EECE 5550
Mobile Robotics

Lecture 13: Planning

Derya Aksaray

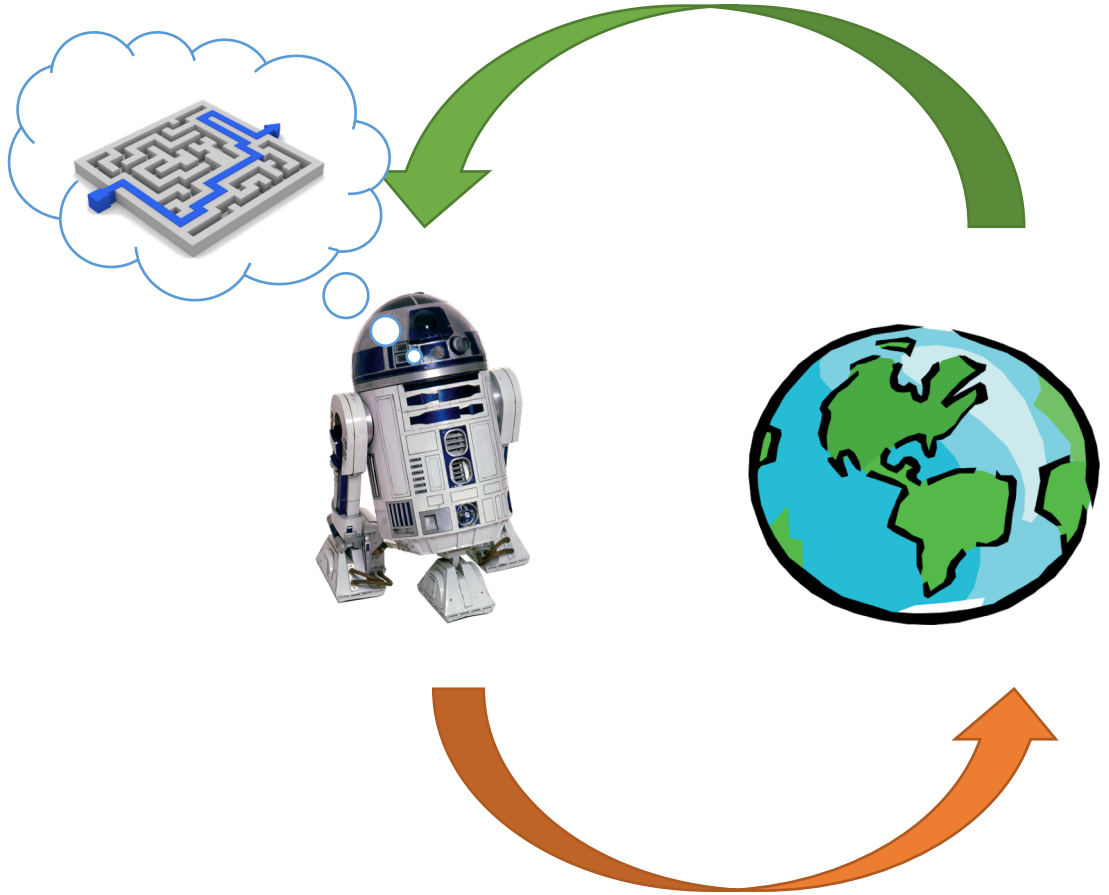
Assistant Professor

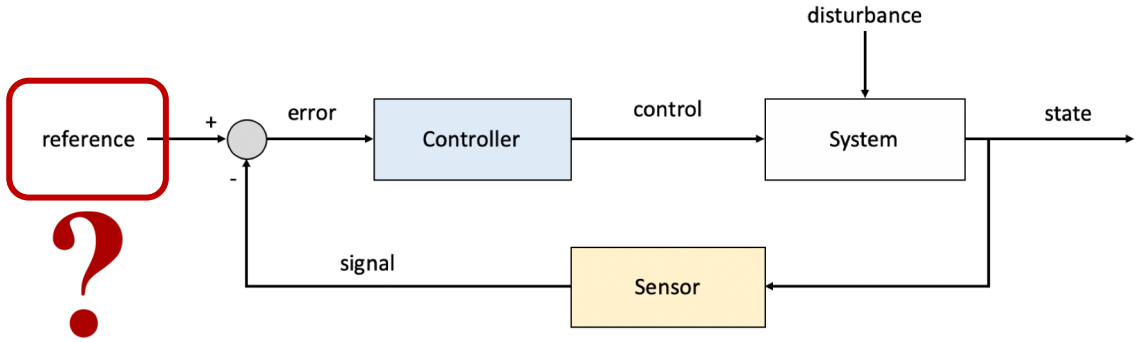
Department of Electrical and Computer Engineering

Northeastern University

The Central Dogma of Robotics: Sense → Think → Act

- Sense: Process sensor data to construct a model of the world
- Think: Construct a plan to move from the current state to the goal state
- Act: Control actuators to execute plan





The story so far

- ✓ Mathematical foundations
- ✓ Robot modeling
- ✓ Sense (Perception)

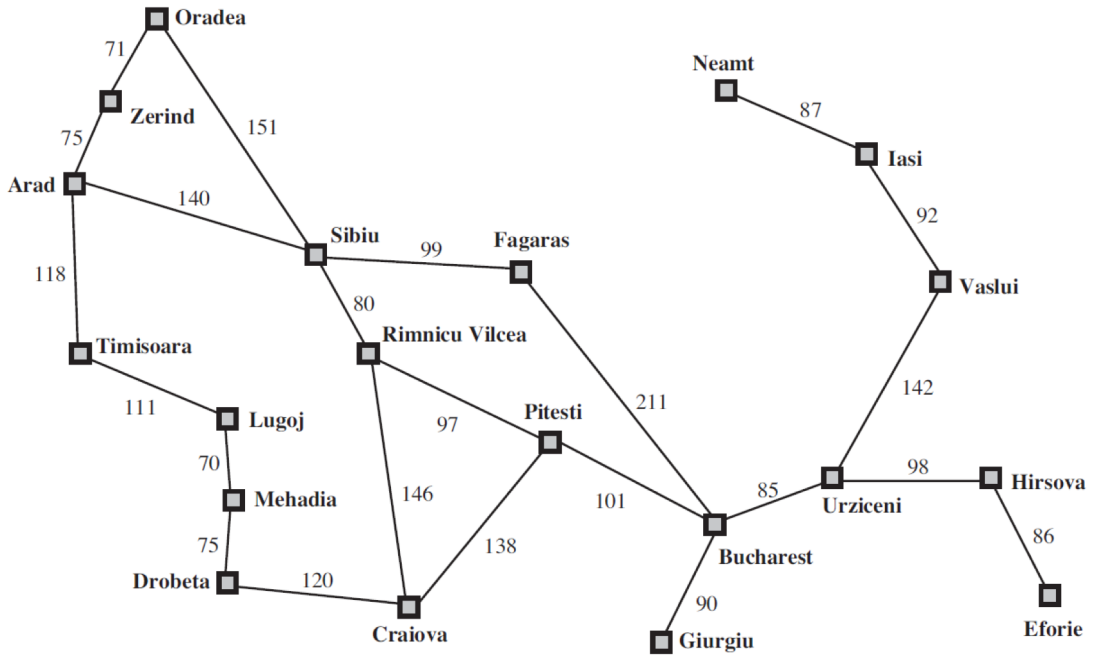
This week: Think (plan)

Tentative Course Outline

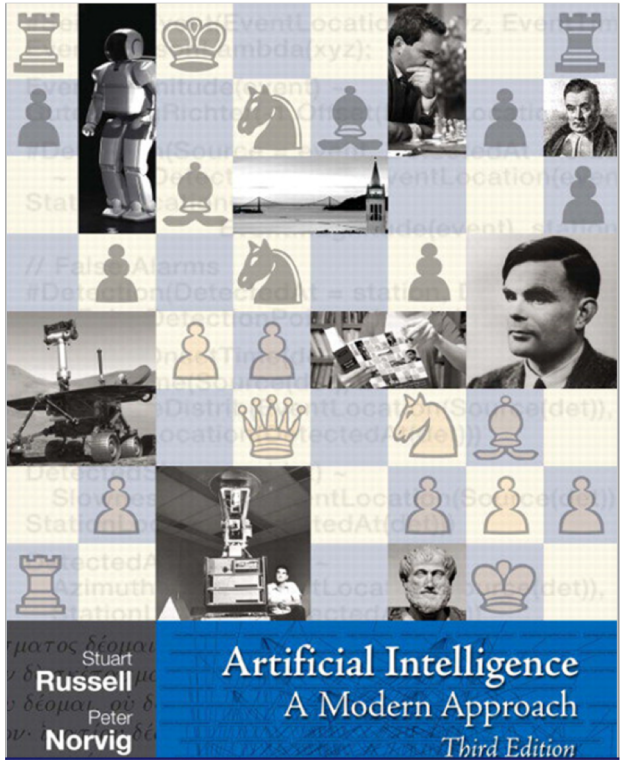
Week	Date	Topic	Assignment
1	Sep 4	Course intro & linear algebra refresher Coordinate systems and geometry, Intro to Manifolds	
2	Sep 11	Robot modeling and sensing Fundamentals of computer vision	Assignment 1
3	Sep 18	Math foundations: Review of Probability Probabilistic Robotics	
4	Sep 25	Bayes filter implementations Mapping	Assignment 2
5	Oct 2	Localization	
6	Oct 9	Simultaneous localization and mapping (SLAM) Math foundations: Optimization	Assignment 3
7	Oct 16	Introduction to planning Robot motion planning	
8	Oct 23	Planning under uncertainty MDPs, POMDPs	Assignment 4
9	Oct 30	Robotic exploration Feedback control I	
10	Nov 6	Feedback control II Stability analysis	Assignment 5
11	Nov 13	Optimal control and model predictive control Advanced topics and recent trends	
12	Nov 20	Reading Week	
13	Nov 27	Thanksgiving	
14	Dec 4	Project presentations	

Plan of the day

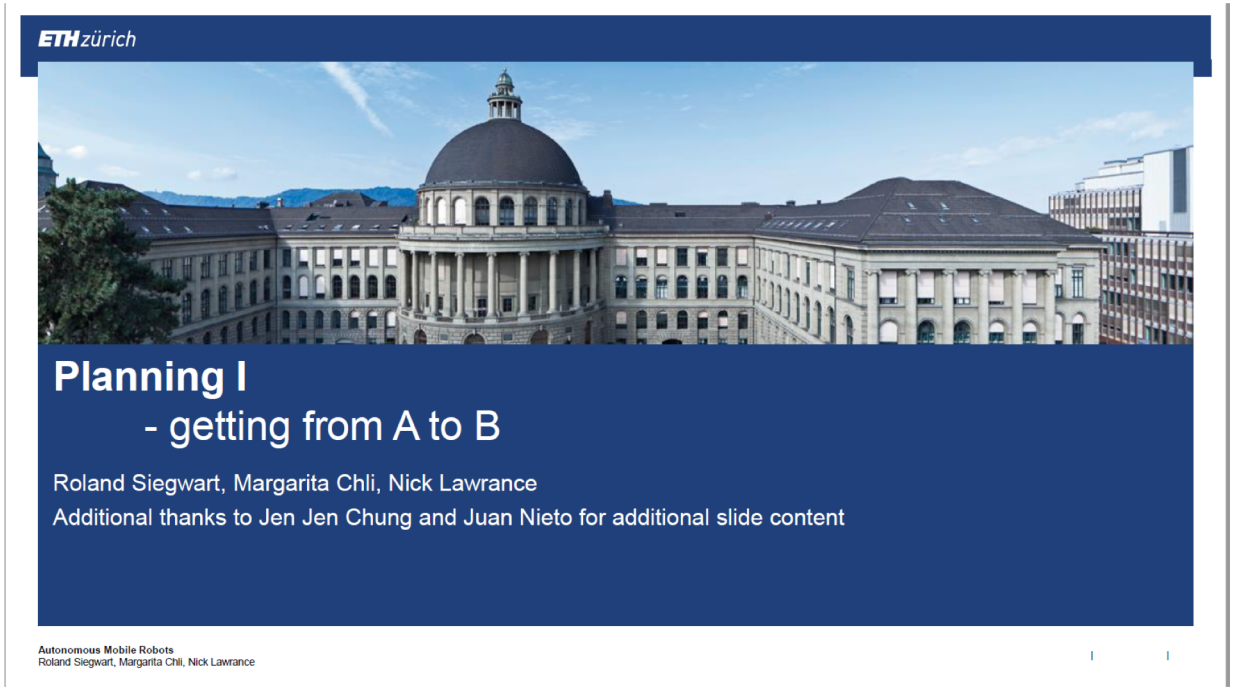
- Introduction to planning
- Formulation of a planning problem
- Planning as graph search
- Four canonical graph search algorithms
  - Breadth-first search
  - Depth-first search
  - Searching for *shortest paths*: Dijkstra's algorithm
  - Informed search: A* search



References



Chapter 3 of “Artificial Intelligence: A Modern Approach”



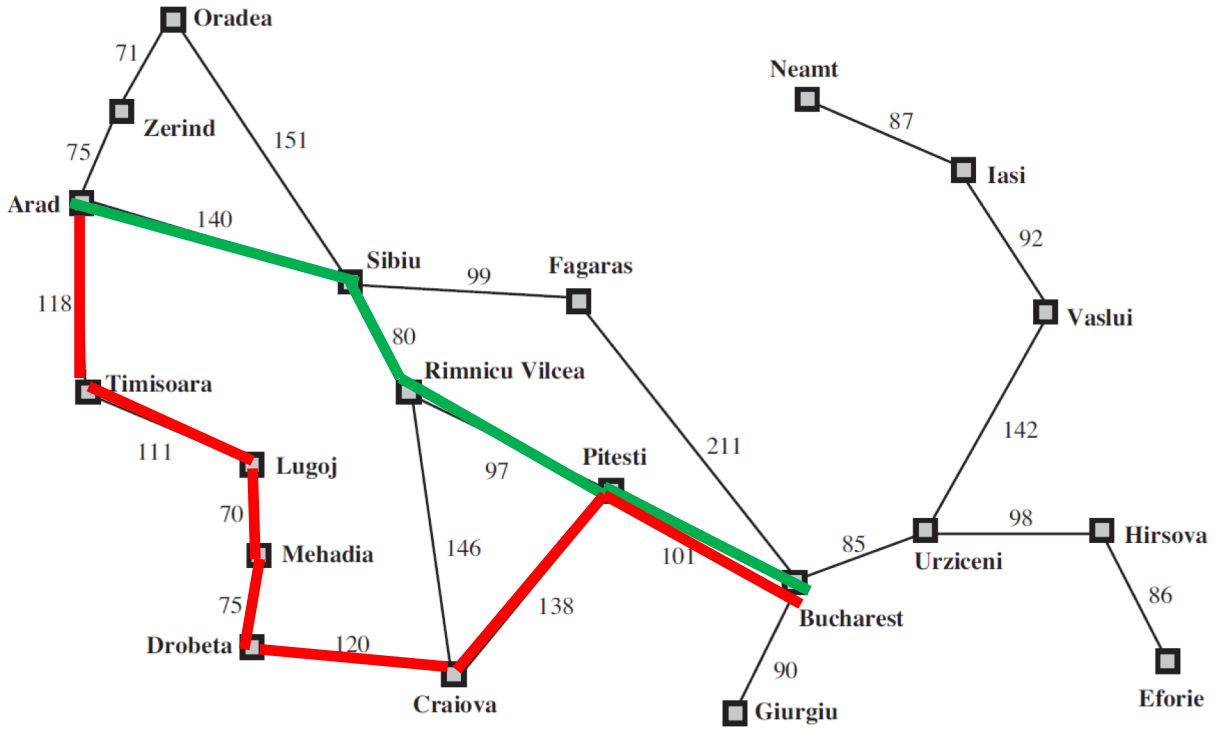
Lecture “Planning I” from ETH Zurich’s Autonomous Mobile Robots course

What is “planning”?

In the context of robot motion planning:

Determine a sequence of actions to drive a robot from a a known *initial state* to another *goal state*.

Main question: Which route to take?



Planning - Control

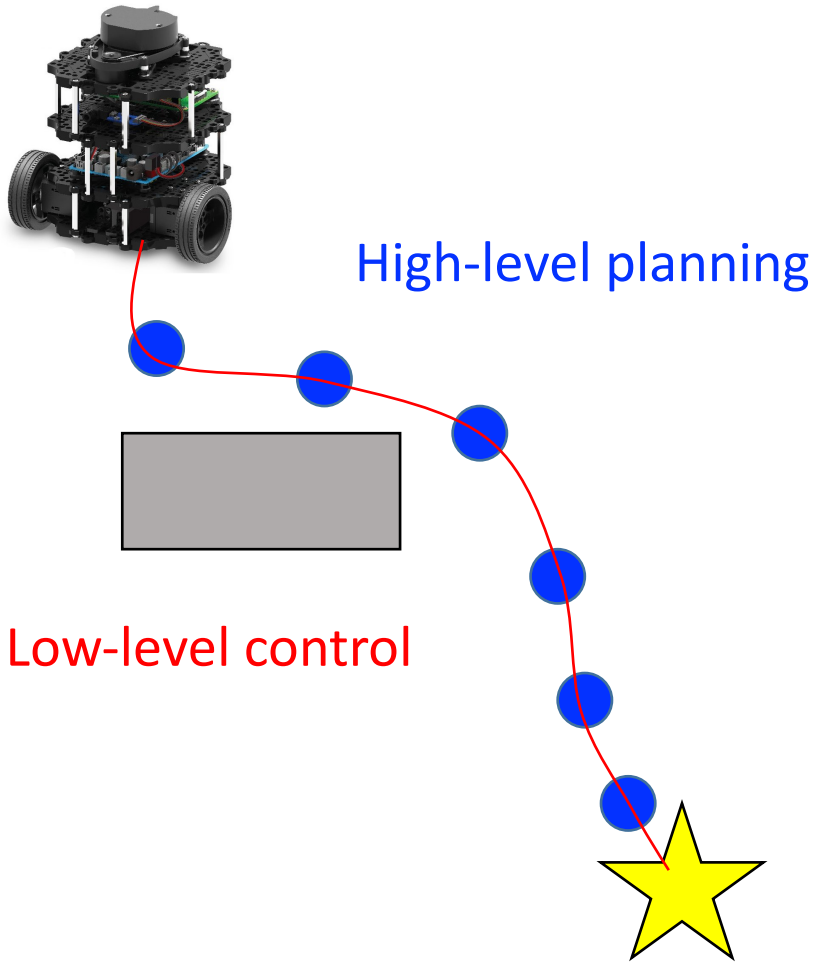
While they have similar objectives like reaching a desired goal state, they have different perspectives:

Control:

- Generally concerned with reaching and maintaining a desired state in some kind of robust way
- Often **feedback-based**
- Success measured in terms of stability, robustness, ability to reject disturbance

Planning

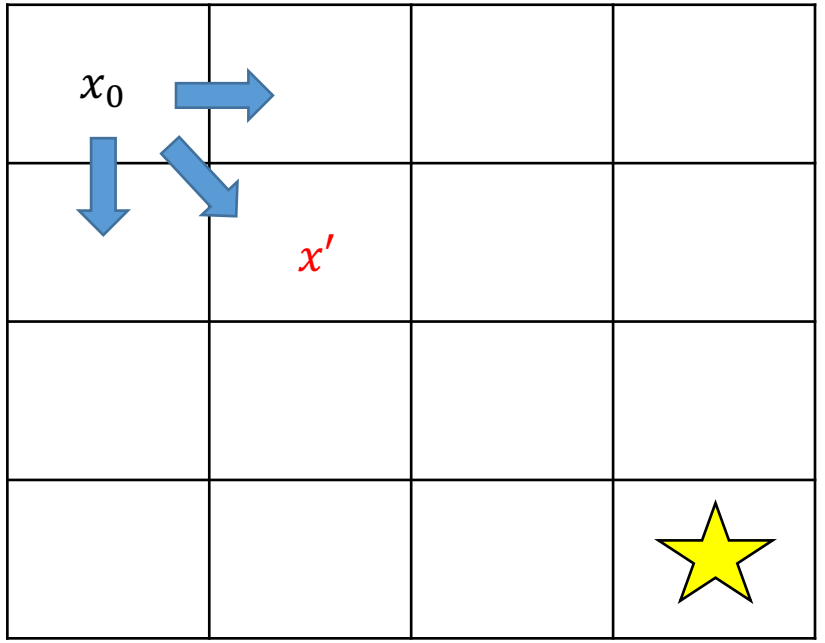
- Generally more focused on discrete problems
- Classical AI planning problems (spanning graphs, travelling salesman, orienteering) often appear in robot planning



Defining a discrete planning problem

We must specify the following elements:

1. State space *X*: possible states of the world
2. Initial state $x_0 \in X$
3. Actions: function A(*x*) returns the set of actions available at each state $x \in X$
4. Transition model: function R(*x*,*a*) describes the *successor* state achieved by applying action *a* in state *x*.
5. Goal test: G(x) returns true/false to indicate whether we have reached a goal
6. Cost: C(*x*,*a*) of applying action *a* in state *x*.



