Sistemi Intelligenti Avanzati
Corso di Laurea in Informatica, A.A. 2021-2022
Università degli Studi di Milano

Search algorithms for planning

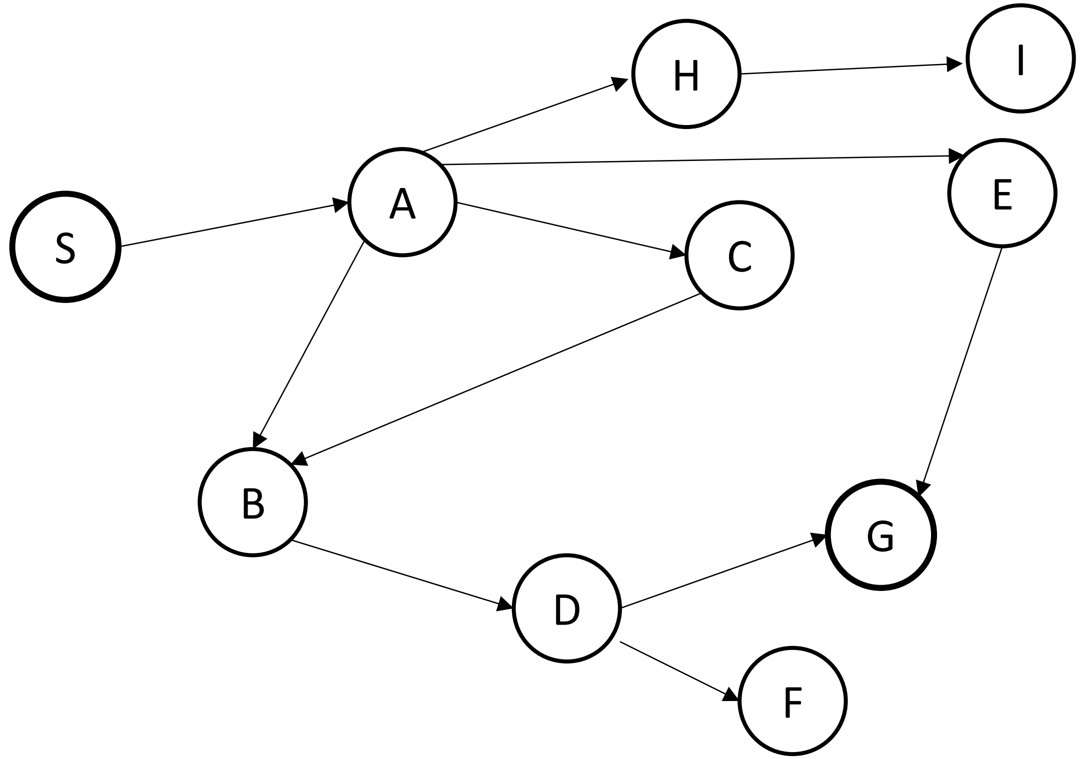
Matteo Luperto
Dipartimento di Informatica
matteo.luperto@unimi.it

Search

Setting:

- Agent
- Goal
- Problem Formulation
  - A Set of Actions
  - A Set of States

What we want to do?



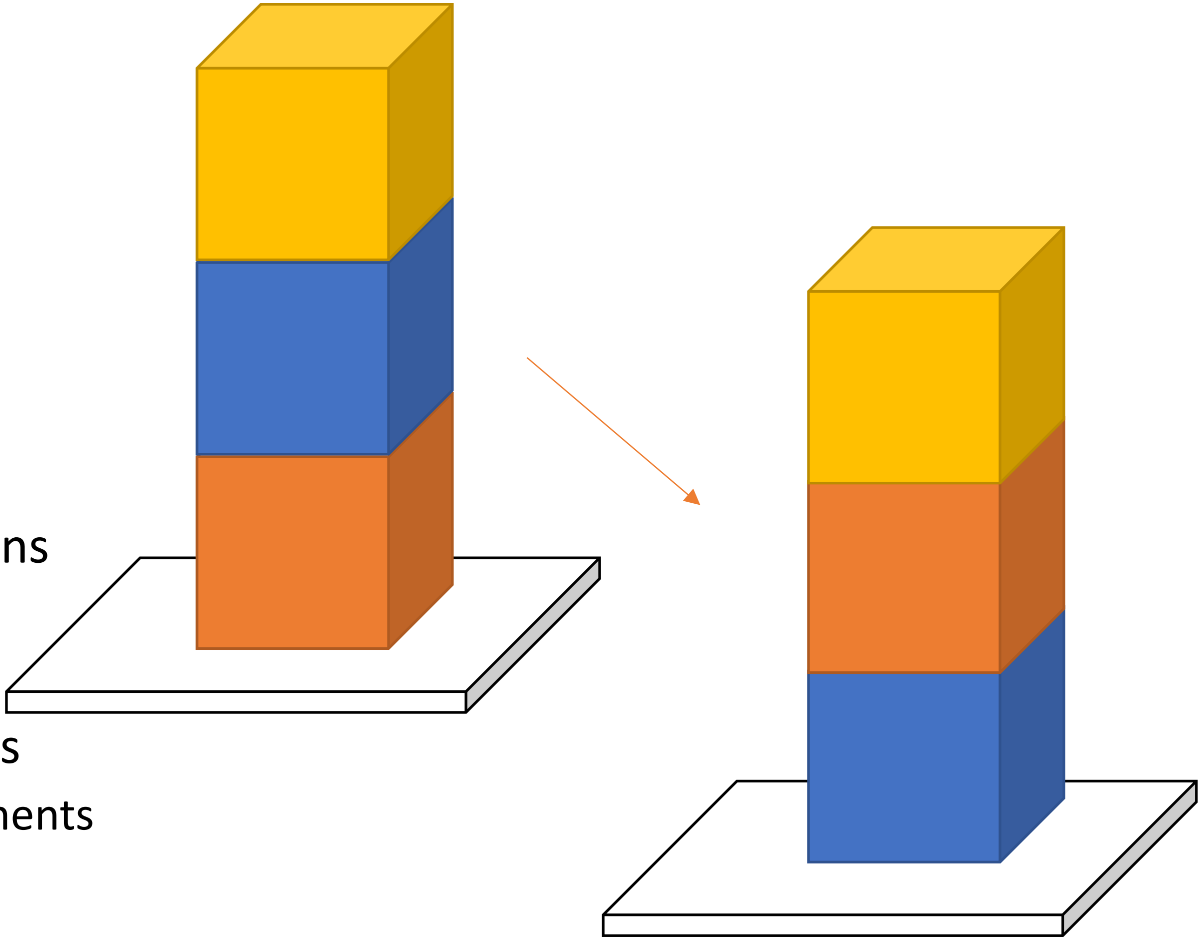
Find a set of actions that achieve the goal

