Motion Planning

Motion Planning

- Motion planning
  - Finding a robot motion from a start state to a goal state that avoids obstacles in the environments and satisfies other constraints, such as joint limits or torque limits

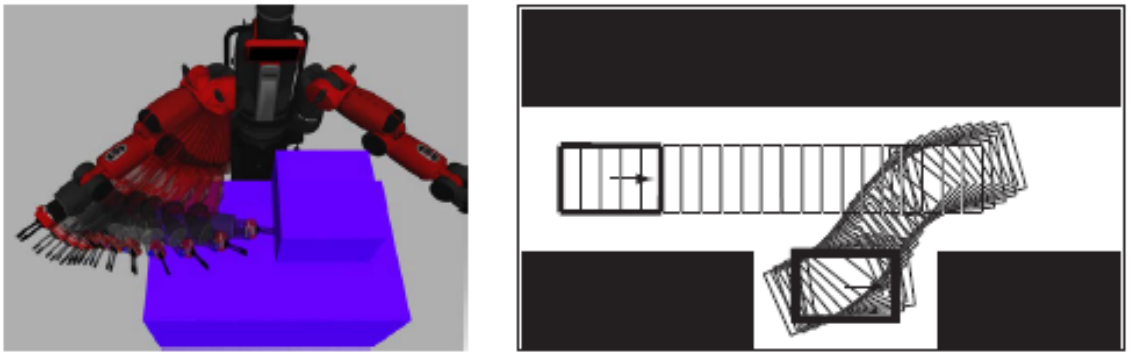


Figure 10.1: (Left) A robot arm executing an obstacle-avoiding motion plan. The motion plan was generated using MoveIt! [180] and visualized using rviz in ROS (the Robot Operating System). (Right) A car-like mobile robot executing parallel parking.

Configuration Space

- C-space
  - Motion planning 이 수행되는 configuration space

$$\mathcal{C} = \mathcal{C}_{\text{free}} \cup \mathcal{C}_{\text{obs}}$$

(ex) 2R Planar Arm

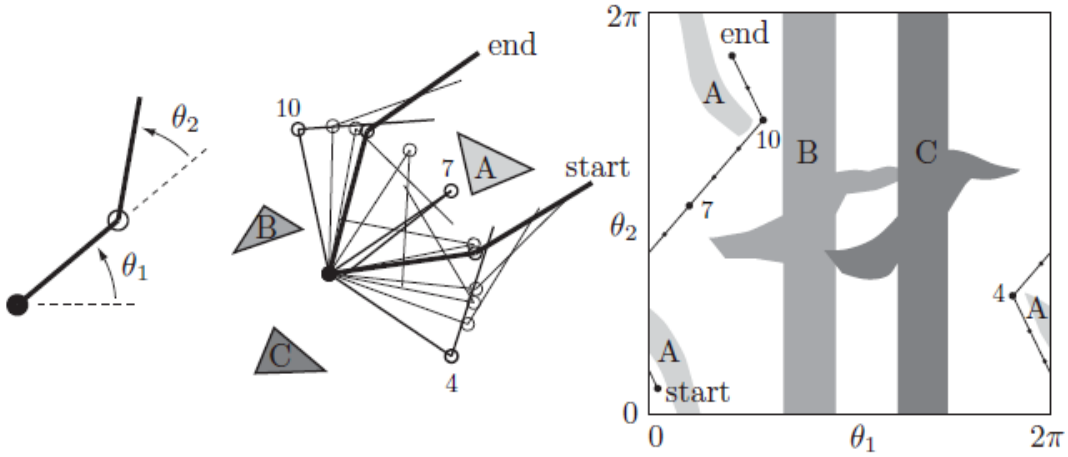


Figure 10.2: (Left) The joint angles of a 2R robot arm. (Middle) The arm navigating among obstacles A, B, and C. (Right) The same motion in C-space. Three intermediate points, 4, 7, and 10, along the path are labeled.

Configuration Space

(ex) Circular Mobile Robot

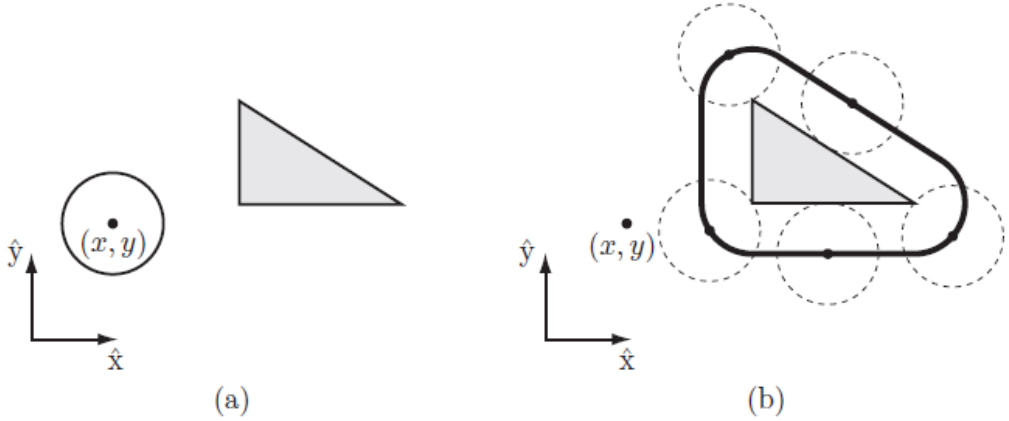
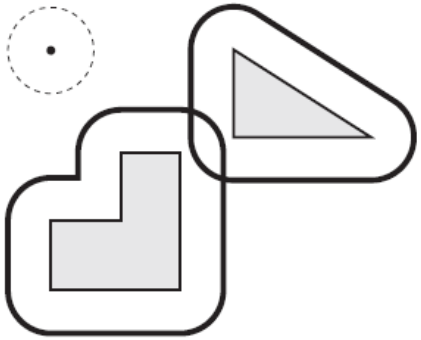


Figure 10.3: (a) A circular mobile robot (open circle) and a workspace obstacle (gray triangle). The configuration of the robot is represented by (x, y), the center of the robot. (b) In the C-space, the obstacle is "grown" by the radius of the robot and the robot is treated as a point. Any (x, y) configuration outside the bold line is collision-free.



