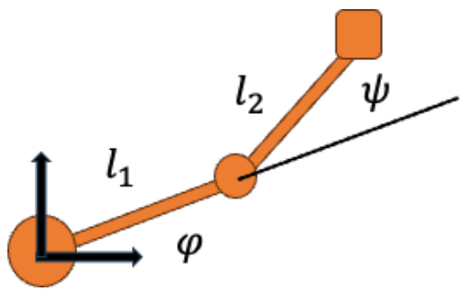
Example. R^2

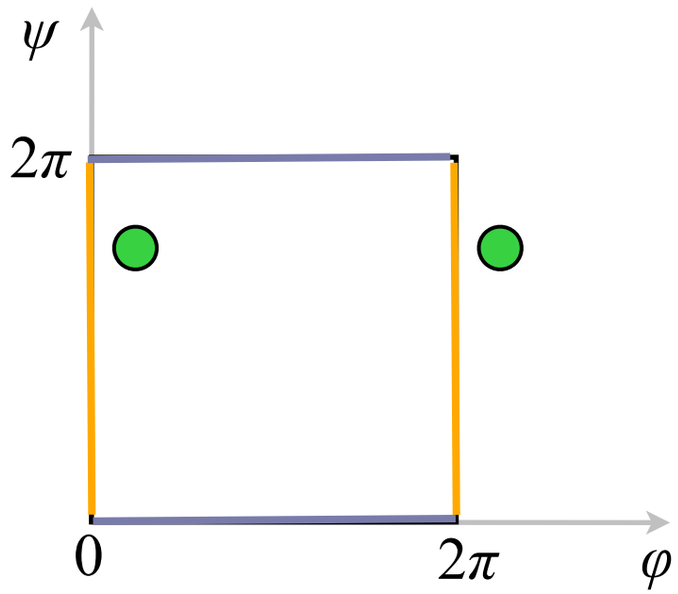


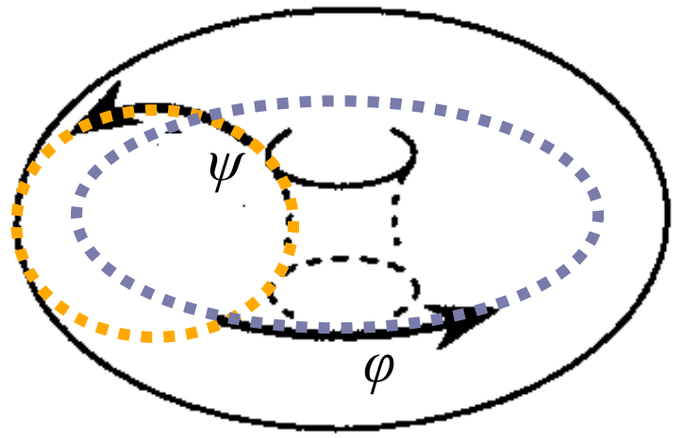


The center of the disc may take any position in the plane.

Example. $S^1 \times S^1$







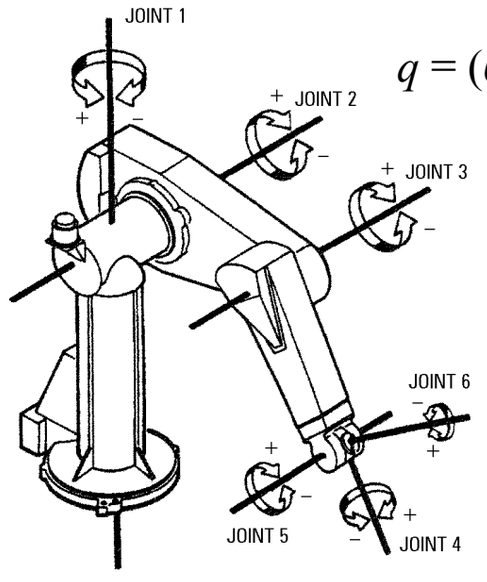
The opposing edges are identified, because 0 and 2π represents the same angle. The space thus “wraps around” to form a torus.

Articulated robots. The configuration is specified by $q = (\theta_1, \theta_2 \ldots, \theta_n)$.

The dimension of the configuration space C is n.

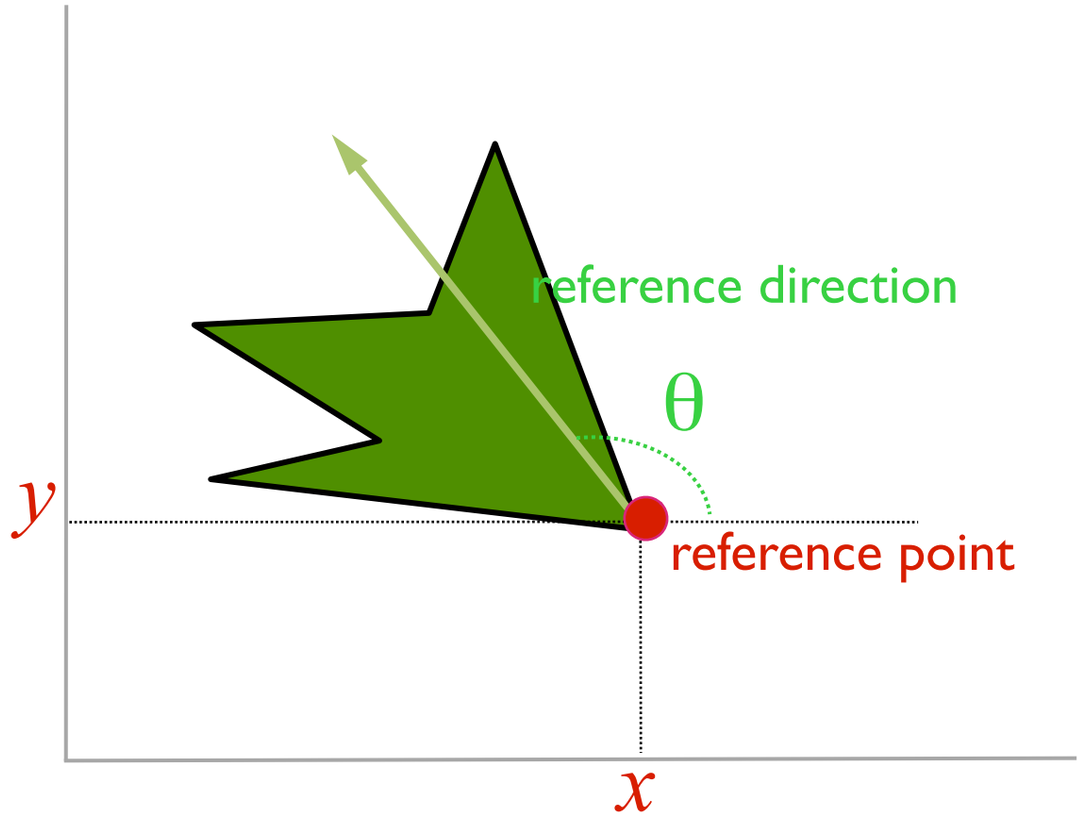
The topology of C is

- a hyper-cube in R^n if there are mechanical stops,
- $S^1 \times S^1 \times \ldots \times S^1$ if there are no mechanical stops.



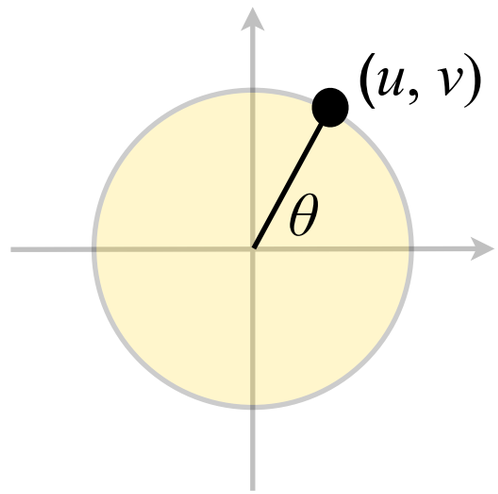
A mechanical stop sets the maximum angle by which a rotational joint can turn

The configuration space of a rigid body translating and rotating in 2D space.

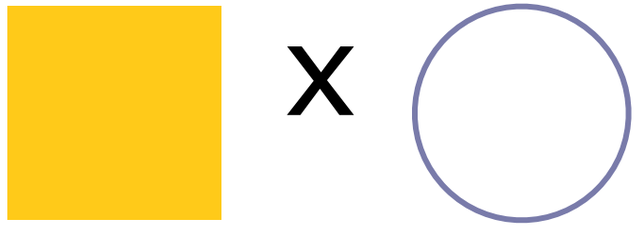


The configuration is specified by 3 parameters: $q = (x, y, \theta)$ with $\theta \in [0, 2\pi)$.

The configuration is specified by 4 parameters: $q = (x, y, u, v)$ with $u^2+v^2 = 1$. Note that $u = \cos\theta$ and $v = \sin\theta$.



The topology is a 3-D cylinder $C = R^2 \times S^1$.



Question. What is the dimension d of the configuration space for a moving object translating and rotating in a 2-D environment?

Question. Does the dimension of the configuration space depend on the parametrization? Does the topology depend on the parametrization?

The configuration space of a rigid body translating and rotating in 3D space.

$q = (x, y, z, ?)$

translation rotation

In 3D space, specifying the translation is straightforward by additional one coordinate. How do we specify the rotation?

Rotation matrix. An orthonormal matrix contains 9 parameters:

$$R = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}$$

with

- $r_{1i}^2 + r_{2i}^2 + r_{3i}^2 = 1$ for all i,
- $r_{1i}\, r_{1j} + r_{2i}\, r_{2j} + r_{3i}\, r_{3j} = 0$ for all $i \neq j$,
- $\det(R) = +1$

The matrix for rotating about the z-axis is

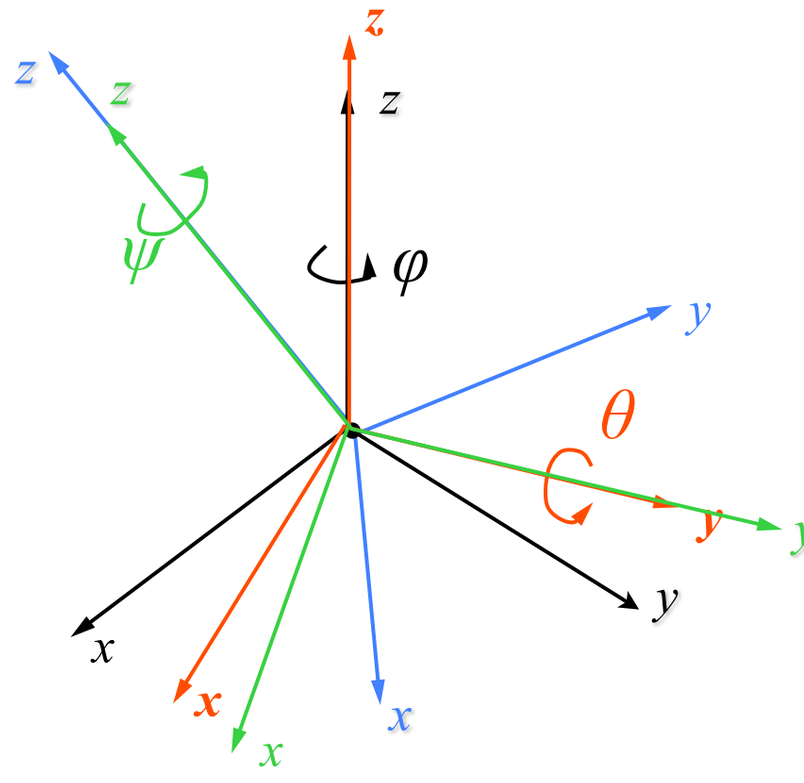
$$R_{z,\varphi} = \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

To confirm, we apply the matrix to the unit vector $(1,0,0)$ and check its effect:

$$\begin{pmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \cos\varphi \\ \sin\varphi \\ 0 \end{pmatrix}$$

Euler angles. Euler angles specify the rotation with 3 angles, one for each chosen axis. In our example, we have chosen z, y, and then z again. There are many valid combinations of axis choices. However, not all combinations are valid. For example, y-y-y is not valid, as it cannot represent rotation about the x-axis.

Euler angles are historically important, but have various undesirable limitations. Suppose that we perform two successive rotations, both represented as Euler angles. It is difficult to calculate the Euler angles for the cumulative final outcome.



1 → 2 → 3 → 4

Euler angles and rotation matrices are related. In our example, $R = R_{z,\varphi} R_{y,\theta} R_{z,\psi}$

Axis-angle. This specification contains 4 parameters: $q = (n_x, n_y, n_z, \theta)$ with $n_x^2 + n_y^2 + n_z^2 = 1$. (n_x, n_y, n_z) is a unit vector representing the axis of rotation, and θ is the angle of rotation about the axis. This is intuitive. However, it is not easy to compose rotations with the axis-angle specification, just like Euler angles.

Unit quaternion. This specification also contains 4 parameters: $u = (u_1, u_2, u_3, u_4)$ with $u_1^2 + u_2^2 + u_3^2 + u_4^2 = 1$. The intuition is similar to that of the axis-angle specification. It is connected to the axis-angle specification through the following relationship:

$$(u_1, u_2, u_3, u_4) = (\cos\frac{\theta}{2}, n_x \sin\frac{\theta}{2}, n_y \sin\frac{\theta}{2}, n_z \sin\frac{\theta}{2})$$

with $n_x^2 + n_y^2 + n_z^2 = 1$. Basically, it is the axis-angle specification re-coded into a different set of 4 numbers.

Compare unit quaternion specification of 3D rotations with the (u,v)-representation of 2-D rotations: (u,v) with $u^2+v^2 = 1$.

The unit quaternion representation offers many advantages over the alternatives:

- Compact
- Naturally capture the topology of 3-D rotation space
- No singularity

It is also easy to compose rotations directly through the quaternion algebra.

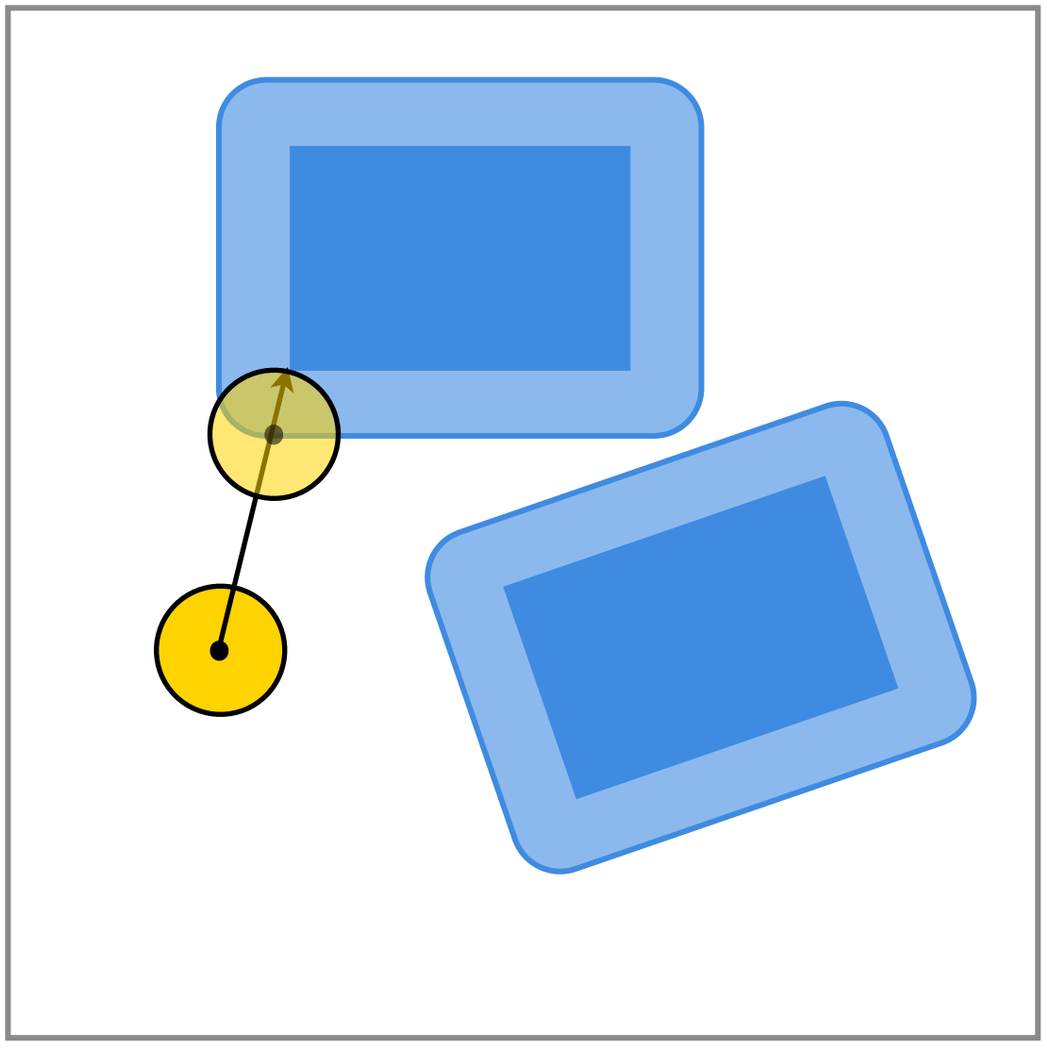
The unit quaternion representation is usually the preferred choice for representing rotations.

C-space obstacles. A configuration q is collision-free, or free, if a moving object placed at q does not intersect any obstacles in the workspace.

The free space F is the set of all free configurations.

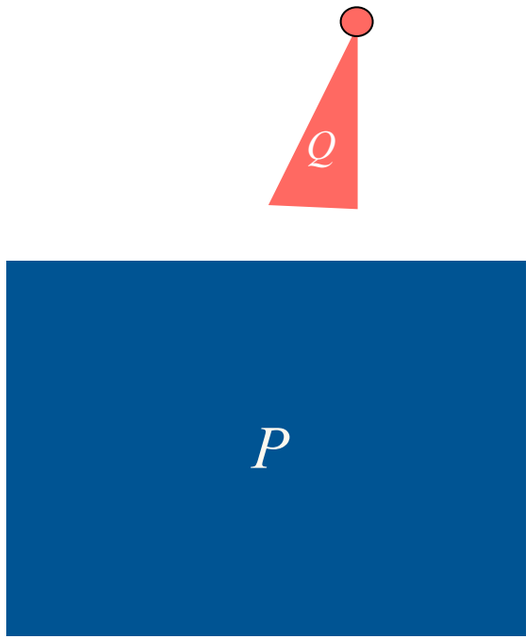
A configuration-space obstacle (C-obstacle) is the set of configurations where the moving object collides with the workspace obstacles or with itself.

In this example, the “expanded” obstacles are in fact C-space obstacles for a disc robot.

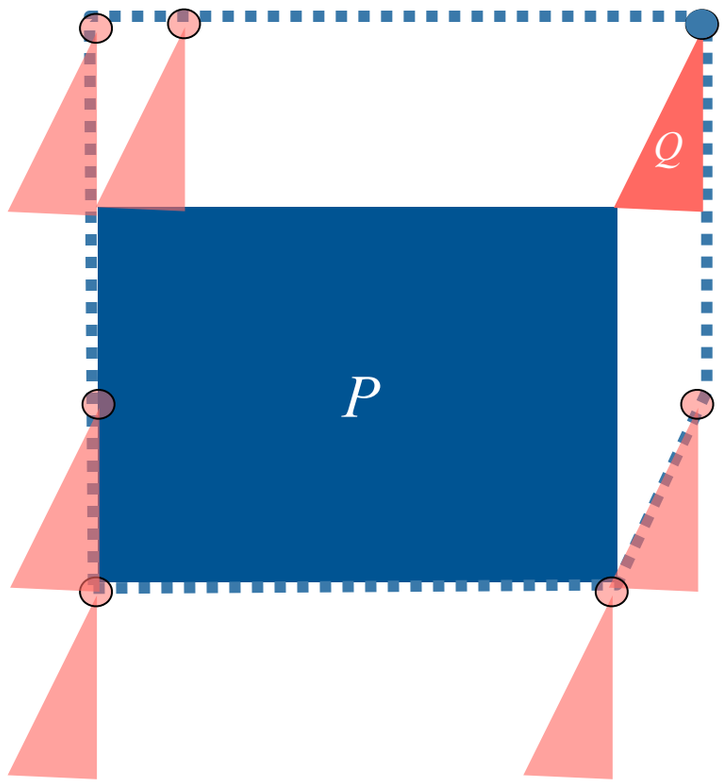


For a polygonal robot, how do we expand the obstacles and compute the C-space obstacles?

