CS 4610/5335 – Lecture 6
Path and Motion Planning

Lawson L.S. Wong
Northeastern University
2/7/22

Material adapted from:
1. Robert Platt, CS 4610/5335
2. Peter Corke, Robotics, Vision and Control
3. Marc Toussaint, U. Stuttgart Robotics Course
4. Oussama Khatib, Stanford CS 223A
5. Howie Choset
6. Kai Arras

Announcements

Ex1 due this Friday

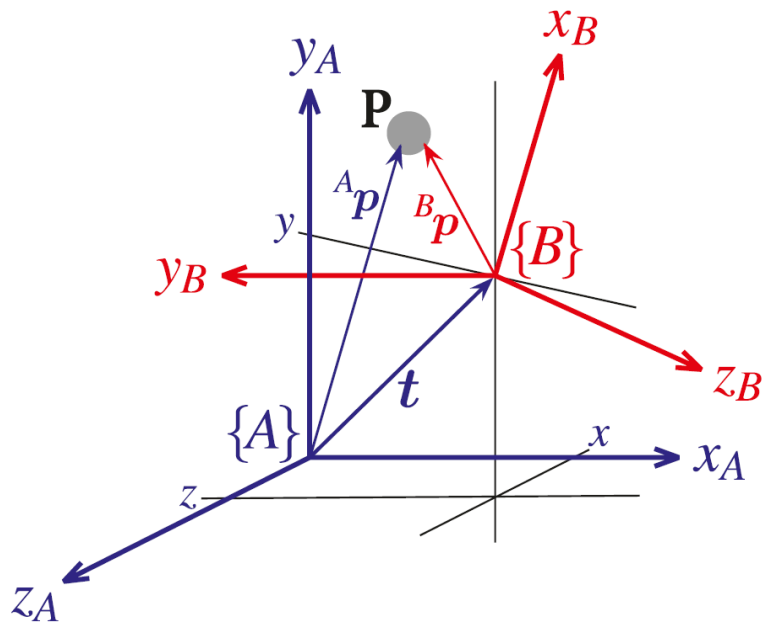
- Start early!
- Make use of office hours and Piazza
- Collaboration is welcome, with acknowledgment
  – but you must write up solutions independently
- Late day penalty is minimal, use them if it helps

Announcements

Ex1 due this Friday
- Start early!
- Make use of office hours and Piazza
- Collaboration is welcome, with acknowledgment
  – but you must write up solutions independently
- Late day penalty is minimal, use them if it helps

Project
- If you have already formed a team
  (or possibly just looking for a final member),
  create a public Piazza thread for your project
- See Piazza @35 for things you should include
  (you can fill these out gradually)

So far



Representations

Transformations

Kinematics

Forward:

$$f(q) = x$$

Inverse:

Input: x*
Output: q*
1. repeat until dx is small:
2. init q to random joint configuration
3. repeat K times:
4. x = FK(q)
5. dx = x*-x
6. dq = stepsize * $J^{\#}$ dx
7. q = q + dq
8. return q* = q

Ex1: Cartesian control using IK

So far

Representations

Transformations

Kinematics (forward and inverse)

This week: Let's get moving!

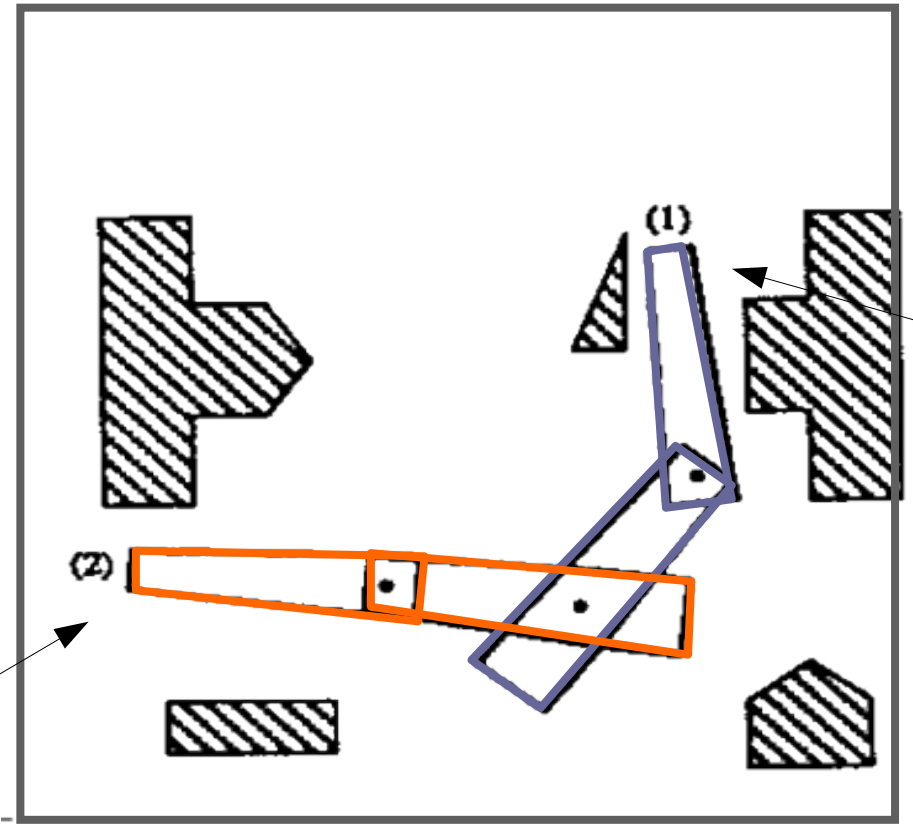
Recalling our earlier definition of a robot as a

goal oriented machine that can sense, plan and act,

this section concentrates on planning.

Problem we want to solve this week

Given: Description of the robot and obstacles
Find: Path from start to goal that is collision-free



Starting configuration

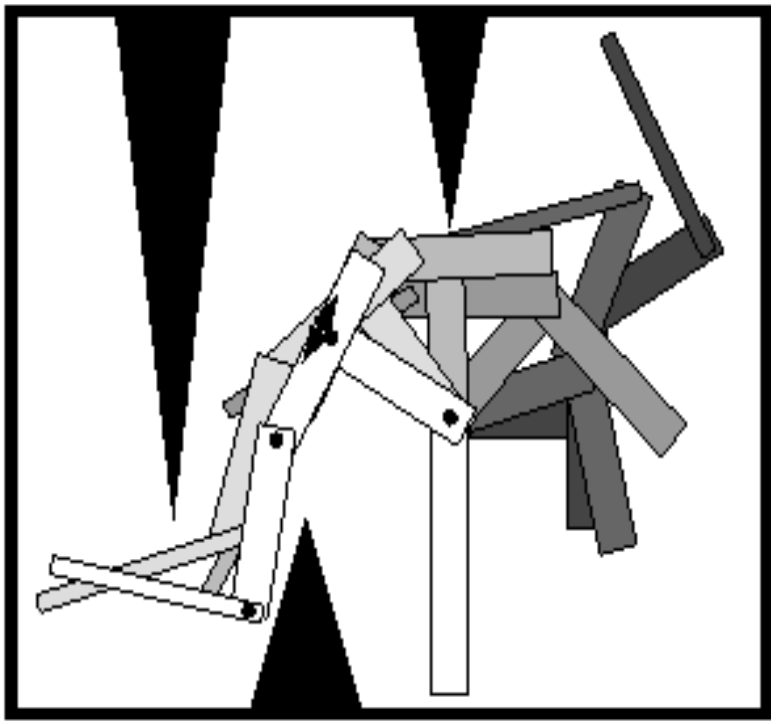
Goal configuration

Problem we want to solve this week

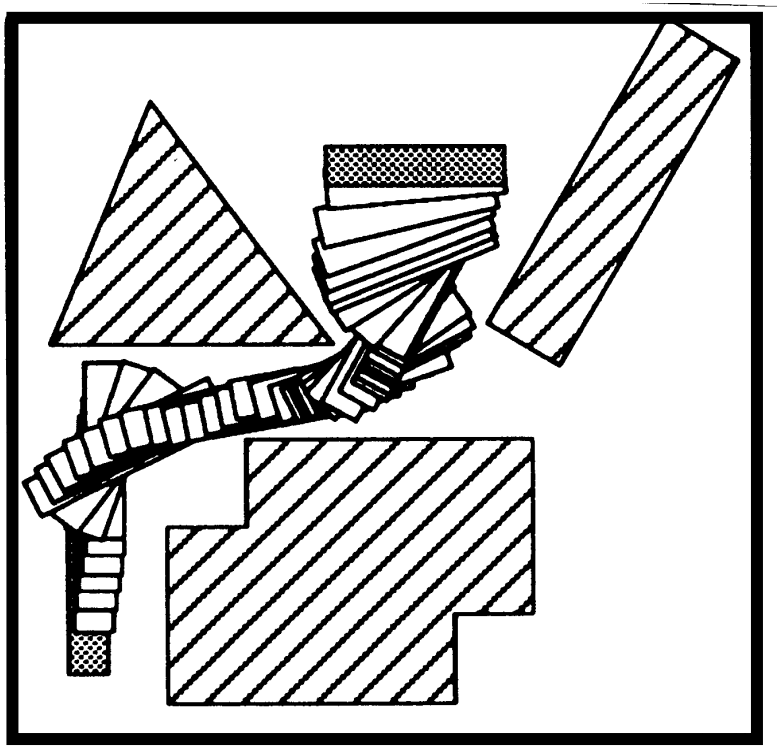
Given: Description of the robot and obstacles
Find: Path from start to goal that is collision-free

This is a very general problem in robotics

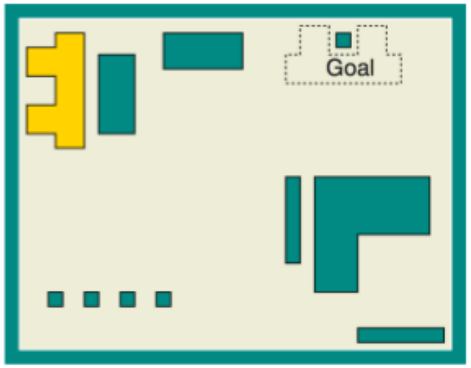
Manipulators

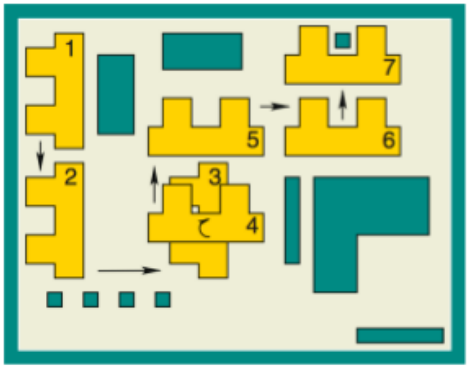


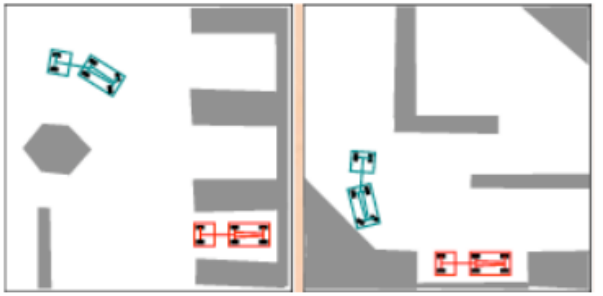
Mobile robots

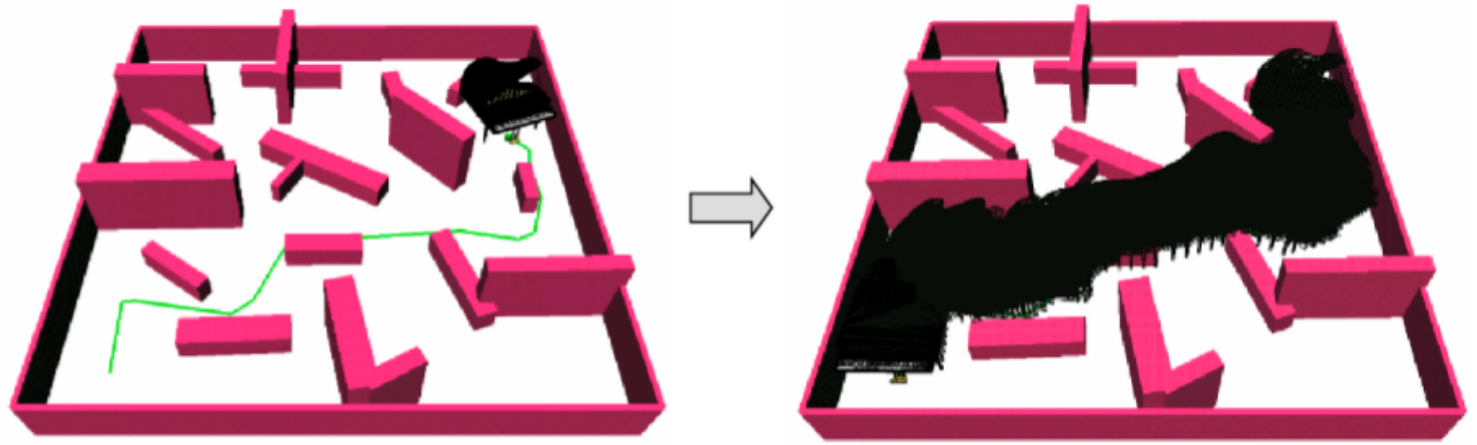


Problem we want to solve this week









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

Problem we want to solve this week

Given: Description of the robot and obstacles
Find: Path from start to goal that is collision-free

More specifically:

Given: A point robot (robot is a point in space)
A start and goal configuration
Shapes and positions of all obstacles

Find: A sequence of collision-free straight-line paths from start to goal

Assumptions: Position of robot known perfectly
Motion of robot controlled perfectly
Robot can move in any direction instantaneously (holonomic)

Outline

Heuristic solutions

Wavefront planner (distance transform)

Configuration-space motion planning

Planning actually not strictly necessary

5.1 Reactive Navigation

Surprisingly complex tasks can be performed by a robot even if it has no map and no real *idea* about where it is. As already mentioned robotic vacuum cleaners use only random motion and information from contact sensors to perform a complex task as shown in Fig. 5.1. Insects such as ants and bees gather food and return it to their nest based on input from their senses, they have far too few neurons to create any kind of mental map of the world and plan paths through it. Even single-celled organisms such as flagellate protozoa exhibit goal-seeking behaviors. In this case we need to temporarily modify our earlier definition of a robot to

> *a goal oriented machine that can sense, ~~plan~~ and act.*

Planning actually not strictly necessary

5.1 Reactive Navigation

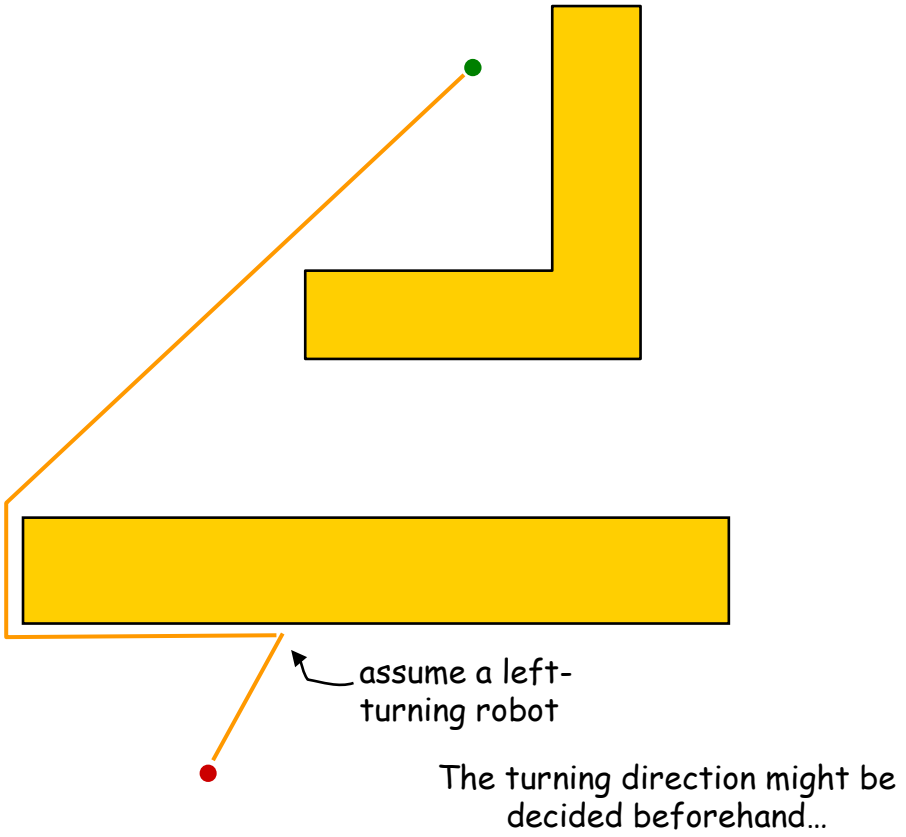
Surprisingly complex tasks can be performed by a robot even if it has no map and no real *idea* about where it is. As already mentioned robotic vacuum cleaners use only random motion and information from contact sensors to perform a complex task as shown in Fig. 5.1. Insects such as ants and bees gather food and return it to their nest based on input from their senses, they have far too few neurons to create any kind of mental map of the world and plan paths through it. Even single-celled organisms such as flagellate protozoa exhibit goal-seeking behaviors. In this case we need to temporarily modify our earlier definition of a robot to

a goal oriented machine that can sense, ~~plan~~ and act.

