UTS:CAS

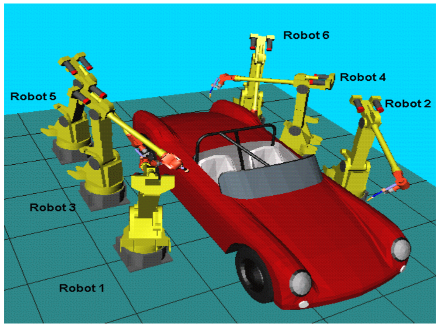
CENTRE FOR AUTONOMOUS SYSTEMS

49274 ADVANCED ROBOTICS
PATH PLANNING

TERESA VIDAL CALLEJA

UNIVERSITY OF TECHNOLOGY SYDNEY

PATH PLANNING APPLICATIONS



Industrial

- painting, welding, sand blasting, loading/unloading machines, assembling parts

Servicing stores, warehouses, and factories

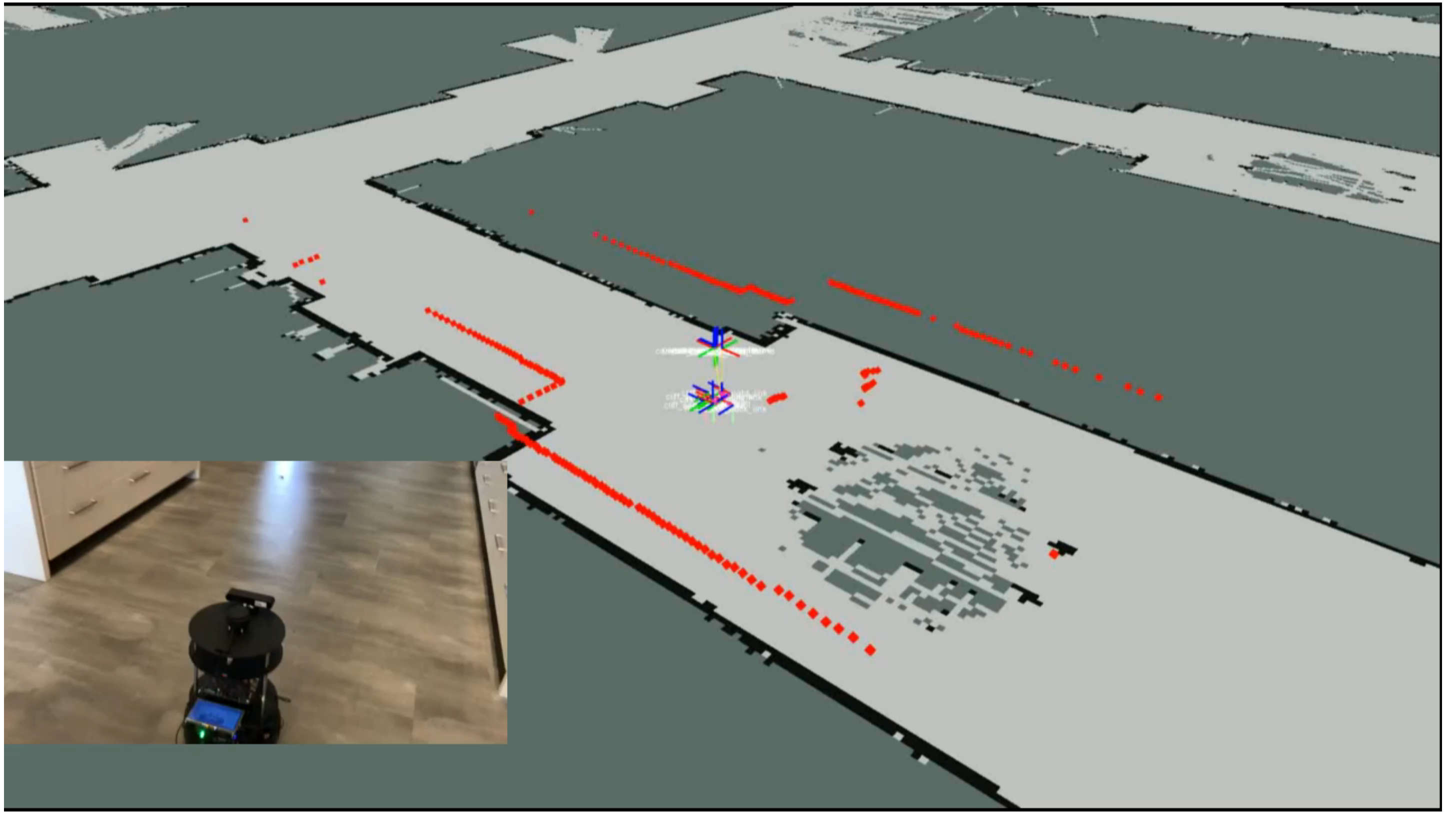
- maintaining, surveying, cleaning, transporting objects

Exploring unknown areas

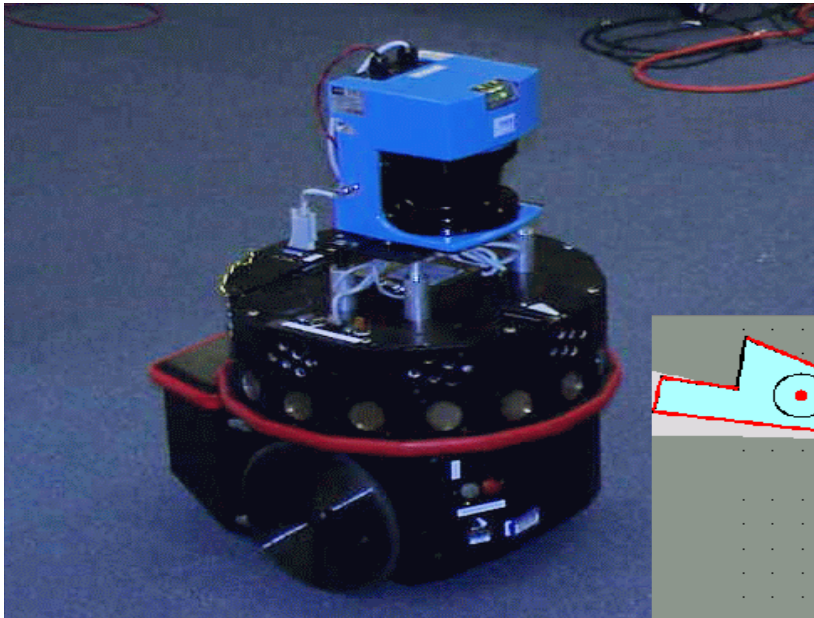
- building a map, extracting samples
- search and rescue operations

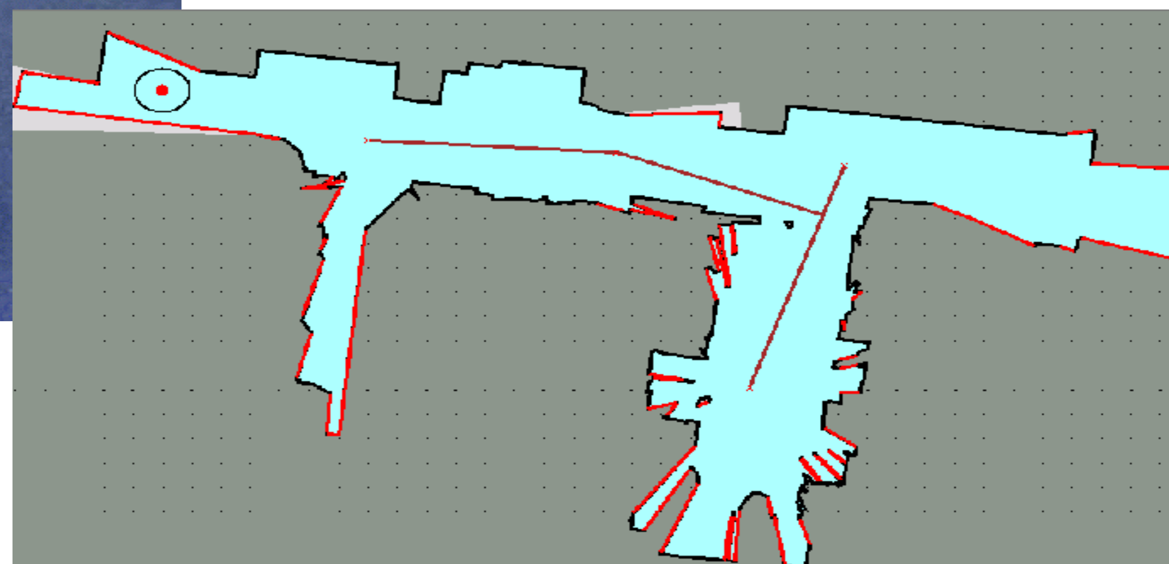
Assisting people in offices, public areas, and homes

NAVIGATION: GO TO MY DESK



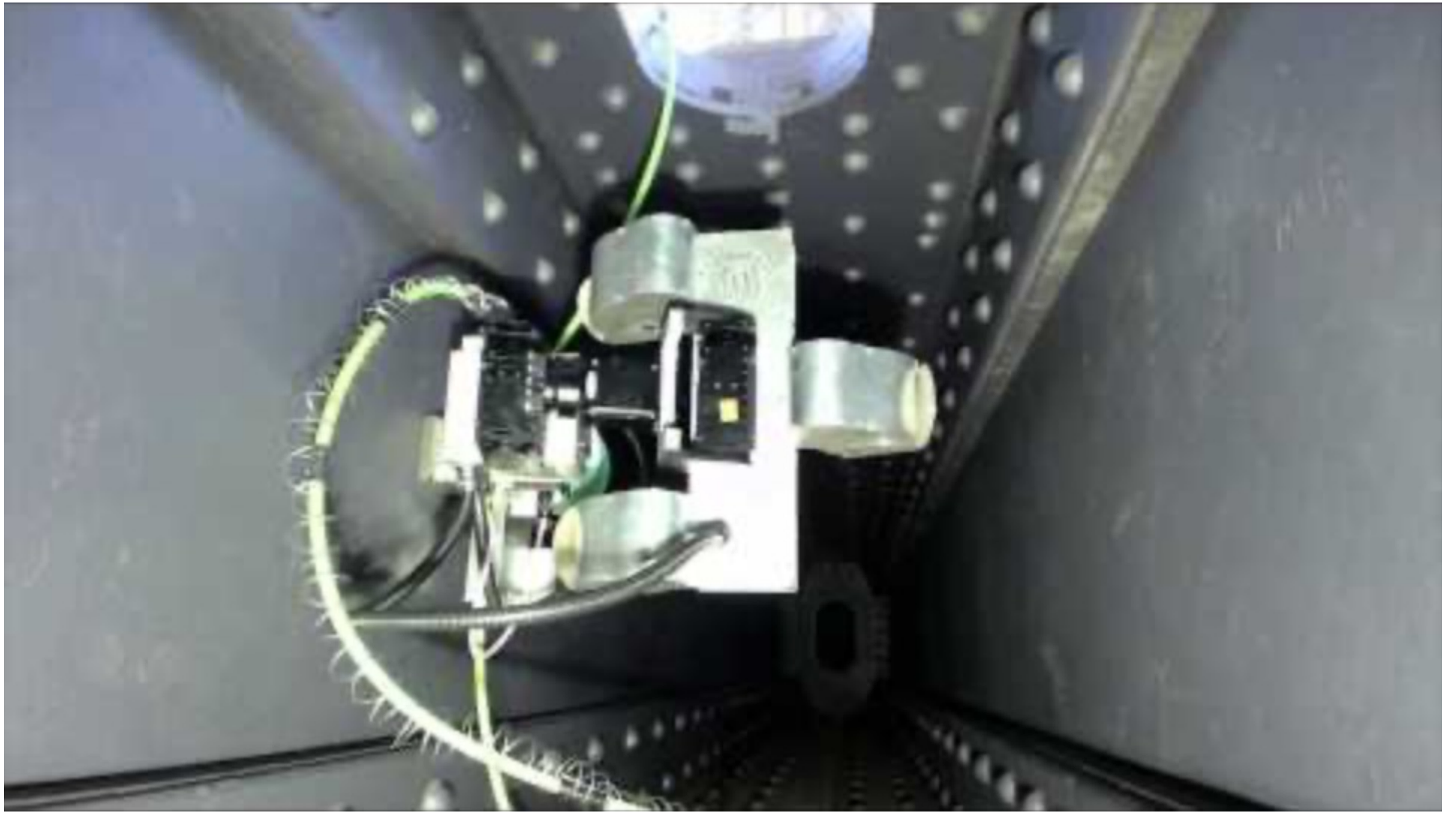
MAPPING: PLAN WHERE TO MOVE NEXT?



