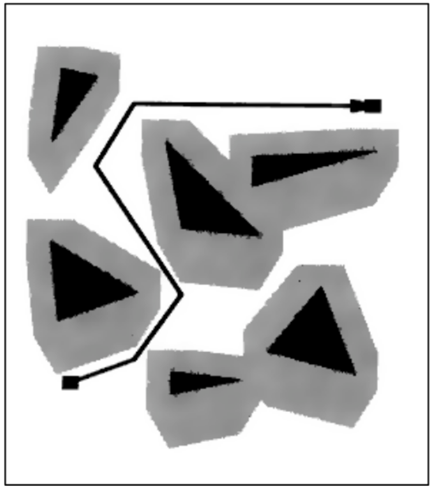
Classical Path Planning

Robert Platt
Northeastern University



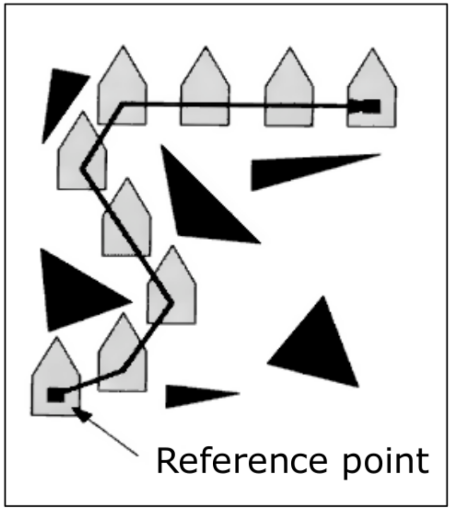
Slides contain significant material from Uni Freiburg course
Original slide author: Kai Arras

Problem we want to solve

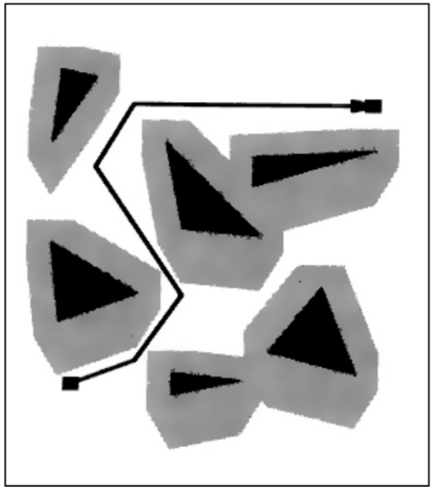
<u>Given:</u>
– a point-robot (robot is a point in space)
– description of obstacle space and free space
– a start configuration and goal region

<u>Find:</u>
– a collision-free path from start to goal

workspace



configuration space

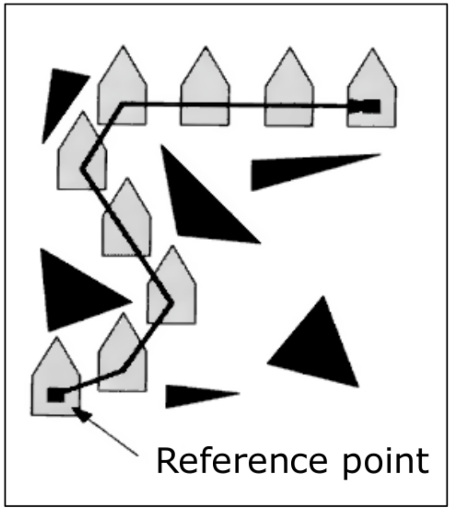


Problem we want to solve

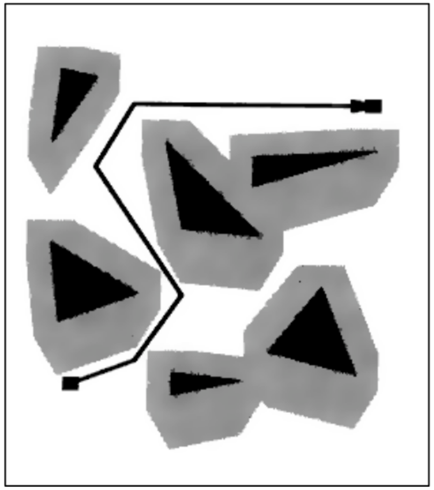
<u>Given:</u>
– configuration space $\mathcal{C}$
– free space $\mathcal{C}_{free}$
– start state $x_{init} \in \mathcal{C}_{free}$
– goal region $X_{goal} \subset \mathcal{C}_{free}$

<u>Find:</u>
– a collision-free path σ, such that $\sigma(0) = x_{init}$ and $\sigma(1) \in X_{goal}$

