ETH Zürich
Chair of Software Engineering

Robotics Programming Laboratory

Bertrand Meyer
Jiwon Shin

Lecture 7: Path Planning

Path planning

Getting to Zurich HB from WEH D4

- 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 to Central, then to HB
- ...

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

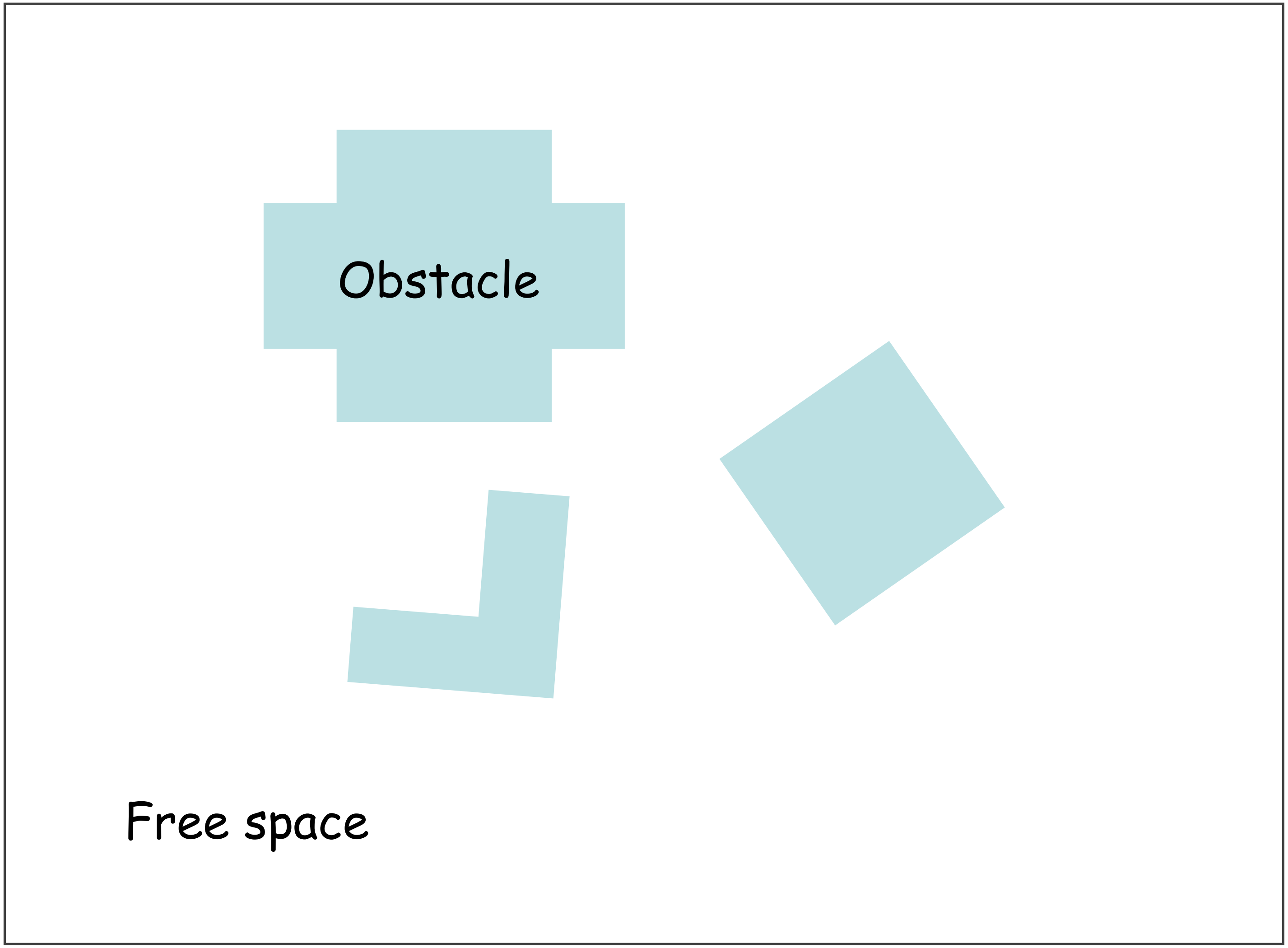
Configuration space C

- 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.

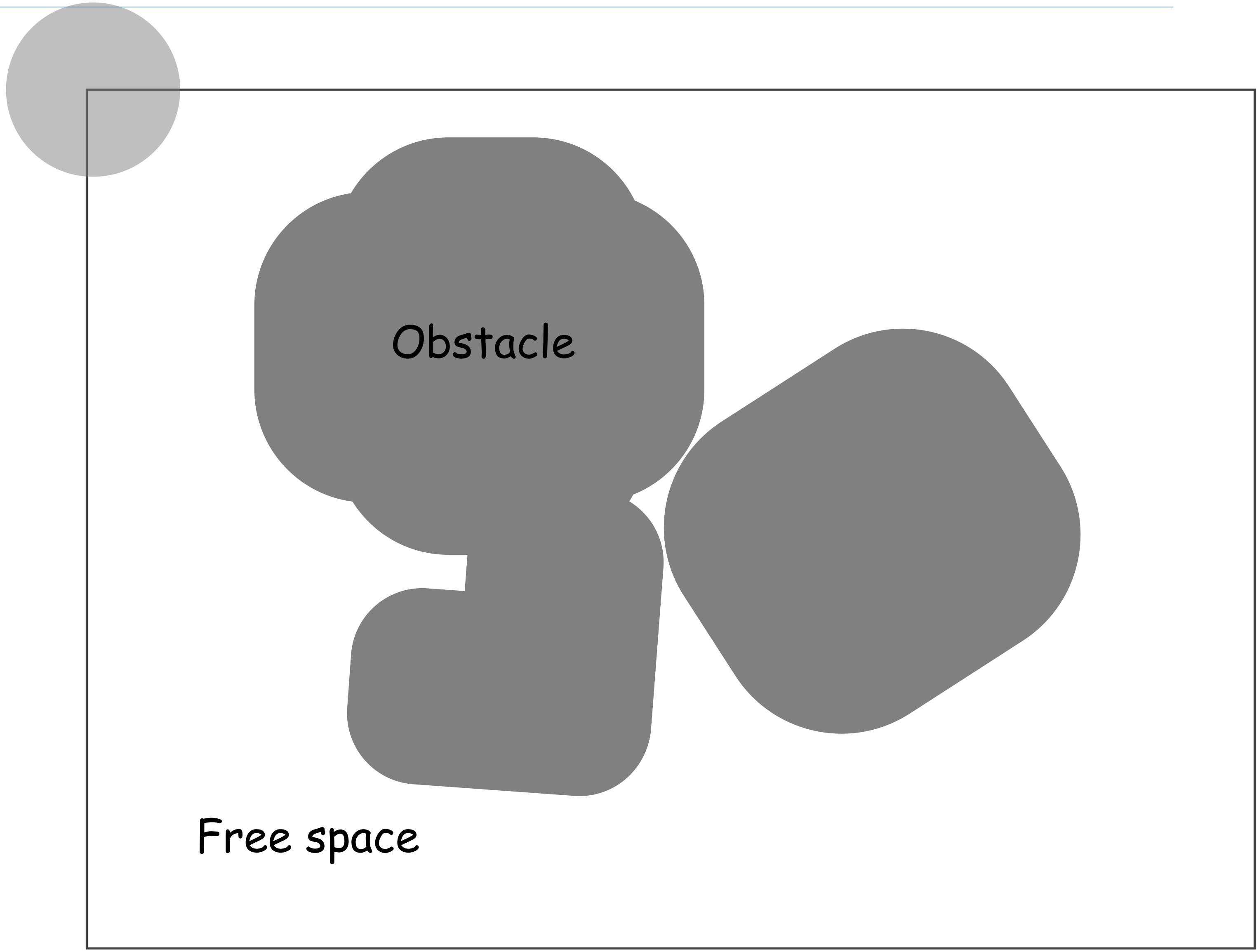
For path planning, assume that

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

Configuration space: point-mass robot



Configuration space: circular robot

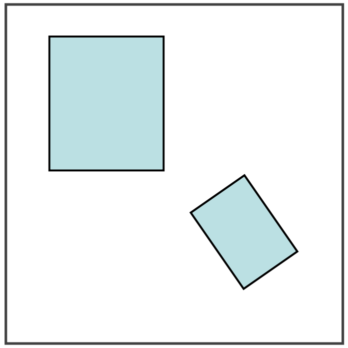


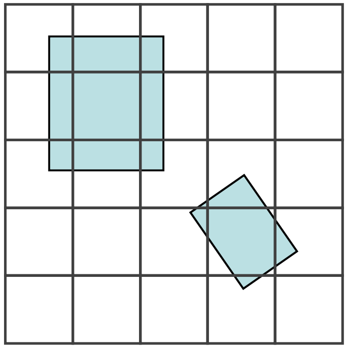
Path planning: graph search

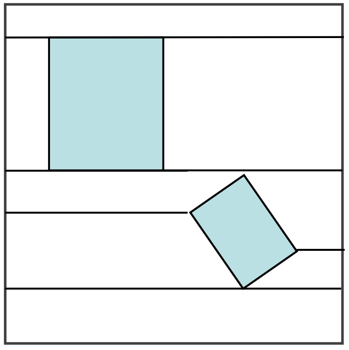
- Graph construction
    - Visibility graph
    - Voronoi diagram
    - Exact cell decomposition
    - Approximate cell decomposition

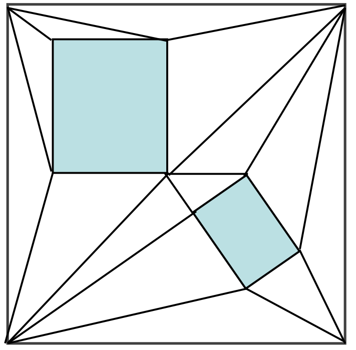
- Graph search
    - Deterministic graph search
    - Randomized graph search