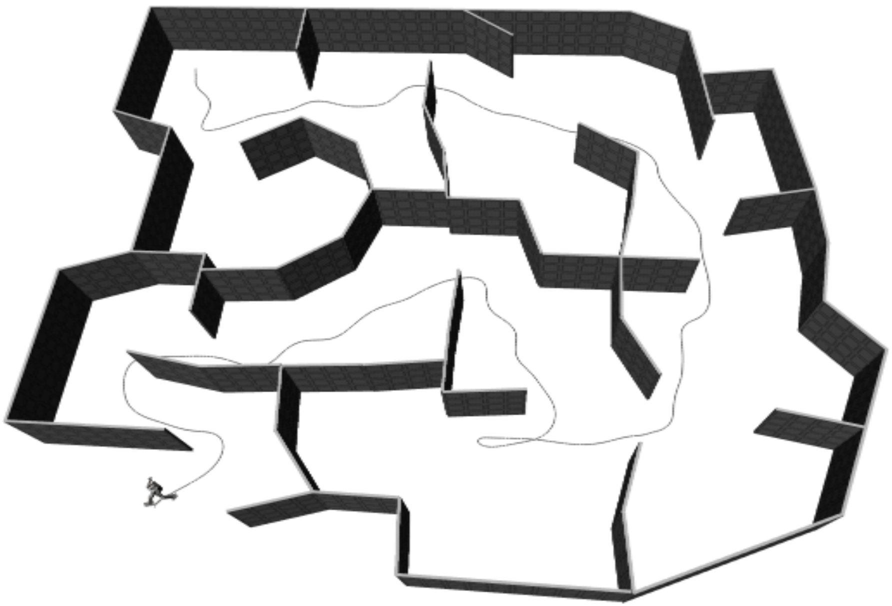


HUMANOID: MOTION PLANNING

BRIDGE INSPECTION



PATH PLANNING DEFINITION

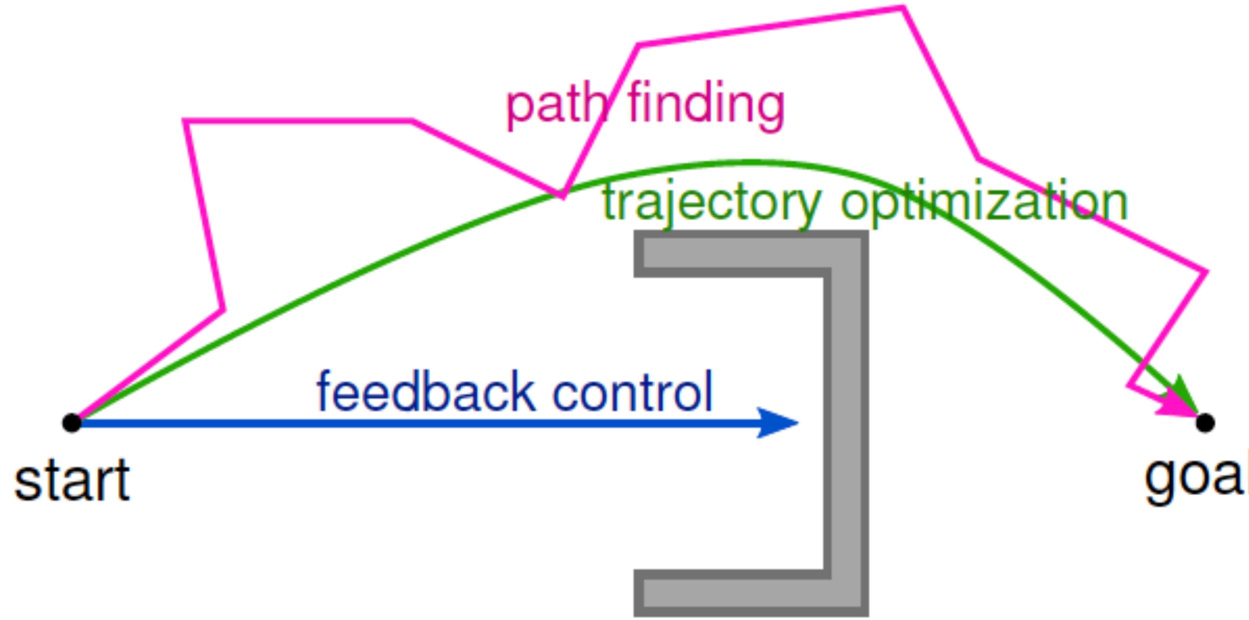


Path Planning

- Given the map, the current robot location, and the goal location

- Path planning involves identify a trajectory that will cause the robot to reach the goal location when executed

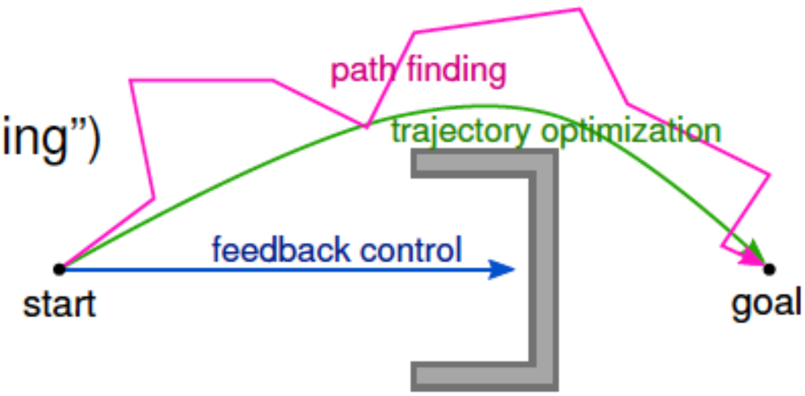
Feedback control, path finding, trajectory optim.



- Feedback Control: E.g., $q_{t+1} = q_t + J^\sharp(y^* - \phi(q_t))$
- Trajectory Optimization: $\mathrm{argmin}_{q_{0:T}} f(q_{0:T})$
- Path Finding: Find some $q_{0:T}$ with only valid configurations

Control, path finding, trajectory optimization

- Combining methods:
  1) Path Finding
  2) Trajectory Optimization ("smoothing")
  3) Feedback Control



- Many problems can be solved with only feedback control (though not optimally)
- Many more problems can be solved *locally* optimal with only trajectory optimization
- Tricky problems need path finding: *global* search for valid paths

OUTLINE

Move from A to B with/without obstacles

Configuration Space

Discretisation

- Wavefront Algorithm
- Dijkstra / A*
- Visibility Graph method
- Potential Fields

Sample-based Path Finding

- Probabilistic road maps (PRMs)
- Rapidly-exploring random trees (RRT)

MOVING FROM A TO B: NO OBSTACLES

A simple discrete-time robot motion model

$$
\begin{aligned}
x(k+1) &= x(k) + v(k)\Delta T \cos[\phi(k)] \\
y(k+1) &= y(k) + v(k)\Delta T \sin[\phi(k)] \\
\phi(k+1) &= \phi(k) + \gamma(k)\Delta T
\end{aligned}
$$

$x(k), y(k)$ — robot position at time k

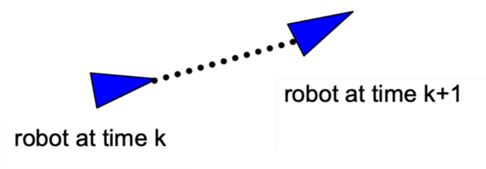
$\phi(k)$ — robot orientation at time k

Control

$v(k)$ — velocity at time k

$\gamma(k)$ — turning rate at time k

ΔT — time interval from step k to step $k+1$



It is obtained from a direct discretization of

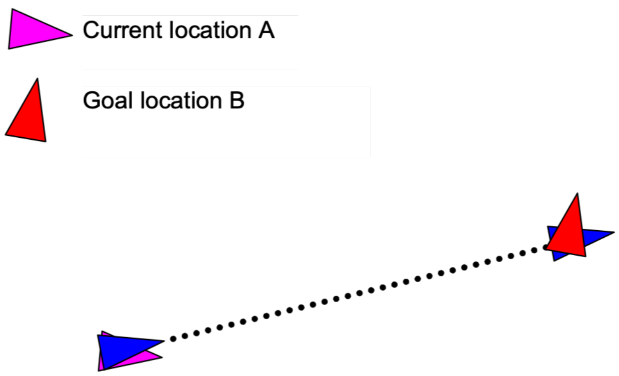
$$
\begin{aligned}
\dot{x} &= v \cos \phi \\
\dot{y} &= v \sin \phi \\
\dot{\phi} &= \gamma
\end{aligned}
$$

MOVING FROM A TO B: WITH OBSTACLES

A simple control strategy:

Rotate → move forward → rotate

- Rotate (e.g. zero velocity, constant turning rate)
- Move forward (e.g. constant velocity, zero turning rate)



A simple discrete-time robot motion model

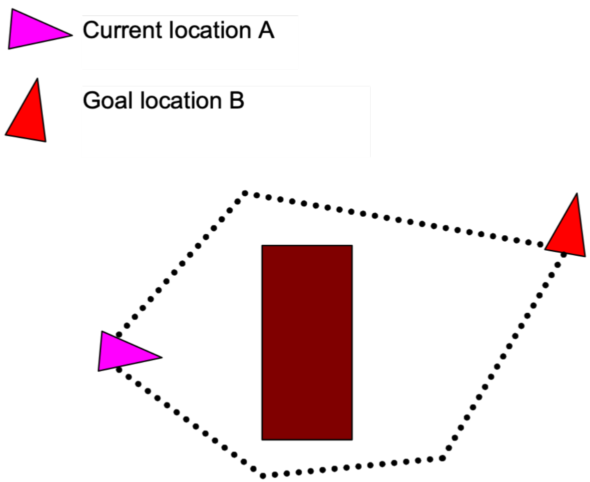
$$
\begin{aligned}
x(k+1) &= x(k) + v(k)\Delta T \cos[\phi(k)] \\
y(k+1) &= y(k) + v(k)\Delta T \sin[\phi(k)] \\
\phi(k+1) &= \phi(k) + \gamma(k)\Delta T
\end{aligned}
$$

MOVING FROM A TO B: WITH OBSTACLES

Rotate -- move forward – rotate

Control problem becomes complicated

– we prefer shorter path than longer path



A simple discrete-time robot motion model

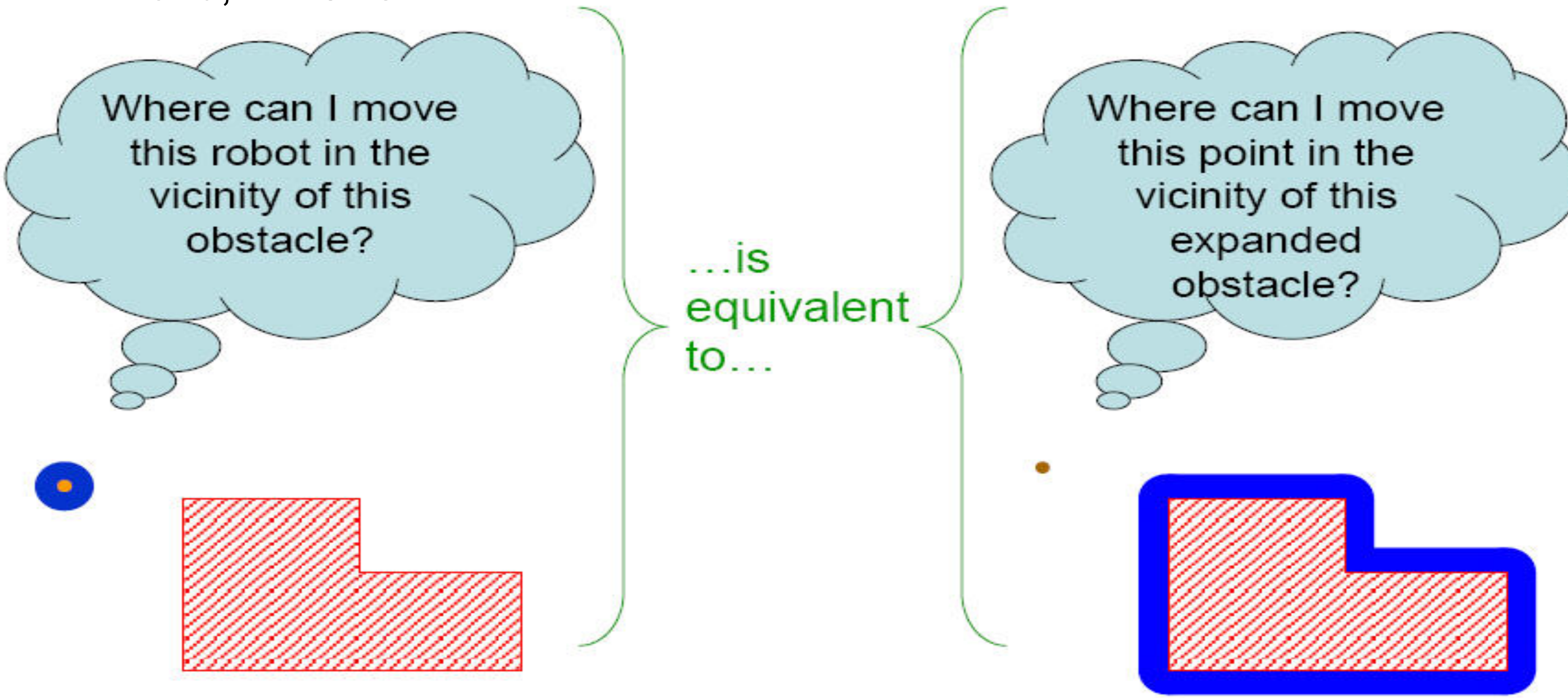
$$
\begin{aligned}
x(k+1) &= x(k) + v(k)\Delta T \cos[\phi(k)] \\
y(k+1) &= y(k) + v(k)\Delta T \sin[\phi(k)] \\
\phi(k+1) &= \phi(k) + \gamma(k)\Delta T
\end{aligned}
$$

