Human-Oriented Robotics

Robot Motion Planning

Kai Arras

Social Robotics Lab, University of Freiburg

Winter term 2014/2015

Contents

- Introduction
- Configuration space
- Combinatorial planning
- Sampling-based planning
- Potential fields methods
- A*, Any-Angle A*, D*/D* Lite
- Dynamic Window Approach (DWA)
- Markov Decision Processes (MDP)

Introduction

- **Robot motion planning** is a fundamental robotics problem since, by definition, a robot accomplishes tasks by **moving in the real world**
- For plan execution, motion planning is tightly coupled to **control theory**
- The **goal** of motion planning is to enable a robot to reach a goal configuration (e.g. a goal location in its workspace) without collisions or self-collisions

- Notice the difference:
  - Perception, sensing, tracking and un/supervised learning are all **estimation-related tasks** where a robot sits still, observes the world and **reasons about the state** of the world or of itself
  - Motion planning, plan execution and control are **planning** and **decision-related tasks** where the robot actually moves and **physically interacts** with the world

Introduction

- The motion planning problem can be **stated as follows**: given
  - an **initial configuration** of the robot (e.g. a pose in 2D)
  - a desired **goal configuration**
  - a **model of the robot** (e.g. in terms of geometry, kinematics and dynamics)
  - A **map of the environment** with obstacles in the workspace,

    find an **admissible**, **collision-free path** that moves the robot gradually from start to goal

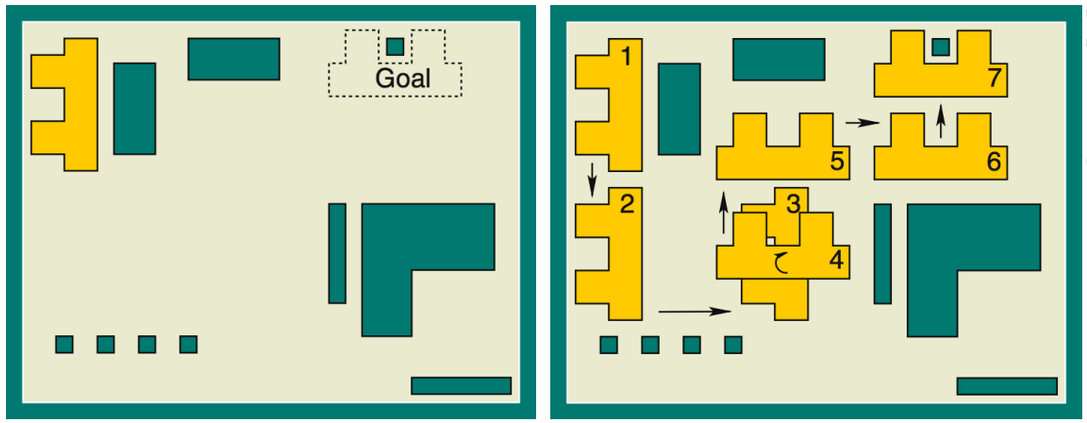
- There are two different criteria that a plan may need to satisfy:
  - **Feasibility**: find a plan that causes arrival at the goal state, regardless of its efficiency. For many interesting planning problems, feasibility is already very challenging
  - **Optimality**: in addition to arriving in a goal state, find a feasible plan that is optimal in some sense (e.g. shortest/smoothest path, minimal time, smallest risk). Achieving optimality can be considerably harder than feasibility

Introduction

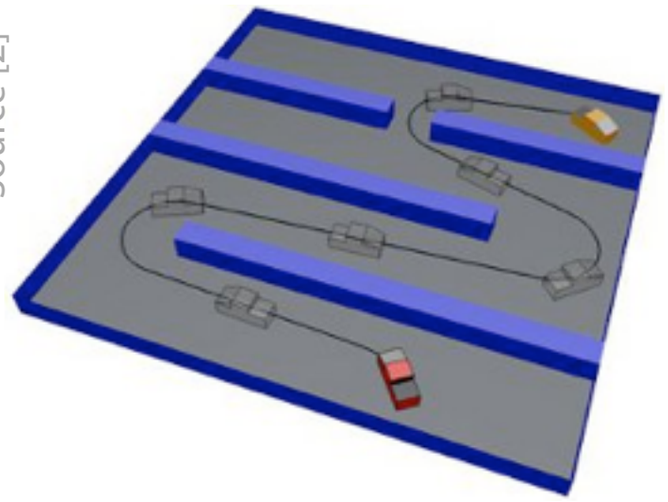
- Motion planning would be **simple** if robots were **free-floating points** in space without physical extension or constraints to their motion
- But considering these rather simple wheeled mobile robots, we have
  - **Physical extension**: non-circular shape
  - **Motion constraints**: they can go anywhere but not by following any trajectory. This is an example of **non-holonomic constraints**
  - If these robots had very weak motors with limited acceleration, **dynamic constraints** would come into play (inertia, mass, torque etc.)
- Let us consider some examples...

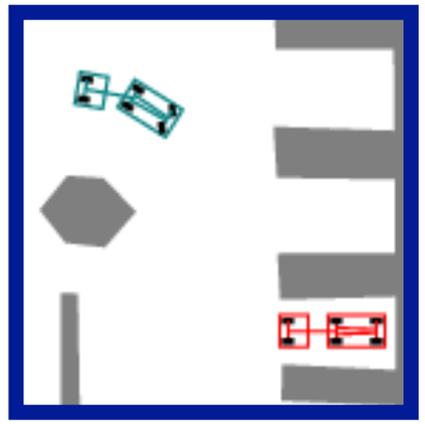
Introduction

- Examples

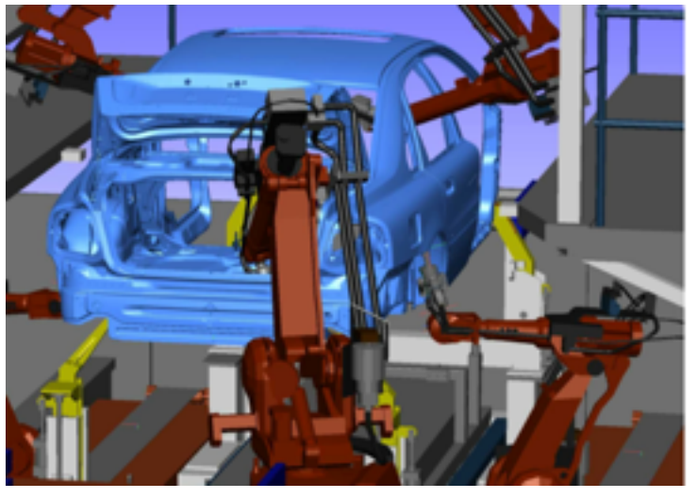


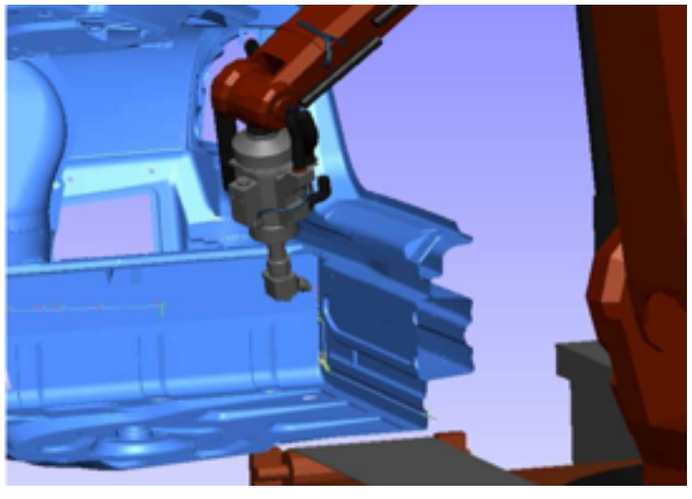
Source [2]





Mobile robots and intelligent vehicles



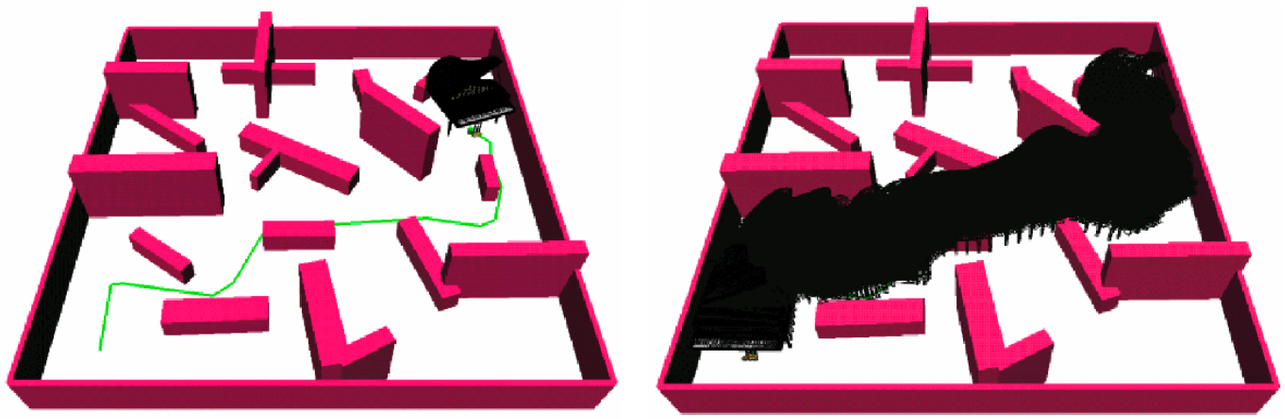


Source [2]

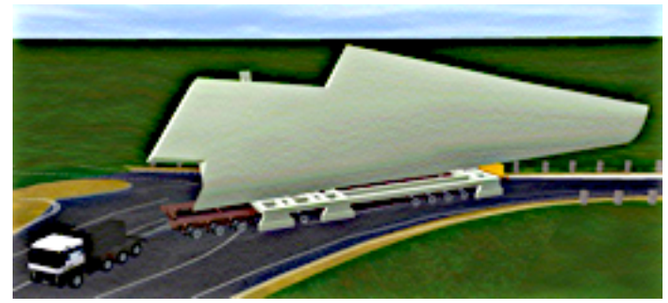
Robot manipulators

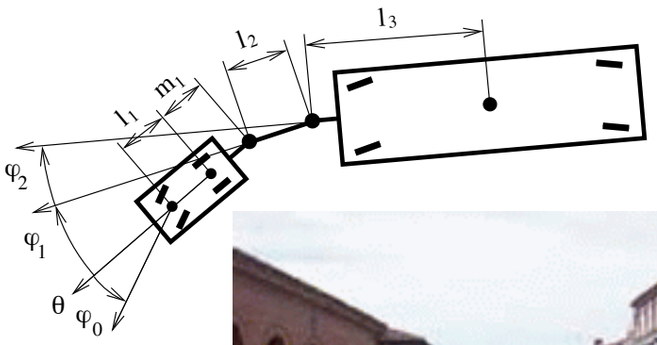
Introduction

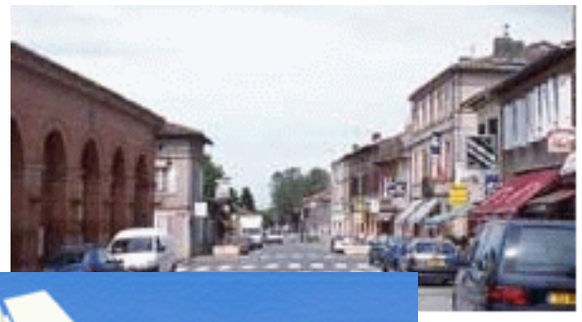
- Examples



Motion planning is also called **piano mover's problem**

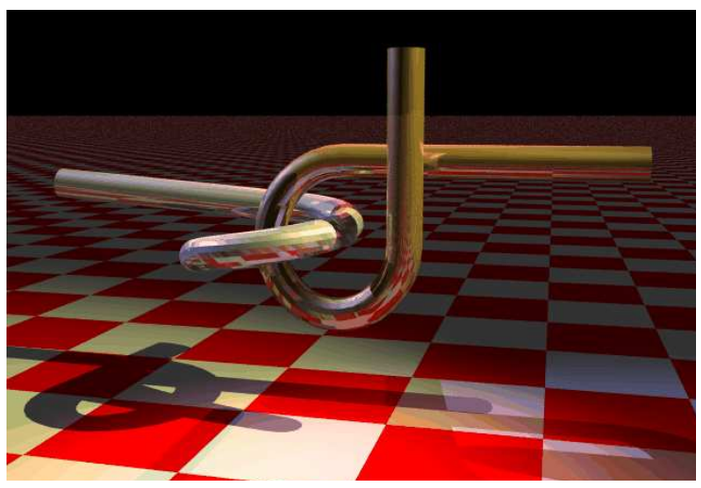






Source [5]

Trailer-truck trajectory optimization for Airbus A380 component transportation



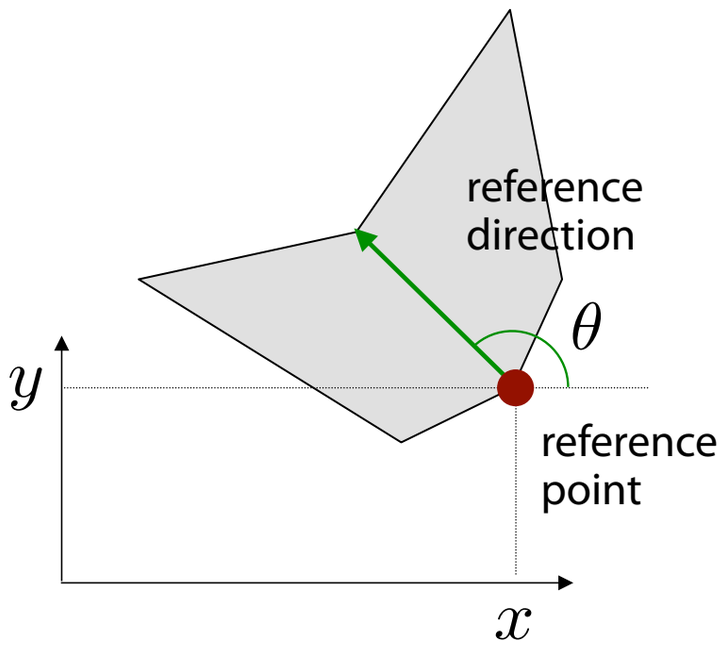
Source [2]

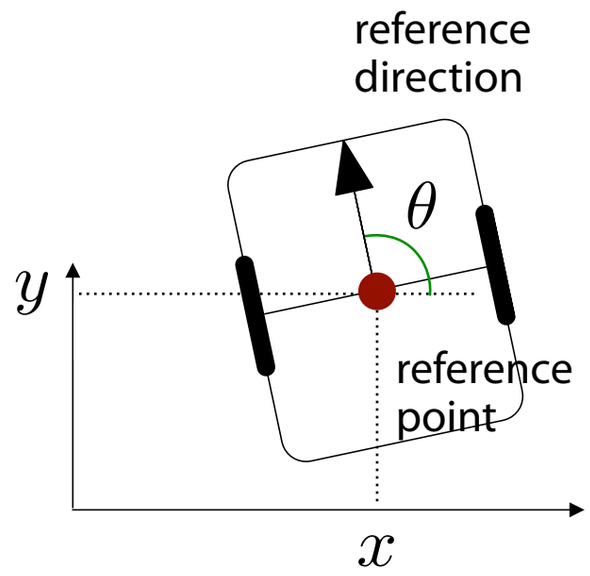
Alpha 1.0 3D puzzle

Configuration Space

- Although plans are executed in the Cartesian world ("the workspace"), motion planning lives in another space: the **configuration space**
- A **robot configuration** q is a specification of the positions of **all** robot points relative to a fixed coordinate system
- Usually a configuration is expressed as a **vector of positions** and **orientations**

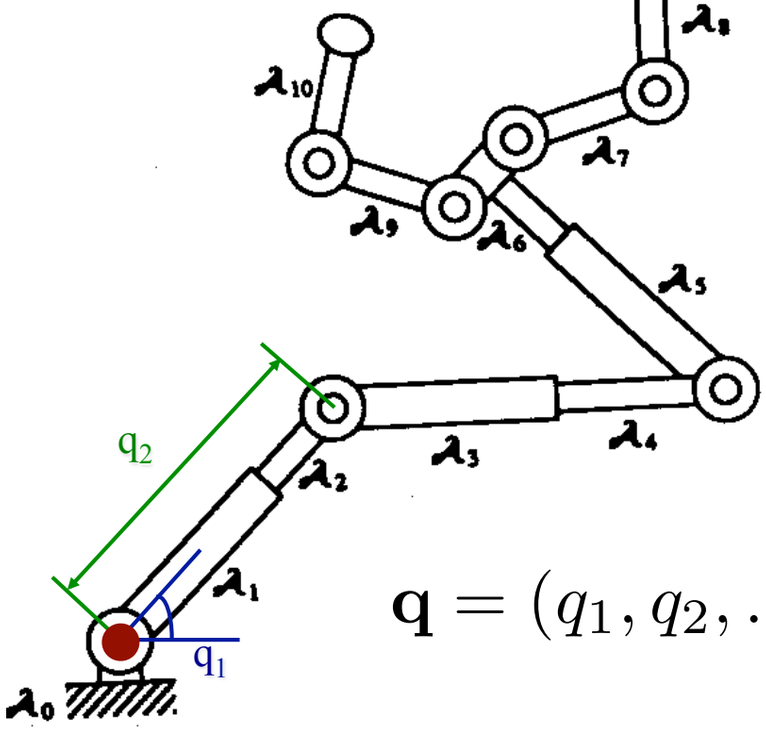
- Rigid body examples with configuration $q = (x, y, \theta)$



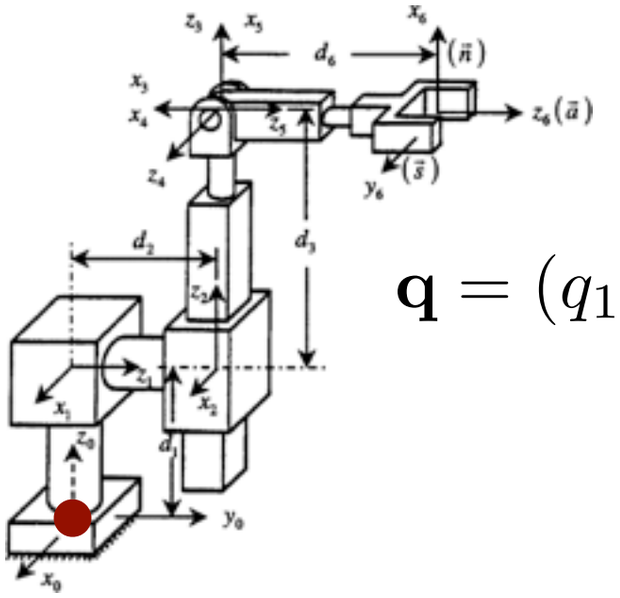


Configuration Space

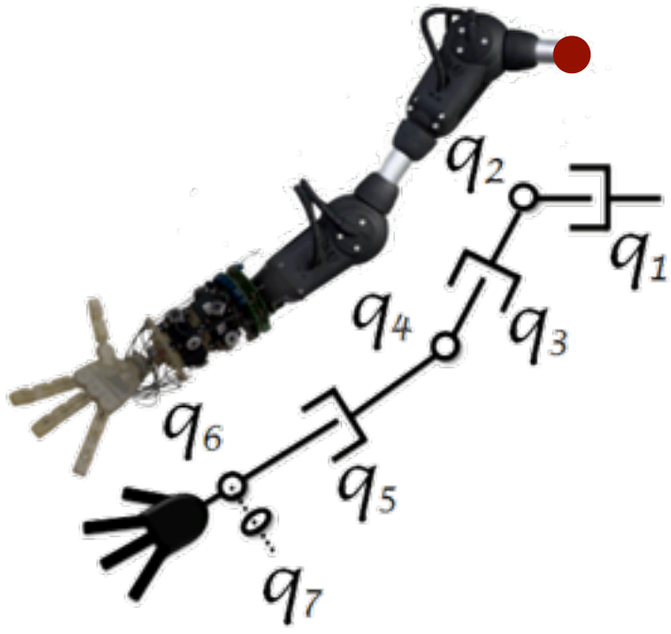
- More complex examples



$$\mathbf{q} = (q_1, q_2, \ldots, q_{10})$$



$$\mathbf{q} = (q_1, q_2, \ldots, q_6)$$

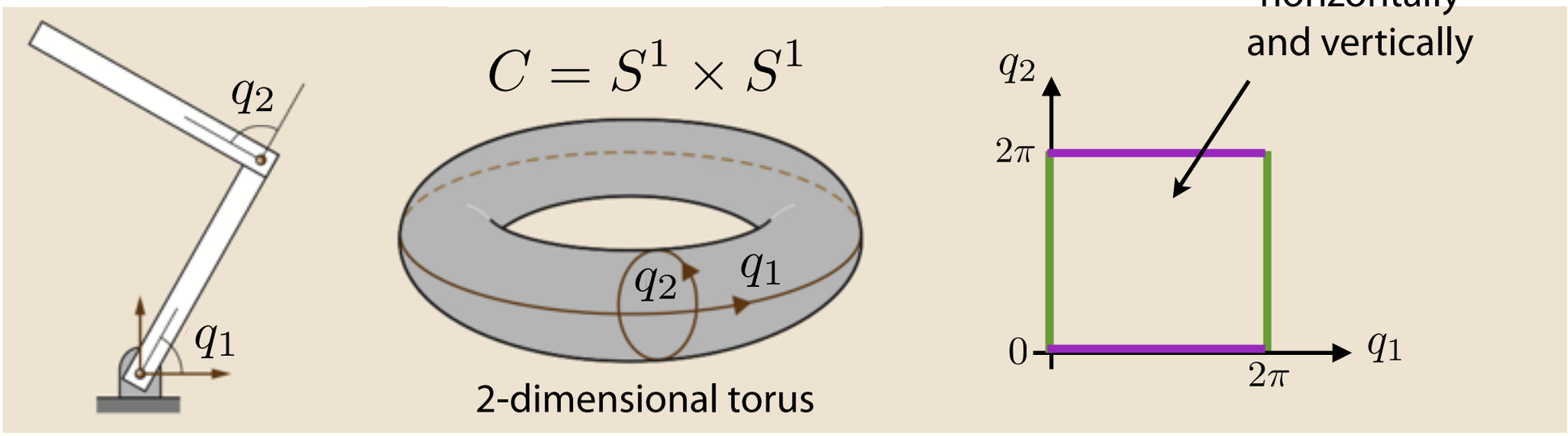


$$\mathbf{q} = (q_1, q_2, \ldots, q_7)$$

Configuration Space

- The configuration space (also called C-space) is the **space of all possible configurations**
- The topology of this space is usually **not** that of a Cartesian space
- The C-space is described as a **topological manifold**