Simple automata / “bug” algorithms:

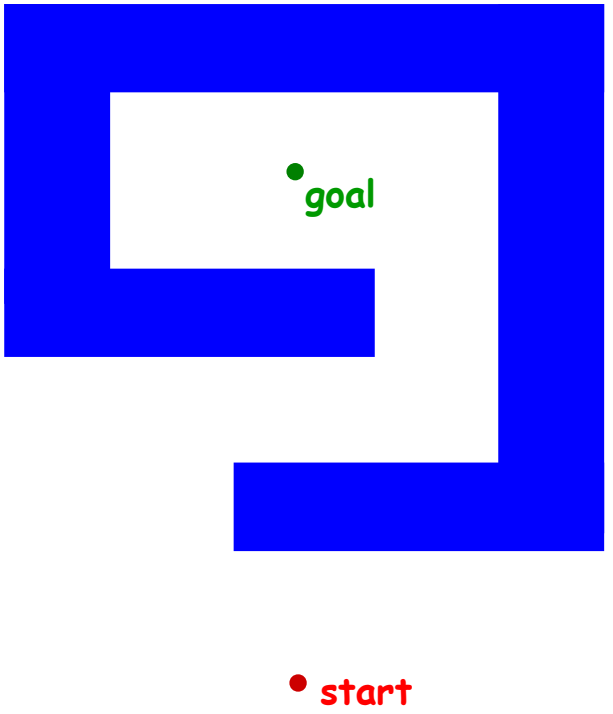
- Assume only local knowledge of environment is available
- Simple behaviors: Follow a wall / straight line to goal

First attempt: BUG 0



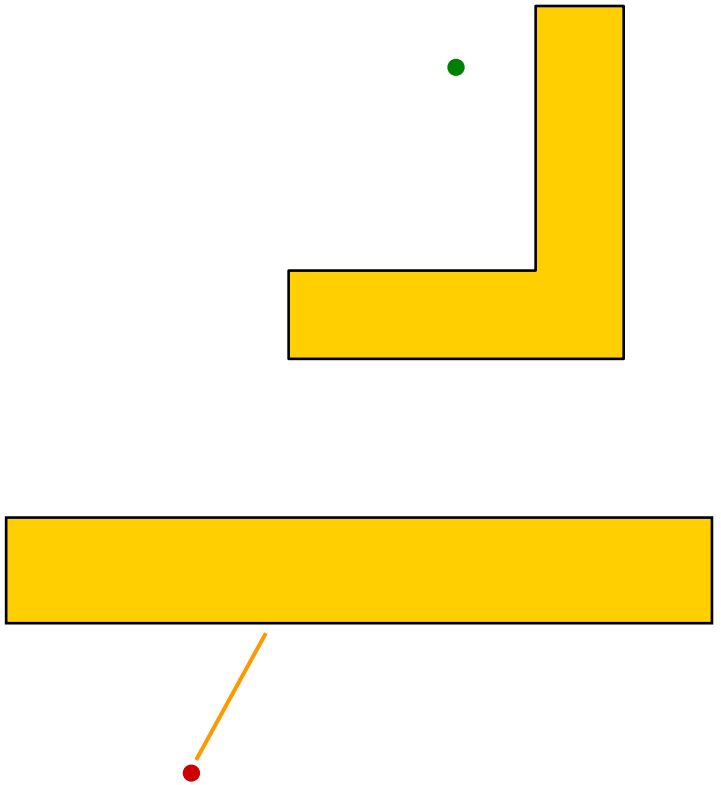
BUG 0:
1. head toward goal
2. if hit a wall, turn left
3. follow wall until a line toward goal will move you away from wall.
   (assume we only have local sensing –
   we cannot sense position of walls we are not touching)

First attempt: BUG 0



BUG 0:
1. head toward goal
2. if hit a wall, turn left
3. follow wall until a line toward goal will move you away from wall.
   (assume we only have local sensing –
   we cannot sense position of walls we are not touching)

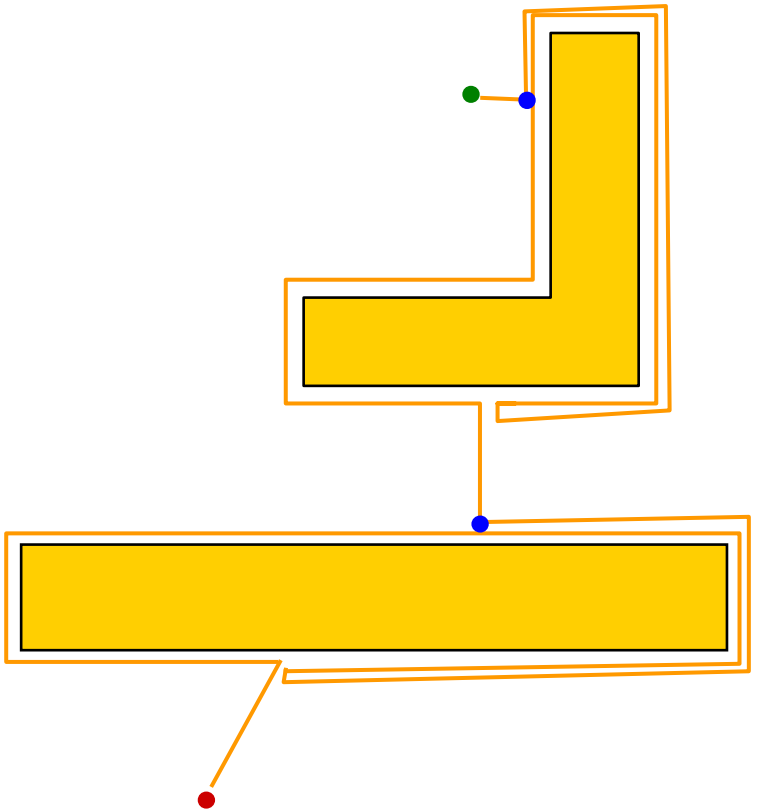
Second attempt: BUG 1



BUG 1:

1. move on straight line toward goal
2. if obstacle encountered, circumnavigate entire obstacle
   and remember how close bug got to goal
3. return to closest point and continue on a straight line toward goal

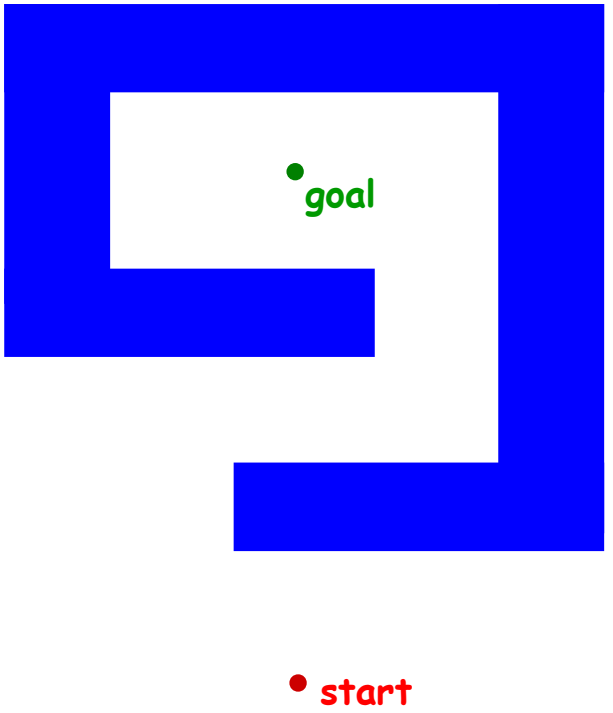
Second attempt: BUG 1



BUG 1:

1. move on straight line toward goal
2. if obstacle encountered, circumnavigate entire obstacle
   and remember how close bug got to goal
3. return to closest point and continue on a straight line toward goal

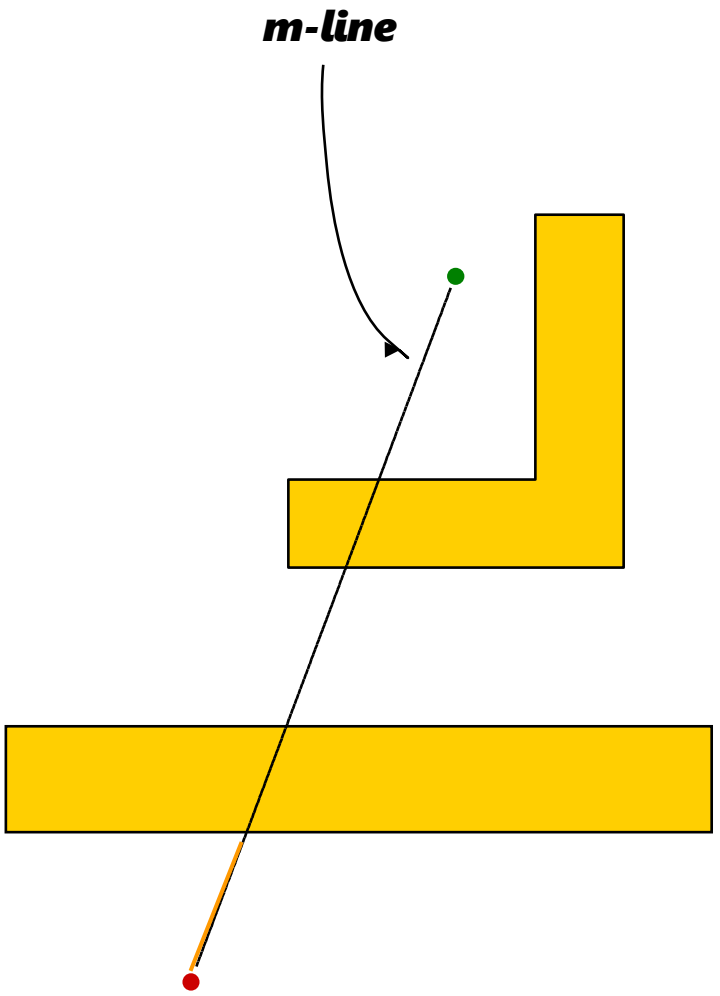
Second attempt: BUG 1



BUG 1:

1. move on straight line toward goal
2. if obstacle encountered, circumnavigate entire obstacle
   and remember how close bug got to goal
3. return to closest point and continue on a straight line toward goal

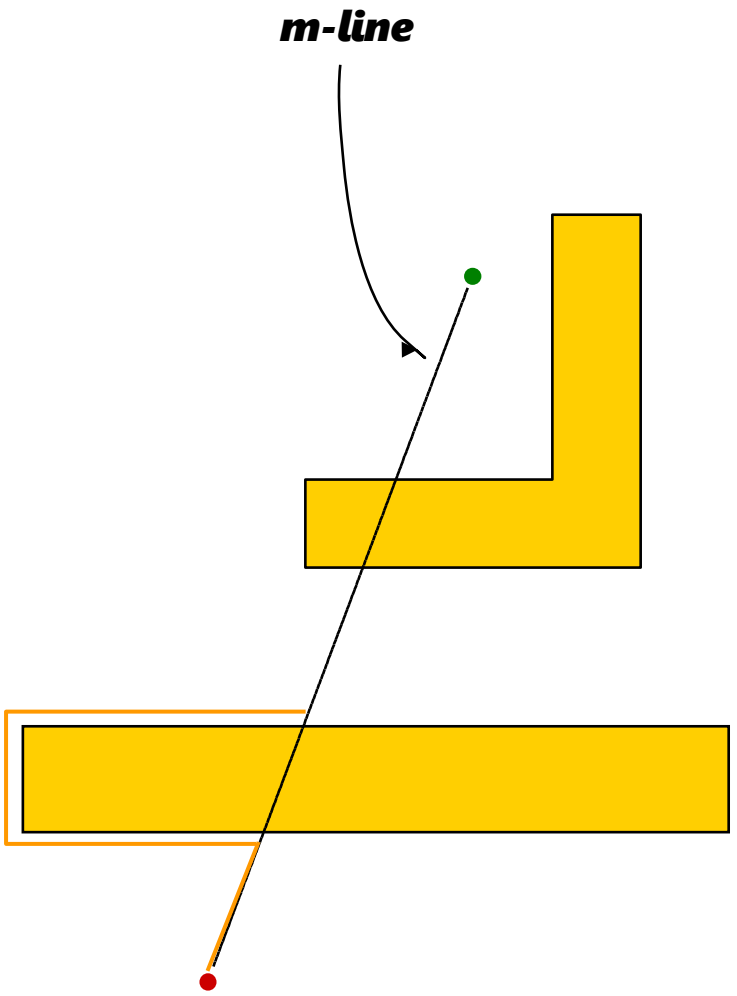
Another bug: BUG 2



BUG 2:

1. head toward goal on m-line

Another bug: BUG 2

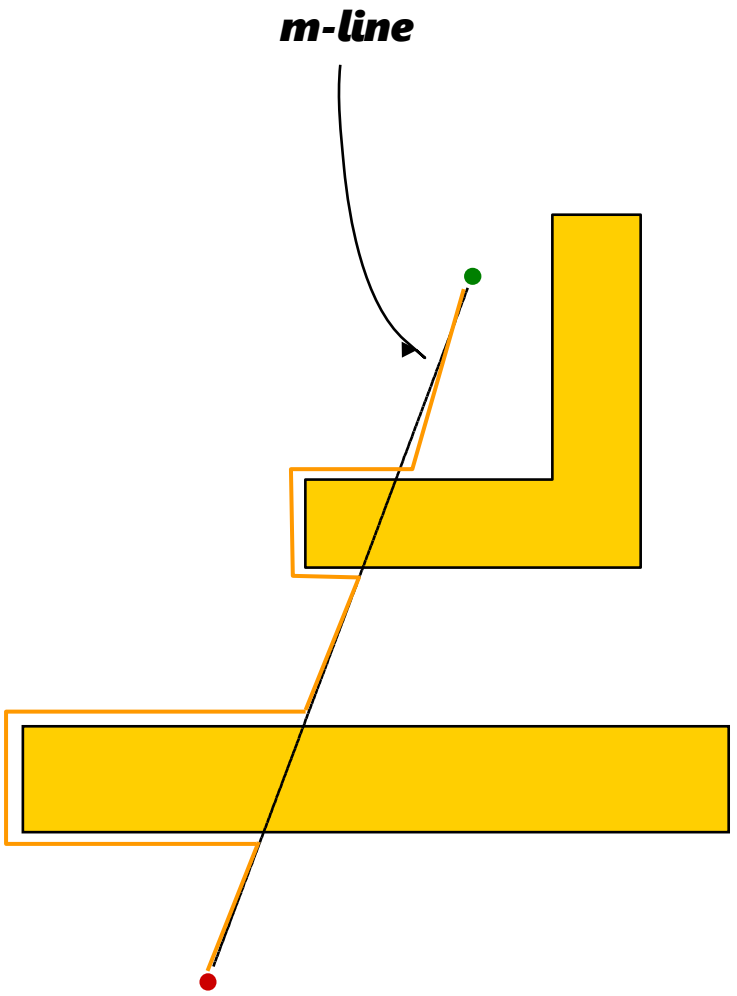


BUG 2:

1. head toward goal on m-line

2. if you encounter obstacle,
   follow it until you encounter
   m-line again at a point closer to goal

Another bug: BUG 2



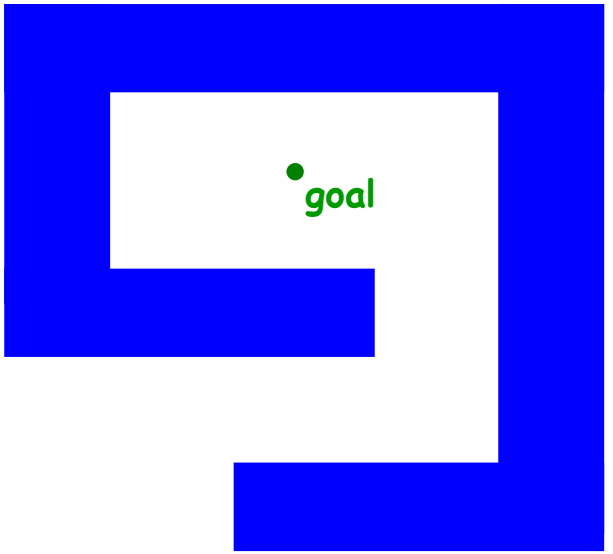
BUG 2:

1. head toward goal on m-line

2. if you encounter obstacle,
   follow it until you encounter
   m-line again at a point closer to goal

3. leave line and head toward goal again

Another bug: BUG 2



BUG 2:

1. head toward goal on m-line

2. if you encounter obstacle,
   follow it until you encounter
   m-line again at a point closer to goal

3. leave line and head toward goal again

BUG 2: Example

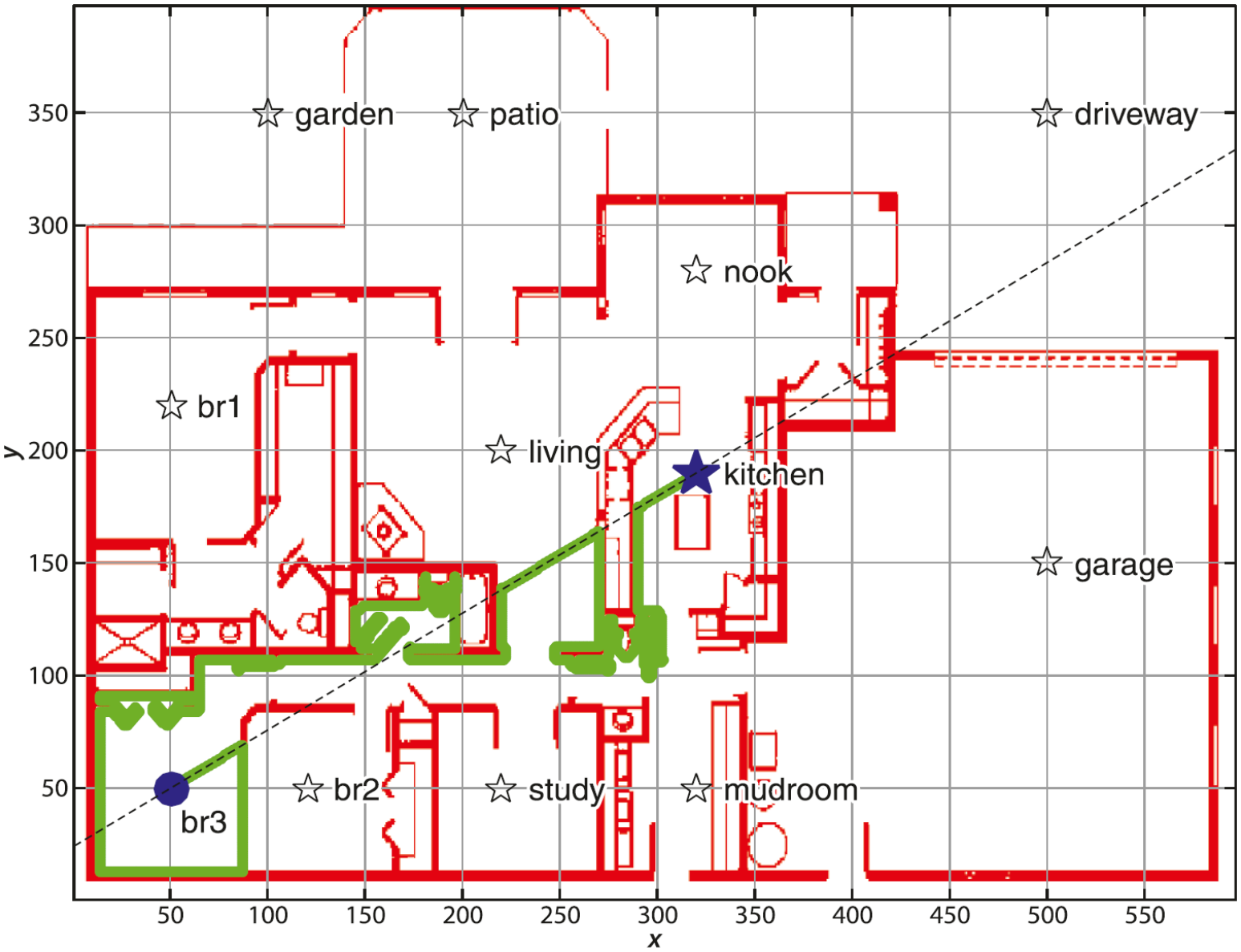


Fig. 5.4.
Obstacles are indicated by *red pixels*. Named places are indicated by *hollow black stars*. Approximate scale is 4.5 cm per cell. The start location is a *solid blue circle* and the goal is a *solid blue star*. The path taken by the bug2 algorithm is marked by a *green line*. The *black dashed line* is the m-line, the direct path from the start to the goal