CONFIGURATION SPACE

- Although the motion planning problem is defined in the regular world, it lives in another space: **the configuration space**
- A robot configuration q is a specification of the positions of all robot points relative to a fixed coordinate system
- Usually a configuration is expressed as a **vector of positions** and **orientations**
- The configuration space (C-space) is the **space of all possible configurations**
- The topology of C-space is usually **not the Cartesian space**

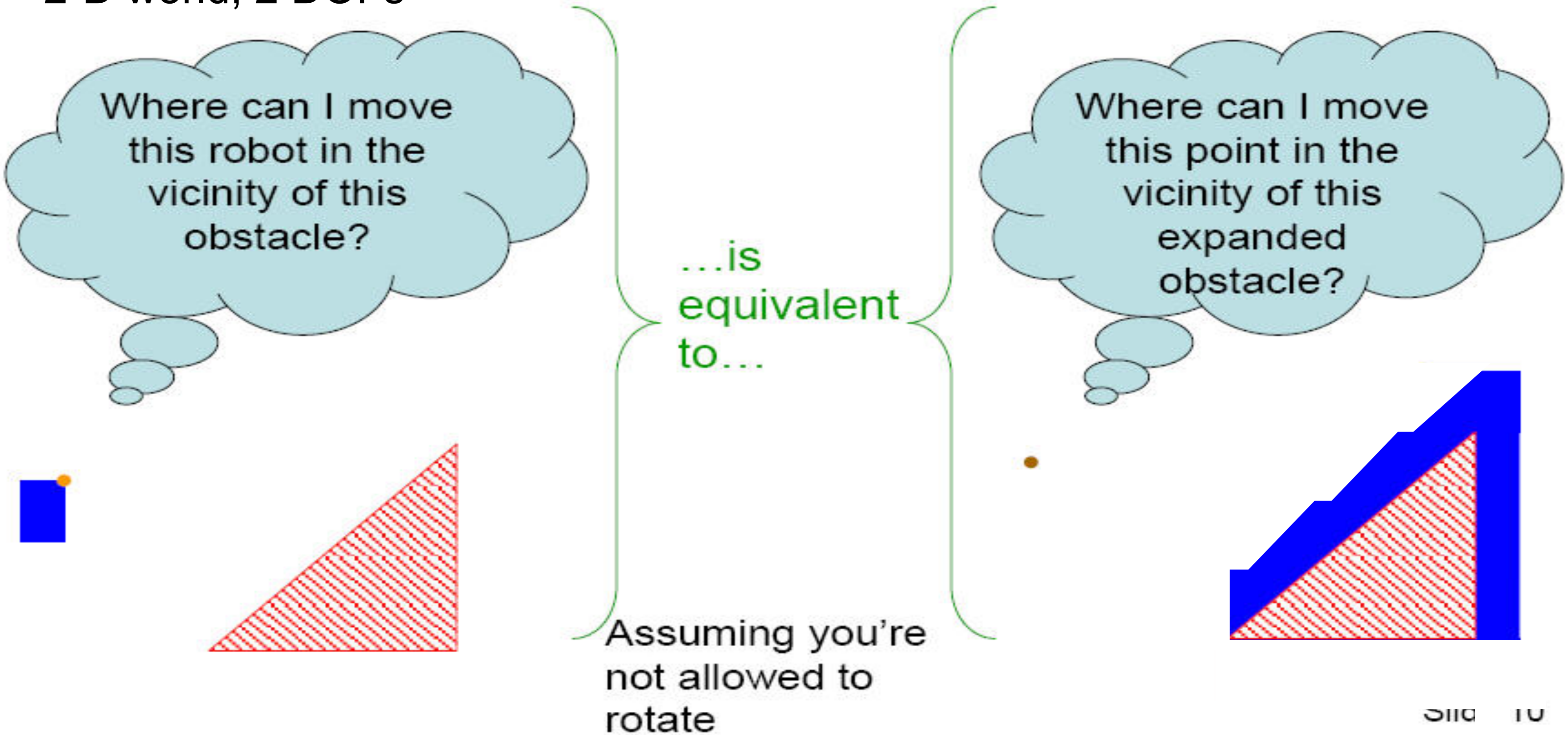
CONFIGURATION SPACE EXAMPLE

2-D world, 2 DOFs



CONFIGURATION SPACE EXAMPLE

2-D world, 2 DOFs



CONFIGURATION SPACE

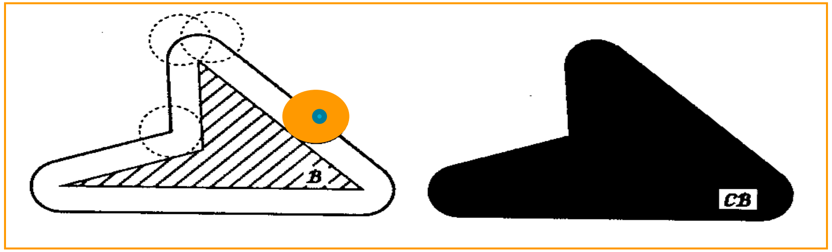
Construct a configuration space (reduces the robot to a point)

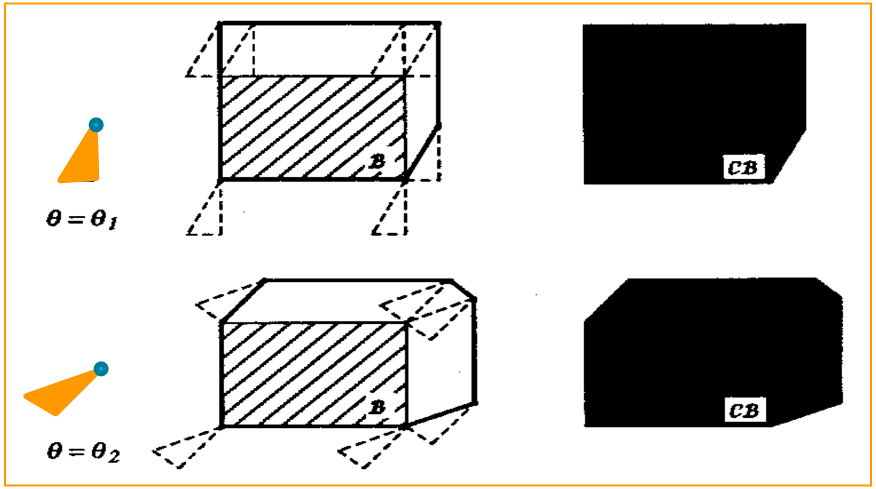
C = the configuration space of the robot

= {all possible configuration of the robot}

C-free = the free configuration space { Robot doesn't collide with the obstacles}

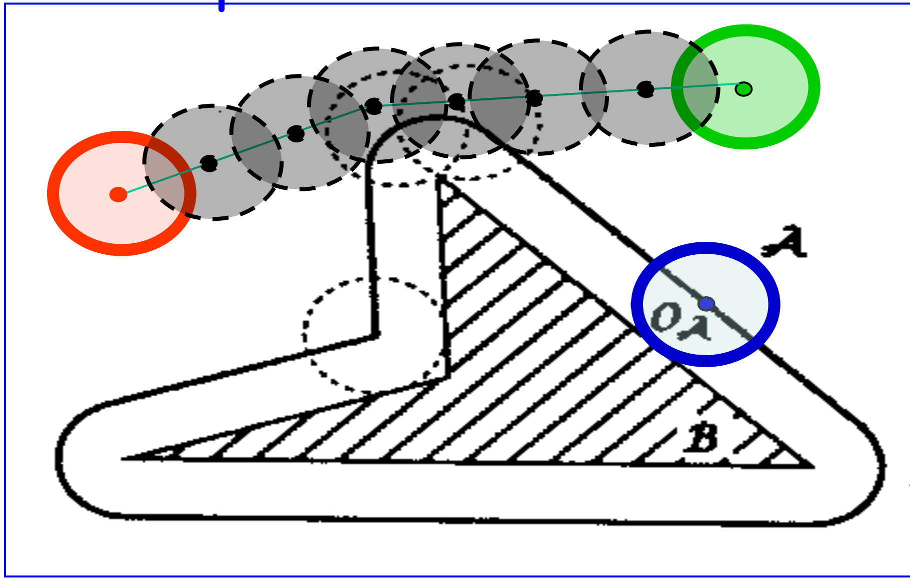
C-obs = C – C-free



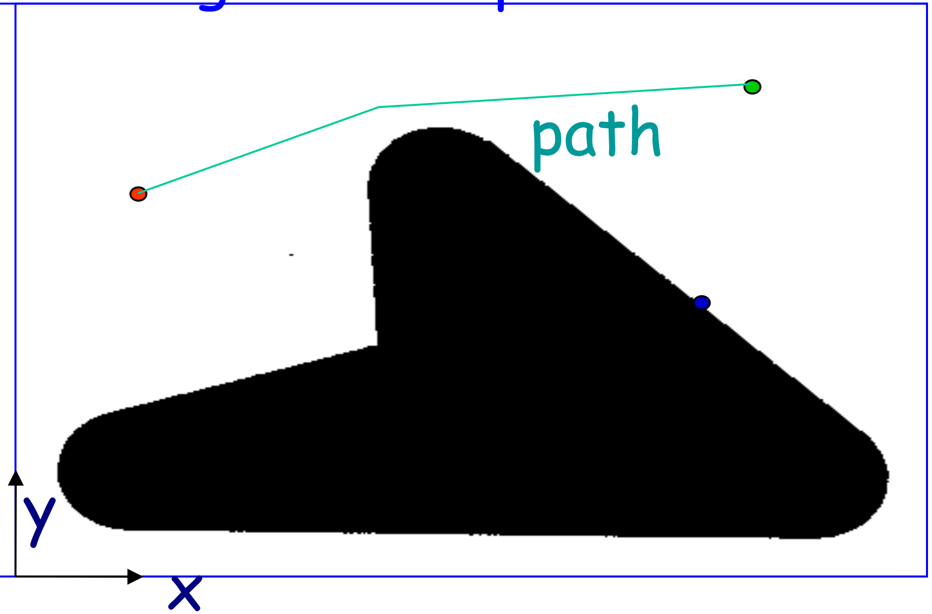


CONFIGURATION SPACE OF A DISK

Workspace W



Configuration space C



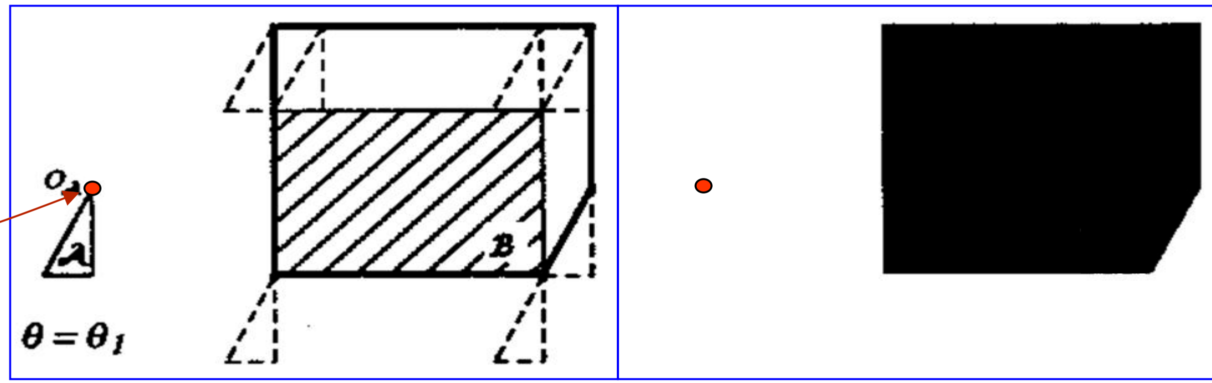
configuration = coordinates (x,y) of robot's centre