when no single action will do

Planning

Setting:

- Agent
- Goal
- Problem Formulation
  - A Complex Set of Actions
    - Preconditions
    - Effects
  - A Complex Set of States
    - Propositional Statements



What we want to do?

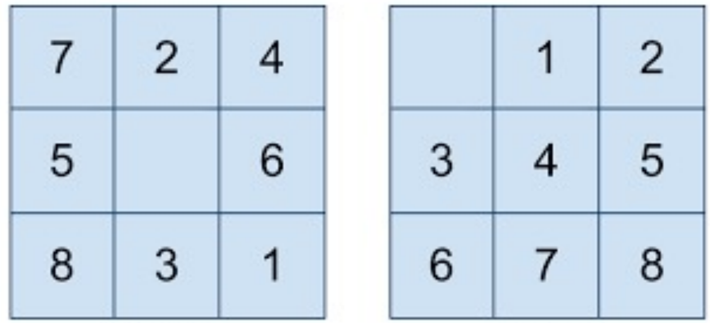
Take advantage of the structure of a problem

to construct complex plans of actions

Search algorithms for Planning

- Search and Planning often addresses similar problems and there is no clear distinction between them.
- On one hand, planning deals with more complex problems w.r.t. how actions are described, states, goals and when is difficult to provide a proper problem formulation.
- As an example, if the conditions can change planning methods are more suited to *adapt* the plan.
- On the other hand, search algorithms are often used where a it is easier to describe the problem in a “mathematical” way.
- Overall, search and planning are deeply connected and overlapped, and planning often requires some form of search and problem solving algorithms.
- Path-planning is one of those problem.