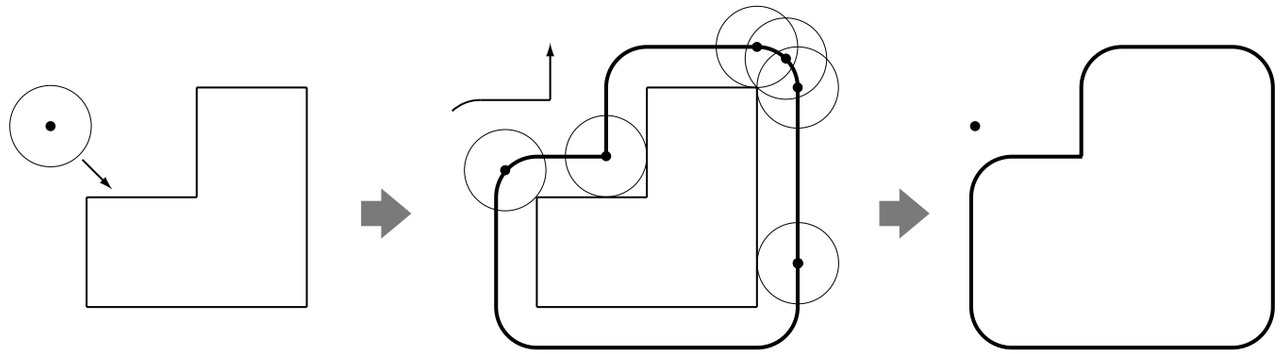
- Example: 2-joint revolute arm in the plane



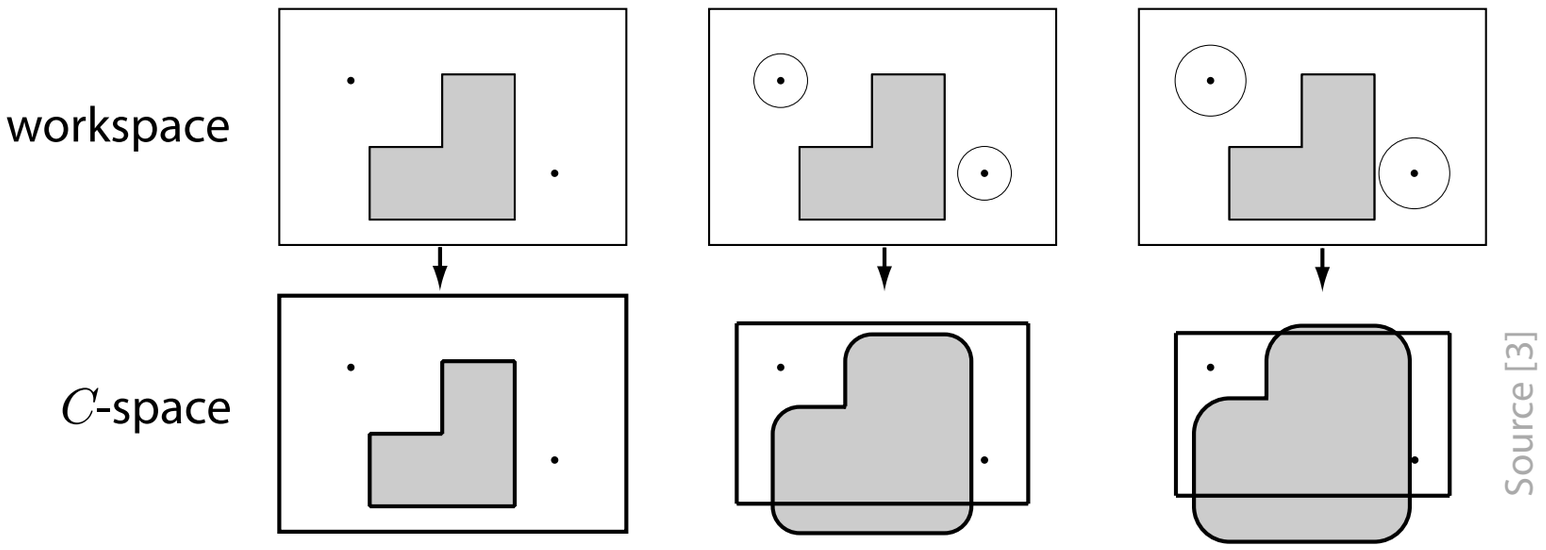
Source [5]

Configuration Space

- Example: circular mobile robot in 2D



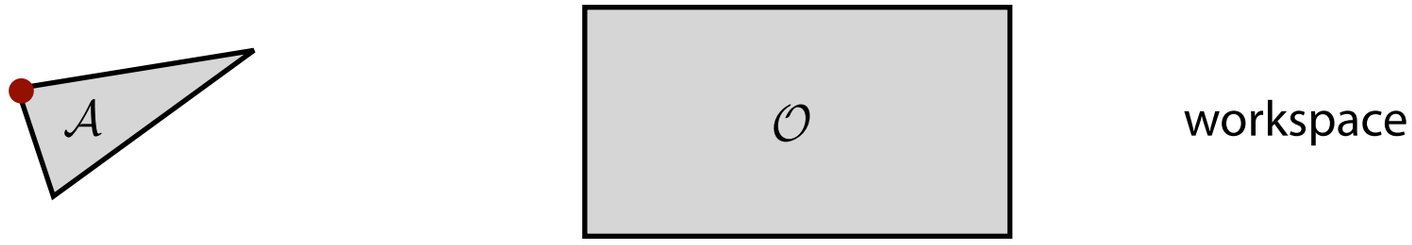
- The C-space is obtained by **sliding the robot along the edge** of the obstacle regions, "blowing them up" by the robot radius

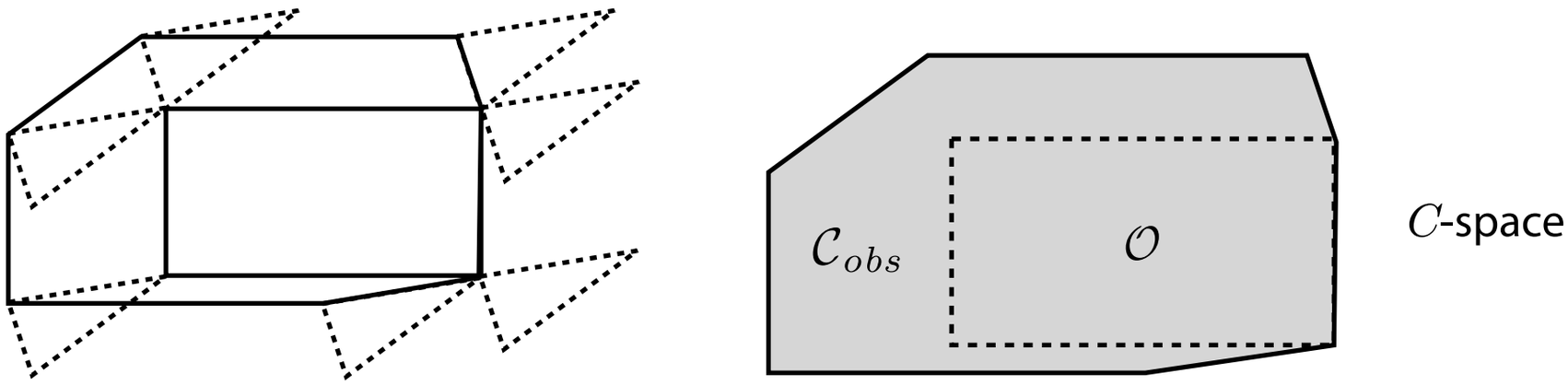


Source [3]

Configuration Space

- Example: polygonal robot, translation only

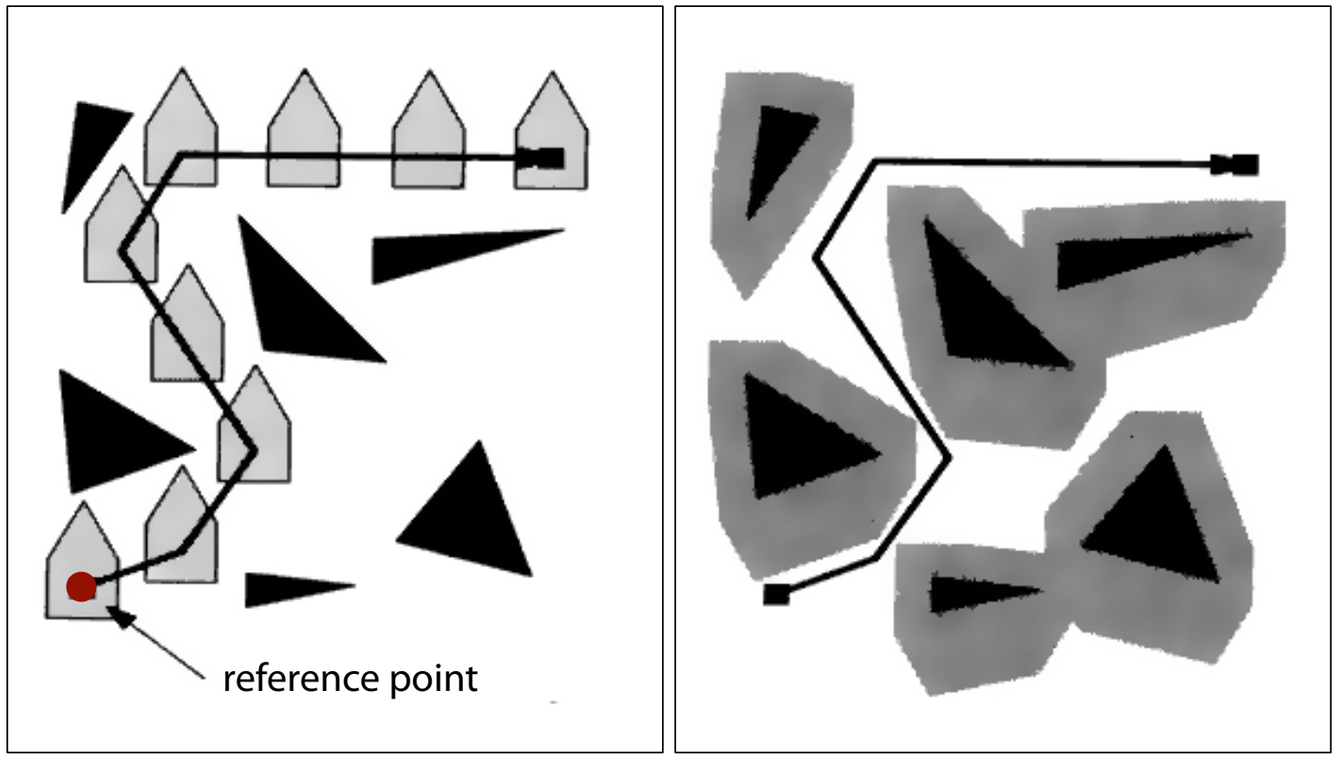




- The C-space is obtained again by **sliding the robot along the edge** of the obstacle regions

Configuration Space

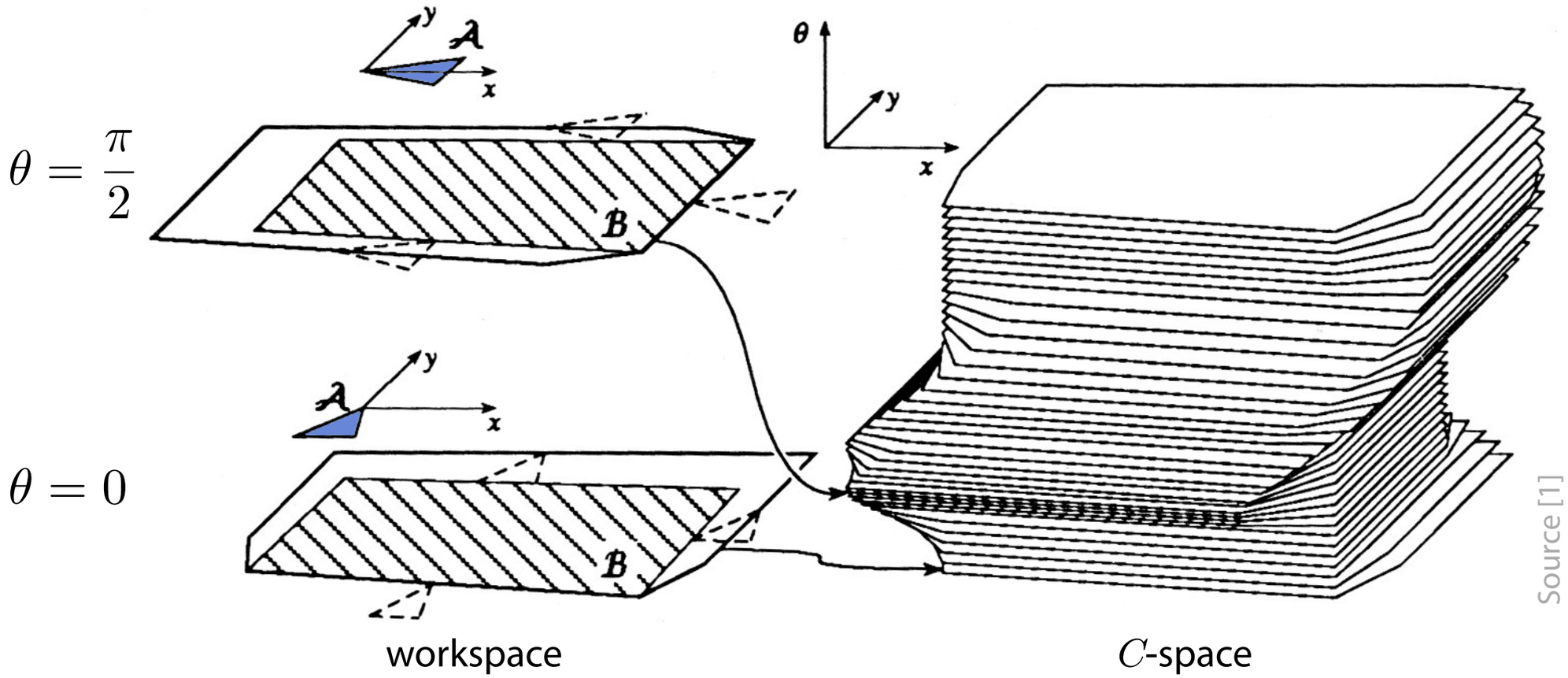
- Example: polygonal robot, translation only



- The C-space is obtained again by **sliding the robot along the edge** of the obstacle regions

Configuration Space

- Example: polygonal robot, **translation and rotation**



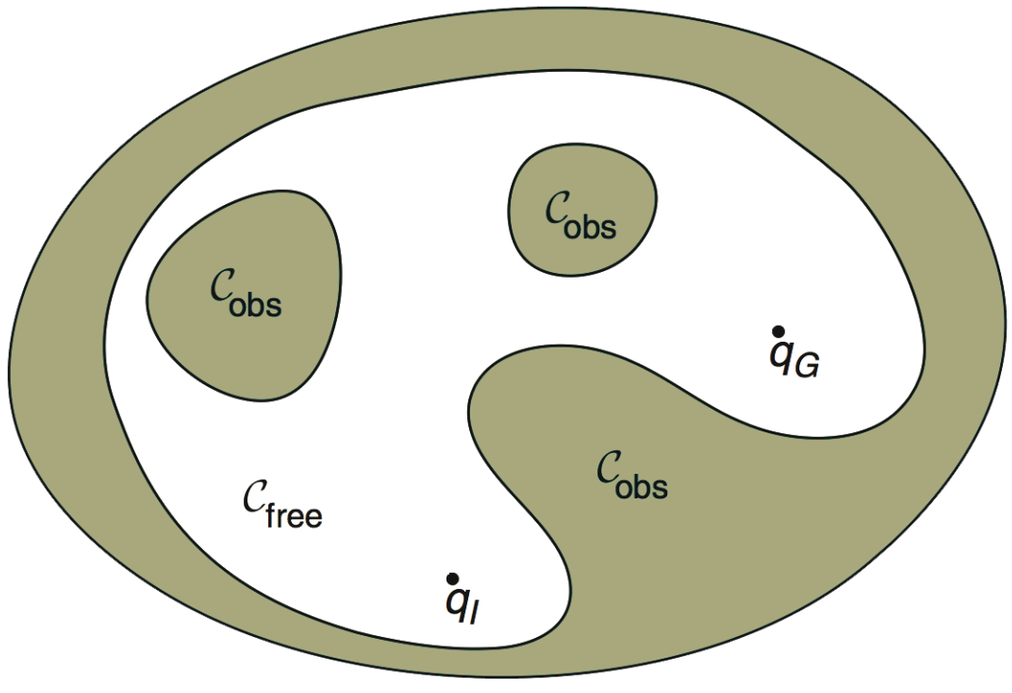
Source [1]

- The C-space is obtained by sliding the robot along the edge of the obstacle regions **in all orientations**

Configuration Space

- Configuration spaces are made up of **free space** and **obstacle regions**
- With $\mathcal{W} = \mathbb{R}^m$ being the workspace, $\mathcal{O} \in \mathcal{W}$ the set of obstacles, $\mathcal{A}(q)$ the robot in configuration $q \in \mathcal{C}$

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



- We further define

  q_I : **initial** configuration

  q_G : **goal** configuration

- Note that $\mathcal{O}$ is defined in the workspace and $\mathcal{C}_{obs}$ in the C-space

Configuration Space

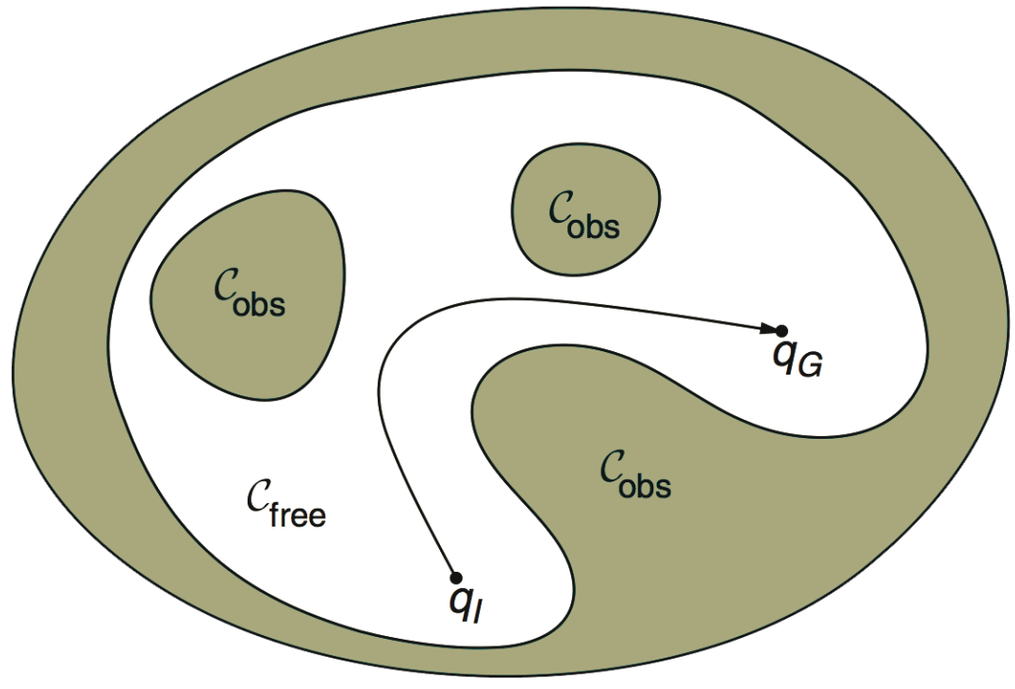
- Then, motion planning amounts to **finding a continuous path**