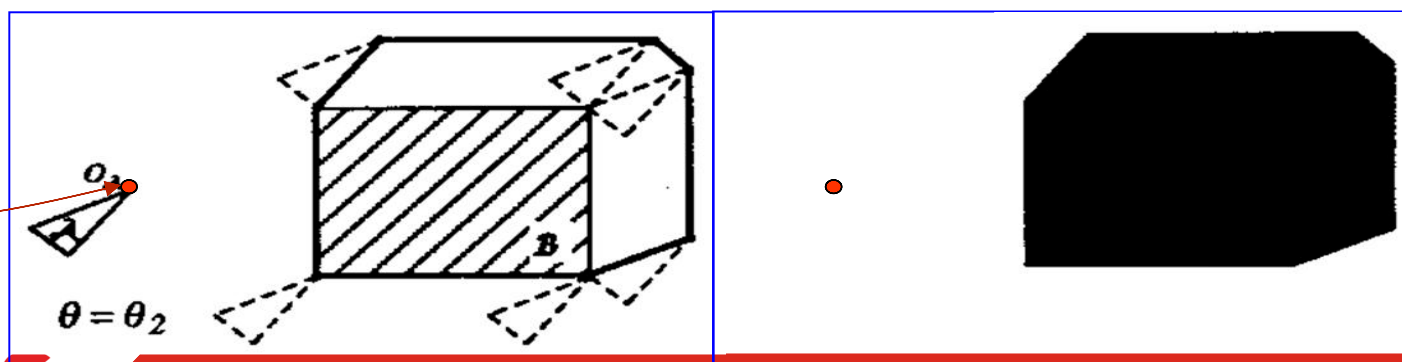
configuration space C = {(x,y)}

free space F = subset of collision-free configurations

CONFIGURATION SPACE OF A TRANSLATION POLYGON

reference point



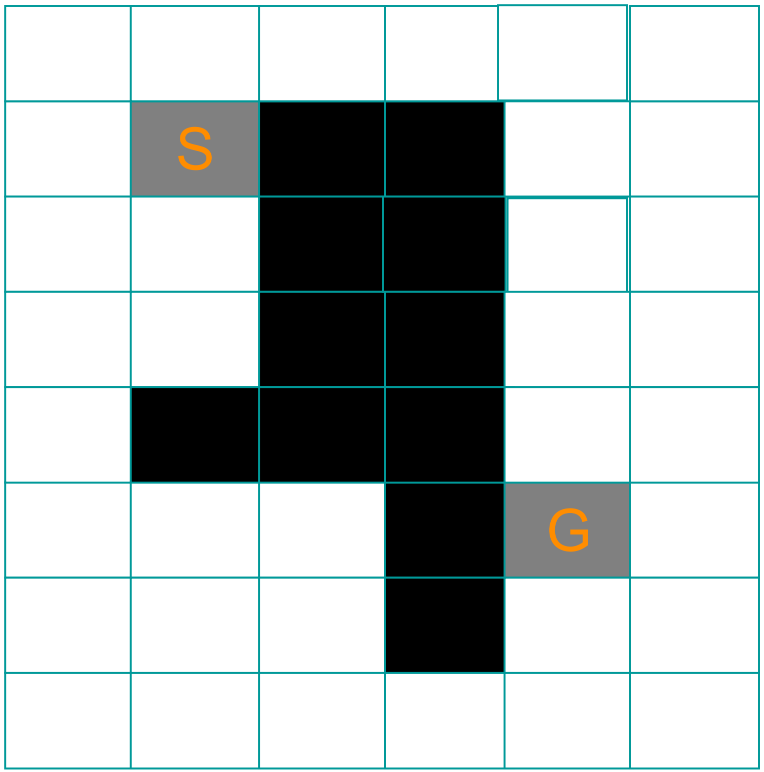


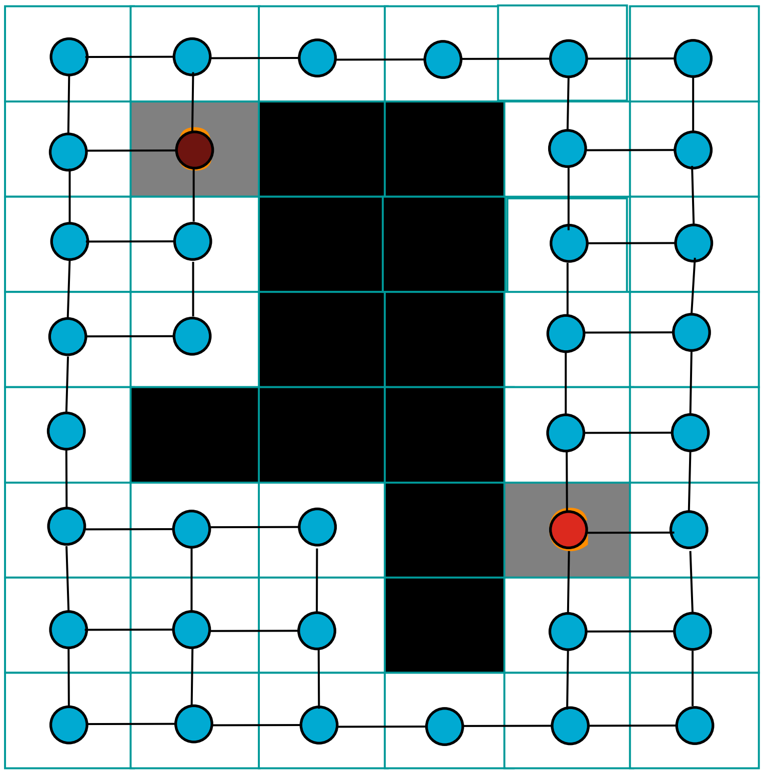
DISCRETISATION METHODS

Cell Decomposition

- Decompose the environment into a number of disjoint cells
- Approximate the obstacles and free-space with cells that have a simple shape like, for example, rectangles
- Build a graph

DISCRETISATION METHODS





WAVEFRONT ALGORITHM

Motivated by the water waves

Step 1: Create a discretised map using cell decomposition

Step 2: Add in start and goal locations

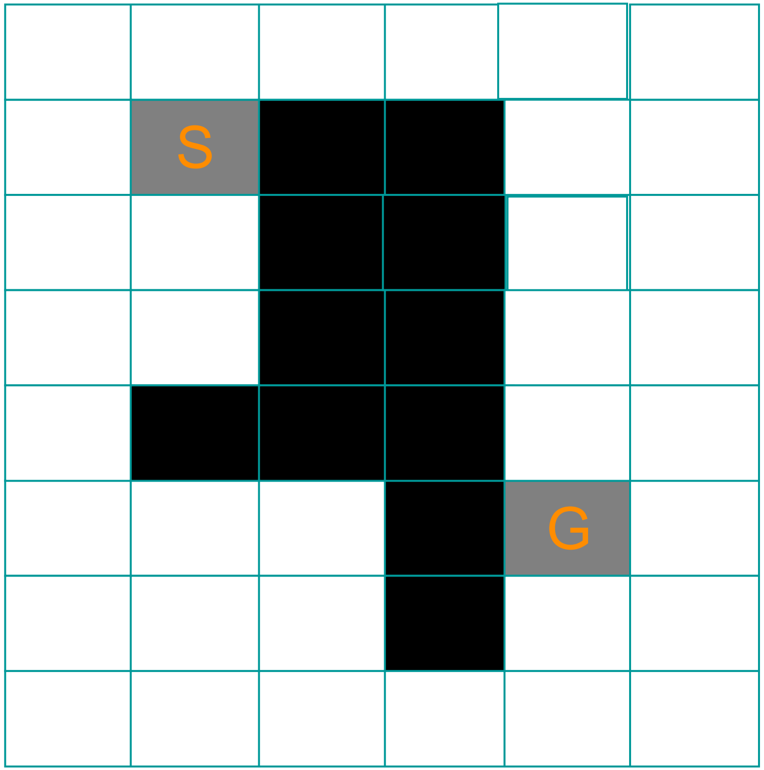
Step 3: Fill in the wavefront table

Step 4: Get the obstacle-free path from the wavefront table

(many different ways for implementation)

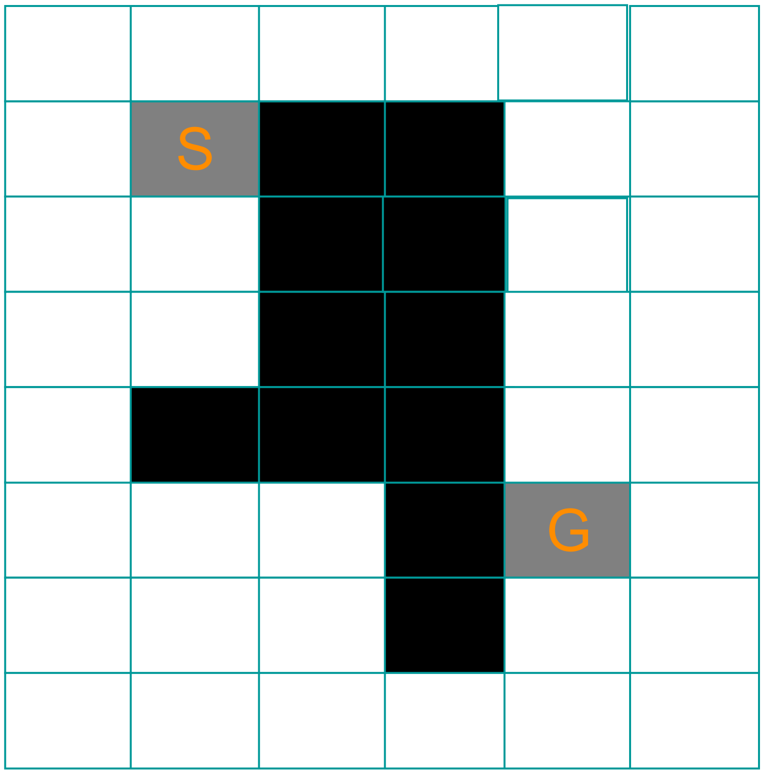
WAVEFRONT ALGORITHM

S– start, G--goal



WAVEFRONT ALGORITHM

S– start, G--goal

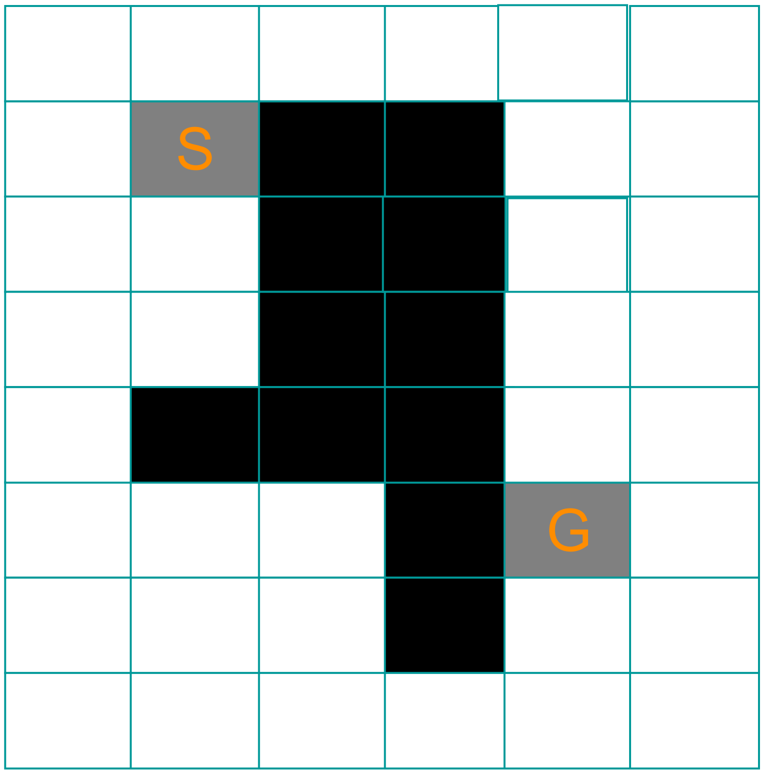


Build the wave (start from goal)

8	7	6	5	5	5
8	S(7)			4	4
7	7			3	3
6	6			2	2
5				1	1
5	4	4		G(0)	1
5	4	3		1	1
5	4	3	2	2	2

WAVEFRONT ALGORITHM

S– start, G--goal

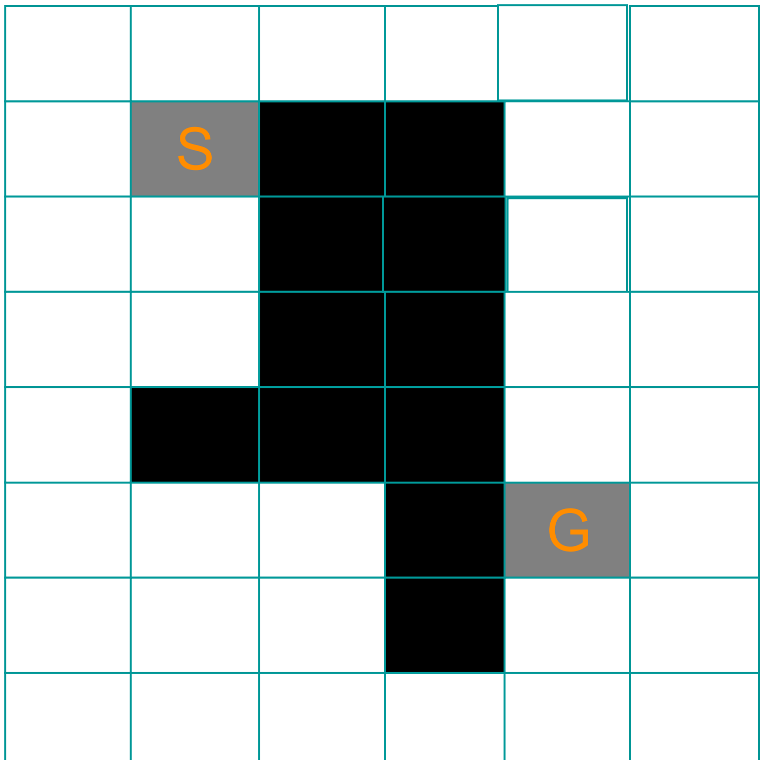


Get the path (reverse order)

8	7	6	5	5	5
8	S(7)			4	4
7	7			3	3
6	6			2	2
5				1	1
5	4	4		G(0)	1
5	4	3		1	1
5	4	3	2	2	2

WAVEFRONT ALGORITHM

S– start, G--goal



If moving diagonally is not allowed

9	8	7	6	5	6
10	S(9)			4	5
11	12			3	4
10	11			2	3
9				1	2
8	7	6		G(0)	1
7	6	5		1	2
6	5	4	3	2	3

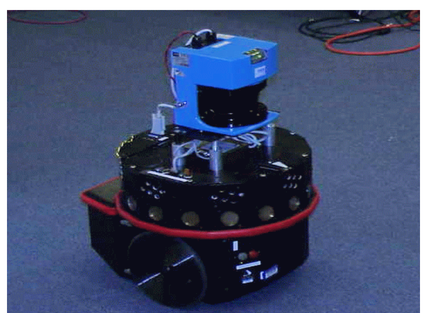
WAVEFRONT - CONFIGURATION SPACE

When planning the path in the wavefront algorithm example, the size of the robot was not taken into account

The size of the robot might be significantly larger than a cell!

