Motion planning

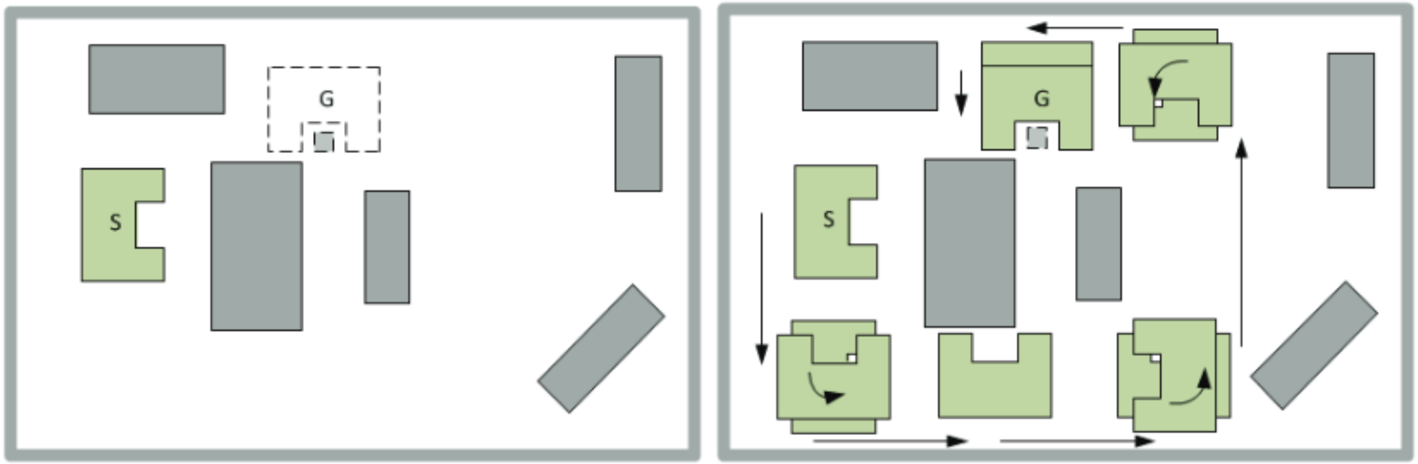
FIELD AND SERVICE ROBOTICS

UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II
DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

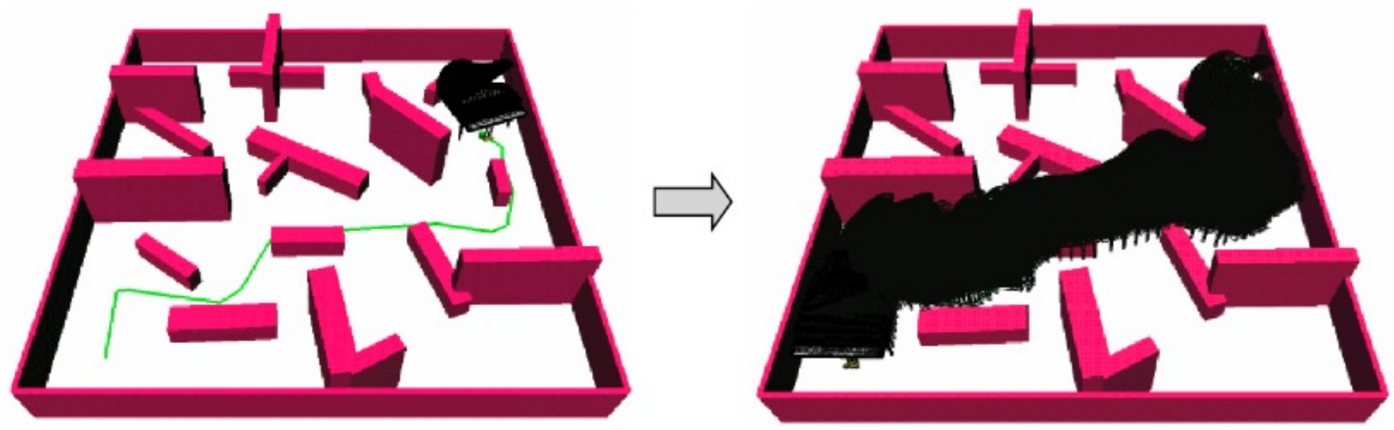
www.prisma.unina.it

- In presence of obstacles, it is necessary to plan motions that enable the robot to execute the assigned task without colliding with them
- One would like that the robot can move from an initial to the desired pose without colliding with obstacles, starting from a high-level description of the task and a geometric characterization of the workspace
  - Offline planning: made in advance, the environment is known
  - Online planning: made at runtime, the environment is discovered through sensors
- Static obstacles
  - Fixed with respect to the environment
    - Walls
    - Desks
    - Doors
- Dynamic obstacles
  - Objects that can appear at any time in the environment
    - Persons
    - Other robots
    - Sliding doors

- A fundamental need in robotics is to have algorithms that convert high-level specifications of tasks from humans into low-level descriptions of how to move
  - Motion planning
- A classical version of motion planning is sometimes referred to as the Piano Mover's Problem (2-D version)
  - Generalized Mover's Problem (3-D version)



- A fundamental need in robotics is to have algorithms that convert high-level specifications of tasks from humans into low-level descriptions of how to move
  - Motion planning
- A classical version of motion planning is sometimes referred to as the Piano Mover's Problem (2-D version)
  - Generalized Mover's Problem (3-D version)



- Consider a generic robot $\mathfrak{B}$ (wheeled robot, aerial robot, legged robot, industrial manipulator, …)
  - The robot moves in its workspace $W \equiv \mathbb{R}^n$, with $n = \{2,3\}$
- Denote with $\mathcal{O}_1, \mathcal{O}_2, \dots, \mathcal{O}_p$ the obstacles
  - We will suppose that, at least for the offline planning, they are fixed in W
- Suppose that, at least for the offline planning, the geometry of $\mathfrak{B}, \mathcal{O}_1, \mathcal{O}_2, \dots, \mathcal{O}_p$ are known
- Suppose also that $\mathfrak{B}$ can instantaneously move everywhere
  - We will relax this assumption

- Motion planning problem
  - Given the initial and the desired poses of $\mathfrak{B}$ in W, we want find, if it exists, a path (i.e., a continuous sequence of poses) that drives the robot between the two poses while avoiding any contact and collision with $\mathcal{O}_1, \mathcal{O}_2, \dots, \mathcal{O}_p$. A failure is reported if such path does not exist
- Hypotheses not present in the practice
  - There are moving robots in W
  - The obstacles are moving in W
  - The environment is unstructured/unknown
  - The robot has nonholonomic constraints
  - The manipulation problem is excluded from the above definition since it intrinsically requires contacts with objects

Distance definition in the configuration space

- Given two points in the configuration space, $q_1 \in \mathcal{C}$ and $q_2 \in \mathcal{C}$, the following Euclidian metrics can be inaccurate as in the example above for the 2-DoF Cartesian manipulator

$$d(q_1, q_2) = \|q_1 - q_2\|$$

  - This is appropriate when $\mathcal{C}$ is the Euclidian space only
- When $\mathcal{C}$ is not an Euclidian space, intuition suggests that the distance between q_1 and q_2 should go to zero when the portion of the space occupied by the robot in q_1 is coincident with the portion of the space occupied by the robot in q_2
- Let $\mathcal{B}(q)$ the subset of W occupied by the robot $\mathcal{B}$ when it is in $q \in \mathcal{C}$
- Let $p(q)$ the position in W of a robot's point, $p \in \mathcal{B}$
- The distance in $\mathcal{C}$ can be defined as

$$d_1(q_1, q_2) = \max_{p \in \mathcal{B}} \|p(q_1) - p(q_2)\|$$

  - In rough words, $d_1(q_1, q_2)$ is the maximum displacement in W that two model-configurations, q_1 and q_2, induce on a point $p \in \mathcal{B}$, as the point moves all around the robot

Obstacles definition in the configuration space

- We have to represent $\mathcal{O}_1, \mathcal{O}_2, \dots, \mathcal{O}_p$ in $\mathcal{C}$
  - It is assumed that the obstacles are closed (they contain their boundaries)
  - In general they are not necessarily limited subsets of W
- Given an obstacle $\mathcal{O}_i \in W,\ (i = 1, \dots, p)$, the image in the configuration space $\mathcal{C}$ is called $\mathcal{C}$-obstacle and it is defined as

$$CO_i = \{q \in \mathcal{C}: \mathcal{B}(q) \cap \mathcal{O}_i \neq \emptyset\}$$

  - That is, the space occupied by the robot in the workspace has some points in common with the space occupied by the i-th obstacle in the workspace
  - In rough words, the $\mathcal{C}$-obstacle, CO_i, is the subset of the model-configurations that cause a collision, or a contact, between the robot $\mathcal{B}$ and the obstacle $\mathcal{O}_i$
- The union of all the $\mathcal{C}$-obstacle spaces defines the $\mathcal{C}$-obstacle region

$$CO = \bigcup_{i=1}^{p} CO_i$$

The free configuration space

- The free configuration space is

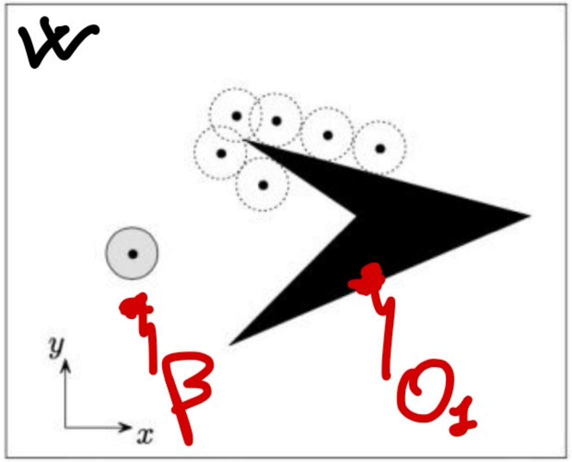
$$\mathcal{C}_{free} = \mathcal{C} - \mathcal{CO} = \{q \in \mathcal{C} : \mathcal{B}(q) \cap (\cup_{i=1}^{p} \mathcal{CO}_i) = \emptyset\}$$

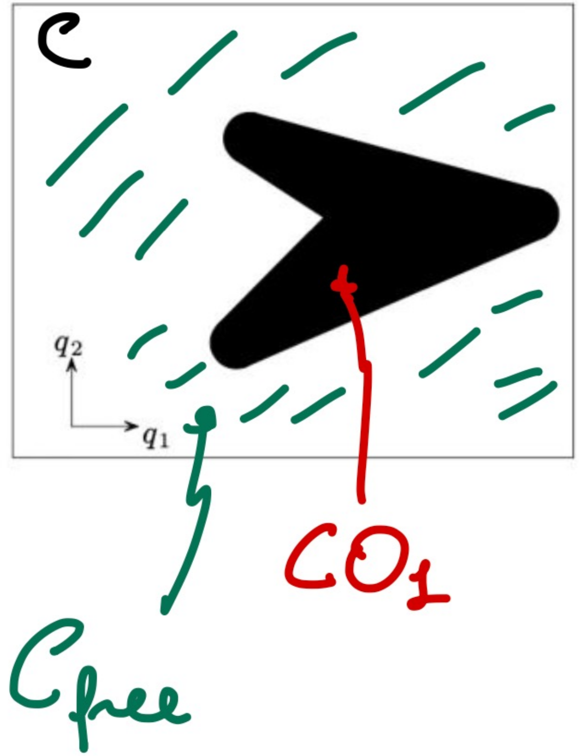
- A free path for the robot is a sequence of model-configurations all beloning to $\mathcal{C}_{free}$
- Notice that, even if $\mathcal{C}$ is a connected space (given two arbitrary model-configurations there exists a path joining them), the subset $\mathcal{C}_{free}$ may not be connected due to the presence of the $\mathcal{C}$-obstacle region

- We can now state a more formal definition for the motion planning problem
  - Let $q_s \in C_{free}$ be the starting model-configuration of the robot $\mathcal{B}$ in its workspace W. Let $q_g \in C_{free}$ be the goal model-configuration. Planning a collision-free motion for $\mathcal{B}$ means generating a path between q_s and q_g as connected components of C_{free}. A failure is reported otherwise.