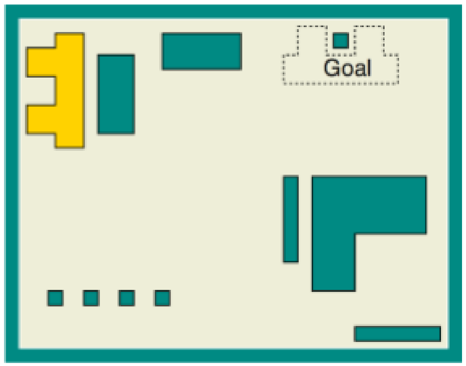
workspace

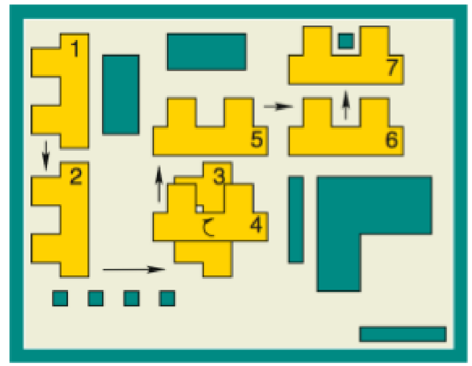


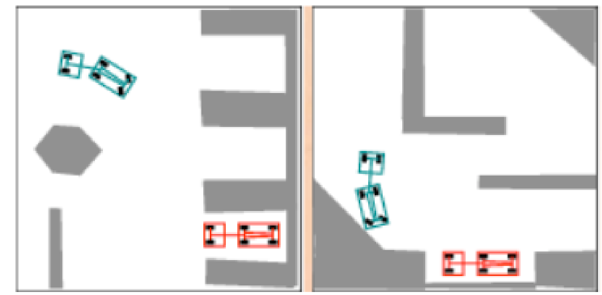
configuration space

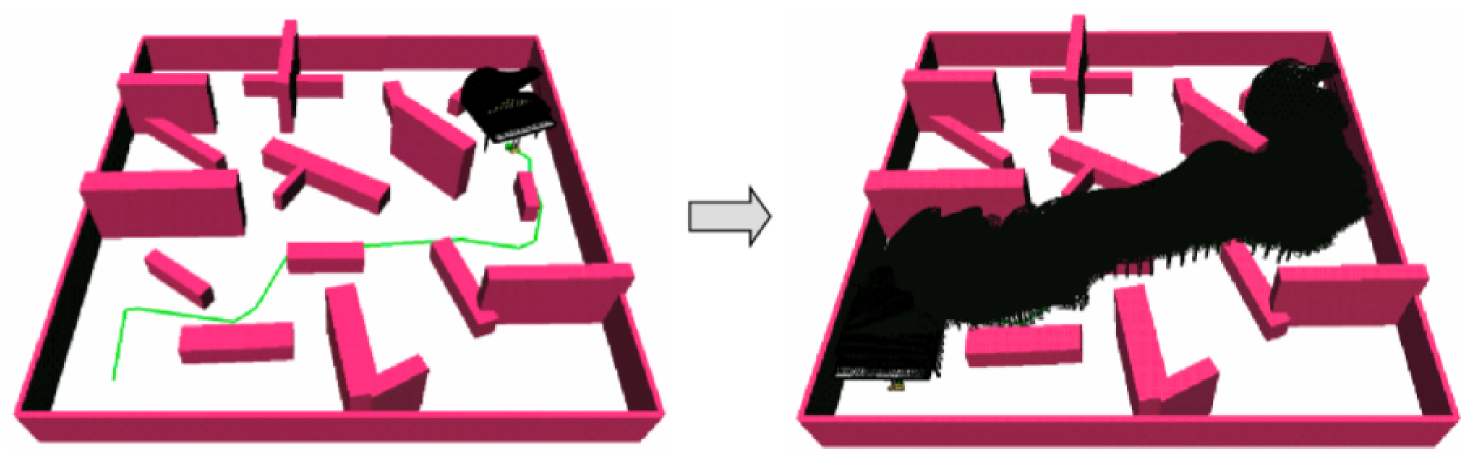


Problem we want to solve





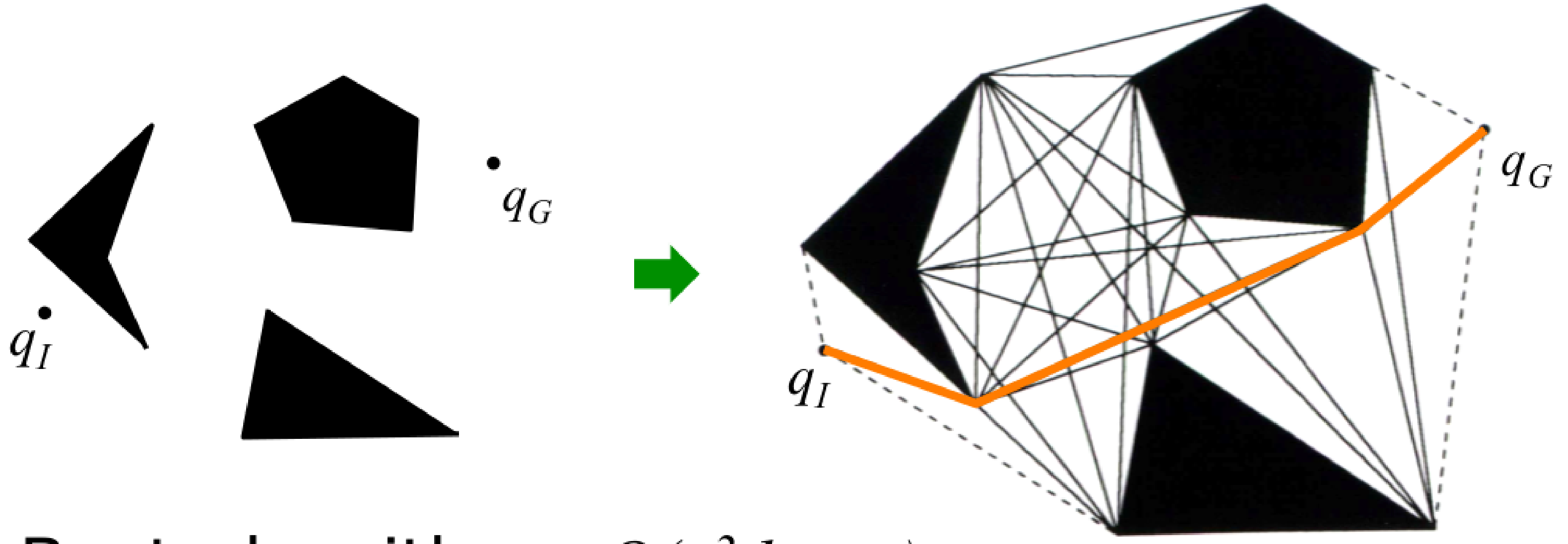




Motion planning is sometimes also called **piano mover's problem**

Method #1: Visibility Graphs

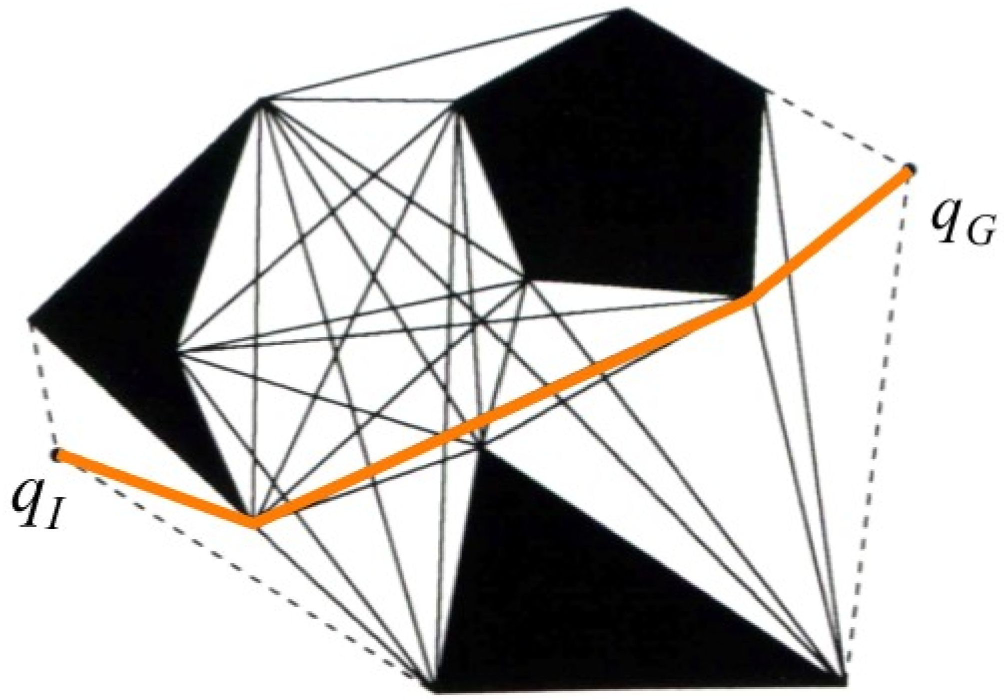
- **Idea:** construct a path as a polygonal line connecting q_I and q_G through vertices of C_{obs}
- Existence proof for such paths, **optimality**
- One of the earliest path planning methods



- Best algorithm: $O(n^2 \log n)$

n = num of obstacle vertices

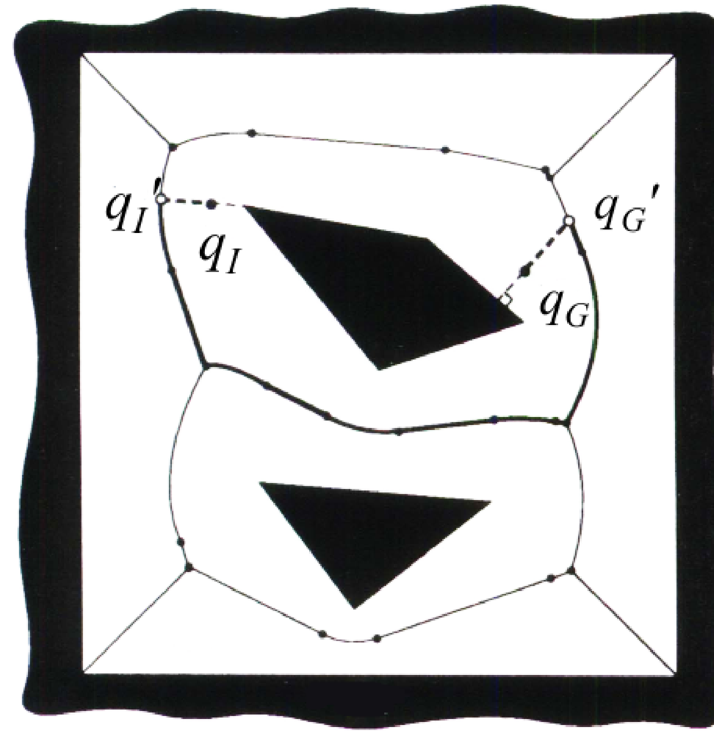
Question



Can you think of an n^3 algorithm to compute the visibility graph?