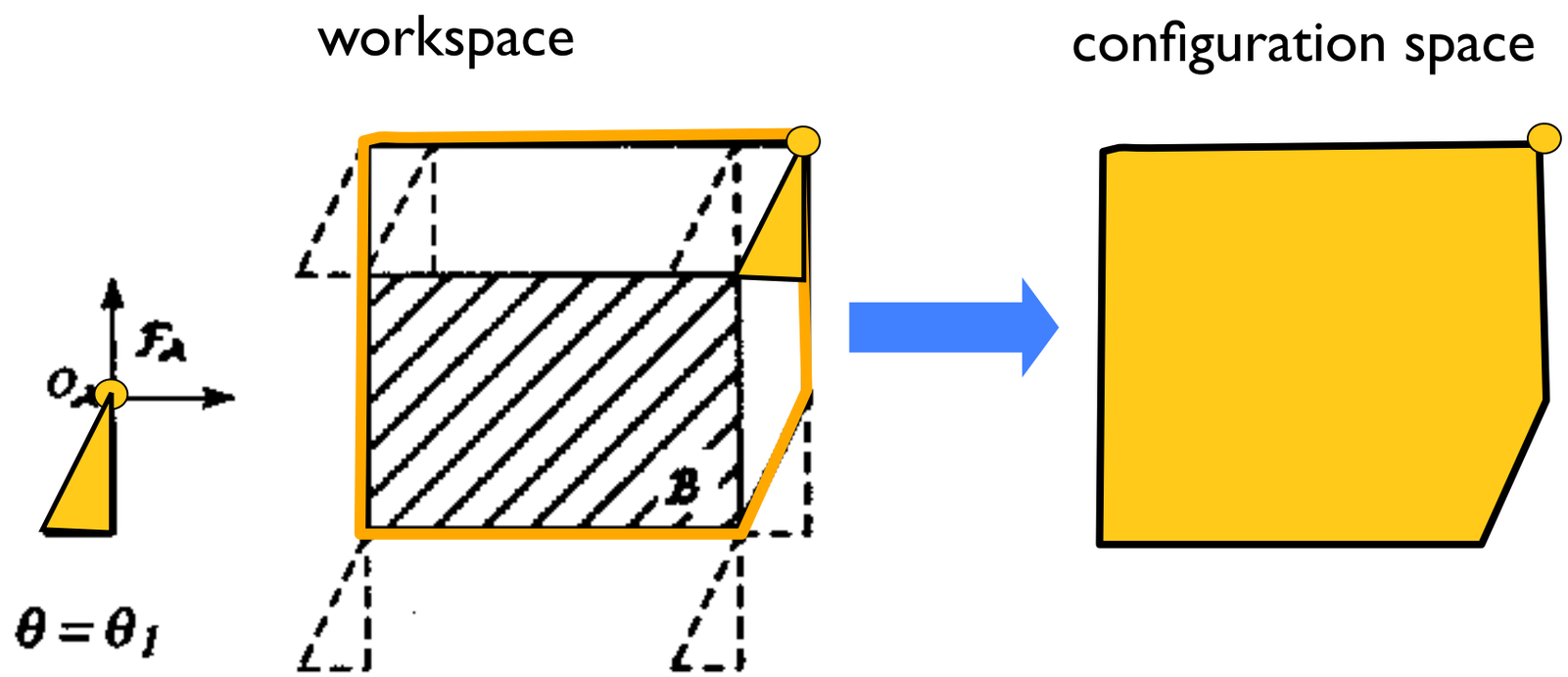
workspace



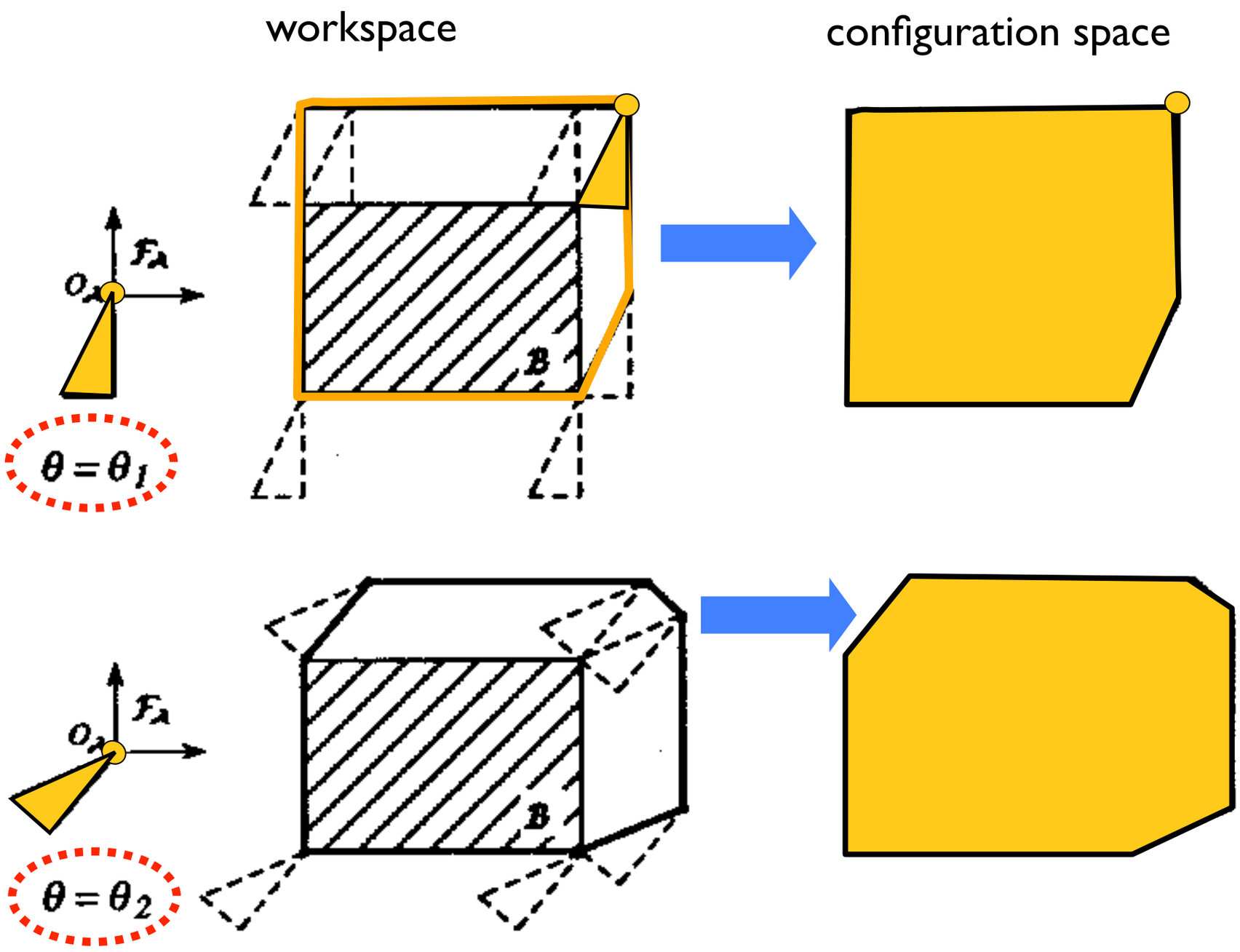
configuration space



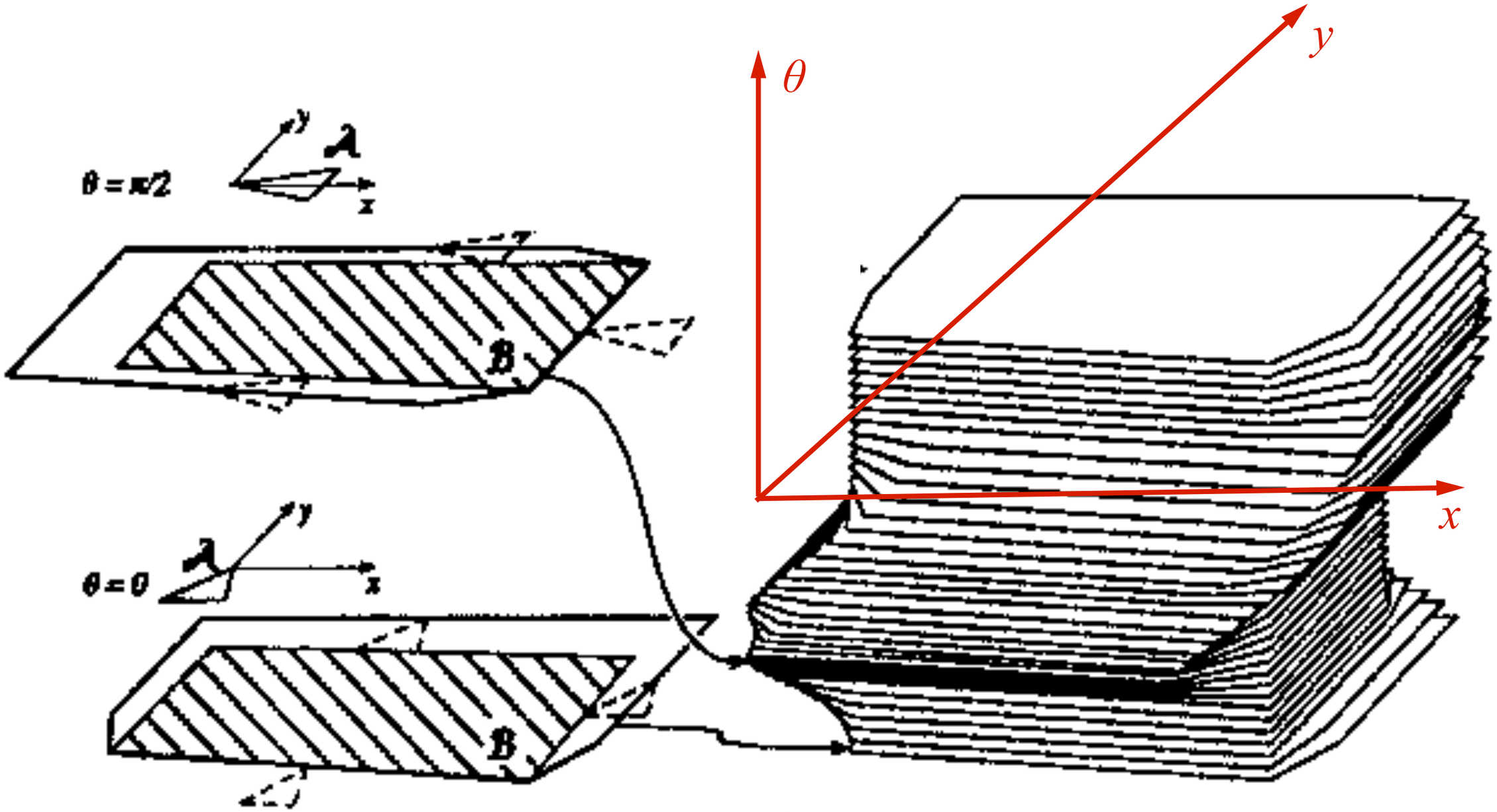
Compute the C-space obstacles for a polygonal robot translating in 2-D workspace.



Compute the C-space obstacles for a polygonal robot translating and rotating in 2-D workspace.

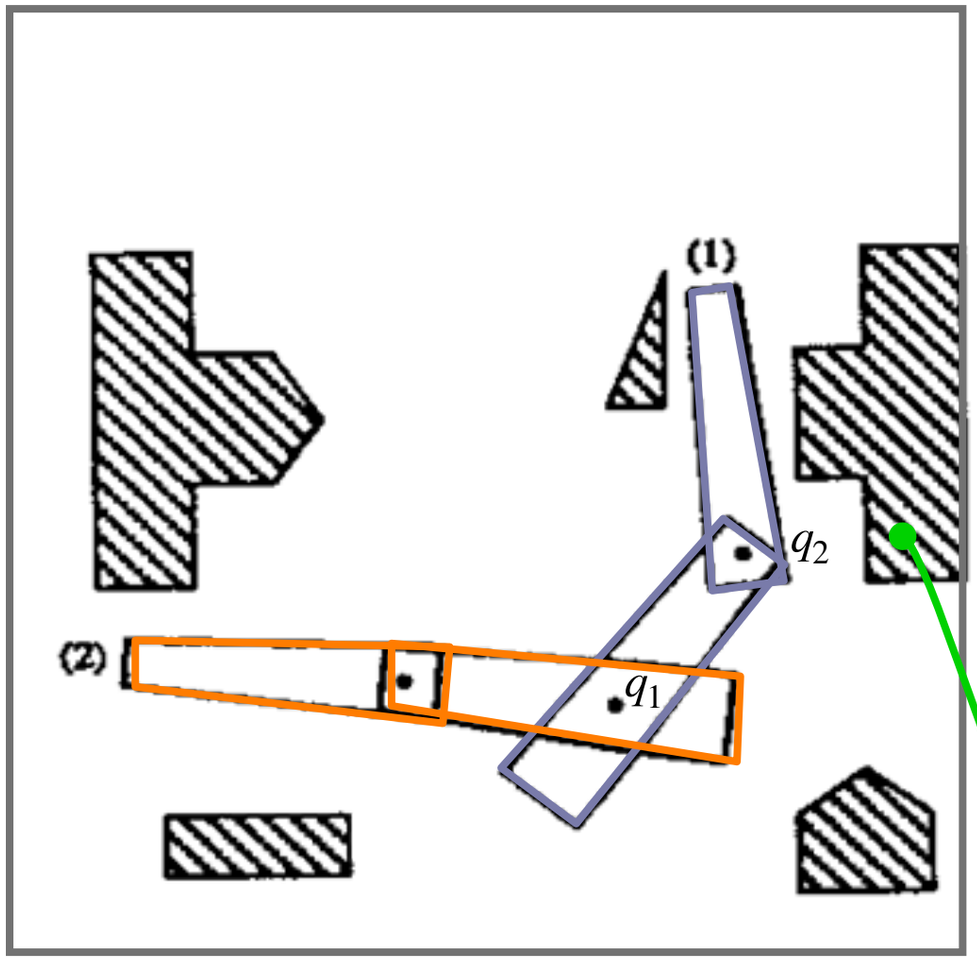


Computing the C-space obstacles for a polygonal robot translating and rotating in 2-D workspace.

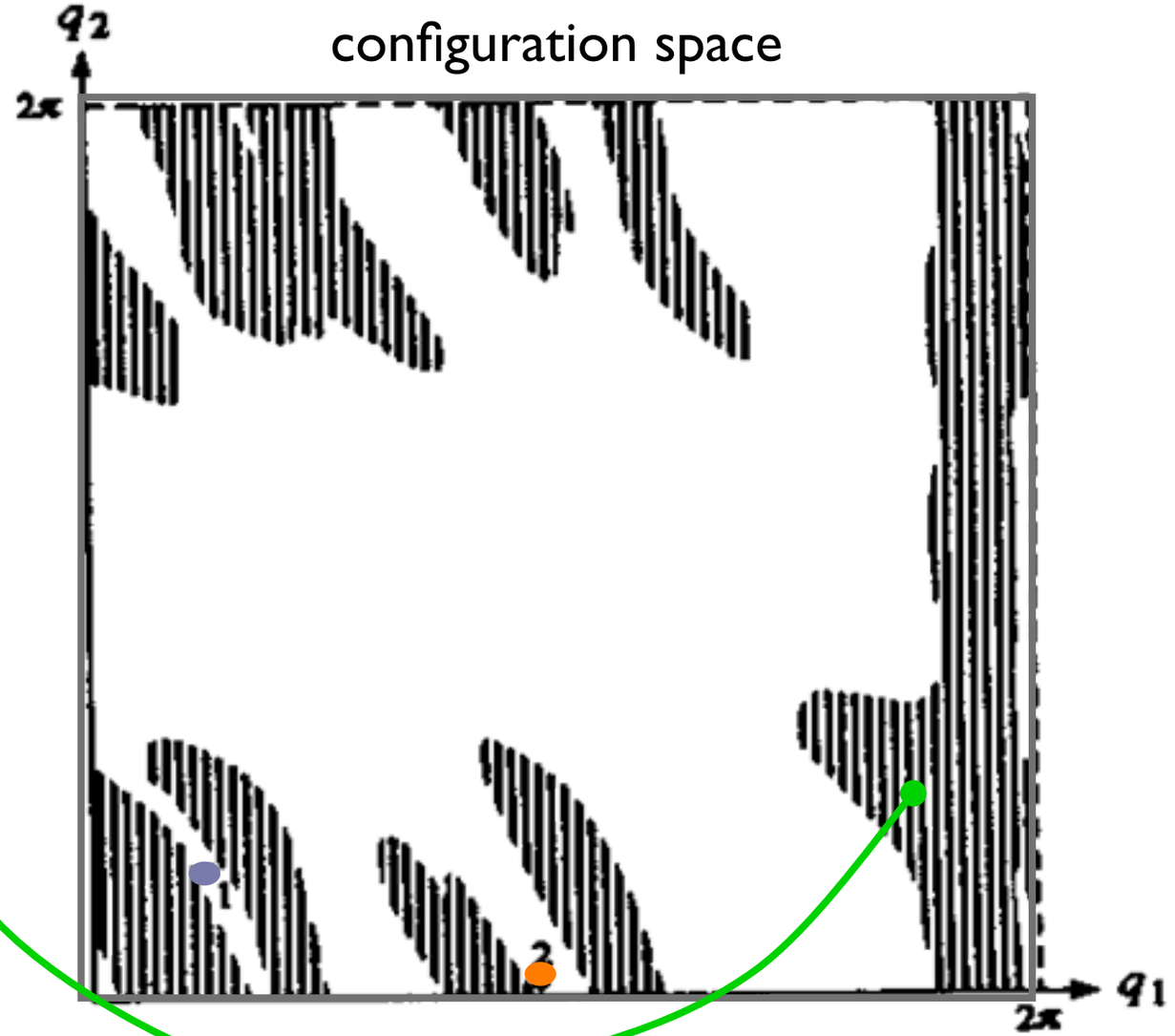


Compute the C-space obstacles for an articulate robot.

workspace



configuration space



q_1: based joint angle wrt the horizontal axis

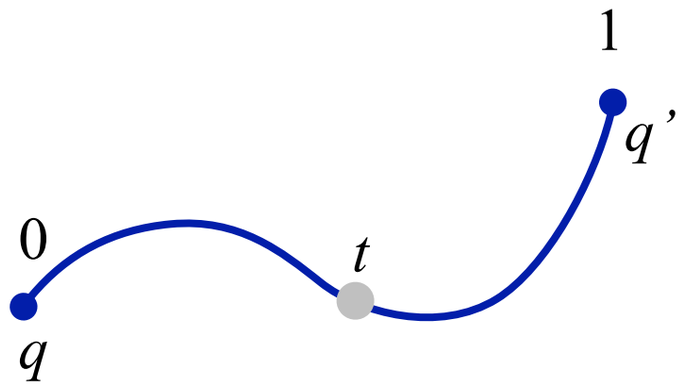
q_2: joint angle of link 2 wrt to the link 1

C-space metrics. A metric or distance function d in a configuration space C satisfies

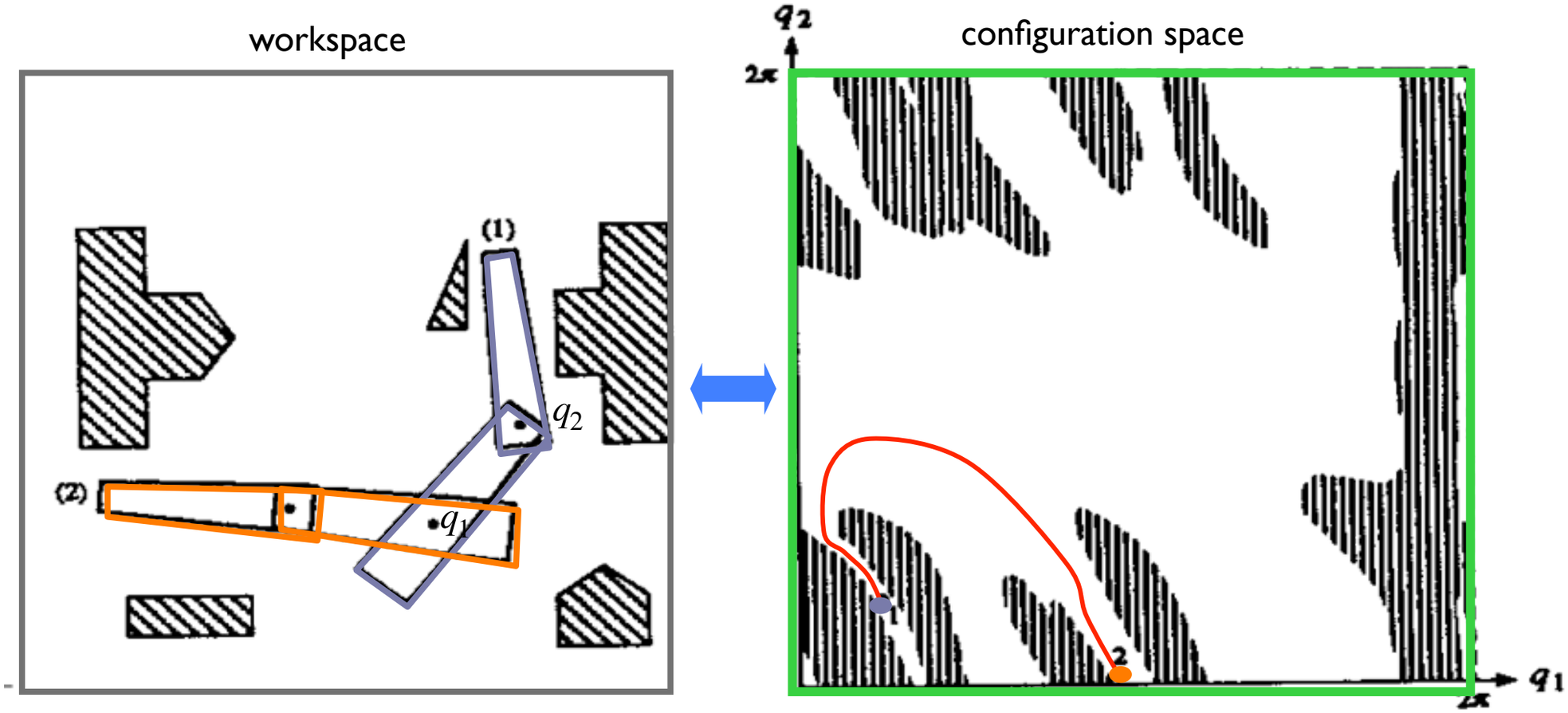
- $d(p,q) = 0$ if and only if $p = q$,
- $d(p,q) = d(q,p)$,
- $d(p,q) \leq d(p,r) + d(r,q)$,

for all $p,q \in C$. A familiar example is the Euclidean metric in the plane.

