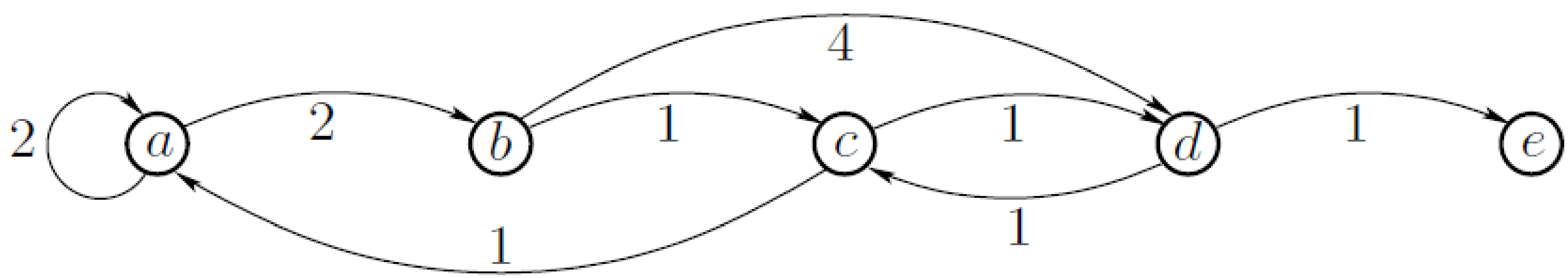
Sistemi Intelligenti
Corso di Laurea in Informatica, A.A. 2019-2020
Università degli Studi di Milan

Search algorithms for planning

Nicola Basilico
Dipartimento di Informatica
Via Celoria 18- 20133 Milano (MI)
Ufficio 4008
nicola.basilico@unimi.it
+39 02.503.16289

State-based problem formulation

- (Single agent: the automated problem solver)
- State space defined as a set of **nodes**, each node represents a state; we assume a finite state space
- For each state, we have set of actions that can be undertaken by the agent from that state
- Transition model: given a starting state and an action, indicates an arrival state (here we assume no uncertainties, i.e., deterministic transitions and full observability)
- Action costs: any transition has a cost, which we assume to be greater than a positive constant (reasonable assumption, useful for deriving some properties of the algorithms we discuss)
- Initial state



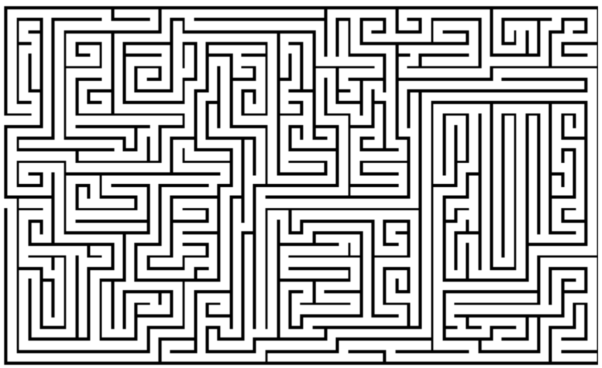
Compact representation: state transition graph G=(V,E)
(We will use "state" and "node" as interchangeable terms)

Formally describing the desired solution

- In the problem formulation we need to formally describe the features of the solution we seek

- Two (three) classes of problems:

feasibility

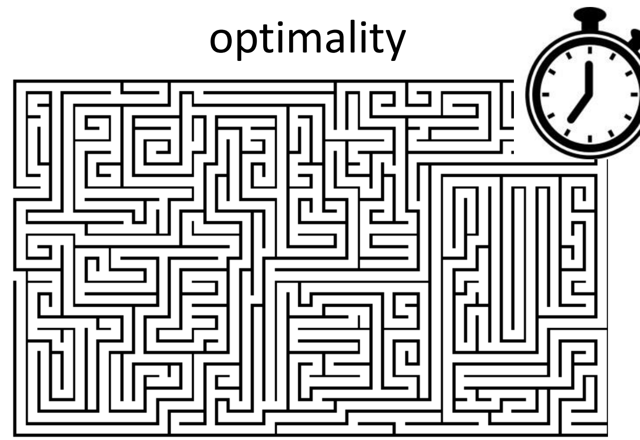


is there a path to an exit?

Set of goal states, find any sequence of actions (path) from the initial state to a goal state

(approximation)

optimality



If at least a path to an exit exists, what is the one with the minimum number of turns?

Set of goal states, find the sequence of actions (path) from the initial state to a goal state that has the minimum cost

Problem example

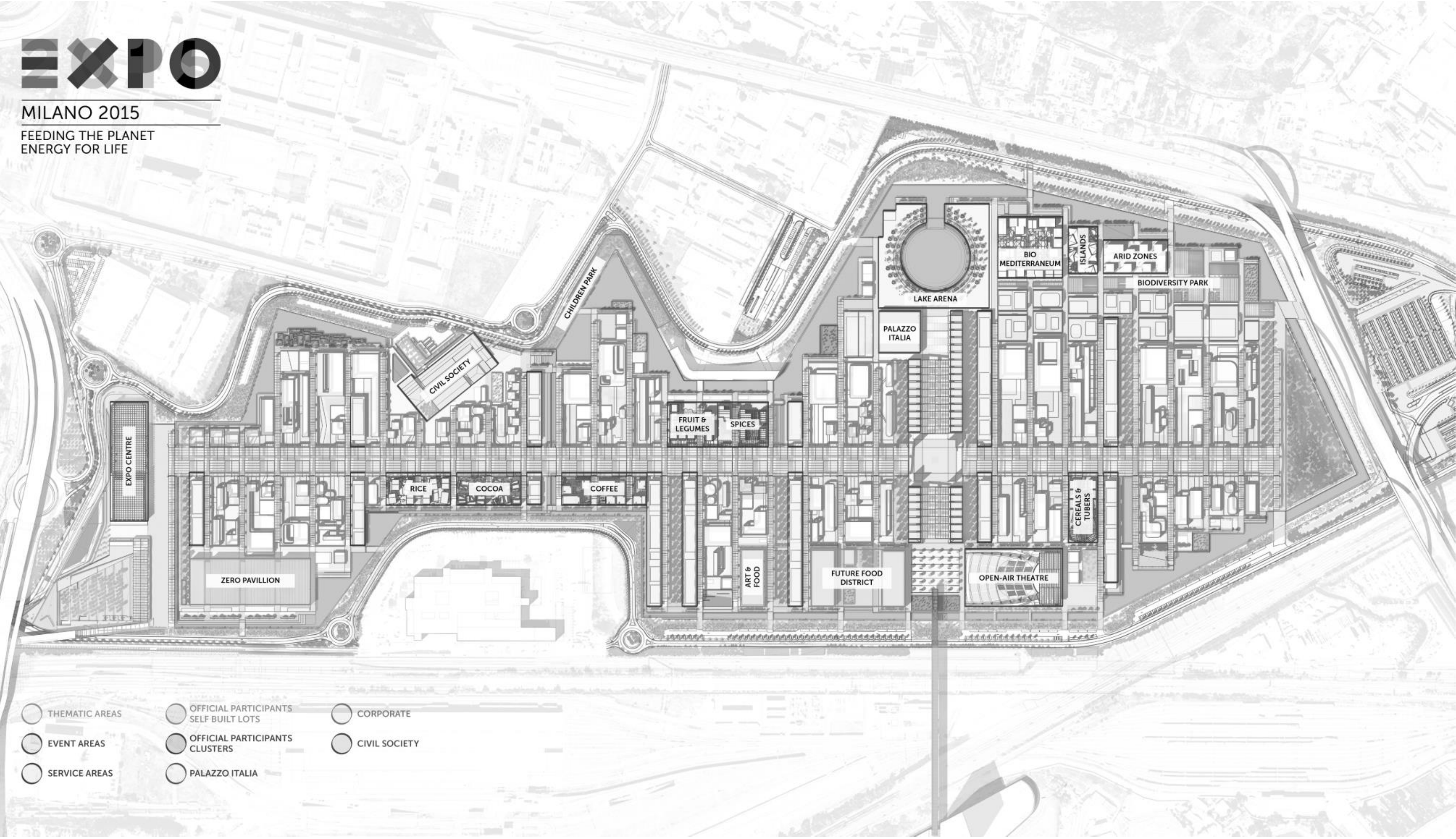
Consider a mobile robot moving on a graph-represented environment:

- **States**: nodes of the graph, they represent physical locations
- **Edges**: represent connections between nearby locations or, equivalently, movement actions
- **Initial state**: some starting location for the robot

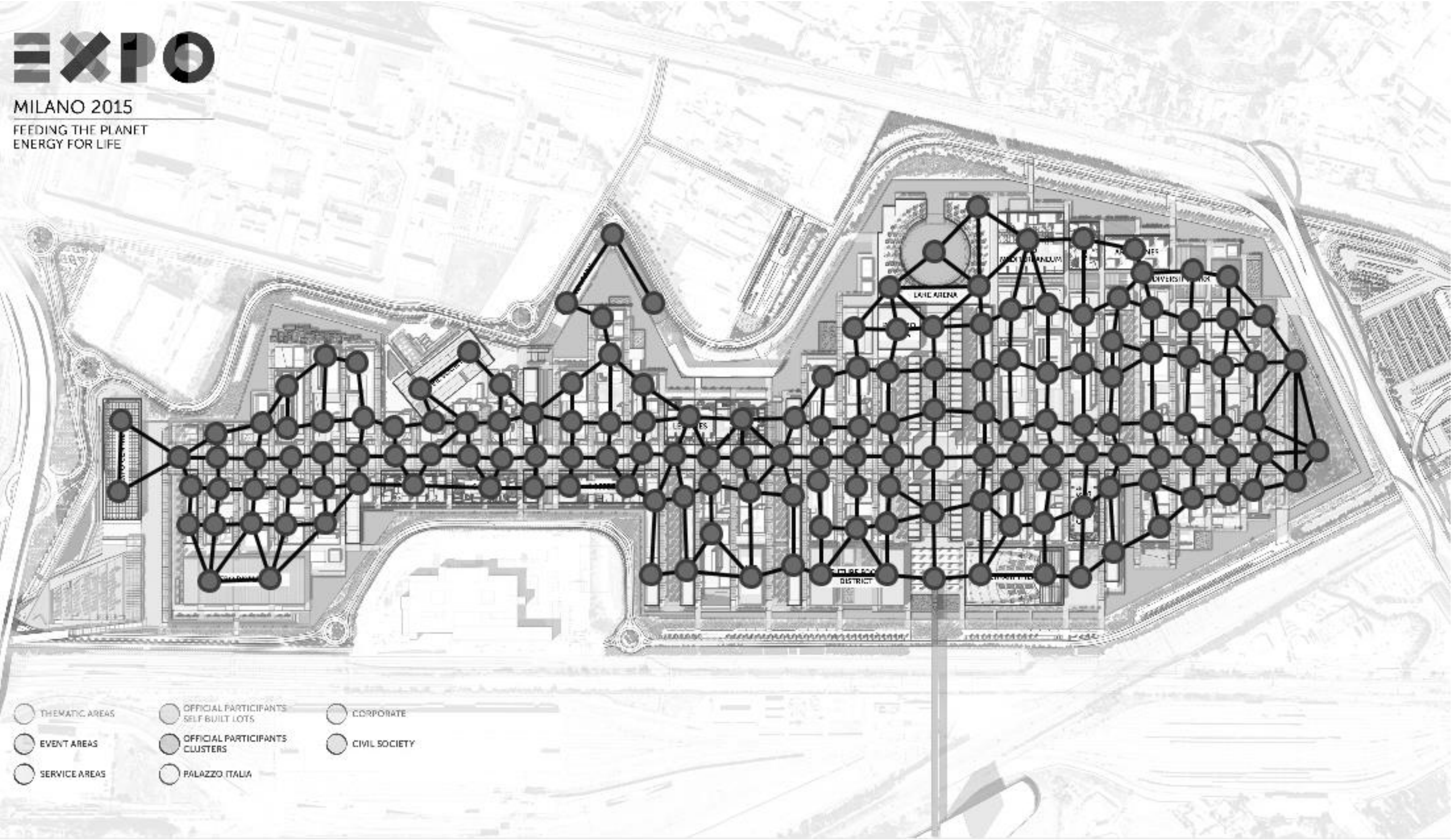
Desired solution:

- **Goal state(s)**: some location(s) to reach, e.g., recharging station, parking depot…
- Find a path to the initial location to a goal one

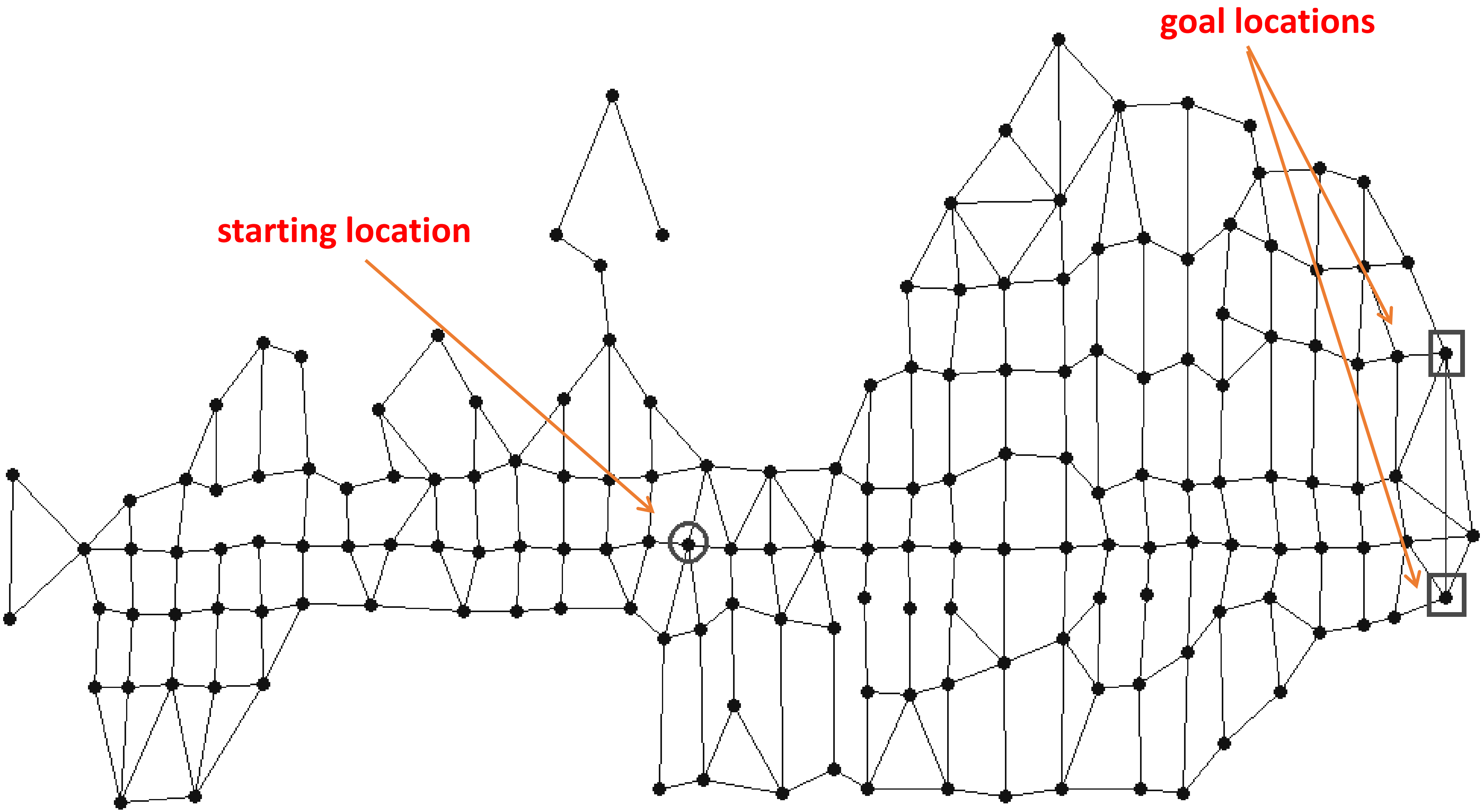
Problem example



Problem example

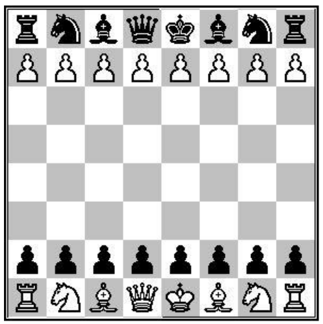


Problem example



Problem specification

- How to **specify** a planning problem?
- First approach: provide the full state transition graph G (as in the previous example)
- Most of the times this is not an affordable option due to the combinatorial nature of the state space:



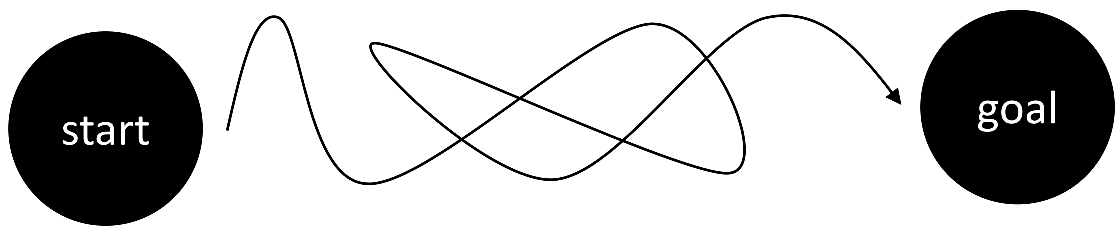
- **Chess board**: approx. 10^{47} states
- We can specify the initial state and the transition function in some compact form (e.g., set of rules to generate next states)
- The planning problem "unfolds" as search progresses

- We need an efficient procedure for *goal checking*

General features of search algorithms

A search algorithm explores the state-transition graph graph G until it discovers the desired solution

- In feasibility: when a goal node is visited the path that led to that node is returned
- In optimality: when a goal node is visited, if any other possible path to that node has higher cost the path that led to that node is returned



It does not suffice to visit a goal node, the algorithm has to reconstruct the path it followed to get there: it must keep a trace of its search

Such a trace can be mapped to a subgraph of G, it is called *search graph*

how to evaluate a (search) algorithm?

- We can evaluate a search algorithm along different dimensions
  - Sound?
  - Complete? (Systematic?)
  - Space complexity?
  - Time complexity?

(The above criteria can actually be used to evaluate a broader class of algorithms)

Soundness

- If the algorithm returns a solution, is it compliant with the desired features specified in the problem formulation?

- Example:
  - In feasibility: *does the returned solution lead to a goal?*
  - In optimality: *does the returned solution lead to a goal with minimum cost?*

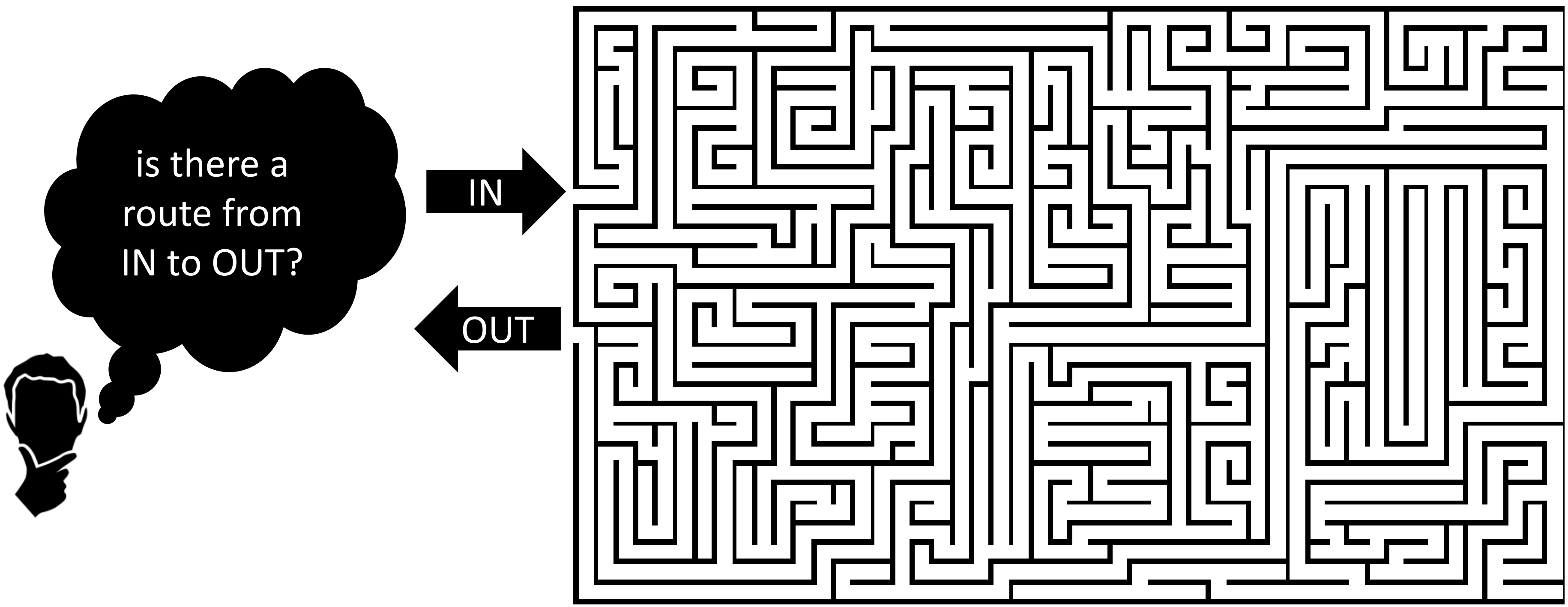
Completeness and the systematic property

- If a solution exists, does the algorithm find it?
- Example:
  - In feasibility: does it always find a path to the goal when it exists?
  - In optimality: does it always find the path to the goal that has minimum cost when at least one exists?