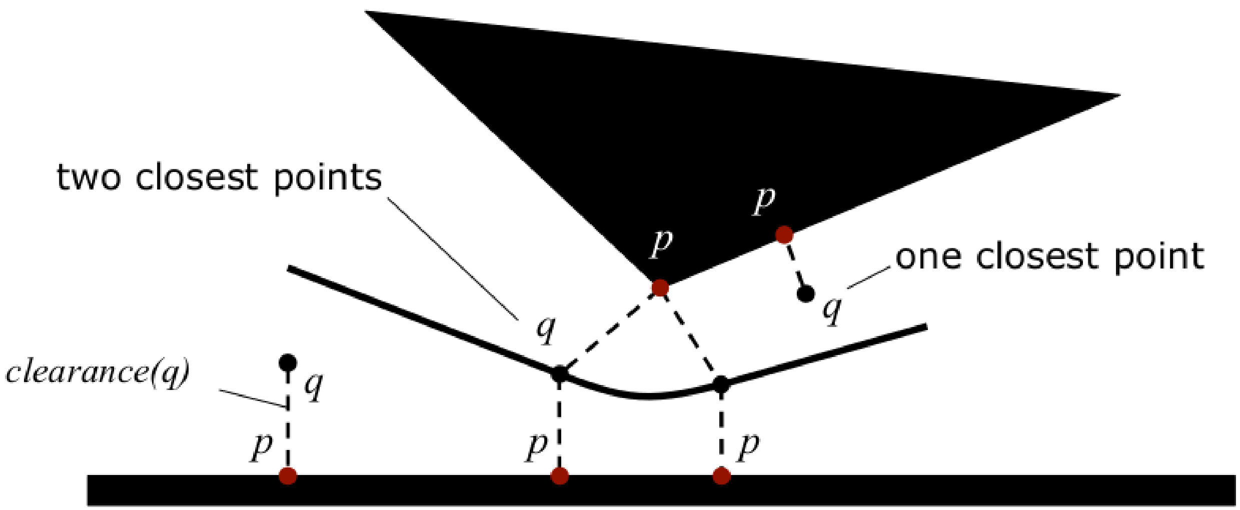
Method #2: Generalized Voronoi Diagram

- **Defined** to be the set of points q whose cardinality of the set of boundary points of C_{obs} with the same distance to q is greater than 1
- Let us decipher this definition...
- **Informally:** the place with the same **maximal clearance** from all nearest obstacles



Method #2: Generalized Voronoi Diagram

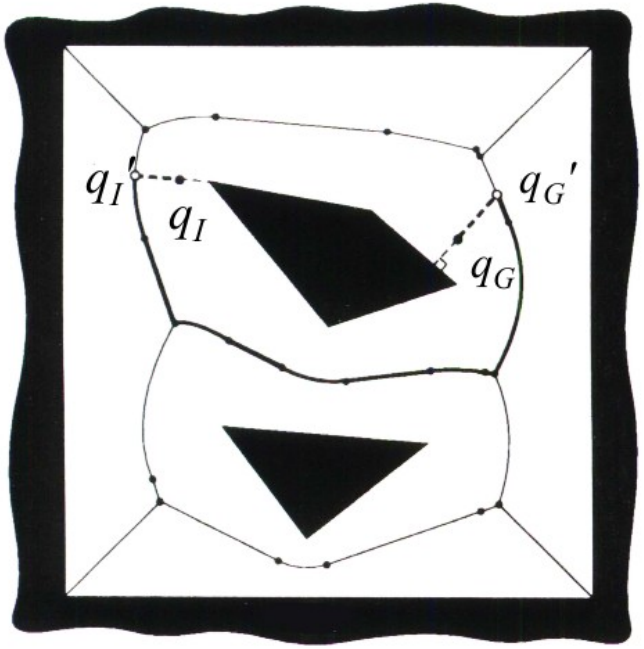
- **Geometrically:**



- For a polygonal C_{obs}, the Voronoi diagram consists of (n) lines and parabolic segments
- Naive algorithm: $O(n^4)$, best: $O(n \log n)$

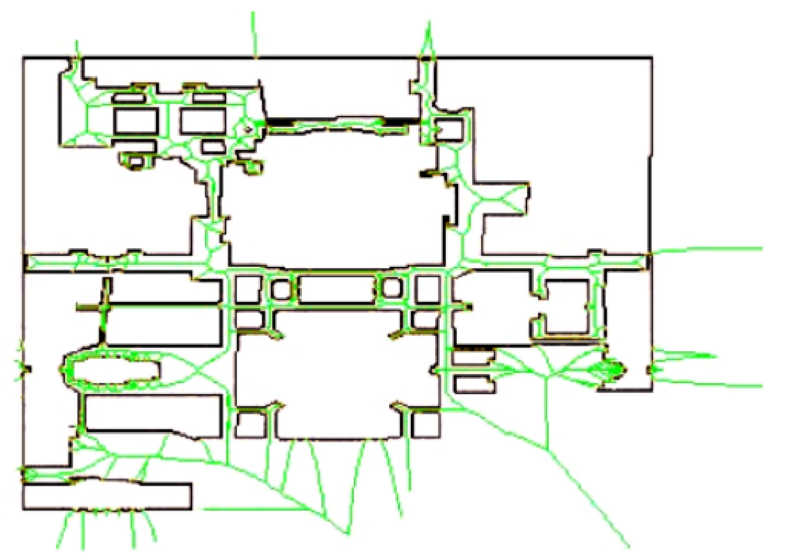
Question

How many regions in a voronoi diagram with n objects?



Method #2: Generalized Voronoi Diagram

- Voronoi diagrams have been well studied for (reactive) **mobile robot** path planning
- Fast methods exist to compute and update the diagram in real-time for low-dim. C's
  - **Pros:** maximize clear-ance is a good idea for an uncertain robot
  - **Cons:** unnatural at-traction to open space, suboptimal paths



- Needs extensions

Method #3: Exact Cell Decomposition

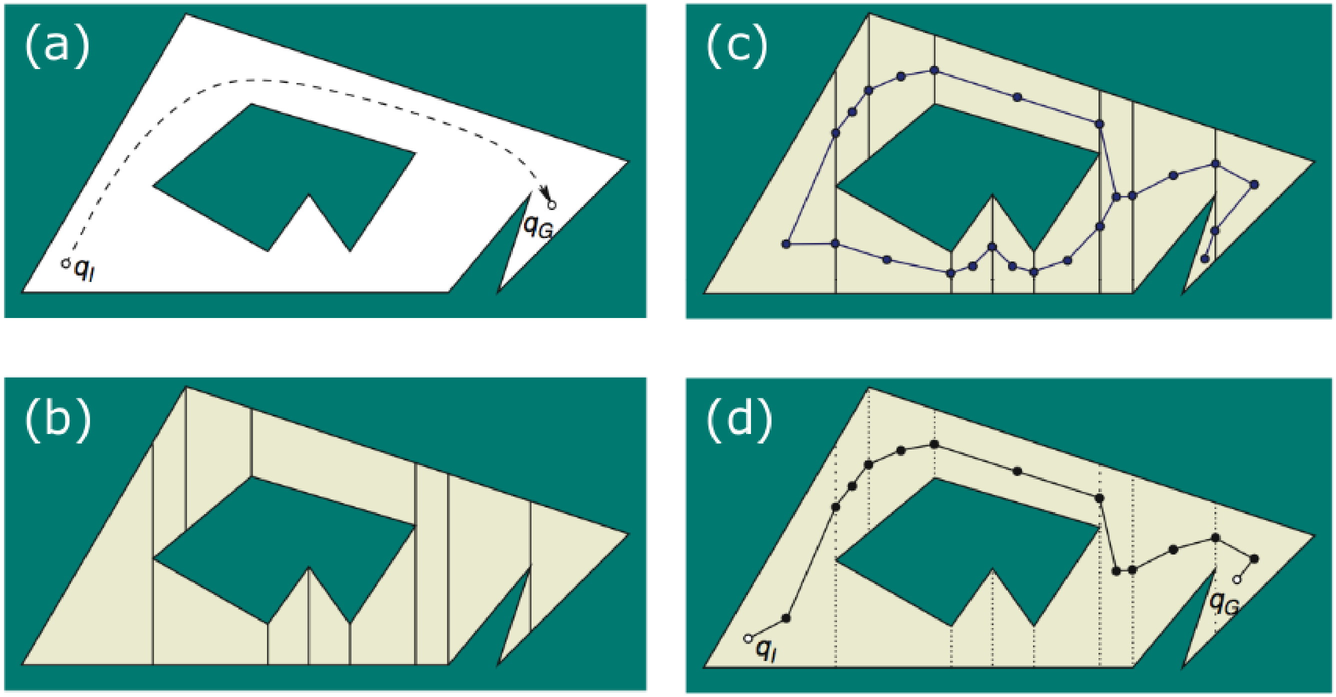
- **Idea:** decompose C_{free} into non-overlapping cells, construct connectivity graph to represent adjacencies, then search