C-obstacle region of a disk-shaped robot

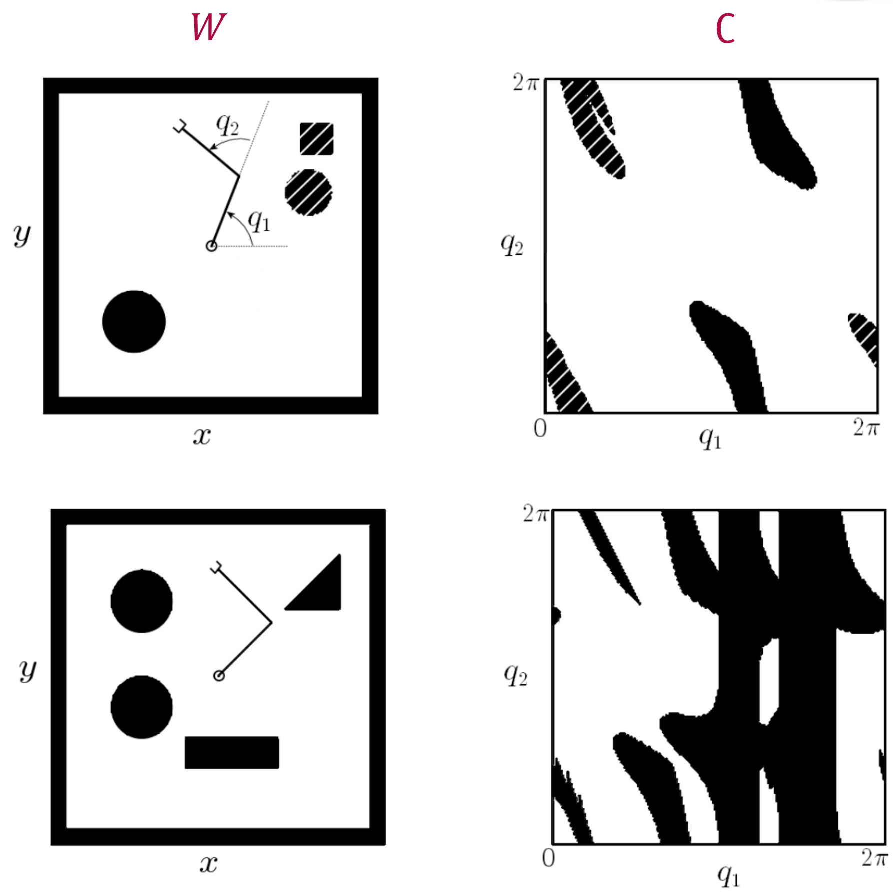
- Consider a disk-shaped robot
  - $W \equiv \mathbb{R}^2$
  - $q = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$
  - $\mathcal{C} \equiv \mathbb{R}^2$
- The boundaries of CO_1 are the locus of the model-configurations in which the robot touches the obstacle in W
- CO_1 can be built through a growing procedure
  - In this case, since the robot is disk-shaped, the procedure is isotropic





C-obstacle region of a robot manipulator

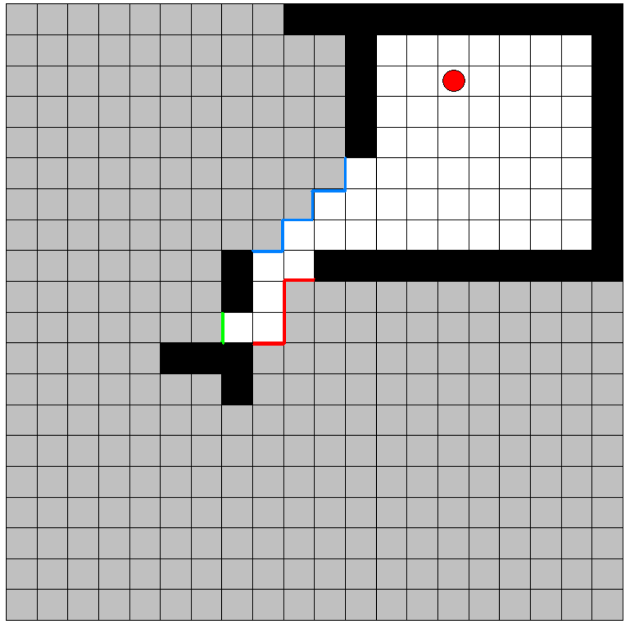
- Consider a robot manipulator $\mathcal{B}$ made by n rigid links $\mathcal{B}_1, \dots, \mathcal{B}_n$ connected by joints
- Two kinds of C-obstacle regions exist
  - Collision between a link $\mathcal{B}_i$ and an obstacle $\mathcal{O}_i$
  - Collision between a link $\mathcal{B}_i$ and another link $\mathcal{B}_j, i \neq j$ (self-collision)
- To obtain the boundaries of $C\mathcal{O}_i$ it is necessary to identify,, through appropriate inverse kinematic computations, all the model-configurations that bring one or more links of $\mathcal{B}$ in contact with $\mathcal{O}_i$
- Notice that, in the second part of this example, there exist three distinct connected regions



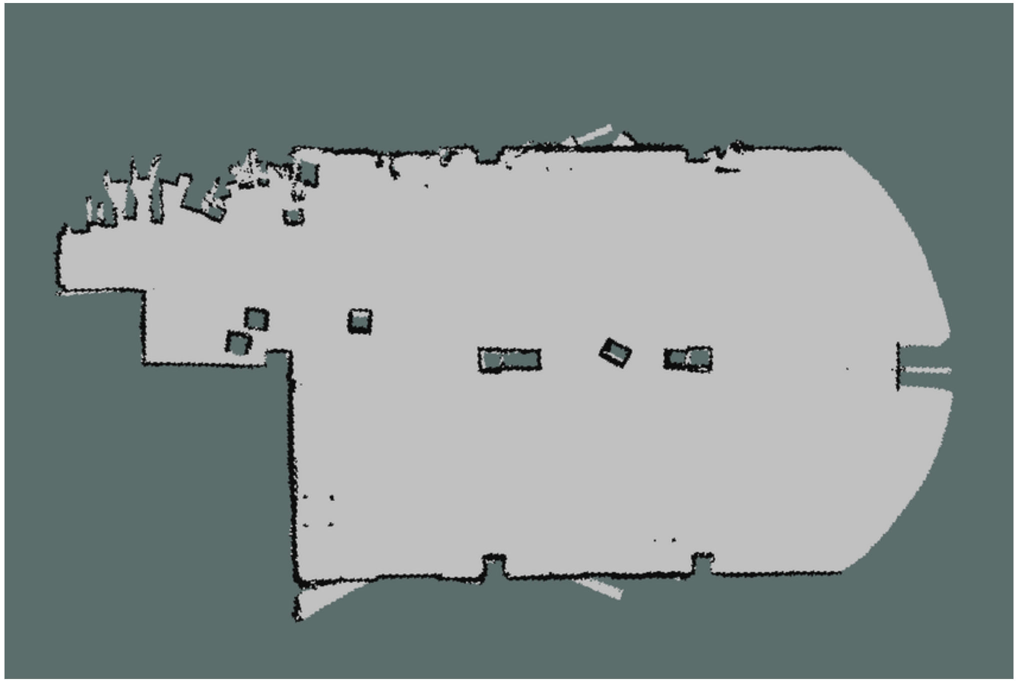
- Therefore, first, we need an algebraic or CAD model of the obstacle $\mathcal{O}_i$ in W
- Then, we need to compute the CO_i image exactly
  - The procedure may be complex
- A simple, but computationally intensive, way to build the CO_i image is to sample $\mathcal{C}$ by a regular grid
  - We then compute the volume occupied by $\mathcal{B}$ via direct kinematics and identify those samples bringing the robot in contact with $\mathcal{O}_i$ through a collision checking algorithm
  - The accuracy improves by increasing the grid resolution

- The idea is to build an approximation of C_{free}
  - Then, we seek the path connecting q_s from q_g, if it exists
- At each iteration of the planner, a sample model-configuration is chosen and it is checked whether there is a collision/contact or not with some obstacles
  - If there is a collision/contact, the sample is discarded from C_{free}
  - If there is not a collision/contact, the sample is saved in the current roadmap
    - The roadmap is a structure representing the approximation of C_{free}

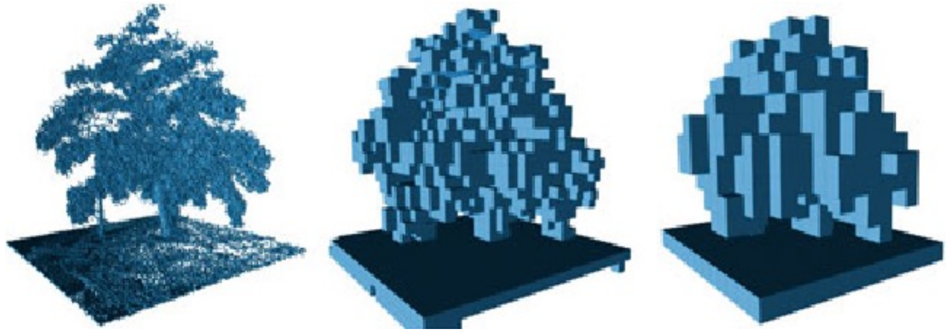
- Discretize the configuration space as a set of occupied/free cells of a matrix
- A common approach:
  - 1 Free cells
  - -1 Unknown cells
  - 0 Occupied cells
- Increasing the resolution of the cells help to speed up the planning process
- Increasing the resolution of the cells generate worst paths

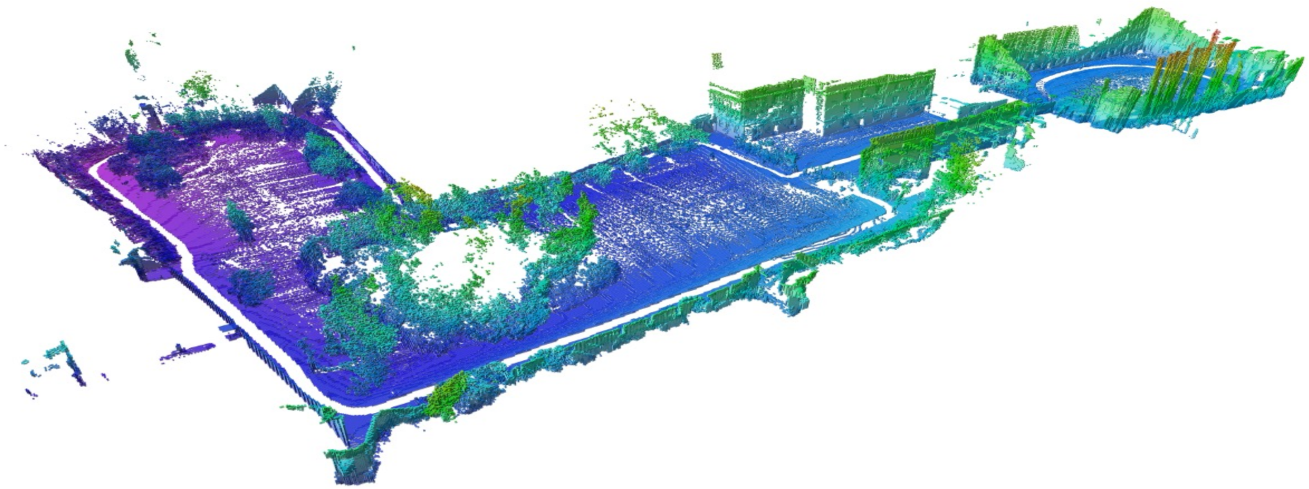


- Map representation is still a problem
  - Which kind of data structure must be used?
  - The data structure must be
    - Easy to read (time complexity)
    - Easy to visualize
    - Easy to maintain (space complexity)



- Wheeled ground-based robots
  - Occupancy grid map
- Aerial robots, legged robots, etc.
  - Discretize the configuration space as a set of occupied/free cells of a matrix



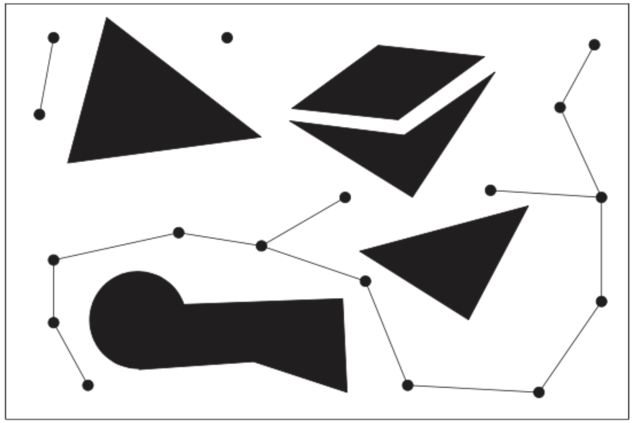


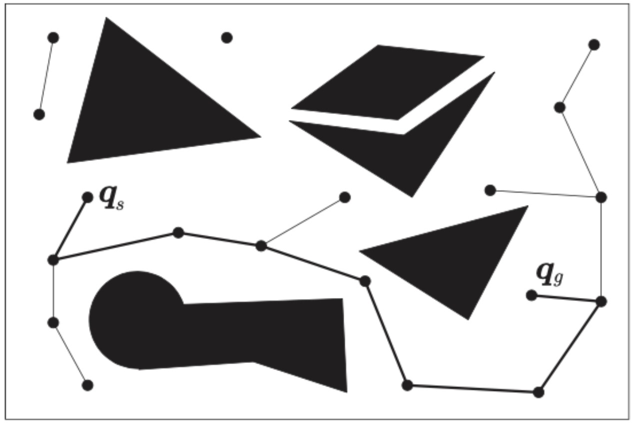
Probabilistic roadmap method (PRM)