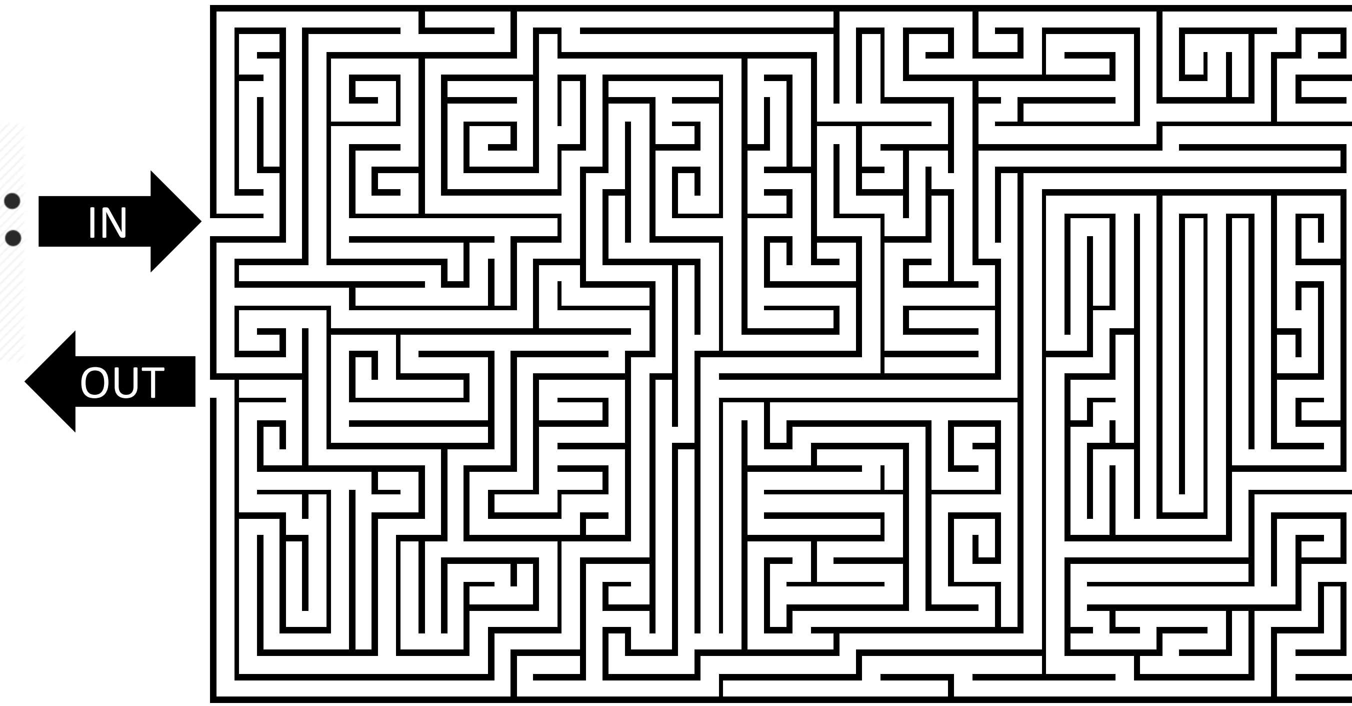
- Typically shown by proving that the search will/will not visit all states if given enough time

- If the state space is infinite, we can ask if the search is systematic:
  - if the answer is “yes” the algorithm must terminate
  - if the answer is “no”, it’s ok if it does not terminate but …
  - … all reachable states must be visited in the limit: as time goes to infinity, all states are visited (this definition is sound under the assumption of countable state space)

Visual example



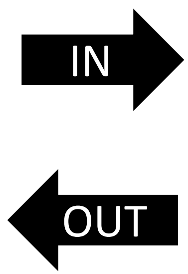
Visual example

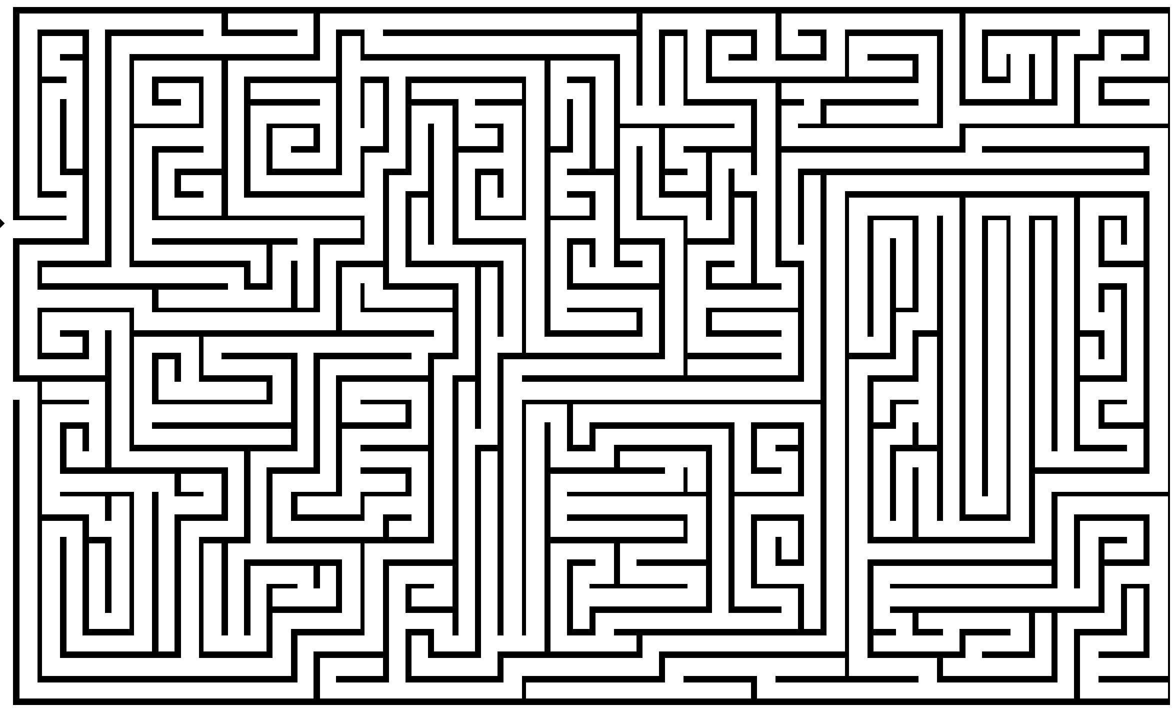


Complete / Systematic

- Searching along **multiple** trajectories (either concurrently or not), eventually covers all the reachable space

Visual example



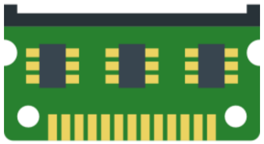


Not complete / Not systematic

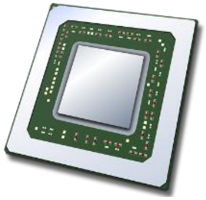
- Searching along a **single** trajectory, eventually gets stuck in a dead end

Space and time complexity

- Space complexity: how does the amount of memory required by the search algorithm grows as a function of the problem's dimension (worst case)?



- Time complexity: how does the time required by the search algorithm grows as a function of the problem's dimension (worst case)?



- Asymptotic trend:
  - We measure complexity with a function $f(n)$ of the input size
  - For analysis purposes, the "Big O" notation is convenient:

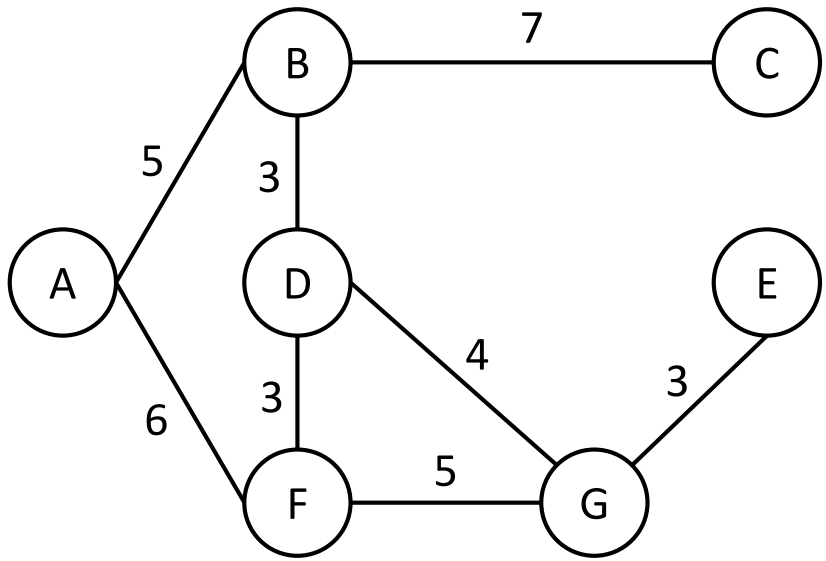
$$\text{A function } f(n) \text{ is } O(g(n)) \text{ if } \exists k > 0, n_0 \text{ such that } f(n) \leq kg(n) \text{ for } n > n_0$$

- An algorithm that is $O(n^2)$ is better than one that is $O(n^5)$
- If $g(n)$ is an exponential, the algorithm is not efficient

Running example

- To present the various search algorithms, we will use this *problem instance* as our running example

State-transition graph:



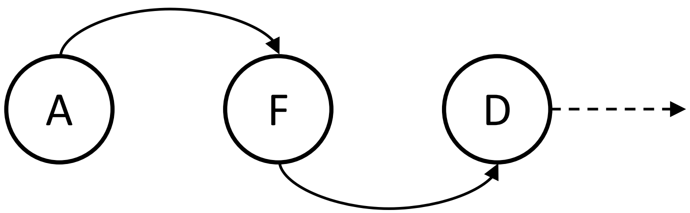
Initial state: A

Desired solution: any path to goal state E

- It might be useful to think it as a map, but keep in mind that this interpretation does not hold for every instance

Search algorithm definition

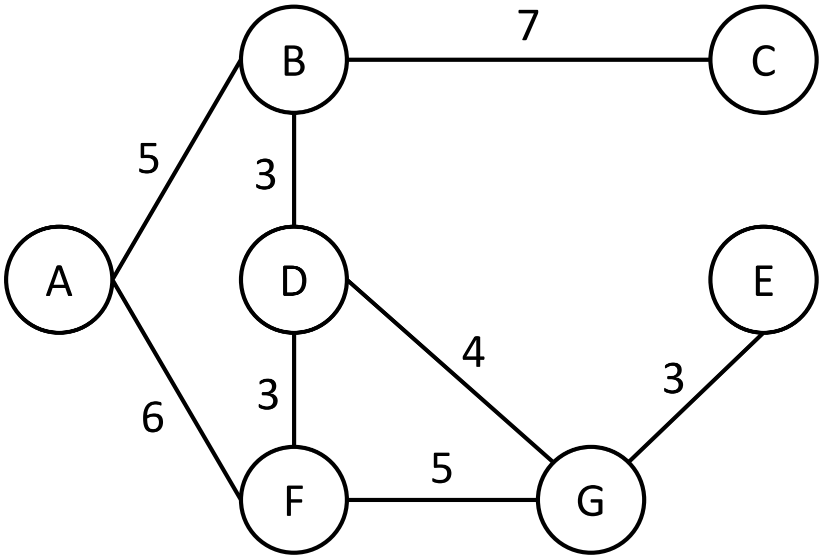
- The different search algorithms are substantially characterized by the answer they provide to the following question:



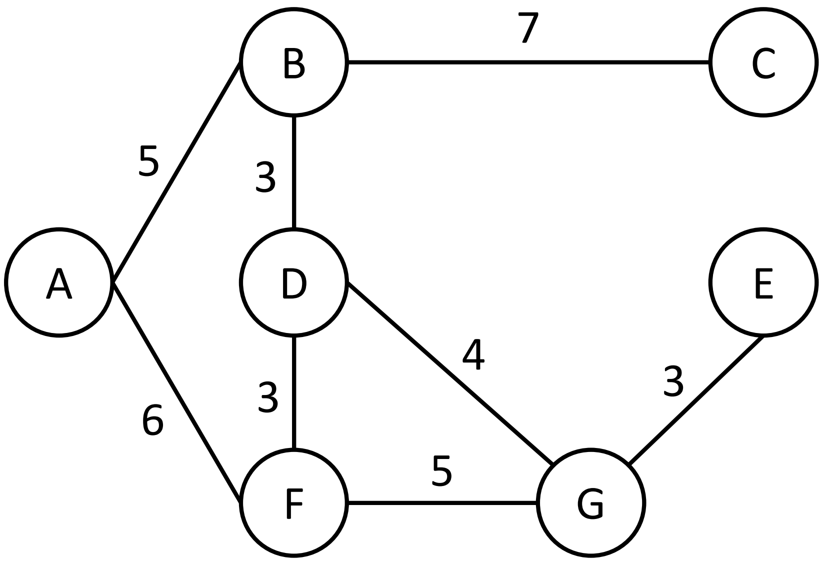
Given what I searched so far,
where to search next?

- The answer is encoded in a set of rules that drives the search and define its type, let's start with the simplest one

Depth-First Search (DFS)

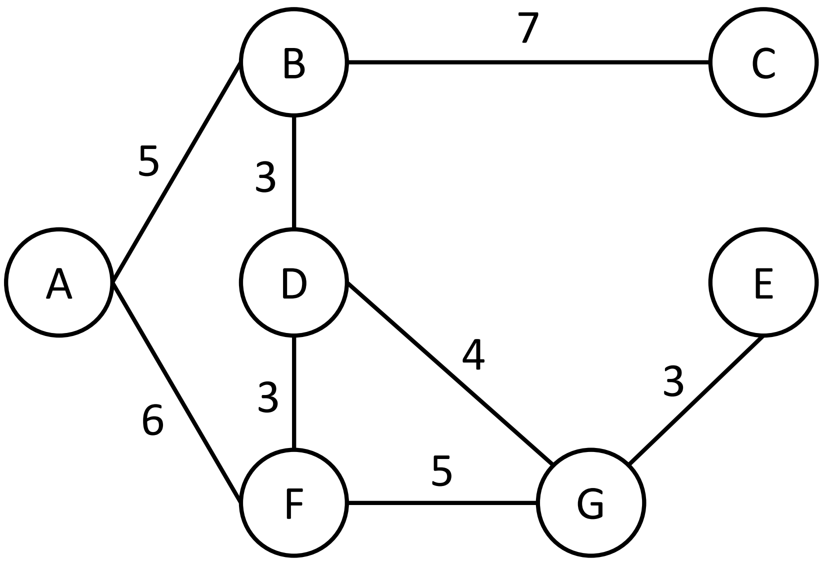


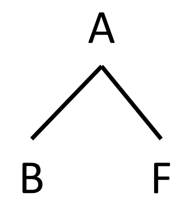
Depth-First Search (DFS)



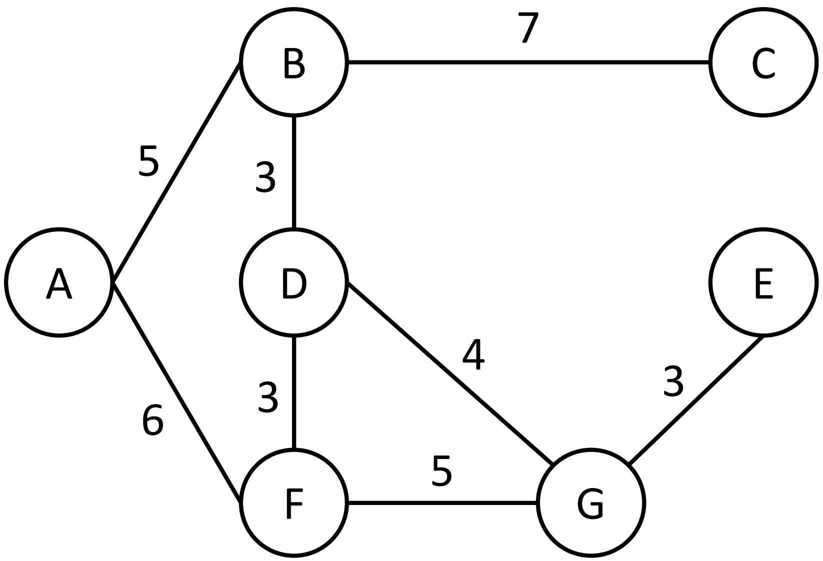
A

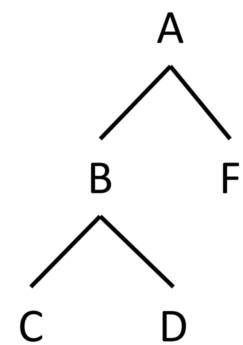
Depth-First Search (DFS)



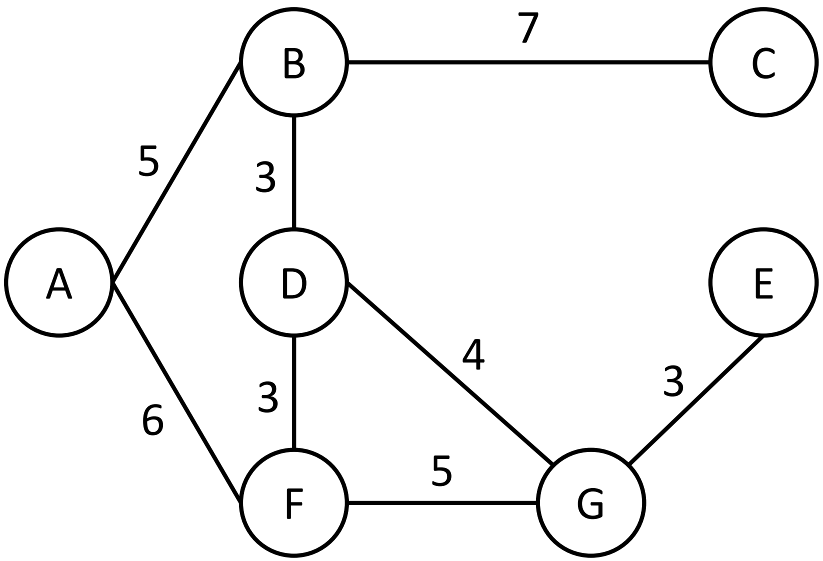


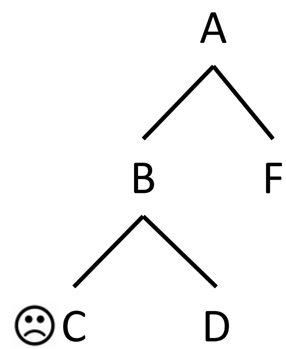
Depth-First Search (DFS)



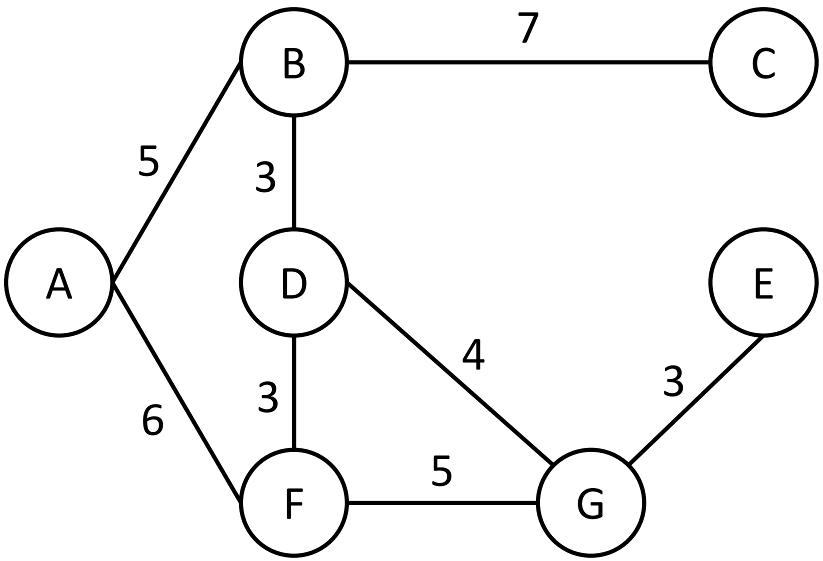


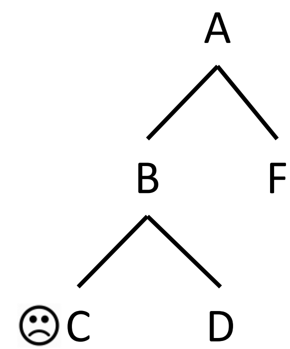
Depth-First Search (DFS)





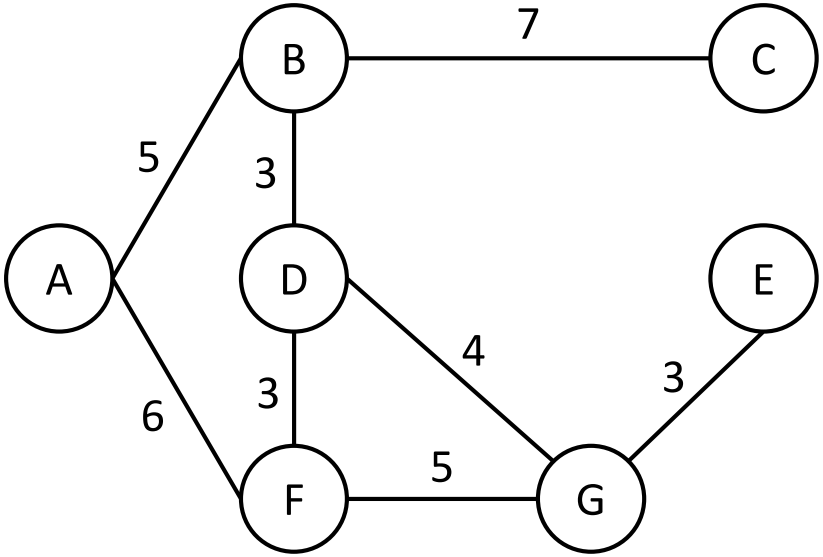
Depth-First Search (DFS)