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

with

$$\tau(0) = q_I \ , \ \tau(1) = q_G$$

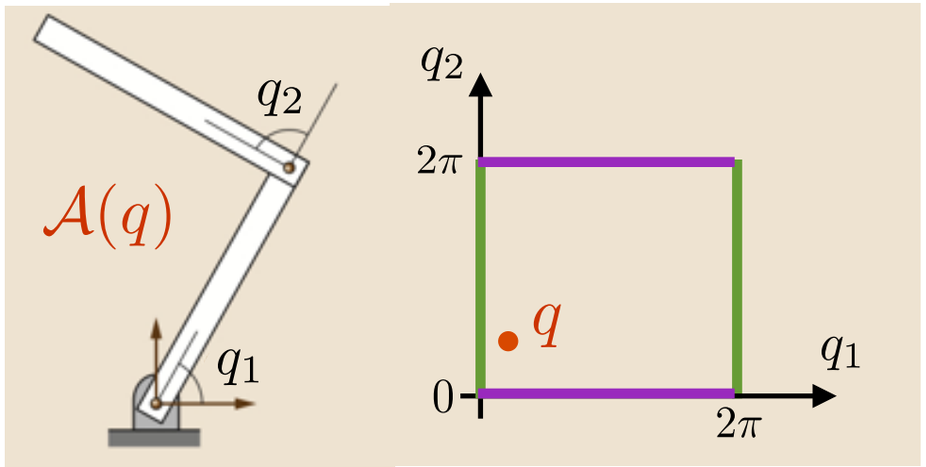
such that no configuration in the path causes a collision between the robot and an obstacle

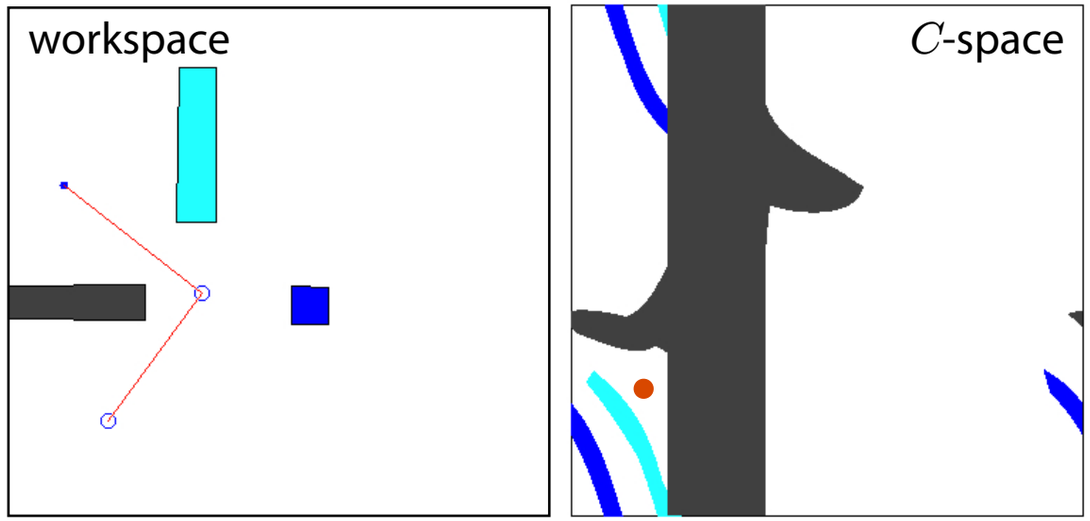


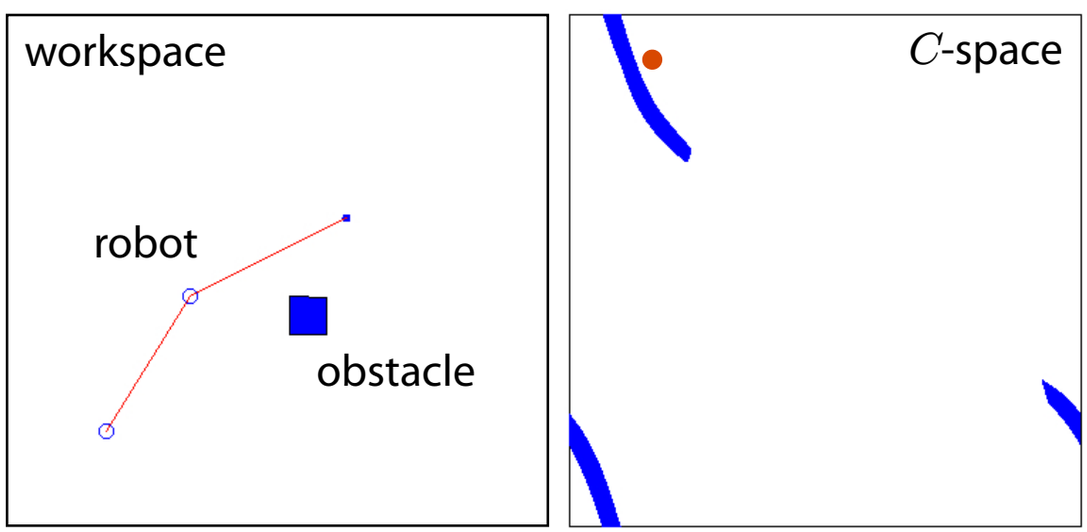
- What do we gain?
- Given this setting, we can do planning with the robot being **a point in C-space!**

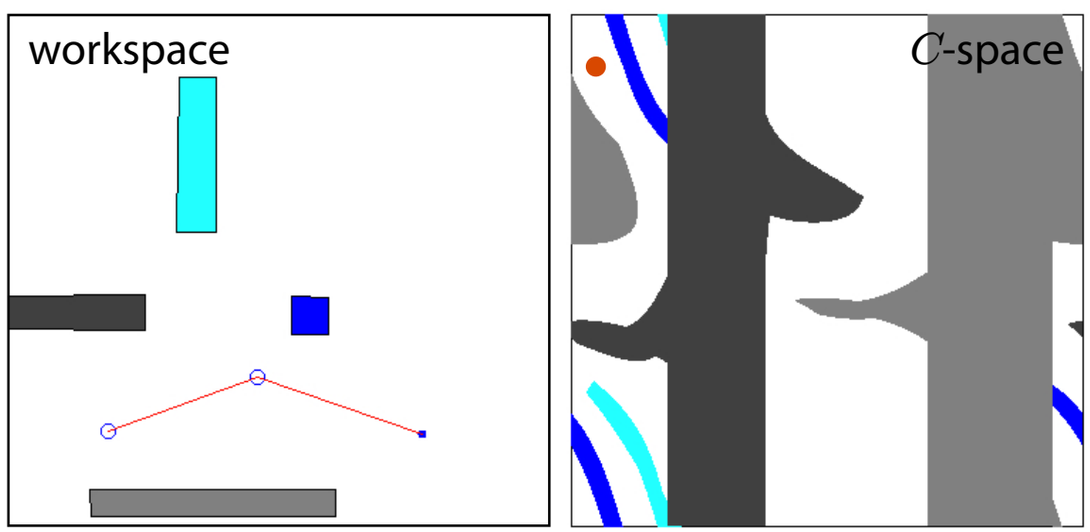
Configuration Space

- Example: 2-joint revolute arm in the plane with obstacles









Source [6]

Configuration Space Discretizations

- In practice, continuous spaces need to be **discretized** for path planning. There are two general approaches to discretize C-spaces:
  - **Combinatorial planning**
    Characterize $\mathcal{C}_{free}$ explicitly by capturing the connectivity of $\mathcal{C}_{free}$ into a graph. Such graphs are also known as roadmaps
  - **Sampling-based planning**
    Randomly probe and incrementally search $\mathcal{C}_{free}$ for a solution. Construct a graph that consist of sampled configurations

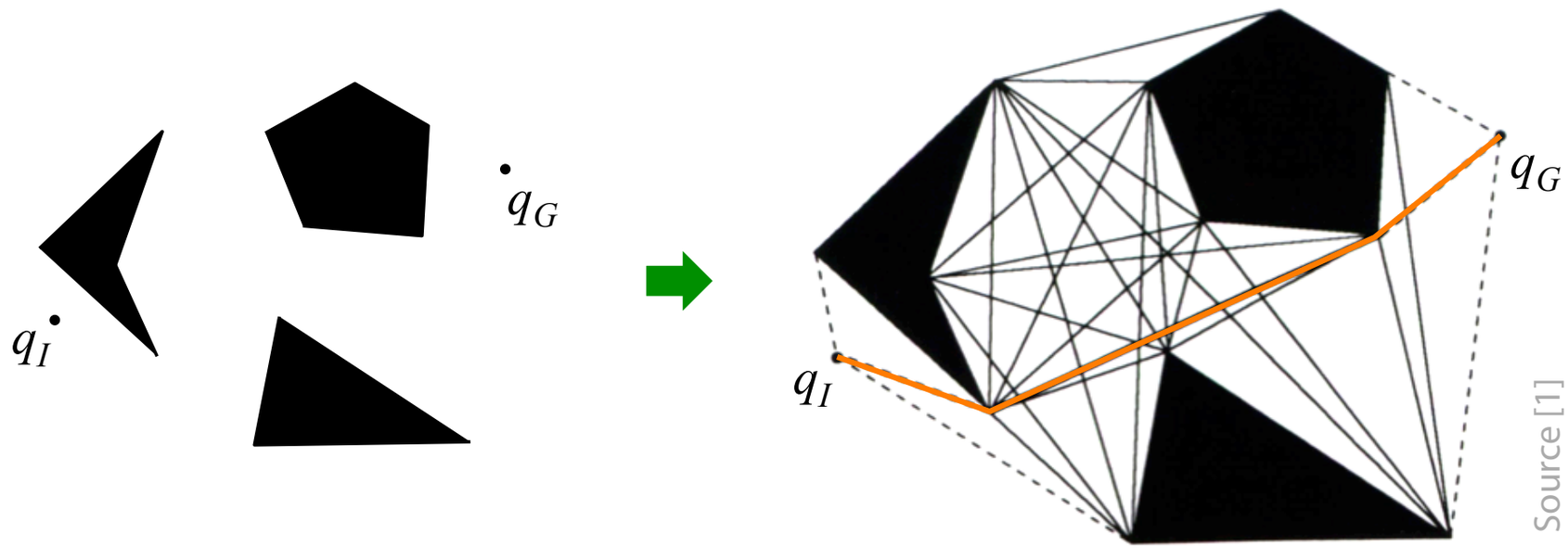
- We will first consider four **combinatorial planning techniques**
  - Visibility graphs
  - Exact and approximate cell decompositions
  - Voronoi diagrams

Roadmap

- The goal of all these approaches is to produce a **roadmap** RM
- A **roadmap** RM is a concise representation of $\mathcal{C}_{free}$ in form of a **graph** that captures its connectivity. Each **vertex** is a configuration in $\mathcal{C}_{free}$ and each **edge** is a collision-free path through $\mathcal{C}_{free}$
- For a roadmap, the following properties hold
  - There is a path from $q_I \in \mathcal{C}_{free}$ to some $q'_I \in RM$
  - There is a path from some $q'_G \in RM$ to $q_G \in \mathcal{C}_{free}$
  - There is a path in RM between q'_I and q'_G
- Given a roadmap, a **planner** can plan paths between configurations using graph-based **search**
- Without loss of generality, we will now consider polygonal worlds in $\mathcal{W} = \mathbb{R}^2$ and a point robot that cannot rotate, that is: $\mathcal{C} = \mathbb{R}^2$

Visibility Graphs

- **Idea:** construct a **graph** of all **intervisible vertices** of obstacles $\mathcal{C}_{obs}$ and plan a path that connects q_I with q_G through those vertices
  - Obstacle edges also serve as edges in the graph, q_I, q_G are also vertices
- Graph contains shortest path among polygonal obstacles in the plane
- Best algorithm is $O(n^2 \log n)$ where n is the number of vertices
- One of the earliest path planning methods (late 1970s)



Exact Cell Decompositions

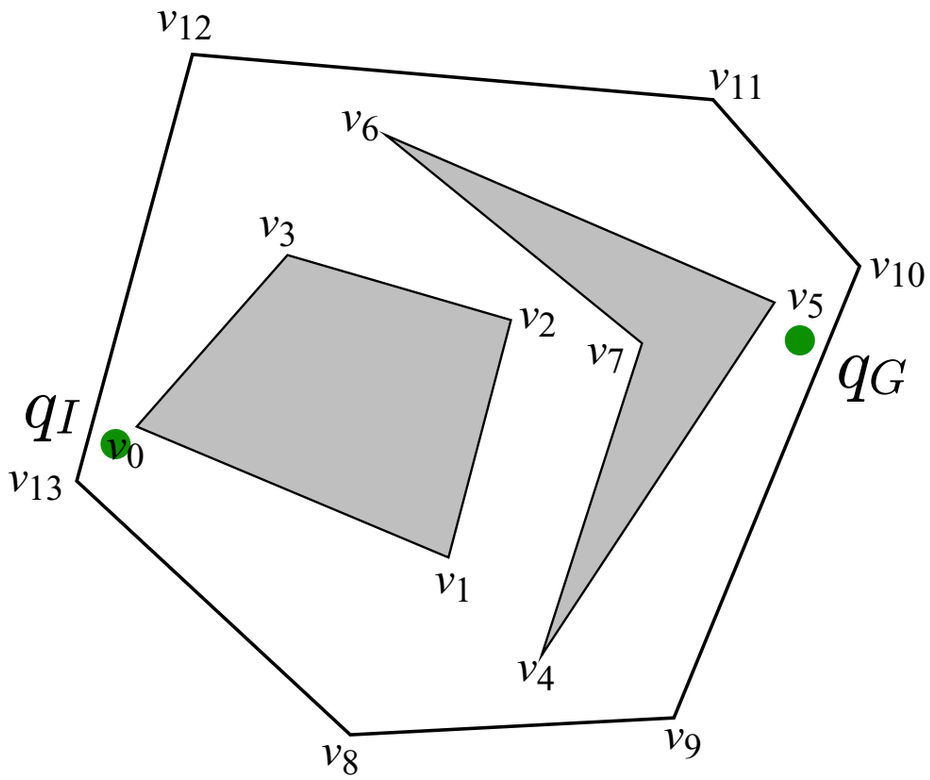
- **Idea:** decompose $\mathcal{C}_{free}$ into **non-overlapping cells**, then construct connectivity graph to represent adjacencies
- A well-known method is the **trapezoidal decomposition**

  1. Decompose $\mathcal{C}_{free}$ into **trapezoids** with vertical side segments by shooting rays upward and downward from each polygon vertex
  2. Place **one vertex** in the interior of every **trapezoid**, pick e.g. the centroid
  3. Place **one vertex** at the midpoint of every vertical **segment**
  4. Connect the vertices to form the **adjacency graph**

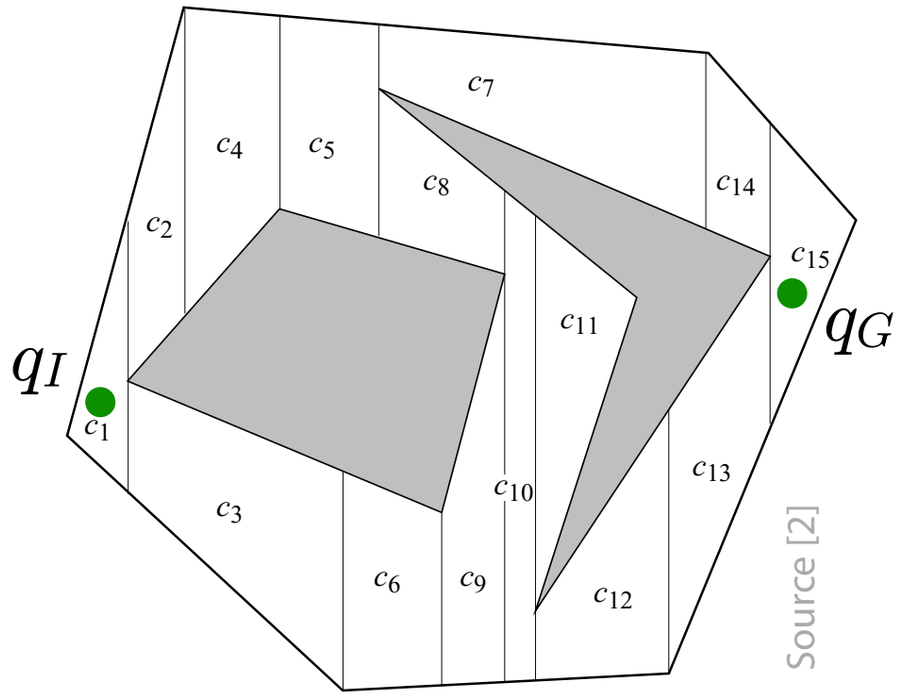
- Method not defined for more than two dimensions
- Best algorithm: $O(n \log n)$ where n is the number of vertices of $\mathcal{C}_{obs}$

Exact Cell Decomposition

- Trapezoidal decomposition



initial problem

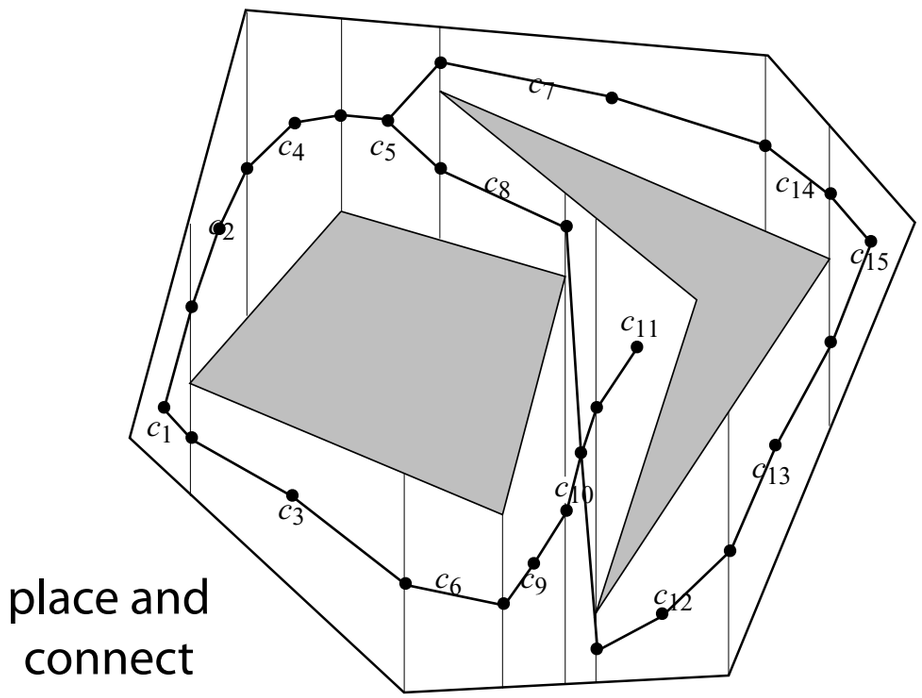


construct trapezoidal cells

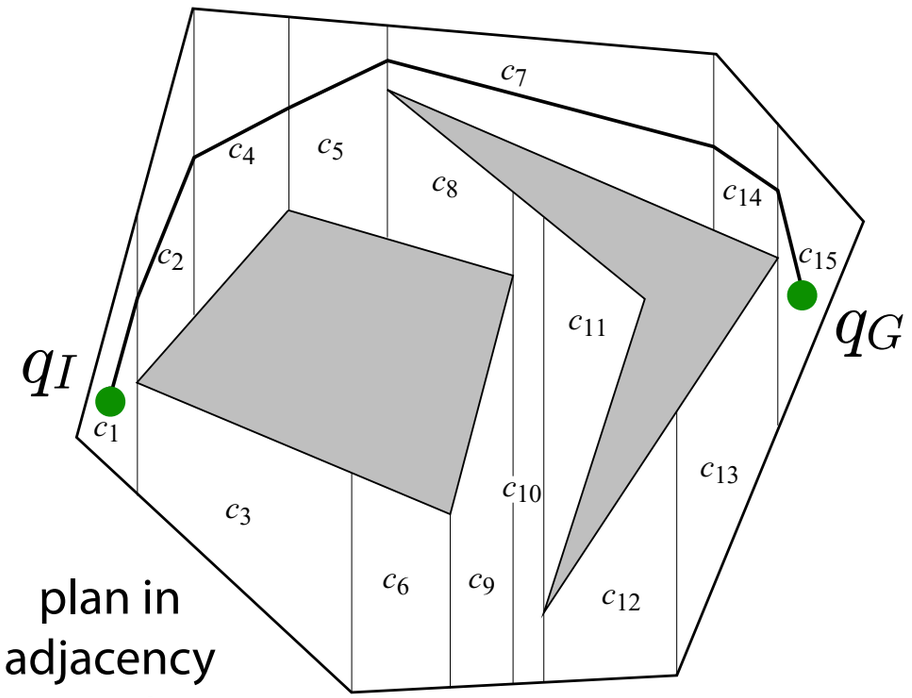
Source [2]