Graph construction

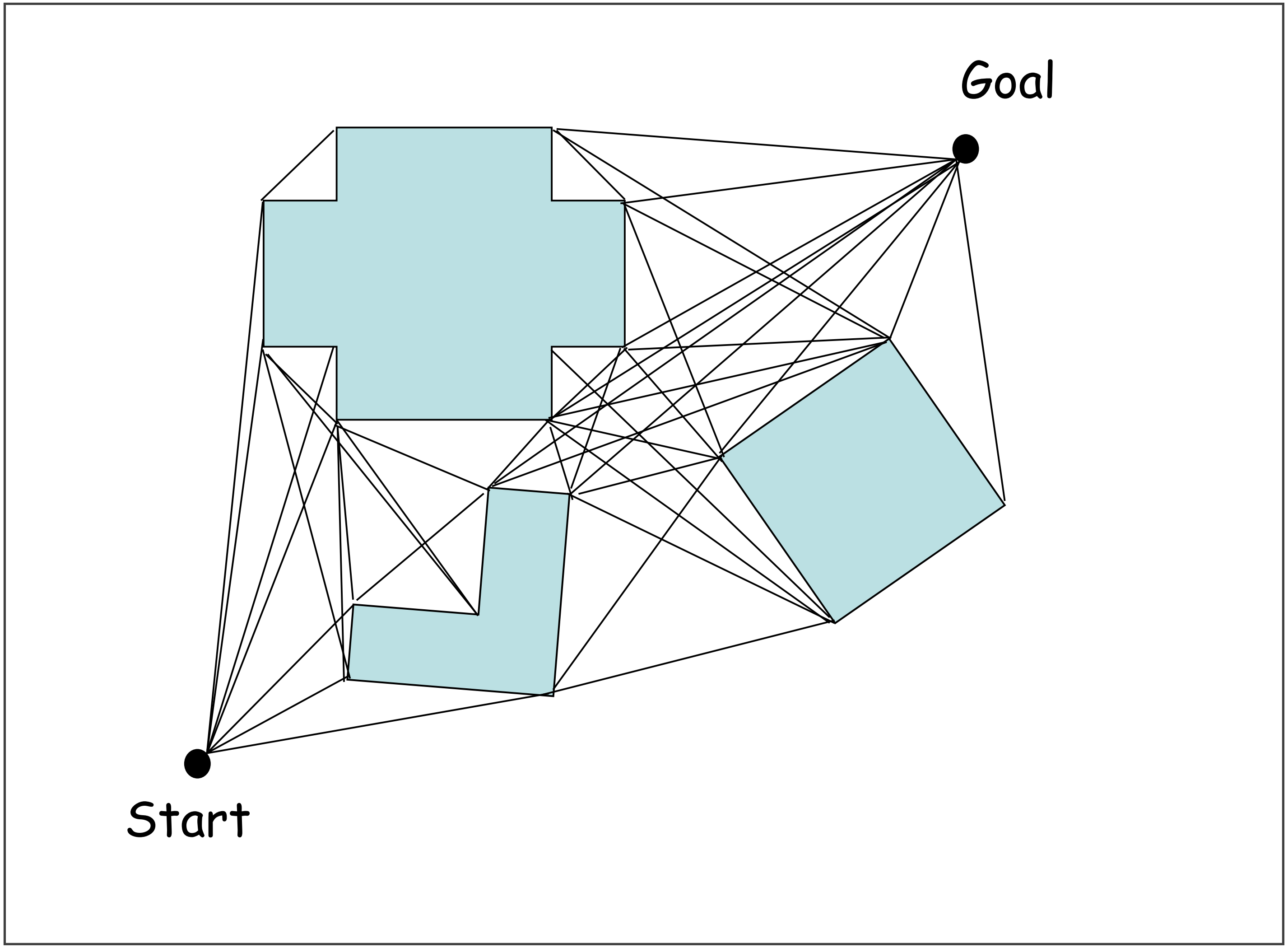








Visibility graph



Visibility graph

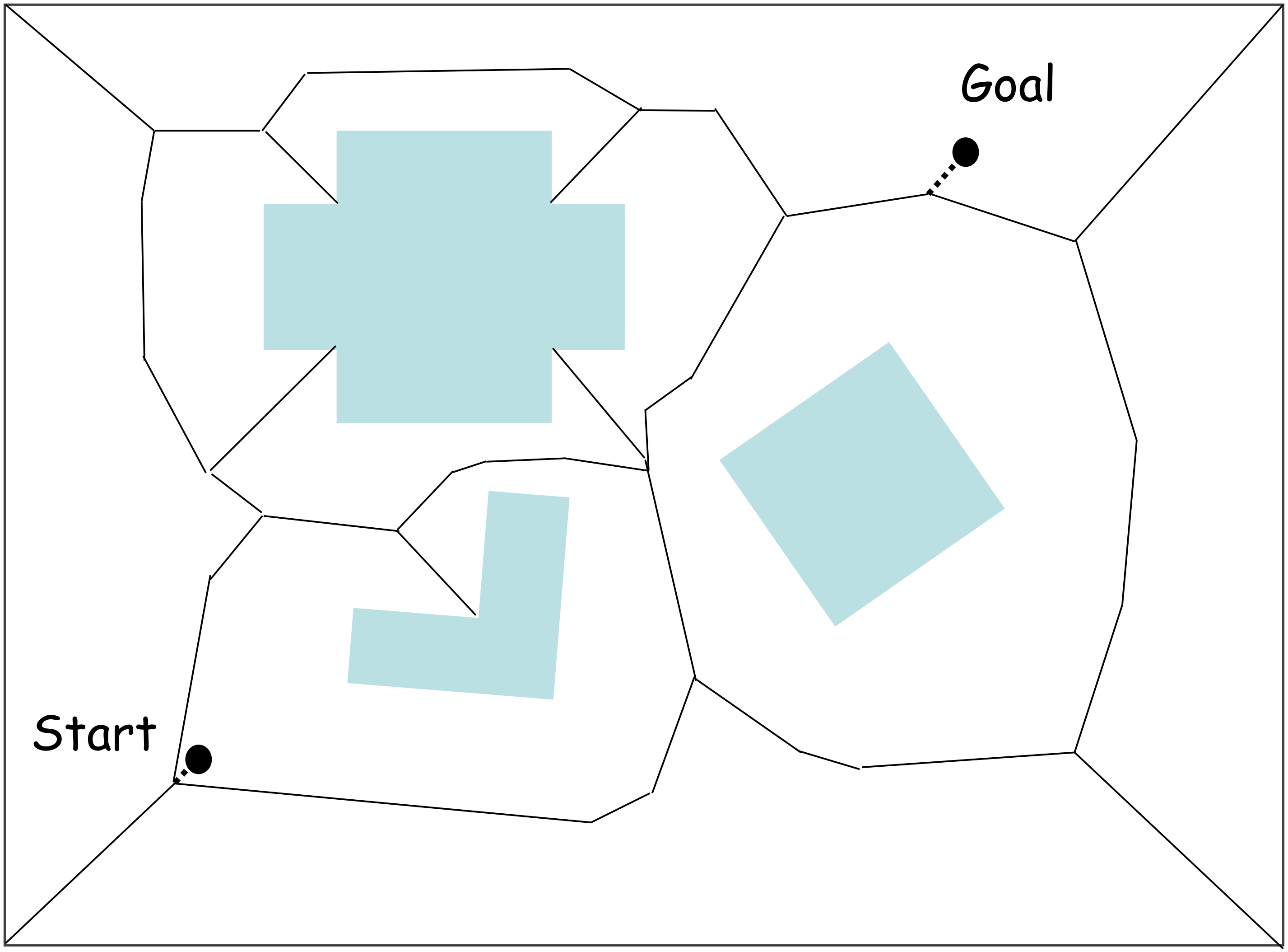
Advantages

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

Issues

- Number of edges and nodes increase with the number of obstacle polygons
  - Fast in sparse environments, but slow and inefficient in densely populated environments
- Resulting path takes the robot as close as possible to obstacles
  - A modification to the optimal solution is necessary to ensure safety
    - Grow obstacles by radius much larger than robot's radius
    - Modify the solution path to be away from obstacles

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

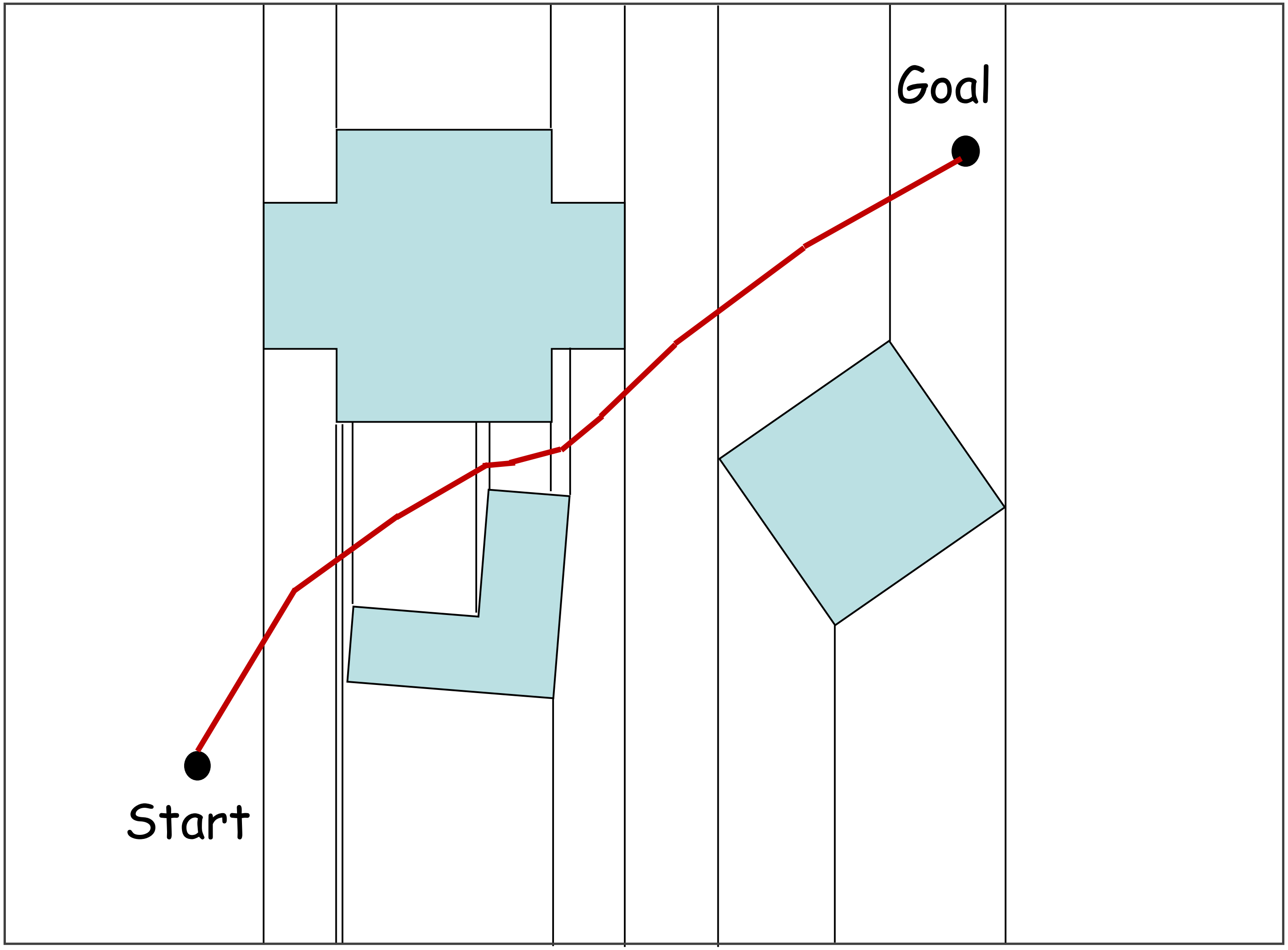
Advantages

- 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

- Not the shortest path in terms of total path length.
- Robots with short-range sensor may fail to localize.

