ETH Zürich
Chair of Software Engineering

Robotics Programming Laboratory

Bertrand Meyer
Jiwon Shin

Lecture 9: Path Planning

Path planning

Getting to Zurich HB from RZ building

- Tram 6, 7 to Bahnhofstrasse/HB
- Tram 10 to Bahnhofplatz/HB
- Walk down on Weinbergstrasse to Central then to HB
- Walk down on Leonhard-Treppe to Walcheplatz to Walchebrücke to HB
- Bike down on Weinbergstrasse
- ...

Each path offers different cost in terms of

- Time
- Convenience
- Crowdedness
- Ease
- ...

Path planning

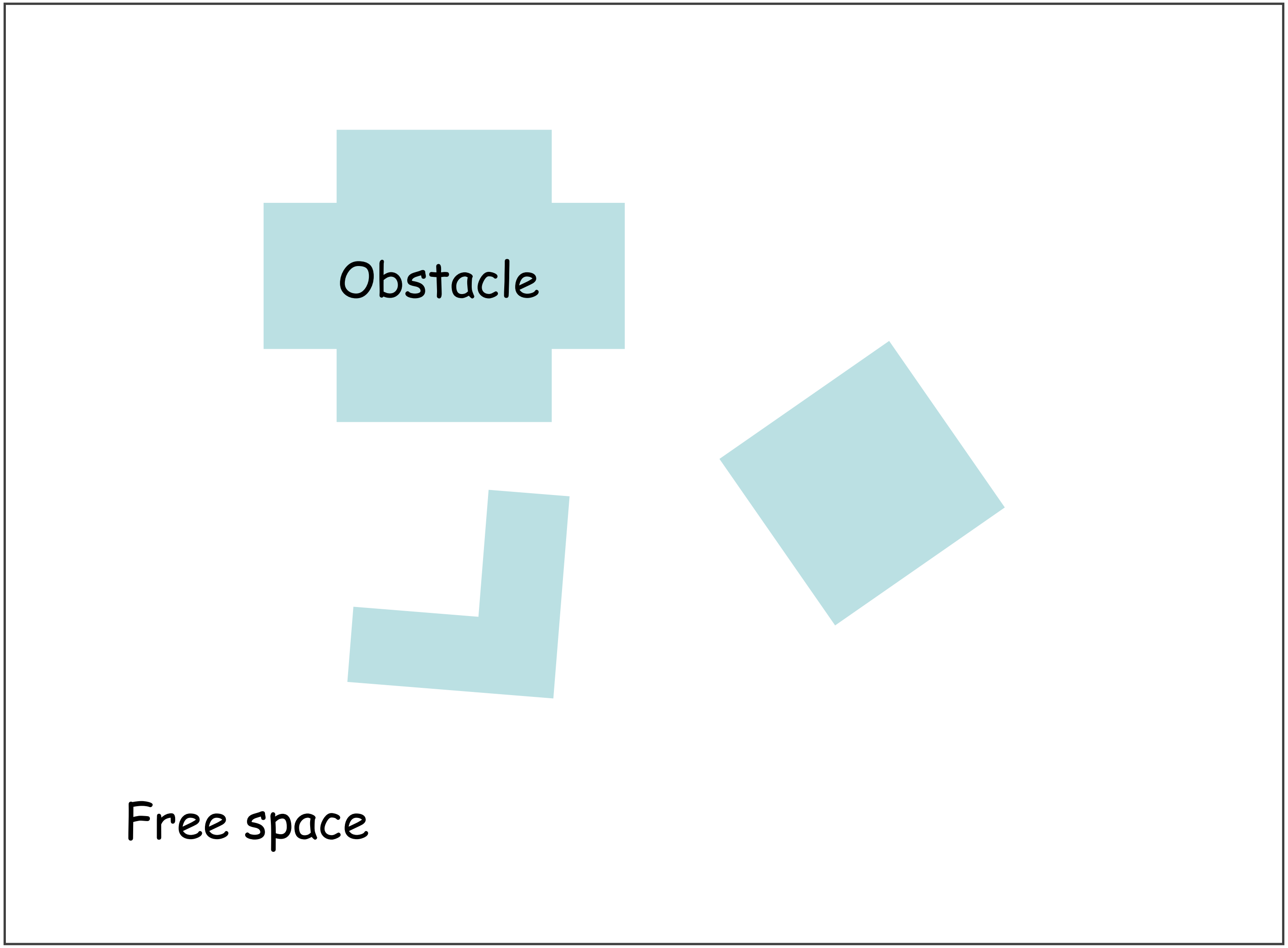
Path planning: a collection of discrete motions between a start and a goal

Strategies

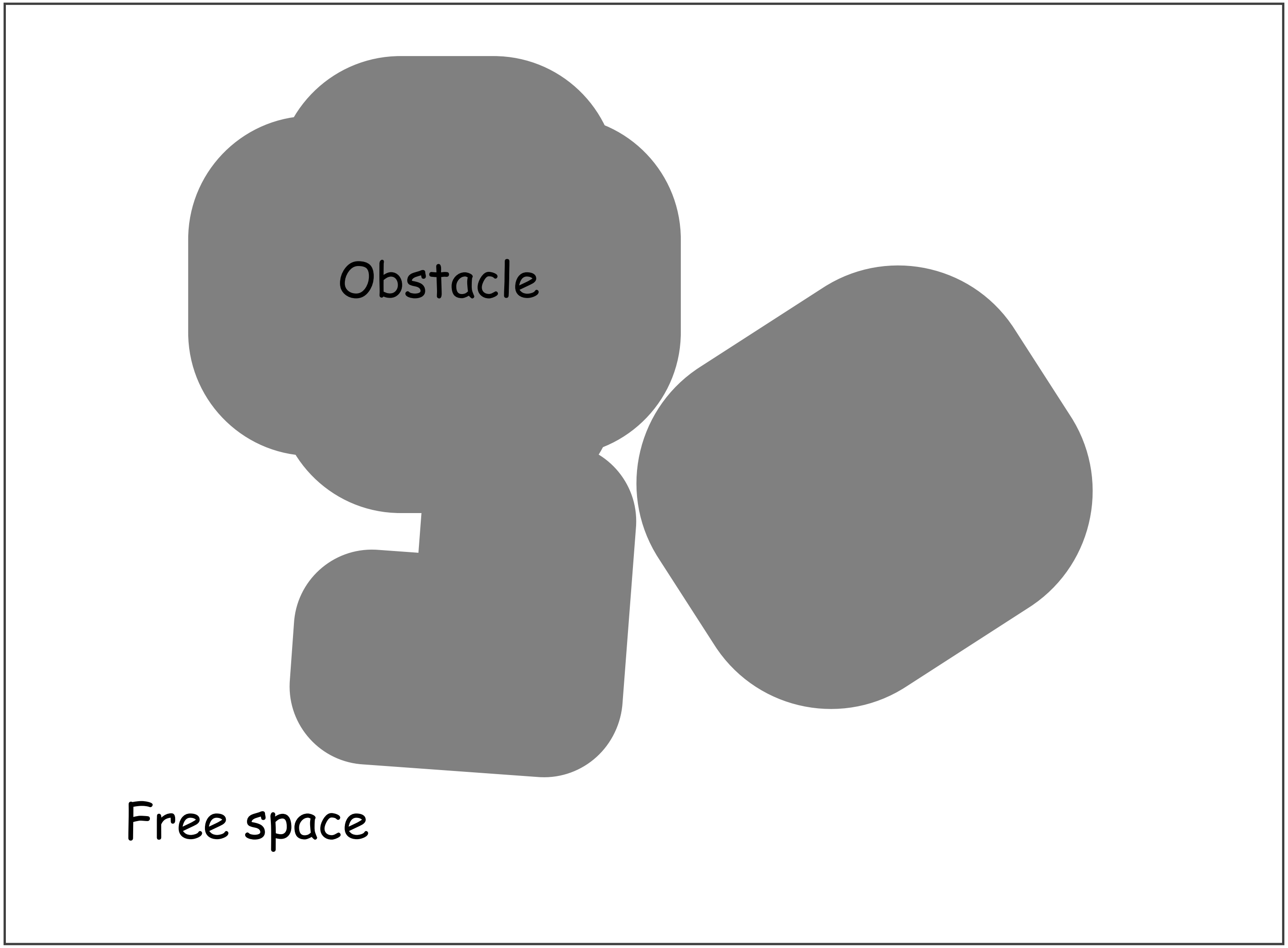
- Graph search
  - Covert free space to a connectivity graph
  - Apply graph search algorithm to find a path to the goal

