Planning

Robot: *goal oriented machine that can sense, plan and act*

To plan or not to plan, that is the question ....

- ***Planning:*** Sequence of actions to achieve a given goal (usually without timing specifications), or towards achieving a given goal by achieving a set of sub-goals
- ***Scheduling:*** Planning + Timing (usually without space considerations)
- ***Task planning:*** Sequence of actions that accomplish a large goal (e.g., building a house)
- ***Motion Planning:*** Generation of motions through space ...

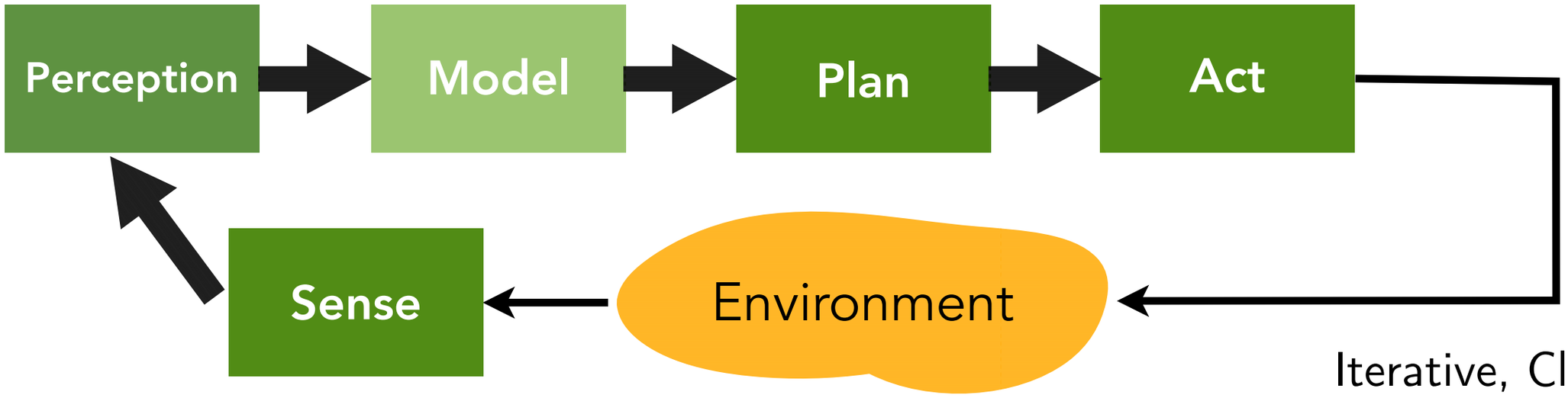
Planning is a **top-down** approach to problem solving that requires a (reliable) ***model*** of the world / problem, to be able to effectively *reason* about it!

Deliberative robot control architectures

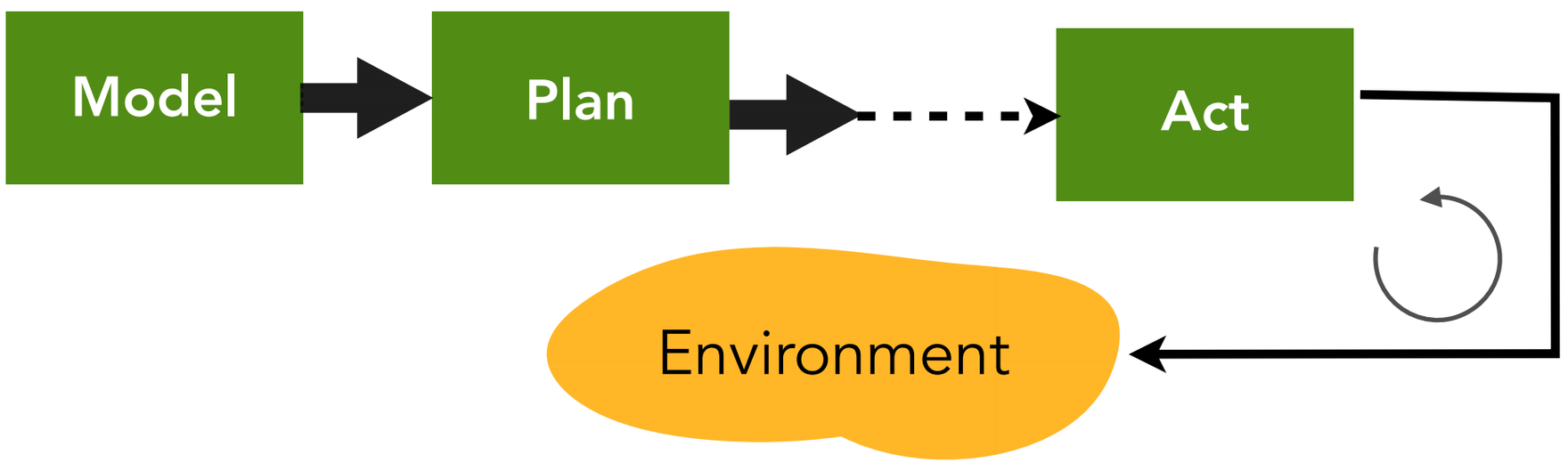
Deliberation: Thoughtfulness in decision and action

→ *Thinking hard: model, reason and plan*

Top-down approach to problem solving

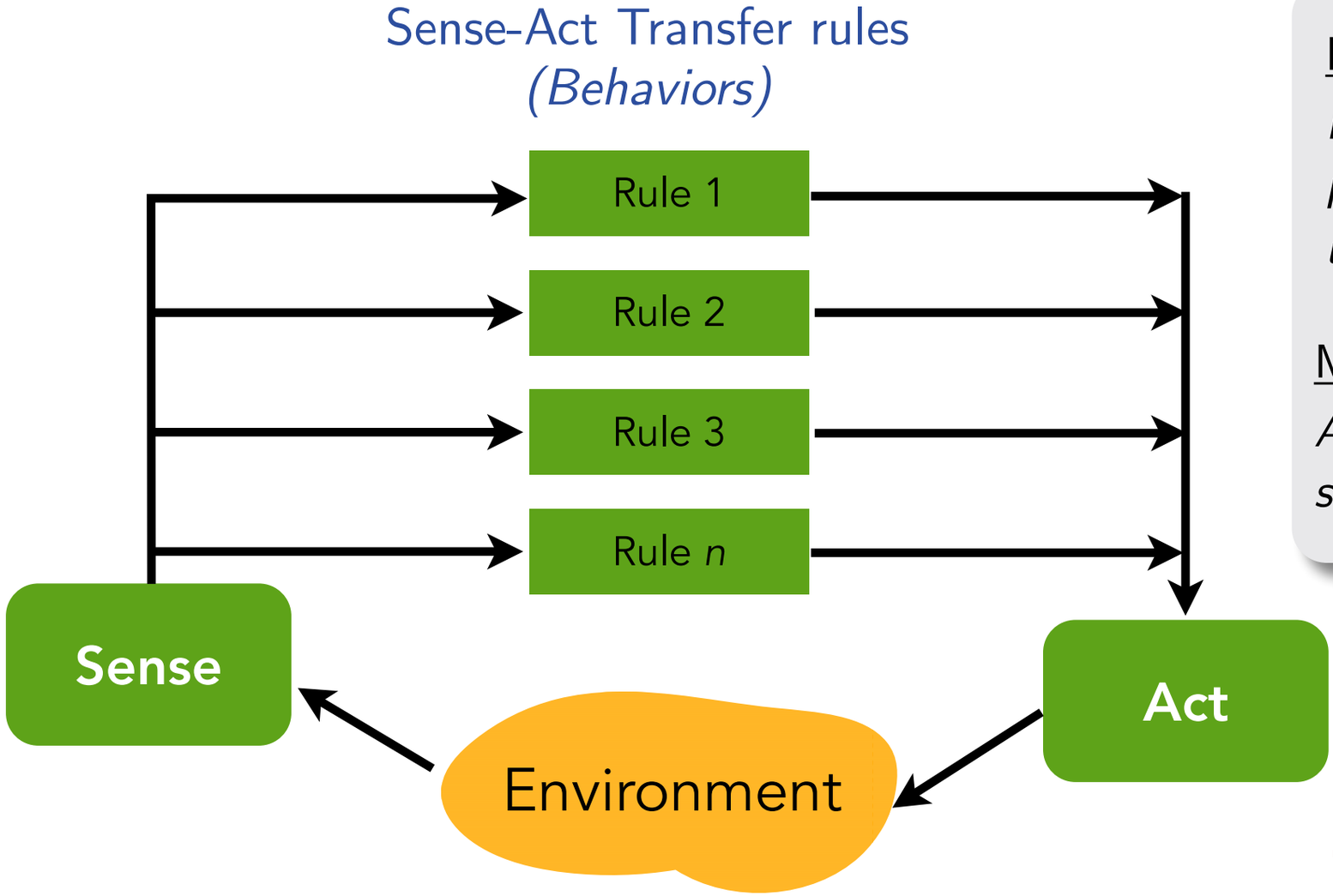


Iterative, Closed-loop
SENSE-(MODEL)-PLAN-ACT



Open-loop
[MODEL-PLAN]-ACT

Reactive control architectures: Don't Think, React!



Ethological view (***Behavior***):

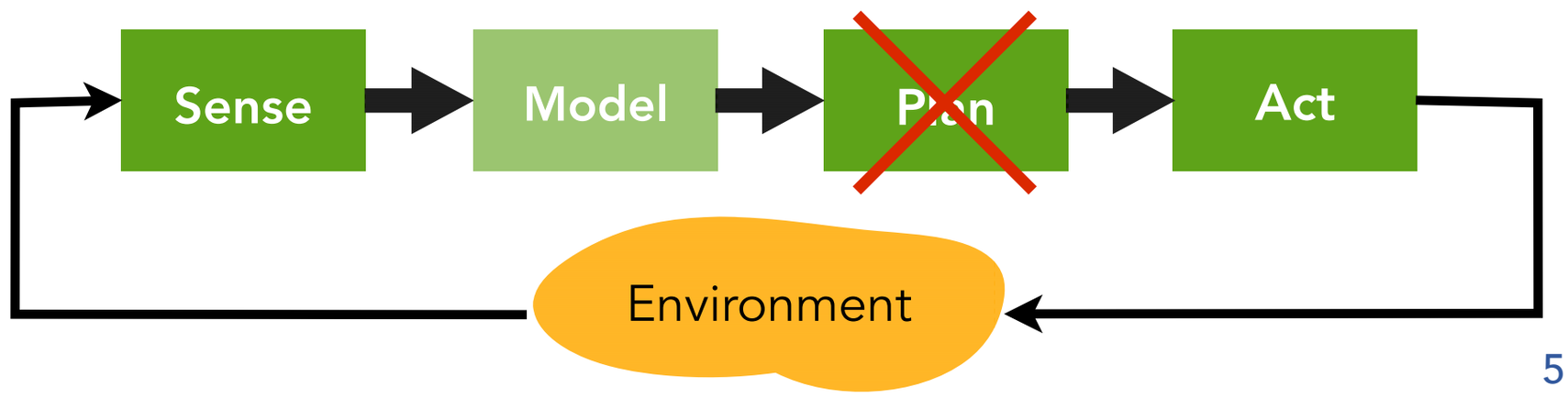
Direct mapping of sensory inputs to a pattern of motor actions that are then used to achieve a task

Mathematical view (***Function***):

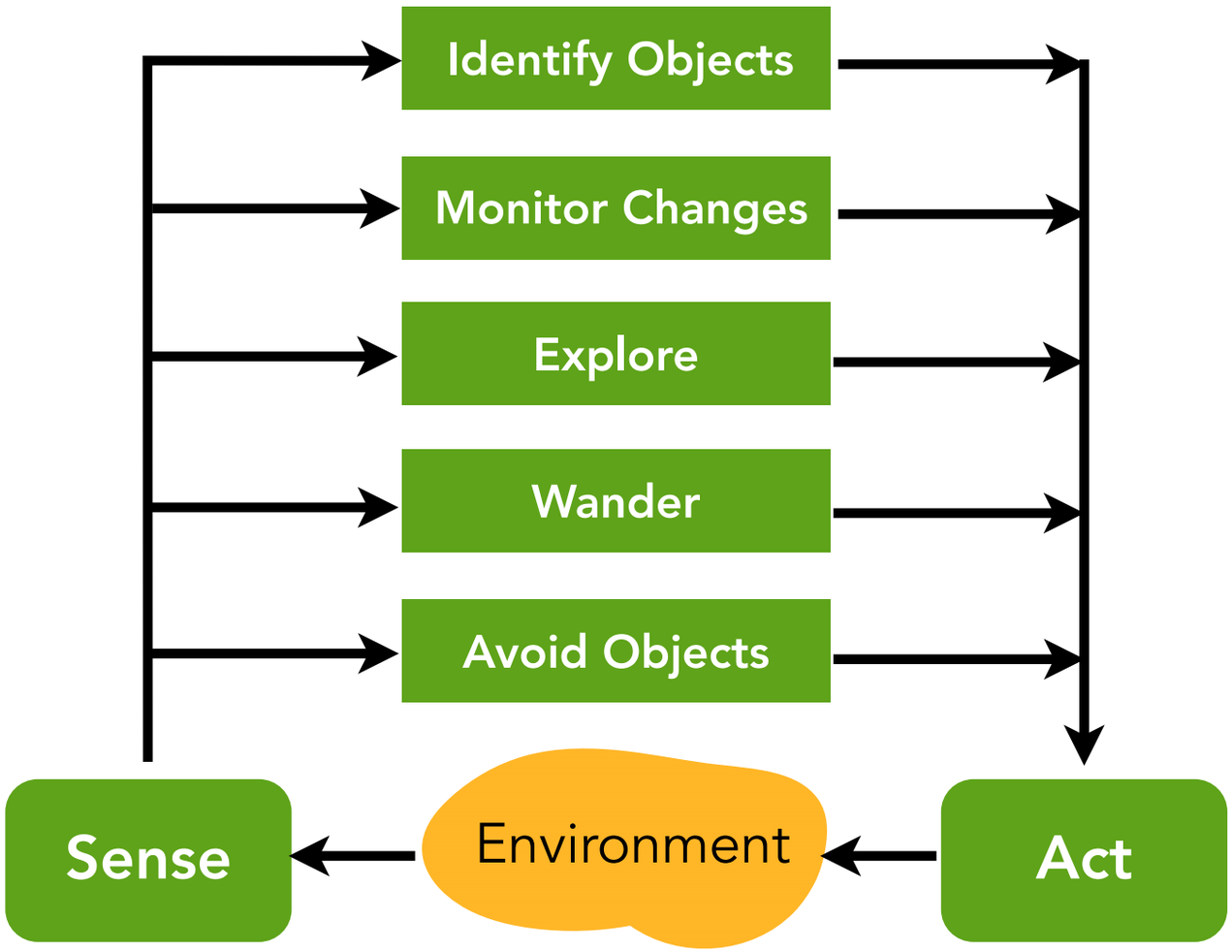
A transfer function, transforming sensory inputs into actuator commands

Vertical decomposition
vs.
Horizontal decomposition

Concurrent mode
vs.
Sequential mode

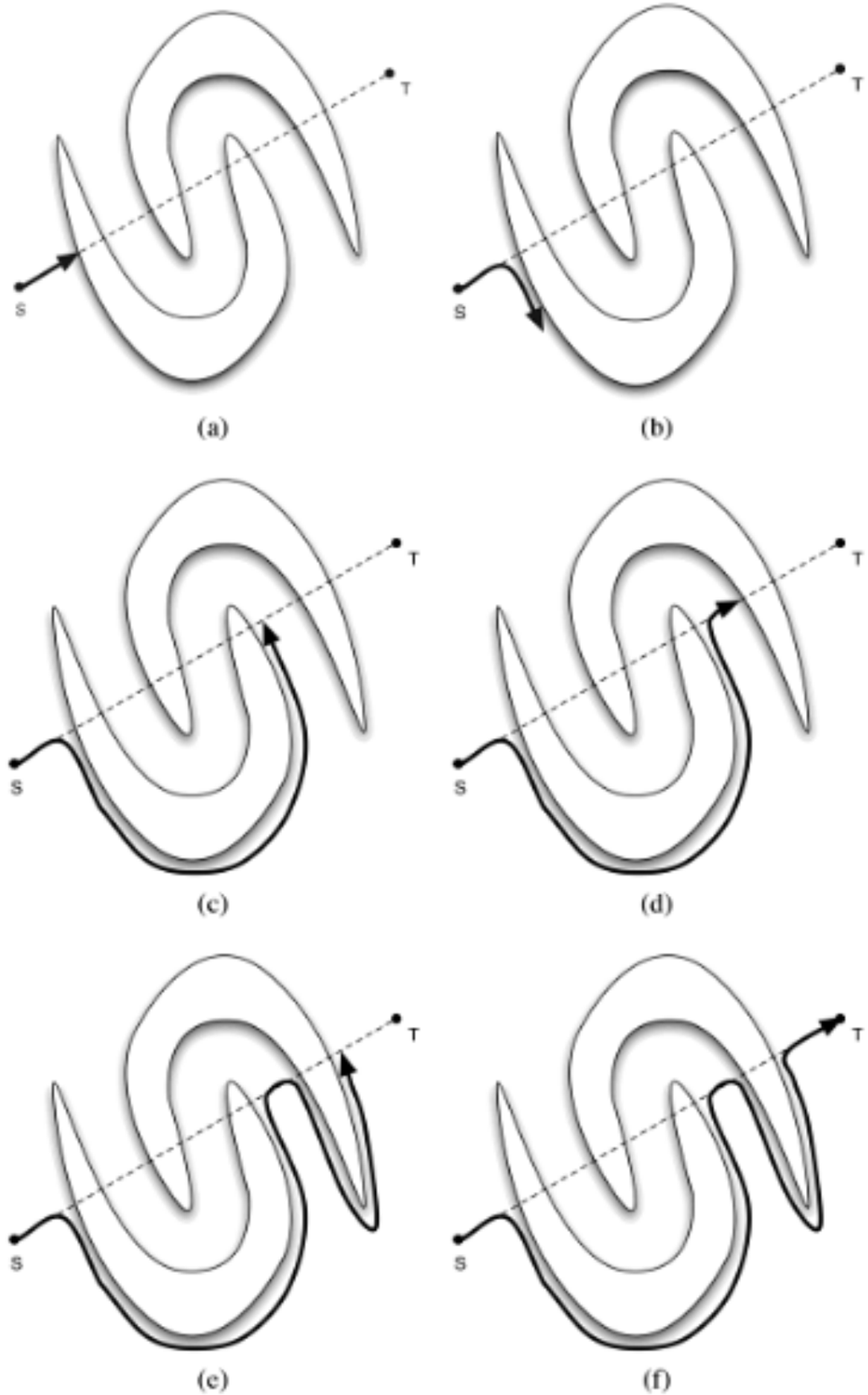


Reactive control architectures: Don’t Think, React!



A navigation behavior

Bug algorithms:
Example of reactive navigation



Deliberative vs. Reactive control architectures

Top-down problem solving

Requires a closed world

Deliberative paradigm

Internal models

Look ahead

Functional pipeline

Model free

React to inputs

Concurrent modules

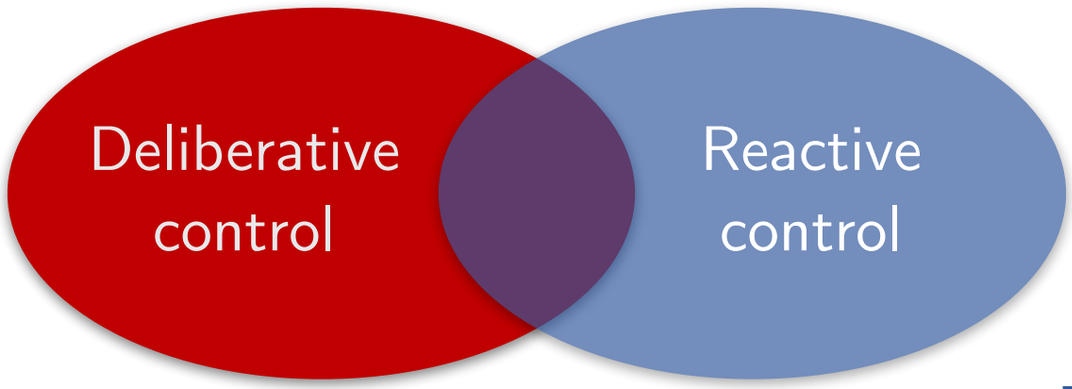
Reactive paradigm

Works in an open world

Bottom-up problem solving

Hybrid control architectures

The best of the two worlds!



Representations

Models

Planning

Hybrid control

Real-time

Multiple goals

Robustness

Flexibility

Modularity

How to combine ?

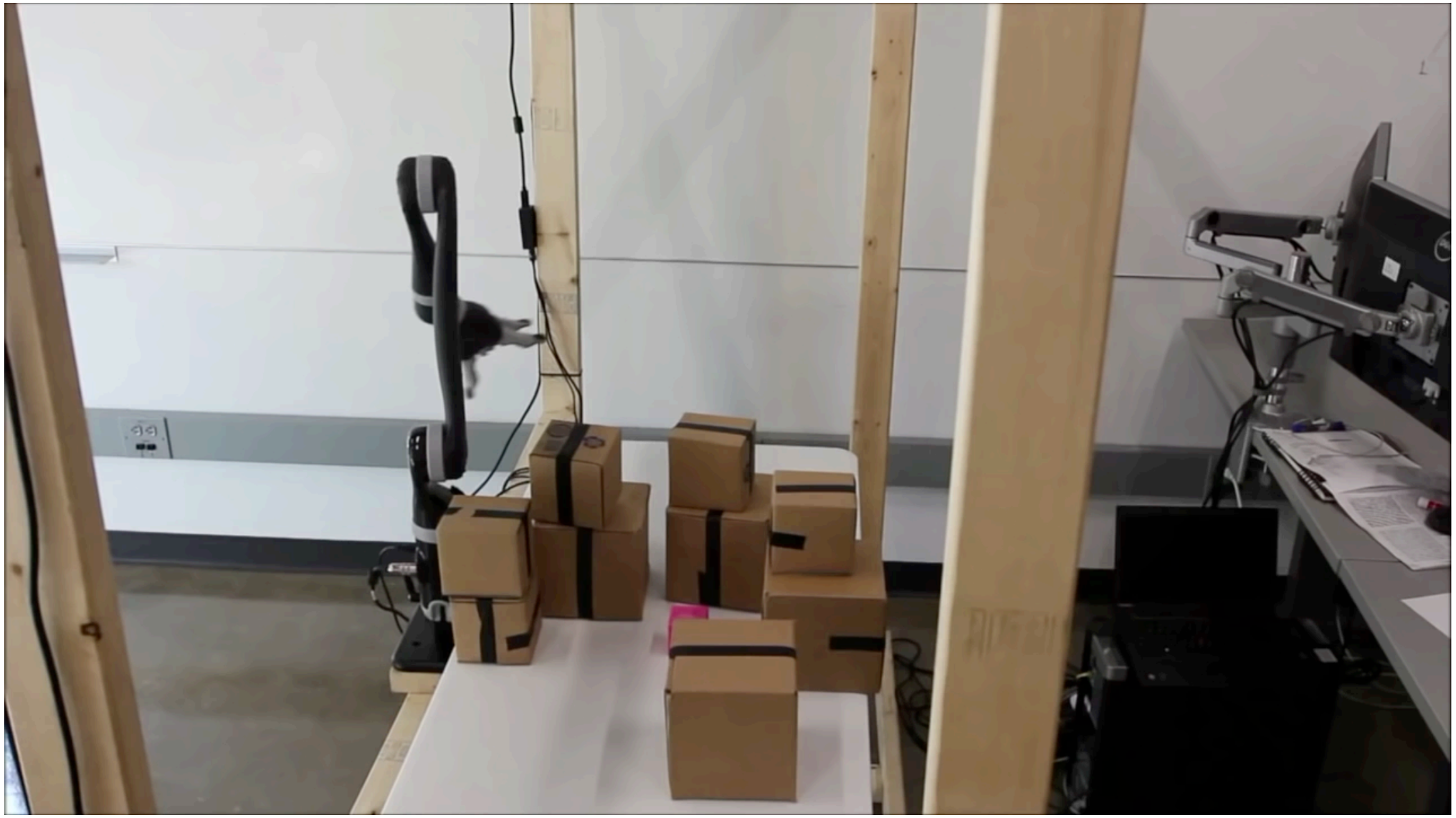
Strategic planning / reasoning

Low(er)-level controls and behaviors

... the most used approach nowadays, but still a sort of art

Motion planning

- ***Motion Planning:*** Specification of motion through space (and time)

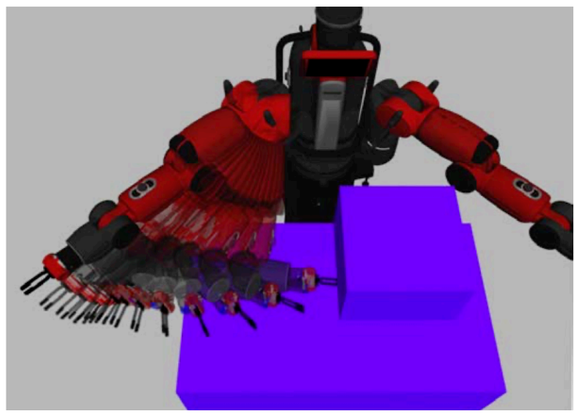


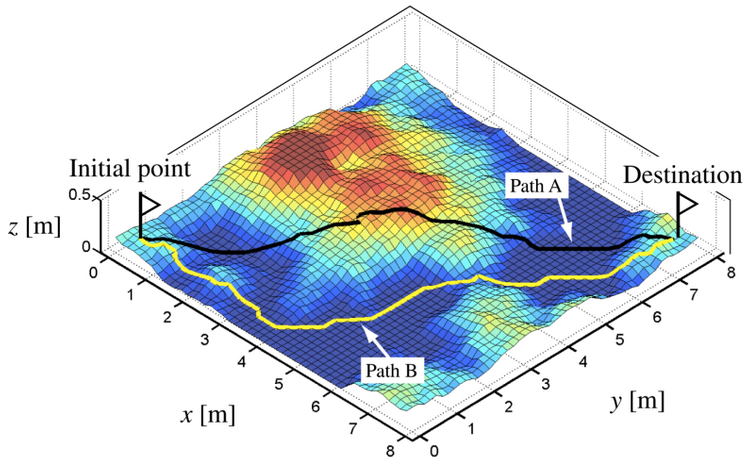
Motion planning

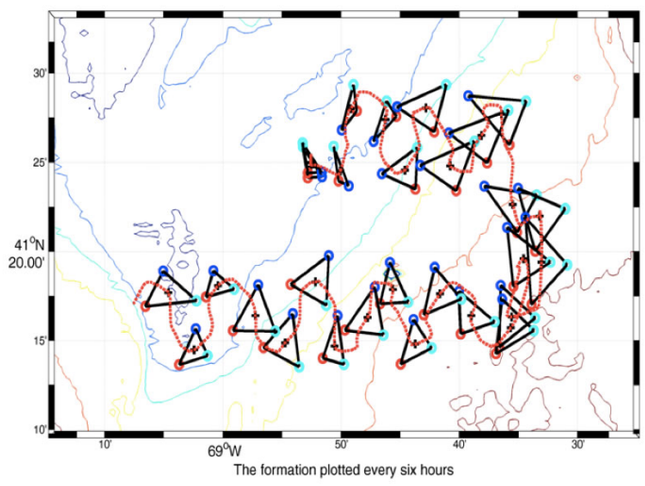
- ***Motion Planning:*** Specification of motion through space (and time)

Motion planning

- ***Motion Planning:*** Specification of motion through space (and time)
  - **Path planning**: Generation of a (feasible) path for going from configuration A to B
  - **Trajectory planning:** Generation of a path for going from A to B specifying how to move based on velocity, time, differential constraints, and dynamics
  - **Coverage planning:** Visit (or observe with sensors) all places (at least or only once)
  - **Formation planning:** Move from A to B in a formation under specified rules ...
  - ...







Open Motion Planning Library

http://ompl.kavrakilab.org/gallery.html

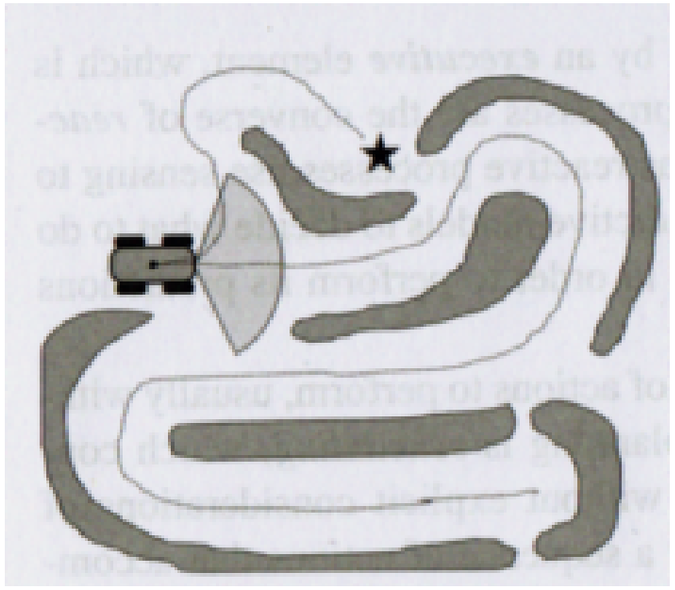
General parameters / design choices

- How far should we look into the future? → **Depth of look ahead**
  The farthest, the more *reliable* the plan is and the more *expensive* computation is

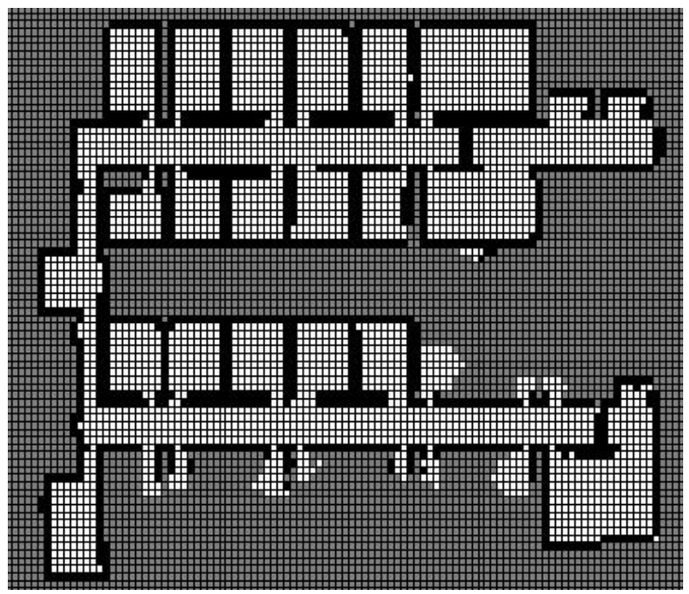
  ✓ Model Predictive Control (MPC): iterative replanning up to a certain horizon

- What *spatial resolution* map model? → **Accuracy of spatial representation**
  The higher the resolution the higher the computational requirements

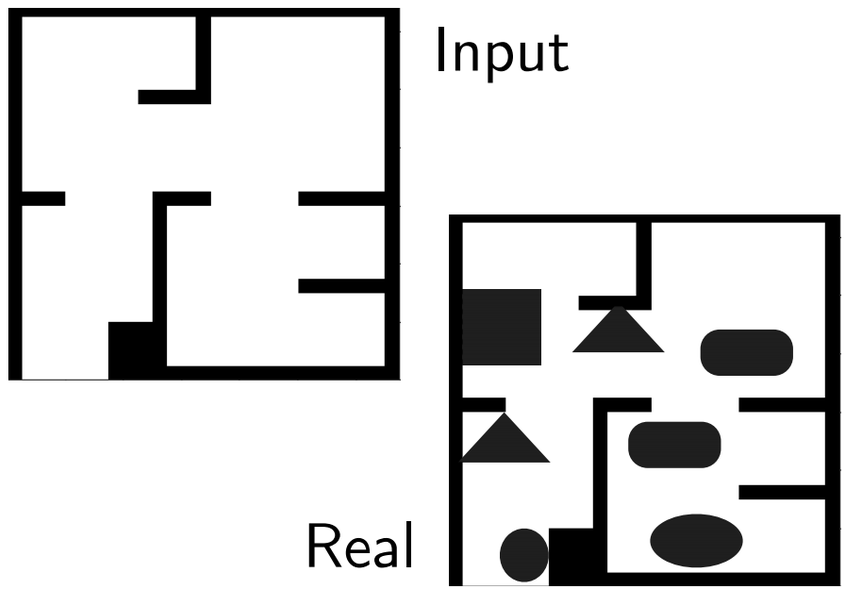
- How realistic / reliable is the input model (the map)?
  What are the effects of **imprecise or partial models**?



Without planning / looking ahead things can get arbitrarily bad!



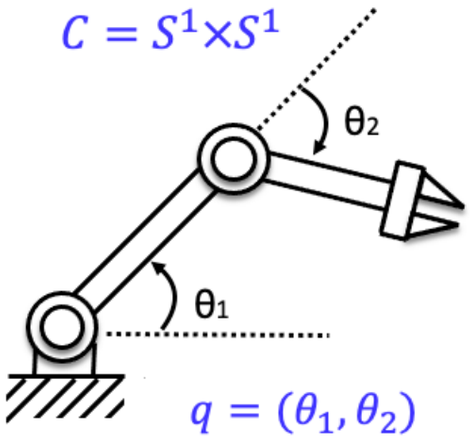
An extremely fine-grained representation (occupancy grid) might require heavy computation



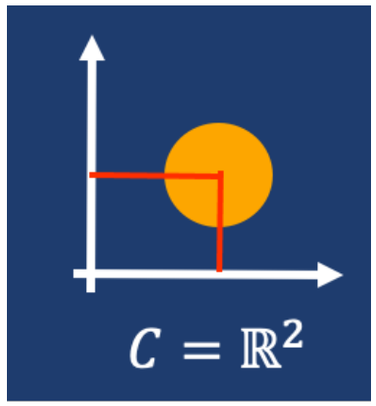
A plan good / feasible in the incomplete input map could easily be not feasible in the real environment

Recap: Configuration space, C-space, $\mathcal{C}$

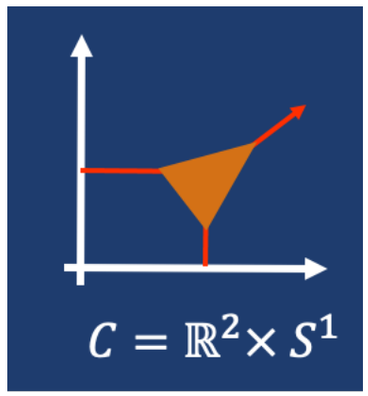
- **Generalized coordinates**: n parameters $\boldsymbol{q} = (q_1, q_2, \ldots, q_n)$ that are sufficient to uniquely describe system configuration relative to some reference (frame, configuration)
- **Configuration space** (C-space, $\mathcal{C}$): n-dimensional space identified by the generalized coordinates, defining the set of possible robot's configurations



$q = (\theta_1, \theta_2)$

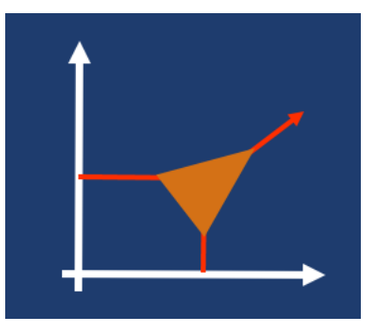


$q = (x, y)$



$q = (x, y, \theta)$

- **Workspace**: set $\mathcal{W}$ of all configurations that the robot, based on its physical structure, can feasibly reach in the embedding environment to perform its *work*



$\mathcal{W} = \mathbb{R}^2$ if robot's orientation doesn't matter for work

$\mathcal{W} = \mathbb{R}^2 \times S^1$ if robot's orientation does matter for work

Configuration space and state evolution

✓ If **control inputs** $u \in \mathcal{U} \subseteq \mathbb{R}^m$ are a vector of m velocities (i.e., kinematics is sufficient)
→ **State** ξ of the robot system coincides with q: $\xi(q) = q$

◉ If control inputs include accelerations (forces), i.e., robot has 2nd order dynamics such that its response is described by 2nd order differential equations of type $F = ma, \rightarrow F = m\dot{q} = m\ddot{q}$
→ **State** ξ of the robot system includes generalized velocities $\dot{q}$: $\xi = (q, \dot{q})$

- **Equations of robot motion:** $\dot{\xi} = f(\xi, u)$ $\quad \xi(T) = \xi(0) + \int_0^T f(\xi(t), u(t))dt$

If $\xi = q$ $\quad \dot{q} = f(q, u)$ $\quad q(T) = q(0) + \int_0^T f(q(t), u(t))dt$

f summarizes the controls over time as defined by the **plan**
+ robot's kinematics/dynamics

Free and occluded space

❖ **Obstacle region**: $\mathcal{O},\ \mathcal{O} \subseteq \mathcal{W}$

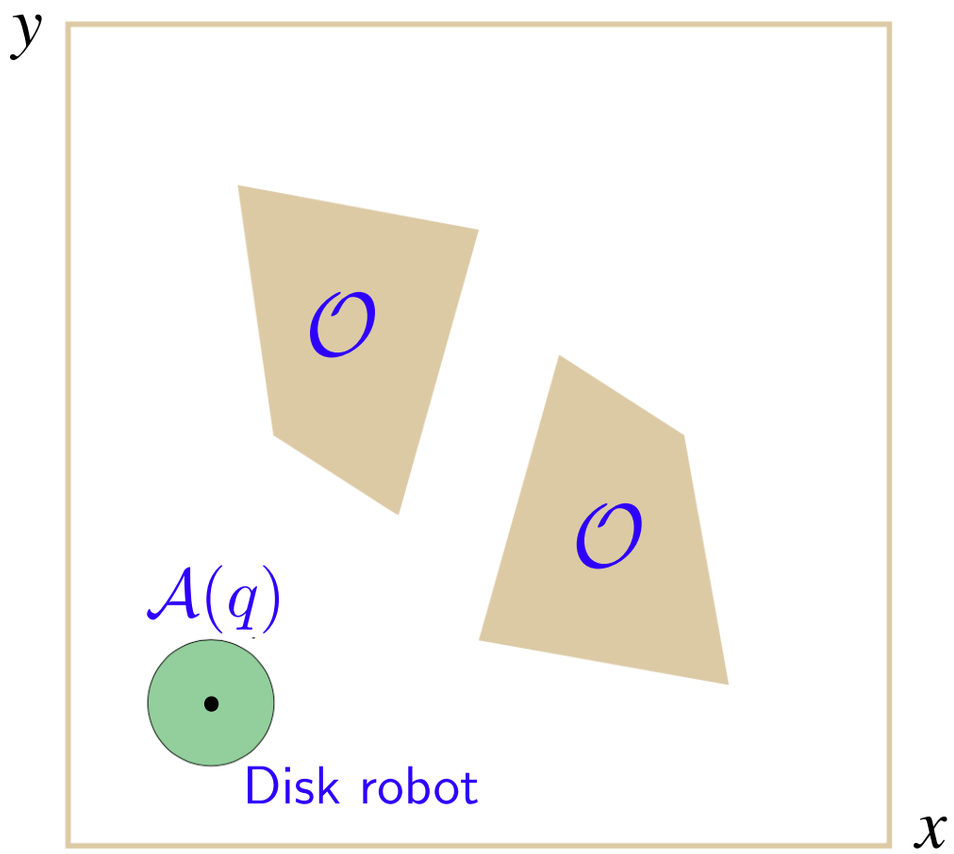
set of all points in the workspace that lie in one or more **obstacles**

- In general, we will safely assume that $\mathcal{W} \subseteq \mathbb{R}^2$ or $\mathcal{W} \subseteq \mathbb{R}^3$

❖ **Robot**: $\mathcal{A},\ \mathcal{A} \subseteq \mathcal{W}$

set of all points in the workspace that are covered/occupied by the robot

- $\mathcal{A}(q)$ is the set of workspace points occupied by the robot when in configuration q



$\mathcal{W} \subset \mathbb{R}^2$

$q = (x, y), \quad \mathcal{C} = \mathcal{W}$

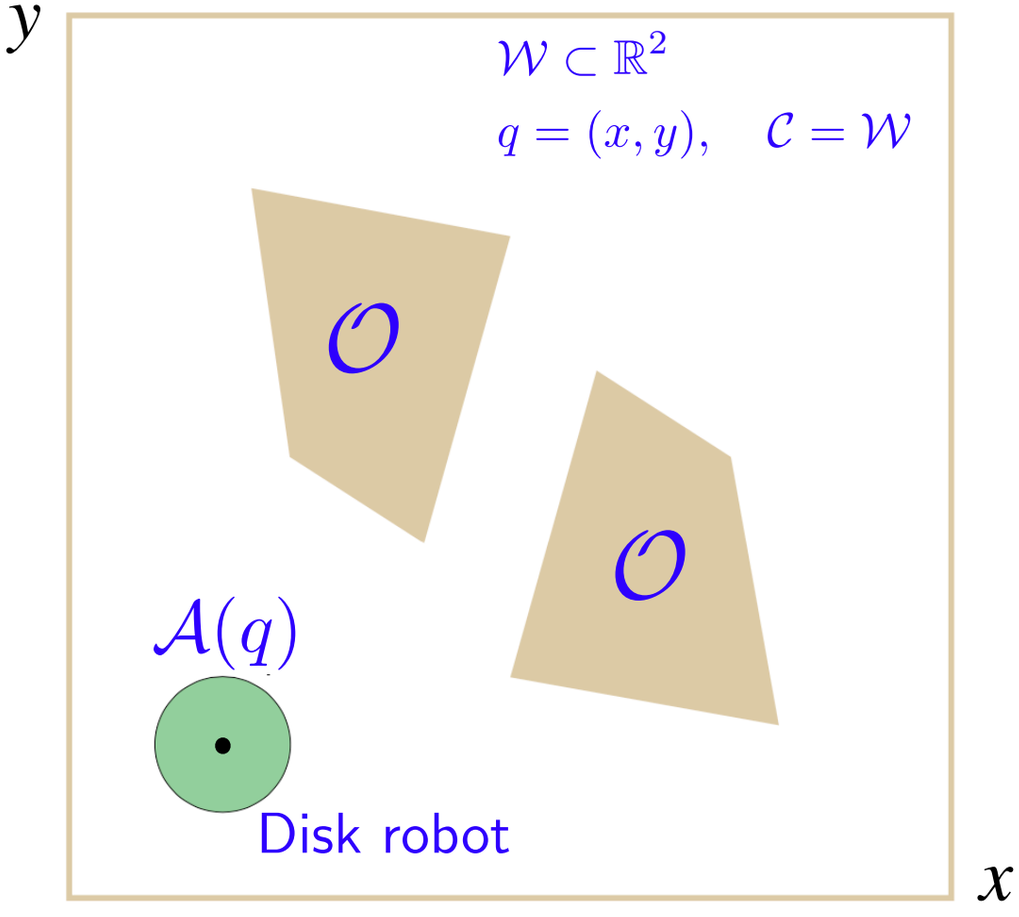
Free and occluded space configuration space

- **Obstacle region for robot in *configuration space***: $\mathcal{C}_{obs} \subseteq \mathcal{C}$

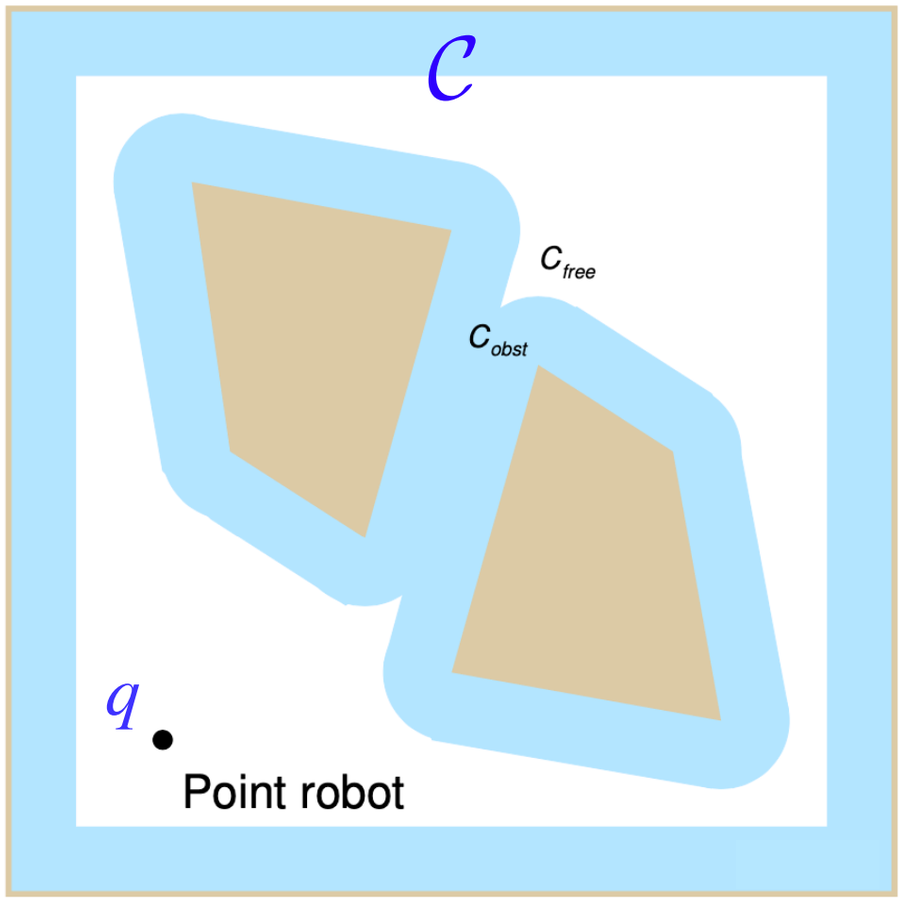
  set of all configurations q at which the *transformed* robot $\mathcal{A}(q)$ would intersect the obstacle region $\mathcal{O}$

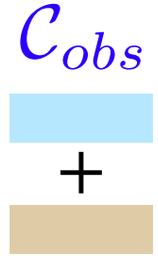
$$\mathcal{C}_{obs} = \{q \in \mathcal{C} \mid \mathcal{A}(q) \cap \mathcal{O} \neq \emptyset\}$$

- **Free space region for robot in *configuration space***: $\mathcal{C}_{free} = \mathcal{C} \setminus \mathcal{C}_{obs}$



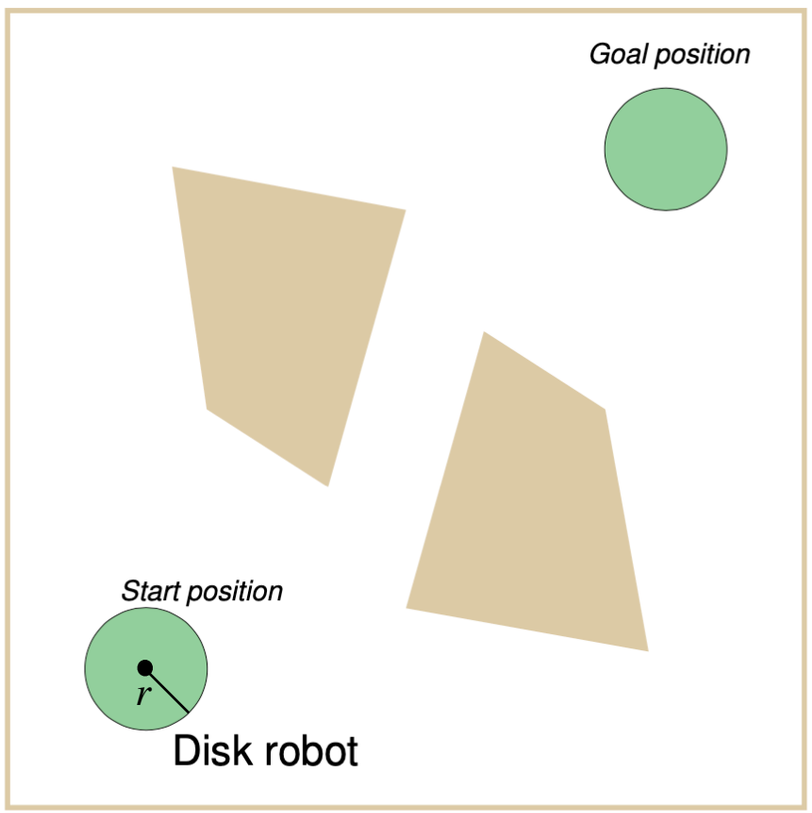
$\Rightarrow$



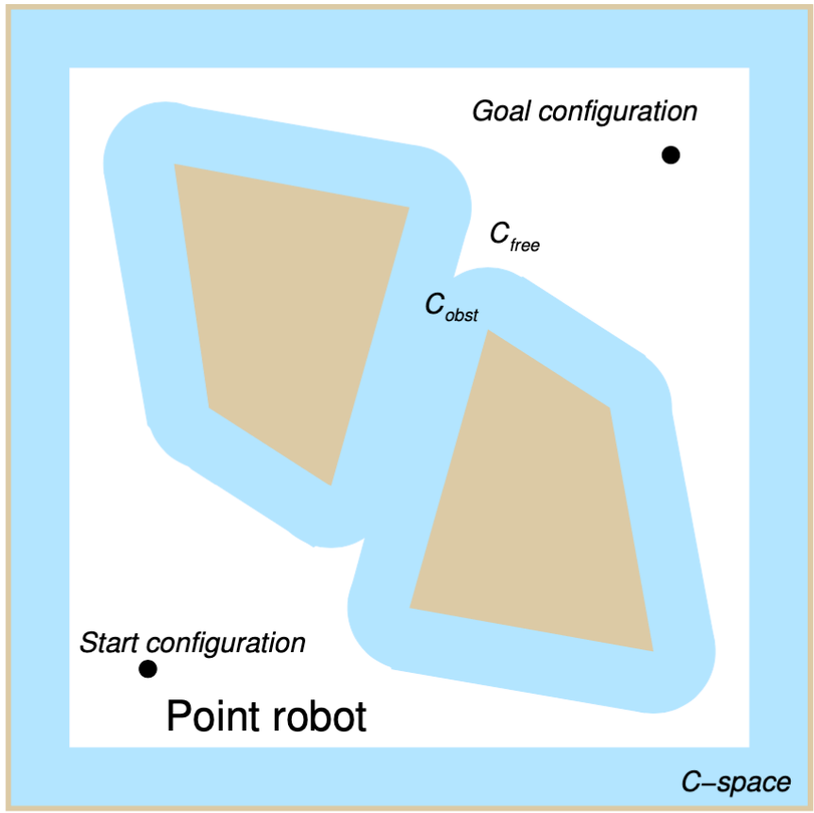


Motion planning: Point robot in configuration space

➡ For the purpose of **motion planning**, once geometry of robot and of environment, including the obstacles, are represented in **configuration space**, the robot is conveniently reduced to a **point**, *since its geometry is already accounted in* $\mathcal{C}_{free}$



Motion planning problem in geometrical representation of $\mathcal{W}$



Motion planning problem in $\mathcal{C}$-space representation

$\mathcal{C}_{obs}$ has been obtained by **enlarging the obstacles** by the disk of radius r

By applying Minkowski sum: $\mathcal{O} \oplus \mathcal{A} = \{x + y \mid x \in \mathcal{O}, y \in \mathcal{A}\}$

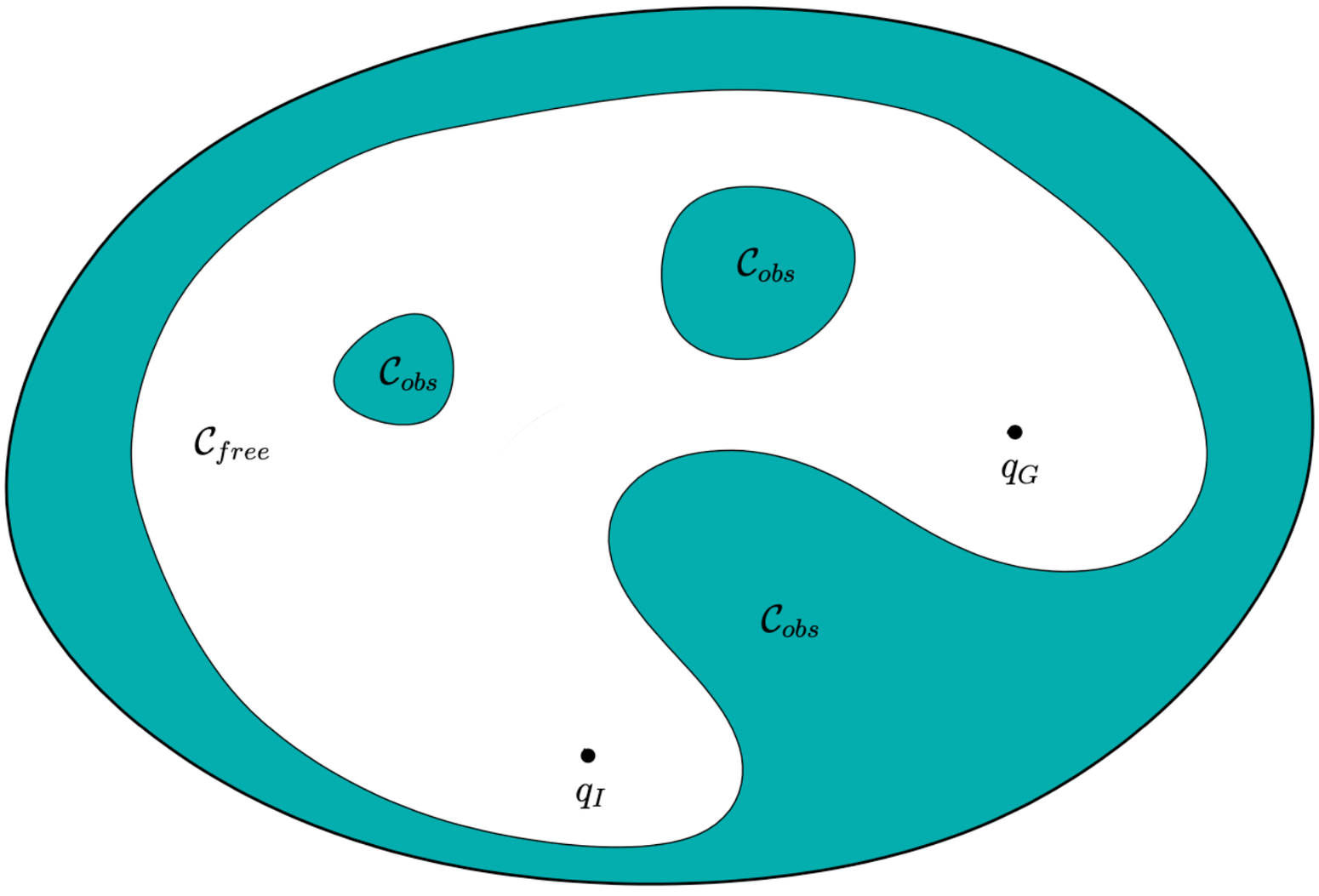
Motion planning: (Basic) Path planning (Piano mover's problem)

Inputs:

- **Initial pose** of the robot, as a configuration-space vector $\boldsymbol{q}_I$
- **Goal pose,** as a configuration vector $\boldsymbol{q}_G$
- **Description of the robot:** Geometry, kinematics, dynamics
- **Description of the world:** Map, including existing obstacles

} Query

$\Rightarrow$ $\mathcal{C}_{free}$, $\mathcal{C}_{obs}$



Motion planning: (Basic) Path planning (Piano mover's problem)

Goal (minimal):

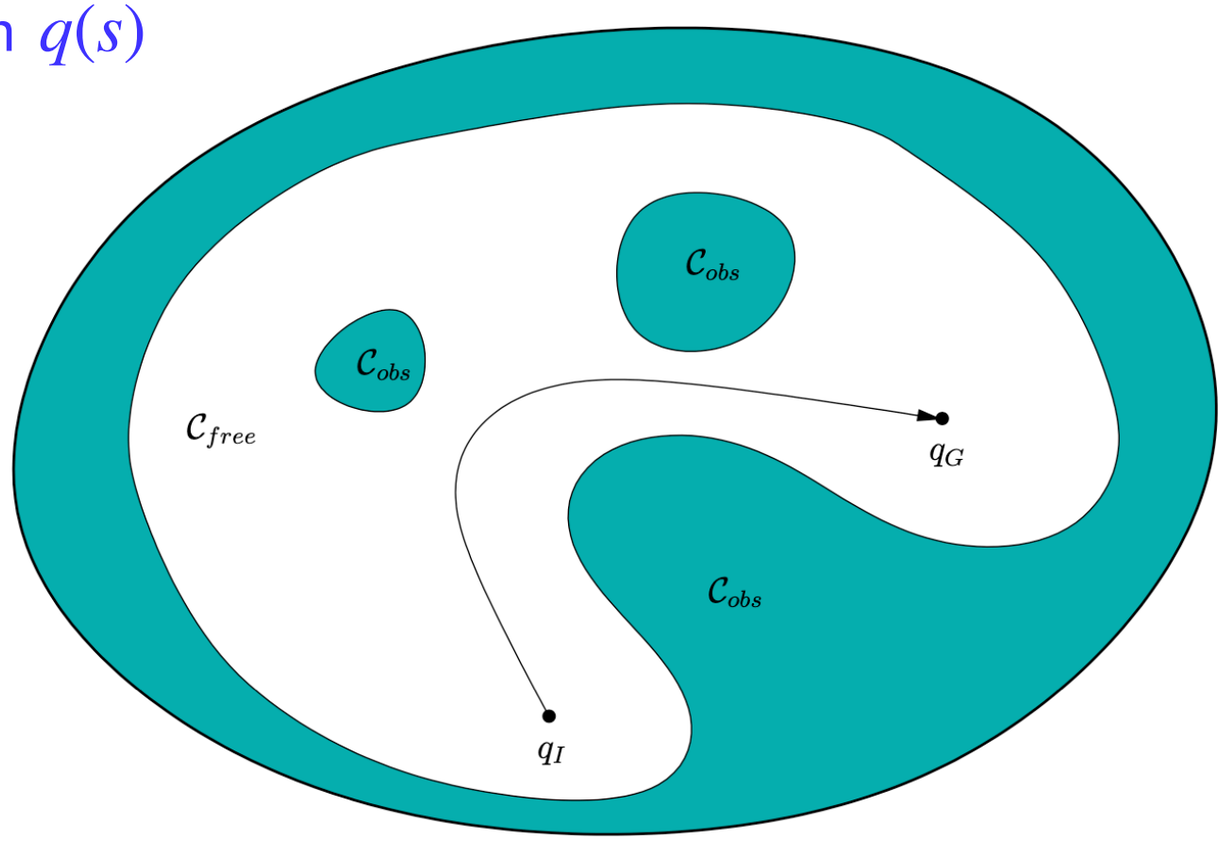
☑ Find a **path** π that moves the robot from initial to goal configurations **never colliding with obstacles** (***admissibility***)

$$\pi : [0,1] \rightarrow \mathcal{C}_{free}$$

$$\pi(0) = q_I$$

$$\pi(1) = q_G$$

- $\pi(s)$, $s \in [0,1]$ is a **continuous parametric function** mapping *geometric* parameter s into an admissible configuration $q(s)$



