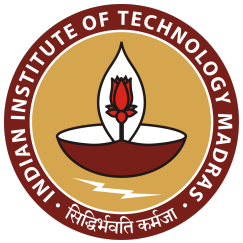
Introduction to Robotics

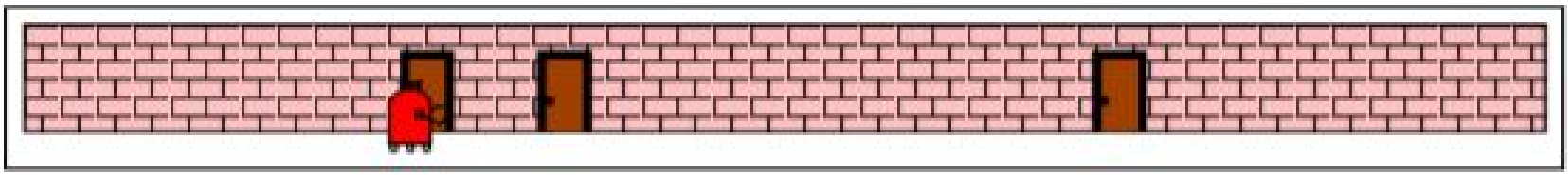
Week-4

Prof. Balaraman Ravindran

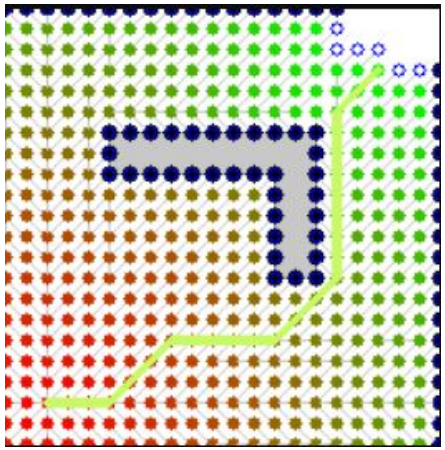
IIT-Madras



- We again consider the problem of Mobile robot localization - that is, determining the pose of a robot relative to a given map of the environment.



- We then examine path planning strategies for robot motion.



A Taxonomy of Localization Problems

We will divide localization problems according to:

- The initial knowledge that a robot may possess relative to the localization problem.
- The nature of the environment.
- Whether or not the localization algorithm controls the motion of the robot.

Local vs Global Localization

(Characterized by the type of knowledge that is available initially and at run-time.)

- Position tracking: Assumes that the *initial robot pose is known*.

  Localizing the robot can be achieved by accommodating the noise in robot motion.

  - Often rely on the assumption that the pose error is small.
  - The pose uncertainty is often approximated by a unimodal distribution (e.g., a Gaussian).
  - The position tracking problem is a local problem, since the uncertainty is local and confined to a region near the robot's true pose.

- Global localization: Here the *initial pose of the robot is unknown*.

  The robot is initially placed somewhere in its environment, but it lacks knowledge of where it is.

  - Approaches to global localization cannot assume boundedness of the pose error, and unimodal probability distributions are usually inappropriate.
  - More difficult, and in fact, subsumes the position tracking problem.

- Kidnapped Robot problem: A variant of the global localization problem, that is even more difficult.

  During operation, the robot can get kidnapped and teleported to some other location.

  - In global localization, the robot knows that it doesn't know where it is. In this problem however, the robot might believe it knows where it is while it does not.
  - The ability to recover from failures is essential for truly autonomous robots. Testing a localization algorithm by kidnapping it measures its ability to recover from global localization failures.

Static vs Dynamic Environments

- Static Environments:
  - Static environments are environments where the only variable quantity (state) is the robot's pose.
  - All other objects in the environments remain at the same location forever
- Dynamic Environments:
  - Dynamic environments possess objects other than the robot whose location or configuration changes over time.
  - Of particular interest are changes that persist over time. (Eg. people, daylight, movable furniture, or doors)
  - Changes that are not measurable are of no relevance to localization, and those that do not persist for multiple sensor readings are best treated as noise.

Passive vs Active Approaches

(Pertains to whether or not the localization algorithm controls the motion of the robot)

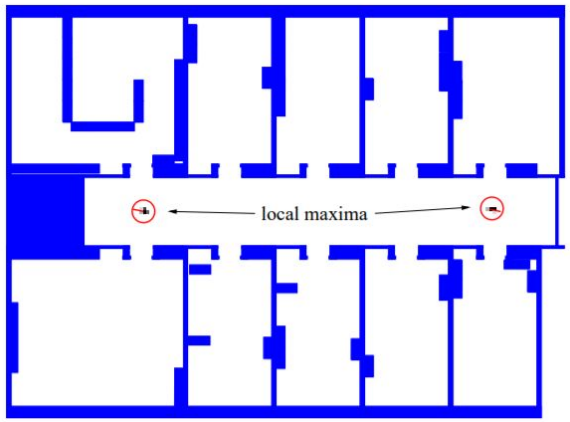
- Passive localization: The localization module only observes the robot operating.

  The robot is controlled through some other means, and the robot's motion is not aimed at facilitating localization. (For example, the robot might move randomly or perform its everyday's tasks.)

- Active localization: Active localization algorithms control the robot so as to minimize the localization error and/or the costs arising from moving a poorly localized robot into a hazardous place.

An example where active localization will perform better than passive localization

- Here the robot is located in a symmetric corridor, and its belief after navigating the corridor for a while is centered at two (symmetric) poses. Only if it moves into a room will it be able to eliminate the ambiguity and determine its pose.



- Instead of merely waiting until the robot incidentally moves into a room, active localization can recognize the impasse and send the robot there directly.

Markov Localization

- Probabilistic localization algorithms are variants of the Bayes filter. The straightforward application of Bayes filters to the localization problem is called Markov localization.

1: **Algorithm Markov_localization**($bel(x_{t-1}), u_t, z_t, m$)**:**
2: *for all* x_t *do*
3: $\overline{bel}(x_t) = \int p(x_t \mid u_t, x_{t-1}, m)\ bel(x_{t-1})\ dx$
4: $bel(x_t) = \eta\ p(z_t \mid x_t, m)\ \overline{bel}(x_t)$
5: *endfor*
6: *return* $bel(x_t)$

- Markov localization addresses the global localization problem, the position tracking problem, and the kidnapped robot problem in static environments.
  - Position Tracking: If initial pose is known exactly, then $bel(x_0)$ is initialized by a point-mass distribution:

$$bel(x_0) = \begin{cases} 1, & \text{if } x_0 = \overline{x_0} \\ 0, & \text{otherwise} \end{cases}$$

The initial pose is often just known in approximation. The belief $bel(x_0)$ is then usually initialized by a narrow Gaussian distribution centered around $\overline{x_0}$.

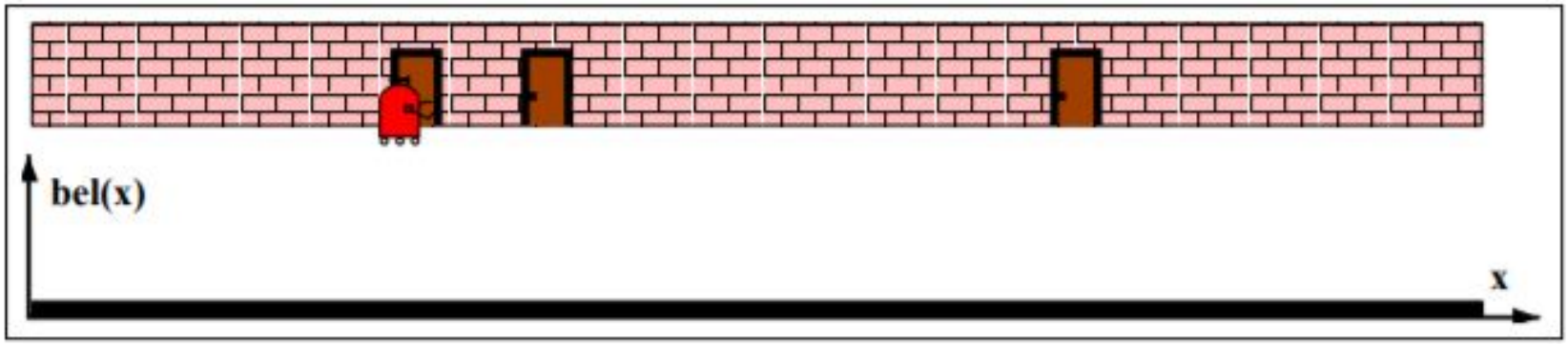
$$bel(x_0) = \mathcal{N}(x_0;\ \overline{x_0},\ \sigma)$$

- Global Localization: If the initial pose is unknown, $bel(x_0)$ is initialized by a uniform distribution over the space of all legal poses in the map.

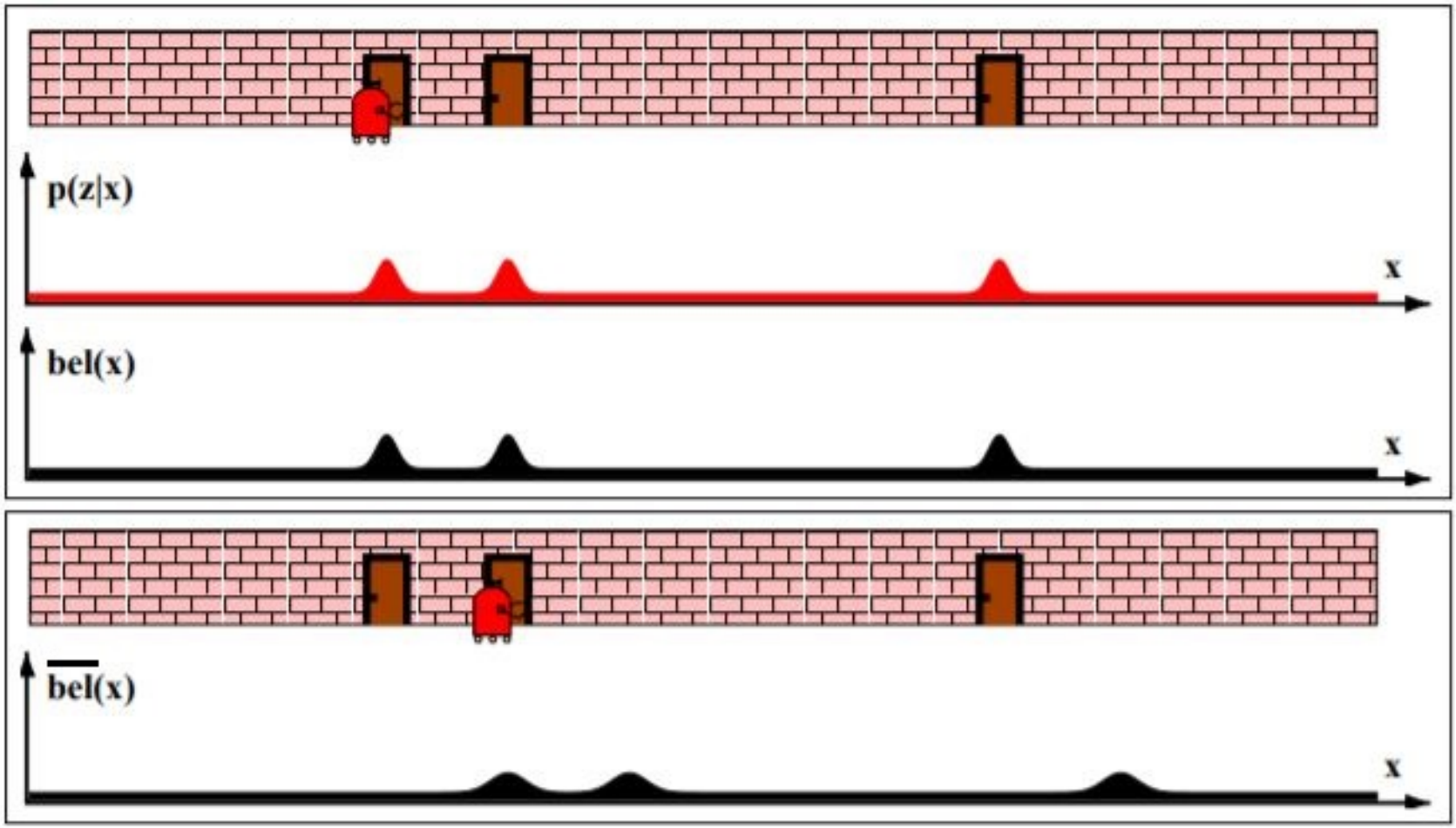
$$bel(x_0) = \frac{1}{|X|}$$

- Partial knowledge: Partial knowledge of the robot's position can usually easily be transformed into an appropriate initial distribution. For example, if the robot is known to start next to a door, one might initialize $bel(x_0)$ using a density that is zero except for places near doors, where it may be uniform.