Exact Cell Decomposition

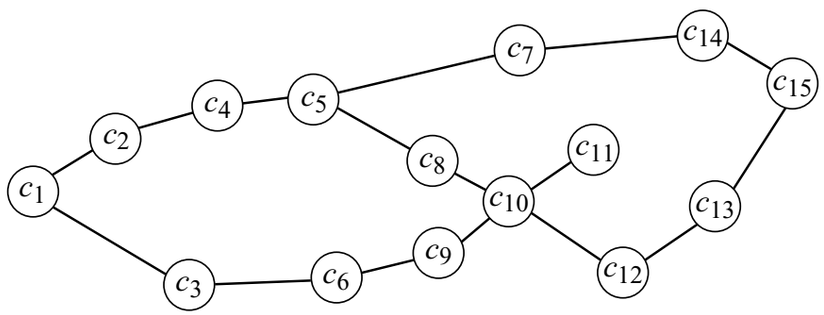
- Trapezoidal decomposition

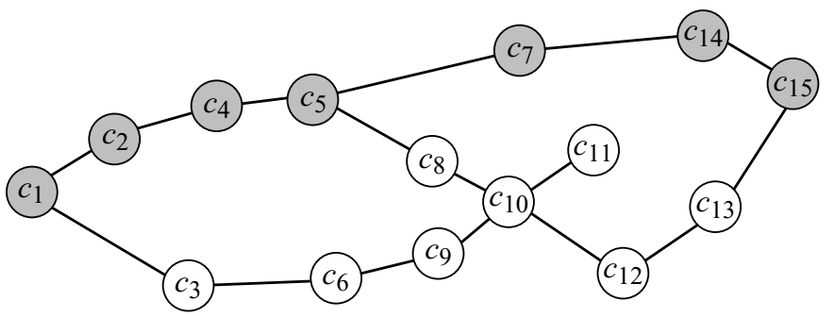


place and connect vertices



plan in adjacency graph

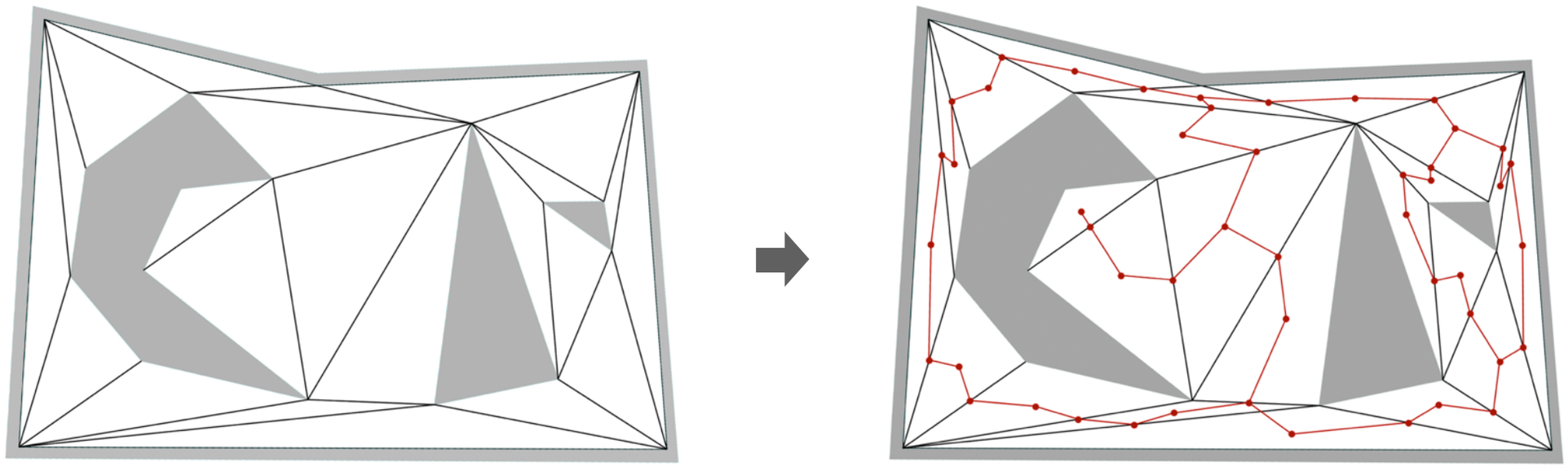




Source [2]

Exact Cell Decomposition

- An alternative approach is to perform a **triangulation**, which in $\mathbb{R}^2$, is a tiling composed of triangular regions

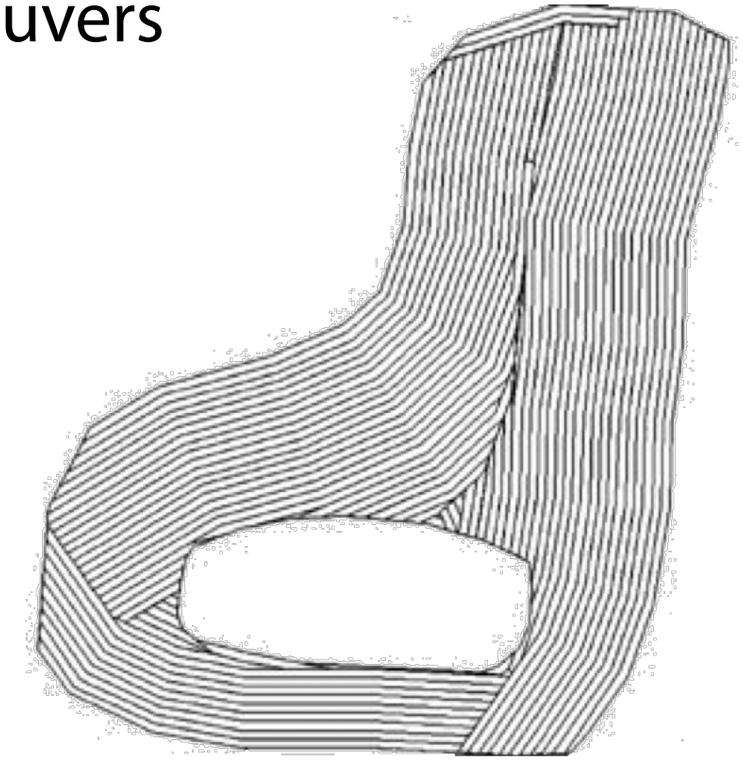


Source [2]

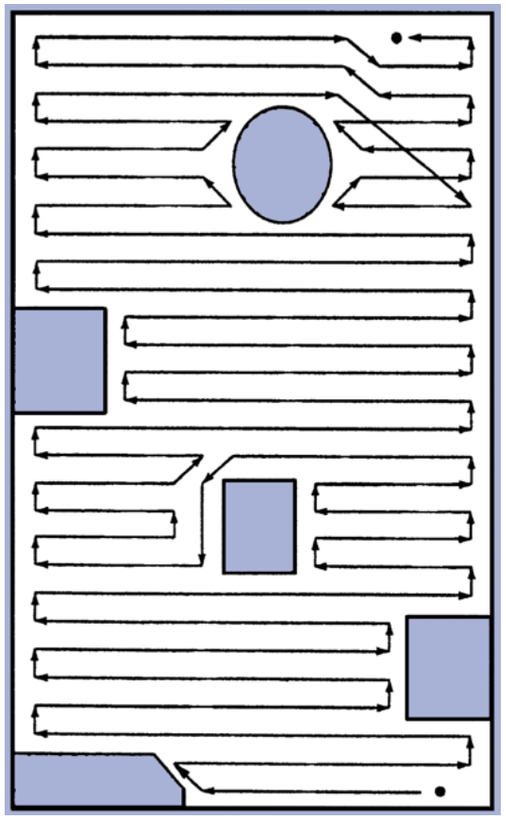
- There are many ways to triangulate $\mathcal{C}_{free}$. Finding the optimal triangulation is exponentially complex in n
- The problem of characterizing a space, capturing its connectivity into a graph, or subdividing an object into simplices (triangles in 2D, tetrahedra in 3D, etc.) are well known problems in **computational geometry**

Exact Cell Decomposition for Coverage Planning

- Cell decompositions can generally be used to achieve **coverage of** $\mathcal{C}_{free}$
- A **coverage path planner** determines a path that passes an end-effector (a mobile robot, a spray can, a sensor) over **all points** in $\mathcal{C}_{free}$
- The assumption is that since each cell has a simple structure, it can be covered with simple motions such as back-and-forth maneuvers
- Coverage planning has applications in
  - **floor care** (e.g. domestic vacuum robots)
  - **farming** (agricultural field robots)
  - robotic **demining**



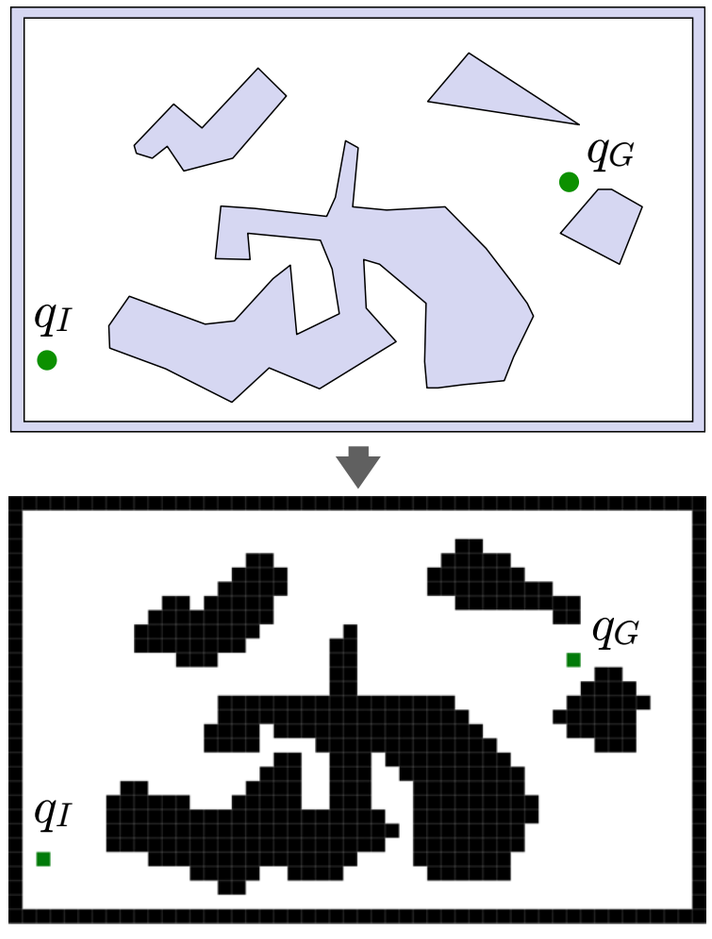
example acre



example living room

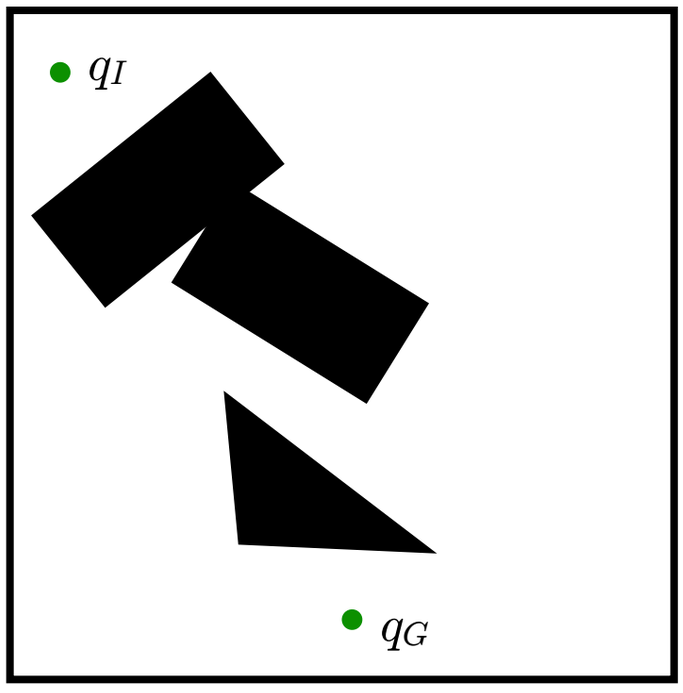
Approximate Cell Decompositions

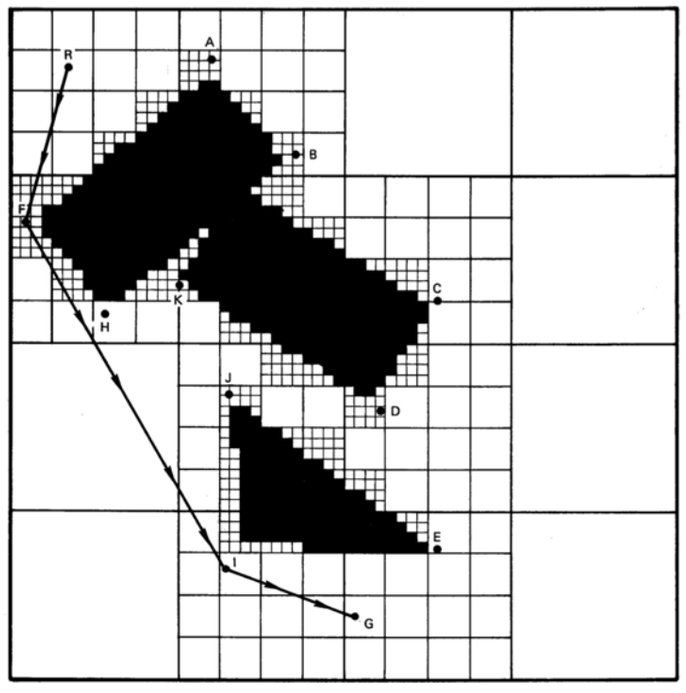
- Exact decomposition methods can be **involved** to implement and **inefficient** to compute for complex problems (large and/or high-dimensional C-spaces, non-polygonal obstacles, etc.)
- One approach is to approximate $\mathcal{C}_{free}$ by cells with the **same** simple predefined shape
- The simplest case is a **grid of rectangular cells** that are either **free** or **occupied**
- The **graph** is built from nodes at cell corners or cell centers and 4- or 8-connected edges
- The resolution of this discretization determines the number of cells and the quality of the approximation



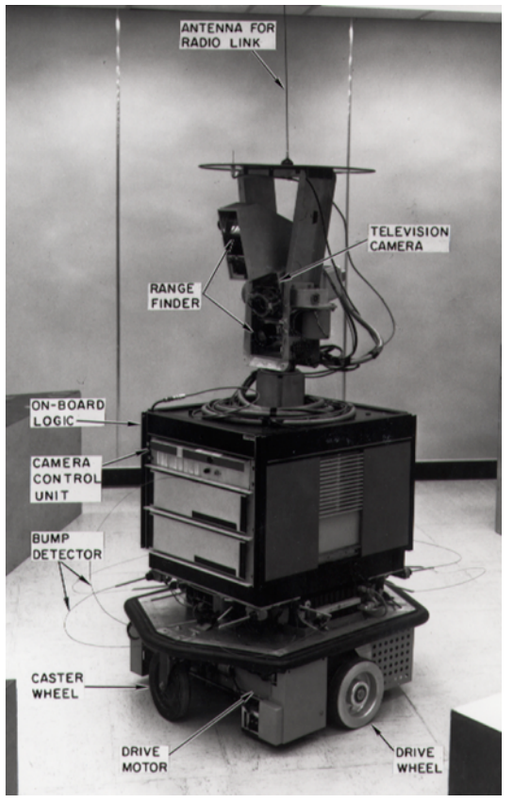
Approximate Cell Decompositions

- An efficient variation of this concept are **quadtrees** in 2D or **octrees** in 3D





Source [8]

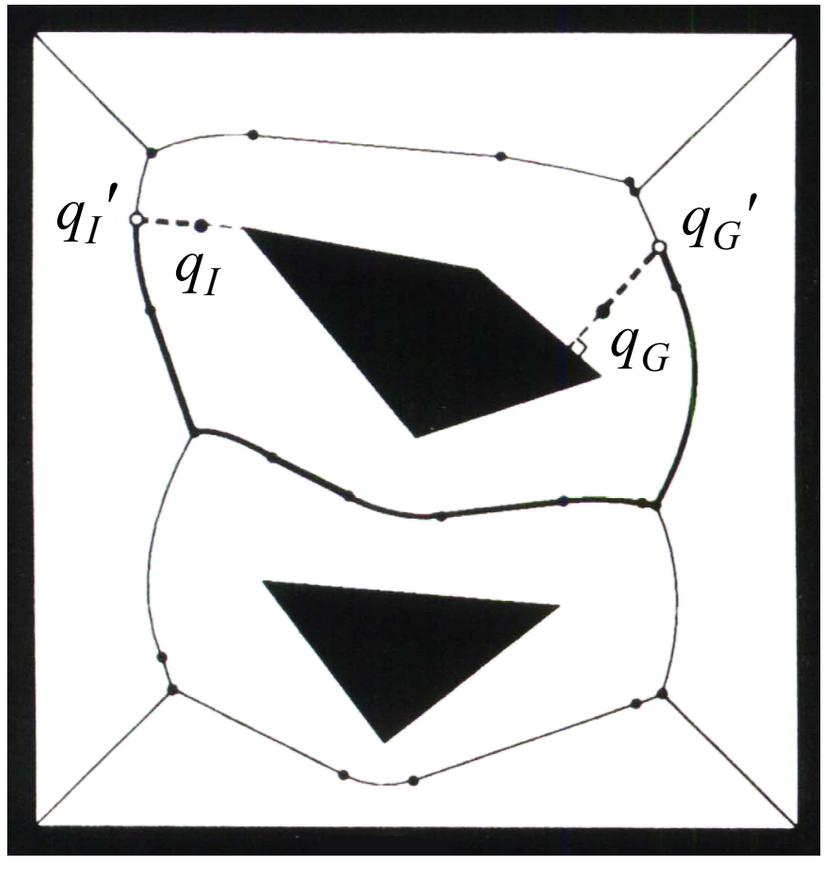


Source [8]

- First used on Shakey the robot (late 1960s)
- First AI-controlled robot. Research on Shakey had several spin-offs: grid-based path planning, visibility graphs, A*

Generalized Voronoi Diagram (GVD)

- Defined to be the set of points q whose cardinality of the set of boundary points of $\mathcal{C}_{obs}$ with the same distance to q is greater than 1
- In other words, the Voronoi diagram is the set of points where the **clearance** to the **closest obstacles is the same**
- Regular Voronoi diagrams are defined for point obstacles only. In the planar case, the diagram is then a collection of line segments
- Since obstacles are not points, the **generalized Voronoi diagram** (GVD) is defined for general extended obstacles



Source [1]

Generalized Voronoi Diagram

- Formally, let $\beta = \partial\, \mathcal{C}_{free}$ be the boundary of $\mathcal{C}_{free}$, and $d(p, q)$ the Euclidian distance between p and q. Then, for all q in $\mathcal{C}_{free}$, let

$$clearance(q) = \min_{p \in \beta} d(p, q)$$

be the clearance of q, and

$$near(q) = \{p \in \beta \;\mid\; d(p, q) = clearance(q)\}$$

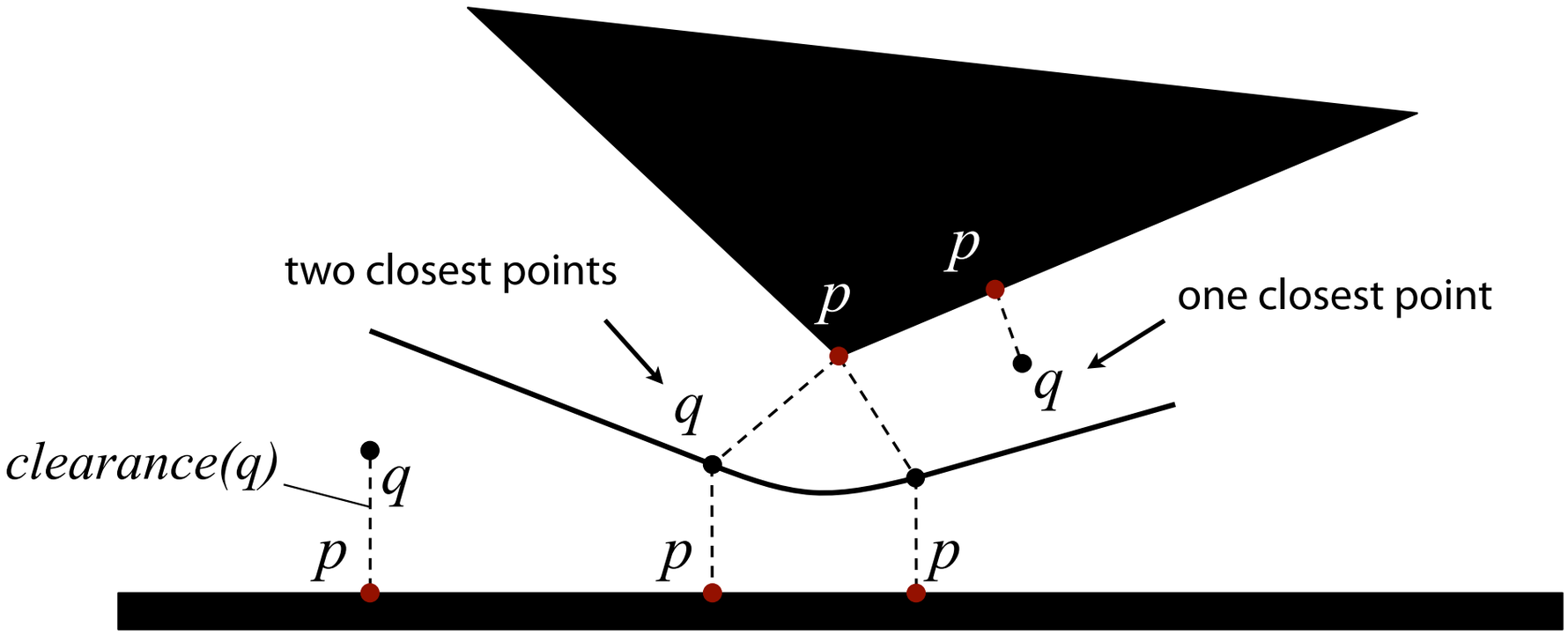
the set of base/foot points on β with the same clearance to q