E.g., a parametric path in configuration space $q = (x, y)$ could be obtained as a 3rd order polynomial

$$x(s) = s^3 x_f - (s-1)^3 x_i + \alpha_x s^2 (s-1) + \beta_x s (s-1)^2$$

$$y(s) = s^3 y_f - (s-1)^3 y_i + \alpha_y s^2 (s-1) + \beta_y s (s-1)^2$$

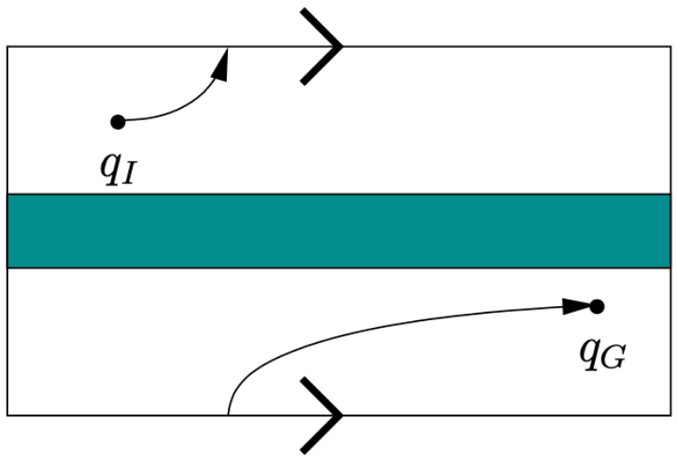
satisfy the boundary conditions on x, y

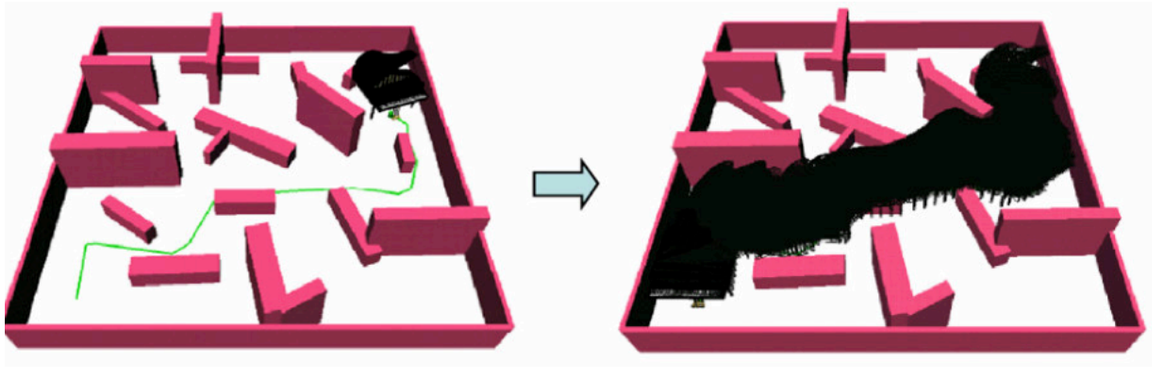
$$x(0) = x_i \qquad x(1) = x_f$$
$$y(0) = y_i \qquad y(1) = y_f$$

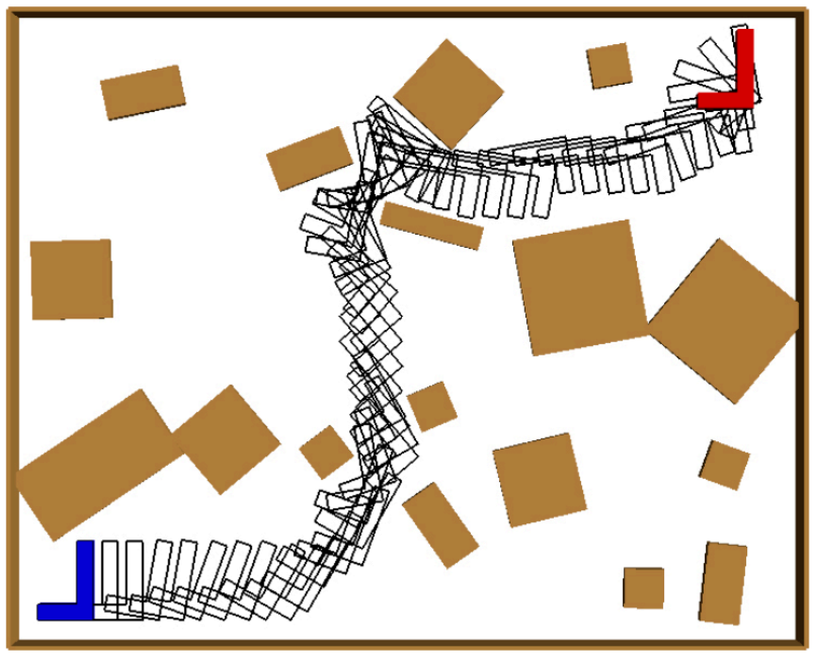
Motion planning: (Basic) Path planning (Piano mover's problem)

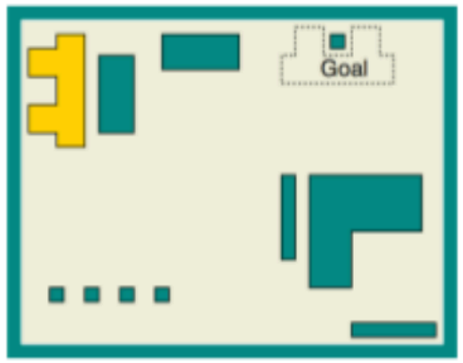
- $\pi(s)$, $s \in [0,1]$ defines a continuous set of transformations / configurations for the robot which are purely **geometric**

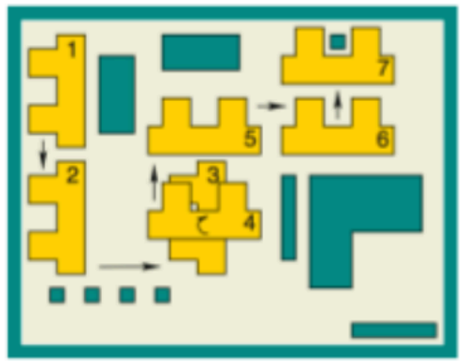
- $q(s), q(s+ds)$ are neighbor configurations in the **topological** space representing $\mathcal{C}$





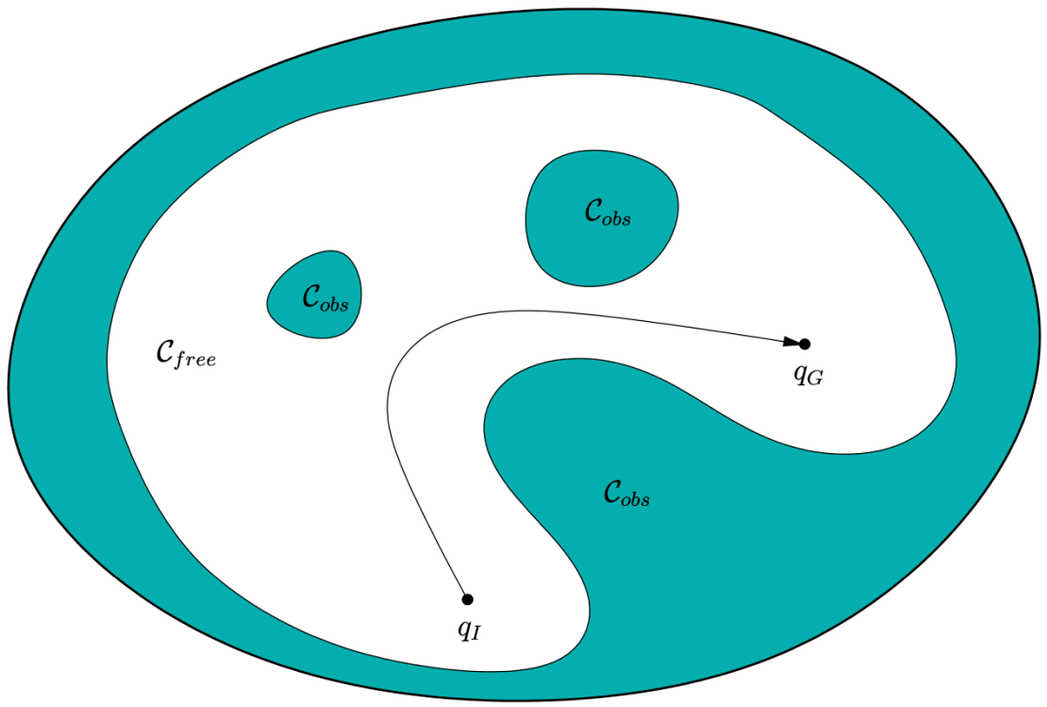






- Velocity controls u e.g., v, ω, to actuate the path?
- What if the robot is non-holonomic?

Motion planning: Trajectory planning



- Velocity controls u e.g., v, ω, to actuate the path?

$$q = (x, y, \theta)$$

$$v_t = \frac{dq}{dt} = (\dot{x}, \dot{y}, \dot{\theta}) \quad \text{velocity controls driving and steering the robot over time } t$$

$$v_s = \frac{dq}{ds} \quad \text{velocity along geometric path } q(s) \text{ in configuration space}$$

$$v_t = \frac{dq}{dt} = \frac{dq}{ds}\frac{ds}{dt} = \frac{dq}{ds}\dot{s} = v_s\,\dot{s}$$

- v_s is the rate of change as defined by the geometric path
- $\dot{s}$ defines a (selected) **time scaling** of geometric velocity, reflected in time velocity v_t

Motion planning: Trajectory planning

Inputs:

- **Start state** of the robot, $\xi(0) = \xi_I = (q_I, \dot{q}_I)$
- **Goal state,** $\xi_G = (q_G, \dot{q}_G)$
- **Description of the robot:** Geometry, kinematics, dynamics
- **Description of the world:** Map, including existing obstacles

Goal:

- Find:
  - **A time T**
  - **A set of controls u to be issued from 0 to T,** $u : [0, T] \rightarrow \mathcal{U}$

    e.g., $(v_0, \omega_0), (v_1, \omega_1), \ldots, (v_T, \omega_T)$
- Such that, based on the equations of motion and robot, and on definition of $\mathcal{C}_{free}$
  - ☑ $\xi(T) = \xi_G$
  - ☑ $q(\xi(t)) \in \mathcal{C}_{free}\ \forall t \in [0, T]$

+ Feedback-based control to ensure correct execution of plan

Path planning: required or desirable objectives

☑ Find a **path** π that moves the robot from initial to goal configurations **never colliding with obstacles** (***admissibility***)

☑ Find a **path that complies with robot motion constraints** (***feasibility***)

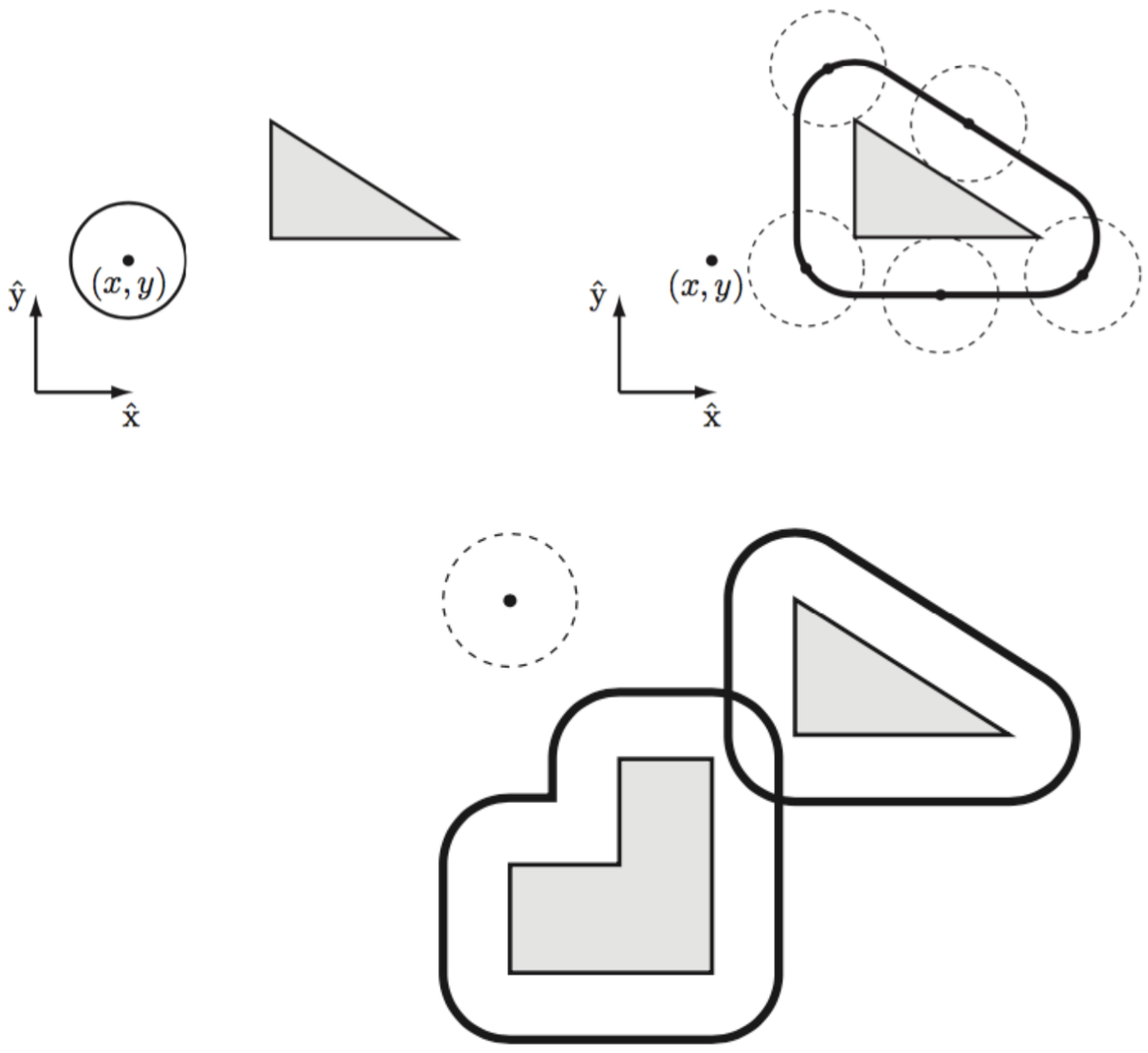
☑ If no feasible + admissible solution exist → Report a *failure* (PSPACE-hard)

☐ ***Optimality:*** If more than one solution exists, select the best

☐ ***Completeness:*** If at least one feasible + admissible path exists, the path planning algorithm is able to *find it.*

Optimality metrics: **Shortest distance**, **Minimum time**, Minimum curvature, Min effort, Max smoothness, Maximal safety, Min/Max ...

Construction of C-free in presence of obstacles

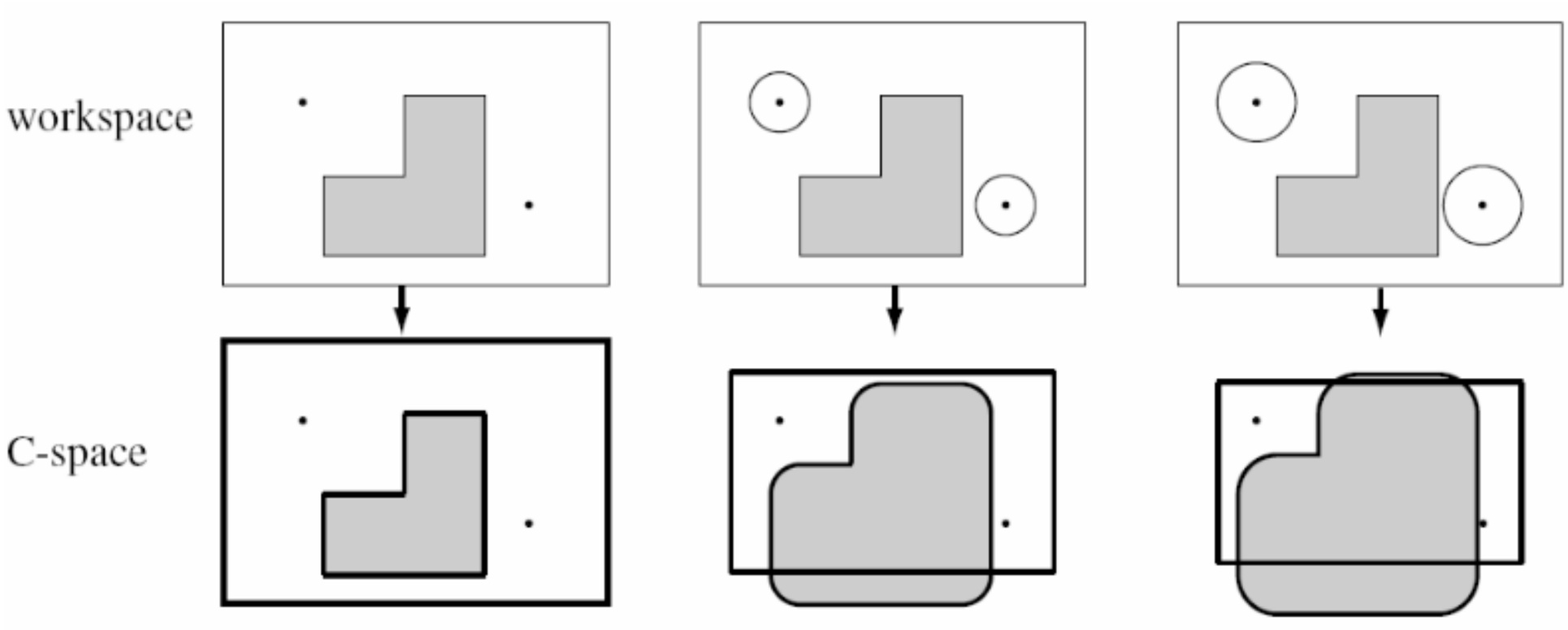


Single *reference point* (single-body robots):

$\mathcal{C}_{free}$ the accessible C-space, is obtained w.r.t. the reference point by sliding the robot along the edge of the obstacle regions "blowing them up" by the robot radius

Construction of C-free in presence of obstacles

Disk robots
with different radius/size



Construction of C-free: polygonal robot, only translation

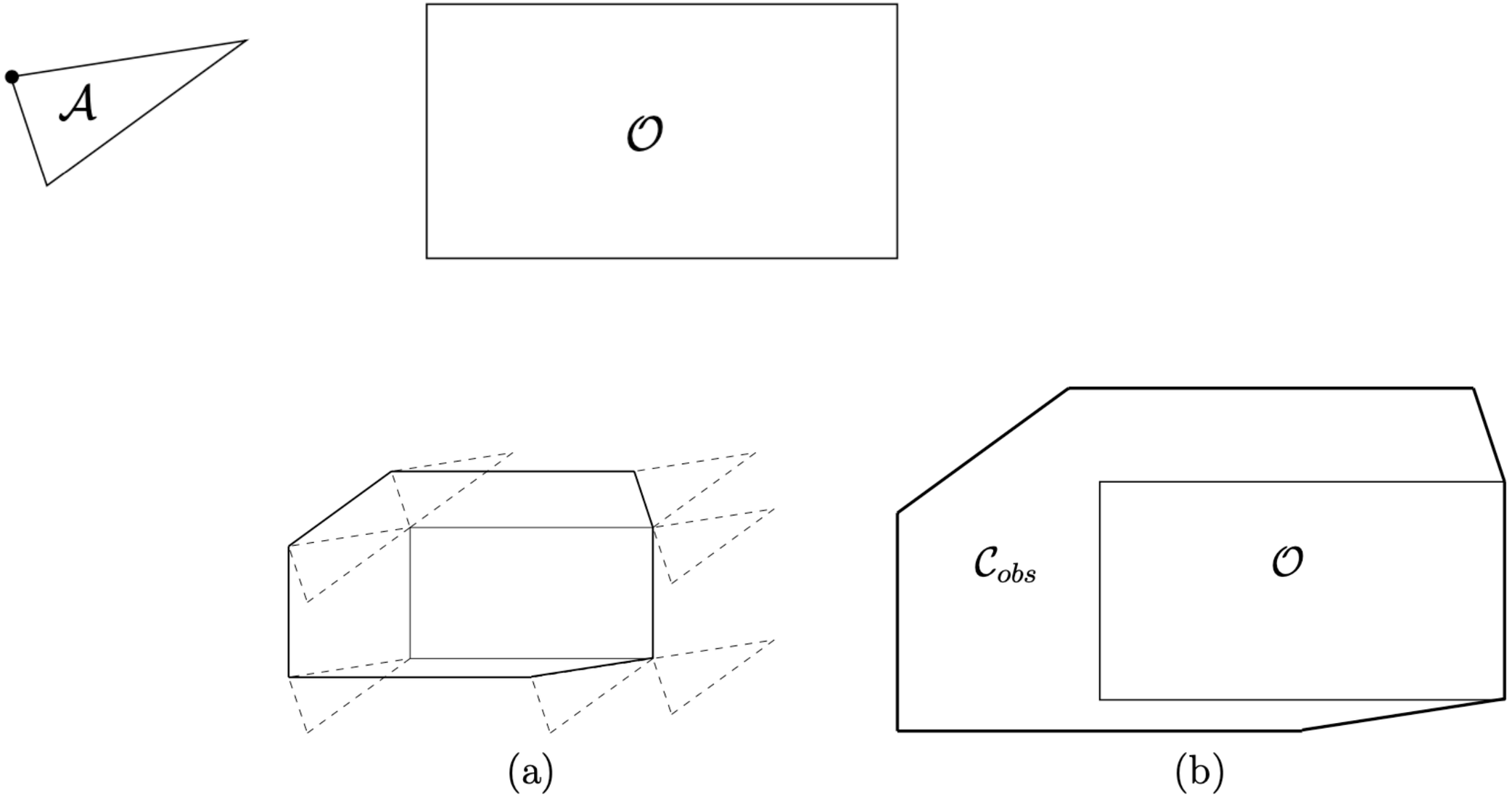
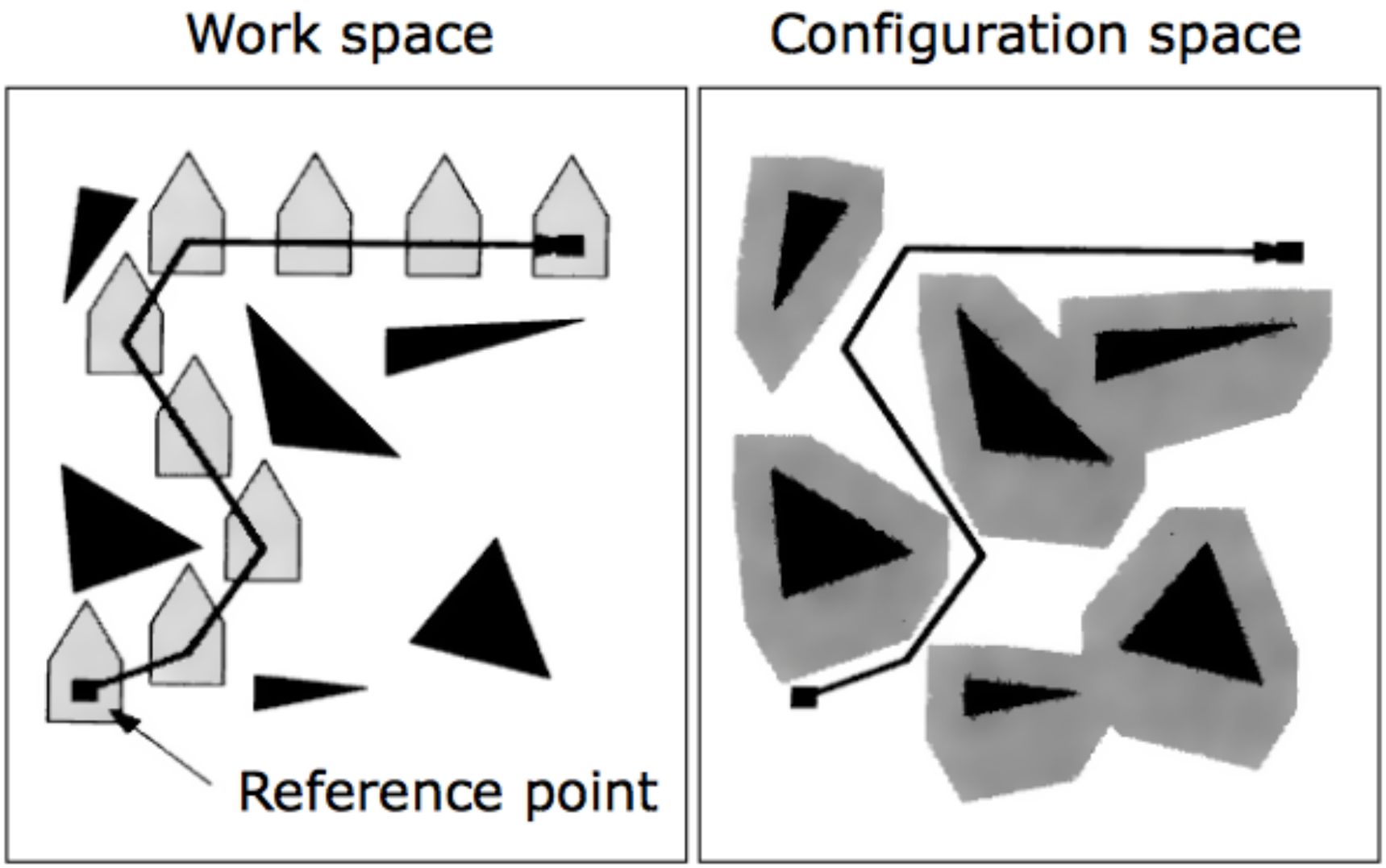


Figure 4.14: (a) Slide the robot around the obstacle while keeping them both in contact. (b) The edges traced out by the origin of $\mathcal{A}$ form $\mathcal{C}_{obs}$.

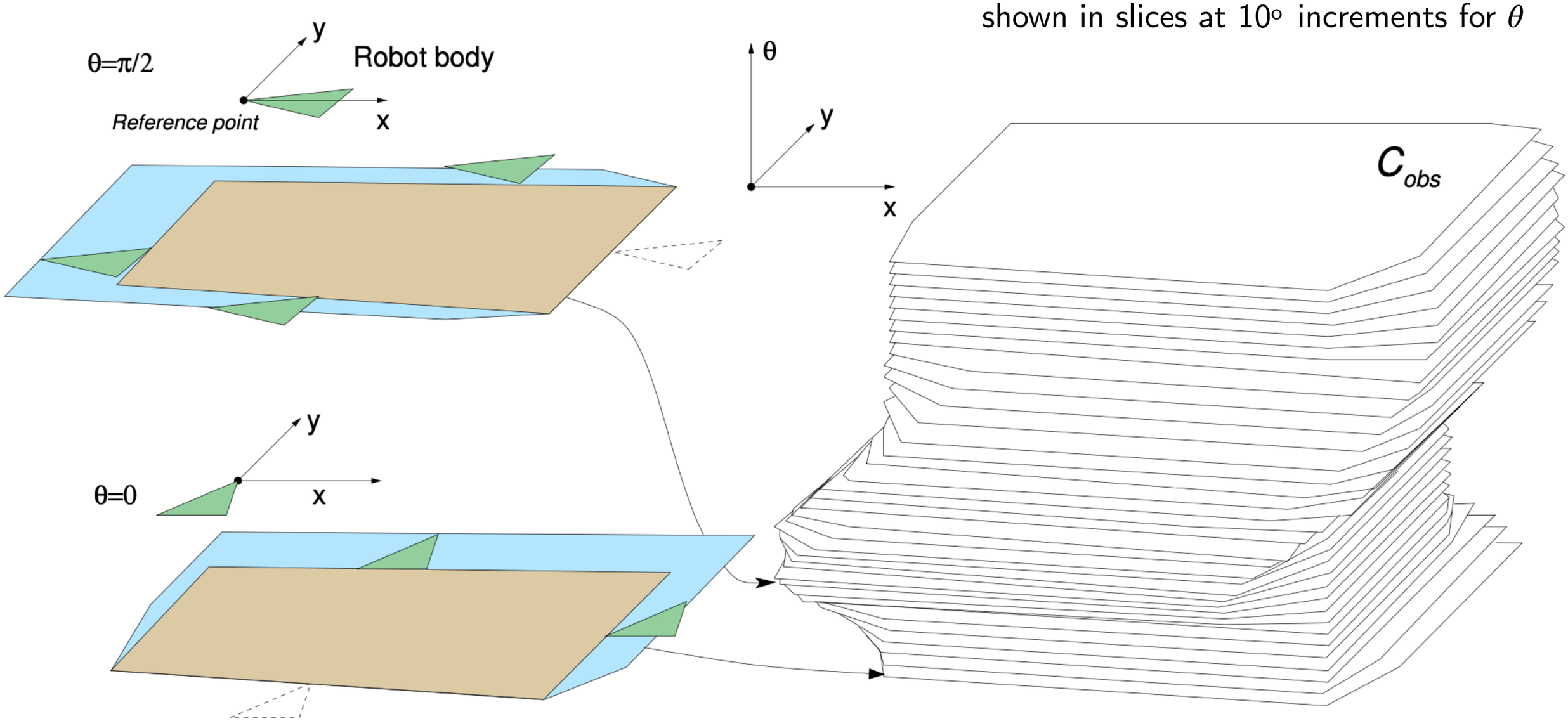
Steven M. LaValle, Planning Algorithms, Cambridge University Press, 844 pp., 2006

Construction of C-free: polygonal robot, only translation



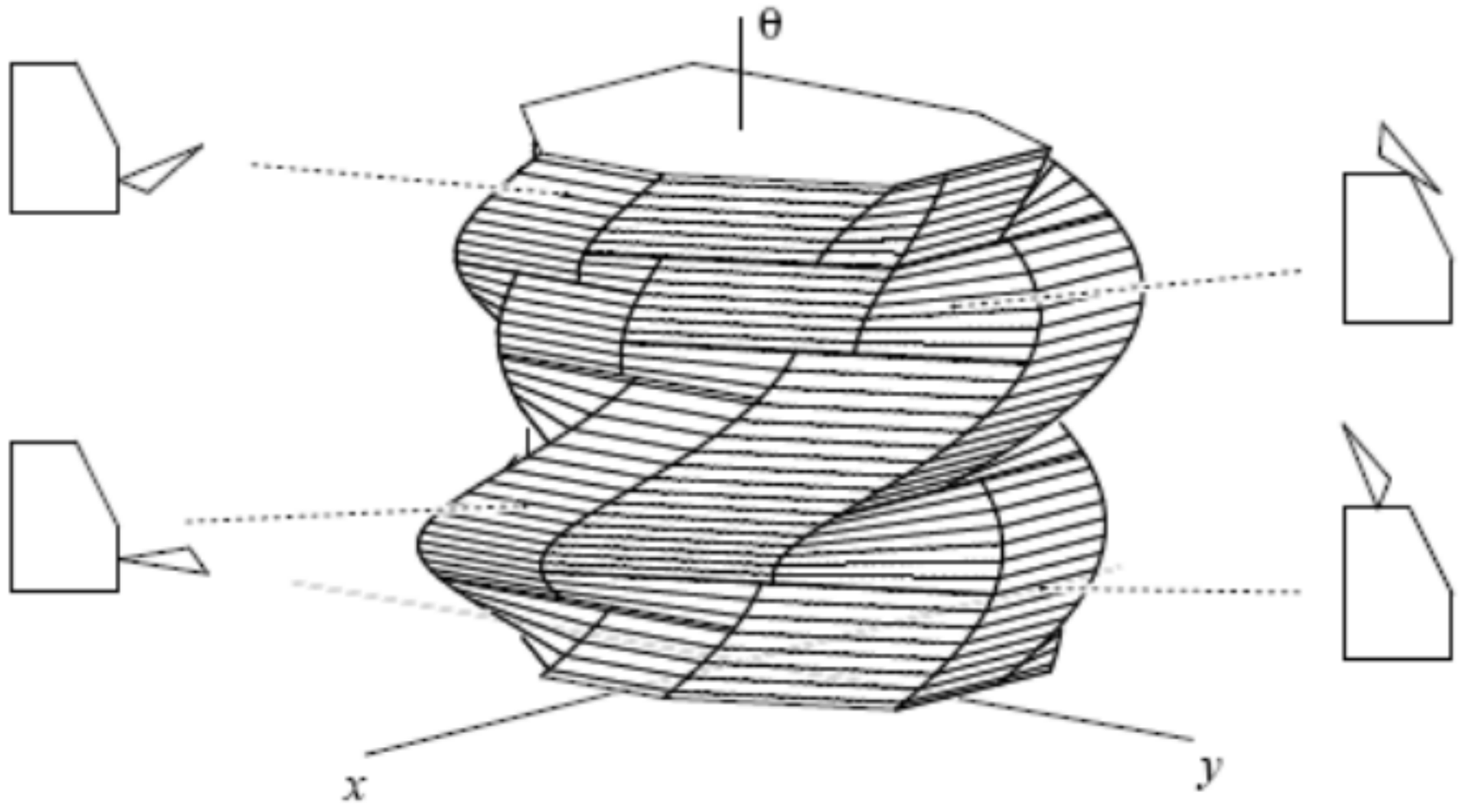
Construction of C-free: polygonal robot, translation + rotation

Full 3-dimensional C-space obstacle shown in slices at 10° increments for θ



Simple polygonal robot and simple polygonal obstacle → Very complex configuration space!

Construction of C-free: polygonal robot, translation + rotation

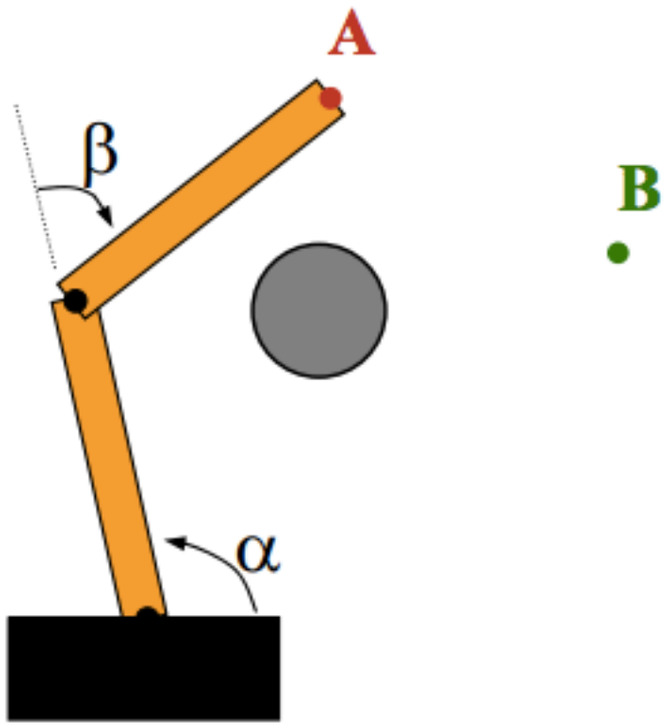


Another similar example

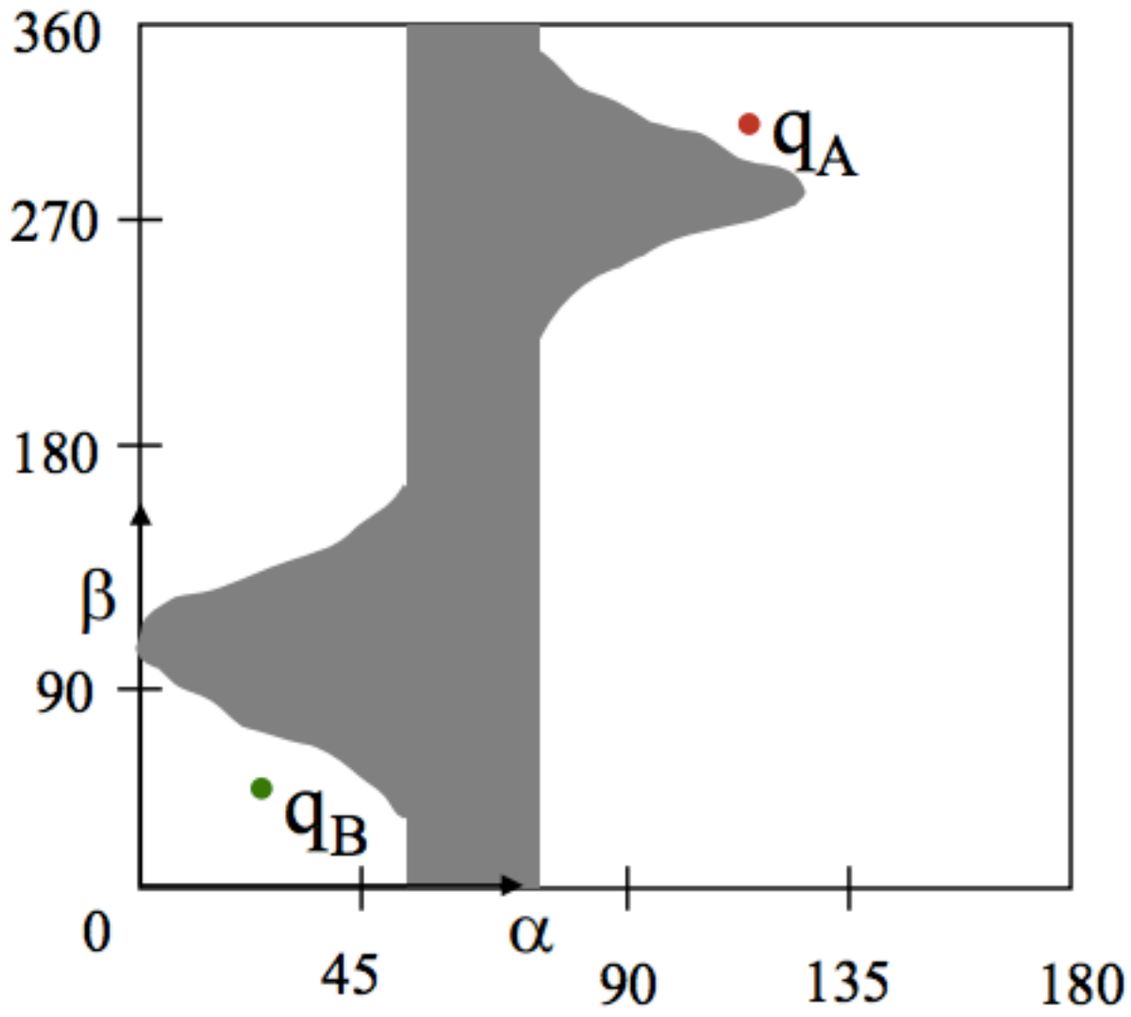
Resulting C-space representation $\mathbb{R}^2 \times S^1$, where configuration vector is $q = (x, y, \theta)$

Construction of C-free: multi-link manipulator

The construction of the free C-space can be very difficult for complex shapes and cluttered environments



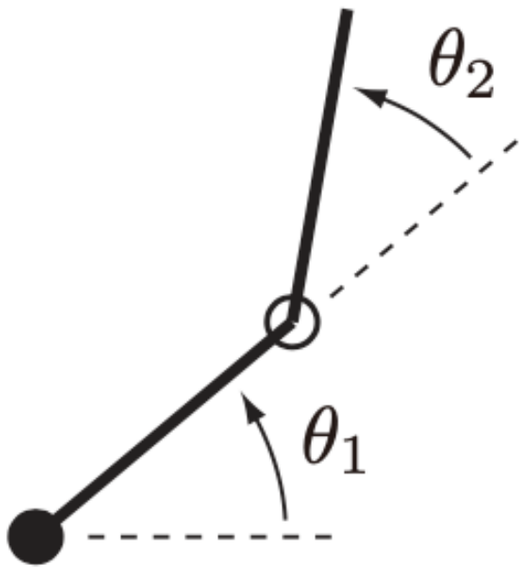
How the two-link robot arm can move from A to B based on the presence of the grey obstacle?

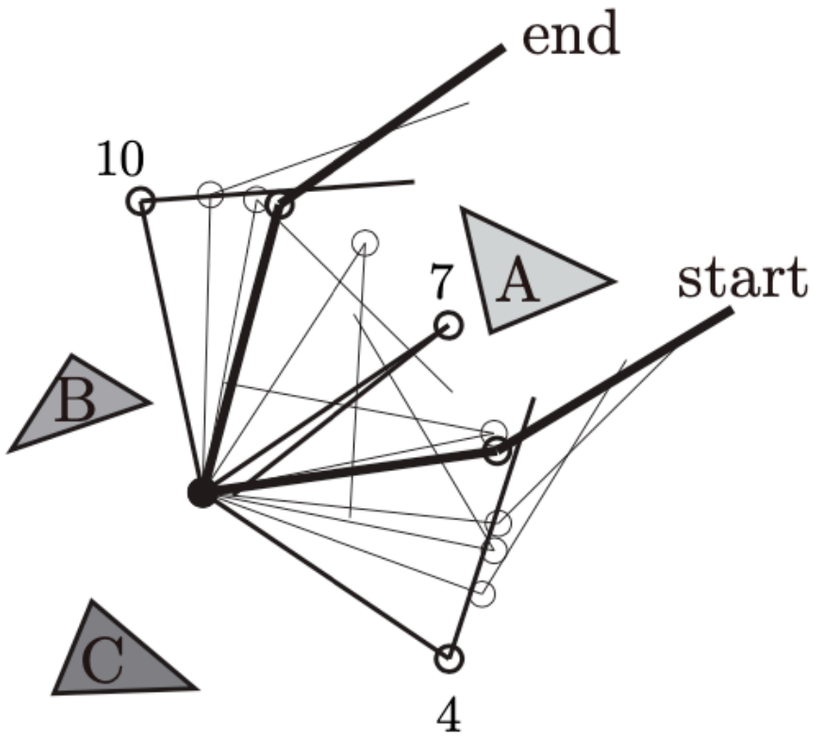


Resulting C-space representation of the obstacle for the robot arm

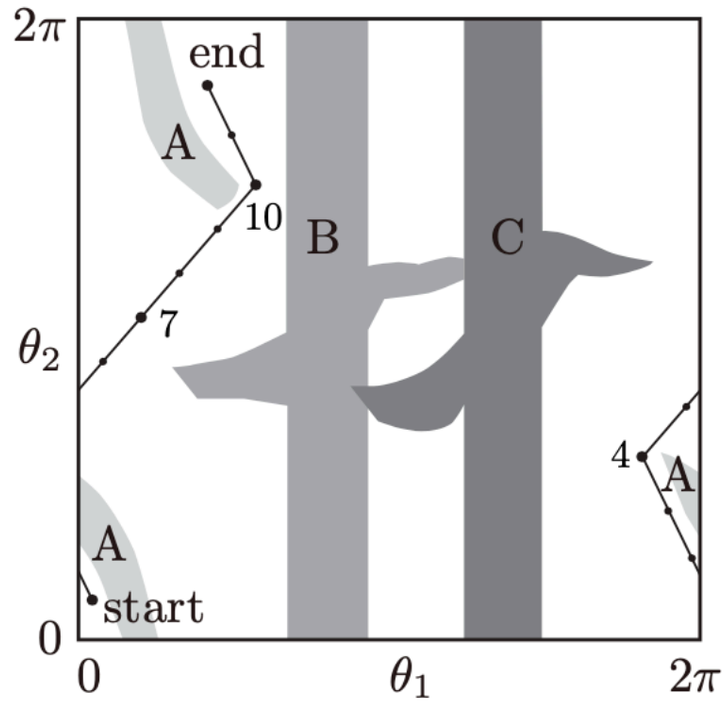
Construction of C-free: multi-link manipulator

Simple(st) arm robot: 2-link planar arm with revolute joints





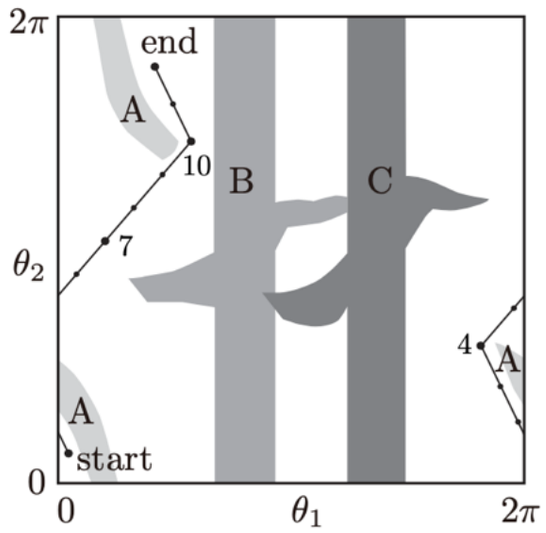
Path from start to end, in workspace

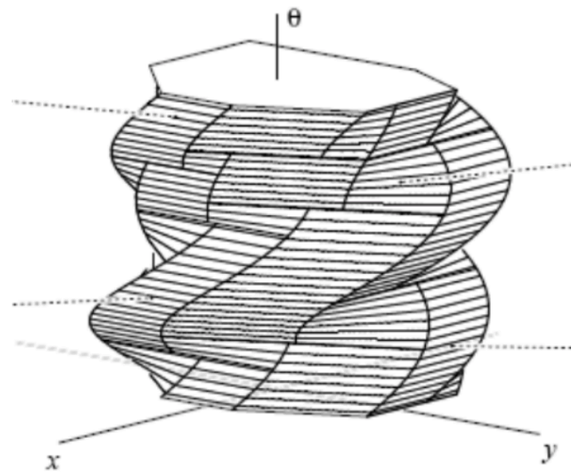


Path from start to end, in configuration space (3 intermediate points are shown)

- Note that the obstacles break $\mathcal{C}_{free}$ into three connected components: for a path to exist, the query pair must lie inside the same connected component

Construction of C-free: impractical





- Deterministic algorithms exist for automatic calculation of configuration space in presence of obstacles, but in the case of general shapes have a complexity which is exponential in the dimension of configuration space

J. Canny, PAMI, 8(2):200–209, 1986

(mathematical)

- Explicit representation of $\mathcal{C}_{free}$ is impractical to compute.

LaValle, Chapter 4

Approximation of C-space

☑ For the purpose of planning, it might be sufficient to be able to efficiently compute for each robot configuration q the (Euclidean) distance $d(q, \beta)$ to a C-obstacle β

$d(q, \beta) > 0$ no contact with the obstacle

$d(q, \beta) = 0$ surface contact with the obstacle

$d(q, \beta) < 0$ penetration into the obstacle

Distance measurement algorithm:

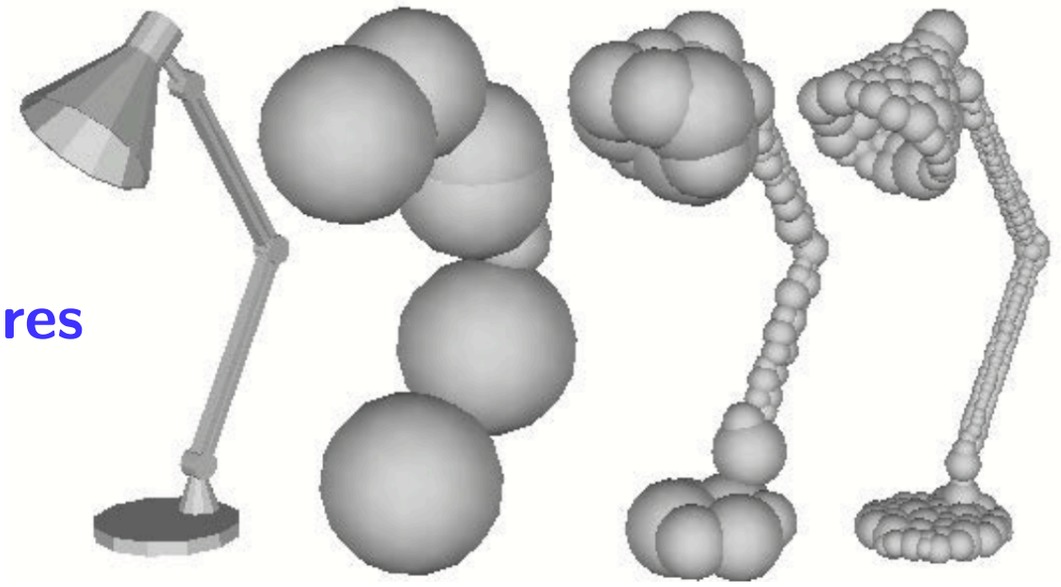
compute $d(q, \beta)$

Collision-detection routine

```
Collision_detection(q, 𝒞):
  forall β_i in 𝒞:
    if d(q, β_i) <= 0:
        return True
    else:
        return False
```

Distance measurement algorithms

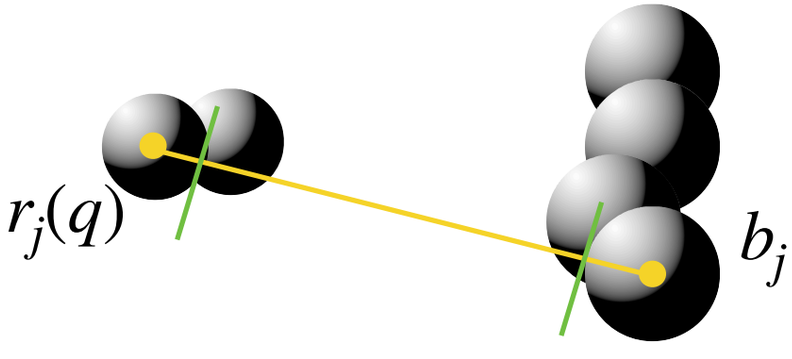
- Gilbert–Johnson–Keerthi (GJK): efficient computation of the distance between two convex bodies, possibly represented by triangular meshes.
  → Any robot or obstacle can be treated as the union of multiple convex bodies



- **Special case** (simple, efficient): approximate robot and obstacles as union of possibly overlapping **spheres**

Given a robot at q represented by k spheres of radius R_i centered at $r_i(q)$, $i = 1, \ldots, k$, and an obstacle $\mathcal{B}$ represented by ℓ spheres of radius B_j centered at b_j, $j = 1, \ldots, \ell$, the distance between the robot and the obstacle can be calculated as

$$d(q, \mathcal{B}) = \min_{i,j} \|r_i(q) - b_j\| - R_i - B_j.$$

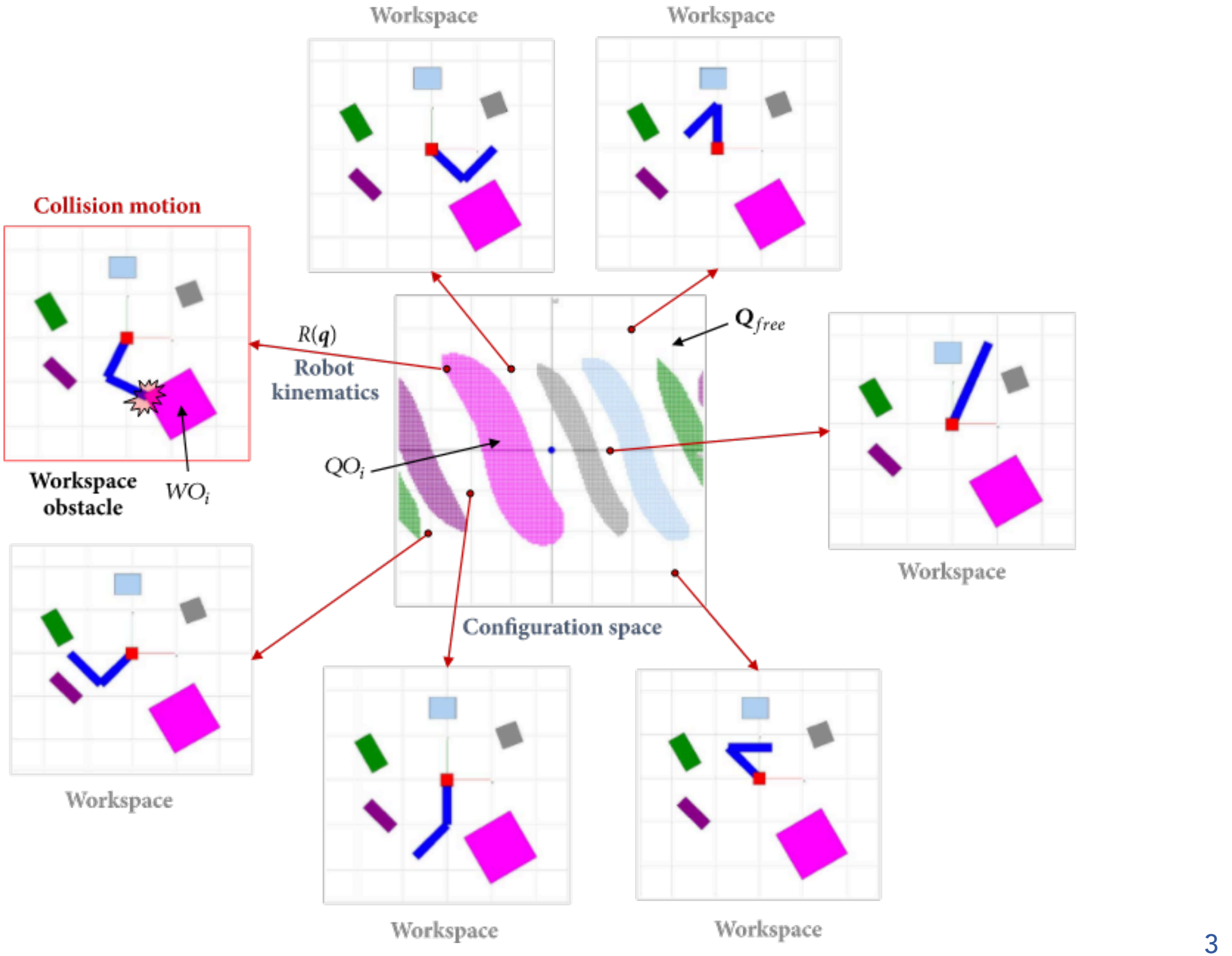


➡ Increasing the number and adapting the size of the spheres increases precision of the approximation, but also increases computation time

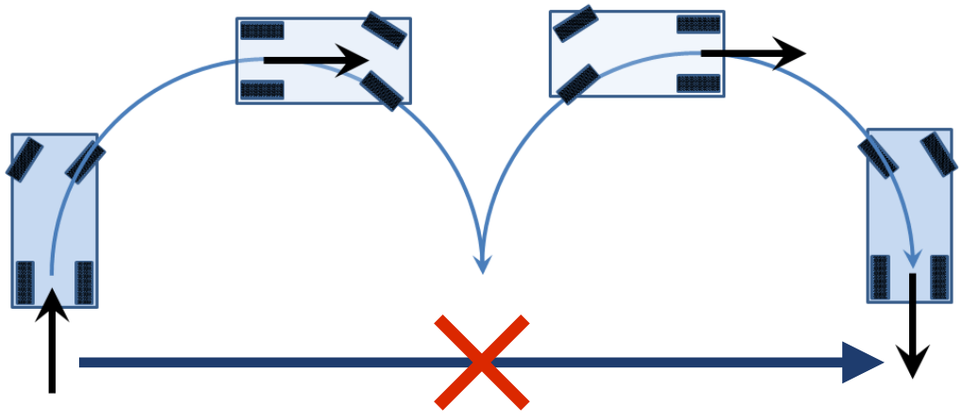
Motion planning: Core Challenges

- Constructing an exact or precise approximation of C-space
- C-space is a potentially highly dimensional, continuous topological space
- Completeness: find a path if it exists, or report a failure
- Optimality vs. the selected metric
- Non-holonomic robots, differential constraints

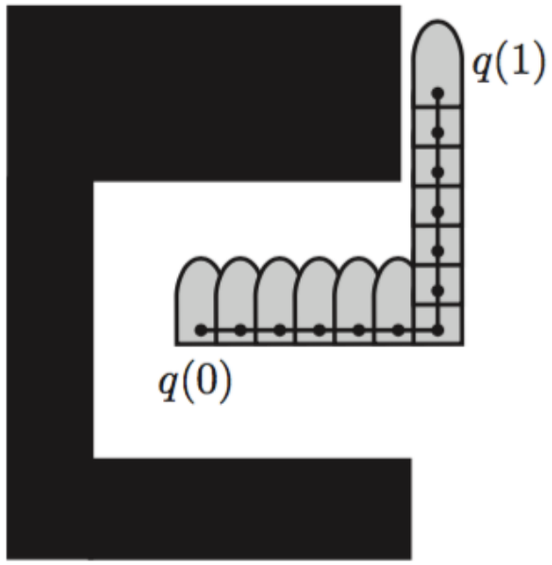
Complexity of C-space



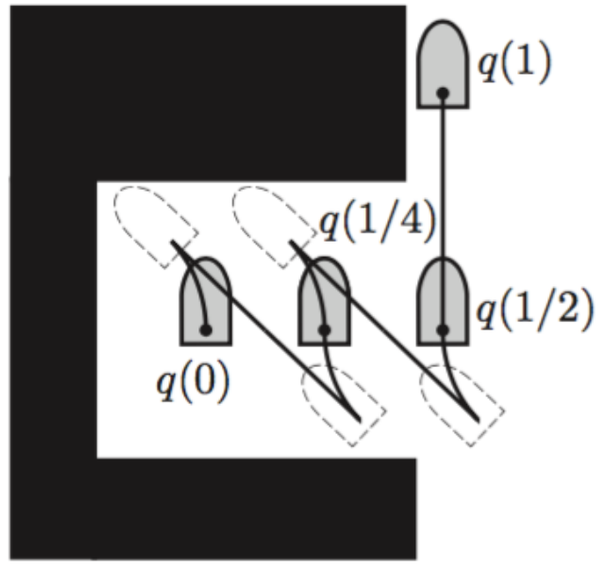
Non-holonomic path planning



Two-moves car parking with forward and backward motion capabilities but no no side-way motion



No constraints on the maximum turning angle



Constraints on the maximum turning angle

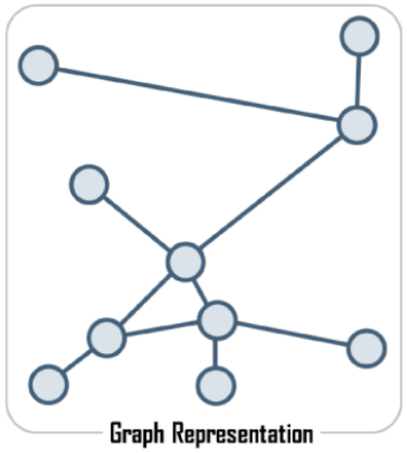
Non Holonomic path planning is a difficult problem!