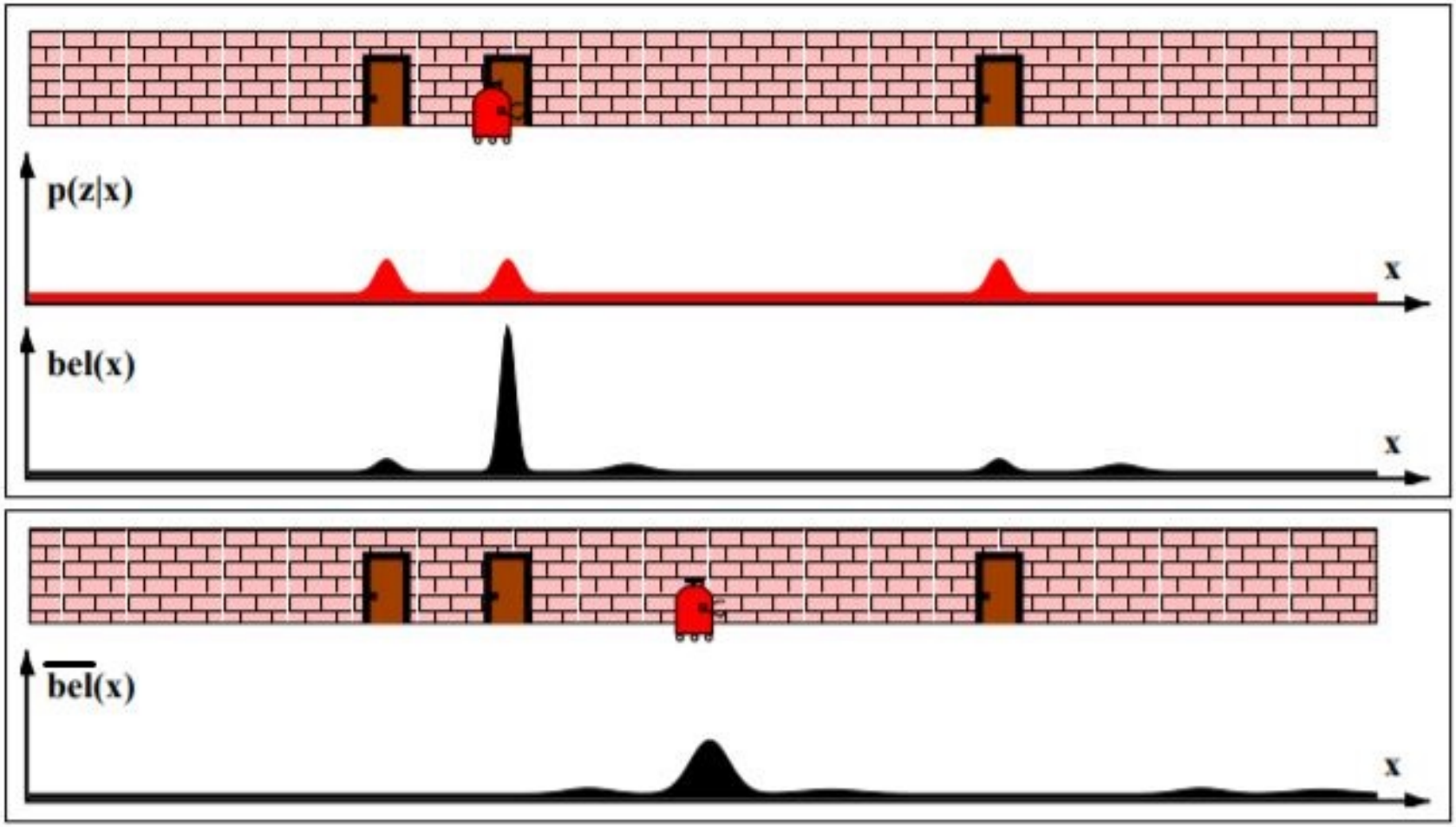
An Illustration of the Markov Localization Algorithm:



Localization



Localization



Sensor Models with Known Correspondence

Alternative to using raw sensor measurements, we can extract features from the measurements.

- In many robotics applications, features may correspond to distinct objects in the physical world (Landmarks).

$f(z_t) = \{f_t^1,\ f_t^2, ...\}$

To make sense of these sensor features, we need to define a variable that establishes correspondence between the feature f_t^i and the landmark m_j in the map.

This variable will be denoted by c_t^i, with:

$c_t^i \in \{1, ..., N+1\}$

Where, N is the number of landmarks in the map m.

If $c_t^i = j \leq N$, then the i^{th} feature observed at time t corresponds to the j^{th} landmark in the map.

- The only exception occurs with $c_t^i = N + 1$ Here a feature observation does not correspond to any feature in the map m. This case is important for handling spurious or unobserved landmarks.

Sensor Models with Unknown Correspondence

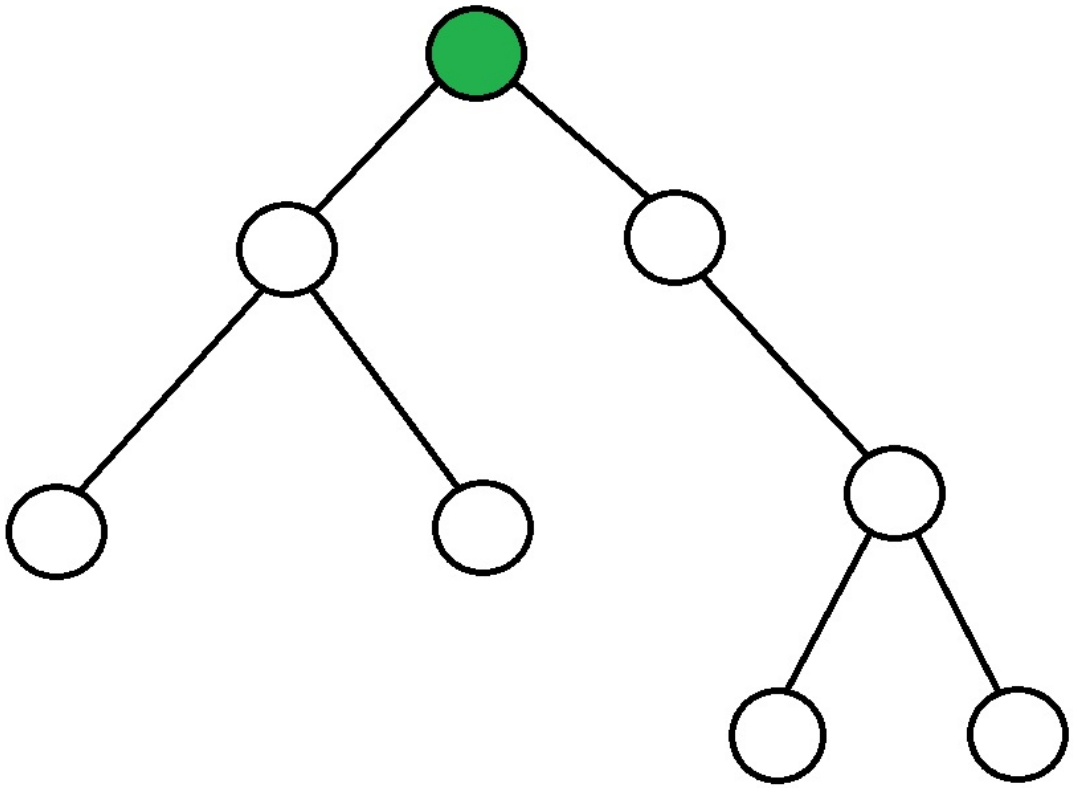
- In practice,landmark correspondences can rarely be determined with absolute certainty.
- A strategy to cope with the unknown correspondence problem is to consider the most likely value of the correspondence variable, and then take this value for granted (maximum likelihood correspondence).

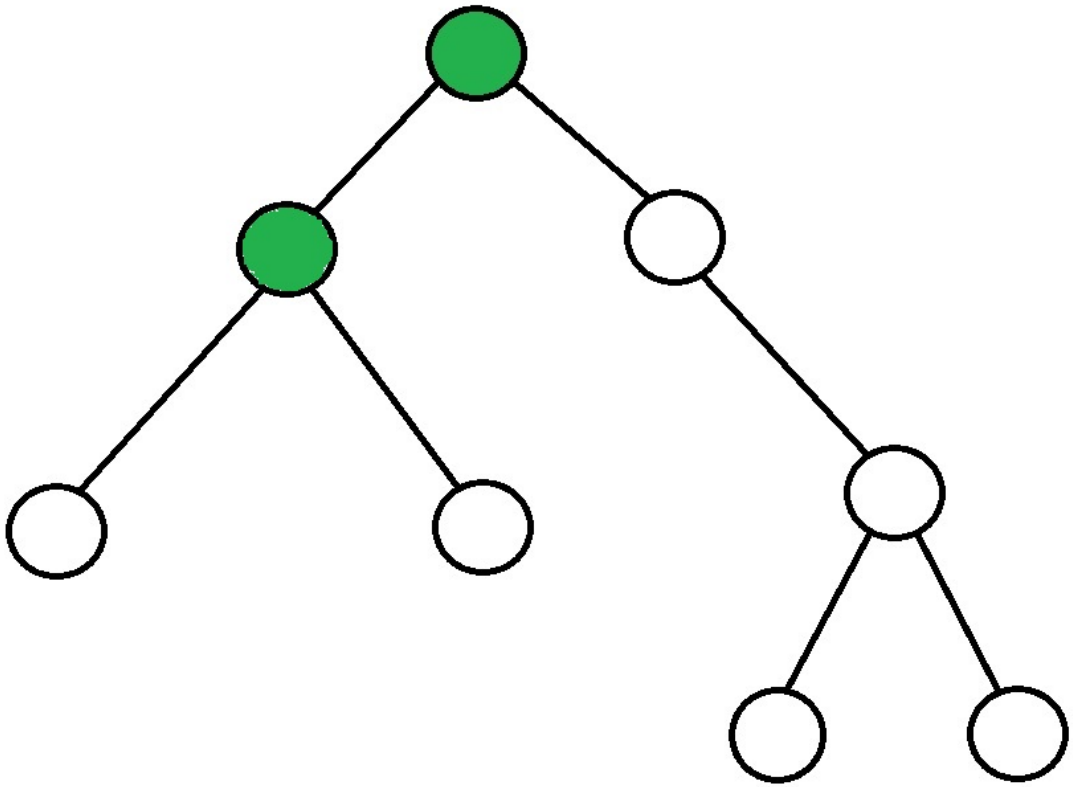
BFS

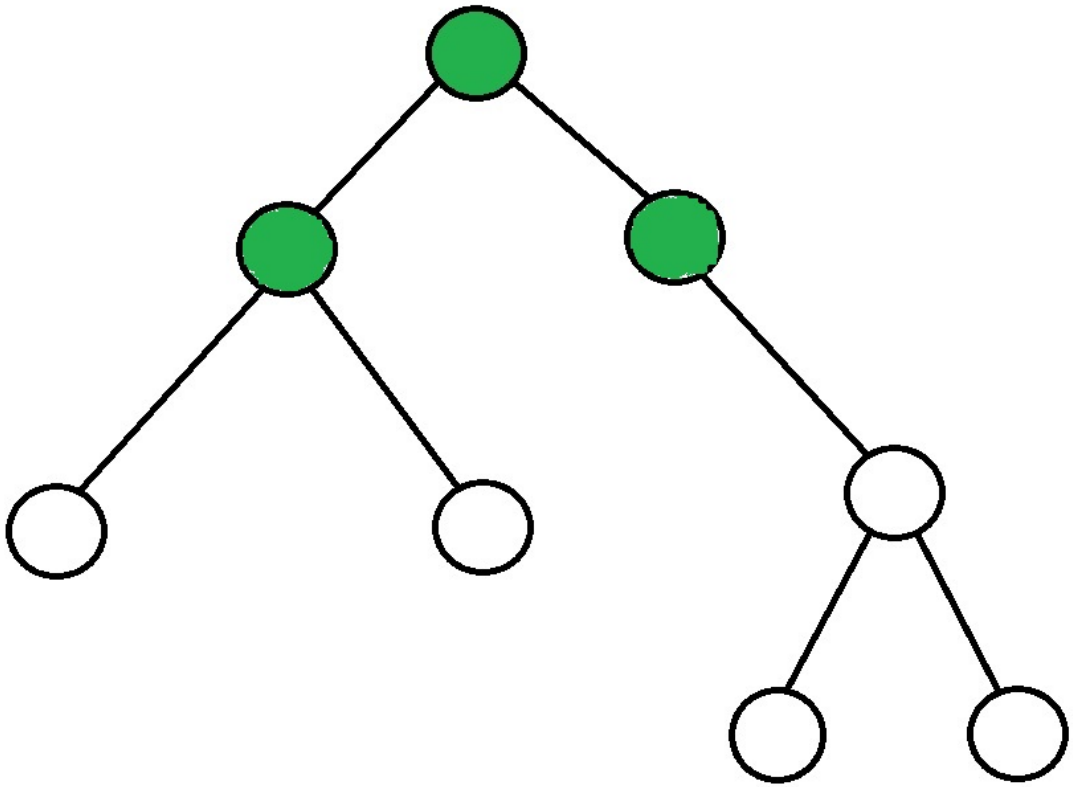
Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