- The basic idea is to generate randomly a sample $q_{rand} \in C$, with uniform probability distribution
  - This sample q_{rand} is tested for collisions
    - If the description of CO is available, it is easy since one must check if q_{rand} belongs to CO or not
    - If the description of CO is not available, kinematics and geometric relationships must be used to check if the robot is in contact or collides in W
  - If q_{rand} des not cause collisions, it is added to the roadmap, otherwise it is discarded
- At the end, many points will belong to C_{free}
- It is now important to create connections between these points approximating C_{free}
  - These connections should be collision-free as well
- The procedure to generate a free local path between q_{rand} and its closest model-configuration q_{near} is a job of the local planner
  - A common choice is a rectilinear path between q_{rand} and q_{near} in C
  - This rectilinear path is sampled, and each sample is checked for collisions
  - A "near" model-configuration, q_{near}, must be determined on the chosen metrics on C

Probabilistic roadmap method (PRM)

- The PRM incremental generation procedure stops when either a maximum pre-determined number of iterations is reached, or the number of connected components (connected regions of C_{free}) are small than a given threshold (the roadmap well describes C_{free})
- At this point, it is necessary to solve the motion planning problem by connecting q_s to q_g
  - First, q_s and q_g are connected to the roadmap through the local planner, if they do not belong yet to it
    - It means that we find a rectilinear path, that is collision-free, connecting q_s and q_g to the respective closest model-configurations of the roadmap
  - Then, the path connecting q_s and q_g is found on the roadmap (graph search algorithms)

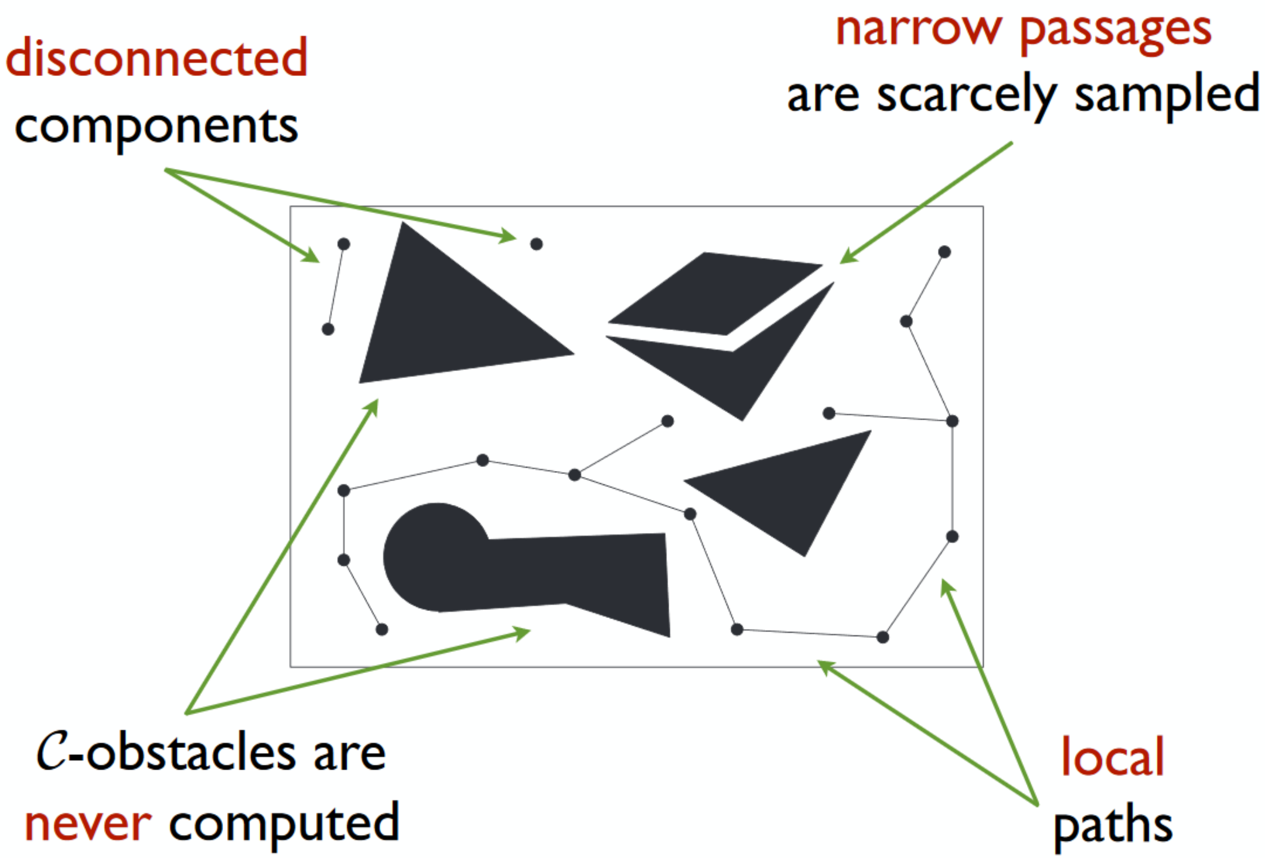




Probabilistic roadmap method (PRM)

- If a solution cannot be found, the PRM can be improved by performing more iterations so that the roadmap is more dense
  - Being the PRM a probabilistic methodology, the probability to find a suitable path connecting q_s and q_g tends to 1 as the execution time increases
  - This means that, if a solution does not exist, the algorithm continues indefinitely
- The PRM is critical for the narrow passages in C_{free}
  - This can be avoided by avoiding a uniform probability distribution
- The PRM describes the whole C_{free}, maybe including portions not of interest for the connection between q_s and q_g
- The PRM does not require the explicit computation of CO since we check only for collisions given a random sample or a local path

Probabilistic roadmap method (PRM)



Notation

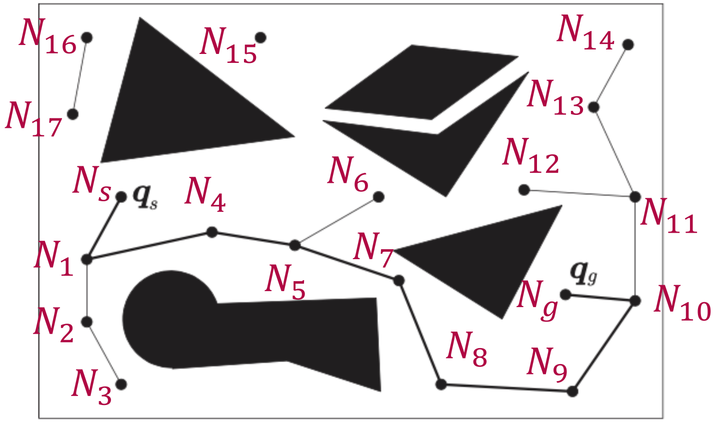
- The graph search algorithms are needed to find the best path connecting two points on the roadmap
- Let $G = (N, A)$ be a graph consisting of N nodes and A arcs
  - $n = card(N)$
  - $a = card(A)$
- G is usually represented by an adjacent list
  - To each node, N_i, is associated a list of nodes connected to N_i itself by arcs
- Consider the problem of searching G to find a path from the starting node, N_s, to the final one, N_g
  - We will see three techniques

- The BFS uses a queue, that is a FIFO (First-Input First-Output) data structure of nodes
  - We refer to this queue as OPEN
- At the beginning, OPEN contains the node N_s only and it is marked as visited
- Then, the other nodes are marked as unvisited
- At each iteration, the first node in OPEN is extracted and all the connect nodes marked as unvisited are inserted into OPEN as visited
- The search terminates once N_g is in OPEN as visited, or OPEN is empty (failure)
- During this search, the algorithm must keep track of the BFS tree, containing all the nodes that have led to unvisited nodes
  - If it exists, the BFS tree contains the path connecting N_s to N_g

Breadth-first search (BFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Unvisited
N_2	Unvisited
N_3	Unvisited
N_4	Unvisited
N_5	Unvisited
N_6	Unvisited
N_7	Unvisited
N_8	Unvisited
N_9	Unvisited
N_{10}	Unvisited
N_{11}	Unvisited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Unvisited



BFS TREE

OPEN

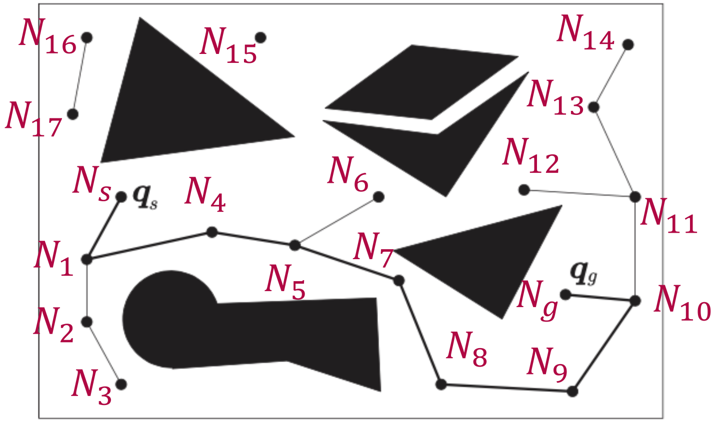
Node (visited from)
N_s

The *visited from* part is taken from the adjacency list

Breadth-first search (BFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Unvisited
N_3	Unvisited
N_4	Unvisited
N_5	Unvisited
N_6	Unvisited
N_7	Unvisited
N_8	Unvisited
N_9	Unvisited
N_{10}	Unvisited
N_{11}	Unvisited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Unvisited



BFS TREE

N_s

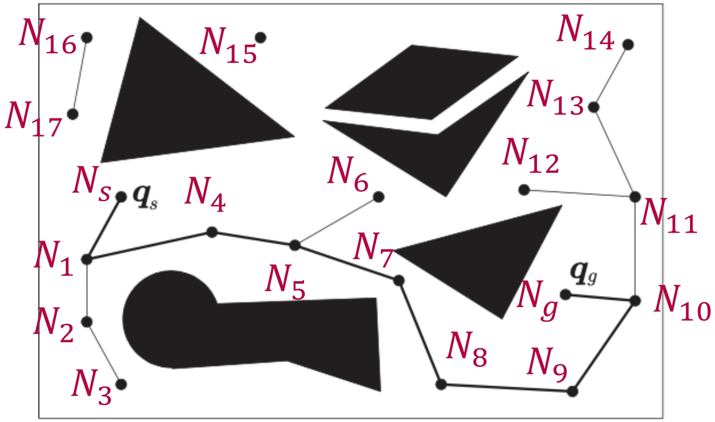
OPEN

Nodes (visited from)
$N_1(N_s)$

Breadth-first search (BFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Visited
N_3	Unvisited
N_4	Visited
N_5	Unvisited
N_6	Unvisited
N_7	Unvisited
N_8	Unvisited
N_9	Unvisited
N_{10}	Unvisited
N_{11}	Unvisited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Unvisited



BFS TREE

N_s

N_1

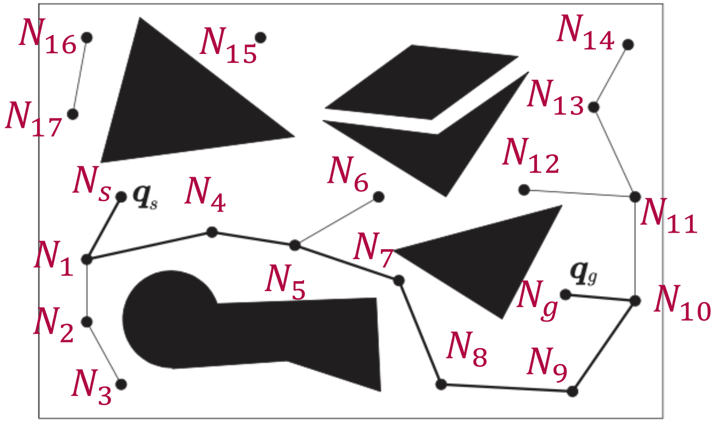
OPEN

Nodes (visited from)
N_2 (N_1)
N_4 (N_1)