- The **Generalized Voronoi diagram** (GVD) is then the set of q's with more than one base point p

$$V(\mathcal{C}_{free}) = \{q \in \mathcal{C}_{free} \;\mid\; |near(q)| > 1\}$$

Generalized Voronoi Diagram

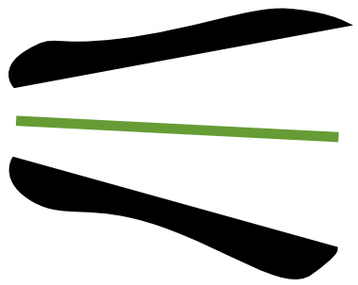
- Geometrically:



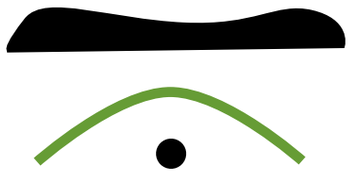
- For **polygonal** $\mathcal{C}_{obs}$, the diagram consists of **lines** and **parabolic edges**
- With n being the number of vertices on β, naive algorithms to construct the diagram have $O(n^4)$, the best algorithm has $O(n \log n)$

Generalized Voronoi Diagram

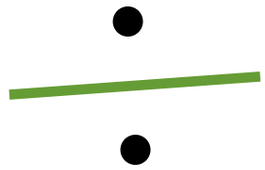
- In 2D, Voronoi edges are formed by three types of interactions



edge – edge
linear edge



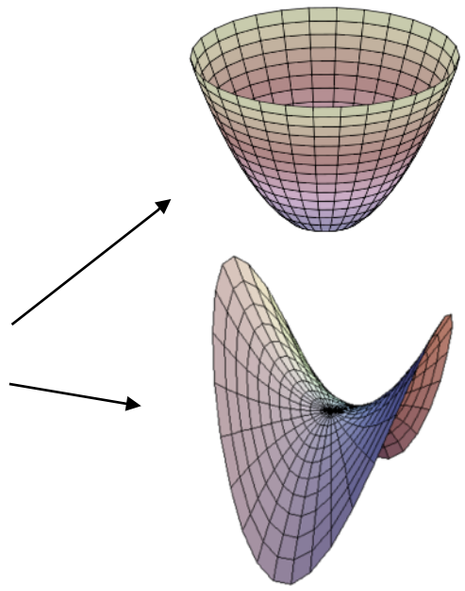
edge – vertex
parabolic edge



vertex – vertex
linear edge

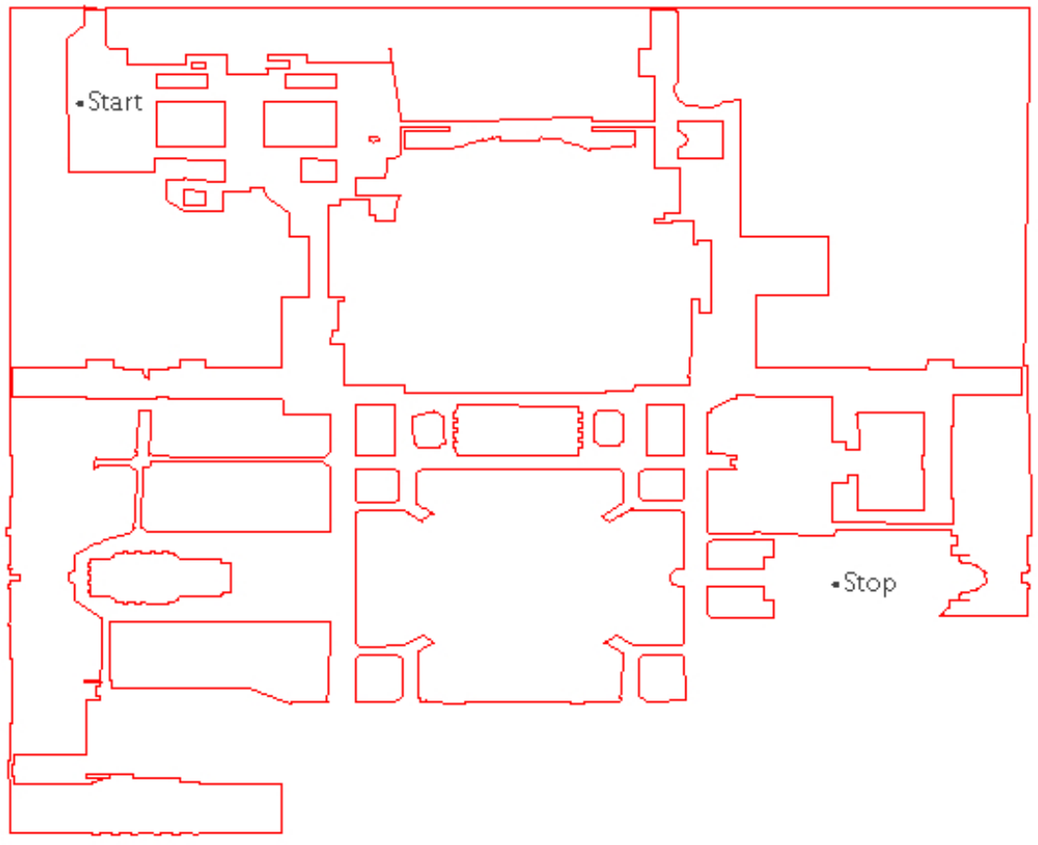
- In 3D, things get more complex

Bisector type	Voronoi element
point – point	plane
point – edge	parabolic cylinder
point – triangle	paraboloid
edge – edge	hyperbolic paraboloid
edge – triangle	parabolic cylinder
triangle – triangle	plane



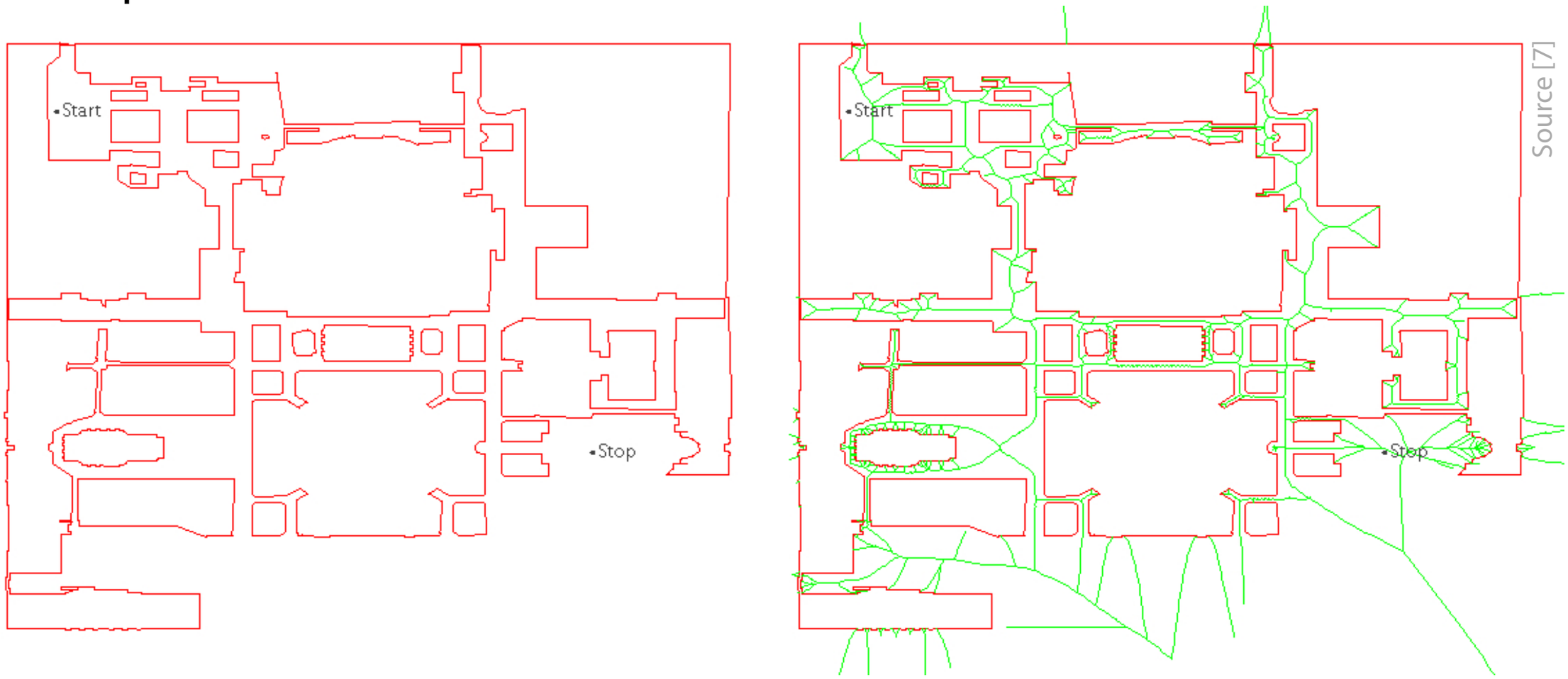
Generalized Voronoi Diagram

- For robot motion planning, Voronoi diagrams can be used to find clear routes which are **furthest from obstacles**
- Example



Generalized Voronoi Diagram

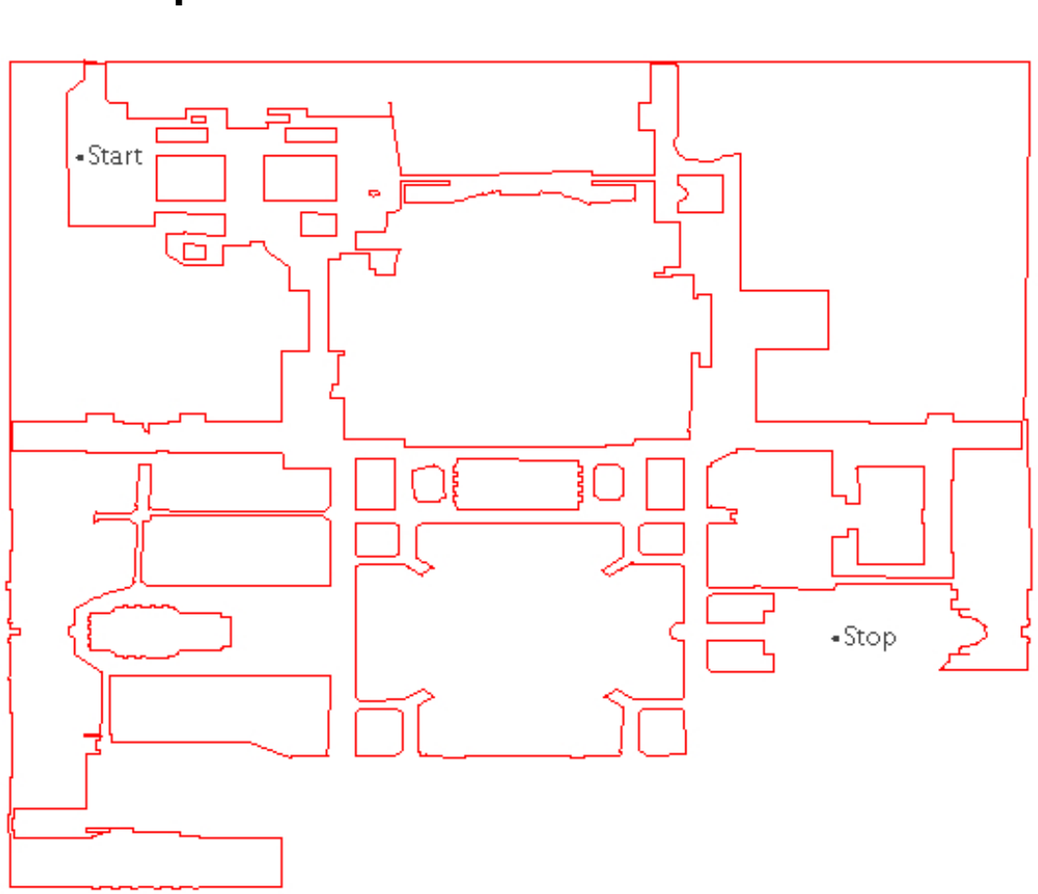
- For robot motion planning, Voronoi diagrams can be used to find clear routes which are **furthest from obstacles**
- Example

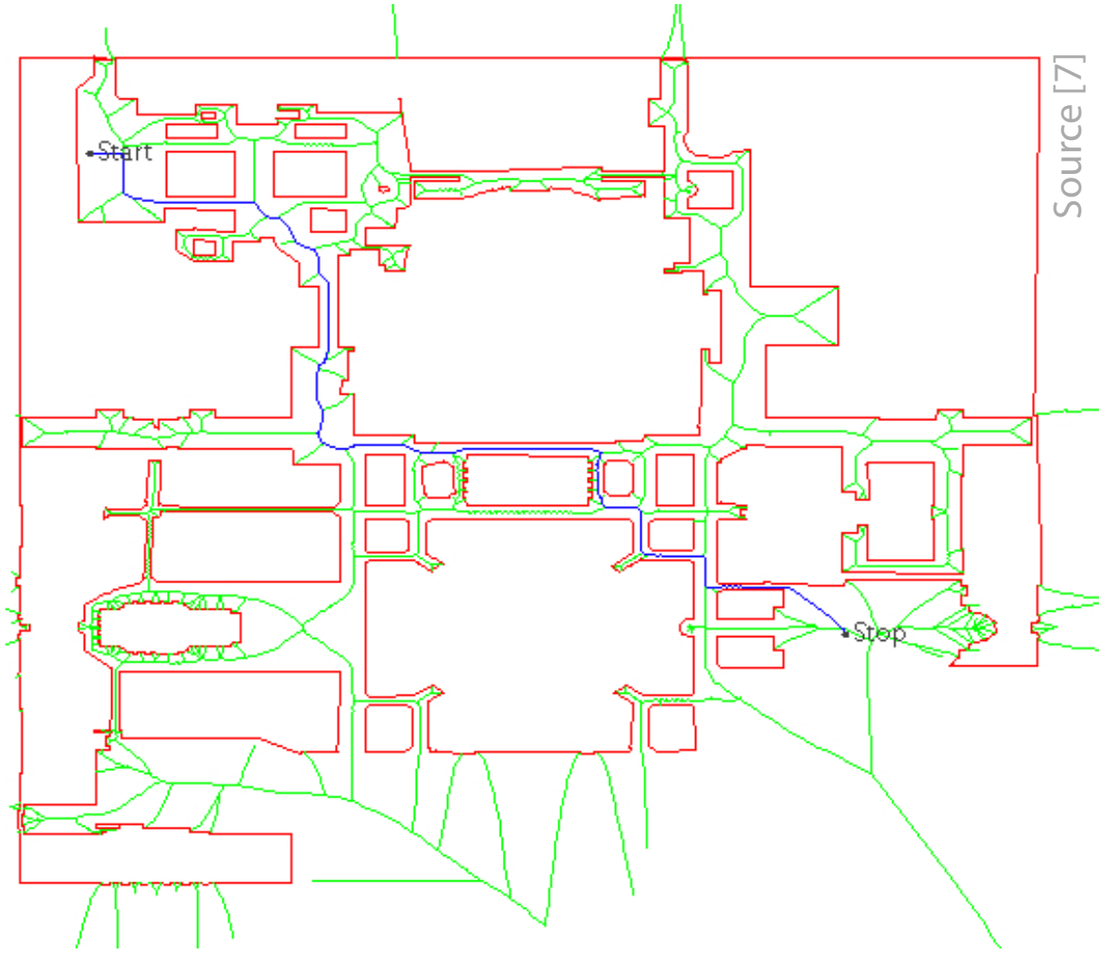


Source [7]

Generalized Voronoi Diagram

- For robot motion planning, Voronoi diagrams can be used to find clear routes which are **furthest from obstacles**
- Example



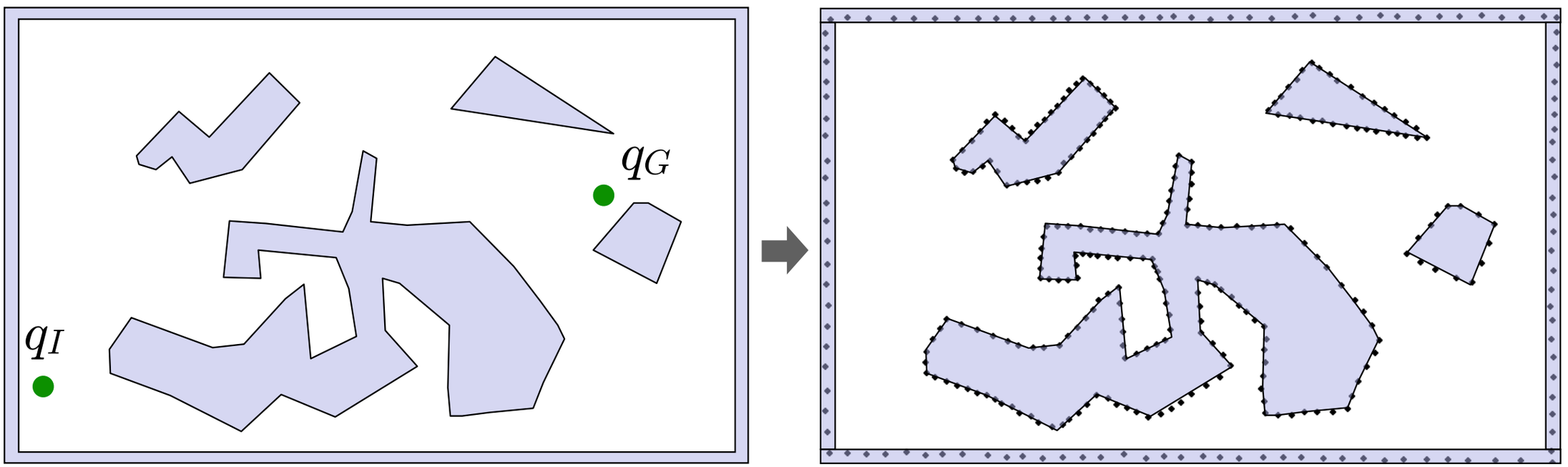


Source [7]

Generalized Voronoi Diagram

- Computing the GVD **exactly** is possible for polygonal C-spaces
- In practice, we need to **approximate** the GVD. Let us consider a method that **discretizes the obstacles**:

  1. Discretize obstacles by discretizing their boundary
  2. Compute regular Voronoi diagram of boundary points

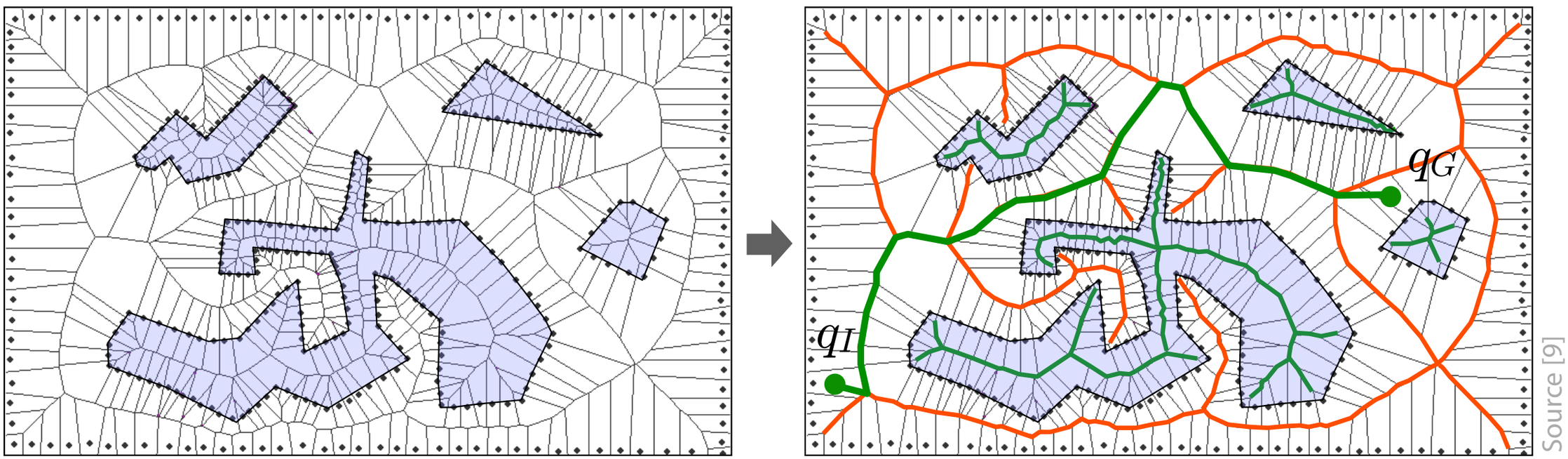


Source [9]

Generalized Voronoi Diagram

- Computing the GVD **exactly** is possible for polygonal C-spaces
- In practice, we need to **approximate** the GVD. Let us consider a method that **discretizes the obstacles**:

1. Discretize obstacles by discretizing their boundary
2. Compute regular Voronoi diagram of boundary points
3. Discard GVD edges from two consecutive points of the same obstacle