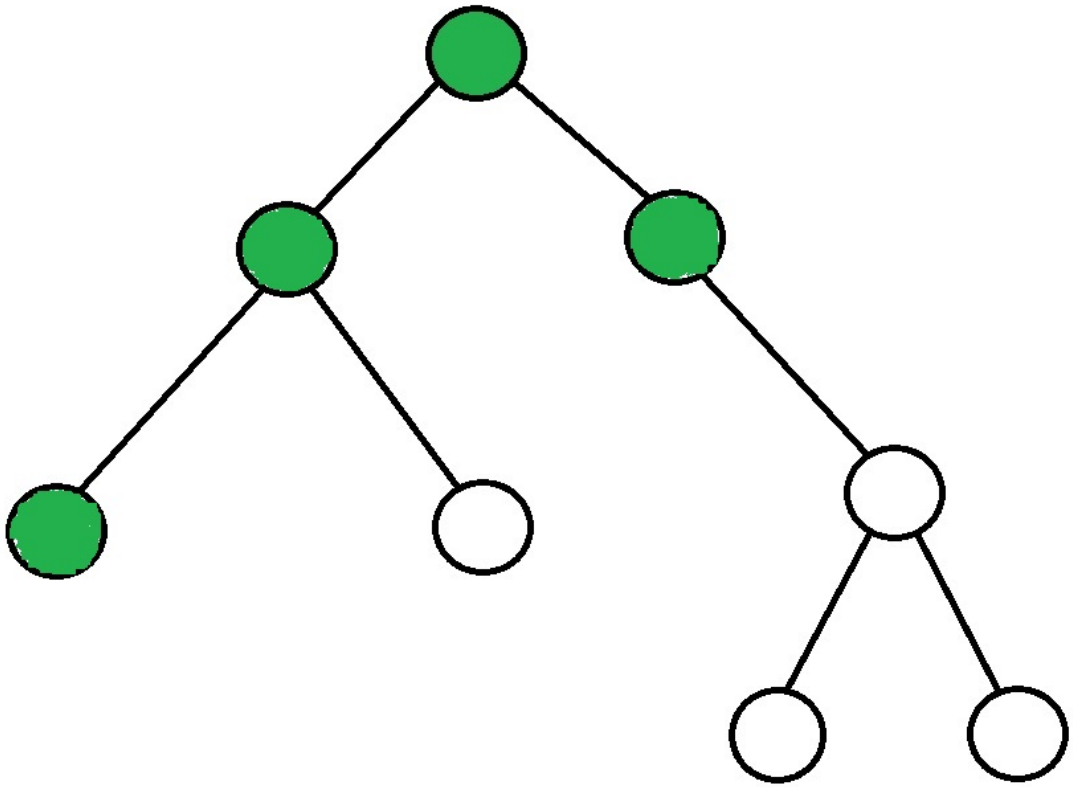
```
1  procedure BFS(G, start_v) is
2      let Q be a queue
3      label start_v as discovered
4      Q.enqueue(start_v)
5      while Q is not empty do
6          v := Q.dequeue()
7          if v is the goal then
8              return v
9          for all edges from v to w in G.adjacentEdges(v) do
10             if w is not labeled as discovered then
11                 label w as discovered
12                 w.parent := v
13                 Q.enqueue(w)
```

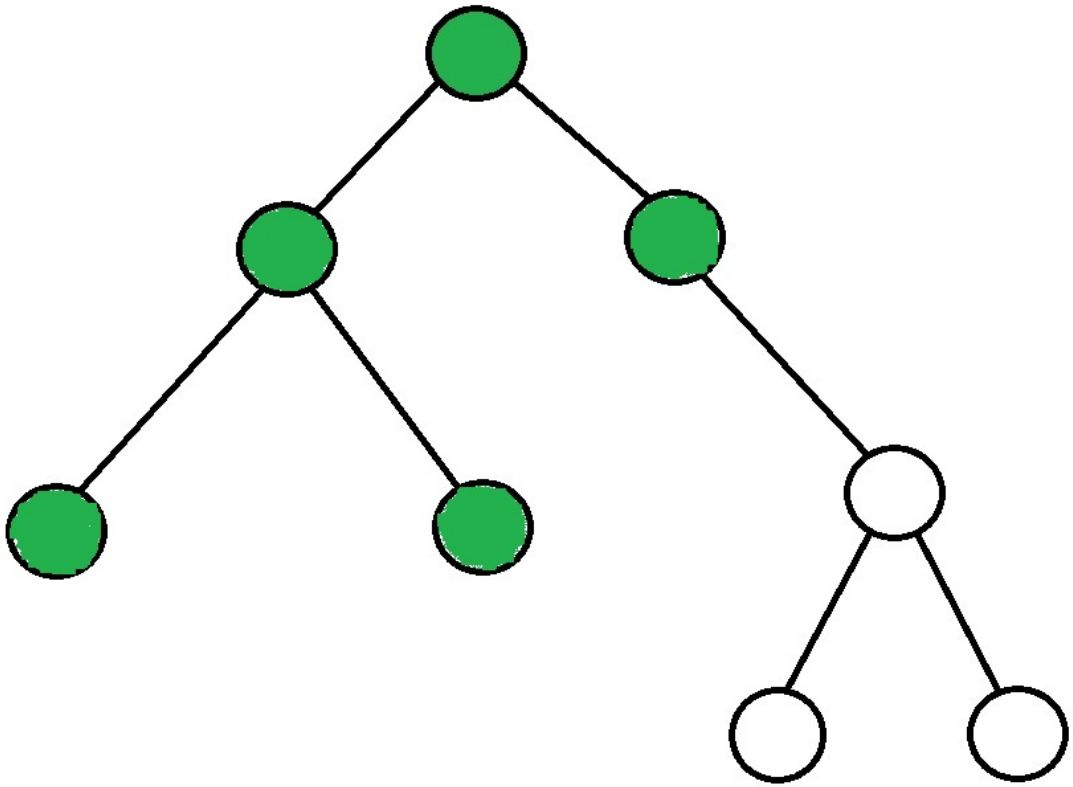
Illustration of BFS

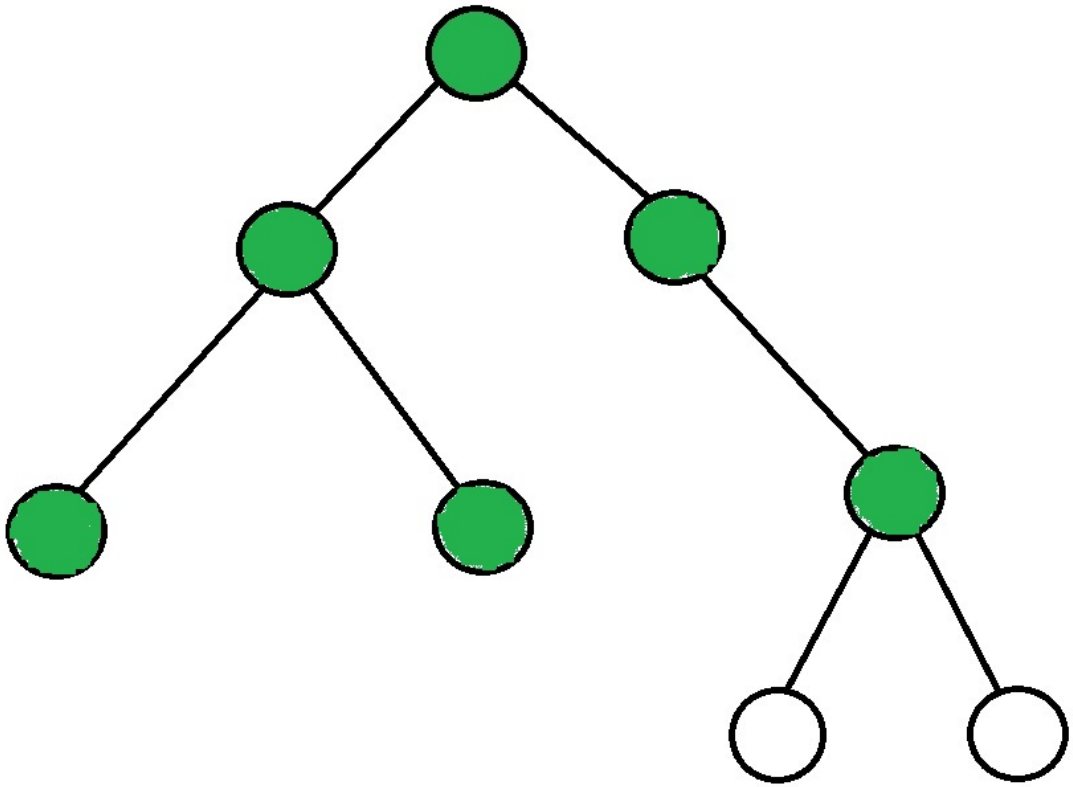


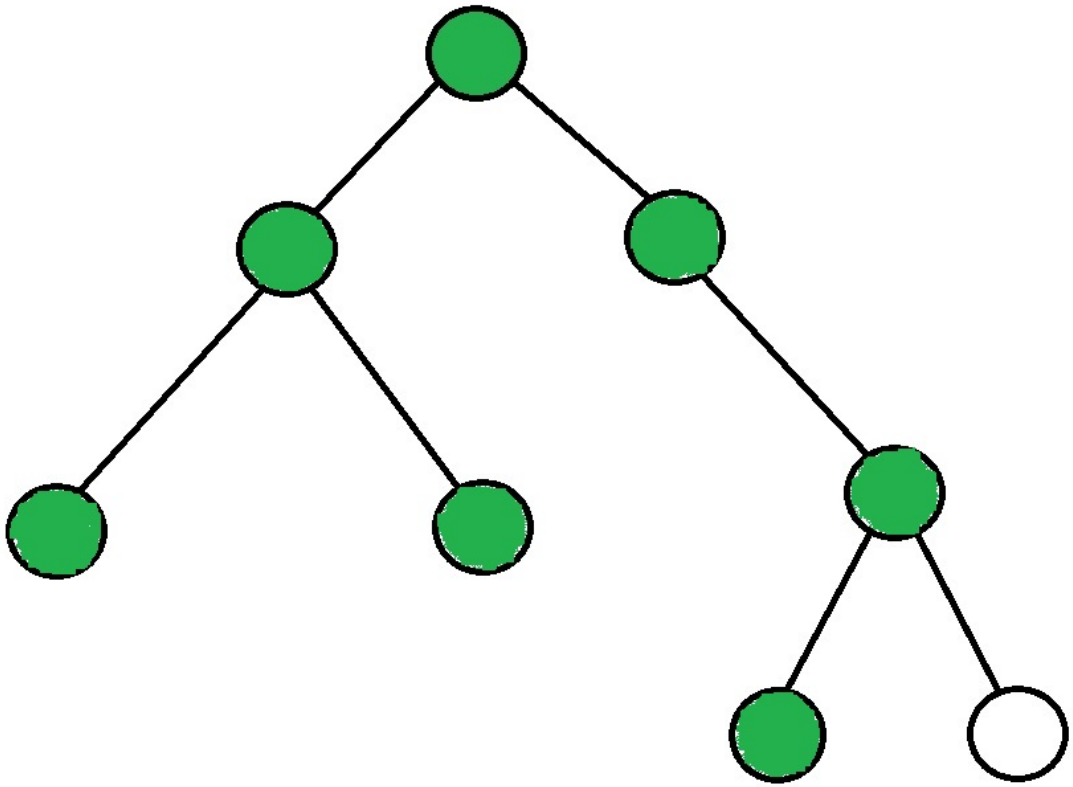


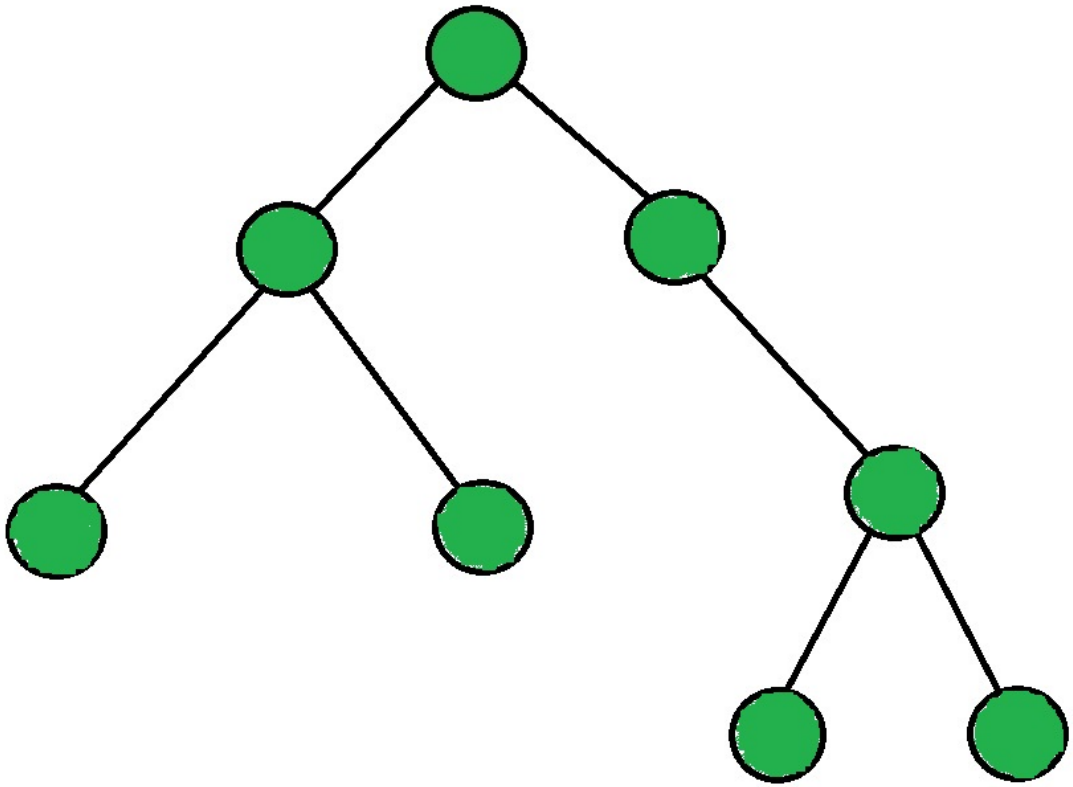












Complexity of BFS

- The time complexity of BFS is $O(|V| + |E|)$
- The space complexity of BFS is $O(|V|)$

Completeness: A search method is described as being complete if it is guaranteed to find a goal state if one exists in an infinite graph.