Breadth-first search (BFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Visited
N_3	Visited
N_4	Visited
N_5	Unvisited
N_6	Unvisited
N_7	Unvisited
N_8	Unvisited
N_9	Unvisited
N_{10}	Unvisited
N_{11}	Unvisited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Unvisited



BFS TREE

N_s

|

N_1

|

N_2

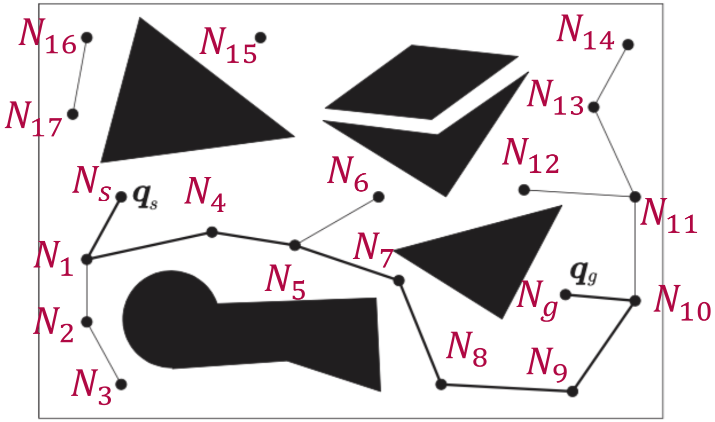
OPEN

Nodes (visited from)
N_4 (N_1)
N_3 (N_2)

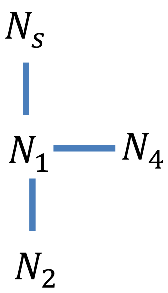
Breadth-first search (BFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Visited
N_3	Visited
N_4	Visited
N_5	Visited
N_6	Unvisited
N_7	Unvisited
N_8	Unvisited
N_9	Unvisited
N_{10}	Unvisited
N_{11}	Unvisited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Unvisited



BFS TREE



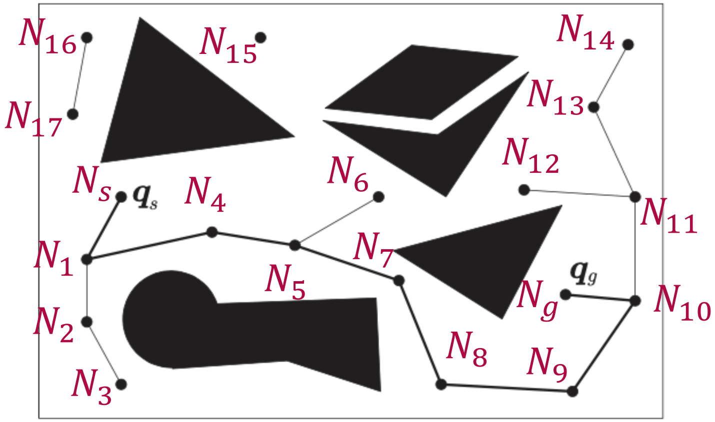
OPEN

Nodes (visited from)
N_3 (N_2)
N_5 (N_4)

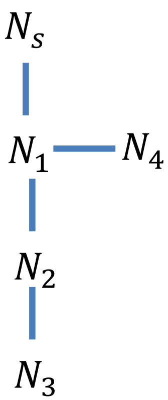
Breadth-first search (BFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Visited
N_3	Visited
N_4	Visited
N_5	Visited
N_6	Unvisited
N_7	Unvisited
N_8	Unvisited
N_9	Unvisited
N_{10}	Unvisited
N_{11}	Unvisited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Unvisited



BFS TREE



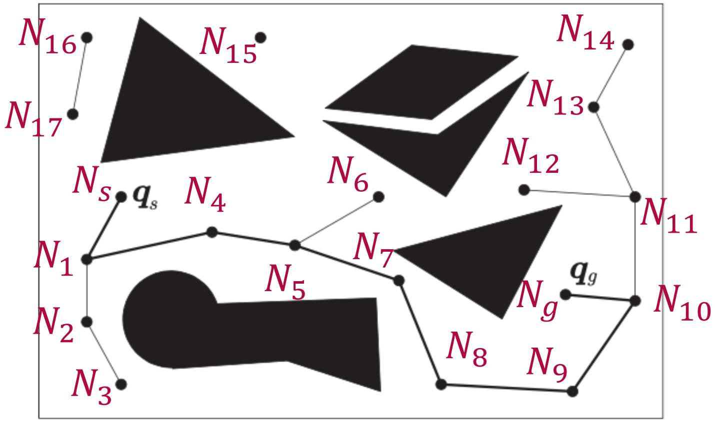
OPEN

Nodes (visited from)
N_5 (N_4)

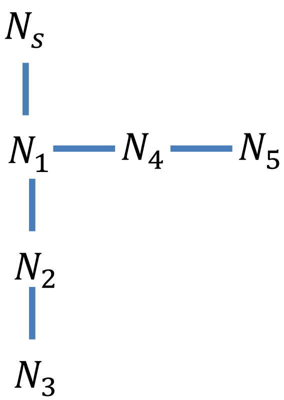
Breadth-first search (BFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Visited
N_3	Visited
N_4	Visited
N_5	Visited
N_6	Visited
N_7	Visited
N_8	Unvisited
N_9	Unvisited
N_{10}	Unvisited
N_{11}	Unvisited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Unvisited



BFS TREE



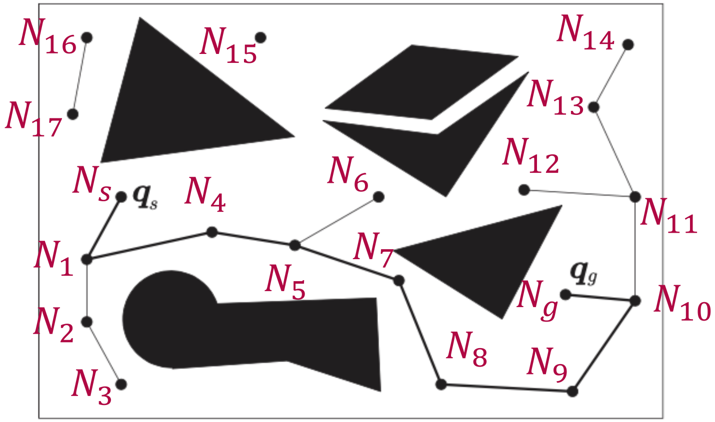
OPEN

Nodes (visited from)
N_6 (N_5)
N_7 (N_5)

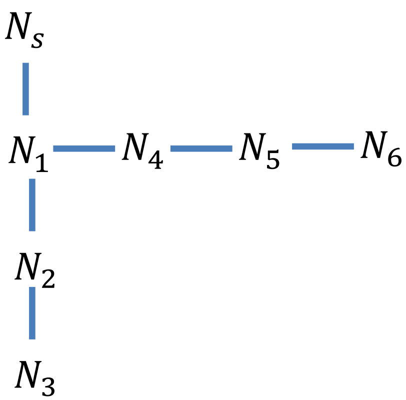
Breadth-first search (BFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Visited
N_3	Visited
N_4	Visited
N_5	Visited
N_6	Visited
N_7	Visited
N_8	Unvisited
N_9	Unvisited
N_{10}	Unvisited
N_{11}	Unvisited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Unvisited



BFS TREE



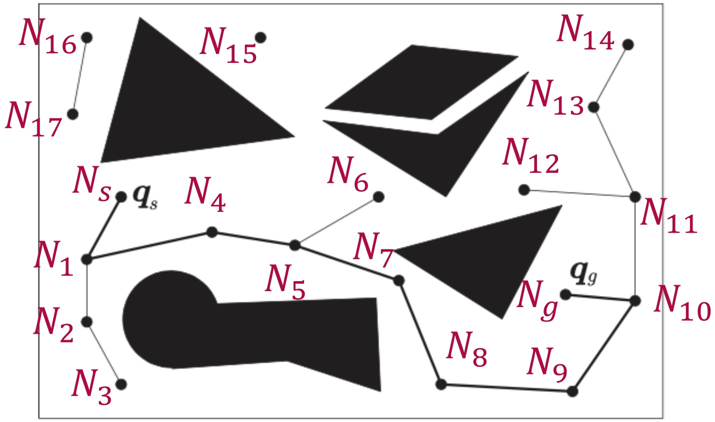
OPEN

Nodes (visited from)
N_7 (N_5)

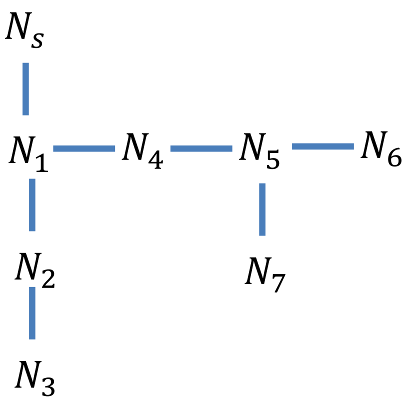
Breadth-first search (BFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Visited
N_3	Visited
N_4	Visited
N_5	Visited
N_6	Visited
N_7	Visited
N_8	Visited
N_9	Unvisited
N_{10}	Unvisited
N_{11}	Unvisited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Unvisited



BFS TREE



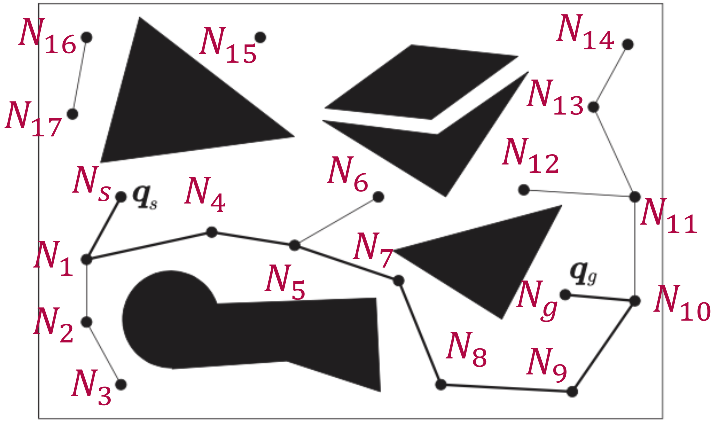
OPEN

Nodes (visited from)
N_8 (N_7)

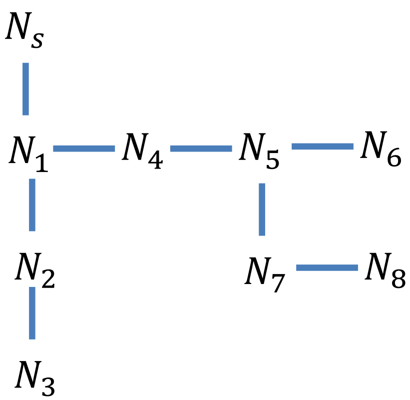
Breadth-first search (BFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Visited
N_3	Visited
N_4	Visited
N_5	Visited
N_6	Visited
N_7	Visited
N_8	Visited
N_9	Visited
N_{10}	Unvisited
N_{11}	Unvisited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Unvisited



BFS TREE



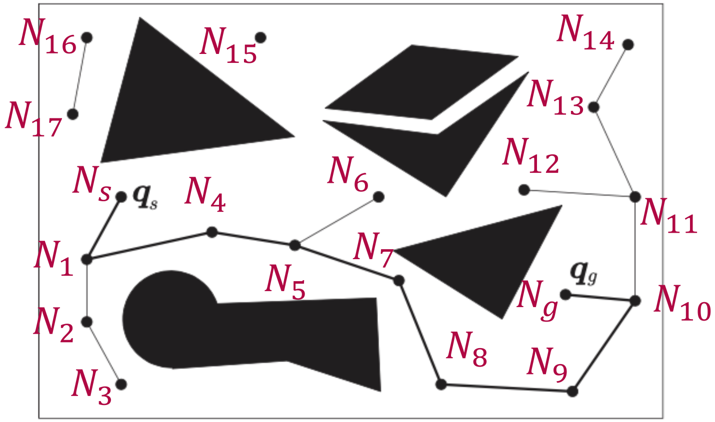
OPEN

Nodes (visited from)
N_9 (N_8)