Exact cell decomposition



Exact cell decomposition

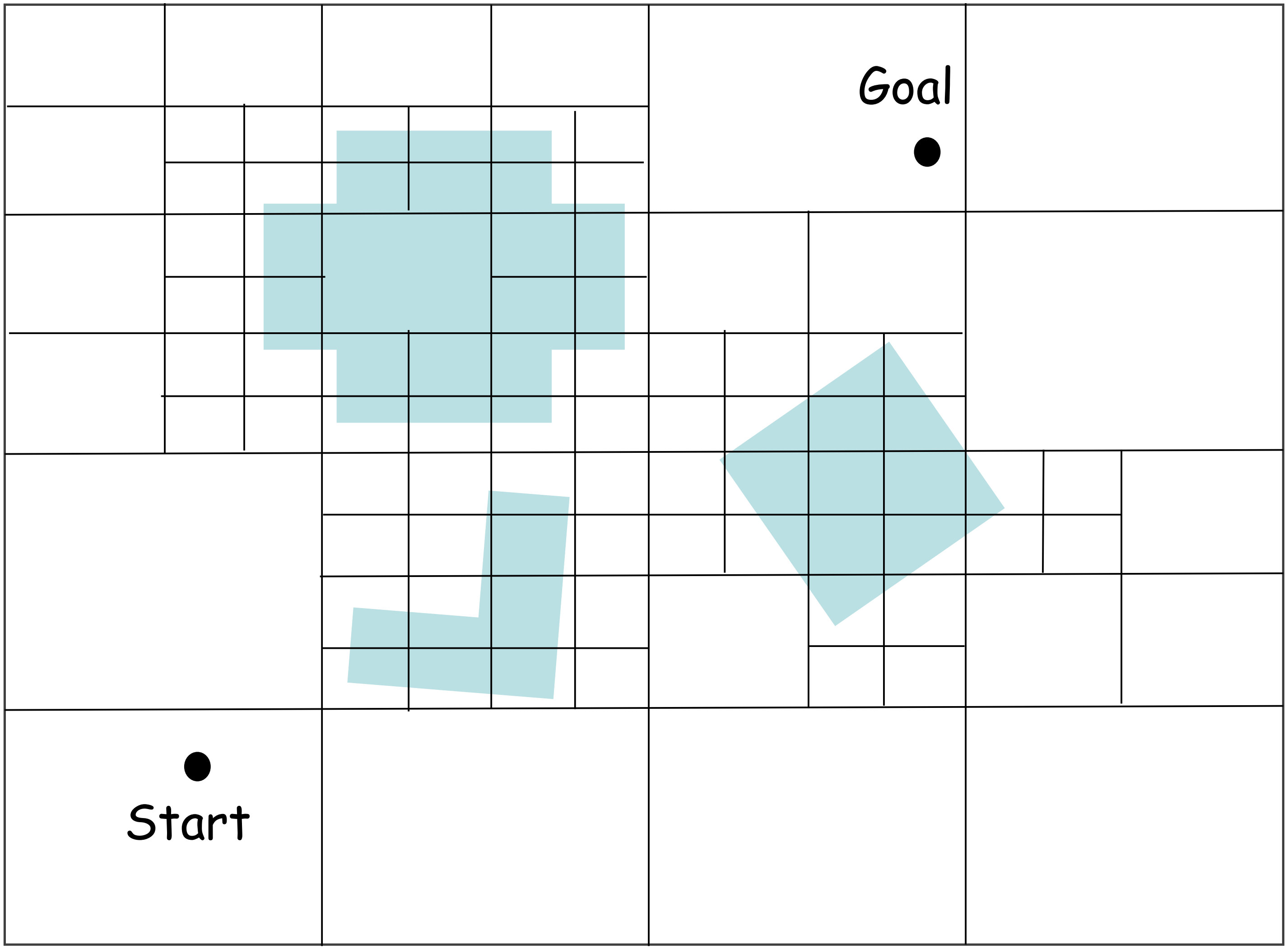
Advantages

- 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

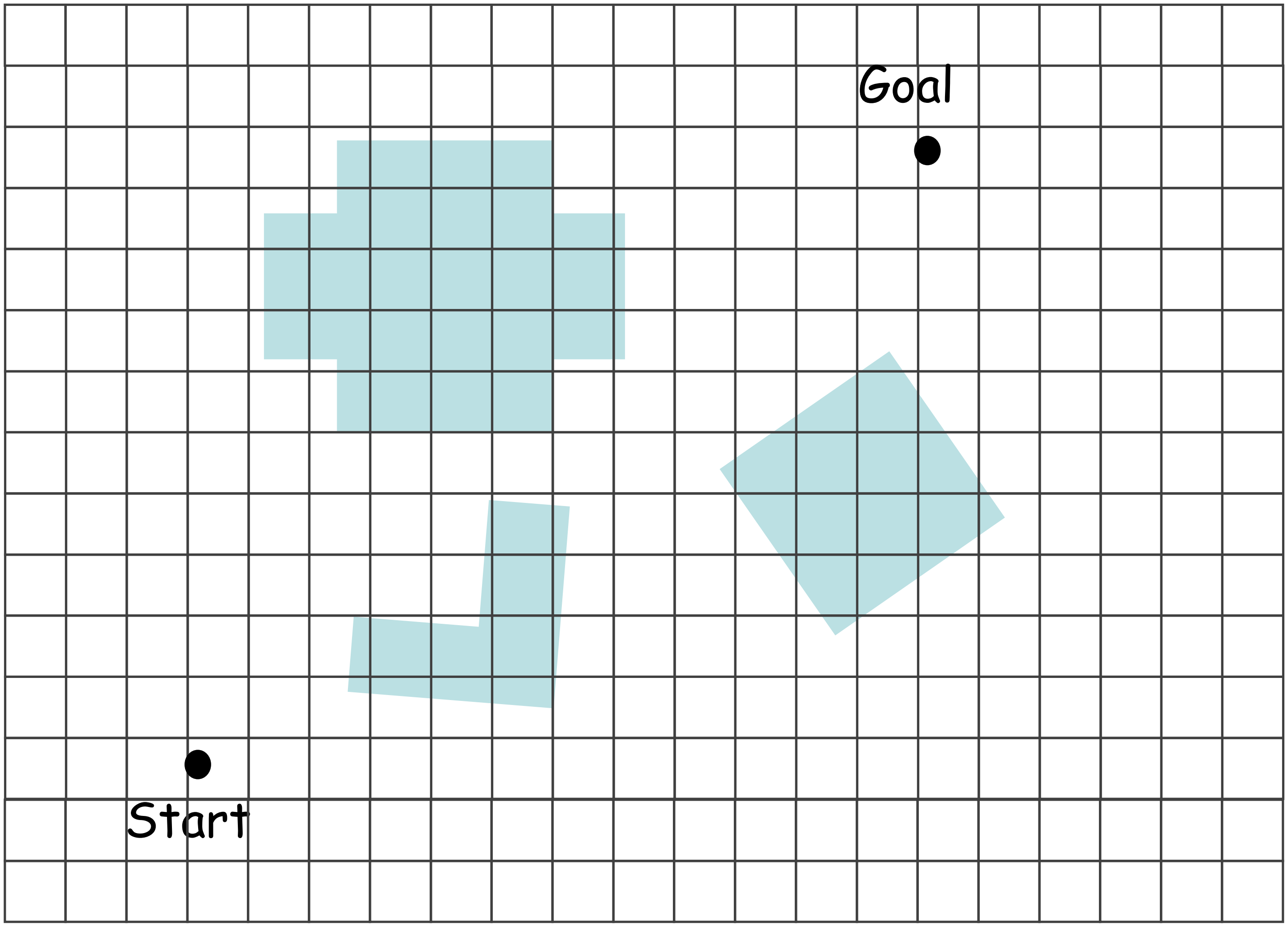
- The number of cells depends on the destiny and complexity of obstacles in the environment