- A popular implementation of this idea:

  1. Decompose 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** in every vertical **segment**
  4. Connect the vertices

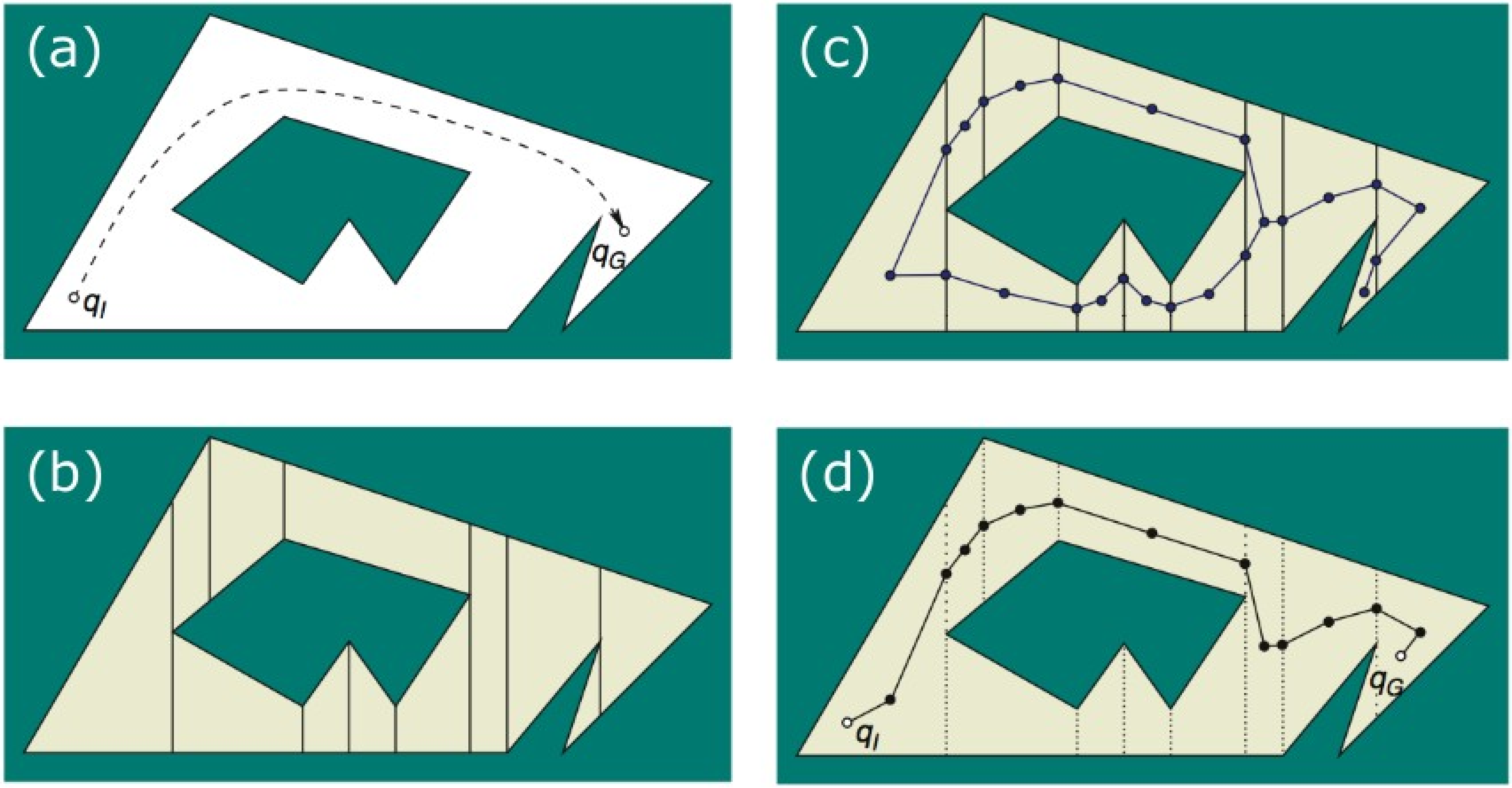
Method #3: Exact Cell Decomposition

- Trapezoidal decomposition ($\mathcal{C} = \mathbb{R}^3$ max)



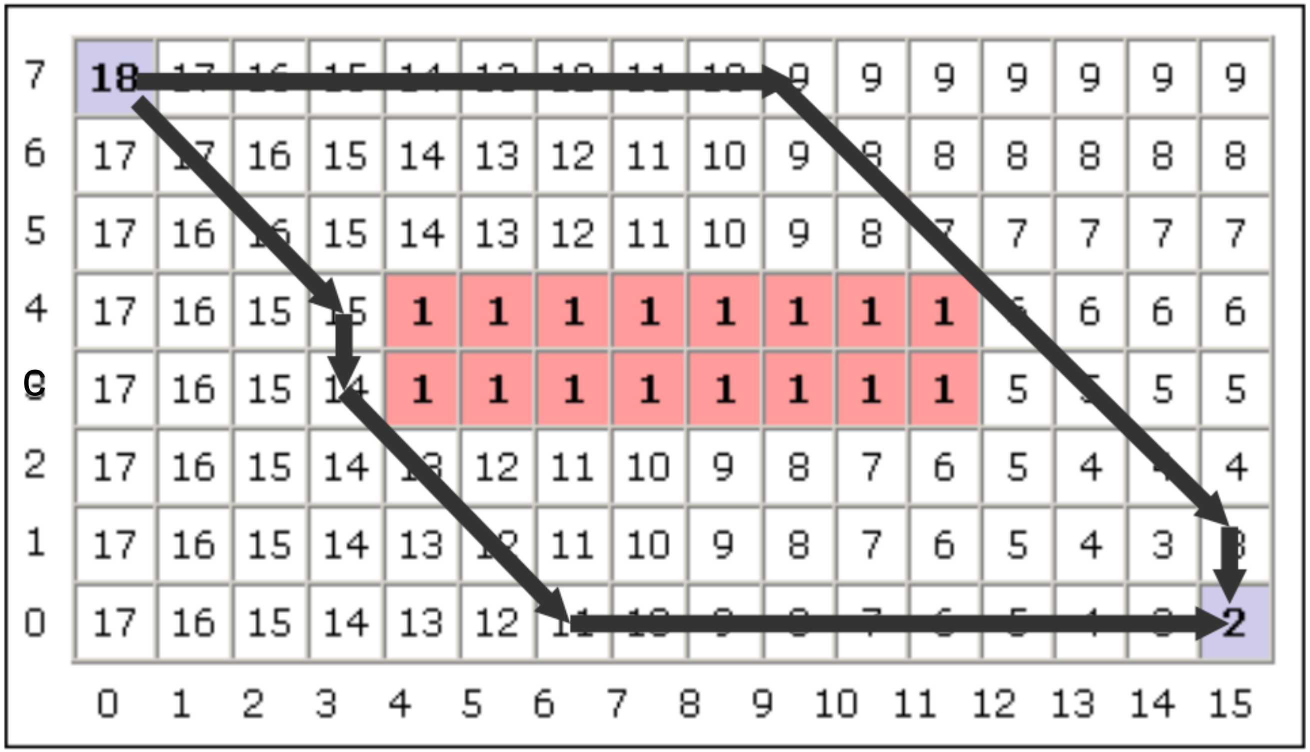
- Best known algorithm: $O(n \log n)$ where n is the number of vertices of C_{obs}