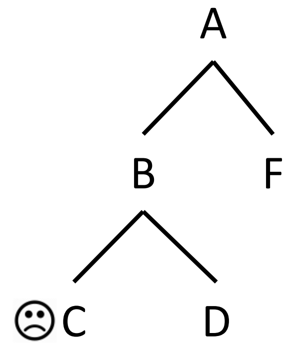




- A Depth-First Search (DFS) chooses the deepest node in the search tree (How to break ties? For now lexicographic order)
- We are **avoiding loops** on the same branch (loops are redundant paths)
- A dead end stopped the search, DFS seems not complete. Can we fix this?
- Let's endow our DFS with **backtracking**: a way to reconsider previously evaluated decisions

Depth-First Search (DFS)

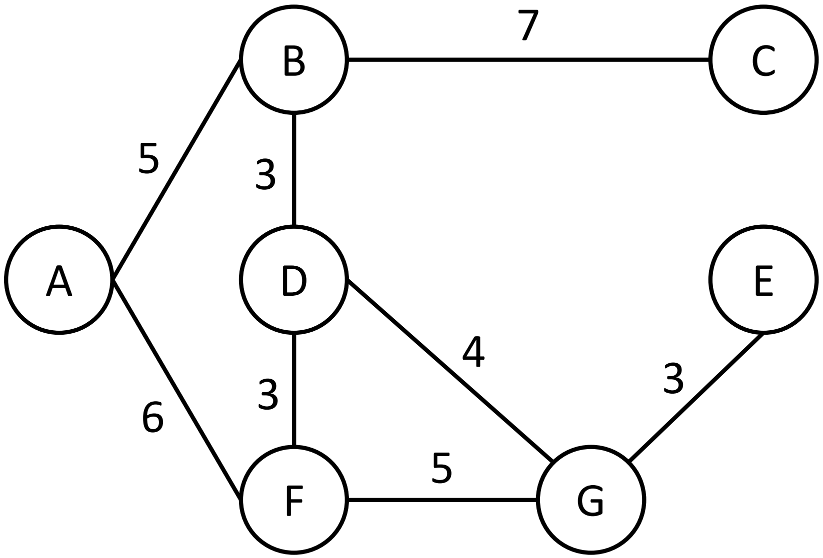


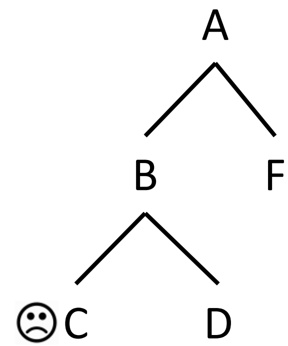


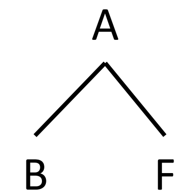
A

- A Depth-First Search (DFS) chooses the deepest node in the search tree (How to break ties? For now lexicographic order)
- We are **avoiding loops** on the same branch (loops are redundant paths)
- A dead end stopped the search, DFS seems not complete. Can we fix this?
- Let’s endow our DFS with **backtracking**: a way to reconsider previously evaluated decisions

Depth-First Search (DFS)

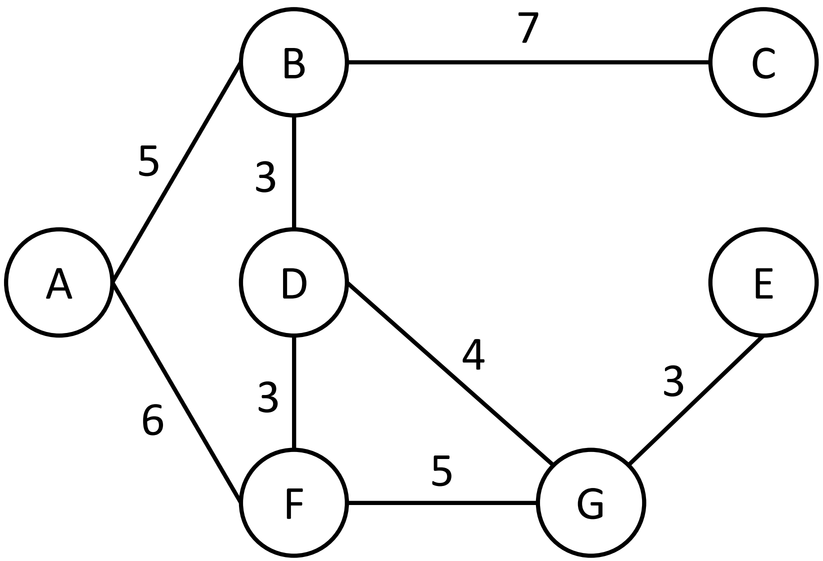


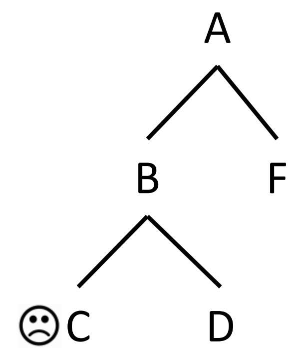


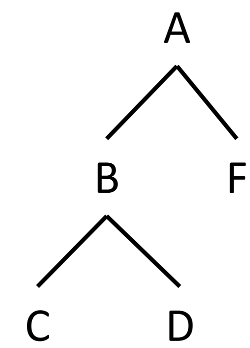


- A Depth-First Search (DFS) chooses the deepest node in the search tree (How to break ties? For now lexicographic order)
- We are **avoiding loops** on the same branch (loops are redundant paths)
- A dead end stopped the search, DFS seems not complete. Can we fix this?
- Let's endow our DFS with **backtracking**: a way to reconsider previously evaluated decisions

Depth-First Search (DFS)

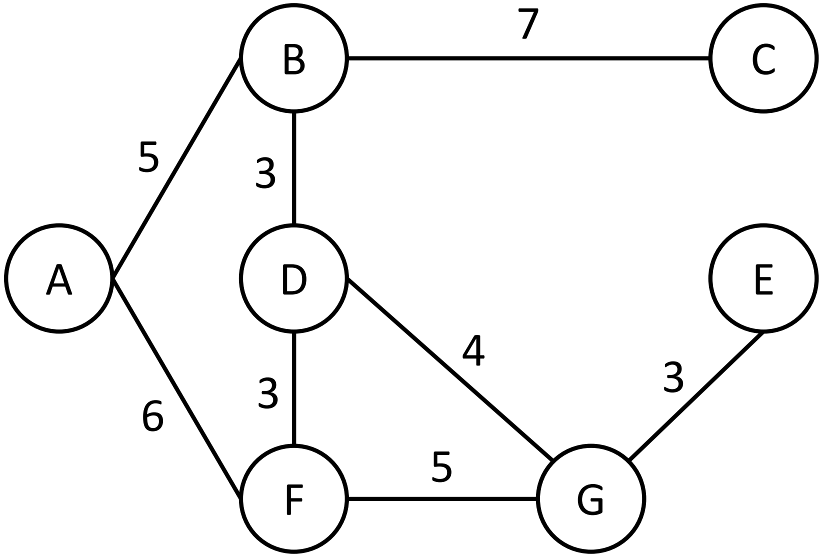


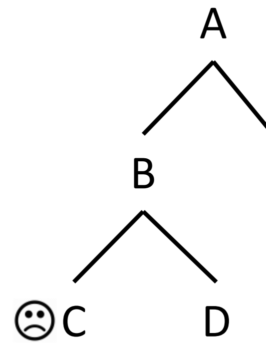


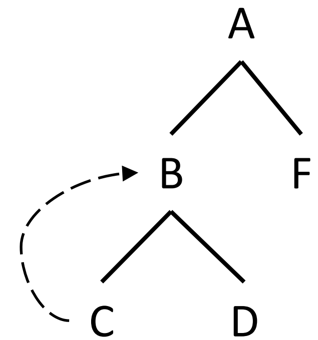


- A Depth-First Search (DFS) chooses the deepest node in the search tree (How to break ties? For now lexicographic order)
- We are **avoiding loops** on the same branch (loops are redundant paths)
- A dead end stopped the search, DFS seems not complete. Can we fix this?
- Let’s endow our DFS with **backtracking**: a way to reconsider previously evaluated decisions

Depth-First Search (DFS)







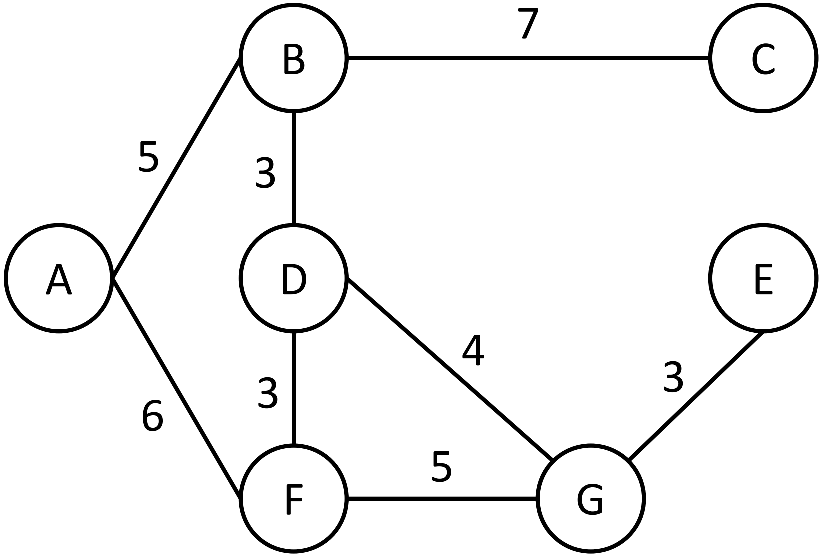
- A Depth-First Search (DFS) chooses the deepest node in the search tree (How to break ties? For now lexicographic order)

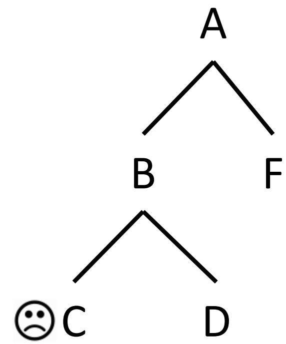
- We are **avoiding loops** on the same branch (loops are redundant paths)

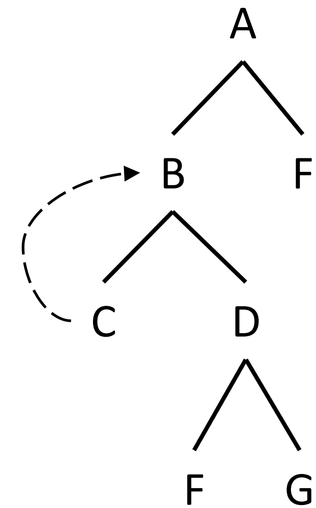
- A dead end stopped the search, DFS seems not complete. Can we fix this?

- Let’s endow our DFS with **backtracking**: a way to reconsider previously evaluated decisions

Depth-First Search (DFS)







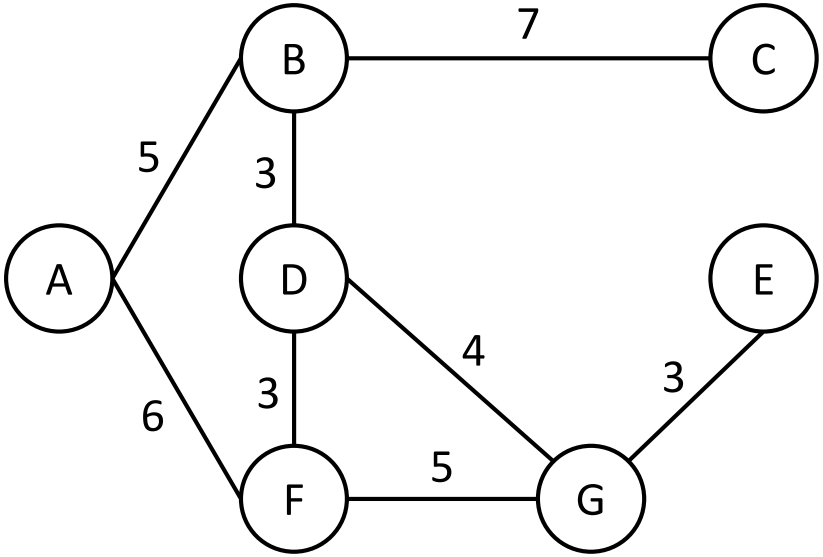
- A Depth-First Search (DFS) chooses the deepest node in the search tree (How to break ties? For now lexicographic order)

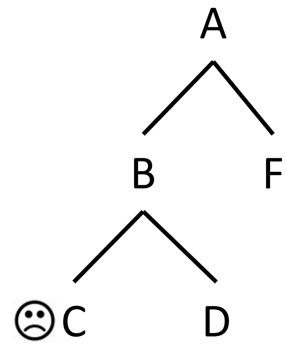
- We are **avoiding loops** on the same branch (loops are redundant paths)

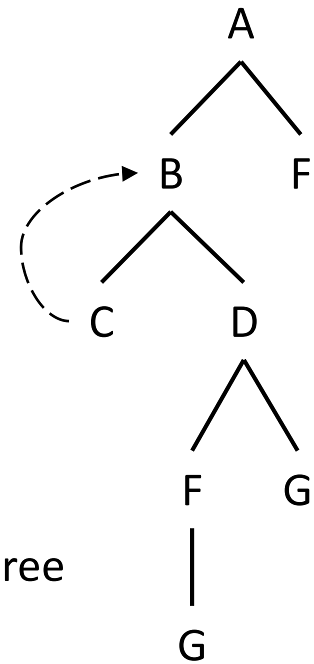
- A dead end stopped the search, DFS seems not complete. Can we fix this?

- Let's endow our DFS with **backtracking**: a way to reconsider previously evaluated decisions

Depth-First Search (DFS)







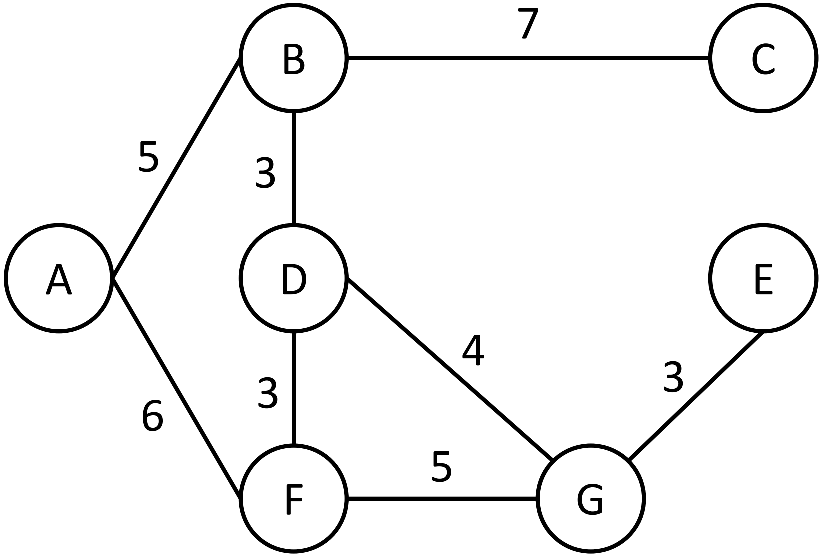
- A Depth-First Search (DFS) chooses the deepest node in the search tree (How to break ties? For now lexicographic order)

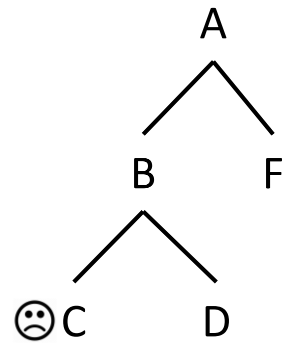
- We are **avoiding loops** on the same branch (loops are redundant paths)

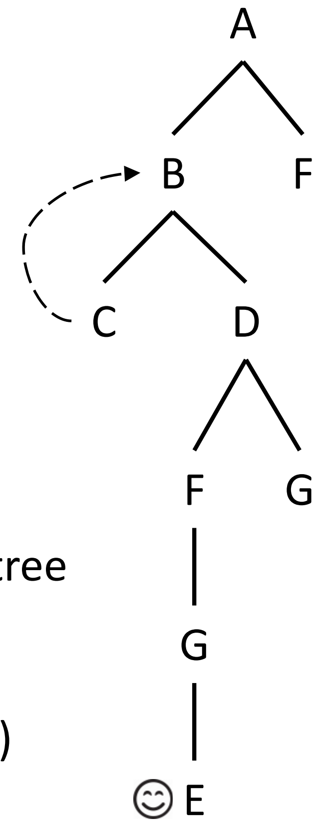
- A dead end stopped the search, DFS seems not complete. Can we fix this?

- Let’s endow our DFS with **backtracking**: a way to reconsider previously evaluated decisions

Depth-First Search (DFS)







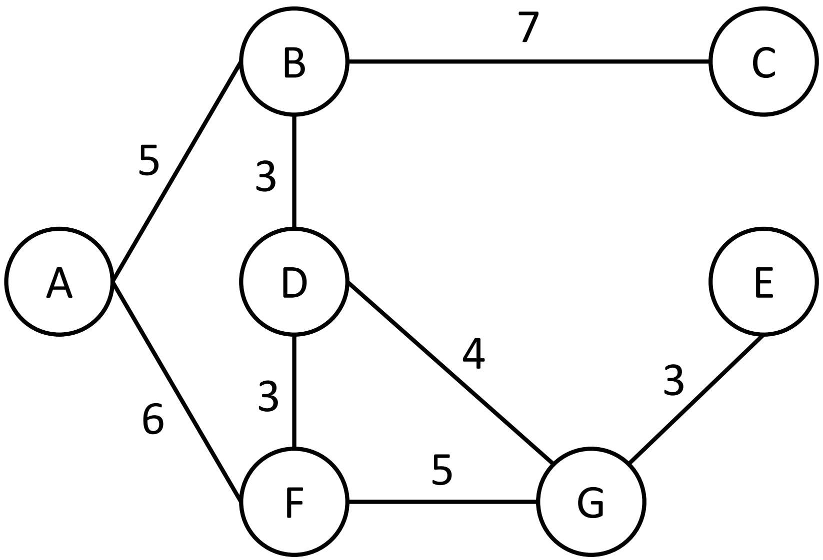
- A Depth-First Search (DFS) chooses the deepest node in the search tree (How to break ties? For now lexicographic order)
- We are **avoiding loops** on the same branch (loops are redundant paths)
- A dead end stopped the search, DFS seems not complete. Can we fix this?
- Let's endow our DFS with **backtracking**: a way to reconsider previously evaluated decisions

Solution: (A->B->D->F->G->E)

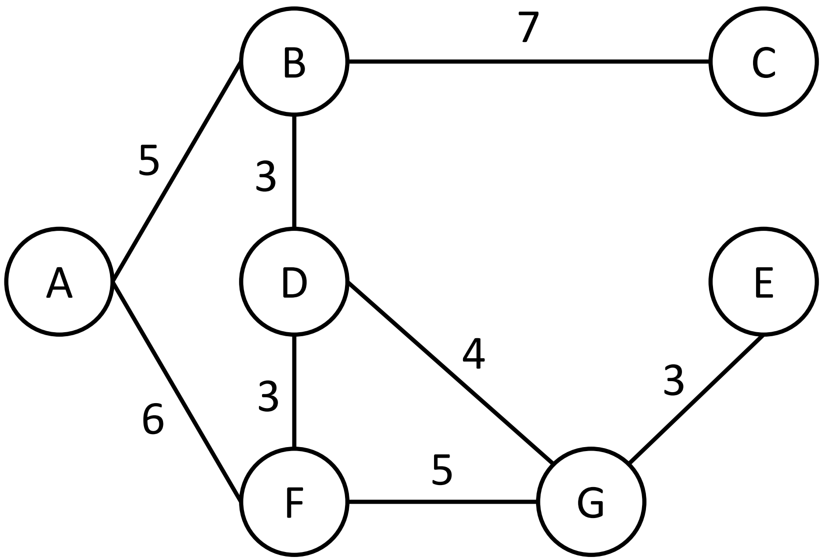
Depth-First Search (DFS)

- DFS with loops removal and BT is sound and complete
- Call b the maximum branching factor, i.e., the maximum number of actions available in a state
- Call d the maximum depth of a solution, i.e., the maximum number of actions in a path
- Space complexity: $O(d)$
- Time complexity: $1 + b + b^2 + \ldots + b^d = O(b^d)$

Breadth-First Search (BFS)