Another heuristic solution: Potential fields

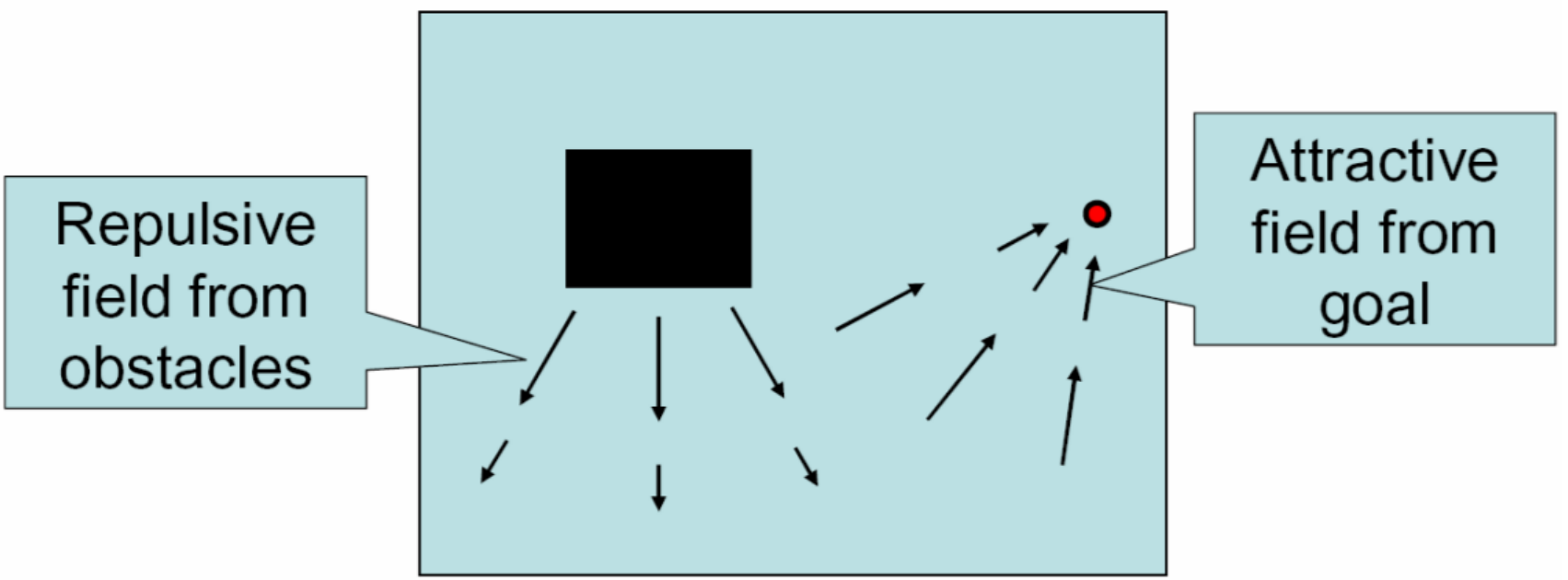
- **Potential Field methods** follow a different idea:

  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

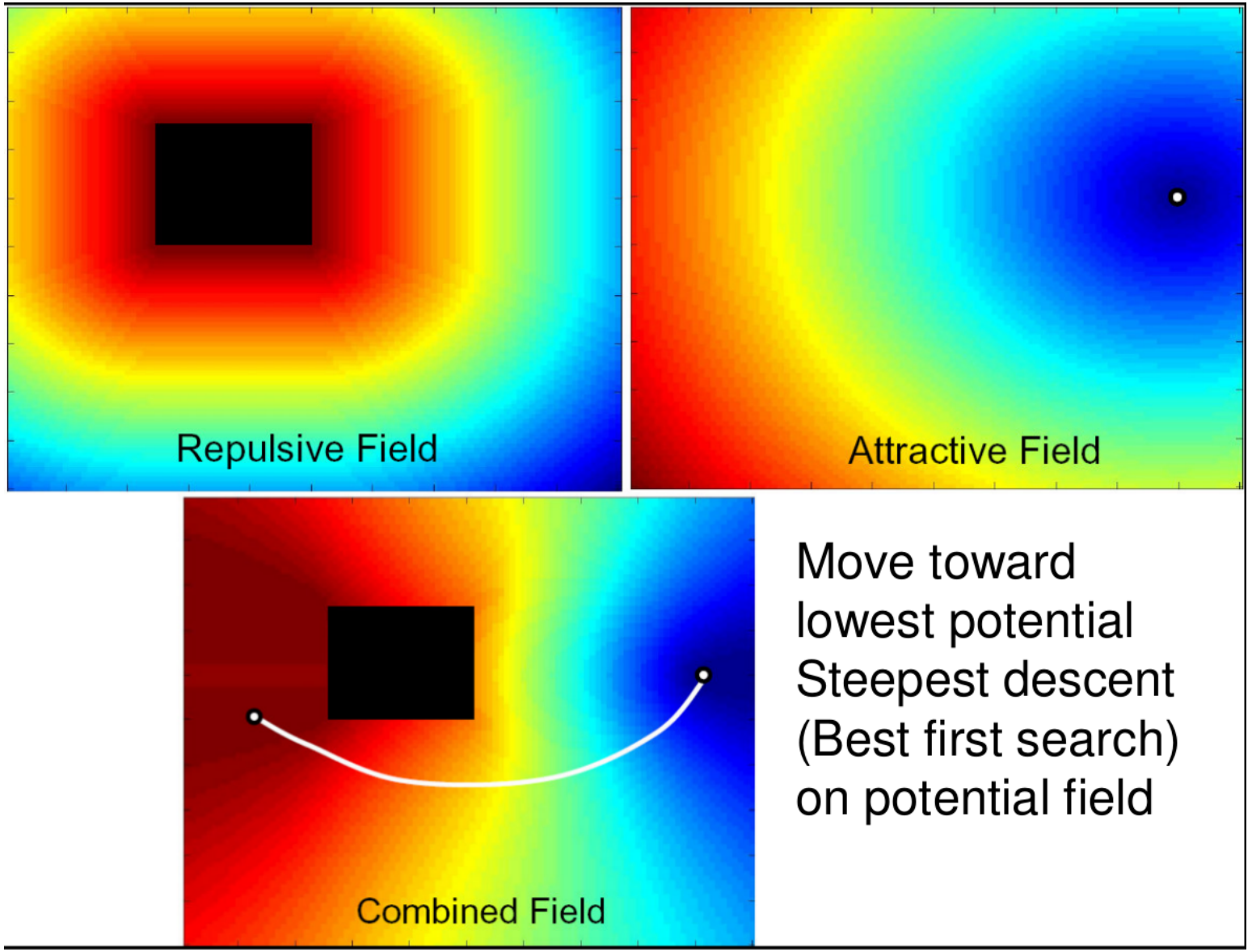
  - **Repulsive forces** from obstacles
  - **Attractive force** from goal

Another heuristic solution: Potential fields



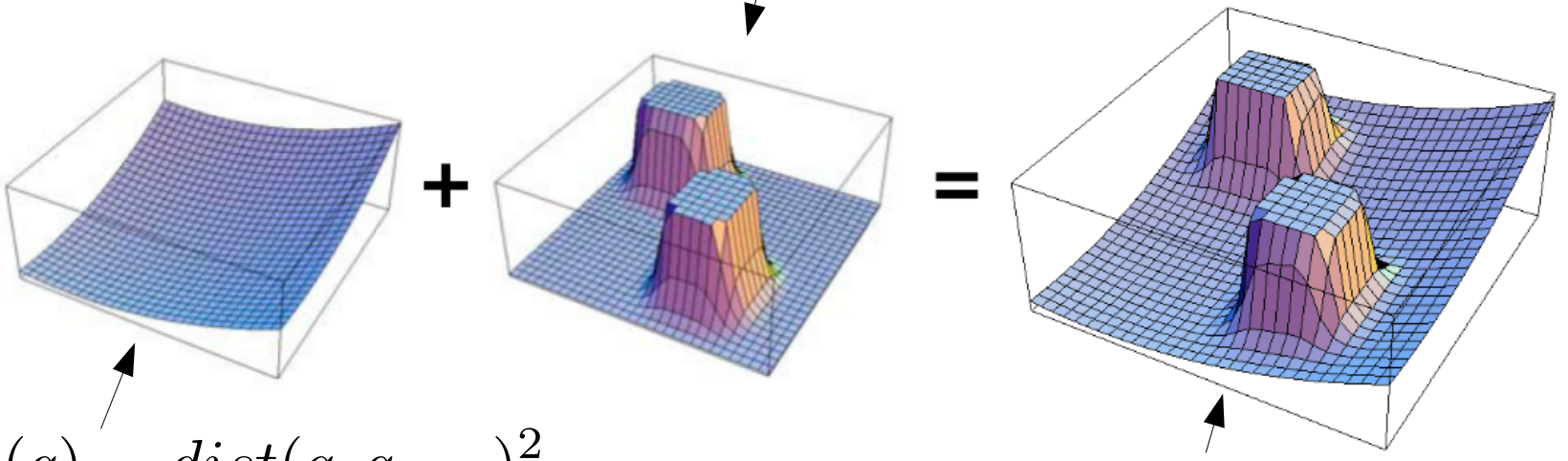
- Stay away from obstacles: Imagine that the obstacles are made of a material that generate a *repulsive* field
- Move closer to the goal: Imagine that the goal location is a particle that generates an *attractive* field

Another heuristic solution: Potential fields



Another heuristic solution: Potential fields

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

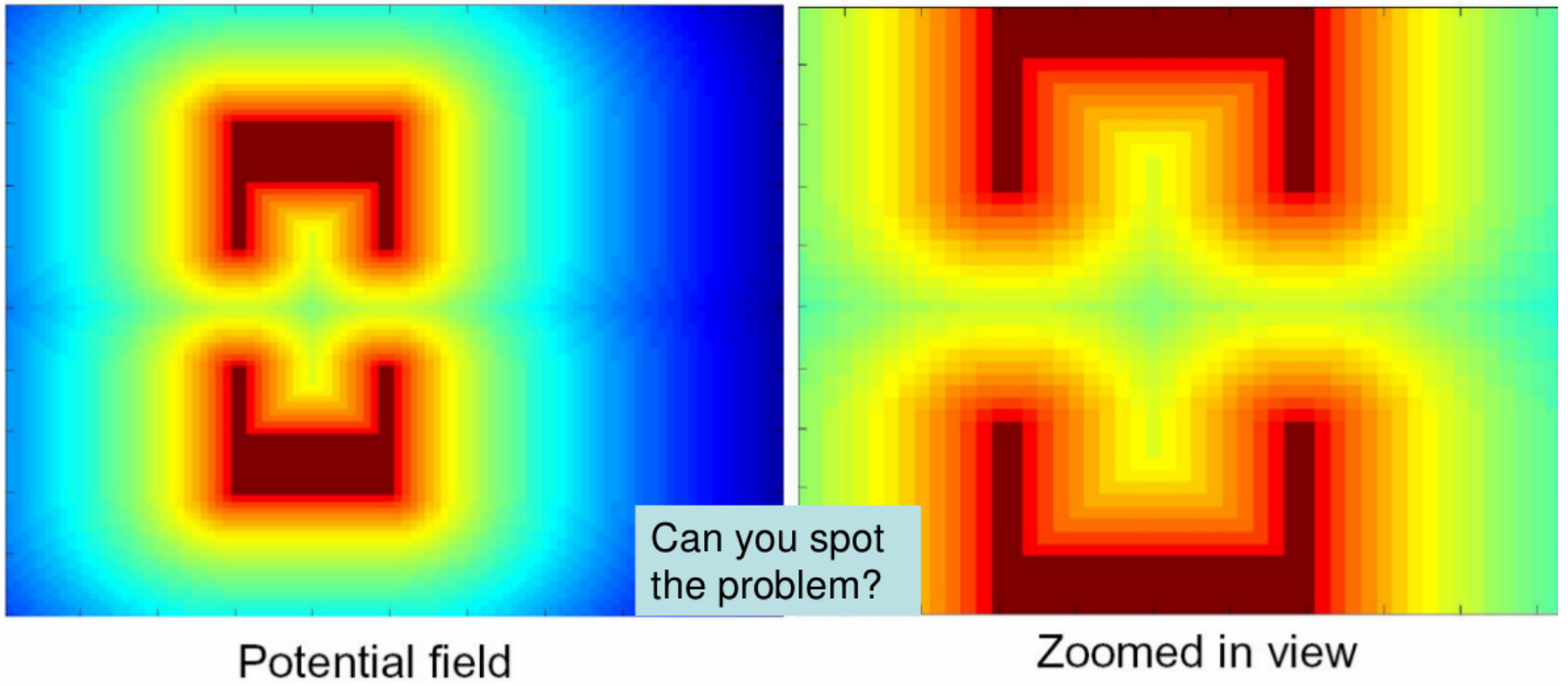


$$U_g(q) = dist(q, q_{goal})^2$$

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

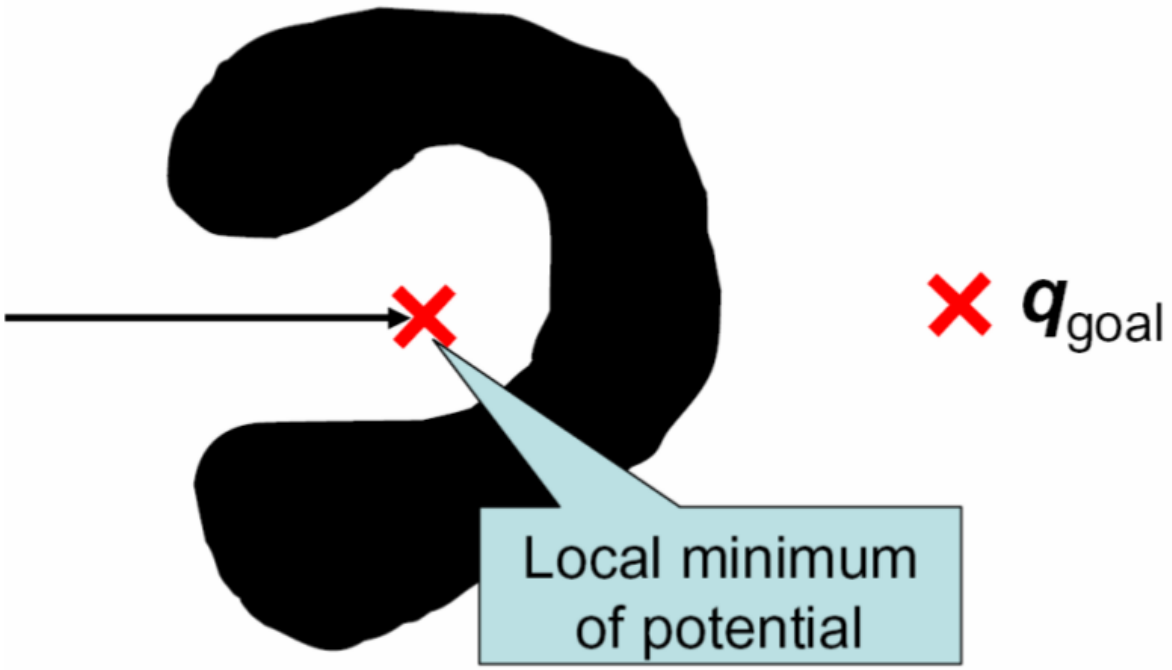
After computing *U*, follow the negative gra$\delta q = -\nabla U(q)$

Limitations of potential fields



- Completeness?
- Problems in higher dimensions

Bug trap



- Potential fields in general exhibit local minima

Outline

- ✔ Heuristic solutions

Wavefront planner (distance transform)

Configuration-space motion planning

Representation: Occupancy grid (2-D)

A simpler and very computer-friendly representation is the occupancy grid which is widely used in robotics.

An occupancy grid treats the world as a grid of cells and each cell is marked as occupied or unoccupied. We use zero to indicate an unoccupied cell or free space where the robot can drive. A value of one indicates an occupied or nondriveable cell. The size of the cell depends on the application. The memory required to hold the occupancy grid increases with the spatial area represented and inversely with the cell size.