Question



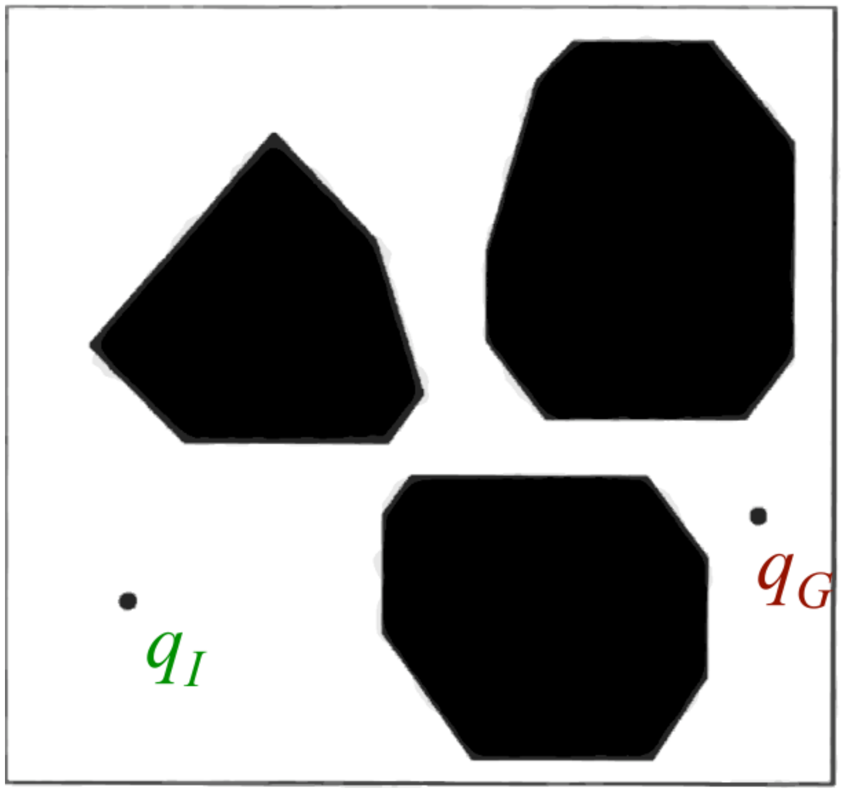
Do you need the vertices at the center of the trapezoids? Why/Why not?

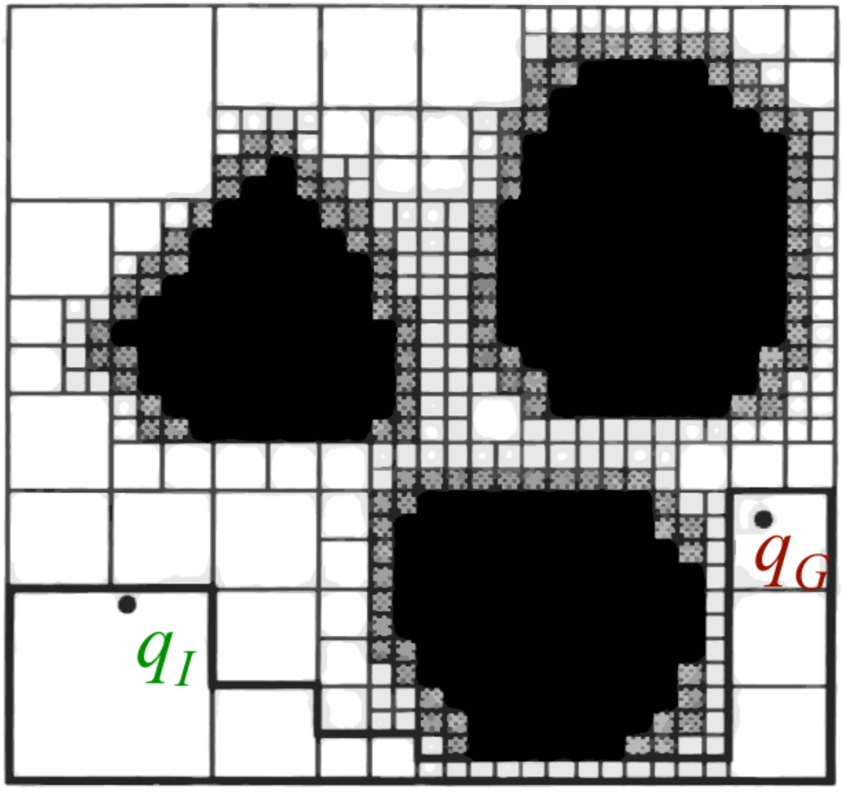
Method #4: Uniform Approximate Cell Decomposition



Uniform cell shape: e.g. wavefront planner

Method #5: Quadtrees





Non-Uniform cell shape: e.g. quadtree decomposition

Method #5: Quadtrees

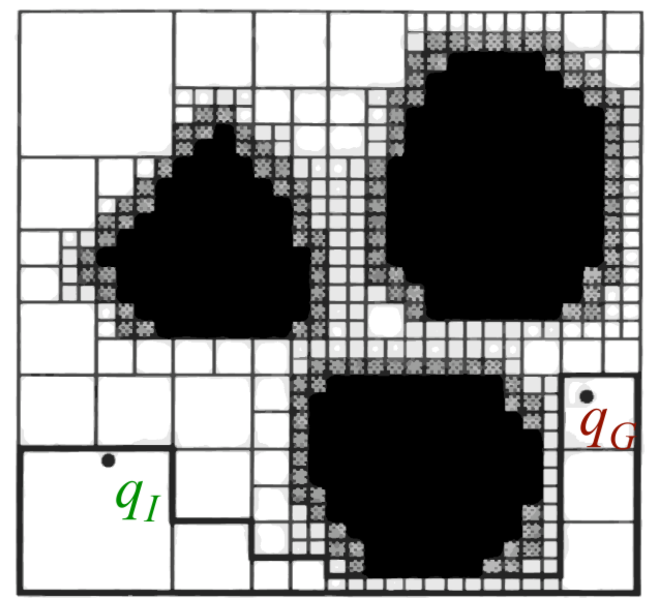
define G = Decompose(G,resolution):

1. if G null:
2. create coarse grid
3. collision-check G
4. for all occupied cells c in G:
5. delete c from G
6. subdivide c into four cells (sub)
7. add sub into G
8. collision-check sub