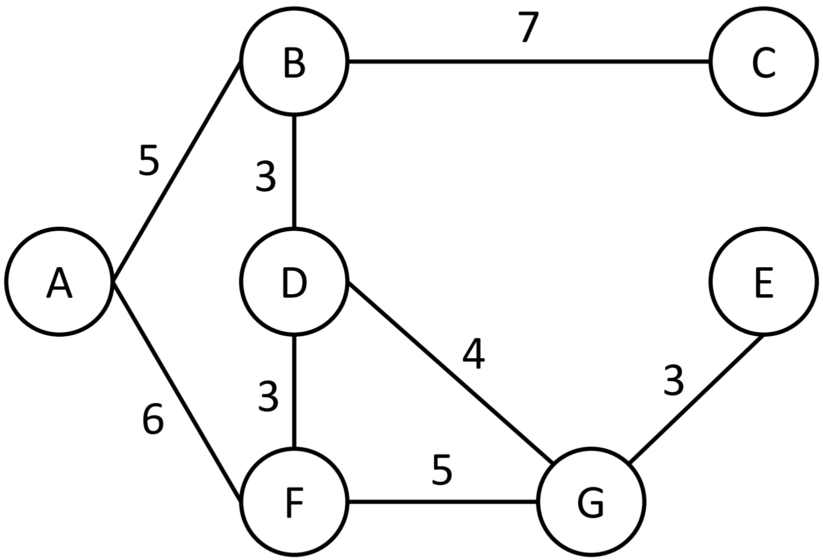


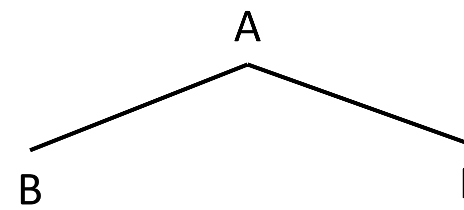
Breadth-First Search (BFS)



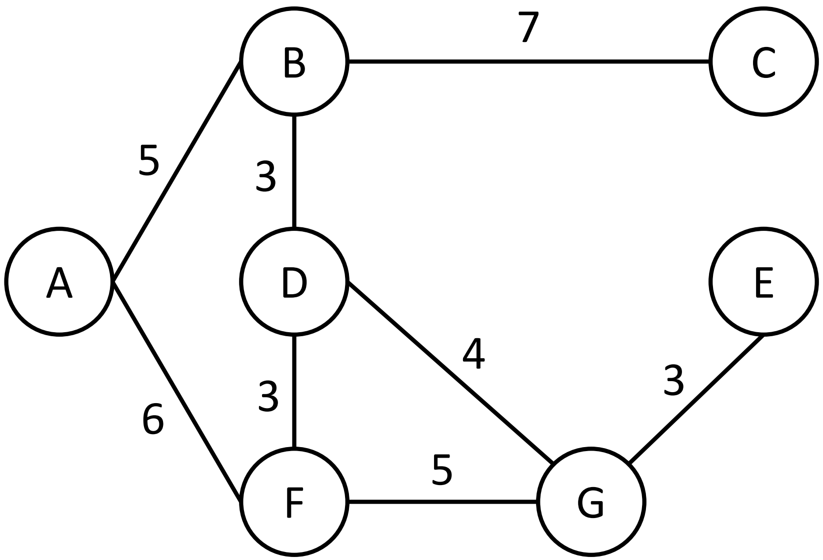
A

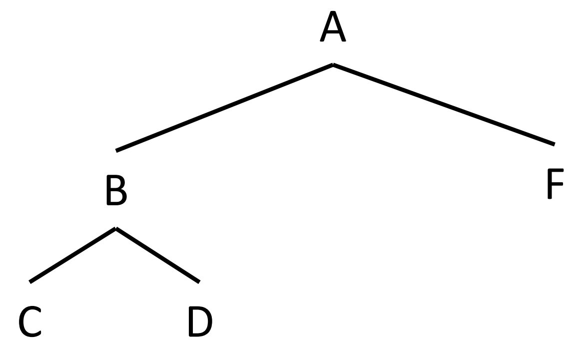
Breadth-First Search (BFS)



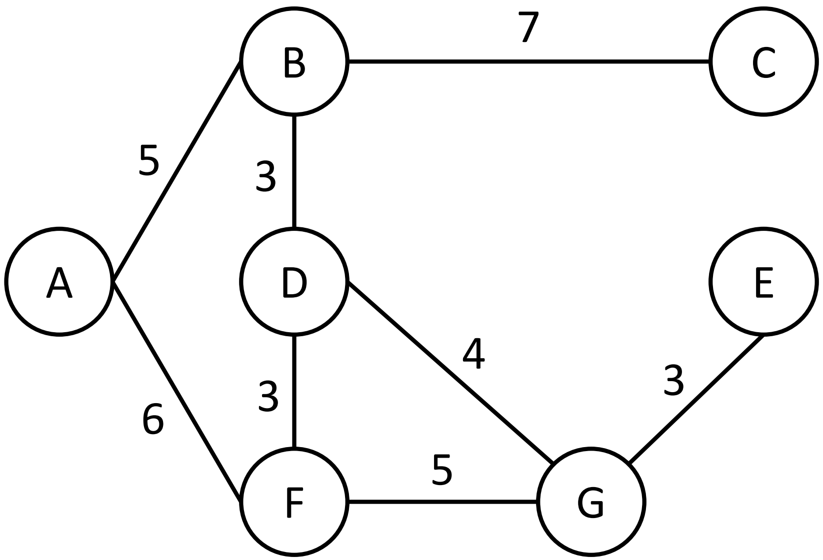


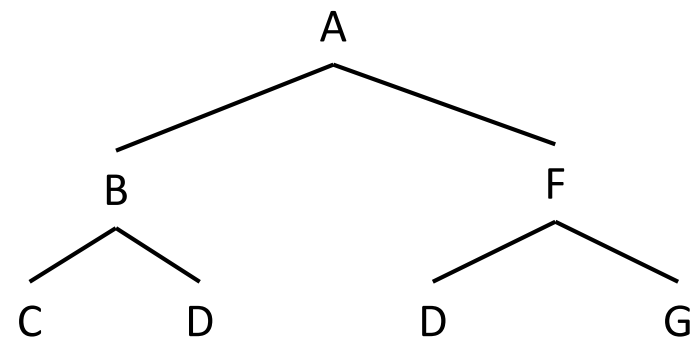
Breadth-First Search (BFS)



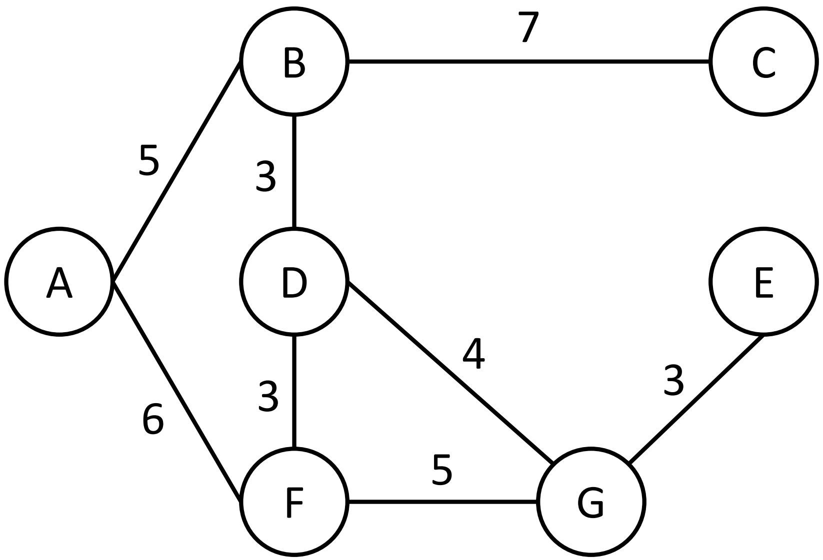


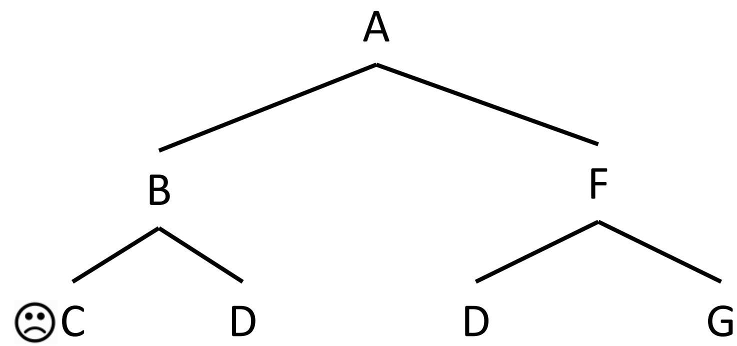
Breadth-First Search (BFS)



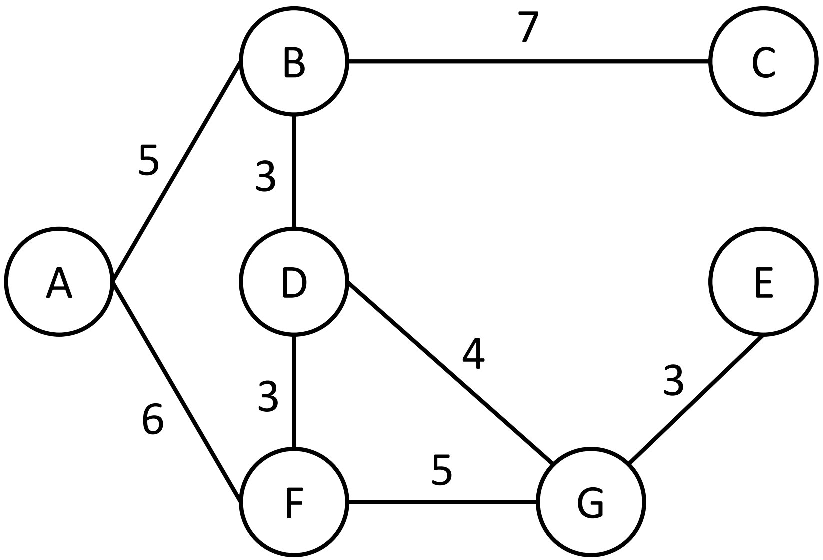


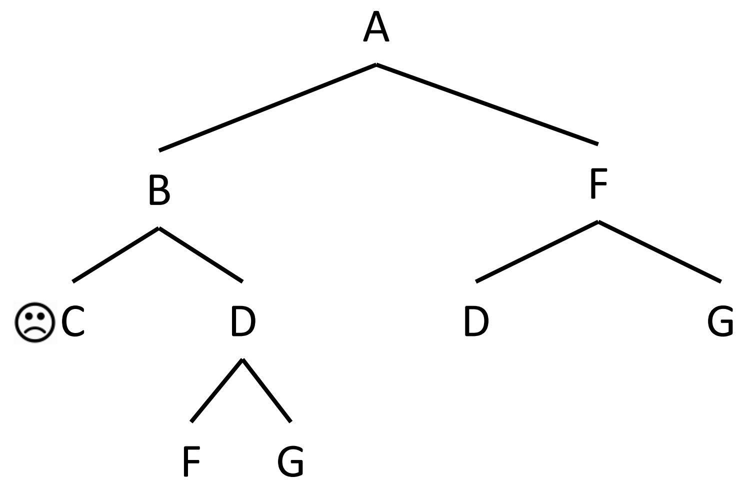
Breadth-First Search (BFS)



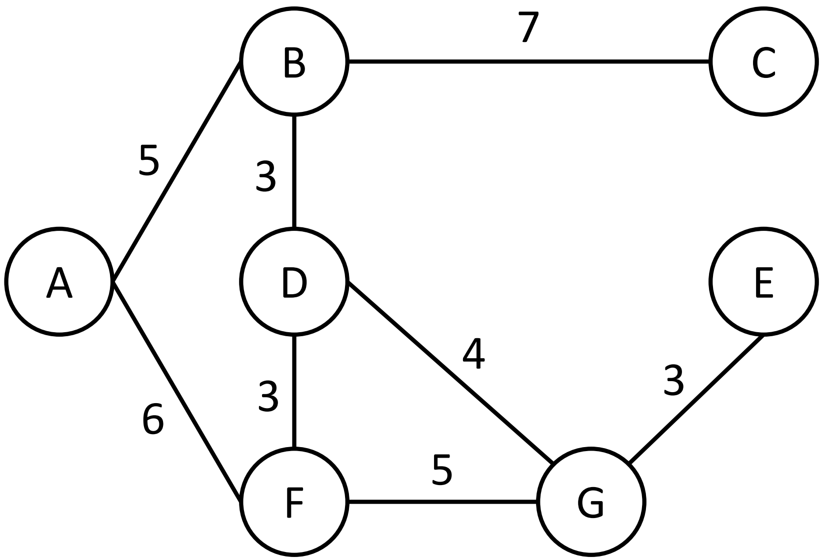


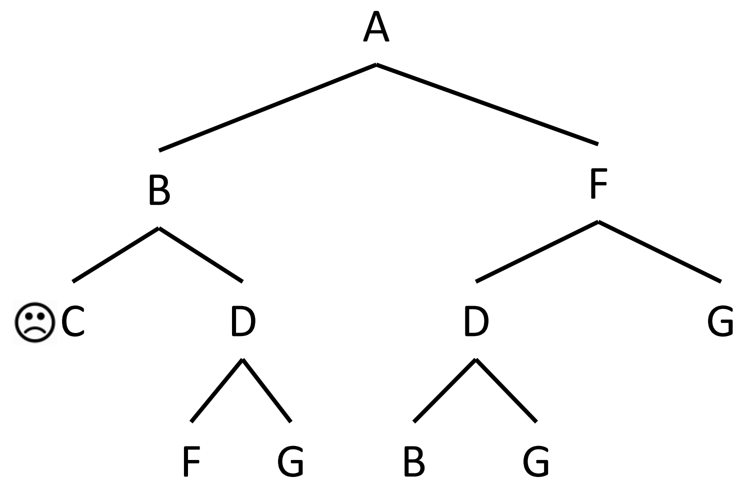
Breadth-First Search (BFS)



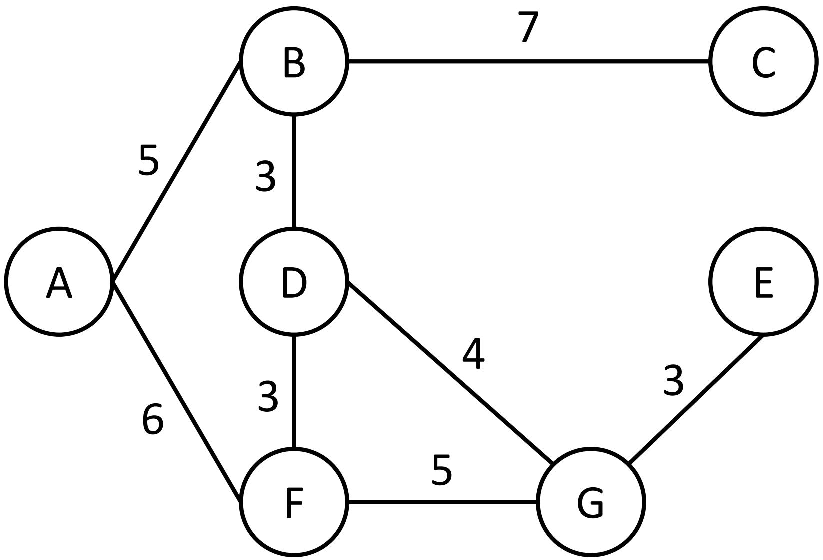


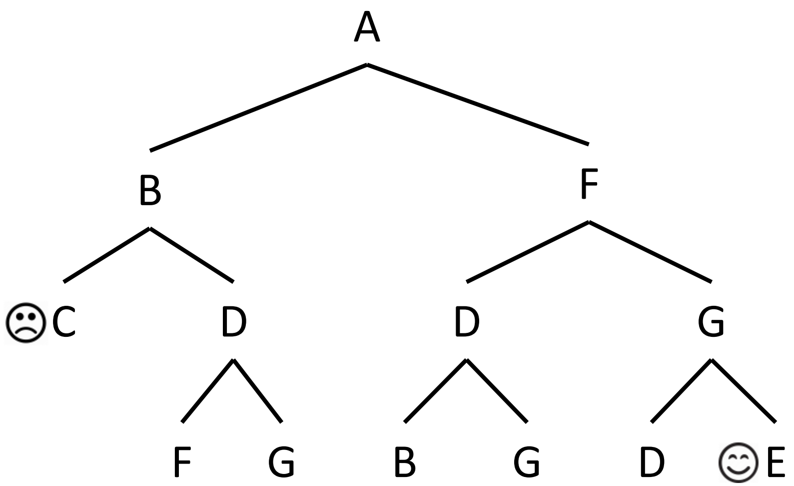
Breadth-First Search (BFS)





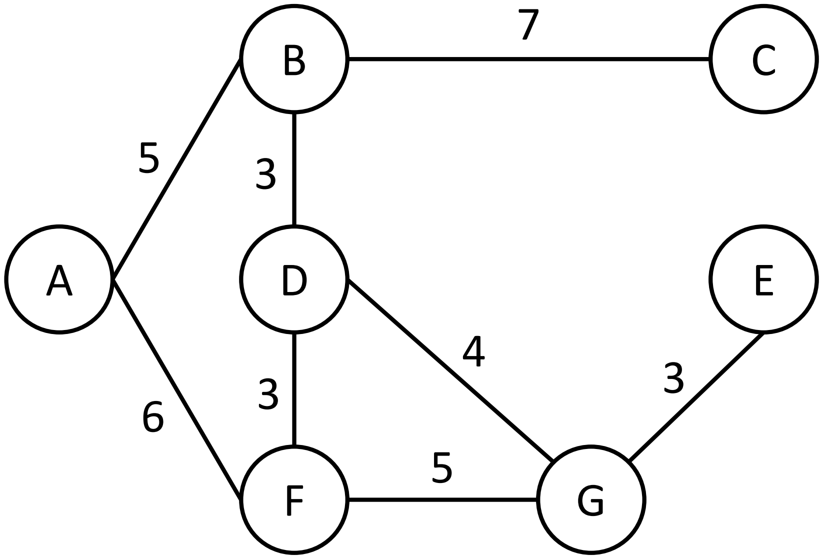
Breadth-First Search (BFS)

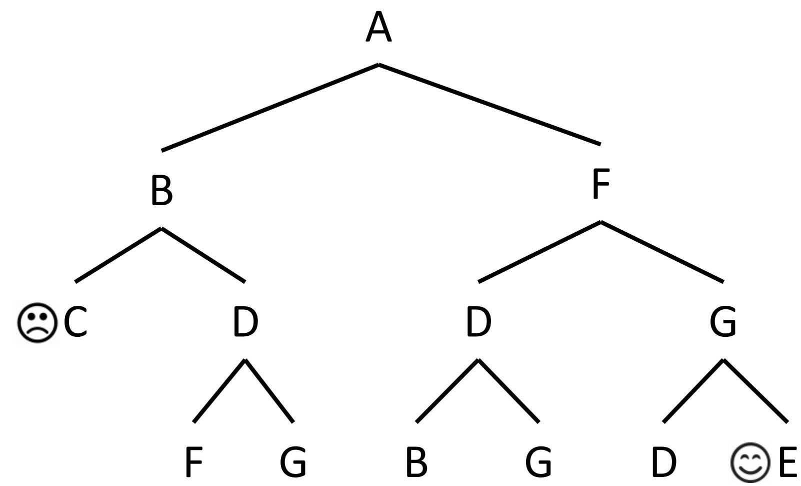




Solution: (A->F->G->E)

Breadth-First Search (BFS)



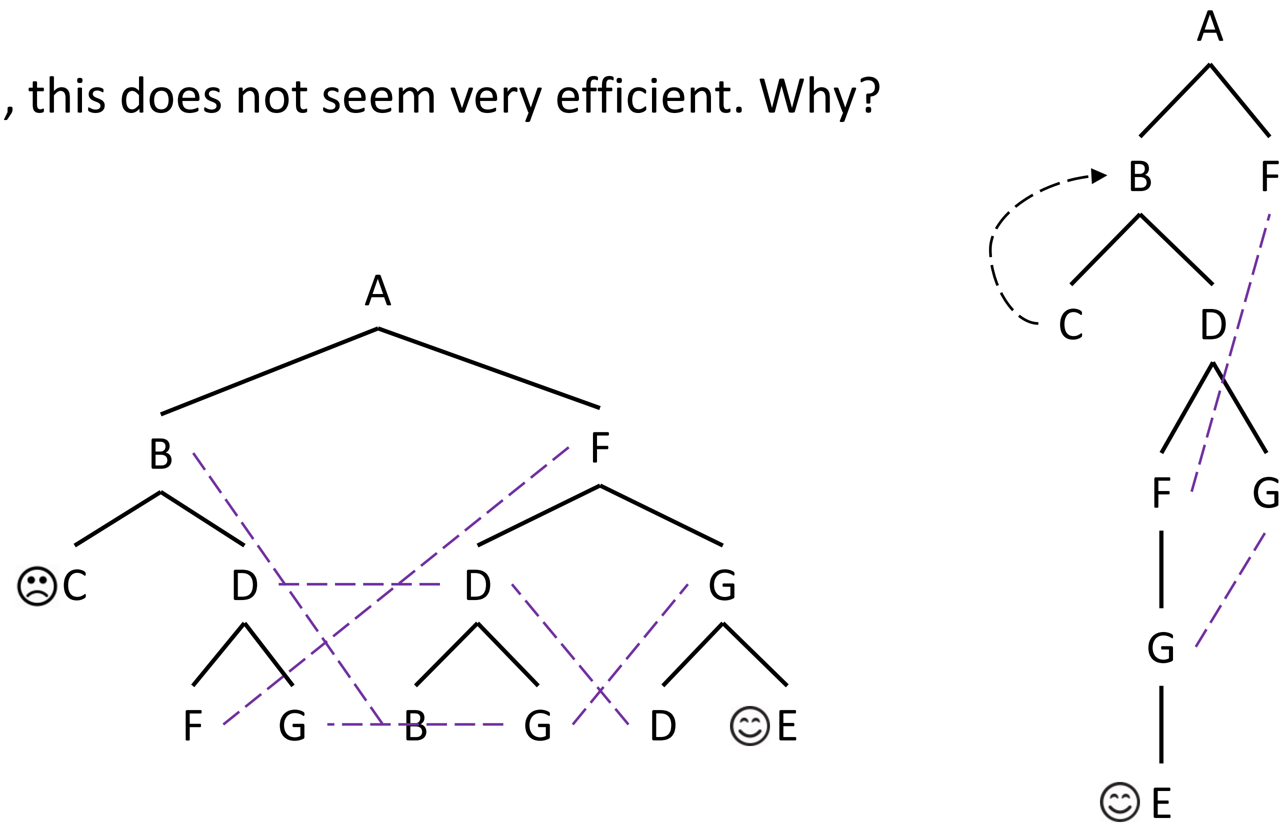


Solution: (A->F->G->E)

- A Breadth-First Search (BFS) chooses the shallowest node, thus exploring in a level by level fashion
- It has a more conservative behavior and does not need to reconsider decisions
- Call q the depth of the shallowest solution (in general $q \leq d$)
- Space complexity: $O(b^q)$
- Time complexity: $O(b^q)$

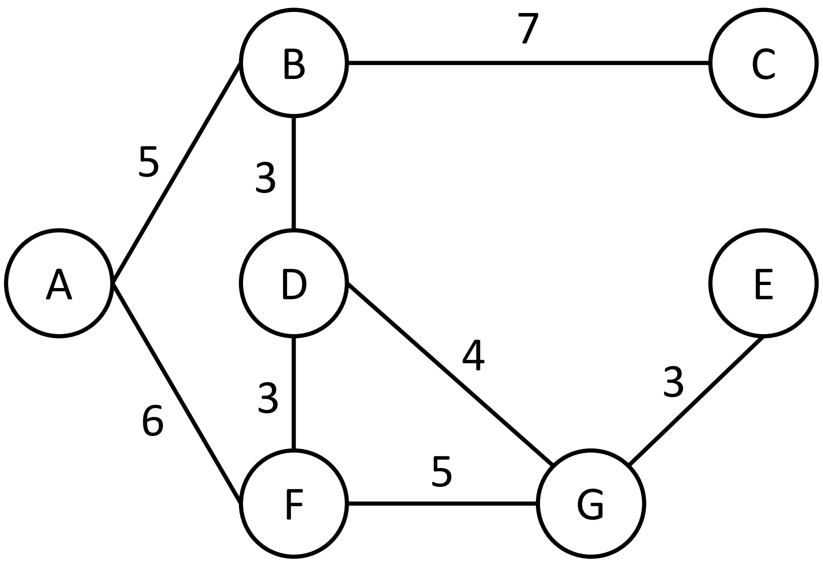
Redundant paths

- Both DFS and BFS visited some nodes multiple times (avoiding loops prevents this to happen only within the same branch)
- In general, this does not seem very efficient. Why?