- Potential field planning
  - Impose a mathematical function directly on the free space
  - Follow the gradient of the function to get to the goal

Configuration space: point-mass robot



Configuration space: circular robot



Configuration space

Configuration space C

- A set of all possible configurations of a robot
- In mobile robots, configuration (pose) is represented by (x, y, θ)
- For a differential-drive robot, there are limited robot velocities in each configuration.

For path planning, assume that

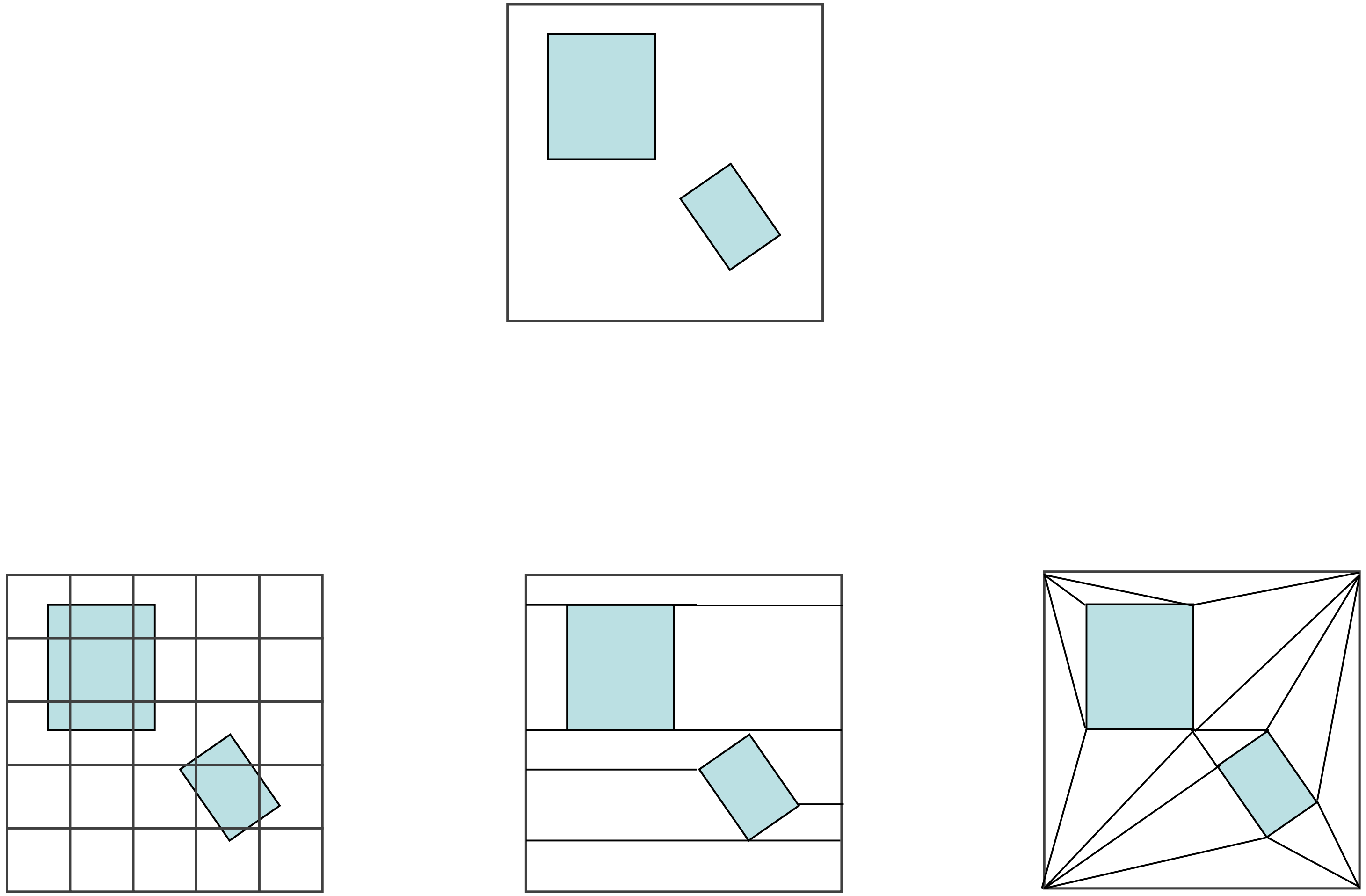
- the robot is holonomic
- the robot has a point-mass
    - Must inflate the obstacles in a map to compensate

Path planning: graph search

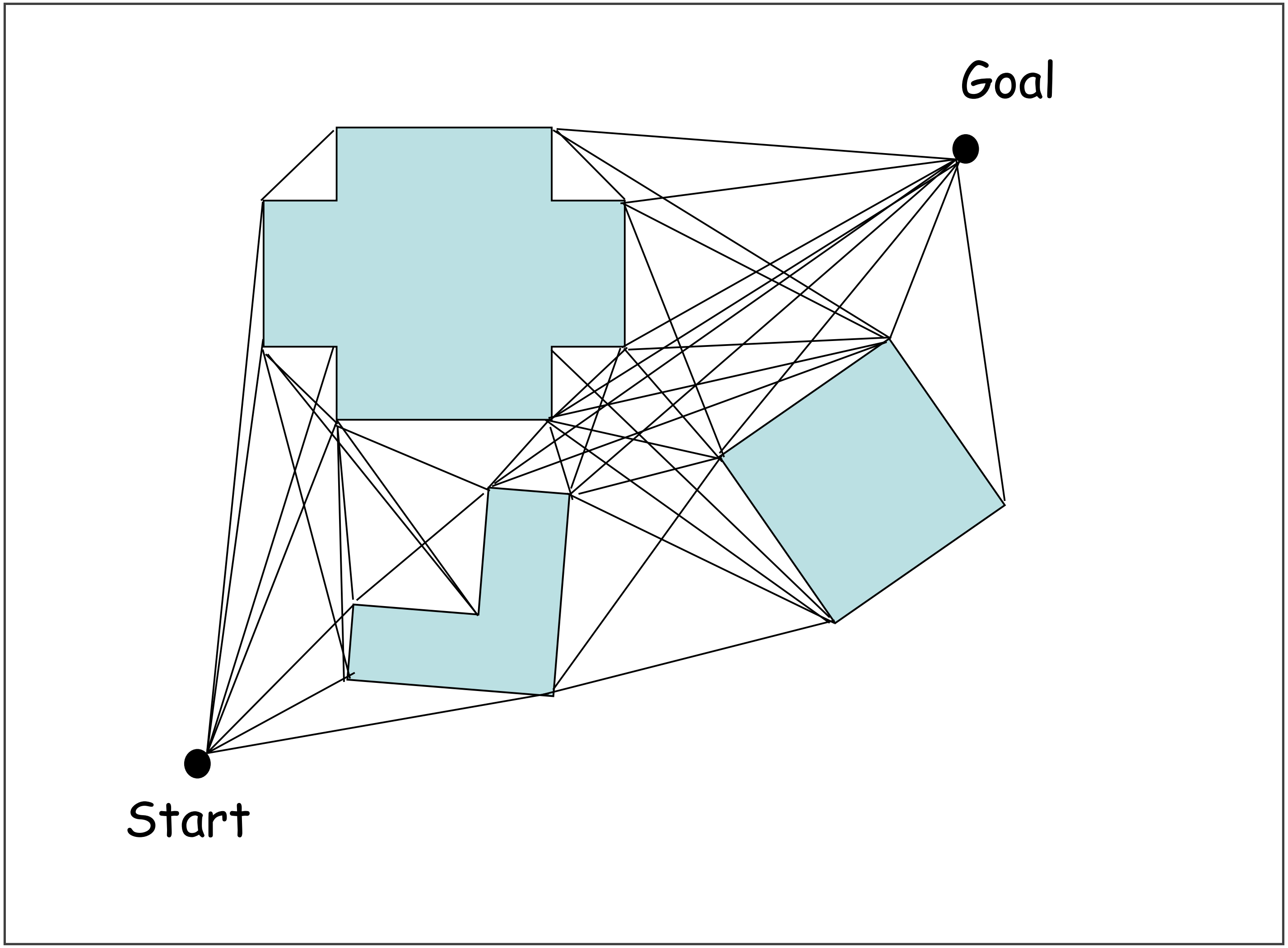
- Graph construction
  - Visibility graph
  - Voronoi diagram
  - Exact cell decomposition
  - Approximate cell decomposition