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

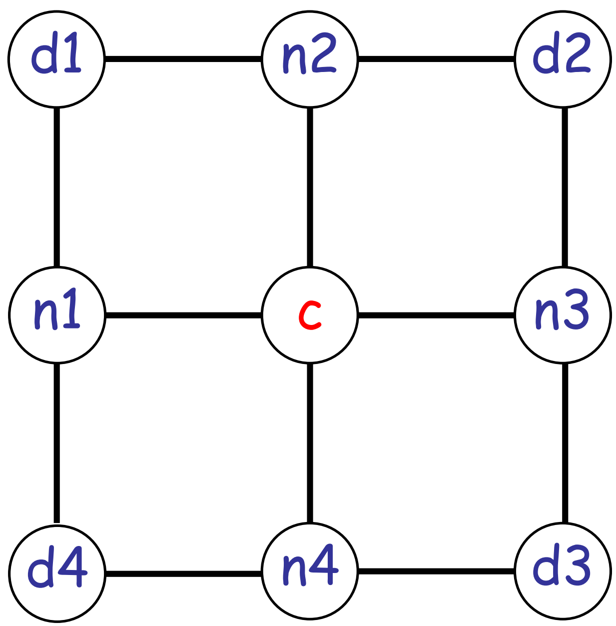
Issues

- Narrow passage ways can be lost

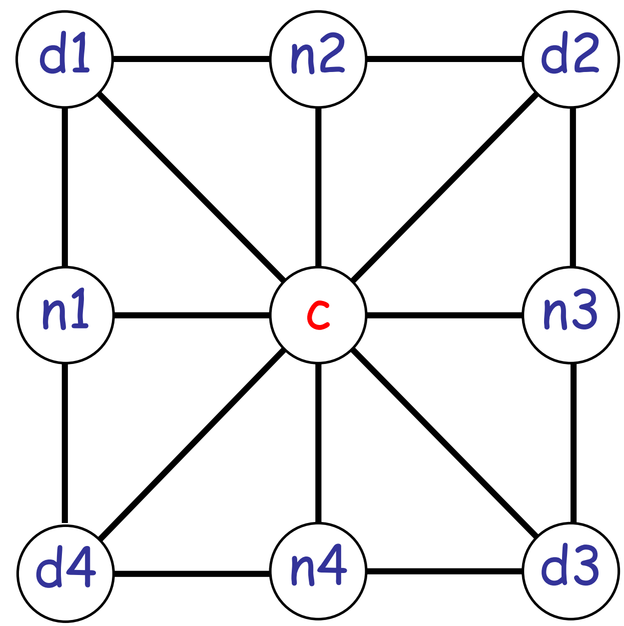
Connectivity

	d1	n2	d2	
	n1	c	n3	
	d4	n4	d3	

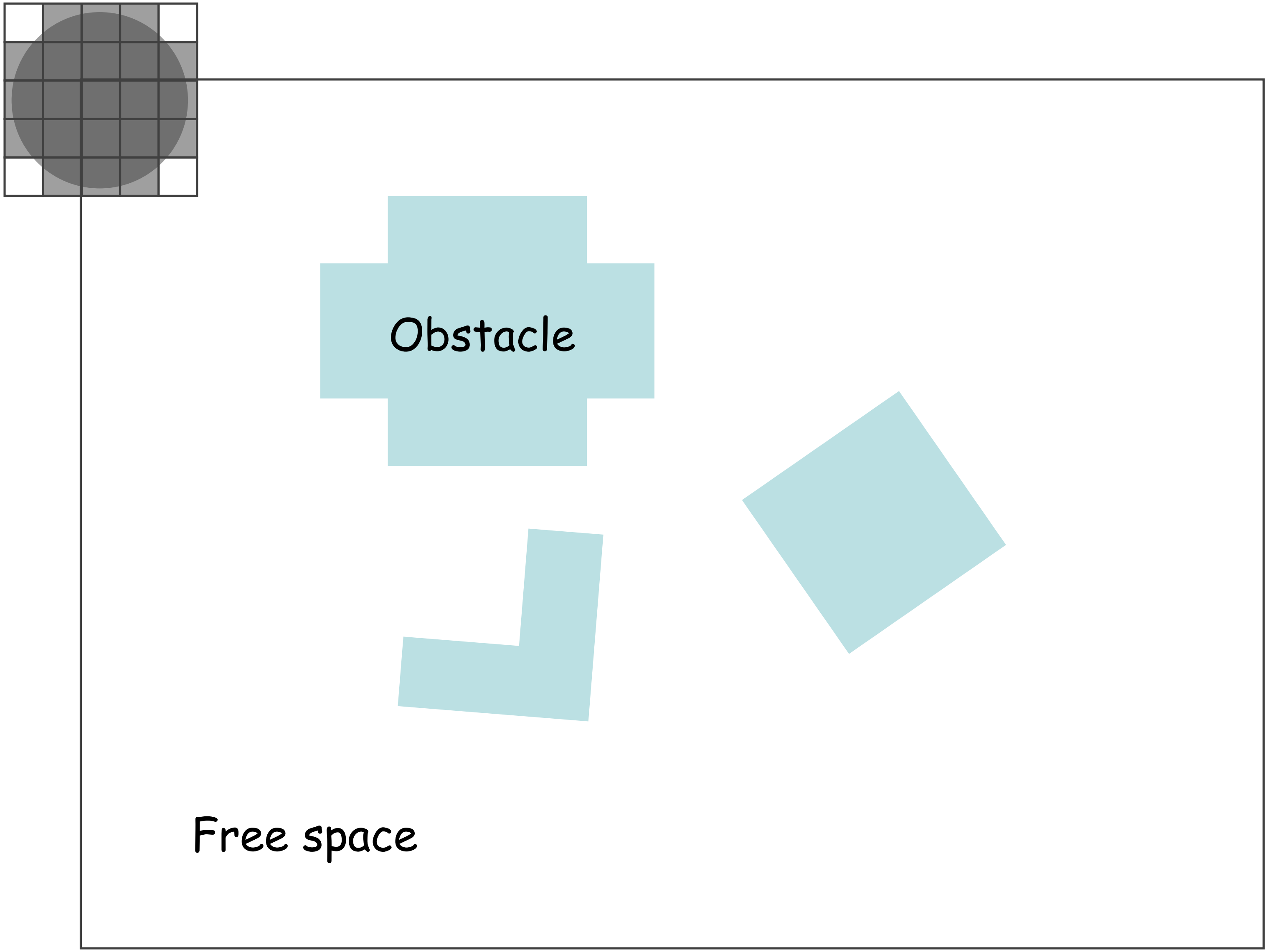
Four-connected



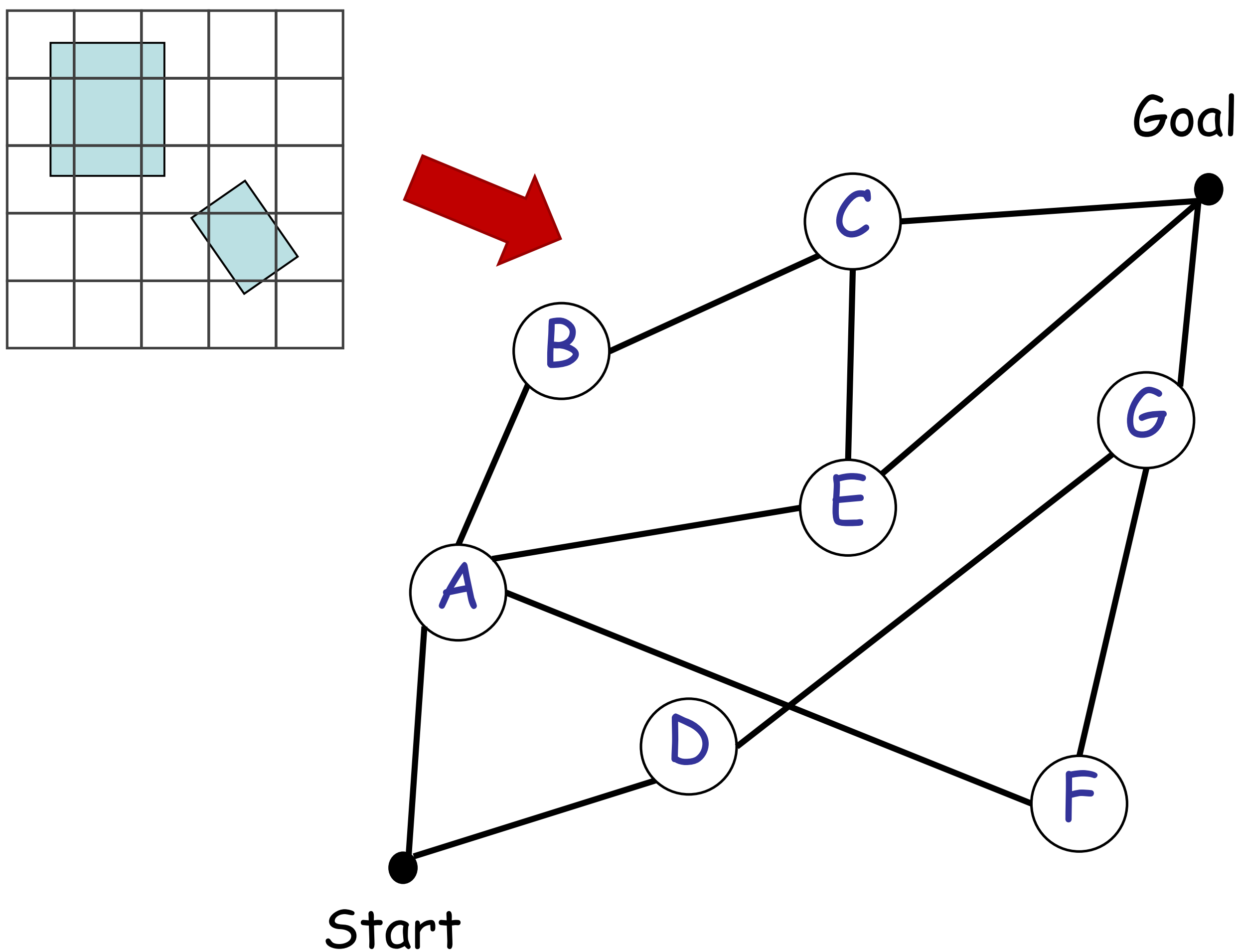
Eight-connected



Grid map inflation



Graph search



Deterministic graph search

Convert the environment map into a connectivity graph

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

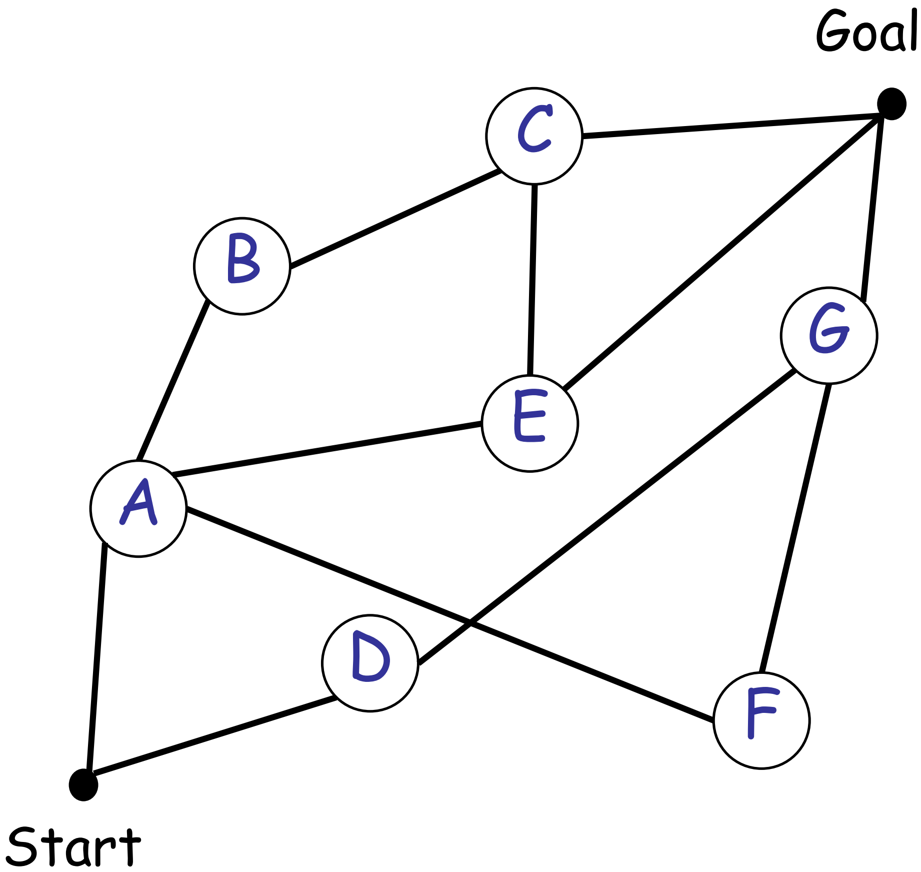
$$f(n) = g(n) + \varepsilon\, h(n)$$

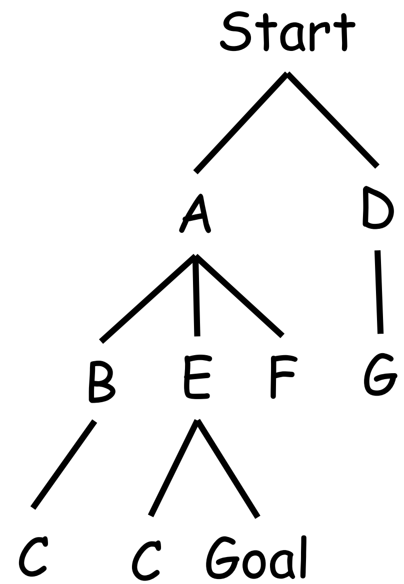
- $f(n)$: Expected total cost
- $g(n)$: Path cost
- $h(n)$: Heuristic cost