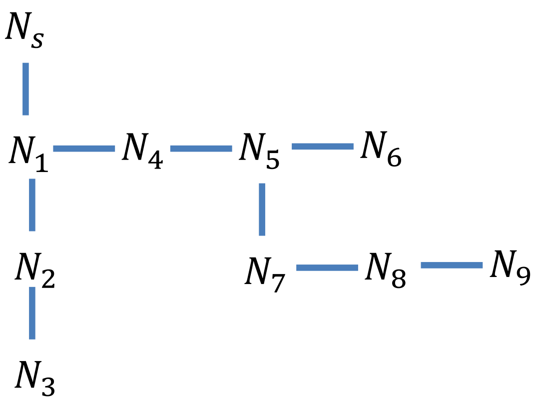
Breadth-first search (BFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Visited
N_3	Visited
N_4	Visited
N_5	Visited
N_6	Visited
N_7	Visited
N_8	Visited
N_9	Visited
N_{10}	Visited
N_{11}	Unvisited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Unvisited



BFS TREE



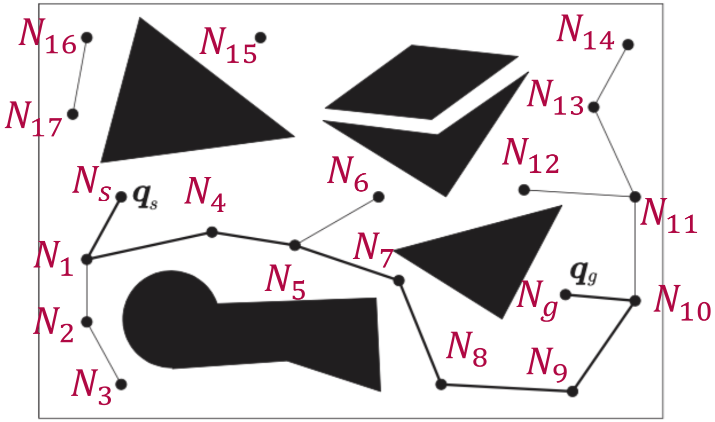
OPEN

Nodes (visited from)
N_{10} (N_9)

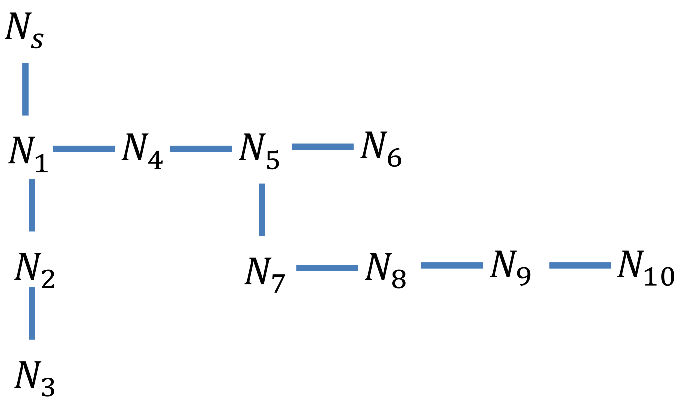
Breadth-first search (BFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Visited
N_3	Visited
N_4	Visited
N_5	Visited
N_6	Visited
N_7	Visited
N_8	Visited
N_9	Visited
N_{10}	Visited
N_{11}	Visited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Visited



BFS TREE



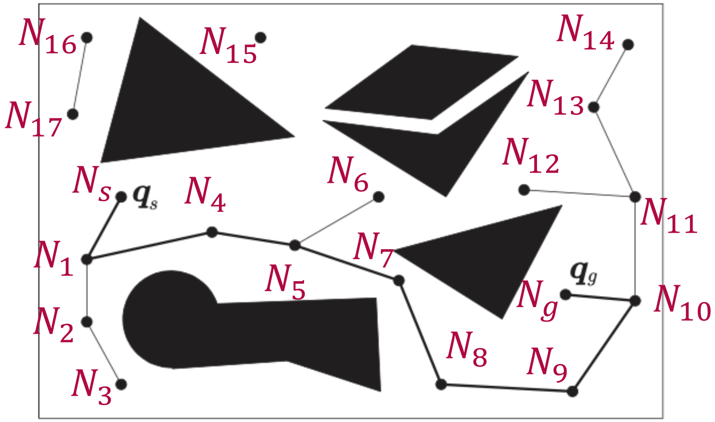
OPEN

Nodes (visited from)
N_g (N_{10})
N_{11} (N_{10})

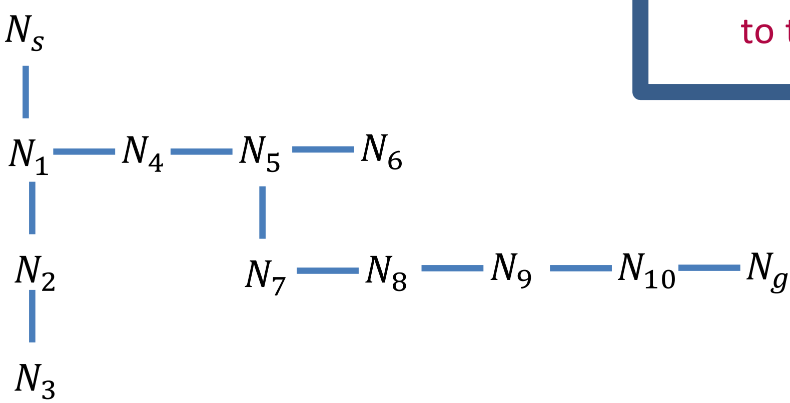
Breadth-first search (BFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Visited
N_3	Visited
N_4	Visited
N_5	Visited
N_6	Visited
N_7	Visited
N_8	Visited
N_9	Visited
N_{10}	Visited
N_{11}	Visited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Visited



BFS TREE



OPEN

Nodes (visited from)
N_g (N_{10})
N_{11} (N_{10})

BFS ends since N_g is in OPEN and it is added to the tree

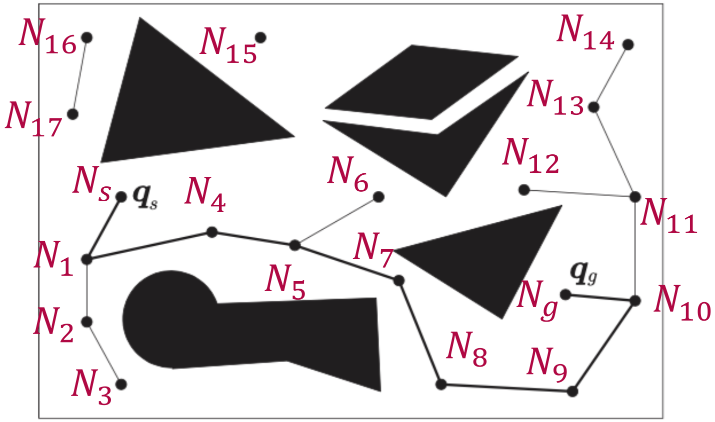
Depth-first search (DFS)

- The DFS uses a stack, that is a LIFO (Last-Input First-Output) data structure of nodes
  - We refer to this stack as OPEN
- At the beginning, OPEN contains the node N_s only and it is marked as visited
- Then, the other nodes are marked as unvisited
- At each iteration, the last node in OPEN is extracted and all the connect nodes marked as unvisited are inserted into OPEN as visited
- The search terminates once N_g is in OPEN as visited, or OPEN is empty (failure)
- During this search, the algorithm must keep track of the DFS tree, containing all the nodes that have led to unvisited nodes
  - If it exists, the DFS tree contains the path connecting N_s to N_g

Depth-first search (DFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Unvisited
N_2	Unvisited
N_3	Unvisited
N_4	Unvisited
N_5	Unvisited
N_6	Unvisited
N_7	Unvisited
N_8	Unvisited
N_9	Unvisited
N_{10}	Unvisited
N_{11}	Unvisited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Unvisited



DFS TREE

OPEN

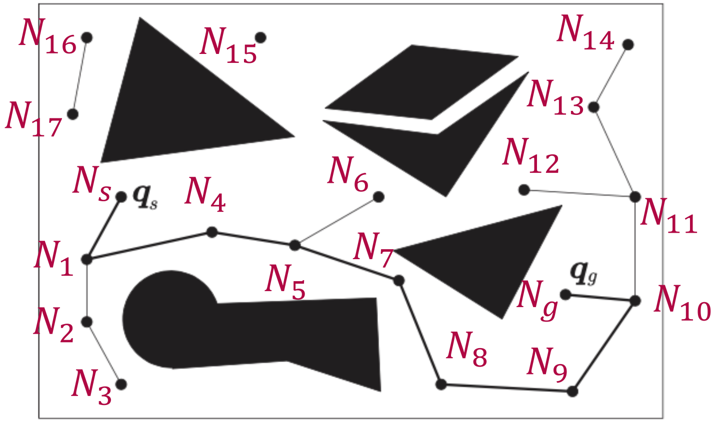
Node (visited from)
N_s

The *visited from* part is taken from the adjacency list

Depth-first search (DFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Unvisited
N_3	Unvisited
N_4	Unvisited
N_5	Unvisited
N_6	Unvisited
N_7	Unvisited
N_8	Unvisited
N_9	Unvisited
N_{10}	Unvisited
N_{11}	Unvisited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Unvisited



DFS TREE

N_s

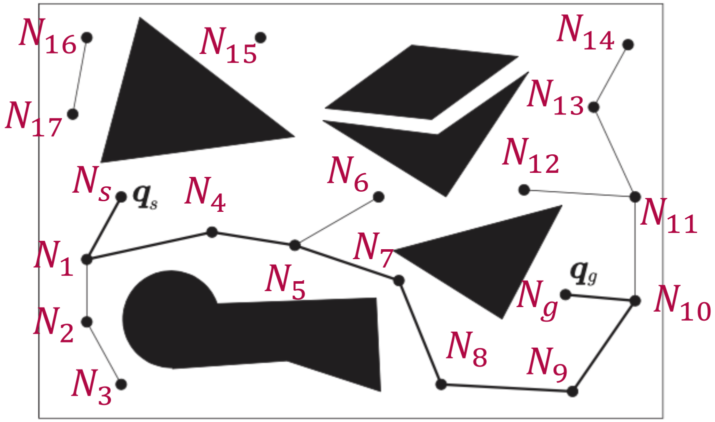
OPEN

Nodes (visited from)
$N_1(N_s)$

Depth-first search (DFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Visited
N_3	Unvisited
N_4	Visited
N_5	Unvisited
N_6	Unvisited
N_7	Unvisited
N_8	Unvisited
N_9	Unvisited
N_{10}	Unvisited
N_{11}	Unvisited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Unvisited



DFS TREE

N_s

|

N_1

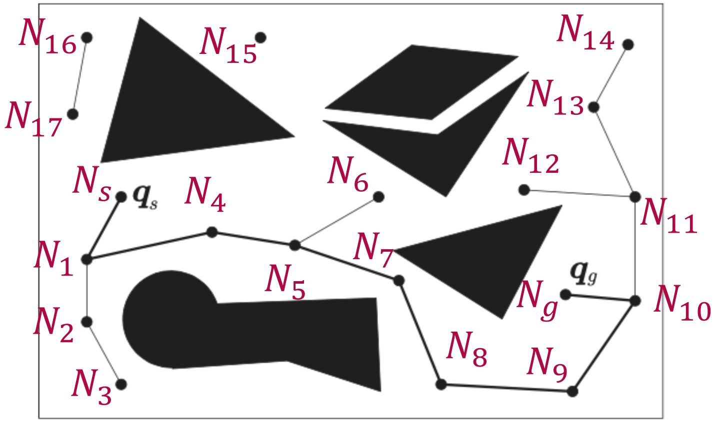
OPEN

Nodes (visited from)
N_2 (N_1)
N_4 (N_1)

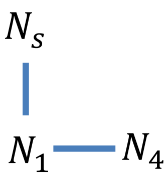
Depth-first search (DFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Visited
N_3	Unvisited
N_4	Visited
N_5	Visited
N_6	Unvisited
N_7	Unvisited
N_8	Unvisited
N_9	Unvisited
N_{10}	Unvisited
N_{11}	Unvisited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Unvisited



DFS TREE



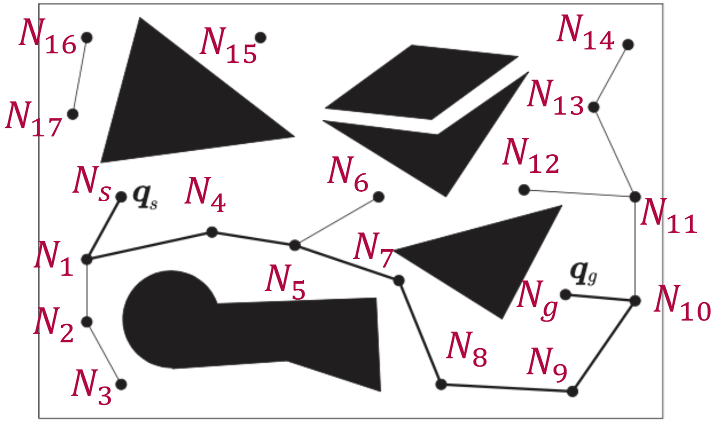
OPEN

Nodes (visited from)
N_2 (N_1)
N_5 (N_4)