- Graph search
  - Deterministic graph search
  - Randomized graph search

Graph construction



Visibility graph



Visibility graph

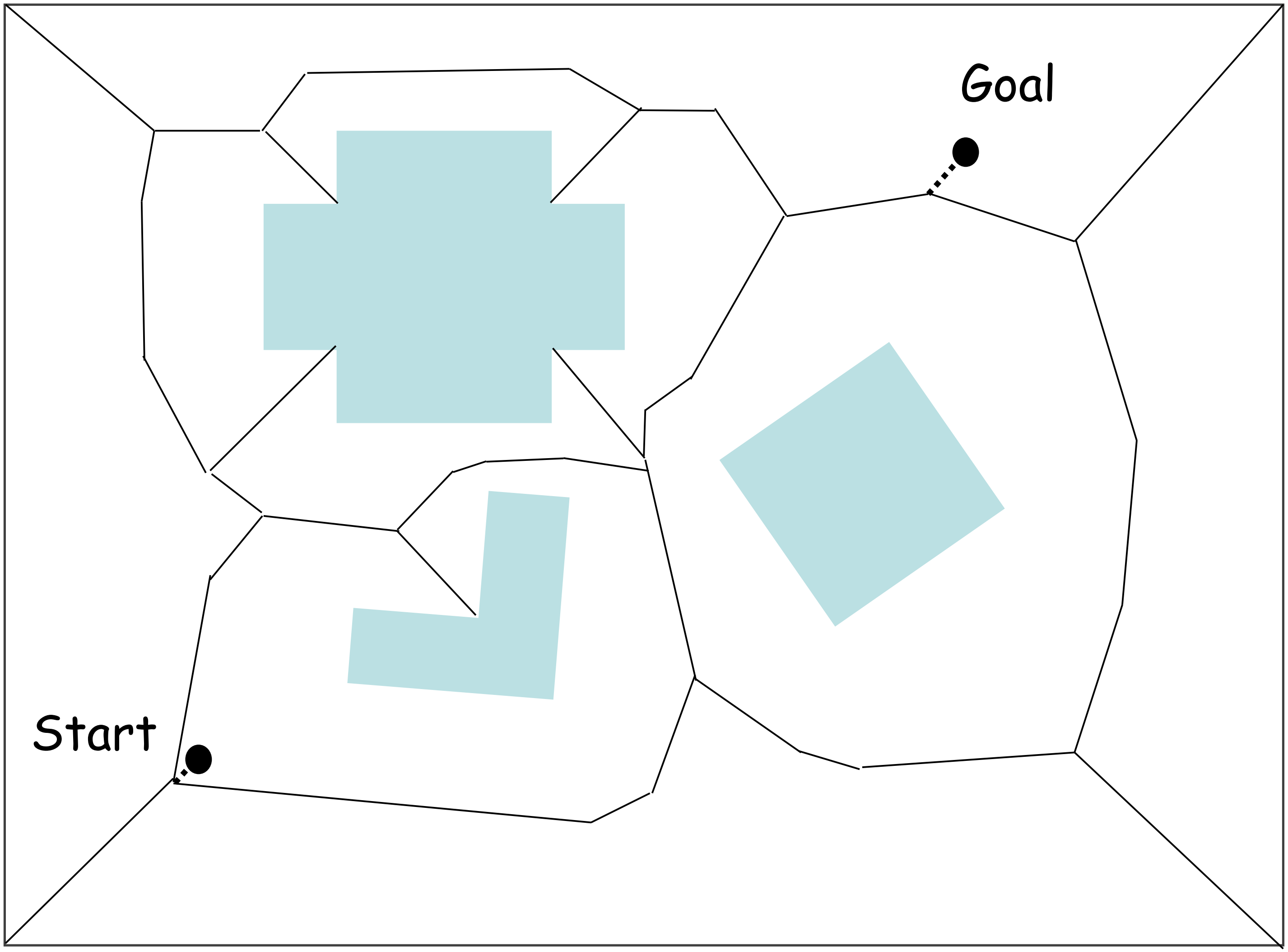
Advantages

- Optimal path in terms of path length
- Simple to implement

Issues

- Number of edges and nodes increase with the number of obstacle polygons
- Resulting path takes the robot as close as possible to obstacles
    - A modification to the optimal solution is necessary to ensure safety

Voronoi diagram



Voronoi diagram

- For each point in free space, compute its distance to the nearest obstacle.
- At points that are equidistant to two or more obstacles, create ridge points.
- Connect the ridge points to create the Voronoi diagram