C-space paths. A path in C is a continuous function $\tau(t)$ connecting two configurations q and q' such that $\tau(0) = q$ and $\tau(1)=q'$.



Example. The path in the C-space corresponds to the motion in the workspace between the two marked configurations. Can you imagine the motion in between?



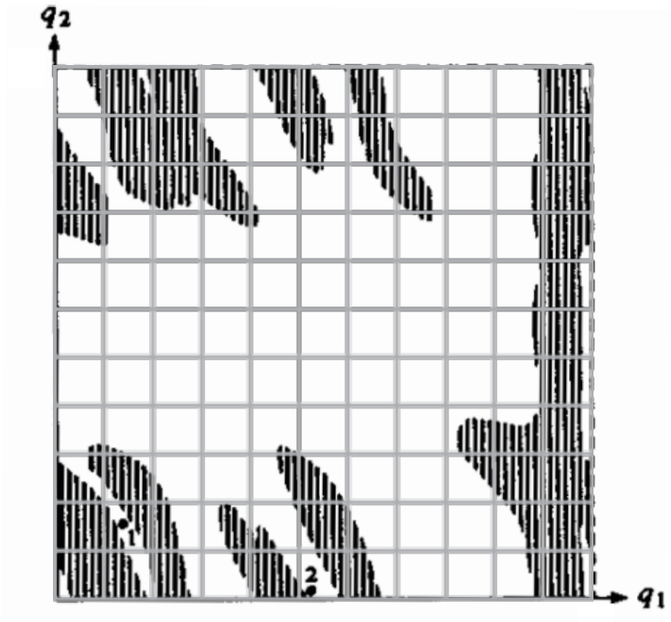
Path constraints. We can place various constraints on C-space paths to induce desirable motions.

- Bounded curvature
  Example. A car has a minimum turning radius.
- Nonholonomic constraints
  Example. A car cannot move sidewise.
- ...

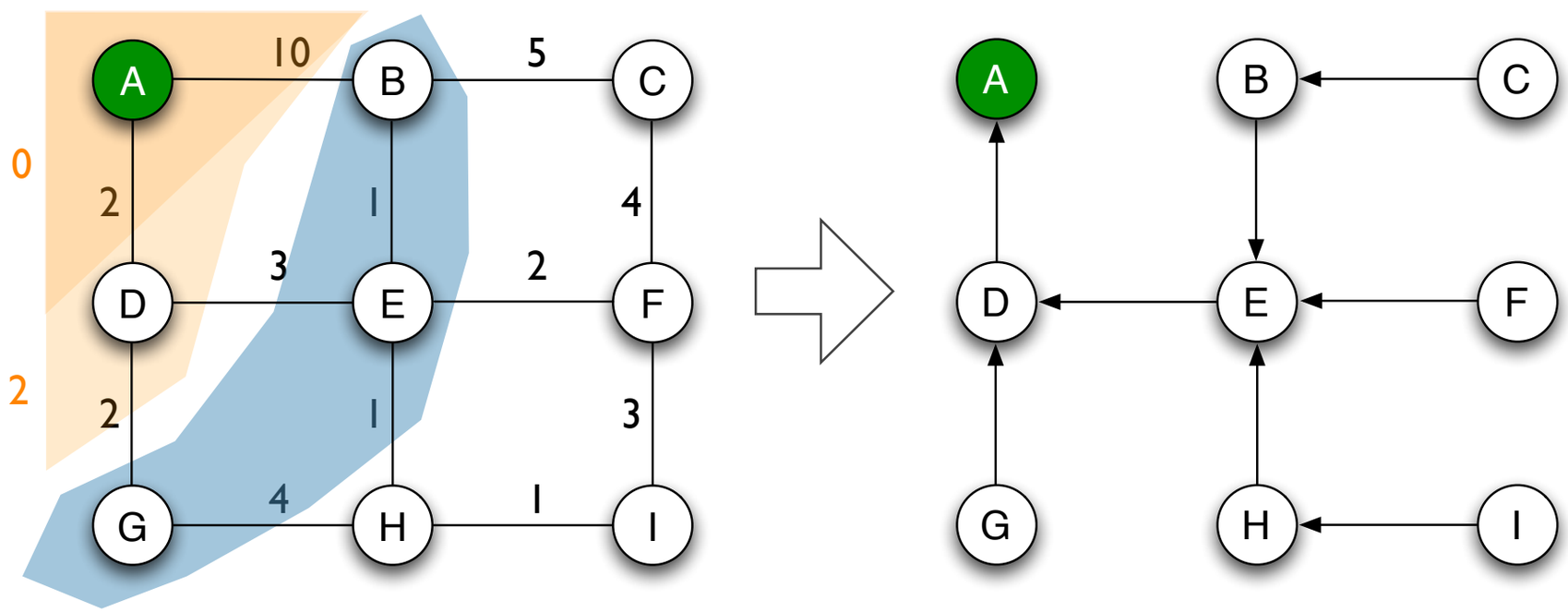
Summary.

- The workspace is the physical space that contains the robot and the obstacles. The configuration space represents the robot posture as a point along with transformed obstacles. The configuration space provides a general and conceptually elegant representation framework.
  - The posture of a robot corresponds to a point in the configuration space.
  - The motion of a robot corresponds to a path in the configuration space.
- Both spaces contain exactly the same information.
  - Given the geometry of a robot and obstacles, we can construct the configuration space (in principle).
  - Given the configuration of a robot, we can determine the position of every point on the robot.

We place a grid of a suitable resolution on the C-space and create a cell-decomposition graph.



Finally, we search the graph, using a graph-search algorithm, e.g., the Dijkstra's algorithm for the shortest path.



Backward dynamic programming. Dynamic programming is a related, but different algorithmic approach to the shortest path problem. It is a very powerful, general idea for not only the shortest-path problem, but also a wide range of optimization problems. We use it for the shortest path here, but will encounter this idea several times later for other problems.

"Configuration" and "state" are often used interchangeably, though "state" is a more general notion. Here we follow the standard convention of DP algorithms and use "state".

Let $V(s)$ be the length of the shortest path to the goal for state s.

- Initialize
  - If s is a goal state, $V_0(s) = 0$
  - Otherwise $V_0(s) = +\infty$
- If there are t time steps to go, recurse

the edge cost from s to s'.

Node expansion $V_t(s) = \min_{s' \in A(s)} \{c(s, s') + V_{t-1}(s')\}$

nodes adjacent to s

for $t = 1, 2, 3, \ldots, N\text{-}1$, where $A(s)$ is the set of nodes adjacent to s, including s.

It is sufficient to choose N to be the total number of nodes, as a shortest path visits each node at most once.

Dynamic programming finds the shortest path from every state to a goal.

Both the computation time and the storage space required depend on the size of the state space, i.e., the **total number of states**.

Question. Would dynamic programming work well for

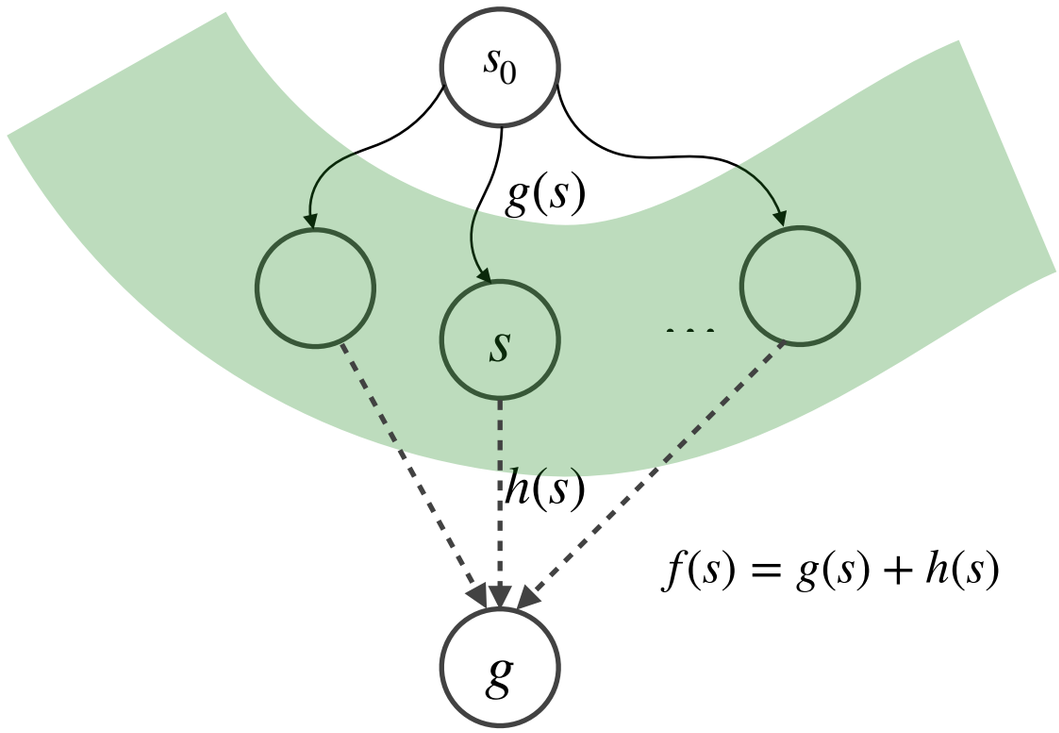
- a rigid body translating and rotating in the plane?
- a hyper-redundant robot arm consisting of 10 rigid pieces linked together at the joints?

Forward search. Dynamic programming is not practical for very large state space. In practice, we often care about the shortest path from the current state s_0.

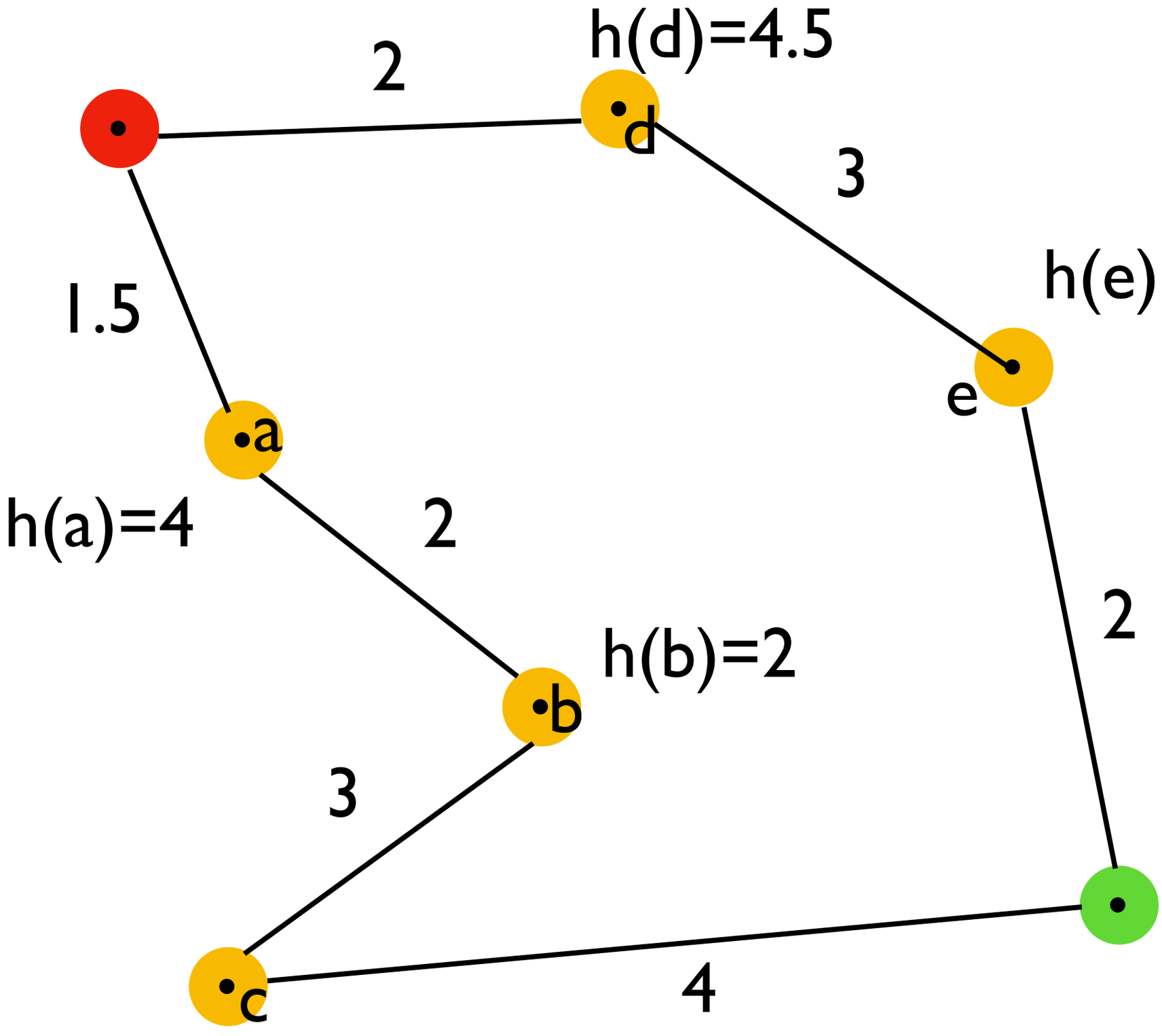
The forward search algorithm starts at the current state and performs lookahead search until it reaches a goal state.

- Start at the current state s_0 .
- Maintain a priority queue Q. Insert s_0 into Q.
- Choose a node s from Q, according to the priority $f(s) = g(s) + h(s)$. For each node s' incident to s, insert s' into Q with updated value of $f(s)$. Repeat until reaching a goal node g.

Node expansion



Intuitively, g(s) is the cost-to-come (from s_0 to s) and h(s) is the cost-to-go (from s to g).

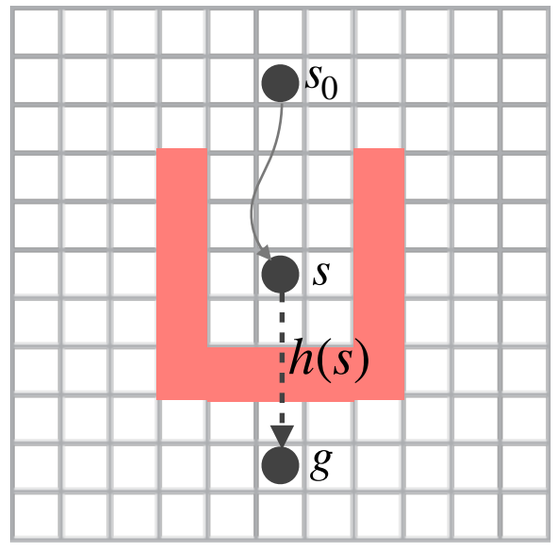

h(d)=4.5
2
d
3
h(e)=2
1.5
e
a
h(a)=4
2
2
h(b)=2
b
3
4
c
h(c)=4

$g(s)$ is the length of the shortest path found so far to go from s_0 to s. $h(s)$ is a heuristic estimating the shortest path from s to g. $g(s)$ represents the "true" shortest path information, found through search; $h(s)$ represents our guess.

- Uninformed search: $f(s) = g(s)$
  Uninformed search sets $h(s) = 0$, effectively ignoring the heuristic. It finds the shortest path. However, the search does not consider the goal and is thus inefficient. An example is Dijkstra's algorithm.

- Greedy search: $f(s) = h(s)$
  Greedy search ignores g(s), effectively not remembering anything about the past search and only trusting the heuristic estimate $h(s)$. However, the heuristic is sometimes misleading. Greedy search may get stuck in a local minimum and fail to find a shortest path.

- A* search: $f(s) = g(s) + h(s)$
  A* search combines information from "cost-to-come" $g(s)$ and "cost-to-go" $h(s)$. It finds the shortest path if the heuristic function $h(s)$ is **admissible**.



Admissibility. A heuristic function $h(s)$ is admissible if it provides an optimistic estimate, in other words, an underestimate of the true shortest path length from s to the goal.

Efficiency. Good heuristics significantly improve computational efficiency of search algorithms. Consider the following two extremes. One extreme is an uninformed heuristic $h(s) = 0$. It is clearly admissible, but not very useful. Suppose that we use it to search a grid graph of 1000x1000, with the lower-left corner and the upper-right corner, as the start and the goal, respectively. With $h(s) = 0$, the A* algorithm expands all1,000,000 nodes, just like the Dijkstra's algorithm. The other extreme is a fully informed heuristic. With the fully informed heuristic, the A* algorithm expands only the nodes on the shortest path, i.e., only about 2,000 nodes in total.

Question. What is the fully informed heuristic?

The computation time of forward search does not depend on the total number of states directly. What does it depend on then? The computation time of a tree search algorithm depends on the **branch factor** and the **depth** of the search tree.

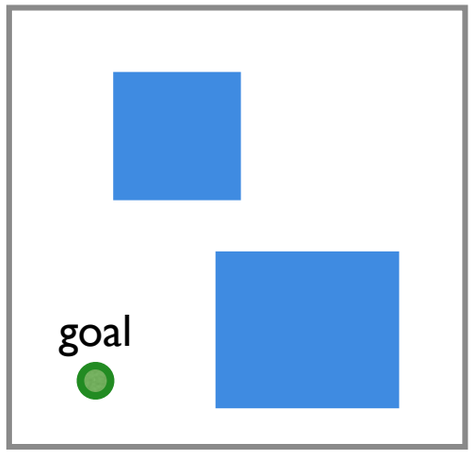
Heuristics are critical to the performance of forward search algorithms. Here are some general ideas for obtaining effective heuristics:

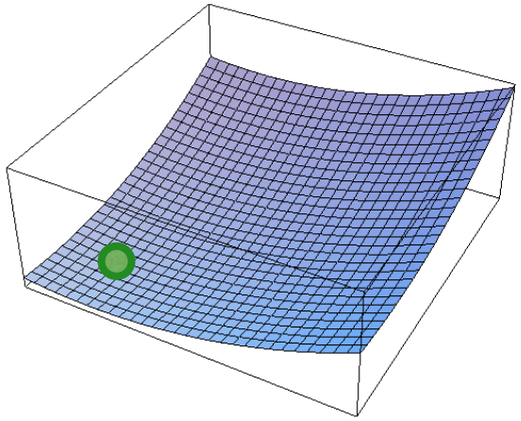
- Use domain-specific knowledge.
- Relax motion constraints the path and calculate the cost of the relaxed path.
- Learn heuristics from experiences.
    - Ask ChatGPT (2023 AD) 😊

A common heuristic function for robot path planning is the Euclidean distance, i.e., the length of the straight-line path to the goal.

Question. Is this heuristic admissible?

Artificial potential field. In essence, the well-known potential field method for motion planning is simply a handcrafted heuristic function based on *a priori* domain knowledge.

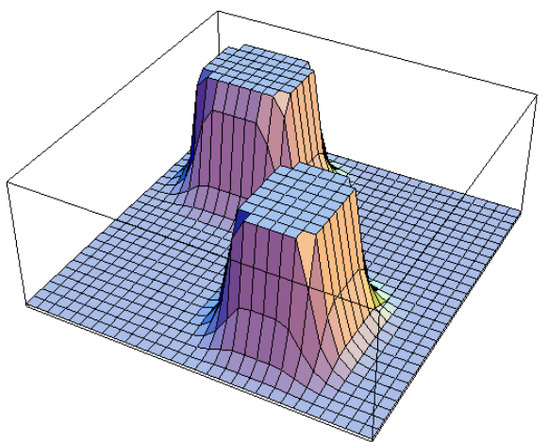




Attractive potential

Activate the attractive potential if the distance to the goal is smaller than a chosen value P.

$$U_{\text{att}}(q) = \frac{1}{2}\xi \left(d(q, q_{\text{goal}}) \right)^2 \quad \text{if } d(q, q_{\text{goal}}) \leq P$$



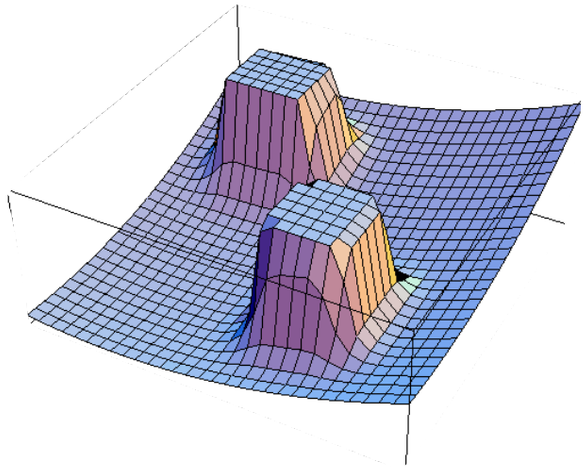
Repulsive potential

Activate the repulsive potential if the distance to an obstacle is smaller than a chosen value Q.

For each obstacle,

$$U_{\text{rep}}(q) = \frac{1}{2}\eta \left(\frac{1}{D(q)} - \frac{1}{Q} \right) \quad \text{if } D(q) \leq Q$$

distance from q to the obstacle



$$U(q) = U_{\text{att}}(q) + U_{\text{rep}}(q)$$

Choose an action to follow the gradient of the artificial potential field $\nabla U = \nabla U_{\text{att}}(q) + \nabla U_{\text{rep}}(q)$.