Let's start with holonomic robots ...

Motion planning: Discretization of C-space

✓ **Discretization** of C-space

Combinatorial problems, Sampling problems

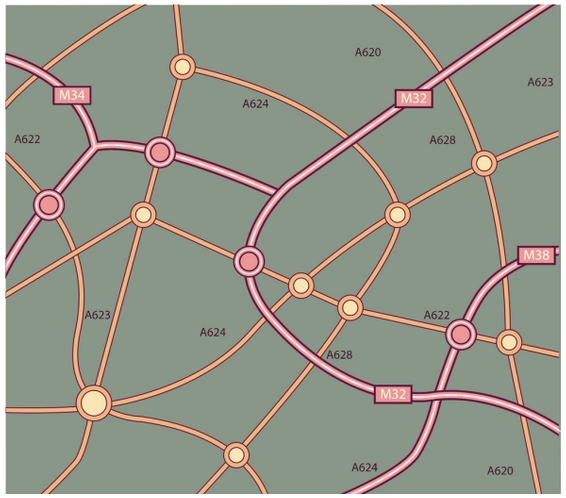


- **Roadmaps** ⇢
  - **Grids**
  - **Spatial decomposition**

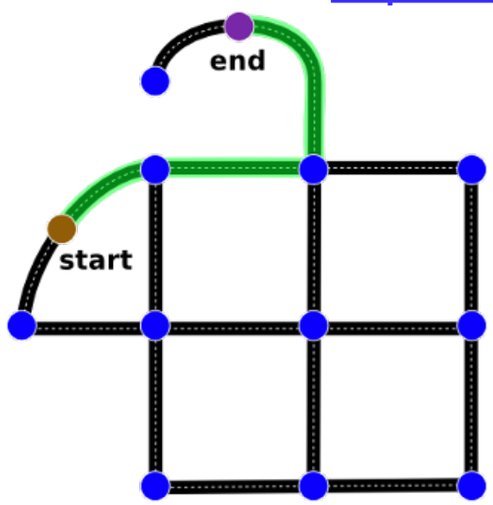
- **Graph search techniques**

  Dijkstra, A*, D*, BFS, DFS, Gradients

Roadmaps

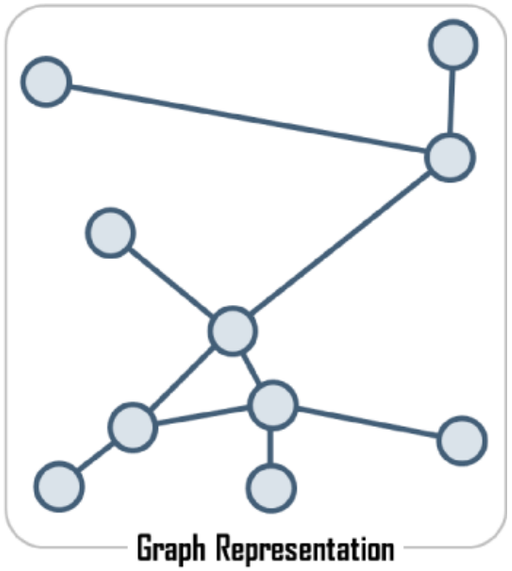


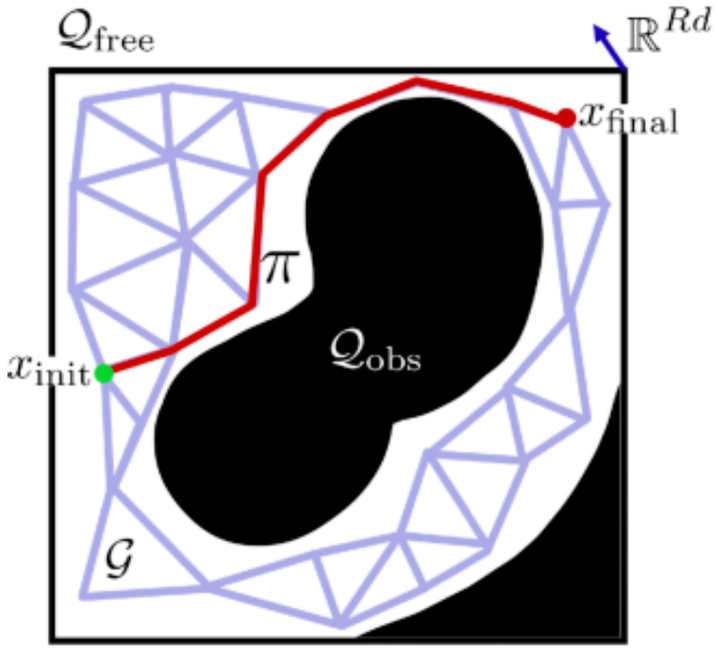
- Roadmaps are compact representations of the space freely accessible for motion
- Staying on the roads guarantees avoiding (known) obstacles
- Roadmaps are not limited to answer one specific query (from A to B), but can be used to find an admissible path for any desired query (**reusability**, **pre-processing**)
- **Roadmap**: Union of curves that create a topological graph

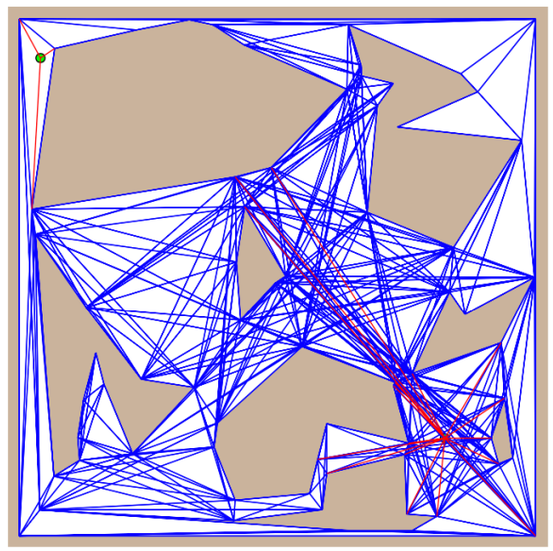


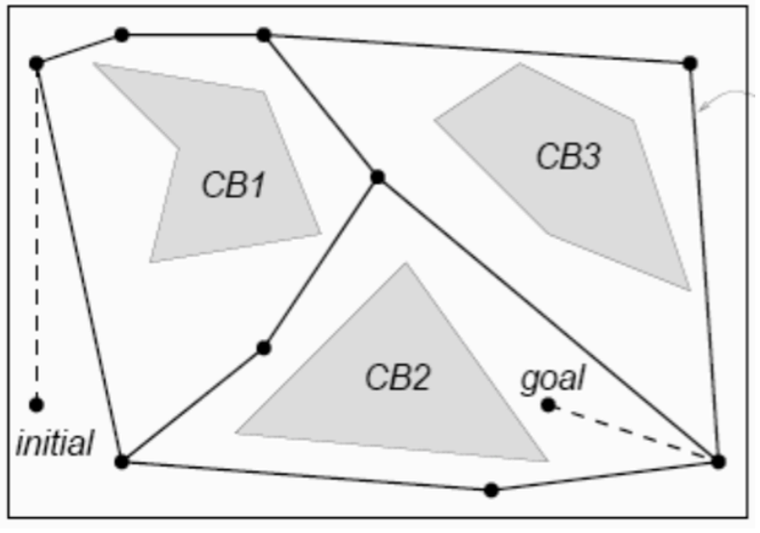
Roadmaps

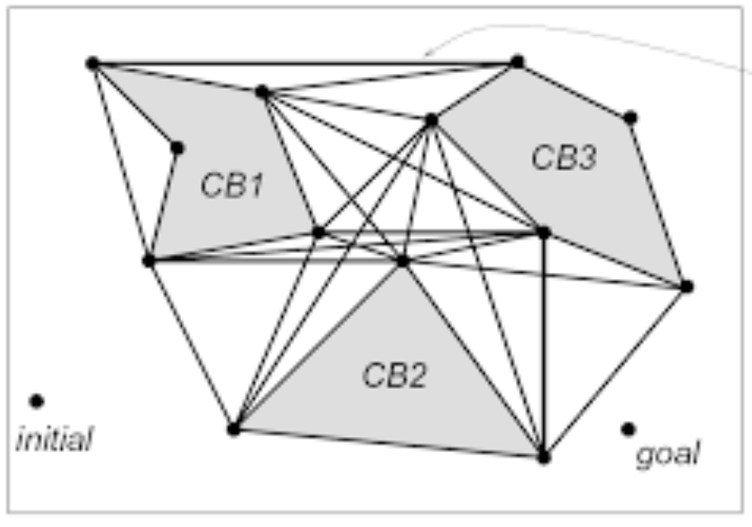
A road map is a graph in $\mathcal{C}_{free}$ in which each vertex is a configuration in $\mathcal{C}_{free}$ and each edge is a collision-free path through $\mathcal{C}_{free}$

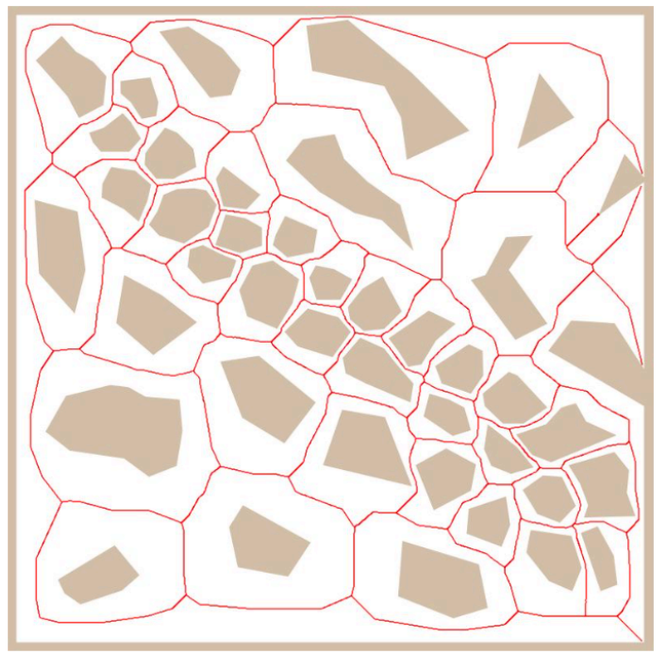




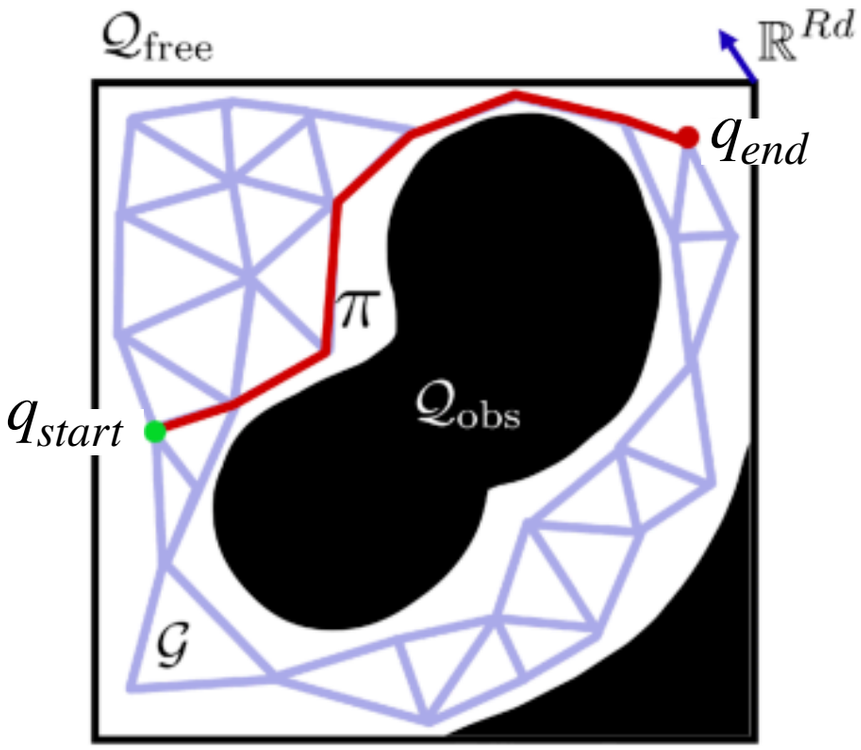


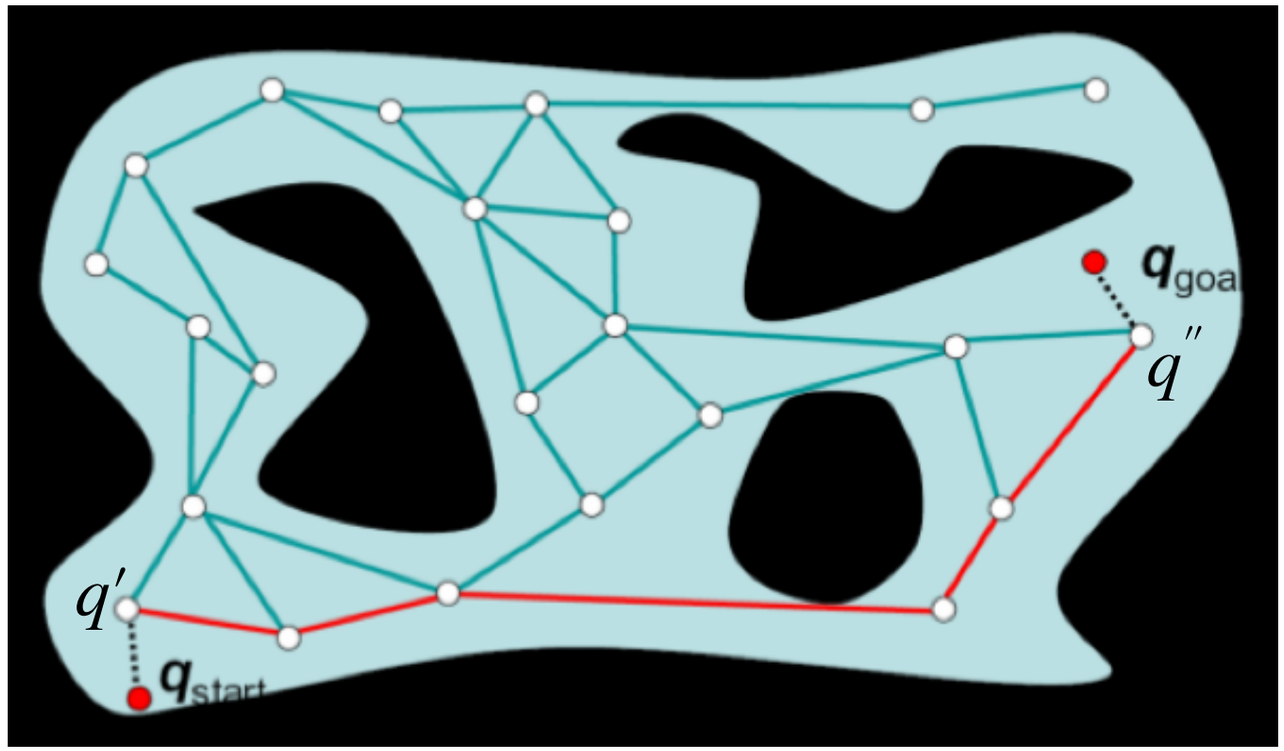






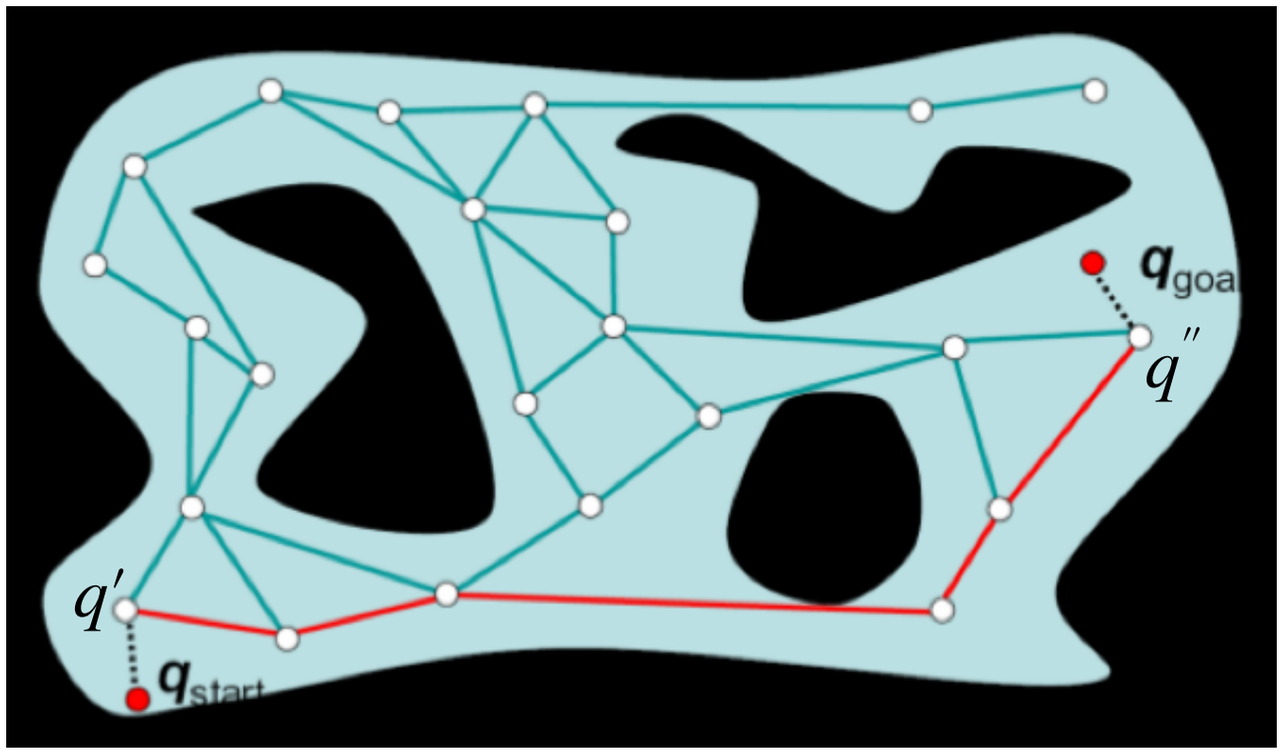
Roadmap for complete algorithms





- A roadmap, RM, is a union of curves such that for all start and goal points in Q_{free} that can be connected by a path:
  - **Accessibility:** There is a path from $q_{start} \in Q_{free}$ to some $q' \in$ RM
  - **Departability:** There is a path from some $q'' \in$ RM to $q_{goal} \in Q_{free}$
  - **Connectivity:** there exists a path in RM between q' and q''
  - **One dimensional**

Build and use a roadmap



1. Build the roadmap
   a) nodes are points in Q_{free} (or its boundary)
   b) two nodes are connected by an edge if there is a free path between them

2. Connect start end goal points to the road map
   at point q' and q'', respectively

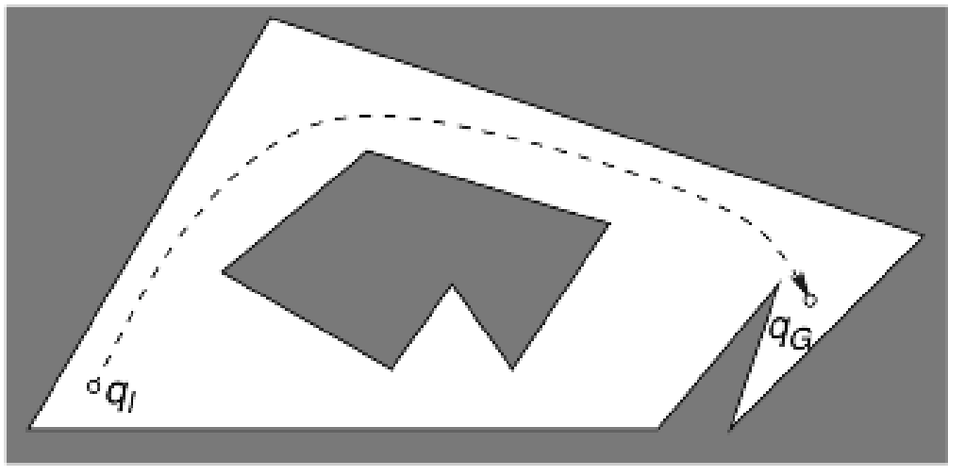
3. Connect find a path on the roadmap betwen q' and q''

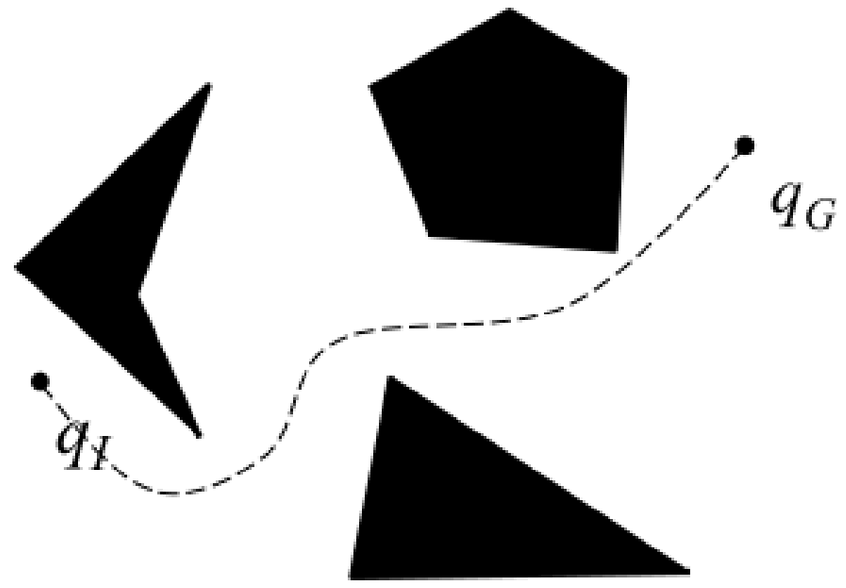
   The result is a path in Q_{free} from start to goal

   Question: what is the hard part here?

What is a good and complete roadmap?

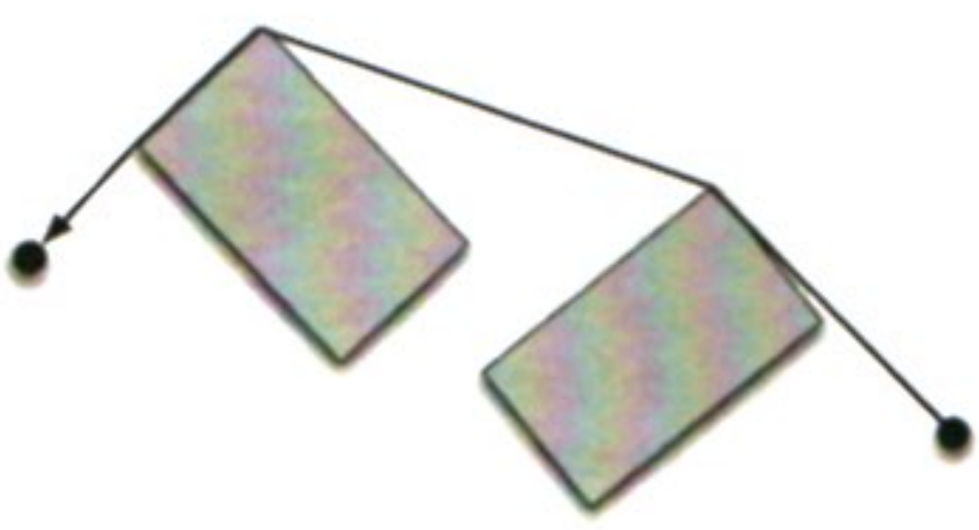
- Let's start, without (much) loss of generality (and performance), to assume to have an omnidirectional (holonomic) point robot moving in a polygonal world: $\mathcal{W} = \mathbb{R}^2$, $\mathcal{C} \subset \mathbb{R}^2$

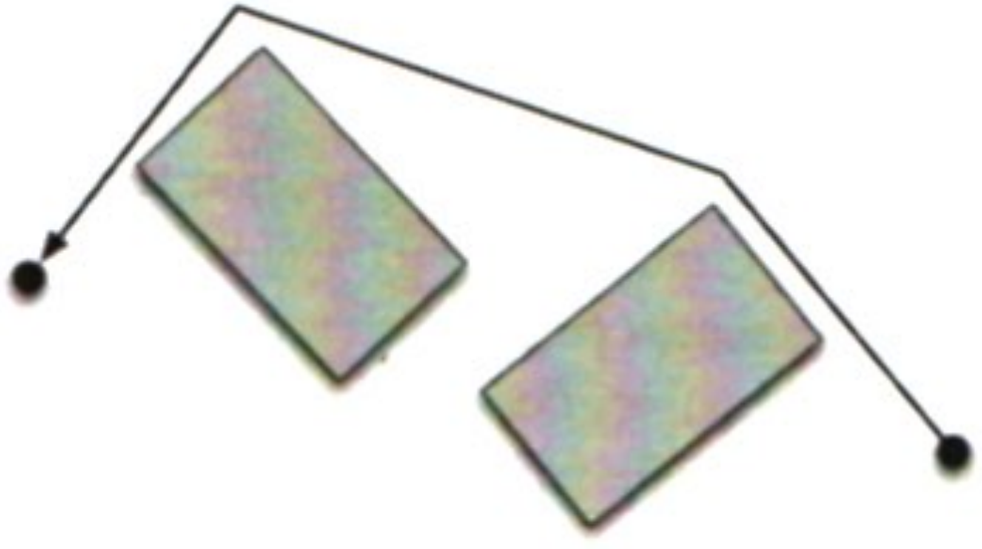




Semi-free space: touching the obstacles

- ***Semi-free paths:*** the obstacles can be touched
- The semi-free space is a *closed set*, which can be (also) useful to prove **optimality** (the free space is an open set, since obstacles' frontier does not belong to the set)

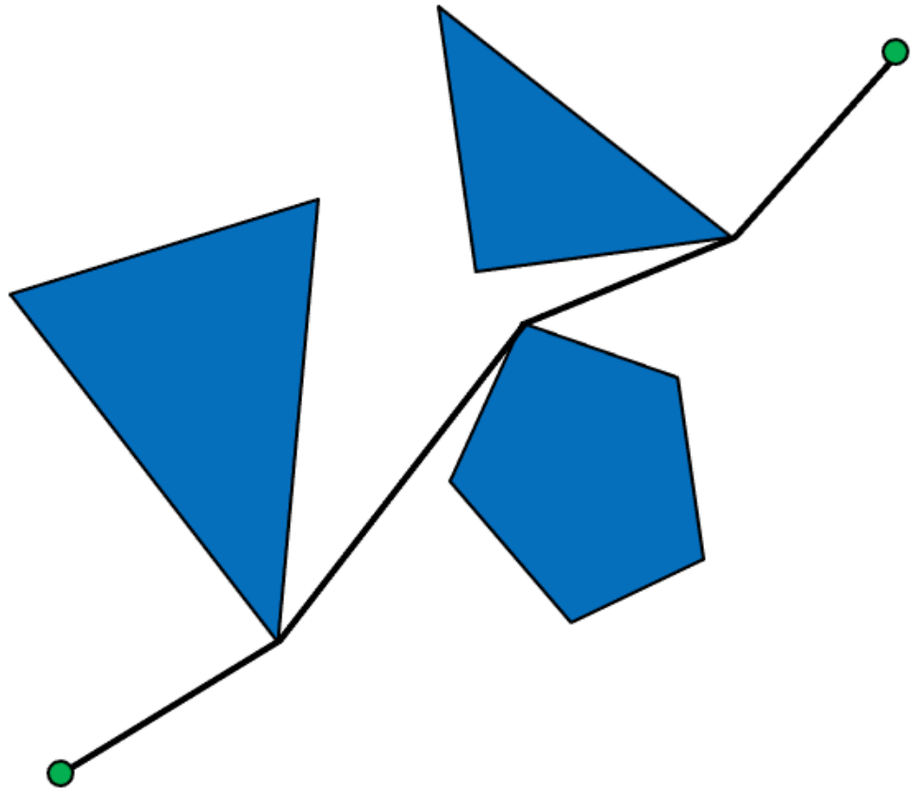




Optimal shortest path?

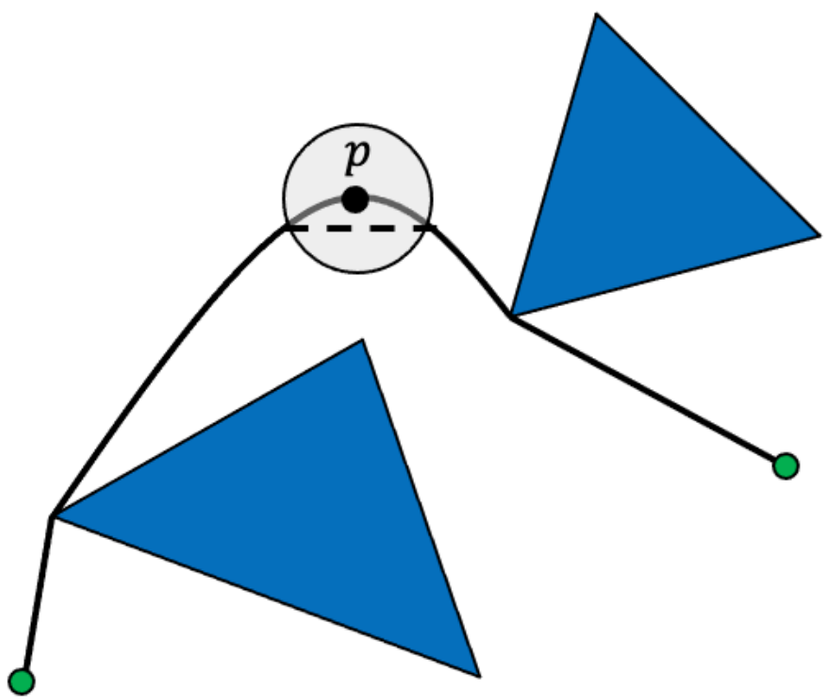
- Polygonal path: sequence of connected straight lines
- Inner vertex of polygonal path: vertex that is not beginning or end

✓Theorem: Assuming *polygonal obstacles*, a shortest path is a *polygonal path* whose inner vertices are vertices of obstacles



Optimal shortest path?

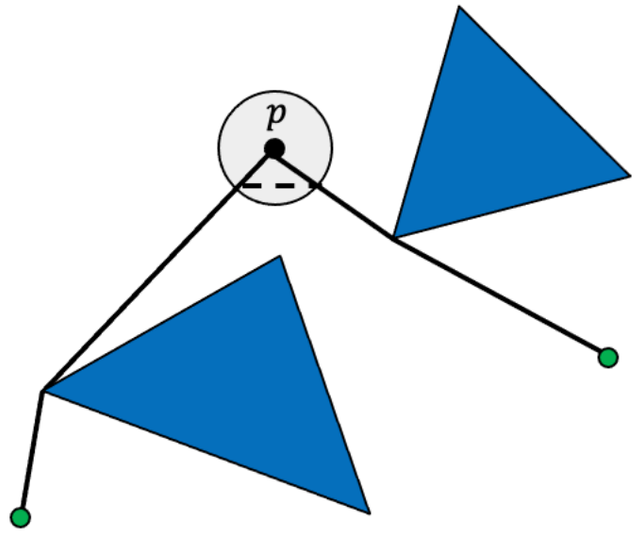
- Suppose for contradiction that shortest path is *not* polygonal
- Obstacles are polygonal ⇒ ∃ point p in interior of free space such that "(shortest) path through p is curved"
- p in free space ⇒ ∃ disc of free space around p
- Path through disc can be shortened by connecting points of entry and exit
- → *Path it's polygonal!* (also true in free space)



Optimal shortest path?

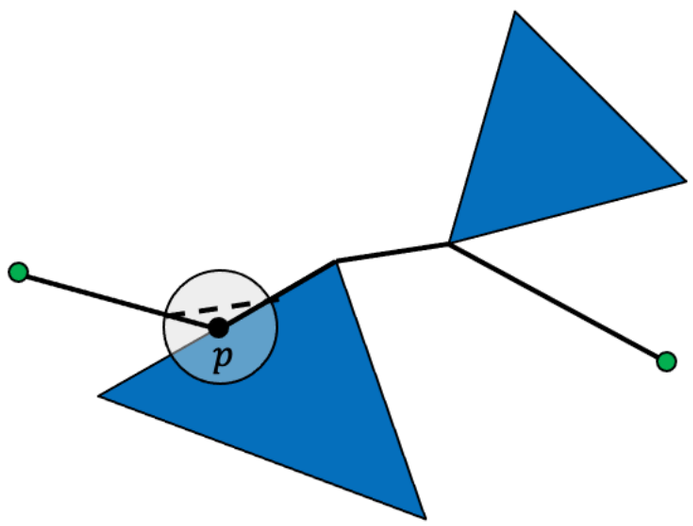
✓ Path is polygonal

- Vertex cannot lie in interior of free space, otherwise we can do the same trick and shorten the path (that would not then be the shortest)



- Vertex cannot lie on an edge, otherwise we can do the same trick

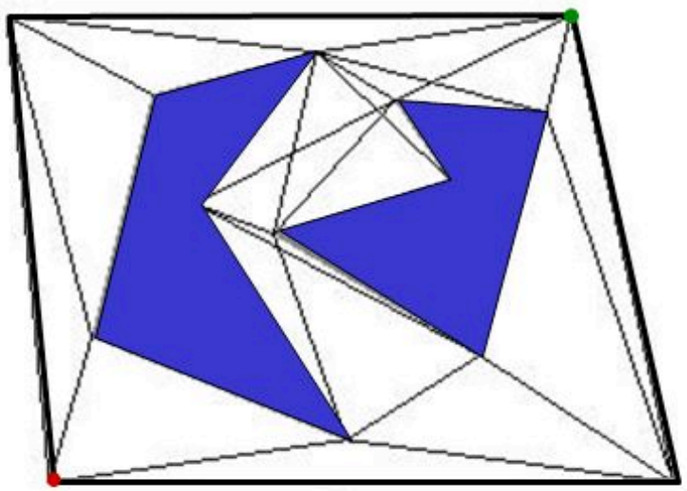
✓ Inner vertices are vertices of obstacles ■

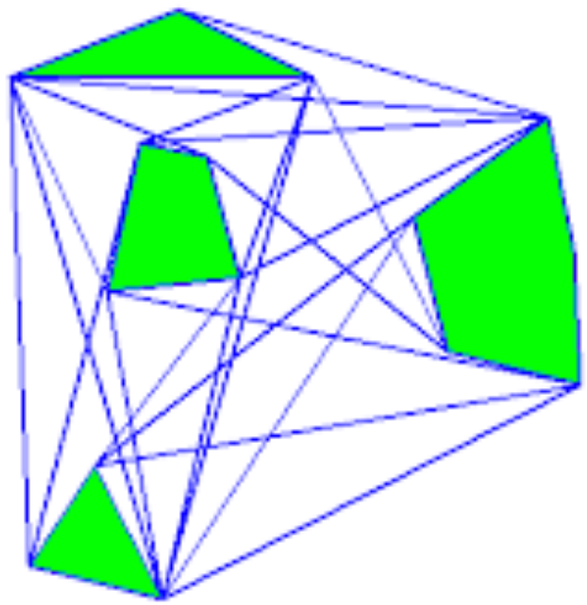


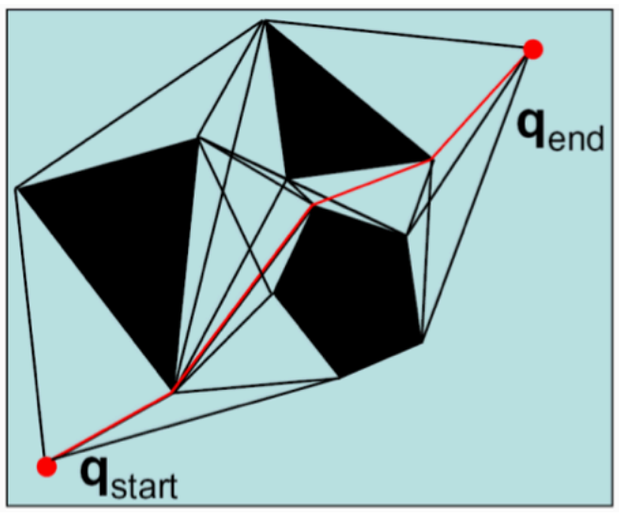
Visibility graphs

Given polygonal obstacles, a road map can be defined as visibility graph $\mathscr{V}$=(V, E):

- V = set of vertices of the polygons (in $\mathscr{C}_{\text{semi-free}}$) $\cup$ $\{q_{start}, q_{end}\}$
- E = set of unblocked (i.e., visible) line segments between the vertices in V

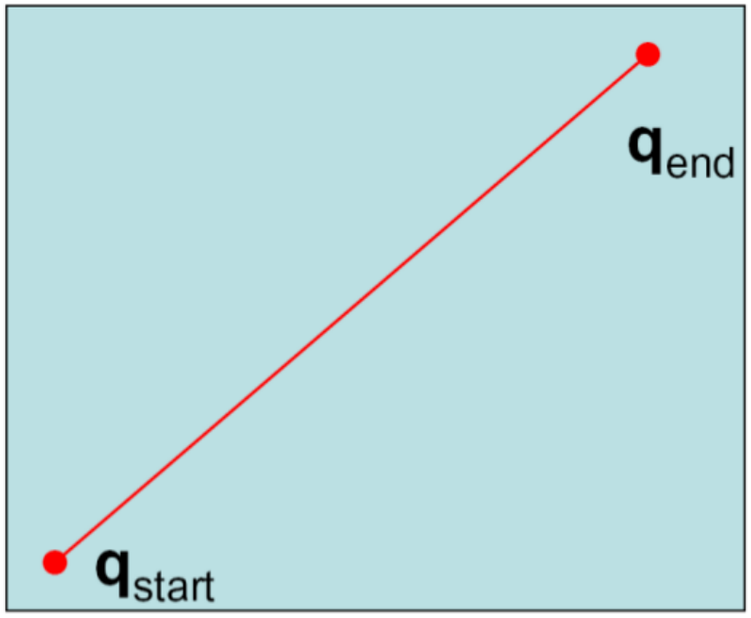




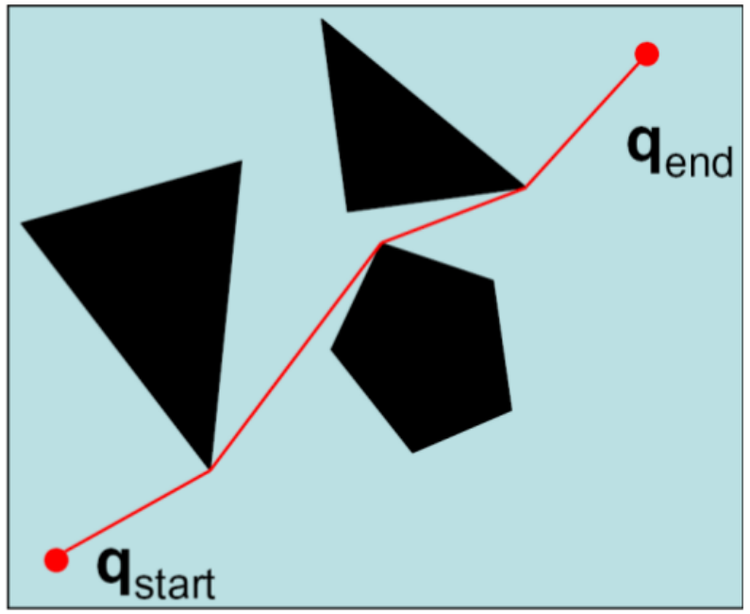


Visibility graphs

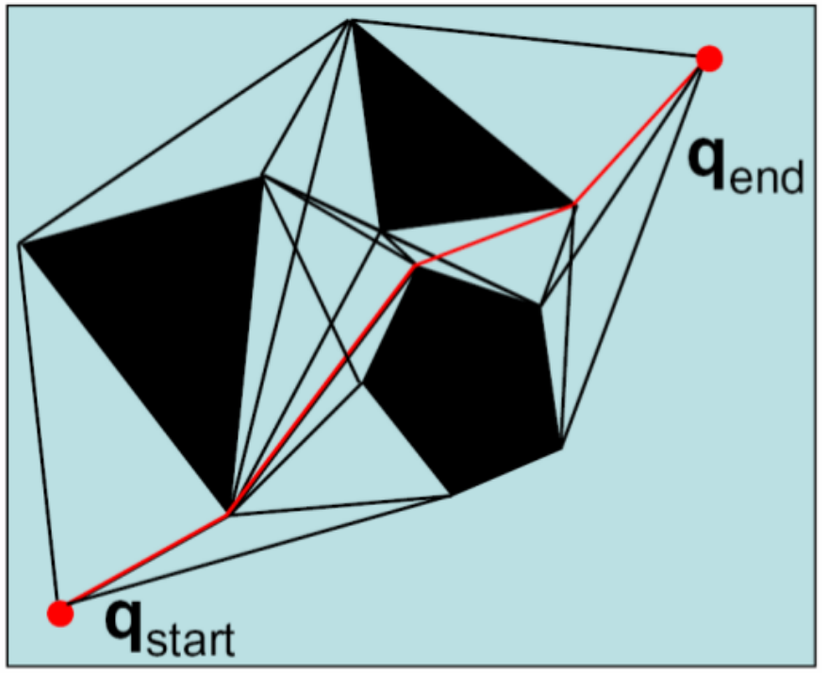
The previous theorem guarantees that the shortest path between $\boldsymbol{q}_{start}$ and $\boldsymbol{q}_{end}$ is a polygonal line connecting start and goal configuration through the vertices of the polygonal obstacles: it corresponds to the **shortest path in the visibility graph**



No obstacles, visibility graph only includes $\boldsymbol{q}_{start}$ and $\boldsymbol{q}_{end}$ and their straight line connection



Obstacles, the shortest path goes trough the vertices.

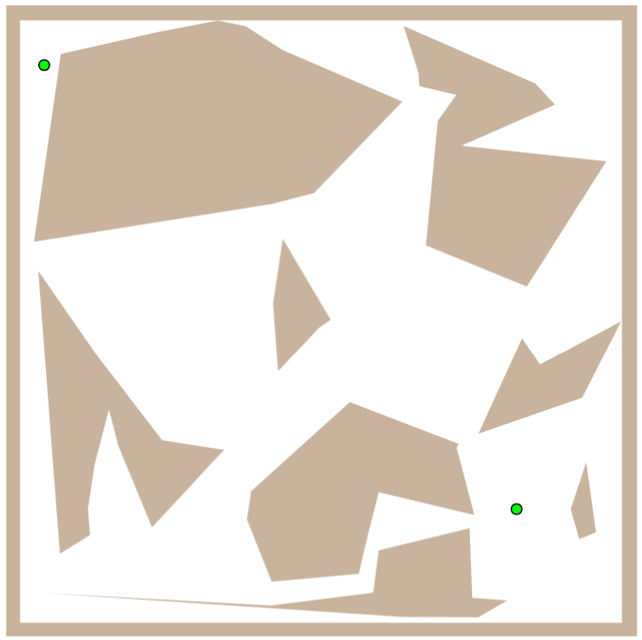


Visibility graph + shortest path.

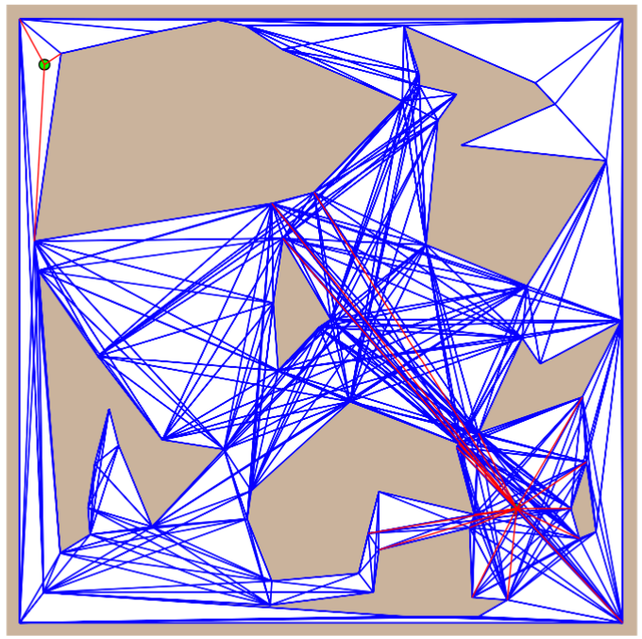
Construct and use visibility graphs

1. Compute visibility graph
2. Find the shortest path

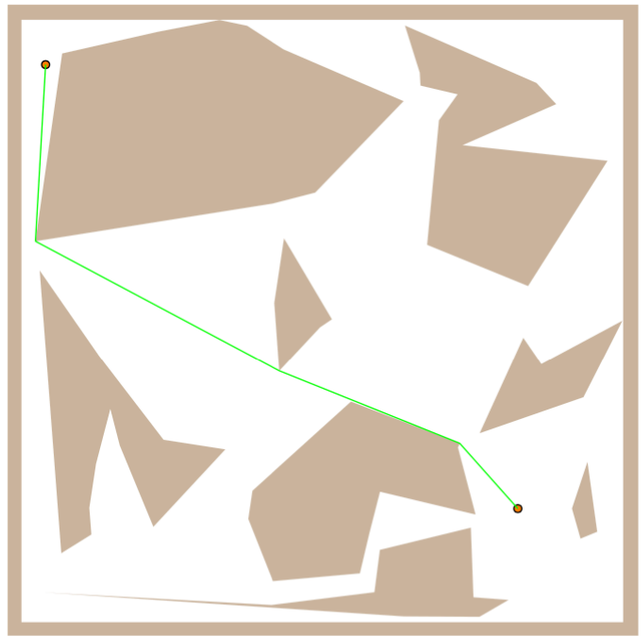
E.g., by Dijkstra's algorithm



Problem



Visibility graph

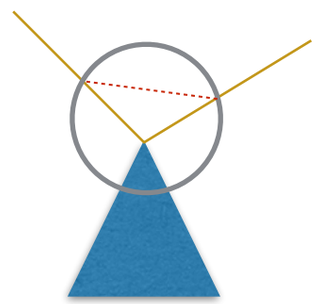


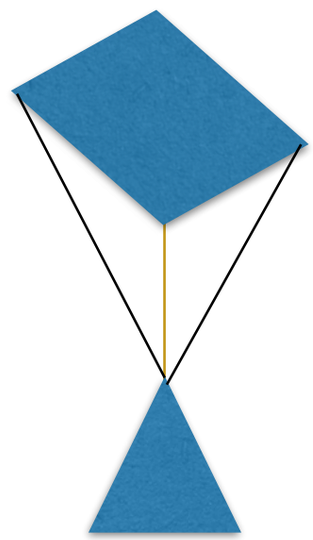
Found shortest path

Reduced Visibility graphs

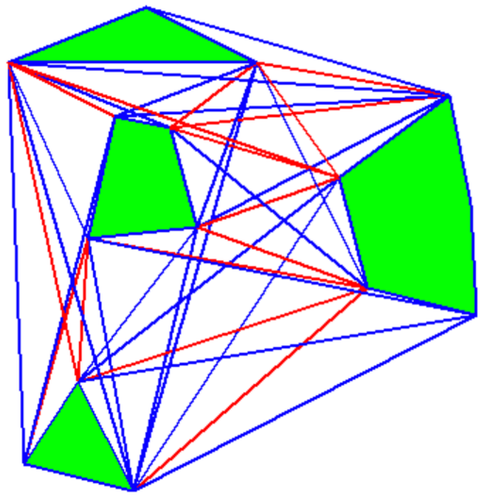
- The current graph as too many lines
  - lines to concave vertices
  - lines that "head into" the object

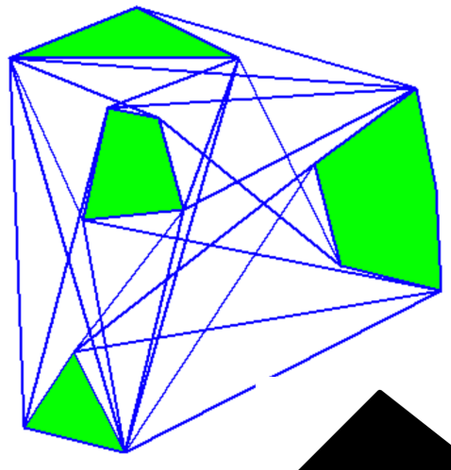
} Can't be on a shortest path

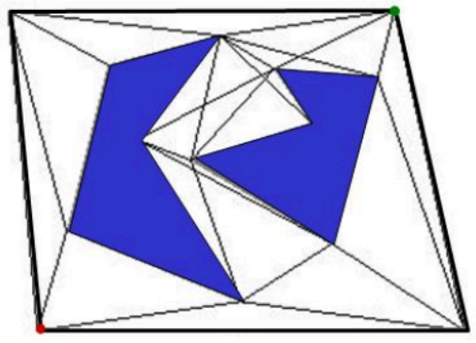


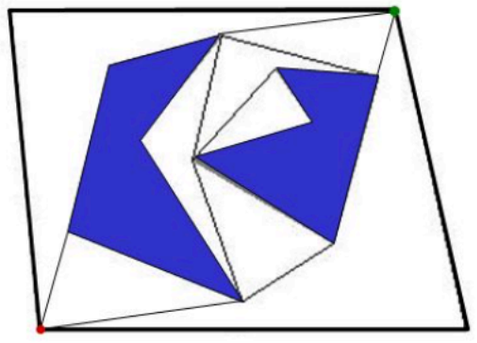


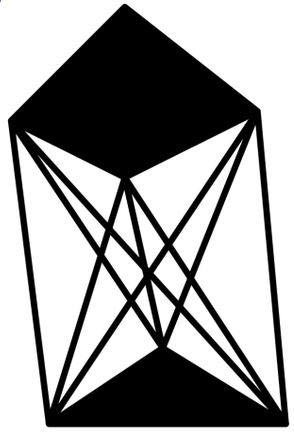
- A reduced visibility graph consists of
  - nodes that are convex
  - edges that are "tangent" (i.e. do not head into the object at either endpoint)

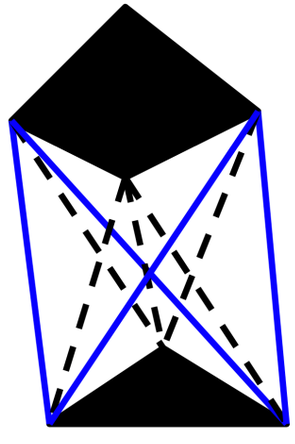








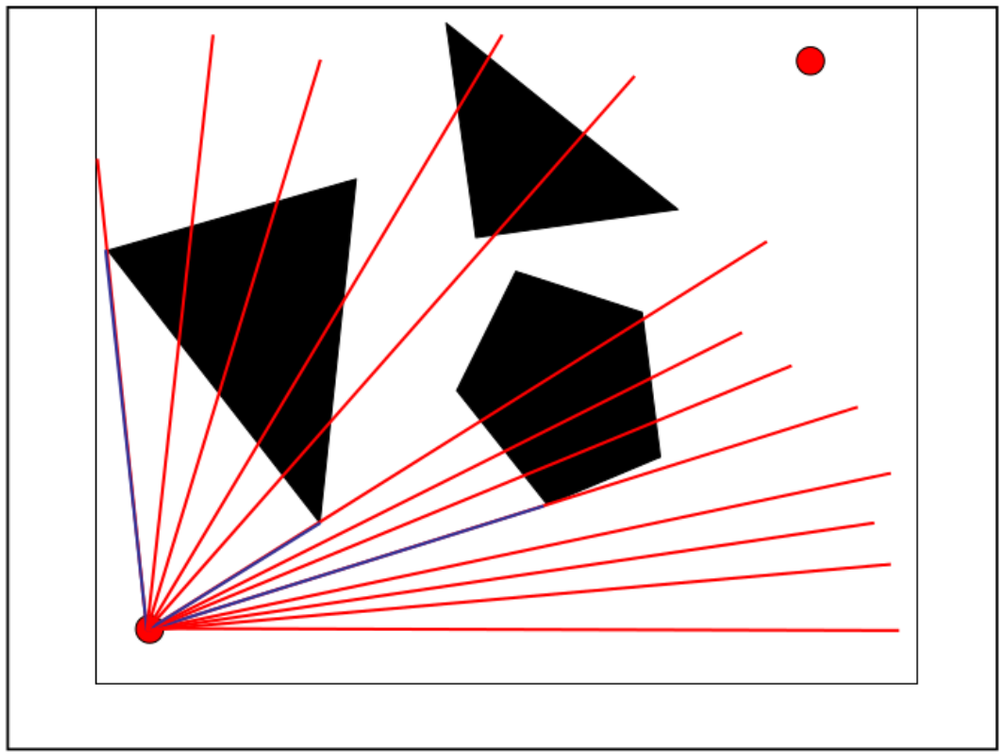


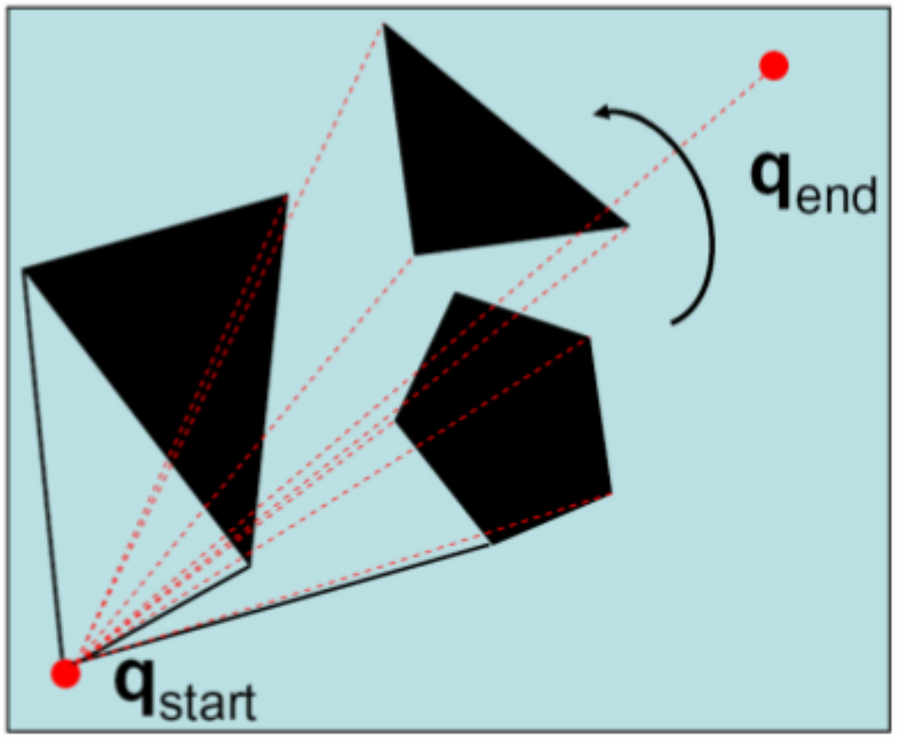


Construction of the visibility graph

- Naive approach: $\mathcal{O}(N^3)$ (N number of vertices)
- Sweep: $\mathcal{O}(N^2 \log N)$
- Optimal: $\mathcal{O}(N^2)$

Sweep: Sweep a line originating at each vertex and record those lines that end at visible vertices





Construction of the visibility graph

- Naive approach: $\mathcal{O}(N^3)$ (N number of vertices)
- Sweep: $\mathcal{O}(N^2 \log N)$
- Optimal: $\mathcal{O}(N^2)$

Naive Algorithm

Overview

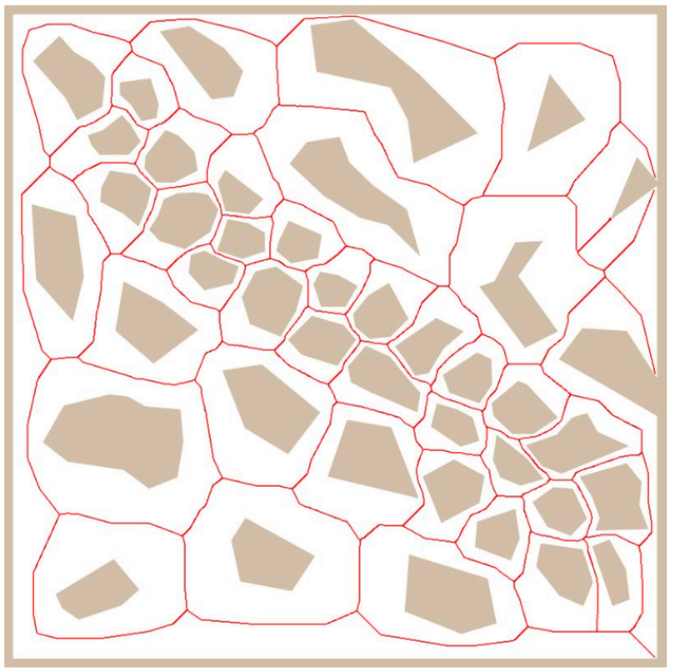
A simple solution to the problem would be to just look at every edge to see if it blocks/interferes with a given pair of vertices. If none interfere, then the two vertices are visible to each other (otherwise not). Of course, to produce the entire visibility graph, the procedure loops through every pair of vertices. The time analysis is simple also: there are (n choose 2) pairs of vertices which is $O(n^2)$ and there are $O(n)$ edges (one for every vertex) so this means the total time is $O(n^3)$. As for storage, the algorithm requires $O(n)$ working space (at least to store the input), and if the visibility graph is stored - not just reported - then it requires $O(|e|)$ memory.