Configuration Space

(ex) Polygon Mobile Robot that Translates

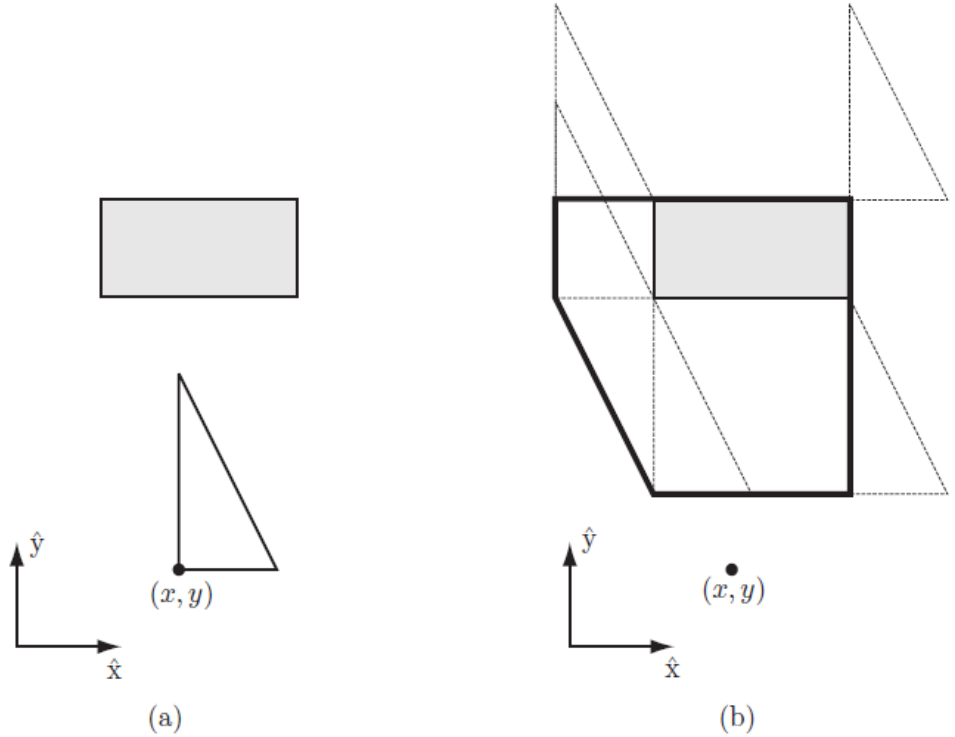


Figure 10.5: (a) The configuration of a triangular mobile robot, which can translate but not rotate, is represented by the (x, y) location of a reference point. Also shown is a workspace obstacle in gray. (b) The corresponding C-space obstacle (bold outline) is obtained by sliding the robot around the boundary of the obstacle and tracing the position of the reference point.

Configuration Space

(ex) Polygon Mobile Robot that Translates and Rotates

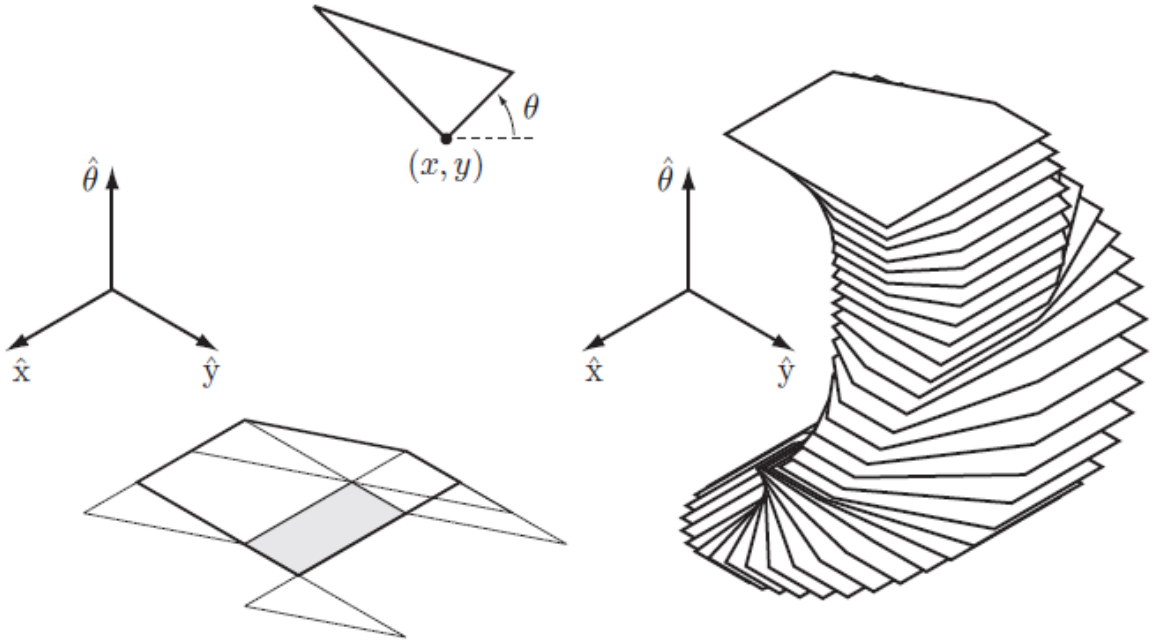


Figure 10.6: (Top) A triangular mobile robot that can both rotate and translate, represented by the configuration (x, y, θ). (Left) The C-space obstacle from Figure 10.5(b) when the robot is restricted to $\theta = 0$. (Right) The full three-dimensional C-space obstacle shown in slices at 10° increments.