Question. As a heuristic, the artificial potential field is intuitively appealing. How does it fail?

Learn a heuristic. The greedy algorithm finds an optimal path if it uses the fully informed heuristic, i.e., the true optimal path length. Of course, the fully informed heuristic is not known in advance, but maybe we can **learn** to approximate it.

From the robot's current state s, it chooses the next s' according the estimated cost-to-go:

node expansion

$$f(s) = \min_{s' \in A(s)} \{c(s, s') + h(s')\}$$

In the standard greedy search, the heuristic h is chosen in advance and does not change during the search. To learn, we set

$$h(s) = f(s)$$

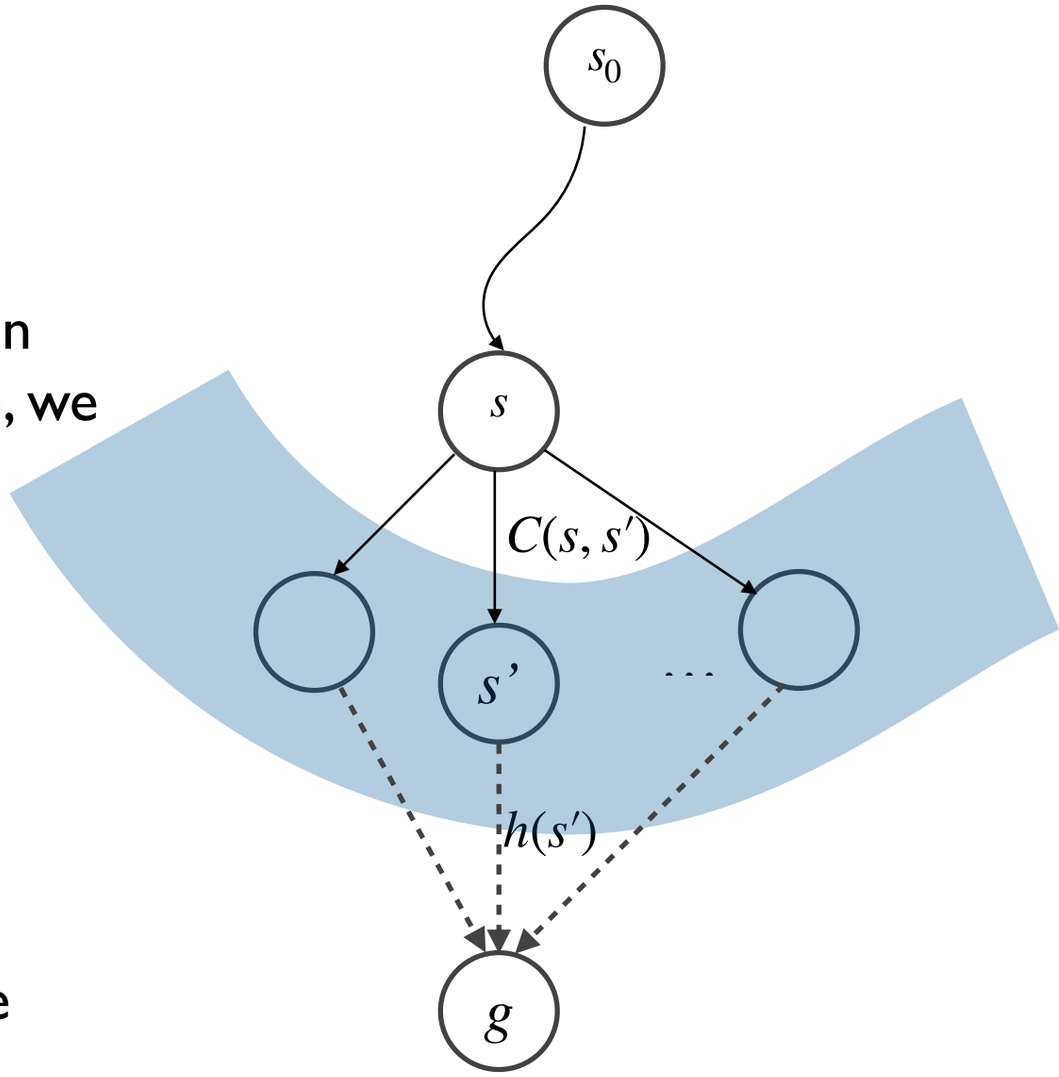
in each step to get an improved heuristic.

Putting the two steps together, we have

$$h(s) \leftarrow \min_{s' \in A(s)} \{c(s, s') + h(s')\}$$

Why does this improve the heuristic? Compare with the recurrence for dynamic programming

$$V_t(s) = \min_{s' \in A(s)} \left\{c(s, s') + V_{t-1}(s')\right\}$$



So heuristic improvement is simply one step of dynamic programming at a particular state s. If we repeat this enough number of steps over all states, then $h(s) = V^*(s)$, i.e., the true shortest path length, the perfect heuristic.

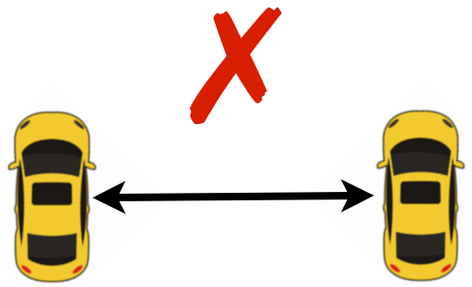
This is a simple example of **learning** by memorizing past experiences.

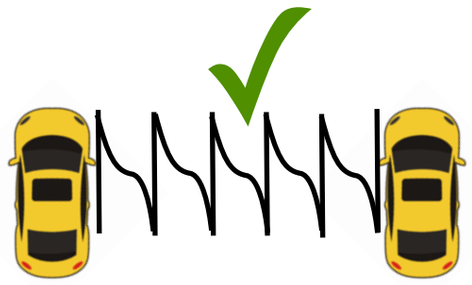
Our derivation also shows that we can perform t steps of dynamic programming for any t and use the resulting $V_t(s)$ as a heuristic function.

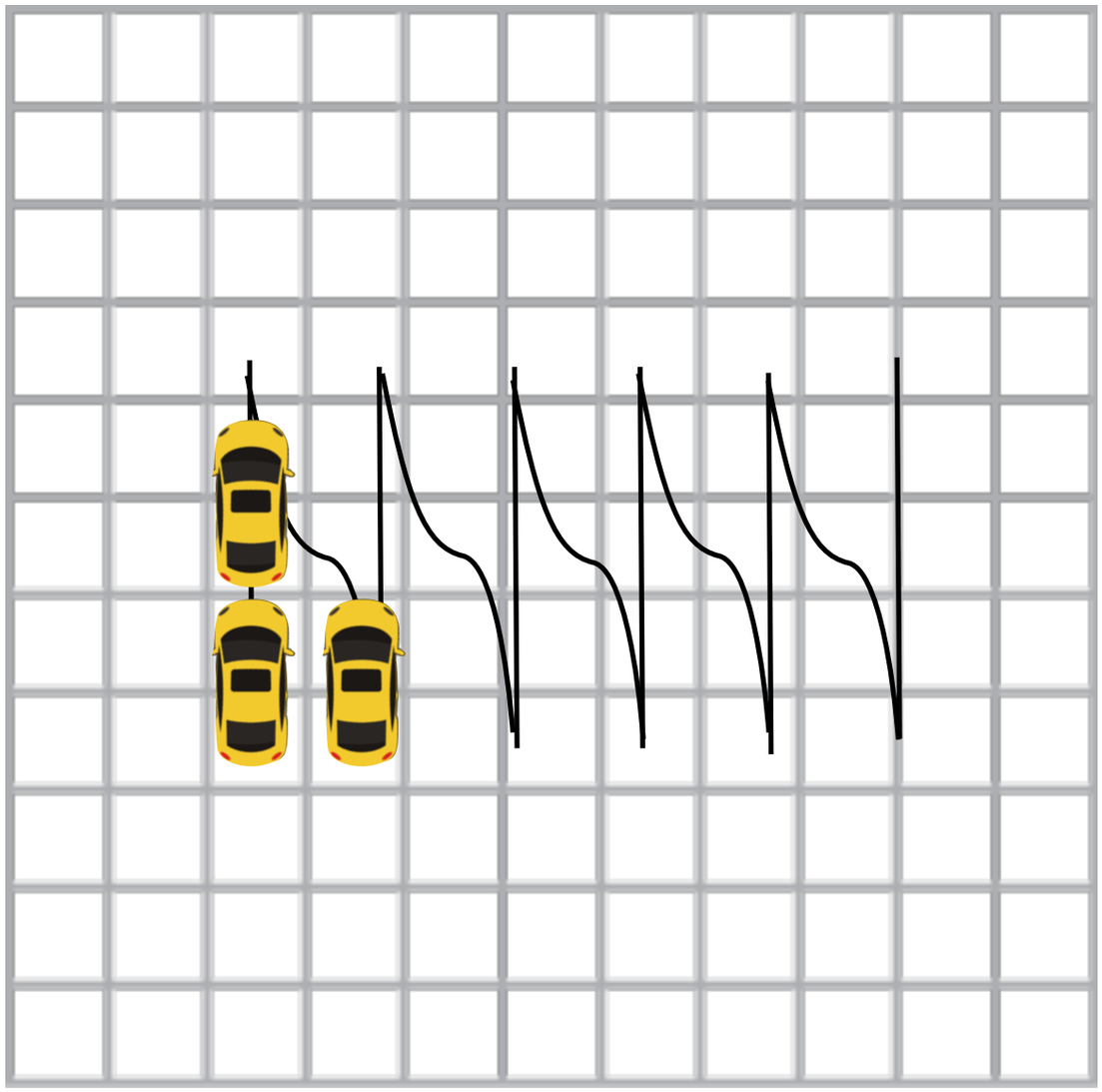
This is the main idea of learning real-time A* (LRTA*).
For more details, see

H. Geffner and B. Bonet. Solving large POMDPs using real time dynamic programming. In Working Notes of AAAI 1998 Fall Symposium on Planning with Partially Observable Markov Decision Processes, 1998.

Question. Can we use the cell-decomposition graph to plan the motion of a car?







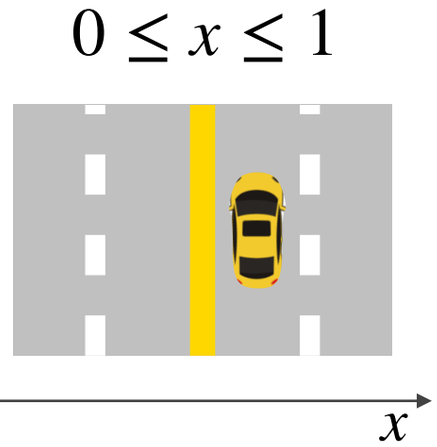
Constraints on motion. Robot motion is subject to various constraints.

A constraint is holonomic if it involves only the configuration q:

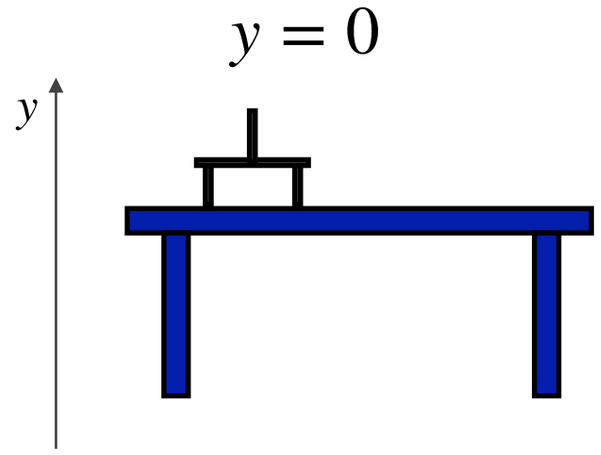
$$g(q) = 0 \text{ or } g(q) \leq 0$$

A holonomic constraint restricts the set of valid configurations.

Example. The in-lane driving constraint is holonomic.



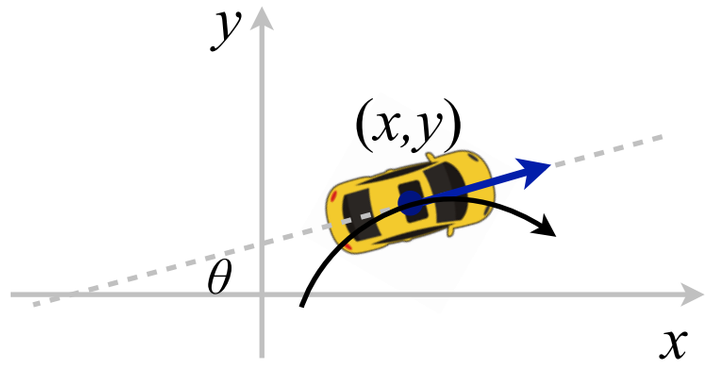
Example. The contact constraint is holonomic. The constraint $y = 0$ ensures that the robot gripper is in contact with the tabletop.



A constraint is nonholonomic if it involves both configuration q and configuration velocity $\dot{q}$:

$$g(q, \dot{q}) = 0 \text{ or } g(q, \dot{q}) \leq 0$$

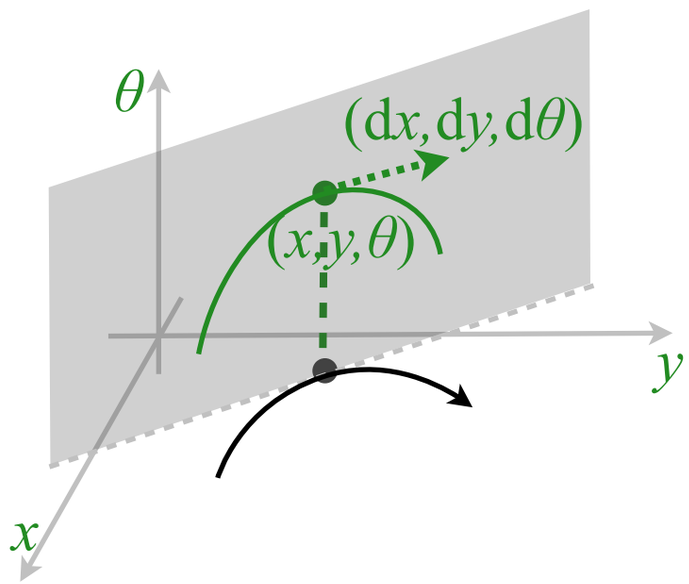
A nonholonomic constraint restricts the set of valid configuration velocities, but not necessarily the set of configurations.



Example. A car cannot drive sidewise. The instantaneous velocity of the car must align with the current orientation of the car:

$$\tan\theta = \frac{\dot{y}}{\dot{x}}$$
$$\Rightarrow \dot{x}\sin\theta - \dot{y}\cos\theta = 0$$

The constraint involves both the configuration $q = (x, y, \theta)$ and the configuration velocity $\dot{q} = (\dot{x}, \dot{y}, \dot{\theta})$. This constraint restricts the instantaneous velocity of the car. However, it does not restrict the set of reachable configurations: a car still can reach any arbitrary configuration under the constraint. Otherwise, the car would not be very useful!



Under the above constraint, we can write down the system equation as

$$\begin{aligned} \dot{x} &= v\cos\theta \\ \dot{y} &= v\sin\theta \\ \dot{\theta} &= (v/L)\tan\phi \end{aligned}$$

speed v

orientation θ

steering angle ϕ

distance L between front and rear wheel axes

Which indeed satisfied the nonholonomic constraint. It is a controlled dynamic system of the form

$$\dot{q} = f(q, u),$$

where control u consists of the speed v and steering angle ϕ.

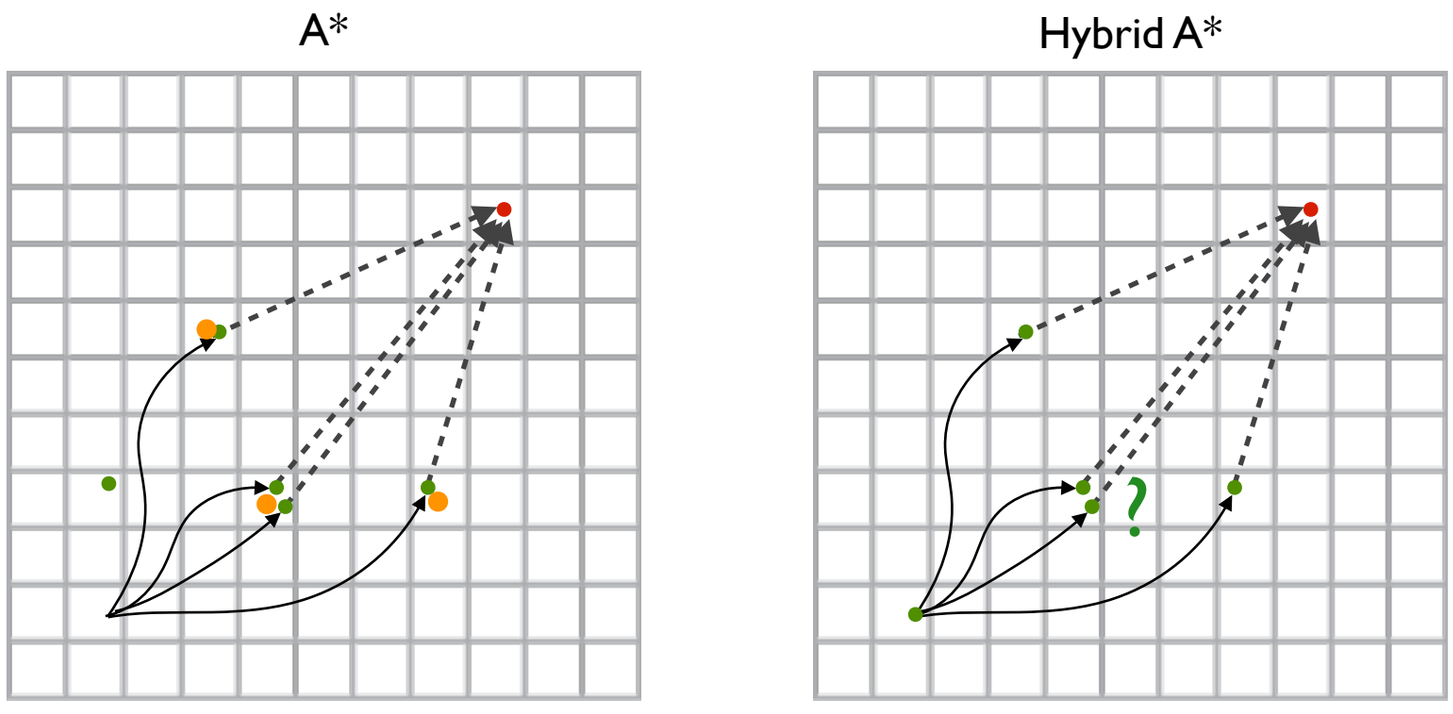
Suppose that we apply some control u at configuration q for a fixed time duration Δt. We arrive at a new configuration

$$q_{t+1} = q_t + \dot{q}_t \Delta t,$$

thus allowing us to "connect" q_t to q_{t+1} via the control u.

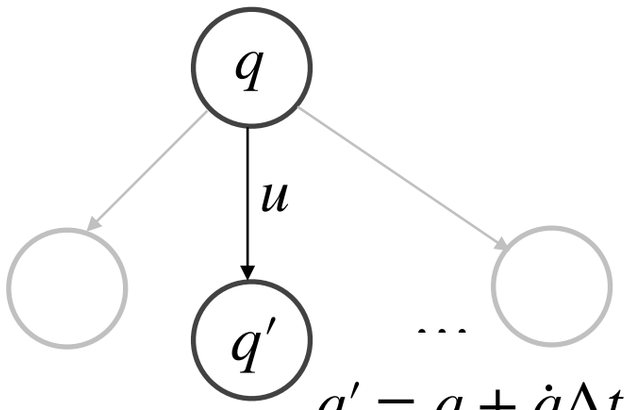
Hybrid A*. The standard A* algorithm associates with each node a configuration at the center of a grid cell. The hybrid A* algorithm associates with each node a **continuous** configuration $q = (x, y, \theta)$.

Question. Is it a good idea to associates with each node a configuration at the center of a grid cell? Why must we use a continuous configuration?



If the search reaches a node visited earlier, we can retain only one associated continuous configuration, the one with lower value of $f(q)$.