- ε: Weighting factor
- n: node/grid cell

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

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

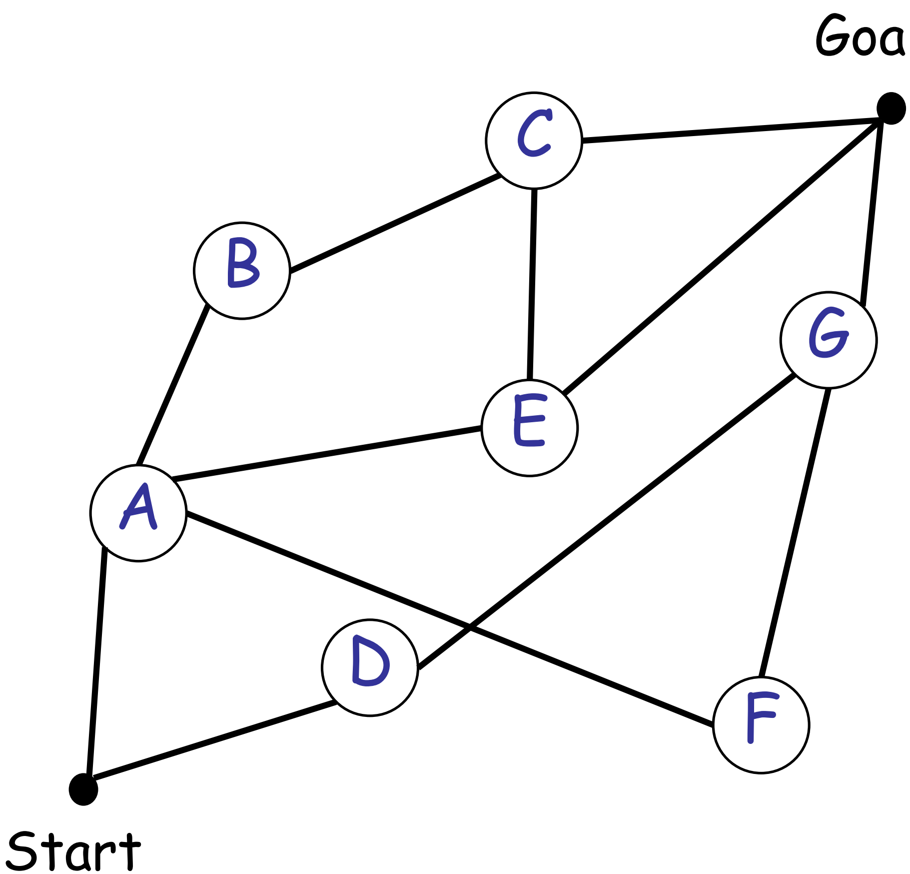
Breadth-first search

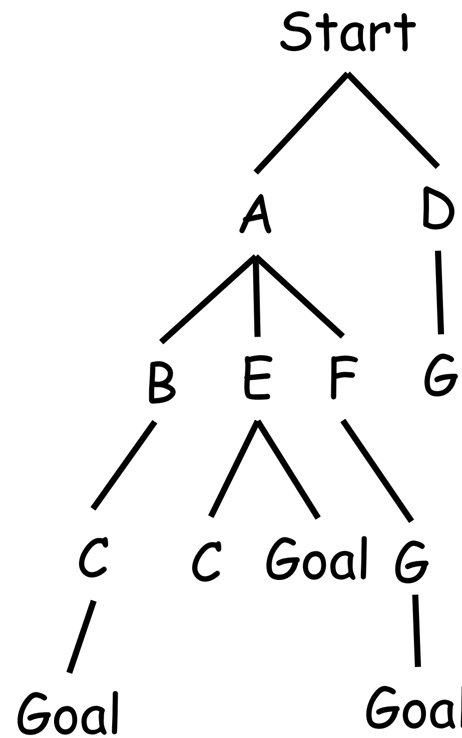




$$f(n) = g(n) \text{ where } c(n, n') = 1$$

Depth-first search





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

Breadth-first search vs depth-first search

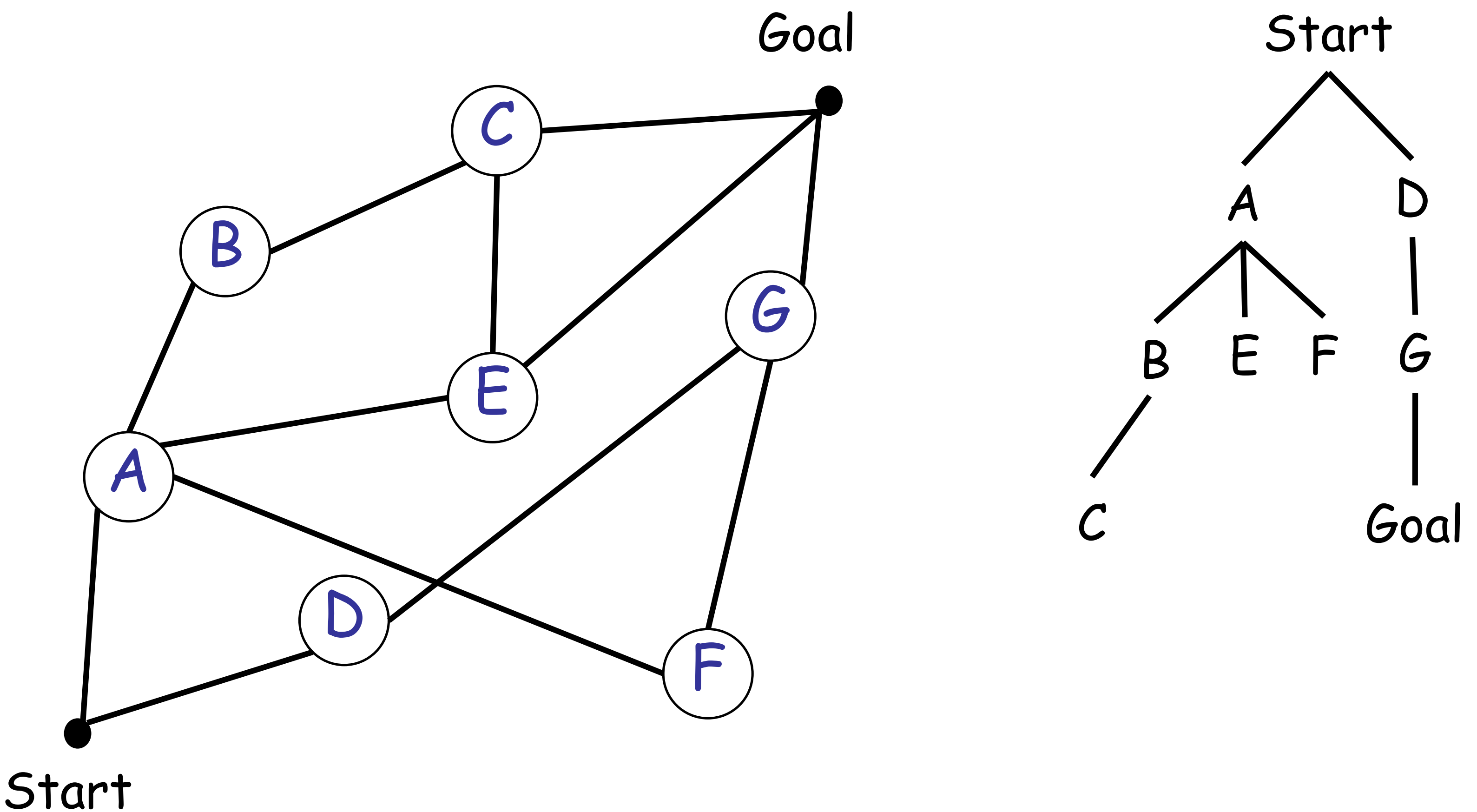
Breadth-first

- Expand all nodes in the order of proximity.
- All paths need to be stored.
- Finds a path has the fewest number of edges between the start and the goal.
- If all edges have the same cost, the solution path is the minimum-cost path.