Graph and Tree

- Graph
  - Nodes + Edges
  - Directed / Undirected
  - Weighted / Unweighted

- Tree
  - Graph with
    - no cycle
    - one parent node

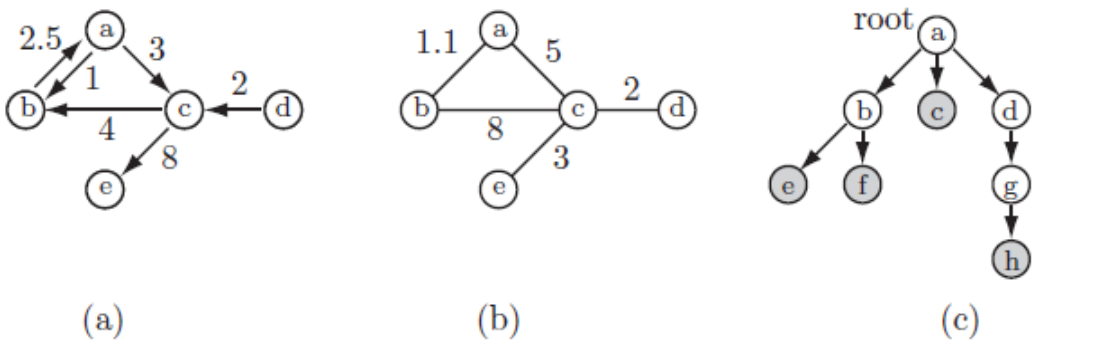


Figure 10.8: (a) A weighted digraph. (b) A weighted undirected graph. (c) A tree. The leaves are shaded gray.

Graph Search – A*

- A* algorithm
  - Effectively finds a minimum-cost path on a graph

- Data structure

  OPEN : 탐색할 노드 집합 (sorted list)
  CLOSED: 탐색이 끝난 노드 집합 (list)
  cost[node1, node2]: node1->node2 이동 시 cost (matrix)
  past_cost[node]: start node 부터 node 까지 소요되는 최소 cost (array)
  parent[node]: node 에 대하여 최소 cost 가 발생되는 선행 node (array)

- Estimated cost

  est_total_cost[node] = past_cost[node] + heuristic_cost_to_go[node]

  heuristic_cost_to_go[node]: node 부터 goal 까지 소요되는 cost 추정치

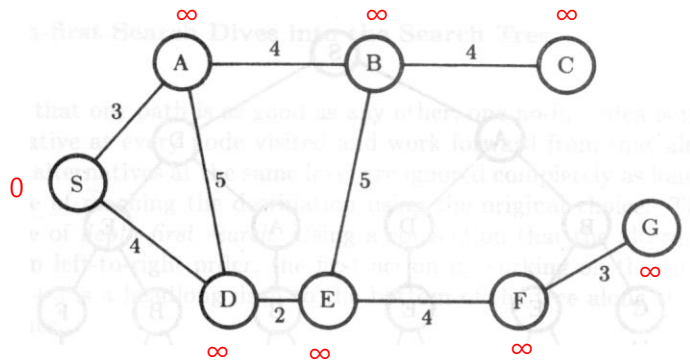
Graph Search – A*

Algorithm 10.1 A^* search.

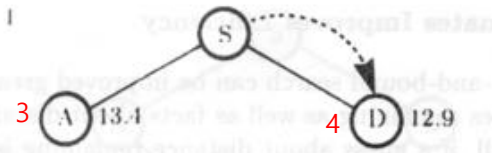
```
1: OPEN ← {1}
2: past_cost[1] ← 0, past_cost[node] ← infinity for node ∈ {2,...,N}
3: while OPEN is not empty do
4:    current ← first node in OPEN, remove from OPEN
5:    add current to CLOSED
6:    if current is in the goal set then
7:       return SUCCESS and the path to current
8:    end if
9:    for each nbr of current not in CLOSED do
10:      tentative_past_cost ← past_cost[current]+cost[current,nbr]
11:      if tentative_past_cost < past_cost[nbr] then
12:         past_cost[nbr] ← tentative_past_cost
13:         parent[nbr] ← current
14:         put (or move) nbr in sorted list OPEN according to
                 est_total_cost[nbr] ← past_cost[nbr] +
                        heuristic_cost_to_go(nbr)
15:      end if
16:   end for
17: end while
18: return FAILURE
```