We now apply the hybrid A* algorithm to car driving. To expand a node,



$q' = q + \dot{q}\Delta t$, where $\dot{q} = f(q, u)$,

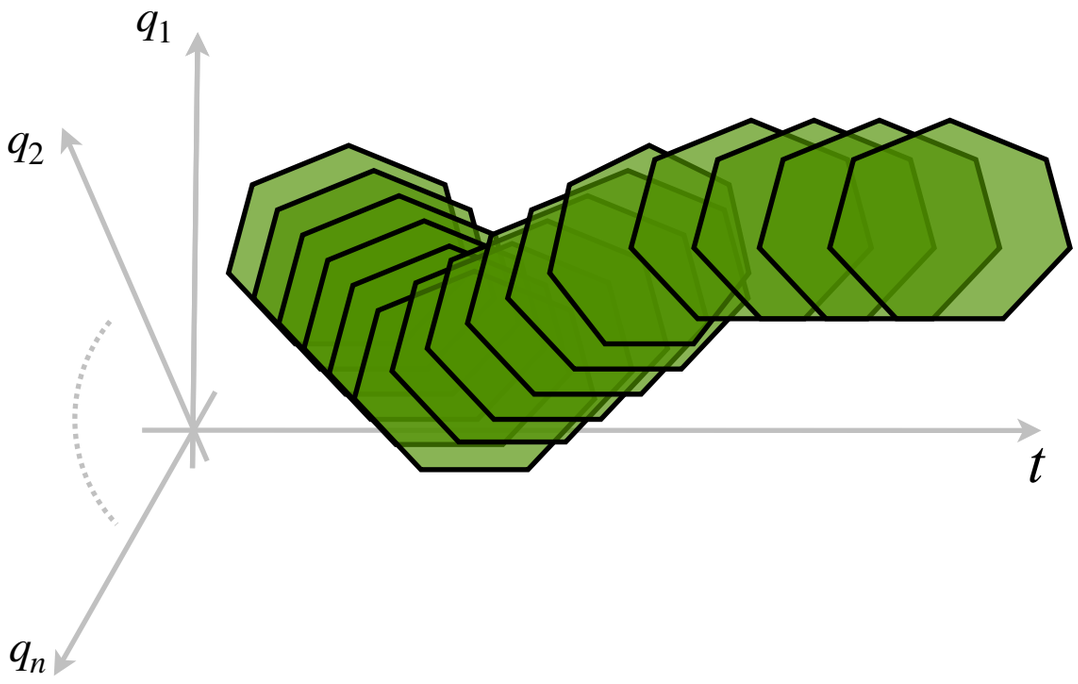
To use the hybrid A* algorithm effectively, we need a good heuristic.

- Nonholomic without obstacles $h_1(q)$. The heuristic calculates the shortest nonholomic path to the goal, ignoring the obstacles.
- Holonmic with obstacles $h_2(q)$. The heuristic calculates the shortest collision-free path to the goal, ignoring the nonholonmic constraints.

Question. Show that both $h_1(q)$ and $h_2(q)$ are admissible.

Question. Show that $h(q) = \max\{h_1(q), h_2(q)\}$ is a better heuristic than either $h_1(q)$ or $h_2(q)$.

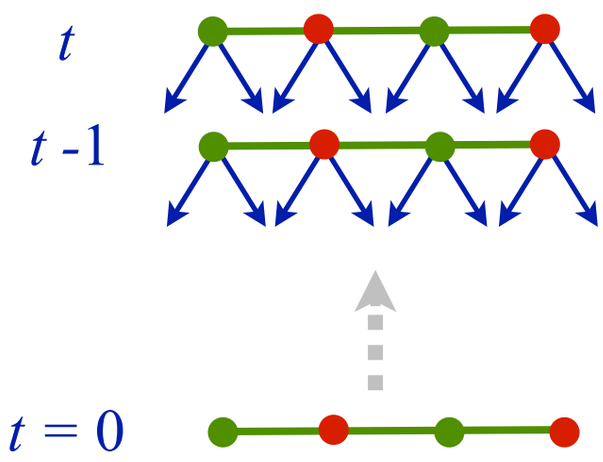
Dynamic environments. So far, we assume that the environment is known and does not change over time. This is somewhat true in a tightly controlled factory. More often, the environment changes. Consider an autonomous vehicle faced with many moving pedestrians. As a first step towards this more realistic setting for motion planning, we assume that future pedestrians are known or predictable. In this case, we just need to add a time axis to the configuration space to form the configuration space-time. The same algorithms that we have studied so far then all apply without change.



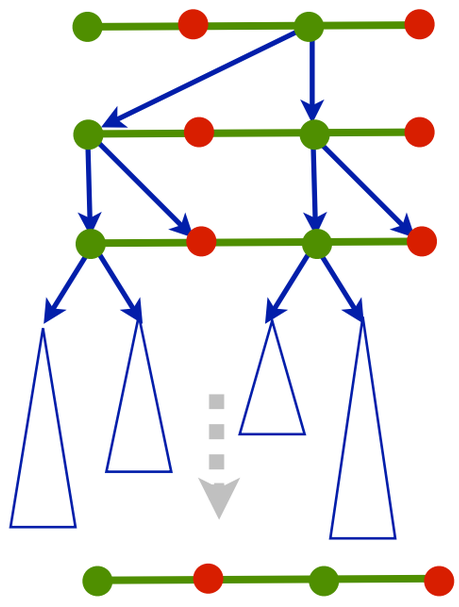
Summary.

- Dynamic programming finds the shortest path for every state. The computation time depends on the size of the state space.
- Forward search finds the shortest path for the current state. The computation time depends on the branch factor and the depth of the search tree.
- The efficiency of forward search depends on the choice of the heuristic function.
- A holonomic constraint involves the configuration only. It restricts the set of valid configurations. A nonlonomic constraint involves both the configuration and the configuration velocity. It restricts the set of valid configuration velocities, but not necessarily the set of configurations.
- Hybrid A* solves the shortest path problem with continuous states.
- We can use hybrid A* to solve for motion planning of a car in a dynamic environment if we know or can predict the future reliably. However, such prediction is not always possible.

dynamic programming



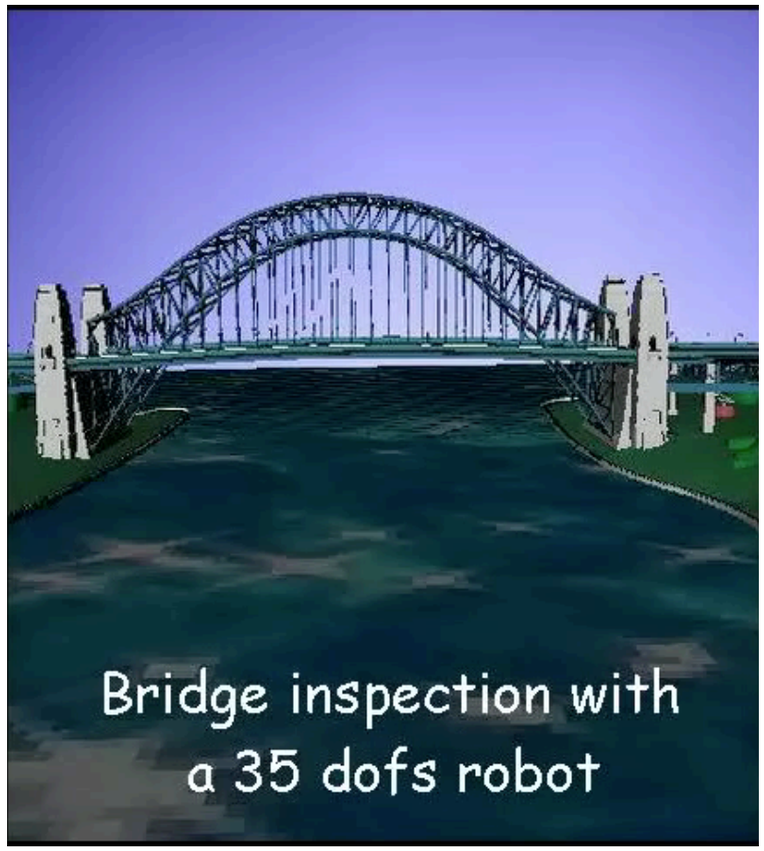
forward search



Key concepts.

- Dynamic programming
- Forward search
- Holonmic and nonholomic constraints
- Hybrid A* algorithm

“Curse of dimensionality”. Hybrid A* cannot handle high-dimensional configuration spaces.



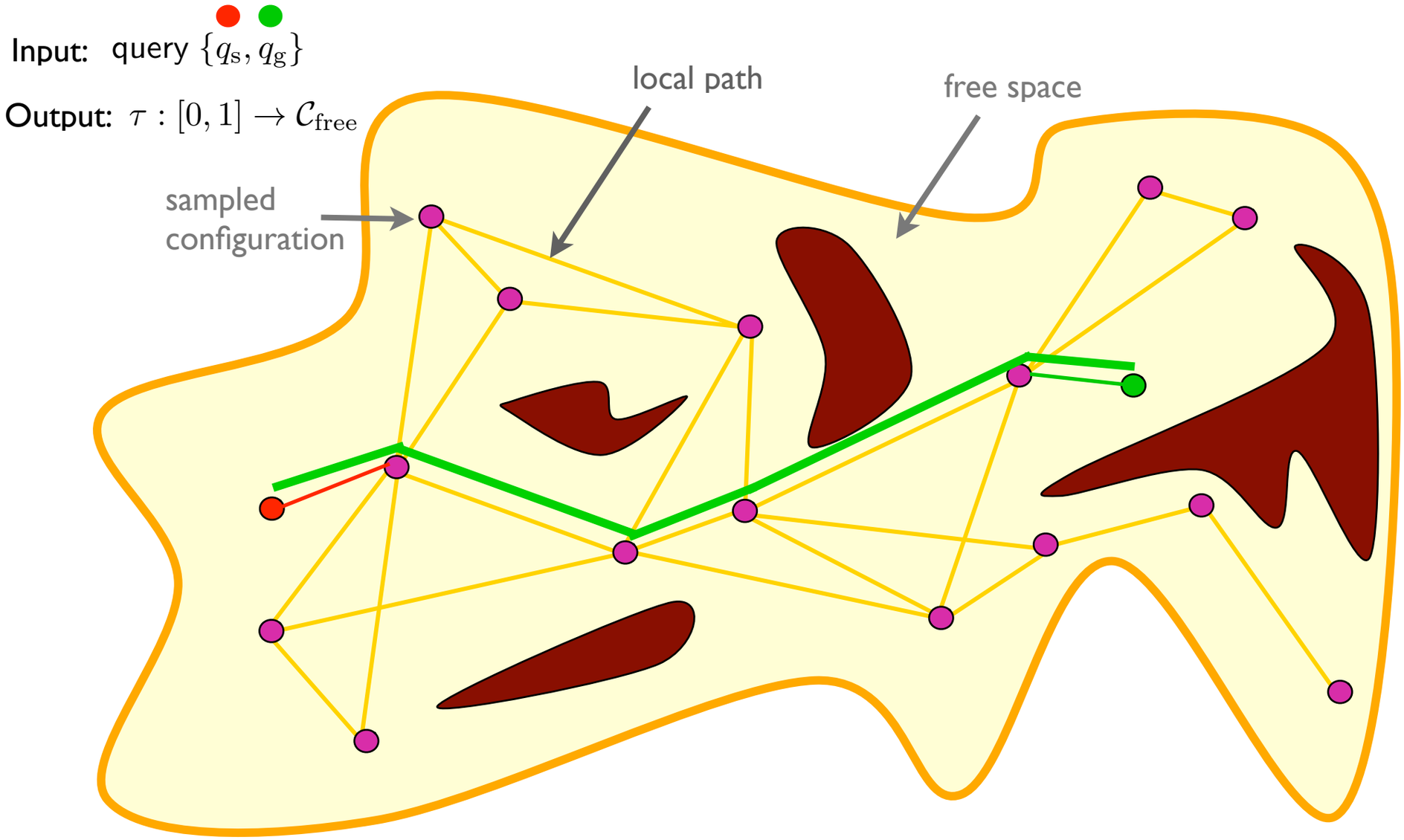
The mobile manipulator has 35 DoFs, i.e., a 35-dimensional configuration space. Placing a grid over this space results in 10^{35} grid cells, if each dimension has 10 discrete value.

The running time increases exponentially with the dimension of the configuration space.

Theoretically, many variants of the path planning problem are PSPACE-hard.

Further, there is geometric complexity, in addition to C-space dimensionality. The rigid body in the example has only 6 DoFs, the required motion to separate the two bodies is intricate and difficult to produce or reproduce.

Probabilistic roadmap (PRM) planning. PRM is a remarkably simple idea that conquers the "curse of dimensionality" through **random sampling**.



PRM planning consists of two phases, offline and online:

- In the offline pre-computation phase, construct a roadmap graph G that captures the connectivity of the underlying configuration space.
- In the online query phase, search G for a path that connects the start and the goal configurations.

A probabilistic roadmap G is a graph.

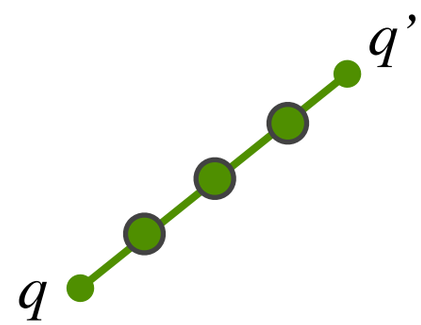
- A node of G represents a randomly sampled collision-free configuration.
- An edge between two nodes of G represents a collision-free "simple" local path (e.g., straight-line path) between the two corresponding configurations.

PRM precomputation.

```
Input:
     N: the number of roadmap nodes
     geometry of a robot and obstacles
Output:
   roadmap G = (V, E)

BasicPRM
1: V ← ∅ and E ← ∅.
2:   while |V| < N
3:     q ← a configuration sampled uniformly at random from C.
4:      if CLEAR(q) = TRUE then
5:        Add q to V.
6:        N_q ← a set of nodes in V such that each is close to q
                according to distance d(q,q')
7:        for each q'∈ N_q
8:          if LINK(q',q) = TRUE then
9:            Add an edge between q and q' to E.
13:return G = (V,E)
```

The PRM algorithm relies on two primitives: **`CLEAR`** and **`LINK`**. **`CLEAR`** checks whether a configuration q is collision-free. **`LINK`** checks whether two configurations q and q' can be connected via a simple path, in particular, a straight-line path. By discretizing the straight-line path, **`LINK`** can be implemented as a sequence of calls to **`CLEAR`**. So, **`LINK`** is computationally much more expensive than **`CLEAR`**.



PRM query.

- Add the start configuration s as a new node in the roadmap G and connect s to other nodes that are close.
- Do the same for the goal configuration g.
- Search in G for a path between s and g. Return the path if one is found. Otherwise, report NONE.

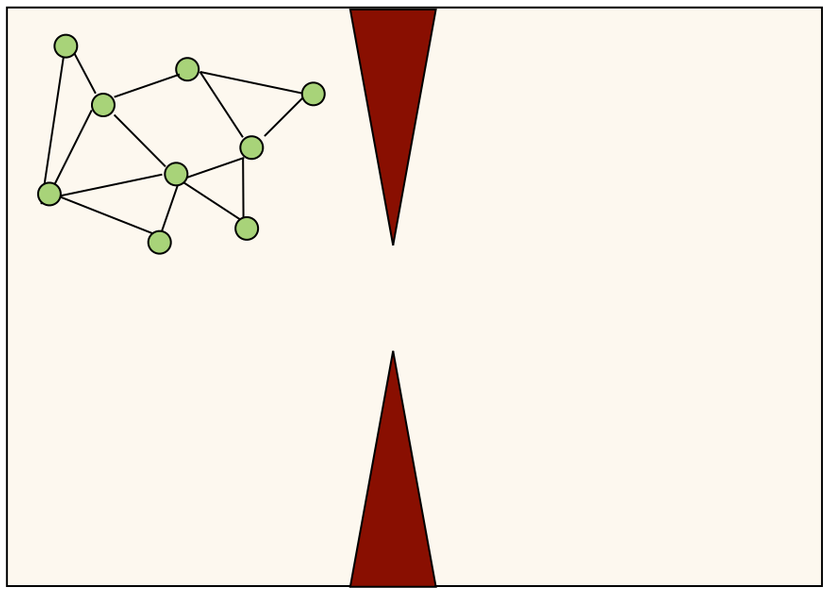
PRM is a major breakthrough in motion planning. It is deceptively simple. Does it work? More importantly, why?

First, we must define what it means to "work".

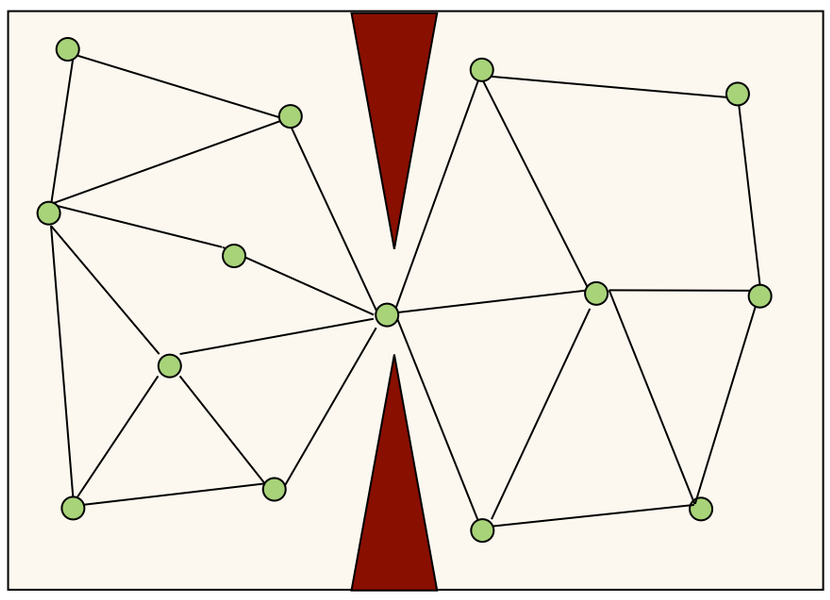
Completeness. The algorithm finds the path if it exists, and reports no otherwise.

Consider the two roadmaps below. The one on the left fails to **cover** the space well.

bad coverage

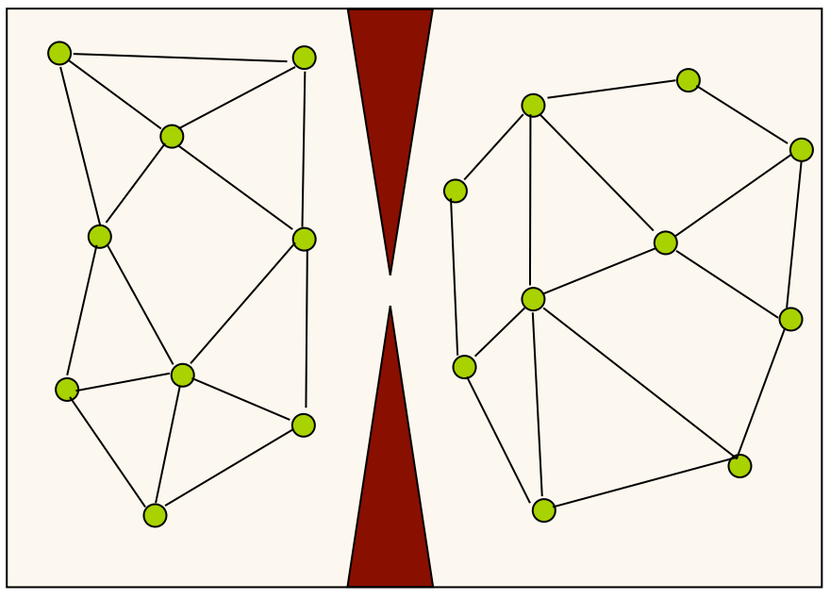


good coverage

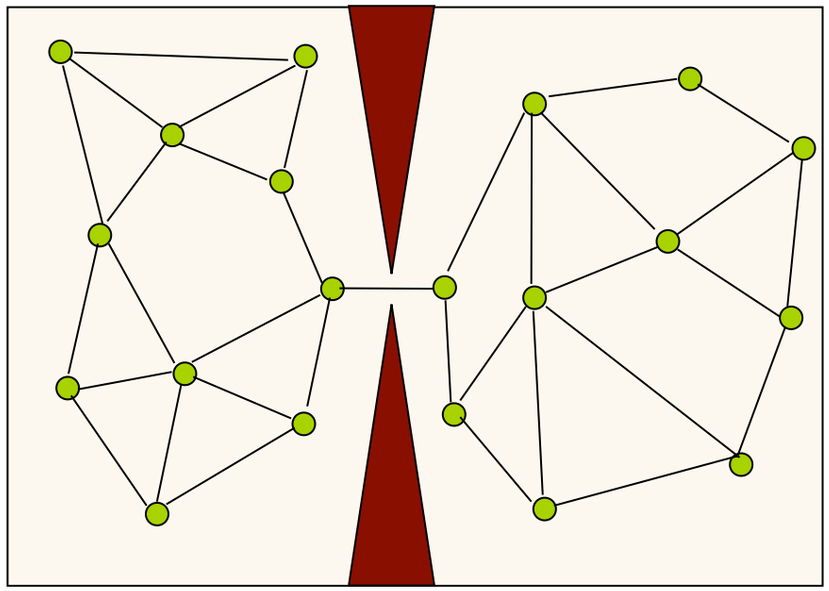


Both roadmaps below cover the space. The one on the left fails to **connect** the space properly.

bad connectivity



good connectivity



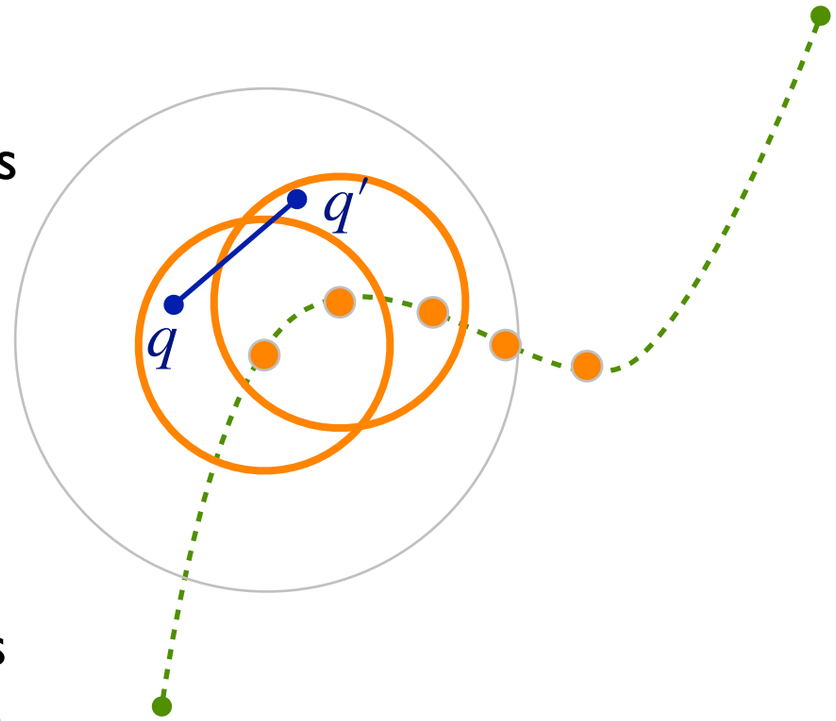
BasicPRM is not complete, for either bad coverage or bad connectivity.