Depth-first search (DFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Visited
N_3	Unvisited
N_4	Visited
N_5	Visited
N_6	Visited
N_7	Visited
N_8	Unvisited
N_9	Unvisited
N_{10}	Unvisited
N_{11}	Unvisited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Unvisited



DFS TREE

N_s

N_1 — N_4 — N_5

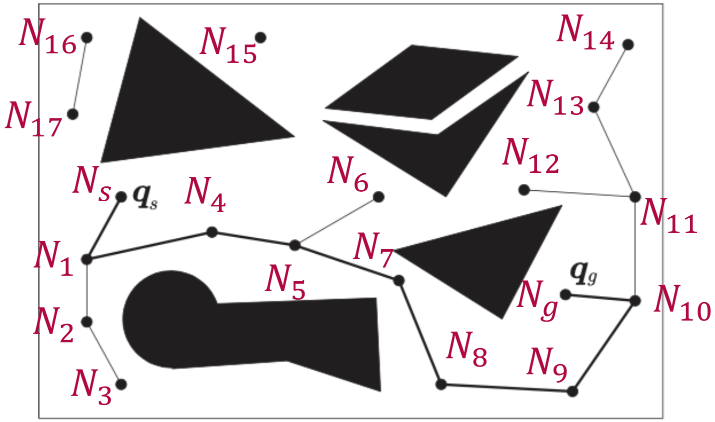
OPEN

Nodes (visited from)
N_2 (N_1)
N_6 (N_5)
N_7 (N_5)

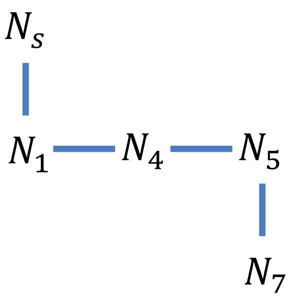
Depth-first search (DFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Visited
N_3	Unvisited
N_4	Visited
N_5	Visited
N_6	Visited
N_7	Visited
N_8	Visited
N_9	Unvisited
N_{10}	Unvisited
N_{11}	Unvisited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Unvisited



DFS TREE



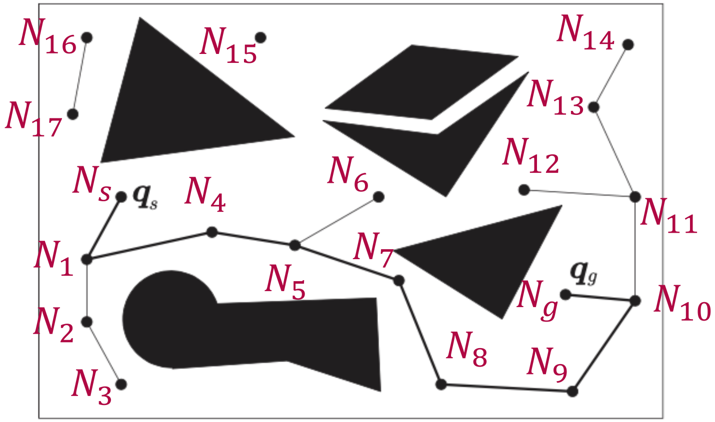
OPEN

Nodes (visited from)
N_2 (N_1)
N_6 (N_5)
N_8 (N_7)

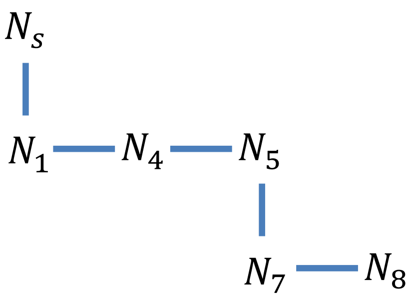
Depth-first search (DFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Visited
N_3	Unvisited
N_4	Visited
N_5	Visited
N_6	Visited
N_7	Visited
N_8	Visited
N_9	Visited
N_{10}	Unvisited
N_{11}	Unvisited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Unvisited



DFS TREE



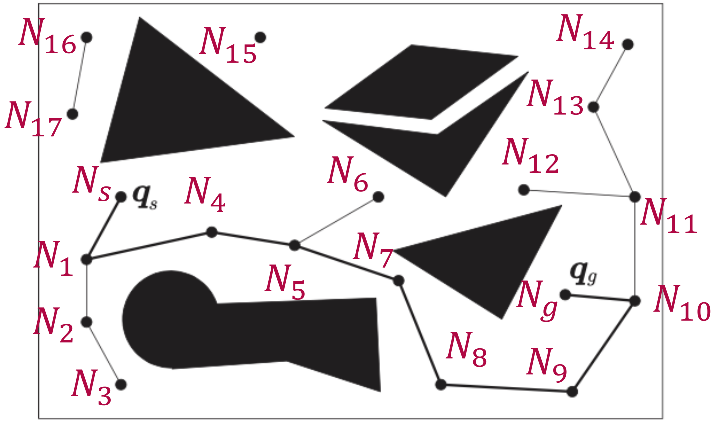
OPEN

Nodes (visited from)
N_2 (N_1)
N_6 (N_5)
N_9 (N_8)

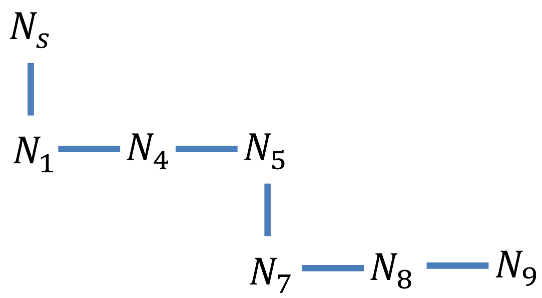
Depth-first search (DFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Visited
N_3	Unvisited
N_4	Visited
N_5	Visited
N_6	Visited
N_7	Visited
N_8	Visited
N_9	Visited
N_{10}	Visited
N_{11}	Unvisited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Unvisited



DFS TREE



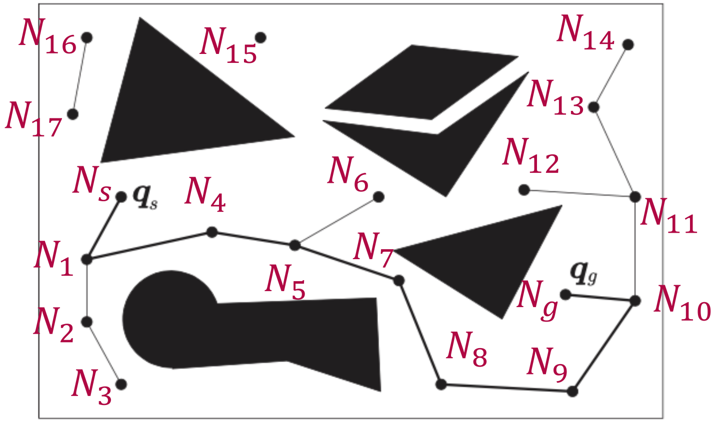
OPEN

Nodes (visited from)
$N_2\ (N_1)$
$N_6\ (N_5)$
$N_{10}\ (N_9)$

Depth-first search (DFS): Example

Node list

Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Visited
N_3	Unvisited
N_4	Visited
N_5	Visited
N_6	Visited
N_7	Visited
N_8	Visited
N_9	Visited
N_{10}	Visited
N_{11}	Visited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Visited



OPEN

Nodes (visited from)
N_2 (N_1)
N_6 (N_5)
N_g (N_{10})
N_{11} (N_{10})

DFS TREE

N_s

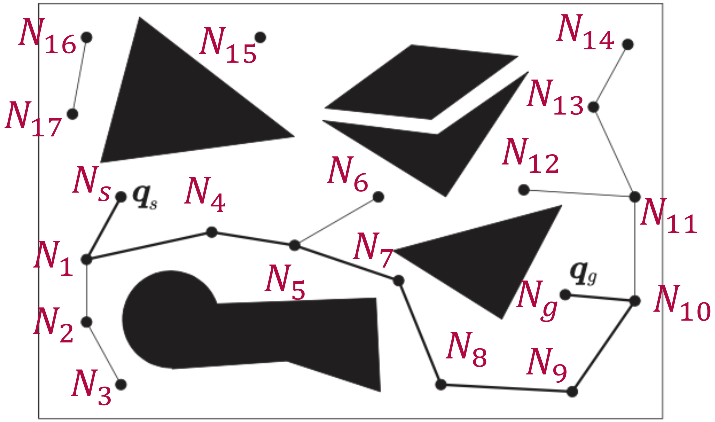
N_1 — N_4 — N_5

N_7 — N_8 — N_9 — N_{10}

Depth-first search (DFS): Example

Node list

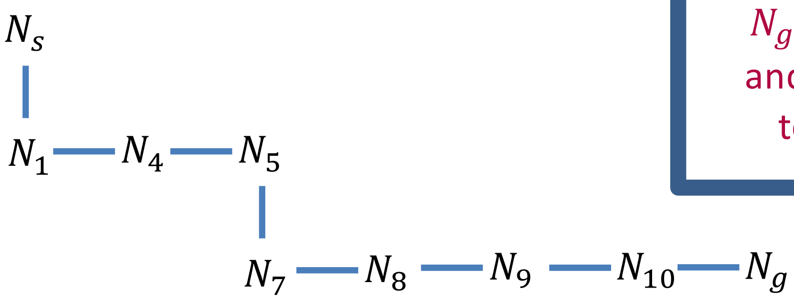
Nodes	Visited/Unvisited
N_s	Visited
N_1	Visited
N_2	Visited
N_3	Unvisited
N_4	Visited
N_5	Visited
N_6	Visited
N_7	Visited
N_8	Visited
N_9	Visited
N_{10}	Visited
N_{11}	Unvisited
N_{12}	Unvisited
N_{13}	Unvisited
N_{14}	Unvisited
N_{15}	Unvisited
N_{16}	Unvisited
N_{17}	Unvisited
N_g	Unvisited



OPEN

Nodes (visited from)
N_2 (N_1)
N_6 (N_5)
N_g (N_{10})
N_{11} (N_{10})

DFS TREE



DFS ends since N_g is in OPEN and it is added to the tree

A* algorithm

- In many applications, the arcs of the graph G are weighted with positive numbers
  - In this way, it is possible to define a cost of a path on G as the sum of the weights on its arcs
  - The problem is now connecting N_s to N_g through a path associated to the minimum cost, called minimum path
- It is possible to associate a cost function associated to a node $N_i, i = 1, \dots, N$

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

  - $g(N_i)$ is the cost of the path from N_s to N_i as stored in the current tree
  - $h(N_i)$ is a heuristic estimate of the cost $h^*(N_i)$ of the minimum path from N_i to N_g
    - Any choice of the heuristic such that $0 \leq h(N_i) \leq h^*(N_i), \forall N_i$ is admissible, meaning that the heuristic should not overestimate the real cost
    - The choice $h(N_i) = 0$ corresponds to the Dijkstra algorithm

- The A* algorithm uses an ordinated list
  - We refer to this list as OPEN
- At the beginning, OPEN contains the node N_s only and it is marked as visited
  - The other nodes are marked as unvisited
- At each iteration, the node in OPEN associated to the minimum cost function is extracted
  - We refer to this node as N_{best}
- The search terminates once N_g is extracted from OPEN, or OPEN is empty (failure)
- During this search, the algorithm must keep track of the A* tree, containing all the nodes that have led to unvisited nodes
  - If it exists, the DFS tree contains the path connecting N_s to N_g
  - The tree adjusts the pointer from N_i to N_{best} if $g(N_{best}) + c(N_{best}, N_i) < g(N_i)$, with $c(N_{best}, N_i)$ the cost from N_i to N_{best}

- Pseudocode

$A^\star$ algorithm

repeat
  find and extract N_{best} from OPEN
  if $N_{\text{best}} = N_g$ **then** exit
  for each node N_i in ADJ(N_{best}) **do**
    if N_i is *unvisited* **then**
      add N_i to T with a pointer toward N_{best}
      insert N_i in OPEN; mark N_i *visited*
    else if $g(N_{\text{best}}) + c(N_{\text{best}}, N_i) < g(N_i)$ **then**
      redirect the pointer of N_i in T toward N_{best}
      if N_i is not in OPEN **then**
        insert N_i in OPEN
      else update $f(N_i)$
      end if
    end if
until OPEN is empty