Taking action a at state x requires:

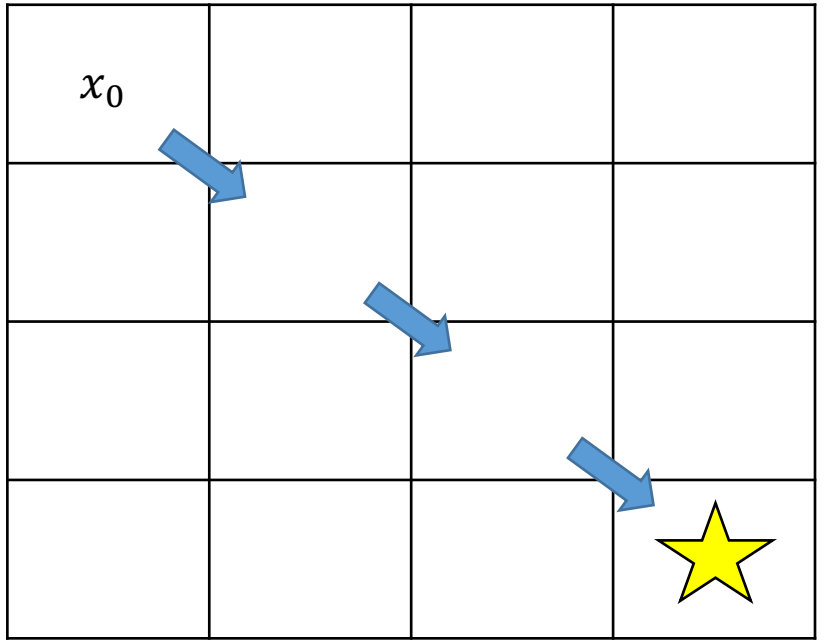
- T amount of time
- F amount of fuel
- D distance to be traversed, etc.

Solutions of a planning problem

Given a planning problem $P = (X, x_0, A, R, G, C)$:

- A *solution* is a *sequence of actions* $a_1, a_2, a_3, \ldots$ that leads from the initial state x_0 to a goal state (a state satisfying G(x) = true).

- An *optimal solution* is a solution attaining the *minimum possible cost*.



Toy example: Vacuum world

- **States:**
  - 2 locations (each of which may contain dirt)
  - Agent location
- **Initial state:** Any
- **Actions:** Left, right, suck
- **Transition model:** As expected (see diagram)
- **Goal test:** No dirt remains in any square
- **Cost:** Each action costs 1

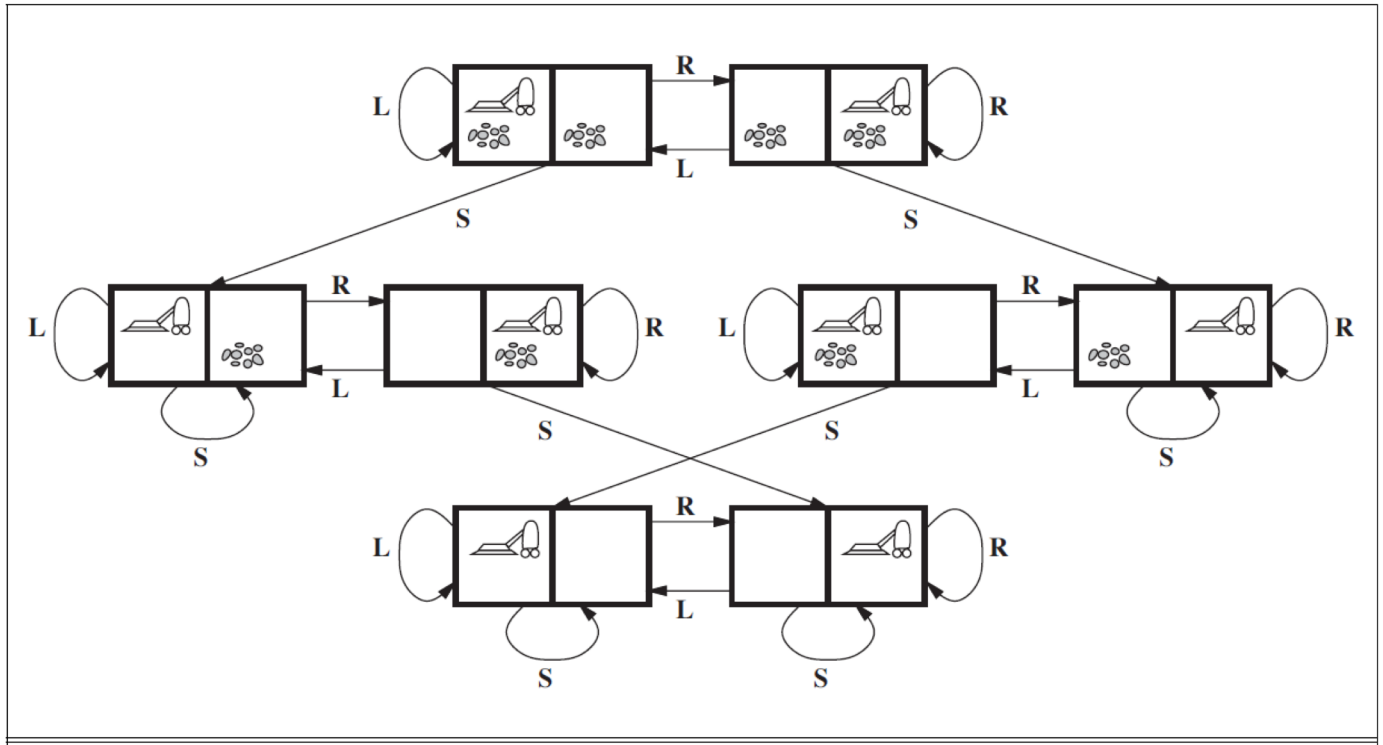
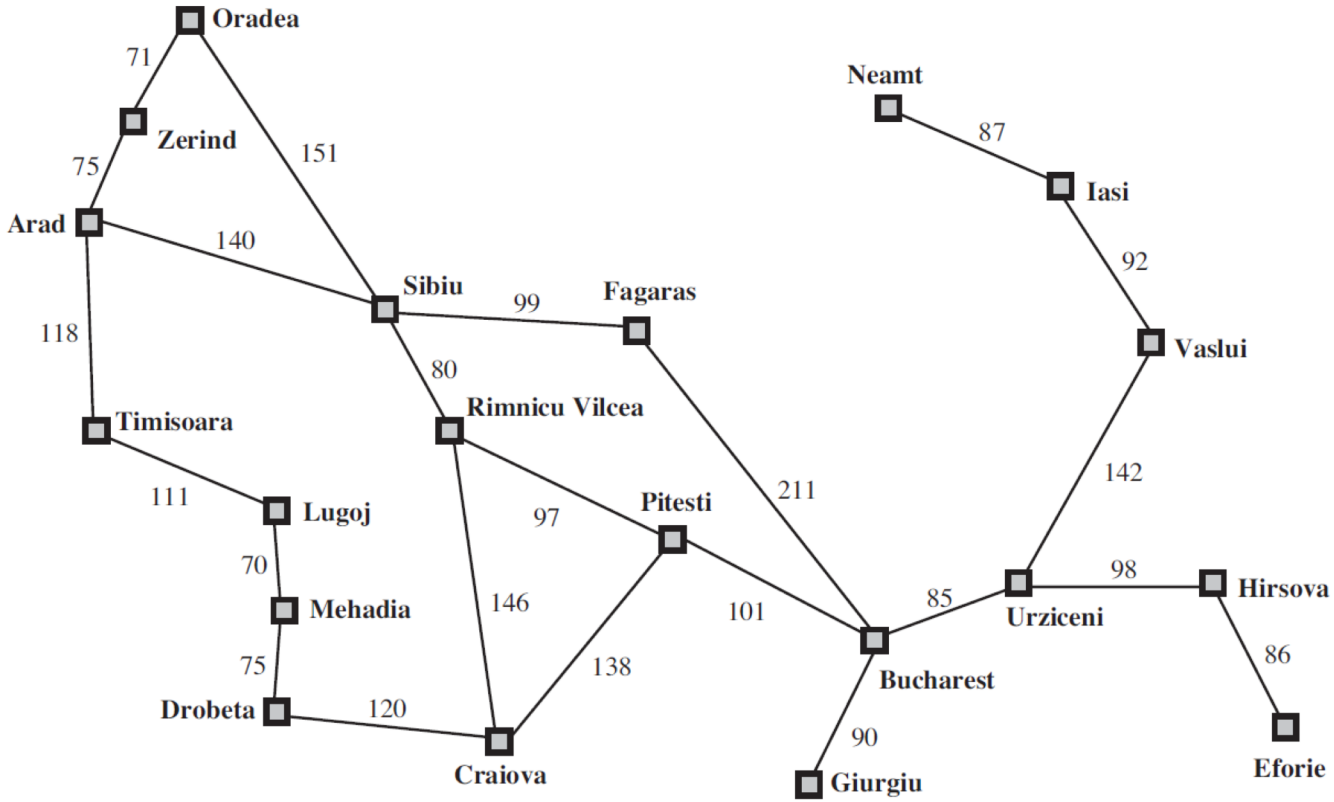


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

Real-world example: Route finding

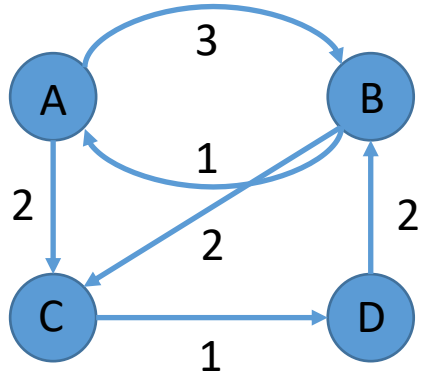
- **States:** Specified set of locations (e.g. cities)
- **Initial state:** Any
- **Actions:** Following any edge (link) between current location and a neighboring location
- **Transition model:** As expected (arrive in the neighboring city)
- **Goal test:** Have you arrived in the goal city?
- **Cost:** Several possible choices:
  - **Money:** Price of tickets, gas, etc.
  - **Time:** Elapsed travel time via the given route



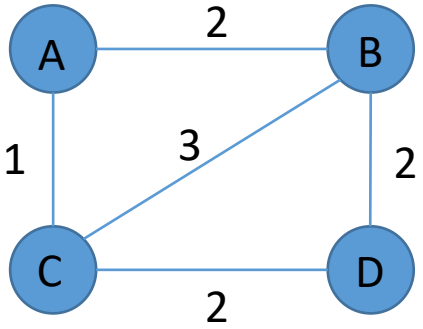
Solving a discrete planning problem

Basic insight: We can model each planning problem $P = (X, x_0, A, R, G, C)$ as a weighted directed graph $G = (X, E, w)$, where:

- Vertices of G are the *states X*
- Edge set *E* contains $i \to j$ iff there is an *action* $a \in A(i)$ for which $j = R(i, a)$.
- The weight $w(i, j)$ of an edge $(i, j) \in E$ is just the cost $C(i, j)$.
- A path is a sequence of edges which joins a sequence of vertices.
- A shortest path between two vertices is a path with the minimum sum of edge weights.
- The graph distance between two vertices is the sum of weights over the shortest path between them.



Weighted directed



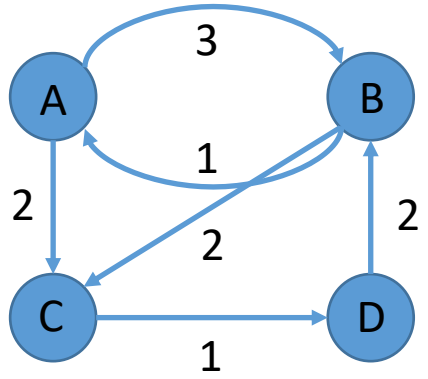
Weighted undirected

(If it is an unweighted graph, the graph distance is the total number of edges on the shortest path.)

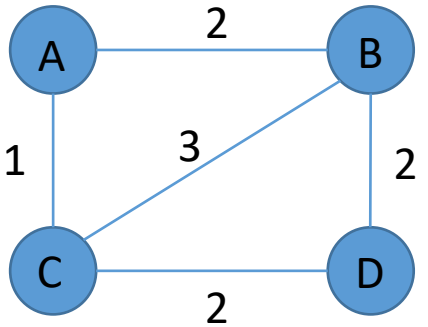
Solving a discrete planning problem

Basic insight: We can model each planning problem $P = (X, x_0, A, R, G, C)$ as a weighted directed graph $G = (X, E, w)$, where:

- Vertices of G are the *states X*
- Edge set *E* contains $i \to j$ iff there is an *action* $a \in A(i)$ for which $j = R(i, a)$.
- The weight $w(i, j)$ of an edge $(i, j) \in E$ is just the cost $C(i, j)$.
- A path is a sequence of edges which joins a sequence of vertices.



Weighted directed

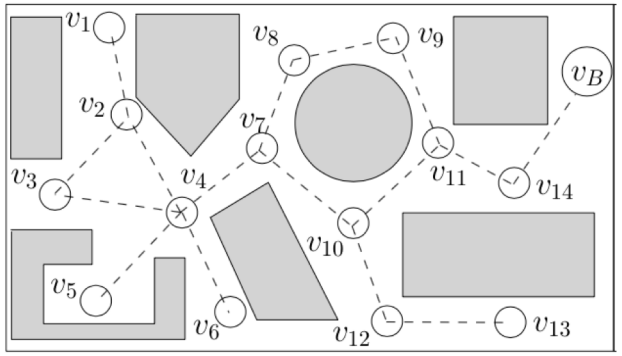


Weighted undirected

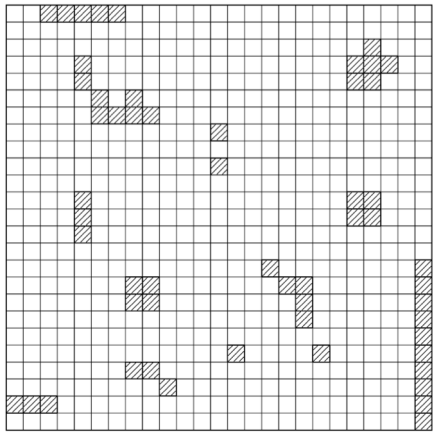
Therefore: We can recast *planning* as *graph search*!

- Solution of planning problem *P* ⇔ *path* from initial state x_0 to goal state in *G*
- Optimal solution of *P* ⇔ *minimum-cost path* from x_0 to goal state in *G*

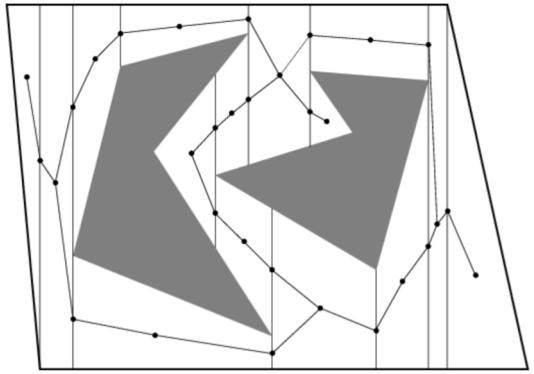
Abstractions of an environment



Graph representation of the environment

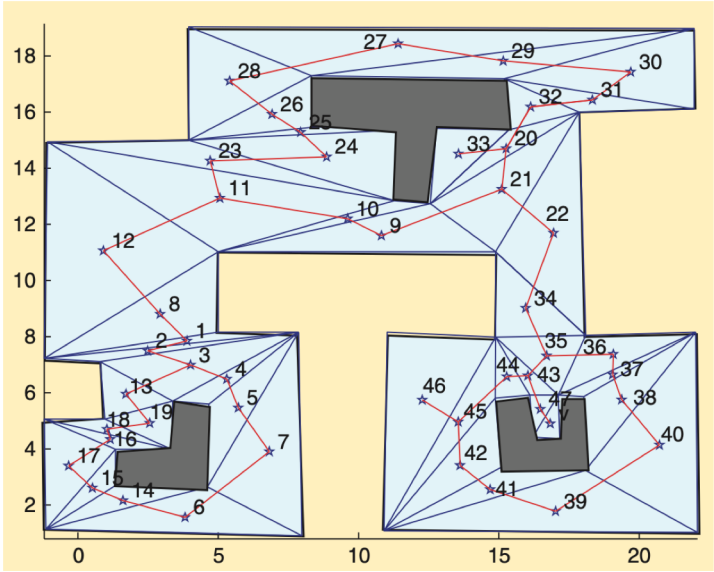


Grid representation of the environment



Vertical cell decomposition

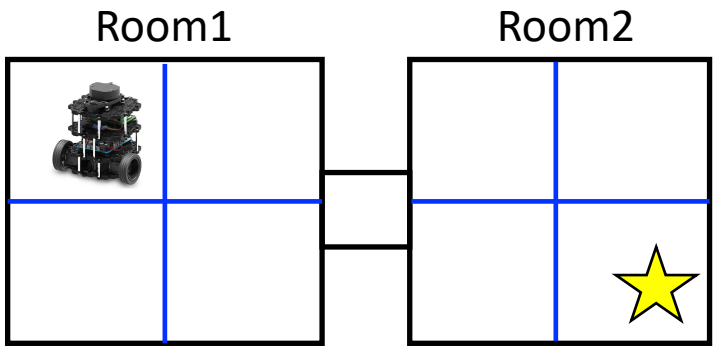
https://lavalle.pl/planning/node262.html

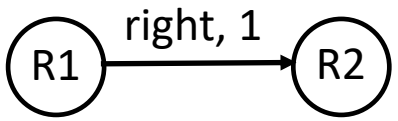


Triangulation of the free space in a polygonal environment and its graph

How to generate graphs?