Probabilistic completeness. A PRM algorithm P is probabilistically complete if P satisfies the two conditions below:

- If no solution path exists, P reports NONE.
- If a solution path exists, P finds it with high probability after a finite amount of computation time.

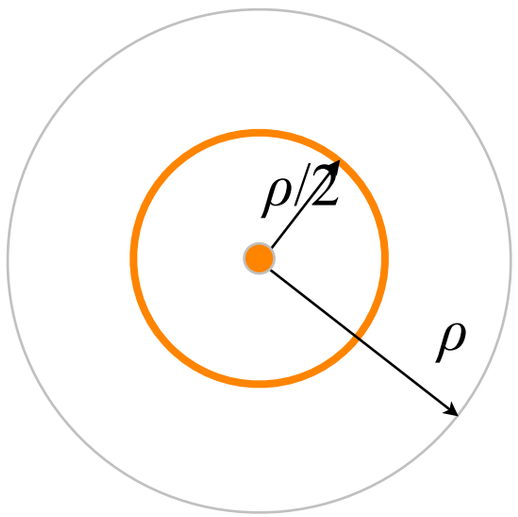
The PRM algorithm is **probabilistically complete**.

Intuitively, the coverage and connectivity of a roadmap improves with the increasing number of samples.



Proof.

- Assume a path of clearance ρ and length L.
- Place balls of radius $\rho/2$ along the path so that the centers of any two adjacent balls are within distance $\rho/2$ along the path.
- Let q be a point in a ball. Let q' be any point in an adjacent ball. The straight-line path between q and q' must be collision-free.
- After sampling a point from every ball, we have a collision-free path.



- Sample n points uniformly at random from the configuration space.

- The failure probability $p(F_i)$ that a given ball i does not get a sample is

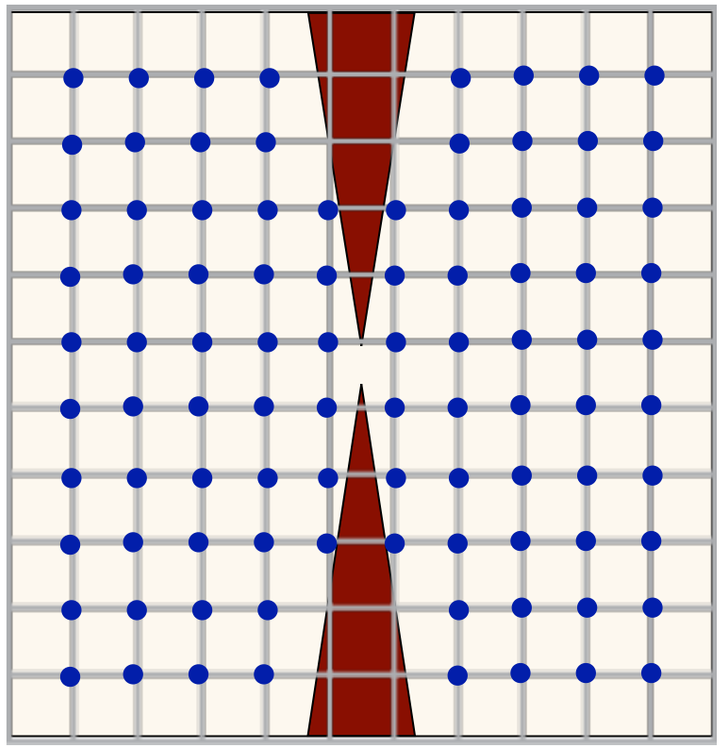
$$p(F_i) = \big(1 - \mu(B_{\rho/2})\big)^n$$

where $\mu(B_{\rho/2})$ is the volume of a ball with radius $\rho/2$.

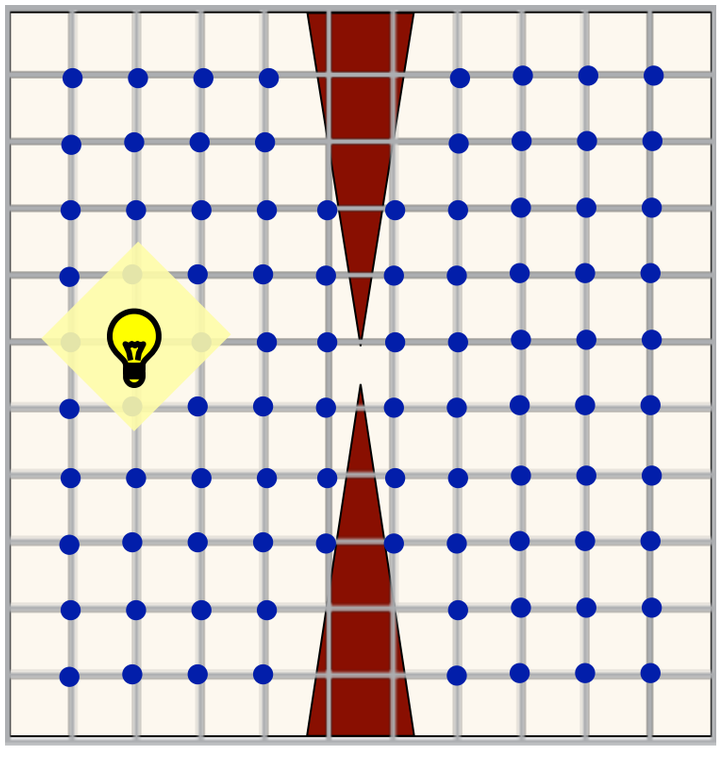
- There are roughly $2L/\rho$ balls. Apply the union bound and obtain the overall failure probability that some ball does not get a sample:

$$\begin{aligned} p(F) &= p(\bigcup_i F_i) \\ &\le \sum_i p(F_i) && \text{Apply the union bound.} \\ &= \frac{2L}{\rho}\big(1 - \mu(B_{\rho/2})\big)^n \\ &\le \frac{2L}{\rho}\mathrm{e}^{-n\mu(B_{\rho/2})} && 1 - x \le \mathrm{e}^{-x} \end{aligned}$$

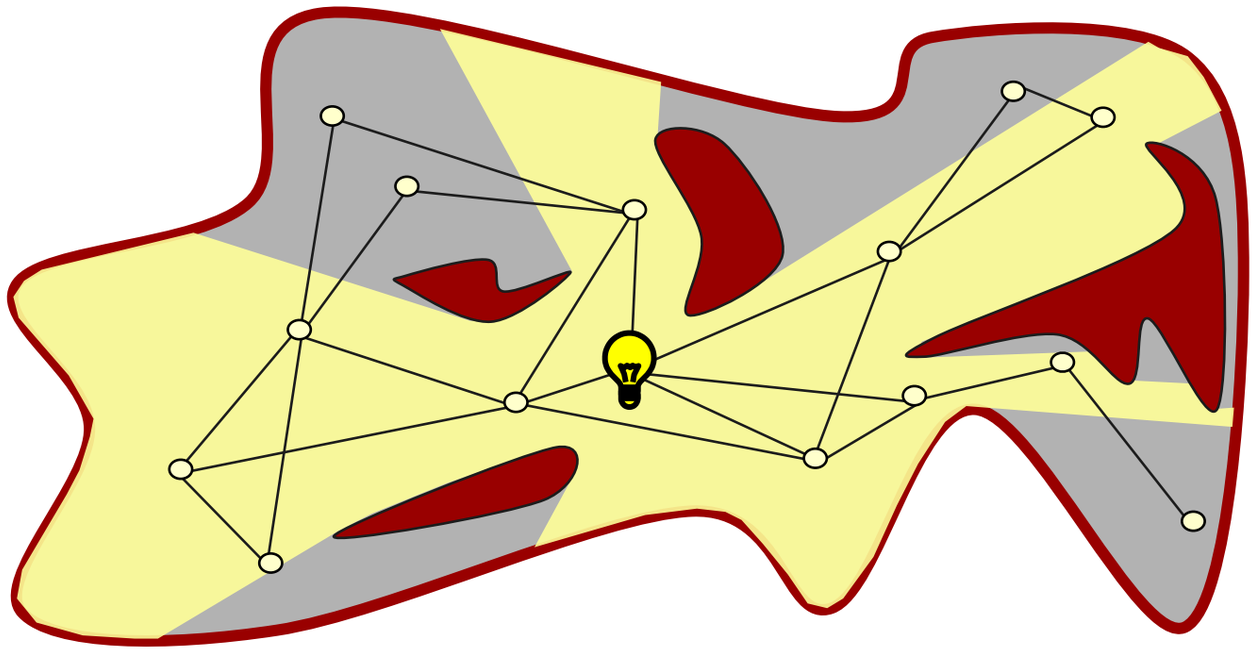
Comparison. Compare the PRM roadmap with the cell decomposition graph. The cell decomposition graph "samples" configurations on a regular grid. How do they differ?



In the cell decomposition graph, each node is connected to nodes of immediately neighboring cells. In contrast, in a PRM roadmap, a node may be connected to a node far away. The "visibility" region of a node in the roadmap is much bigger than that of a node in the cell decomposition graph. As a result, the roadmap covers the space with much fewer samples.



A dense graph



A very sparse graph

Importance (re-)sampling. Suppose that we have a roadmap consisting of configurations sampled uniformly at random. We can improve the connectivity of the roadmap by sampling in "difficult" regions with poor connectivity.

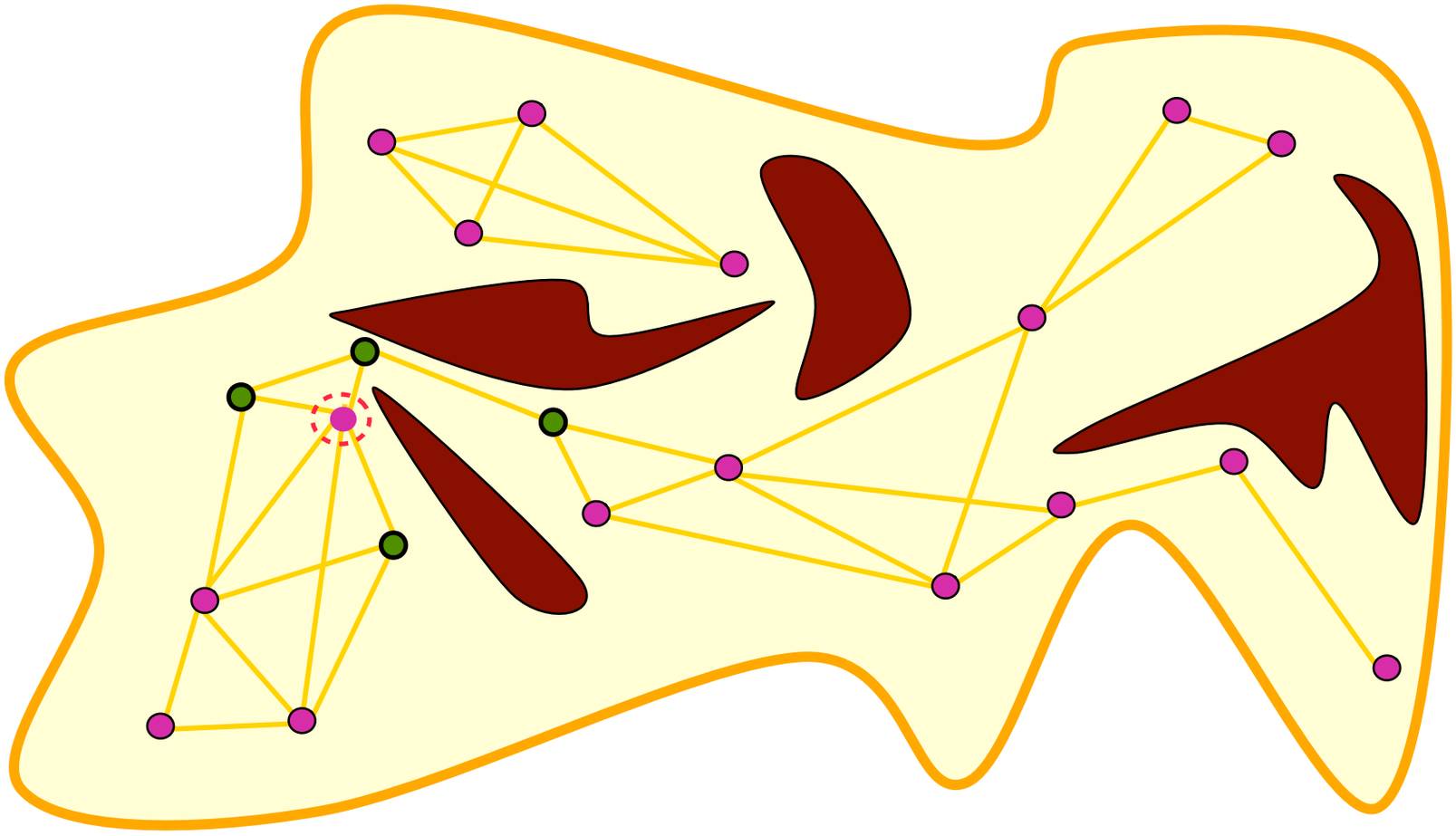
- For each node q in a roadmap, define the importance weight

$$\pi(q) = \frac{1/(\deg(q)+1)}{\sum_{p \in V} 1/(\deg(p)+1)}$$

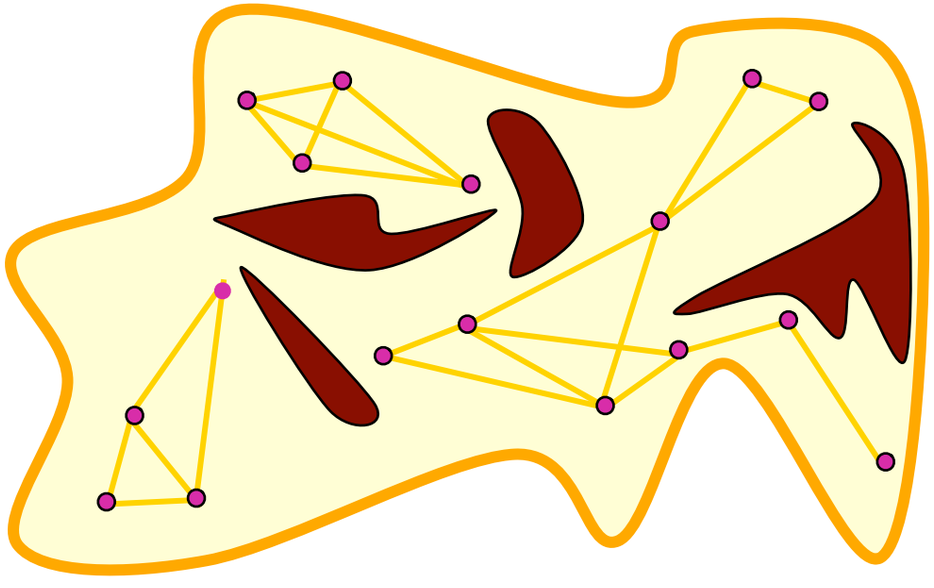
  where $\deg(q)$ is the degree of the node q in the roadmap.

If a node has low degree, it is not well connected with its neighbors. The connections are blocked by obstacles. It is a reasonable heuristic to add more samples in the "difficult" regions.

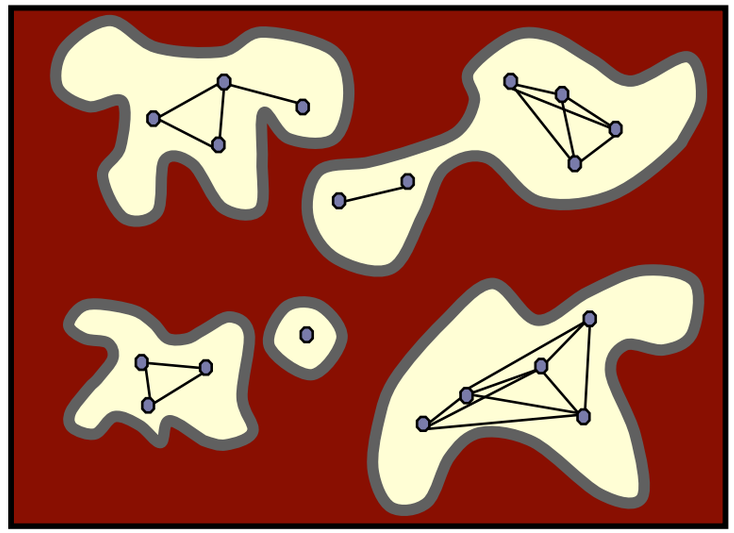
- Choose a node q from V with probability $\pi(q)$.
- Sample new nodes in the neighborhood of q.
- Connect the new nodes to existing nodes in the roadmap.



Question. According to the importance sampling criteria, which node is most likely to be chosen? Is it a good choice?

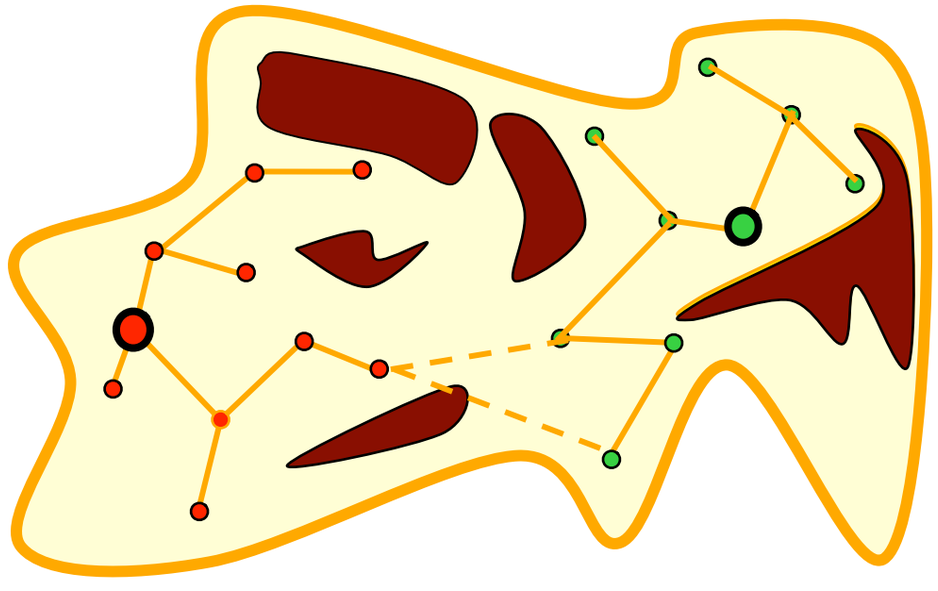


Tree search with random sampling. If we want to answer a single path planning query from the current configuration to a goal. It seems "overkill" to build an entire roadmap. In the configuration space, there are at most two connected components relevant to the query, one containing the start configuration and one containing the goal configuration. The motivation here is similar to that of forward search.

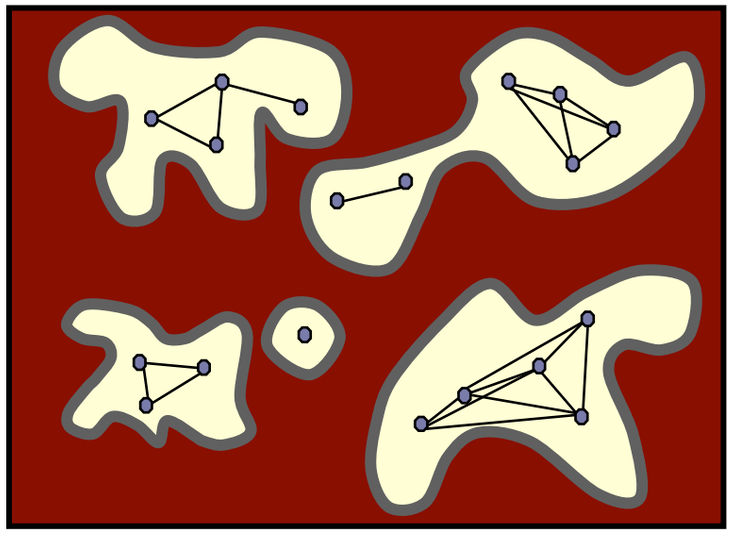


Here we must combine tree search with random sampling.

- Incrementally "grow" two trees T_s and T_g, rooted at s and g, respectively.
- **Expand** a tree by choosing a node from a tree and sample in its neighborhood.
- Connect the two trees.