Shortcoming of visibility graphs

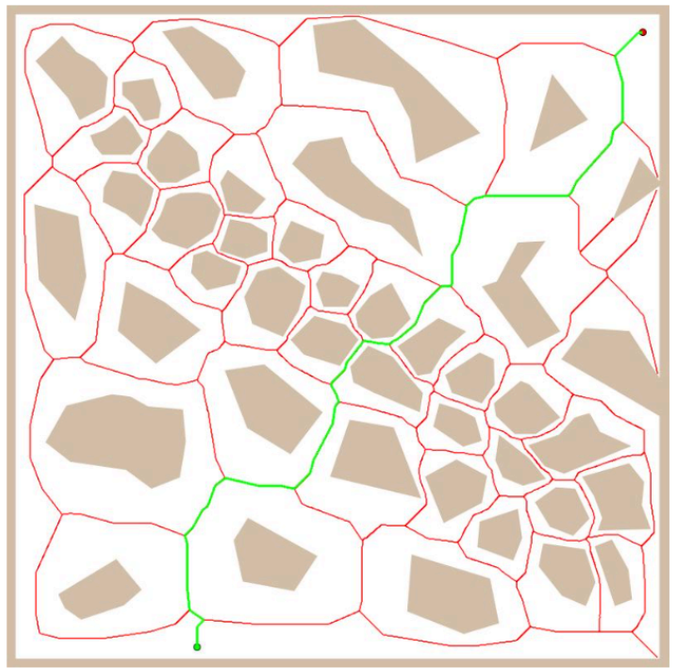
- Guarantee of shortest path but tries to stay as close as possible to obstacles, which is not precisely what we want in practice $\rightarrow$
- Any execution error will lead to a collision
- Complicated in more than 2 dimensions
- Finding a safe path (in practice) is maybe more important than strict optimality . . . $\rightarrow$

Roadmaps as Voronoi graph

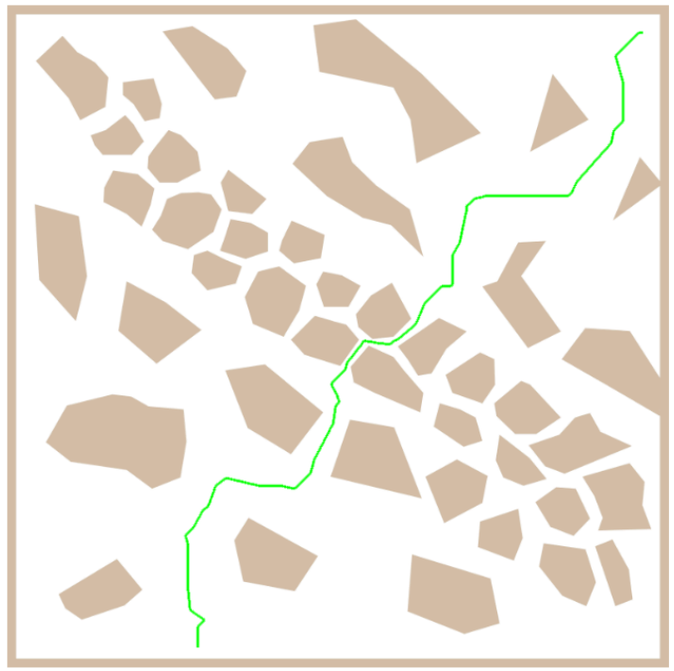
1. Roadmap is Voronoi graph that maximizes clearance from the obstacles
2. Start and goal positions are connected to the graph
3. Path is found using a graph search algorithm



Voronoi graph



Path in graph

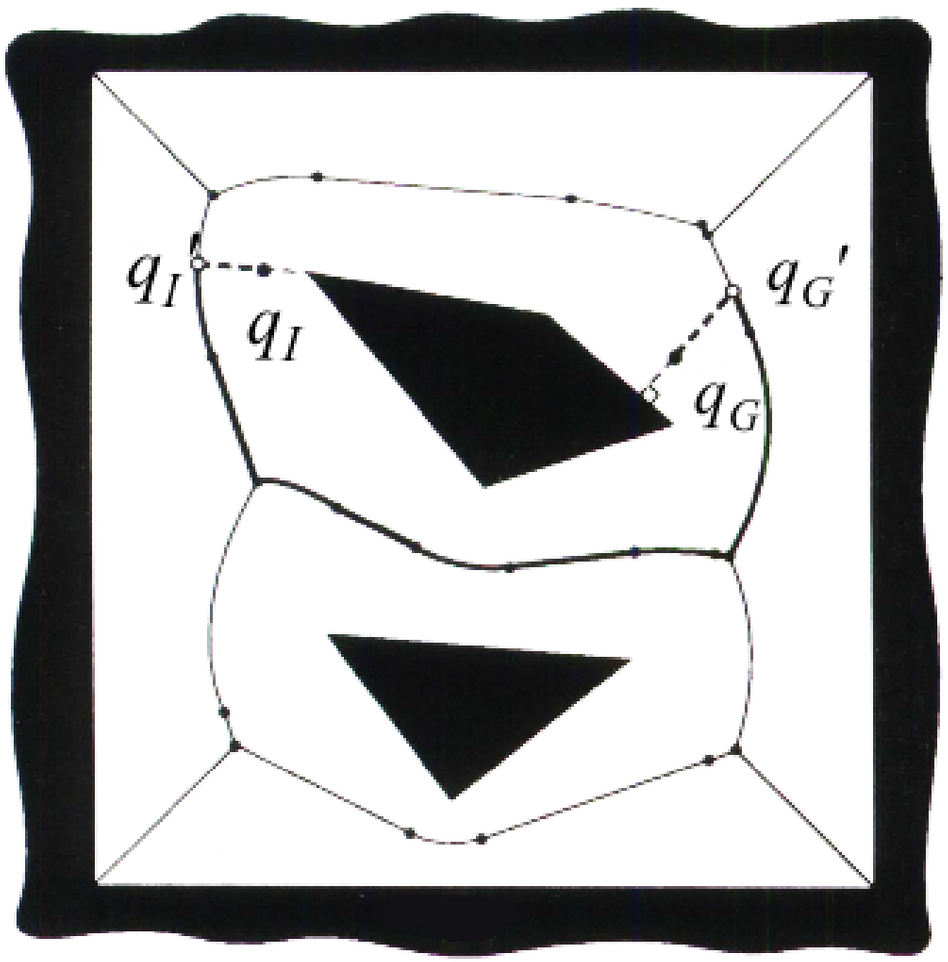


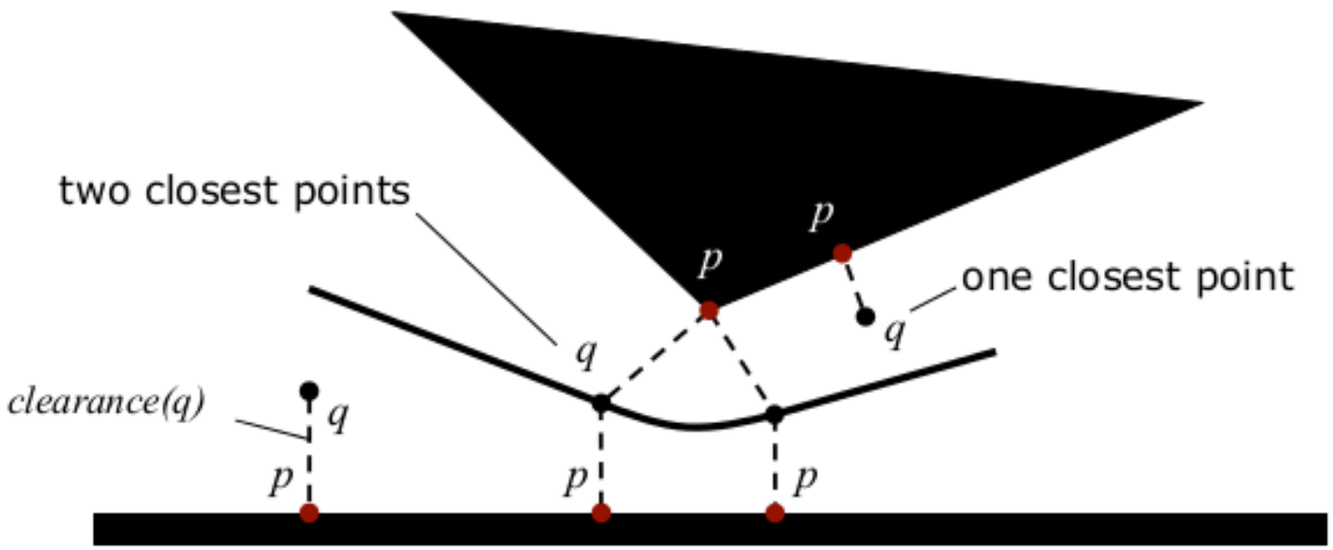
Found path

Generalized voronoi diagrams

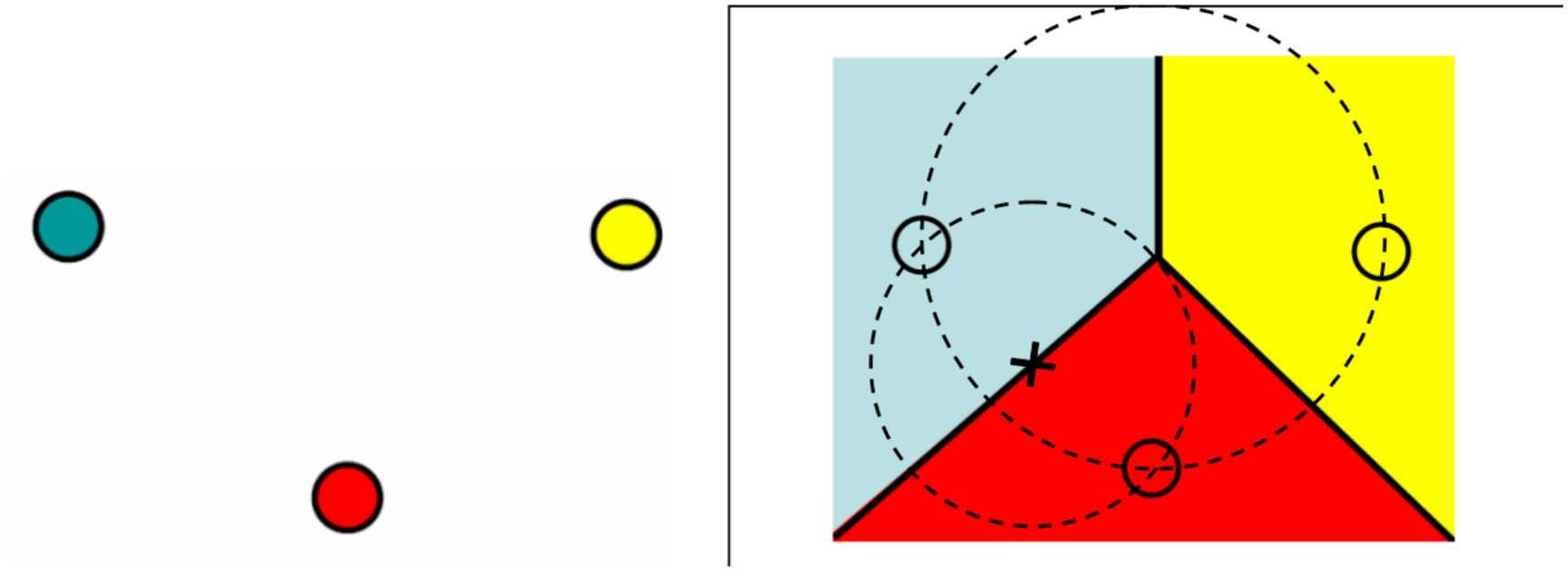
The locus of points that are equidistant from the closest two or more obstacle boundaries (in C_{obs}), including the workspace boundaries. In other words, the set of points q whose cardinality of the set of boundary points (in C_{obs}) with the same distance to q is greater than 1

The region with the same maximal clearance from all nearest obstacles



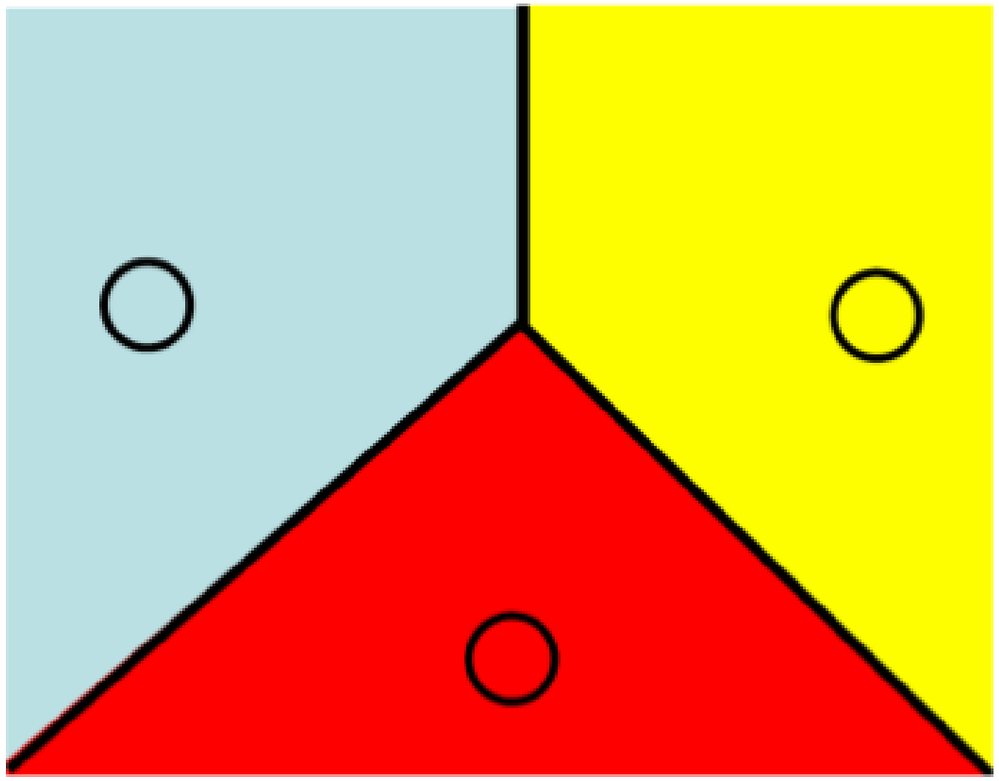


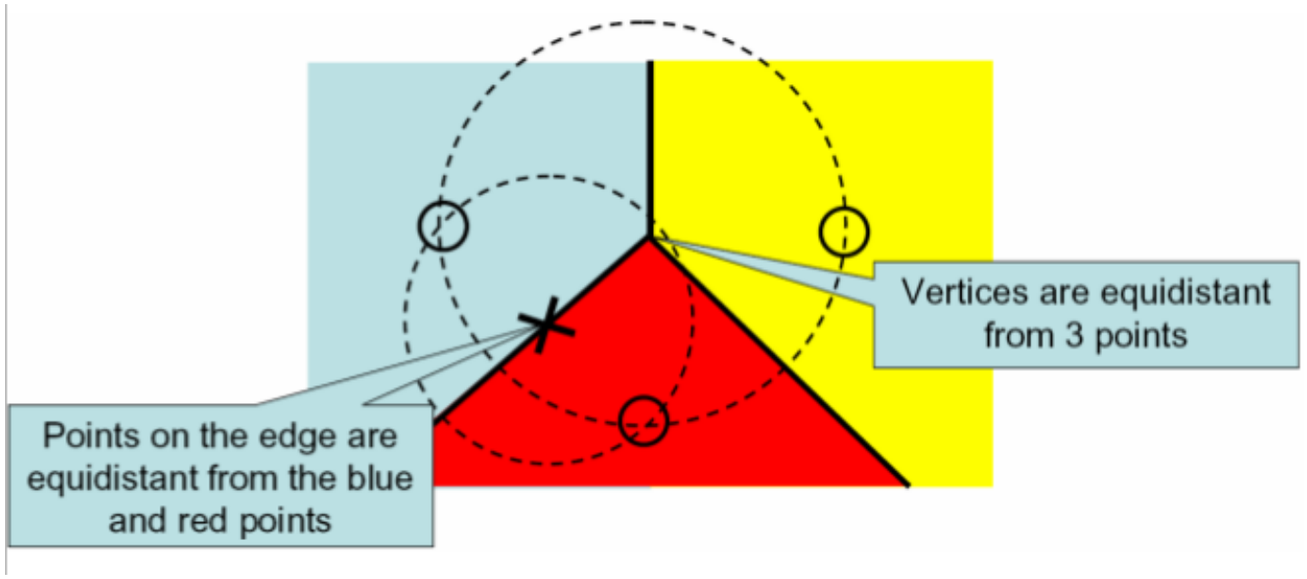
Vornonoi cells



- A set of data points, the *generators*, is given
- Each generator point defines a ***Voronoi cell*** that consists of every other point whose "distance" to the generator is less than or equal to the its distance to any other generator point (distance is well defined in Euclidean spaces, but it can be generalized)
- Each cell is obtained from the intersection of half-spaces → *Convex polygon*
- ***Voronoi diagram:*** line segments that correspond to all the points in the plane that are equidistant to the two nearest generator points
- ***Voronoi vertices:*** the points equidistant to three (or more) generators

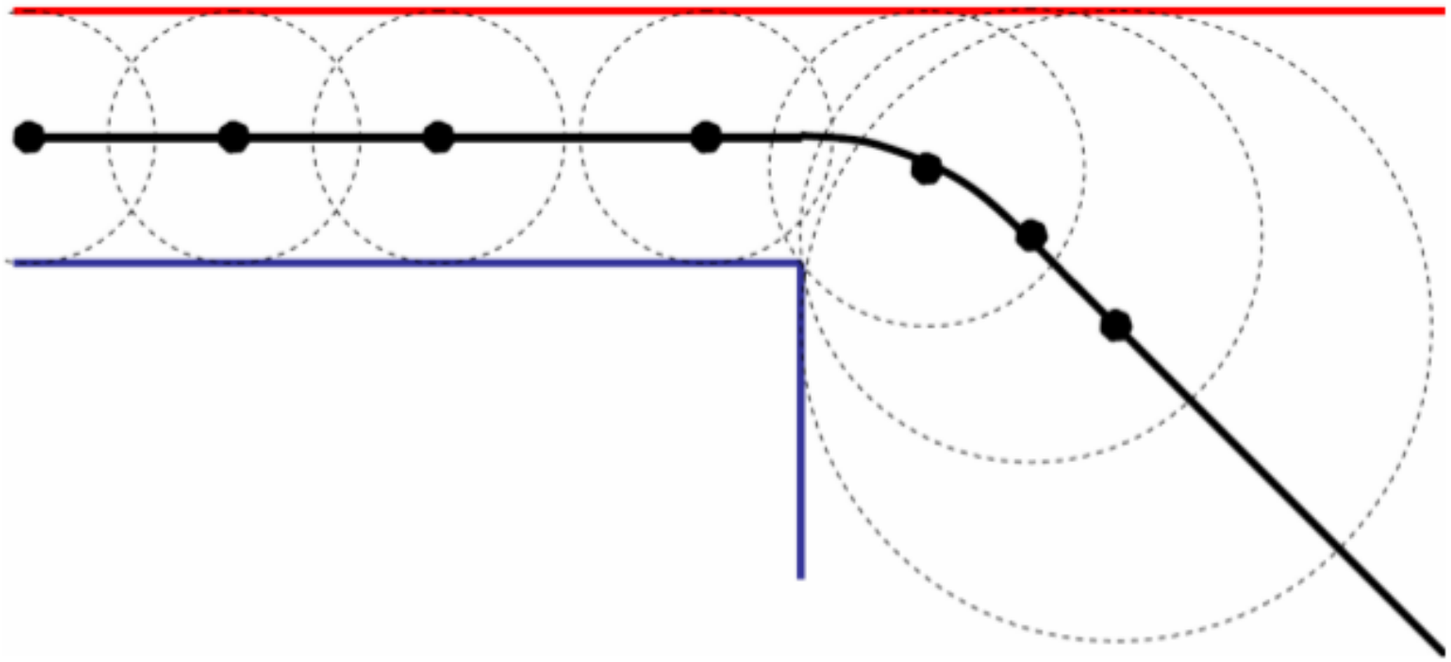
Voronoi diagrams





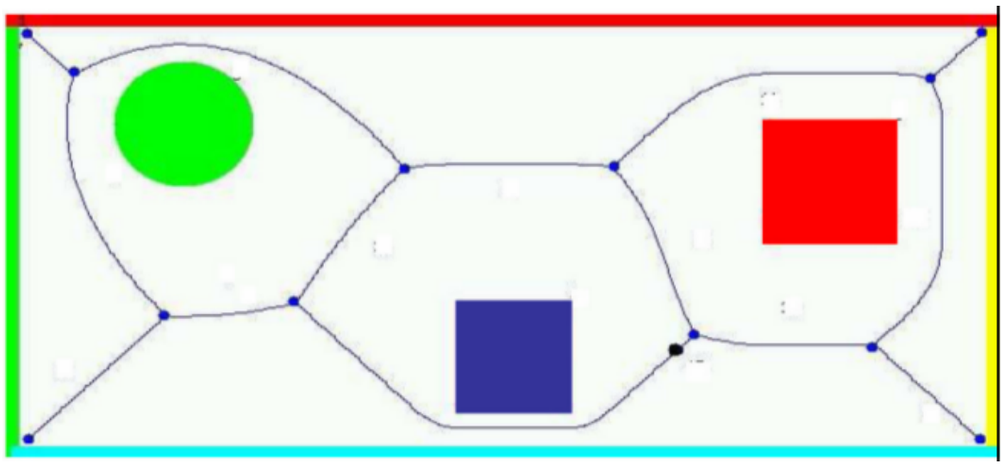
- Voronoi diagram: The set of line segments separating the regions corresponding to different colors
- Line segment: points equidistant from 2 data points
- Vertices: points equidistant from > 2 data points
- Complexity (in the plane): $\mathcal{O}(N \log N)$ in time, $\mathcal{O}(N)$ in space

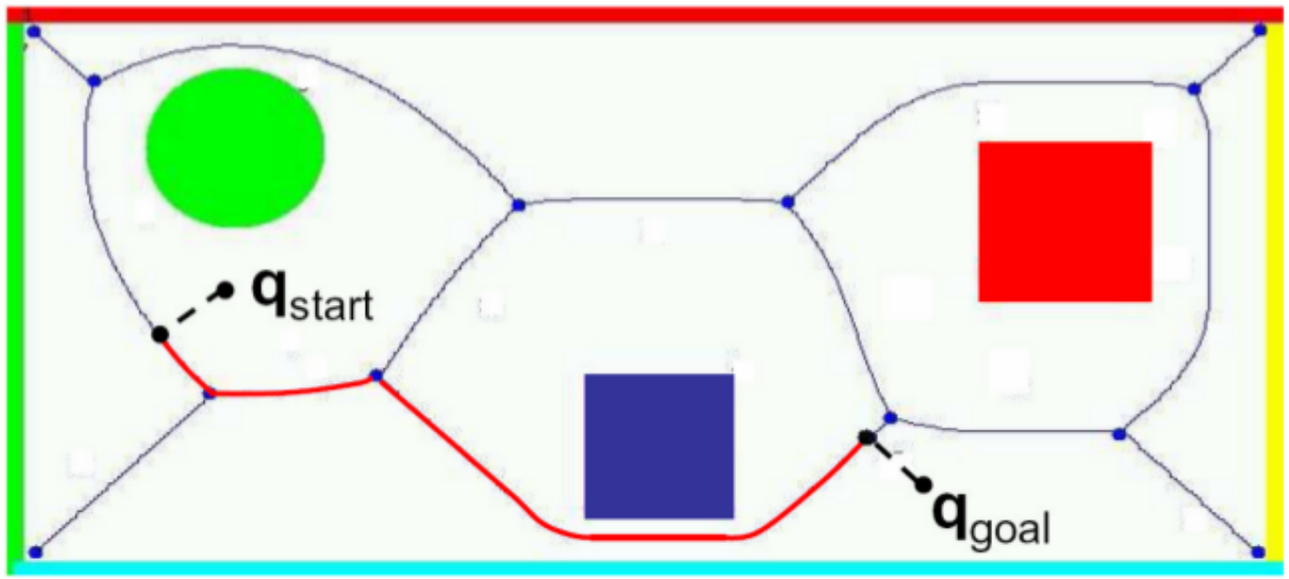
Voronoi diagrams for cluttered environments



- Edges are combinations of straight line segments and segments of quadratic curves
- Straight edges: Points equidistant from 2 lines
- Curved edges: Points equidistant from one corner and one line
- At any point on the Voronoi diagram, the distance to nearby obstacles cannot be increased by any (differential) motion local to the diagram

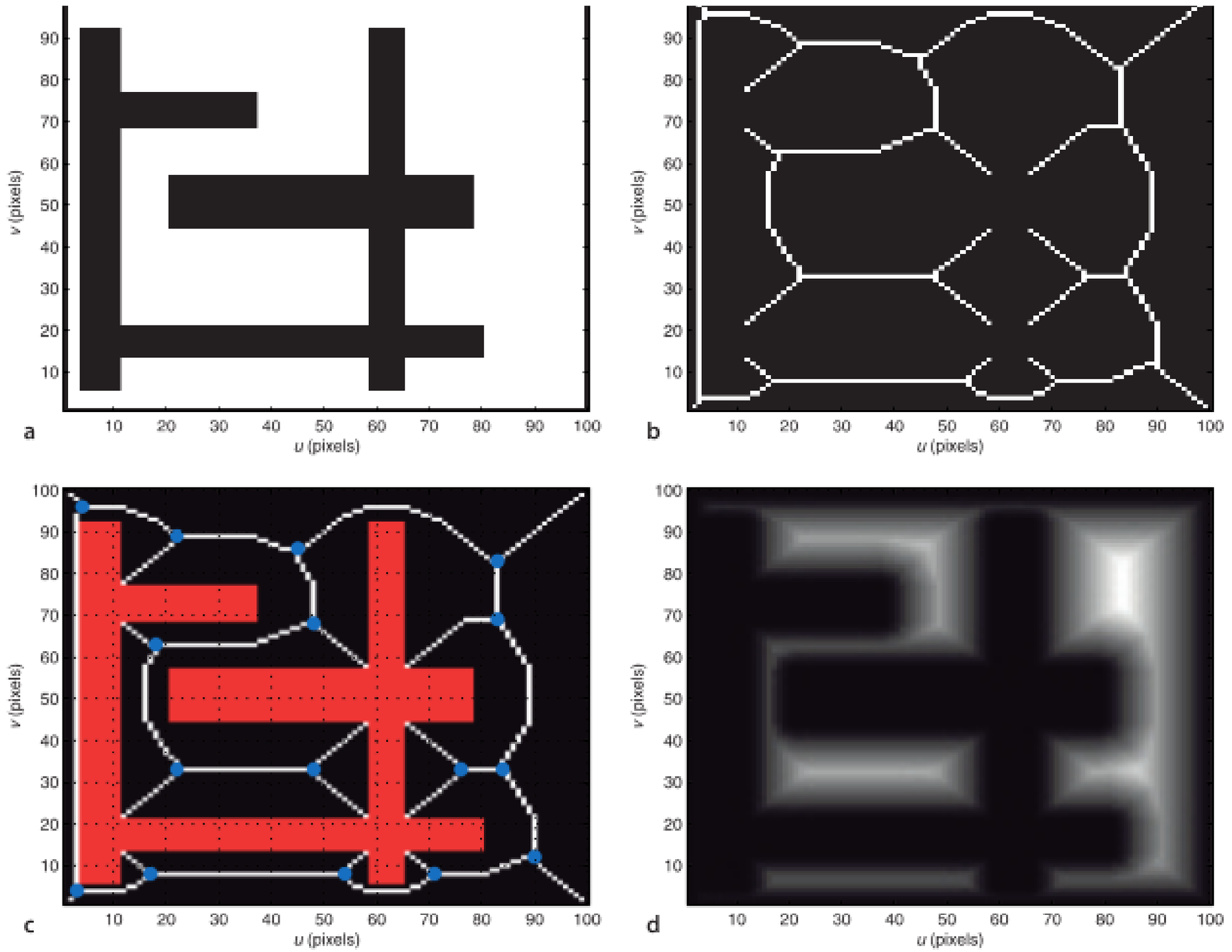
Planning in generalized Voronoi diagrams





- Find the closest points on the Voronoi skeleton to the desired start and goal points
- Compute the shortest path on the Voronoi graph

Planning in generalized Voronoi graphs

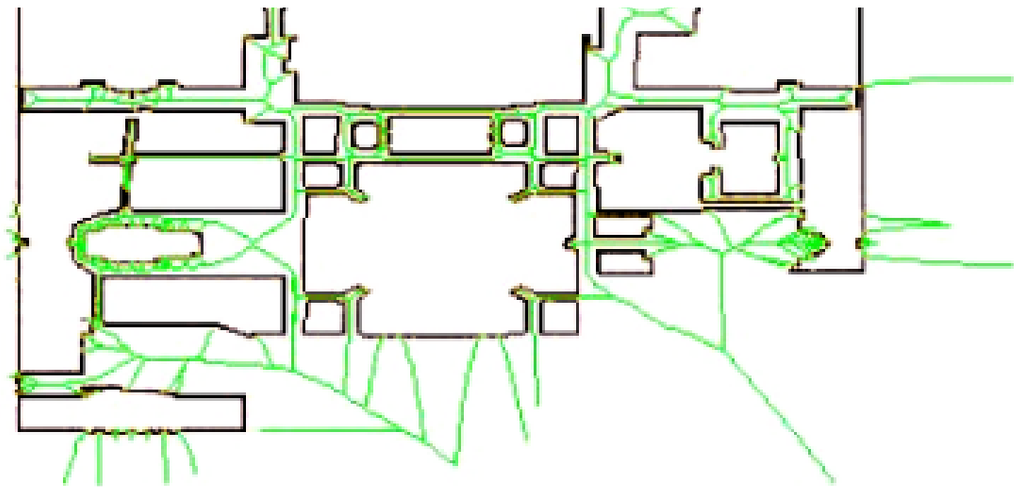


- Real environment → Voronoi skeleton → Overlapping the environment with the skeleton → A heat map for the distance from the nearest obstacle (*distance transform*)

Pros and cons of voronoi planning

- Difficult to compute in higher dimensions or non polygonal worlds
- . . . But approximate algorithms exist
- Use of Voronoi is not necessarily the best heuristic (“stay away from obstacles”) $\rightarrow$ Can lead to paths that are too much conservative
- . . . But for an uncertain robot staying away from the obstacles is a good idea
- Unnatural/counterproductive attraction to open space (see figure)
- Can be unstable: Small changes in obstacle configuration can lead to large changes in the diagram

Pros and cons of Voronoi planning

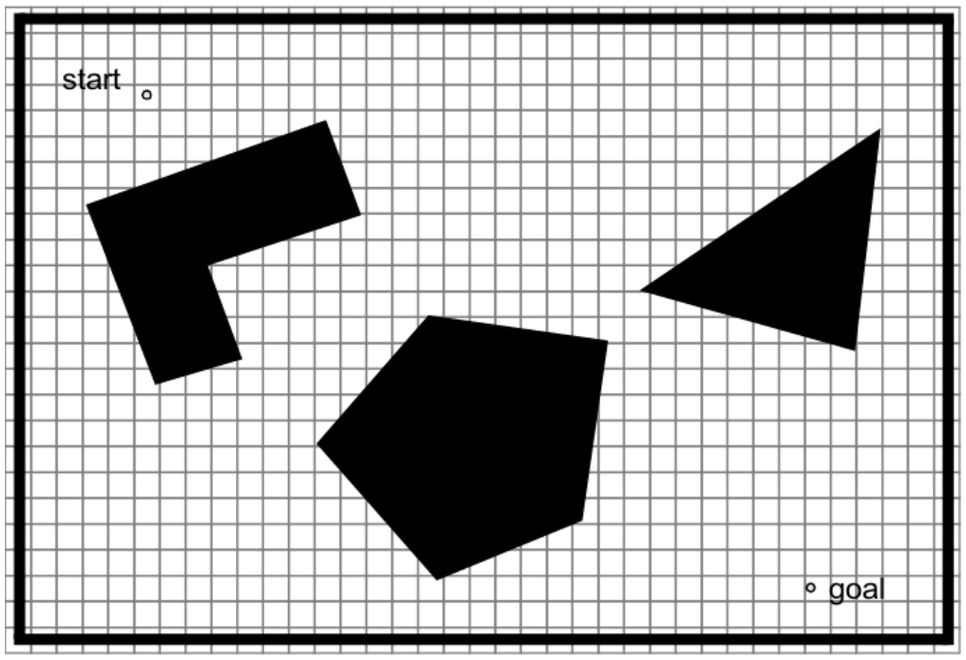


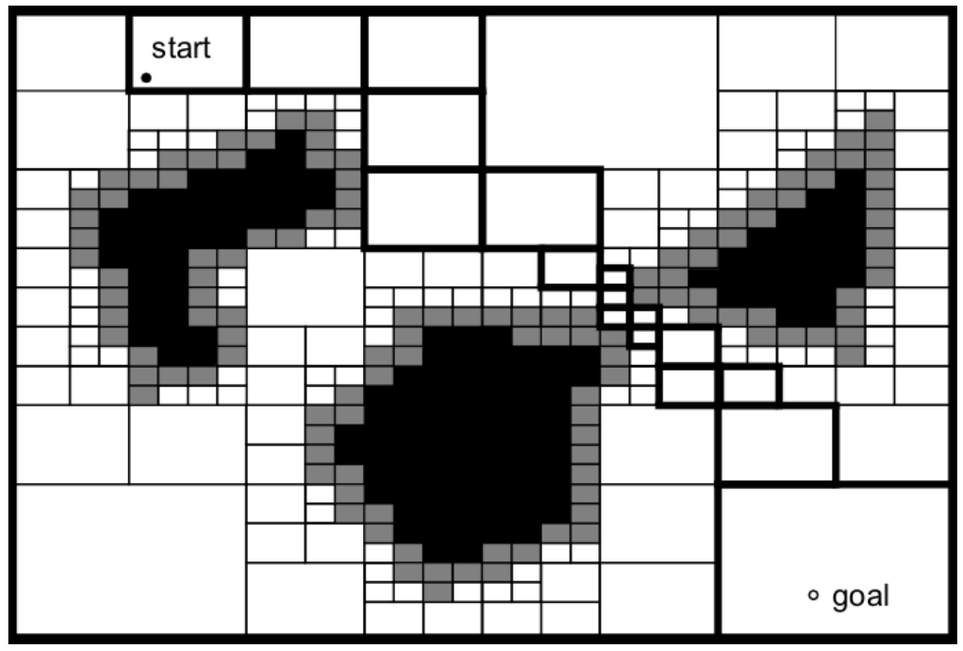
- **For robots with short-range sensors**: the path-planning algorithm maximizes the distance between the robot and objects in the environment, therefore navigating on a Voronoi path might be problematic, since the robot might not be able to use sensing to detect the objects around it and localize itself, being always too distant from the obstacles to sense them.

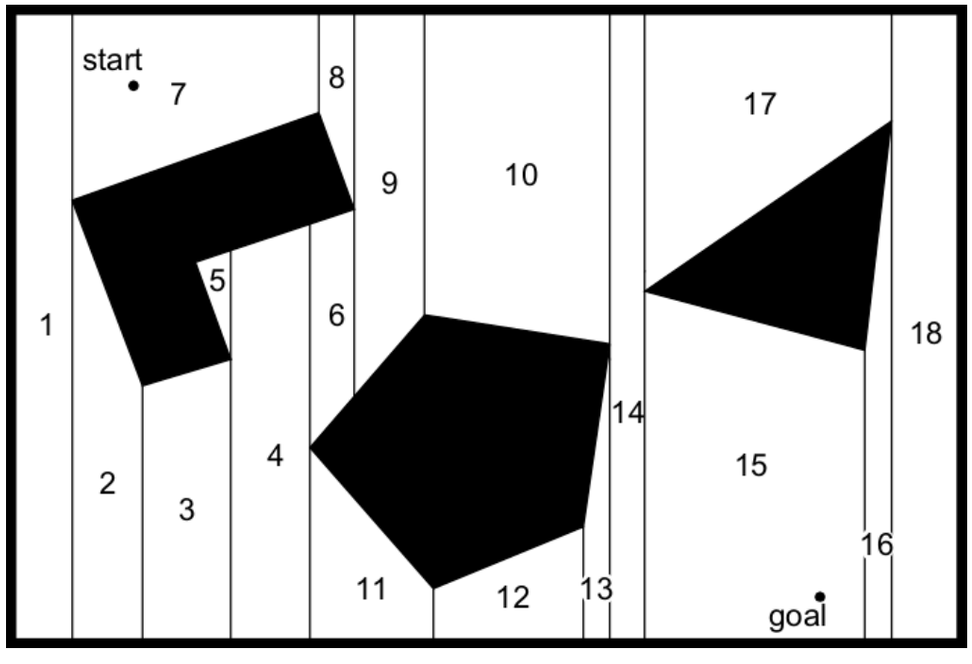
- **For robots with long-range sensors**: the Voronoi diagram method has over most other obstacle avoidance techniques the advantage of *executability*. In fact, following the Voronoi path results from maximizing the distance while maintaining equidistant from the surrounding objects, which can be done relatively easily with a good range finder. In this way, the robot can naturally stay on the Voronoi edges, mitigating, for instance, odometry inaccuracy for localization and path-following.

Discretization → Roadmap by spatial decomposition

1. A discrete, **cell-based spatial decomposition** is defined over the C-space, where any cell intersecting $\mathscr{C}_{\text{obs}}$ is marked as BLOCKED, FREE otherwise



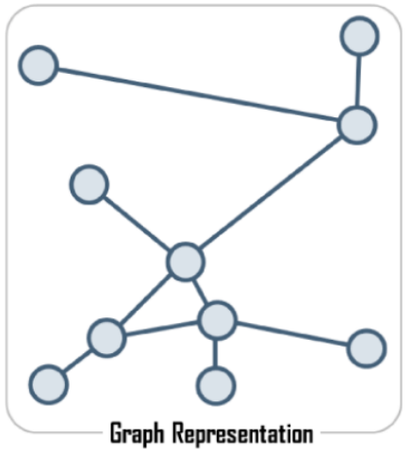




Recap: Motion planning, Discretization of C-space

✓ **Discretization** of C-space

Combinatorial problems, Sampling problems



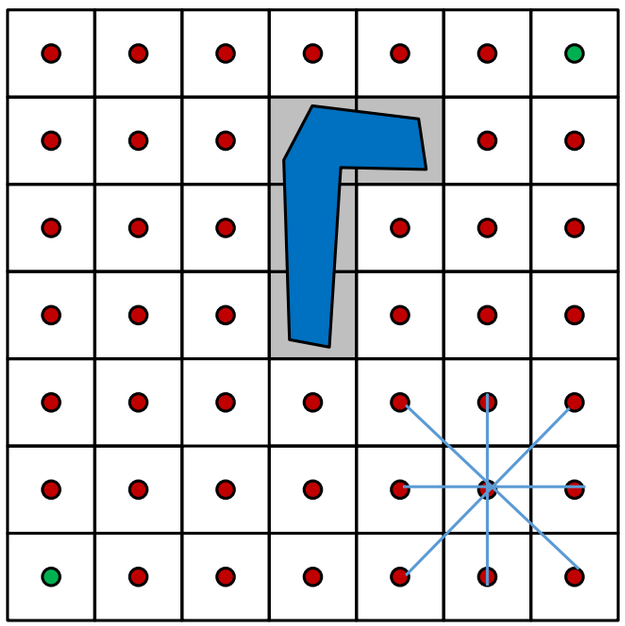
- **Roadmaps** ⇢
  - **Grids**
  - **Spatial decomposition**

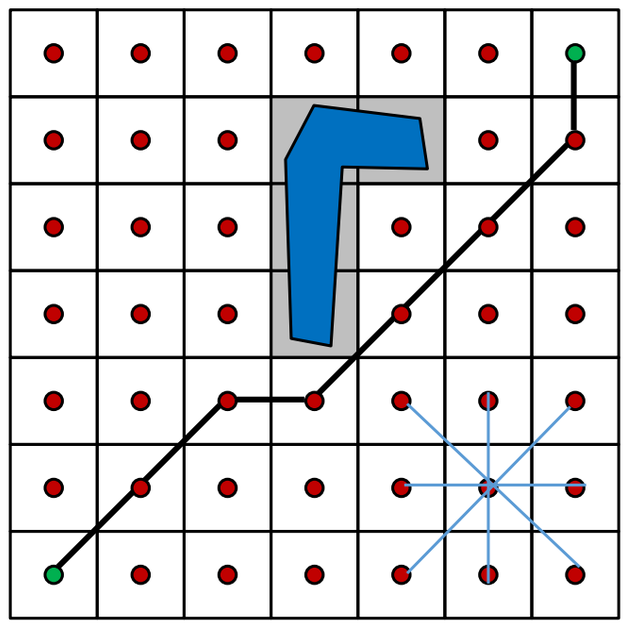
- **Graph search techniques**

  Dijkstra, A*, D*, BFS, DFS, Gradients

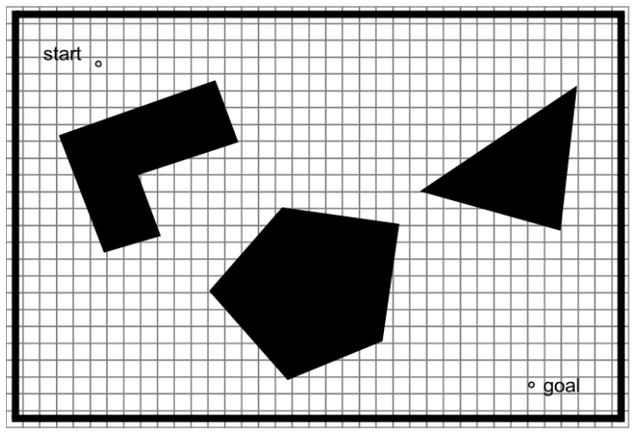
Recap: Discretization → Roadmap by spatial decomposition

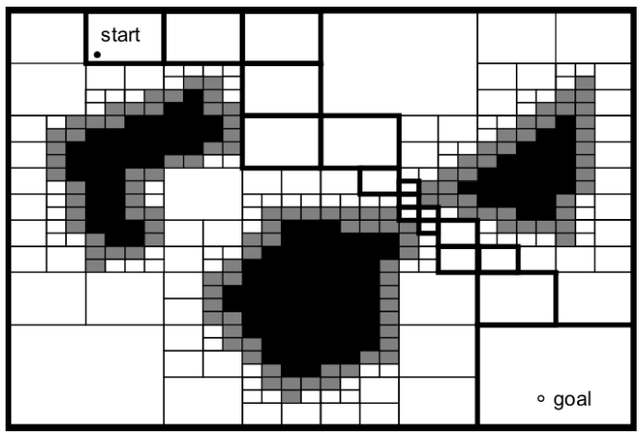
1. A discrete, **cell-based spatial decomposition** is defined over the C-space, where any cell intersecting $\mathscr{C}_{\text{obs}}$ is marked as BLOCKED, FREE otherwise
2. Determine adjacent FREE cells and construct a **traversability / adjacency** among them: E.g., vertices can be placed in the mid of each cell, edges connect them
3. Find the cells in which the initial and goal configurations lie, and search for the best (e.g., shortest) path in the connectivity graph that joins initial and goal cells: the path consists of a sequence of FREE cells ↔ **node path in the graph**

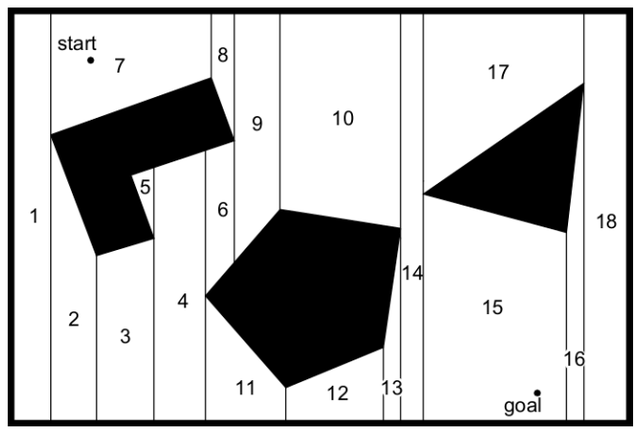


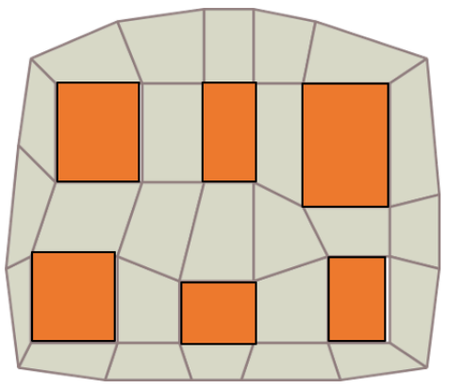


Recap: Discretization → Roadmap by spatial decomposition

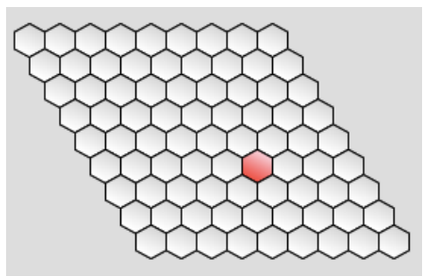


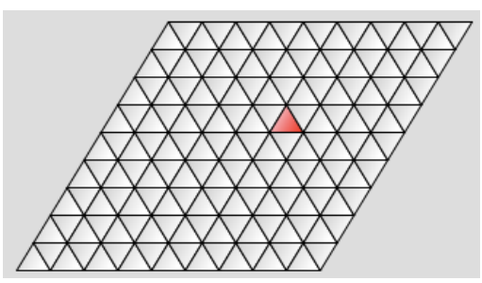




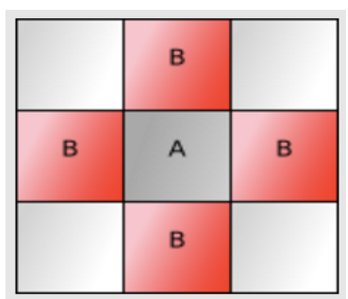


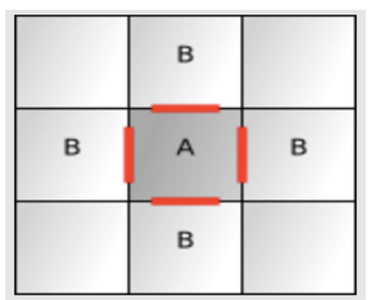
Cell spatial decomposition can be done in many different ways, using different geometries

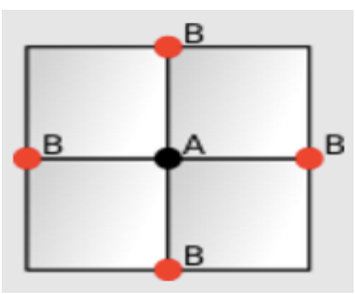


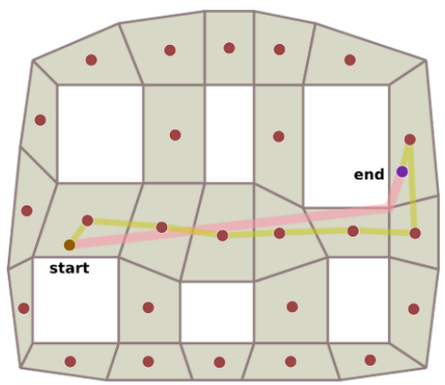


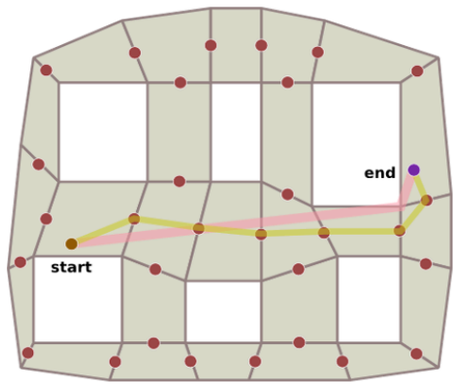
Cell's reference point(s) for navigation can be selected at center, edges, corners

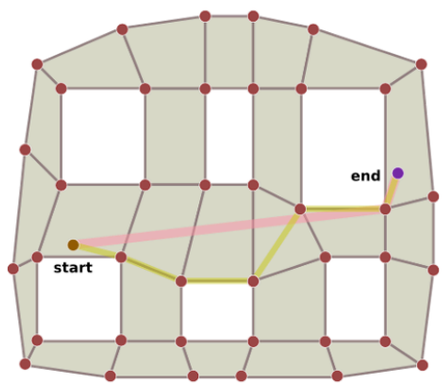


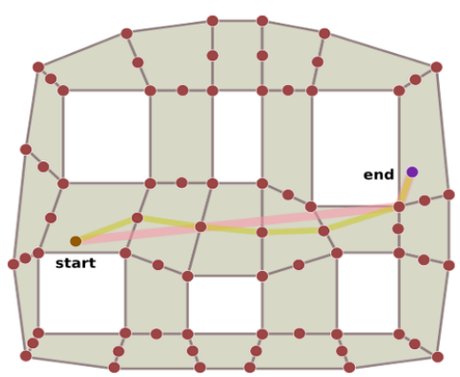












Recap: Discretization by spatial decomposition

Classification of spatial decomposition approaches:

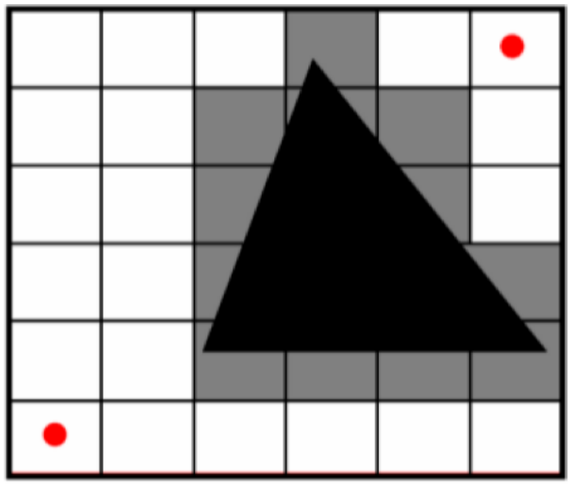
- ***Exact cell decomposition:*** Cell boundaries are placed as a function of the structure of the environment, such that the decomposition is *lossless,* no parts of $\mathscr{C}_{\text{free}}$ are removed because of the decomposition

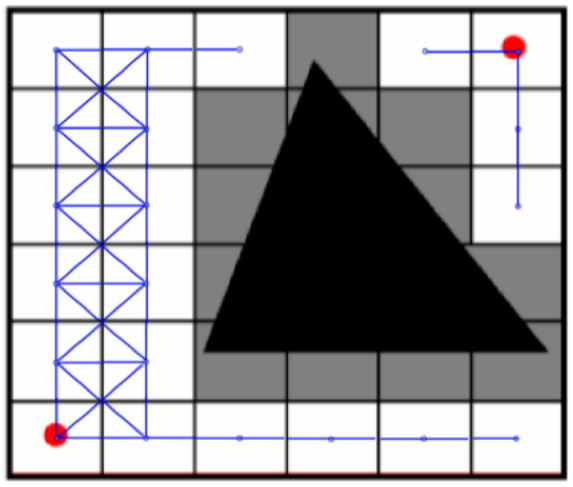
- ***Approximate cell decomposition:*** The decomposition is *lossy,* resulting in an approximation of $\mathscr{C}_{\text{free}}$, that removes parts that are potentially accessible

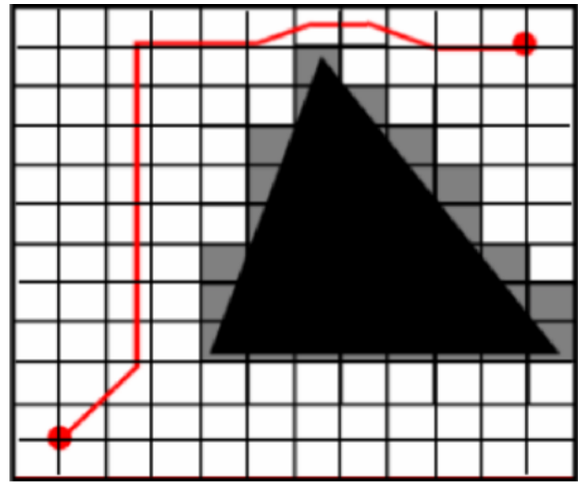
The way we perform decomposition matters for completeness!

The shortest path through cell centers is the optimal path on the graph!