nbr: neighborhood

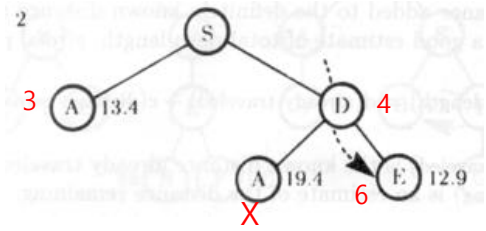
Graph Search – A*



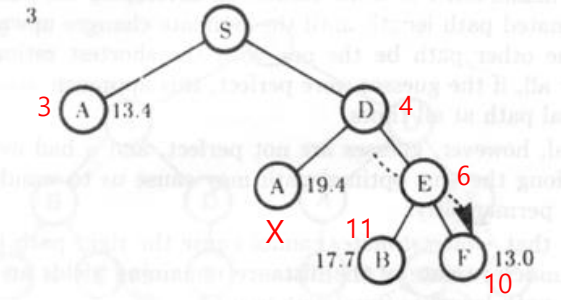
OPEN = {S}



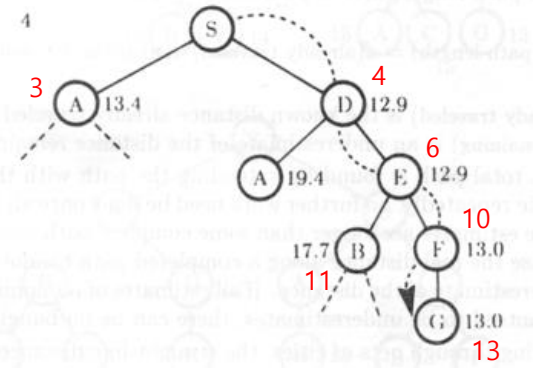
OPEN = {D, A} CLOSED = {S}



OPEN = {E, A} CLOSED = {S, D}



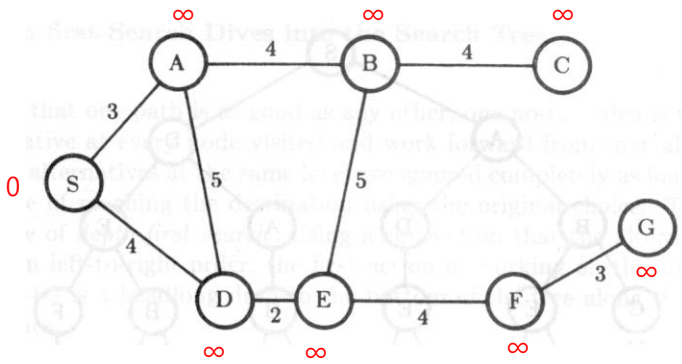
OPEN = {F, A, B} CLOSED = {S, D, E}



OPEN = {G, A, B} CLOSED = {S, D, E, F}

Graph Search-Dijkstra's Algorithm

- Dijkstra's algorithm
  - A* 알고리즘의 변형: heuristic_cost_to_go[node] = 0
  - Guarantees to find a minimum-cost path
  - Runs more slowly than A*



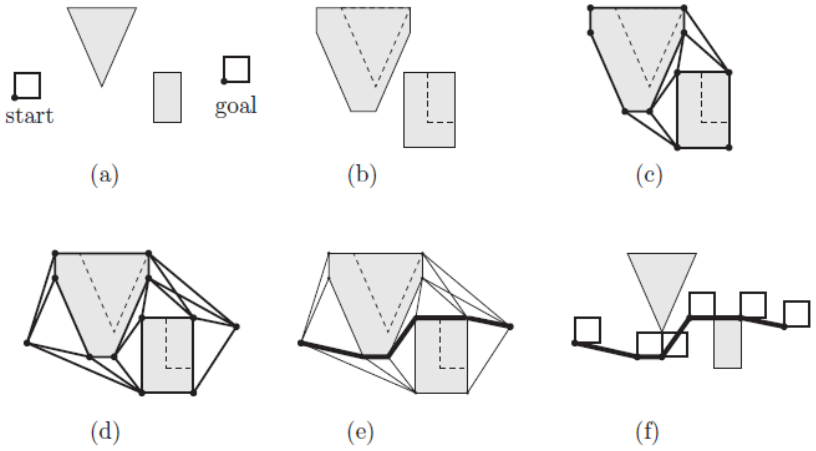
Motion Planning Methods

- Search space 형식에 따라 분류

1. Complete methods
   - Exact representations of geometry or topology of C_{free}
   - V graph
2. Grid methods
   - Discretize C_{free} into a grid and search the grid for a motion
   - Grid-based A*
3. Virtual potential fields
   - Creates virtual potential field (forces on the robot that pull it toward the goal and push it away from obstacles)
4. Sampling methods
   - Builds up a graph or tree by choosing a sample from the C space or state space
   - High d.o.f motion planning
   - RRT, PRM

Complete Methods

- Visibility graph (V graph)
  - C-space 에서 시작점, 목표점, 장애물의 꼭지점으로 구성된 graph
    - Nodes: vertices of C-obstacle
    - Edges: lines between nodes that can "see" each other
      (line segment between vertices dose not intersect an obstacle)
- Graph search
  - Undirected weight graph
    - Weight = Euclidian distance
  - A* search algorithm
- 특징
  - 꼭지점을 지나가는 경로 생성
    ⇒ 충돌주의



Grid Methods

- Grid representation
  - C-space 를 grid 로 분할
  - 4-connected vs. 8-connected
  - Euclidean distance vs. Manhattan distance