Depth-first

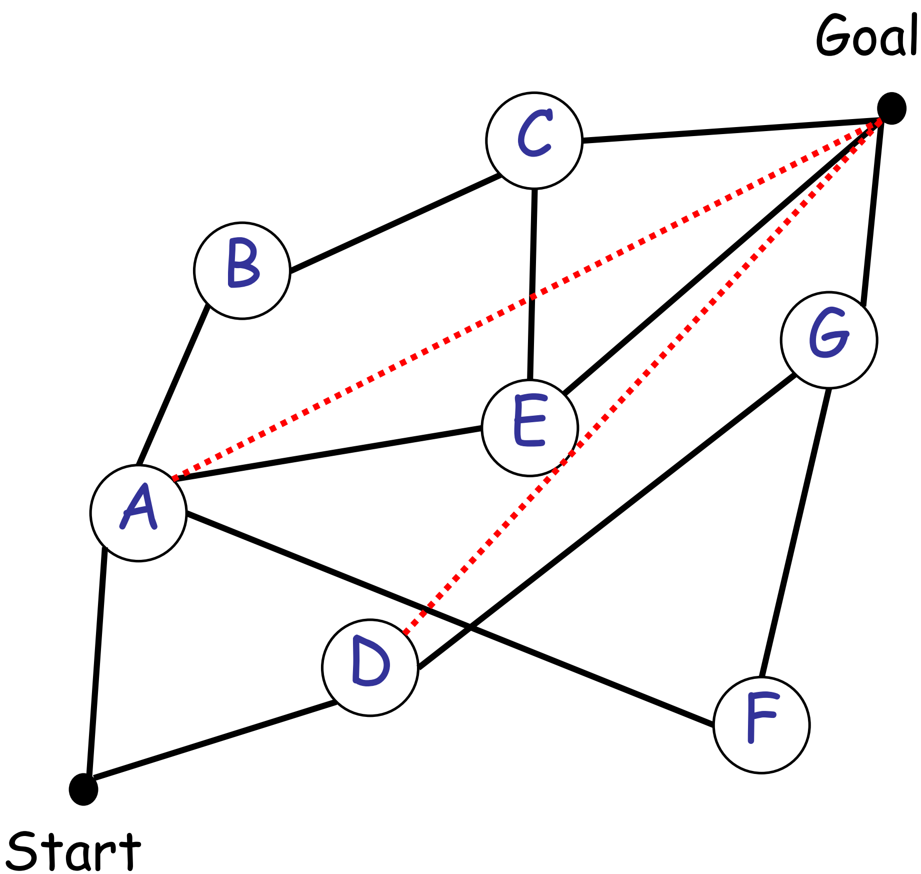
- Expand each node up to the deepest level of the graph first.
- May revisit previously visited nodes or redundant paths.
- Reduction in space complexity: Only need to store a single path.

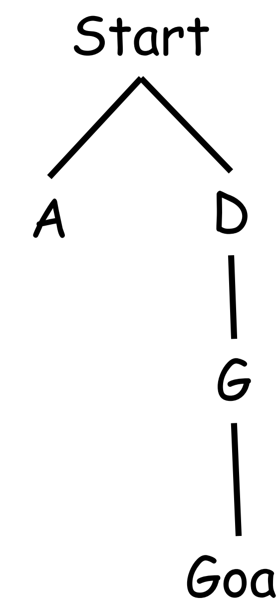
Dijkstra's algorithm



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

A* algorithm



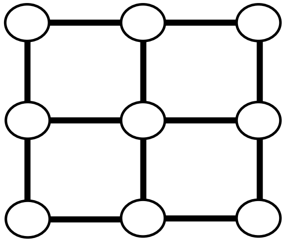


$$f(n) = g(n) + h(n)$$

A* algorithm

```
A*_shortest_path ( map: GRAPH; start_node: NODE; goal_node: NODE )
    local
        c : NODE
    do
        initialize_search ( start_node, goal_node )
        from until is_closed( goal_node ) or not has_open_node loop
            c := open_list.loweset_expected_cost_node
            open_list.remove( c )
            closed_list.add( c )
            if c = goal_node then
                reconstruct_path ( c )
            elseif
                across map.neighboring_nodes( c ) as n loop
                    if not map.is_occupied( n ) and not closed_list.has( n ) then
                        if not open_list.has( n ) then
                            open_list.add( n, c )
                        elseif compute_expected_cost( n, c ) < n.expected_cost then
                            open_list.update( n, c )
                    end
                end
            end
        end
    end
```

A* algorithm: cost computation

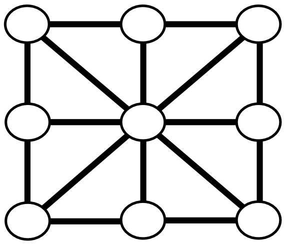


Manhattan distance (4-connected path)

- Path cost $g(n) = g(n') + c(n,n')$
- Edge traversal cost: $c(n,n') = 1$
- Heuristic cost: $h(n) = \#x + \#y$
    - $\#x$ = # of cells between n and goal in x-direction
    - $\#y$ = # of cells between n and goal in x-direction

A* algorithm: cost computation

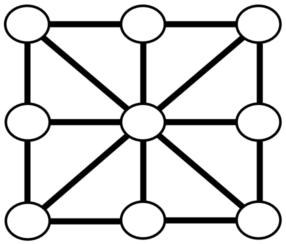
Diagonal distance (8-connected path): *Case 1*



- Path cost g(n) = g(n') + c(n,n')
- Edge traversal cost: c(n,n') = 1
- Heuristic cost: h(n) = max (#x, #y)
  - #x = # of cells between n and goal in x-direction
  - #y = # of cells between n and goal in y-direction

A* algorithm: cost computation

Diagonal distance (8-connected path): *Case 2*



- Path cost $g(n) = g(n') + c(n,n')$

- Edge traversal cost:

  $c(n,n') = 1$ if n is north, south, east, west of n'

  $c(n,n') = \sqrt{2}$ if n is a diagonal neighbor of n'

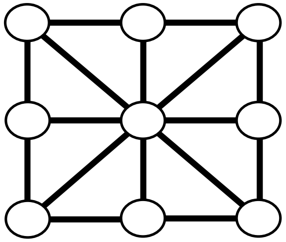
- Heuristic cost:

  $h(n) = (\#y * \sqrt{2} + \#x - \#y)$ if $\#x > \#y$

  $h(n) = (\#x * \sqrt{2} + \#y - \#x)$ if $\#x < \#y$

  - $\#x$ = # of cells between n and goal in x-direction
  - $\#y$ = # of cells between n and goal in y-direction

A* algorithm: cost computation



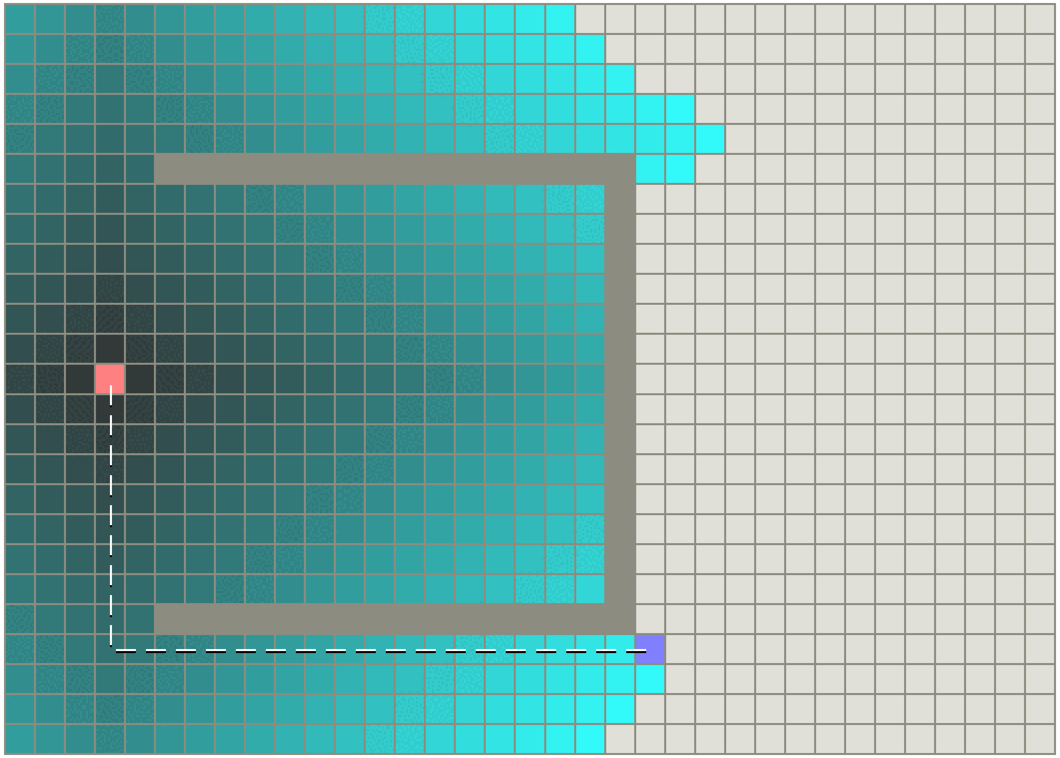
Diagonal distance (8-connected path): *Case 3*

- Path cost $g(n) = g(n') + c(n,n')$
- Edge traversal cost:
  $c(n,n')$ = Euclidean distance
- Heuristic cost: $h(n) = D*\sqrt{(dx*dx + dy*dy)}$
  - $dx = || n.x - goal.x ||$
  - $dy = || n.y - goal.y ||$