<u>Collision-check</u>: check whether each cell is completely free or not

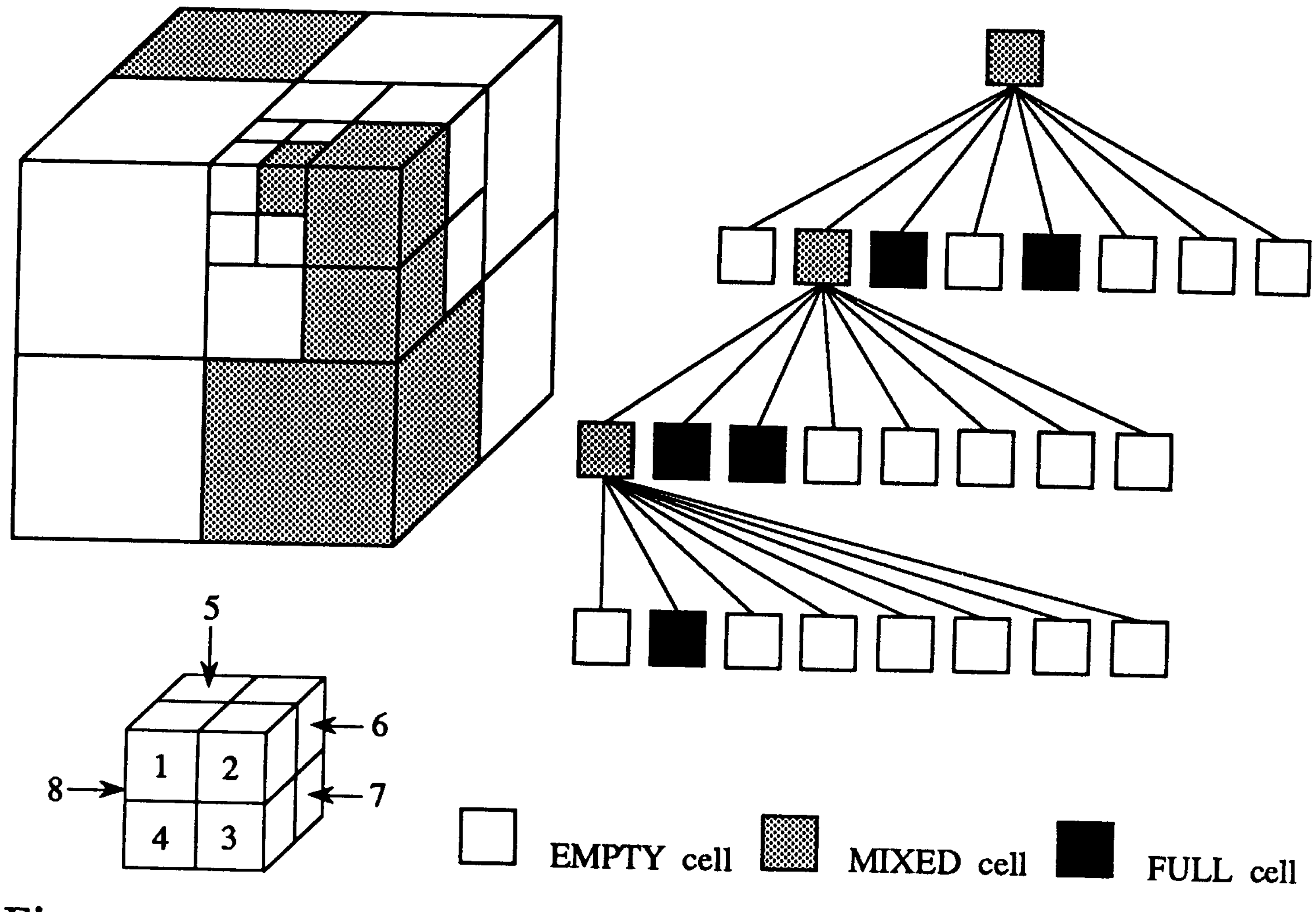
define FindPath(maxresolution):

1. for resolution = coarse to maxresolution:
2. G = Decompose(G,resolution)
3. if Check-for-path(G) == True:
4. Success!



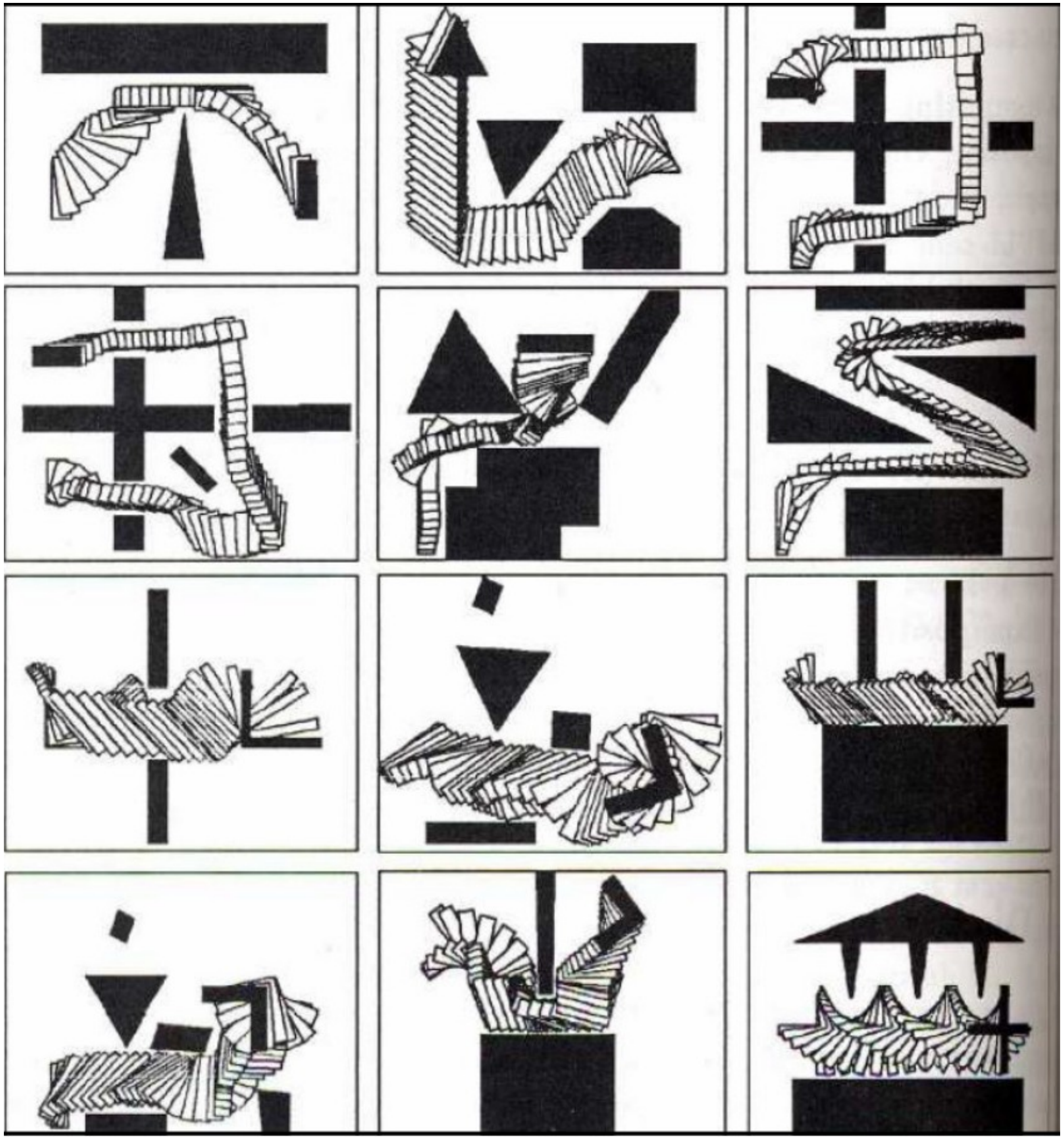
Why do you think this method is called "quadtree"?

Method #5: Octomaps



Same as quadtrees, but in three dimensions...

Examples of solutions found using octomaps



Exact vs approximate cell decomposition

- Exact decomposition methods can be involved and inefficient for complex problems
- Approximate decomposition uses cells with the **same simple predefined shape**
- **Pros:**
  - Iterating the **same** simple computations
  - Numerically more **stable**
  - **Simpler** to implement
  - Can be made **complete**