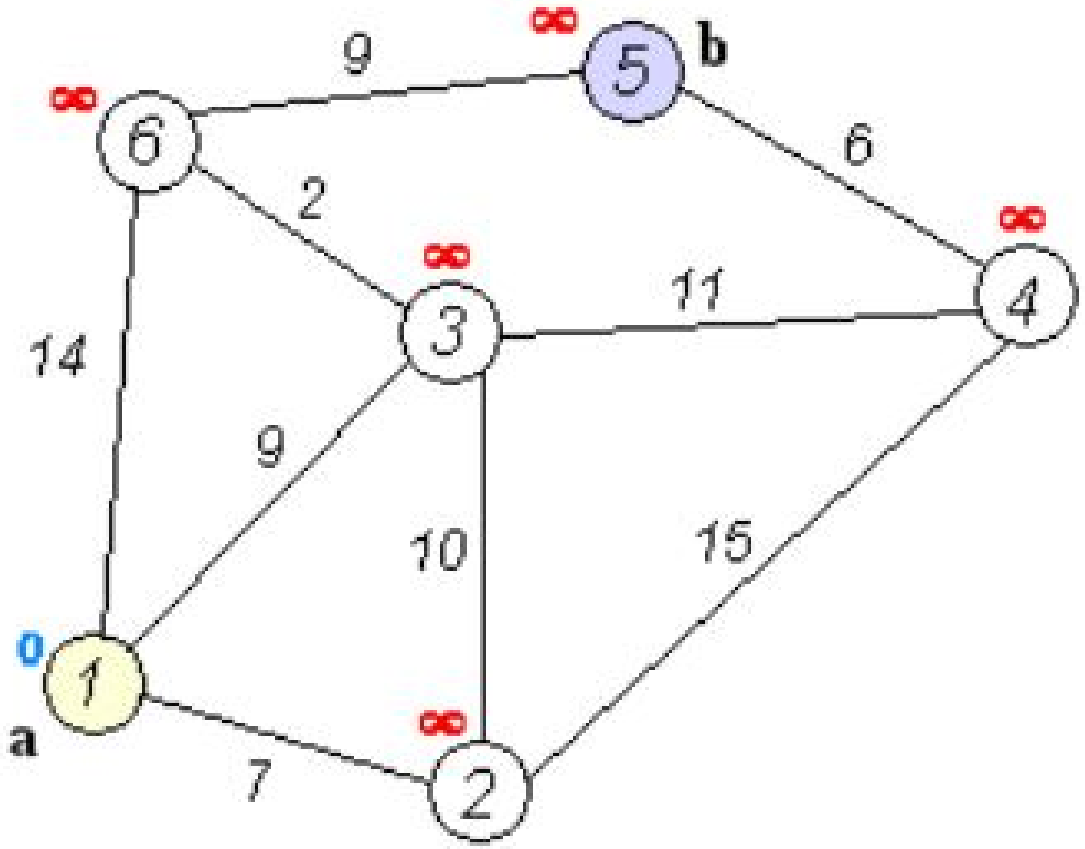
- Breadth-first search is complete, but depth-first search is not.

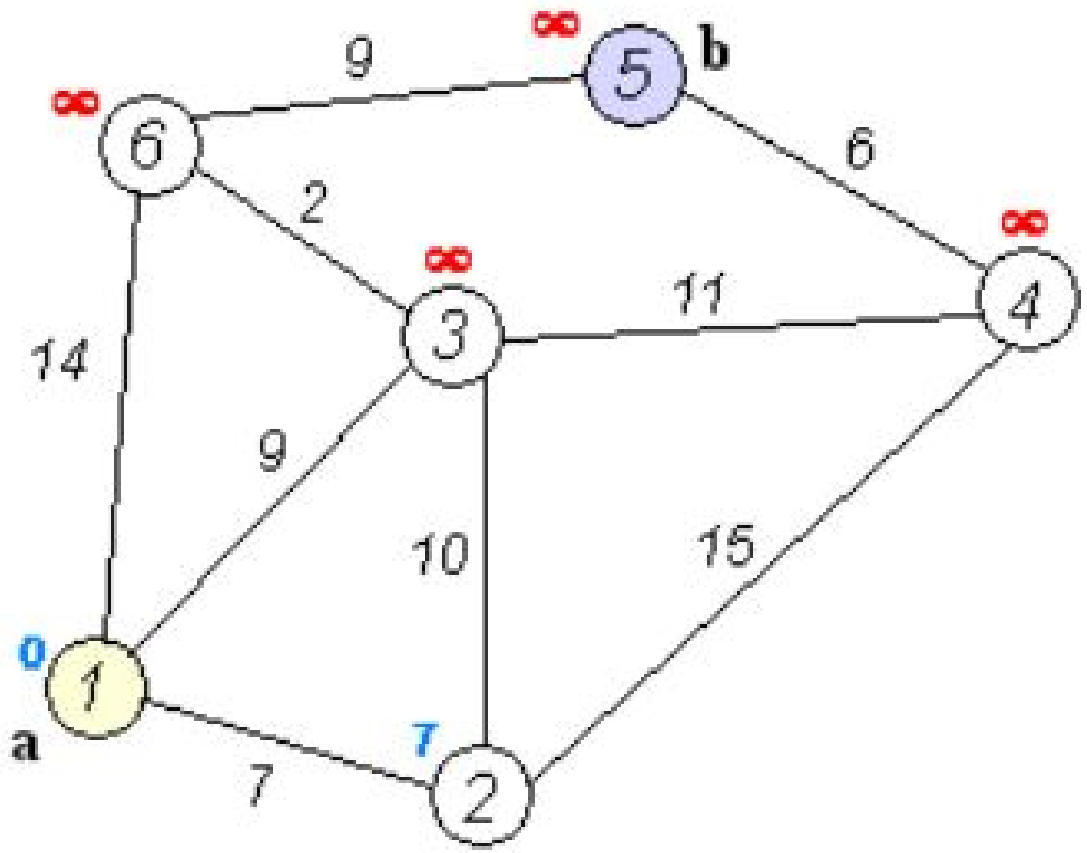
Dijkstra's Algorithm

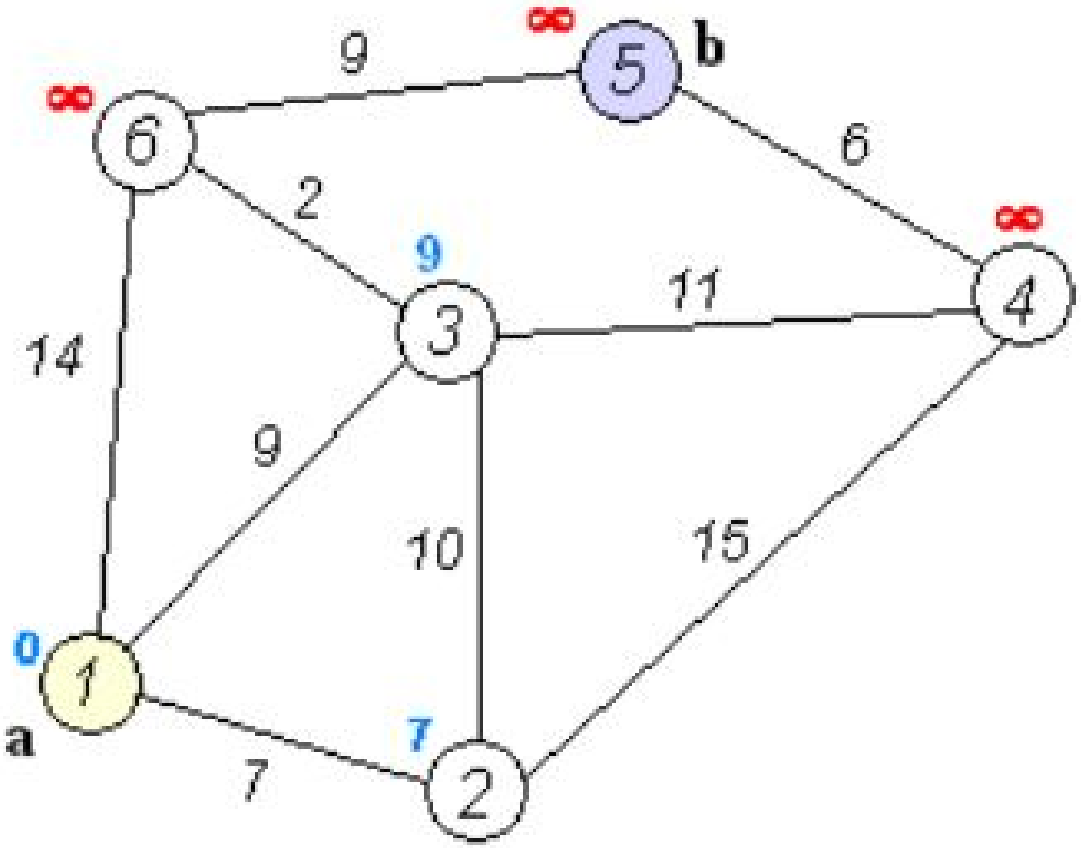
Dijkstra's algorithm finds the shortest path between a and b. It picks the unvisited vertex with the lowest distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller.

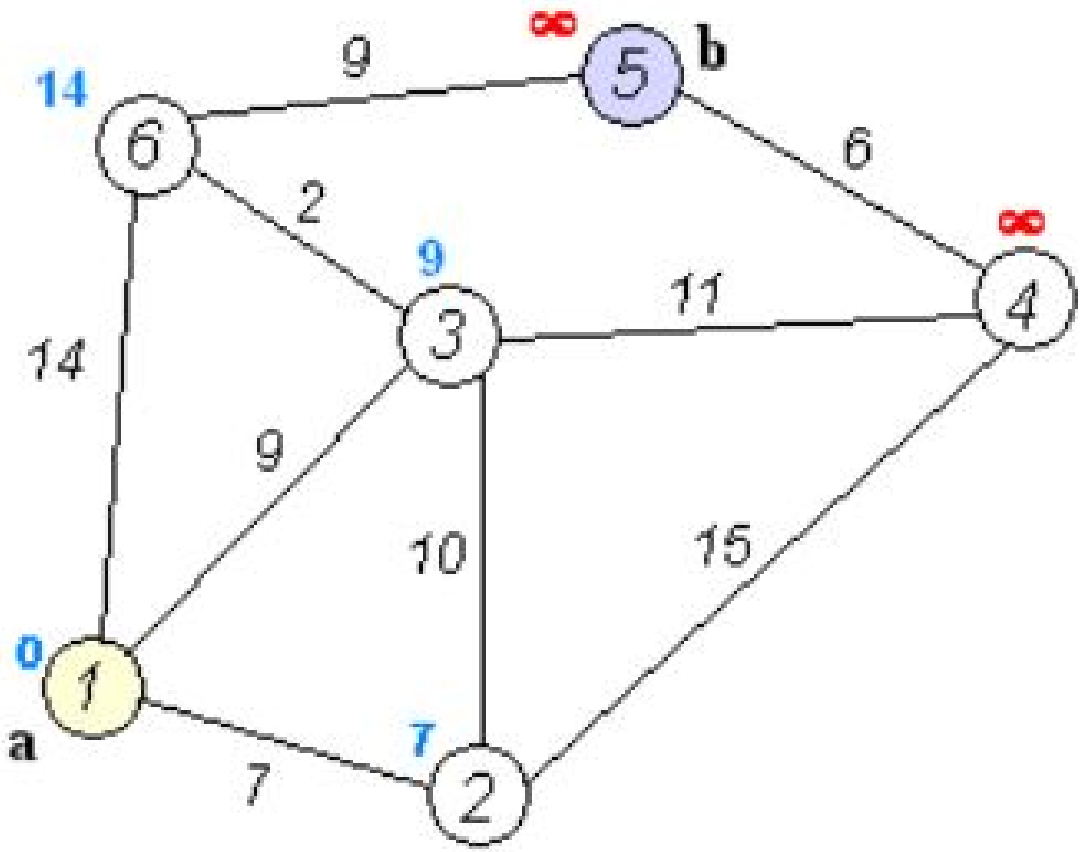
```
1  function Dijkstra(Graph, source):
2
3      create vertex set Q
4
5      for each vertex v in Graph:
6          dist[v] ← INFINITY
7          prev[v] ← UNDEFINED
8          add v to Q
10     dist[source] ← 0
11
12     while Q is not empty:
13         u ← vertex in Q with min dist[u]
14
15         remove u from Q
16
17         for each neighbor v of u:           // only v that are still in Q
18             alt ← dist[u] + length(u, v)
19             if alt < dist[v]:
20                 dist[v] ← alt
21                 prev[v] ← u
22
23     return dist[], prev[]
```