How to guarantee collision free?

– use configuration space



Get the path (reverse order)

8	7	6	5	5	5
8	S(7)			4	4
7	7			3	3
6	6			2	2
5				1	1
5	4	4		G(0)	1
5	4	3		1	1
5	4	3	2	2	2

A GENERAL SHORTEST PATH PROBLEM

In the wavefront algorithm example, the cost for moving diagonally is assumed to be the same as that of moving vertically or horizontally

A more general Shortest Path Problem:

Given a connected graph *G*=(V,E), a cost function d:E→R+ and two fixed vertex s and g in V, find a shortest path from s to g

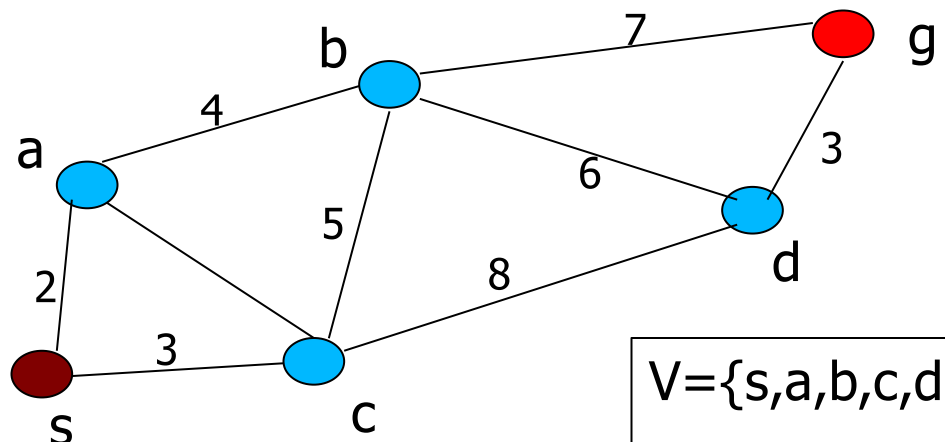
Get the path (reverse order)

8	7	6	5	5	5
8	S(7)			4	4
7	7			3	3
6	6			2	2
5				1	1
5	4	4		G(0)	1
5	4	3		1	1
5	4	3	2	2	2

A GENERAL SHORTEST PATH PROBLEM

Shortest Path Problem:

Given a connected graph G=(V,E), a cost function d:E→R+ and two fixed vertex s and g in V, find a shortest path from s to g (the path with least cost)



V={s,a,b,c,d,g} ---- vertices

E={(s,a),(s,c),…,(d,g)} ---- edges

d((s,a))=2 ---- cost of edge (s,a)

DIJKSTRA'S ALGORITHM

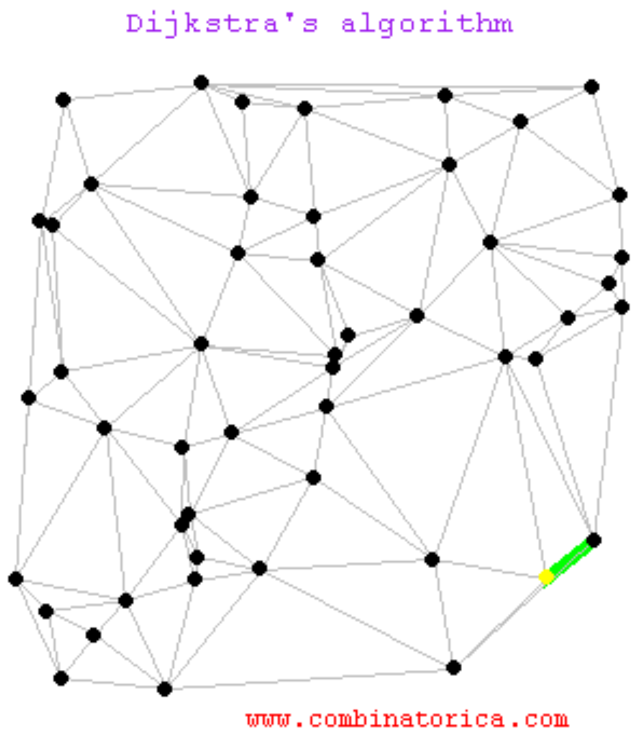
An algorithm that solves the general Shortest Path Problem:

1. Create a distance list (cost list), a previous vertex list, a visited list, and a current vertex
2. All the values in the distance list are set to infinity except the starting vertex which is set to zero
3. All values in visited list are set to false
4. All values in the previous list are set to a special value signifying that they are undefined, such as null
5. Current vertex is set as the starting vertex
6. Mark the current vertex as visited
7. Update distance and previous lists based on those vertices which can be immediately reached from the current vertex
8. Update the current vertex to the unvisited vertex that can be reached by the shortest path from the starting vertex
9. Repeat (from step 6) until the goal vertex is visited

https://www-m9.ma.tum.de/graph-algorithms/spp-dijkstra/index_en.html

DIJKSTRA'S ALGORITHM

An algorithm that solves the general Shortest Path Problem:



A STAR (A*) ALGORITHM

An algorithm that solves the general Shortest Path Problem – faster than Dijkstra's Algorithm in most cases:

- It uses a distance-plus-cost function $f(x) = g(x) + h(x)$ to determine the order of search
- The path-cost function $g(x)$ is the cost from the starting node to the current node
- $h(x)$ is an admissible "heuristic estimate" of the distance to the goal, it must not overestimate the distance to the goal. It is used to guide the search to the desired outcome

Dijkstra's Algorithm is the special case of A* where $h(x) = 0$ for all x

A* EXAMPLE

$g(n)$ = cost from start

2	1	2	3	4	5
1	S			5	6
2	1			6	7
3	2			7	8
4				8	9
5	6	7		G	
6	7	8			

$h(n)$ = cost to goal (estimate)

9	8	7	6	5	6
8	S			4	5
7	6			3	4
6	5			2	3
5				1	2
4	3	2		G	
5	4	3			

$h(n)$ is the distance to goal assuming no obstacle

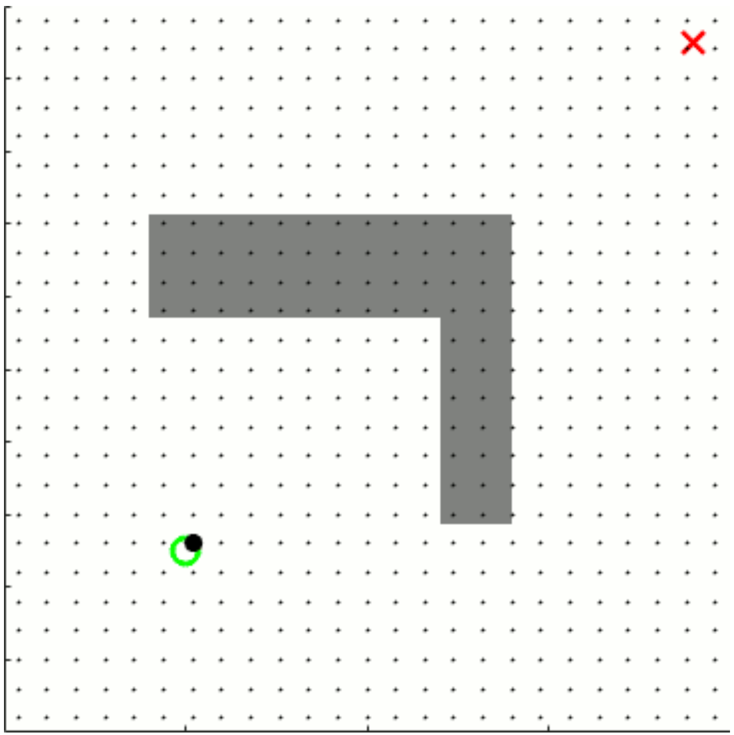
A* EXAMPLE

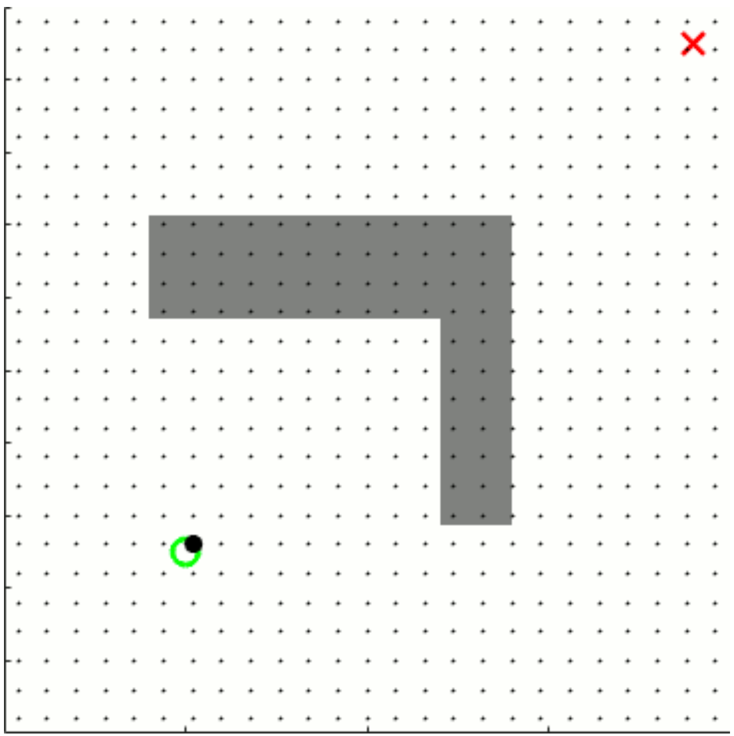
$$f(n) = h(n) + g(n)$$

9+2	8+1	7+2	6+3	5+4	6+5
8+1	S			4+5	5+6
7+2	6+1			3+6	4+7
6+3	5+2			2+7	3+8
5+4				1+8	2+9
4+5	3+6	2+7		G	
5+6	4+7	3+8			

9+2	1(9)	2(9)	3(9)	4(9)	6+5
8+1	S			5(9)	5+6
7+2	1(7)			6(9)	4+7
3(9)	2(7)			7(9)	3+8
4(9)				8(9)	2+9
5(9)	6(9)	7(9)		G	
5+6	4+7	3+8			