Voronoi diagram

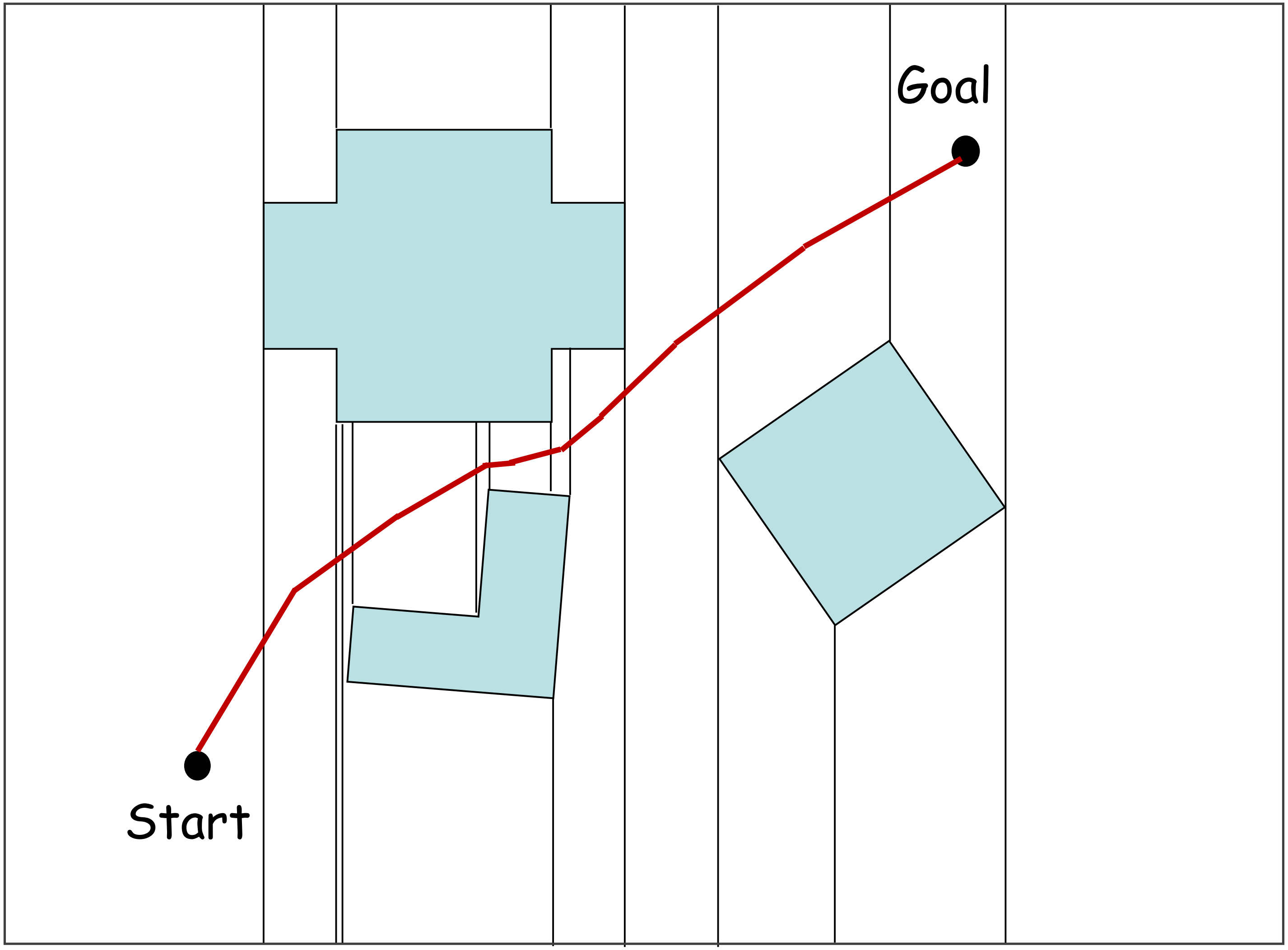
Advantages

- Maximize the distance between a robot and obstacles
  - Keeps the robot as safe as possible
- Executability
  - A robot with a long-range sensor can follow a Voronoi edge in the physical world using simple control rules: maximize the readings of local minima in the sensor values.

Issues

- Robots with short-range sensor may fail to localize.

Exact cell decomposition



Exact cell decomposition

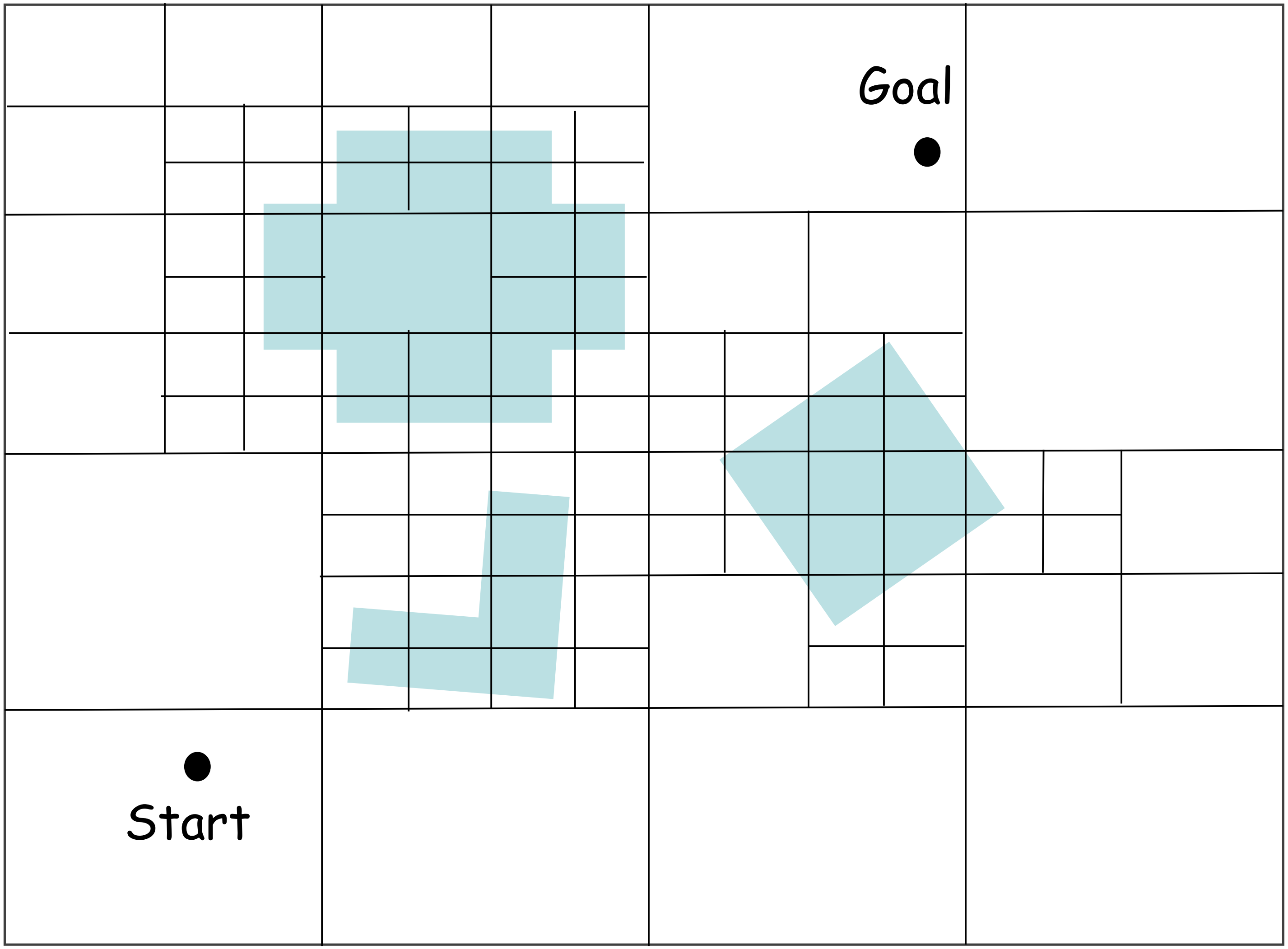
Advantages

- In a sparse environment, the number of cells is small regardless of actual environment size.
- Robots can move around freely within a free cell.