Method #6: Potential Functions

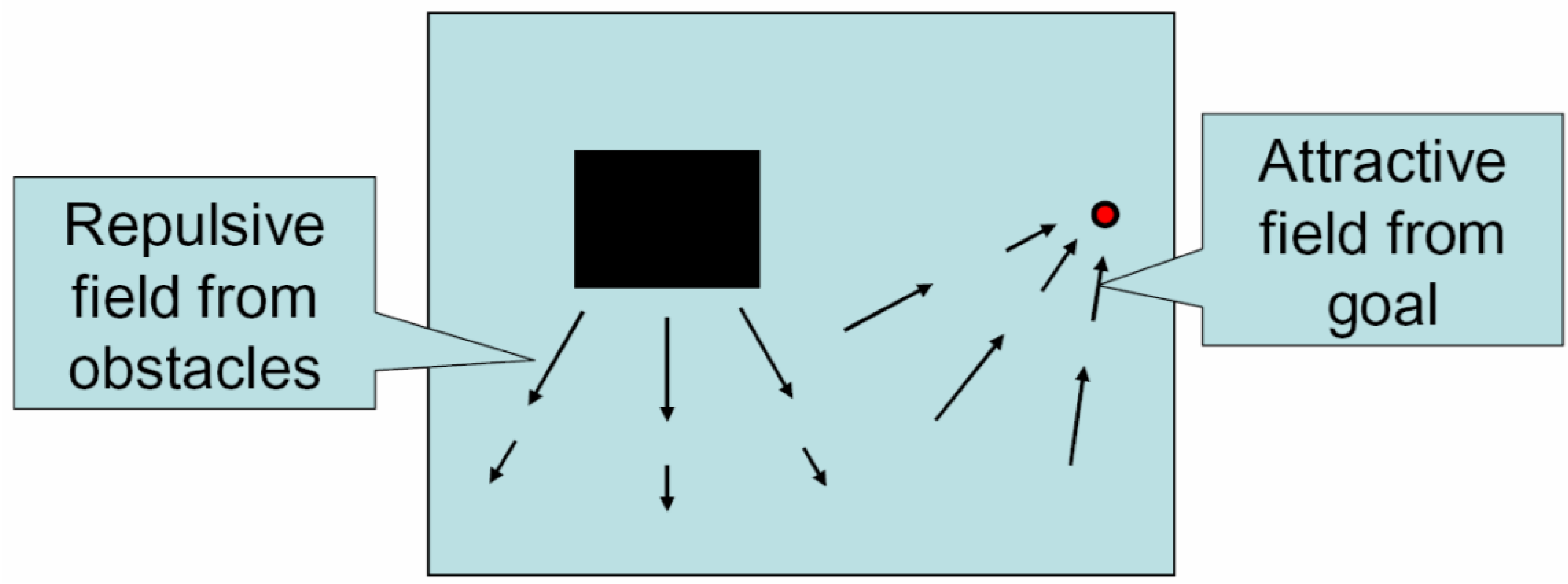
- All techniques discussed so far aim at cap-turing the connectivity of C_{free} into a graph
- **Potential Field methods** follow a different idea:

  The robot, represented as a point in C, is modeled as a **particle** under the influence of a **artificial potential field** $\mathbf{U}$

  $\mathbf{U}$ superimposes

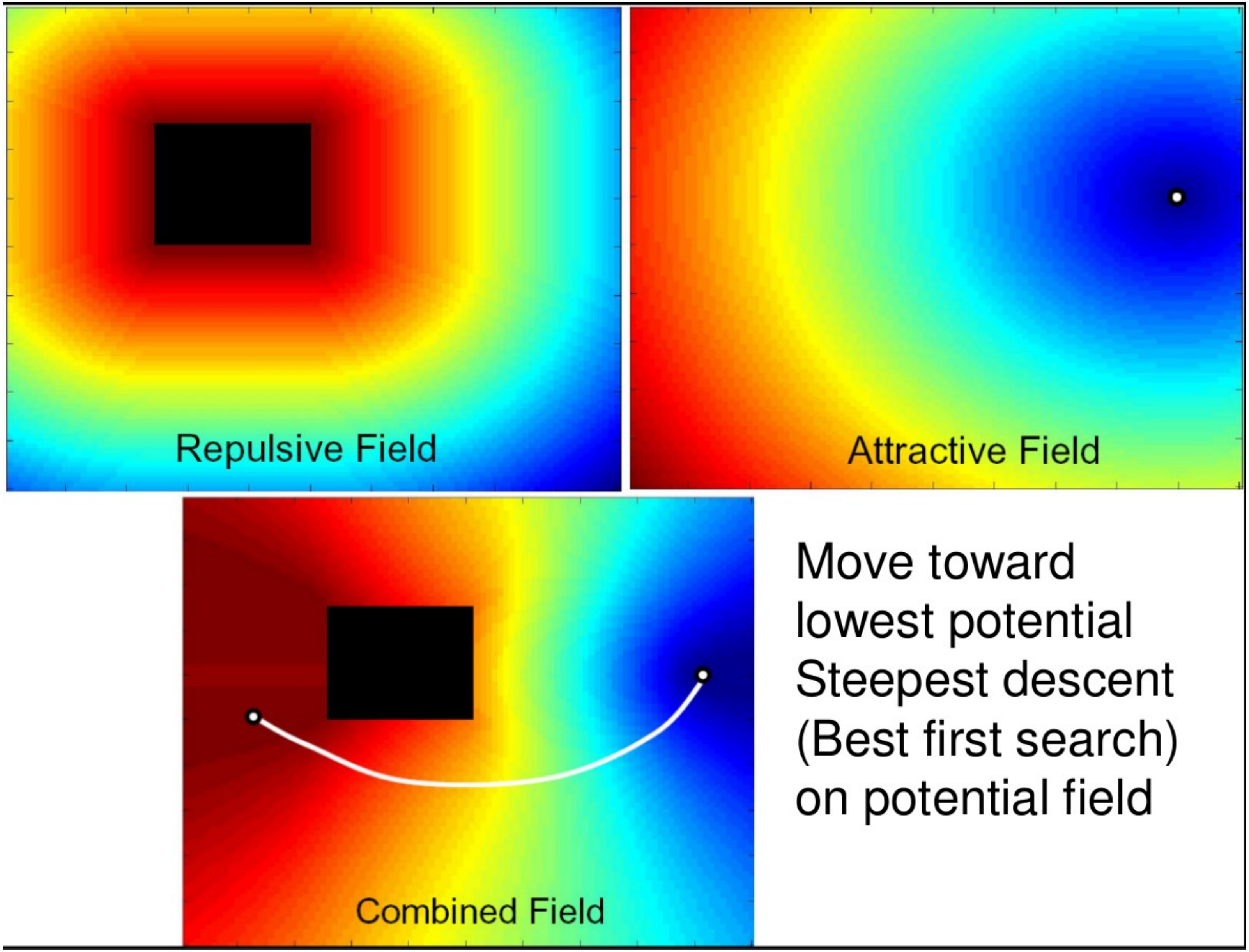
  - **Repulsive forces** from obstacles
  - **Attractive force** from goal

Method #6: Potential Functions



- Stay away from obstacles: Imagine that the obstacles are made of a material that generate a *repulsive* field
- Move closer to the goal: Imagine that the goal location is a particle that generates an *attractive* field

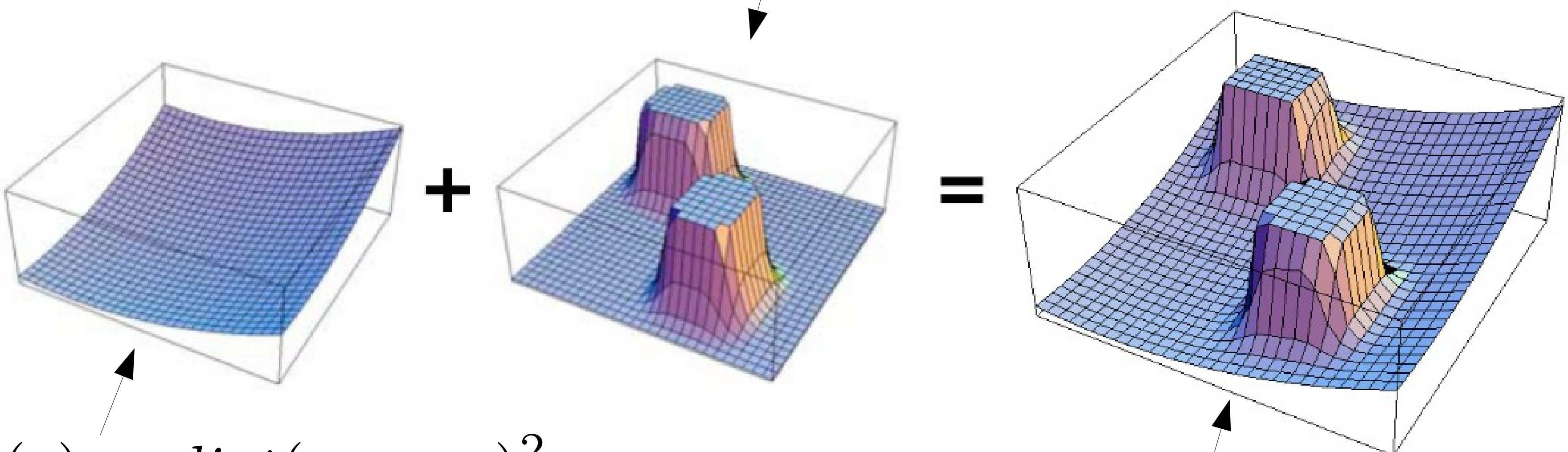
Method #6: Potential Functions



Move toward lowest potential
Steepest descent
(Best first search)
on potential field