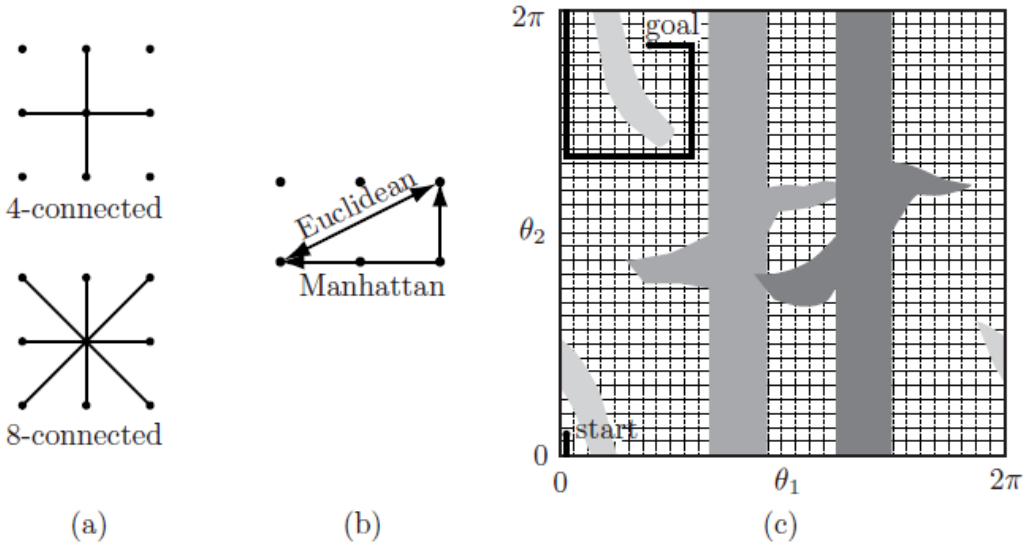


Figure 10.10: (a) A 4-connected grid point and an 8-connected grid point for a space $n = 2$. (b) Grid points spaced at unit intervals. The Euclidean distance between the two points indicated is $\sqrt{5}$ while the Manhattan distance is 3. (c) A grid representation of the C-space and a minimum-length Manhattan-distance path for the problem of Figure 10.2.

Grid Methods

- Wavefront Planner
  - Score of goal: 0
  - Scores of collision-free-neighbors: +1
  - Breadth-first search

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

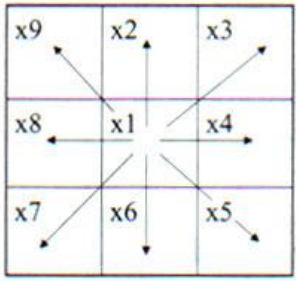
Grid Methods

- A* grid-based path planner

6	h =6 f = b=()	h =5 f = b=()	h =4 f = b=()	h =3 f = b=()	h =2 f = b=()	h =1 f = b=()	h =0 f = b=()
5	h =6.4 f = b=()	h =5.4 f = b=()	h =4.4 f = b=()	h =3.4 f = b=()	h =2.4 f = b=()	h =1.4 f = b=()	h =1 f = b=()
4	h =6.8 f = b=()	h =5.8 f = b=()	h =4.8 f = b=()	h =3.8 f = b=()	h =2.8 f = b=()	h =2.4 f = b=()	h =2 f = b=()
3	h =7.2 f = b=()	h =6.2 f = b=()	h =5.2 f = b=()	h =4.2 f = b=()	h =3.8 f = b=()	h =3.4 f = b=()	h =3 f = b=()
2	h =7.6 f = b=()	h =6.6 f = b=()	h =5.6 f = b=()	h =5.2 f = b=()	h =4.8 f = b=()	h =4.4 f = b=()	h =4 f = b=()
1	h =8.0 f = b=()	h =7.0 f = b=()	h =6.6 f = b=()	h=6.2 f= b=()	h =5.8 f = b=()	h =5.4 f = b=()	h =5 f = b=()
r/c	1	2	3	4	5	6	7

Goal

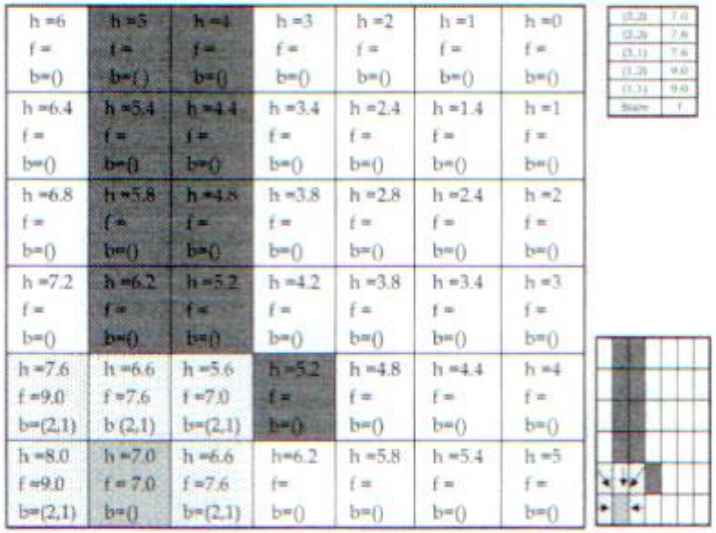
Start

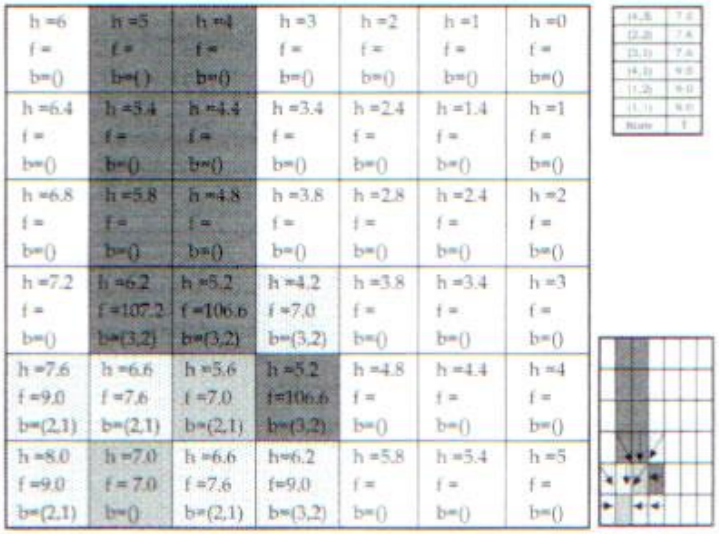


c(x1, x2) = 1
c(x1, x9) = 1.4

c(x1, x8) = 10000, if x8 is in obstacle, x1 is a free cell

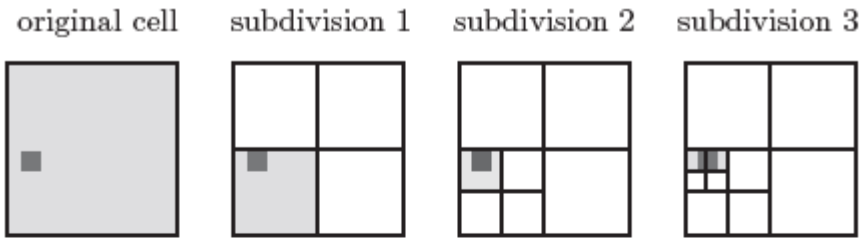
c(x1,x9) = 10000.4, if x9 is in obstacle, x1 is a free cell