DIJKSTRA VS A*


Dijkstra

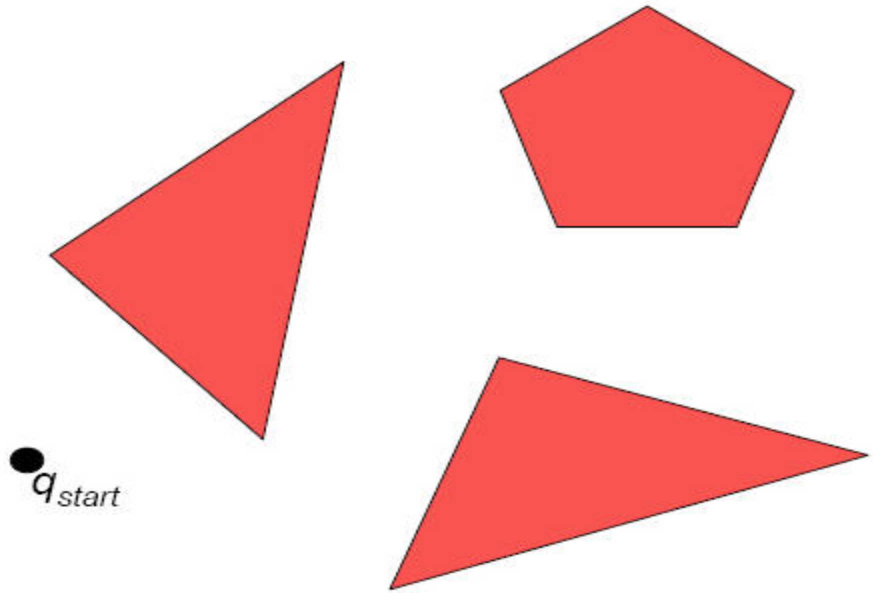

A*

A* PSEUDO CODE

```
Priority Queue Open List
Closed List
AStarSearch
    s.g = 0 // s is the start node
    s.h = GoalDistEstimate(s)
    s.f = s.g + s.h
    s.parent = null
    push s on Open
    while Open is not empty
        pop node n from Open // n has the lowest f
        if n is a goal node
            construct path
            return success
        for each successor n' of n
            newg = n.g + cost(n,n')
            if n' is in Open or Closed
               if n'.g <= newg skip
               remove n' from Open or Closed
            n'.parent = n
            n'.g = newg
            n'.h = GoalDistEstimate(n')
            n'.f = n'.g + n'.h
            push n' on Open
        push n onto Closed
    return failure // if no path found
```

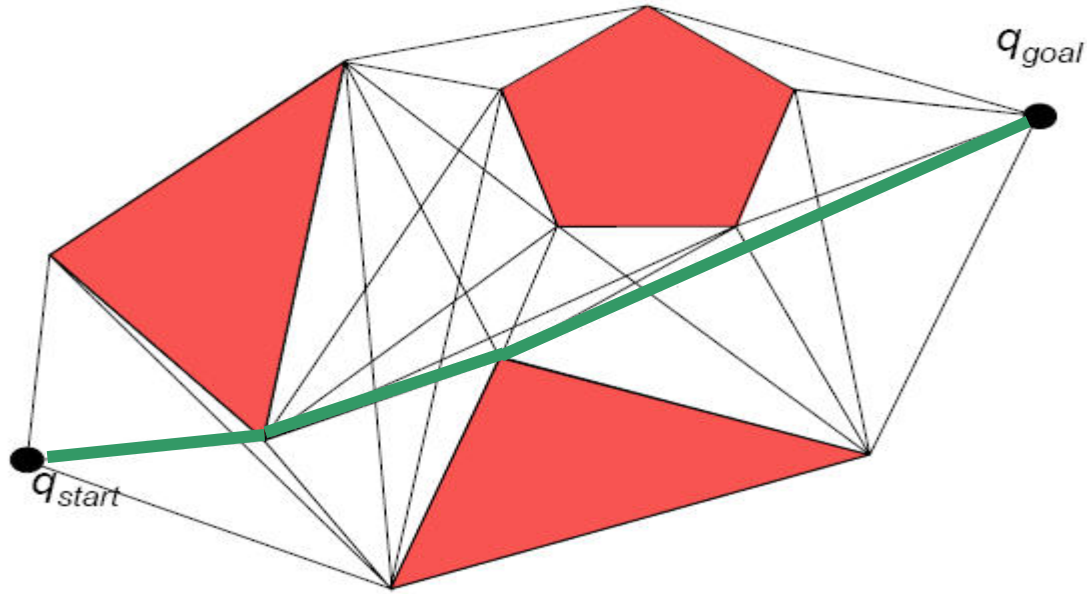
https://qiao.github.io/PathFinding.js/visual/

VISIBILITY GRAPH (NO SAMPLING)



- Suppose we have a Configuration space with polygonal obstacles
- If there were no blocks, shortest path would be a straight line. Else it must be a sequence of straight lines “shaving” corners of obstacles

VISIBILITY GRAPH (NO SAMPLING)



1. Find all non-blocked lines between polygon vertices, start and goal

2. Search the graph of these lines for the shortest paths

VISIBILITY GRAPH ALGORITHMS

- Visibility Graph method finds the shortest path
- But it does so by skirting along and close to obstacles
- Any errors in control, or model of obstacle locations, bad thing happens

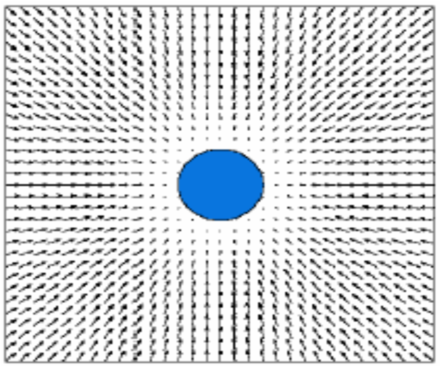
POTENTIAL FIELDS

- Tries to guide the robot from the initial configuration to the goal configuration using an artificial potential field
- The robot can follow a virtual force defined at each point by the gradient of the potential field
- The potential field is composed of one field attracting the agent to the goal configuration and one field repelling the agent from configuration space obstacles

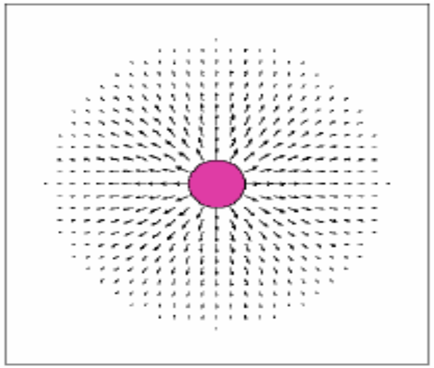
POTENTIAL FIELD METHOD

- Approach initially proposed for real-time collision avoidance [Khatib, 86]

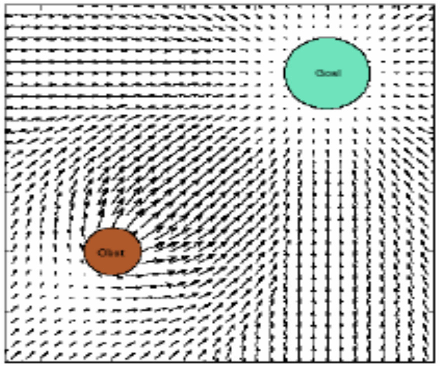
Khatib 1986
Latombe 1991
Koditschek 1998



Attractive Potential for goals



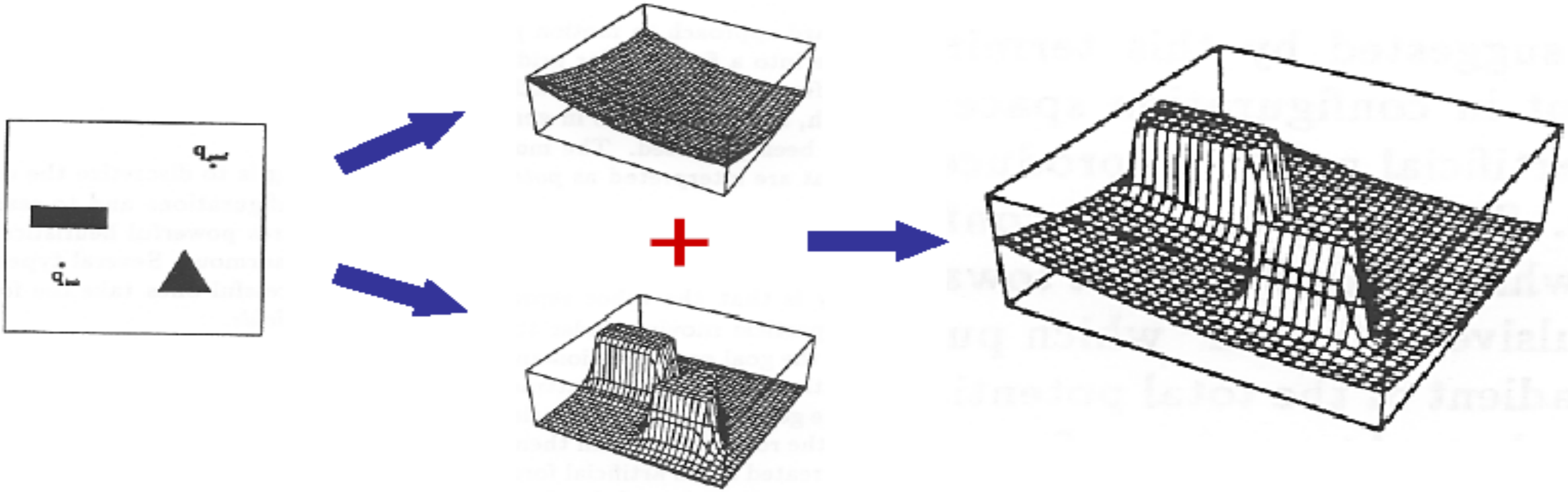
Repulsive Potential for obstacles



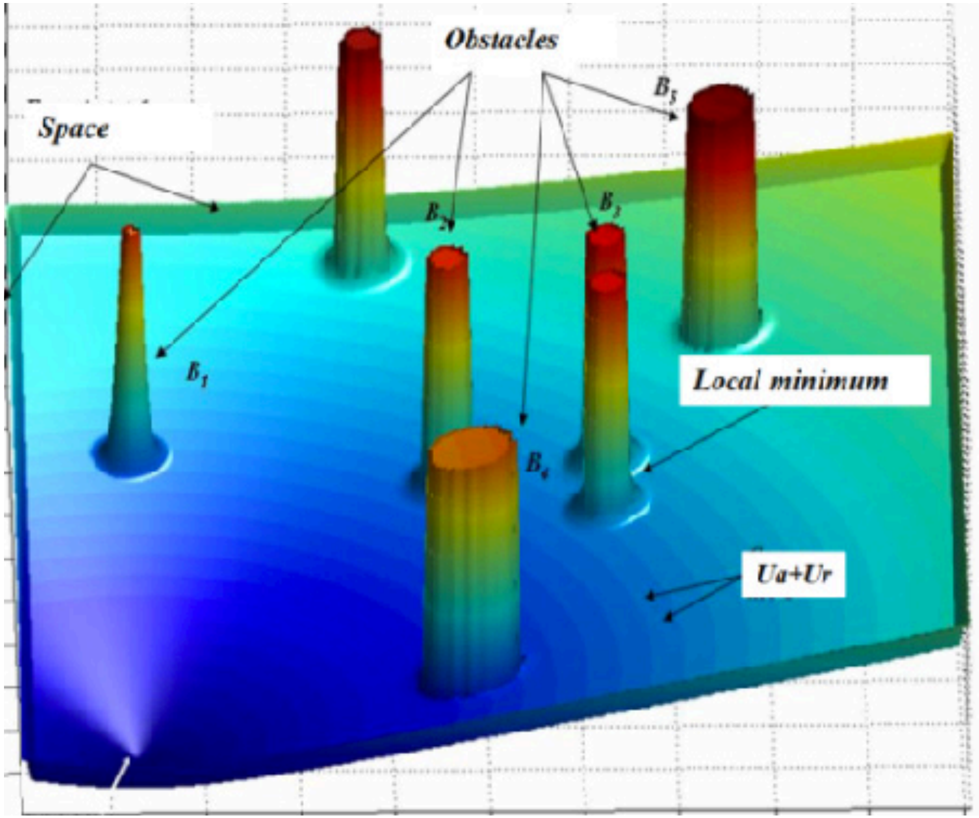
Combined Potential Field

Move along force: $F(x) = \nabla U_{att}(x) - \nabla U_{rep}(x)$

ATTRACTIVE AND REPULSIVE FIELDS/FORCES

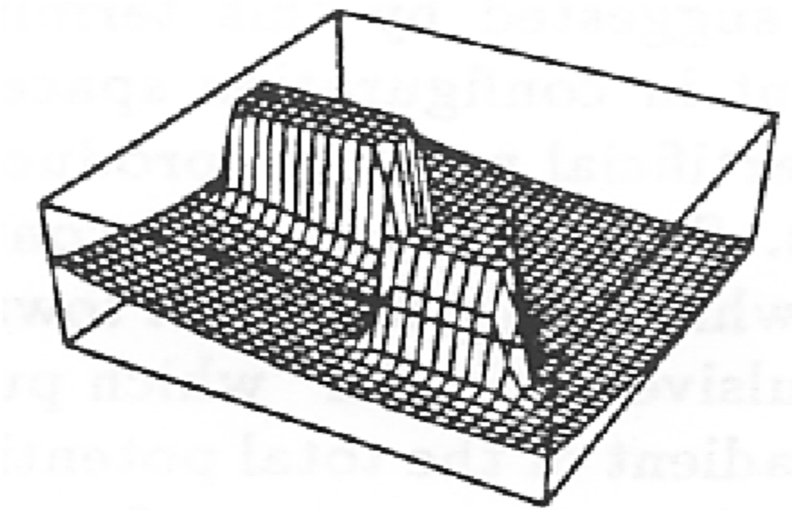


ATTRACTIVE AND REPULSIVE FIELDS/FORCES



LOCAL MINIMUM ISSUE

- Perform best-first search (possibility of combining with approximate cell decomposition)
- Alternate descents and random walks
- Use local-minimum-free potential (navigation function)