Issues

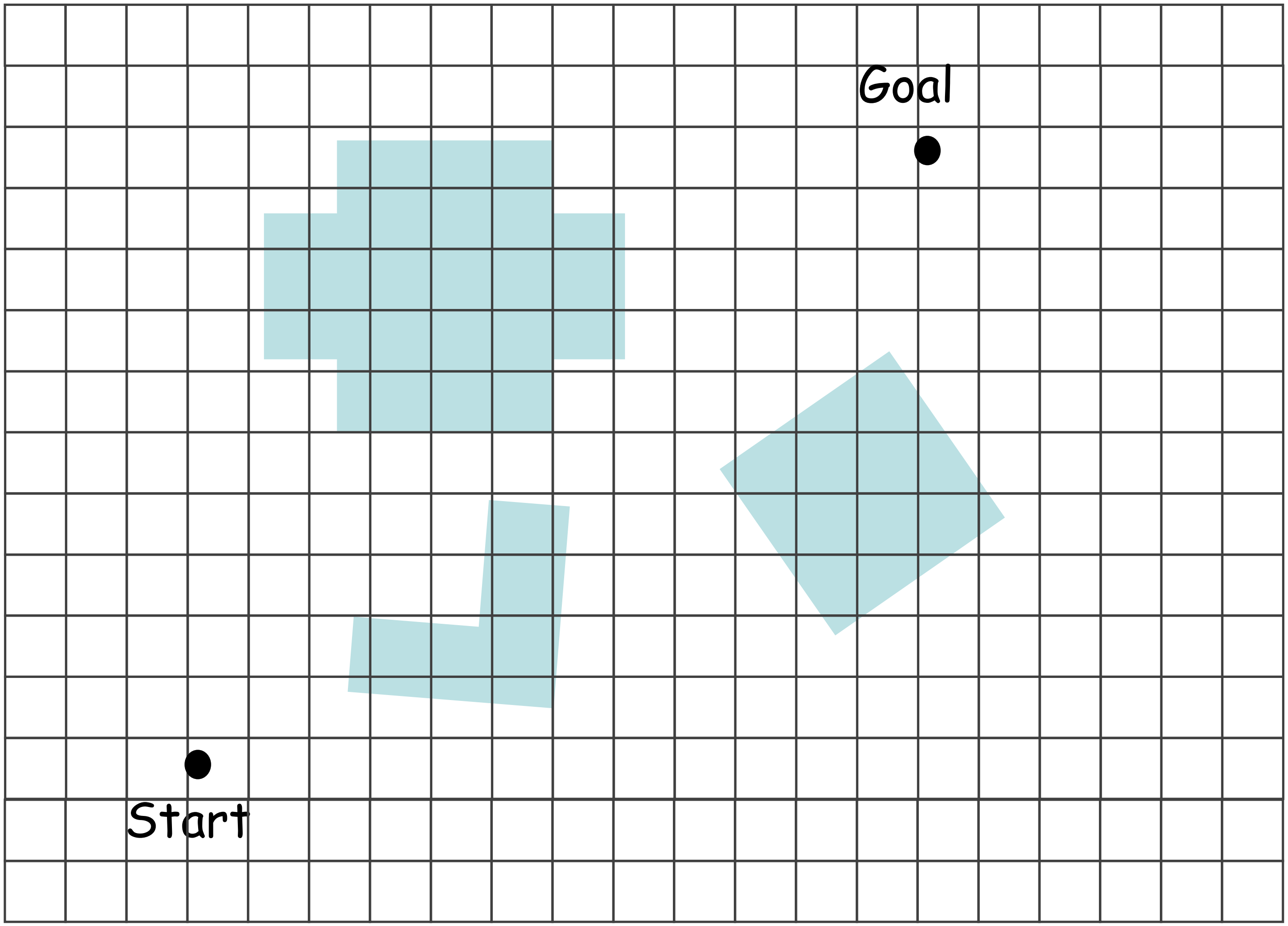
- The number of cells depends on the destiny and complexity of obstacles in the environment

Approximate cell decomposition



Variable-size cell decomposition

Approximate cell decomposition



Fixed-size cell decomposition

Approximate cell decomposition

Variable-size

- Recursively divide the space into rectangles unless
  - A rectangle is completely occupied or completely free
- Stop the recursion when
  - A path planner can compute a solution, or
  - A limit on resolution is attained