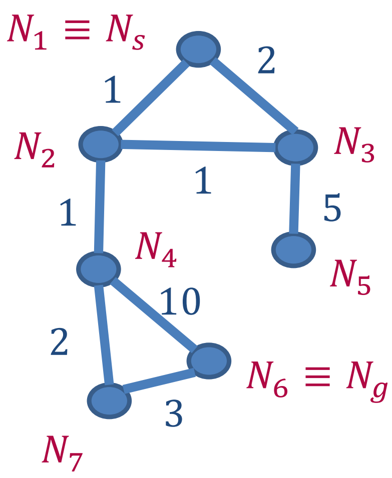
A* algorithm: Example

Node list

Nodes	Visited/Unvisited
N_1	Visited
N_2	Unvisited
N_3	Unvisited
N_4	Unvisited
N_5	Unvisited
N_6	Unvisited
N_7	Unvisited

Adjacency list

Nodes	Adjacent nodes
N_1	N_2, N_3
N_2	N_1, N_3, N_4
N_3	N_1, N_2, N_5
N_4	N_2, N_6, N_7
N_5	N_3
N_6	N_4, N_7
N_7	N_4, N_6



DFS TREE

OPEN

Node (visited from / cost)
N_1

The *visited from* part is taken from the adjacency list

A^* algorithm

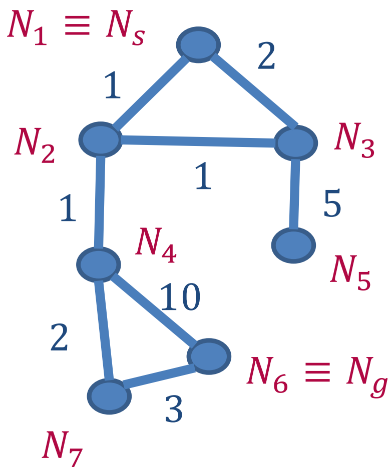
repeat
  find and extract N_{best} from OPEN
  if $N_{\text{best}} = N_g$ **then** exit
  for each node N_i in ADJ(N_{best}) **do**
    if N_i is *unvisited* **then**
      add N_i to T with a pointer toward N_{best}
      insert N_i in OPEN; mark N_i *visited*
    else if $g(N_{\text{best}}) + c(N_{\text{best}}, N_i) < g(N_i)$ **then**
      redirect the pointer of N_i in T toward N_{best}
      if N_i is not in OPEN **then**
        insert N_i in OPEN
      else update $f(N_i)$
      end if
    end if
until OPEN is empty

Node list

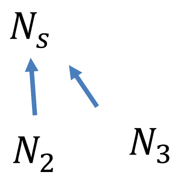
Nodes	Visited/Unvisited
N_1	Visited
N_2	Visited
N_3	Visited
N_4	Unvisited
N_5	Unvisited
N_6	Unvisited
N_7	Unvisited

Adjacency list

Nodes	Adjacent nodes
N_1	N_2, N_3
N_2	N_1, N_3, N_4
N_3	N_1, N_2, N_5
N_4	N_2, N_6, N_7
N_5	N_3
N_6	N_4, N_7
N_7	N_4, N_6



DFS TREE



OPEN

Node (visited from / cost)
$N_2(N_1/1)$
$N_3(N_1/2)$

```
A* algorithm
  repeat
    find and extract N_best from OPEN
    if N_best = N_g then exit
    for each node N_i in ADJ(N_best) do
      if N_i is unvisited then
        add N_i to T with a pointer toward N_best
        insert N_i in OPEN; mark N_i visited
      else if g(N_best) + c(N_best, N_i) < g(N_i) then
        redirect the pointer of N_i in T toward N_best
        if N_i is not in OPEN then
          insert N_i in OPEN
        else update f(N_i)
        end if
      end if
  until OPEN is empty
```

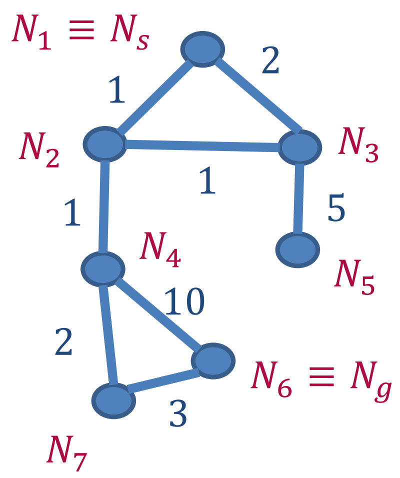
A* algorithm: Example

Node list

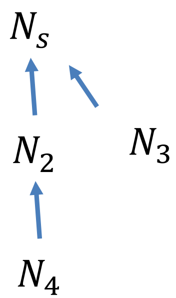
Nodes	Visited/Unvisited
N_1	Visited
N_2	Visited
N_3	Visited
N_4	Visited
N_5	Unvisited
N_6	Unvisited
N_7	Unvisited

Adjacency list

Nodes	Adjacent nodes
N_1	N_2, N_3
N_2	N_1, N_3, N_4
N_3	N_1, N_2, N_5
N_4	N_2, N_6, N_7
N_5	N_3
N_6	N_4, N_7
N_7	N_4, N_6



DFS TREE



OPEN

Node (visited from / cost)
$N_3(N_1/2)$
$N_4(N_2/2)$

$g(N_2) + c(N_2, N_1) < g(N_1)$ is not verified since 1+1<0

$g(N_2) + c(N_2, N_3) < g(N_3)$ is not verified since 1+1<2

```
A* algorithm
  repeat
    find and extract N_best from OPEN
    if N_best = N_g then exit
    for each node N_i in ADJ(N_best) do
      if N_i is unvisited then
        add N_i to T with a pointer toward N_best
        insert N_i in OPEN; mark N_i visited
      else if g(N_best) + c(N_best, N_i) < g(N_i) then
        redirect the pointer of N_i in T toward N_best
        if N_i is not in OPEN then
          insert N_i in OPEN
        else update f(N_i)
        end if
      end if
  until OPEN is empty
```

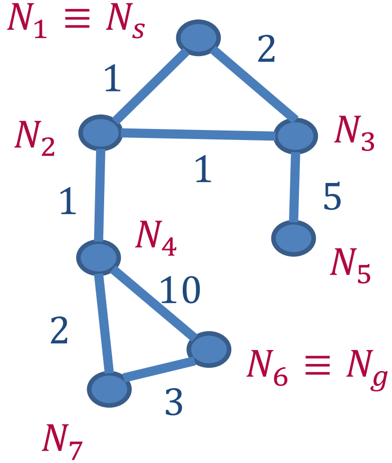
A* algorithm: Example

Node list

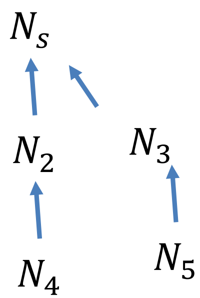
Nodes	Visited/Unvisited
N_1	Visited
N_2	Visited
N_3	Visited
N_4	Visited
N_5	Visited
N_6	Unvisited
N_7	Unvisited

Adjacency list

Nodes	Adjacent nodes
N_1	N_2, N_3
N_2	N_1, N_3, N_4
N_3	N_1, N_2, N_5
N_4	N_2, N_6, N_7
N_5	N_3
N_6	N_4, N_7
N_7	N_4, N_6



DFS TREE



OPEN

Node (visited from / cost)
$N_4(N_2/2)$
$N_5(N_3/7)$

$g(N_3) + c(N_3, N_1) < g(N_1)$ is not verified since 2+2<0

$g(N_3) + c(N_2, N_3) < g(N_2)$ is not verified since 2+1<1

```
A* algorithm
  repeat
    find and extract N_best from OPEN
    if N_best = N_g then exit
    for each node N_i in ADJ(N_best) do
      if N_i is unvisited then
        add N_i to T with a pointer toward N_best
        insert N_i in OPEN; mark N_i visited
      else if g(N_best) + c(N_best, N_i) < g(N_i) then
        redirect the pointer of N_i in T toward N_best
        if N_i is not in OPEN then
          insert N_i in OPEN
        else update f(N_i)
        end if
      end if
  until OPEN is empty
```

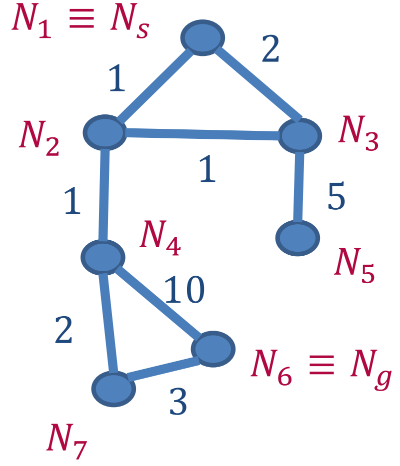
A* algorithm: Example

Node list

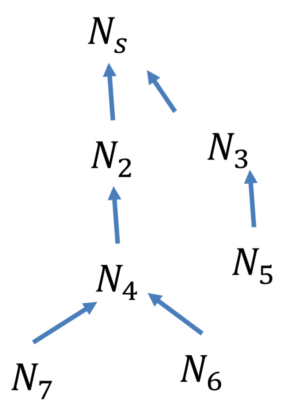
Nodes	Visited/Unvisited
N_1	Visited
N_2	Visited
N_3	Visited
N_4	Visited
N_5	Visited
N_6	Visited
N_7	Visited

Adjacency list

Nodes	Adjacent nodes
N_1	N_2, N_3
N_2	N_1, N_3, N_4
N_3	N_1, N_2, N_5
N_4	N_2, N_6, N_7
N_5	N_3
N_6	N_4, N_7
N_7	N_4, N_6



DFS TREE



OPEN

Node (visited from / cost)
$N_7(N_4/4)$
$N_5(N_3/7)$
$N_6(N_4/12)$

$g(N_4) + c(N_4, N_2) < g(N_2)$ is not verified since 2+1<1

```
A* algorithm
  repeat
    find and extract N_best from OPEN
    if N_best = N_g then exit
    for each node N_i in ADJ(N_best) do
      if N_i is unvisited then
        add N_i to T with a pointer toward N_best
        insert N_i in OPEN; mark N_i visited
      else if g(N_best) + c(N_best, N_i) < g(N_i) then
        redirect the pointer of N_i in T toward N_best
        if N_i is not in OPEN then
          insert N_i in OPEN
        else update f(N_i)
        end if
      end if
  until OPEN is empty
```

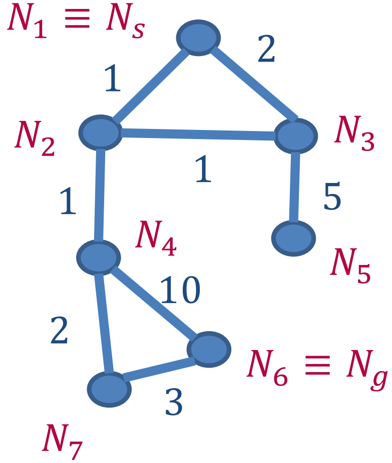
A* algorithm: Example

Node list

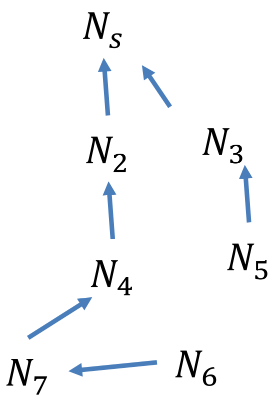
Nodes	Visited/Unvisited
N_1	Visited
N_2	Visited
N_3	Visited
N_4	Visited
N_5	Visited
N_6	Visited
N_7	Visited

Adjacency list

Nodes	Adjacent nodes
N_1	N_2, N_3
N_2	N_1, N_3, N_4
N_3	N_1, N_2, N_5
N_4	N_2, N_6, N_7
N_5	N_3
N_6	N_4, N_7
N_7	N_4, N_6



DFS TREE



OPEN

Node (visited from / cost)
$N_5(N_3/7)$
$N_6(N_4/12)$

$g(N_7) + c(N_7, N_4) < g(N_4)$ is not verified since 4+2<2
$g(N_7) + c(N_7, N_6) < g(N_6)$ IS VERIFIED since 4+3<12

```
A* algorithm
repeat
  find and extract N_best from OPEN
  if N_best = N_g then exit
  for each node N_i in ADJ(N_best) do
    if N_i is unvisited then
      add N_i to T with a pointer toward N_best
      insert N_i in OPEN; mark N_i visited
    else if g(N_best) + c(N_best, N_i) < g(N_i) then
      redirect the pointer of N_i in T toward N_best
      if N_i is not in OPEN then
        insert N_i in OPEN
      else update f(N_i)
      end if
    end if
until OPEN is empty
```