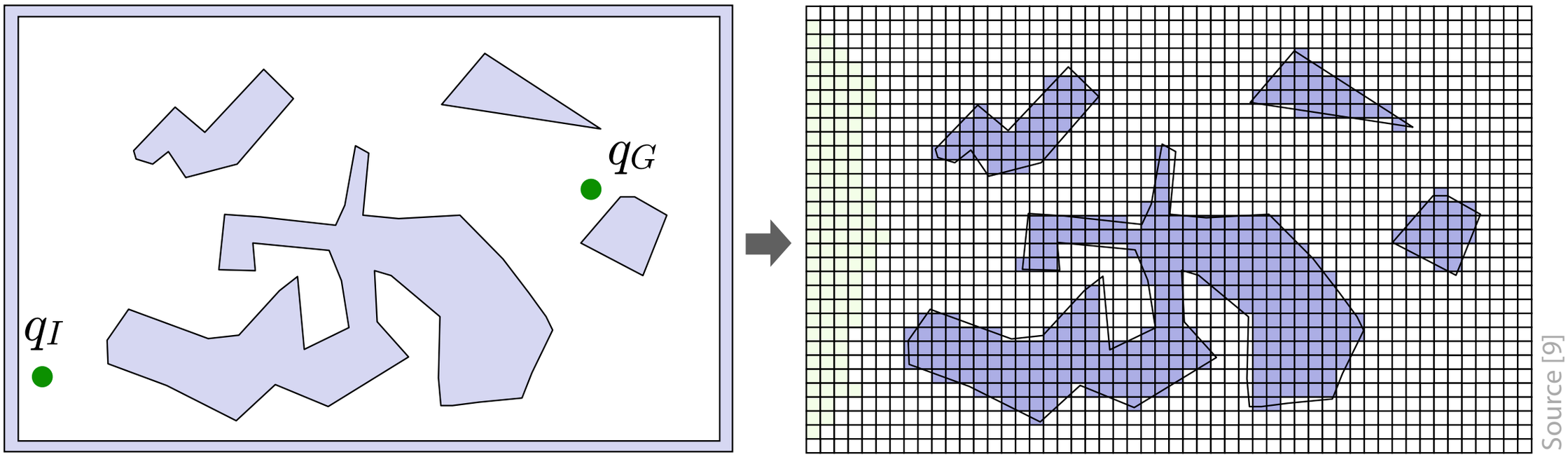


Source [9]

Generalized Voronoi Diagram

- Computing the GVD **exactly** is possible for polygonal C-spaces
- In practice, we need to **approximate** the GVD. Let us consider a method that **discretizes the C-space**:

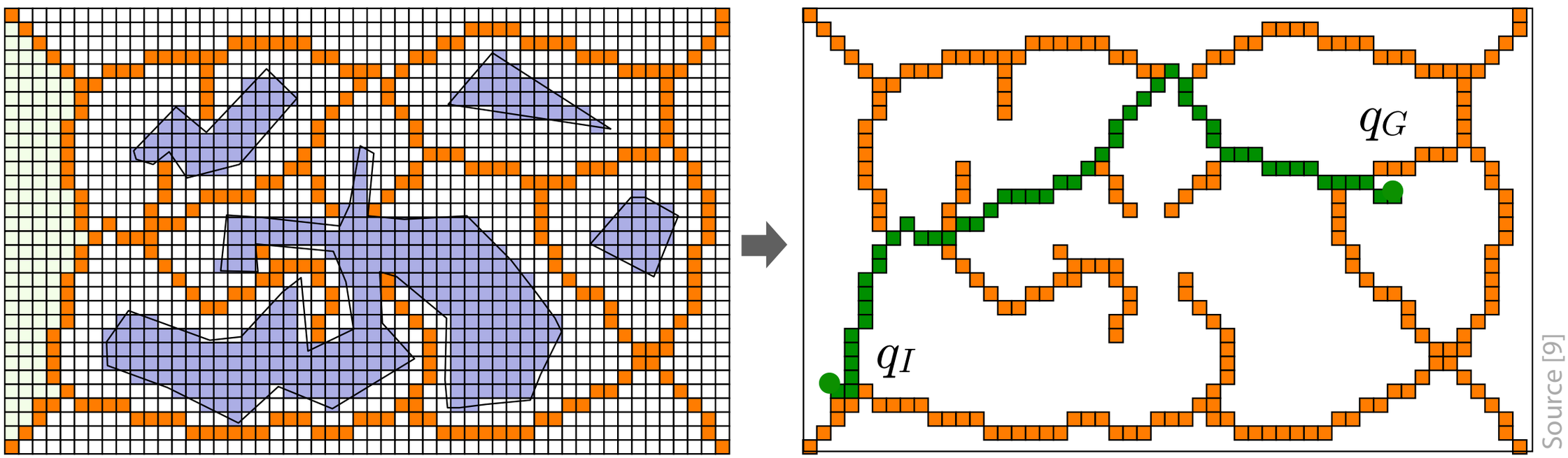
  1. Discretize obstacles onto a regular grid



Source [9]

Generalized Voronoi Diagram

- Computing the GVD **exactly** is possible for polygonal C-spaces
- In practice, we need to **approximate** the GVD. Let us consider a method that **discretizes the C-space**:
  1. Discretize obstacles onto a regular grid
  2. Compute Voronoi diagram by running a wavefront algorithm



Source [9]

Generalized Voronoi Diagram

- Voronoi diagrams have been **well studied** for **mobile robot** motion planning. Maximum clearance paths are a good idea for robots with uncertain plan execution
- For mobile robots, fast methods exist to compute the generalized Voronoi diagram **from sensory data** (e.g. sonar, laser) in real-time
- Produces natural paths in corridor-like environments such as offices or man-made buildings
- In general spaces, however, the Voronoi set has a unnatural attraction to open space and may lead to **far-optimal** paths
- There are **problems** with the GVD in **higher dimensions**. Roadmaps are not connected anymore. A hierarchy of higher-order GVDs based on the nth-closest neighbors helps in some cases.

Combinatorial Planning

- Let us **wrap up**:
- Combinatorial planning techniques are elegant and, more importantly, **complete**: they find a solution if it exists and report failure otherwise
- However, they become quickly intractable when C-space dimensionality increases (or the number of vertices n of $\mathcal{C}_{obs}$, respectively)
- **Combinatorial explosion** in terms of vertices to represent the robot $\mathcal{A}(q)$, the obstacle in the workspace $\mathcal{O}$, and the obstacle in the C-space $\mathcal{C}_{obs}$, especially when rotative degrees of freedom make $\mathcal{C}$ a complex manifold
- Algorithms scale **poorly** in both space and time complexity

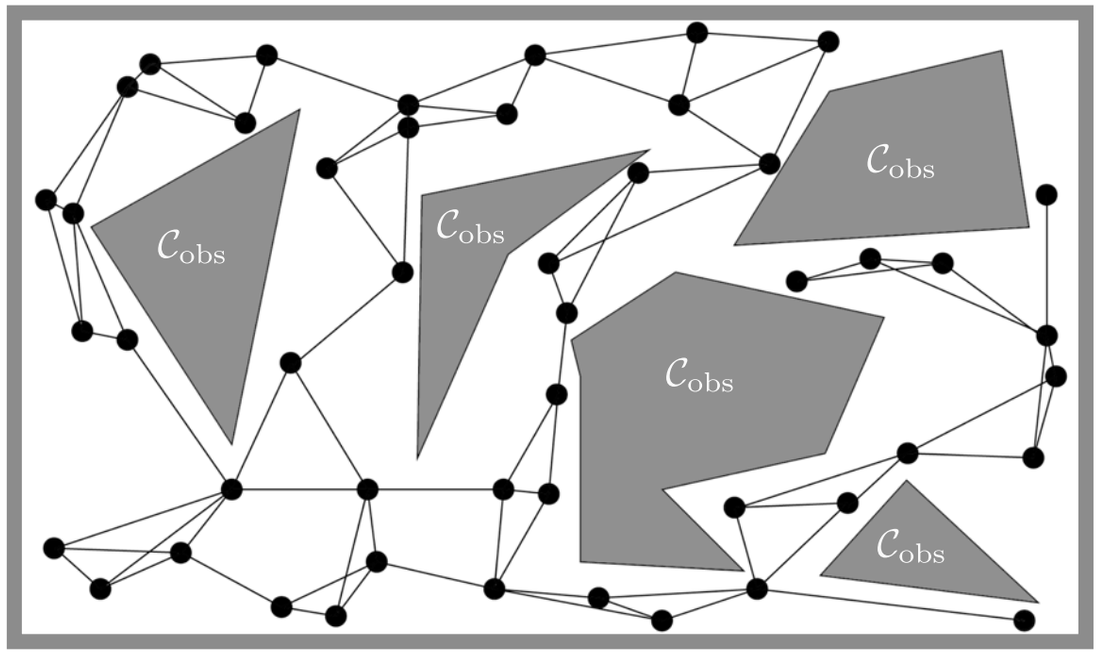
- We therefore consider an alternative group of approaches to discretize the configuration space: **sampling-based planning**

Sampling-based planning

- In practical planning problems it may be difficult to explicitly represent $\mathcal{C}_{free}$ but testing whether a given configuration is in $\mathcal{C}_{free}$ is easy and fast
- Thus, we abandon the concept of explicitly characterizing $\mathcal{C}_{free}$ and $\mathcal{C}_{obs}$ and leave the algorithm **in the dark** when exploring $\mathcal{C}_{free}$
- The only "light" is provided by a **collision-detection algorithm**, that probes $\mathcal{C}$ to see whether some configuration lies in $\mathcal{C}_{free}$
- Using such methods, we **trade completeness guarantees** for a **reduction in planning time**
- We will consider two popular sampling-based methods that implement this idea:
  - **Probabilistic road maps** (PRM)
  - **Rapidly exploring random trees** (RRT)

Probabilistic Roadmaps (PRM)

- **Idea:** we approximate $\mathcal{C}_{free}$ by randomly drawing samples from $\mathcal{C}$, insert them as vertices into the roadmap if they lie in $\mathcal{C}_{free}$, and then try to connect them with nearby vertices
- Collision checks are carried out in the **workspace**. Given a configuration q, $\mathcal{A}(q)$ is computed and checked for collision with $\mathcal{O}$
  - If collision occurs, q is in $\mathcal{C}_{obs}$
  - Otherwise, q is in $\mathcal{C}_{free}$
- Connecting vertices with the roadmap is the task of a **local planner**. It checks if moving from q to some nearby roadmap vertex q' is collision-free



Source [3]

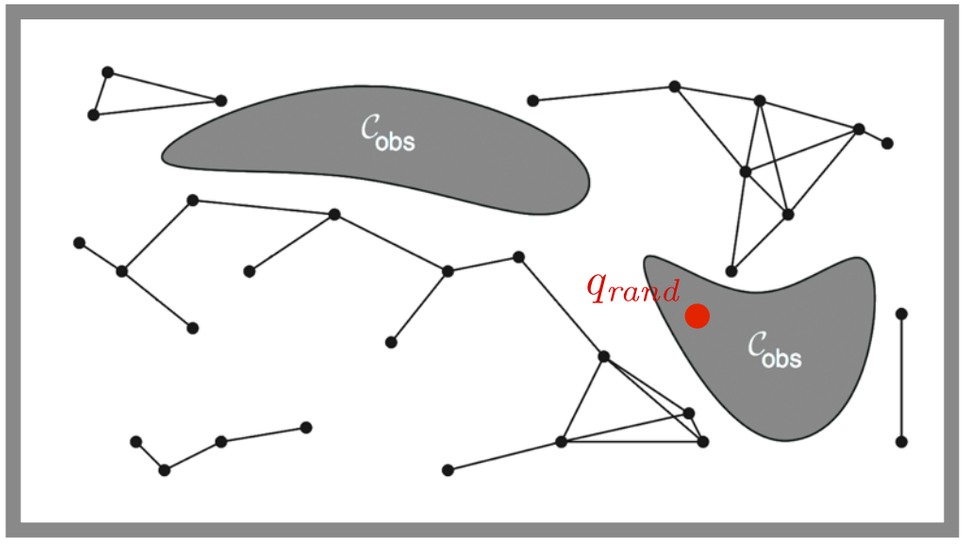
Probabilistic Roadmaps

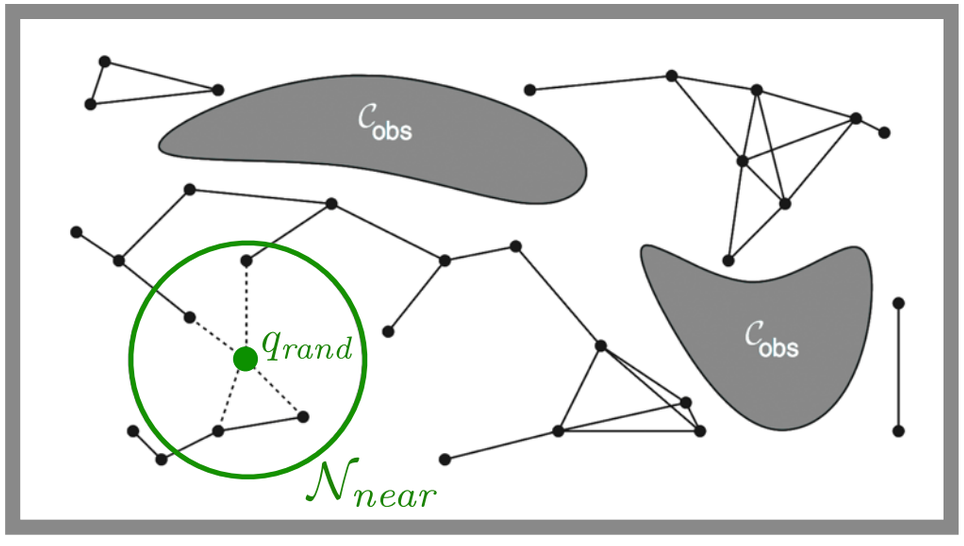
- Basic algorithm to construct PRMs

Algorithm 1: Construct PRM

In: Configuration space $\mathcal{C}$
Out: Roadmap $G = (V, E)$

```
G ← ∅
repeat
    q_rand → RANDOM_CONFIG(C)
    if CLEAR(q_rand) then
        G.add_vertex(q_rand)
        N_near ← NEAREST(G, q_rand)
        for q_near ∈ N_near do
            if LINK(q_rand, q_near) then
                G.add_edge(q_near, q_rand)
            end
        end
    end
until condition
```





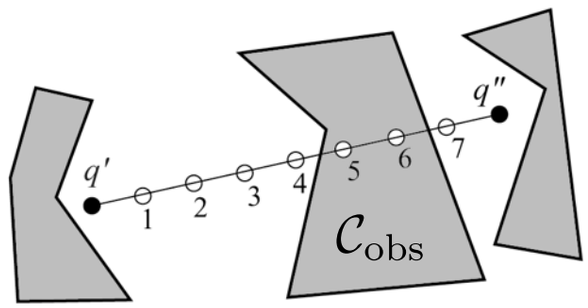
Source [2]

Probabilistic Roadmaps

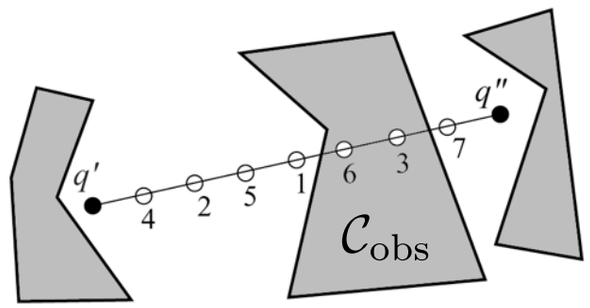
- The algorithm comprises the following steps:
  - **Sampling** (function $RANDOM\_CONFIG$): samples are usually drawn uniformly over $\mathcal{C}$. This is a general scheme that works well for many planning problems. Specialized non-uniform sampling strategies may be needed for difficult problems
  - **Collision check** (function $CLEAR$): carried out in the workspace
  - **Selecting neighbors** (function $NEAREST$): strategies may select the k nearest neighbors or all neighbors within a given radius. kd-trees can be used for speed up
  - **Local planning** (function $LINK$): the most basic local planner, applicable to all holonomic robots, connects two configurations by a straight **line-of-sight segment**. More complex planners may be used, e.g. for robots with non-holonomic motion constraints

Probabilistic Roadmaps

- The **local planner discretizes** the local path to detect collisions, either incrementally or by subdivision. The latter method is faster



incremental: 5 checks needed



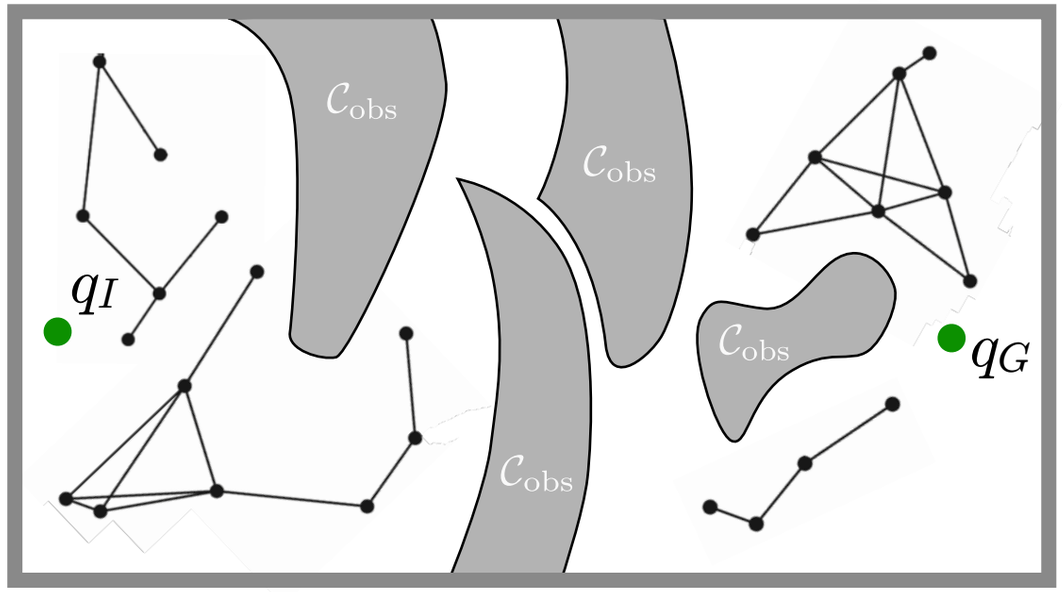
binary: 3 checks needed

Source [3]

- What means "**nearest**"? Choosing a proper **distance metric** is a difficult problem in general, e.g. when $\mathcal{C}$ is a complex manifold due to rotative degrees of freedom. Requires task-specific choices, active area of research
- The **termination condition** may be the number of vertices to put into the roadmap or a maximal density of nodes. The roadmap should be dense enough to always be able to connect q_I and q_G

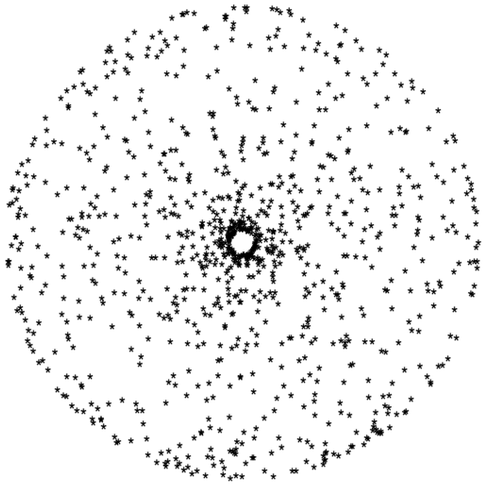
Probabilistic Roadmaps: Sampling

- PRMs have problems with **narrow passages**. Such cases require specialized sampling strategies
  - Examples include: **sampling near obstacles**, bridge sampling, GVD-inspired sampling, random-bounce walks, deterministic or quasi-random sampling