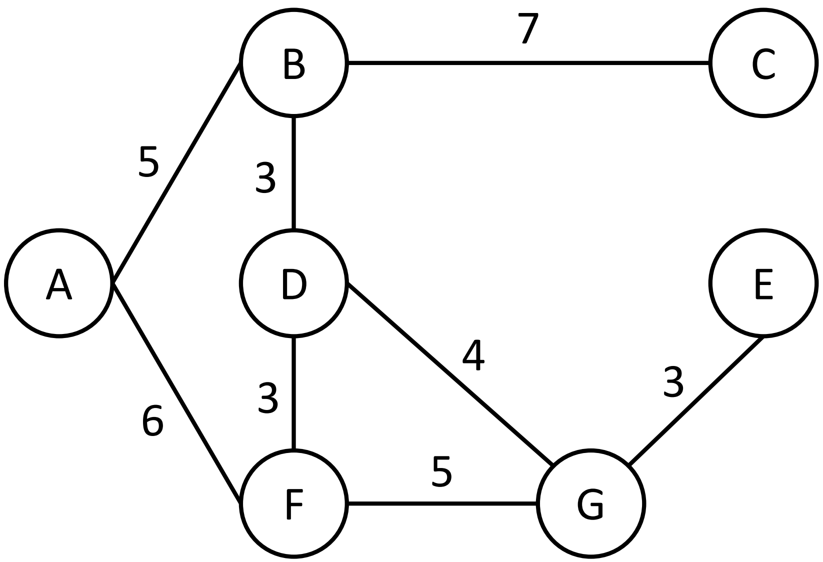


- Idea: discard a newly generated node if already present somewhere on the tree, we can do this with an **enqueued list**

DFS with Enqueued List

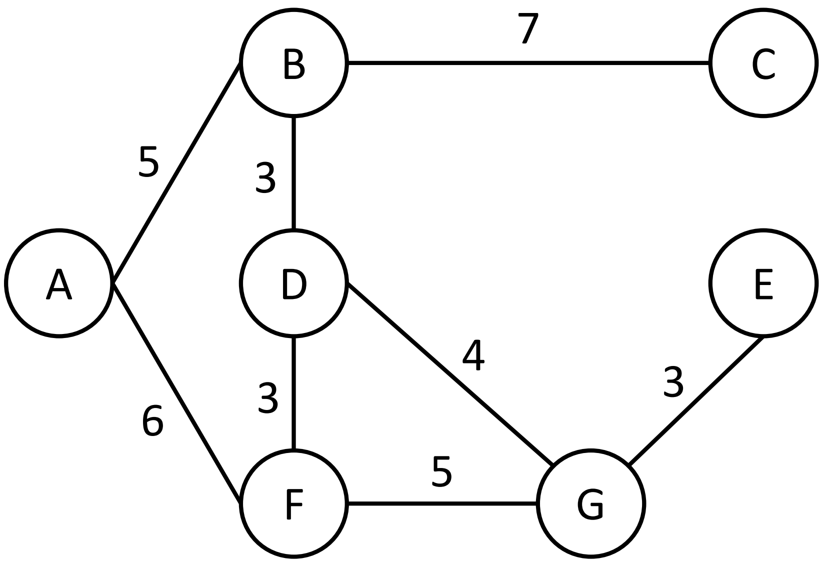


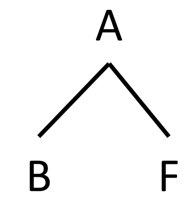
DFS with Enqueued List



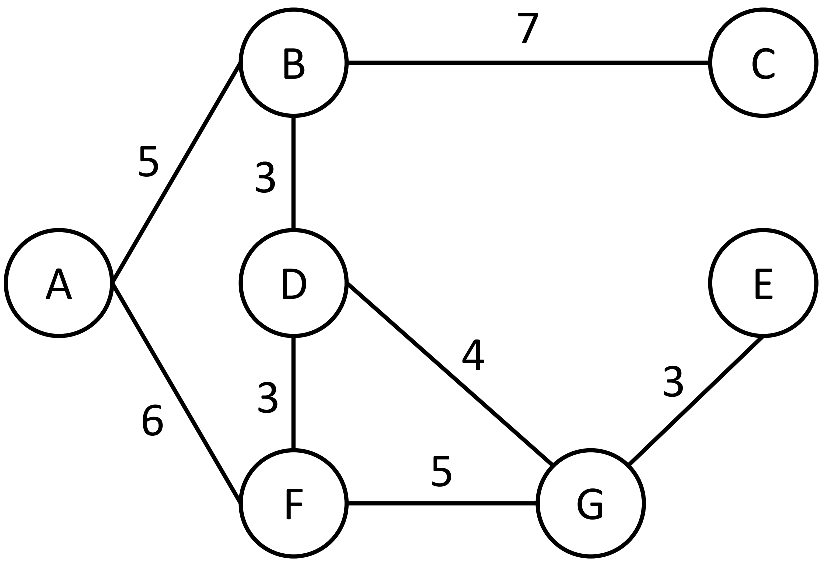
A

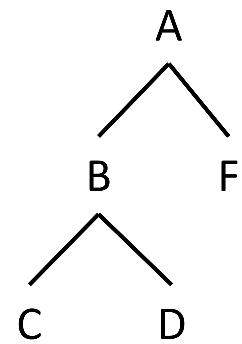
DFS with Enqueued List



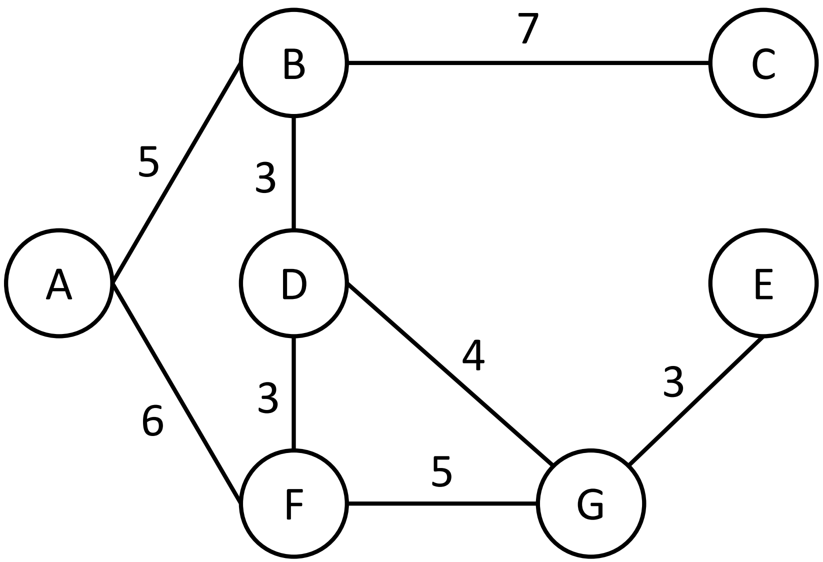


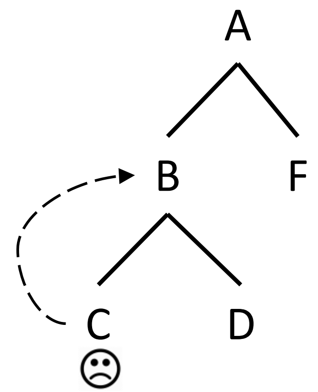
DFS with Enqueued List



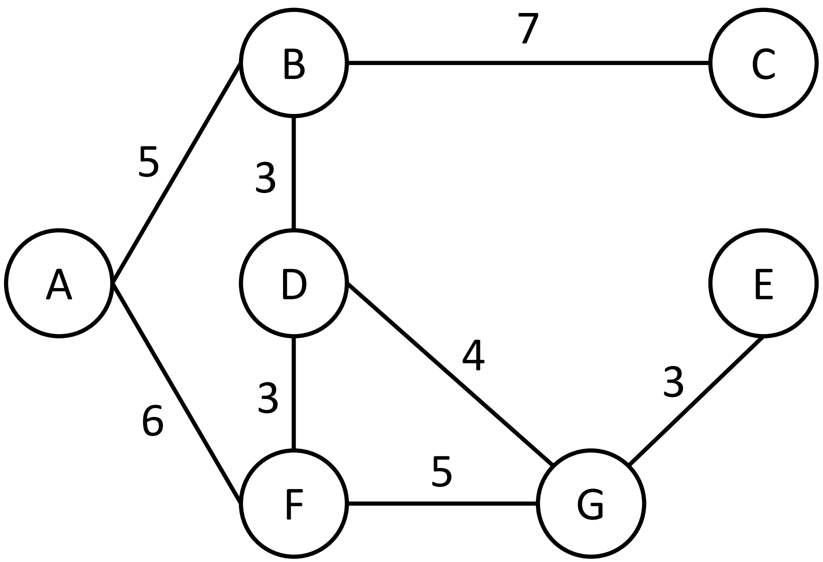


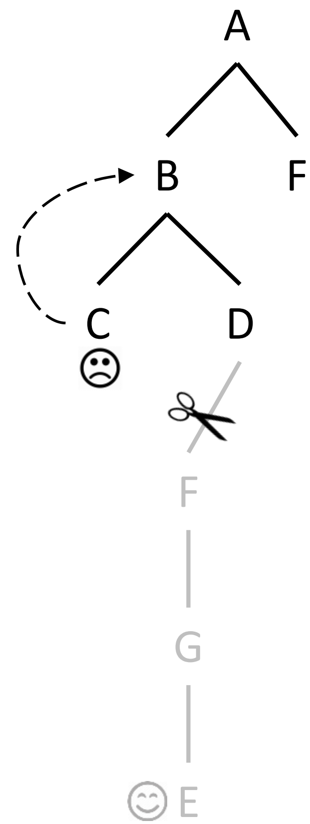
DFS with Enqueued List





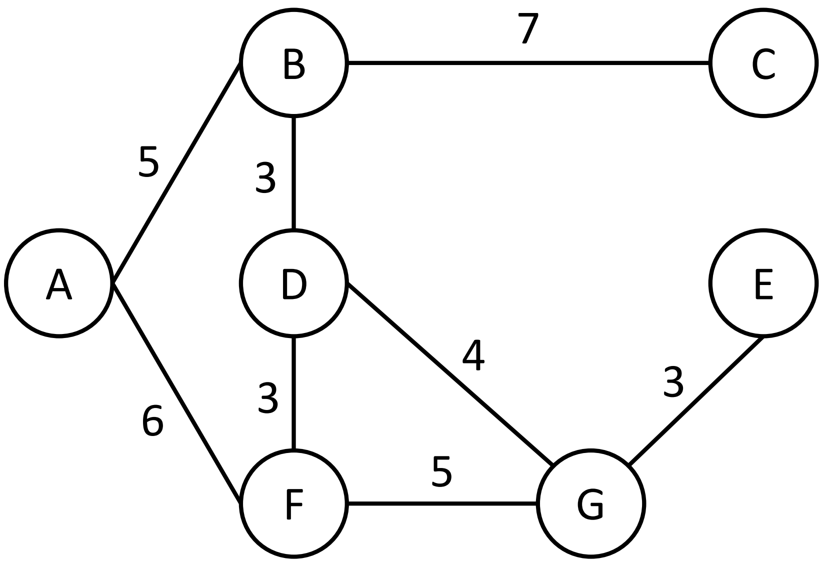
DFS with Enqueued List

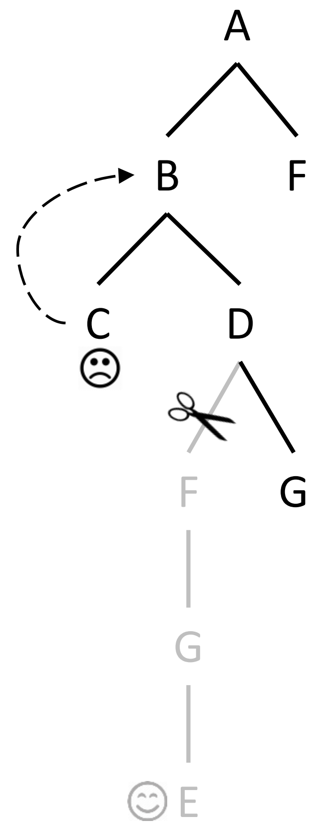




- Node F ha already been "enqueued" on the tree, by discarding it we *prune* a branch of the tree

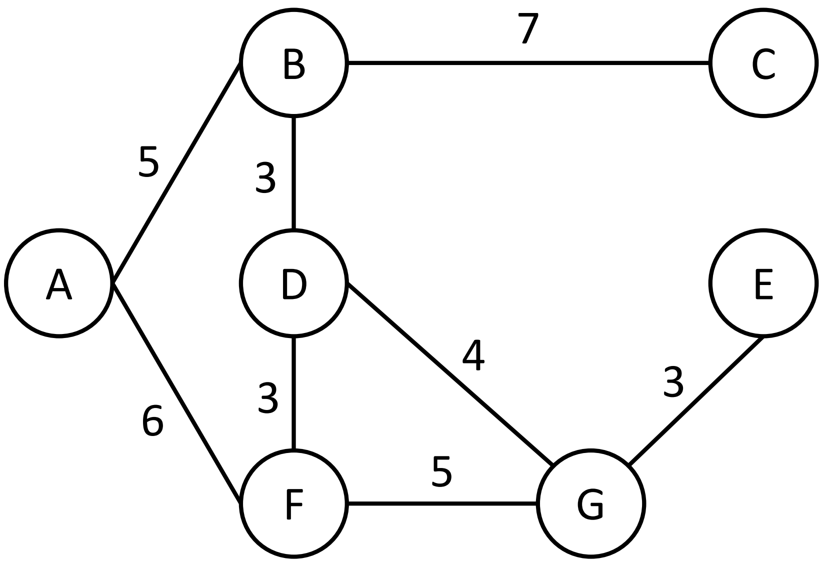
DFS with Enqueued List

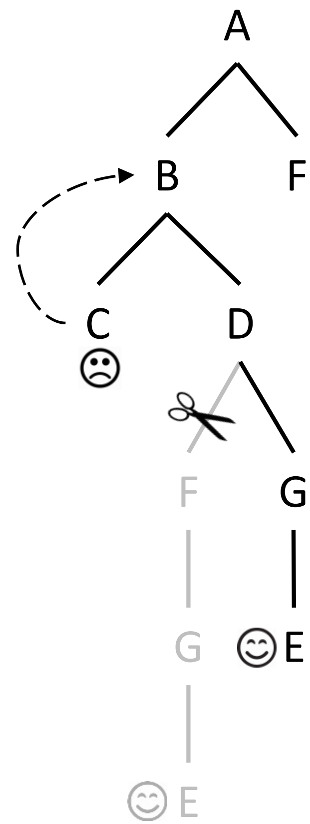




- Node F ha already been "enqueued" on the tree, by discarding it we *prune* a branch of the tree

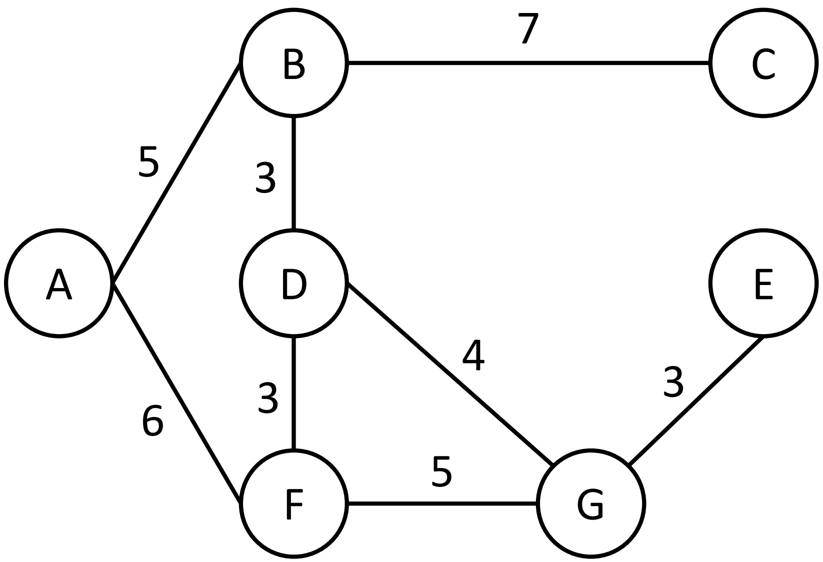
DFS with Enqueued List



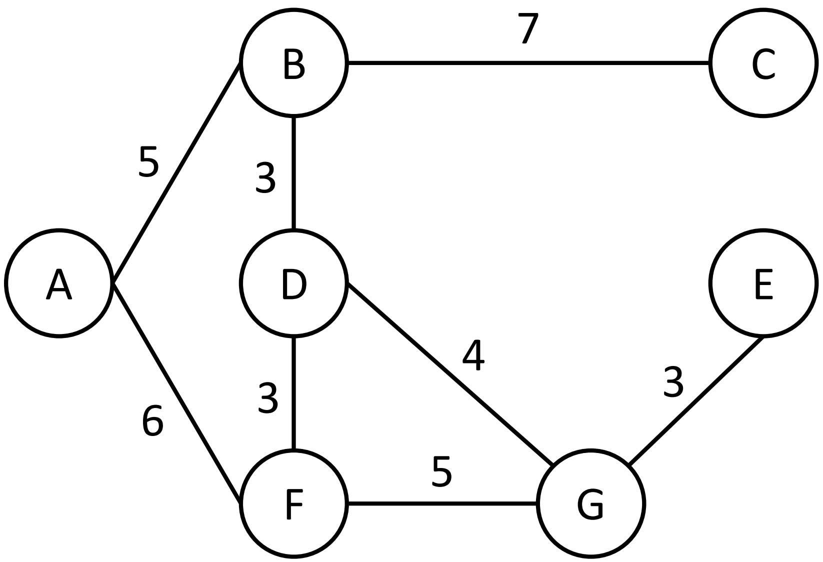


- Node F ha already been "enqueued" on the tree, by discarding it we *prune* a branch of the tree