SAMPLING-BASED METHOS

- Probabilistic road maps **(PRM)**

[Kavraki et al., 92]

- Rapidly exploring random trees **(RRT)**

[Lavalle and Kuffner, 99]

PRMS (PROBABILISTIC ROAD MAPS)

Idea: Take random samples from **C**, declare them as vertices if in **C-free**, try to connect nearby vertices with local planner

- In a probabilistic roadmap, we first sample the C-space, by generating a large number of samples inside our space, and see if the sample collides with anything
- For every sample we generate, we try to hook it up to its nearest neighbours with a short, local path.
- After we've generated enough nodes we think we have a path, search the graph for the path (Dijkstra's or A*)

THE PRM ALGORITHM

Let: $V \leftarrow \emptyset$; $E \leftarrow \emptyset$;

loop

    $c \leftarrow$ a (useful) configuration in $\mathcal{C}_{\text{free}}$

    $V \leftarrow V \cup \{c\}$

    $N_c \leftarrow$ a set of (useful) nodes chosen from V

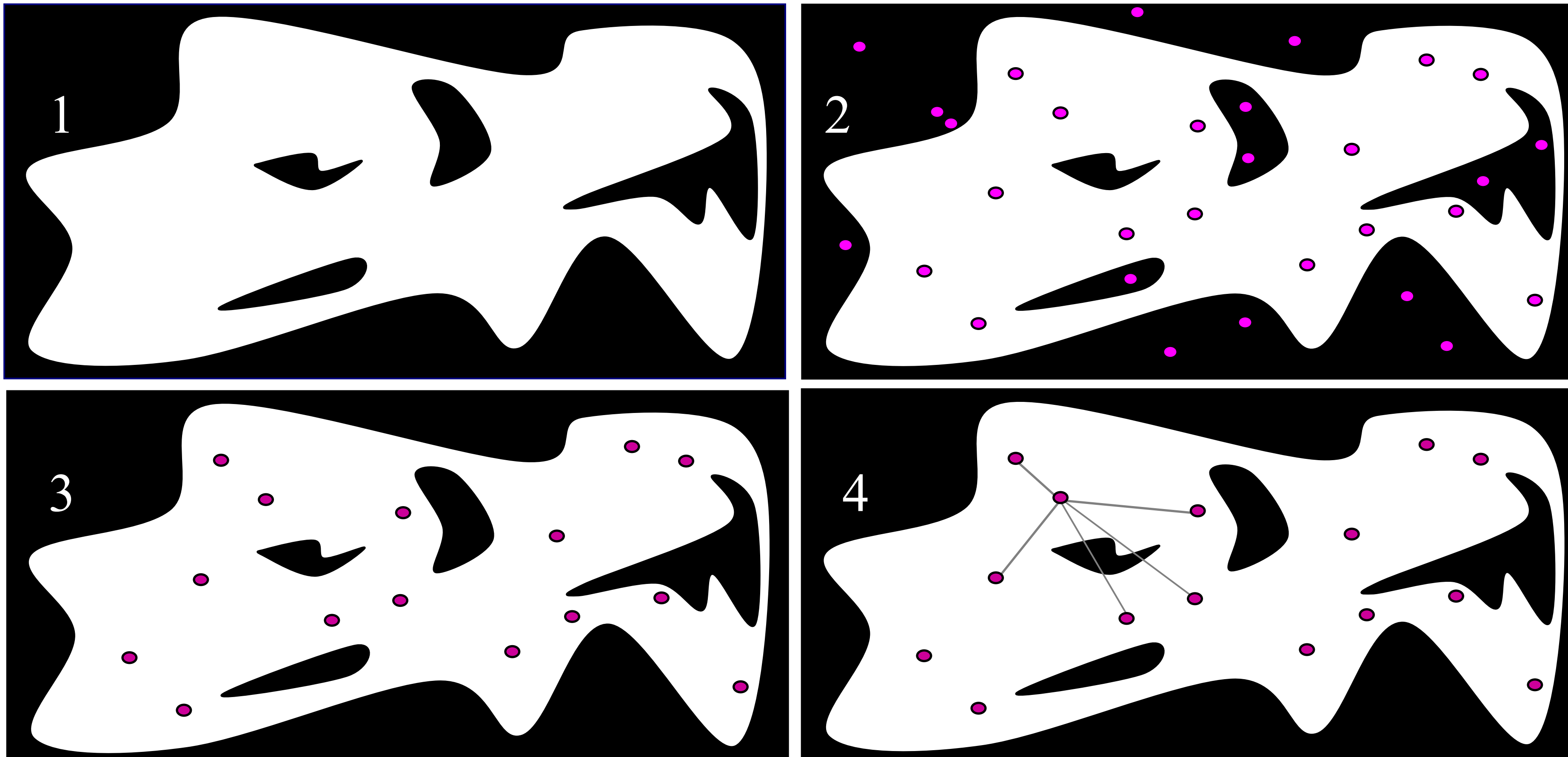
    for all $c' \in N_c$, in order of increasing distance from c **do**

        if c' and c are not connected in G **then**

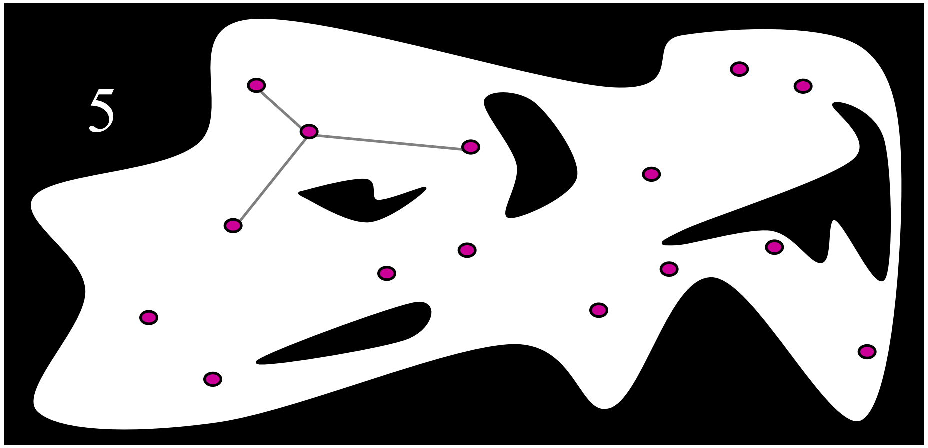
            if the local planner finds a path between c' and c **then**

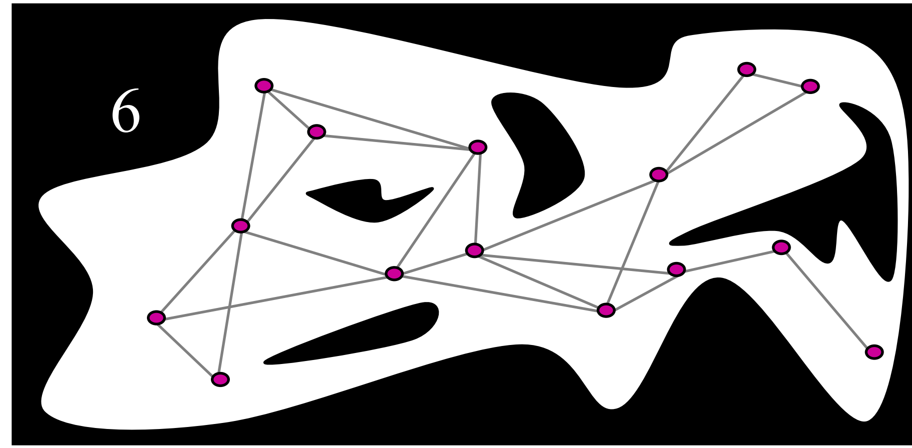
                add the edge $c'c$ to E

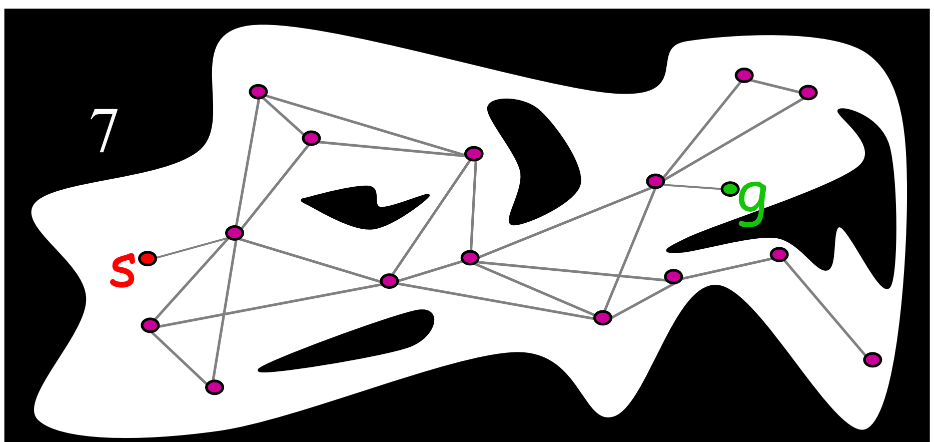
PRM EXAMPLE

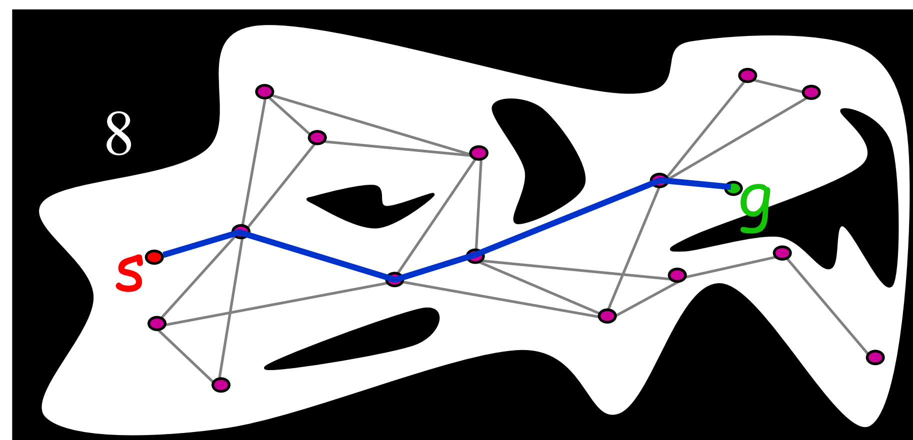


PRM EXAMPLE





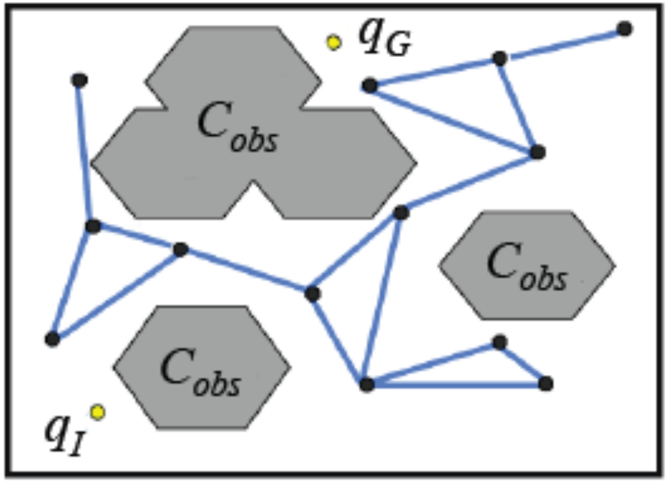


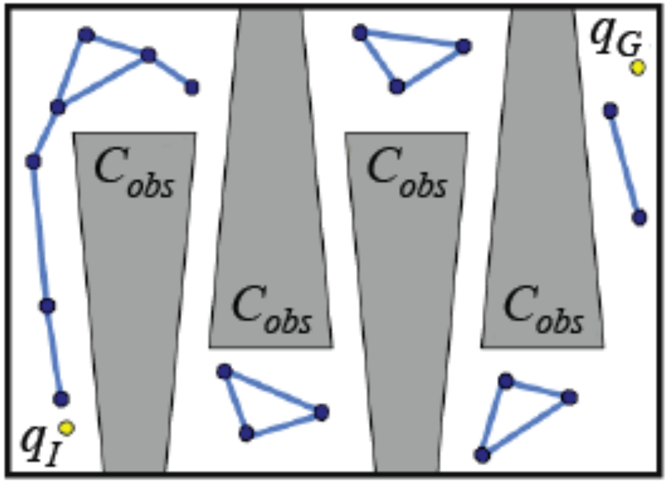


PRM

Pros:
- Probabilistically complete
- Do not construct C-space
- Apply easily to high-dim C's
- PRMs have solved previously unsolved problems

Cons:
- Do not work well for some problems, narrow passages
- Not optimal, not complete





RRTS (RAPIDLY-EXPLORING RANDOM TREES)

- Randomised algorithm that is designed for a broad class of path planning problems
- Conceptually Simple
- Quick exploration of the high-dimensional state space
- Easily incorporate differential constraints (dynamics can be considered directly)
- Applicable to a broad class of problems (holonomic, nonholonomic, kinodynamic)