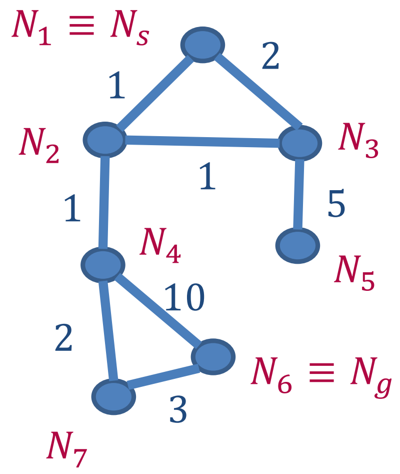
A* algorithm: Example

Node list

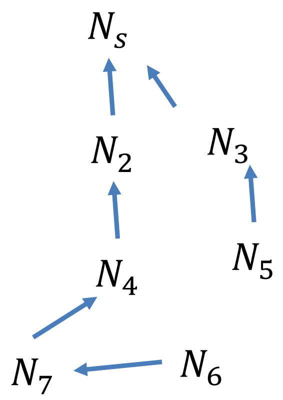
Nodes	Visited/Unvisited
N_1	Visited
N_2	Visited
N_3	Visited
N_4	Visited
N_5	Visited
N_6	Visited
N_7	Visited

Adjacency list

Nodes	Adjacent nodes
N_1	N_2, N_3
N_2	N_1, N_3, N_4
N_3	N_1, N_2, N_5
N_4	N_2, N_6, N_7
N_5	N_3
N_6	N_4, N_7
N_7	N_4, N_6



DFS TREE



OPEN

Node (visited from / cost)
$N_6(N_4/12)$

$g(N_5) + c(N_5, N_3) < g(N_3)$ is not verified since 7+5<2

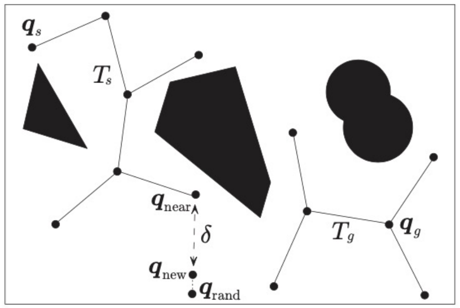
N_6 is then extracted and the algorithm ends

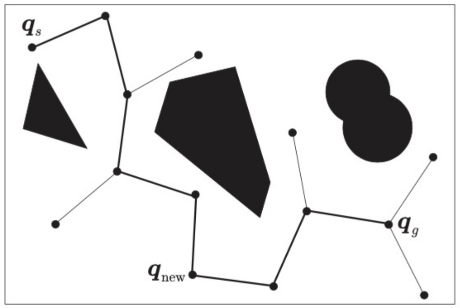
```
A* algorithm
  repeat
    find and extract N_best from OPEN
    if N_best = N_g then exit
    for each node N_i in ADJ(N_best) do
      if N_i is unvisited then
        add N_i to T with a pointer toward N_best
        insert N_i in OPEN; mark N_i visited
      else if g(N_best) + c(N_best, N_i) < g(N_i) then
        redirect the pointer of N_i in T toward N_best
        if N_i is not in OPEN then
          insert N_i in OPEN
        else update f(N_i)
        end if
      end if
  until OPEN is empty
```

Rapidly-exploring random tree method (RRT)

- It might be a good idea not to build a roadmap describing the entire C_{free}
  - It may be useful to explore only the part of C_{free} connecting q_s to q_g
- Denote with G the current graph
  - The first step is the generation of a random model-configuration q_{rand} as for the PRM with a uniform probability distribution
  - Given G, the model-configuration $q_{near} \in G$ closer to q_{rand} is found
  - A new model-configuration, q_{new}, is chosen as the segment joining q_{near} and q_{rand} at a pre-determined distance δ from q_{near}
  - A collision-check is carried out for q_{new} and the segment connecting q_{near} and q_{new}
  - If there are no collisions, q_{new} and its segment connecting q_{near} are added to G
  - Notice that q_{rand} can also be a point belonging to the CO-obstacle region

- To speed-up the search, two graphs are considered
  - One graph, G_s, starts from q_s
  - The other graph, G_g, starts from q_g
  - They evolve in parallel
- At a certain point (i.e., after a certain number of iterations), G_s must be connected to G_g
  - In this phase, q_{new} acts as q_{rand} for G_g
  - One finds the closest q_{near} in G_g and moves it trying to have $q_{rand} = q_{new}$ with a variable step-size instead of a fixed δ
  - If this segment is free from collisions, the graphs are connected





Artificial potentials method

- This method is suitable for on-line applications
  - The obstacles are not known in advance
  - Sensors are indeed needed
  - It is employed also off-line
- The aim is not to build C_{free}, but only to reach q_g
- The artificial potentials method exploits the superimposition of two terms
  - An attractive potential to the goal
  - A repulsive potential away from the CO-obstacle region

Attractive potential

- The robot must be guided to q_g through the potential

$$U_a(q) = \frac{1}{2}k_a e^T(q)e(q) = \frac{1}{2}k_a\|e(q)\|^2$$

  - $k_a > 0$
  - $e(q) = q_g - q$
  - $U_a(q)$ has the global minimum in $e(q) = 0$
- The resulting force from this potential is

$$f_a(q) = -\nabla U_a(q) = k_a e(q)$$

  - Applying this force to the robot, q tends to q_g linearly

Attractive potential

- Another choice for the potential might be

$$U_b(q) = k_a ||e(q)||$$

  - $k_a > 0$
  - $e(q) = q_g - q$
  - $U_b(q)$ has the global minimum in $e(q) = 0$

- The resulting force from this potential is

$$f_b(q) = -\nabla U_b(q) = k_a \frac{e(q)}{||e(q)||}$$

  - Notice that this force is not defined when $e(q) = 0$, therefore it is not denoted when $q = q_g$, the desired model-configuration
  - The advantage in applying this force is that $f_b(q)$ does not go to infinite when $e(q)$ increases in norm, therefore it is suitable for large initial errors
  - It is slower than $f_a(q)$ close to the desired model-configuration q_g

- A way to combine the two forces is to use

$$f(x) = \begin{cases} f_a(q), & \|e(q)\| < 1 \\ f_b(q), & \|e(q)\| \geq 1 \end{cases}$$

  - The transition is smooth when $e(q) = 1$, avoiding jumps in the control actions

Repulsive potential

- The repulsive potential is needed to avoid collisions
  - The idea is to build a sort of a virtual barrier potential around the obstacles
- It is assumed that CO is convex, or partitioned in convex components $CO_i, i = 1, \dots, p$
- For each CO_i, the associated potential is

$$U_{r,i}(q) = \begin{cases} \dfrac{k_{r,i}}{\gamma}\left(\dfrac{1}{\eta_i(q)} - \dfrac{1}{\eta_{o,i}}\right)^{\gamma}, & \eta_i(q) \leq \eta_{o,i} \\ 0, & \eta_i(q) > \eta_{o,i} \end{cases}$$

  - $k_{r,i} > 0$ is a gain
  - $\eta_i\,(q) = \min_{q' \in CO_i} \|q - q'\|$ is the distance from the obstacle
  - $\eta_{o,i}$ is the range of influence of the obstacle
  - $\gamma = \{2,3\}$ is a factor
  - Notice that $U_{r,i}(q)$ is zero outside the range of influence and positive inside

- The repulsive force is

$$f_{r,i}(q) = -\nabla U_{r,i}(q) = \begin{cases} \dfrac{k_{r,i}}{\eta_i(q)^2}\left(\dfrac{1}{\eta_i\,(q)} - \dfrac{1}{\eta_{o,i}}\right)^{\gamma-1} \nabla\eta_i(q), & \eta_i(q) \le \eta_{o,i} \\ 0, & \eta_i(q) > \eta_{o,i} \end{cases}$$

  - The convexity hypothesis regarding the obstacles is necessary to compute $\nabla\eta_i(q)$ analytically
  - During on-line planning, $\nabla\eta_i(q)$ must be computed numerically
  - Notice that $\eta_i(q_g) \ge \eta_{o,i}, i = 1, \dots, p$, meaning that the final model-configuration is sufficiently outside any influence region

- The sum of all the repulsive potentials and forces is

$$U_r(q) = \sum_{i=1}^{p} U_{r,i}(q)$$

$$f_r(q) = -\nabla U_r(q) = \sum_{i=1}^{p} f_{r,i}(q)$$

- The total potential is the sum of the attractive and the repulsive one

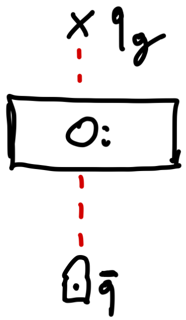
$$U_t(q) = U_a(q) + U_r(q) > 0$$

  - This total potential has a global minimum in q_g by construction
- The resulting force field is

$$f_t(q) = -\nabla U_t(q) = f_a(q) + f_r(q)$$

  - This is called deepest descent method
- The artificial potentials method suffers of local minima in which

$$\begin{cases} f_t(\bar{q}) = 0 \\ U_t(\bar{q}) > 0 \end{cases} \text{with } \bar{q} \neq q_g$$



$f_a(\bar{q}) = f_r(\bar{q})$

$U_T(\bar{q}) > 0$

- There are different interpretations for the total force
  - $f_t(q) = \tau$
    - The total force is seen as the vector of generalized forces (force plus torques) that induce a motion of the robot in accordance with its dynamic model
    - In case of on-line planning, the total force is directly the control input
    - In case of off-line planning, it generates smooth trajectories since the reactions of the robot are filtered by the robot dynamcis
  - $f_t(q) = \ddot{q}$
    - The total force is seen as the acceleration moving the robot that is considered as a point mass
    - In case of on-line planning, it requires the solution of the inverse dynamic problem or an on-line 2nd order kinematic control scheme
    - In case of off-line planning, it is an intermediate case as the above one and the below one
  - $f_t(q) = \dot{q}$
    - The total force is seen as the velocity vector moving the robot, that is considered from a kinematic viewpoint only
    - In case of on-line planning, it requires the solution of an on-line 1st order kinematic control scheme
    - In case of off-line planning, it is faster to generate trajectories as $q(t)$
    - This is the only method ensuring that q_g is reached with zero velocity, $\dot{q} = 0$, while the other methods require the addition of a damping term in $f_a(q)$
    - This is the most common choice, where $q_{k+1} = q_k + T_s f_t(q)$, that is the next model-configuration is the actual one plus the sampling time multiplied by the total force given by the artificial potentials methodology

- If a local minima is recognised, one solution is to stop the execution of the artificial potentials method and perform some random motions
  - Be careful of the environment!
- The artificial potentials method can be re-activated afterwards
- In case of on-line planning without any prior information or reconstruction of the environment, this is the only method that we consider here
  - Other methods can be used in case of off-line planning and they are explained in the following

Local minima – Best-first algorithm

- Suppose to discretize C_{free} using a regular grid
- Each free cell of this grid is assigned to a value of $U_t(q_c)$, where q_c is the model-configuration of the cell's centre
- The algorithm builds a graph rooted at q_s aiming at q_g
- At each iteration, the adjacent cells (4,8,...) of the node with a minimum $U_t(q_c)$ are considered
  - Those nodes that are not in the graph are added as children of the considered node
- The algorithm stops when the cell containing q_g is in the graph
  - Otherwise, a failure is reported
- This best-first algorithm is the discretized version of the steepest descent method
- This best-first algorithm is used to exit a local minimum, the artificial potentials method is re-activated afterwards, or it is employed from the beginning

- The best-first algorithm, in general, may lead to inefficient paths
  - The robot can enter into other local minima
- The numerical navigation function is a potential built on the grid-map of C_{free} associated to its distance from q_g
  - $0 \rightarrow$ is assigned to the cell containing q_g
  - $1 \rightarrow$ is assigned to the adjacent cells of q_g
  - $2 \rightarrow$ is assigned to the unmarked cells among those adjacent to the ones marked with 1
  - …
- The steepest descent method can be applied on this grid-map
  - Again, the navigation function can be applied to exit from the local minima only
  - It can be also applied from the beginning as an off-line planning

2	1	2	3	4	5	6	7	8	9		19
1	0	1			6	7	8	9	10		18
2	1	2	3		7	8		10	11		17
3		3	4	5	6	7	8		12		16
4			5	6	7			12	13		15
5	6	7	6	7	8	9	10	11	12	13	14
6	7	8	7	8	9	10	11	12	13	14	15