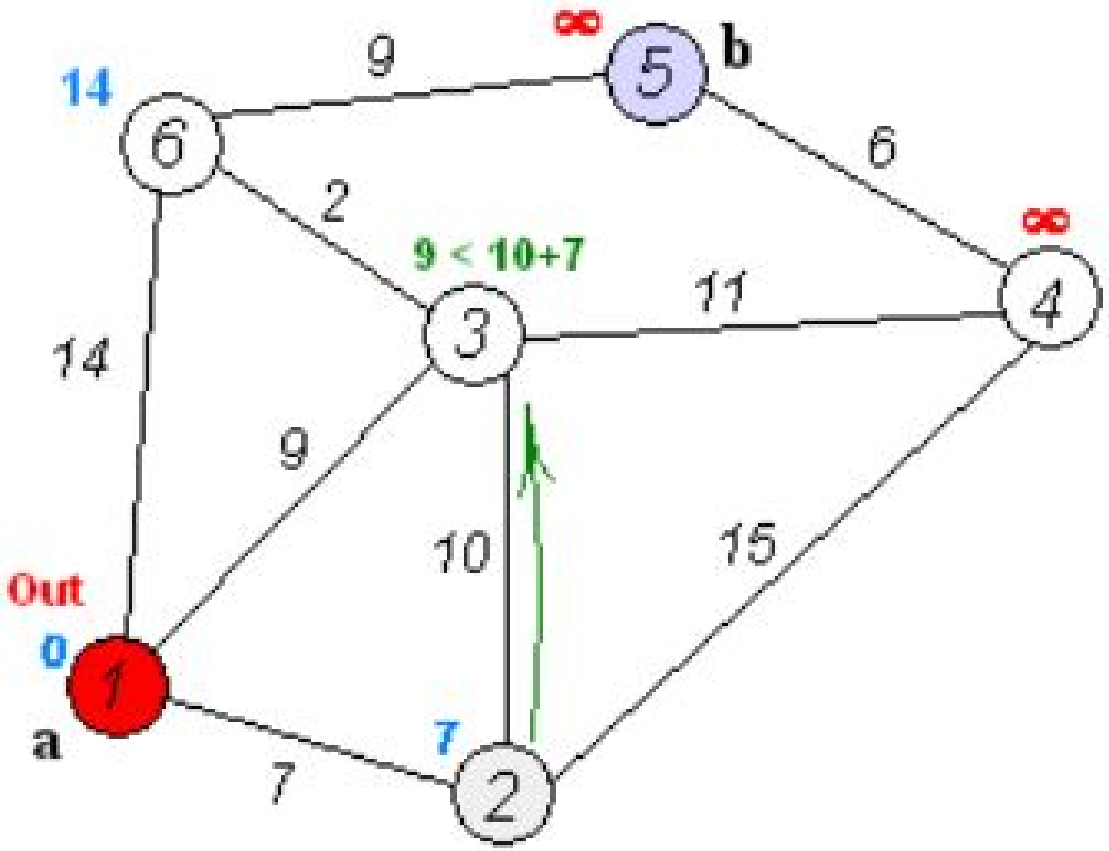
Illustration of Dijkstra's algorithm

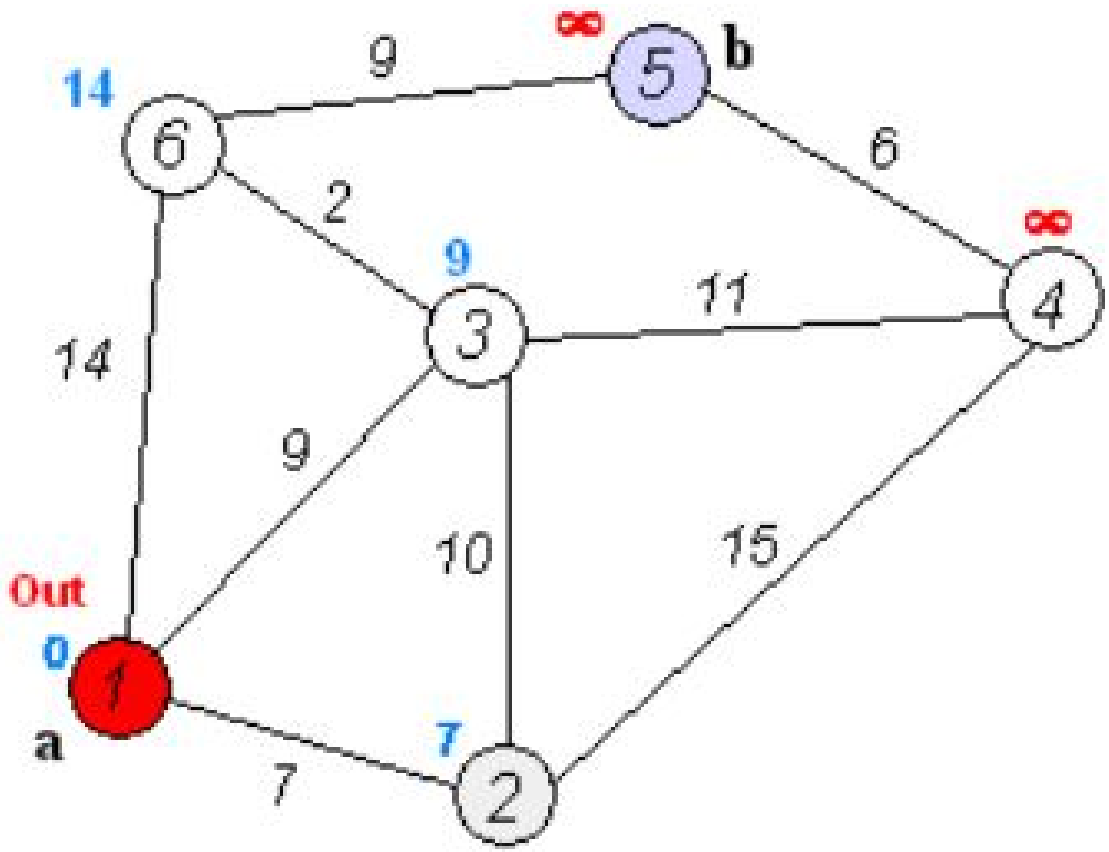


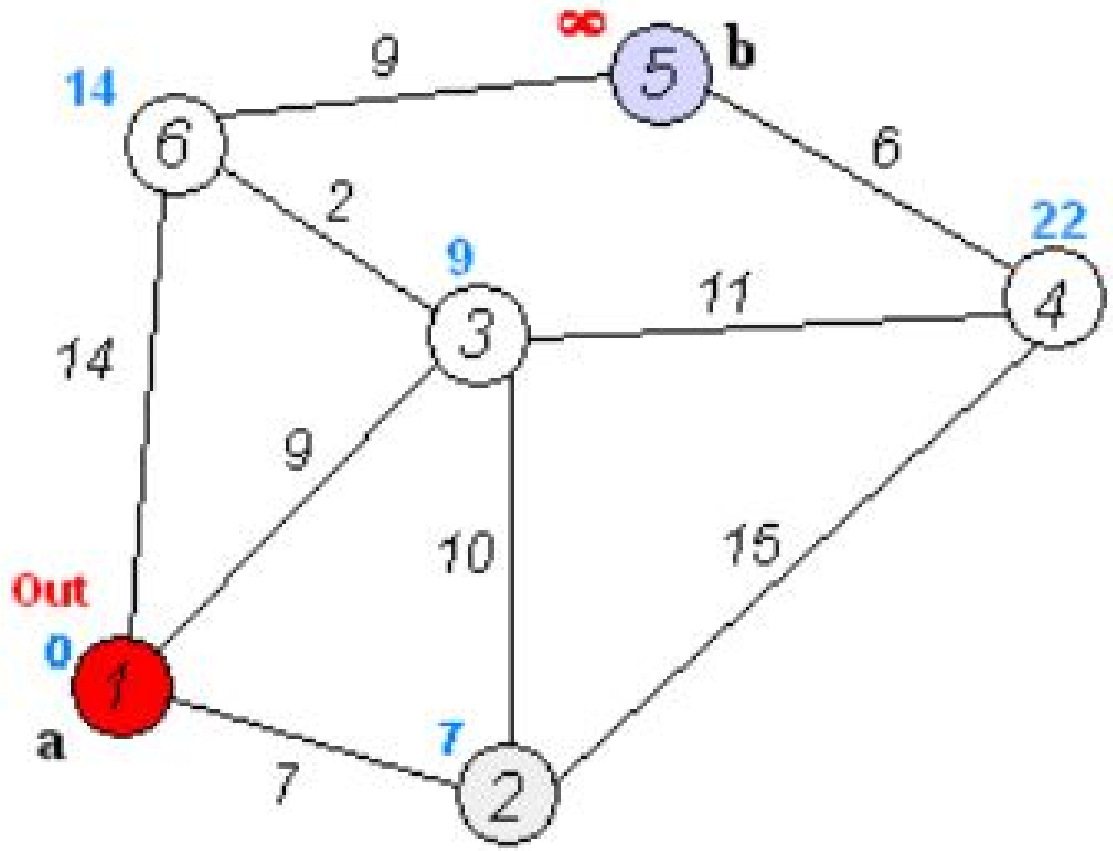

∞
9
5
b
∞
6
6
2
∞
∞
11
4
3
14
9
10
15
0
1
a
7
7
2

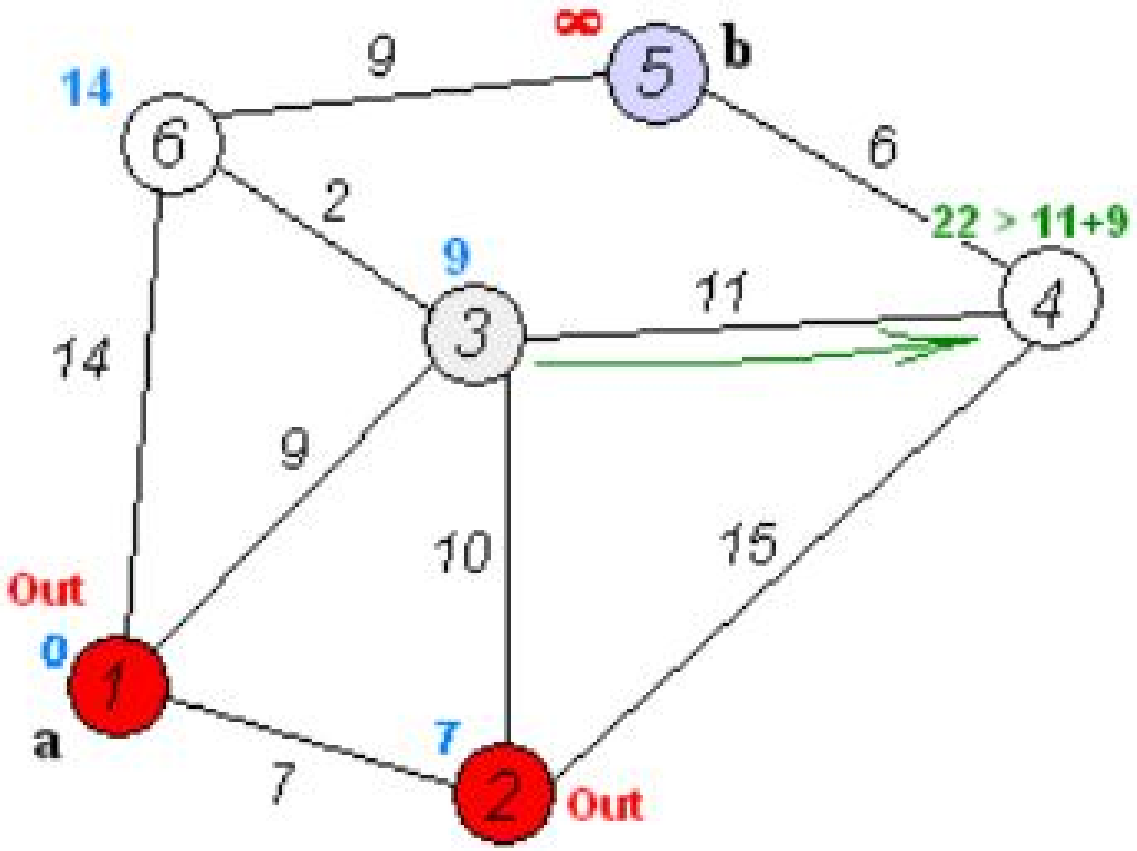

∞
9
∞
b
5
6
∞
6
2
9
11
4
3
14
9
10
15
0
1
a
7
7
2

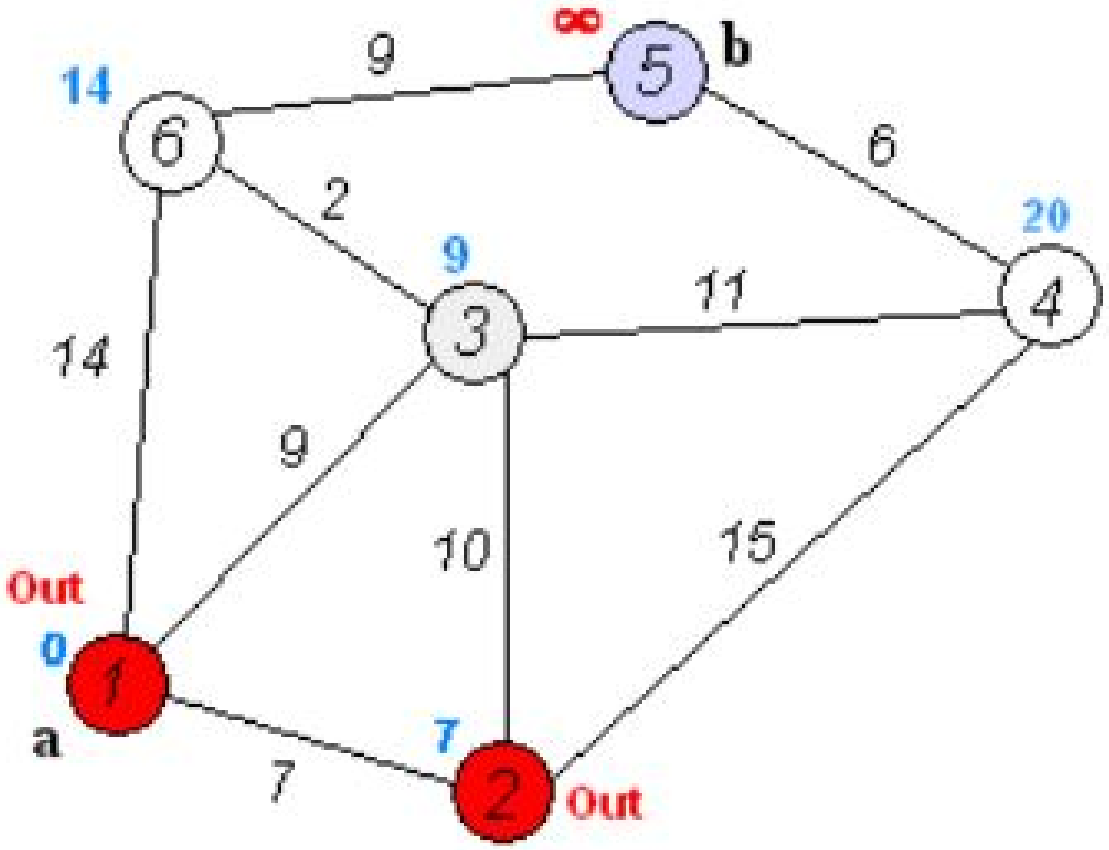

∞
5
b
9
14
6
6
2
∞
9
4
11
3
14
9
10
15
0
1
a
7
7
2

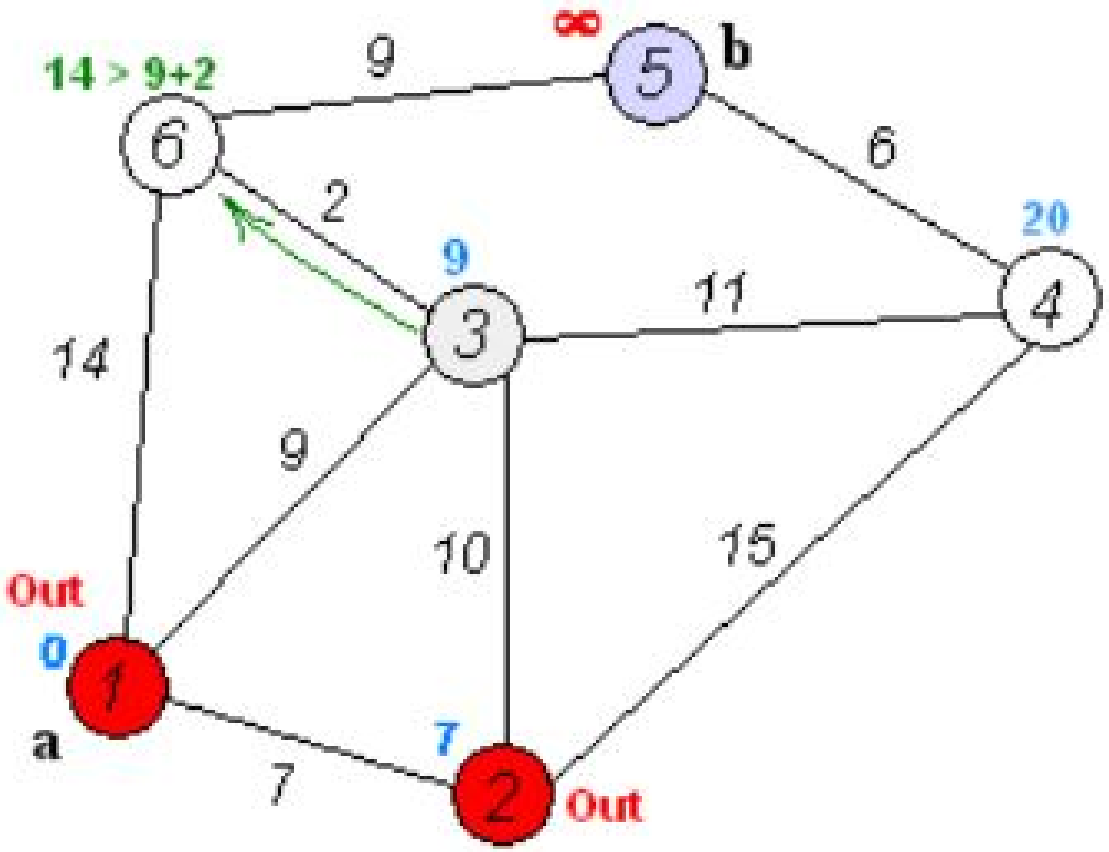

∞
9
5
b
14
6
6
2
∞
9 < 10+7
11
4
3
14
9
10
15
Out
0
1
a
7
7
2

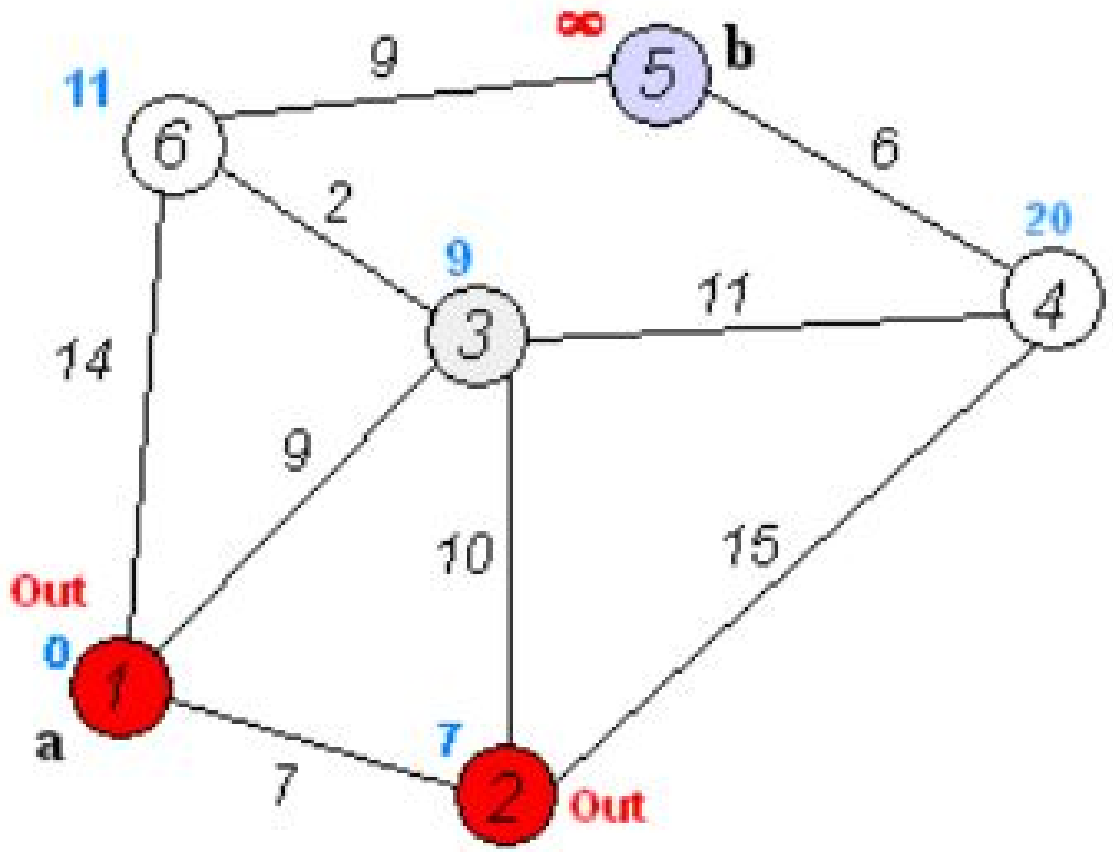

∞
5
b
9
14
6
6
2
∞
9
4
11
3
14
9
10
15
Out
0
1
a
7
7
2

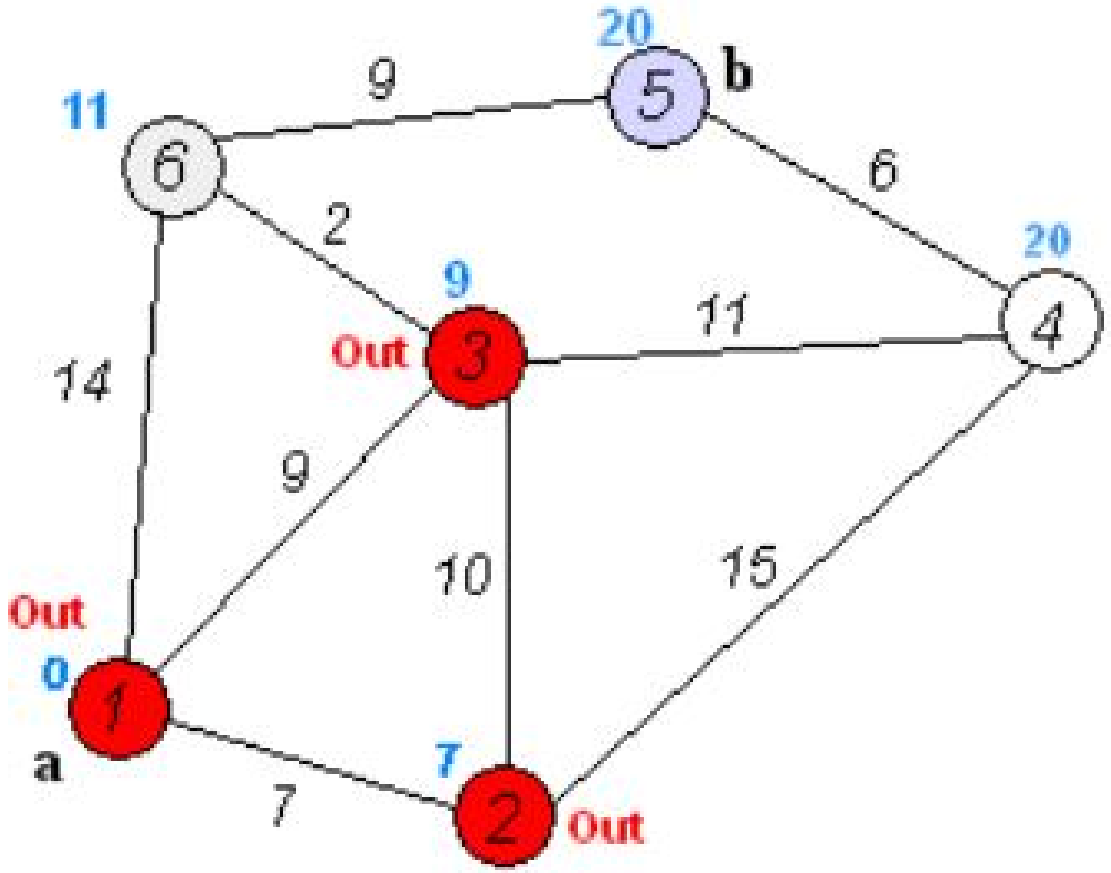

∞
5
b
9
14
6
6
2
22
9
11
4
3
14
9
10
15
Out
0
1
a
7
7
2

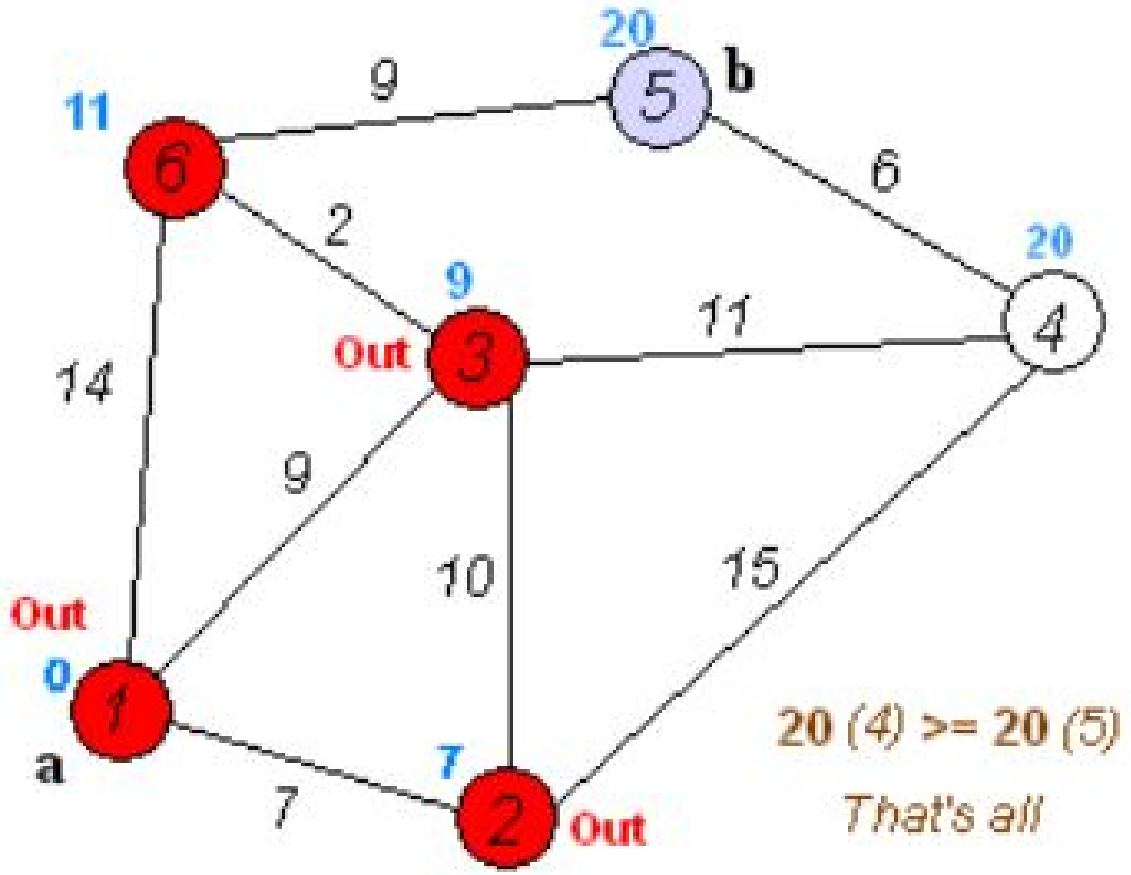

∞
b
5
9
14
6
6
2
22 > 11+9
9
11
4
3
14
9
10
15
Out
0
1
a
7
7
2
Out


∞
5
b
9
14
6
6
2
20
9
11
4
3
14
9
10
15
Out
0
1
a
7
7
2
Out


∞
b
5
9
14 > 9+2
6
6
2
20
9
11
4
3
14
9
10
15
Out
0
1
a
7
7
2
Out


∞
b
5
9
11
6
6
2
20
9
11
4
3
14
9
10
15
Out
0
1
a
7
7
2
Out


20
b
5
9
11
6
6
2
20
9
Out
3
11
4
14
9
10
15
Out
0
1
a
7
7
2
Out


20
b
5
9
11
6
6
2
20
4
9
Out
3
11
14
9
10
15
Out
0
1
a
7
7
2
Out
20 (4) >= 20 (5)
That's all

GIF of Dijkstra's algorithm for Robot Motion:
<CLICK>

Analysis of Dijkstra's algorithm