Wavefront planner (distance transform)

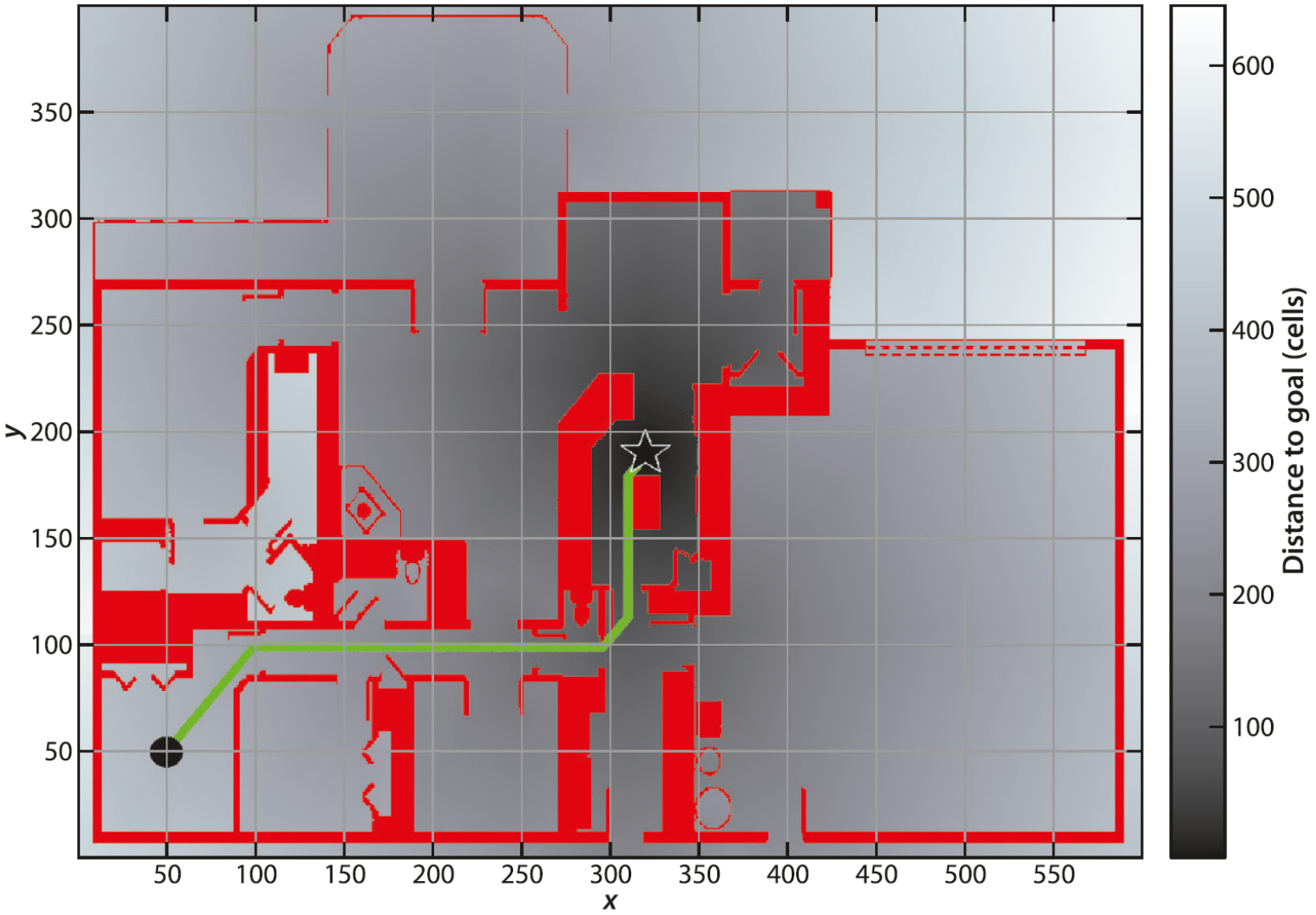
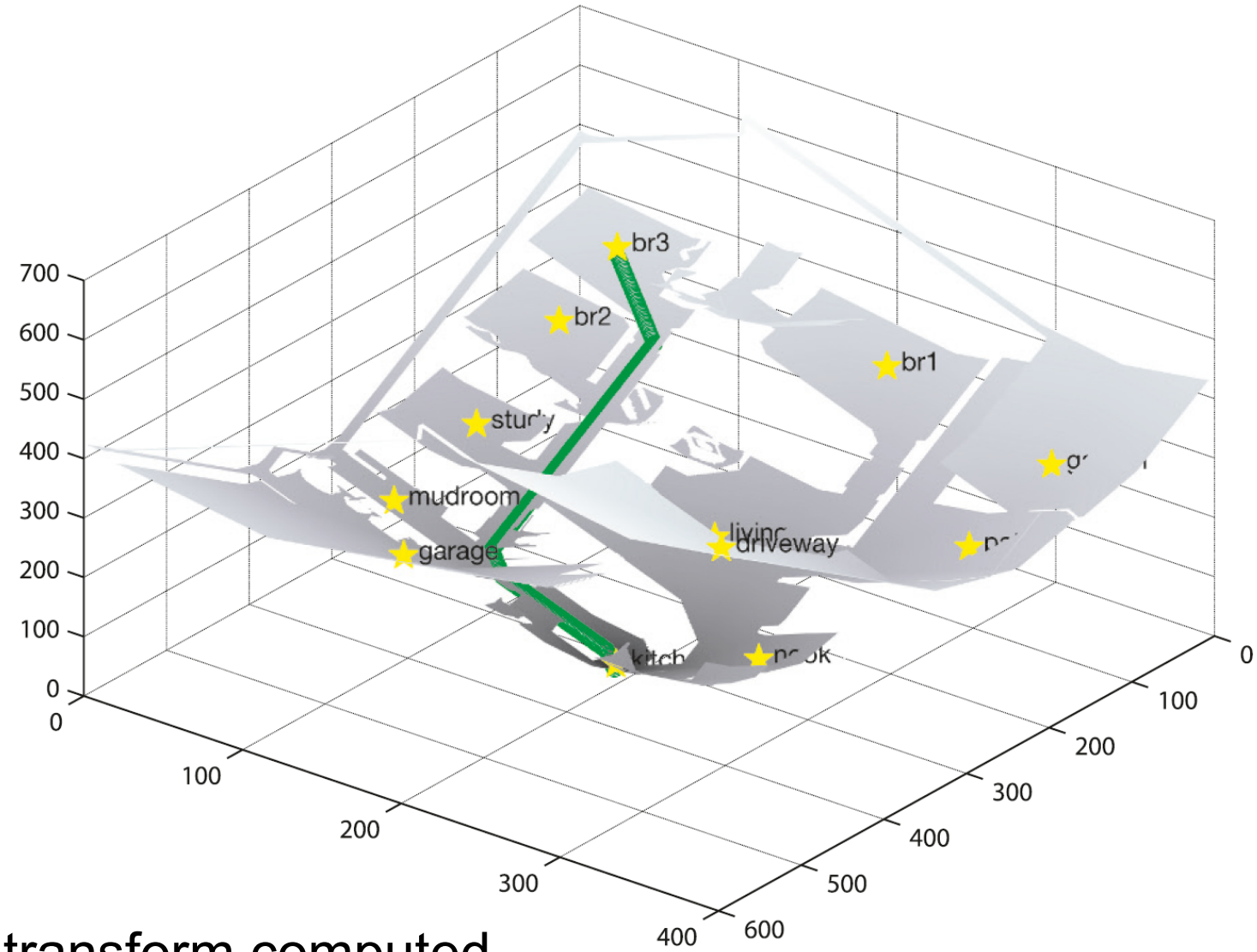


Fig. 5.5.
The distance transform path. Obstacles are indicated by *red cells*. The background grey intensity represents the cell's distance from the goal in units of cell size as indicated by the *scale* on the right-hand side

Intensity of a point denotes its obstacle-respecting distance to the goal

Wavefront planner (distance transform)

Fig. 5.6.
The distance transform as a 3D function, where height is distance from the goal. Navigation is simply a downhill run. Note the discontinuity in the distance transform where the split wavefronts met



Once distance transform computed,
greedily follow it “downhill” to reach goal

Wavefront planner (distance transform)

Algorithm:

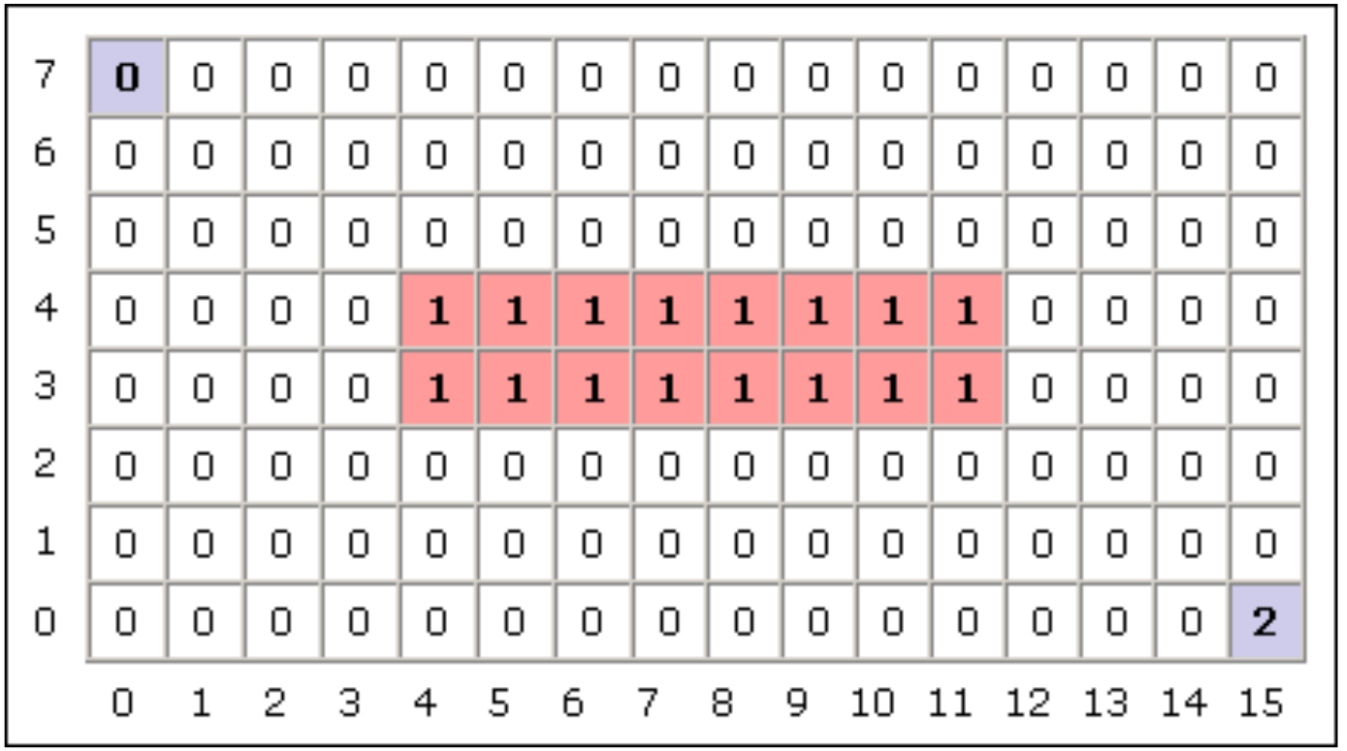
1. L = {goal state}, d(goal state) = 2, d(obstacle states) = 1, d(rest of states) = 0
2. while L != null
3. pop item i from L
4. for all neighbors j of i such that d(j)==0
5. d(j) = d(i)+1
6. push j onto L

L: List of nodes in wavefront; initially just the goal state

d: Distance function over nodes; initially
d(goal) = 2, d(obstacle) = 1, d(all other states) = 0

Wavefront planner (distance transform)

Algorithm:
1. L = {goal state}, d(goal state) = 2, d(obstacle states) = 1, d(rest of states) = 0
2. while L != null
3. pop item i from L
4. for all neighbors j of i such that d(j)==0
5. d(j) = d(i)+1
6. push j onto L



Wavefront planner (distance transform)

Algorithm:
1. L = {goal state}, d(goal state) = 2, d(obstacle states) = 1, d(rest of states) = 0
2. while L != null
3. pop item i from L
4. for all neighbors j of i such that d(j)==0
5. d(j) = d(i)+1
6. push j onto L

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
7	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0
3	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	**2**

Wavefront planner (distance transform)

Algorithm:
1. L = {goal state}, d(goal state) = 2, d(obstacle states) = 1, d(rest of states) = 0
2. while L != null
3. pop item i from L
4. for all neighbors j of i such that d(j)==0
5. d(j) = d(i)+1
6. push j onto L

7	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0
3	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4
1	0	0	0	0	0	0	0	0	0	0	0	0	0	4	3	3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	3	**2**
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Wavefront planner (distance transform)

Algorithm:
1. L = {goal state}, d(goal state) = 2, d(obstacle states) = 1, d(rest of states) = 0
2. while L != null
3. pop item i from L
4. for all neighbors j of i such that d(j)==0
5. d(j) = d(i)+1
6. push j onto L

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
7	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0
3	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	5	5	5	5
2	0	0	0	0	0	0	0	0	0	0	0	0	5	4	4	4
1	0	0	0	0	0	0	0	0	0	0	0	0	5	4	3	3
0	0	0	0	0	0	0	0	0	0	0	0	0	5	4	3	**2**

Wavefront planner (distance transform)

Algorithm:
1. L = {goal state}, d(goal state) = 2, d(obstacle states) = 1, d(rest of states) = 0
2. while L != null
3. pop item i from L
4. for all neighbors j of i such that d(j)==0
5. d(j) = d(i)+1
6. push j onto L

7	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	6	6	6	6
3	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	5	5	5	5
2	0	0	0	0	0	0	0	0	0	0	0	6	5	4	4	4
1	0	0	0	0	0	0	0	0	0	0	0	6	5	4	3	3
0	0	0	0	0	0	0	0	0	0	0	0	6	5	4	3	**2**
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Wavefront planner (distance transform)

Algorithm:
1. L = {goal state}, d(goal state) = 2, d(obstacle states) = 1, d(rest of states) = 0
2. while L != null
3. pop item i from L
4. for all neighbors j of i such that d(j)==0
5. d(j) = d(i)+1
6. push j onto L

7	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	7	7	7	7	7
4	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	6	6	6	6
3	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	5	5	5	5
2	0	0	0	0	0	0	0	0	0	0	7	6	5	4	4	4
1	0	0	0	0	0	0	0	0	0	0	7	6	5	4	3	3
0	0	0	0	0	0	0	0	0	0	0	7	6	5	4	3	**2**
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Wavefront planner (distance transform)

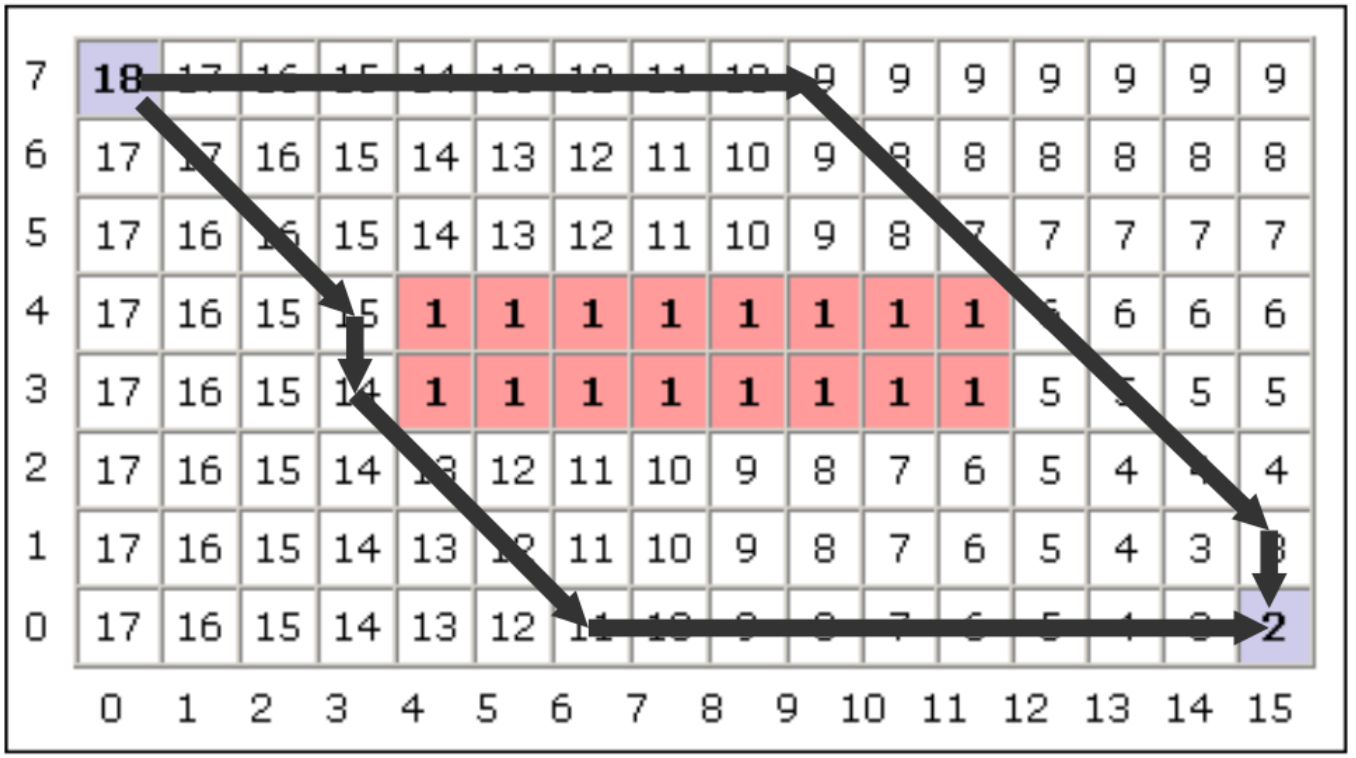
Algorithm:
1. L = {goal state}, d(goal state) = 2, d(obstacle states) = 1, d(rest of states) = 0
2. while L != null
3. pop item i from L
4. for all neighbors j of i such that d(j)==0
5. d(j) = d(i)+1
6. push j onto L

(… more steps later)

7	**18**	17	16	15	14	13	12	11	10	9	9	9	9	9	9	9
6	17	17	16	15	14	13	12	11	10	9	8	8	8	8	8	8
5	17	16	16	15	14	13	12	11	10	9	8	7	7	7	7	7
4	17	16	15	15	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	6	6	6	6
3	17	16	15	14	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	5	5	5	5
2	17	16	15	14	13	12	11	10	9	8	7	6	5	4	4	4
1	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	3
0	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	**2**
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

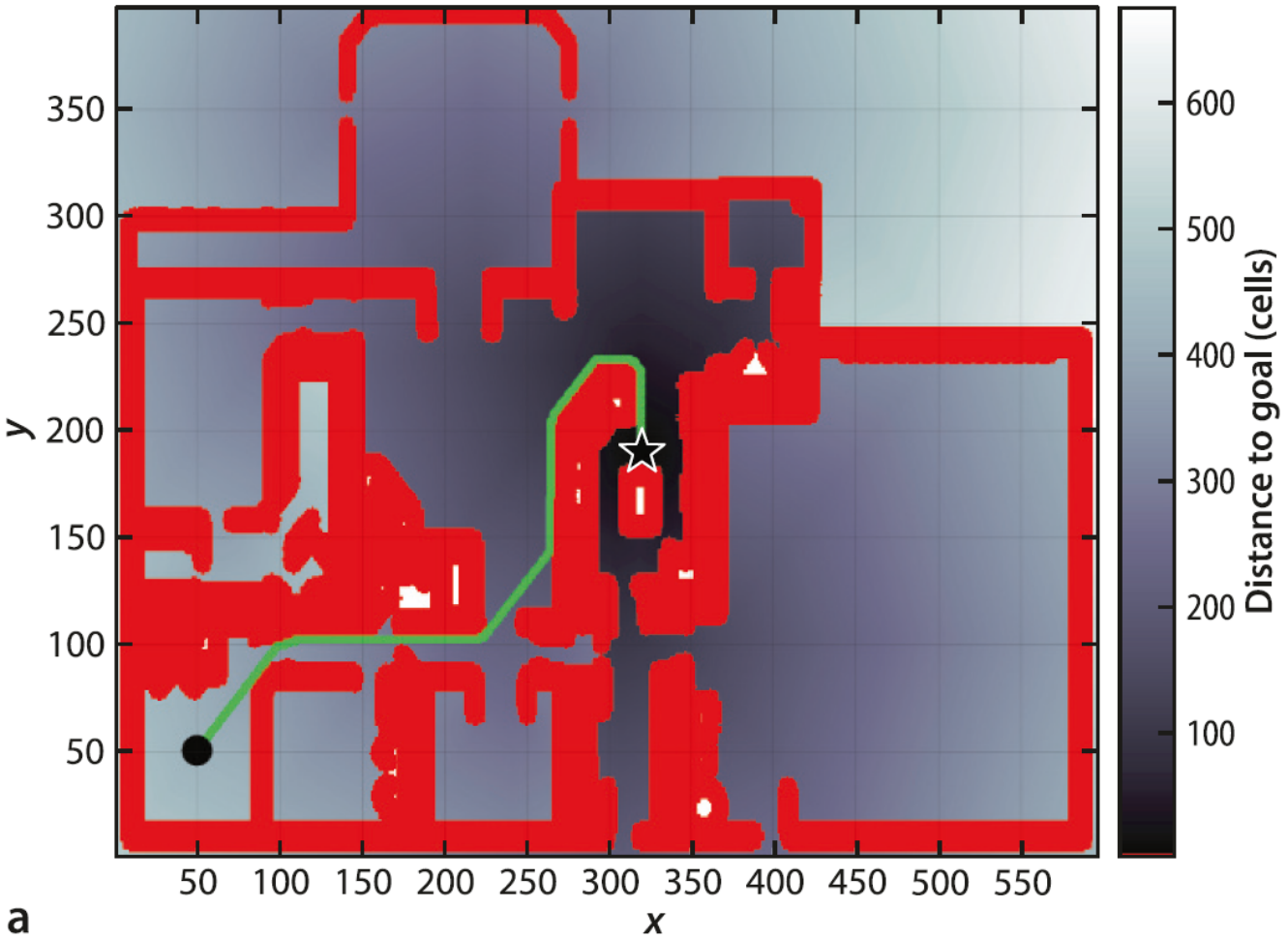
Wavefront planner (distance transform)

Algorithm:
1. L = {goal state}, d(goal state) = 2, d(obstacle states) = 1, d(rest of states) = 0
2. while L != null
3. pop item i from L
4. for all neighbors j of i such that d(j)==0
5. d(j) = d(i)+1
6. push j onto L



What if not a point robot?