Grid Methods

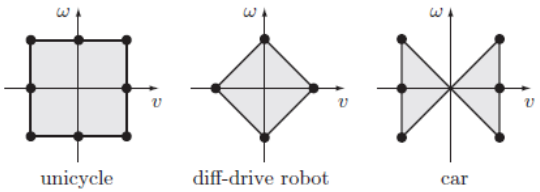
- Multi-resolution grid representation
  - Reduce the computational complexity
  - Subdivided by quad-tree



Fixed-grid: 64 cells
Mult-resolution grid: 10 cells

Grid Methods

- Grid-based path planning for a wheeled mobile robot
  - Discretizations of control sets



v : linear velocity
ω : angular velocity

Algorithm 10.2 Grid-based Dijkstra planner for a wheeled mobile robot.

```
1: OPEN ← {q_start}
2: past_cost[q_start] ← 0
3: counter ← 1
4: while OPEN is not empty and counter < MAXCOUNT do
5:    current ← first node in OPEN, remove from OPEN
6:    if current is in the goal set then
7:       return SUCCESS and the path to current
8:    end if
9:    if current is not in a previously occupied C-space grid cell then
10:      mark grid cell occupied
11:      counter ← counter + 1
12:      for each control in the discrete control set do
13:         integrate control forward a short time Δt from current to q_new
14:         if the path to q_new is collision-free then
15:            compute cost of the path to q_new
16:            place q_new in OPEN, sorted by cost
17:            parent[q_new] ← current
18:         end if
19:      end for
20:   end if
21: end while
22: return FAILURE
```

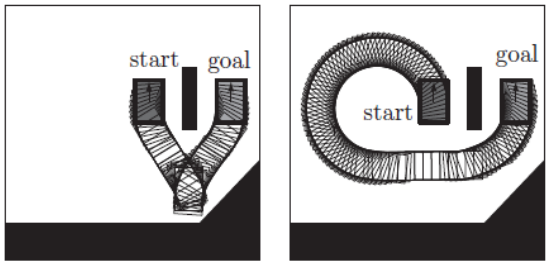


Figure 10.15: (Left) A minimum-cost path for a car-like robot where each action has identical cost, favoring a short path. (Right) A minimum-cost path where reversals are penalized. Penalizing reversals requires a modification to Algorithm 10.2.

Virtual Potential Fields

- Potential energy from goal $\mathcal{P}_{\text{goal}}(q) = \frac{1}{2}(q - q_{\text{goal}})^{\mathrm{T}} K (q - q_{\text{goal}})$
  - Attractive force from goal $F_{\text{goal}}(q) = -\frac{\partial \mathcal{P}_{\text{goal}}}{\partial q} = K(q_{\text{goal}} - q)$
- Repulsive potential energy from obstacle β $\mathcal{P}_{\mathcal{B}}(q) = \frac{k}{2d^2(q, \mathcal{B})},$
  - Repulsive force from obstacle $F_{\mathcal{B}}(q) = -\frac{\partial \mathcal{P}_{\mathcal{B}}}{\partial q} = \frac{k}{d^3(q, \mathcal{B})} \frac{\partial d}{\partial q}.$
- Total potential & force

$$\mathcal{P}(q) = \mathcal{P}_{\text{goal}}(q) + \sum \mathcal{P}_{\mathcal{B}_i}(q)$$
$$F(q) = F_{\text{goal}}(q) + \sum_i F_{\mathcal{B}_i}(q)$$