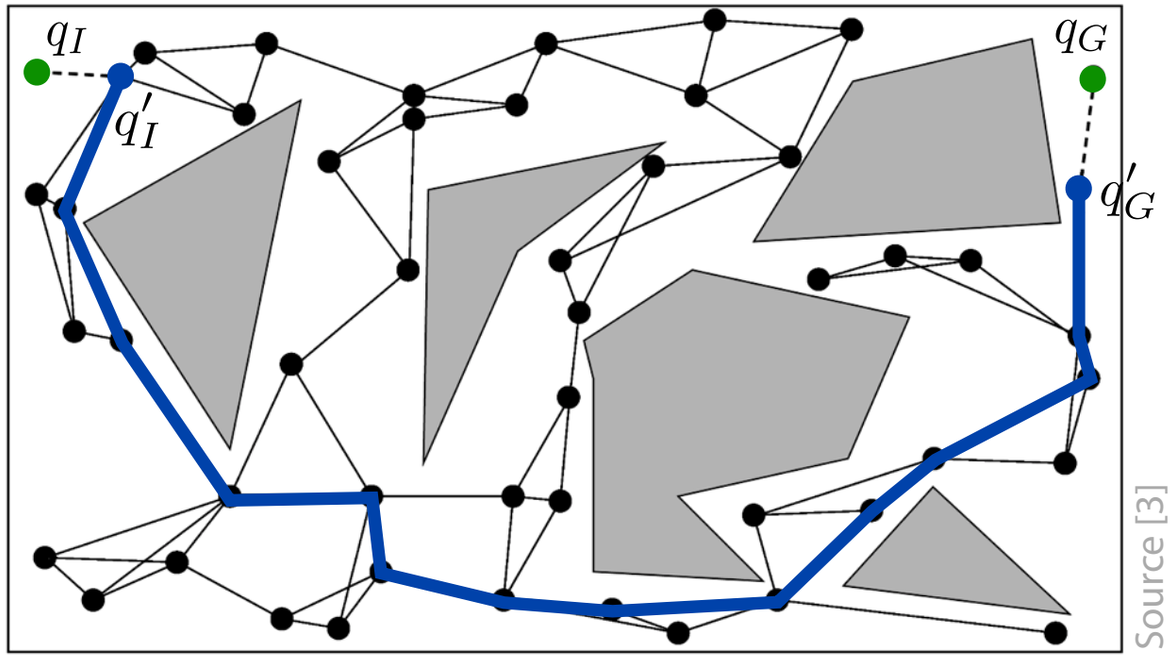
narrow passage

- Care has also to be taken when sampling non-Euclidian spaces such as **spaces of rotations**: sampling Euler angles uniformly gives more samples near poles, not uniform over $SO(3)$. Use alternative ways to represent rotations such as quaternions or Hopf coordinates


sampling on a torus

Probabilistic Roadmaps

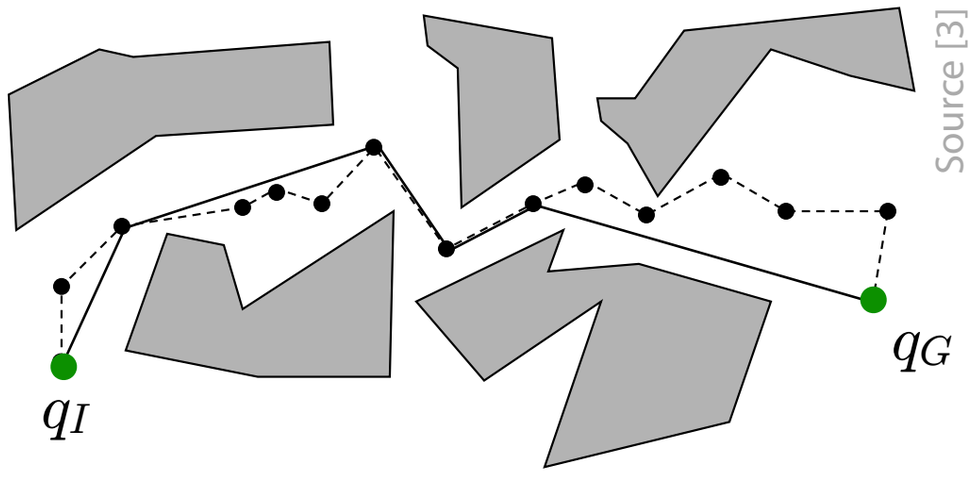
- After the roadmap has been **constructed** (also called "learned") there is the **query phase** (analogue to the application phase of a classifier)
- A **query** consists in passing q_I and q_G to a PRM planner which computes the path from q_I to q_G by connecting them to the roadmap and planning a path between the connection points q'_I and q'_G
- Once a roadmap has been created, it can be used to process **multiple queries** very **efficiently**
- Only when the workspace or robot changes (e.g. dynamic obstacles or loaded cargo), a roadmap needs to be rebuilt



Source [3]

Probabilistic Roadmaps

- PRM paths are examples of feasible but **non-optimal paths**: solutions are typically jagged and overly long
- A popular **path post-processing** technique is **smoothing**: try to connect non-adjacent nodes along the path (either sampled or chosen greedily) with the local planner



path post-processing

- An important theoretical result is that probabilistic roadmaps are **probabilistically complete**: the probability of finding a solution if one exists tends to one
- However, when there is no solution (path is blocked), the planner may run **forever**. This is of course weaker than combinatorial planners that are able to report failure after a finite time

Probabilistic Roadmaps

- Despite its lack of optimality and completeness, sampling-based methods, and in particular PRMs, are the **preferred choice** for most practical planning problems
- Combinatorial methods are rarely tractable for robots with more than three degrees of freedom. PRM planners were able to solve problems that were **previously unsolved**
- The goal of PRMs are to create a roadmap that captures the connectivity of $\mathcal{C}_{free}$ and to answer **multiple queries** very fast
- In many planning problems, however, we are only interested in **single queries**. Instead of focussing on the exploration of $\mathcal{C}$, **single-query planners** attempt to solve a planning problem as fast as possible
- An important single-query sampling-based technique: RRTs

Rapidly-Exploring Random Trees (RRT)

- **Idea:** aggressively probe and explore the C-space by expanding incrementally from an initial configuration q_0
- The explored territory is marked by a **tree** rooted at q_0

after 45 iterations

after 2'345 iterations

Source [2]

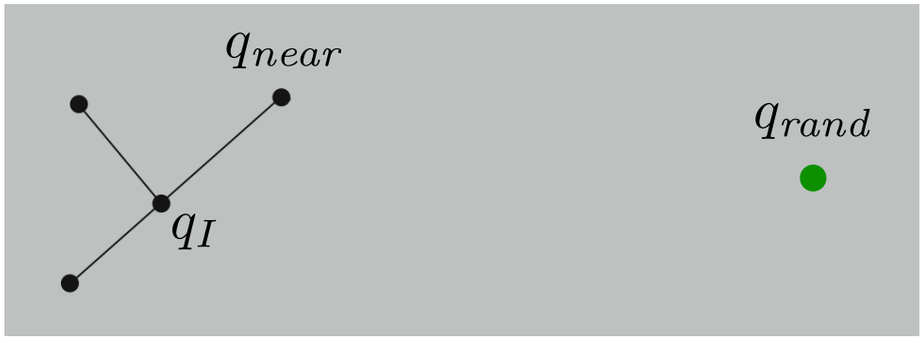
Rapidly-Exploring Random Trees

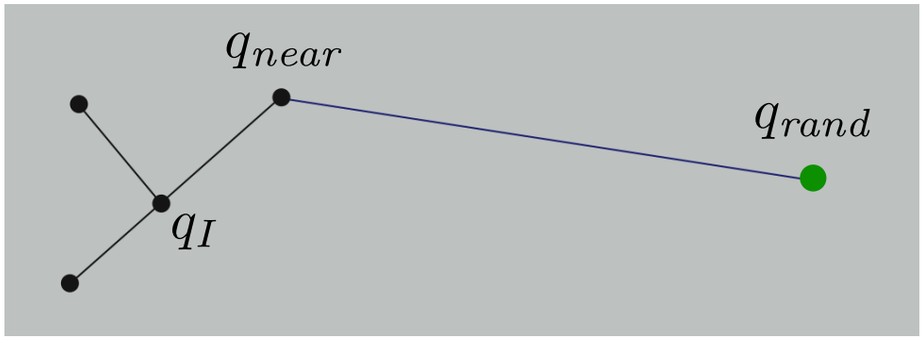
- Basic RRT algorithm

Algorithm 1: RRT

In: $\mathcal{C}$, q_I, q_G
Out: Tree $G = (V, E)$

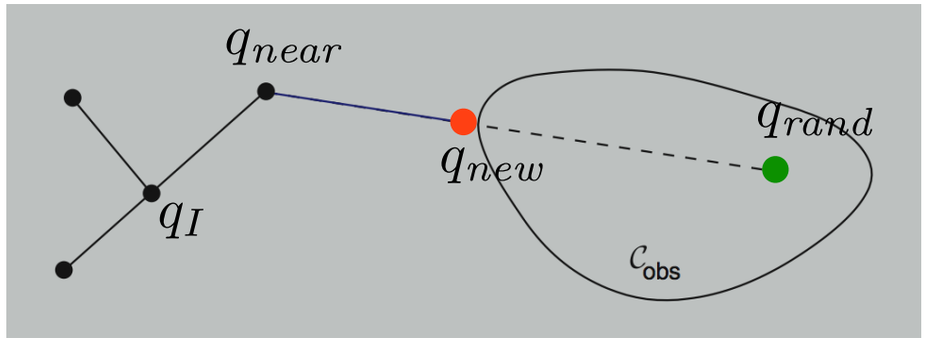
```
1  G.add_vertex(q_I)
2  repeat
3  |   q_rand → RANDOM_CONFIG(C)
4  |   if CLEAR(q_rand) then
5  |   |   q_near ← NEAREST(G, q_rand)
6  |   |   if LINK(q_rand, q_near) then
7  |   |   |   G.add_vertex(q_rand)
8  |   |   |   G.add_edge(q_near, q_rand)
9  |   |   end
10 |   end
11 until until q_G reached
```





Rapidly-Exploring Random Trees

- Unlike PRMs that connect newly sampled configurations to a set of nearest neighbors, RRT selects a **single neighbor**
- Again, what means "**nearest**" on a manifold? Requires the choice of a proper distance metric
- Instead of discarding q_{rand} when the **local planner** reports a **collision**, we can also add the configuration along the local path which is **closest to** $\mathcal{C}_{obs}$



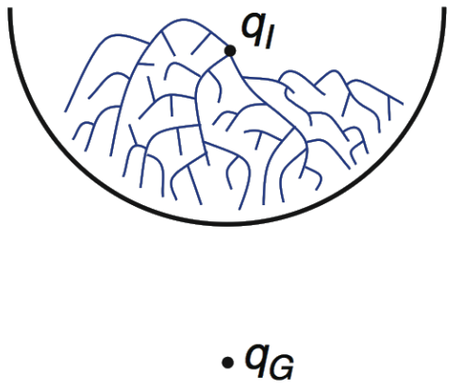
- q_{rand} may not be reachable from q_{near} also due to other reasons than collision with obstacles (such as kinodynamic constraints). Thus, it is most general to say that the tree can only be extended by a **configuration** q_{new} **close to** q_{rand} and the **corresponding edge** from q_{near} to q_{new}

Rapidly-Exploring Random Trees

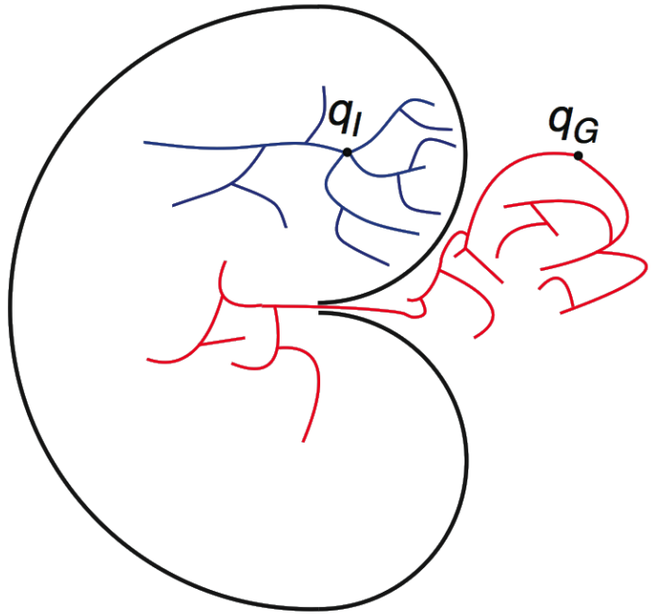
- So far, there is little consideration of the goal q_G for growing the tree. The basic algorithm usually **terminates** by checking if q_{rand} (or q_{new}) is **near the goal** (“has reached the goal region”)
- Inducing a gentle bias toward the goal can accelerate the method
  - Seed a tree at q_I
  - At every, say, 100th iteration, force $q_{rand} = q_G$
  - If q_G is reached, problem is solved
- Picking q_G every time would fail and waste much effort in bumping into $\mathcal{C}_{obs}$ instead of exploring the space
- However, some problems require more effective methods. One way is to perform **bidirectional search**

Rapidly-Exploring Random Trees

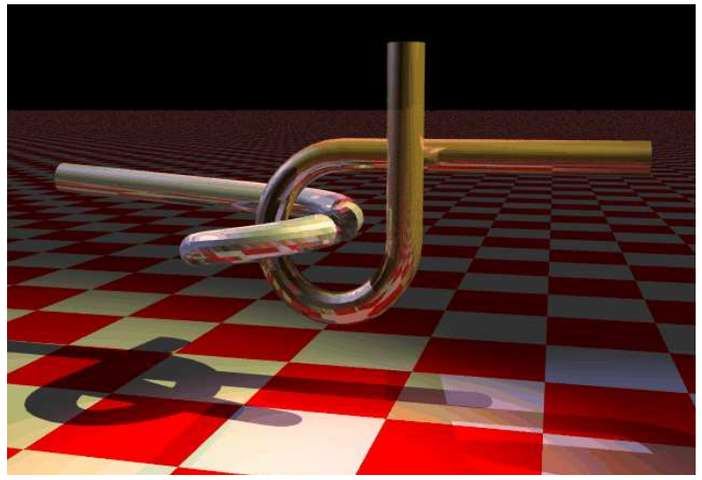
- **Bidirectional RRT** grow two trees, one from q_I, one from q_G
- In every other step, the method tries to extend each tree towards the newest vertex of the other tree
- Examples



local minima (filling a well)



one-way door (bug trap)

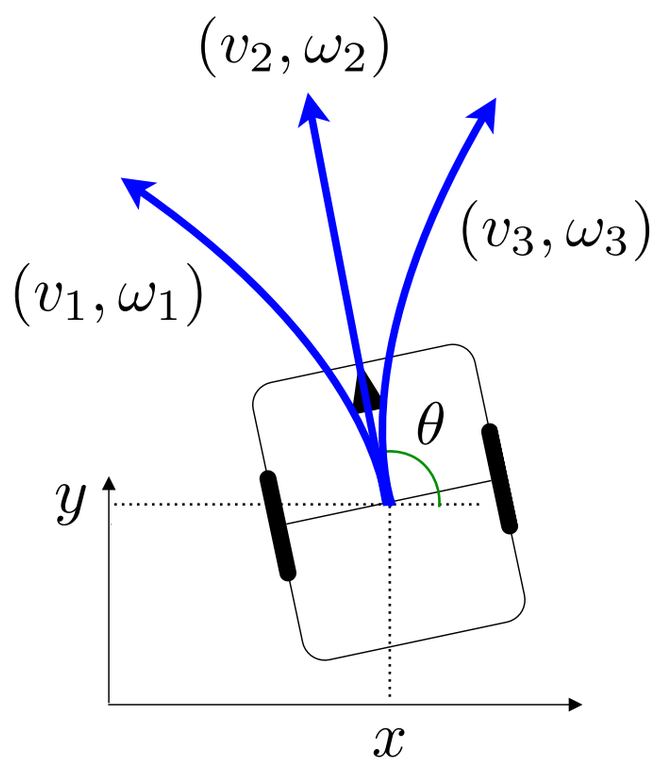


Source [2]

The Alpha 1.0 3D puzzle can be solved using the bidirectional RRT in a few minutes

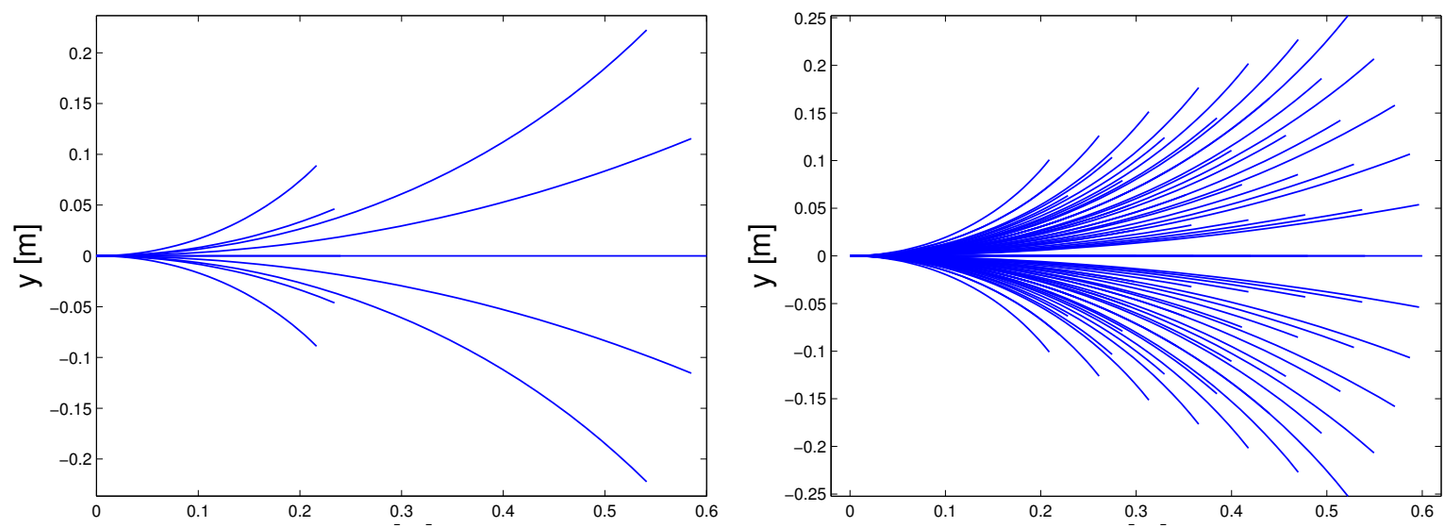
Rapidly-Exploring Random Trees

- RRTs have initially been developed for **kinodynamic motion planning**, that is, motion planning under kinematic and dynamic constraints
- The basic RRT algorithm can be easily extended for planning under such constraints
- Let us consider a wheeled differential-drive robot with controls $u = (v, \omega)$ where v is the translational and ω the angular robot velocity
- Example of non-holonomic constraints: the robot can go anywhere but **not** by following any trajectory
- Thus, the original **local planner**, which connects new vertices to the tree over linear edges, **cannot** be used anymore



Rapidly-Exploring Random Trees

- Let us **precompute some controls** and use them as a sort of "prefabricated edges" to connect new vertices to the tree
- Such sets of precomputed controls are called **motion primitives**
- Like linear edges, they consist of **many nearby points** that allow for incremental or binary collision checking



Example sets with 10 (left) and 77 (right) motion primitives

- When extending the tree to a new configuration q_{rand}, we then **select** the motion primitive that **comes closest** to q_{rand}. Its **terminal point** becomes q_{new}

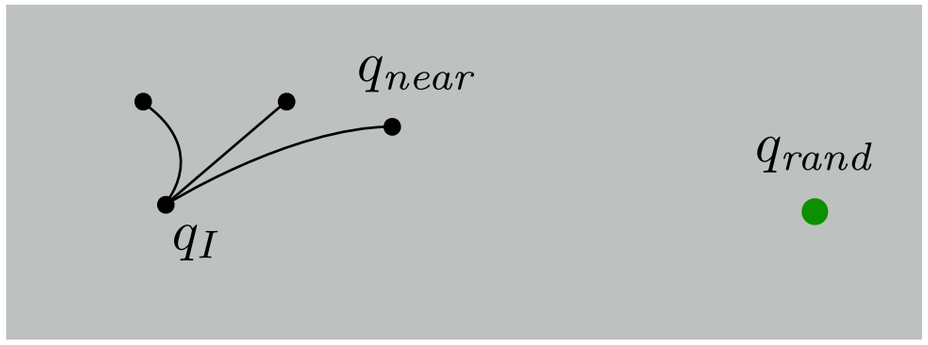
Rapidly-Exploring Random Trees

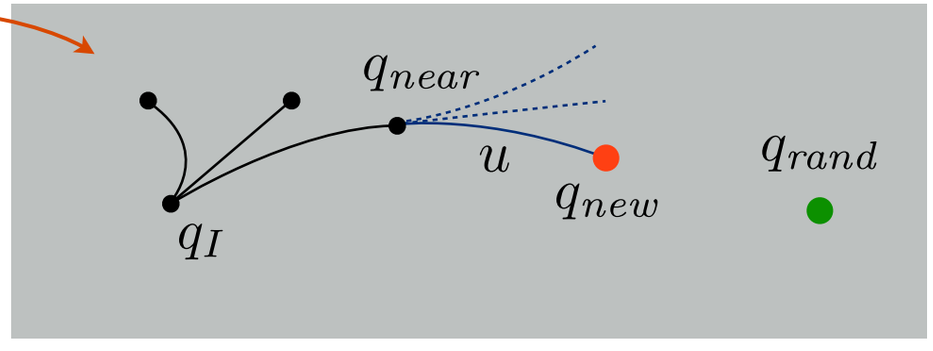
- Basic RRT under kinodynamic constraints

Algorithm 1: RRT under constraints

In: $\mathcal{C}$, q_I, q_G
Out: Tree $G = (V, E)$

```
1  G.add_vertex(q_I)
2  repeat
3      q_rand → RANDOM_CONFIG(C)
4      if CLEAR(q_rand) then
5          q_near ← NEAREST(G, q_rand)
6          q_new, u, collision ← EXTEND(q_near, q_rand)
7          if ¬collision then
8              G.add_vertex(q_new)
9              G.add_edge(q_near, q_new, u)
10         end
11     end
12 until until q_G reached
```