Fig. 5.7. Distance transform path with obstacles inflated by 5 cells. **a** Path shown with inflated obstacles; **b** path computed for inflated obstacles overlaid on original obstacle map, black regions are where no distance was computed



Other related algorithms

Dijkstra's algorithm – shortest paths on a graph

A* – forward search from start to goal (+ heuristic function)

D* – incremental A* supporting changes in the map

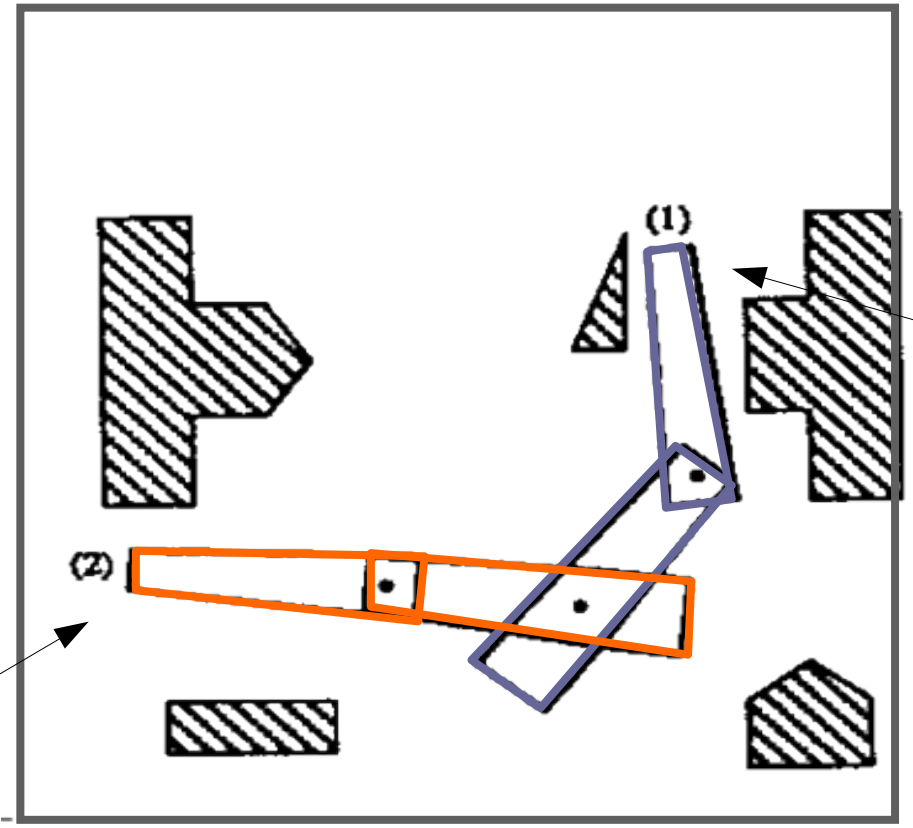
Outline

- ✔ Heuristic solutions
- ✔ Wavefront planner (distance transform)

Configuration-space motion planning

Problem we want to solve this week

Given: Description of the robot and obstacles
Find: Path from start to goal that is collision-free



Starting configuration

Goal configuration

Planning in “configuration space”

Given: Description of the robot and obstacles
Find: Path from start to goal that is collision-free

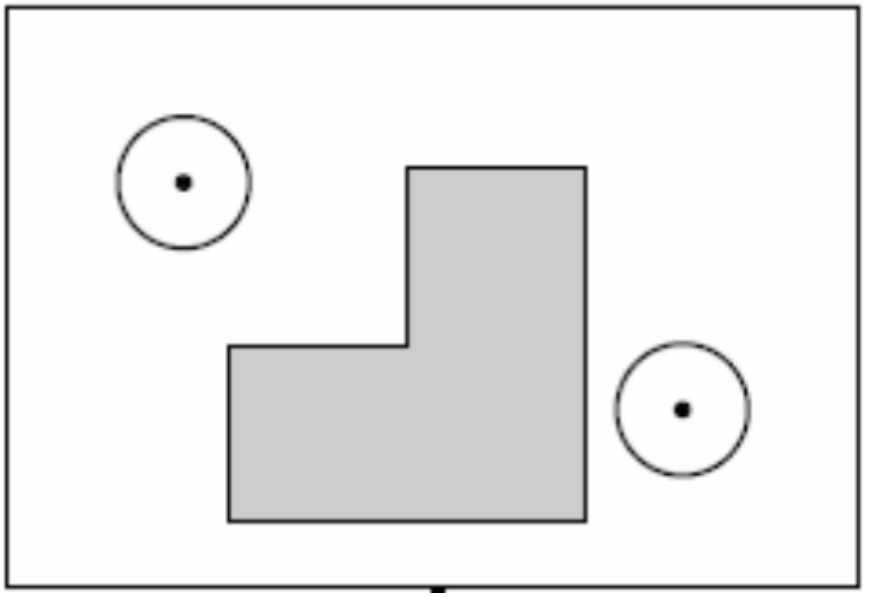
Key idea: Convert original planning problem
into a planning problem for a single point

Planning in “configuration space”

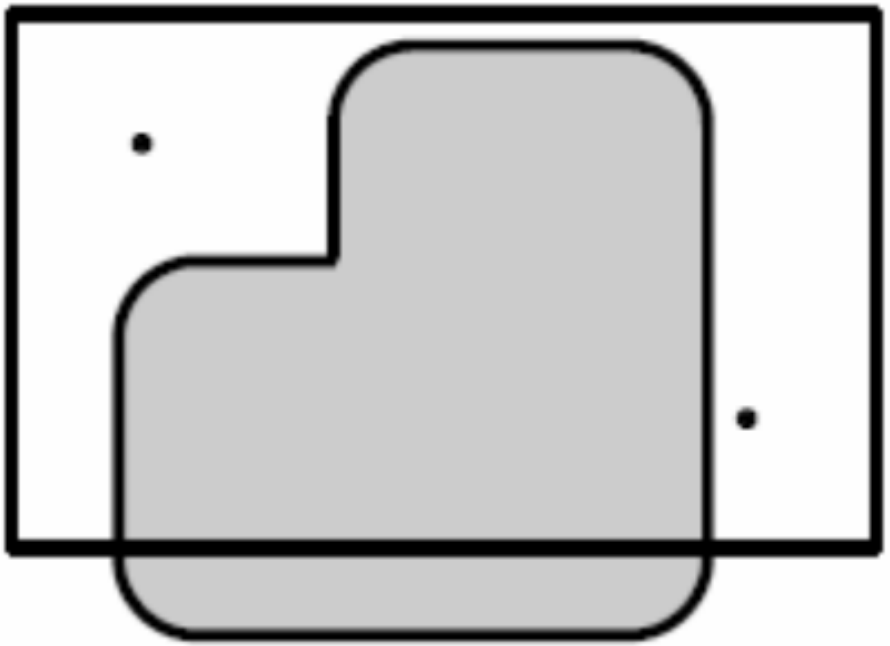
Given: Description of the robot and obstacles
Find: Path from start to goal that is collision-free

Key idea: Convert original planning problem
into a planning problem for a single point

workspace



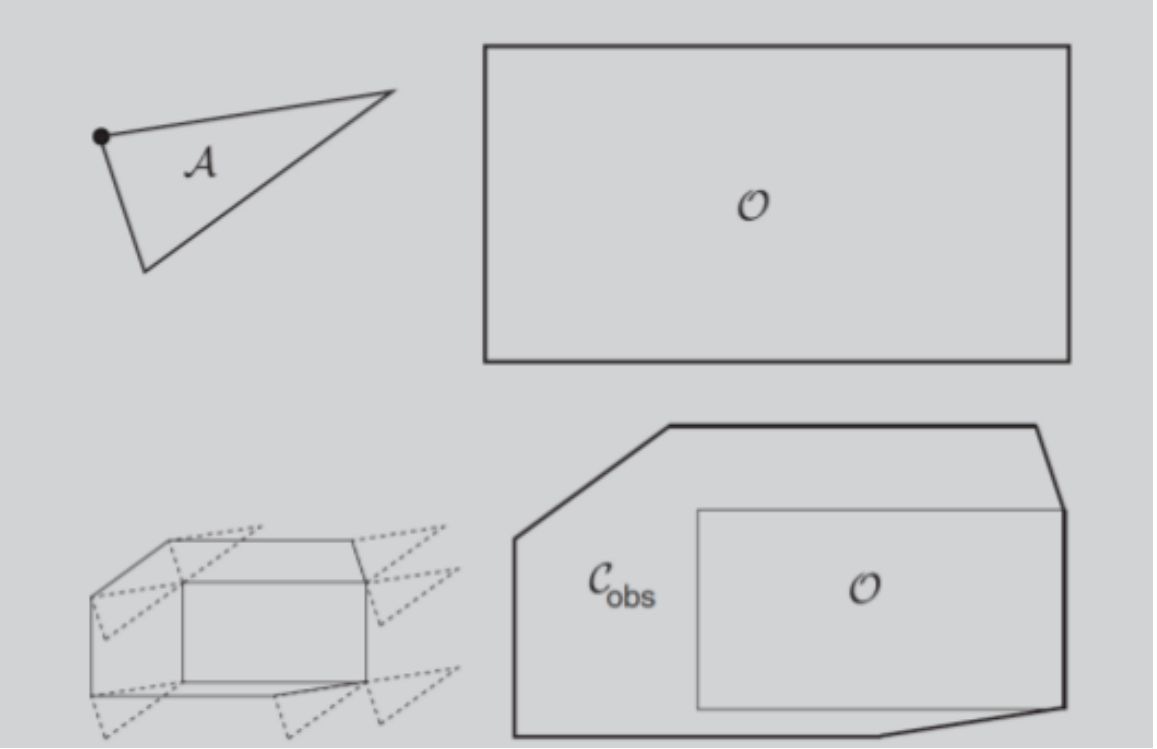
configuration space



Planning in “configuration space”

Given: Description of the robot and obstacles
Find: Path from start to goal that is collision-free

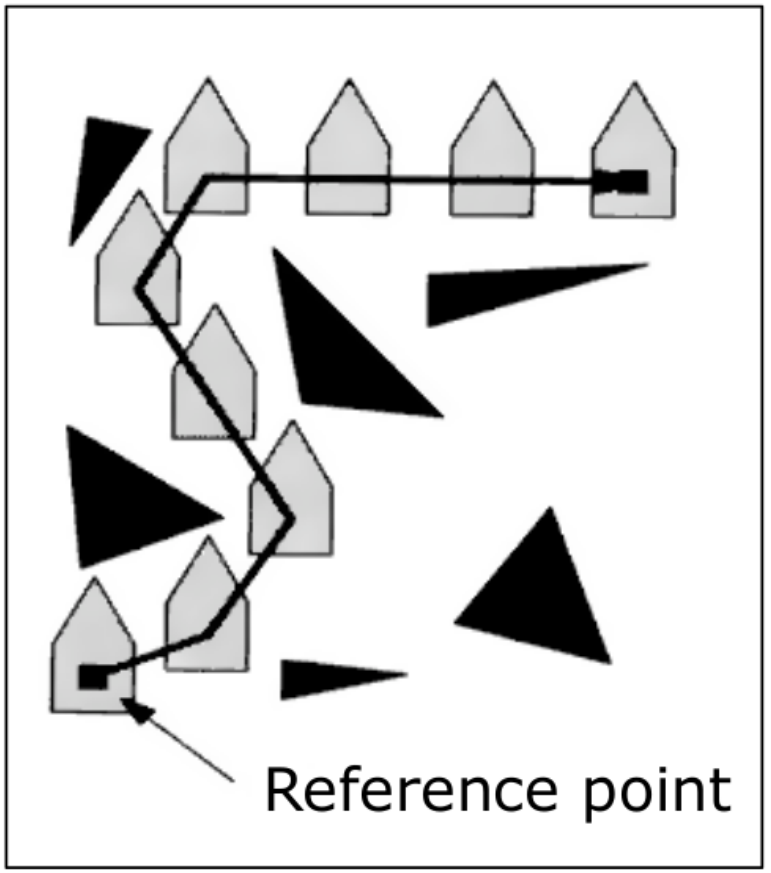
Key idea: Convert original planning problem
into a planning problem for a single point
(convert using Minkowski sum)



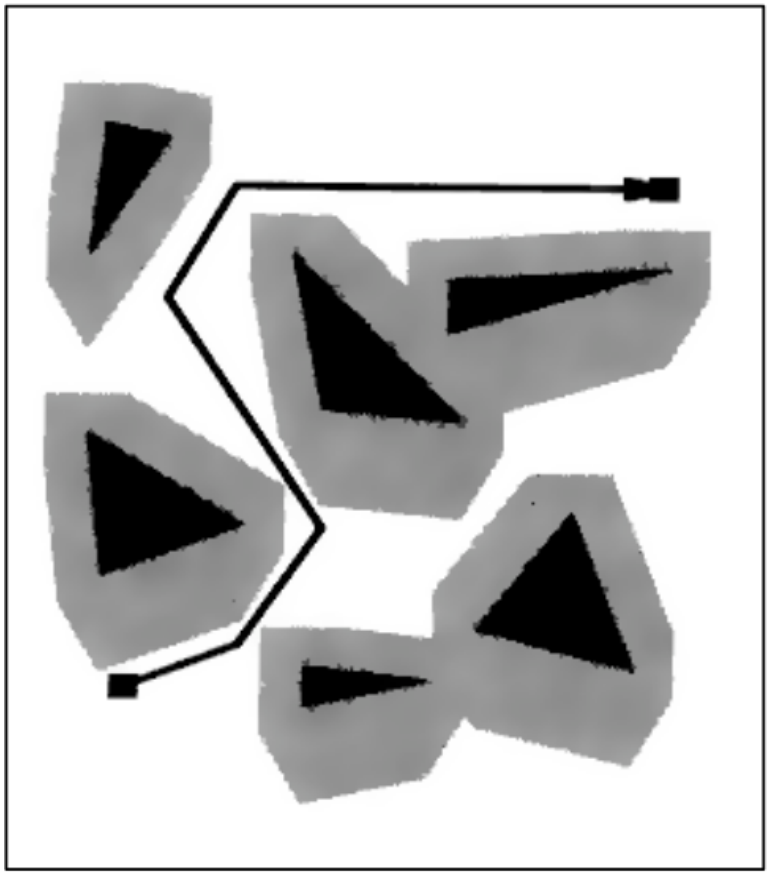
Planning in “configuration space”