- Velocity command

$$\dot{q} = F(q)$$

Virtual Potential Fields

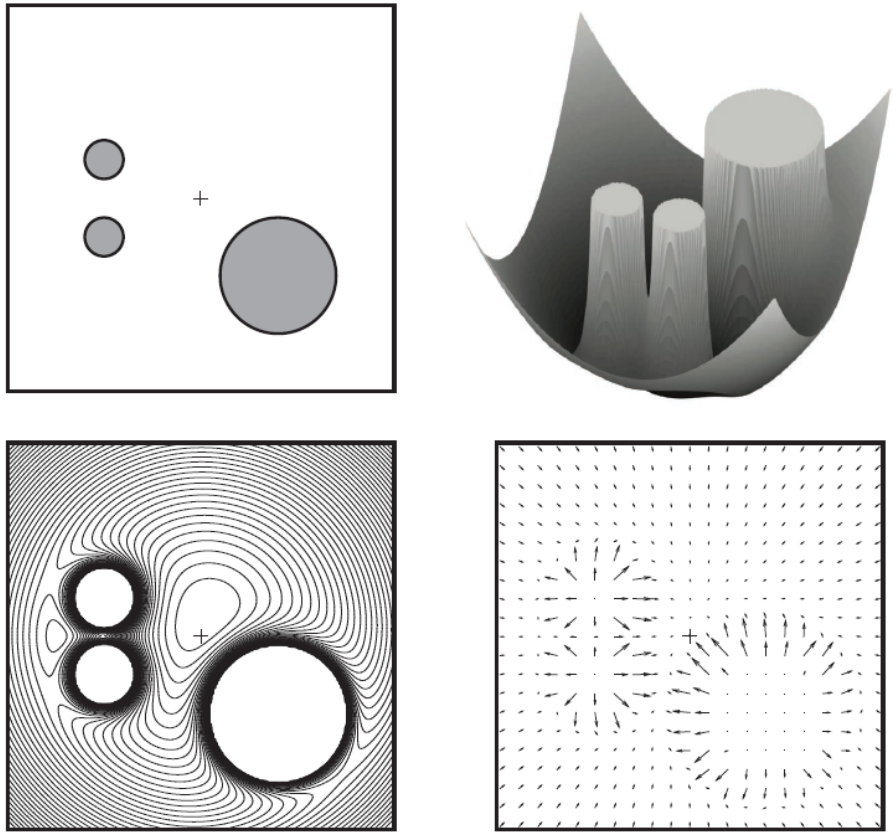


Figure 10.21: (Top left) Three obstacles and a goal point, marked with a +, in $\mathbb{R}^2$. (Top right) The potential function summing the bowl-shaped potential pulling the robot to the goal with the repulsive potentials of the three obstacles. The potential function saturates at a specified maximum value. (Bottom left) A contour plot of the potential function, showing the global minimum, a local minimum, and four saddles: between each obstacle and the boundary of the workspace, and between the two small obstacles. (Bottom right) Forces induced by the potential function.

global minimum
local minima

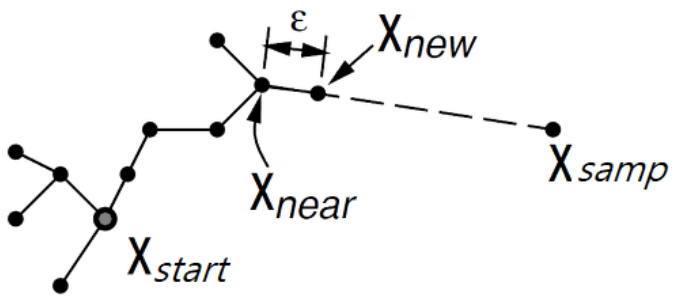
Sampling Methods

- RRT algorithm
  - Rapidly exploring random trees
  - Single-query planning in C-space or state space
  - Good for complex motion constraints or high d.o.f systems
  - Not optimal motion

Algorithm 10.3 RRT algorithm.

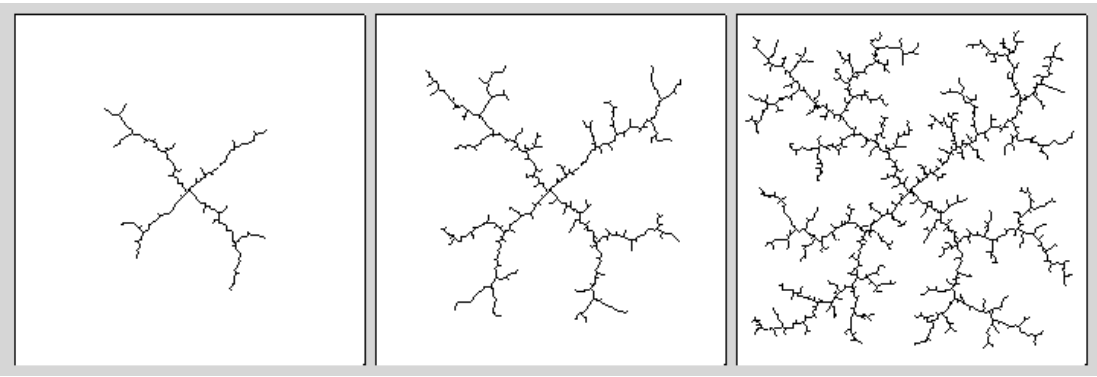
```
1: initialize search tree T with x_start
2: while T is less than the maximum tree size do
3:    x_samp ← sample from X
4:    x_nearest ← nearest node in T to x_samp
5:    employ a local planner to find a motion from x_nearest to x_new in
          the direction of x_samp
6:    if the motion is collision-free then
7:       add x_new to T with an edge from x_nearest to x_new
8:       if x_new is in X_goal then
9:          return SUCCESS and the motion to x_new
10:      end if
11:   end if
12: end while
13: return FAILURE
```

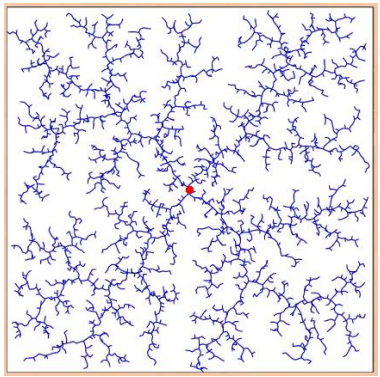
x : state
(ex) configuration, velocity, ...