Approximate cell decomposition



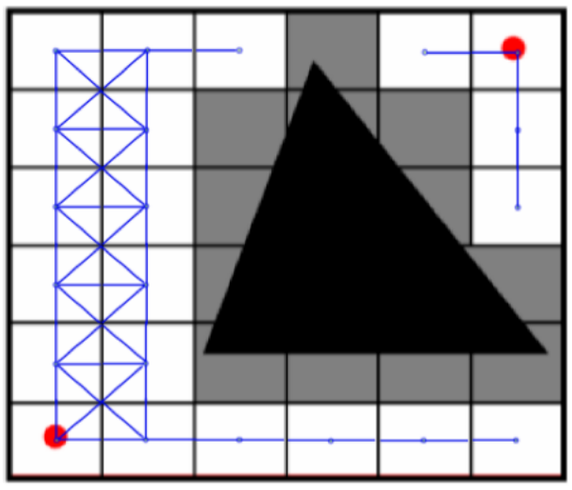


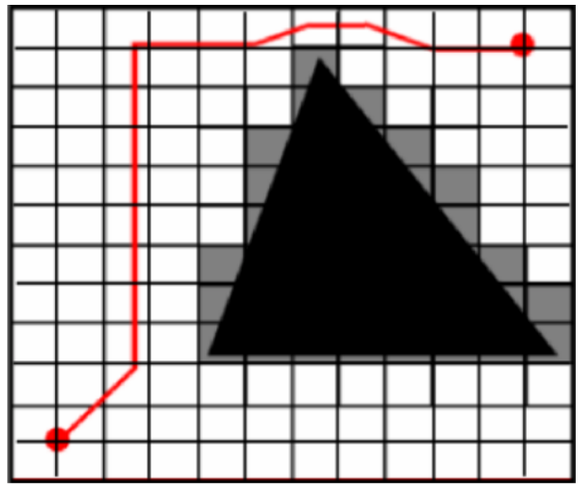


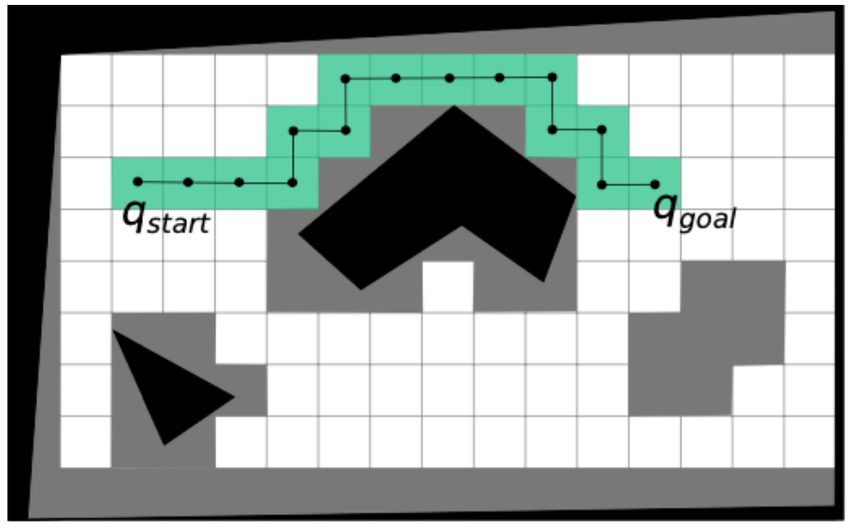
- Let's consider a spatial decomposition implemented using a uniform grid of cells, all of the same shape (e.g., squares)

1. A discrete grid is defined over the entire C-space using a predefined shape
2. Any cell intersecting an obstacle is marked as BLOCKED, as FREE otherwise
3. An **adjacency graph** $G(V, E)$ is constructed by placing a vertex in (e.g., the middle of) each cell and connecting the adjacent FREE cells
4. Given a query, the solution is the shortest path on the resulting graph

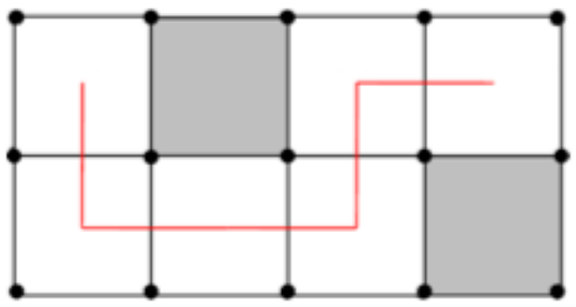
Approximate cell decomposition: neighboring

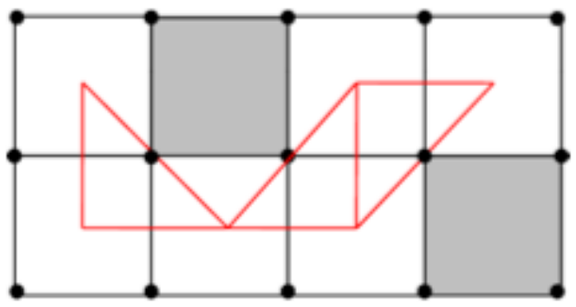




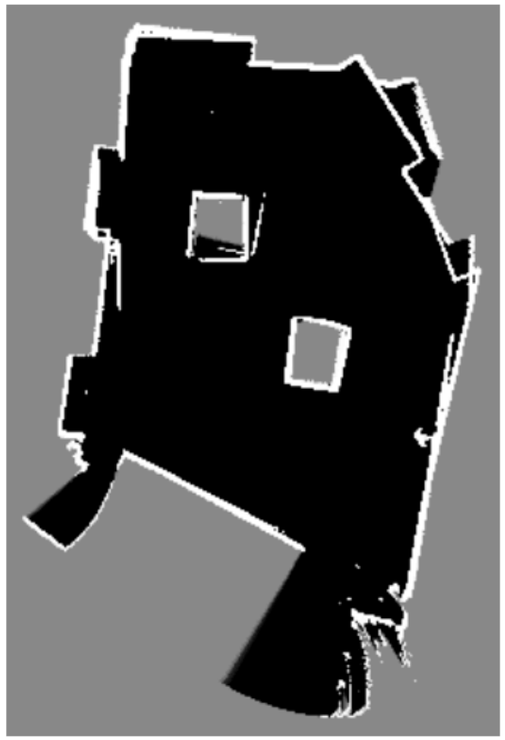


- The resulting adjacency graph can be constructed considering, for instance, a 4-neighbor topology, or an 8-neighbor topology, that result in different traversability of the free space





Approximate cell decomposition



➡ A grid map can be obtained from CAD images / bitmaps or constructed from occupancy grids built from sensor data (e.g., range finder scans)

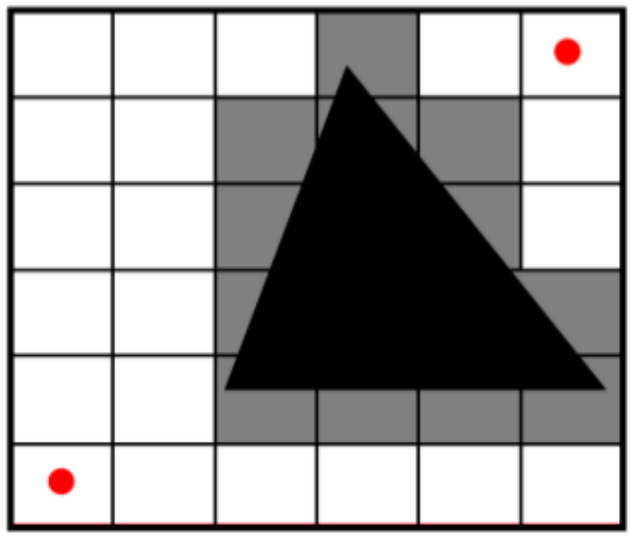
- $\{0,1\}$ grid → Bitmap occupancy grid
- $[0,1]$ grid → Probabilistic occupancy grid

Not complete

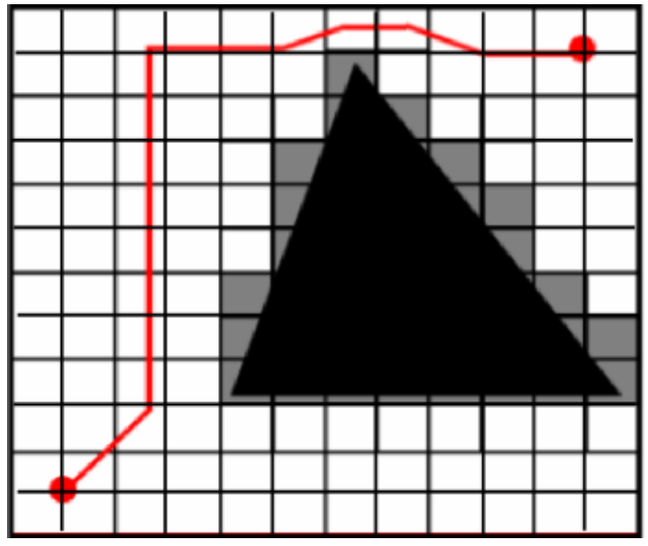
☑ An approximate cell decomposition is easy to construct, complexity is linear in number of cells, and it allows the use of efficient path search algorithms specialized for grids

➡ The effective portion of C-space tagged as $\mathcal{C}_{obs}$ results enlarged because of discretization process → Resulting connectivity graph is an **approximate roadmap**

➡ **Narrow passages problem**: Even if a solution path exists in $\mathcal{C}_{free}$ the algorithm might report a failure, depending on grid resolution → Incompleteness



With this relatively large cell size (and small graph) an admissible path doesn't exist

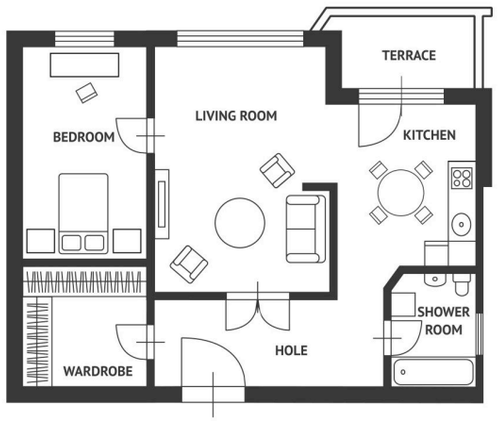


With this more fine-grade grid an admissible path does exist, but connectivity graph is much larger now

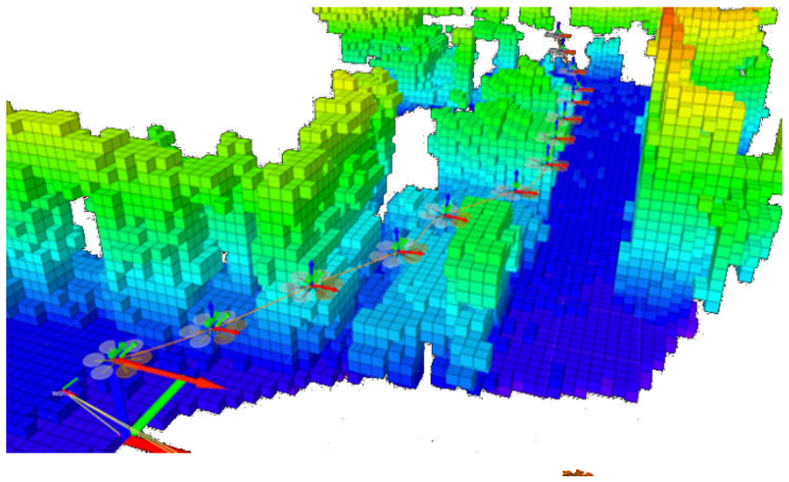
✓ **Variable cell size** to set a good trade-off between completeness and computations

Combinatorial explosion

- How many grid cells do we need?
- E.g., mobile planar omnidirectional robot in a workspace area of $100\ m^2$



➡ If we want cm-level precision: $(100 \times 100)^2 = 10^8$ cells!

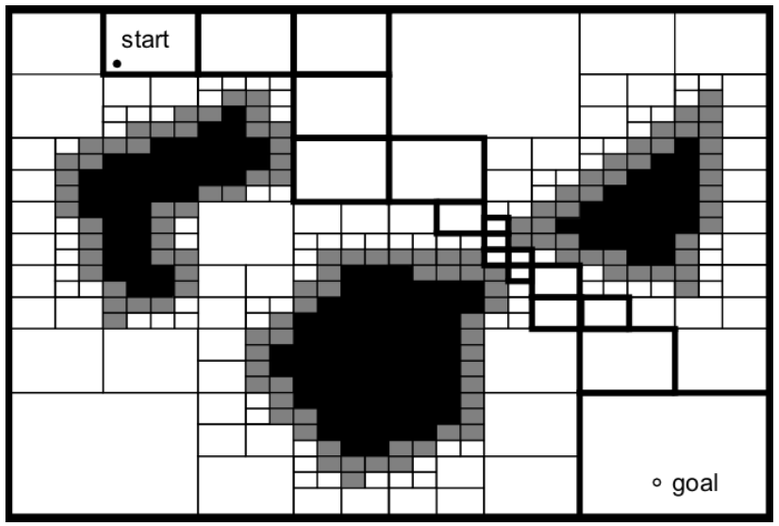


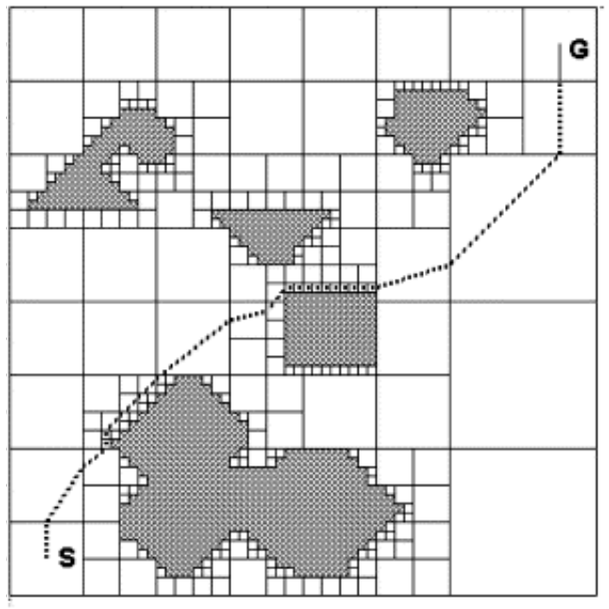
➡ If robot moves in 3D space: $(100 \times 100)^3 = 10^{12}$ cells!

➡ For robot arms, configuration space if n-dimensional, one for each joint, which can take values based on the type of joint (e.g., in $[0, 2\pi]$ for an unconstrained revolute joint)

✓ **Variable cell size** to set a good trade-off between completeness and computations

Quad-trees (n-trees) iterative decomposition

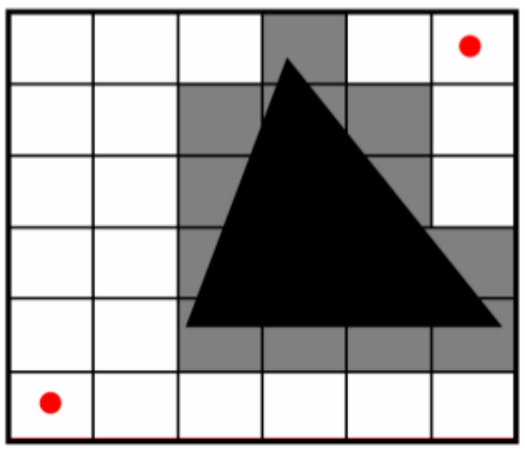


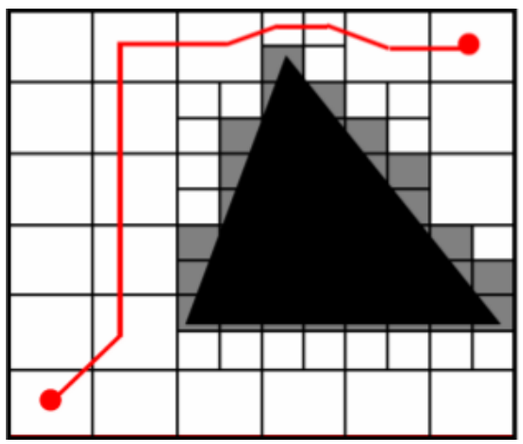


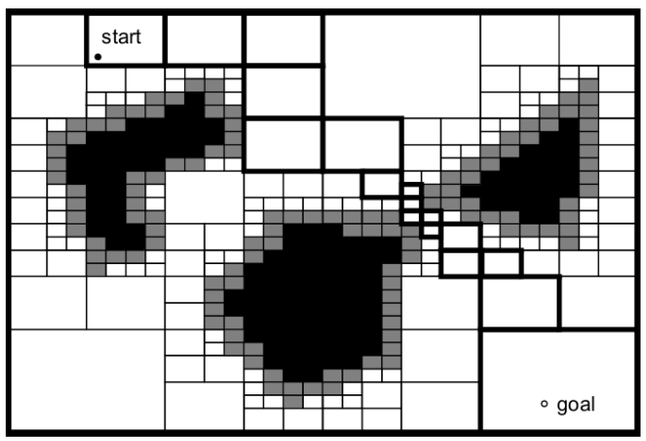
Quadtree spatial decomposition:

1. Define an initial cell size
2. **Distinguish** among:
   A. Cells that are entirely contained in $\mathcal{C}_{obs}$ (FULL cells)
   B. Cells that partially intersect $\mathcal{C}_{obs}$ (MIXED cells)
   C. Cells that do not intersect $\mathcal{C}_{obs}$ (FREE cells)
3. **Recursively subdivide** in $n = 4$ cells the MIXED cells until either
   - all cells are FULL or FREE, or
   - a predefined resolution limit is reached

Quad-trees (n-trees) iterative decomposition

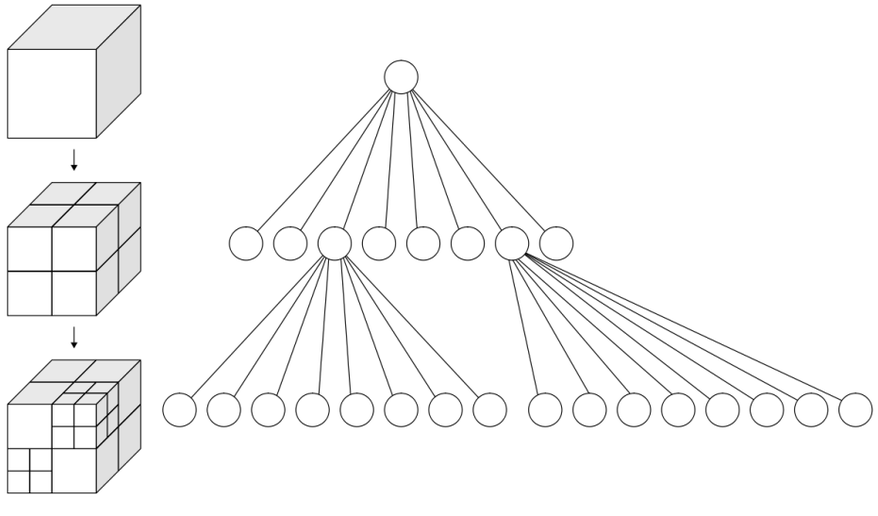


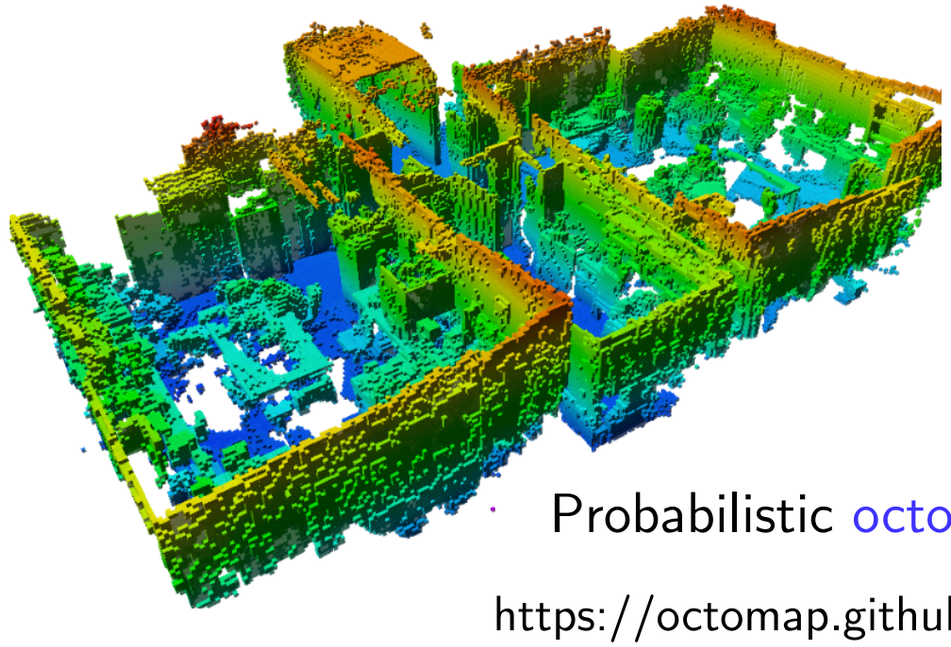




- **Resolution limit can be set iteratively**: first, a pass of quad tree construction of performed, IF an admissible path is found, it is returned
- Otherwise, another pass of the recursive decomposition is performed, until an admissible path is found, or some time/space limit is exceeded

- In 3D, an **octree** can be effectively used, which is a tree data structure where each cell gets uniformly decomposed in 8 smaller cells



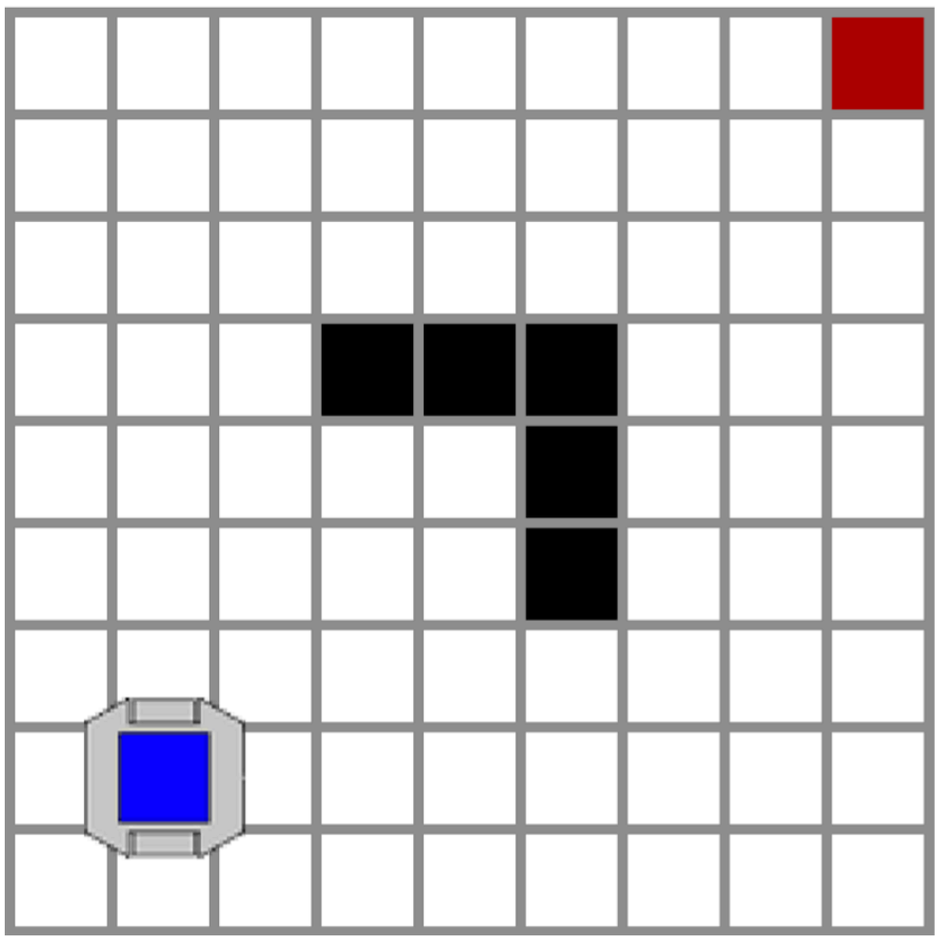


Probabilistic octomap

https://octomap.github.io

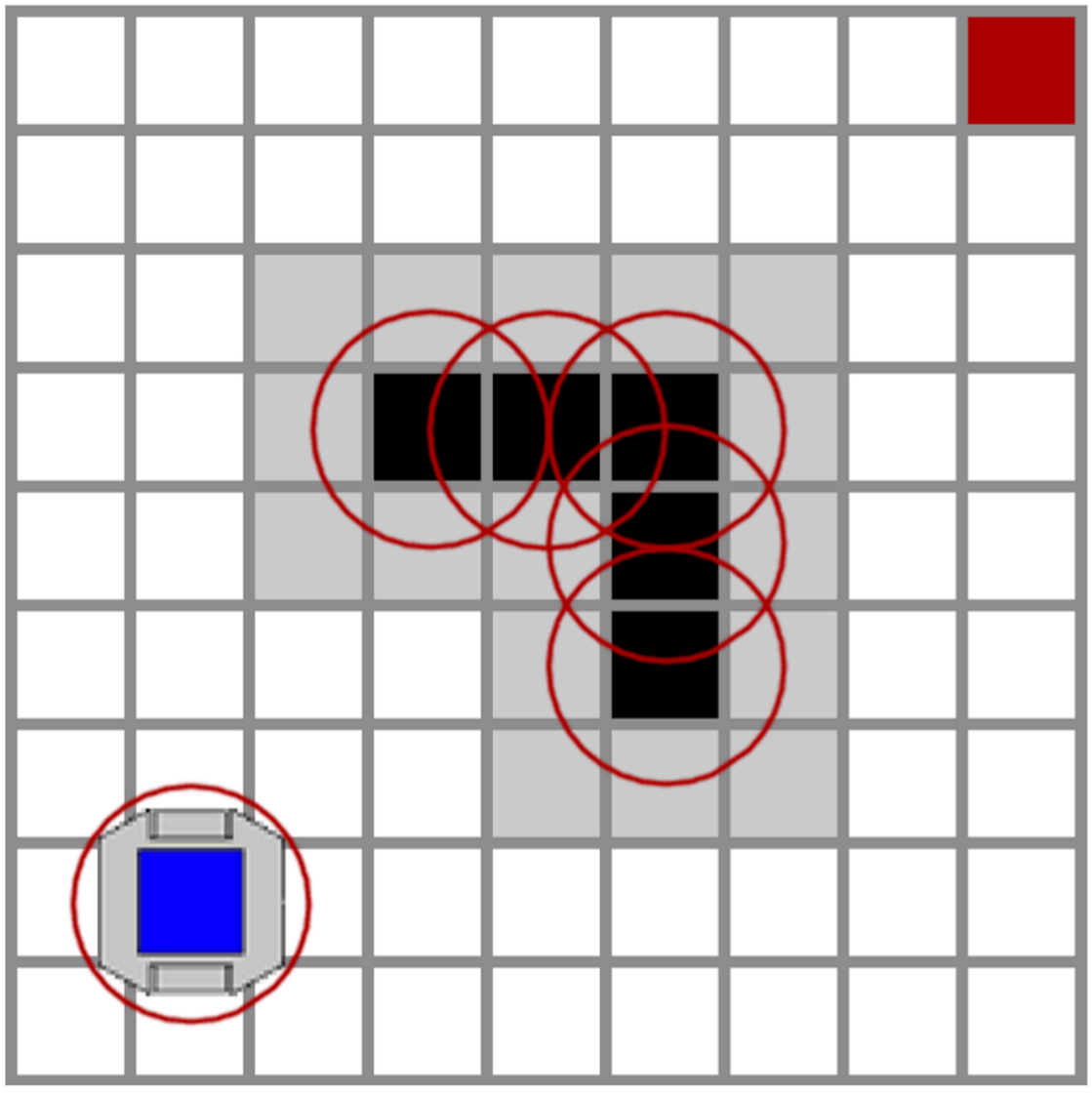
Grid-based planning: wavefront propagation approach

- Given: (Geometric) map, a query (from blue to red)
- A uniform grid is placed over the configuration space

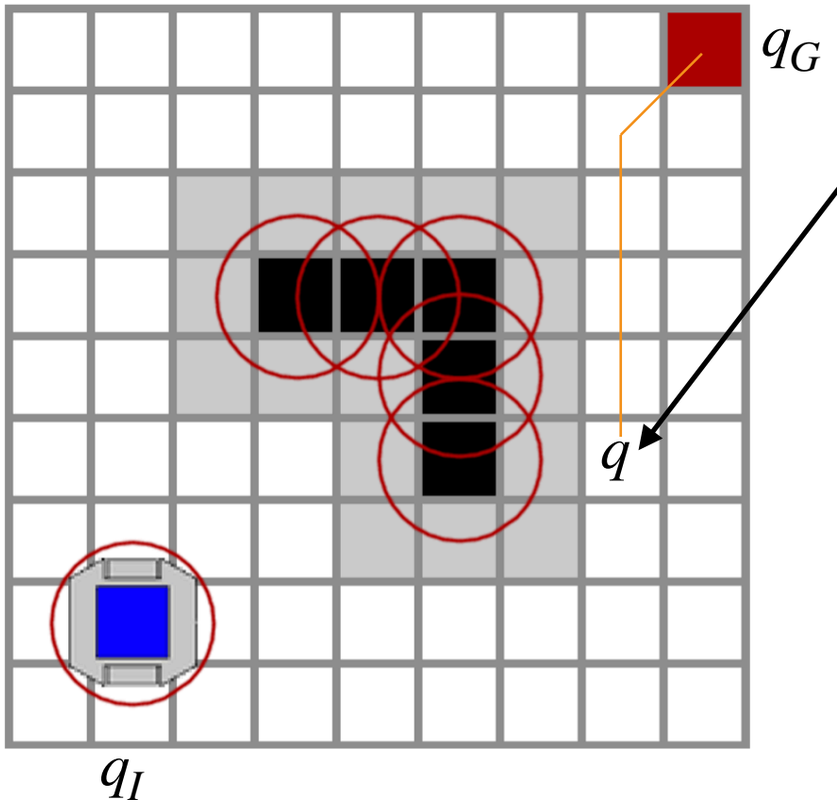


Wavefront propagation: object growing

- Obstacle / robot growing: Not mandatory but useful / recommended



Navigation functions



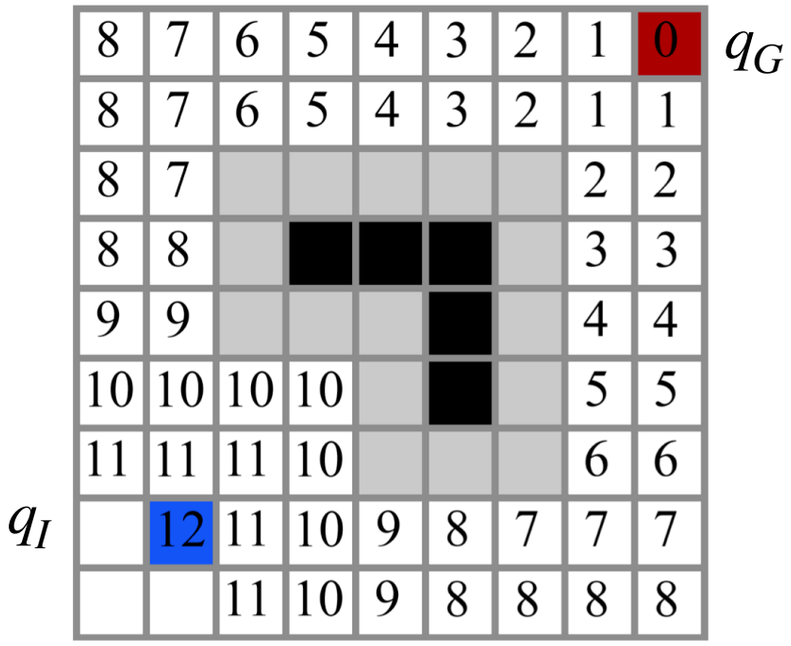
- What is the *cost* $C(q)$ for the robot of being at a specific configuration given that it has to reach the goal configuration q_G minimizing the cost of the traveling path?

- $C(q)$ represents an estimated or exact **cost-to-go** from $q \rightarrow q_G$ (based on some performance metrics)

- 8-neighborhood
- $C(q)$ = Steps to go(al) configuration

8	7	6	5	4	3	2	1	0
8	7	6	5	4	3	2	1	1
8	7						2	2
8	8						3	3
9	9						4	4
10	10	10	10				5	5
11	11	11	10				6	6
	12	11	10	9	8	7	7	7
		11	10	9	8	8	8	8

Navigation functions



- $C(q)$ represent an estimated or exact **cost-to-go** from $q \to q_G$ (based on some performance metrics)

Navigation function $C : \mathcal{C} \to \mathbb{R}_0^+$

- $C(q)$ can be used by a ***navigation*** algorithm to:

  - directly select the next configuration, q_{next} in the local neighborhood $\mathcal{N}(q)$

$$q_{next} = \arg \min_{q' \in \mathcal{N}(q)} C(q')$$

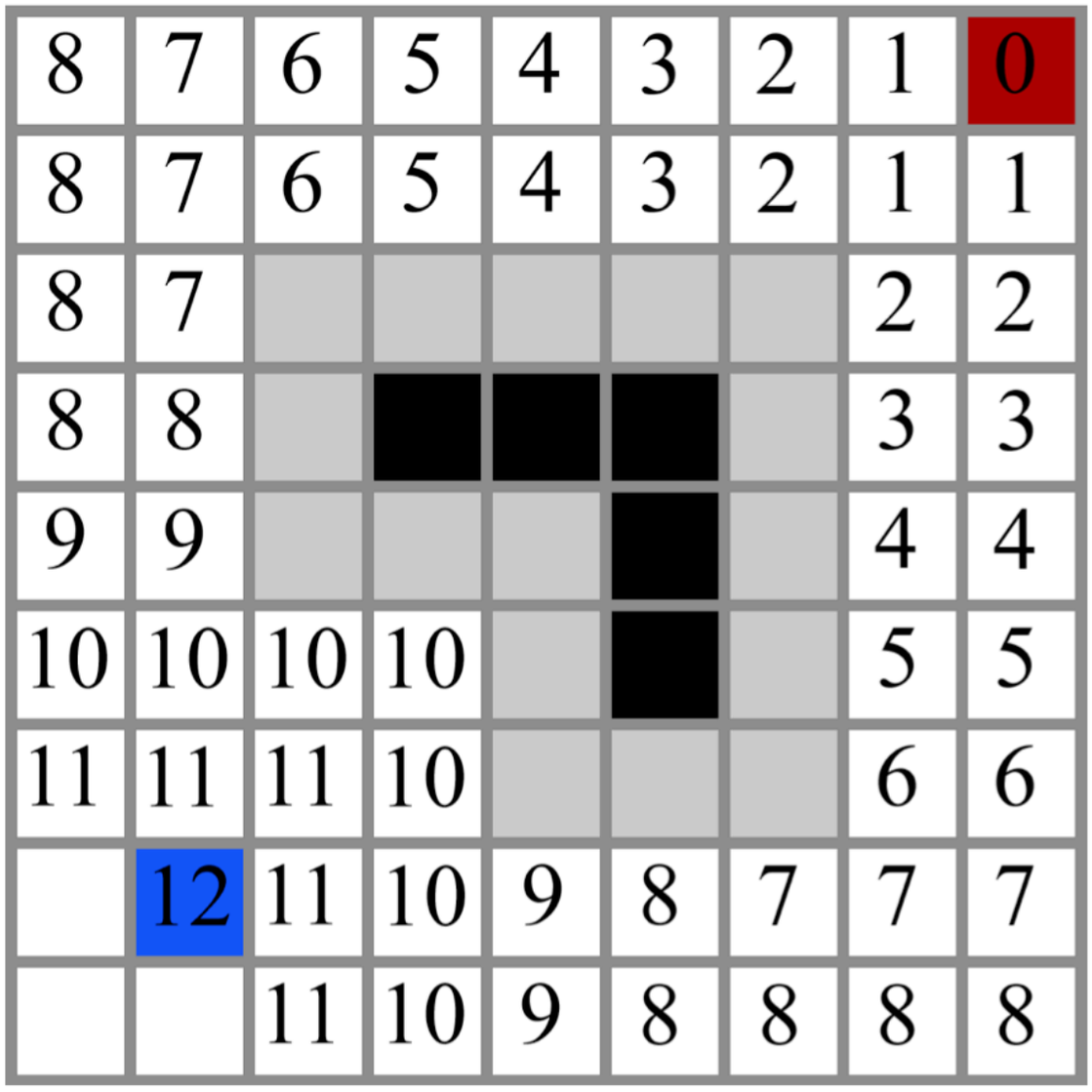
  - select next control action u_{next} among the locally feasible ones in set $\mathcal{U}(q)$ (e.g., v, ω, accounting for non-holonomicity), f is robot's dynamics

$$u_{next} = \arg \min_{u' \in \mathcal{U}(q)} C\big(f(q, u')\big)$$

Wavefront propagation: flood-fill configurations with cost-to-go values

- Navigation function by flood-fill configurations with cost-to-go values

- Compute steps starting from goal, q_G, moving as a wavefront, step-by-step until q_I is reached

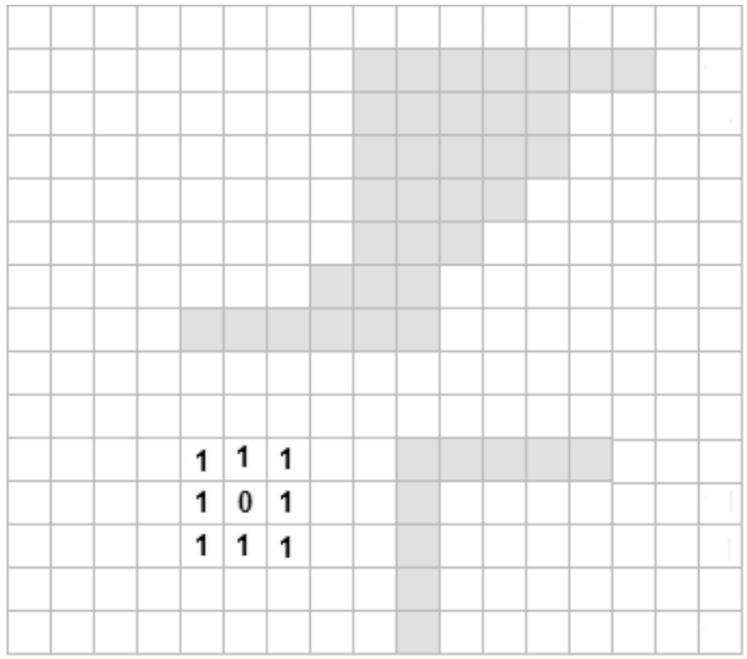


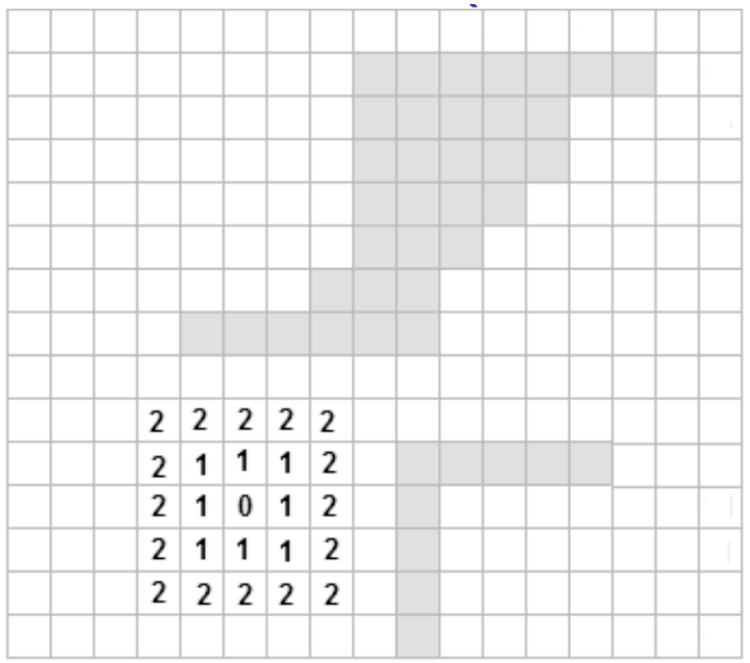
Wavefront propagation: flood-fill configurations with cost-to-go values

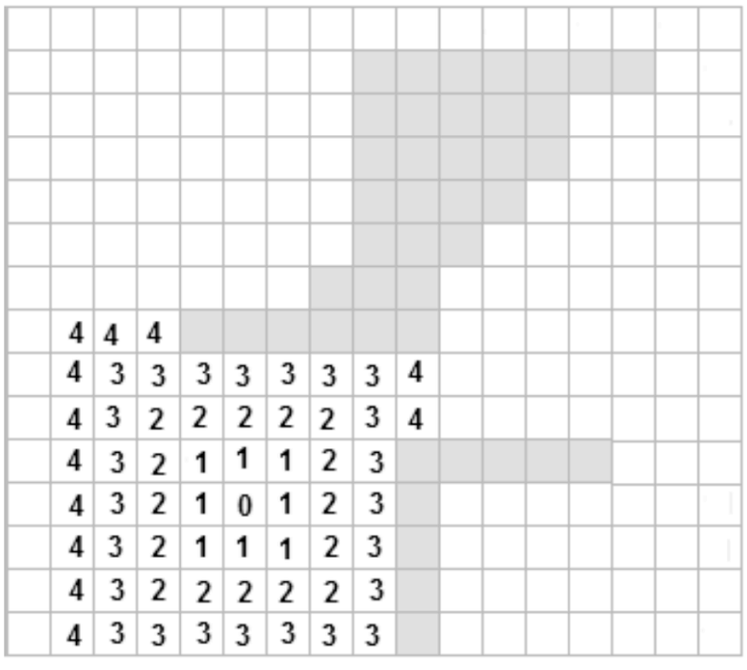
A slightly different, a bit more exciting scenario ...

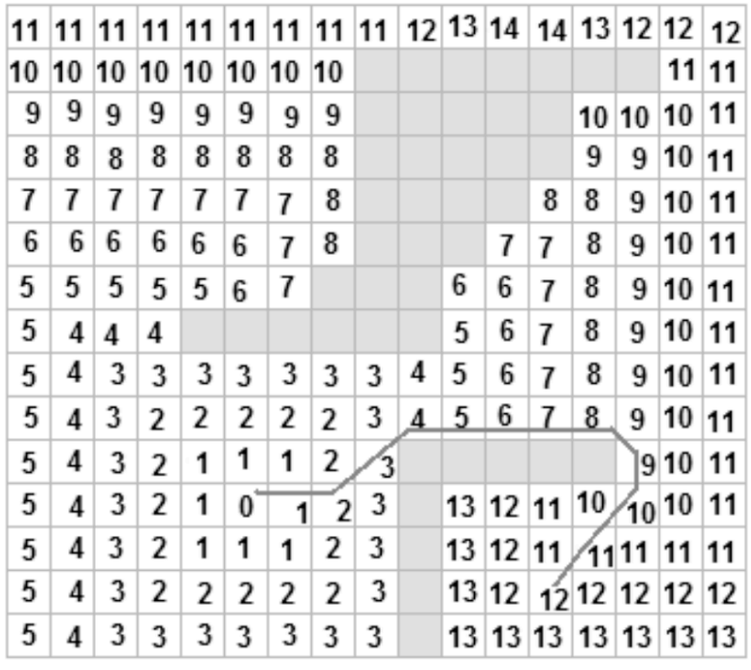
➡ If ALL cells are filled with a $C(q)$ value → Any query $(*, q_G)$ can be answered!

q_G



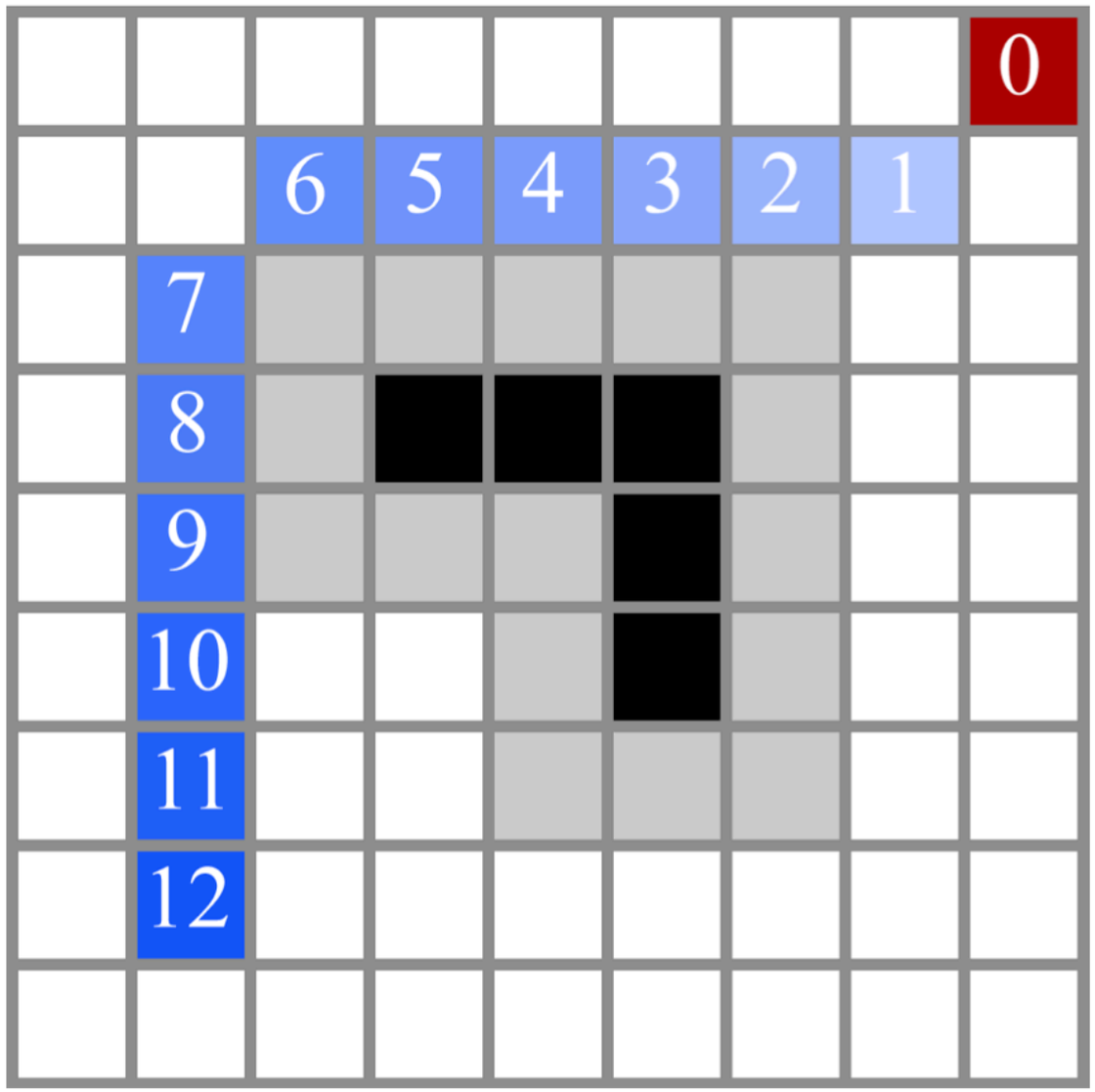




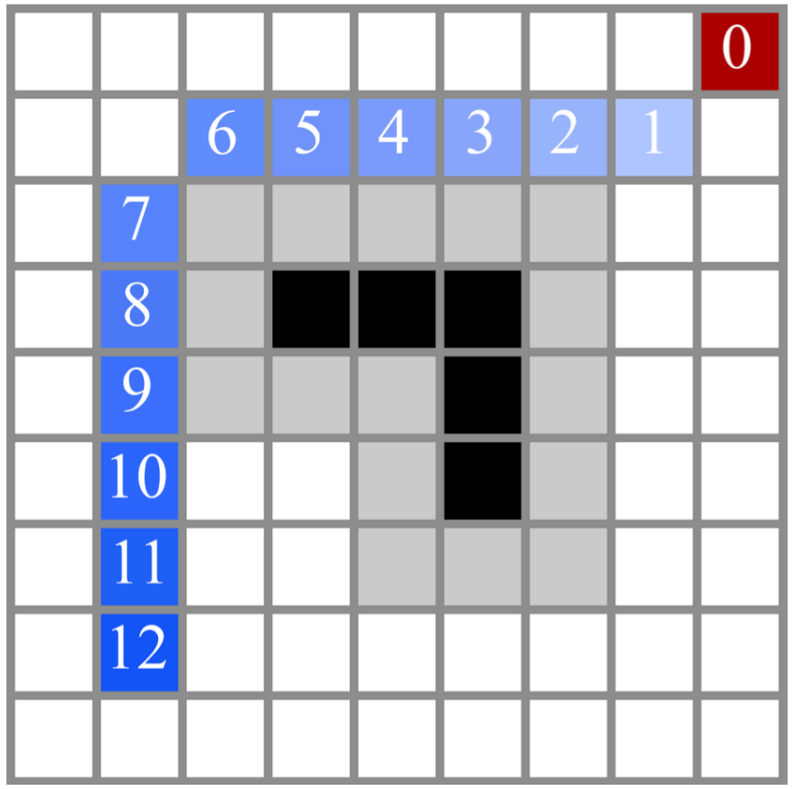


Wavefront propagation: find path using navigation function

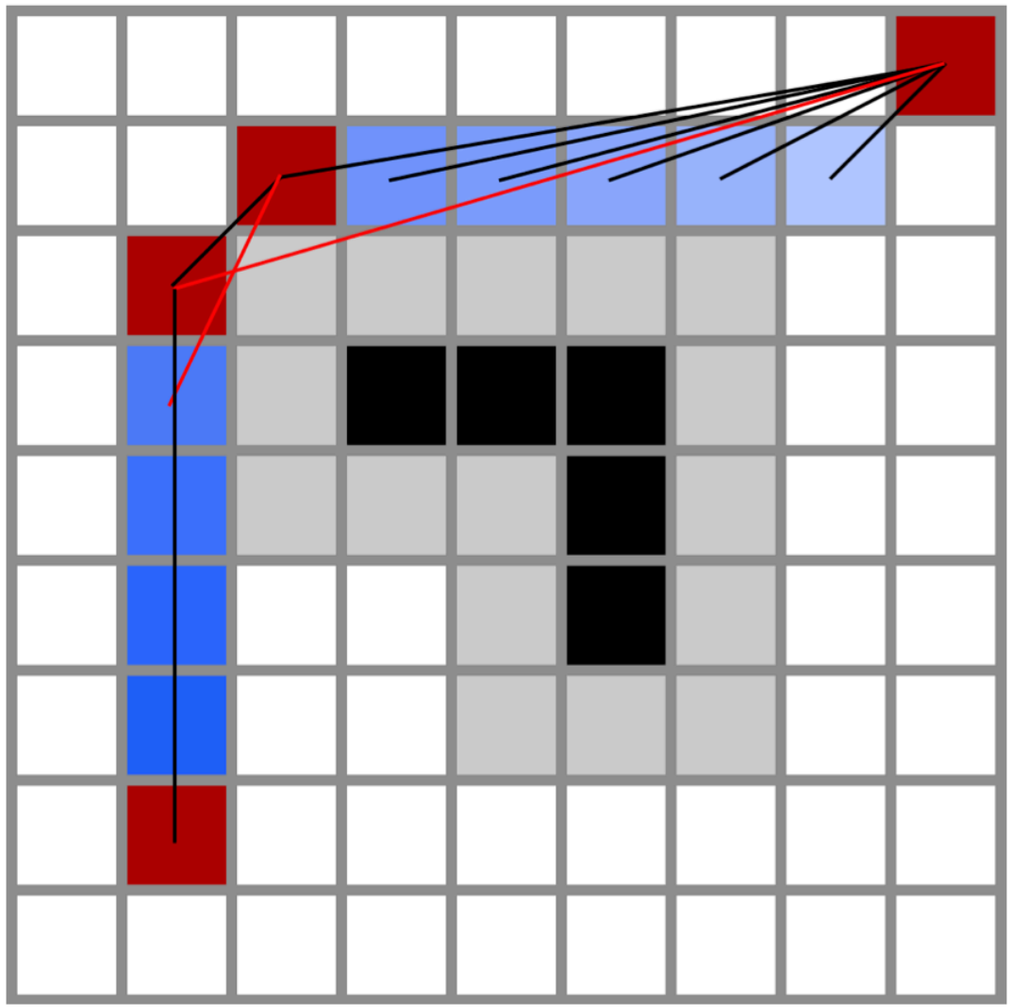
- Find a **path** using the navigation function $C(q)$:
  at each step, move to the configuration q' with the lowest $C(q')$ value in neighborhood



Wavefront propagation: improve path by finding shortcuts



- Is the path *good*?
  - ✓ Admissible
  - \- Not optimal in configuration space

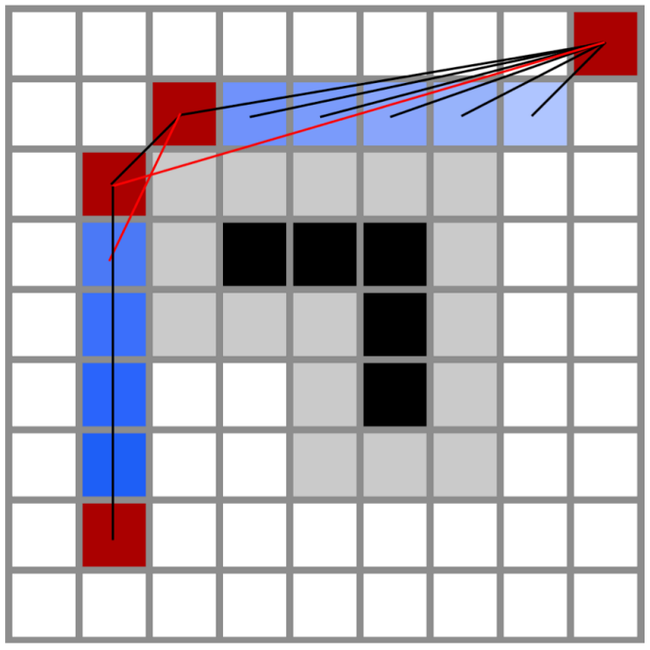


➡ Find shortcuts / smoothing the path

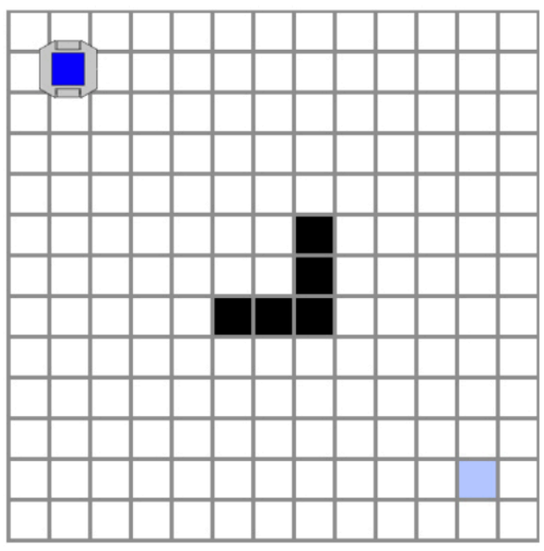
- Any two non-contiguous configurations in the path that can be directly connected by a path in C_free can be directly joined in the path, skipping the intermediate configurations

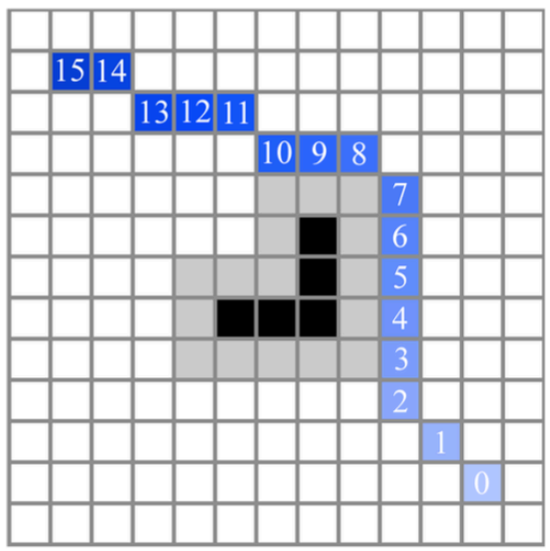
- **Ray-shooting** for finding the shortcuts: Bresenham's line algorithm

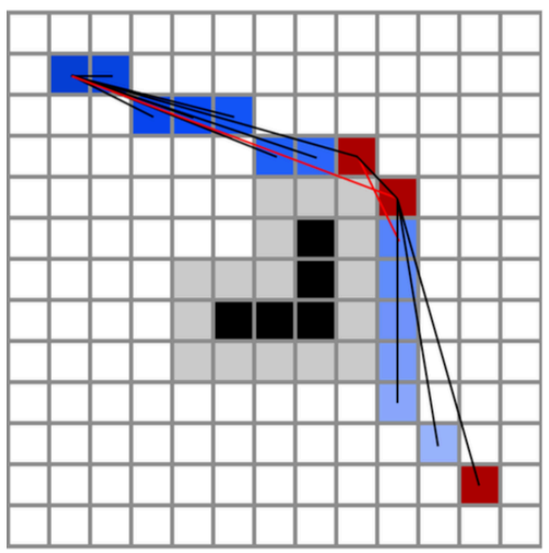
Wavefront propagation: improve path by finding shortcuts

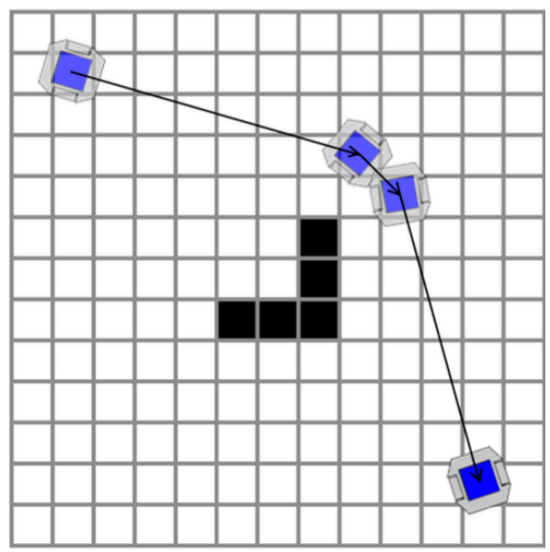


- Using a 8-neighborhood is expensive
  - Find first a **coarse path** using a 4-neighborhood
  - **Improve** the path with **ray-shooting shortcuts**









Exact spatial decomposition

- **Exactly** decompose C_free into a **finite set of non-overlapping cells,** $cell_i$, $i = 1,\ldots,n$

$$\mathcal{C}_{free} = \cup_{i=1}^{n} cell_i, \quad \cap_{i=1}^{n} cell_i = \emptyset$$

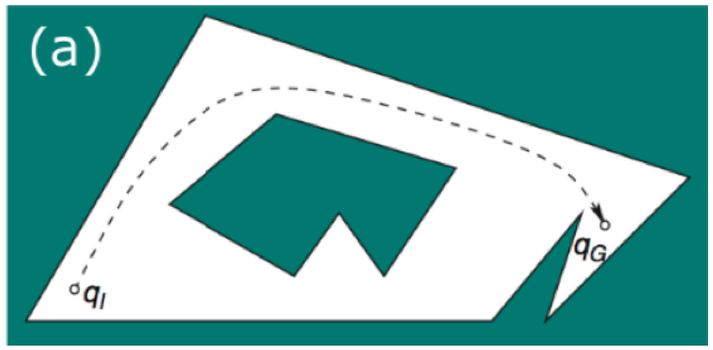
  - Construct the adjacency graph to represent adjacencies among the cells
  - Search the for the shortest path on the graph

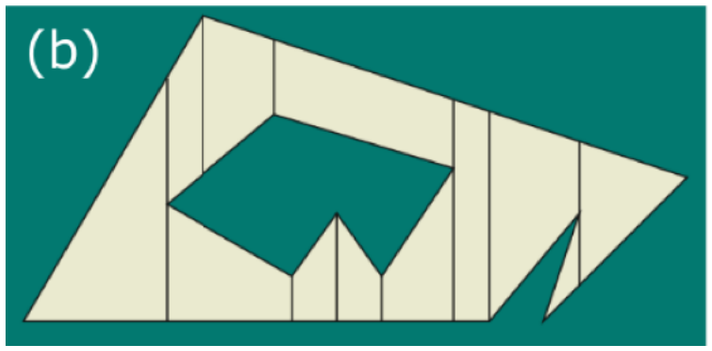
☑ Since no portions of free space are removed, the roadmap is ***complete***, i.e., if a complete search algorithm is employed, it is guaranteed to find a path, if it exists

- The particular position / navigation path of the robot within a cell doesn't matter for completeness, it only matters for **optimality**

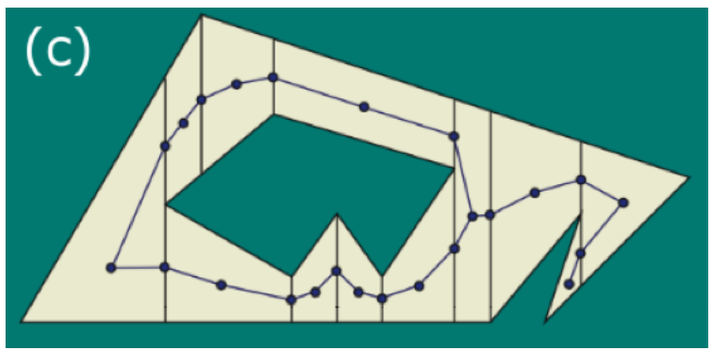
  - A lossless cell decomposition can be implemented in many different ways

Exact cell decomposition





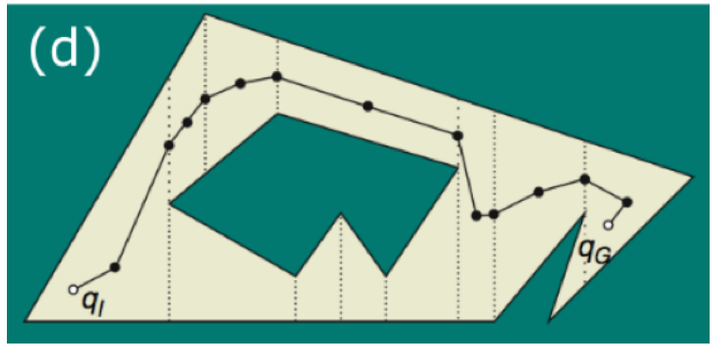
1. From each polygon vertex, shoot upward and/or downward *vertical rays* in C_{free} → This results in decomposing C_{free} into **trapezoids** with two parallel vertical sides



2. Place one **vertex in the interior of every trapezoid** (e.g., at centroid)

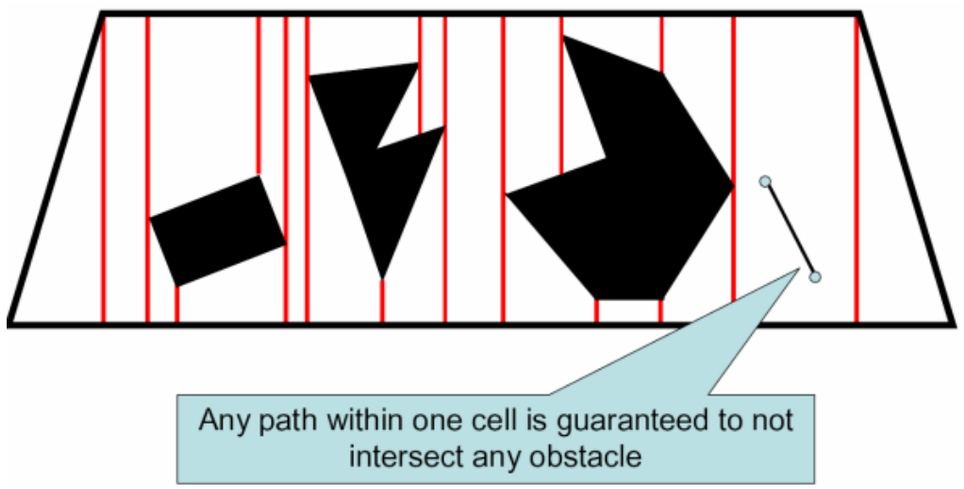
3. Place one **vertex in every vertical segment** (e.g., at middle point)

4. Connect the vertices to create a **connectivity graph** → **Roadmap graph**



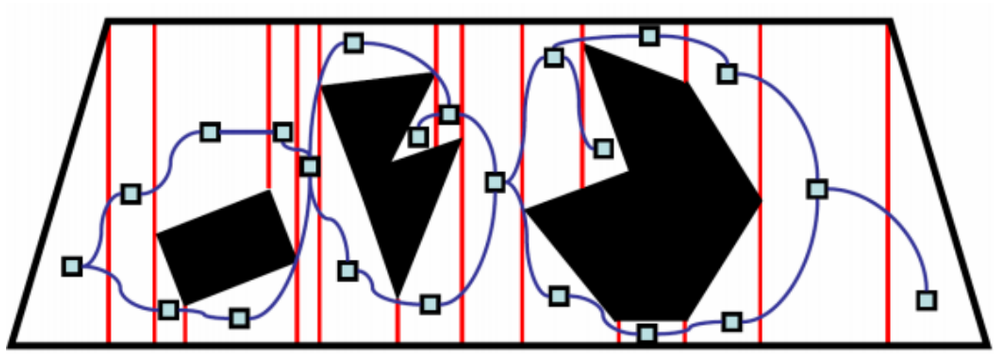
- Find the **shortest path** for the query (q_I, q_G) on the roadmap graph

Exact cell decomposition



- Within a cell the free space is guaranteed, such that a path connecting adjacent cells is a guaranteed free space path.