BFS with Enqueued List

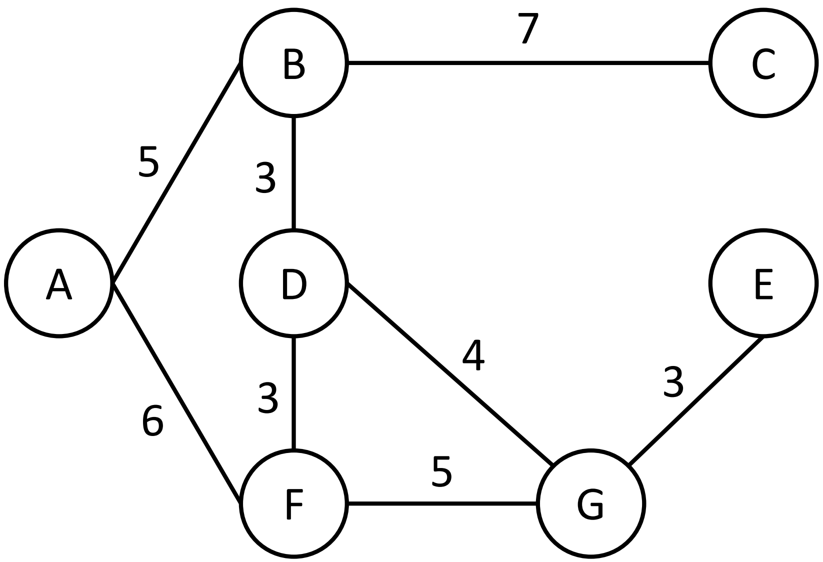


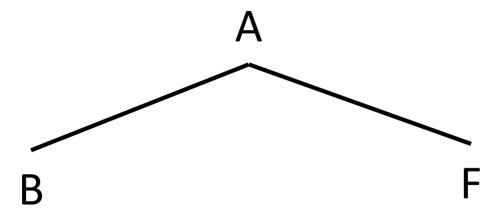
BFS with Enqueued List



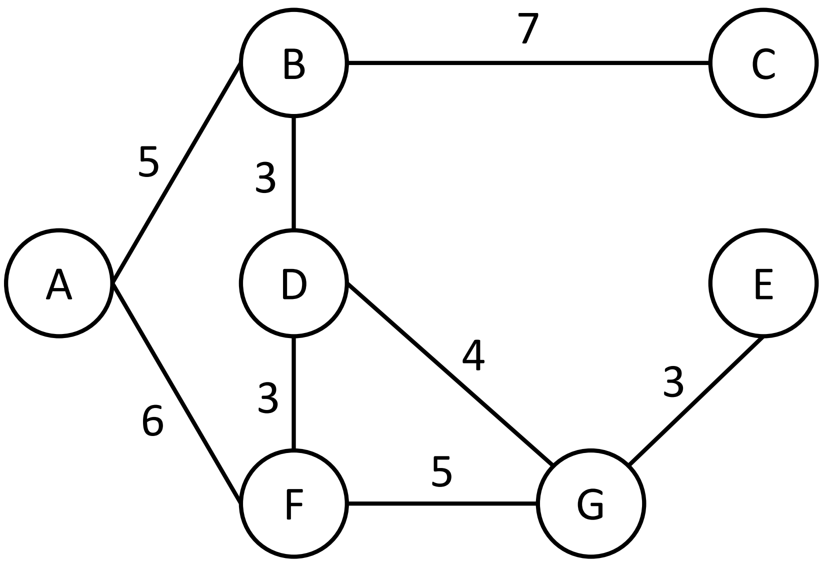
A

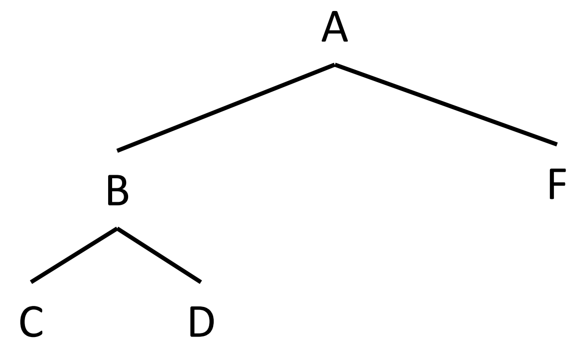
BFS with Enqueued List



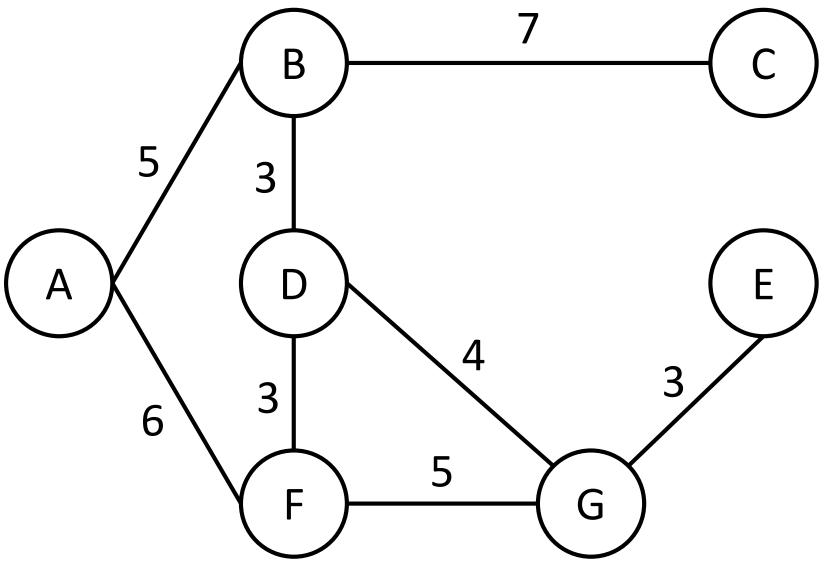


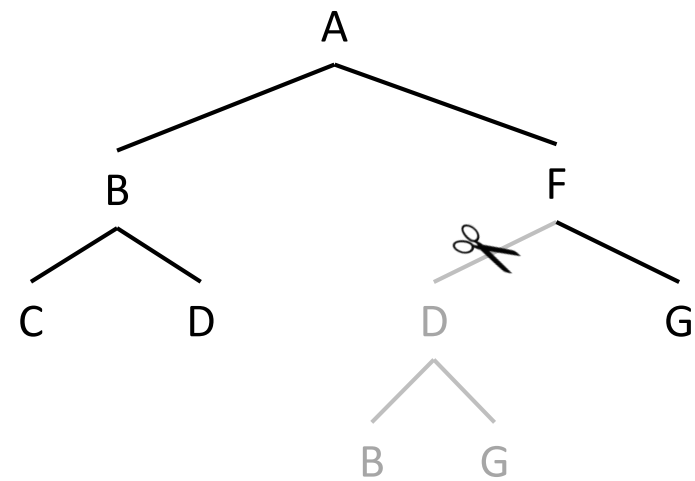
BFS with Enqueued List



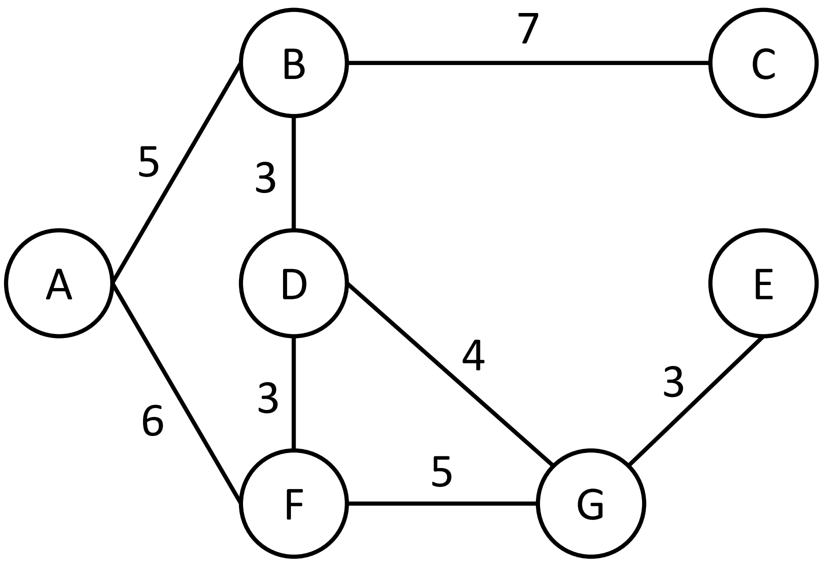


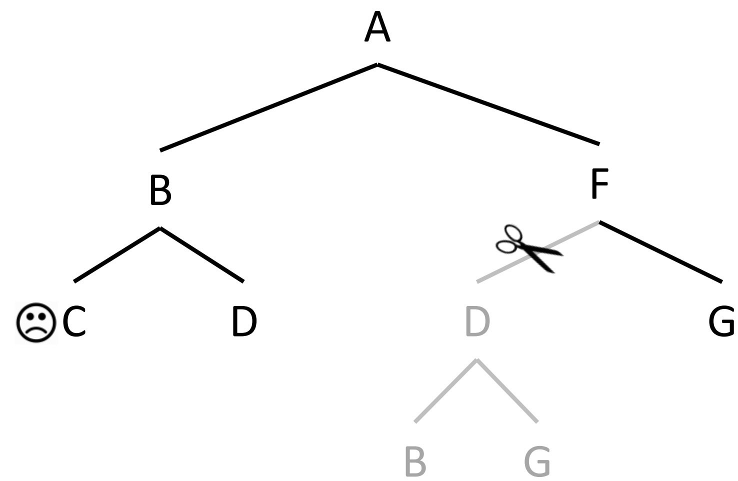
BFS with Enqueued List



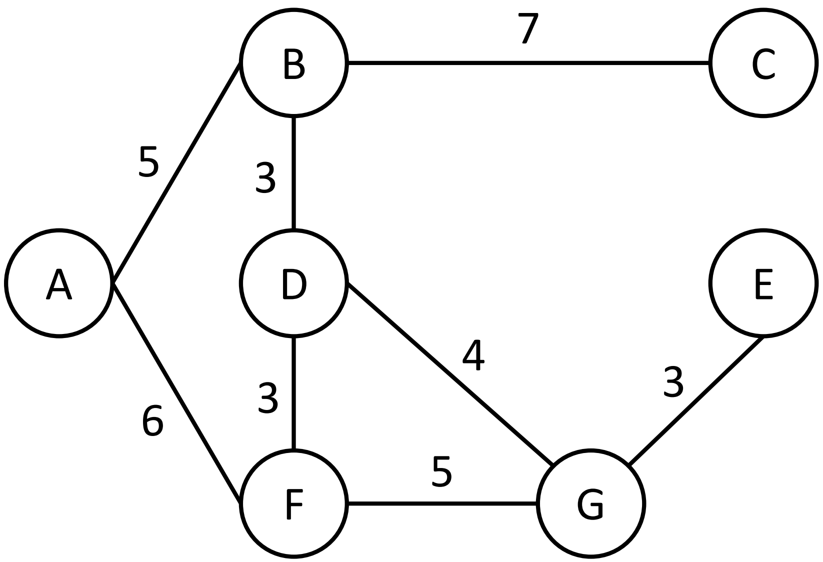


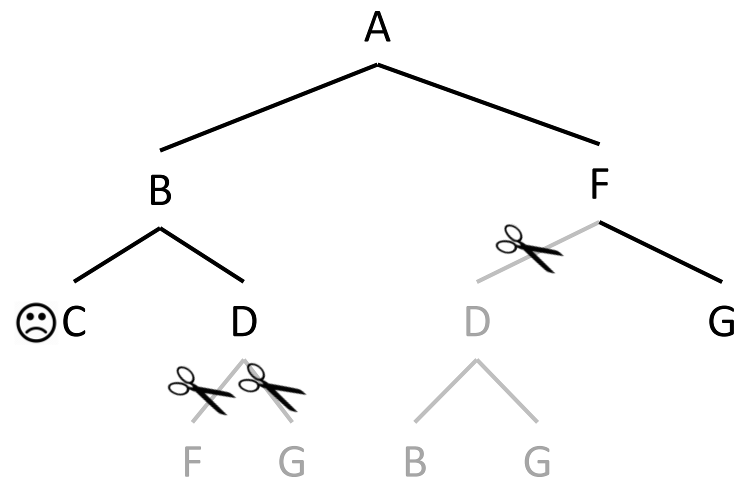
BFS with Enqueued List



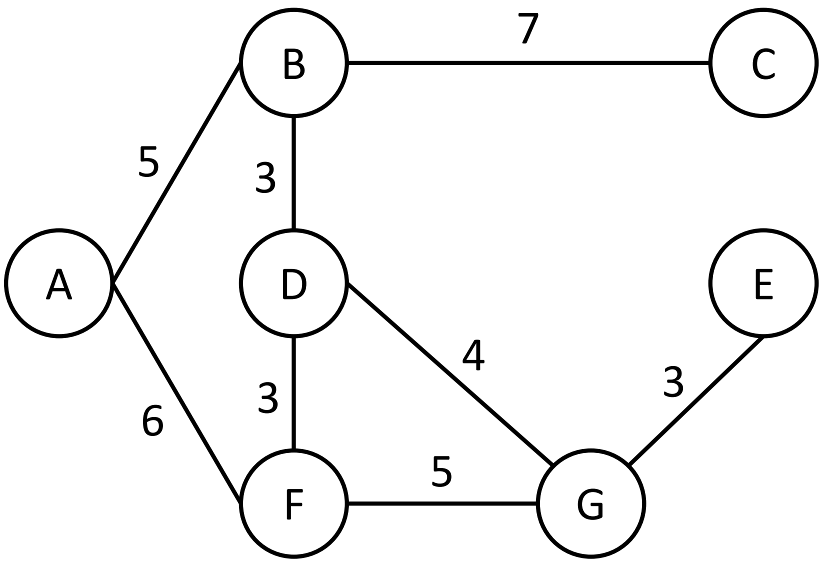


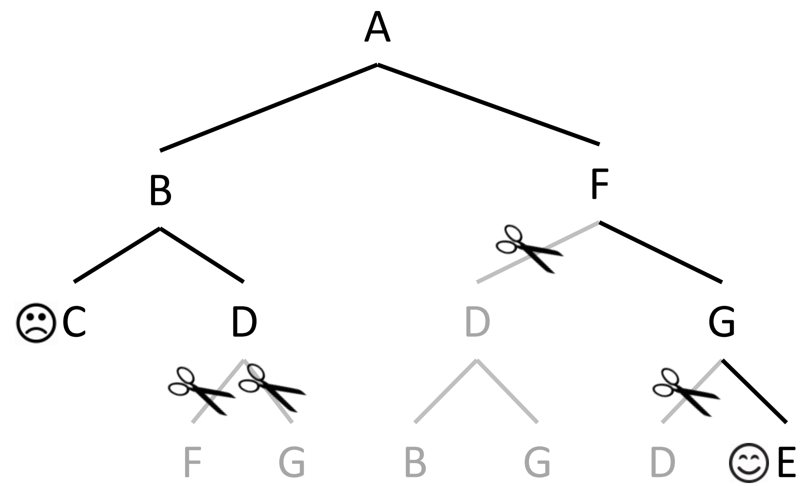
BFS with Enqueued List





BFS with Enqueued List

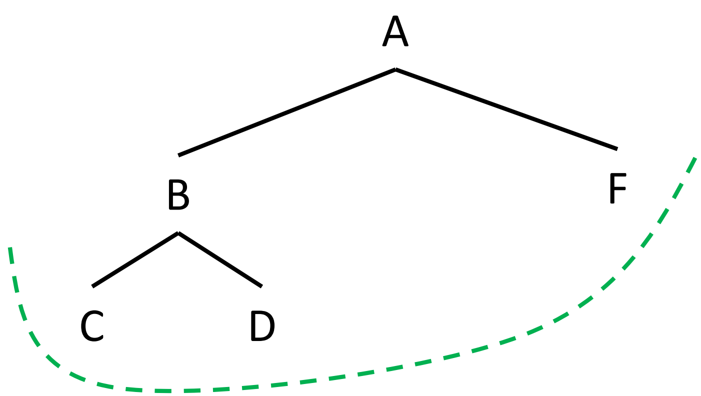


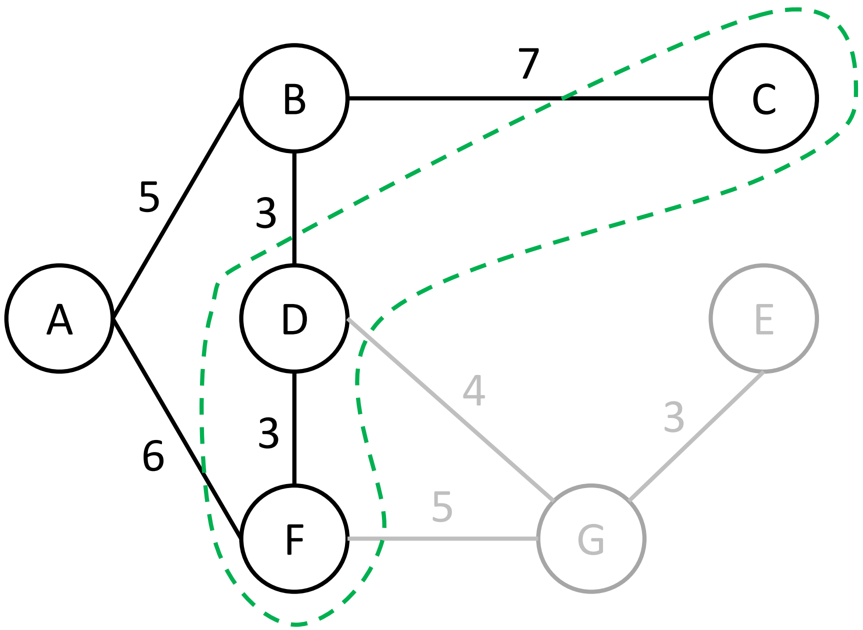


Implementation

- The implementation of the previous algorithms is based on two data structures:
  - A queue **F** (Frontier), elements ordered by priority, a selection consumes the element with highest priority
  - A list **EL** (Enqueued List, nodes that have already been put on the tree)

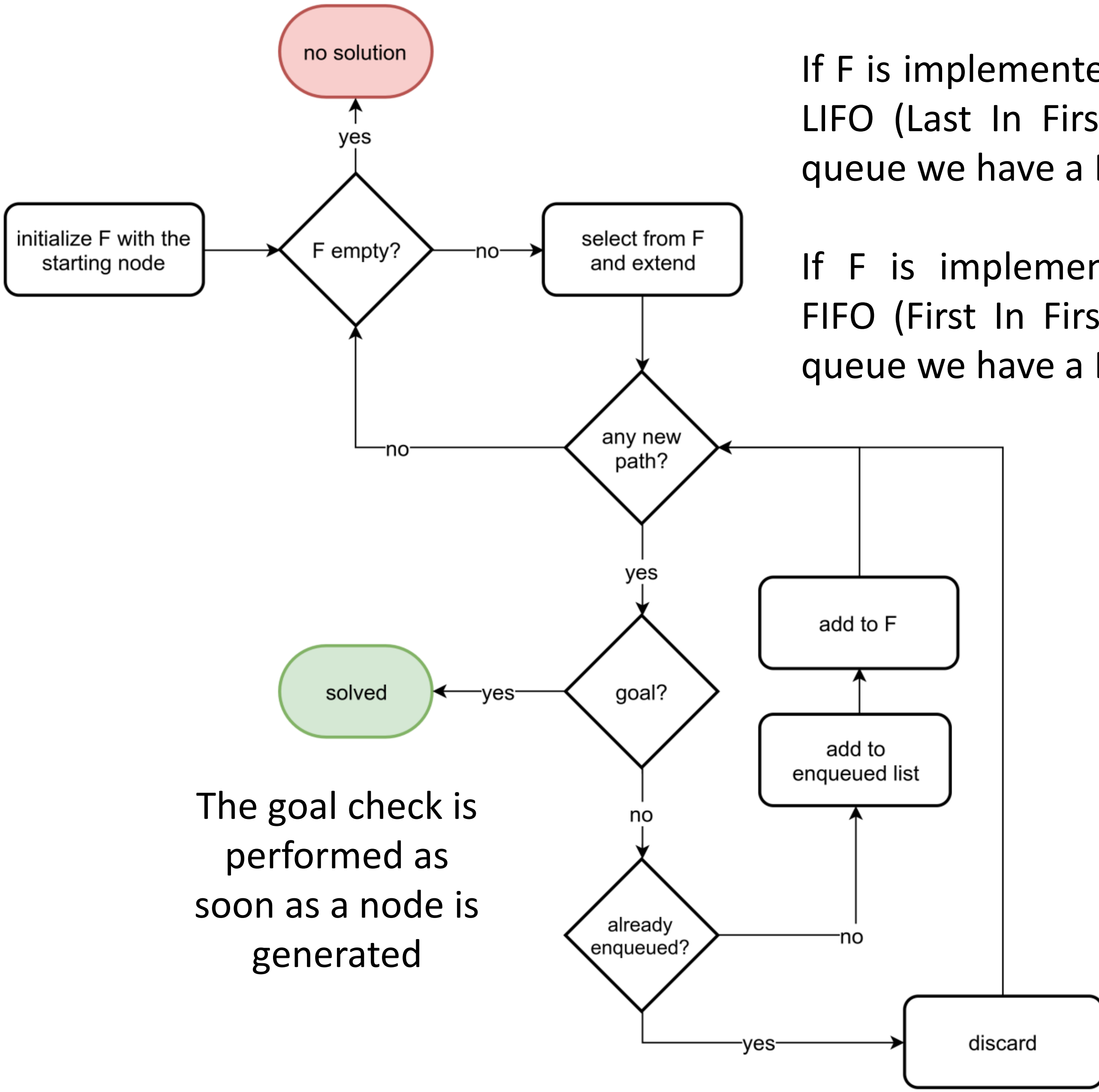
- The frontier F contains the terminal nodes of all the paths currently under exploration on the tree





- The frontier **separates** the explored part of the state space from the unexplored part
- In order to reach a state that we still did not searched, we need to pass from the frontier (separation property)

Implementation



If F is implemented as a LIFO (Last In First Out) queue we have a DFS

If F is implemented a FIFO (First In First Out) queue we have a BFS

The goal check is performed as soon as a node is generated

Informed vs non-informed search

- Besides its own rules, any search algorithm decides where to search next by leveraging some knowledge
- **Non-informed** search uses only knowledge specified at problem-definition time (e.g., goal and start nodes, edge costs), just like we saw in the previous examples
- An **informed** search might go beyond such knowledge
- Idea: using an estimate of how far a given node is from the goal
- Such an estimate is often called a **heuristic**

Estimate of the cost of the optimal path from node v to the goal: $h(v)$

Informed vs non-informed search

- We can enrich DFS and BFS to obtain their an informed versions
- Both search methods break ties in lexicographical order, but it seems reasonable to do that in favor of nodes that are believed to be closer to the goal
- **Hill climbing**
  - A DFS where ties are broken in favor the node with smallest h
- **Beam** (of width w)
  - A BFS where at each level we keep the first w nodes in increasing order of h

Search for the optimal solution

- Now we assume to be interested in the solution with minimum cost (not just any path to the goal, but the cheapest possible)

- To devise an optimal search algorithm we take the moves from BFS. Why it seems reasonable to do that?