Fixed-size

- Divide the space evenly
  - The cell size is often independent of obstacles

Approximate cell decomposition

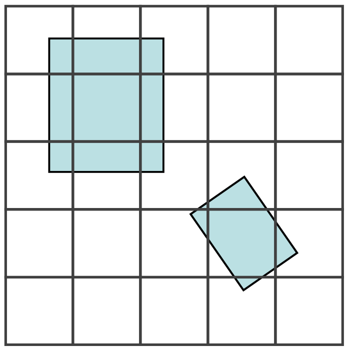
Advantages

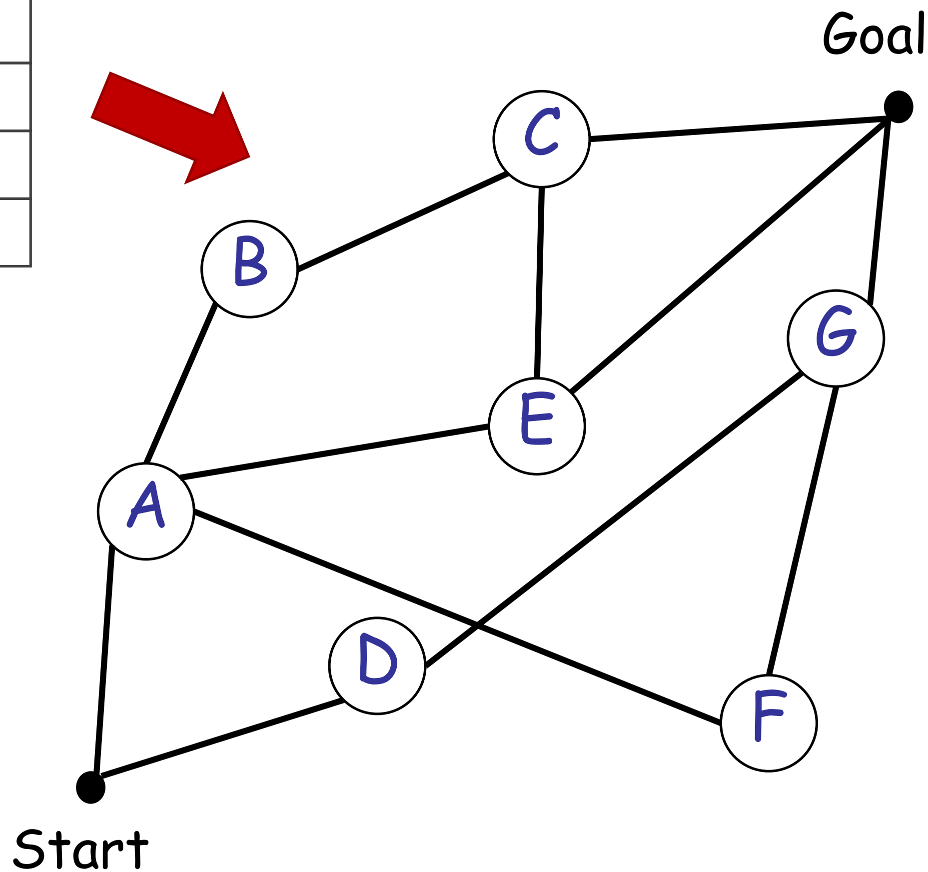
- Low computational complexity

Issues

- Narrow passage ways can be lost

Graph search





Deterministic graph search

Convert the environment map into a connectivity graph

Find the best path (lowest cost) in the connectivity graph

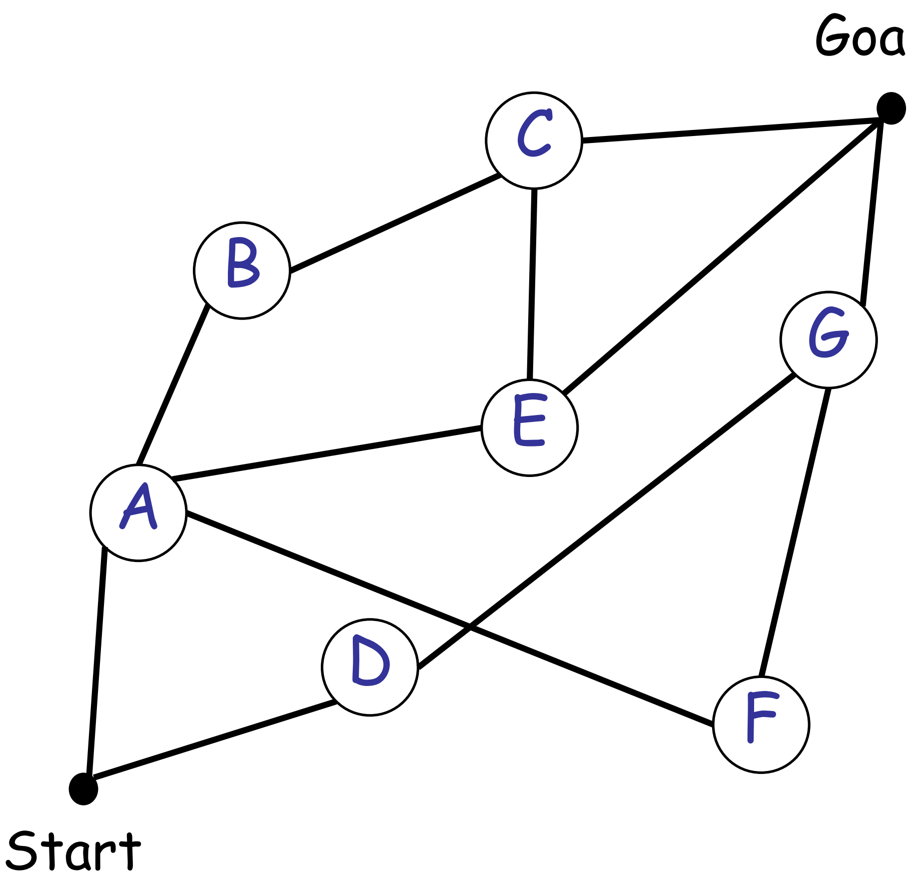
$$f(n) = g(n) + \varepsilon\, h(n)$$

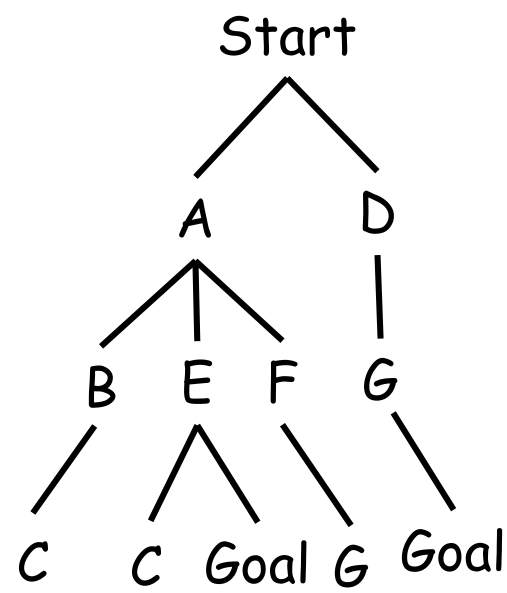
- $f(n)$: Expected total cost
- $g(n)$: Path cost
- $h(n)$: Heuristic cost
- ε: Weighting factor
- n: node/grid cell