- Runs in $O(V^2)$ time using an array, and $O((|V| + |E|) \log |V|)$ time using a min-heap.
- Space complexity is $O(|V|)$

A^* Search

At each iteration of its main loop, A^* needs to determine which of its paths to extend. It does so based on the cost of the path (stored in g) and an estimate of the cost required to extend the path all the way to the goal (computed by a heuristic function h).

Specifically, A^* selects the path that minimizes:

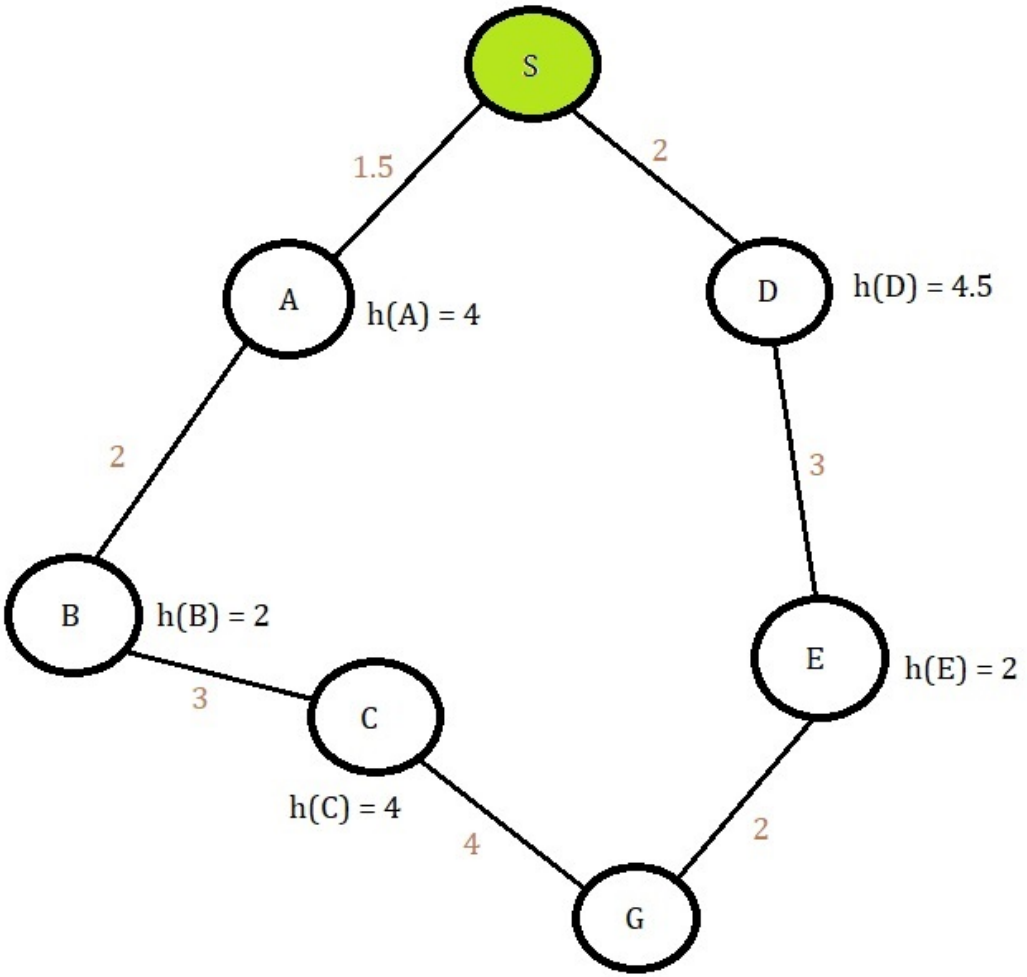
$$f(n) = g(n) + h(n)$$

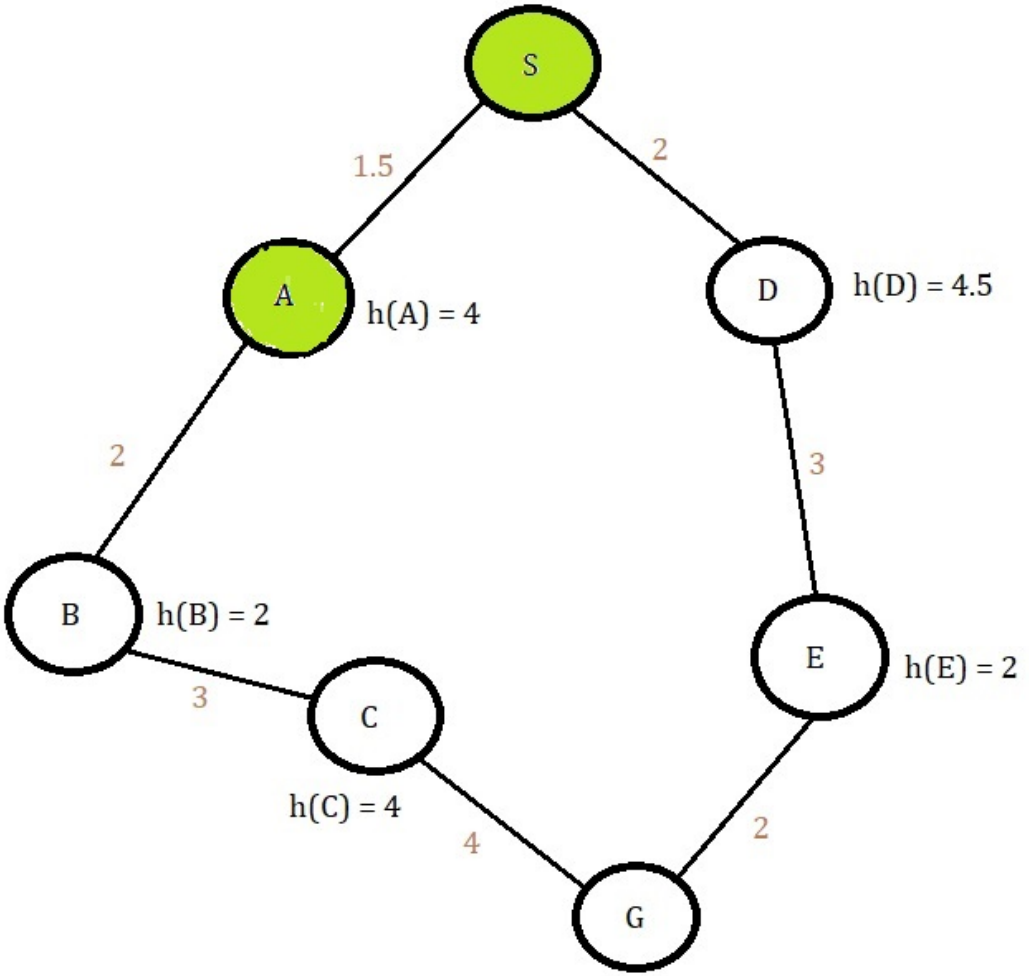
The heuristic function is problem-specific. If the heuristic function is *admissible* - meaning that it never overestimates the actual cost to get to the goal, then A^* is guaranteed to return a least-cost path from start to goal.

Path Planning

```
function A_Star(start, goal, h)
    // The set of discovered nodes that may need to be (re-)expanded.
    // Initially, only the start node is known.
    // This is usually implemented as a min-heap or priority queue rather than a hash-set.
    openSet := {start}

    // For node n, cameFrom[n] is the node immediately preceding it on the cheapest path from start
    // to n currently known.
    cameFrom := an empty map

    // For node n, gScore[n] is the cost of the cheapest path from start to n currently known.
    gScore := map with default value of Infinity
    gScore[start] := 0

    // For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our current best guess as to
    // how short a path from start to finish can be if it goes through n.
    fScore := map with default value of Infinity
    fScore[start] := h(start)

    while openSet is not empty
        // This operation can occur in O(1) time if openSet is a min-heap or a priority queue
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        for each neighbor of current
            // d(current,neighbor) is the weight of the edge from current to neighbor
            // tentative_gScore is the distance from start to the neighbor through current
            tentative_gScore := gScore[current] + d(current, neighbor)
            if tentative_gScore < gScore[neighbor]
                // This path to neighbor is better than any previous one. Record it!
                cameFrom[neighbor] := current
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := gScore[neighbor] + h(neighbor)
                if neighbor not in openSet
                    openSet.add(neighbor)

    // Open set is empty but goal was never reached
    return failure
```