A*: heuristic cost and speed

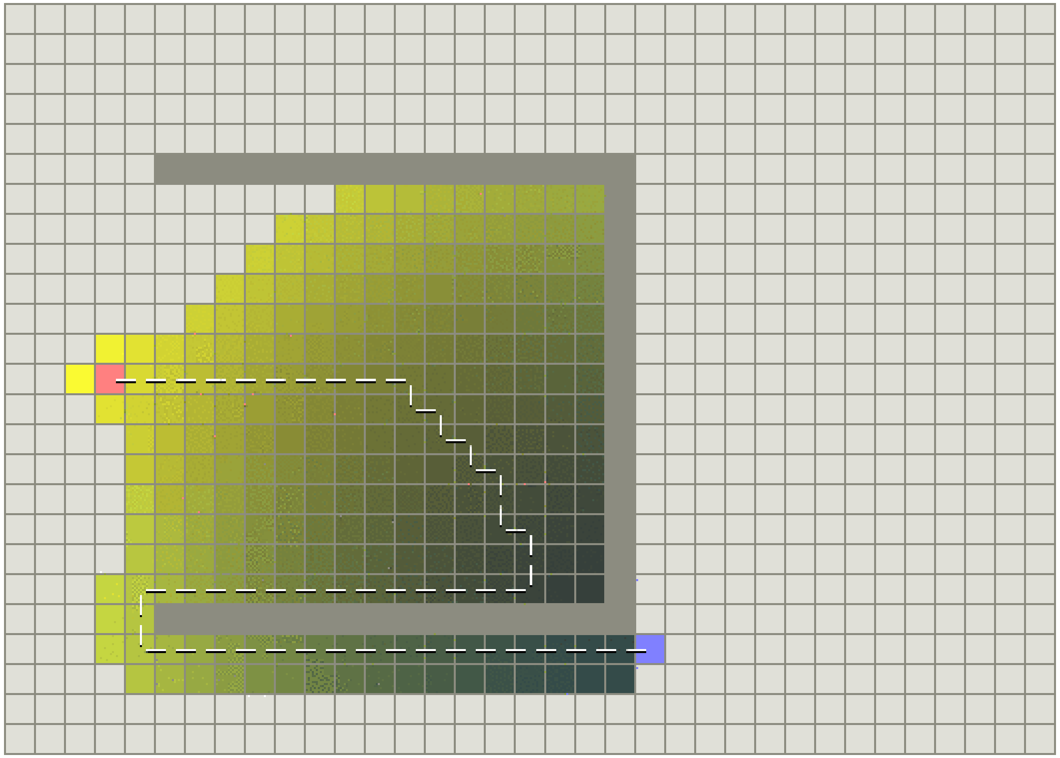
- h(n) <= actual cost from n to goal
  - A* is guaranteed to find a shortest path. The lower h(n) is, the more node A* expands, making it slower.
  - h(n) = 0, then we have Dijkstra's algorithm
- h(n) = actual cost from n to goal
  - A* will only follow the best path and never expand anything else, making it very fast.
- h(n) > actual cost from n to goal
  - A* is not guaranteed to find a shortest path, but it can run faster.
  - h(n) >> g(n), then we have Greedy Best-First-Search: selects vertex closest to the goal

Dijkstra's algorithm



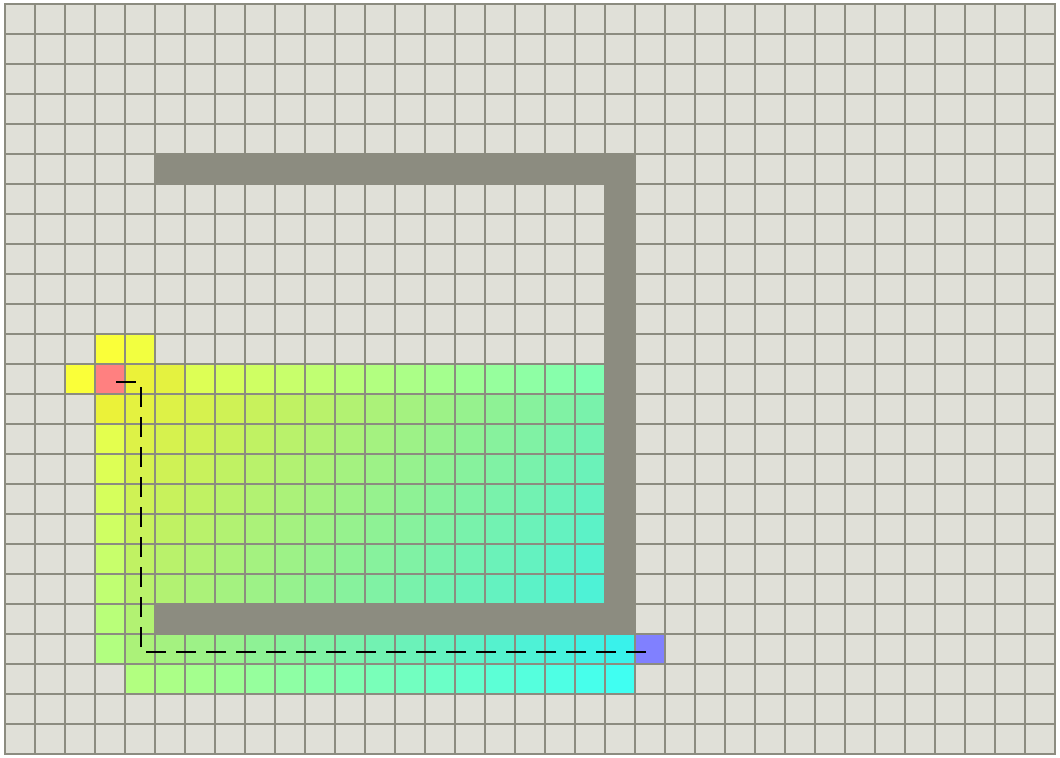
http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