  However, **how to navigate inside a cell and from one cell to another** is an additional task delegated to a local navigation controller.

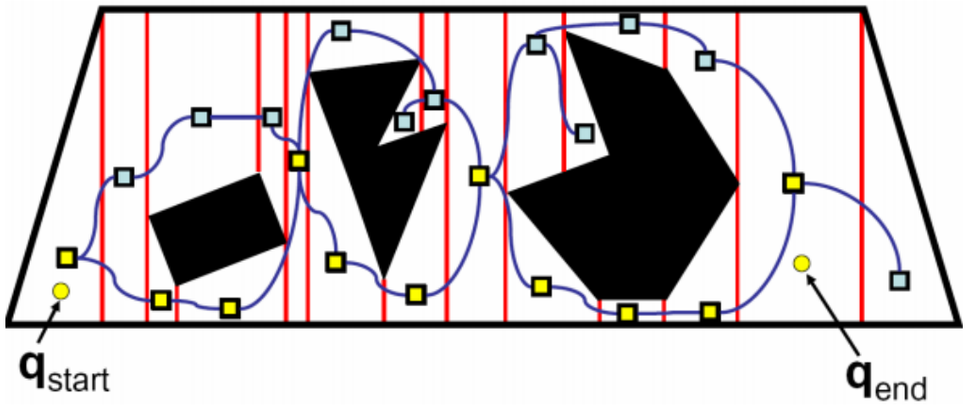


- The graph of cells defines a roadmap. **The roadmap graph depends upon the density and complexity of objects in the environment**.

  In presence of a *highly cluttered* environment the size of the graph can rapidly grow.

  In *sparse environments* the connectivity graph will be small, even if the geometric size of the environment is very large.

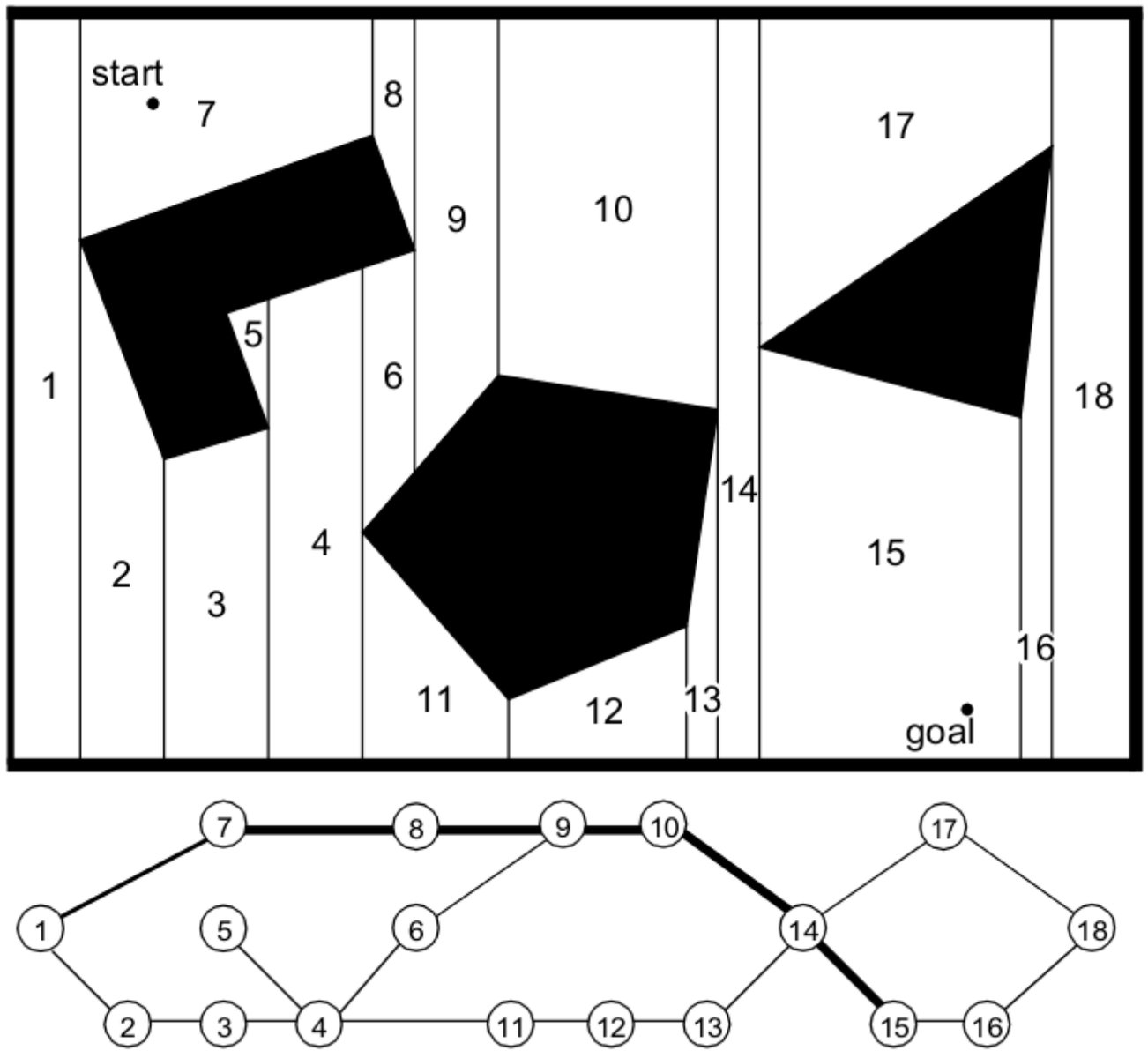
☑ Exact cell decomposition guarantees **completeness**



- The connectivity graph can be used to find a path between **any two configurations**.

Note: Vertices in the vertical segments can be added/used

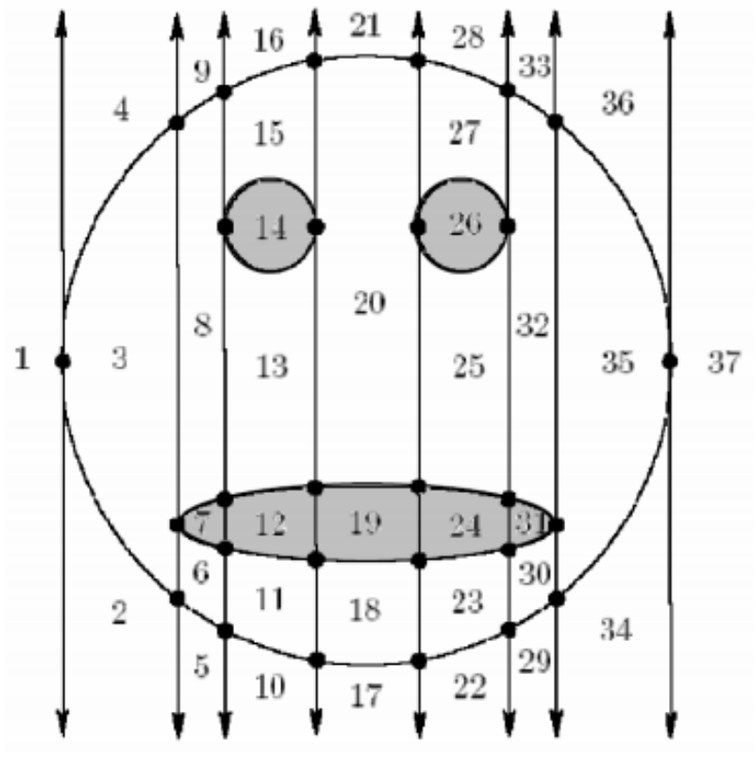
Exact cell decomposition



A sort of layered graph layout can be defined

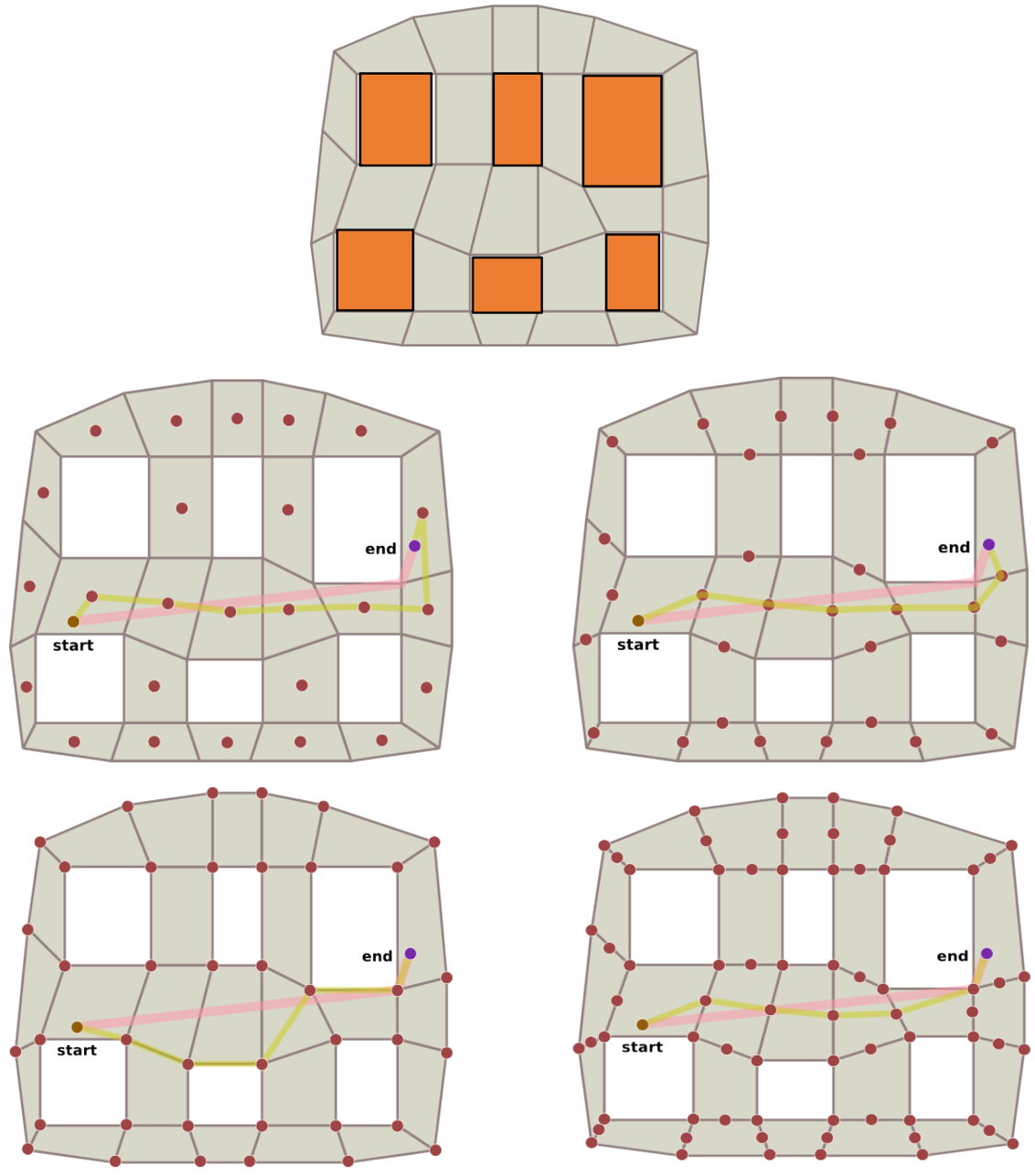
Exact cell decomposition

- A version of exact cell decomposition can be extended to higher dimensions and non-polygonal boundaries, such as *cylindrical objects*



- In 2D, the complexity of construction is $\mathcal{O}(N \log N)$ in time and $\mathcal{O}(N)$ in space

- In more than three dimensions exact decomposition becomes expensive and difficult to implement

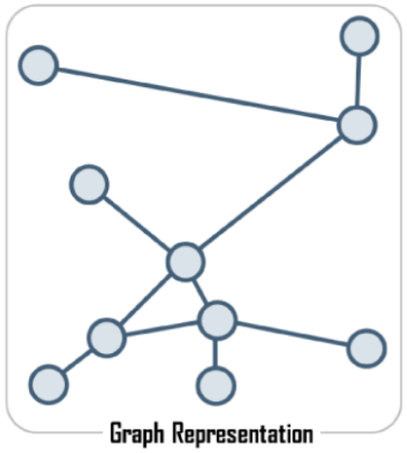


- Meshes can be used to define exact decompositions
- Reference points can be placed in the centers, on the edges, or at the vertices

Motion planning: Discretization of C-space

✓ **Discretization** of C-space

Combinatorial problems, Sampling problems



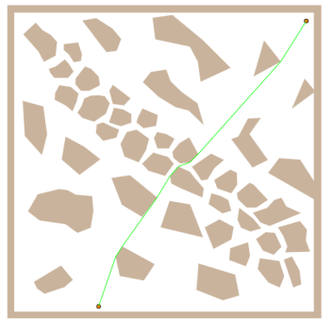
- **Roadmaps** ⇢
  - **Grids**
  - **Spatial decomposition**

- **Graph search techniques**

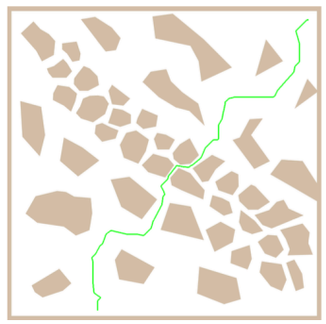
  Dijkstra, A*, D*, BFS, DFS, Gradients

Motion planning by C-space discretization: Summary so far

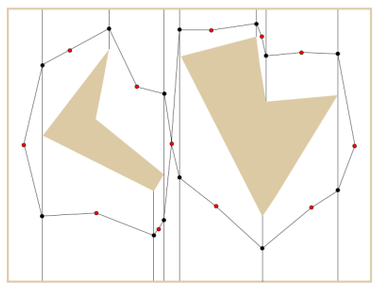
- **Roadmaps**
  - Visibility graph
    (complete, optimal as minimum cost, require polygonal obstacles)

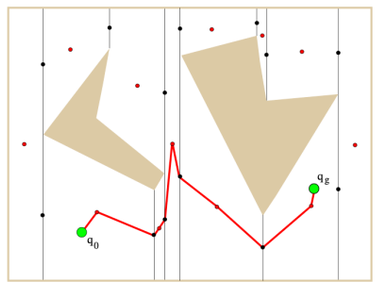


  - Generalized Voronoi skeleton
    (complete, optimal as max clearance)

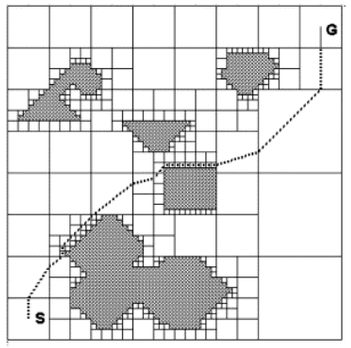


- **Grids**
- **Spatial decomposition**
  - Exact cell decomposition
    (complete, optimality based on navigation, require some regularity in obstacles' shape)





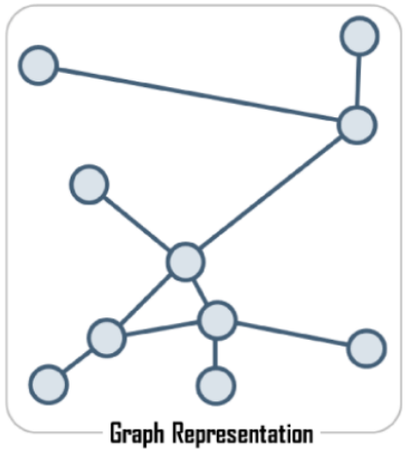
  - Approximate cell decomposition:
    uniform grids, quadtrees, ...
    (in general incomplete & non optimal, obstacles get approximated)



Motion planning: Discretization of C-space

✓ **Discretization** of C-space

Combinatorial problems, **Sampling problems**



- **Roadmaps** ⇢
  - **Grids**
  - **Spatial decomposition**

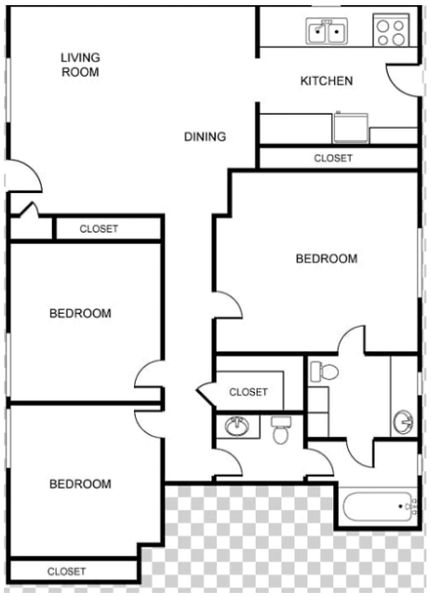
- **Graph search techniques**

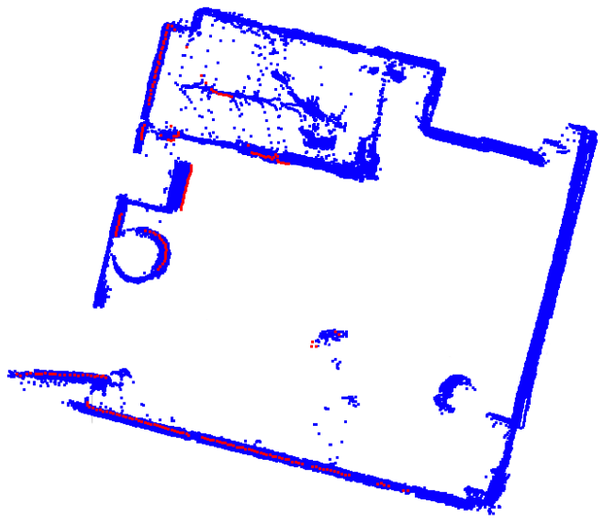
  Dijkstra, A*, D*, BFS, DFS, Gradients

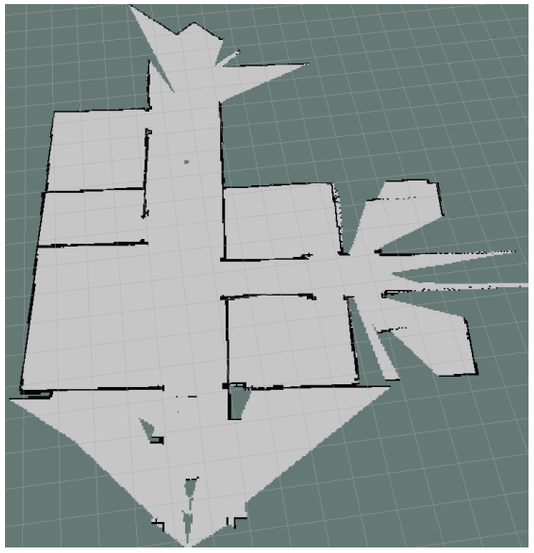
Recap: From combinatorial planning to sampling-based planning

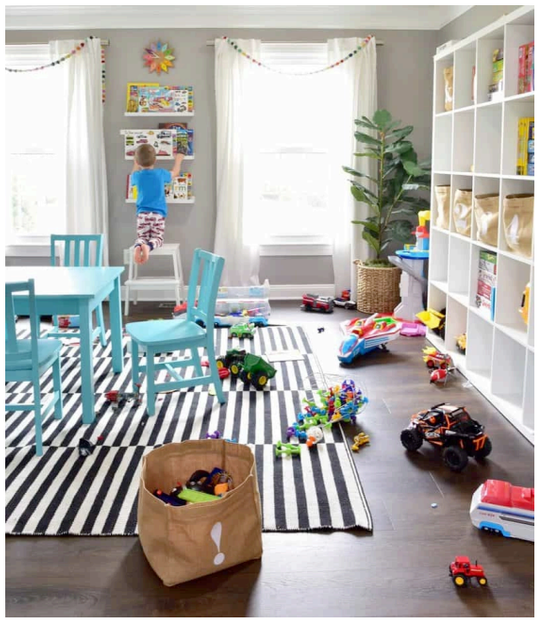
- (Deterministic) Combinatorial techniques are elegant, relatively easy to implement, and can enjoy completeness, but become quickly **intractable** when C-space dimensionality and/or number of obstacles increase (e.g., highly over actuated arm robots, highly cluttered scenarios)

- In a way or the other, they perform or require or assume some **approximation** of the real $\mathcal{C}_{obs}$
  - CAD maps are good and clean, but most likely do not reflect the actual scene
  - Maps from sensors are noisy
  - Objects have irregular shapes (might need to be embedded in regular geometries, e.g., polygons)





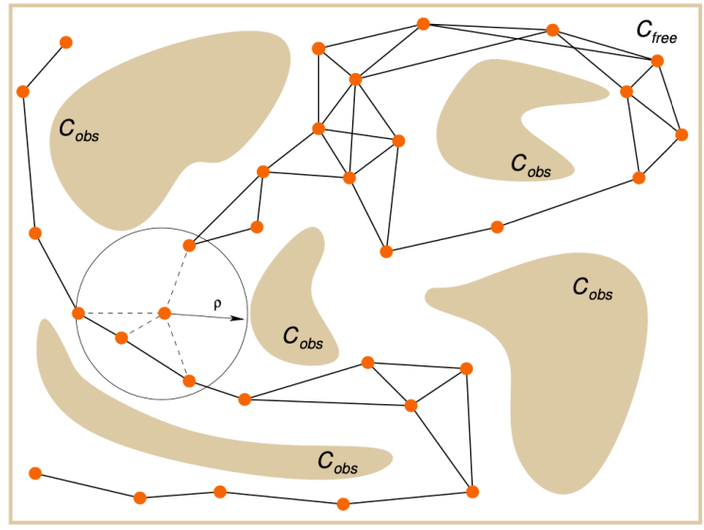




Sampling-based planning, weaker guarantees but more efficient in high dimensions

Recap: Sampling-based planning

- Abandon the concept of explicitly describe and model (exactly or approximately, yet, deterministically) $\mathcal{C}_{free}$ and $\mathcal{C}_{obs}$



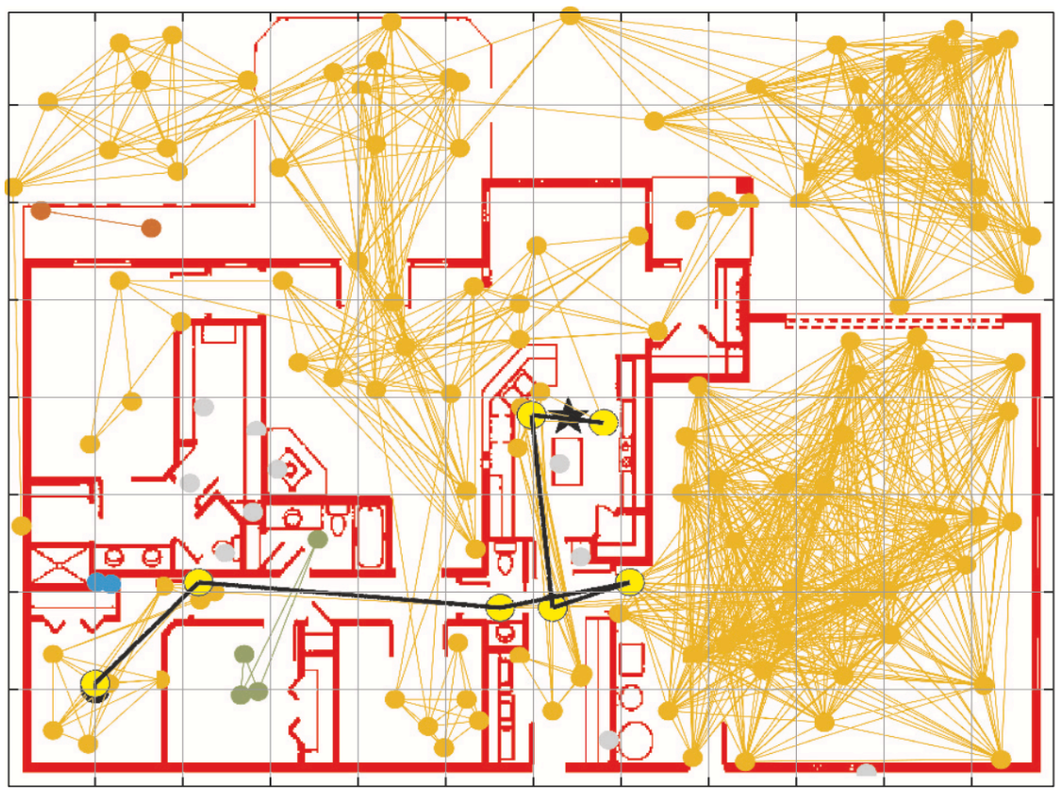
- Instead, rely on a sort of *blind* **exploration** of $\mathcal{C}_{free}$
  - Random **sampling** of configuration points from $\mathcal{C}$
  - Connect them to form a roadmap graph (incomplete)
  - Find the minimum cost path on the roadmap

- The only *light*, i.e., **informed source**, during the sampling process is provided by:
  - A **collision-detection** tool, that can probe the C-space to check whether a sampled configuration lies in or not in $\mathcal{C}_{free}$
  - A **local planner / navigation algorithm**, that checks whether a sampled configuration can be connected or not to another nearby configuration by a admissible path

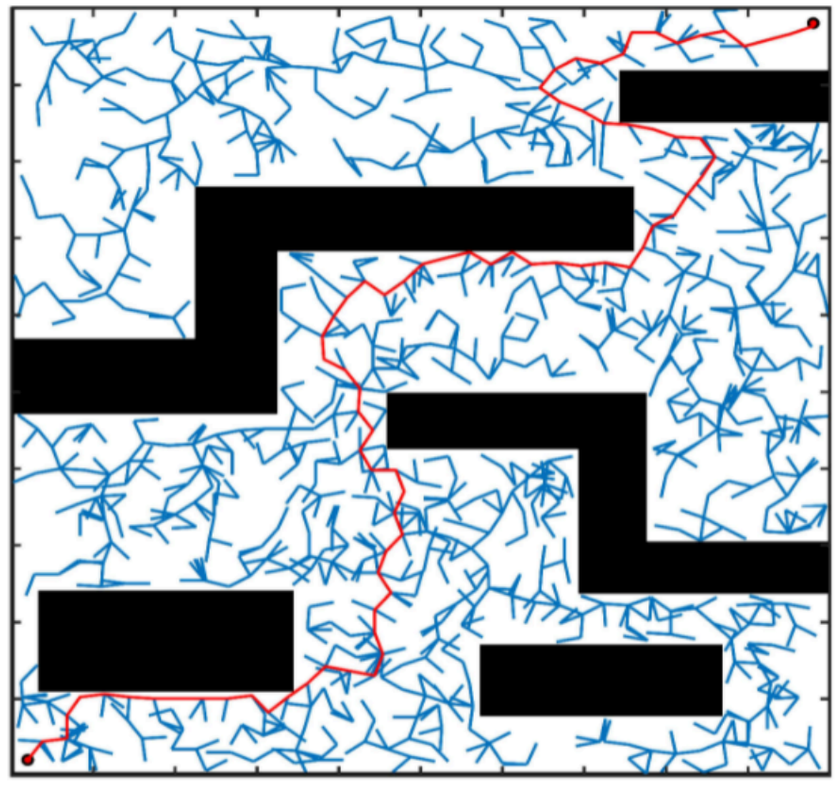
    1. In the free space, a geometric path exists that connects the two configurations
    2. The path can be feasibly executed by the robot given its differential constraints (account for **non-holonomic** robots)

Sampling-based planning

Probabilistic roadmaps (PRM)



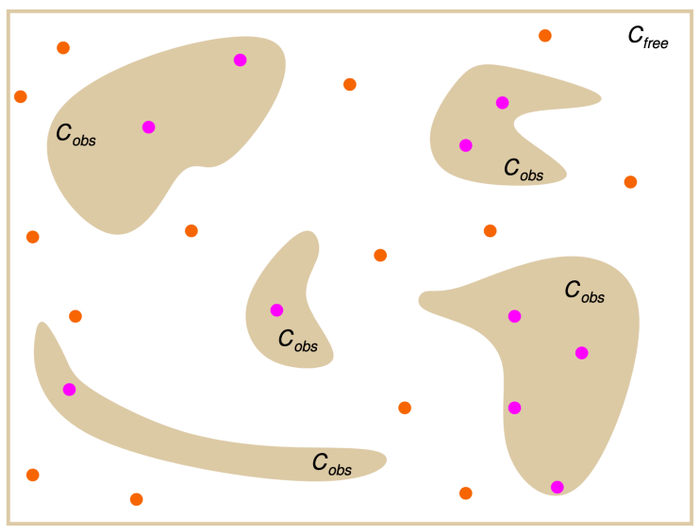
Rapidly-exploring Random Trees (RRT)

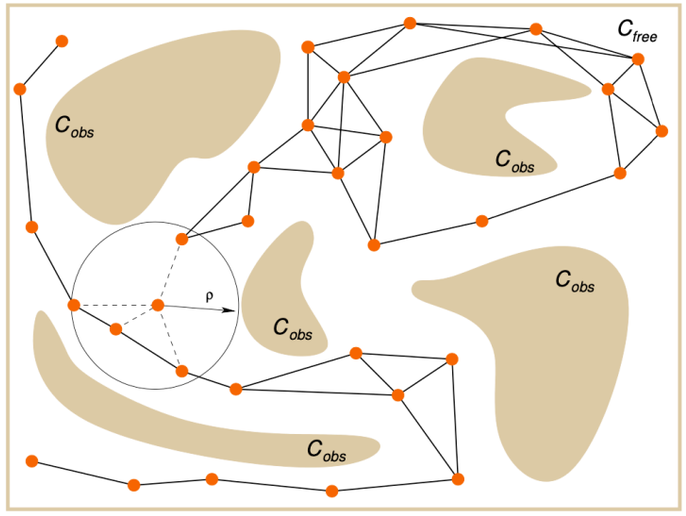


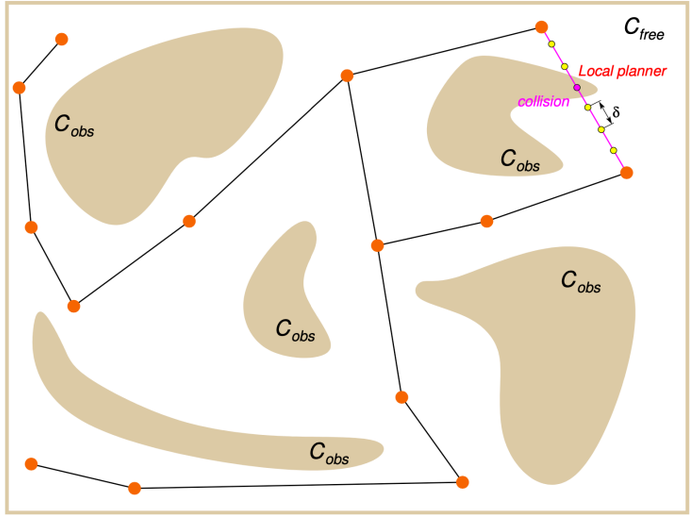
Construction of Probabilistic road maps (PRM)

1. **Start** with an empty road map graph, $G(V, E)$, where $G = \emptyset$, $E = \emptyset$
2. **Generate** one **random sample** q (i.e., a random configuration) from $\mathcal{C}$ (or a batch of samples)
3. **Add** q to V if $q \in \mathcal{C}_{free}$ (use **collision-detection** tool)
4. **Select one set of vertices** in V that are neighbors to q: k-Nearest neighbors or *all* neighbors within a specified radius ρ (note that the notion of proximity on a topological manifold needs to be defined)
5. Try to **connect** the configuration q to the neighbor vertices q_i $\forall i$, using a **local planner**:
   - Holonomic robot: local planner checks if the line-of-sight $q \rightarrow q_i$ is collision free (at resolution steps δ)
   - Non-Holonomic robot: the local planner checks if the neighbor vertex q_i can be reached with a collision-free trajectory compliant with robot's non-holonomic constraints

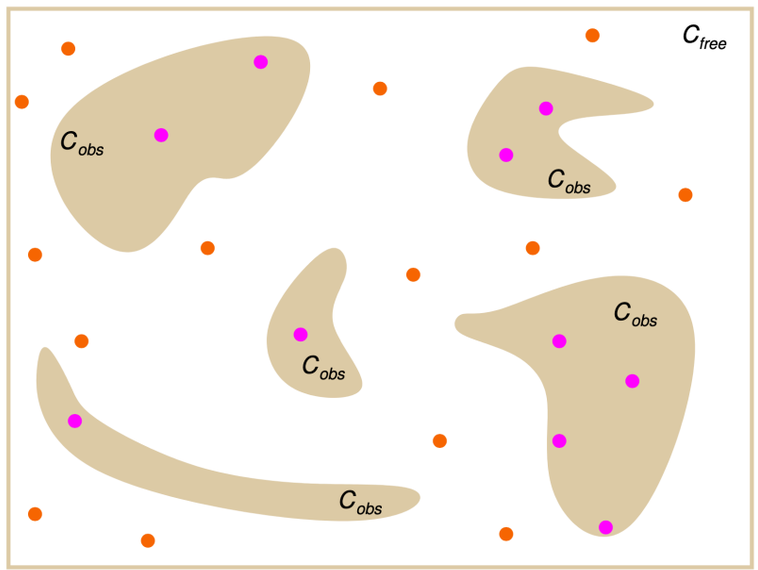
   Add edge (q, q_i) to E if the two vertices could get connected
6. **Iterate** 2-5, keeping adding vertices and edges to the road map graph G until it gets dense enough (or a predefined number of vertices/edges is added)

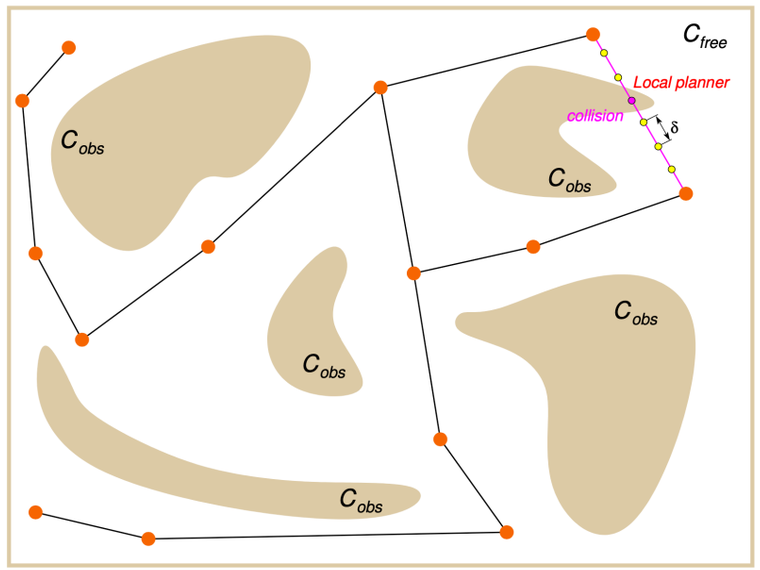


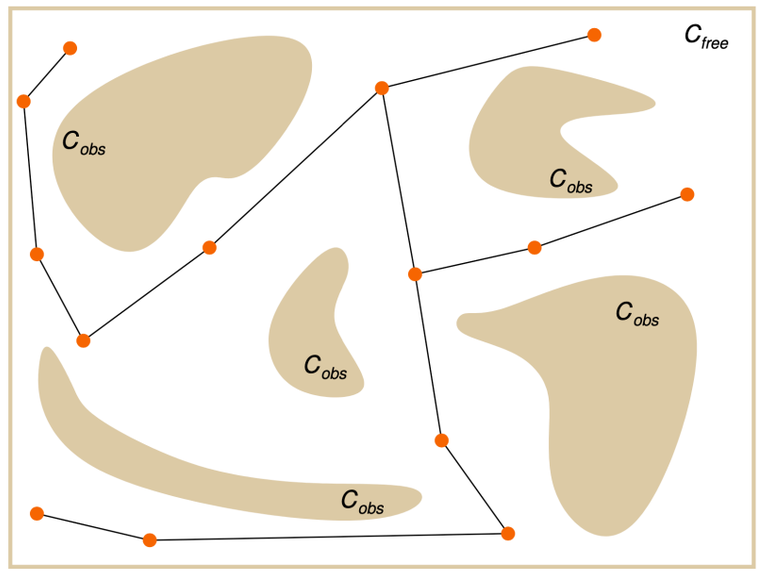


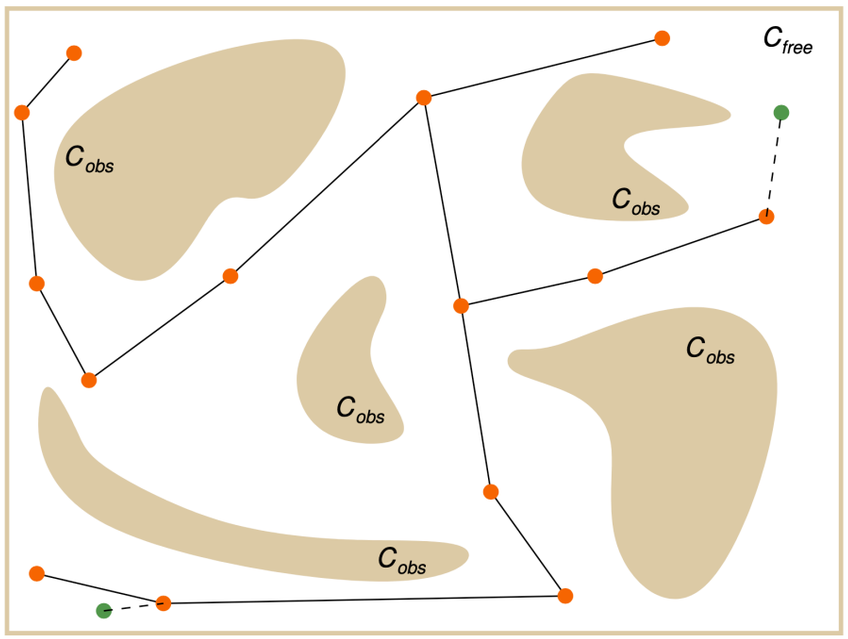


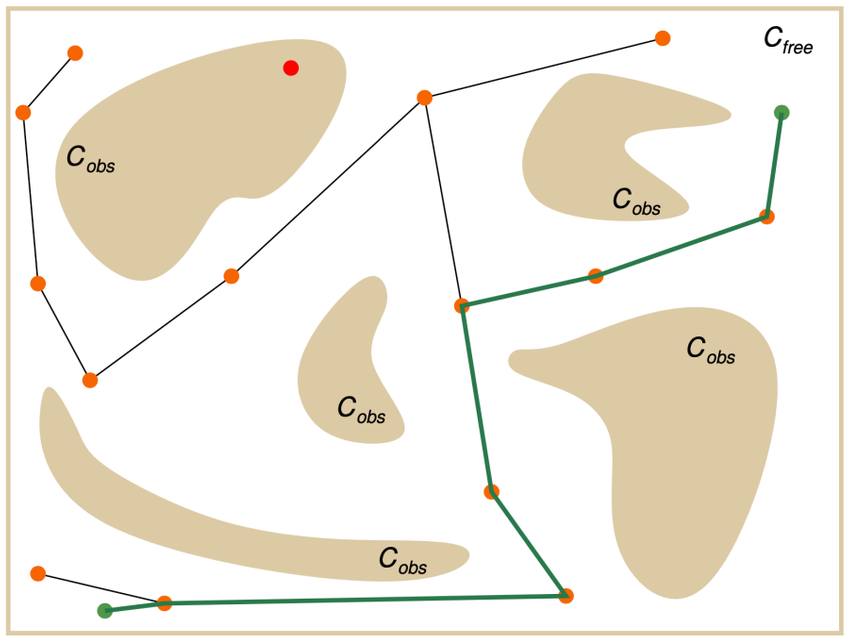
Construction of Probabilistic road maps (PRM)



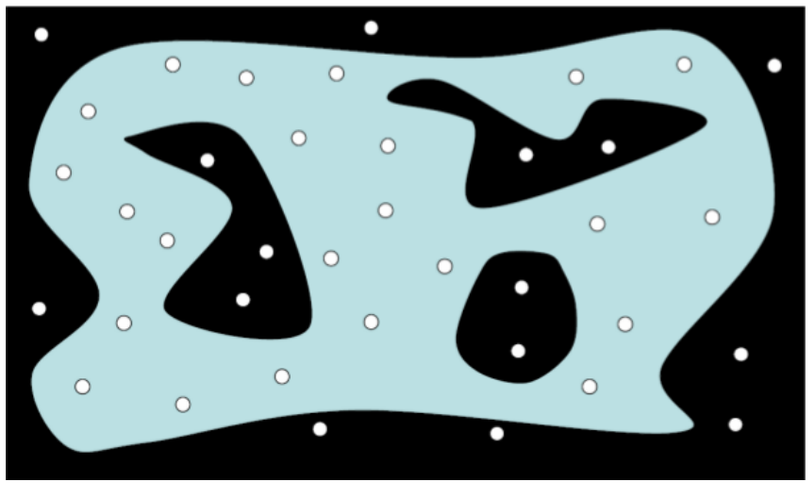




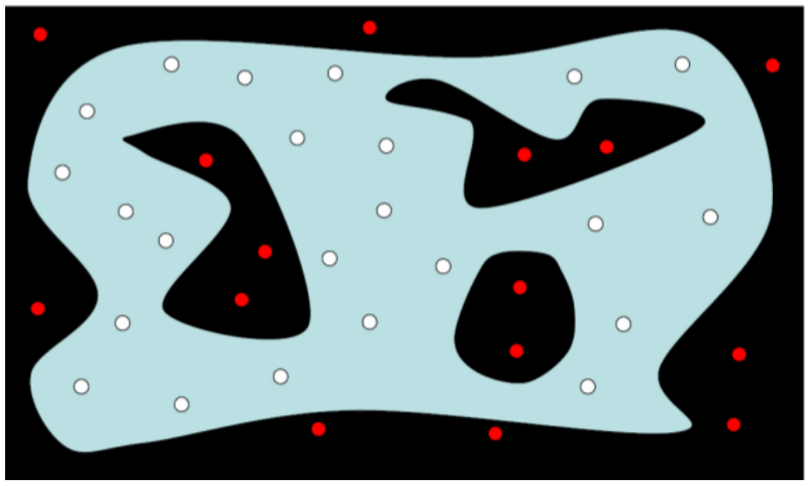




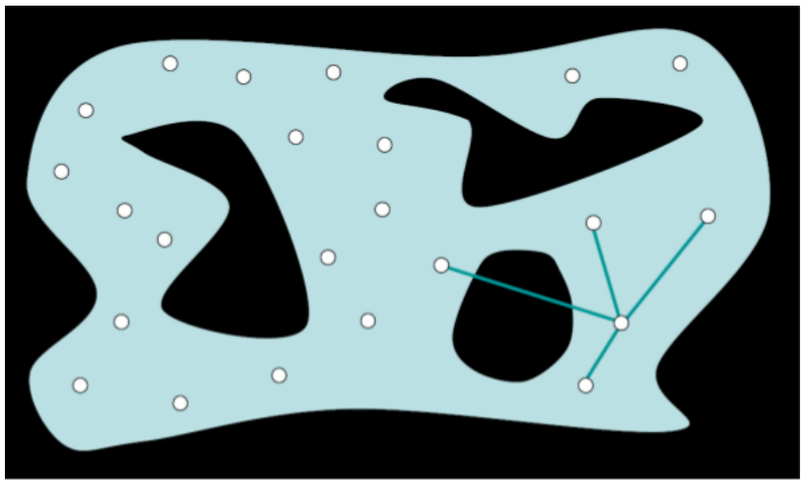
Construction of Probabilistic road maps (PRM), batch mode



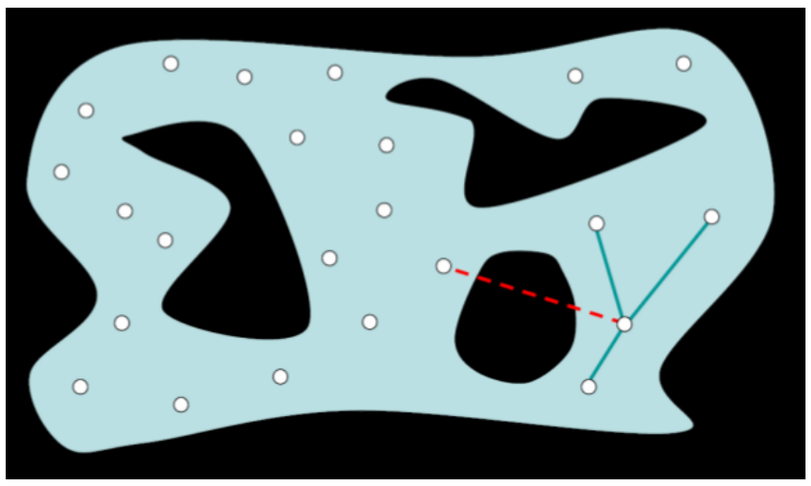
Sample random configurations



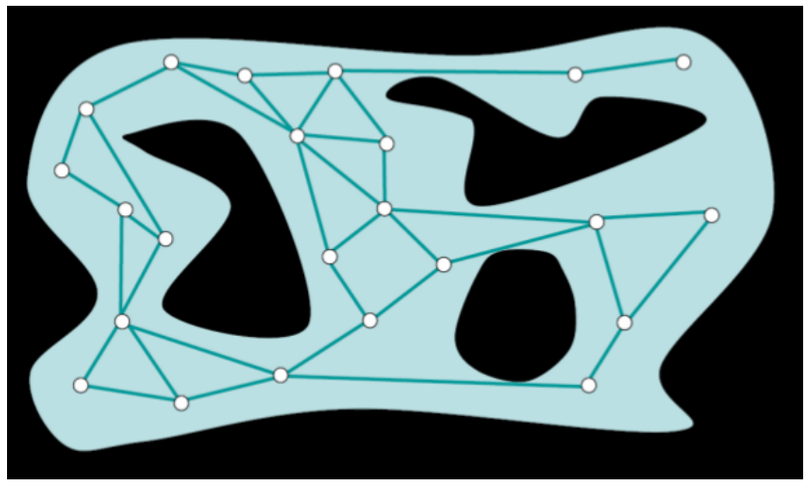
Remove the configurations that are not in $\mathcal{C}_{free}$



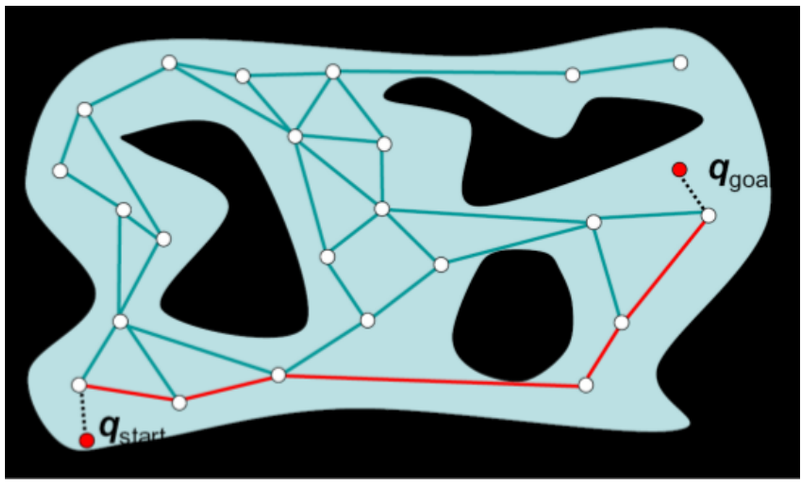
Try to connect each sample with selected nearby samples



Remove links intersecting $\mathcal{C}_{obs}$



Construct the road map graph

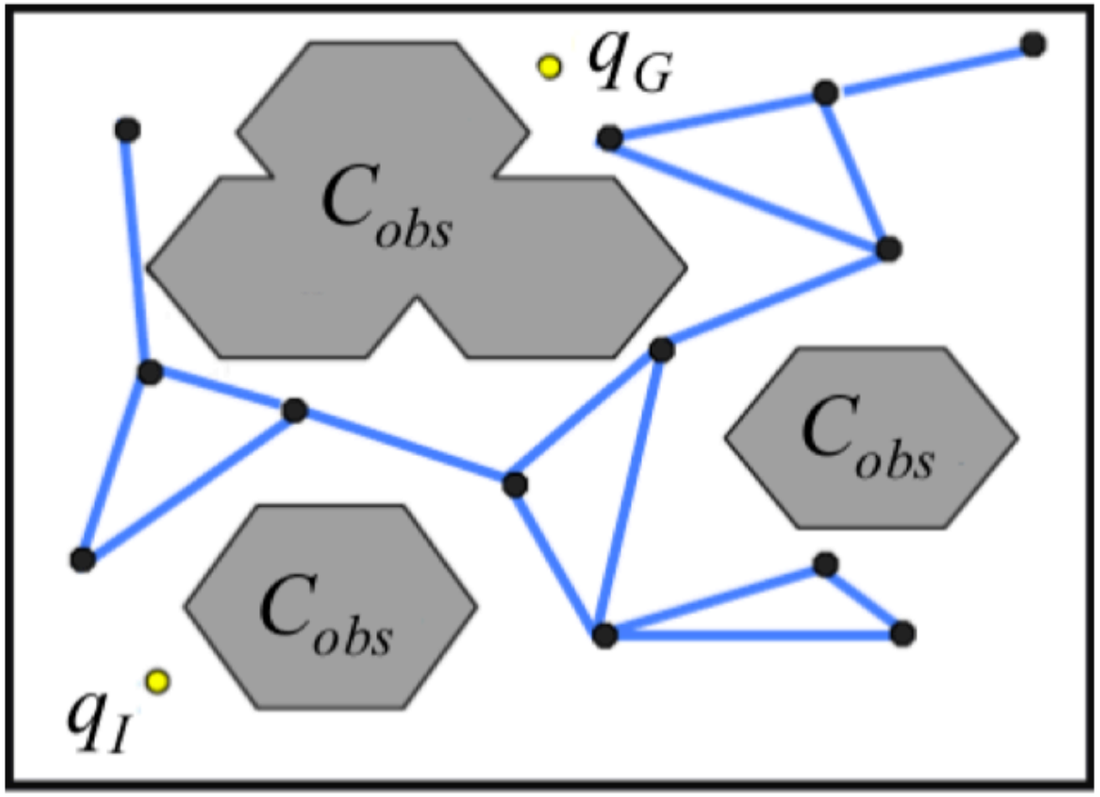


Connect the given start and goal configurations to the road map, and use a graph search algorithm (e.g., A*) to compute the best path between them

Probabilistic road maps: connecting to the query's points

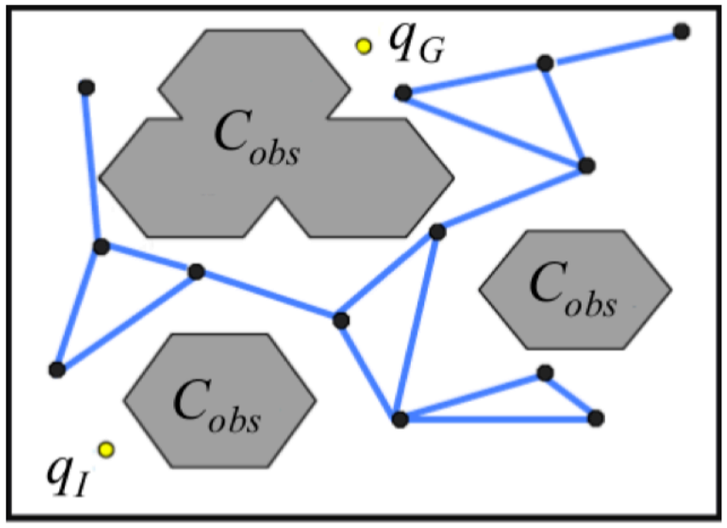
- How to connect the start and the goal configurations, q_I, q_G specified by the current **query** to the road map? E.g., look for the points on the graph at *minimal distance* from q_I and q_G, or perform a number of *random walks* with the local planner from both q_I and q_G trying to reach the graph.