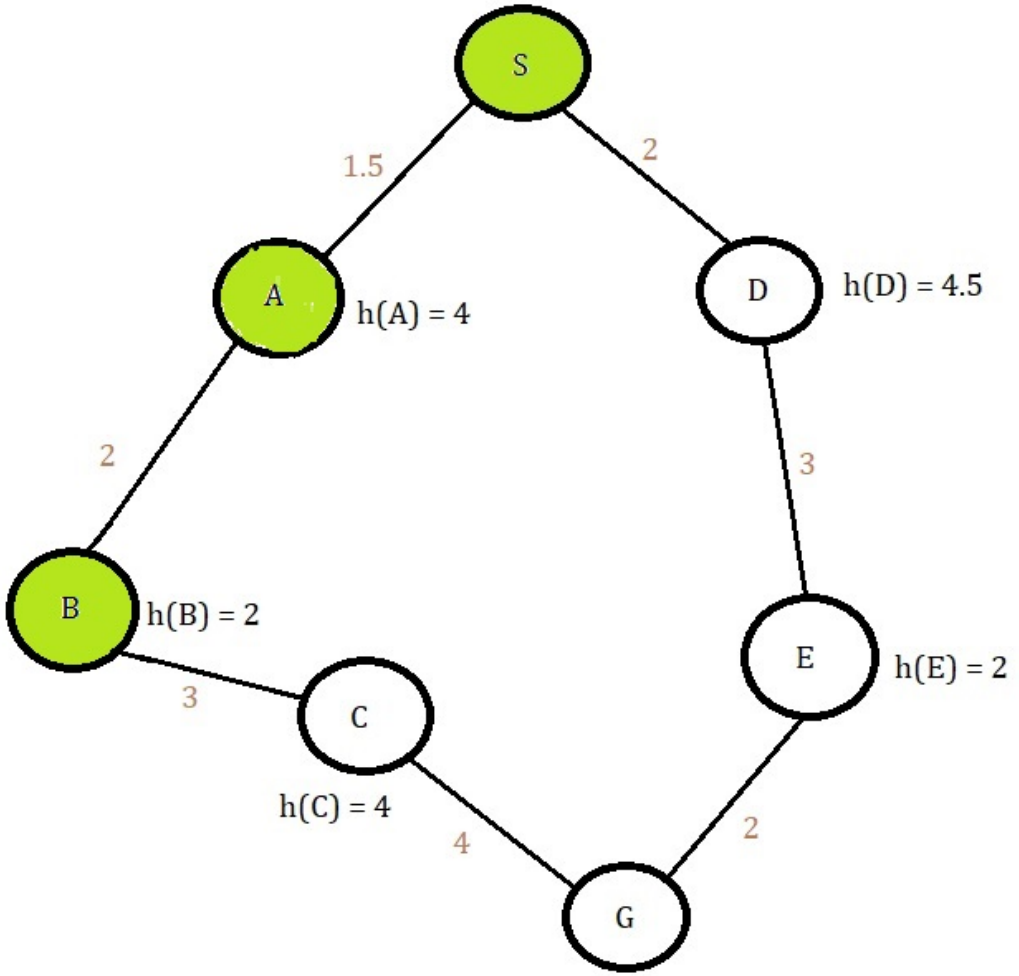
Illustration of A^* Search





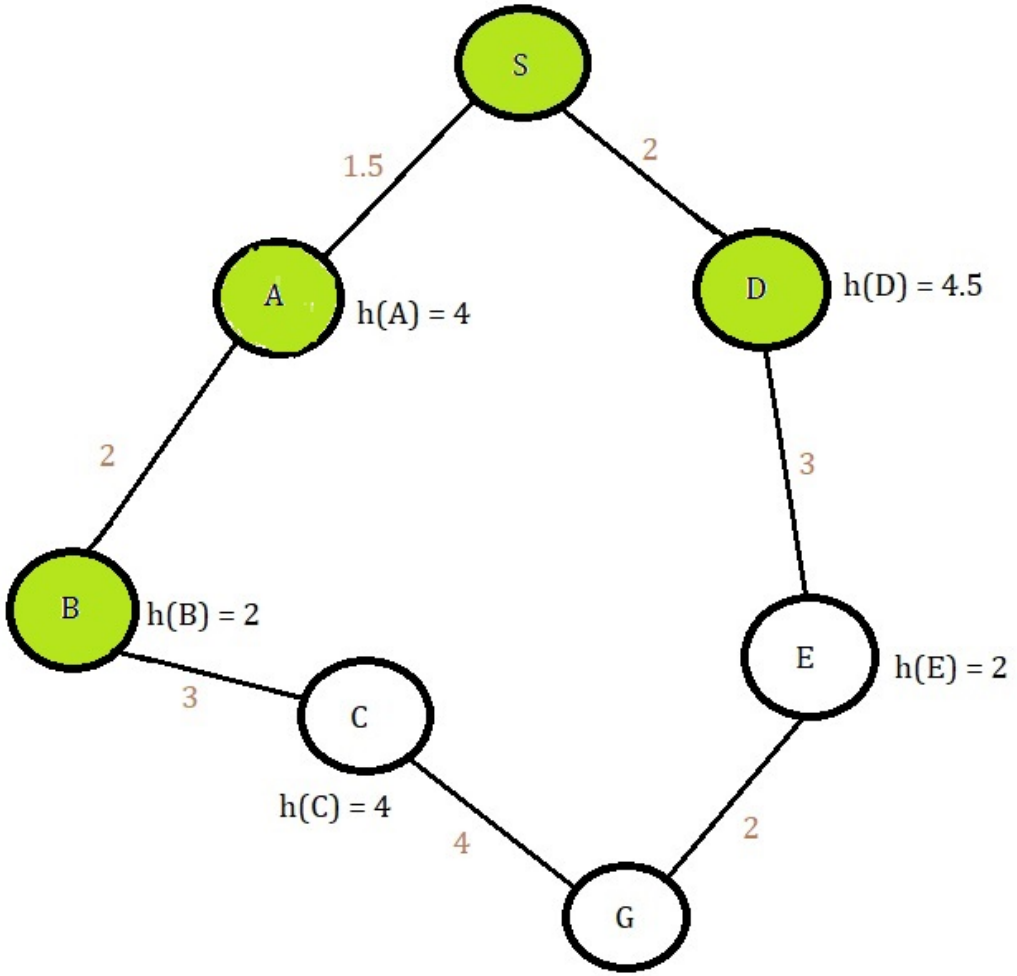
f(B) = 3.5 + 2 = 5.5

f(D) = 2 + 4.5 = 6.5



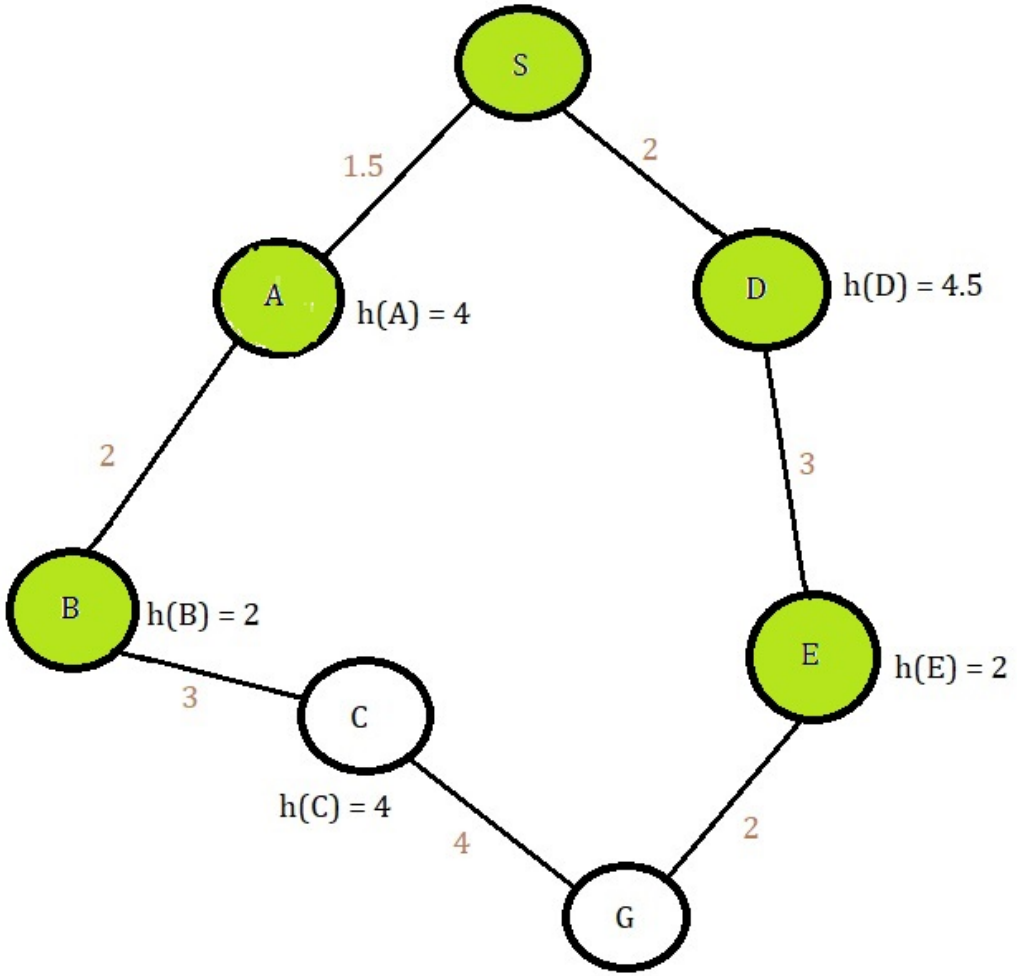
f(C) = 6.5 + 4 = 10.5

f(D) = 2 + 4.5 = 6.5



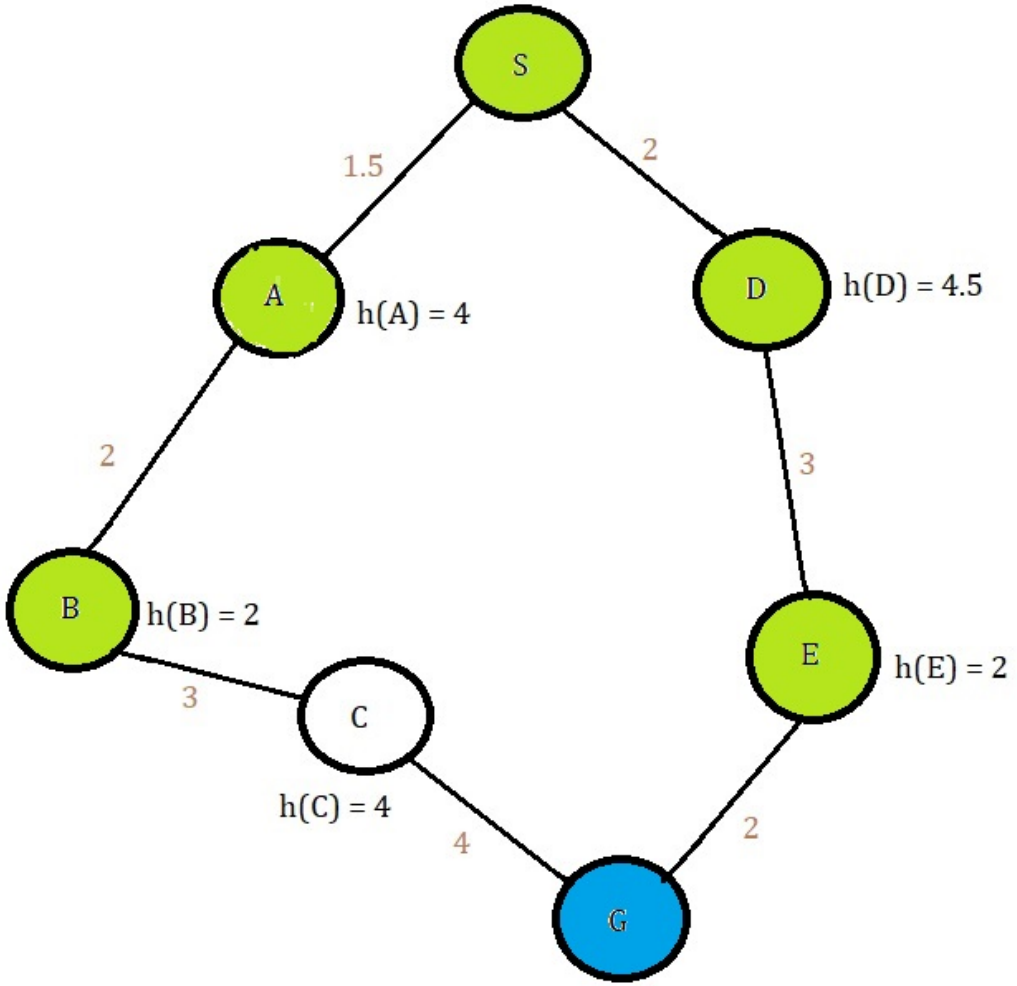
f(C) = 6.5 + 4 = 10.5

f(E) = 5 + 2 = 7



f(C) = 6.5 + 4 = 10.5

f(G) = 7 + 0 = 7


S
1.5
2
A
h(A) = 4
D
h(D) = 4.5
2
3
B
h(B) = 2
E
h(E) = 2
3
C
h(C) = 4
4
2
G

GIF of A^* Search for Robot Motion:

(h(x) used here is the straight-line distance to the target point)
<CLICK>

Analysis of A^* Search

- The time complexity of A^* depends on the heuristic.
- Assuming that the goal exists and is reachable from the start state, in the worst case of an unbounded search space, the number of nodes expanded is exponential in the depth of the shortest path d: $O(b^d)$, where b is the branching factor (the average number of successors per state).

  The heuristic function has a major effect on the practical performance of A^* search, since a good heuristic allows A^* to prune away many of the b^d nodes that an un-informed search would expand.