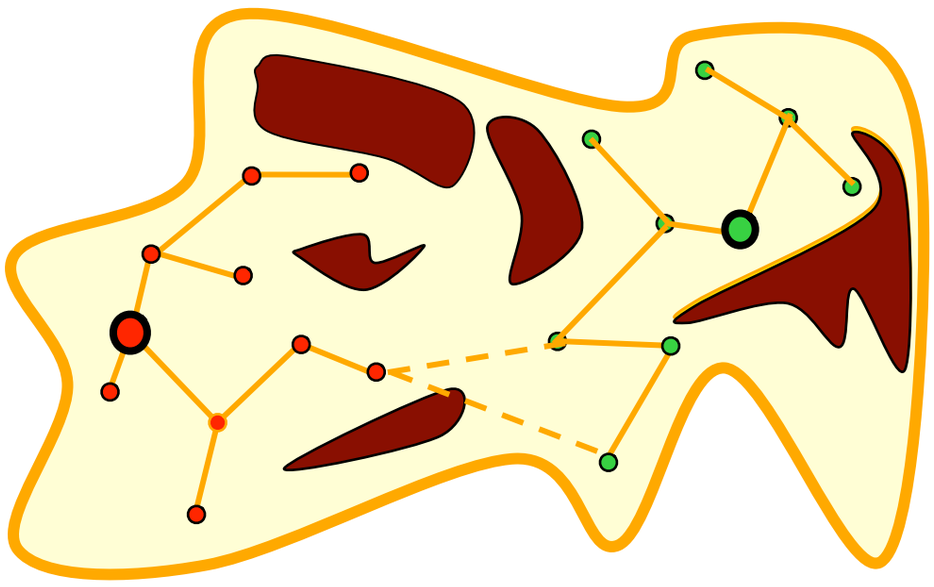


Interaleave sampling and search



must combine tree search with random sampling.

- Incrementally "grow" two trees T_s and T_g, rooted at s and g, respectively.
- **Expand** a tree by choosing a node from a tree and sample in its neighborhood.
- Connect the two trees.



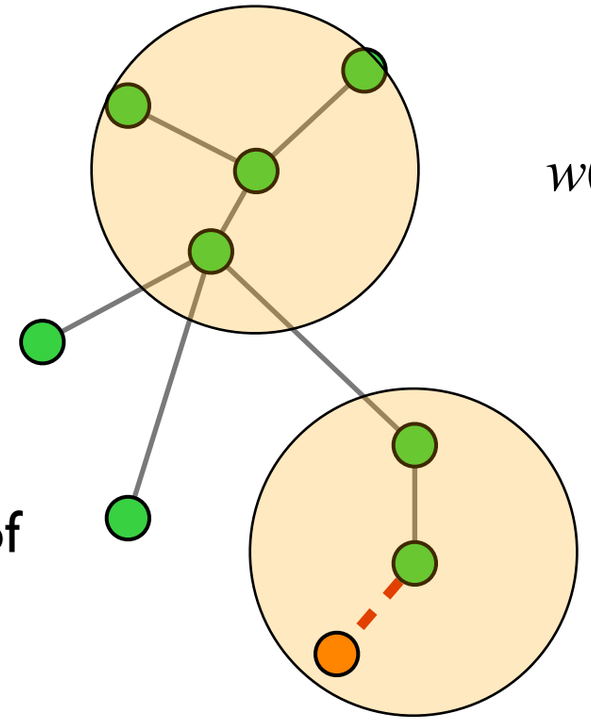
There are different ways of expanding tree nodes.

Expansive-space tree (EST). The main idea is to expand tree nodes in low-density regions.

- For a node q, define the weight $w(q)$ to be the number of nodes in a sphere with the center q and some chosen radius ρ.
- Choose a node q from T with probability

$$\pi(q) = \frac{1/(w(q))}{\sum_{p \in V} 1/w(p)}.$$

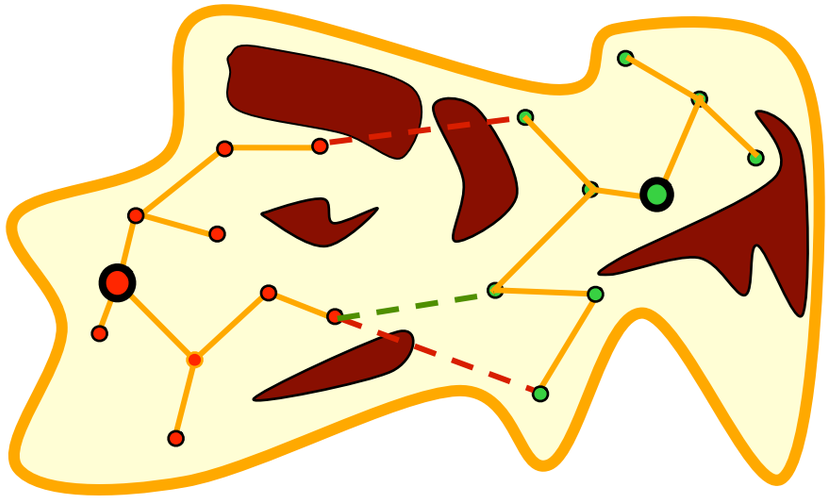
- Sample a new configuration q' from the neighborhood of q. Add q' to T if q' is collision-free and there is a collision-free straight-line path between q and q'.



```
Input:
  q0: the configuration for the root of the tree
  N : the number of tree nodes
  geometry of a robot and obstacles
Output:
  tree T = (V, E)

EST-Expand
1: V ← {q0} and E ← ∅.
2: while |V| <= N
3:    Choose a node q from V with probability π(q).
4:    Sample a configuration q' from the neighborhood of q.
5:       if CLEAR(q') = TRUE and LINK(q',q) = TRUE then
6:          Add q' to V.
7:          Add an edge between q and q' to E.
8:          N_q' ← a set of nodes in V whose distance to q' is
9:                 no greater than δ.
10:         w(q') ← |N_q'|+1
11:         for each p ∈ N_q'
12:           w(p) ← w(p)+1.
13:return T = (V,E)
```

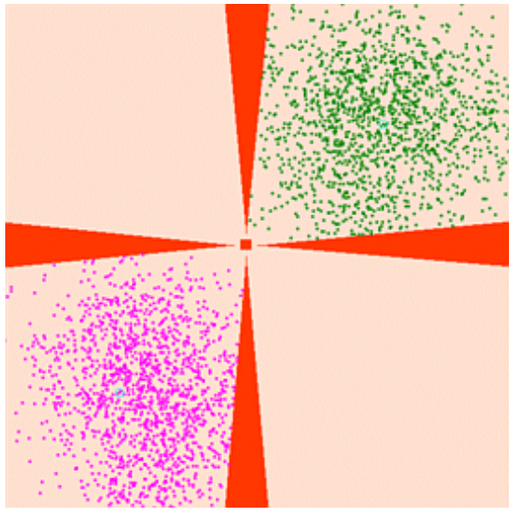
Expand two EST trees, rooted at the start and the goal configurations, and connect the two trees. For each node q in T and q' in T', if the distance $d(q, q')$ is small, call **`LINK`**(q, q') to check whether there is a collision-free straight-line path between q and q'.



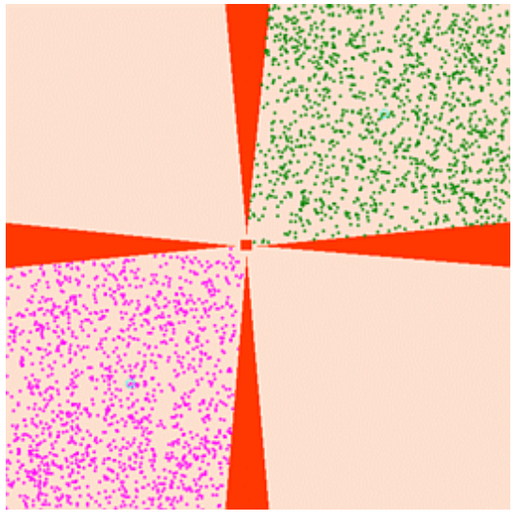
Question.

- What is the difference between the two sets of samples below? Which one is obtained with the weighted sampling?

unweighted EST



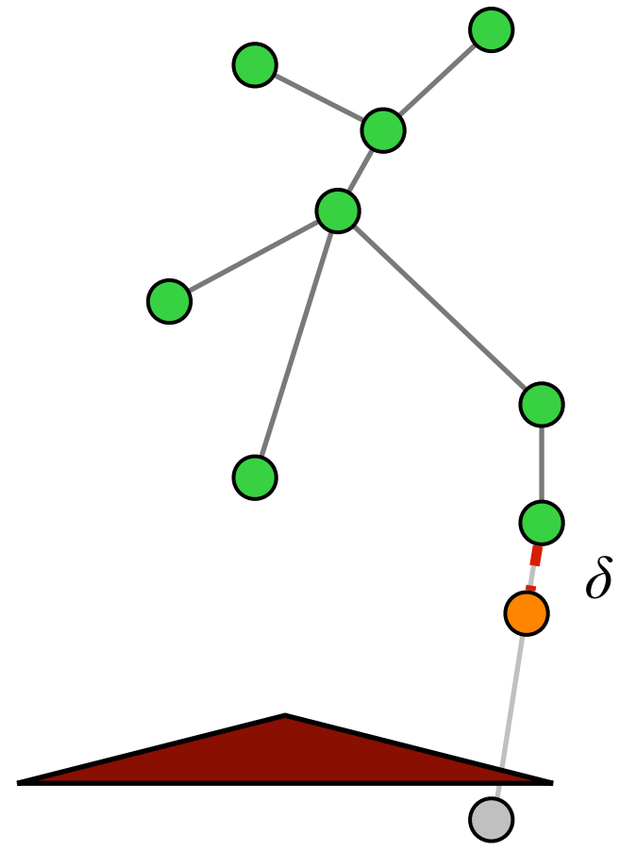
EST



- Why do we use weighted sampling when choosing the nodes from V ?

Rapidly-exploring random tree. Approach samples from a target sampling distribution, e.g., the uniform distribution.

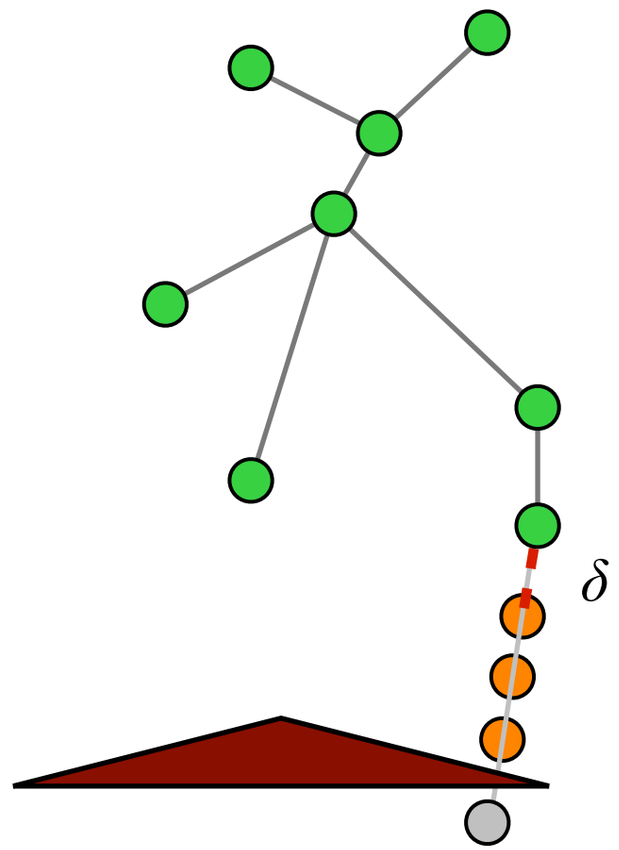
- Sample a collision-free configuration p uniformly at random.
- Find a node q from T, which is closest to p.
- Set q' to be the configuration at a fixed distance δ from q along the straight line from q to p. Add q' to T if q' is collision-free and there is a collision-free straight-line path between q and q'.



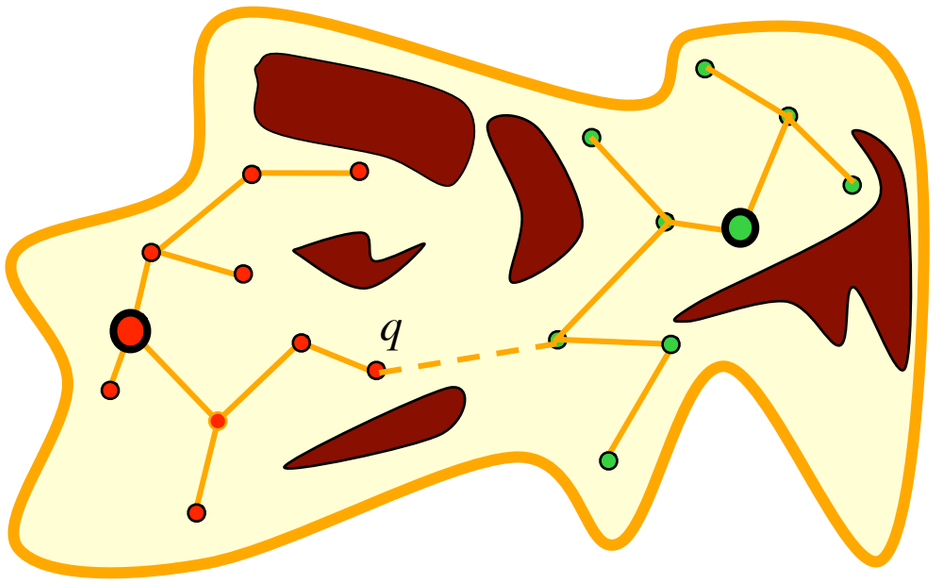
We can also approach the target sample in a more aggressive manner by the replacing the last step above with

- Set q' to be the configuration a fixed distance δ from q along the straight line from q to p. Add q' to T if q' is collision-free and there is a collision-free straight-line path between q and q'.

Set $q = q'$. Repeat until blocked by obstacles.



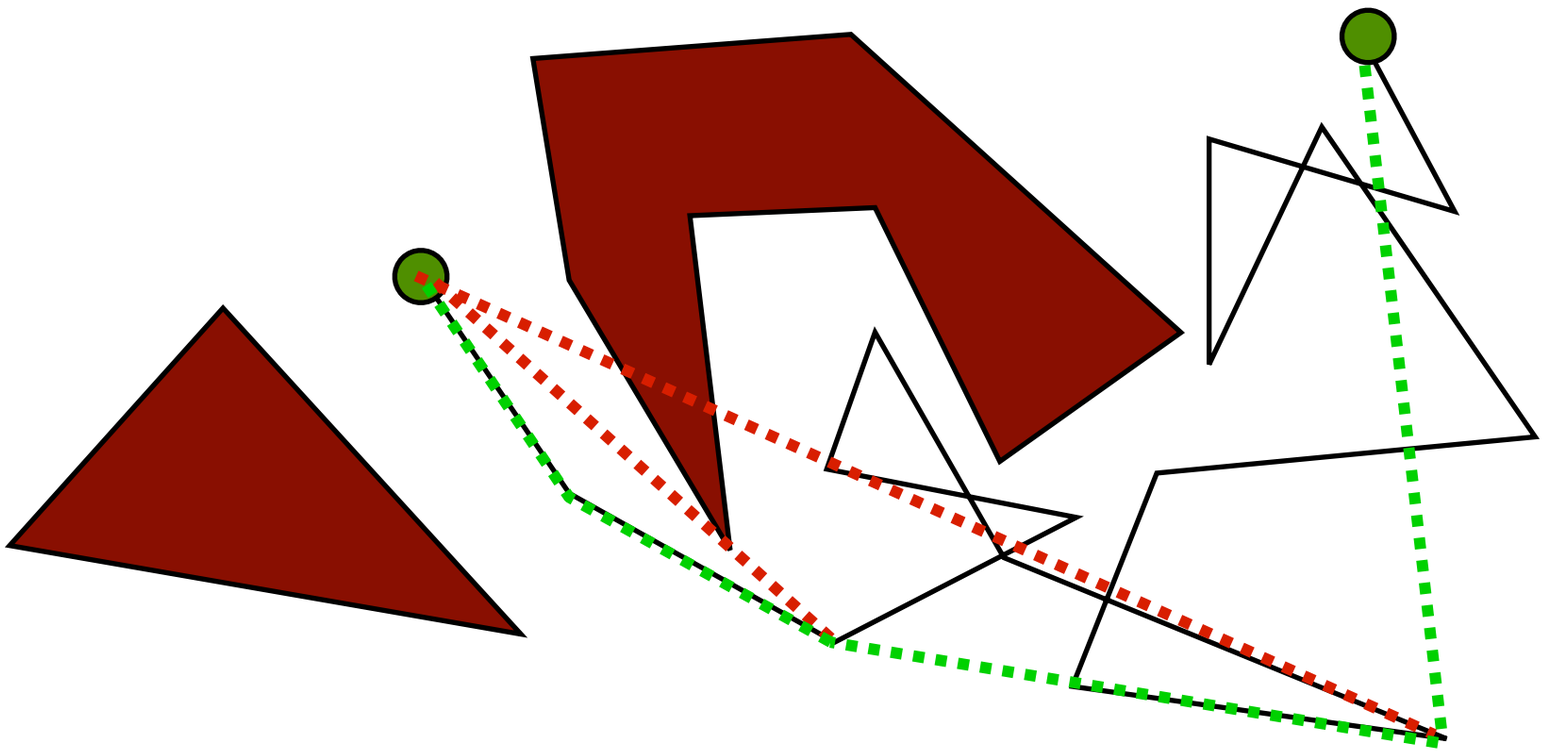
Expand two RRT trees T and T', rooted at the start and the goal configurations. For each new node q of a tree T, (greedily) extend T' towards q instead of a randomly sampled configuration p. Terminate if the two trees are connected.



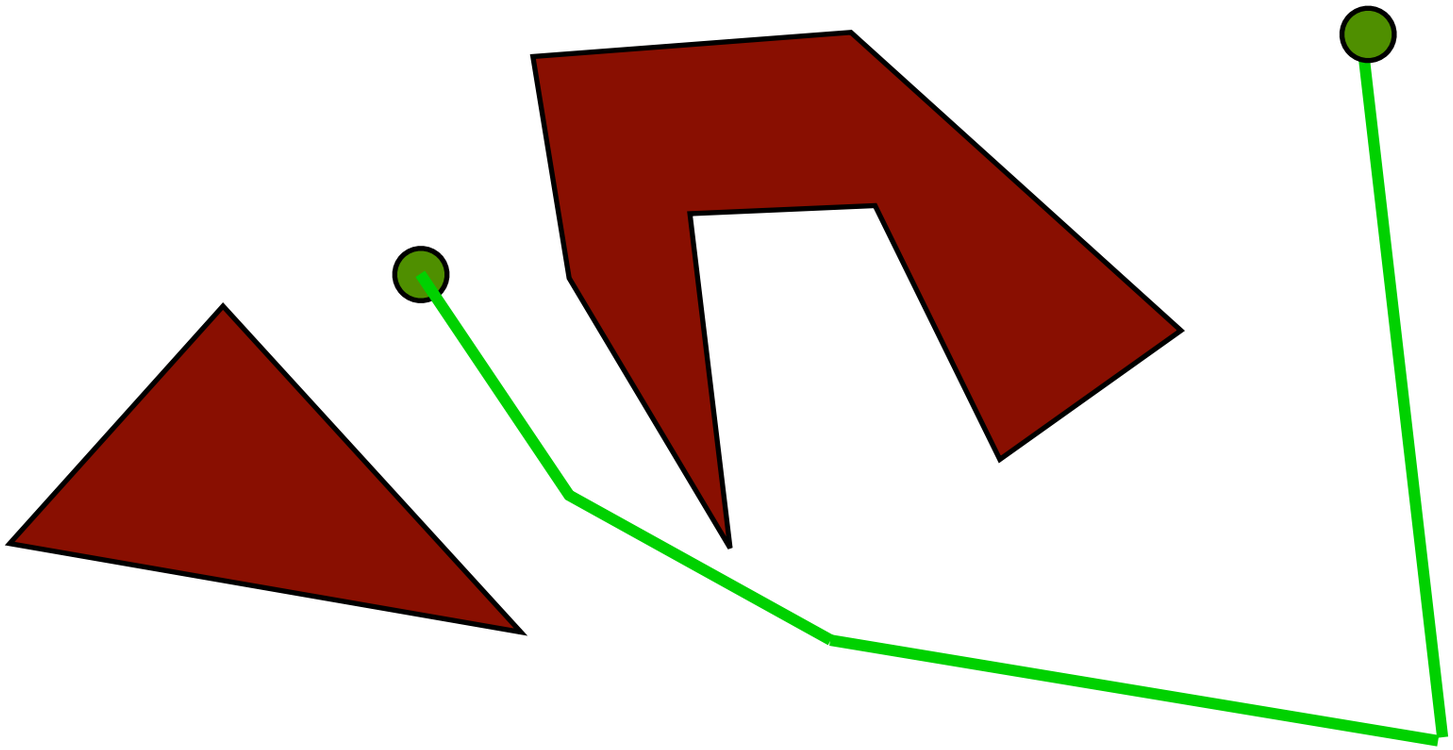
Comparison of EST and RRT.

- Both EST and RRT can be shown to be probabilistically complete.
- They are in fact very similar, but differ mainly in how new configurations are sampled. RRT places strong confidence in the sampling heuristic that there are straight-line paths between nodes of the two trees and is more **greedy** in trying to sample nodes that connect the two trees togethers. This is often good, but gets in trouble when the heuristic fails. As a rough analogy, EST behaves more like breadth-first search, and RRT behaves more like depth-first search.
- EST and RRT find valid collision-free path, but **not** the shortest path.