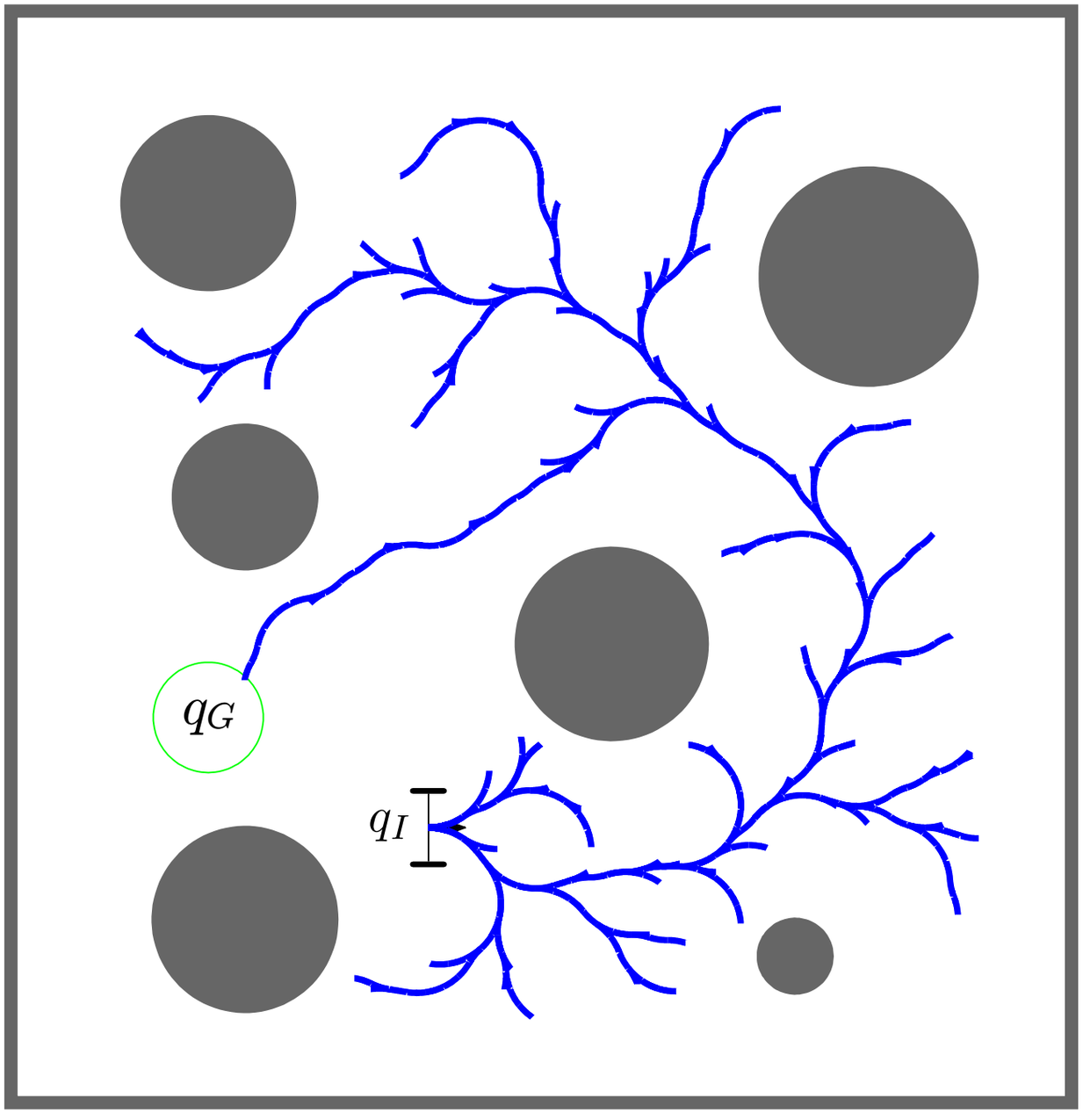


Rapidly-Exploring Random Trees

- Example using three motion primitives

$$u_1 = (1, 0)$$
$$u_2 = (1, 3)$$
$$u_3 = (1, -3)$$

- The **number and shape** of motion primitives are parameters with a strong effect on the resulting tree and the performance of the algorithm

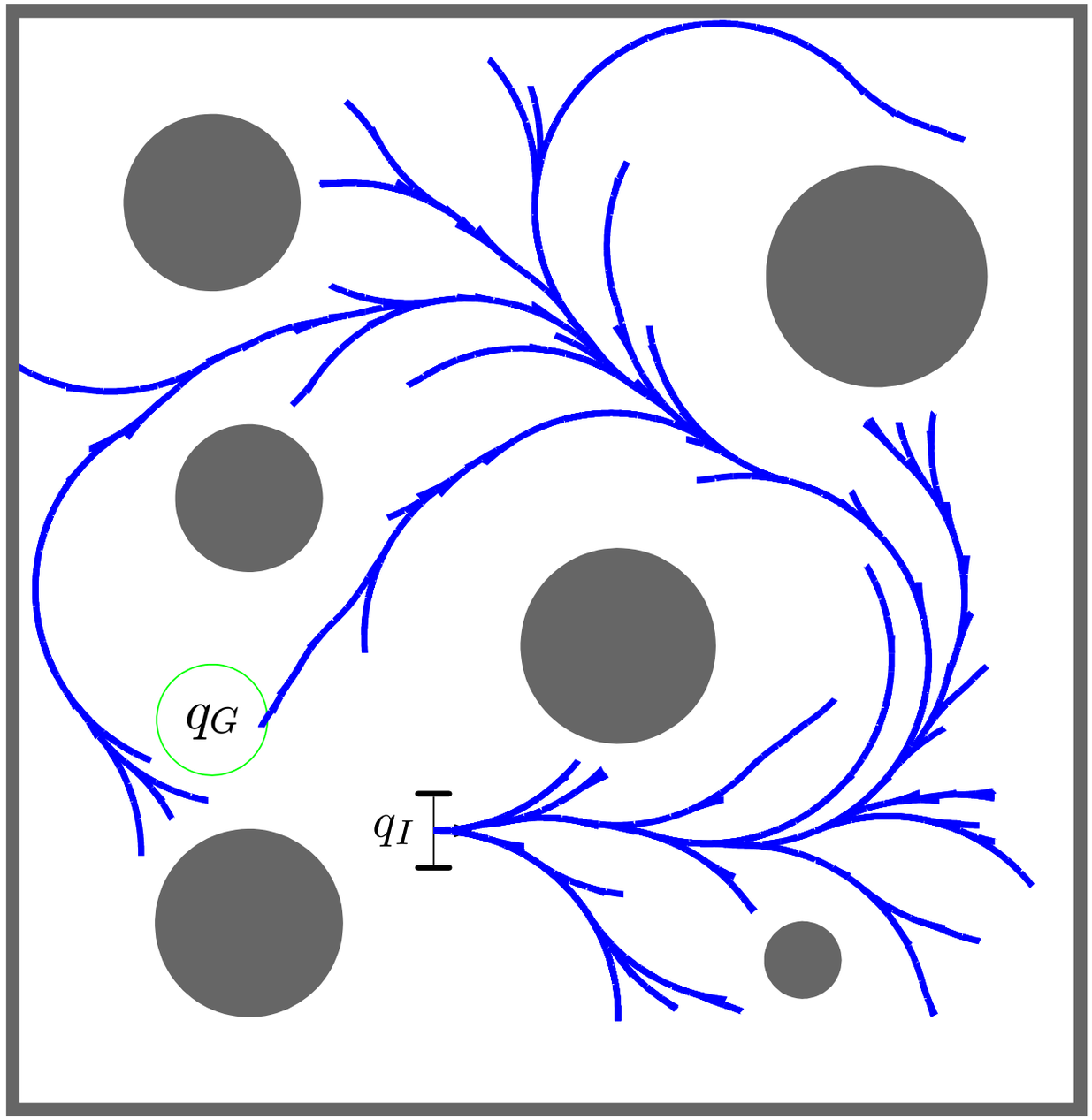


Rapidly-Exploring Random Trees

- Example using three motion primitives

$$u_1 = (1, 0)$$
$$u_2 = (1, 1)$$
$$u_3 = (1, -1)$$

- The **number and shape** of motion primitives are parameters with a strong effect on the resulting tree and the performance of the algorithm

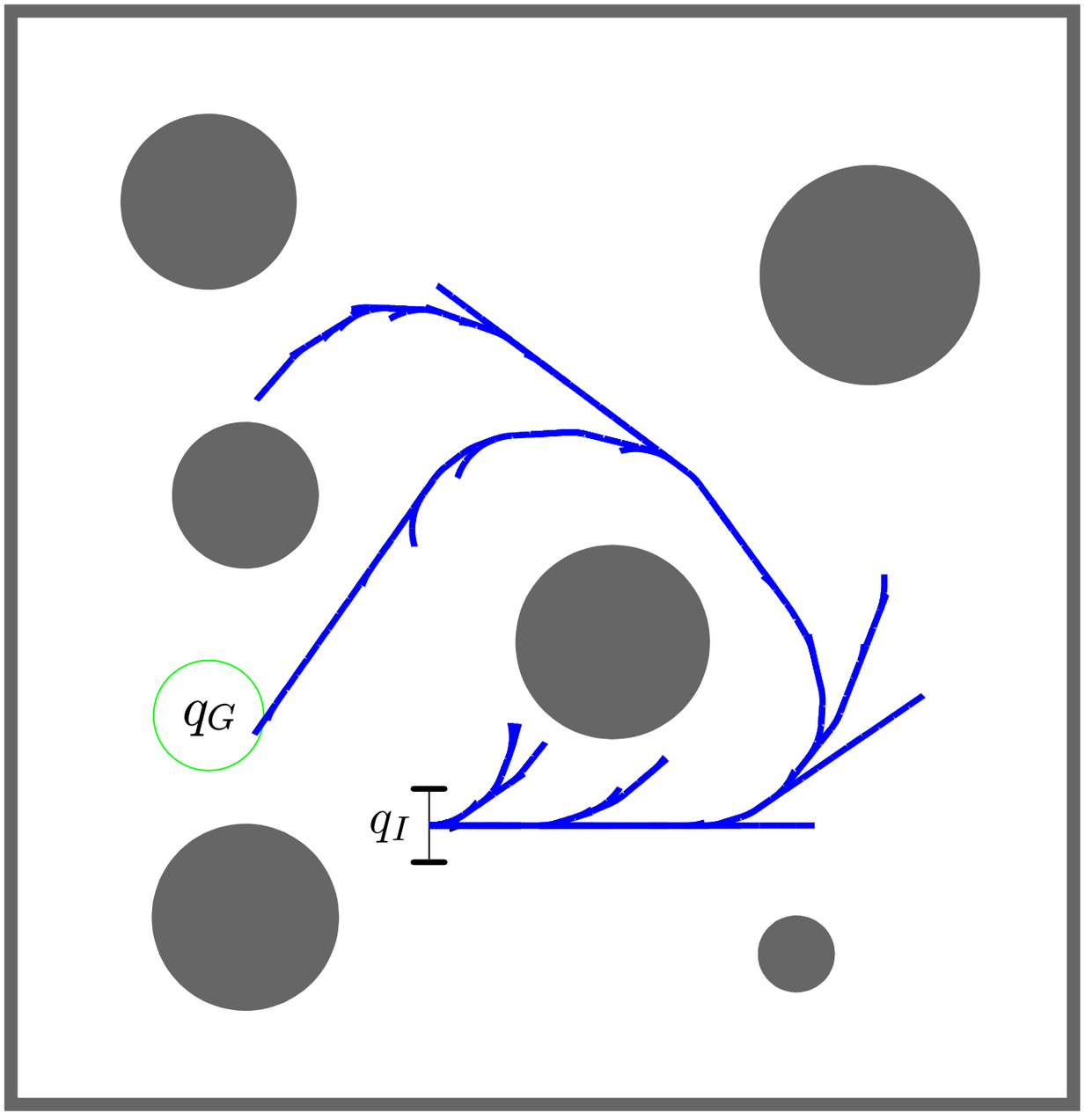


Rapidly-Exploring Random Trees

- Example using three **fun** motion primitives

$$u_1 = (1, 0)$$
$$u_2 = (1, 1)$$
$$u_3 = (1, 3)$$

- Robot can go only **straight** or make **left turns**



Sampling-Based Planning

- Let us **wrap up**:
- Sampling-based planners are **more efficient** but have **weaker guarantees**
- They are **probabilistically complete**: the probability tends to **one** that a solution is found if one exists. Otherwise they may still run forever
- It is easy to construct examples that cause sampling-based algorithm to fail or converge slowly. In some cases (e.g. narrow passages), **problem-specific heuristics** can be developed
- Problems with high-dimensional and complex C-spaces are still also hard for sampling-based methods
- However, they have solved **previously unsolved** problems and have become the **preferred choice** for many practical problems

Potential Field Methods

- All techniques discussed so far aim at capturing the connectivity of $\mathcal{C}_{free}$ into a roadmap
- **Potential field methods** follow a different idea: the robot, represented as a point in $\mathcal{C}$, is modeled as a **particle** under the influence of a **artificial potential field** U
- A potential field (or potential function) U is a differentiable real-valued function

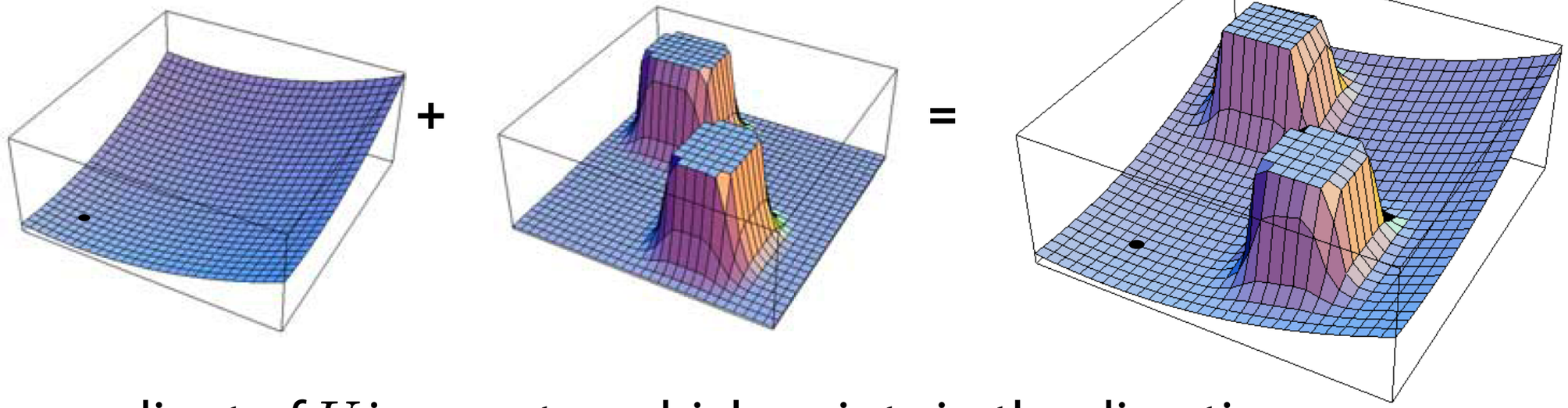
$$U : \mathbb{R}^m \rightarrow \mathbb{R}$$

- The value of U can be seen as an energy or **superposition of forces**
  - **Repulsive forces** from **obstacles**
  - **Attractive force** from the **goal**

Potential Field Methods

- Potential function

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



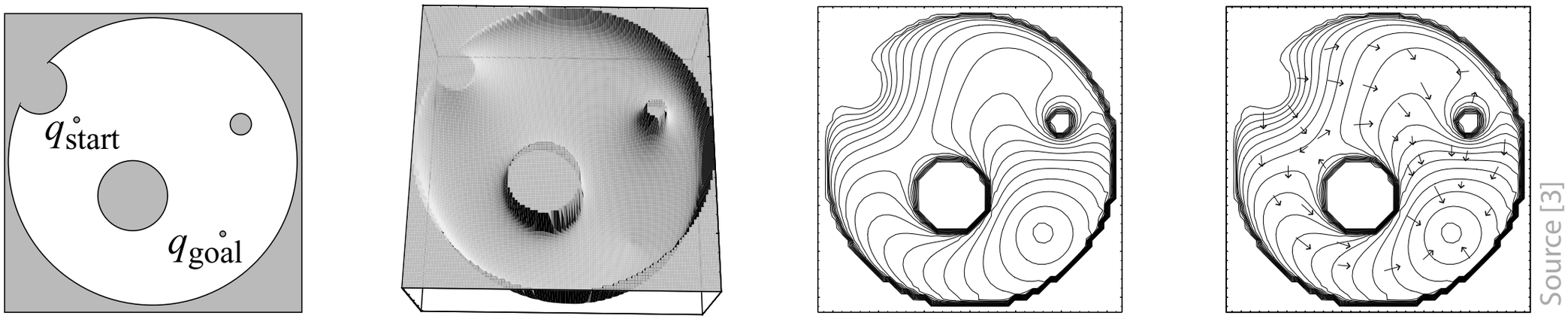
- The gradient of U is a vector which points in the direction that locally maximally increases U

$$F(q) = -\nabla U(q)$$

- The (negative) gradient **directs the robot to the goal**, avoiding obstacles

Potential Field Methods

- Example with three circular obstacles

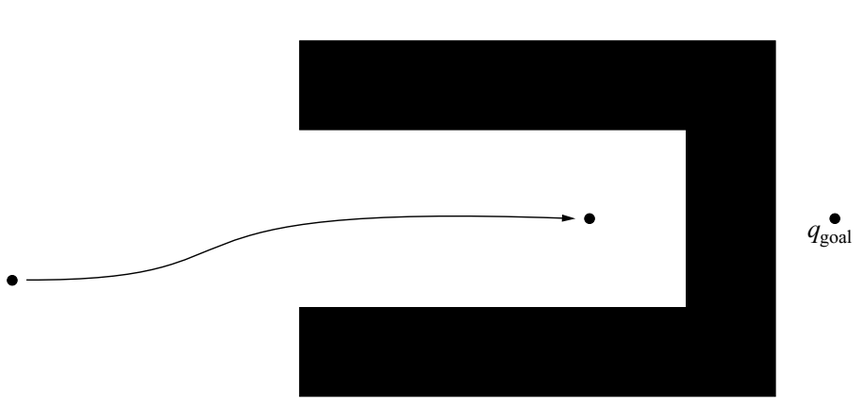


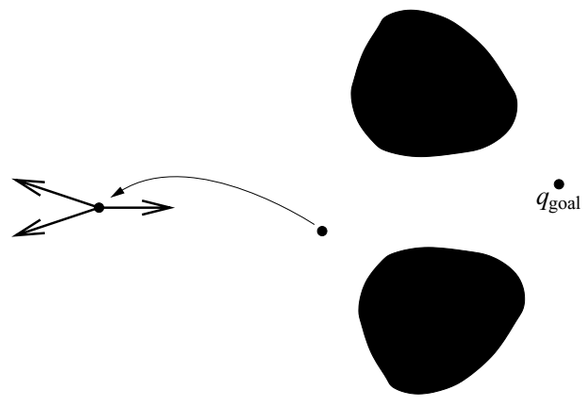
Source [3]

- The gradient defines a **vector field** that can be used as feedback control strategy, relevant for an uncertain robot
- Gradient descent, a well-know optimization problem, has particularly **simple implementations** when $\mathcal{C}$ is discretized as a **grid**

Potential Field Methods

- One of the major problems of potential field methods are **local minima**





Source [3]

- A solution are so called **navigation functions**, **local-minima-free** potential function (e.g. NF1). Then, gradient descent works
- Do not work in general configuration spaces, only in a limited class
- However, potential fields need to represent $\mathcal{C}_{free}$ **explicitely**. This is, as we have learned, **too costly** in many practical motion planning problems

Sources and Further Reading

These slides partly follow the books of Latombe [1], LaValle [2], and Choset et al. [3]. Chapters 5 and 6 of [4] are well-written compact introductions to the field. Some GVD-related pictures have been taken from the lecture notes by Johnson [9]. You are also encouraged to try the nice and instructive online applets [6] and [7]. Readers interested in AI/robotics history, refer to the amazing work by Nilson and colleagues [8].

[1] J.C. Latombe, Robot Motion Planning, Kluwer Academic Publishers, 1991

[2] S. LaValle, Planning Algorithms, Cambridge University Press, 2006. See http://planning.cs.uiuc.edu

[3] H. Choset, K.M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L.E. Kavraki, S. Thrun, Principles of Robot Motion: Theory, Algorithms, and Implementations, MIT Press, 2005. See http://biorobotics.ri.cmu.edu/book

[4] B. Siciliano, O. Khatib (editors), Handbook of Robotics, Chapters 5 and 6, Springer, 2008. See http://www.springer.com/engineering/robotics/book/978-3-540-23957-4

[5] F. Lamiraux, J. P. Laumond, C. VanGeem, D. Boutonnet, and G. Raust, "Trailer-truck trajectory optimization for Airbus A380 component transportation," IEEE Robotics and Automation Magazine, 2003

Sources and Further Reading

[6] Planar Robot Simulator with Obstacle Avoidance Applet, by K. Goldberg, E. Lee, J. Wiegley, http://ford.ieor.berkeley.edu/cspace

[7] P. Blear, Path Planning Applet, http://www.cs.columbia.edu/~pblaer/projects/path_planner/applet.shtml

[8] N.J. Nilson, "A mobile automaton: an application of artificial intelligence techniques", 1st Int. Joint Conference on Artificial Intelligence (IJCAI), 1969. See http://www.ai.sri.com/pubs/files/tn040-nilsson69.pdf

[9] D.E. Johnson, "CS 6370: Motion Planning", lecture notes, University of Utah, 2011