Greedy best-first search



http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

A* algorithm



http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

Randomized graph search

http://msl.cs.uiuc.edu/rrt/gallery_2drrt.html

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

Potential 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

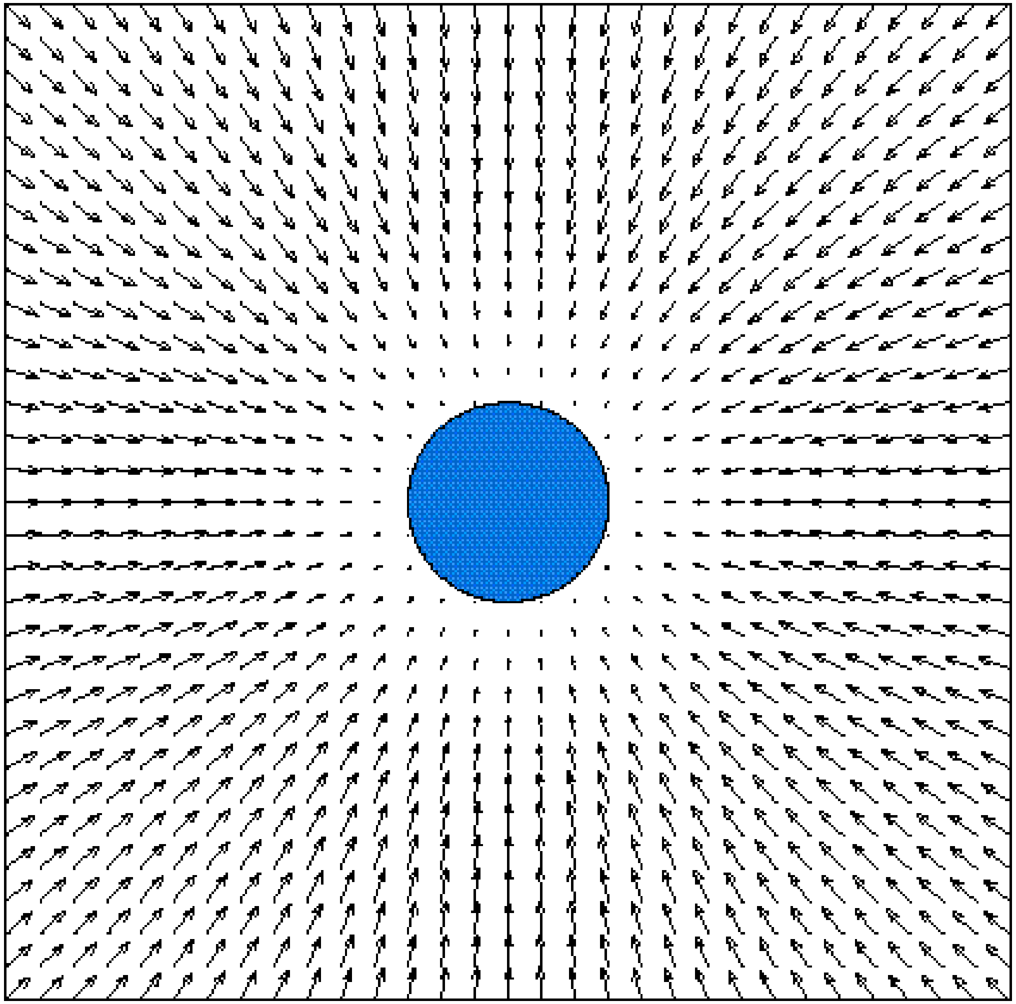


Image from lecture notes by Benjamin Kuipers

Repulsive potential

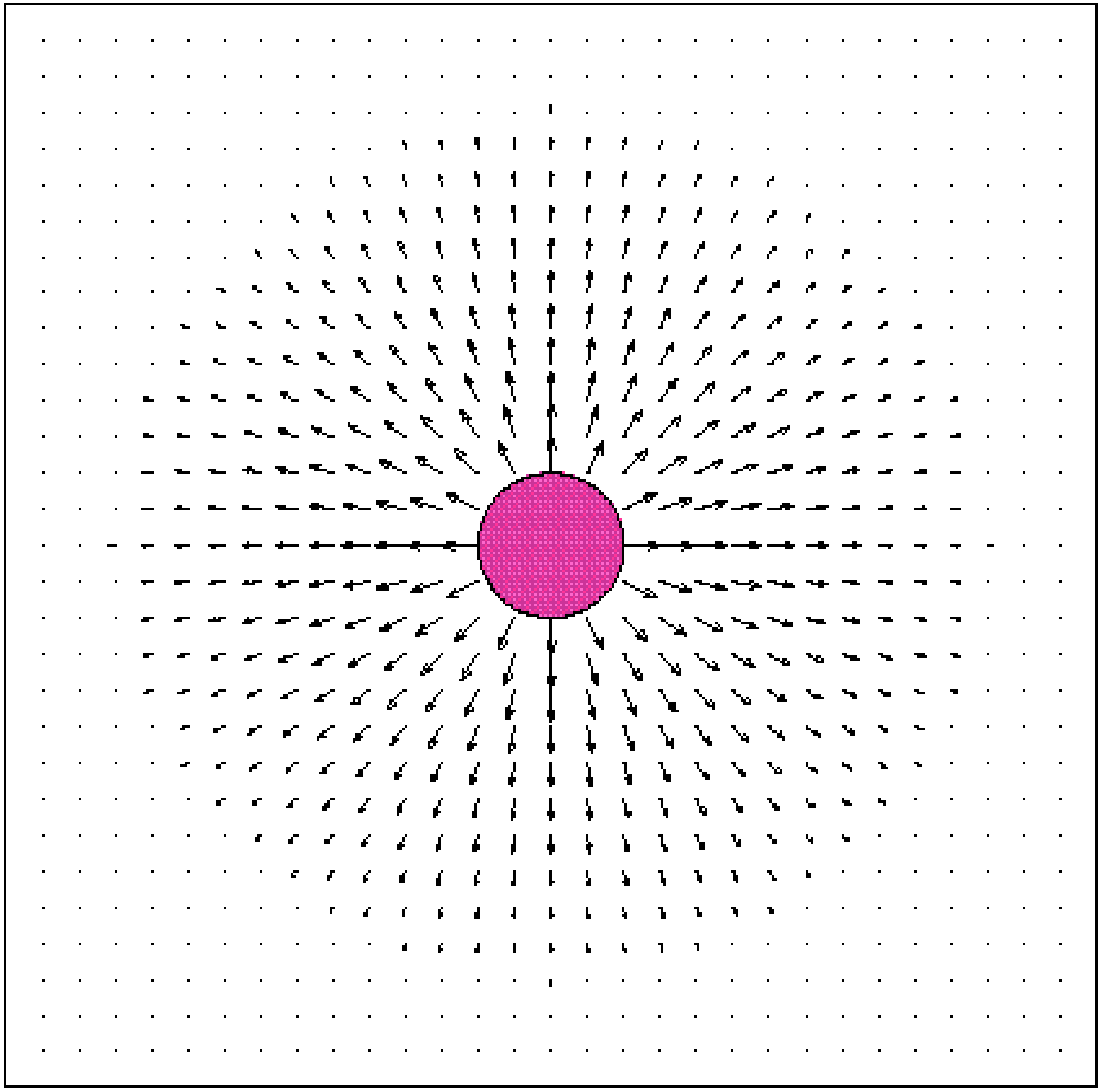


Image from lecture notes by Benjamin Kuipers

Sum of two fields

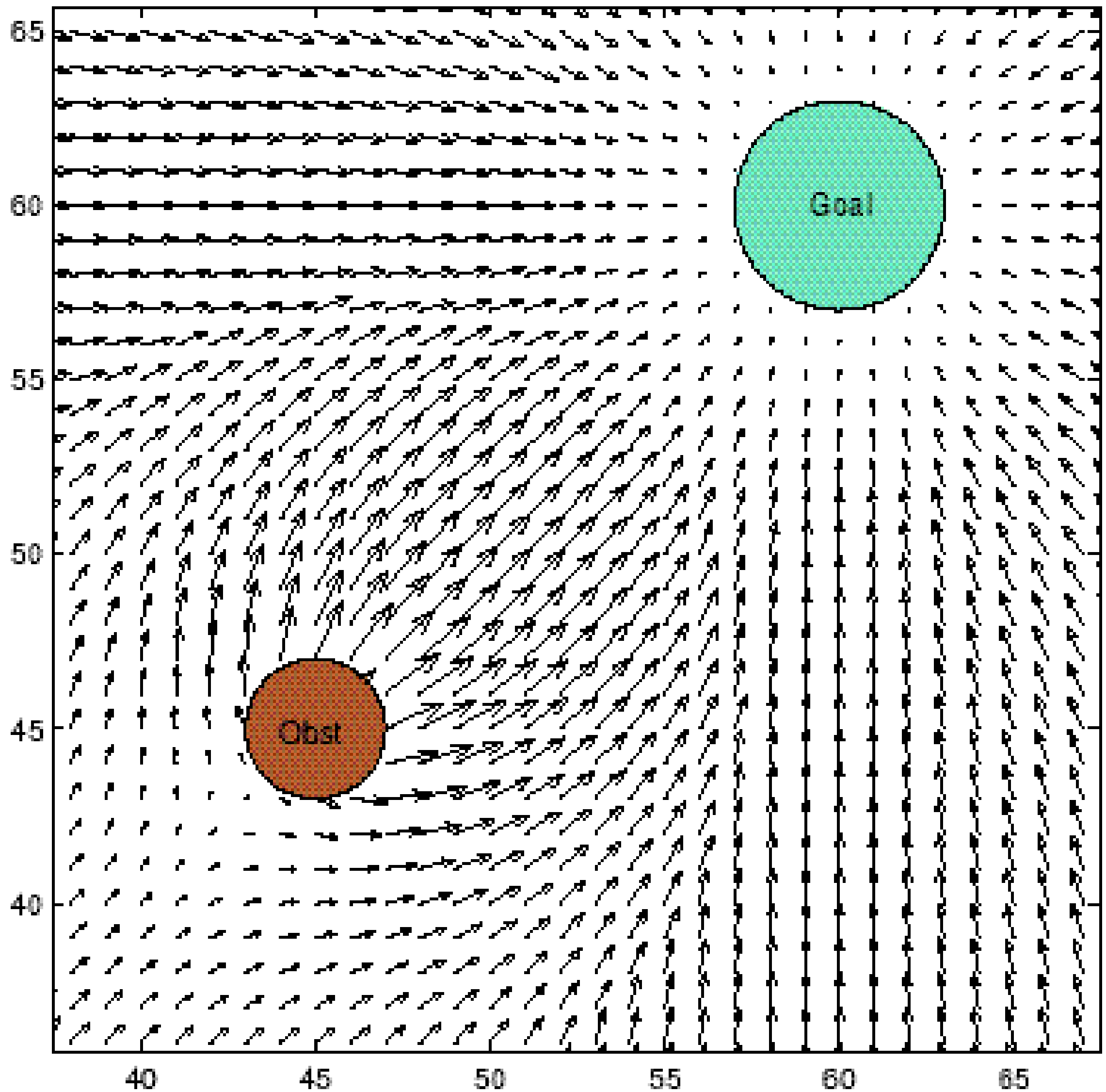


Image from lecture notes by Benjamin Kuipers

Resulting path

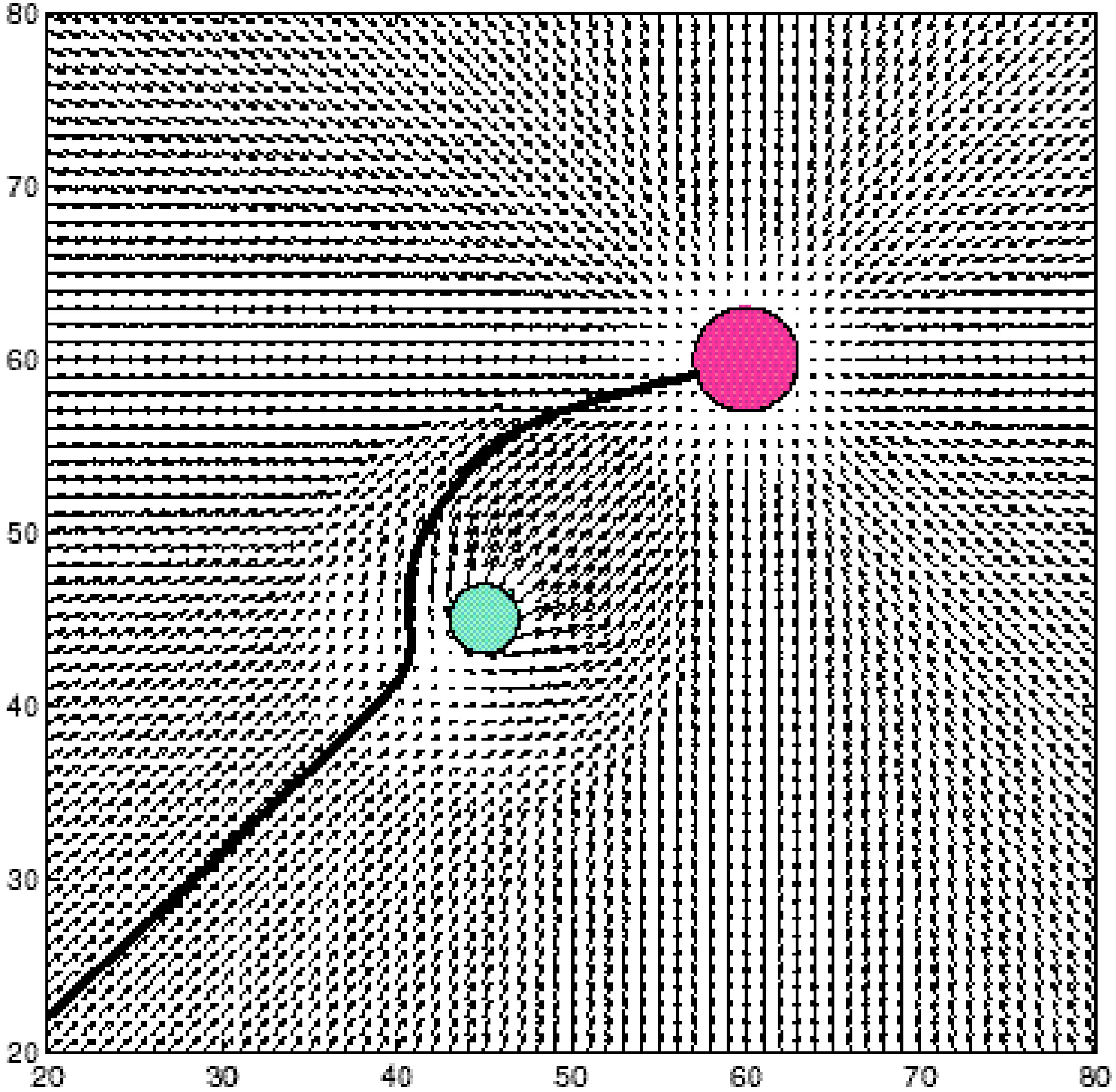


Image from lecture notes by Benjamin Kuipers

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

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

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

- 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