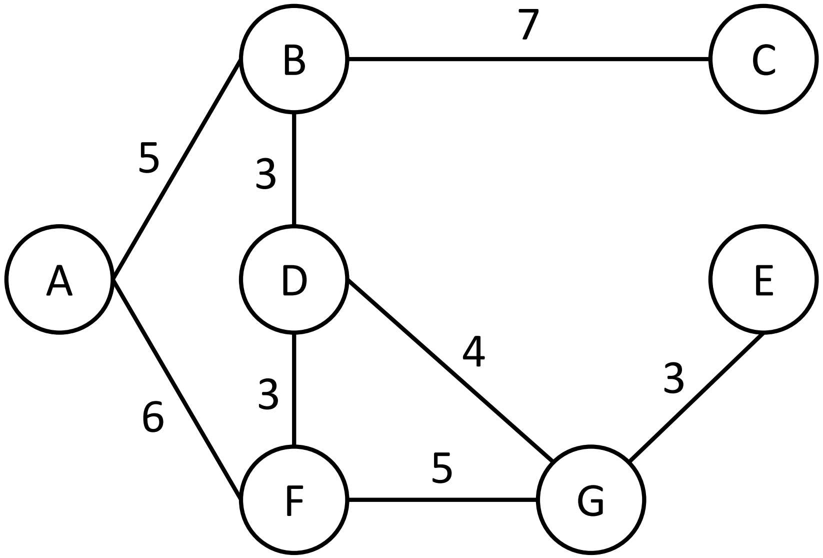
- We generalize the idea of BFS to that of Uniform Cost Search (UCS)

- BFS proceeds by *depth* levels, UCS does that by *cost* levels (as a consequence, if costs are all equal to some constant BFS and UCS coincide)

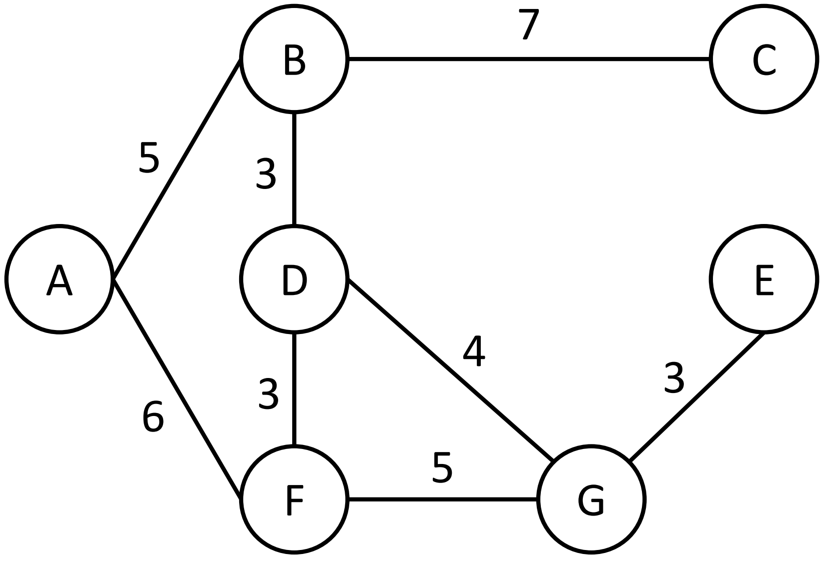
- Cost accumulated on a path from the start node to v: $g(v)$ (we should include a dependency on the path, but it will always be clear from the context)

- For now let's remove the enqueued list and the goal checking as we know it

Uniform Cost Search (UCS)

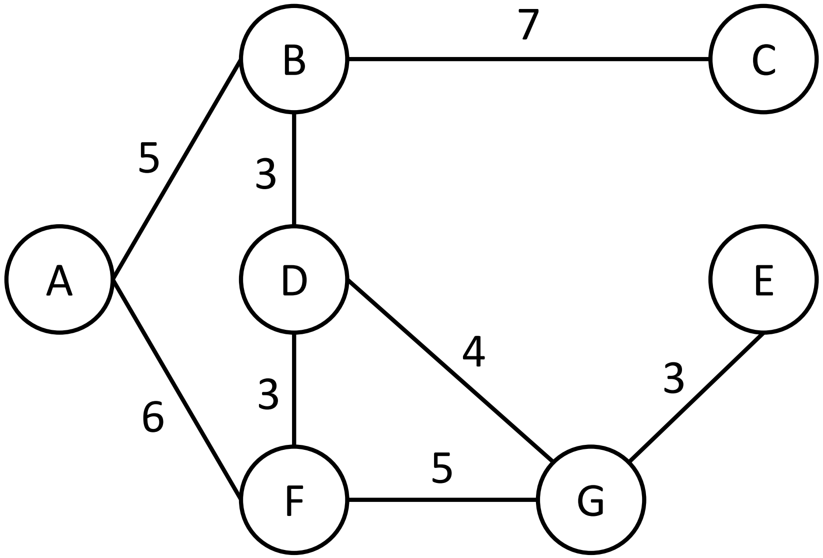


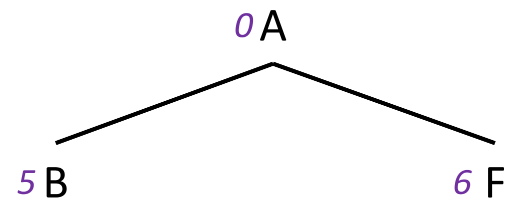
Uniform Cost Search (UCS)



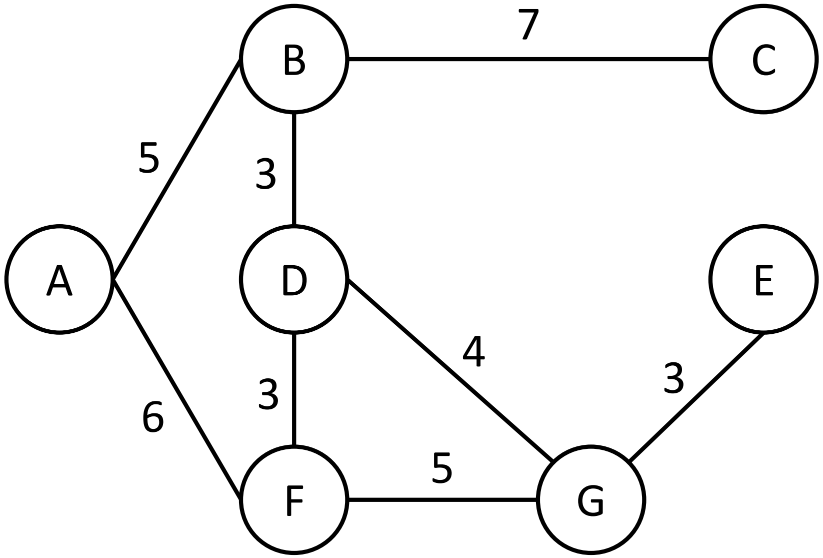
0 A

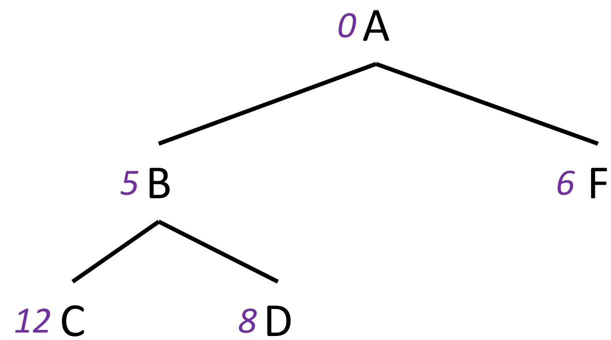
Uniform Cost Search (UCS)



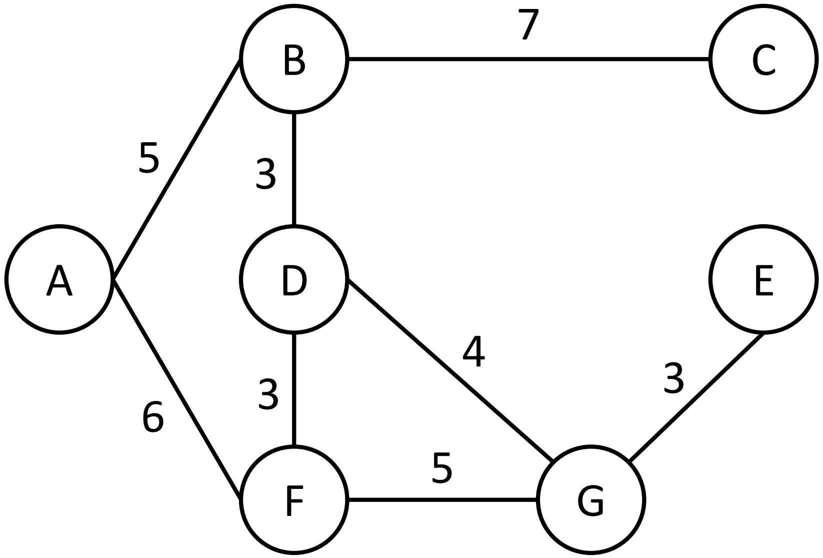


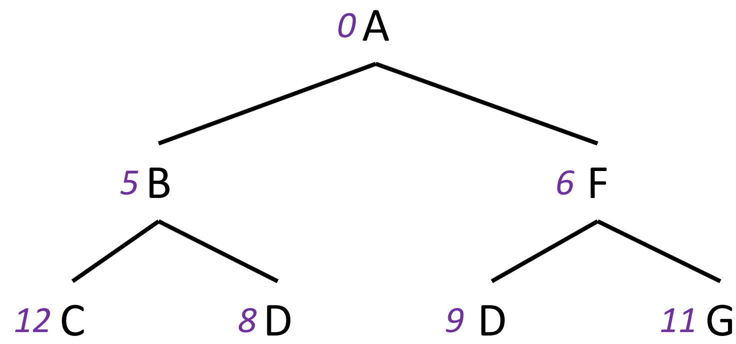
Uniform Cost Search (UCS)



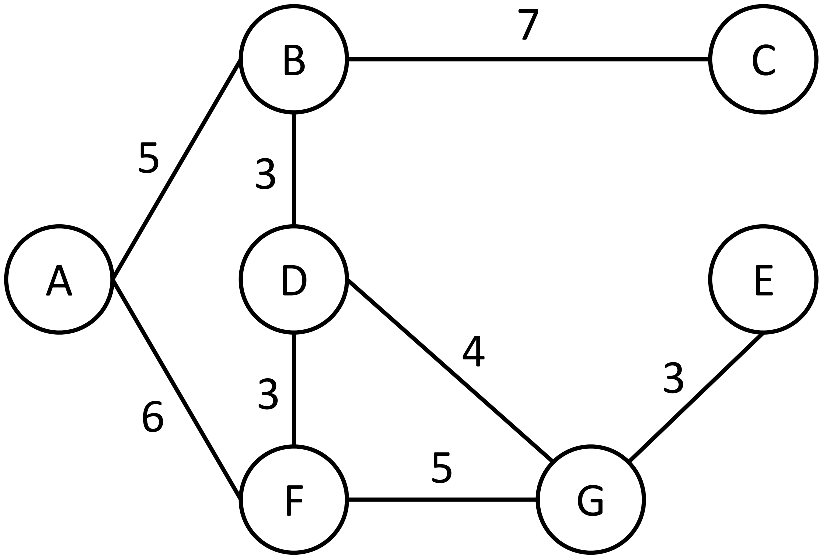


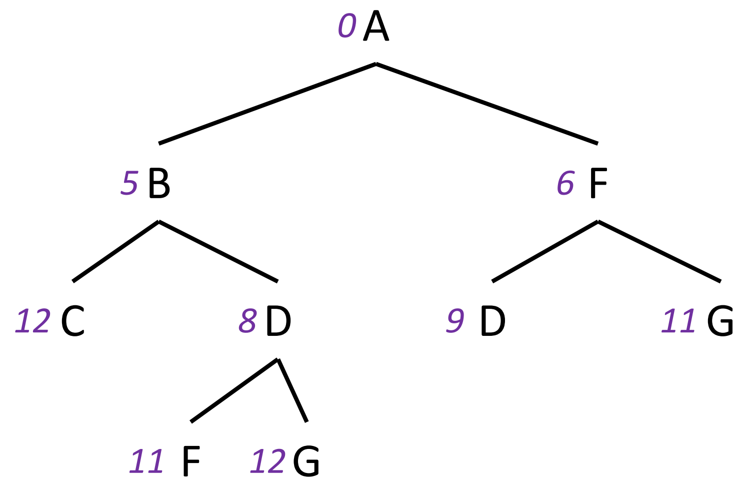
Uniform Cost Search (UCS)



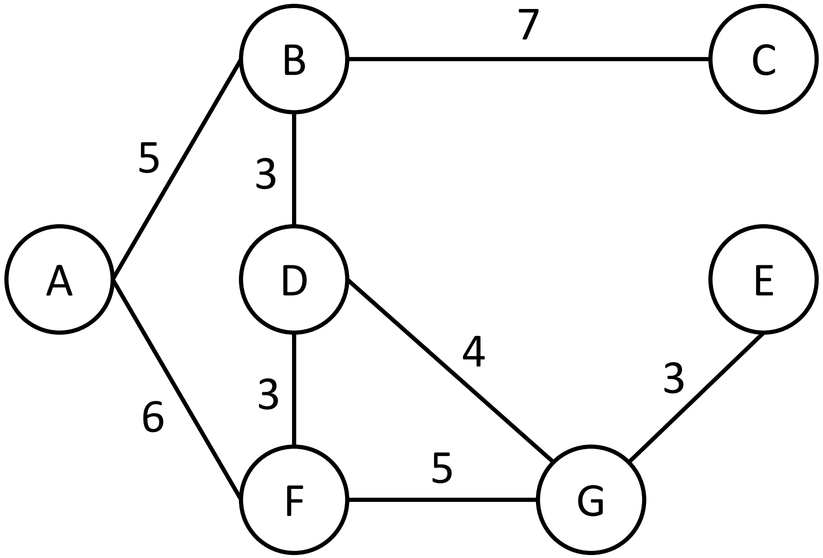


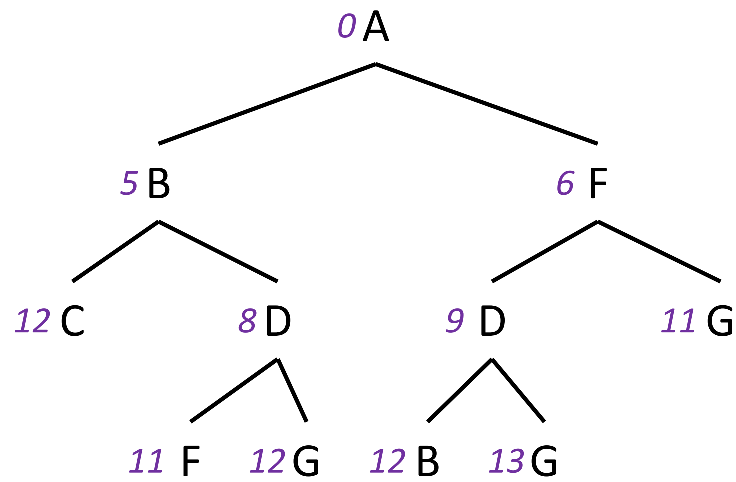
Uniform Cost Search (UCS)



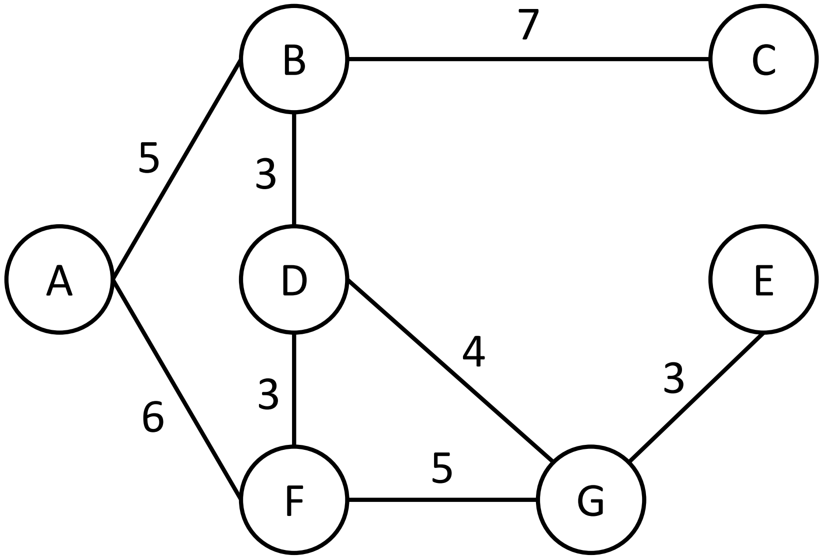


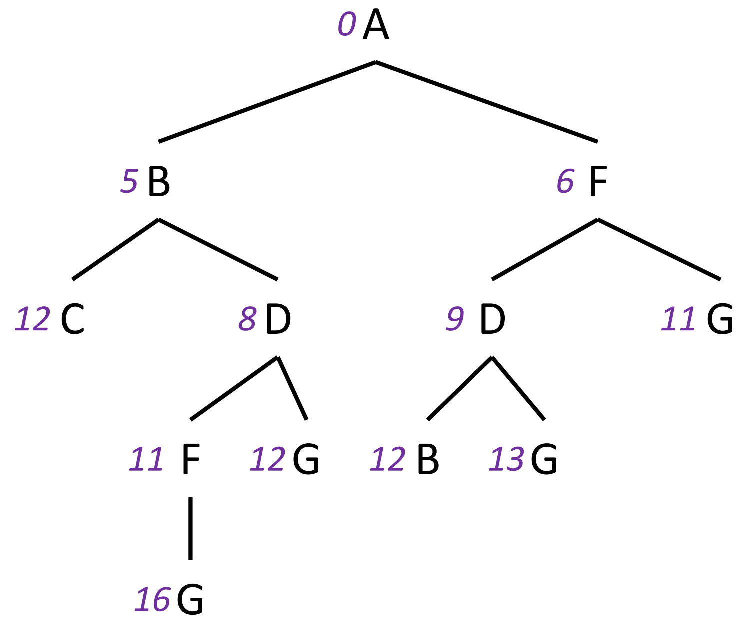
Uniform Cost Search (UCS)



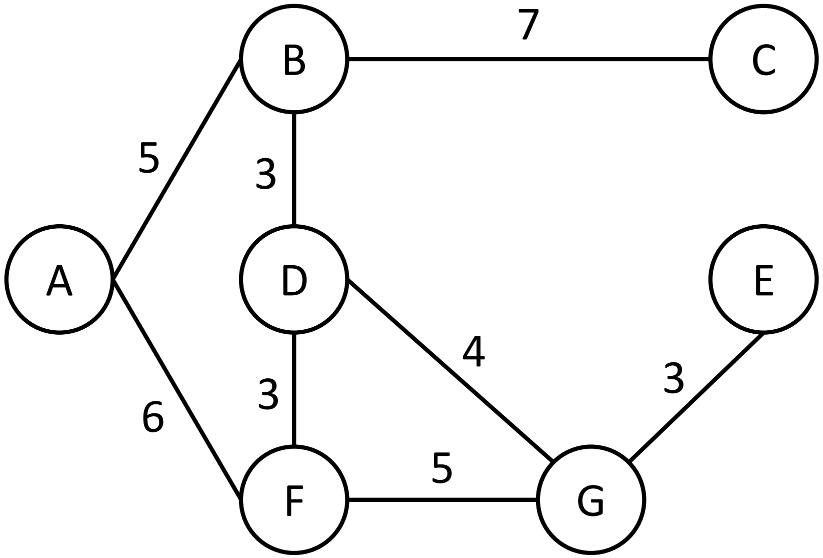


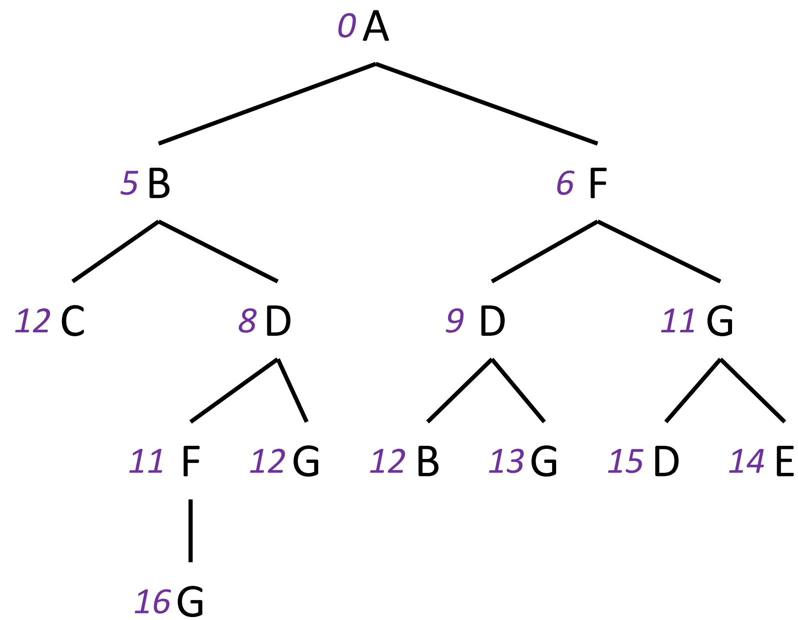
Uniform Cost Search (UCS)