- Coarser representations are desirable (less states, less actions, quick graph search)
- However, we have to be careful in abstracting environments.





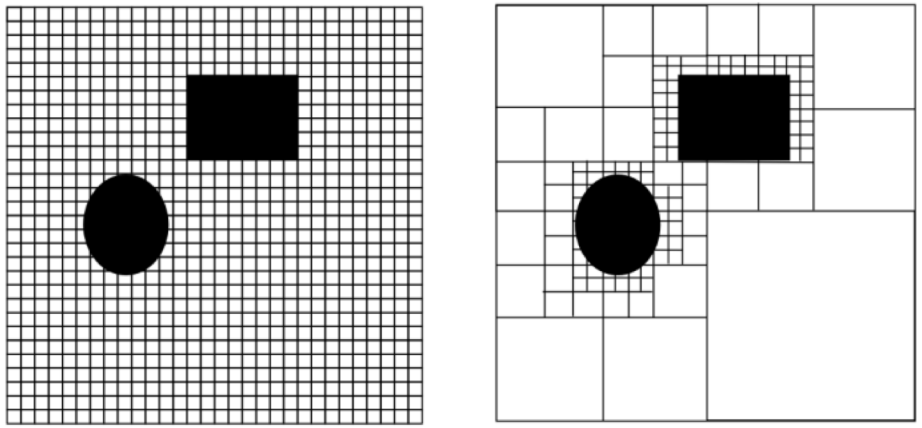
- The edge in this graph means that the robot moves from any state in room 1 to any state in room 2 in one time step.
- Suppose that the robot is slow so it is not possible to move to room 2 in one time step from the upper corner of room 1.
- Then, we might need additional states…
- It is very crucial to take into account robot's dynamical constraints when generating graphs for planning! Otherwise, the plan will not be applicable…

How to generate graphs?

Various ways are used to generate graphs:

Grid based representation

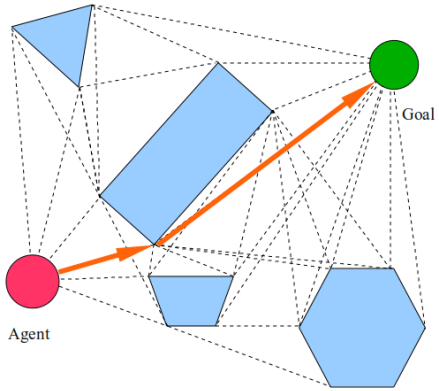
Very easy to implement
Usually suffer some loss of precision
Selecting an appropriate grid resolution is a challenge.



Fixed vs. variable grid cells

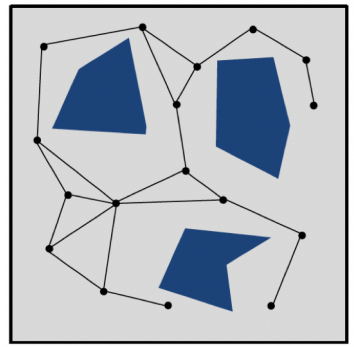
Visibility graphs

Create edges between all pairs of mutually visible vertices
Optimal plans
Limited to straight motion, 2D, polygonal obstacles



Randomly-sampled graphs

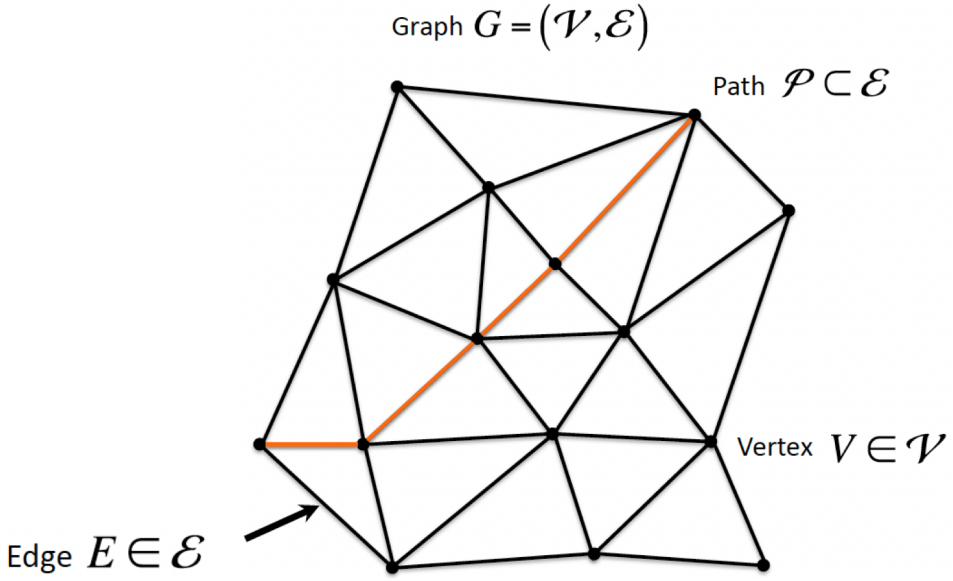
We will discuss more in next lecture.
Require careful consideration to construct graphs with guarantees



Graph search methods

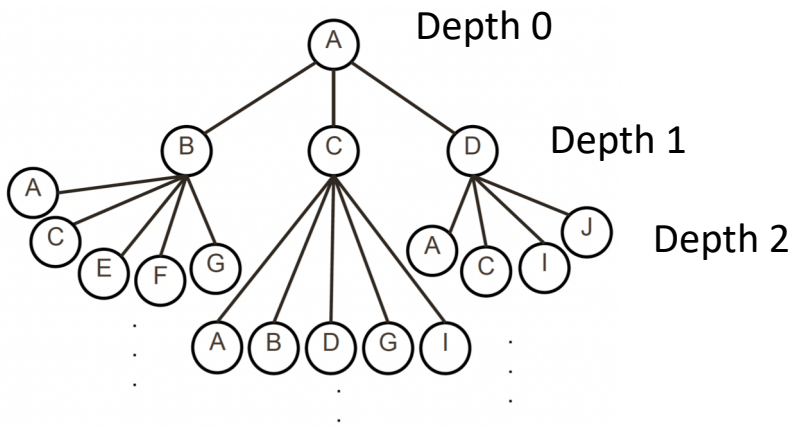
- Suppose that we convert the planning problem into a graph.
- Now, we will learn graph search algorithms to find the min cost path.
  - A min cost path never revisits a node.

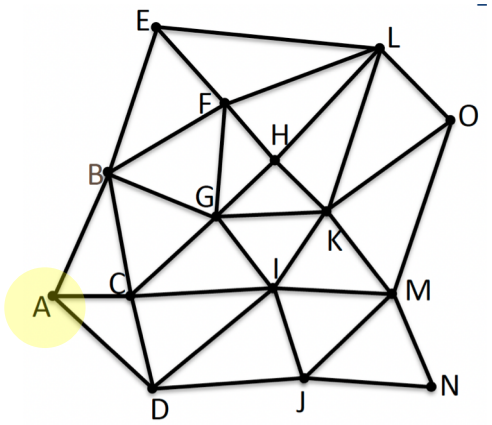
- Graph search methods
  - Breadth-first search (BFS)
  - Depth-first search (DFS)
  - Dijkstra's algorithm
  - A*



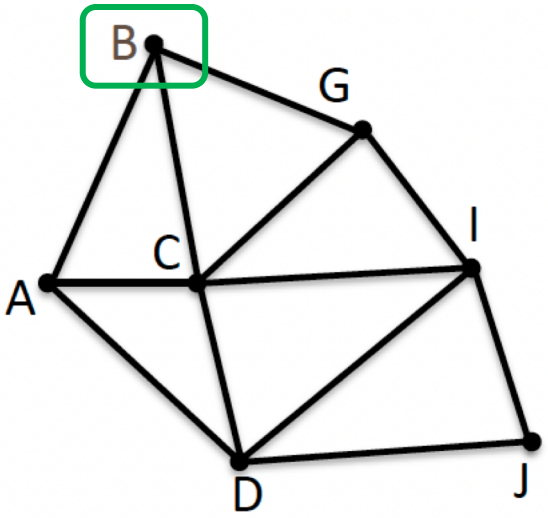
Search trees

- A tree is constructed to search for optimal paths through the environment.
- Start state at the root note
- Children correspond to successors
- For most problems, we avoid constructing the whole tree…

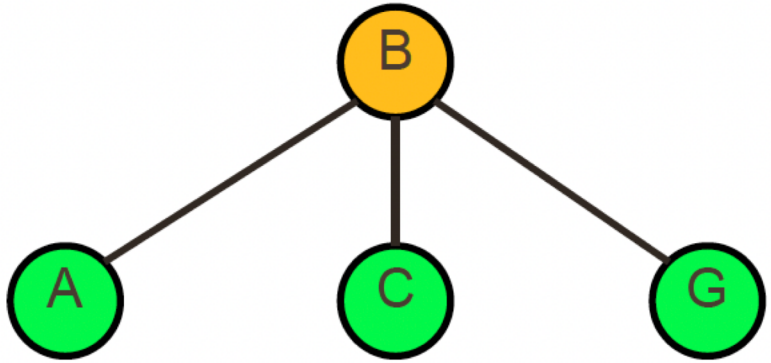


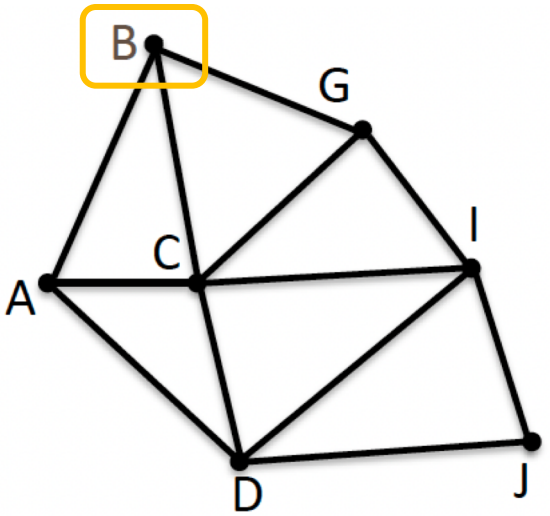


Breadth-first search



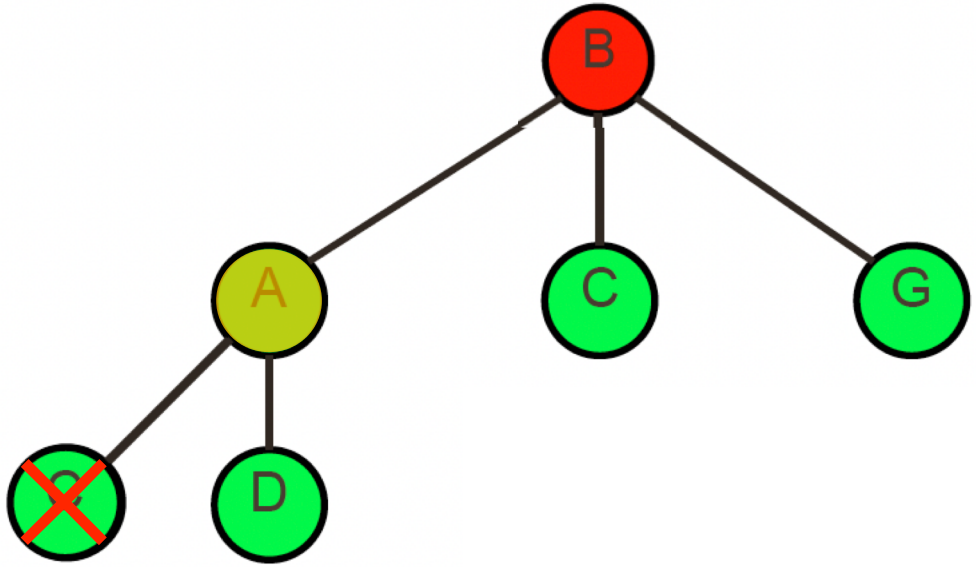
Breadth-first search

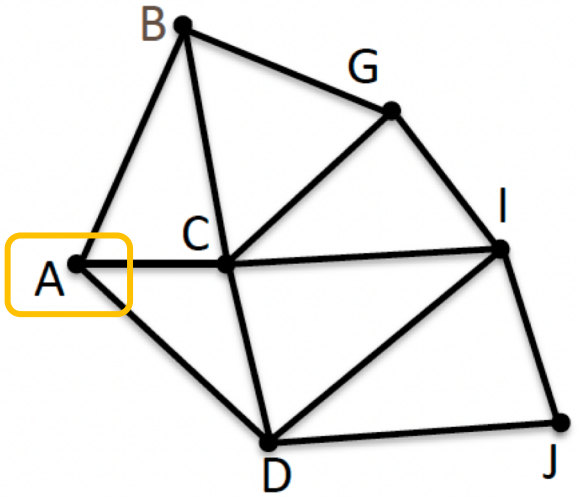




Current: B
Queue: ACG
Visited:

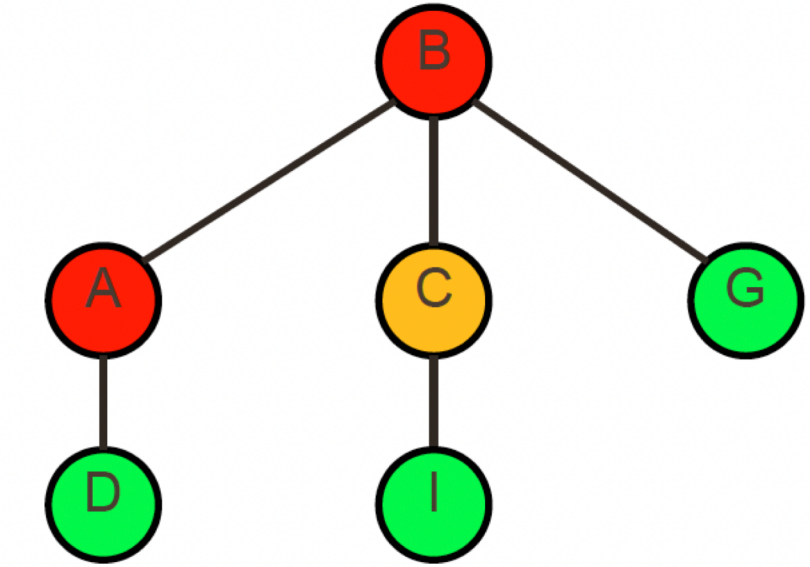
Breadth-first search

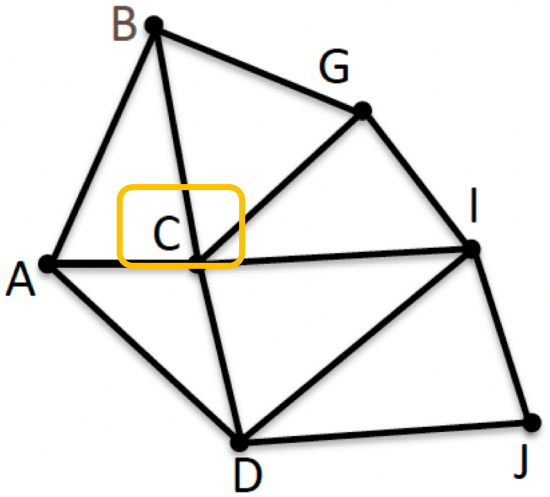




Current: A
Queue: CGD
Visited: B

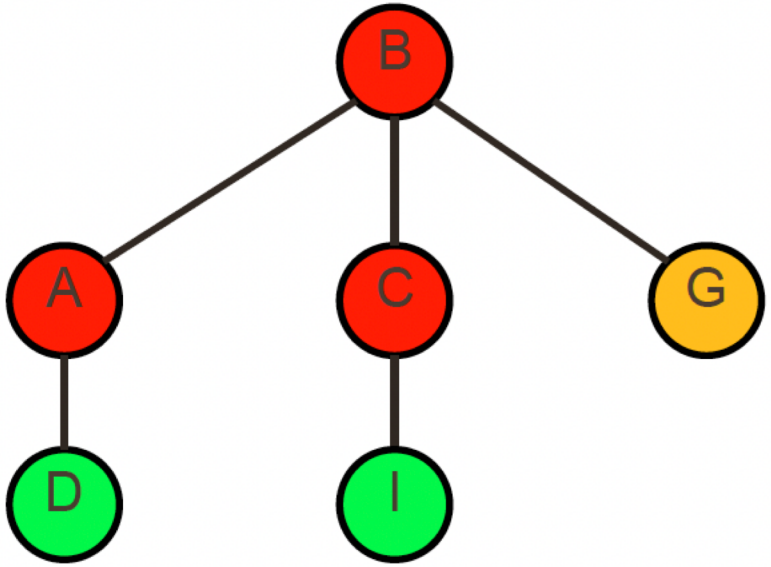
Breadth-first search

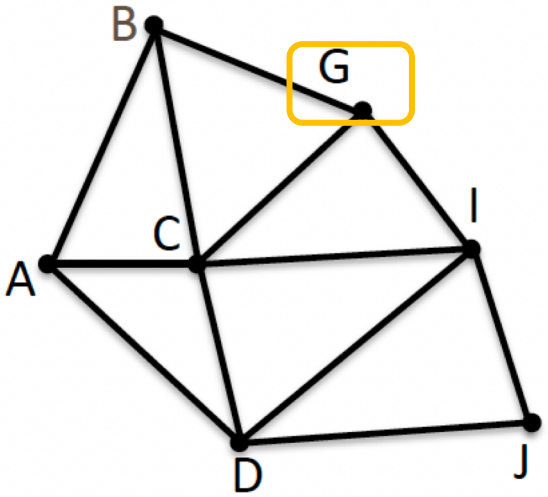




Current: C
Queue: GDI
Visited: BA

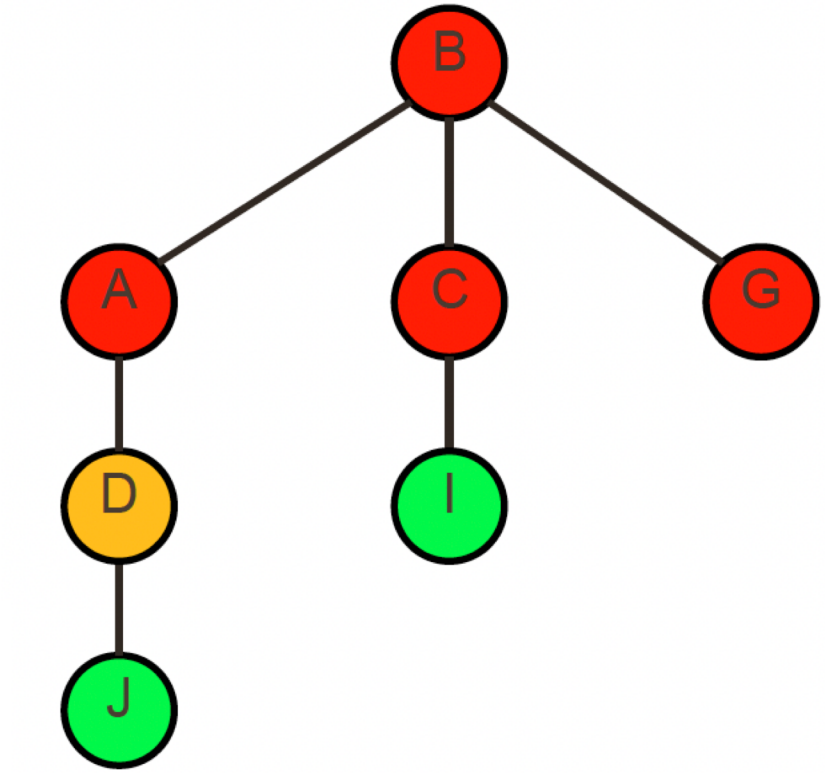
Breadth-first search

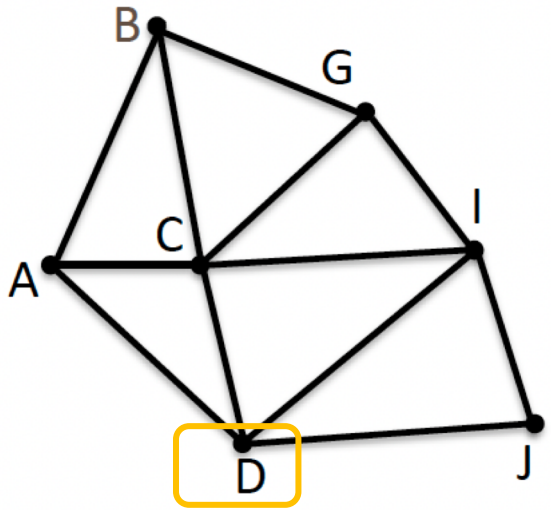




Current: G
Queue: DI
Visited: BAC

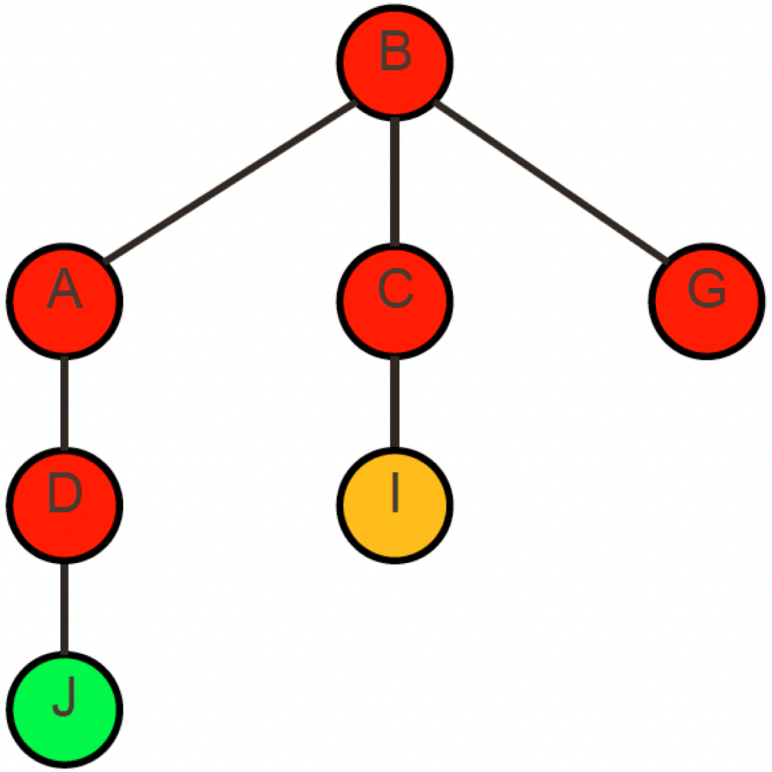
Breadth-first search

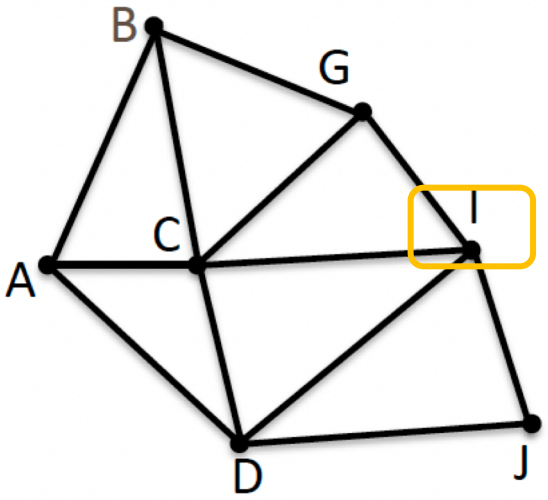




Current: D
Queue: IJ
Visited: BACG

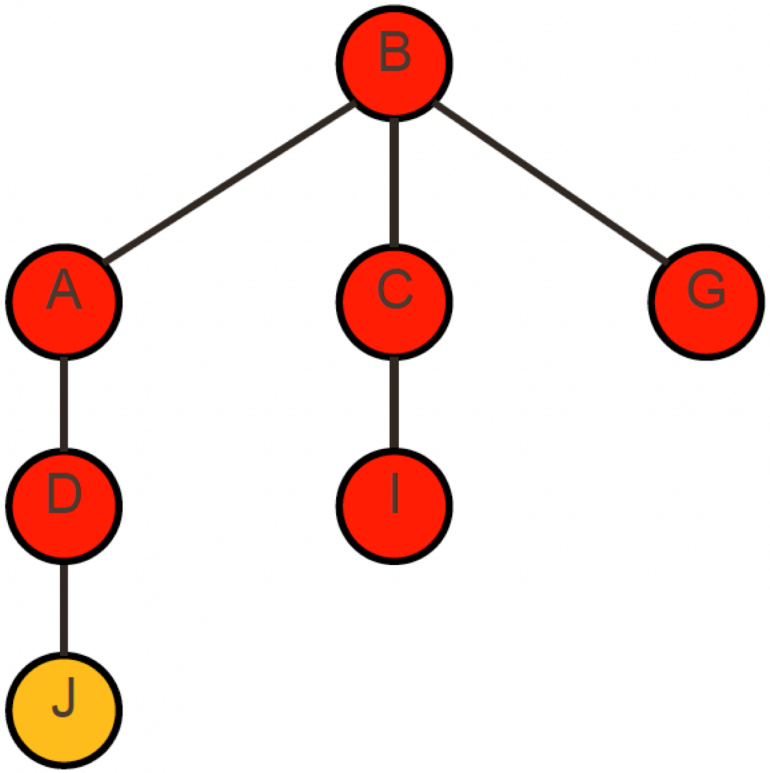
Breadth-first search

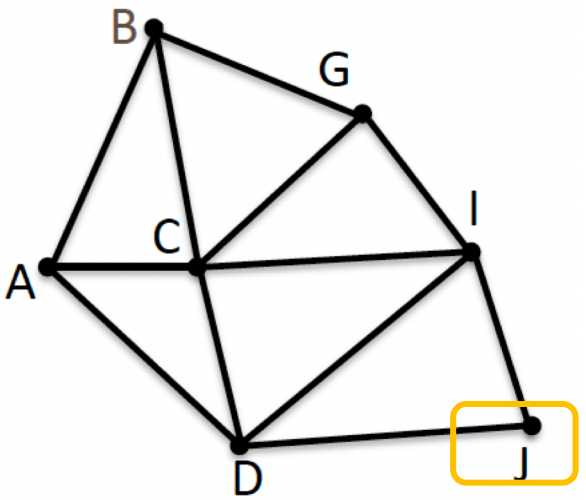




Current: I
Queue: J
Visited: BACGD

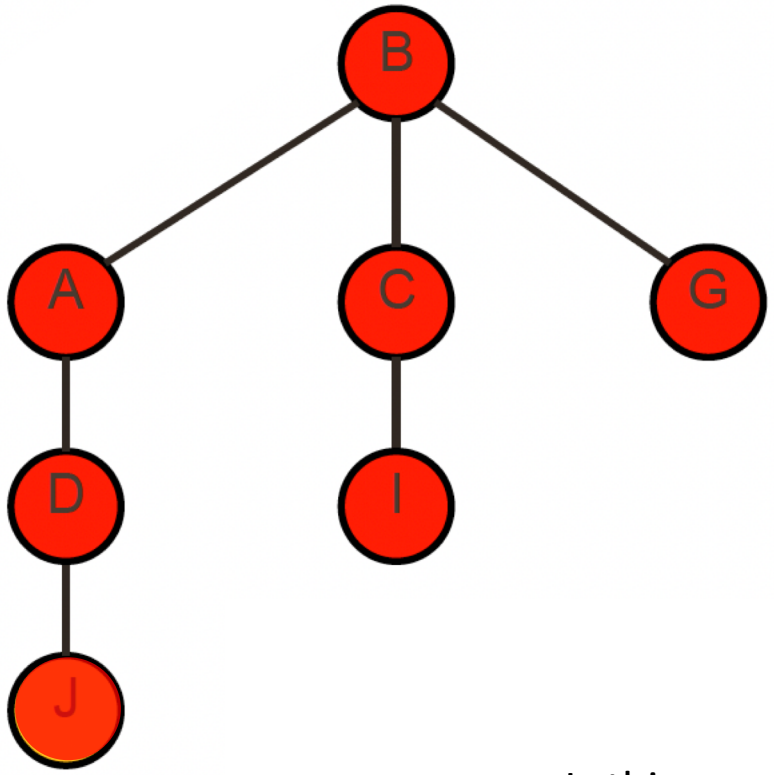
Breadth-first search

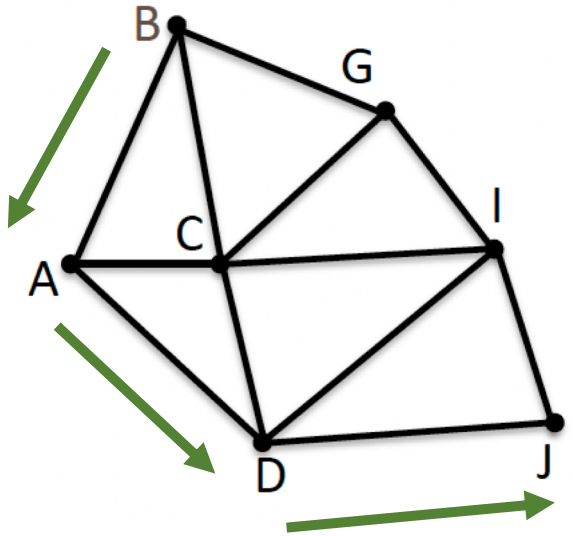




Current: J
Queue:
Visited: BACGDI

Breadth-first search





Current:
Queue:
Visited: BACGDIJ

Is this a unique solution?

Shortest path from B to J: B -> A -> D -> J

Other solutions may exist but have the same number or more transitions

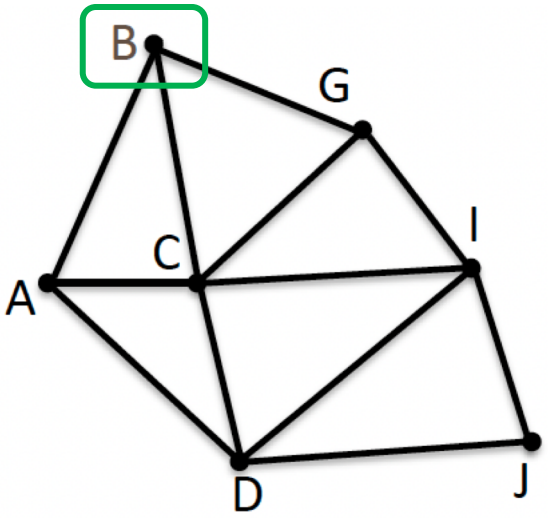
Breadth-first search

- Applicable to graphs with uniform edge weight
- Complete (will find the solution if it exists)
- Guaranteed to find the shortest (number of edges) path
- Time complexity O(|V|+|E|)

  (the time it takes to find a solution)

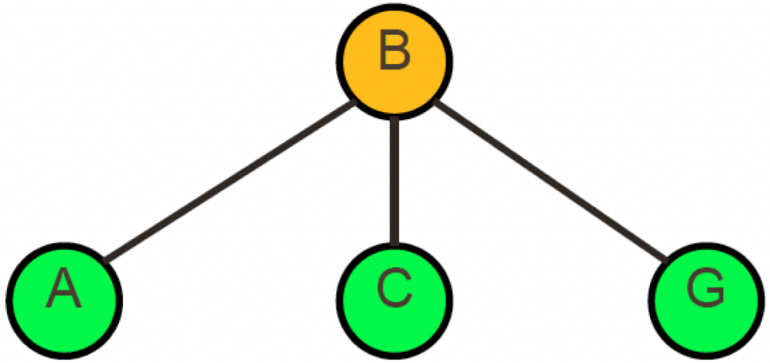
```
BFS

add start vertex to Q
mark start as visited


while Q not empty
        v ← dequeue from Q
        for each adj vertex av of v
                //.... process vertex av....
                if av is not visited
                        add av to Q
                        mark av as visited
```

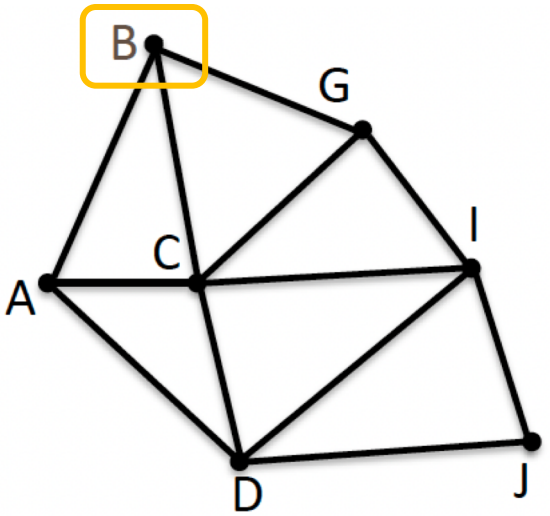
Depth-first search



Instead of searching across levels of the tree, DFS starts at the root node and explores as far as possible along each branch before backtracking

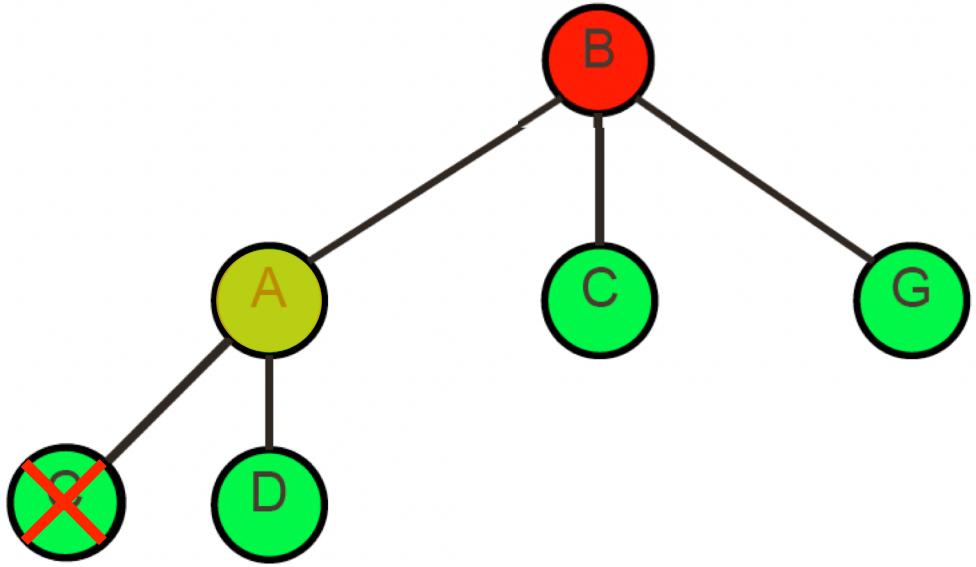
Depth-first search

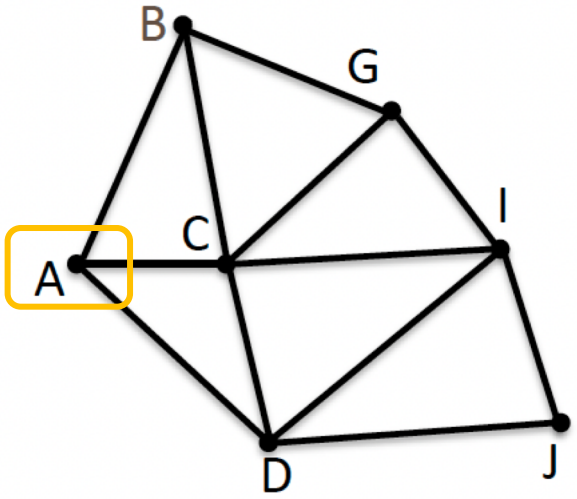




Current: B
Queue: ACG
Visited:

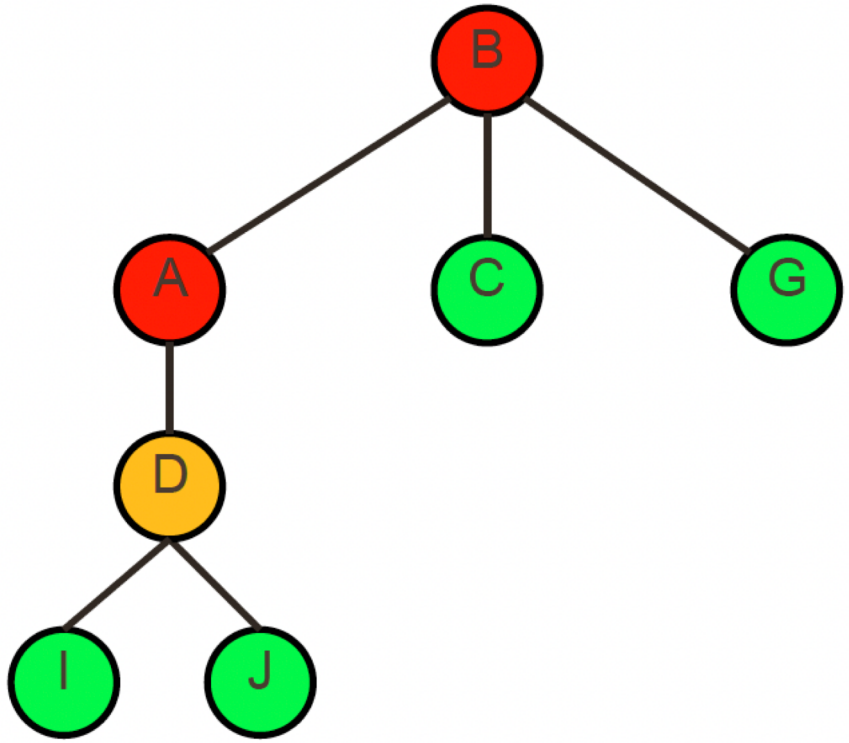
Depth-first search

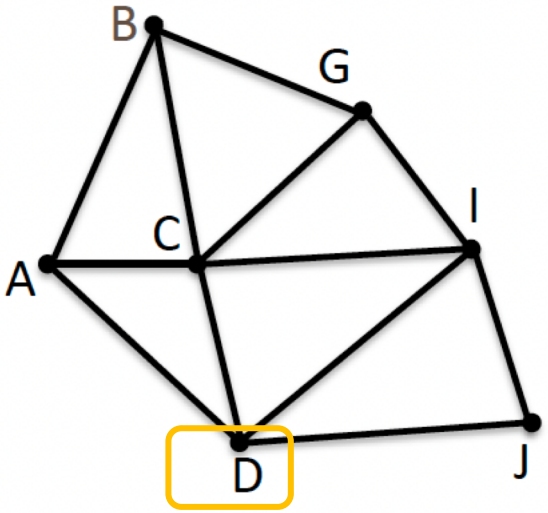




Current: A
Queue: CGD
Visited: B

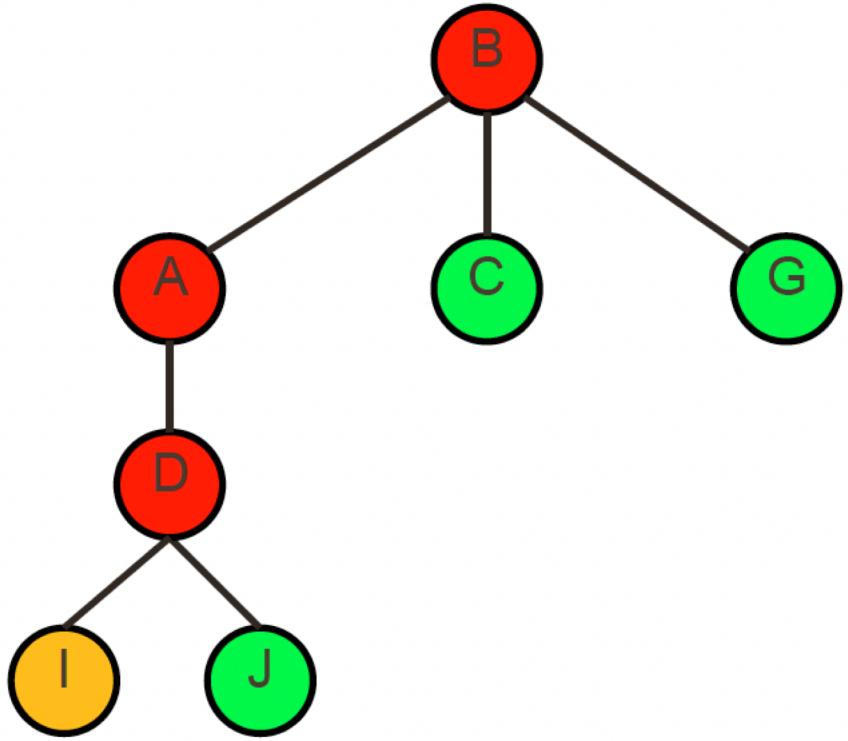
Depth-first search

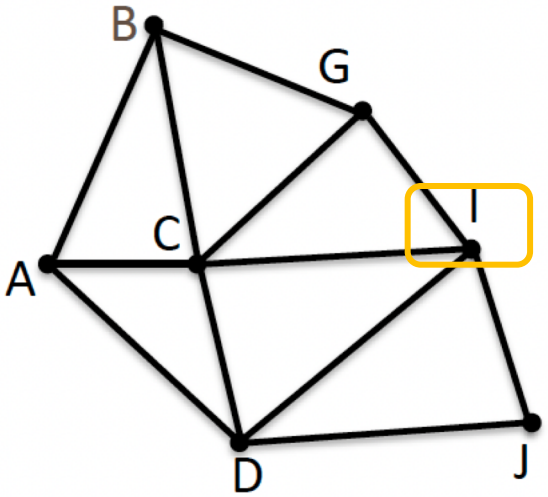




Current: D
Queue: CGIJ
Visited: BA

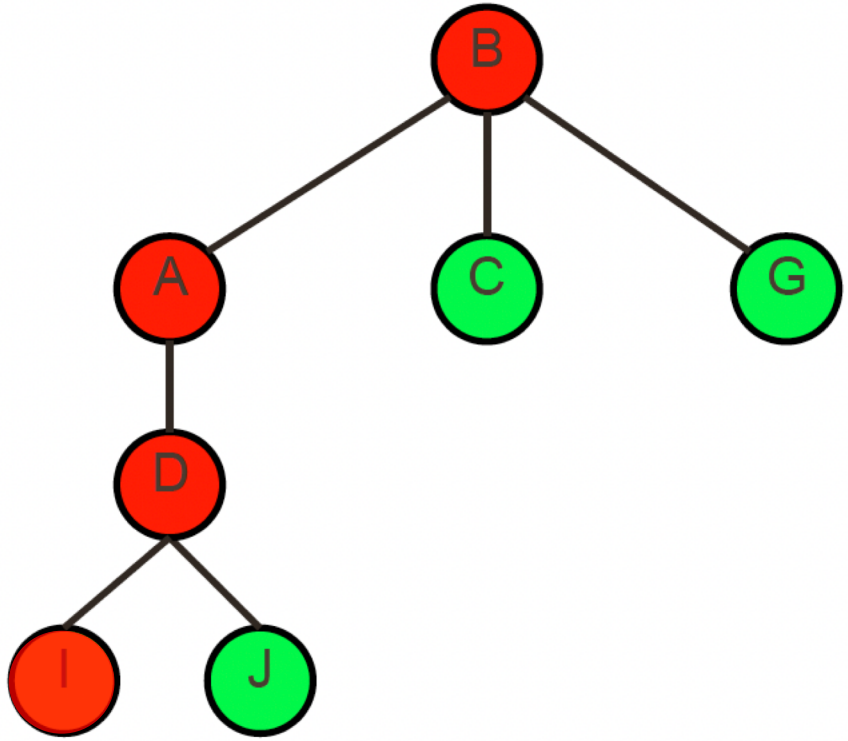
Depth-first search

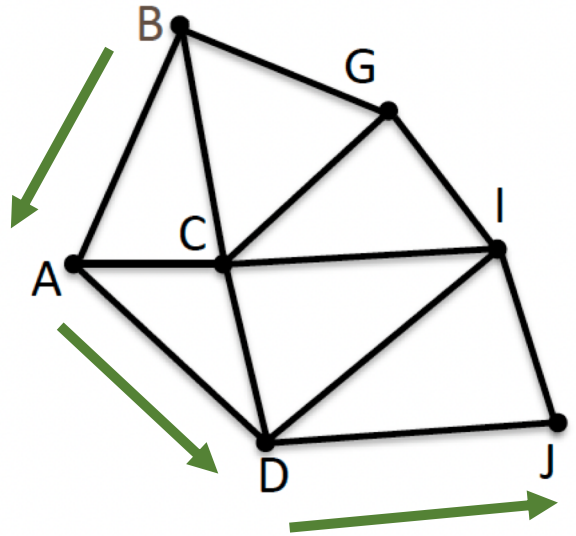




Current: I
Queue: CGJ
Visited: BAD

Depth-first search

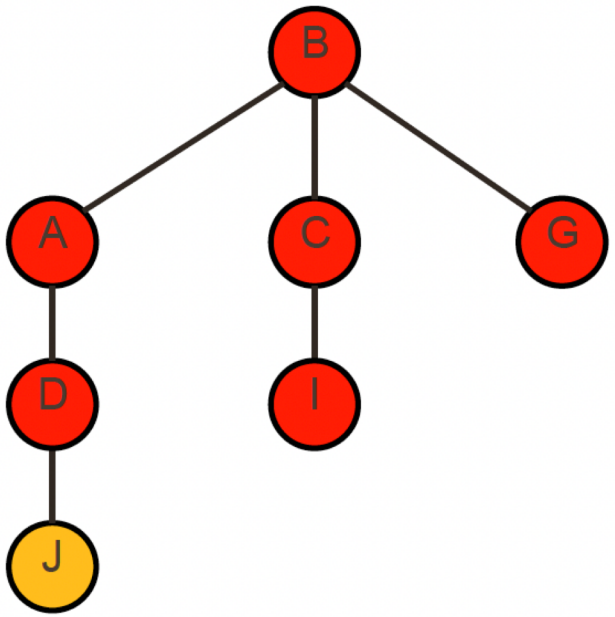




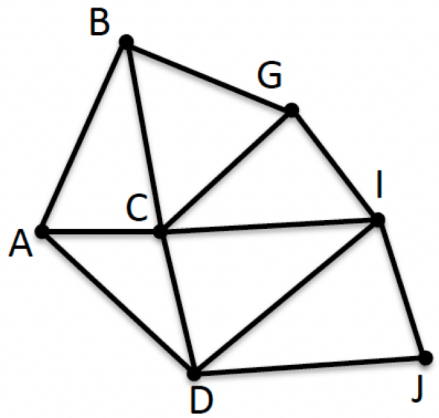
Current:
Queue:
Visited: BADICGJ

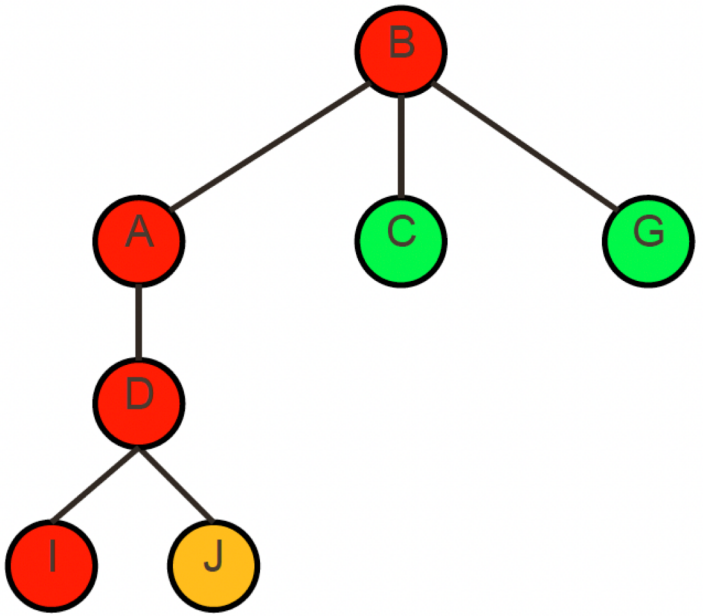
Path from B to J: B -> A -> D -> J

Search tree comparison



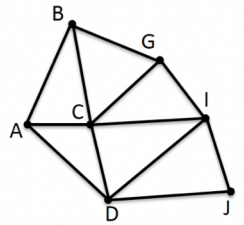
BFS

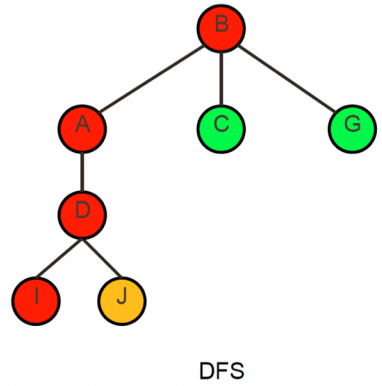




DFS

Depth-first search





DFS

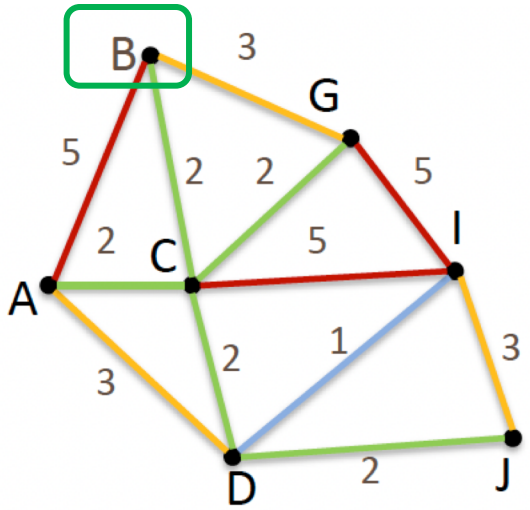
- Applicable to graphs with uniform edge weight
- DFS is not guaranteed to find the shortest path (can find the shortest path if the graph is a tree).
- Lower memory footprint than BFS with high-branching
- Time complexity O(|V|+|E|)
- When target is close to the initial state, BFS is preferable. When target is far from the initial state, DFS is preferable.
- Both BFS and DFS are simple to implement, but might be inefficient. More complex algorithms are faster, but generally more difficult to implement

Dijkstra's algorithm

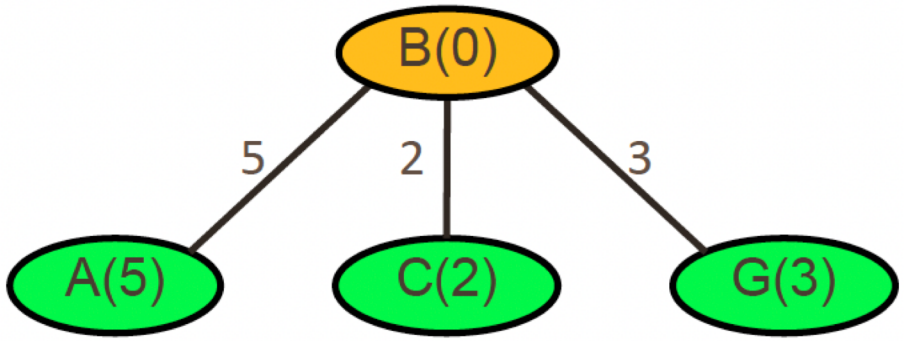
- Published by Edsger Dijkstra in 1959
- Basic idea of expanding in order of closest to start (BFS with edge costs)
- One of the most commonly used routing algorithms in graph traversal problems
- Asymptotically the fastest known single-source shortest path algorithm for arbitrary directed graphs
- Time complexity $O((|V|+|E|)\log|V|)$
- Complete and optimal

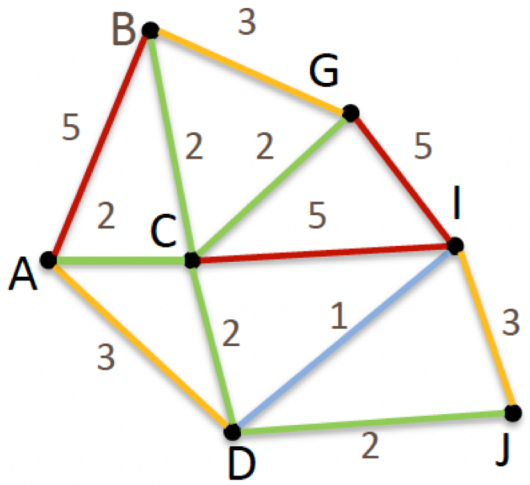
Dijkstra's algorithm

B(0)



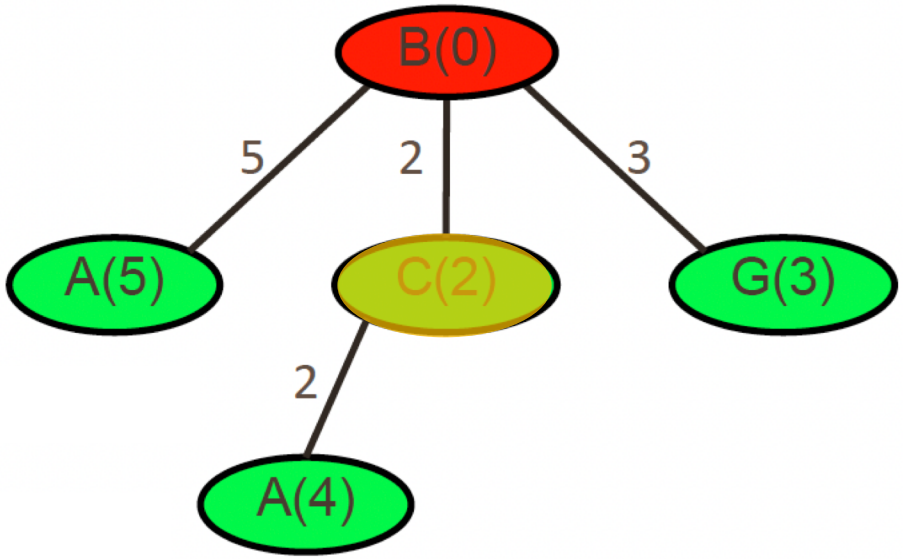
Dijkstra's algorithm

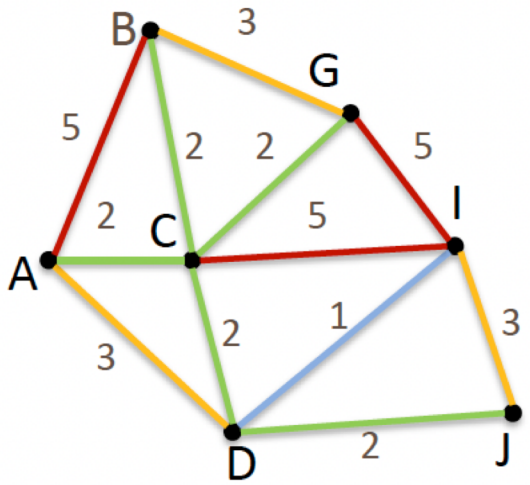




Current: B(0)
Queue: A(5),C(2), G(3)
Visited:

Dijkstra's algorithm

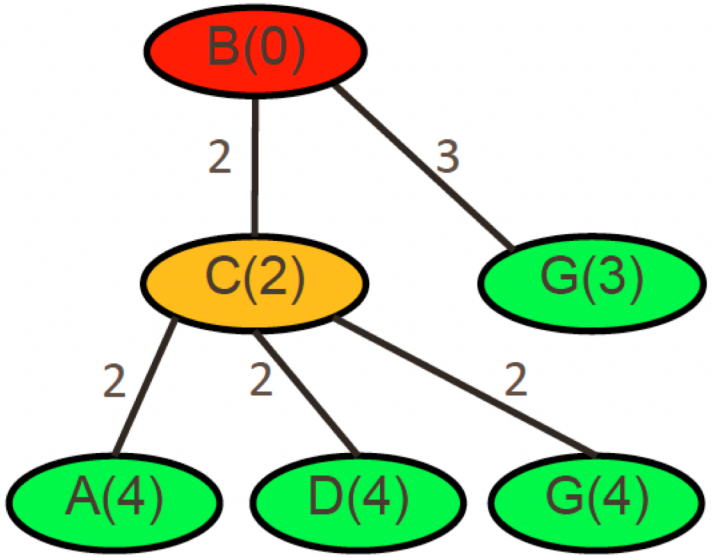


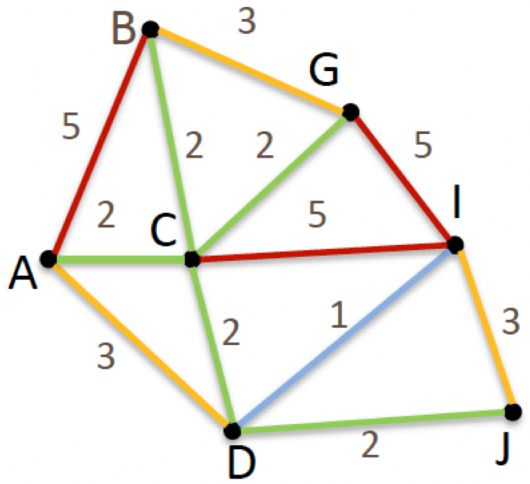


Current: C(2)
Queue: G(3), A(4)
Visited: B(0)

Resolve duplicate

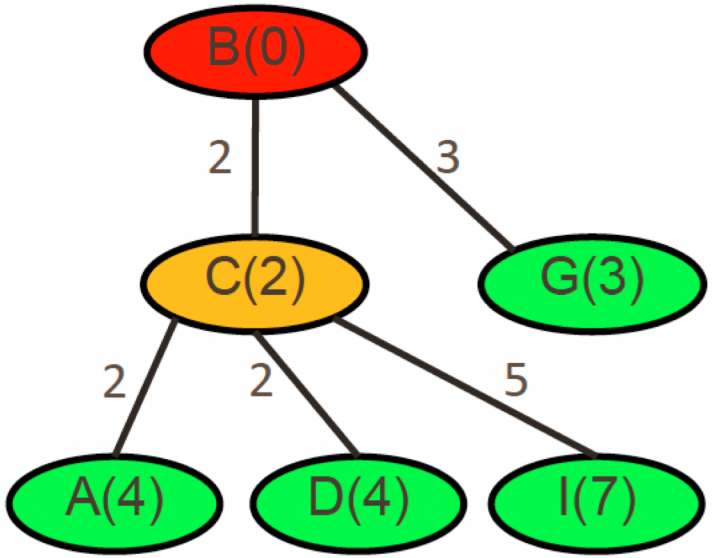
Dijkstra's algorithm

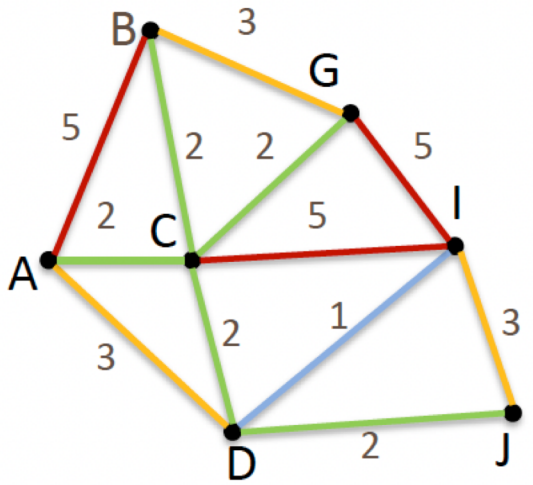




Current: C(2)
Queue: G(3), A(4), D(4)
Visited: B(0)

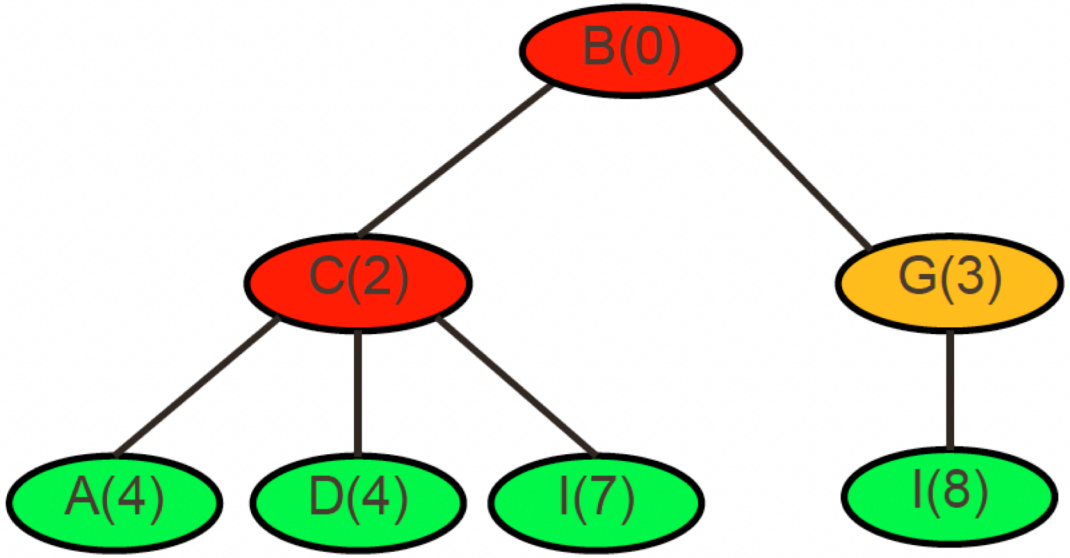
Dijkstra's algorithm

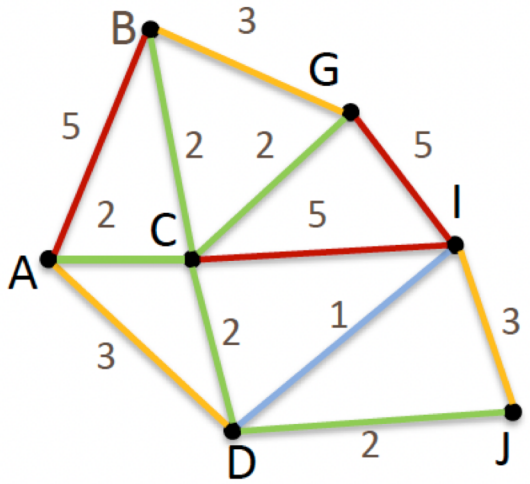




Current: C(2)
Queue: G(3), A(4), D(4), I(7)
Visited: B(0)

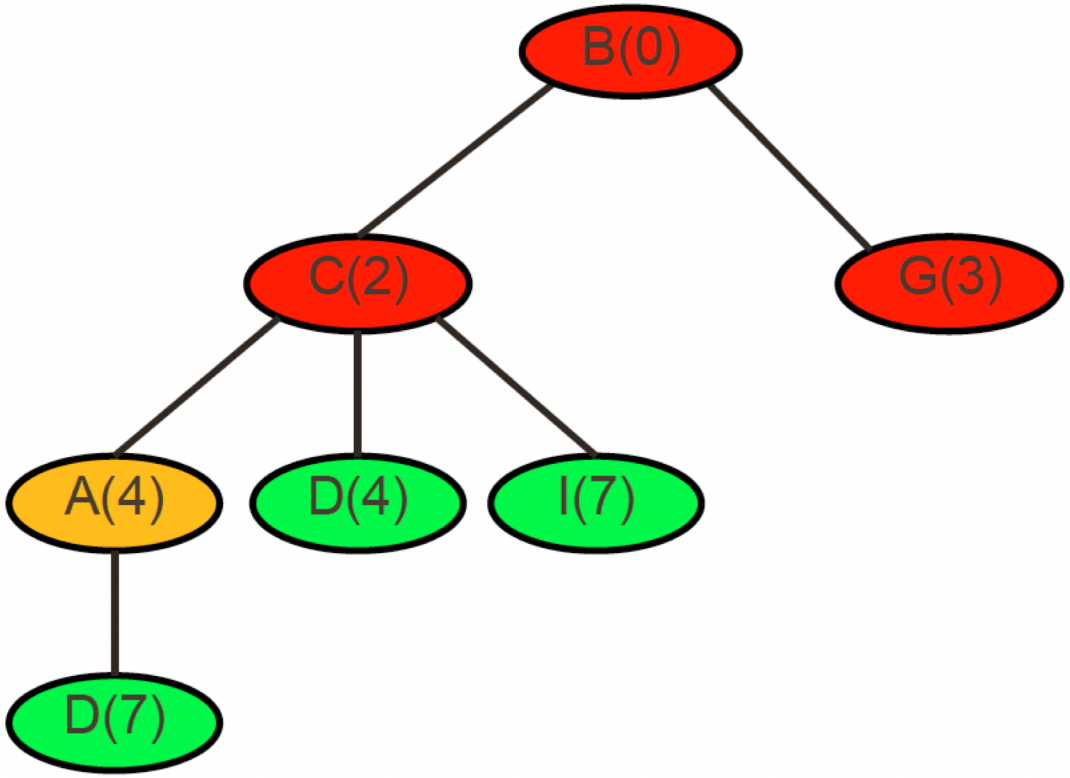
Dijkstra's algorithm

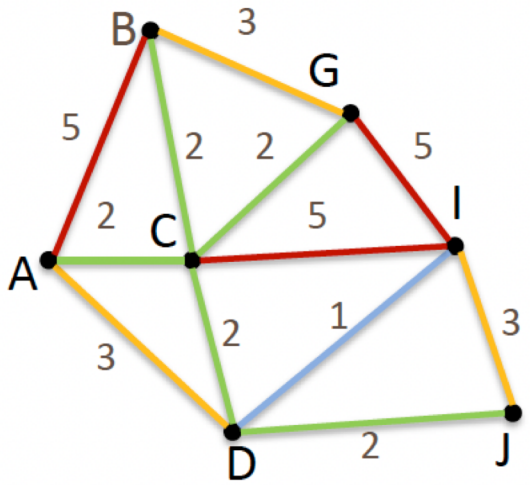




Current: G(3)
Queue: A(4), D(4), I(7)
Visited: B(0), C(2)

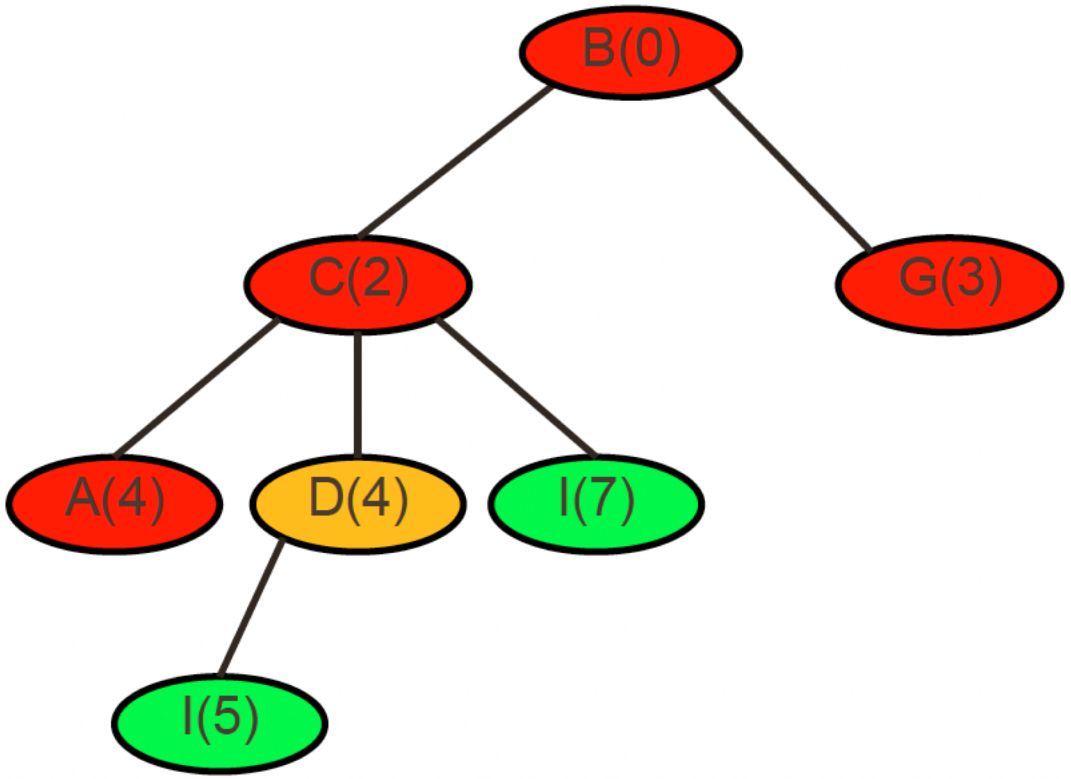
Dijkstra's algorithm

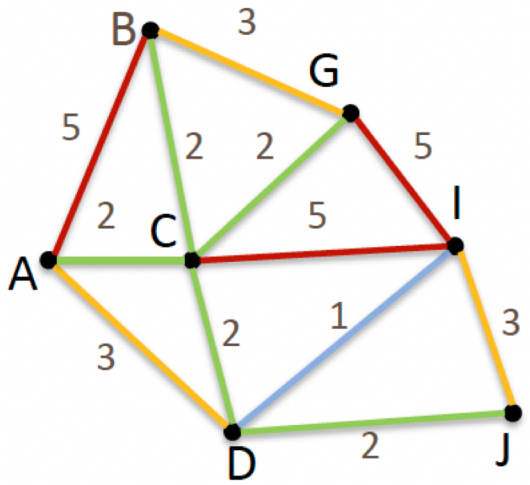




Current: A(4)
Queue: D(4), I(7)
Visited: B(0), C(2), G(3)

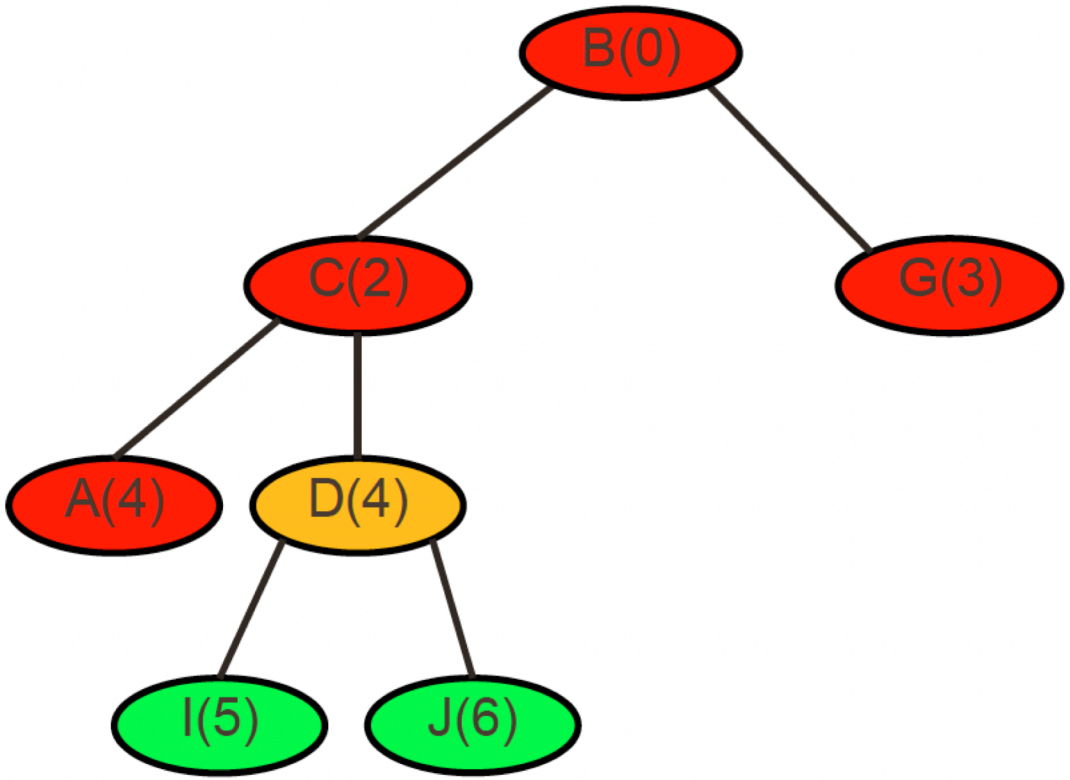
Dijkstra's algorithm

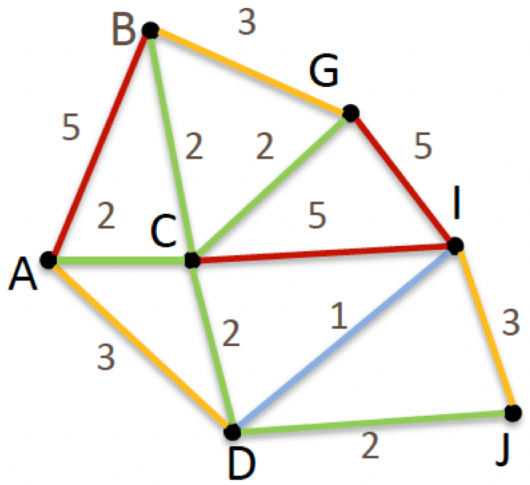




Current: D(4)
Queue: I(7)
Visited: B(0), C(2), G(3), A(4)