$$g(n) = g(n') + c(n, n')$$

- $c(n, n')$: edge traversal cost

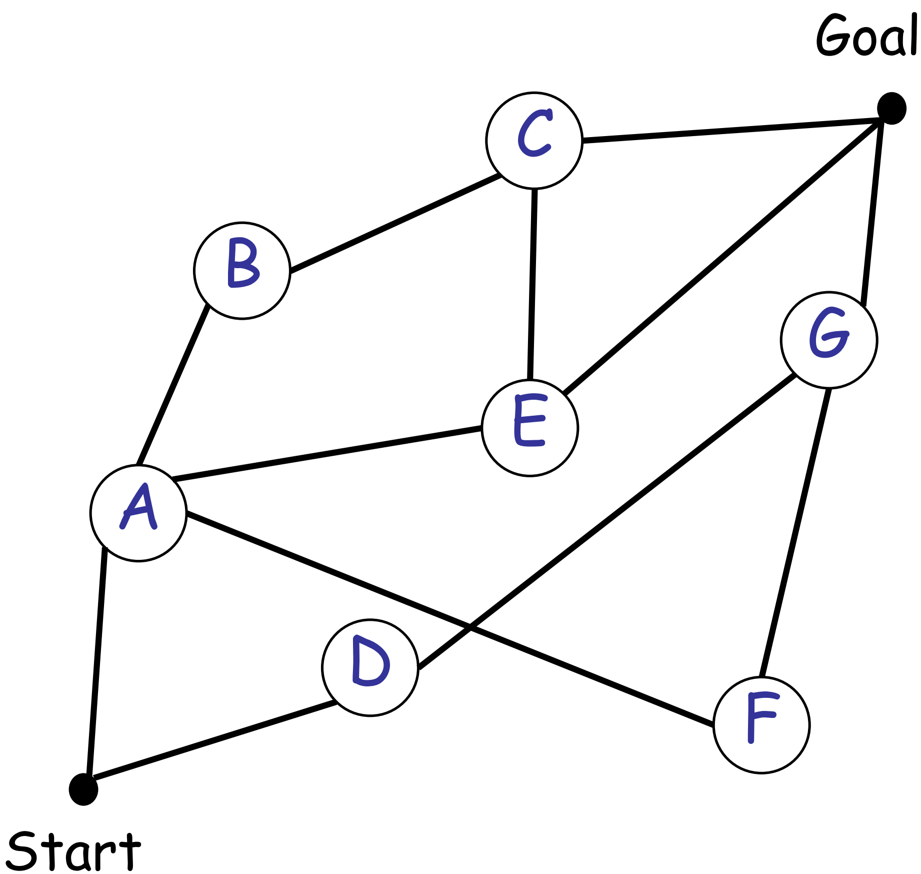
Breadth-first search

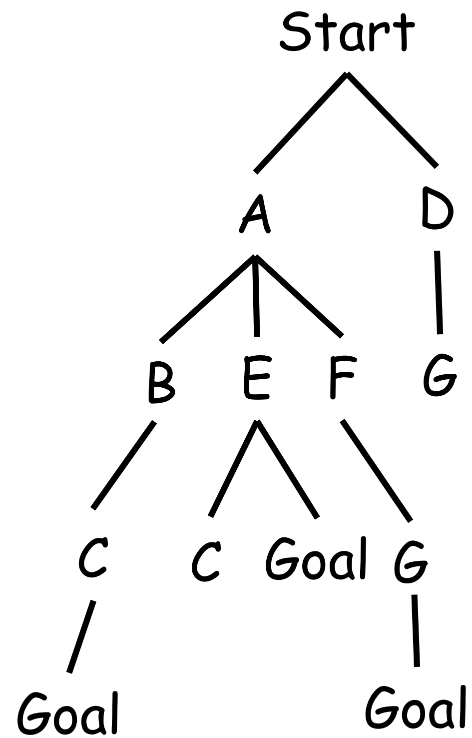




f(n) = g(n) where c(n, n') = 1

Depth-first search





$$f(n) = g(n) \text{ where } c(n, n') = 1$$

Breadth-first search vs depth-first search

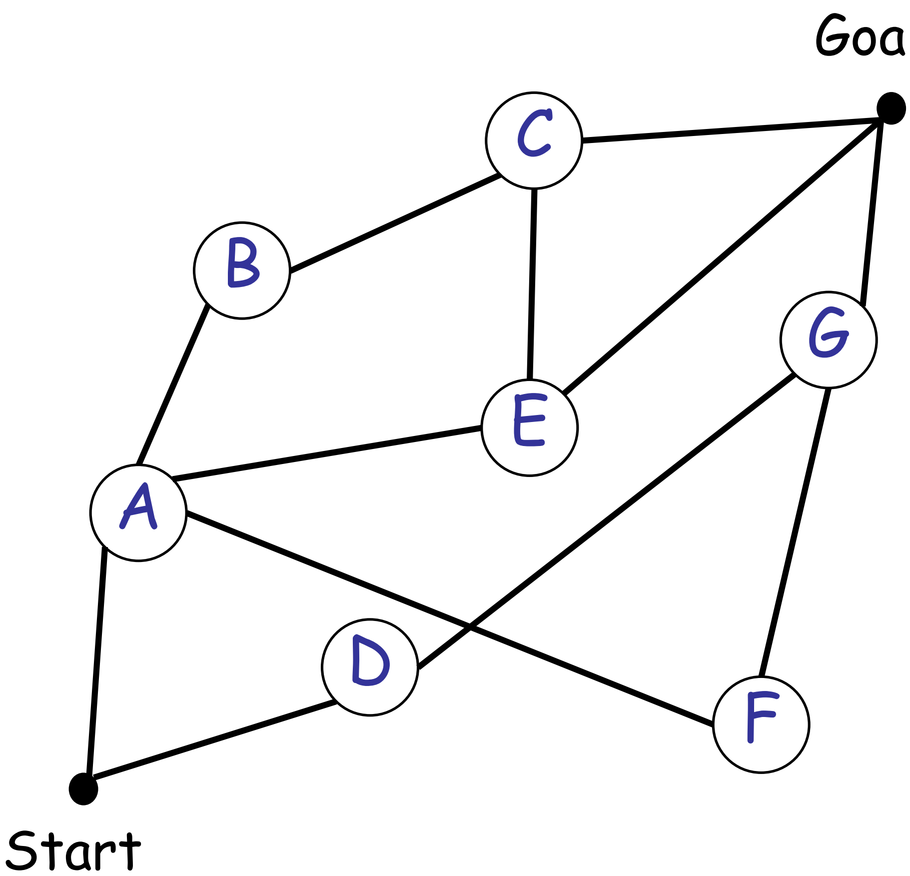
Breadth-first

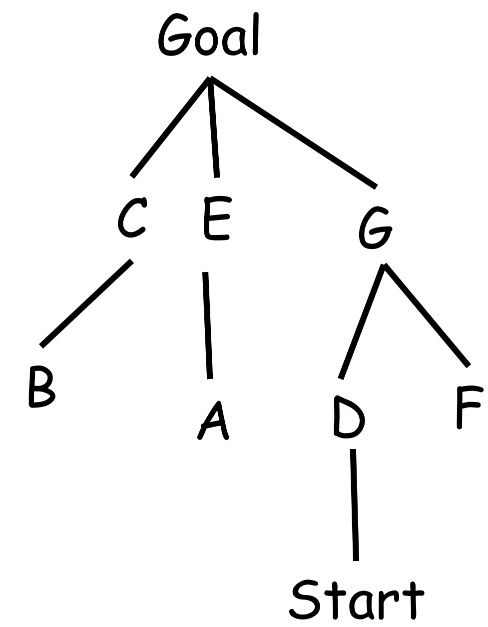
- All the paths need to be stored.
- When a path to the goal is

Depth-first

- Expand each node up to the deepest level of the graph first.
- May revisit previously visited nodes or redundant paths.
- Reduction in space complexity: Only need to store a single node

Dijkstra's algorithm





$$f(n) = g(n) + 0 * h(n)$$

Dijkstra's algorithm

```
dijkstra_shortest_path ( map: GRID_MAP; start_cell: GRID_CELL; goal_cell: GRID_CELL )
	local
		c: GRID_CELL
	do
		map.initialize_all_cells( start_cell )
		from  until map.is_visited( goal_cell ) or map.all_cells_visited loop
				c := map.closest_unvisited_cell -- with minimum path cost
				map.set_visited( c )
				map.update_expected_cost( c )
		end
	end
```

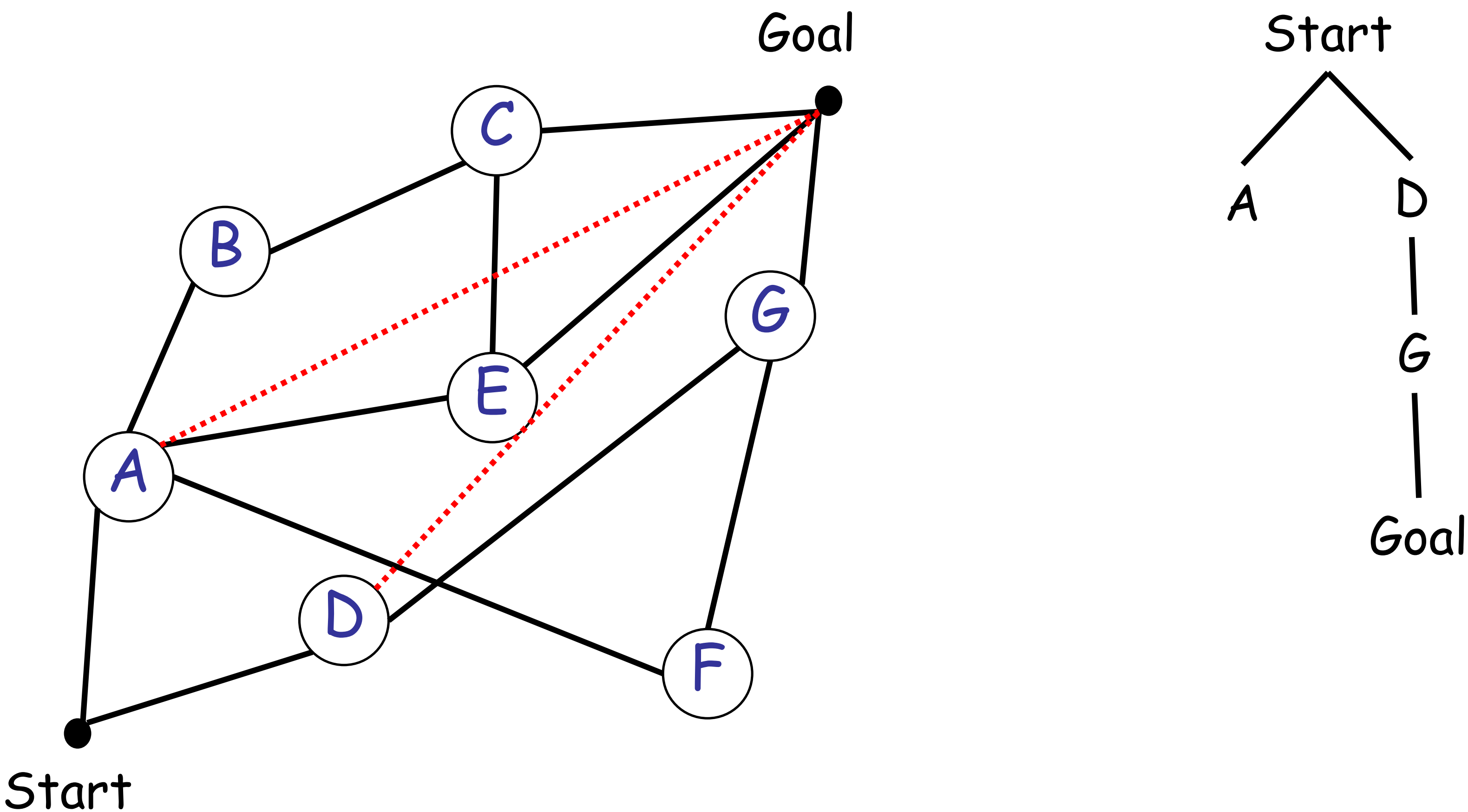
Dijkstra's algorithm

```
initialize_all_cells ( start_cell : GRID_CELL )
    do
        across grid_cells as c loop
            if c = start_cell then
                c.set_distance( 0 )
                c.set_visited( true )
            else
                c.set_distance( (create {REAL_32}).max_value )
                c.set_visited( false )
            end
        end
    end
```