Convert original planning problem into
a planning problem for a single point

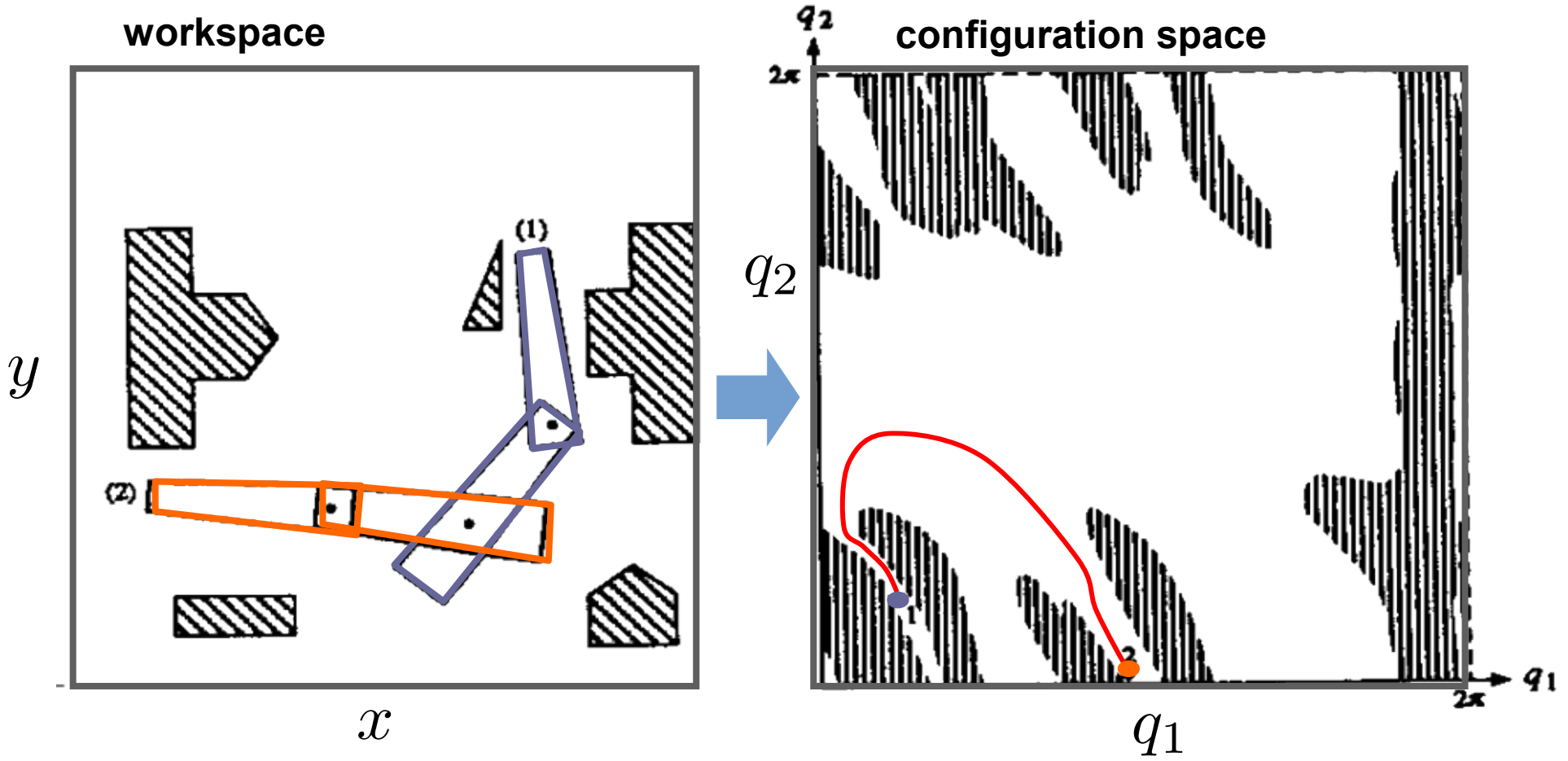
workspace



configuration space



Configuration space

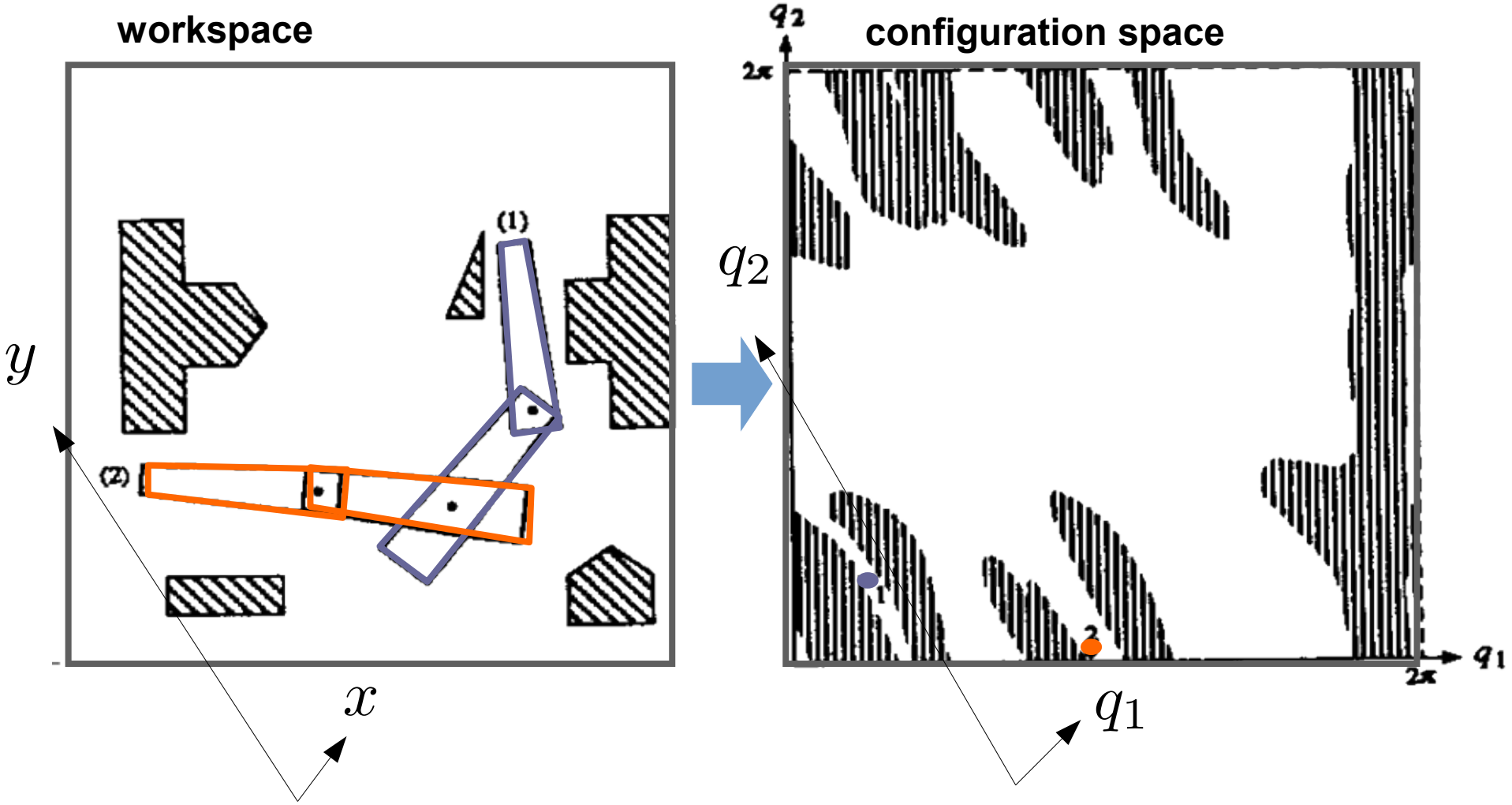


Original problem
– plan path for robot arm

Equivalent problem:
– plan path for a point

Configuration space

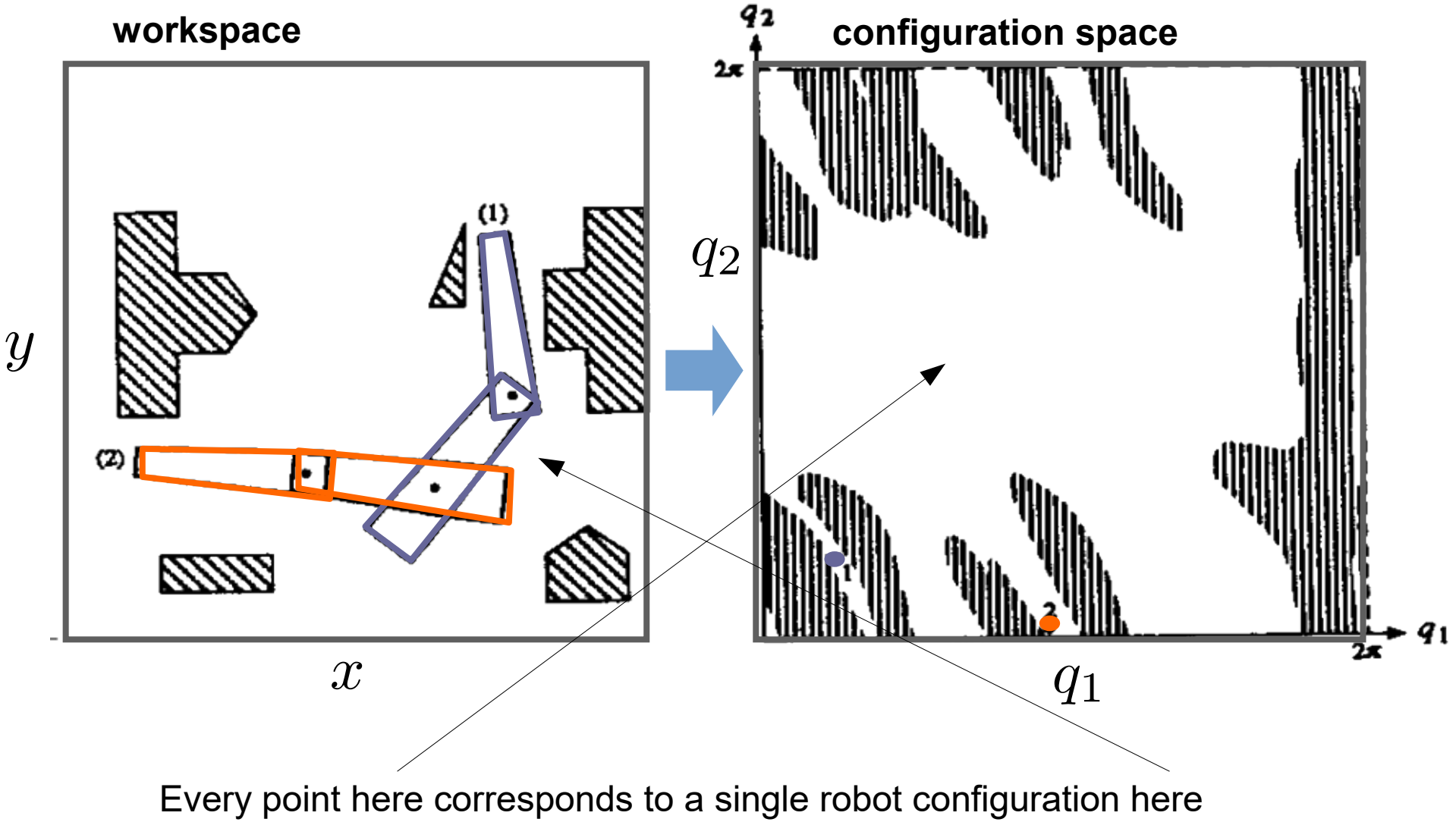
Notice the axes!



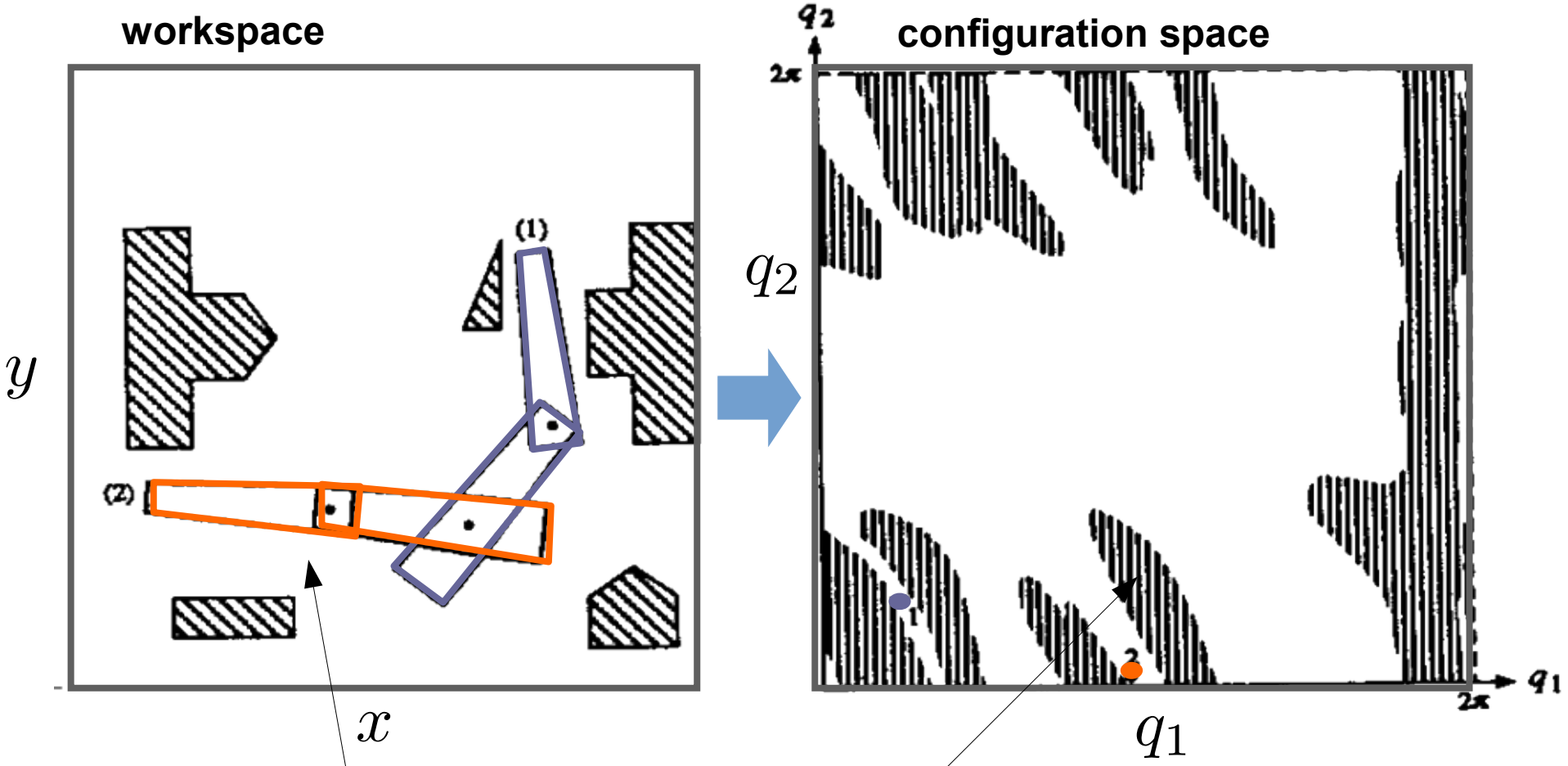
Cartesian space!

Joint angles!

Configuration space

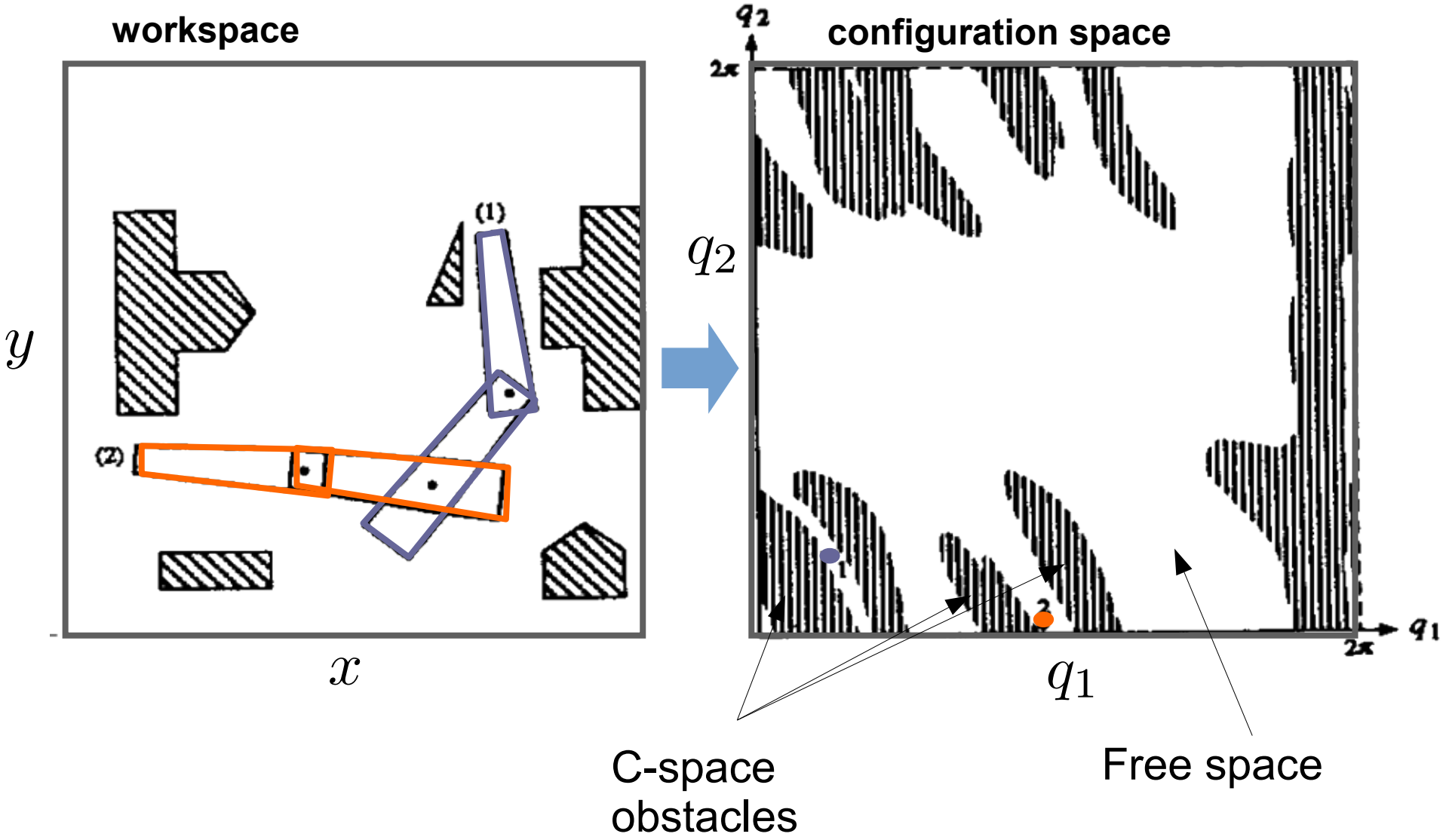


Configuration space

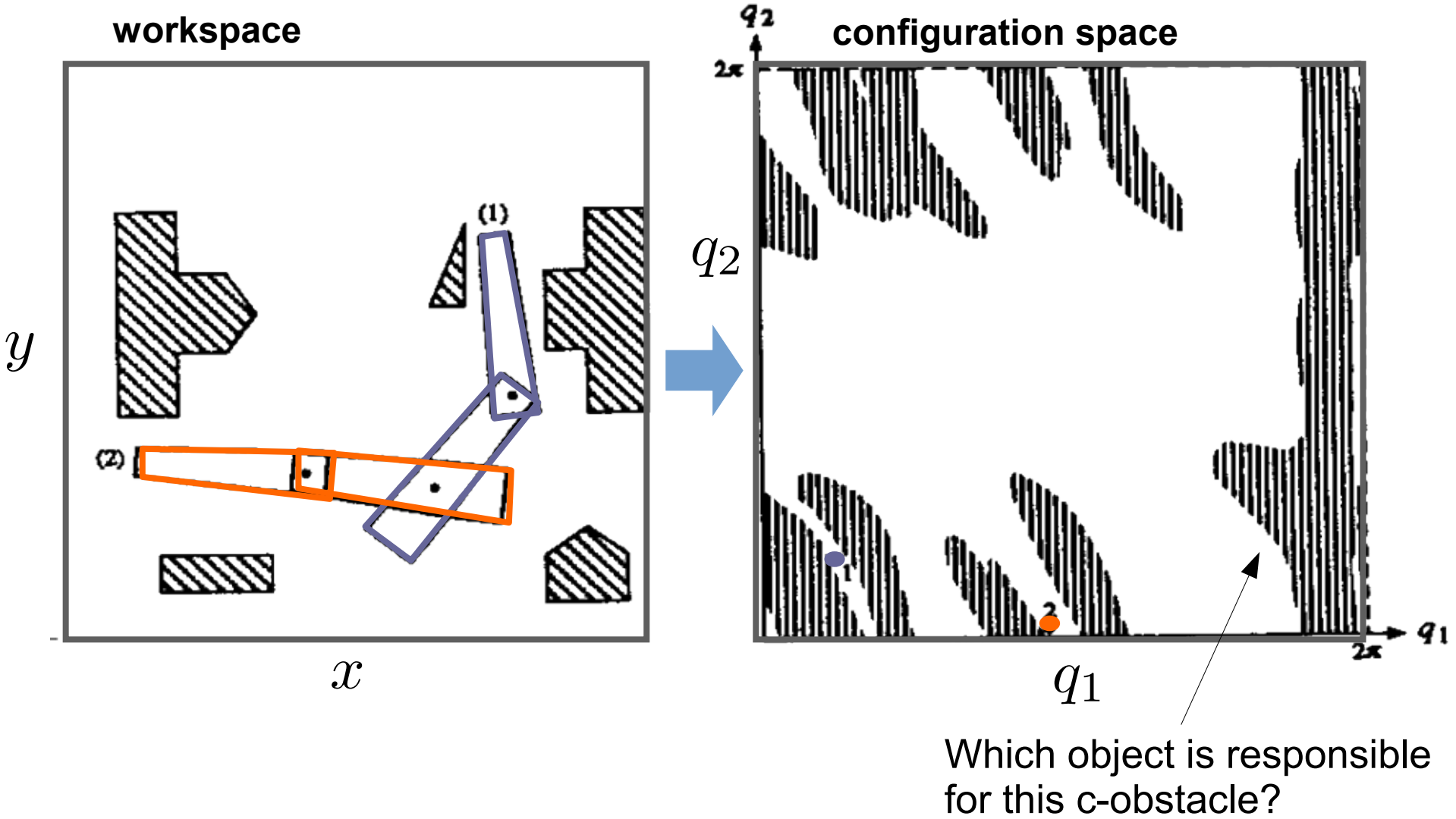


Every point that intersects an obstacle here corresponds to an arm configuration that intersects an obstacle