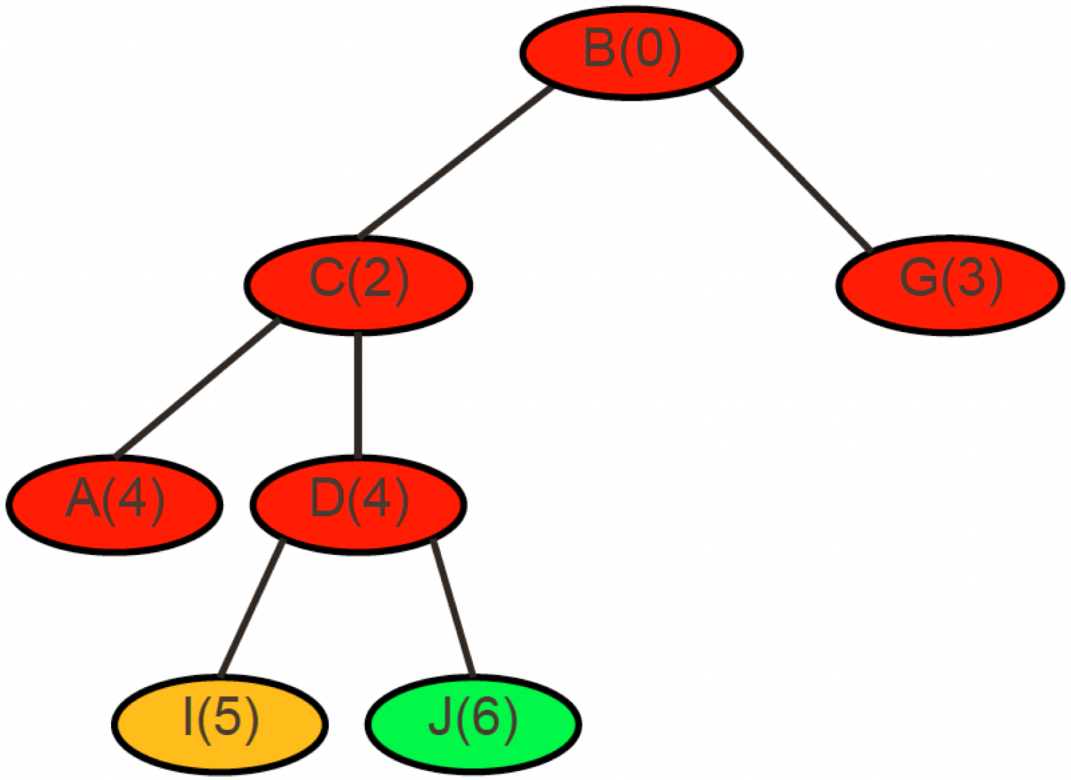
Dijkstra's algorithm

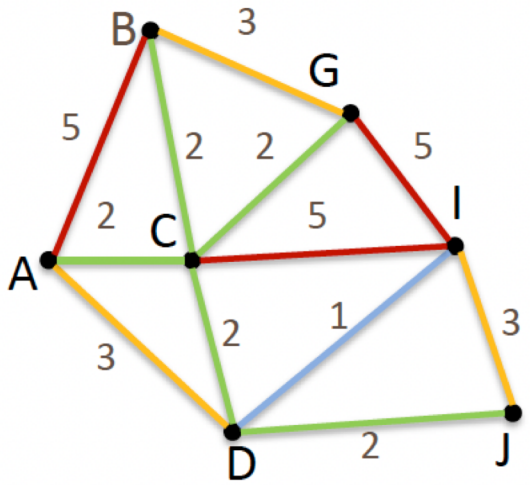




Current: D(4)
Queue: I(5), J(6)
Visited: B(0), C(2), G(3), A(4)

Dijkstra's algorithm

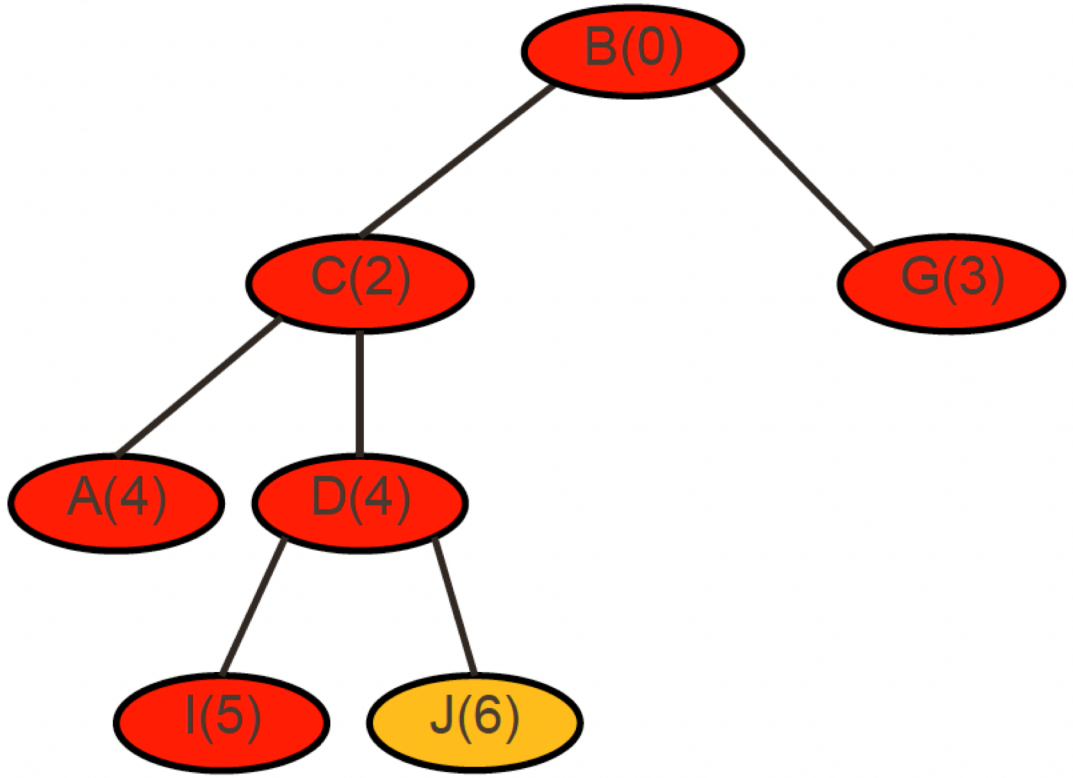


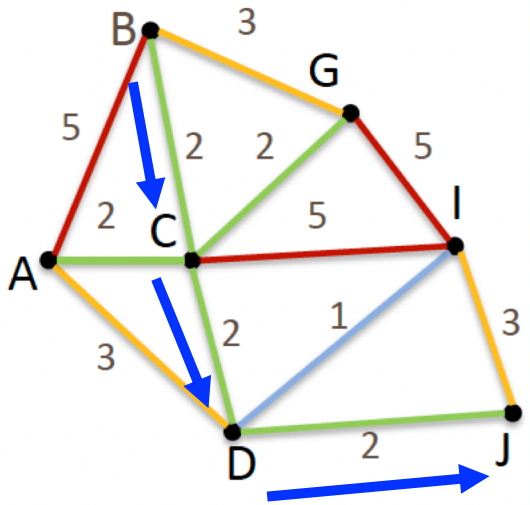


Current: I(5)
Queue: J(6)
Visited: B(0), C(2), G(3), A(4), D(4)

Dijkstra's algorithm

Go from B to J: B -> C -> D -> J
Cost: 6



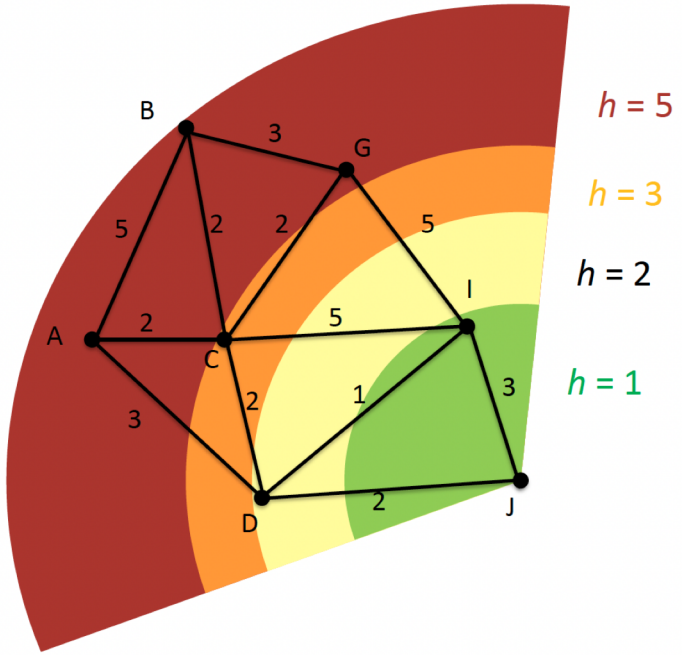


Current: J(6)
Queue:
Visited: B(0), C(2), G(3), A(4), D(4), I(5)

Does not know the goal exists until it reaches it

Could we guide the search to expand nodes that are closer to the goal earlier?

A* heuristic search



Heuristic:

- Any optimistic estimate of how close a state is to a goal

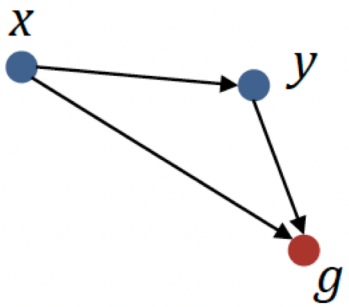
For every node (vertex):

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

Cost to arrive

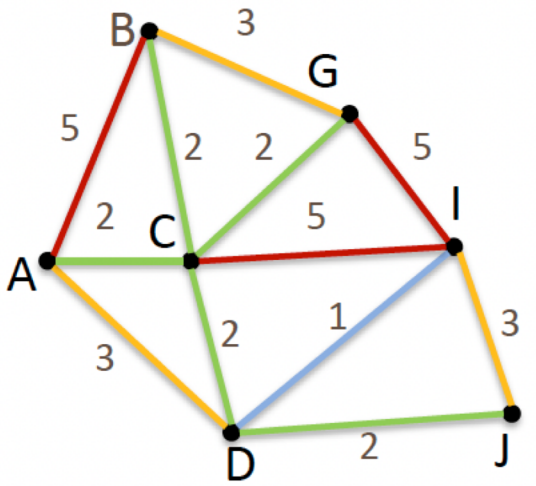
Heuristic cost to goal

- Let $d(x, y)$ be the cost of going from x to y
- The heuristic should never overestimate the true cost: $h(n) \leq d(x, goal)$
- The heuristic must be consistent: $h(x) \leq d(x, y) + h(y)$
- Examples: Manhattan distance, Euclidian distance, Zero (Dijkstra's algorithm)

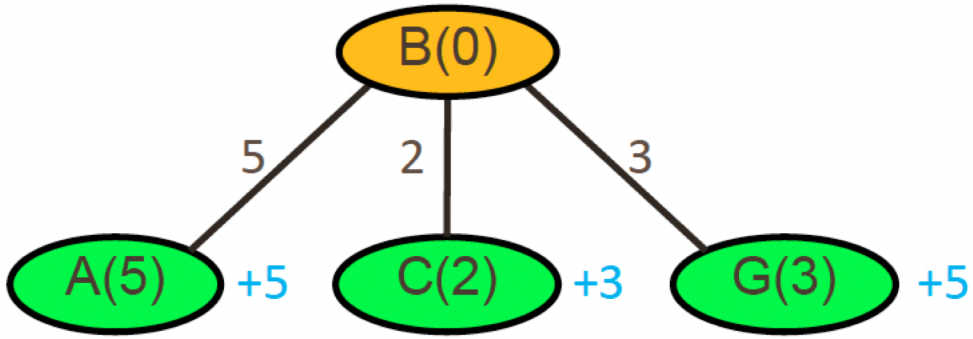


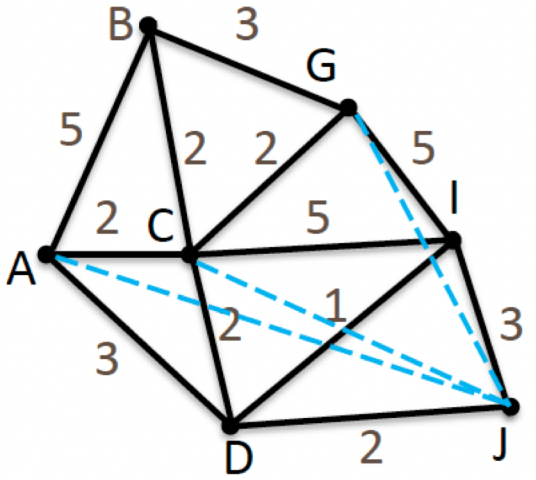
A*

B(0)



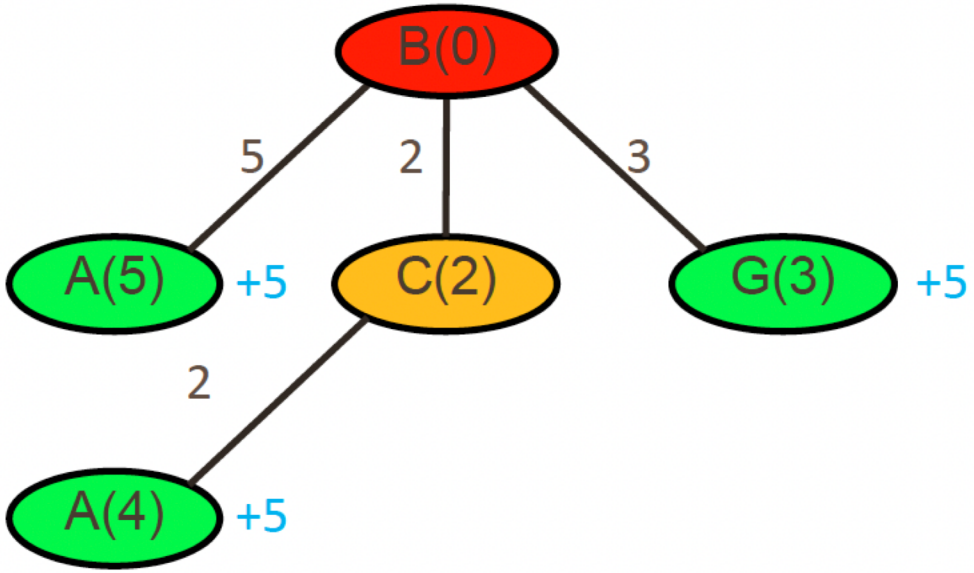
A*

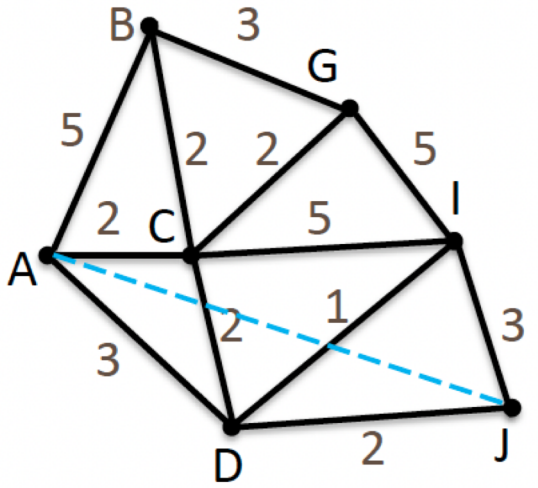




Current: B(0)
Queue: A(5+5), C(2+3), G(3+5)
Visited:

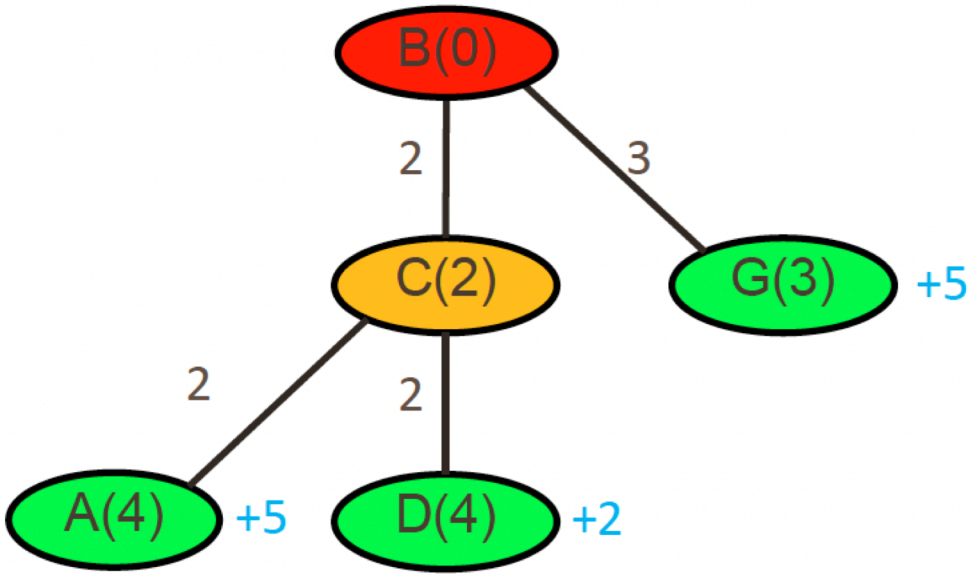
A*

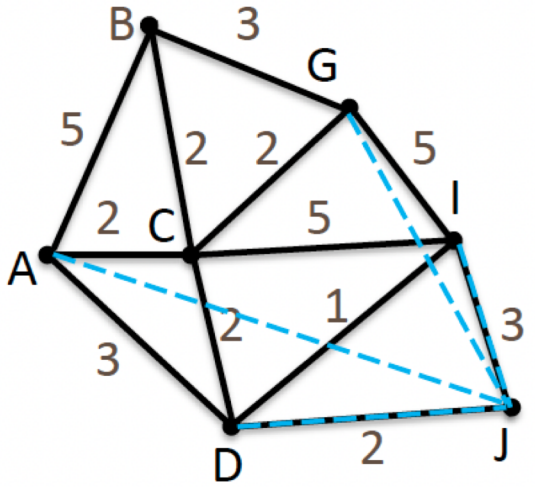




Current: C(2)
Queue: A(4+5), G(3+5)
Visited: B(0)