Method #6: Potential Functions

$$U_o(q) = \frac{1}{dist(q, q_{Obstacles})^2}$$

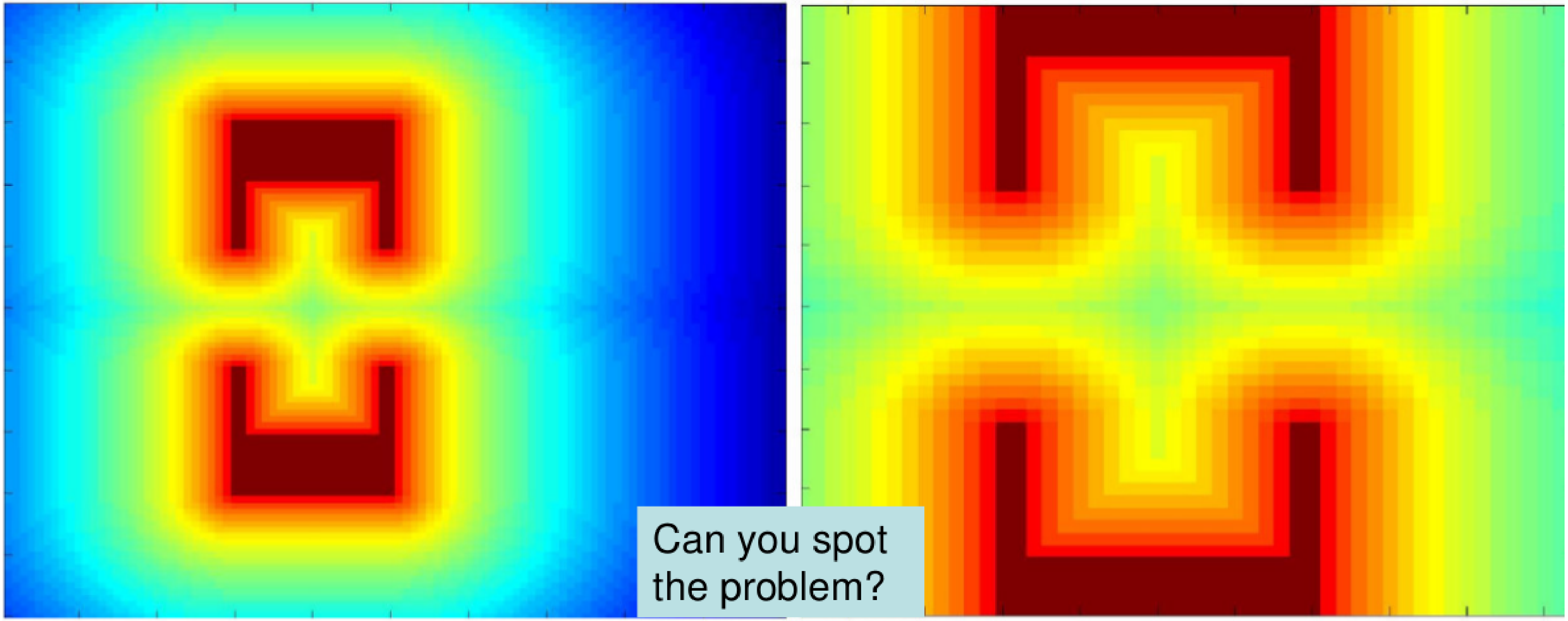


$$U_g(q) = dist(q, q_{goal})^2$$

$$U(q) = U_g(q) + \lambda U_o(q)$$

After computing U, follow the negative gradient: $\delta q = -\nabla U(q)$

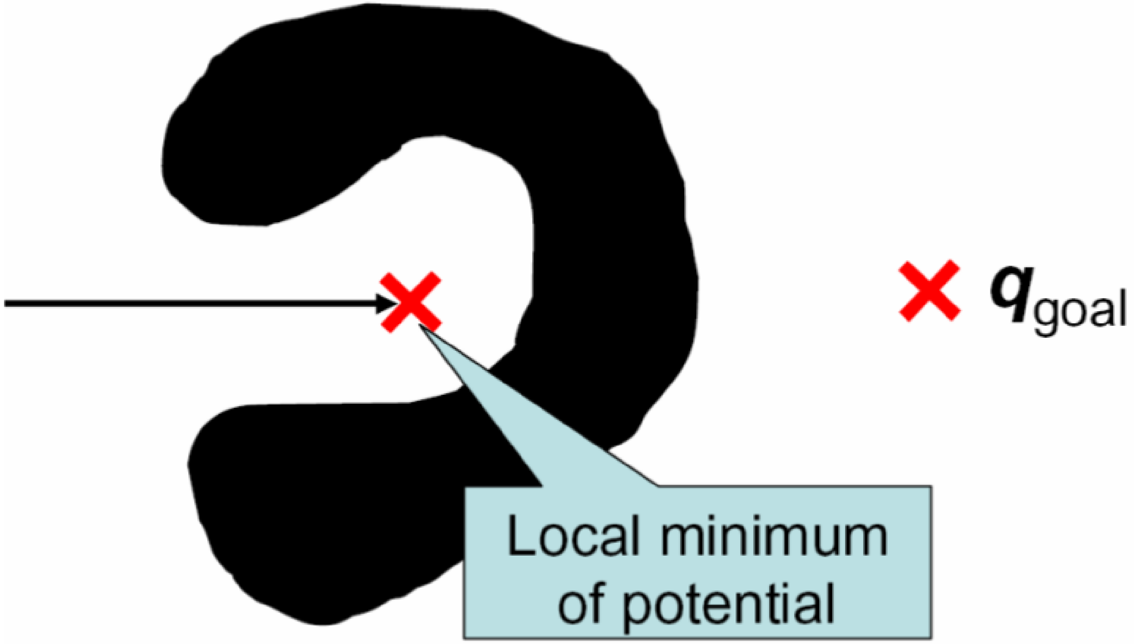
Potential Function Limitations



Potential field   Zoomed in view

- Completeness?
- Problems in higher dimensions

Potential Function Limitations



- Potential fields in general exhibit local minima

Applications to manipulators

Compute potential function in Cartesian space: $\delta x = -\nabla U(x)$

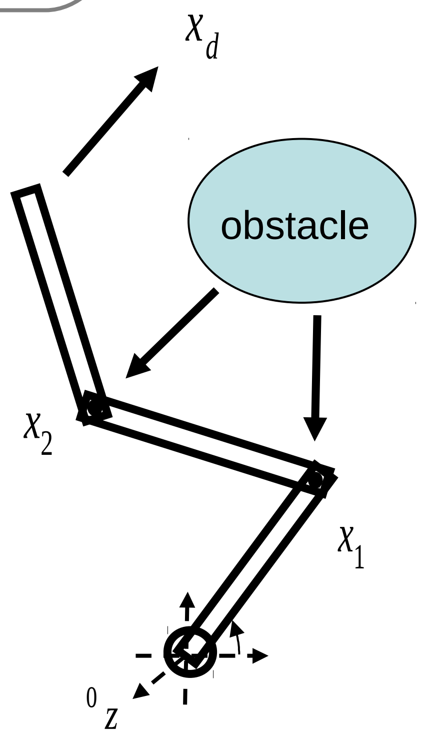
Project into joint space: $\delta q = -J_{eff}^{\#} \nabla U(x)$

Compute goal velocities at different points on the arm:

$$\delta q = -J_{eff}^{\#} \nabla U(x) - J_2^{\#} \nabla U_o(x) - J_1^{\#} \nabla U_o(x)$$

Pull eff toward goal and away from obstacles

Push x1 and x2 away from obstacles



Applications to manipulators

Can you draw a bug-trap-like scenario where this approach won't work?