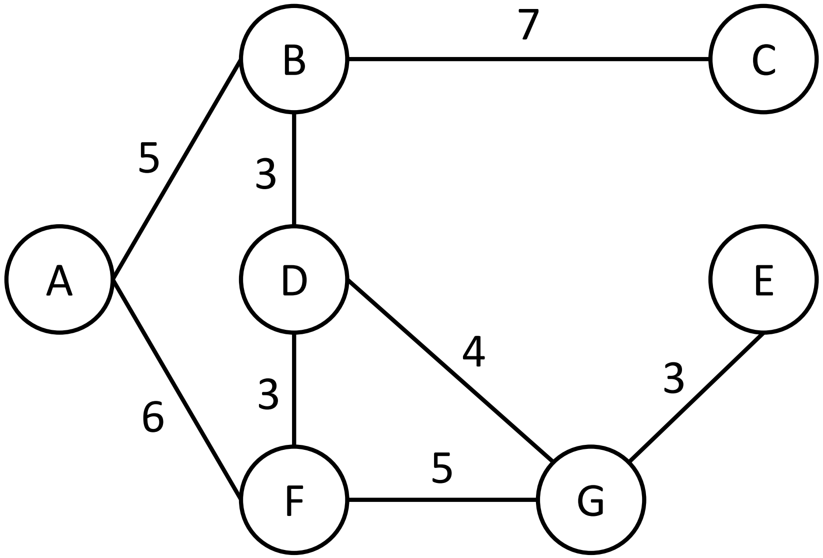


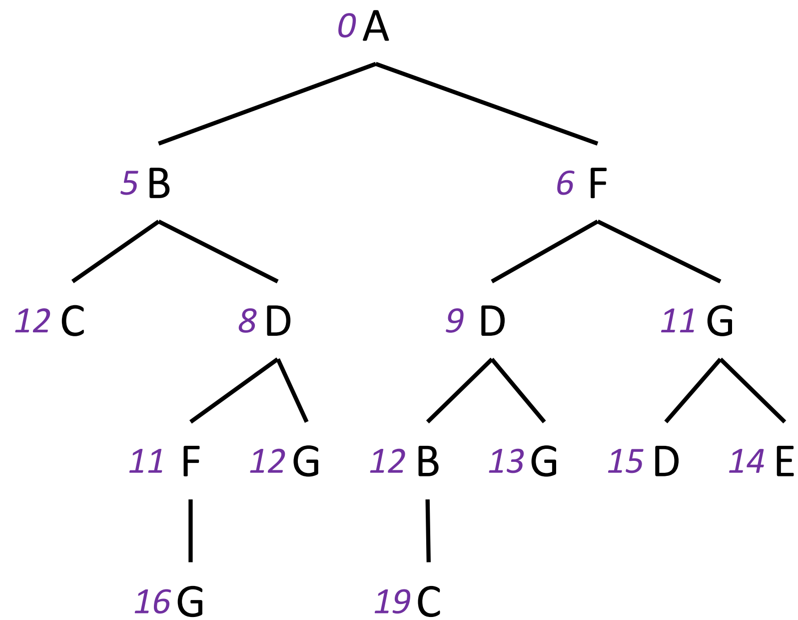
Uniform Cost Search (UCS)



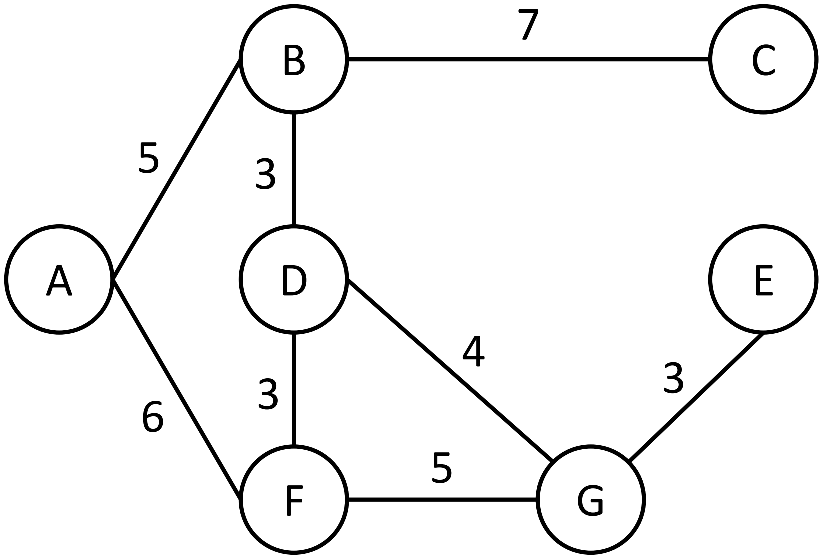


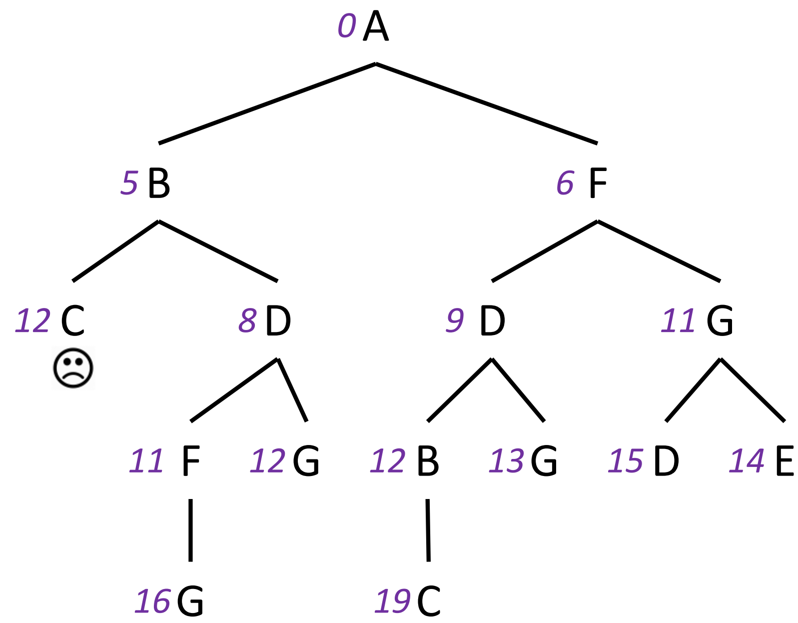
Uniform Cost Search (UCS)



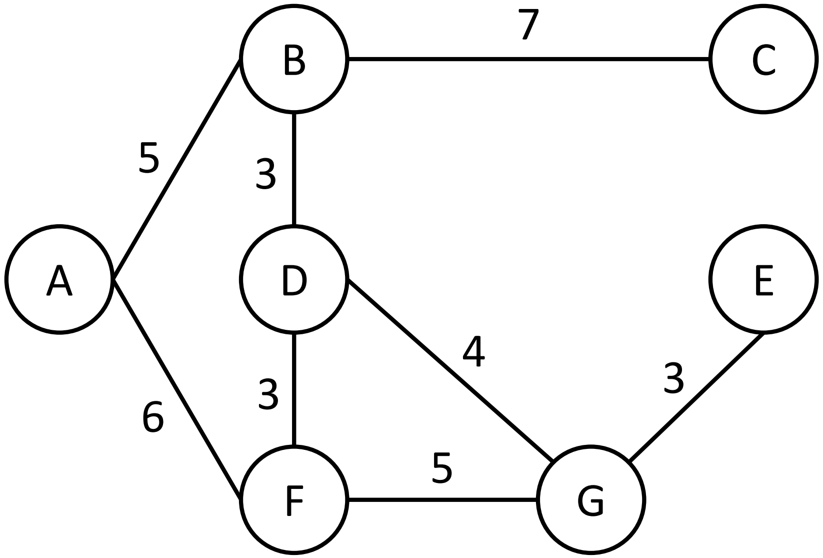


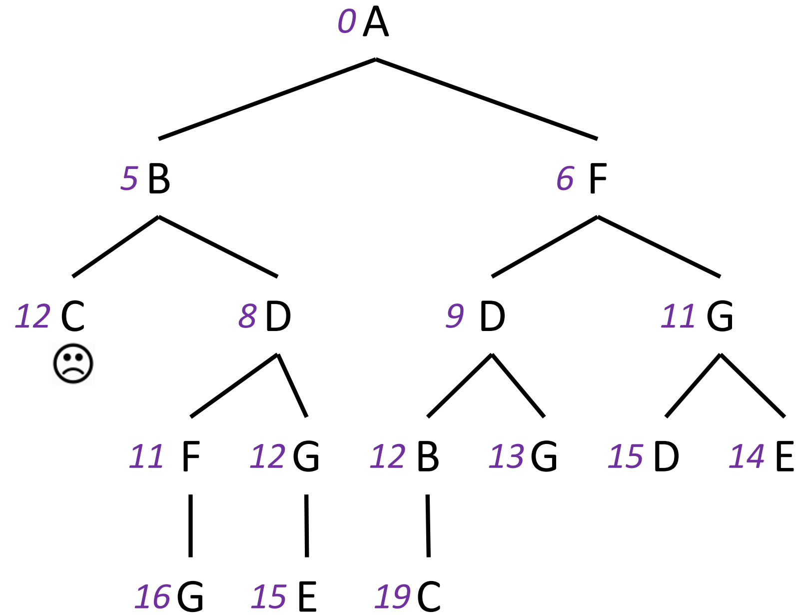
Uniform Cost Search (UCS)



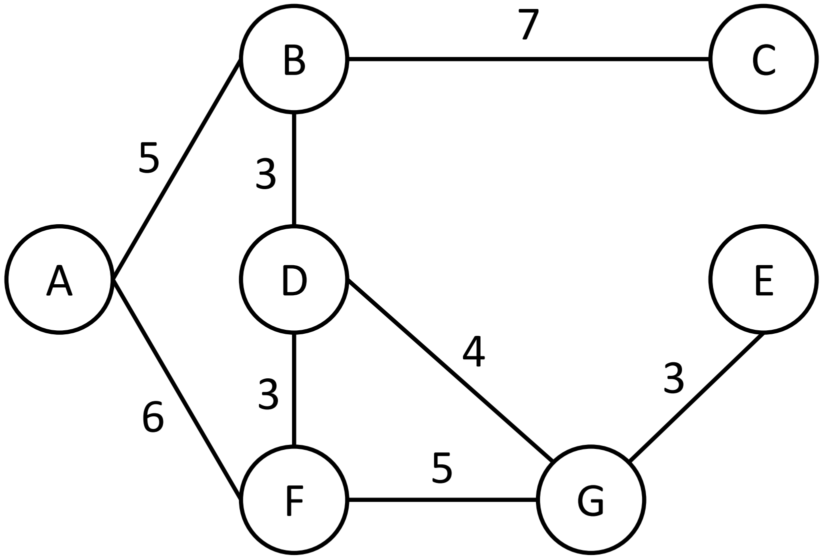


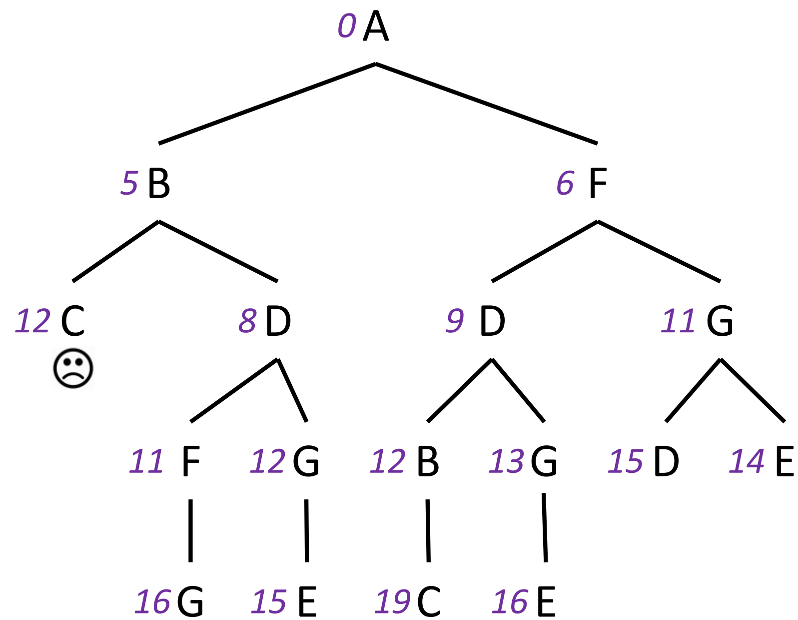
Uniform Cost Search (UCS)



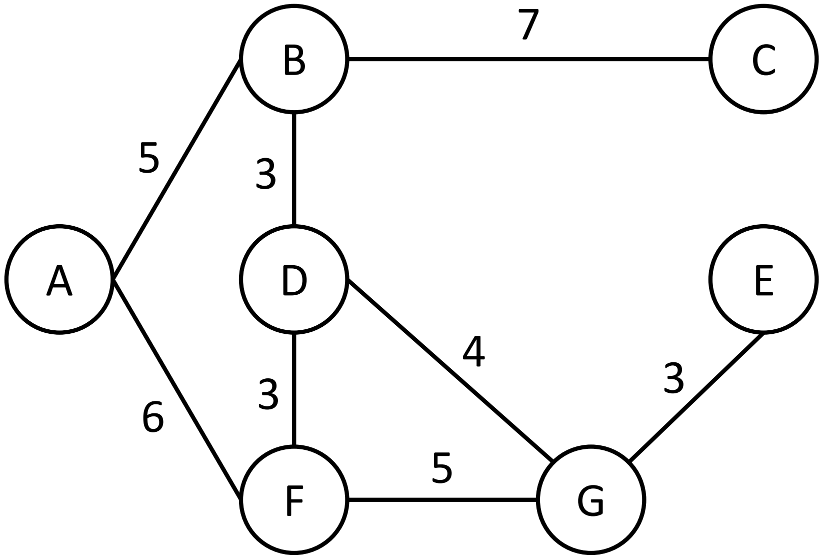


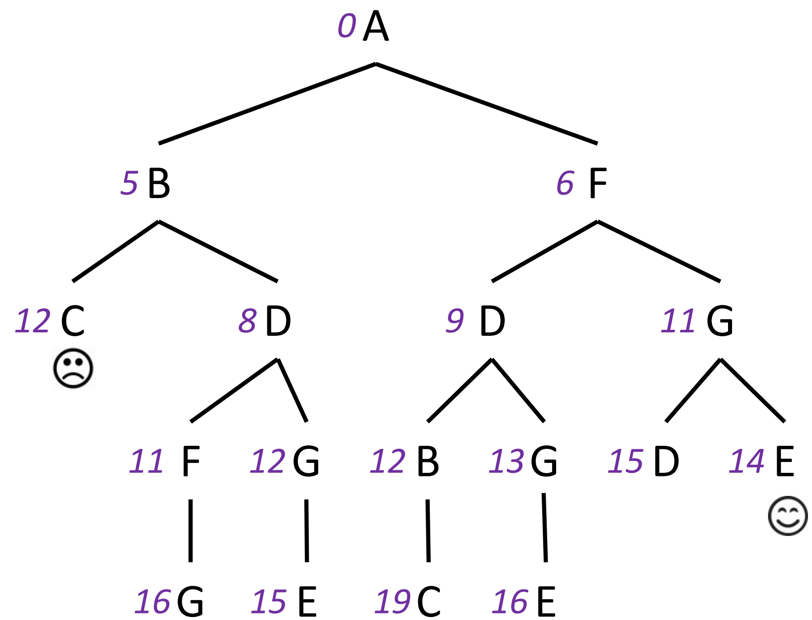
Uniform Cost Search (UCS)



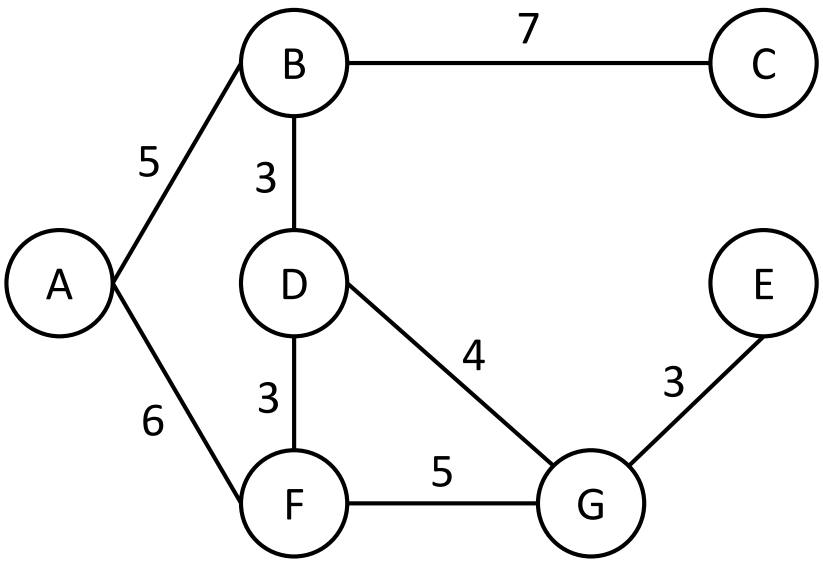


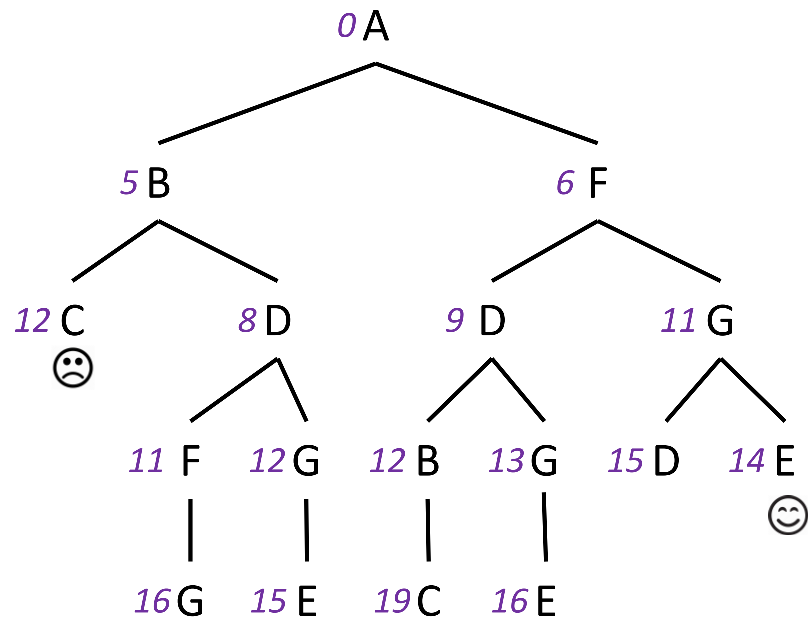
Uniform Cost Search (UCS)





Uniform Cost Search (UCS)



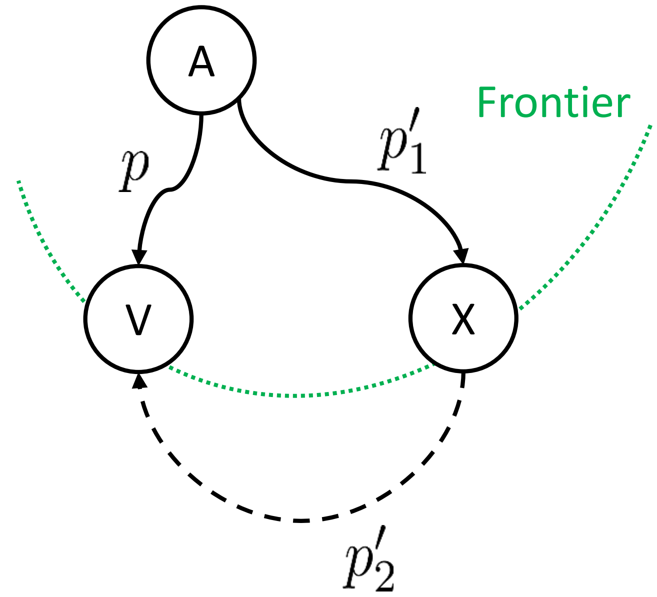


- Have we found the optimal path to the goal? In this problem instance, we can answer *yes* by inspecting the graph
- How about larger instances? Can we prove optimality?
- Actually, we can prove a stronger claim: every time UCS selects **for the first time** a node for expansion, the associated path leading to that node has minimum cost

Optimality of UCS

Hypotheses:

1. UCS selects from the frontier a node V that has been generated through a path p
2. p is not the optimal path to V



Given 2 and the frontier separation property, we know that there must exist a node X on the frontier, generated through a path p'_1 that is on the optimal path $p' \neq p$ to V; let assume $p' = p'_1 + p'_2$

$c(p') = c(p'_1) + c(p'_2) < c(p)$ since, from Hp, p' is optimal

$c(p'_1) < c(p'_1) + c(p'_2) < c(p)$ since costs are positive

$c(p'_1) < c(p)$ X would have been chosen before V, then 1 is false

Optimality of UCS

If when we select for the first time we discover the optimal path, there is no reason to select the same node a second time: **extended list**

Every time we select a node for extension:

- If the node is already in the extended list we discard it
- Otherwise we extend it and we put it the extended list

- (Warning: we are not using an enqueued list, it would actually make the search not sound!)

Sistemi Intelligenti
Corso di Laurea in Informatica, A.A. 2019-2020
Università degli Studi di Milano

Nicola Basilico
Dipartimento di Informatica
Via Celoria 18- 20133 Milano (MI)
Ufficio 4008
nicola.basilico@unimi.it
+39 02.503.16289