RANDOMIZED PLANNERS / RRTS

- Selects **random points** from an environment and moves towards that point that is an incremental distance away from the nearest node of an expanding tree

- The movement from existing traversed points to random points in the environment will lead to a **path that branches** out somewhat like a tree, and will cover most of the free space in the environment

- A planned path will be found once a branch in the tree comes close the destination (goal)

- The algorithm: Given C and q_0

Algorithm 1: RRT

G.init(q_0)
repeat
    $q_{rand} \rightarrow$ RANDOM_CONFIG($\mathcal{C}$)
    $q_{near} \leftarrow$ NEAREST(G, q_{rand})
    G.add_edge(q_{near}, q_{rand})
until *condition*

Sample from a **bounded region** centered around q_0

E.g. an axis-aligned relative random translation or random rotation

(but recall sampling over rotation spaces problem)

- The algorithm

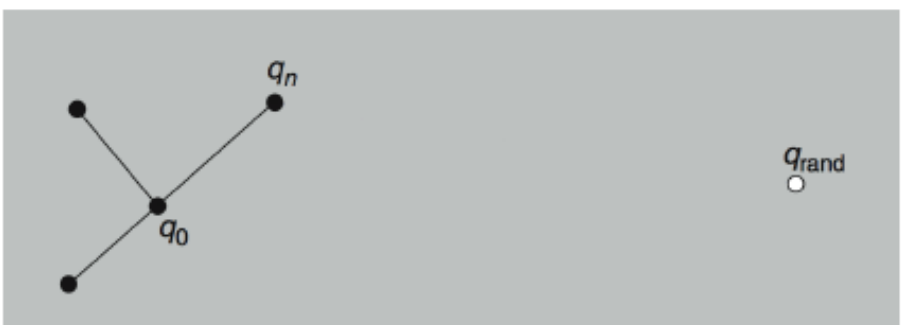
Algorithm 1: RRT

$G.\text{init}(q_0)$
repeat
$\quad q_{rand} \rightarrow \text{RANDOM\_CONFIG}(\mathcal{C})$
$\quad q_{near} \leftarrow \text{NEAREST}(G, q_{rand})$
$\quad G.\text{add\_edge}(q_{near}, q_{rand})$
until *condition*

Finds closest vertex in G using a **distance function**

$$\rho \;:\; \mathcal{C} \times \mathcal{C} \rightarrow [0, \infty)$$

formally a ***metric*** defined on C



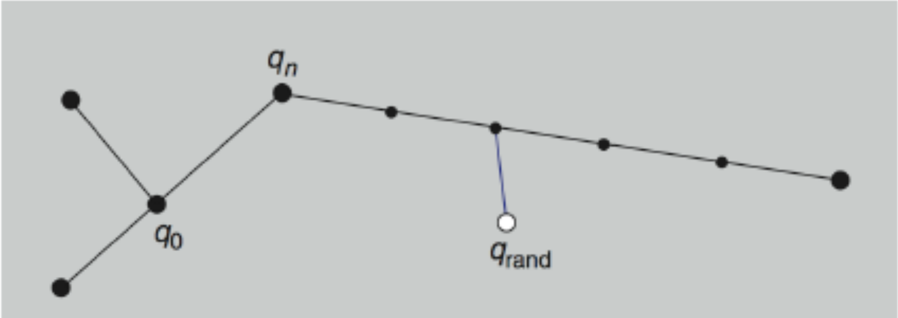
- The algorithm

Algorithm 1: RRT

$G.\text{init}(q_0)$
repeat
  $q_{rand} \rightarrow \text{RANDOM\_CONFIG}(\mathcal{C})$
  $q_{near} \leftarrow \text{NEAREST}(G, q_{rand})$
  $G.\text{add\_edge}(q_{near}, q_{rand})$
until *condition*

Several stategies to find q_{near} given the closest vertex on G:

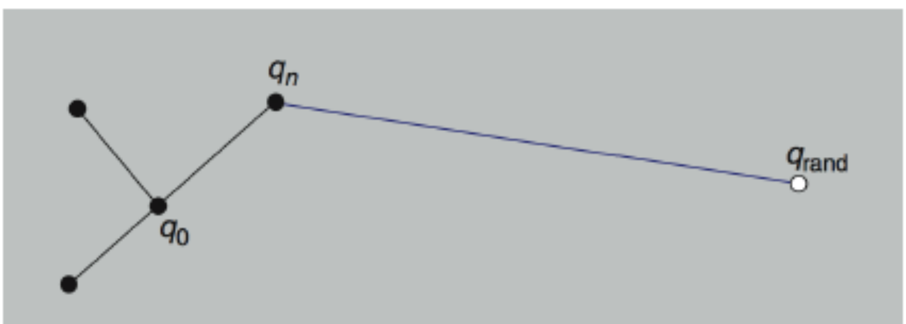
- Take closest vertex
- Check intermediate points at regular intervals and split edge at q_{near}



- The algorithm

Algorithm 1: RRT

G.init(q_0)
repeat
    $q_{rand} \rightarrow$ RANDOM_CONFIG($\mathcal{C}$)
    $q_{near} \leftarrow$ NEAREST(G, q_{rand})
    G.add_edge(q_{near}, q_{rand})
until *condition*



Connect nearest point with random point using a **local planner** that travels from q_{near} to q_{rand}

- No collision: add edge
- Collision: new vertex is q_i, as close as possible to C_{obs}

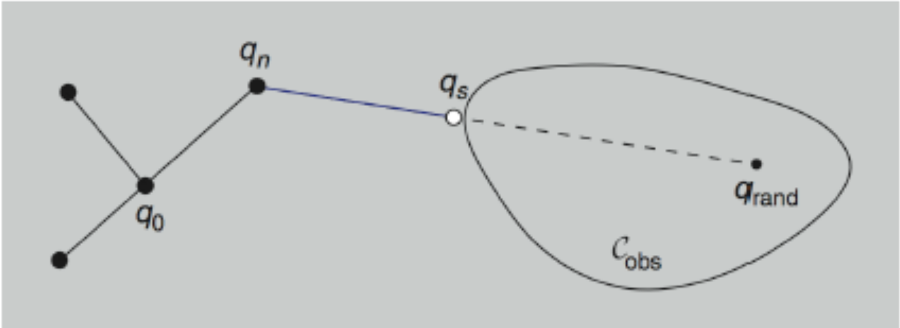
The algorithm

Algorithm 1: RRT

G.init(q_0)
repeat
  $q_{rand} \rightarrow$ RANDOM_CONFIG$(\mathcal{C})$
  $q_{near} \leftarrow$ NEAREST(G, q_{rand})
  G.add_edge(q_{near}, q_{rand})
until *condition*

Connect nearest point with random point using a **local planner** that travels from q_{near} to q_{rand}

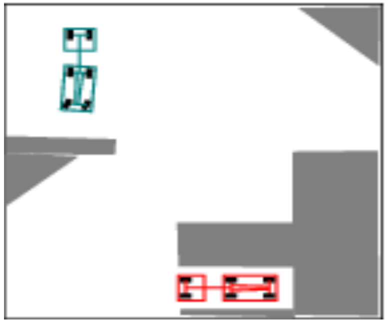
- No collision: add edge
- Collision: new vertex is q_i, as close as possible to C_{obs}

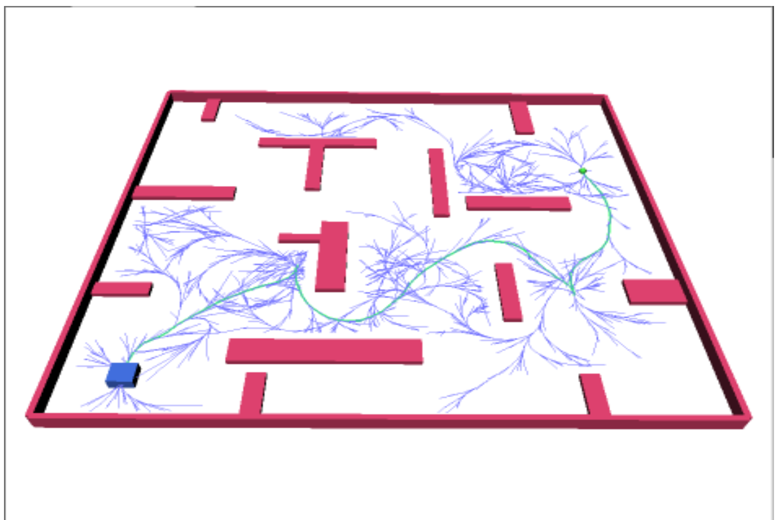


NICE RRT PROPERTIES

- Distribution of vertices approaches sampling distribution
- Probabilistically complete
- Simple implementation
- Always connected
- Does not require ability to steer between prescribed states
- Expansion biased towards unexplored space

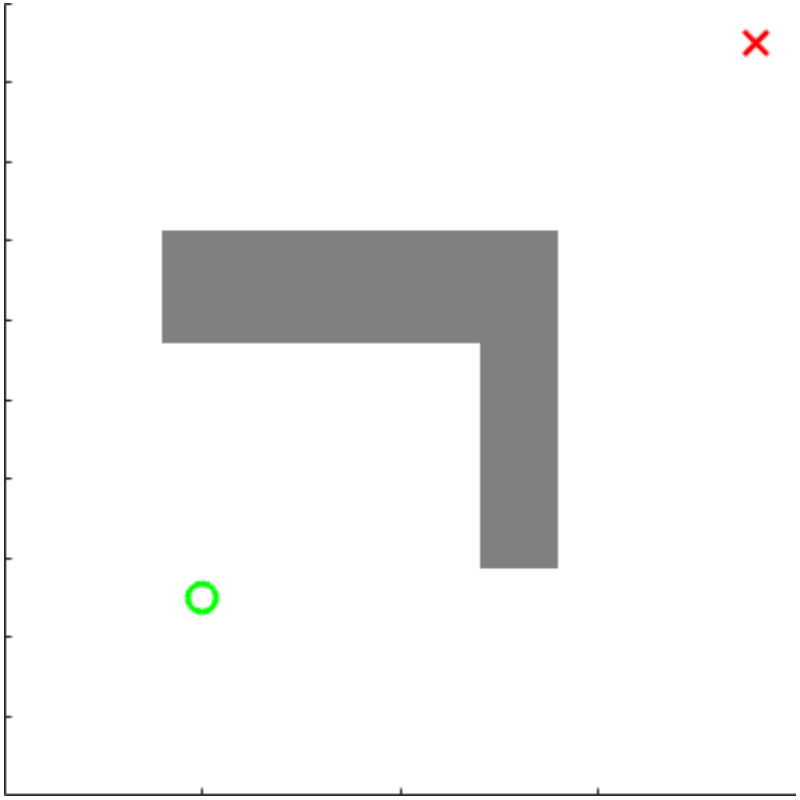
RRT EXAMPLES



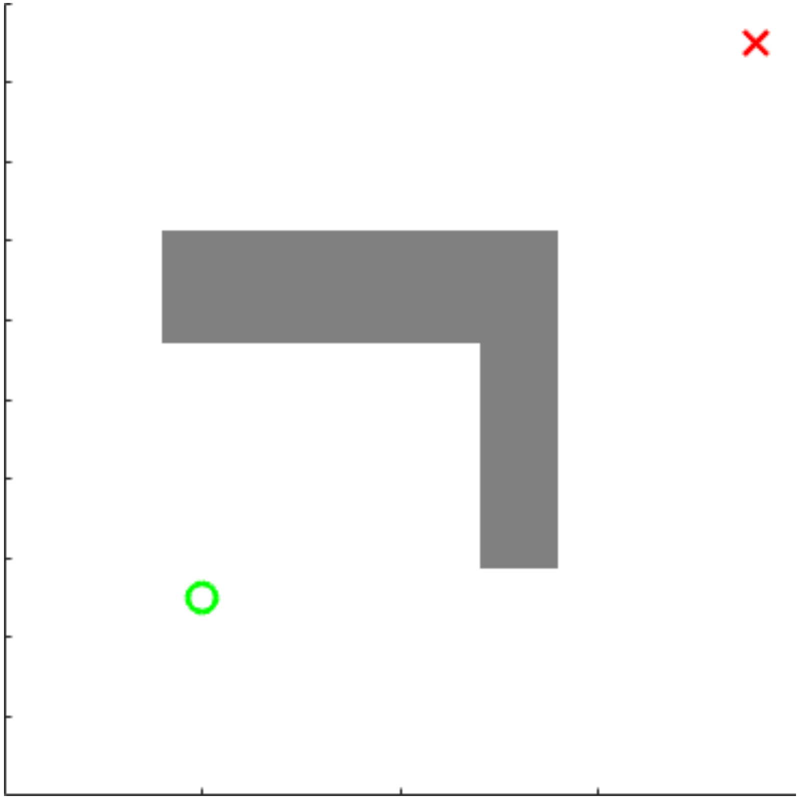


Source: http://msl.cs.uiuc.edu/rrt/gallery.html

RRT VS PRM



RRT



PRM $K = 11$