Dijkstra's algorithm

```
update_expected_cost_cost ( start_cell : GRID_CELL )
    local
        d : REAL_32
    do
        across neighboring_cells( start_cell ) as c loop
            d := start_cell.distance + start_cell.compute_distance( c )
            if d < c.distance and not c.is_visited then
                c.distance := d
                c.previous_vertex := start_cell
            end
        end
    end
```

A* algorithm



$$f(n) = g(n) + h(n)$$

A* algorithm

```
A*_shortest_path ( map: GRID_MAP; start_cell: GRID_CELL; goal_cell: GRID_CELL )
    local
        c : GRID_CELL
    do
        map.initialize_open_list( start_cell )
        map.set_goal( goal_cell )
        from until map.is_closed( goal_cell ) or not map.has_open_vertices loop
            c := map.loweset_expected_cost_cell_in_open_list
            map.move_to_closed_list( c )
            across map.neighboring_cells( c ) as n loop
                if not map.is_occupied( n ) and not map.is_in_closed_list( n ) then
                    if not map.is_in_open_list( n ) then
                        map.add_to_open_list( n, c )
                    elseif map.has_lower_expected_cost( n, c ) then
                        map.update_open_list( n, c )
                    end
                end
            end
        end
    end
```

A* algorithm

```
initialize_open_list ( cell : GRID_CELL )
    do
        cell.set_g_score( 0 )
        cell.compute_f_score( heurist_cost(cell, goal_cell) )
        open_list.add( cell )
    end

add_to_open_list ( cell : GRID_CELL; parent_cell : GRID_CELL )
    do
        cell.set_previous_cell( parent_cell )
        cell.set_g_score( parent_cell.g_score + compute_distance( cell, parent_cell ) )
        cell.compute_f_score( heurist_cost(cell, goal_cell) )
        open_list.add( cell )
    end
```

A* algorithm

```
has_lower_expected_cost ( cell : GRID_CELL; parent_cell : GRID_CELL ) : BOOLEAN
    local
        g_score : REAL_32
        f_score : REAL_32
    do
        g_score := parent_cell.g_score + compute_distance( cell, parent_cell )
        f_score := g_score + heuristic_cost( cell, goal_cell )
        if f_score < cell.f_score then
            Result := true
        else
            Result := false
        end
    end

update_open_list ( cell : GRID_CELL; parent_cell : GRID_CELL )
    do
        cell.set_previous_cell( parent_cell )
        cell.set_g_score( parent_cell.g_score + compute_distance( cell, parent_cell ) )
        cell.compute_f_score( heurist_cost(cell, goal_cell) )
    end
```

Dijkstra algorithm vs A* algorithm

Dijkstra

- $f(n) = g(n)$
  - $H(n) = 0$
- When computed from the goal, the best path from any cell to the goal can be found

A*

- $f(n) = g(n) + \varepsilon\, h(n)$
  - $h(n) = || n - n_{goal} ||$
- $\varepsilon = 1$ leads to the optimal A* solution
- $\varepsilon > 1$ results in a greedy solution

Randomized graph search

- Initialize a tree
- Add nodes to the tree until a termination condition is triggered
- During each step:
  - Pick a random configuration q_{rand} in the free space.
  - Compute the tree node q_{near} closest to q_{rand}
  - Grow an edge (with a fixed length) from q_{near} to q_{rand}
  - Add the end q_{new} of the edge if it is collision free

Randomized graph search

Advantages

- Can address situations in which exhaustive search is not an option.

Issues

- Cannot guarantee solution optimality.
- Cannot guarantee deterministic completeness.

- If there is a solution, the algorithm will eventually find it as the number of nodes added to the tree grows to infinity.

Path planning strategies

- Graph search
  - Covert free space to a connectivity graph
  - Apply graph search algorithm to find a path to the goal
- Potential field planning
  - Impose a mathematical function directly on the free space
  - Follow the gradient of the function to get to the goal

Pontential field

Create a gradient to direct the robot to the goal position

Main idea

- Robots are attracted toward the goal.
- Robots are repulsed by obstacles.

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

- $F(q)$: artificial force acting on the robot at the position $q = (x, y)$
- $U(q)$: potential field function
- $\nabla U(q)$: gradient vector of U at position q

- $U(q) = U_{attractive}(q) + U_{repulsive}(q)$
- $F(q) = F_{attractive}(q) + F_{repulsive}(q) = -\nabla U_{attractive}(q) - \nabla U_{repulsive}(q)$

Attractive potential

$$U_{attractive}(q) = \frac{1}{2} k_{attrative} \cdot \rho^2_{goal}(q)$$

- $k_{attrative}$: a positive scaling factor
- $\rho_{goal}(q)$: Euclidean distance $||q - q_{goal}||$

$$\begin{aligned} F_{attractive}(q) &= -\nabla U_{attractive}(q) \\ &= -k_{attrative}\, \rho_{goal}(q)\, \nabla \rho_{goal}(q) \\ &= -k_{attrative}\, (q - q_{goal}) \end{aligned}$$

- Linearly converges toward 0 as the robot reaches the goal

Repulsive potential

$$U_{repulsive}(q) = \begin{cases} \frac{1}{2} k_{repulsive} \left(\frac{1}{\rho(q)} - \frac{1}{\rho_0} \right)^2 & \rho(q) \leq \rho_0 \\ 0 & \rho(q) > \rho_0 \end{cases}$$

- $k_{repulsive}$: a positive scaling factor
- $\rho(q)$: minimum distance from q to an object
- ρ_0: distance of influence of the object

$$F_{repulsive}(q) = -\nabla U_{repulsive}(q)$$

$$= \begin{cases} k_{repulsive} \left(\frac{1}{\rho(q)} - \frac{1}{\rho_0} \right) \frac{1}{\rho^2(q)} \frac{q - qobstacle}{\rho(q)} & \rho(q) \leq \rho_0 \\ 0 & \rho(q) \leq \rho_0 \end{cases}$$

- Only for convex obstacles that are piecewise differentiable

Potential field

Advantages

- Both plans the path and determines the control for the robot.
- Smoothly guides the robot towards the goal.

Issues

- Local minima are dependent on the obstacle shape and size.
- Concave objects may lead to several minimal distances, which can cause oscillation