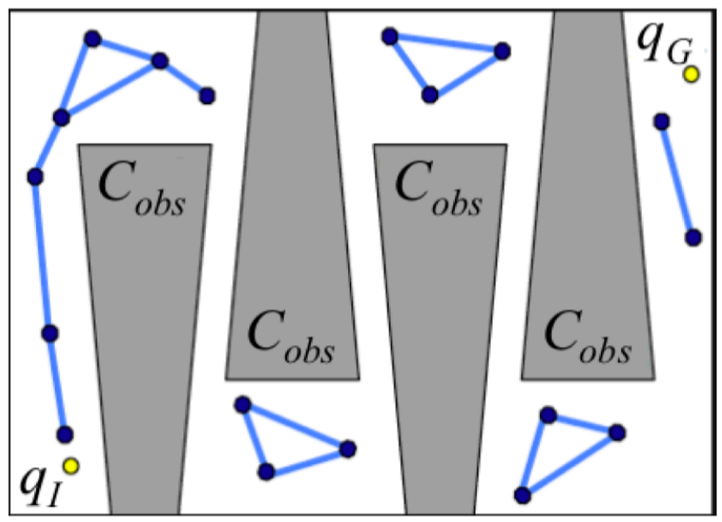
☑ A PRM can handle multiple queries



Properties of probabilistic road maps

- ➡ PRMs do not explicitly construct C-space
  - C-space is only used to check if a configuration belongs to C_free or not and if two configurations can be connected (via an admissible velocity trajectory) in C_free
- Problems with narrow passages





- ○ Neither complete nor optimal
- ☑ Probabilistically complete

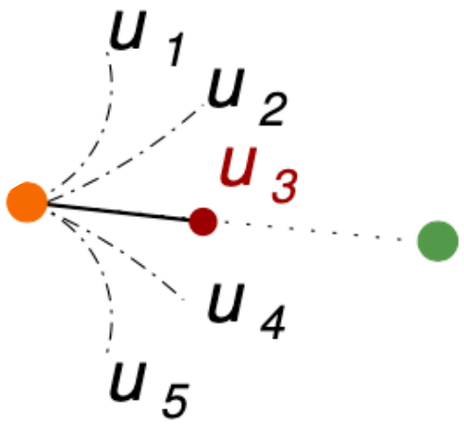
- ✓ Easily apply to high-dimensional problems
- ✓ Handle multiple queries
- ✓ Handle non-holonomic robots (via local planner)
- ✓ A lot of powerful extensions exists, quite popular!

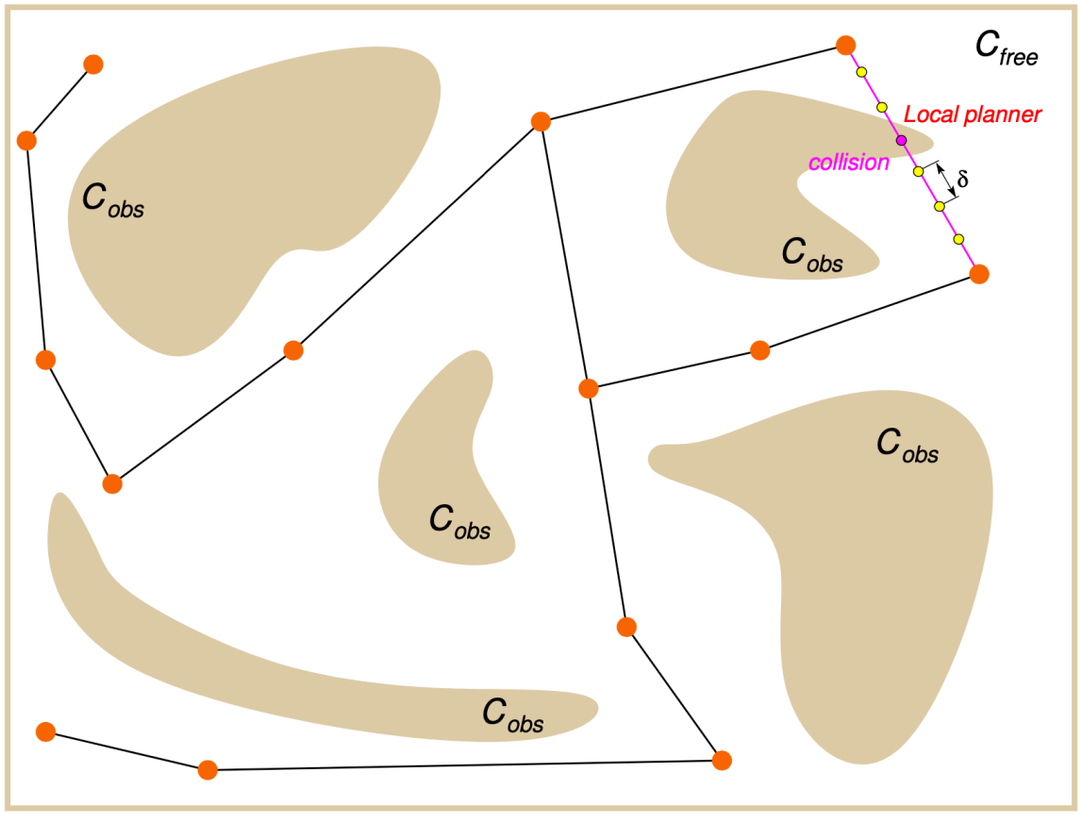
Local planner and neighbor selection

Local planner:

- A core ingredient in all sampling-based methods: it must locally construct a path that is feasible for the robot given its holonomic and non-holonomic constraints
- Define paths that the robot can perform *in practice* for moving between two selected configurations. Limited by a resolution step δ

➡ The **control inputs** u_i that are used to drive the robot along the path can be attached to the edge, allowing for a fast and reliable plan execution for the actual robot after the query

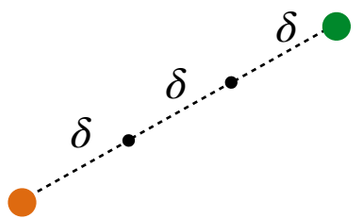




$(v_1, \omega_1), (v_2, \omega_2), \ldots, (v_n, \omega_n)$

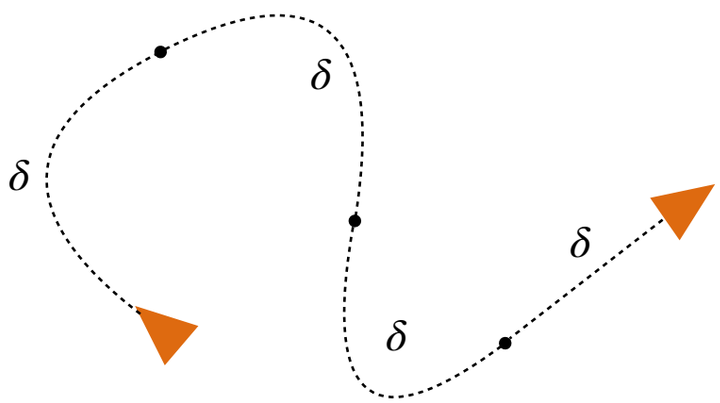
Local planner

- E.g, for a *omnidirectional, holonomic robot* the local planner can be implemented as a controller moving the robot along the straight line between the two configuration points



- E.g., for a *differential drive robot*, the local planner can locally rotate the robot in the direction of the target point in $\mathbb{R}^2$ and then drive it along a straight line

- E.g., for a *car-like, non-holonomic robot*, a local planner that would connect the two given configurations with a straight-line segment (in C-space) is not a suitable one, since it would potentially violate the non-holonomic constraint that do not allow motion perpendicular to the wheels.



- In general, the local planner must define a *feasible motion* and be *computationally fast*, since it has to be called for each sampled point × the number of selected neighbors.

Local planner

- Time efficient local planner → Implement robot's dynamics as simply as possible
  - **Holonomic robots** → **Generic local planner:** given two configurations, it connects them by a **straight line** segment in C-space and checks this line segment for collision.
  - **Non-holonomic robots** → The control inputs $\boldsymbol{u}$ that would define a feasible and fast, approximation of the local path must be devised **case by case** based on robot's constraints. The controls $\boldsymbol{u}$ are attached to each edge, to later permit a fast and coherent execution of the path.

- Avoid to waste computations → The probability that the local paths intersect with obstacles must be low to (i.e., calls to the local planner that do not result in new edges on the graph) → Consider short distances (on the manifold) for neighbor selection, and/or sample the points in confined volumes around selected current points, and/or avoid to connect nodes that already have connection edges (they are already in the graph)

Example of local planner for a non-holonomic robot

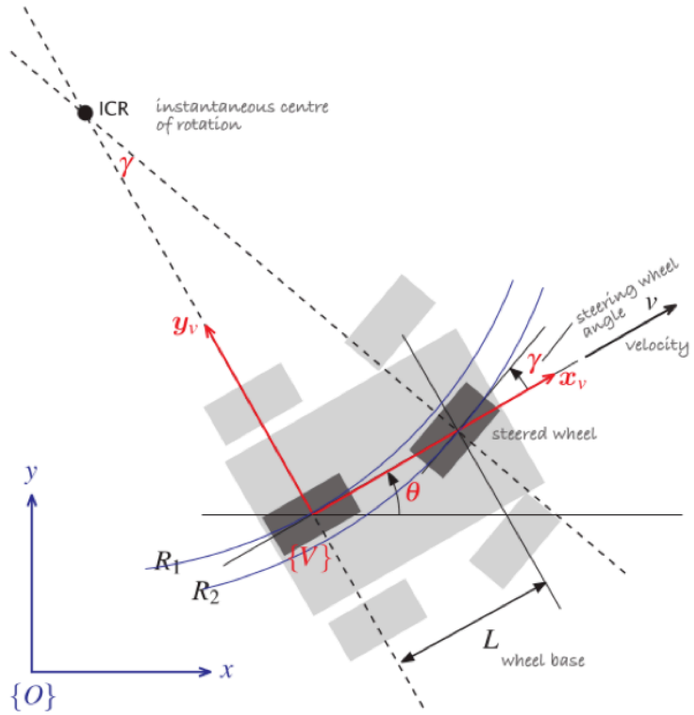
- A *proper* (but rather expensive, since the kinematic model is relatively hard to control) way to implement a local planner for a *car-like robot*:

  1. Use the **kinematic equations** in the configuration space $\mathbb{R}^2 \times \mathbb{S}$:
     $\dot{x} = v_t \cos\theta_t,\ \ \dot{y} = v_t \sin\theta_t,\ \ \dot{\theta} = \frac{v_t}{L} \tan\gamma_t$, where v is the rear wheels' velocity and γ is the steering angle of the front wheels

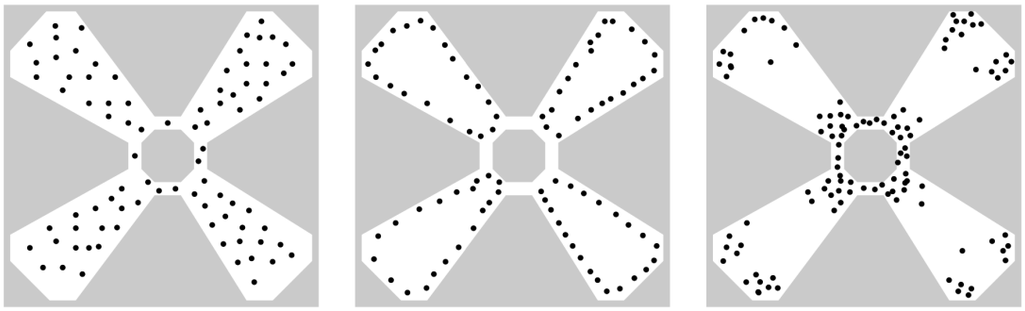
  2. Starting from the currently sampled configuration $\begin{bmatrix} x_t & y_t & \theta_t \end{bmatrix}^T$, **apply a sequence of n control inputs** $\boldsymbol{u}_i,\ \ i = t, t + \Delta t, t + 2\Delta t, \ldots, t + n\Delta t$, for v and γ.
     The controls are aimed at connecting the sampled configuration to a selected neighbor.

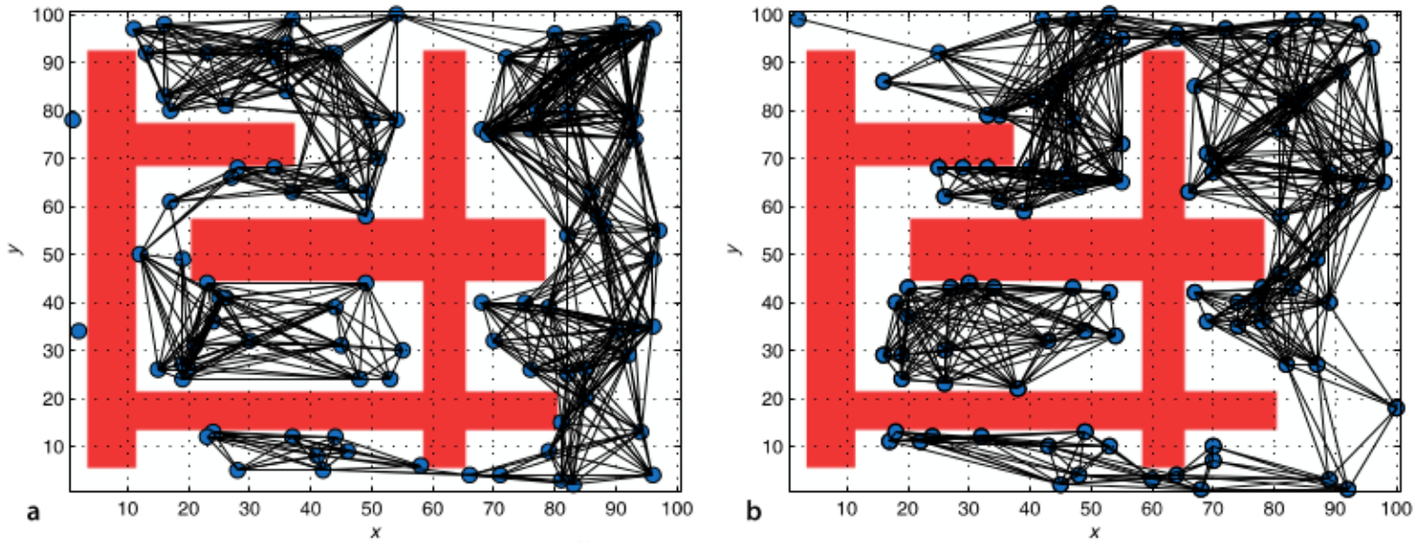
  3. **Numerically integrate the motion equations** (e.g., using Runge-Kutta) with the applied controls, for getting the robot to the desired neighbor configuration.

  4. **Report a failure if an obstacle is encountered** while executing the path, success is the neighbor configuration is eventually reached.



Sampling strategy is important!





- How to **uniformly sample** $\mathcal{C}$? This is not trivial given its usually complex topology
- Sample more $\rightarrow$ near points with few neighbors
- Sample more $\rightarrow$ close to the obstacles
- Sample more $\rightarrow$ disconnected areas
- Sample more $\rightarrow$ where the local planner fails more frequently (*difficult areas*)
- Use pre-computed sequence of samples
- ...

- Remarkably, a solution can be often found by using *relatively few randomly sampled points*
- In most problems, a relatively small number of samples is sufficient to cover most of the feasible space with high probability
- For a large class of problems the **probability of finding a path goes to 1 exponentially with the number of samples** ... but cannot detect that a path does not exist

Rapidly exploring random trees (RRT)

Rapidly Exploring Random Trees:

Aggressively probe and explore the C-space by incrementally expanding the sampling process from an initial configuration q_0

A random tree rooted at q_0 is incrementally generated, marking the explored region of the $\mathcal{C}$ space.

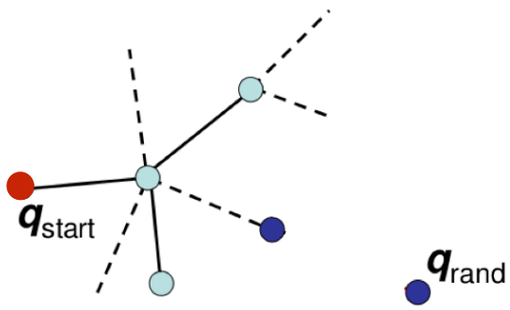
q_0 is set in q_{start}, with the tree which (hopefully) grows towards q_{goal}.

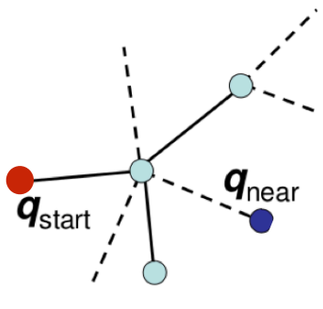
To improve the efficiency **two trees can be made growing**, one seeded in q_{start} and the other seeded in q_{goal} with the aim of meeting and merging the two trees.

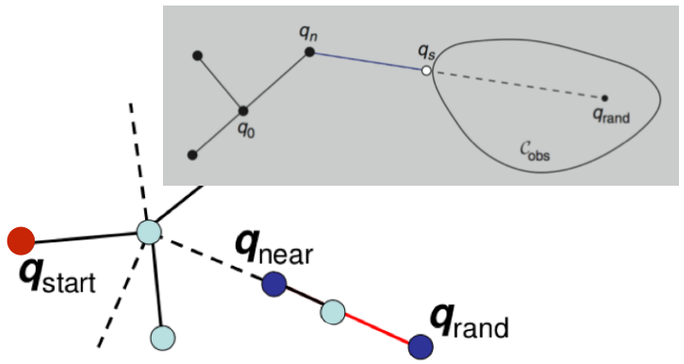
45 iterations

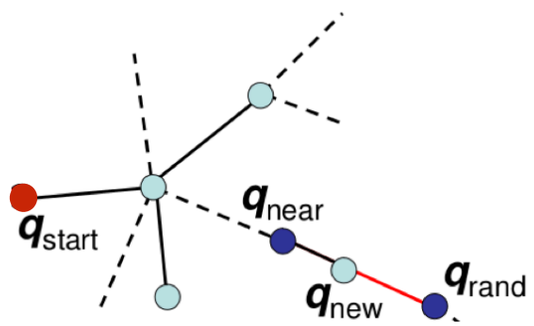
2345 iterations

Rapidly exploring random trees (RRT)









Input: Initial configuration q_{start}, termination criterion, **growth factor** Δq

1. Root the tree R in q_{start}
2. Generate a **random configuration** $q_{rand} \in \mathcal{C}_{free}$
3. Use a **distance criterion** (on the C-space manifold) to find the vertex $q_{near} \in R$ which is the *closest* to q_{rand}
4. From q_{near}, **move an incremental distance** Δq in direction of q_{rand} using a **local planner**, that applies a sequence of control inputs $\boldsymbol{u}_{near \to rand}$ compliant with robot's motion constraints
5. The local planner stops either after moving the robot for an incremental distance Δq over a **collision-free path**, or as soon as a collision is detected. In either cases, the final point in $\mathcal{C}_{free}$ reached, indicated with q_{new}, is **added to vertex set of** R and the edge (q_{near}, q_{new}) between q_{near} and q_{new} is **added to the edge set of** R
6. **Attach the applied control inputs** $\boldsymbol{u}_{near \to rand}$ to the edge (q_{near}, q_{new}) (to speedup execution)
7. If the termination criterion is not satisfied, go to 2 and iterate the process

Algorithm for generating an RRT with n nodes

Build_RRT(q_{start}, N, Δq)

R.start(q_{start});

for k = 1 to N

$q_{rand} \leftarrow$ RANDOM_CONFIGURATION($\mathcal{C}_{free}$);

$q_{near} \leftarrow$ NEAREST_VERTEX(q_{rand}, R);

$q_{new}, u_{near \to new} \leftarrow$ NEW_CONFIGURATION($q_{near}, q_{rand}, \Delta q, LocalPlanner()$);

R.add_vertex(q_{new});

R.add_edge(q_{near}, q_{new});

R.add_controls((q_{near}, q_{new}), $u_{near \to new}$);

Return R;

Algorithm 3: 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**

    $V \leftarrow V \cup \{x_{\text{new}}\}$; $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$;

return $G = (V, E)$;

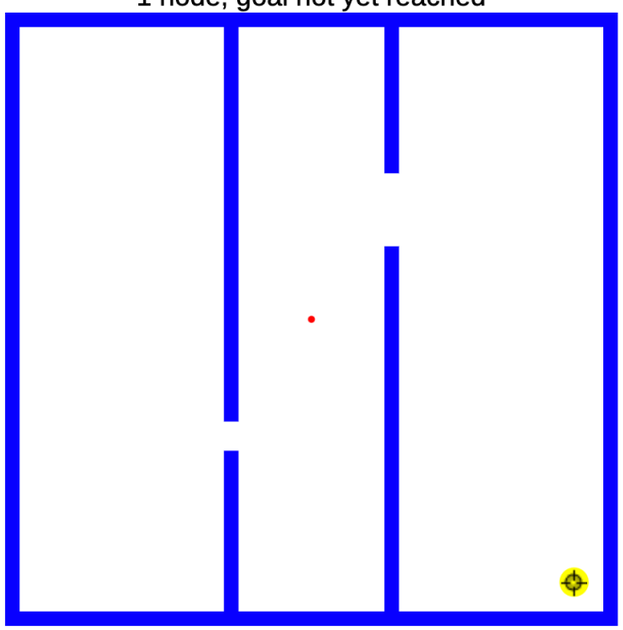
S. Karaman and E. Frazzoli, *Sampling-Based Algorithms for Optimal Motion Planning*, The International Journal of Robotics Research, 30(7), pp. 846-894, 2011

- For implementation issues and optimization actions, check the paper:

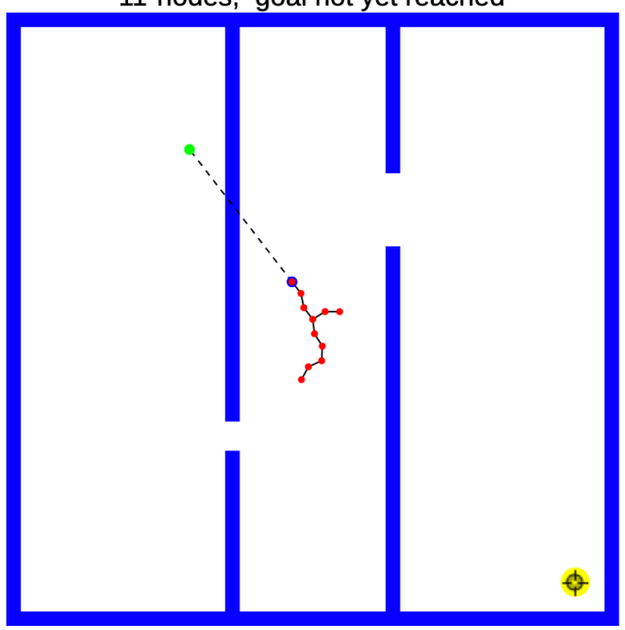
- I. Sucan and L. Kavraki, *On the implementation of single-query sampling-based motion planners*, Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), 2010

RRT at work

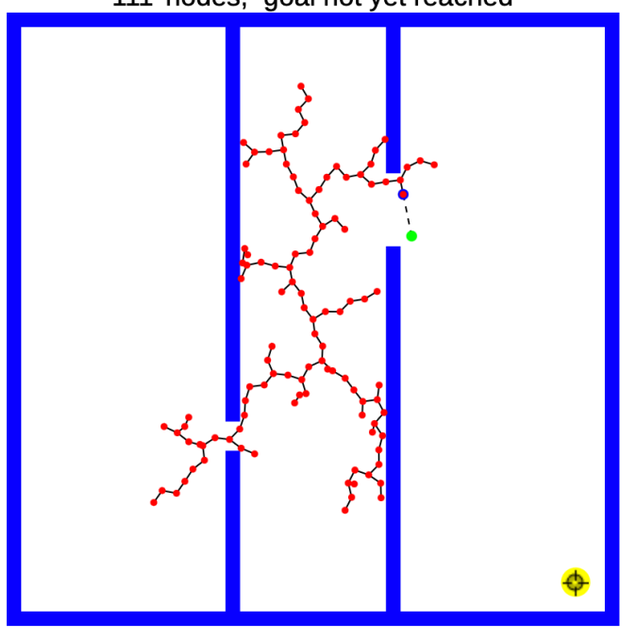
1 node, goal not yet reached



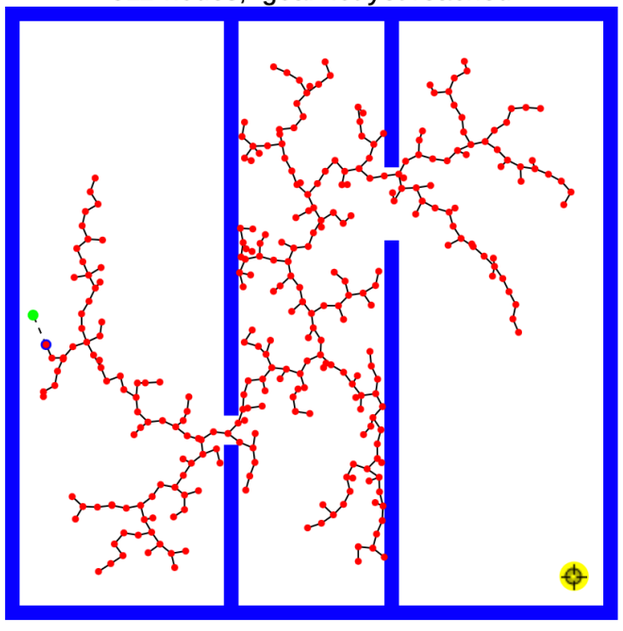
11 nodes, goal not yet reached



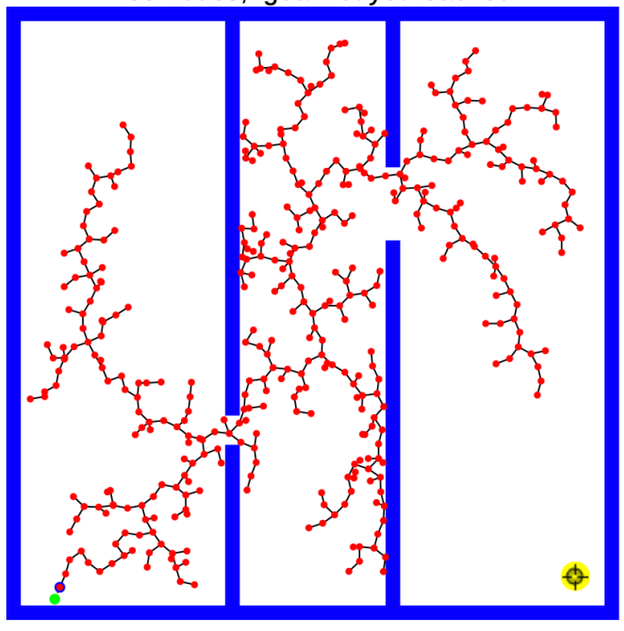
111 nodes, goal not yet reached



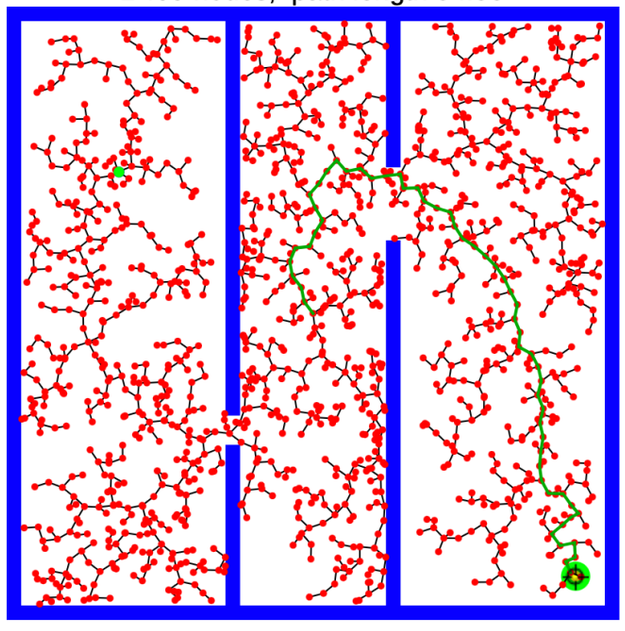
311 nodes, goal not yet reached



409 nodes, goal not yet reached



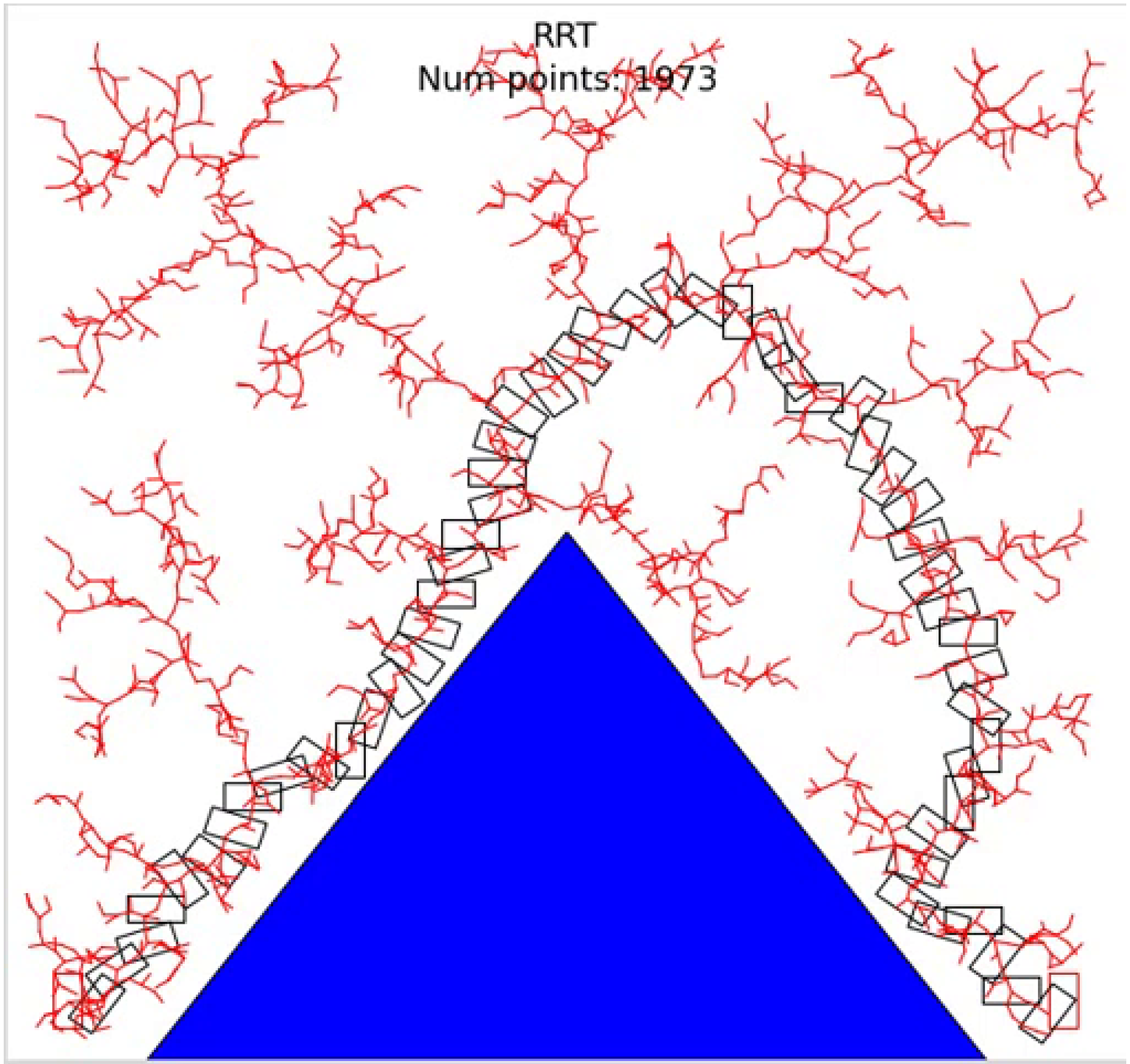
1 409 nodes, path length 54.58



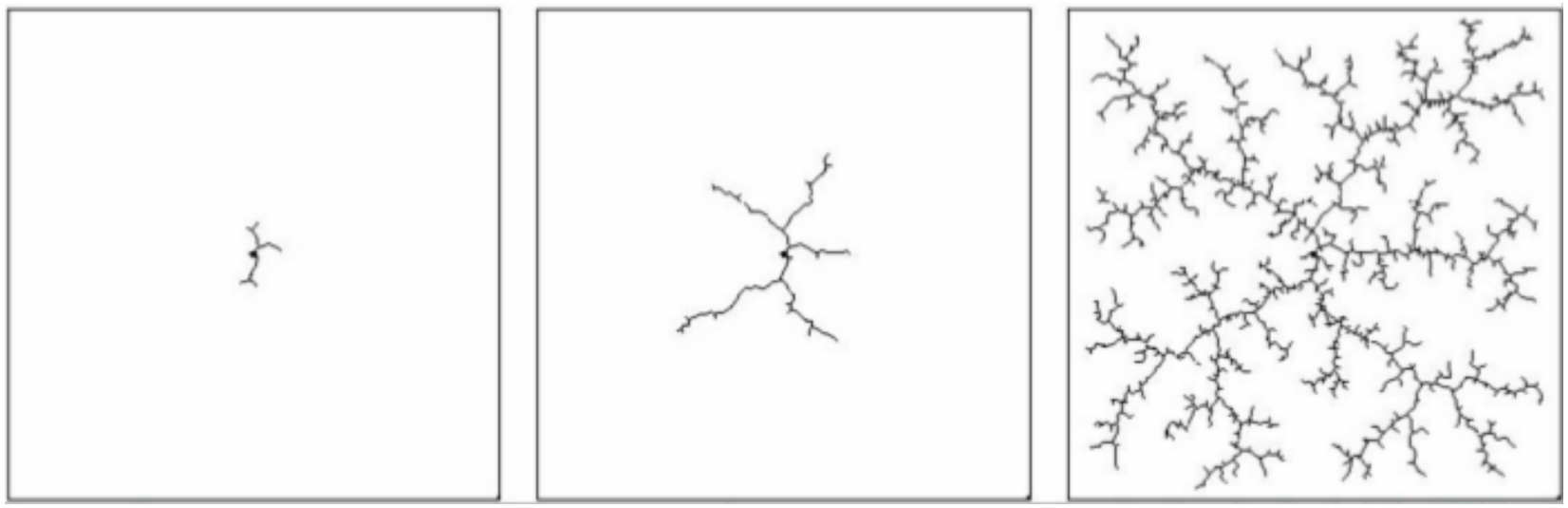
Play with this online app!

https://www.wolframcloud.com/objects/demonstrations/RapidlyExploringRandomTreeRRTAndRRT-source.nb

RRT at work



RRT properties



- Tends to explore the space rapidly in all directions, since it has a bias towards point with large Voronoi regions
- Does not require extensive pre-processing
- Designed for single query problems
- It has been combined with PRMs to be used for multiple queries
- Needs only collision detection test + Local planner
- No need to represent/pre-compute the entire C-space
- Highly popular, many extensions exist: real-time RRTs, anytime RRTs, for dynamic environments etc.
- Good balance between greedy and exploration
- Easy to implement
- Sensitive to the distance metric
- Unknown rate of probabilistic convergence

RRT* and its variants for getting optimal or near-optimal paths

- It has been shown that, under 'mild technical conditions', the cost of the best path in the RRT converges almost surely to a **non-optimal value**.

➡ Many variants of the RRT have been developed that asymptotically converge to an optimum, or near an optimum.

➡ The RRT* algorithm is a variation on the single-tree RRT that continually rewires the search tree to ensure that it always encodes the shortest path from start to each node in the tree.

- Rapidly-exploring random graph (RRG) and RRT*,[9][10][11] a variant of RRT that converges towards an optimal solution
- RRT+, [12] a family of RRT-based planners aiming to generate solutions for high-dimensional systems in real-time, by progressively searching in lower-dimensional subspaces.
- RRT*-Smart,[13] a method for accelerating the convergence rate of RRT* by using path optimization (in a similar fashion to Theta*) and intelligent sampling (by biasing sampling towards path vertices, which – after path optimization – are likely to be close to obstacles)
- A*-RRT and A*-RRT*,[14] a two-phase motion planning method that uses a graph search algorithm to search for an initial feasible path in a low-dimensional space (not considering the complete state space) in a first phase, avoiding hazardous areas and preferring low-risk routes, which is then used to focus the RRT* search in the continuous high-dimensional space in a second phase
- RRT*FN,[15][16][17] RRT* with a fixed number of nodes, which randomly removes a leaf node in the tree in every iteration
- RRT*-AR,[18] sampling-based alternate routes planning
- Informed RRT*,[19][20] improves the convergence speed of RRT* by introducing a heuristic, similar to the way in which A* improves upon Dijkstra's algorithm
- Real-Time RRT* (RT-RRT*),[21] a variant of RRT* and informed RRT* that uses an online tree rewiring strategy that allows the tree root to move with the agent without discarding previously sampled paths, in order to obtain real-time path-planning in a dynamic environment such as a computer game
- RRT^X and RRT$^\#$,[22][23] optimization of RRT* for dynamic environments
- Theta*-RRT,[24] a two-phase motion planning method similar to A*-RRT* that uses a hierarchical combination of any-angle search with RRT motion planning for fast trajectory generation in environments with complex nonholonomic constraints
- RRT* FND,[25] extension of RRT* for dynamic environments
- RRT-GPU,[26] three-dimensional RRT implementation that utilizes hardware acceleration
- APF-RRT,[27] a combination of RRT planner with Artificial Potential Fields method that simplify the replanning task
- CERRT,[28] a RRT planner modeling uncertainty, which is reduced exploiting contacts
- MVRRT*,[29] Minimum violation RRT*, an algorithm that finds the shortest route that minimizes the level of unsafety (the "cost" of the environment rules that have been violated, e.g. traffic laws)
- RRT-Blossom,[30] RRT planner for highly constrained environments.
- TB-RRT,[31] Time-based RRT algorithm for rendezvous planning of two dynamic systems.
- RRdT*,[32][33] a RRT*-based planner that uses multiple local trees to actively balances the exploration and exploitation of the space by performing local sampling.

From Wikipedia

RRT vs. RRT*

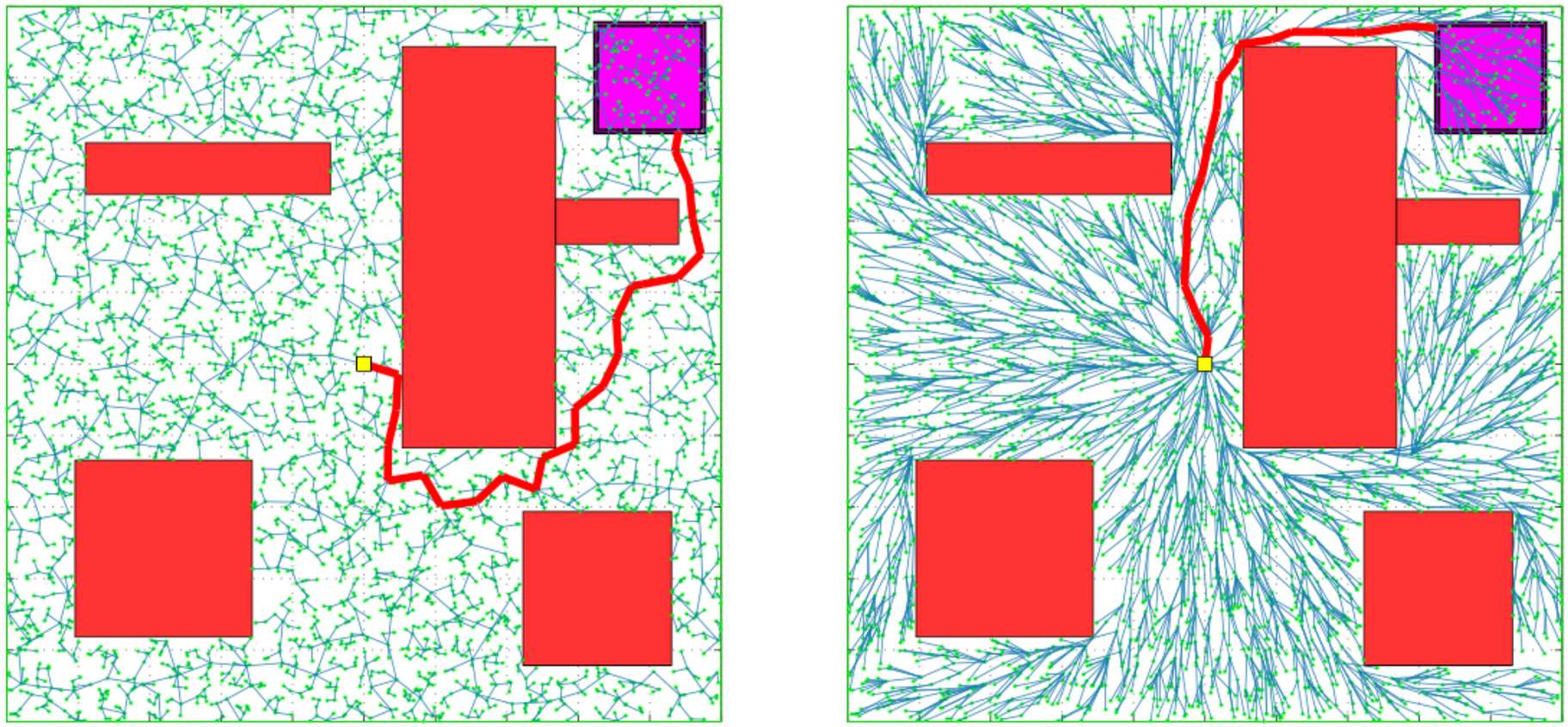


Figure 10.19: (Left) The tree generated by an RRT after 5,000 nodes. The goal region is the square at the top right corner, and the shortest path is indicated. (Right) The tree generated by RRT* after 5,000 nodes. Figure from [67] used with permission.

From: S. Karaman and E. Frazzoli, *Sampling-Based Algorithms for Optimal Motion Planning*, The International Journal of Robotics Research, 30(7), pp. 846-894, 2011

RRT* and its variants for getting optimal or near-optimal paths

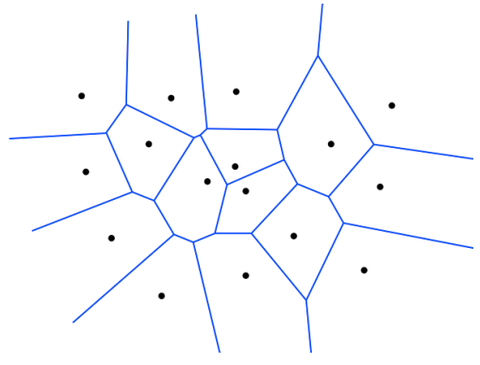
Table 1. Summary of results. Time and space complexity are expressed as a function of the number of samples n, for a fixed environment.

	Algorithm	Probabilistic completeness	Asymptotic optimality	Monotone convergence	Time complexity		Space complexity
					Processing	Query	
Existing algorithms	PRM	Yes	no	Yes	$O(n \log n)$	$O(n \log n)$	$O(n)$
	sPRM	Yes	Yes	Yes	$O(n^2)$	$O(n^2)$	$O(n^2)$
	k-sPRM	Conditional	No	No	$O(n \log n)$	$O(n \log n)$	$O(n)$
	RRT	Yes	No	Yes	$O(n \log n)$	$O(n)$	$O(n)$
Proposed algorithms	PRM* k-PRM*	Yes	Yes	No	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
	RRG k-RRG	Yes	Yes	Yes	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
	RRT* k-RRT*	Yes	Yes	Yes	$O(n \log n)$	$O(n)$	$O(n)$

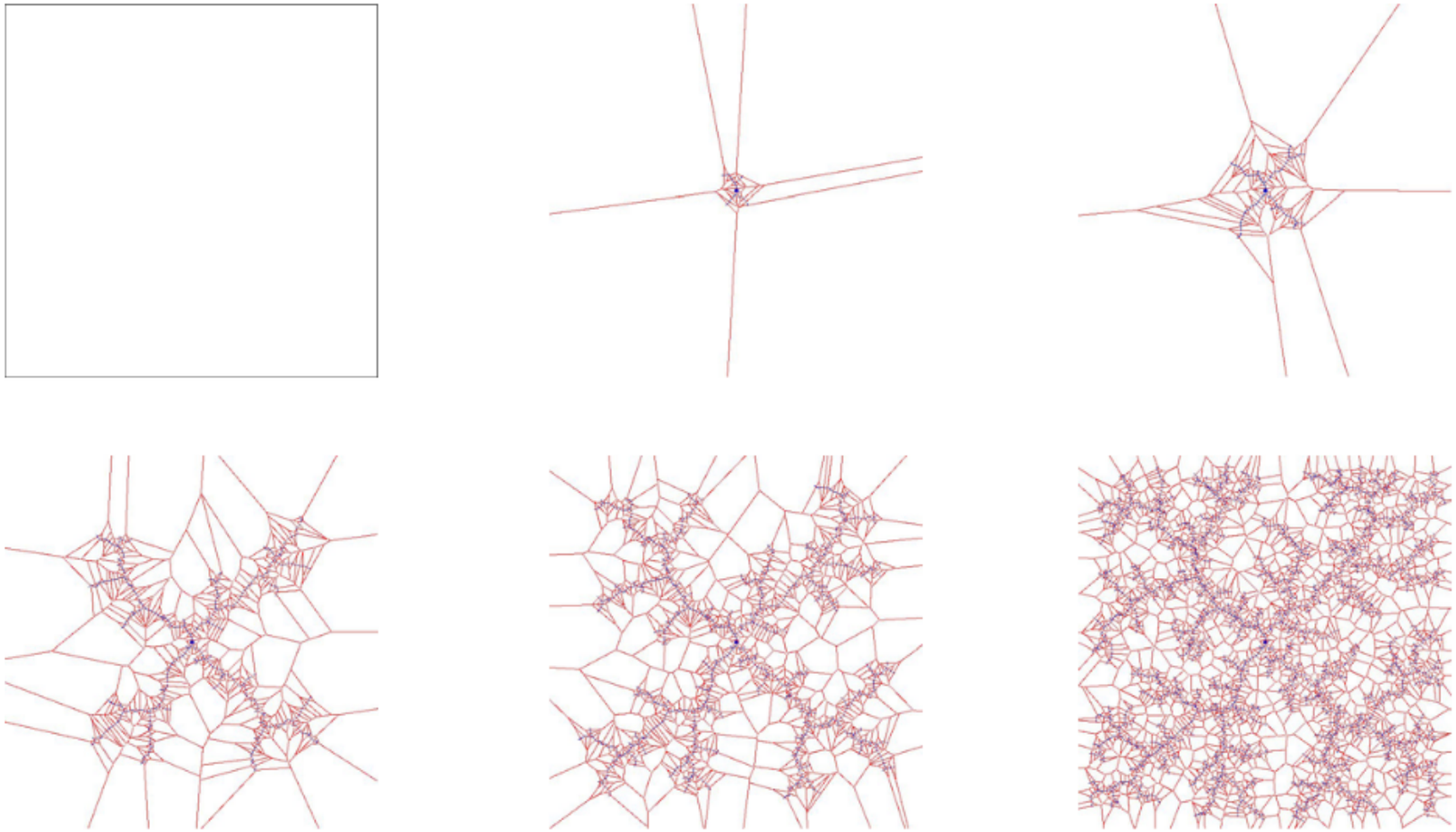
S. Karaman and E. Frazzoli, *Sampling-Based Algorithms for Optimal Motion Planning*, The International Journal of Robotics Research, 30(7), pp. 846-894, 2011

RRT bias toward large voronoi regions

- *The 2D Voronoi diagram induced by a set of seed sites:* Subdivision of the plane in cells, where each cell corresponds to the region where one site is the closest site for all the points in the region.



- The largest Voronoi regions belong to the states on the frontier of the search. Since sampling is done uniformly, the probability of expanding an existing state is proportional to the size of its Voronoi region, such that the tree preferentially expands towards these large, so far unsearched areas.



Planning with RRT

1. Start the RRT at q_{start}
2. At every nth iteration (e.g., $n = 100$), **force** $q_{rand} = q_{goal}$
3. If q_{goal} is reached $\rightarrow$ problem is solved
4. The path from q_{start} to q_{goal} is immediately ready from the *rooted tree* structure: each node has a unique *parent*, such that retracing the tree from the q_{goal} node based on the parent relation, the q_{start} node (the root of the tree) is eventually reached.
5. Picking q_{goal} at every iteration would fail and waste much effort in running into $\mathcal{C}_{obs}$ instead of exploring the space

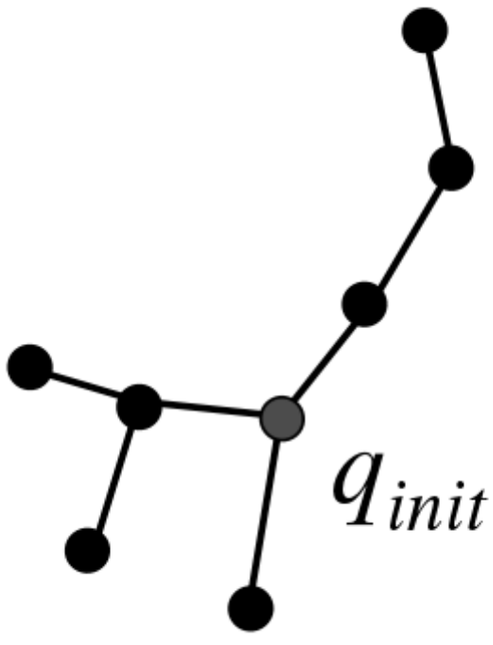
Bi-directional search with RRT

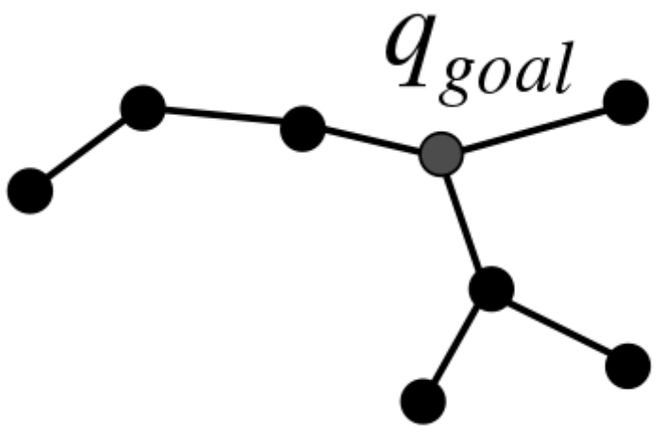
Bidirectional search:

- Grow two RRTs, one from q_{start} and one from q_{goal}
- Periodically, try to extend each tree towards the newest vertex of the other tree

bidirectional search and tree growth

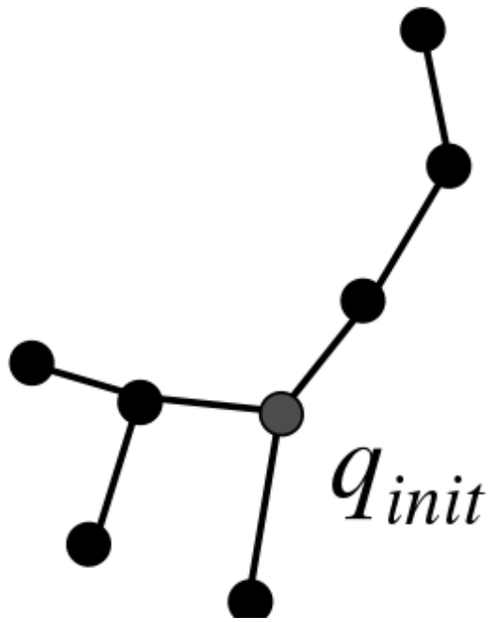
A single RRT-Connect iteration...