Sampling Methods

- Tree expansion

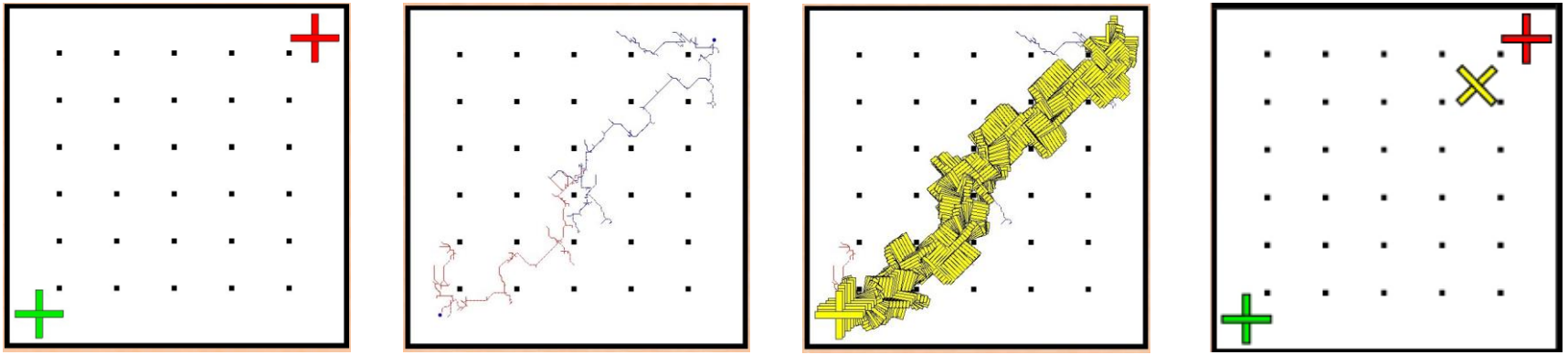




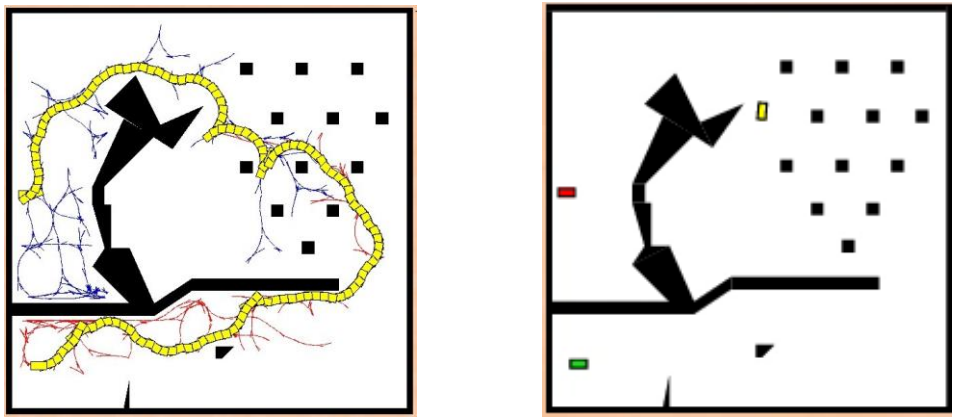
- Local path planner
  - Straight-line planner: system without motion constraints
  - Discretized controls planner: system with motion constraints
    - Determine x_{new} from $\dot{x} = f(x, u)$ and Δt
  - Wheeled robot planner: Reeds-Shepp curves

Sampling Methods

(Ex) Rigid robot



(Ex) Car-like robot



Sampling Methods

- RRT*
  - Continually rewires the search tree to ensure that it always encodes the shortest path from x_{start} to each node in the tree.

Algorithm 10.3 RRT algorithm.

```
1: initialize search tree T with x_start
2: while T is less than the maximum tree size do
3:    x_samp ← sample from X
4:    x_nearest ← nearest node in T to x_samp
5:    employ a local planner to find a motion from x_nearest to x_new in
          the direction of x_samp
6:    if the motion is collision-free then
7:       add x_new to T with an edge from x_nearest to x_new
8:       if x_new is in X_goal then
9:          return SUCCESS and the motion to x_new
10:      end if
11:   end if
12: end while
13: return FAILURE
```

Add a node after testing of all the nodes $x \in \chi_{near}$
1) collision-free
2) Minimizes the total cost of the path from x_{start}

Sampling Methods

- RRT*

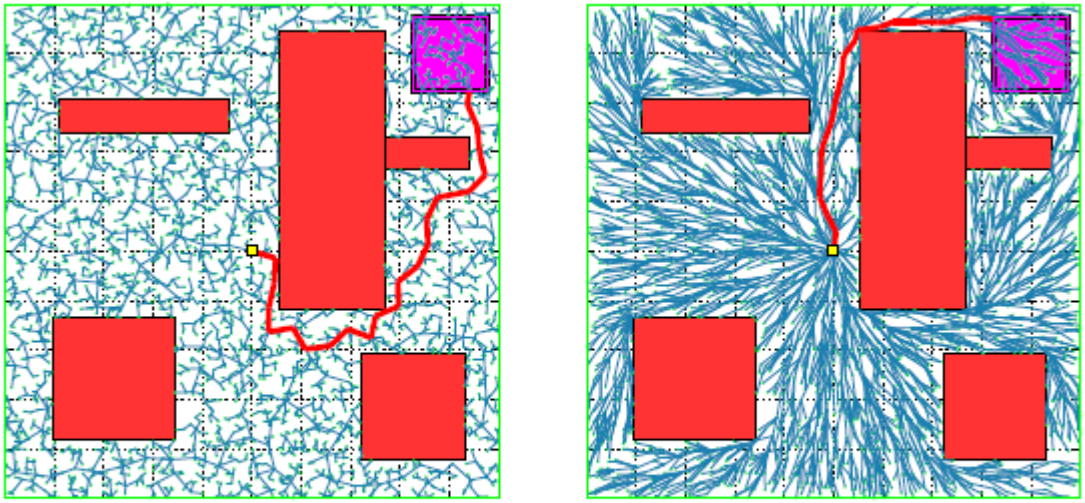


Figure 10.19: (Left) The tree generated by an RRT after 5,000 nodes. The goal region is the square at the top right corner, and the shortest path is indicated. (Right) The tree generated by RRT* after 5,000 nodes. Figure from [67] used with permission.