Configuration space

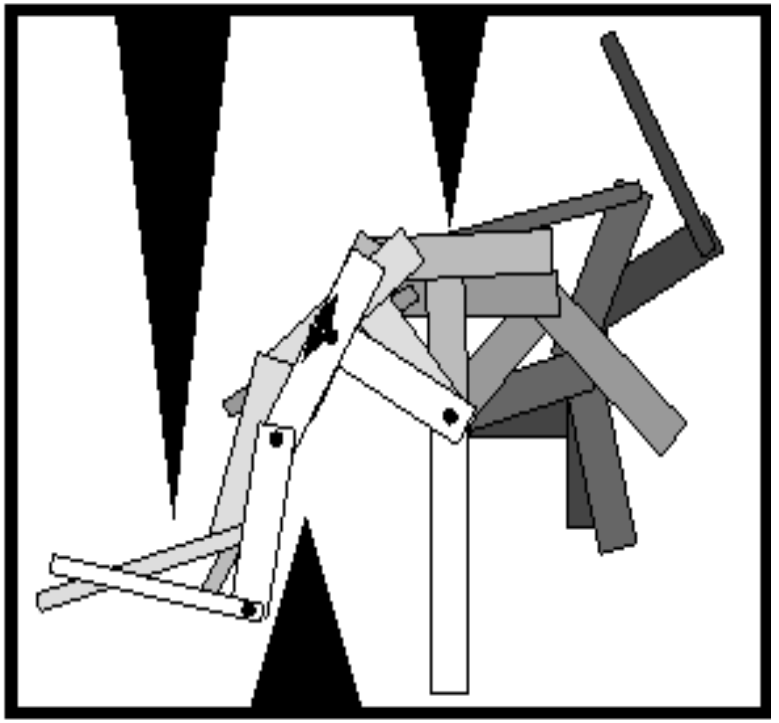


Configuration space



Configuration space

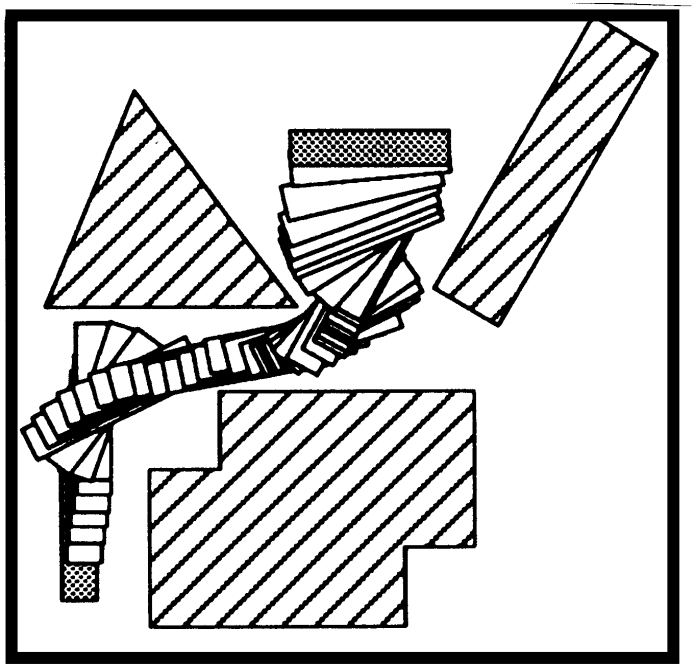
Dimension of a configuration space is
the minimum number of parameters needed
to specify the configuration of the robot completely
- i.e., the number of “degrees of freedom” (DOFs)



Dimension = 3

Configuration space

Dimension of a configuration space is
the minimum number of parameters needed
to specify the configuration of the robot completely
- i.e., the number of “degrees of freedom” (DOFs)



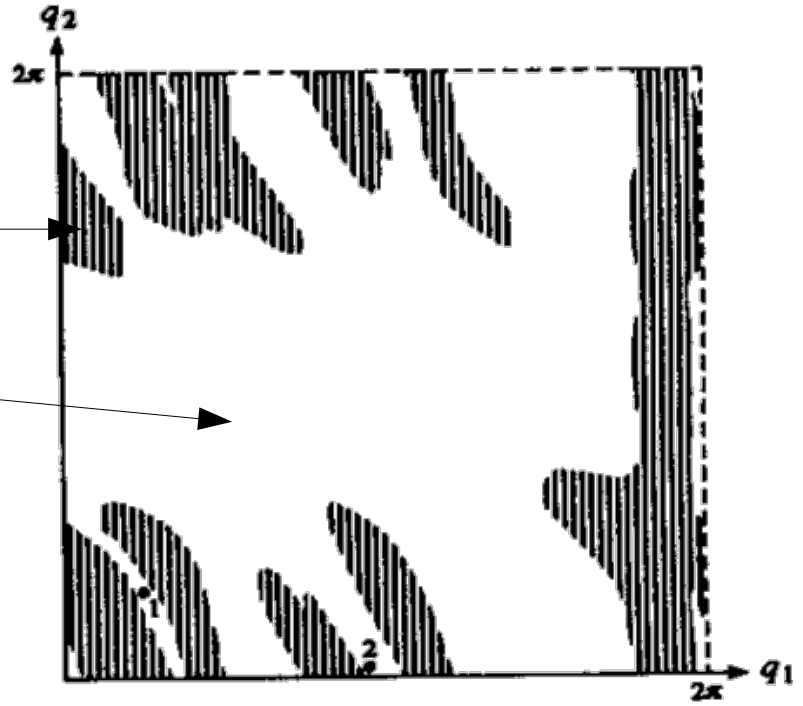
Dimension = ?

Formalization of the motion planning problem

Configuration space: $\mathcal{C}$

Obstacle space: $\mathcal{C}_{obs}$

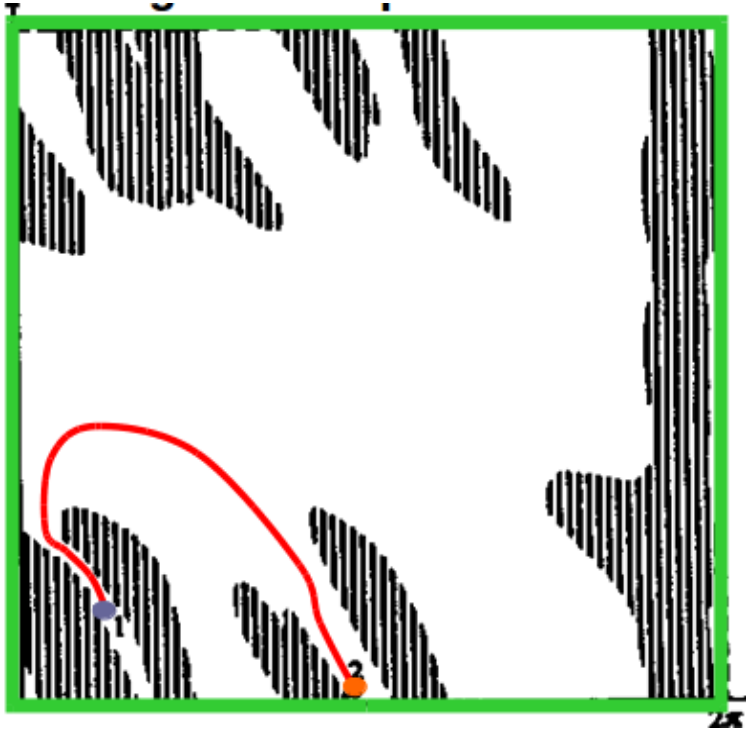
Free space: $\mathcal{C}_{free} = \mathcal{C} \setminus \mathcal{C}_{obs}$



Path: $\sigma : [0, 1] \rightarrow \mathcal{C}$ where σ must be continuous

Collision-free path: $\sigma(\tau) \in \mathcal{C}_{free}, \tau \in [0, 1]$

Formalization of the motion planning problem



Given:

– configuration space $\mathcal{C}$

– free space $\mathcal{C}_{free}$

– start state $x_{init} \in \mathcal{C}_{free}$

– goal region $X_{goal} \subset \mathcal{C}_{free}$

Find:

– a collision-free path σ such that $\sigma(0) = x_{init}$ and $\sigma(1) \in X_{goal}$

Methods for configuration-space motion planning

Good to know:

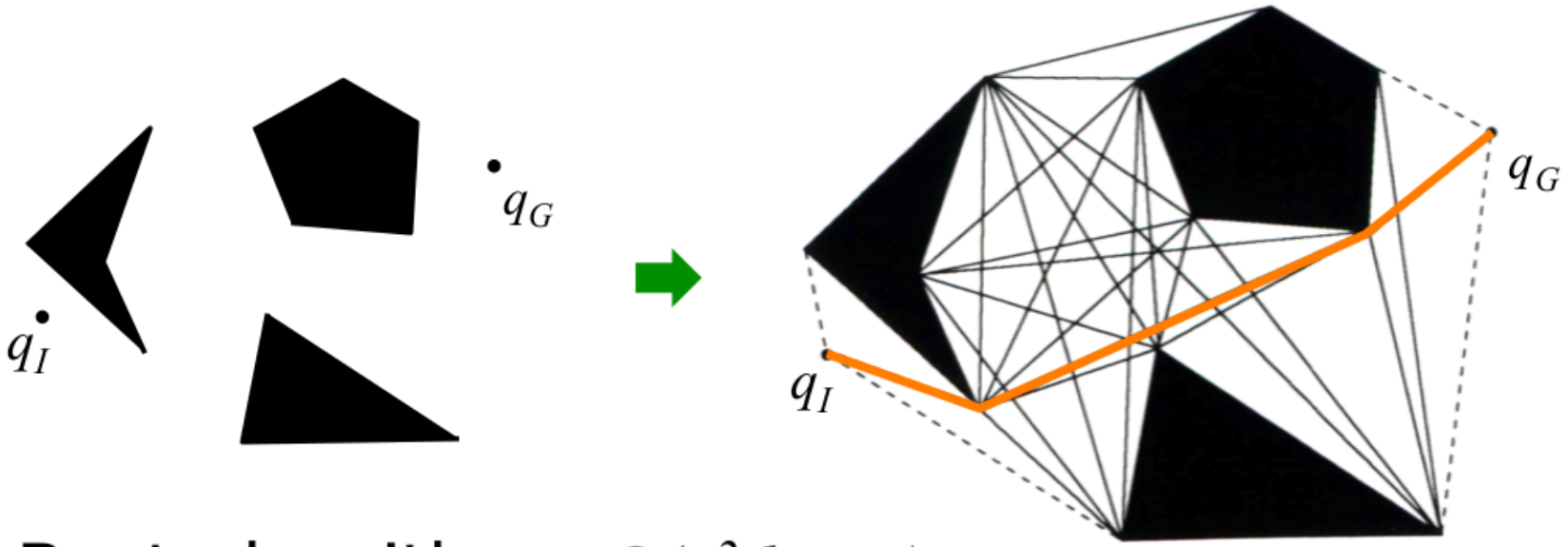
- Visibility graphs
- Voronoi diagrams
- Exact cell decomposition
- Approximate cell decomposition

Must know (next time):

- Sampling-based motion planning
  - Probabilistic roadmap (PRM)
  - Rapidly-exploring random tree (RRT)

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

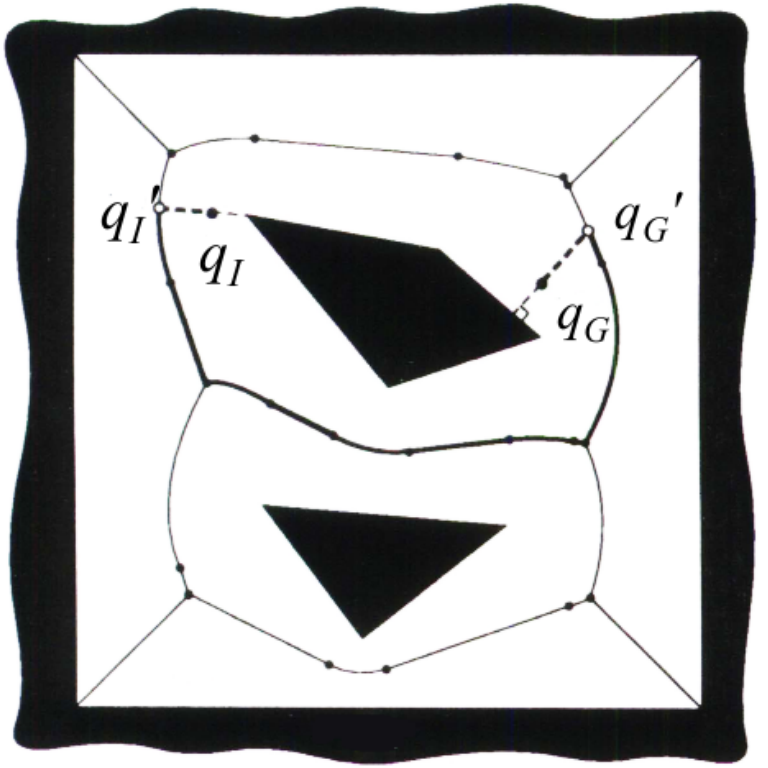


- Best algorithm: $O(n^2 \log n)$

n = num of obstacle vertices

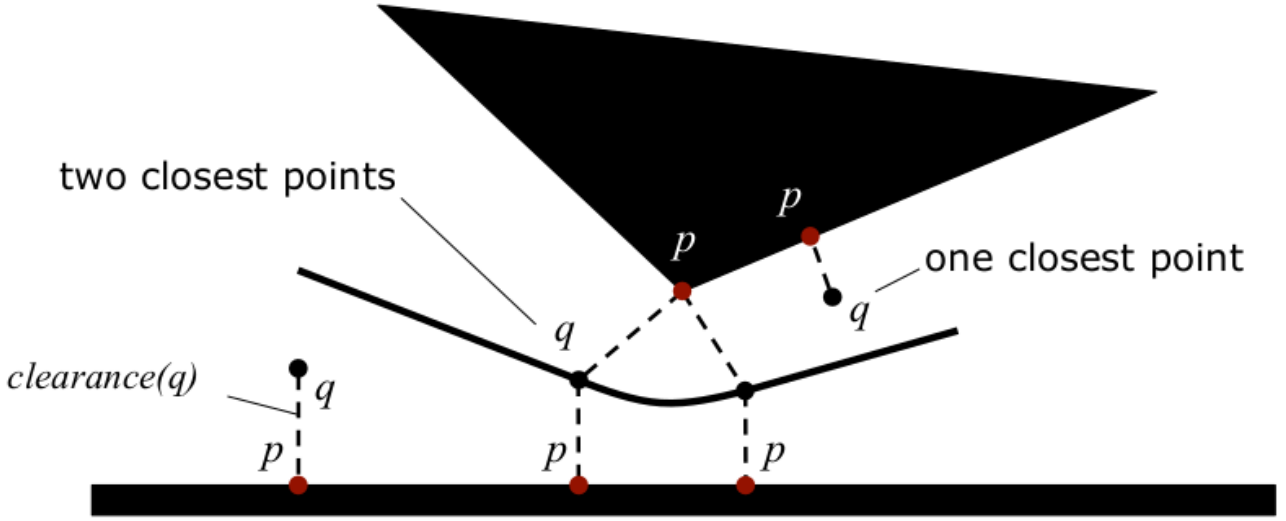
Method 2: Voronoi diagrams

- **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: Voronoi diagrams

- **Geometrically:**



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

Method 2: Voronoi diagrams

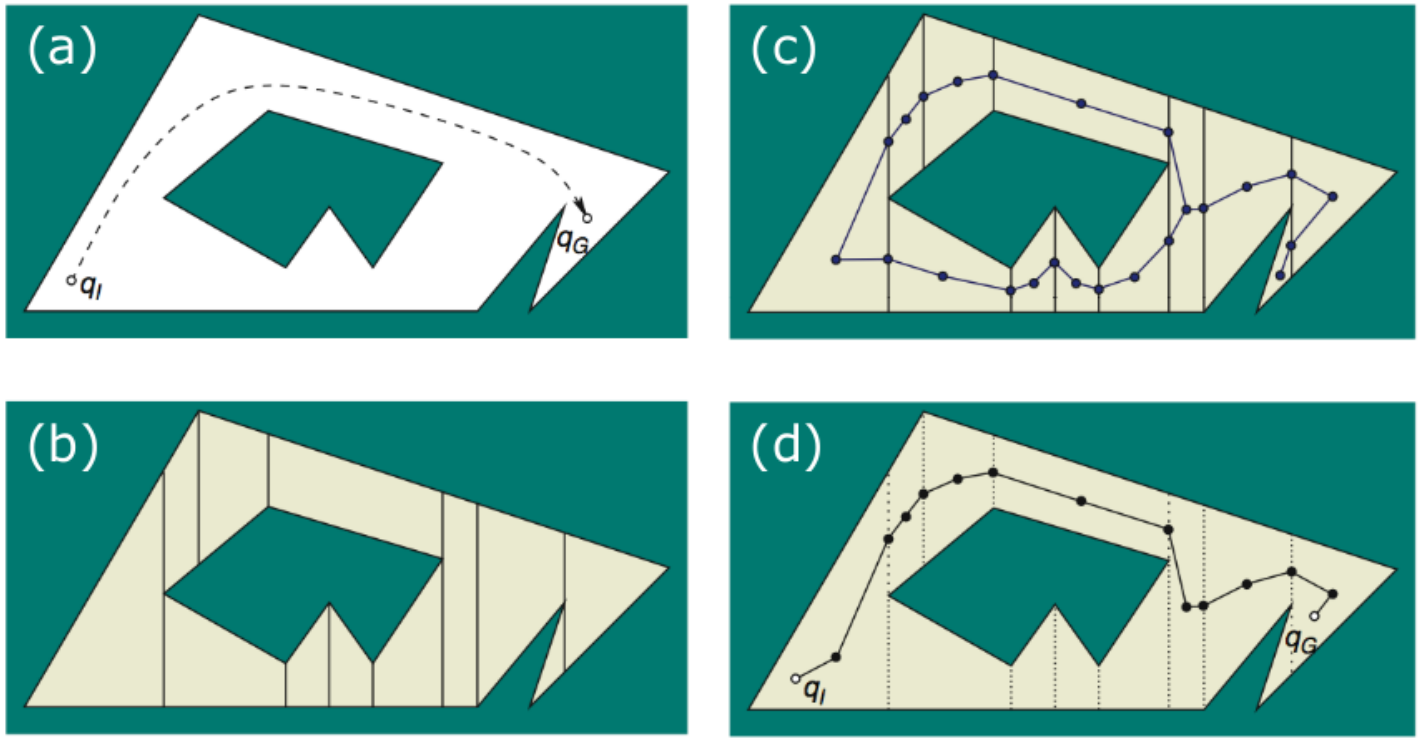
- Voronoi diagrams have been well studied for (reactive) **mobile robot** path planning
- Fast methods exist to compute and update the diagram in real-time for low-dim. C's
  - **Pros:** maximize clearance is a good idea for an uncertain robot
  - **Cons:** unnatural attraction to open space, suboptimal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