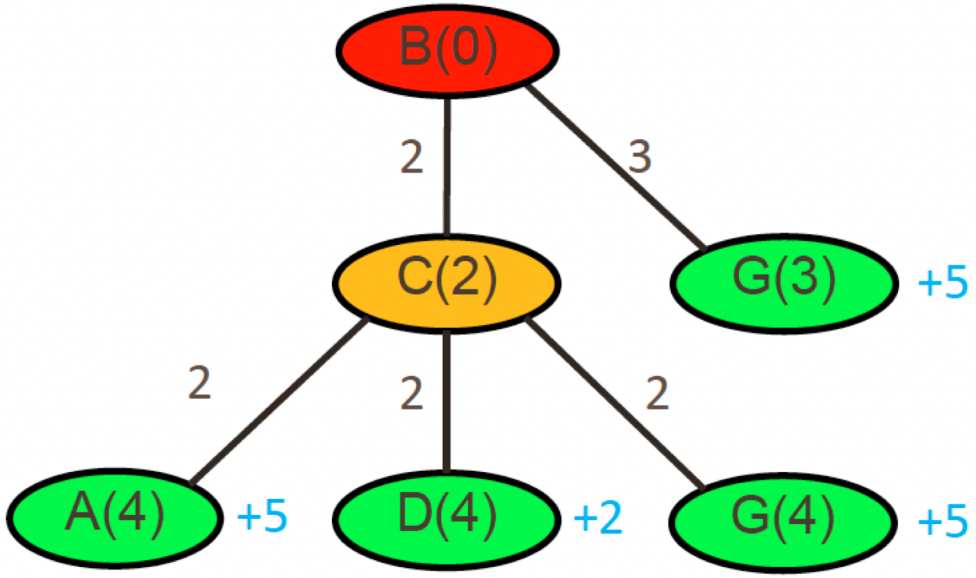
A*

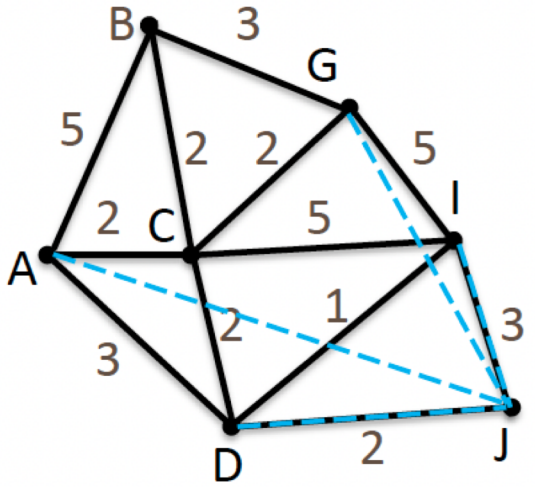




Current: C(2)
Queue: A(4+5), G(3+5), D(4+2)
Visited: B(0)

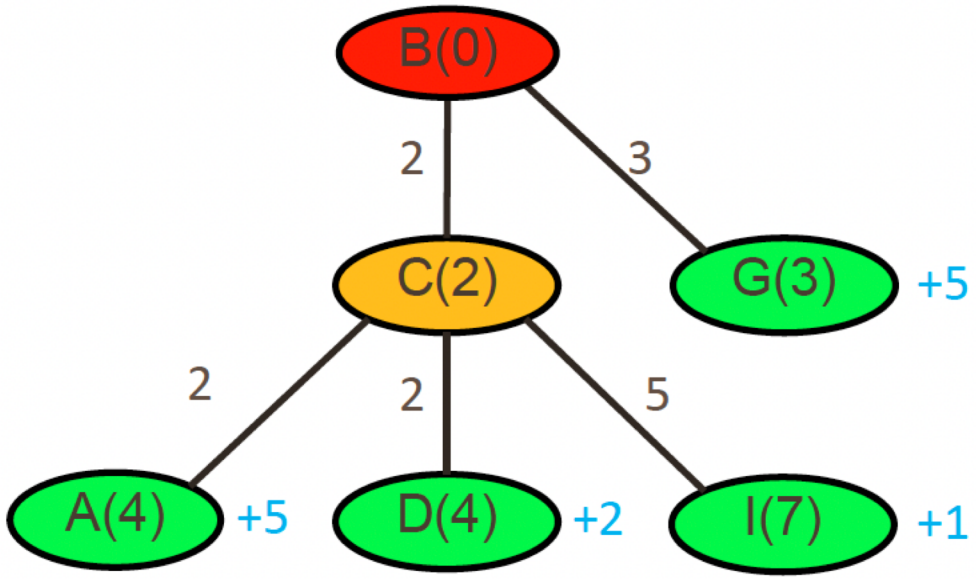
A*

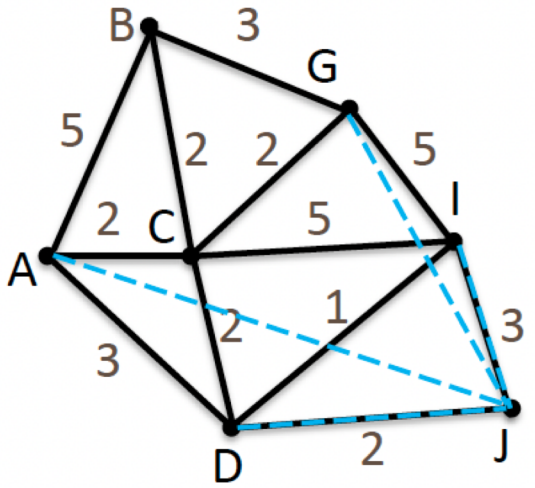




Current: C(2)
Queue: A(4+5), G(3+5), D(4+2)
Visited: B(0)

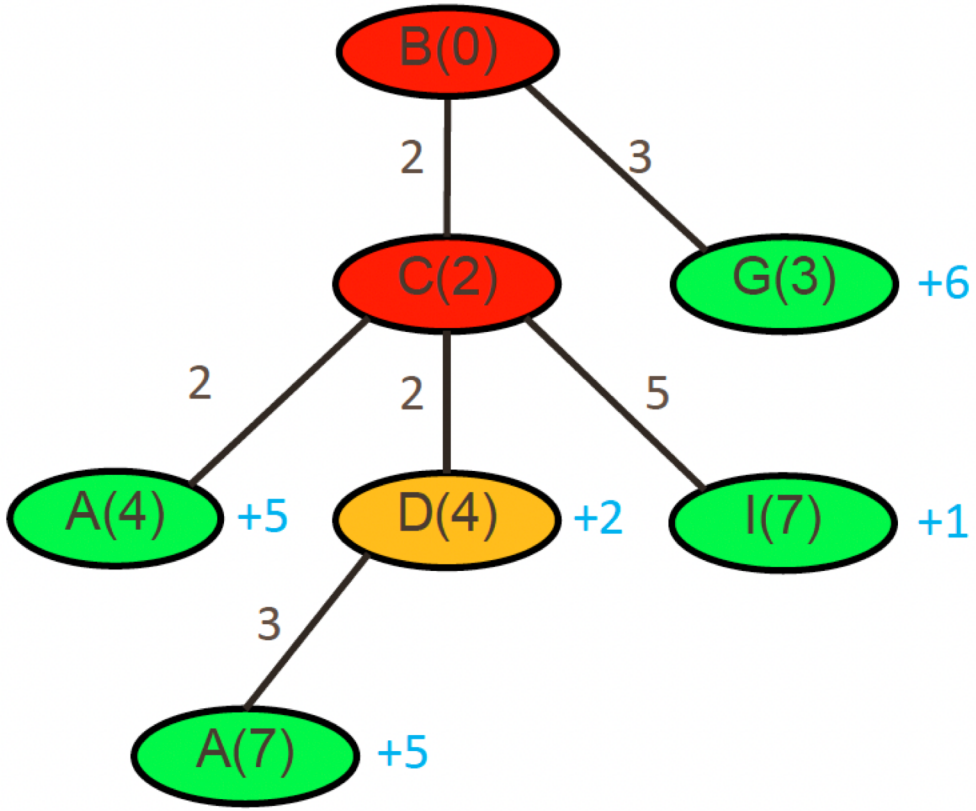
A*

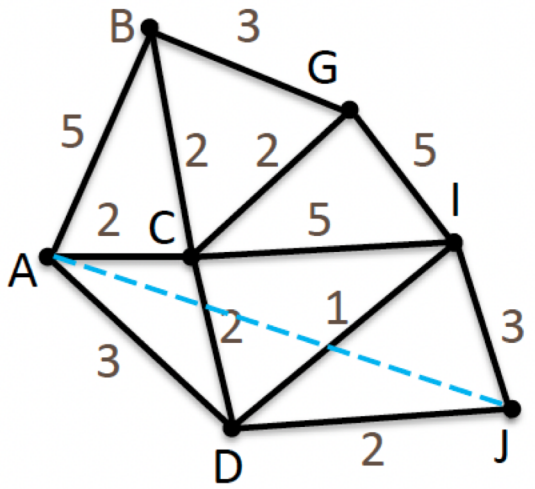




Current: C(2)
Queue: A(4+5), G(3+5), D(4+2), I(7+1)
Visited: B(0)

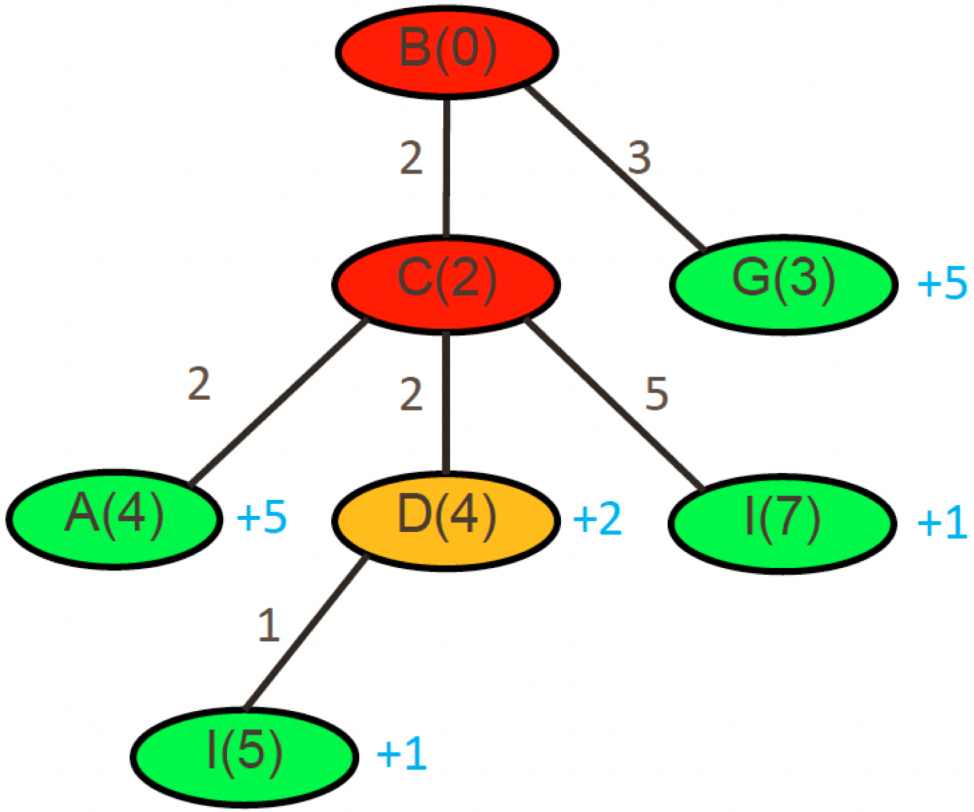
A*

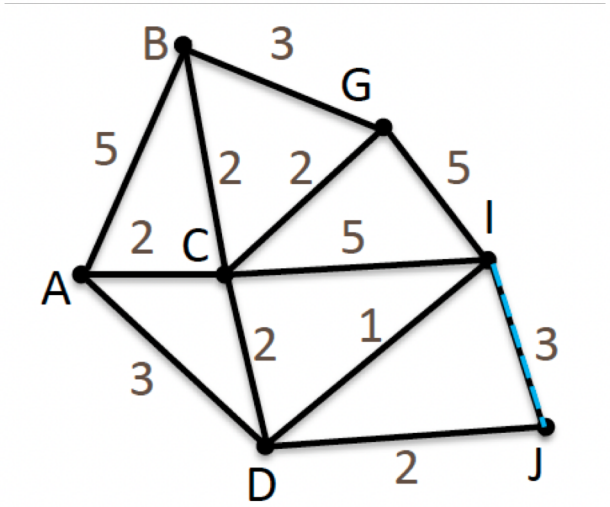




Current: D(4)
Queue: A(4+5), G(3+5), I(7+1)
Visited: B(0), C(2)

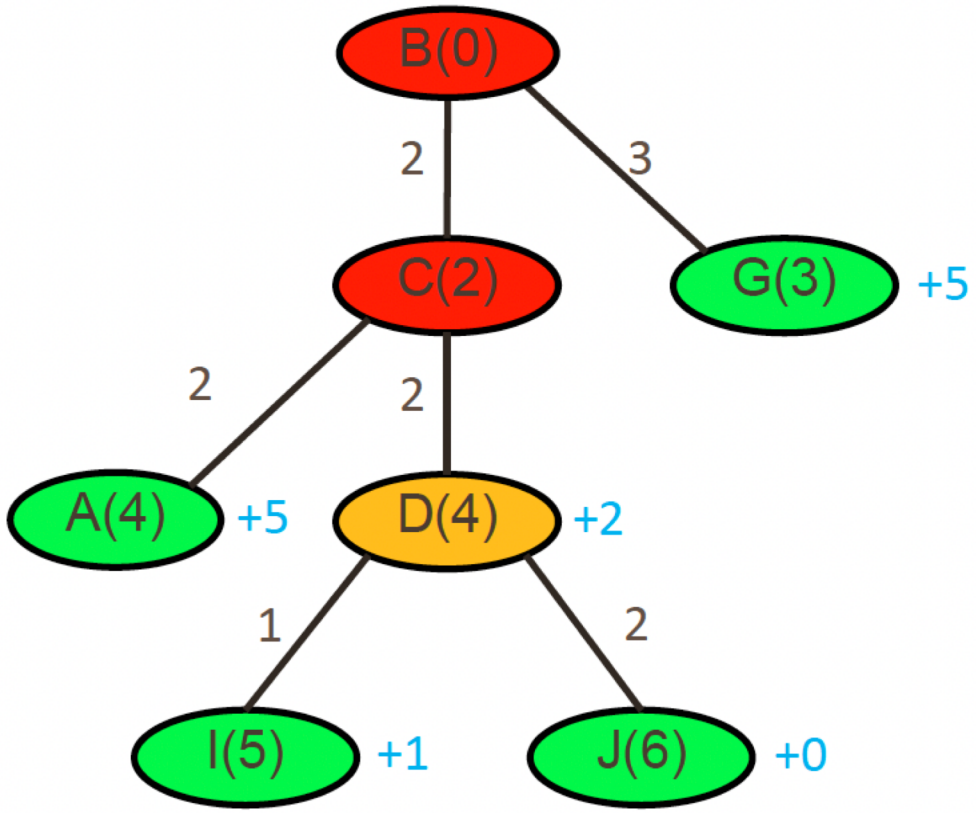
A*

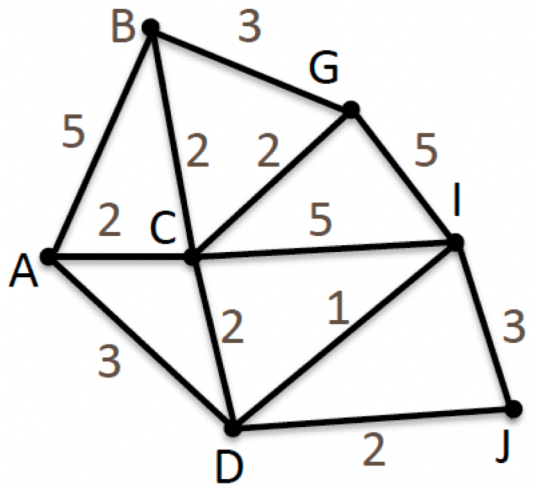




Current: D(4)
Queue: A(4+5), G(3+5), I(5+1)
Visited: B(0), C(2)

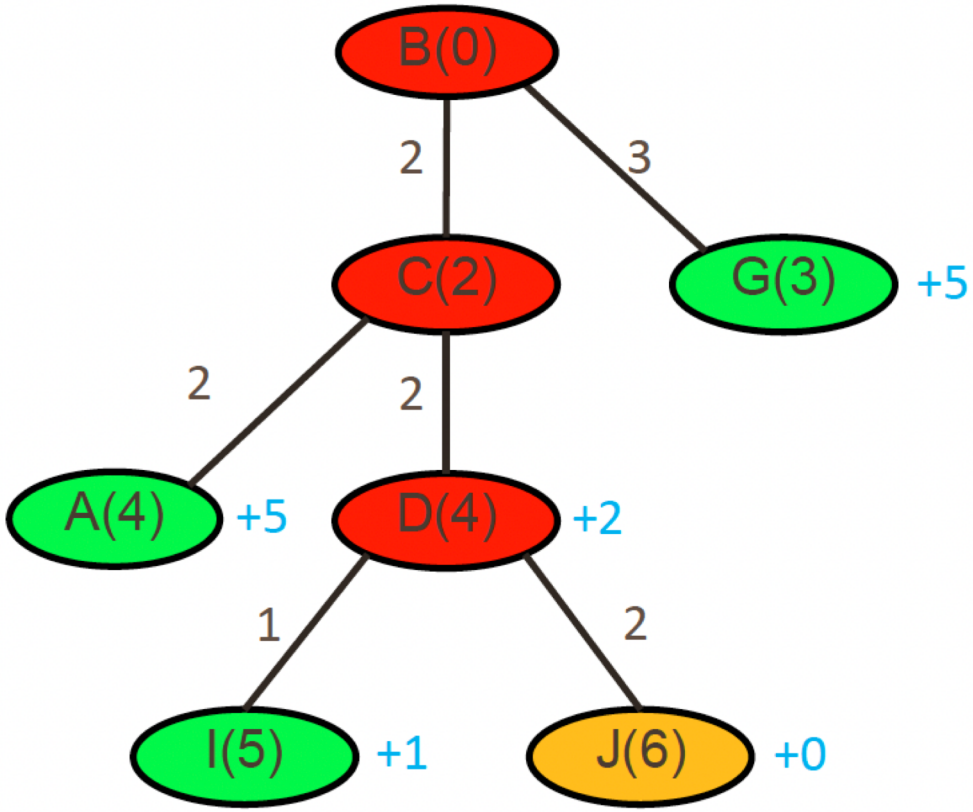
A*



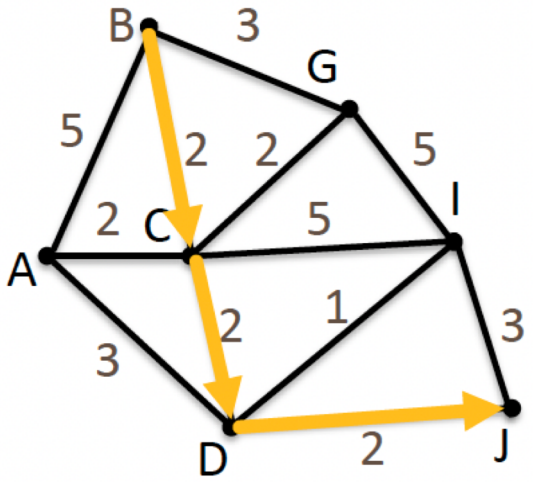


Current: D(4)
Queue: A(4+5), G(3+5), I(5+1), J(6+0)
Visited: B(0), C(2)

A*



Go from B to J: B -> C -> D -> J
Cost: 6



Current: J(6)
Queue: A(4+5), G(3+5), I(5+1)
Visited: B(0), C(2), D(4)

A*

- Dijkstra's algorithm is a type of BFS that accommodates edge weights.
- A* is an extension of Dijkstra's algorithm.
- It achieves faster performance by using heuristics
- Commonly used in robot planning
- It may not be always easy to find a proper heuristic function.

Summary

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