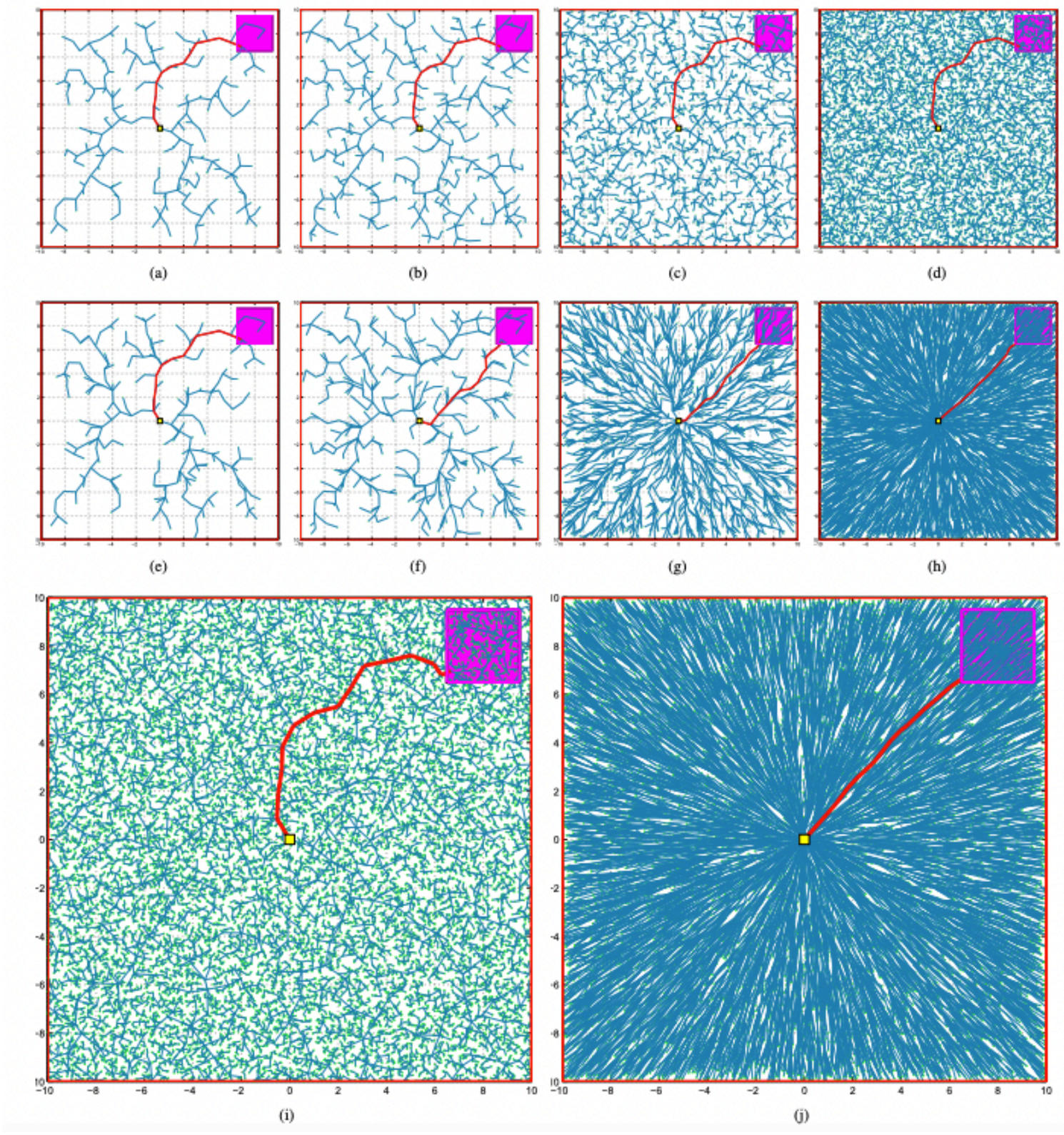
Path smoothing. Random sampling often results in a jagged path, which is undesirable in practice. To smooth the path, perform "shortcutting" recursively.



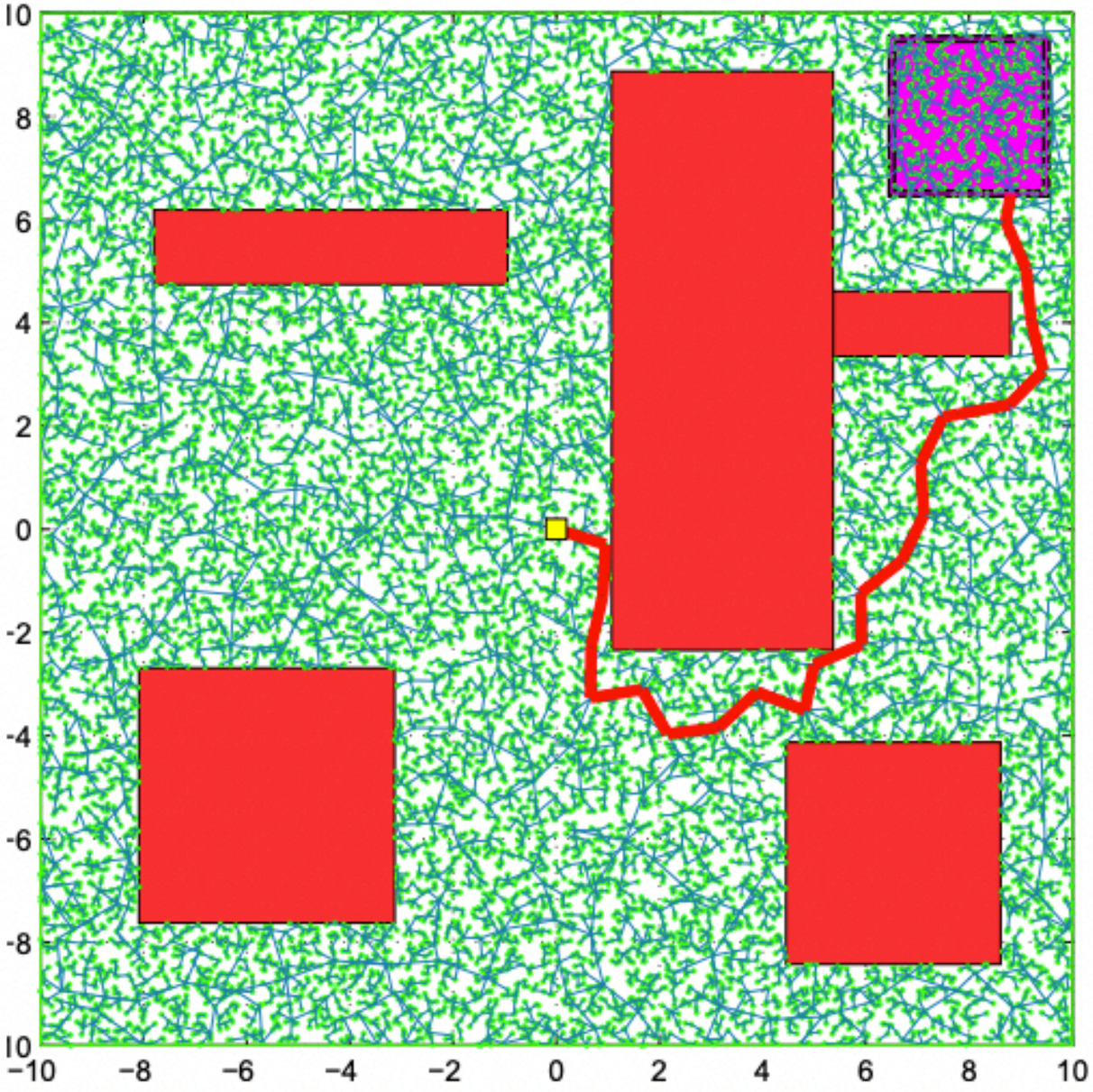
Path smoothing. Random sampling often results in a jagged path, which is undesirable in practice. To smooth the path, perform "shortcutting" recursively.



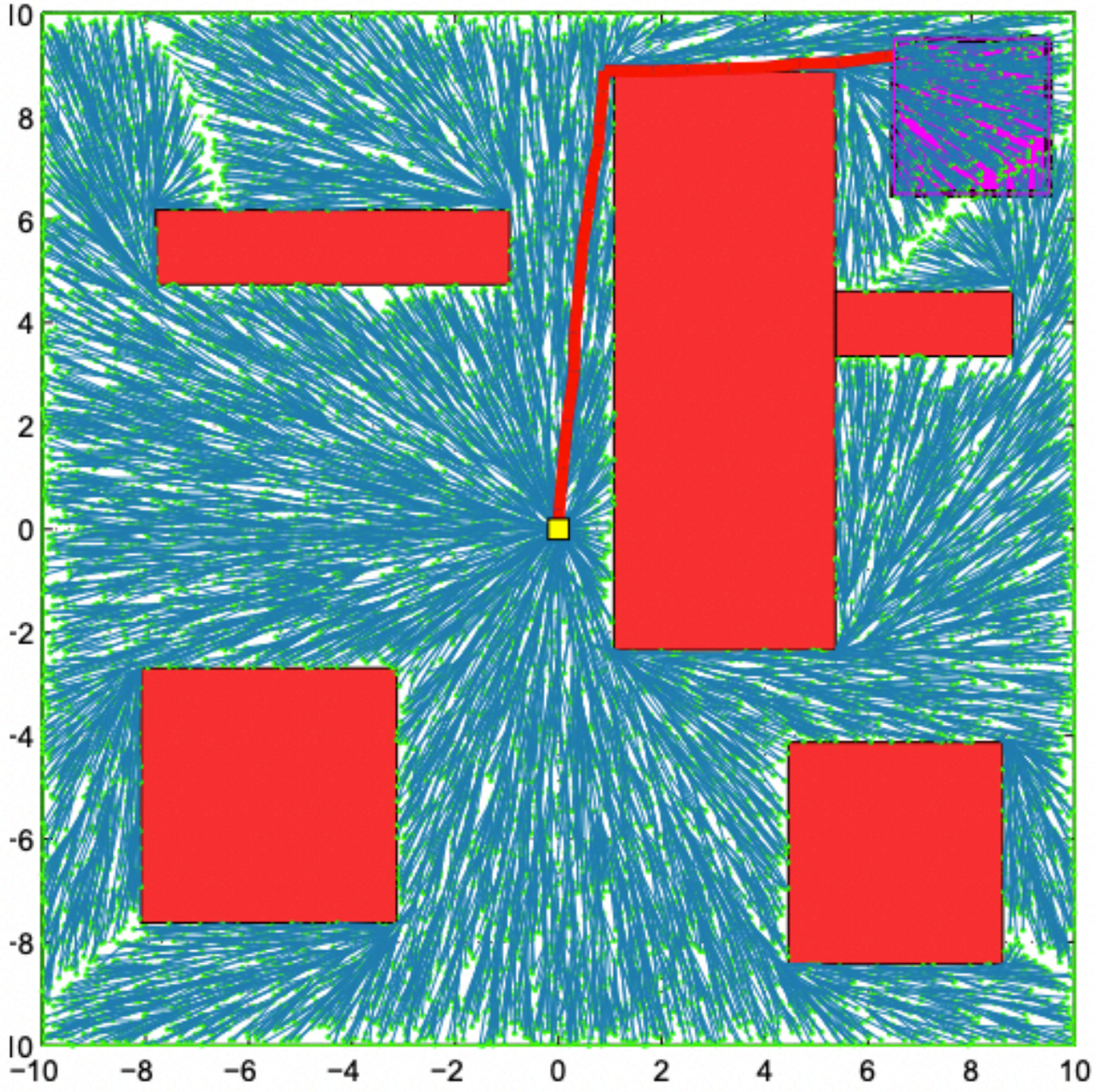
RRT vs RRT*



RRT vs RRT*


10
8
6
4
2
0
-2
-4
-6
-8
10
−10 −8 −6 −4 −2 0 2 4 6 8 10

(a)


10
8
6
4
2
0
-2
-4
-6
-8
10
−10 −8 −6 −4 −2 0 2 4 6 8 10

(b)

RRT*

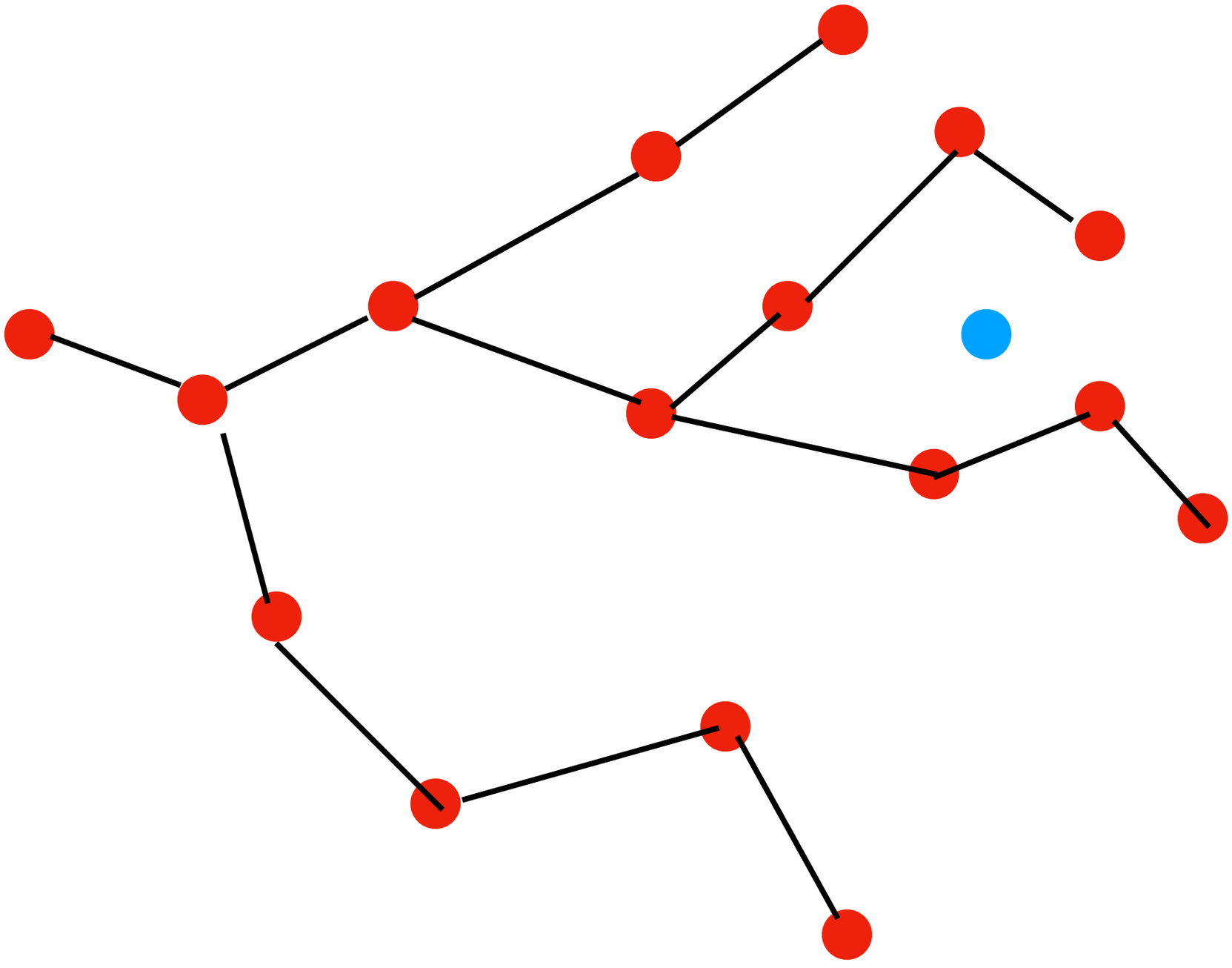
parent(node)

cost(node)

cost(v)=cost(parent(v))+ c(Line(parent(v),v)

RRT*

$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
for $i = 1, \ldots, n$ **do**
  $x_{\text{rand}} \leftarrow \texttt{SampleFree}_i$;
  $x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$;
  $x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
  if $\texttt{ObtacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $X_{\text{near}} \leftarrow \texttt{Near}(G = (V, E), x_{\text{new}}, \min\{\gamma_{\text{RRT}^*}(\log(\text{card}\,(V))/\,\text{card}\,(V))^{1/d}, \eta\})$;
    $V \leftarrow V \cup \{x_{\text{new}}\}$;
    $x_{\min} \leftarrow x_{\text{nearest}}$; $c_{\min} \leftarrow \texttt{Cost}(x_{\text{nearest}}) + c(\texttt{Line}(x_{\text{nearest}}, x_{\text{new}}))$;
    foreach $x_{\text{near}} \in X_{\text{near}}$ **do** // Connect along a minimum-cost path
      if $\texttt{CollisionFree}(x_{\text{near}}, x_{\text{new}}) \wedge \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}})) < c_{\min}$ **then**
        $x_{\min} \leftarrow x_{\text{near}}$; $c_{\min} \leftarrow \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}}))$
    $E \leftarrow E \cup \{(x_{\min}, x_{\text{new}})\}$;
    foreach $x_{\text{near}} \in X_{\text{near}}$ **do** // Rewire the tree
      if $\texttt{CollisionFree}(x_{\text{new}}, x_{\text{near}}) \wedge \texttt{Cost}(x_{\text{new}}) + c(\texttt{Line}(x_{\text{new}}, x_{\text{near}})) < \texttt{Cost}(x_{\text{near}})$
      then $x_{\text{parent}} \leftarrow \texttt{Parent}(x_{\text{near}})$;
      $E \leftarrow (E \setminus \{(x_{\text{parent}}, x_{\text{near}})\}) \cup \{(x_{\text{new}}, x_{\text{near}})\}$
return $G = (V, E)$;



Learning-based RRT

Learning What?

Sampling Function?

Collision Checking?

Online and offline planning. There are two common planning and execution architectures, offline and online.

Offline planning consists of two phases:

- Planning. Plan a path for every possible start and goal configuration pairs.
- Plan execution. Execute a pre-plannned path, given the current configuration and the goal configuration.

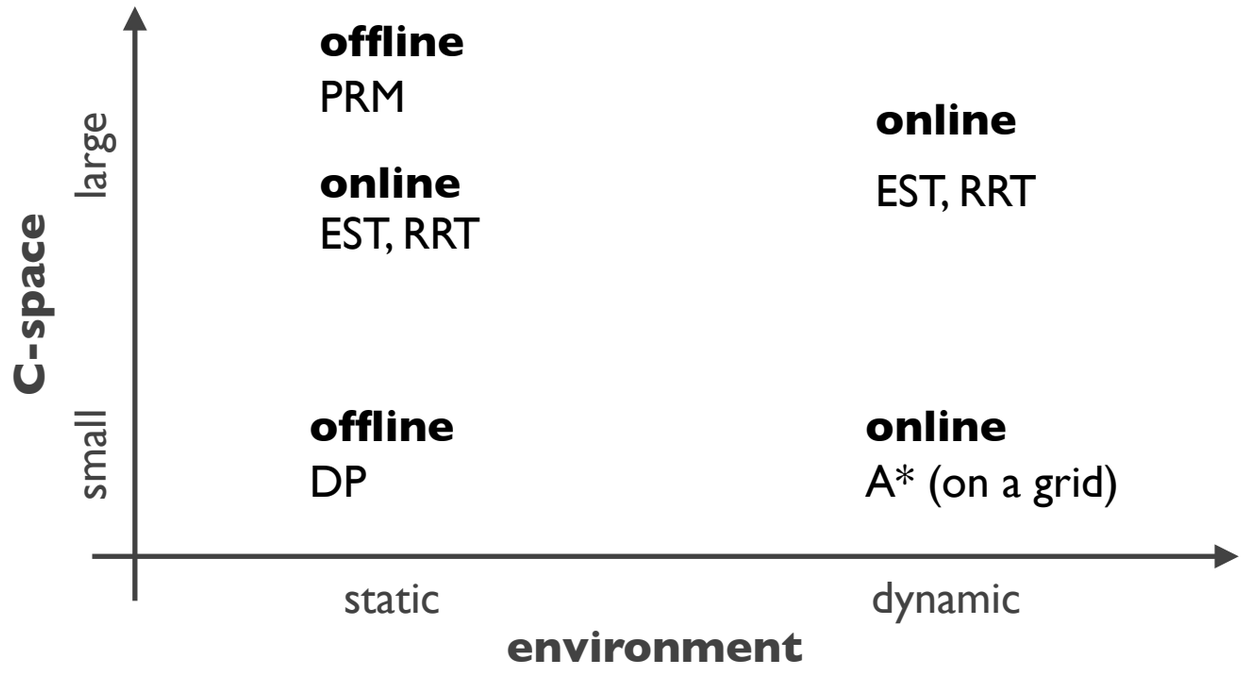
If we have sufficient information about the execution environment and sufficient computational resources, then we can pre-compute all possible solutions in advance. The execution phase is then very simple and efficient. The robot looks up the pre-computed solution based on the current configuration and executes it.

Alternatively, online planning repeats the following steps:

- Plan a path for the current configuration.
- Execute one step along the path.
- Update the current configuration and repeat

Online planning is common in dynamic environments, where pre-computed solutions quickly become invalid because of environment changes.

Summary.



- Dynamic environments requires online planning. Extensive pre-computation is wasteful, as pre-computed solutions or partial solutions quickly become invalid because of environment changes. Forward search algorithms such as A*, EST, RRT, … are suitable candidates.
- In static environments, we can pre-compute all solutions, using, e.g., dynamic programming, if the C-space size is moderate and sufficient computational resources are available.

Key concepts.

- PRM
- EST
- RRT
- Completeness, probabilistic completeness
- Offline and online planning

Tools.

- MoveIt
- OMPL