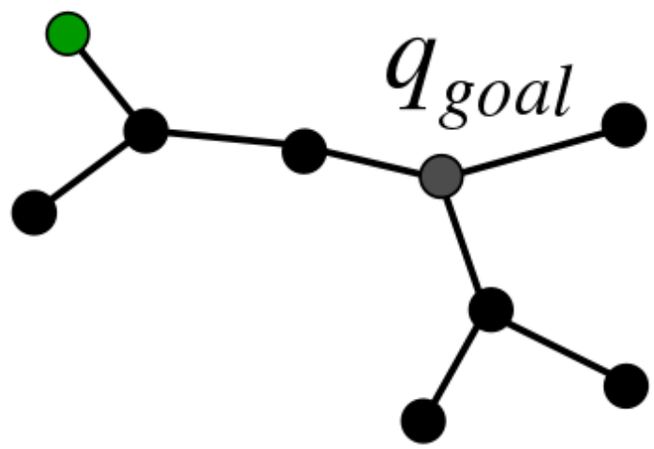




bidirectional search and tree growth

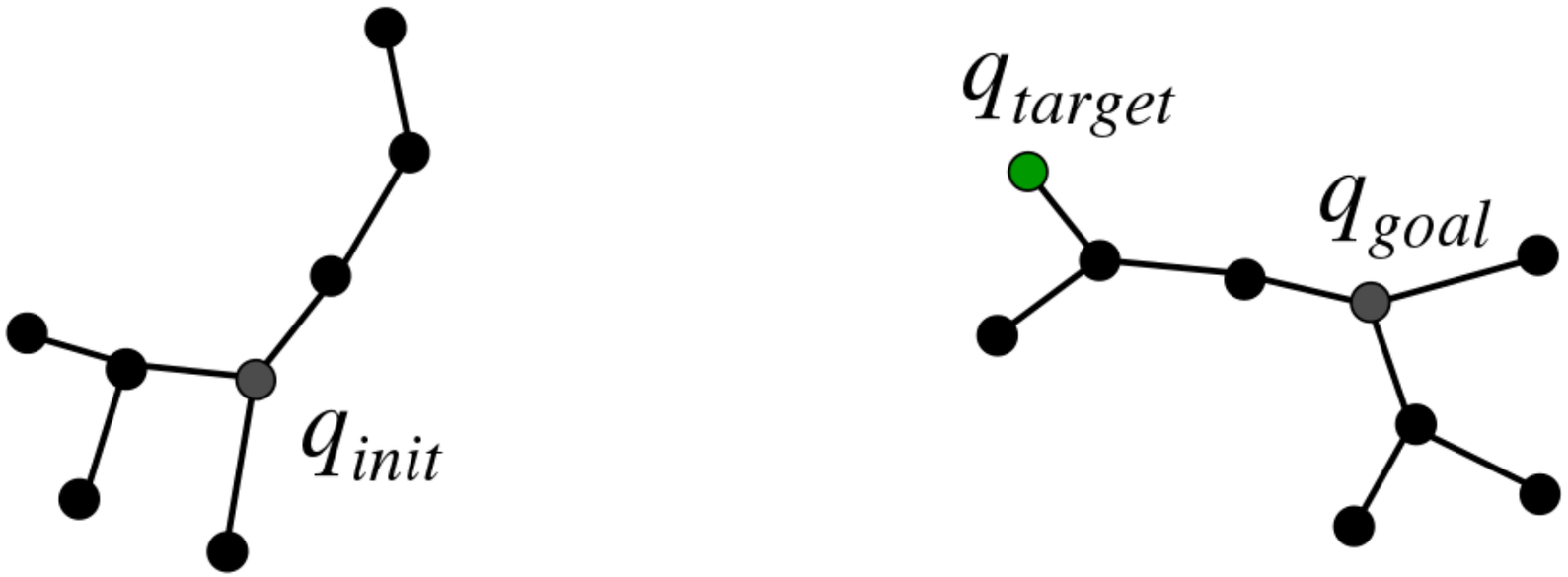
1) One tree grown using random target





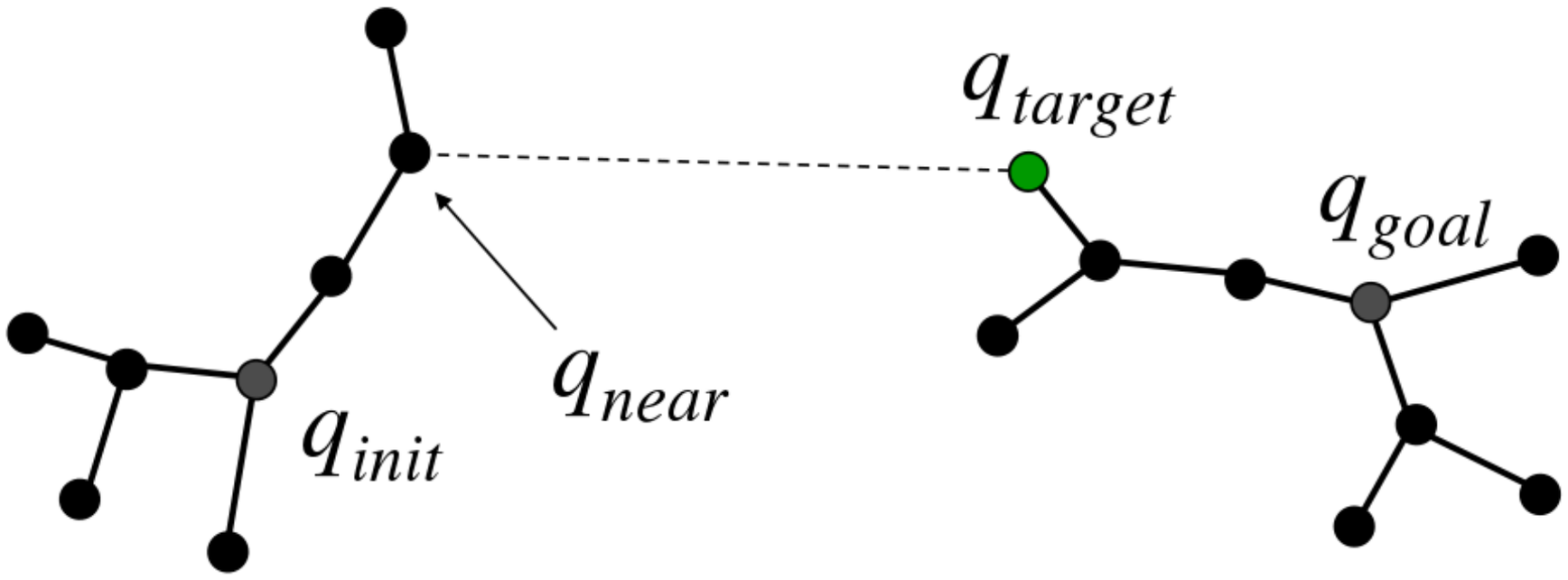
bidirectional search and tree growth

2) New node becomes target for other tree



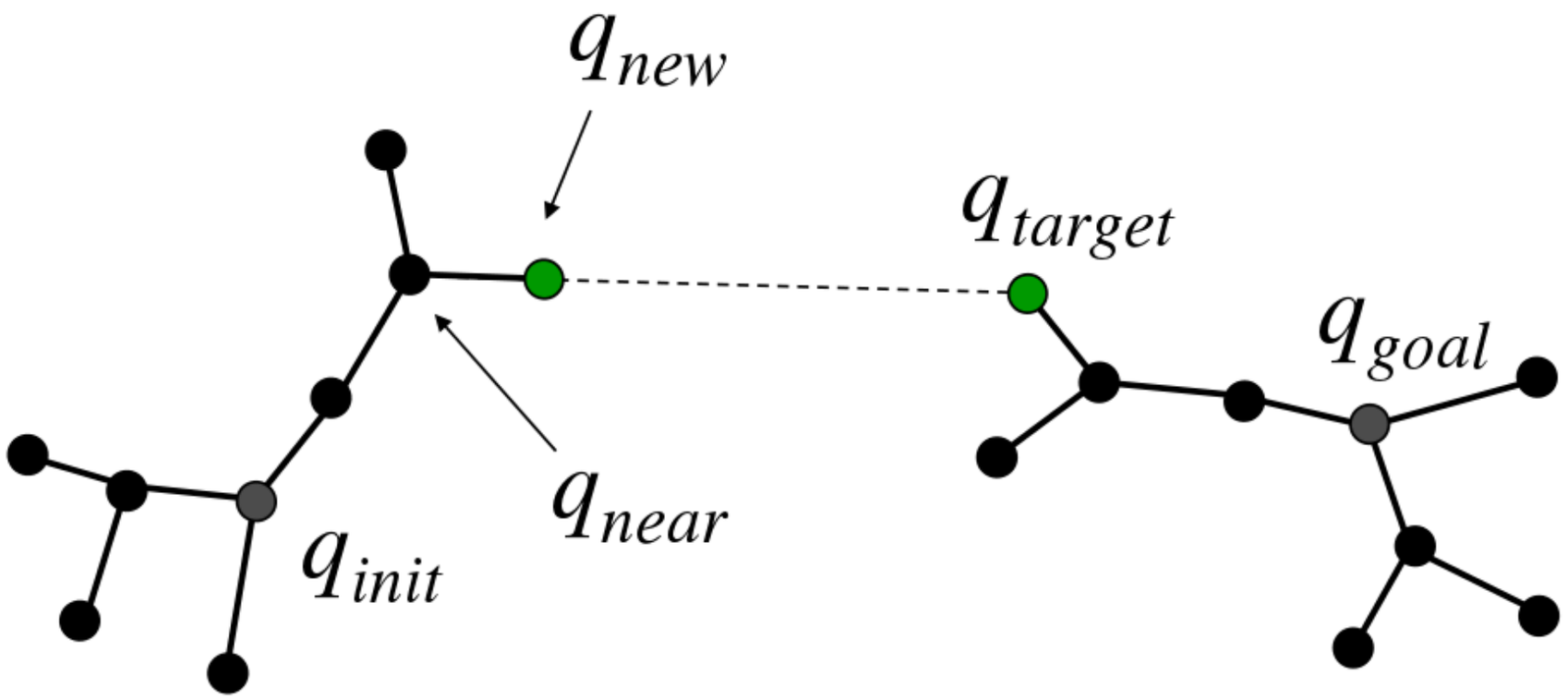
Bidirectional search and tree growth

3) Calculate node “nearest” to target



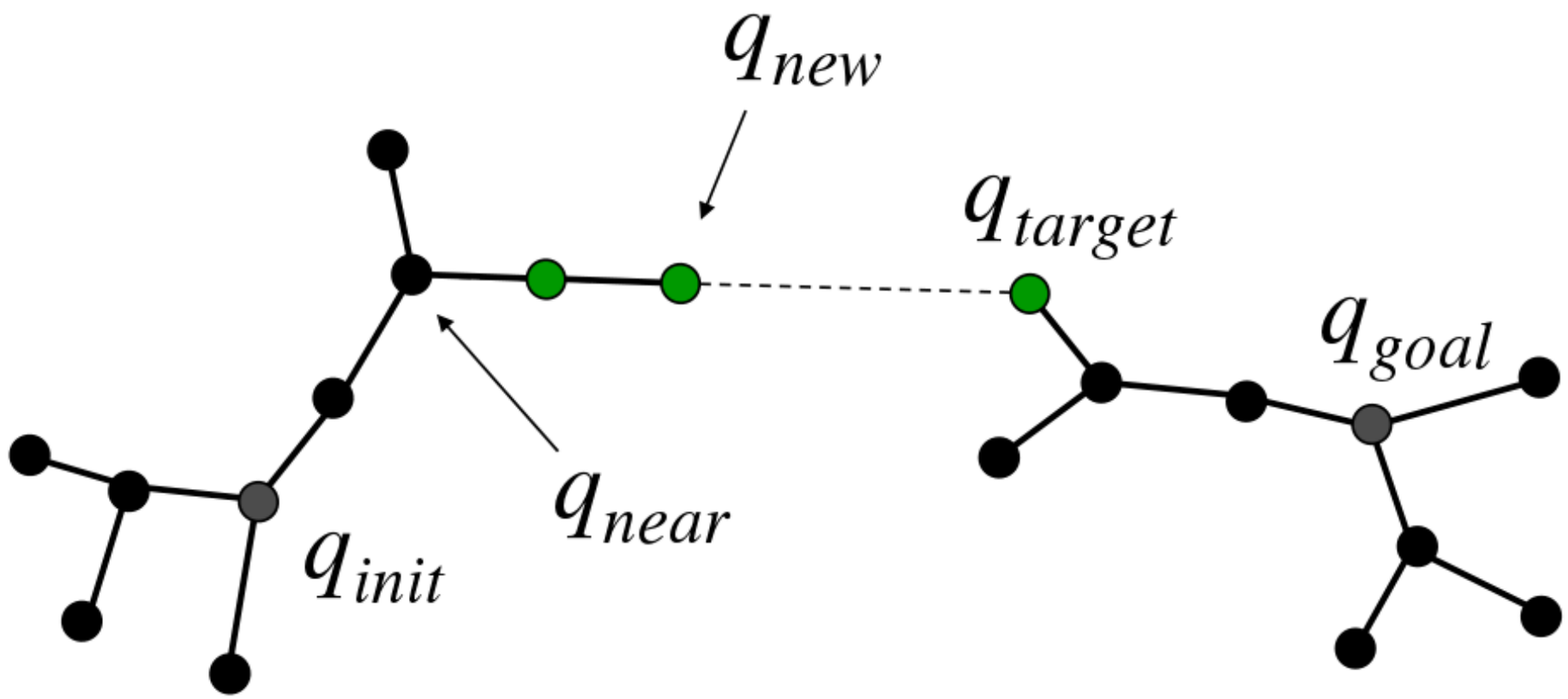
Bidirectional search and tree growth

4) Try to add new collision-free branch



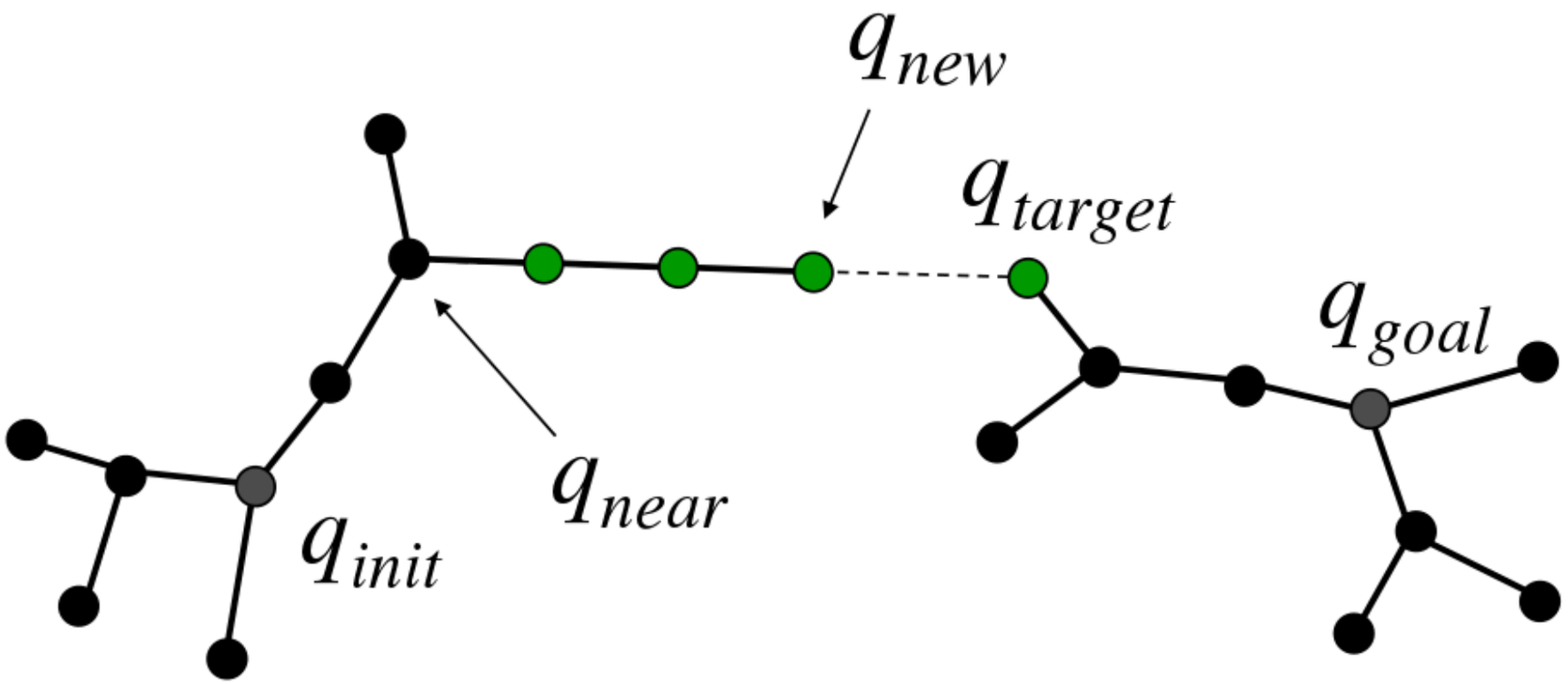
Bidirectional search and tree growth

5) If successful, keep extending branch



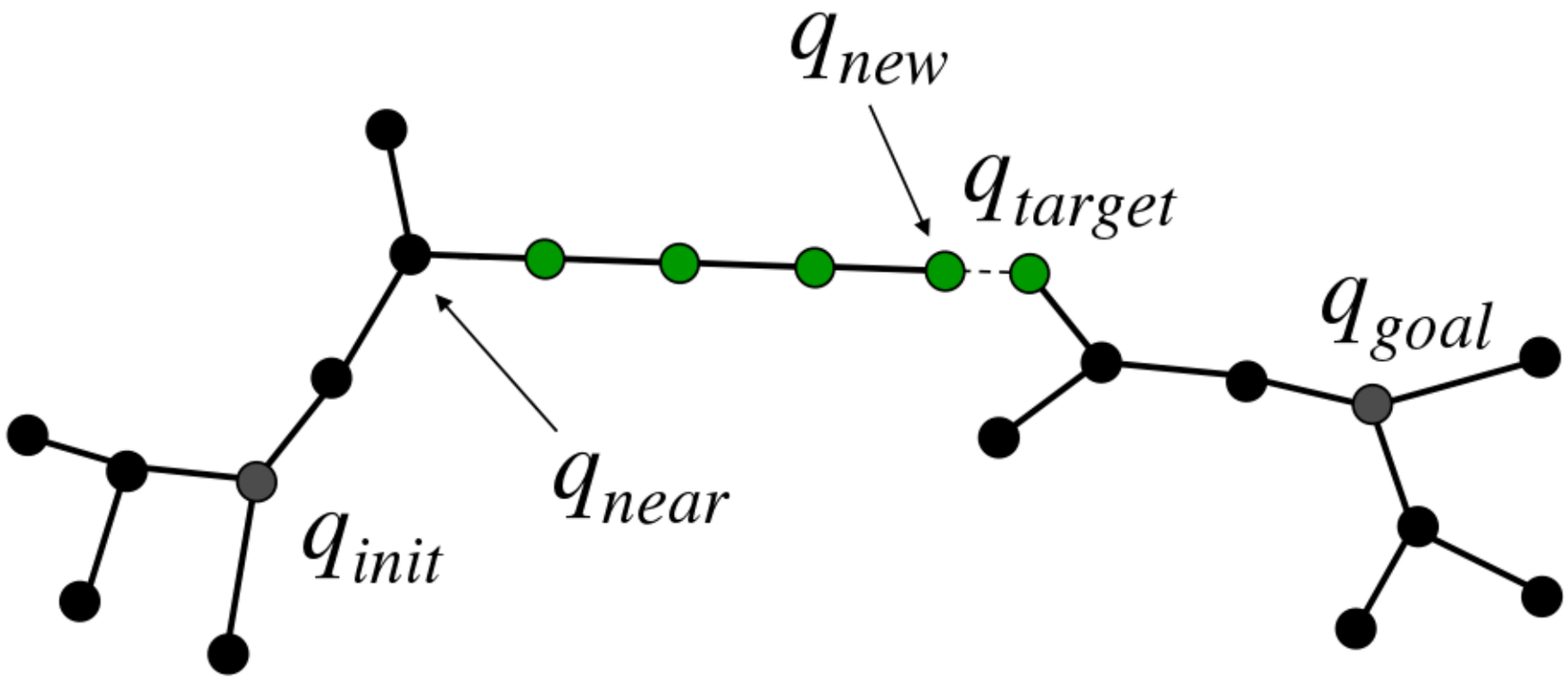
Bidirectional search and tree growth

5) If successful, keep extending branch



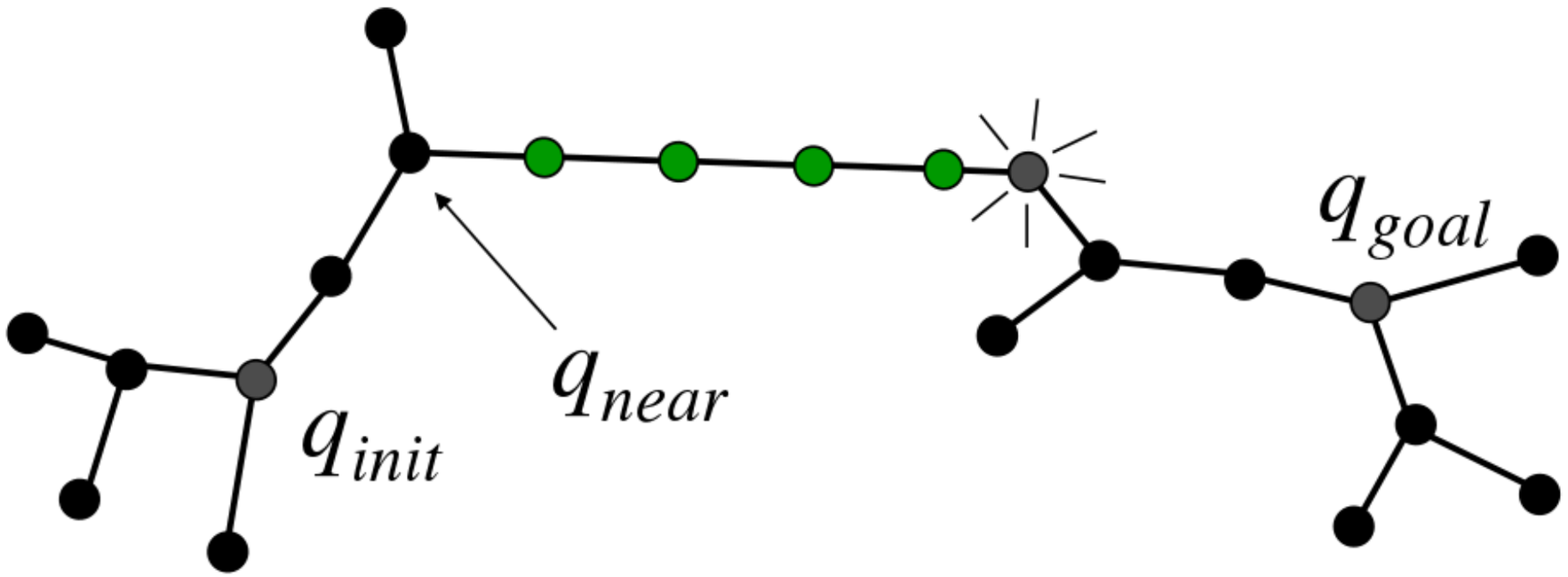
Bidirectional search and tree growth

5) If successful, keep extending branch



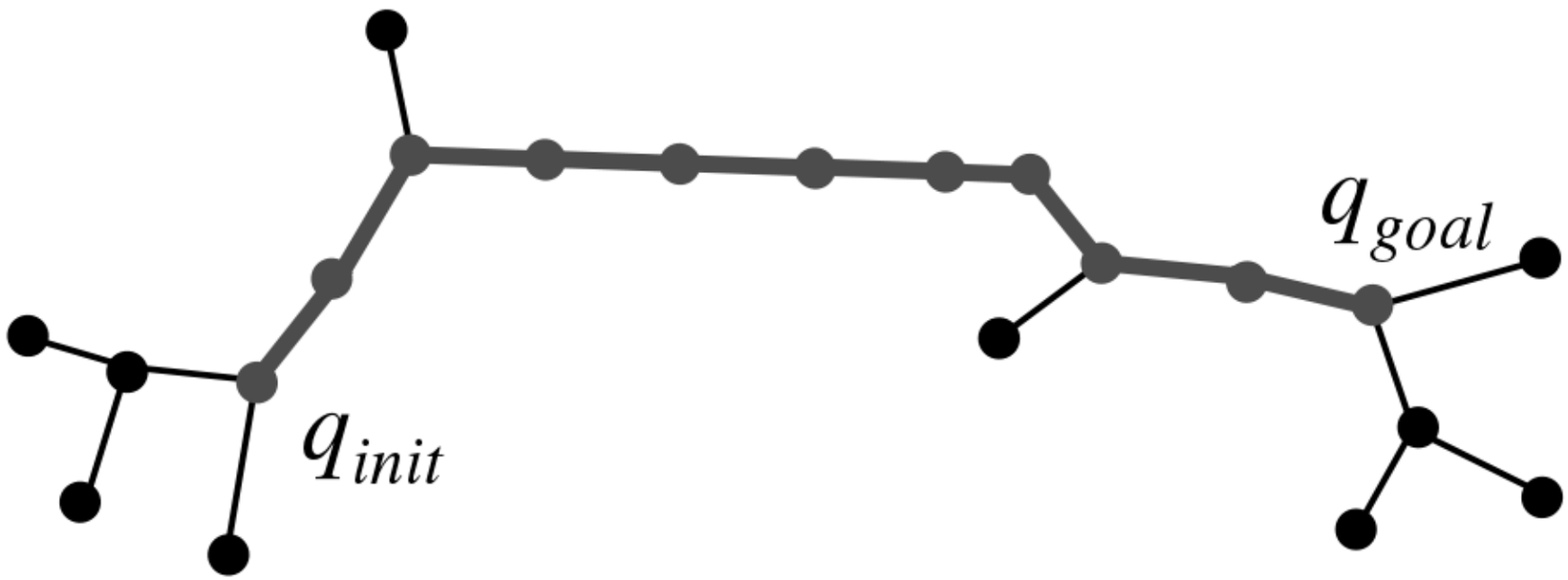
Bidirectional search and tree growth

6) Path found if branch reaches target

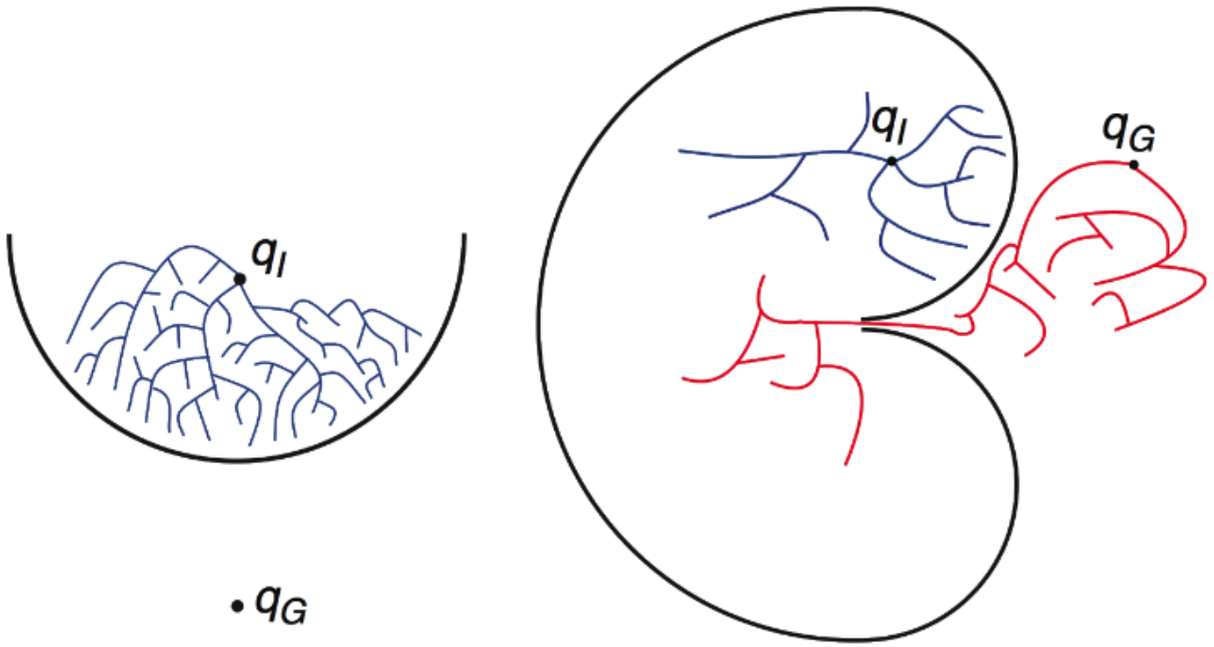


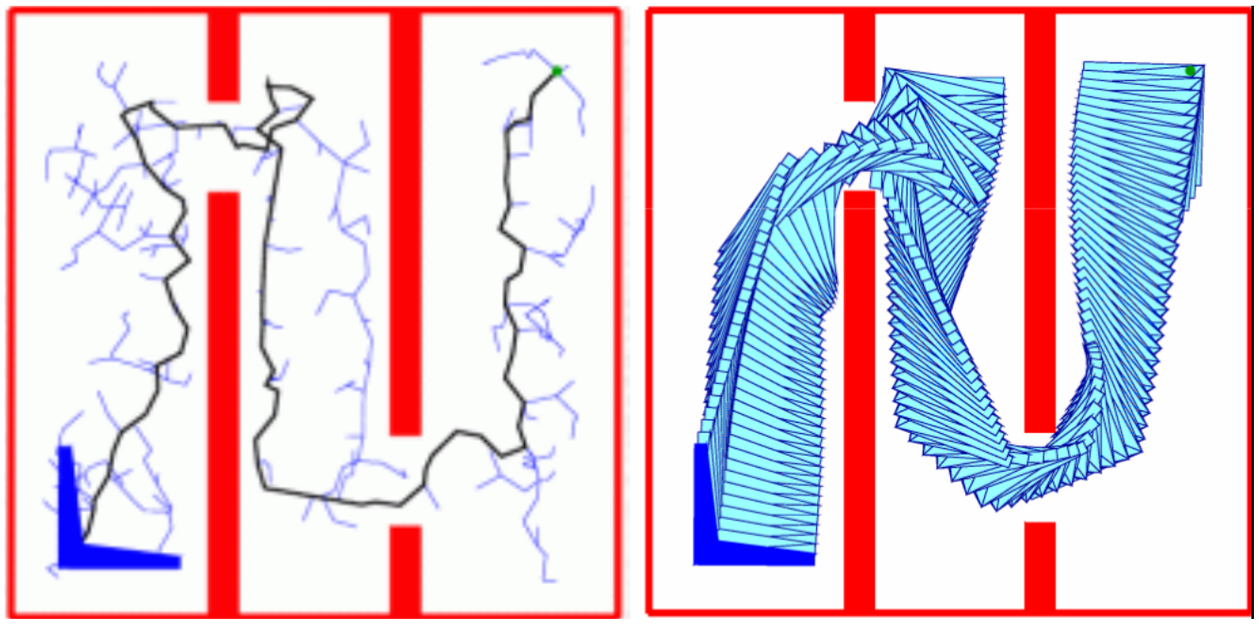
Bidirectional search and tree growth

7) Return path connecting start and goal



Difficult cases





A* or Wavefront search would likely do a good job here since they would use a full (deterministic) model of the C-space environment

The slides that follow present the approach to motion based on potential field, that in principle tackles the (original) continuous problem of path planning.
These slides are fully optional, since they haven't been presented in the class.

Behavior composition approach: potential fields

*Potential field methods for **navigation tasks***

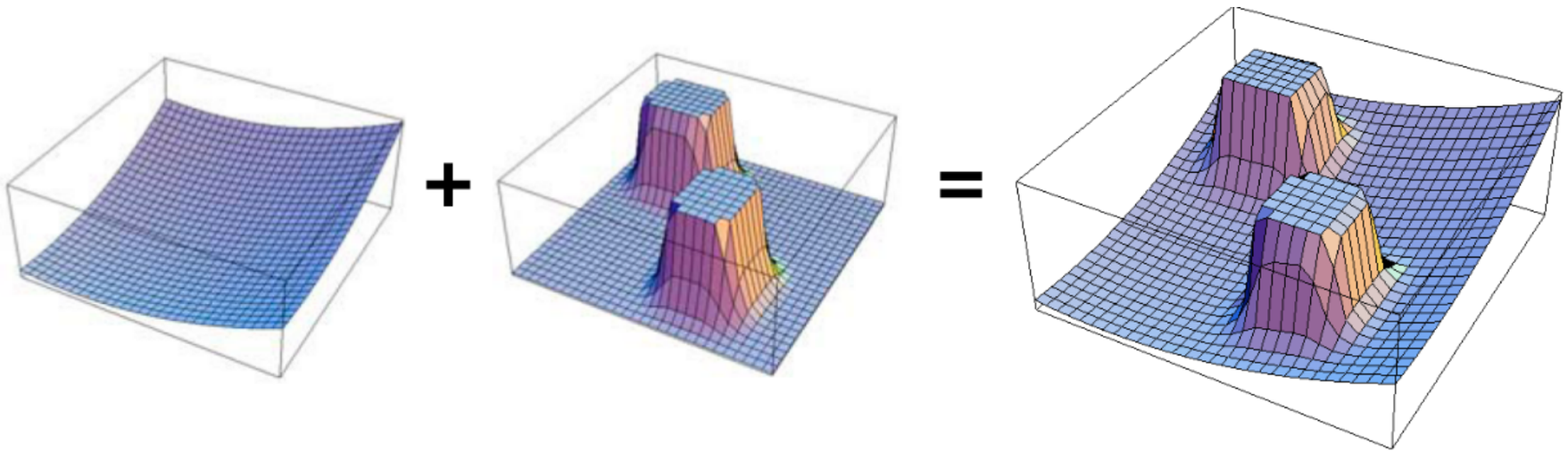
The robot is represented in configuration space as a particle under the influence of an *artificial potential field U(q)* which superimposes:

1. **Repulsive** forces from obstacles
2. **Attractive** force from goal(s)

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

$$\vec{F}(q) = -\vec{\nabla} U(q)$$

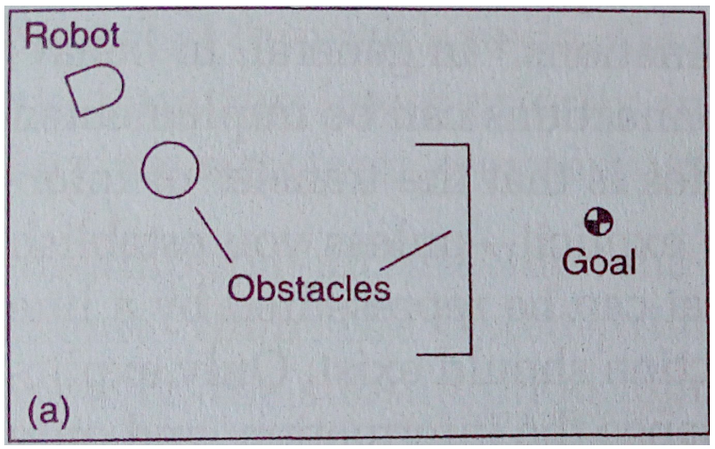
Different behaviors *feels* different fields, and an arbiter combines their proposed motion vectors

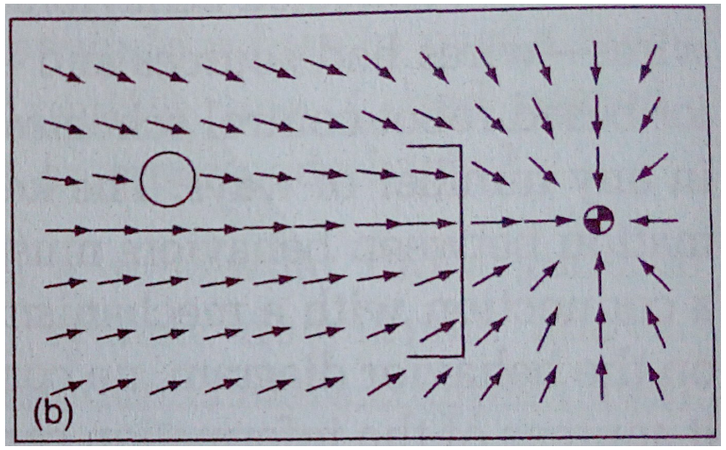


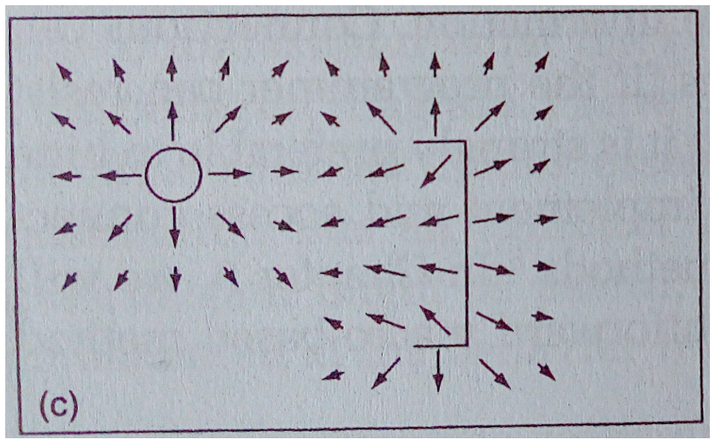
Following a **gradient descent** moves the robot towards the minima (goal = global minimum)

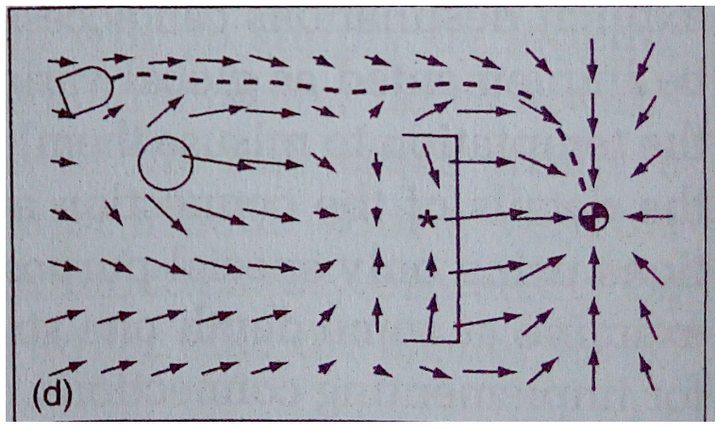
R. Arkin, *Behavior-Based Robotics*, MIT Press, 1998

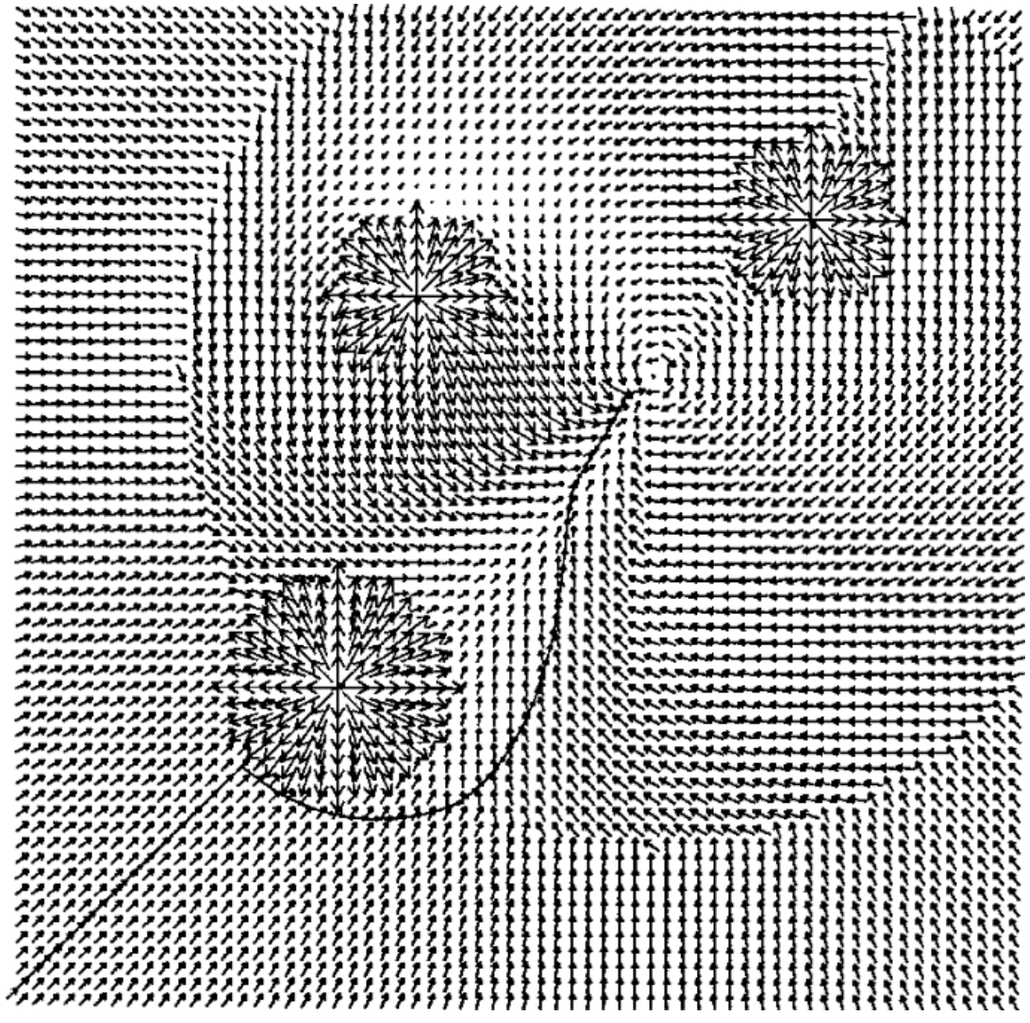
Potential fields at work

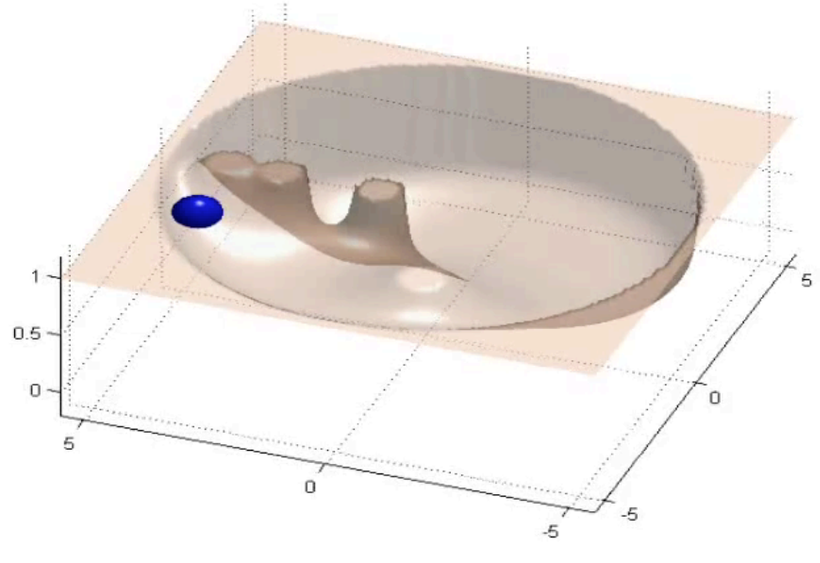




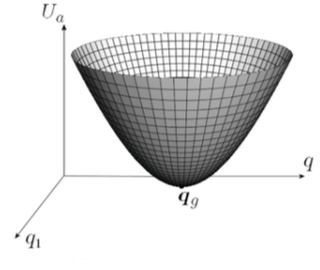




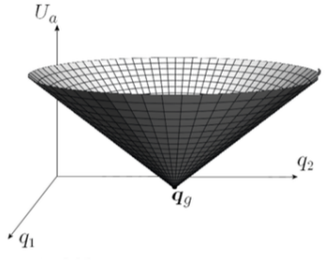




Shape and mathematical properties of the potential functions matter …



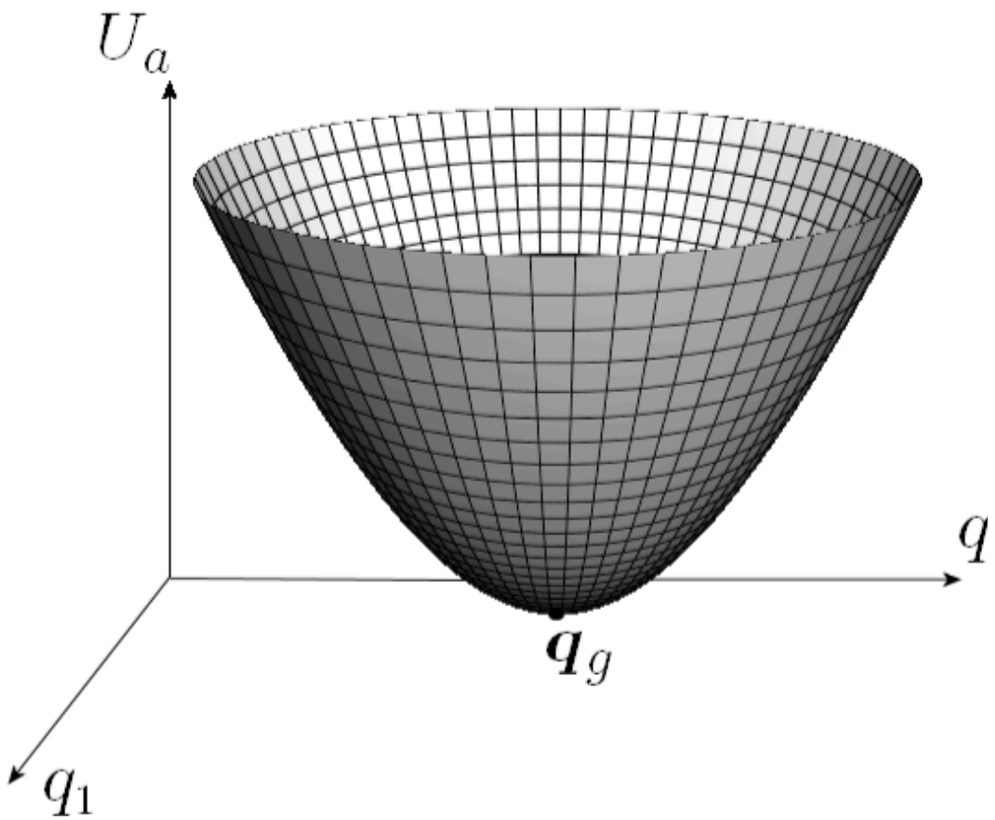
paraboloidal



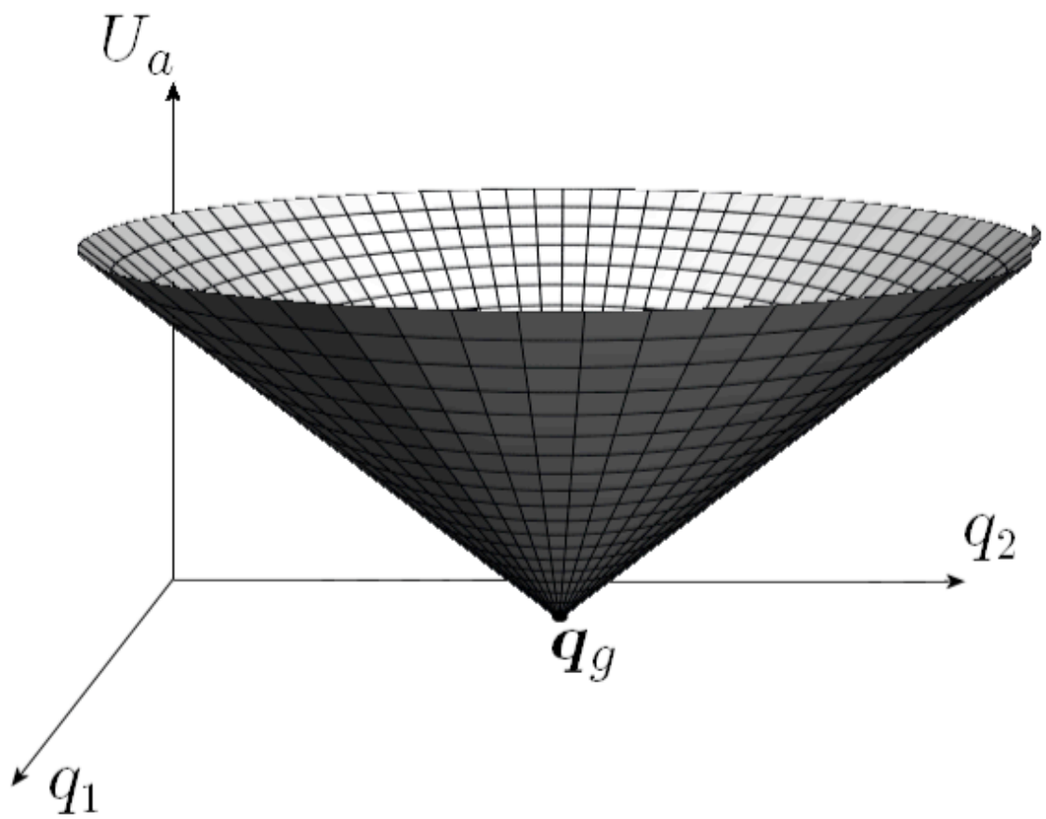
conical

Attractive potential

- objective: to guide the robot to the goal q_g
- two possibilities; e.g., in $\mathcal{C} = \mathrm{R}^2$



paraboloidal



conical

Attractive potential

- paraboloidal: let $\boldsymbol{e} = \boldsymbol{q}_g - \boldsymbol{q}$ and choose $k_a > 0$

$$U_{a1}(\boldsymbol{q}) = \frac{1}{2} k_a \boldsymbol{e}^T(\boldsymbol{q})\boldsymbol{e}(\boldsymbol{q}) = \frac{1}{2} k_a \|\boldsymbol{e}(\boldsymbol{q})\|^2$$

- the resulting attractive force is linear in $\boldsymbol{e}$

$$\boldsymbol{f}_{a1}(\boldsymbol{q}) = -\nabla U_{a1}(\boldsymbol{q}) = k_a \boldsymbol{e}(\boldsymbol{q})$$

- conical:

$$U_{a2}(\boldsymbol{q}) = k_a \|\boldsymbol{e}(\boldsymbol{q})\|$$

- the resulting attractive force is constant

$$\boldsymbol{f}_{a2}(\boldsymbol{q}) = -\nabla U_{a2}(\boldsymbol{q}) = k_a \frac{\boldsymbol{e}(\boldsymbol{q})}{\|\boldsymbol{e}(\boldsymbol{q})\|}$$

Attractive potential

- $\boldsymbol{f}_{a1}$ behaves better than $\boldsymbol{f}_{a2}$ in the vicinity of $\boldsymbol{q}_g$ but increases indefinitely with $\boldsymbol{e}$

- a convenient solution is to combine the two profiles: conical away from $\boldsymbol{q}_g$ and paraboloidal close to $\boldsymbol{q}_g$

$$U_a(\boldsymbol{q}) = \begin{cases} \frac{1}{2} k_a \|\boldsymbol{e}(\boldsymbol{q})\|^2 & \text{if} \quad \|\boldsymbol{e}(\boldsymbol{q})\| \leq \rho \\ k_b \|\boldsymbol{e}(\boldsymbol{q})\| & \text{if} \quad \|\boldsymbol{e}(\boldsymbol{q})\| > \rho \end{cases}$$

continuity of $\boldsymbol{f}_a$ at the transition requires

$$k_a \boldsymbol{e}(\boldsymbol{q}) = k_b \frac{\boldsymbol{e}(\boldsymbol{q})}{\|\boldsymbol{e}(\boldsymbol{q})\|} \qquad \text{for} \quad \|\boldsymbol{e}(\boldsymbol{q})\| = \rho$$

i.e., $k_b = \rho k_a$

Repulsive potential

- objective: keep the robot away from $\mathcal{CO}$
- assume that $\mathcal{CO}$ has been partitioned in advance in convex components $\mathcal{CO}_i$
- for each $\mathcal{CO}_i$ define a repulsive field

$$U_{r,i}(\boldsymbol{q}) = \begin{cases} \frac{k_{r,i}}{\gamma}\left(\frac{1}{\eta_i(\boldsymbol{q})} - \frac{1}{\eta_{0,i}}\right)^{\gamma} & \text{if } \eta_i(\boldsymbol{q}) \leq \eta_{0,i} \\ 0 & \text{if } \eta_i(\boldsymbol{q}) > \eta_{0,i} \end{cases}$$

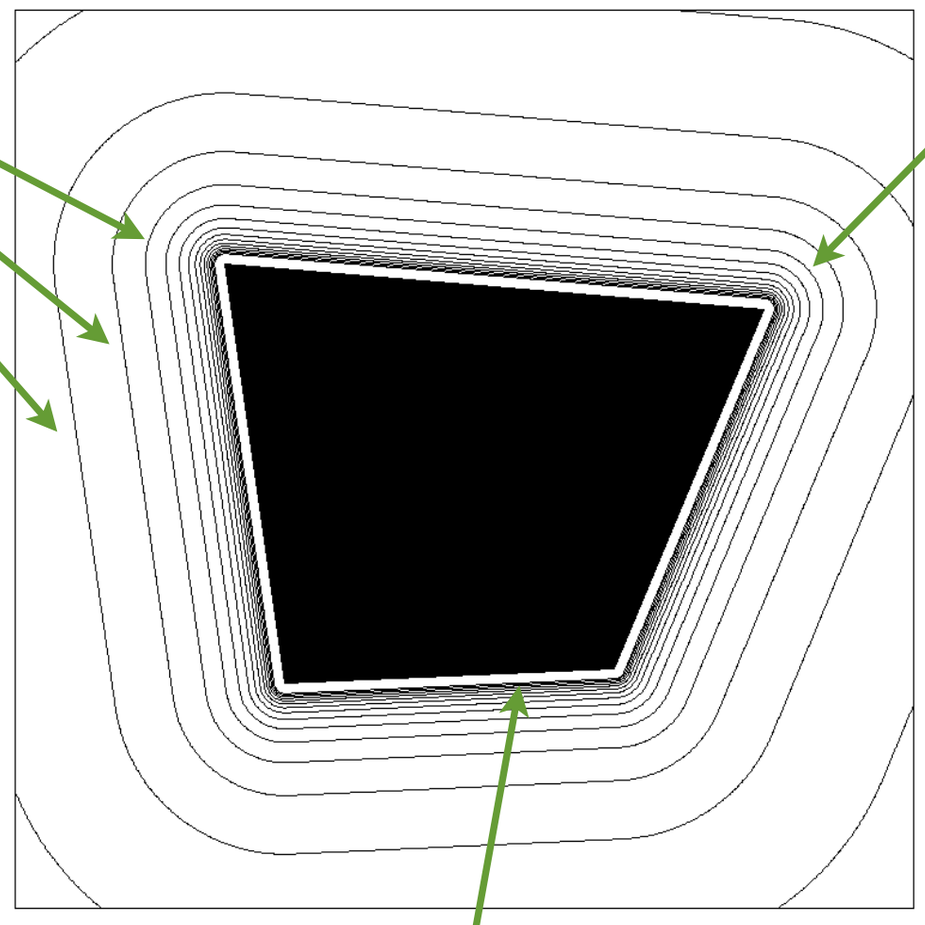
where $k_{r,i} > 0; \gamma = 2,3,\ldots;$ $\eta_{0,i}$ is the range of influence of $\mathcal{CO}_i$ and

$$\eta_i(\boldsymbol{q}) = \min_{\boldsymbol{q}' \in \mathcal{CO}_i} \|\boldsymbol{q} - \boldsymbol{q}'\|$$

Repulsive potential

equipotential contours

the higher γ,
the steepest the slope



$U_{r,i}$ goes to ∞
at the boundary of $\mathcal{CO}_i$

Repulsive potential

- the resulting repulsive force is

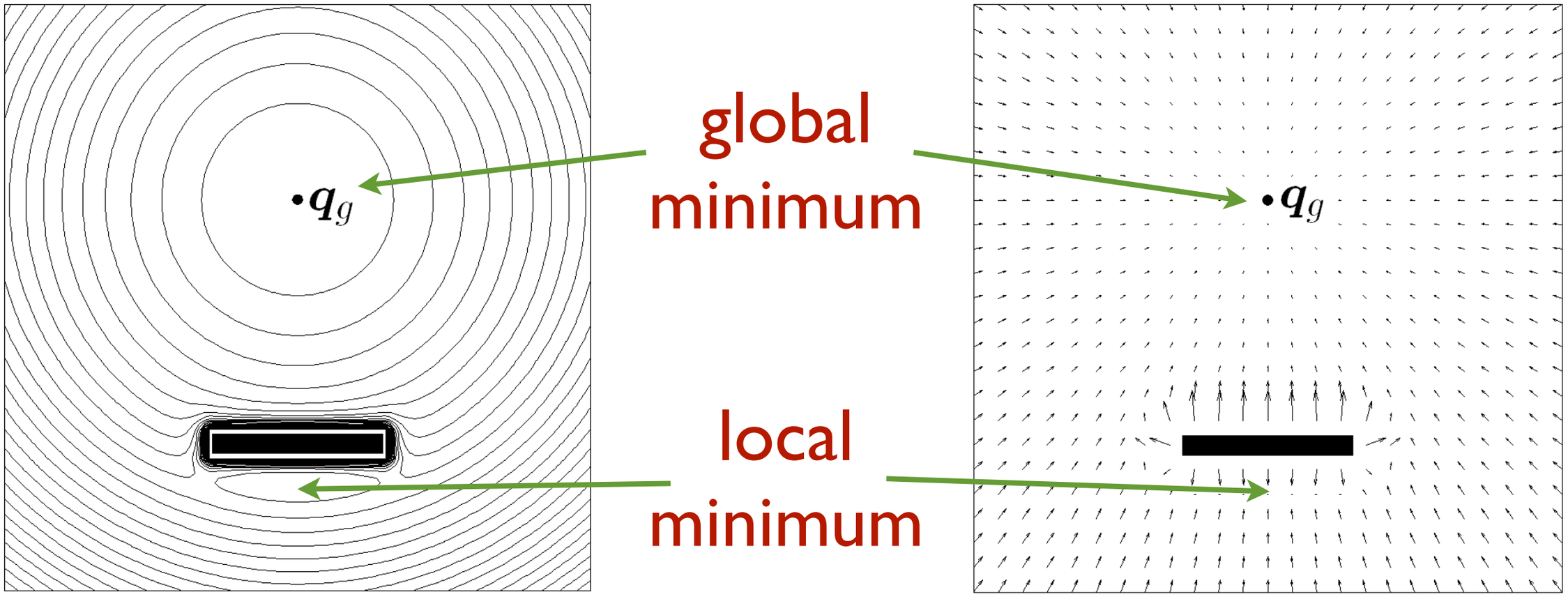
$$\boldsymbol{f}_{r,i}(\boldsymbol{q}) = -\nabla U_{r,i}(\boldsymbol{q}) = \begin{cases} \dfrac{k_{r,i}}{\eta_i^2(\boldsymbol{q})}\left(\dfrac{1}{\eta_i(\boldsymbol{q})} - \dfrac{1}{\eta_{0,i}}\right)^{\gamma-1}\nabla\eta_i(\boldsymbol{q}) & \text{if } \eta_i(\boldsymbol{q}) \leq \eta_{0,i} \\ 0 & \text{if } \eta_i(\boldsymbol{q}) > \eta_{0,i} \end{cases}$$

- $\boldsymbol{f}_{r,i}$ is orthogonal to the equipotential contour passing through $\boldsymbol{q}$ and points away from the obstacle
- $\boldsymbol{f}_{r,i}$ is continuous everywhere thanks to the convex decomposition of $\mathcal{CO}$
- aggregate repulsive potential of $\mathcal{CO}$

$$U_r(\boldsymbol{q}) = \sum_{i=1}^{p} U_{r,i}(\boldsymbol{q})$$

Total potential

- superposition: $U_t(\boldsymbol{q}) = U_a(\boldsymbol{q}) + U_r(\boldsymbol{q})$
- force field: $\boldsymbol{f}_t(\boldsymbol{q}) = -\nabla U_t(\boldsymbol{q}) = \boldsymbol{f}_a(\boldsymbol{q}) + \sum_{i=1}^{p} \boldsymbol{f}_{r,i}(\boldsymbol{q})$



Planning/navigation using potential field

- three techniques for planning on the basis of $\boldsymbol{f}_t$

1. consider $\boldsymbol{f}_t$ as generalized forces: $\boldsymbol{\tau} = \boldsymbol{f}_t(\boldsymbol{q})$

   the effect on the robot is filtered by its dynamics (generalized accelerations are scaled) Robot has *mass (inertia)*!

2. consider $\boldsymbol{f}_t$ as generalized accelerations: $\ddot{\boldsymbol{q}} = \boldsymbol{f}_t(\boldsymbol{q})$

   the effect on the robot is independent on its dynamics (generalized forces are scaled)

3. consider $\boldsymbol{f}_t$ as generalized velocities: $\dot{\boldsymbol{q}} = \boldsymbol{f}_t(\boldsymbol{q})$

   the effect on the robot is independent on its dynamics (generalized forces are scaled)

Kinematics

Planning/navigation using potential field

- technique 1 generates smoother movements, while technique 3 is quicker (irrespective of robot dynamics) to realize motion corrections; technique 2 gives intermediate results

- strictly speaking, only technique 3 guarantees (in the absence of local minima) asymptotic stability of $\boldsymbol{q}_g$; velocity damping is necessary to achieve the same with techniques 1 and 2

Planning/navigation using potential field

- off-line planning

  paths in $\mathcal{C}$ are generated by numerical integration of the dynamic model (if technique 1), of $\ddot{\boldsymbol{q}} = \boldsymbol{f}_t(\boldsymbol{q})$ (if technique 2), of $\dot{\boldsymbol{q}} = \boldsymbol{f}_t(\boldsymbol{q})$ (if technique 3)

  the most popular choice is 3 and in particular

$$\boldsymbol{q}_{k+1} = \boldsymbol{q}_k + T\boldsymbol{f}_t(\boldsymbol{q}_k)$$

  i.e., the algorithm of steepest descent

- on-line planning (is actually feedback!)

  technique 1 directly provides control inputs, technique 2 too (via inverse dynamics), technique 3 provides reference velocities for low-level control loops

  the most popular choice is 3

Local minima: a complication

- if a planned path enters the basin of attraction of a local minimum $\boldsymbol{q}_m$ of U_t, it will reach $\boldsymbol{q}_m$ and stop there, because $\boldsymbol{f}_t\left(\boldsymbol{q}_m\right) = -\nabla U_t(\boldsymbol{q}_m) = 0$; whereas saddle points are not an issue

- repulsive fields generally create local minima, hence motion planning based on artificial potential fields is not complete (the path may not reach $\boldsymbol{q}_g$ even if a solution exists)

- workarounds exist but consider that artificial potential fields are mainly used for on-line motion planning, where completeness may not be required

Workaround 1: Best-First algorithm

- build a discretized representation (by defect) of $\mathcal{C}_{\text{free}}$ using a regular grid, and associate to each free cell of the grid the value of U_t at its centroid

- build a tree T rooted at $\boldsymbol{q}_s$: at each iteration, select the leaf of T with the minimum value of U_t and add as children its adjacent free cells that are not in T

- planning stops when $\boldsymbol{q}_g$ is reached (success) or no further cells can be added to T (failure)

- if success, build a solution path by tracing back the arcs from $\boldsymbol{q}_g$ to $\boldsymbol{q}_s$

Workaround 1: Best-First algorithm

- best-first evolves as a grid-discretized version of steepest descent until a local minimum is met

- at a local minimum, best-first will "fill" its basin of attraction until it finds a way out

- the best-first algorithm is resolution complete

- its complexity is exponential in the dimension of $\mathcal{C}$, hence it is only applicable in low-dimensional spaces

- efficiency improves if random walks are alternated with basin-filling iterations (randomized best-first)

Workaround 2: navigation functions

- path generated by the best-first algorithm are not efficient (local minima are not avoided)

- a different approach: build navigation functions, i.e., potentials without local minima

- if the $\mathcal{C}$-obstacles are star-shaped, one can map $\mathcal{CO}$ to a collection of spheres via a diffeomorphism, build a potential in transformed space and map it back to $\mathcal{C}$

- another possibility is to define the potential as an harmonic function (solution of Laplace's equation)

Workaround 3: wavefront expansion

- an efficient alternative: numerical navigation functions
- with $\mathcal{C}_{\text{free}}$ represented as a gridmap, assign 0 to start cell, 1 to cells adjacent to the 0-cell, 2 to unvisited cells adjacent to 1-cells, ... (wavefront expansion)