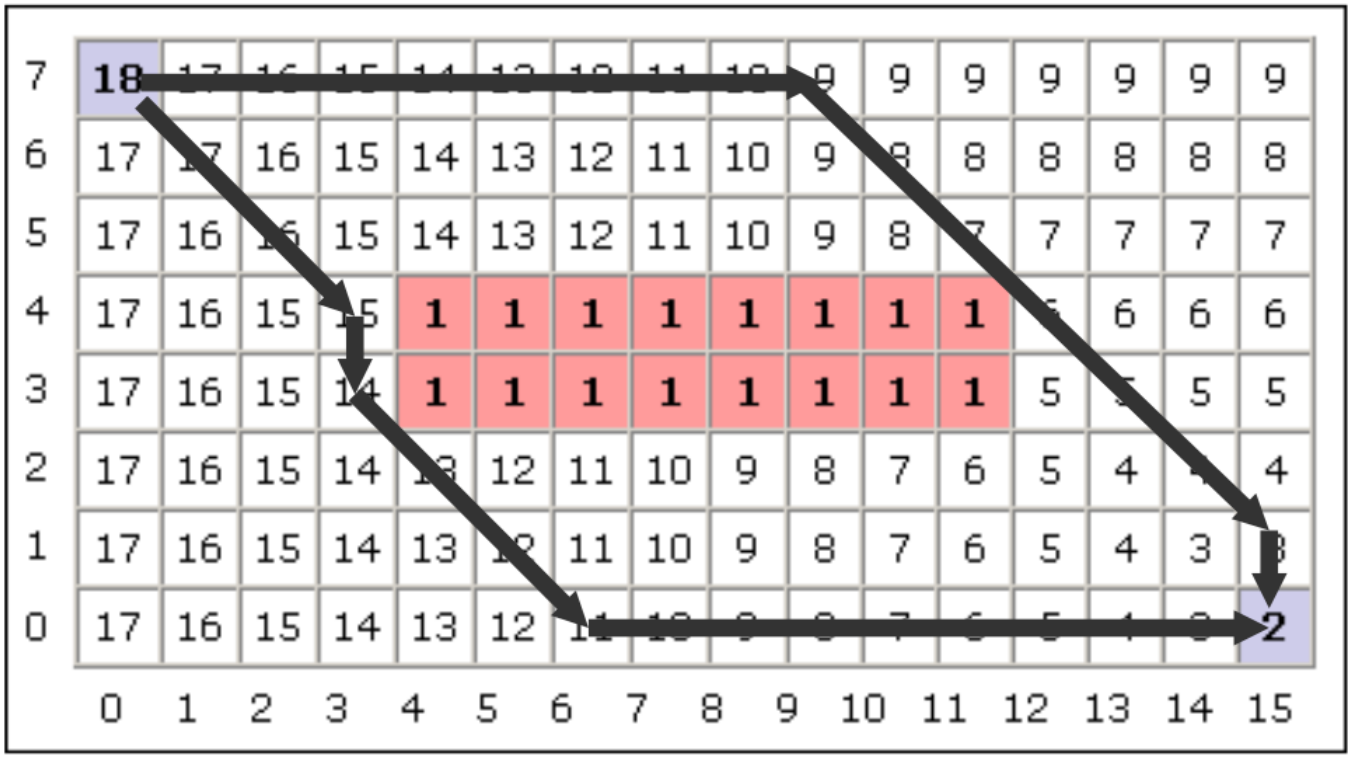
Method 3: Exact cell decomposition

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

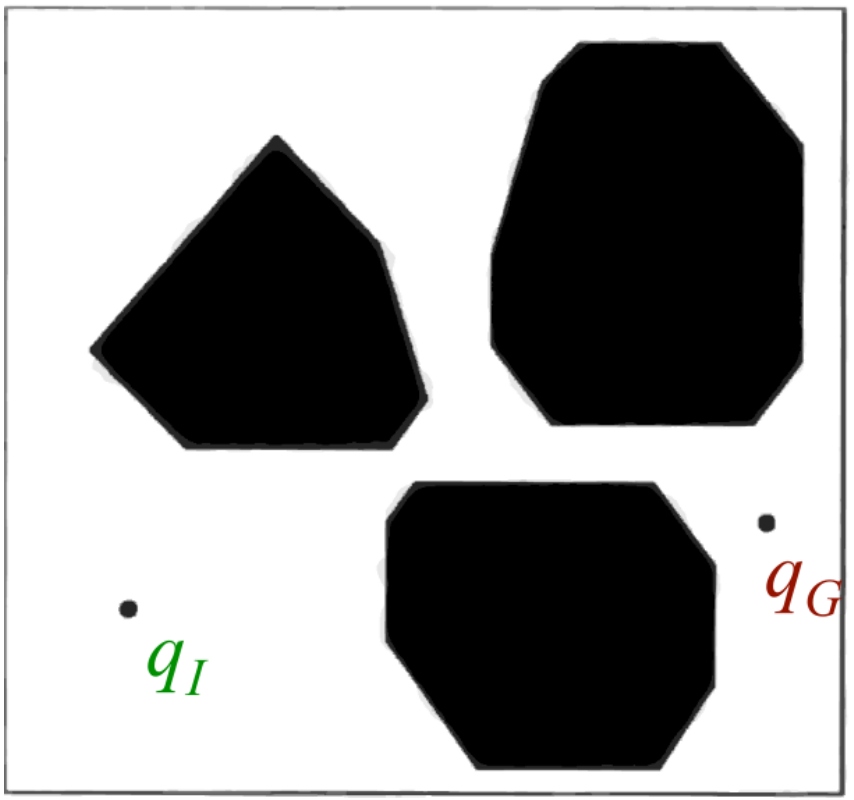


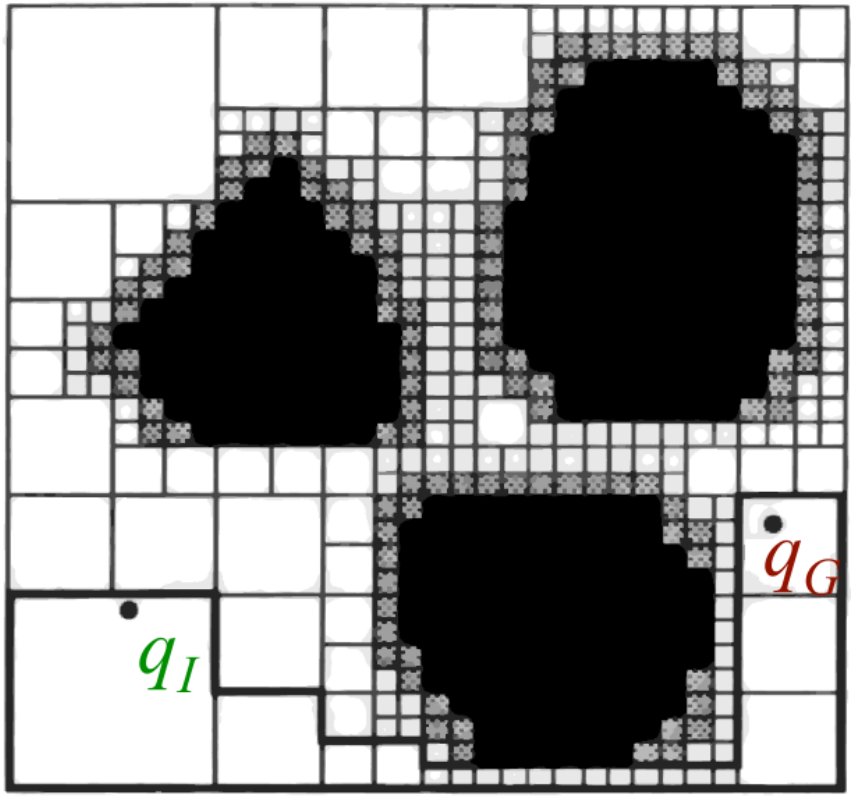
- Best known algorithm: $O(n \log n)$ where n is the number of vertices of C_{obs}

Method 4: Approximate cell decomposition (uniform cells)

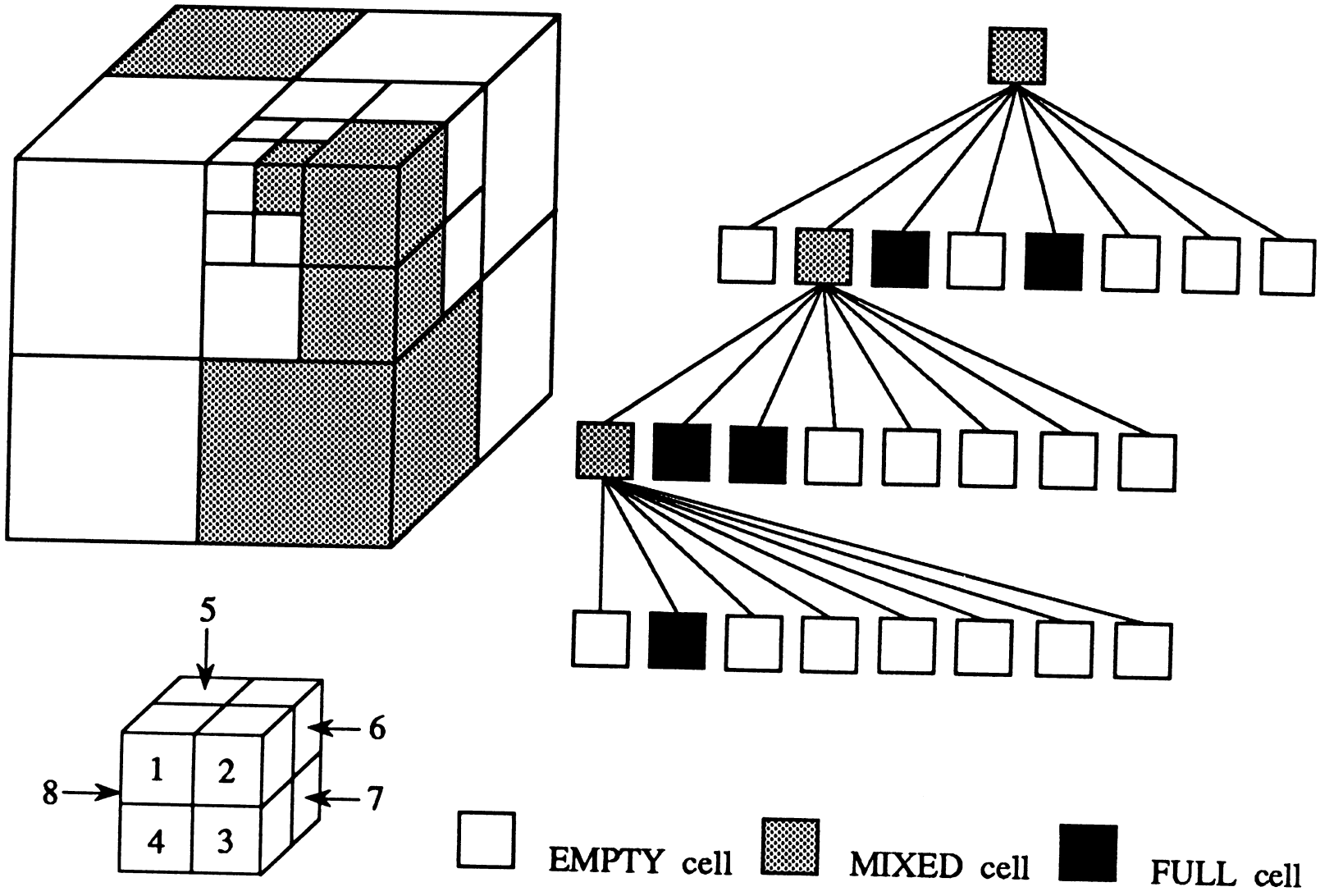


Method 4: Approximate cell decomposition (non-uniform cells: quadtree)

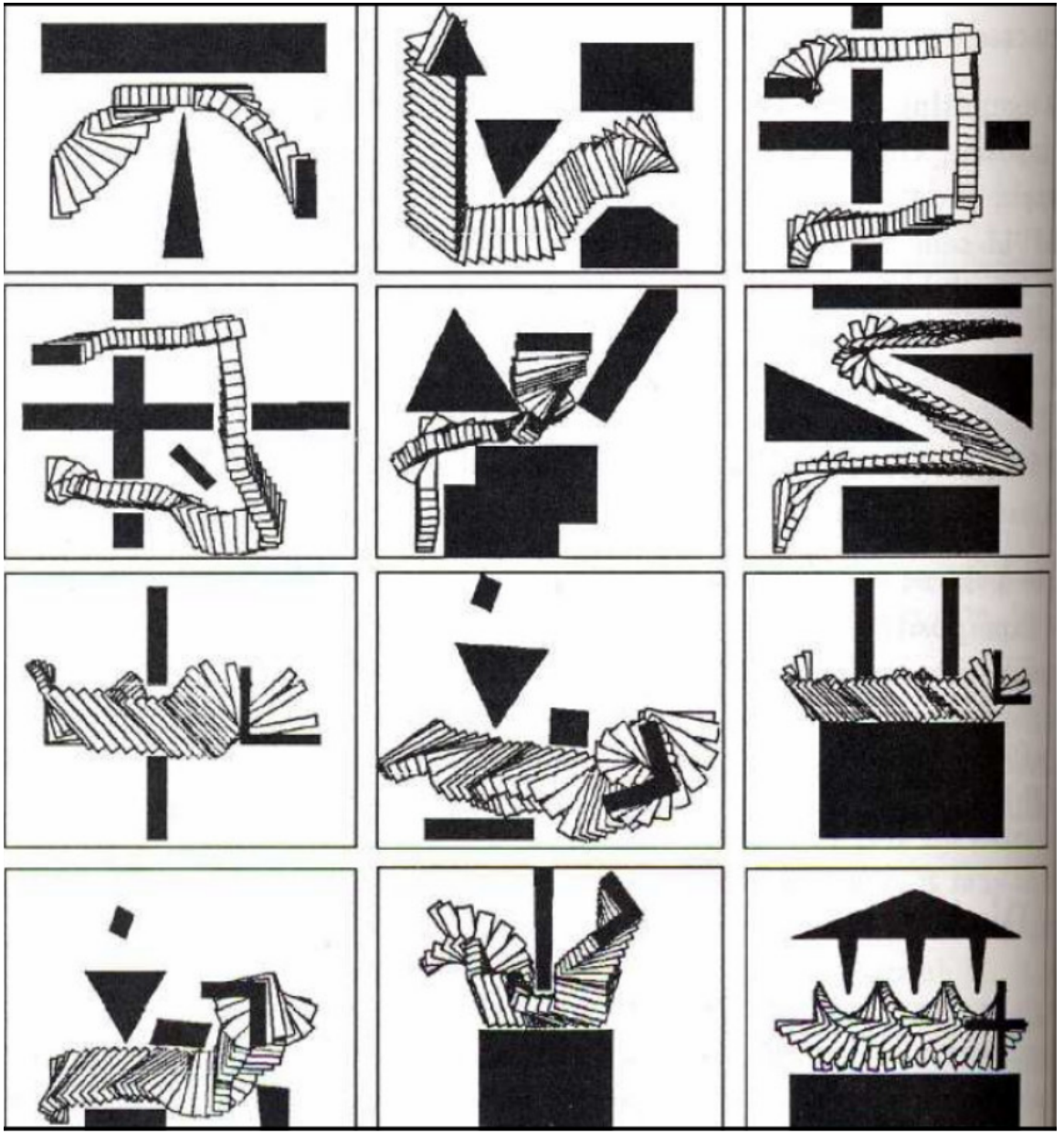




Method 4: Approximate cell decomposition (non-uniform cells: octree / octomap)



Method 4: Approximate cell decomposition (non-uniform cells: octree / octomap)



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**

Methods for configuration-space motion planning

Good to know:

- Visibility graphs
- Voronoi diagrams
- Exact cell decomposition
- Approximate cell decomposition

Unfortunately, most of the above methods
only work for a small number of obstacles
in low-dimensional configuration spaces (2-3 DOF)

Must know (next time):

- Sampling-based motion planning
  - Probabilistic roadmap (PRM)
  - Rapidly-exploring random tree (RRT)

Feedback

Piazza thread: 2/7 Lec 06 Feedback

Please post your answers to the following anonymously.

1. What did you like so far?

2. What was unclear?

3. Have you started Ex1 yet?
   If so, how are you finding it so far?

4. Any additional feedback / comments?