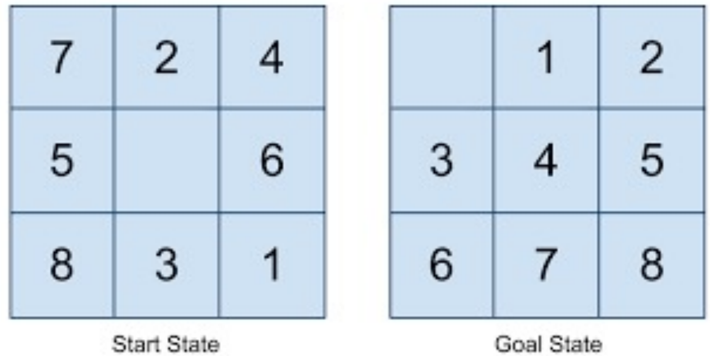
Discrete Search Problems: 8-Puzzle



Start State Goal State

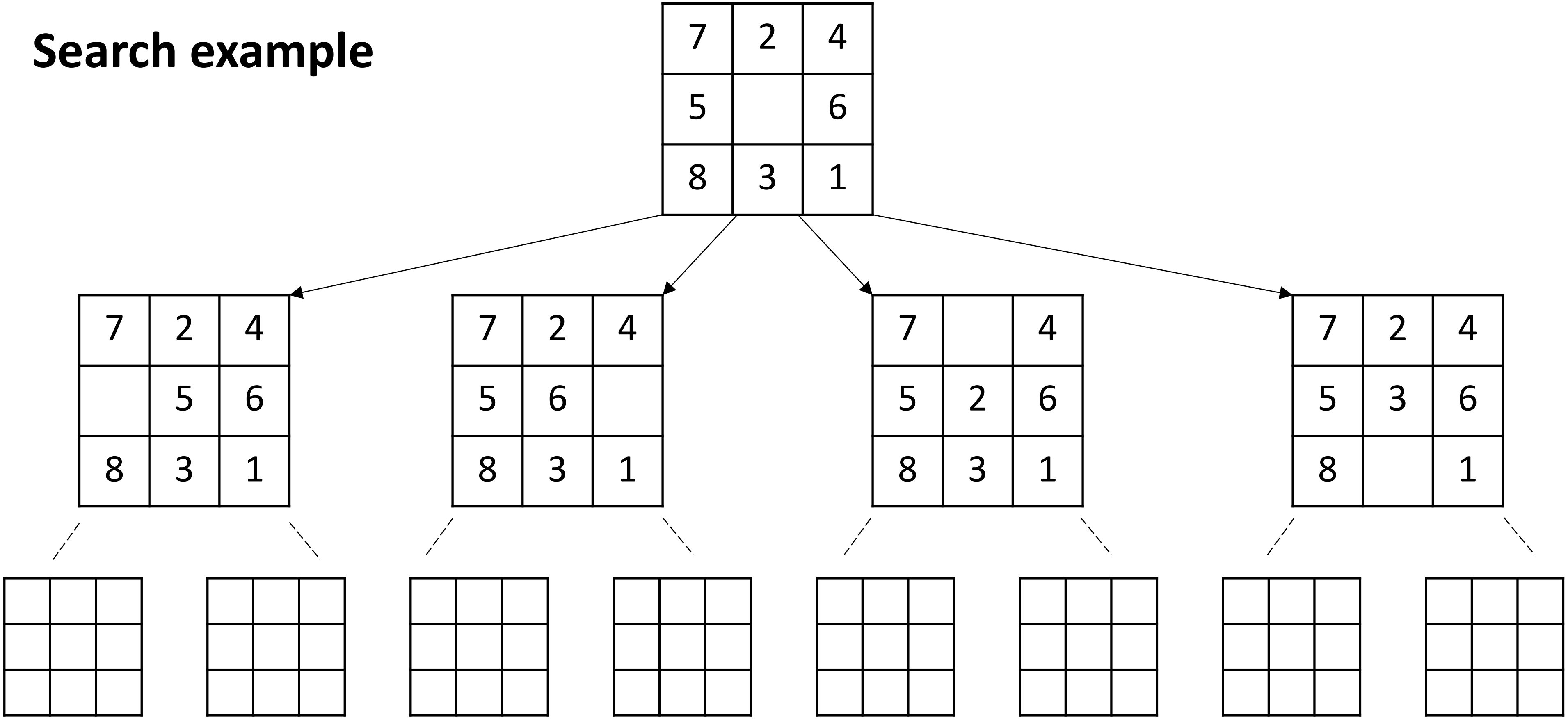
- States: location of each digits in the eight tiles + blank one
- Initial State
- Goal State
- Actions: Left, Right, Up, Down
- Transition: given a state and an action, the resulting board

Discrete Search Problems: 8-Puzzle



- States: location of each digits in the eight tiles + blank one
- Initial State
- Goal State
- Actions: Left, Right, Up, Down
- Transition: given a state and an action, the resulting board
- Goal Test: if the states are equal to the goal state
- Cost: each movement costs 1, the lowest number of tile move the lowest the cost

Search example

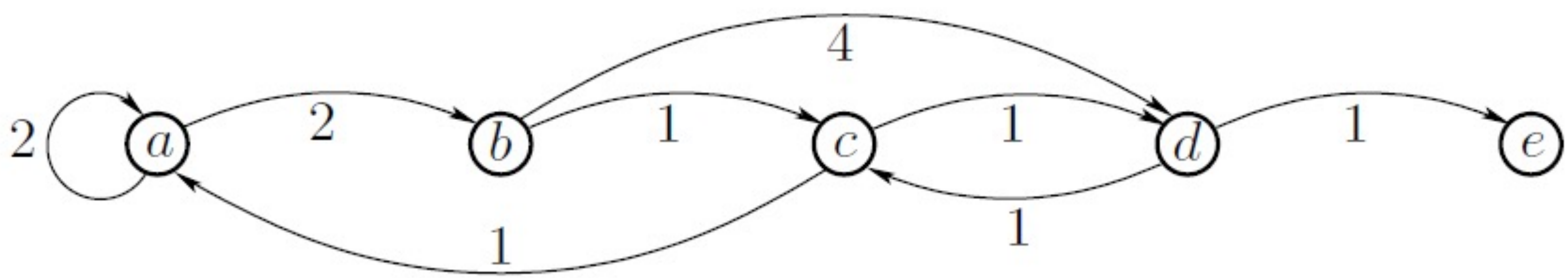


Expanding the current state by applying a legal action generating a new set of states, then…

…following up one option and putting aside others in case the first choice does not lead to a solution

State-based problem formulation

- State space defined as a set of **nodes**, each node represents a state; we assume a finite state space (and discrete)
- 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; 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
- Goal State



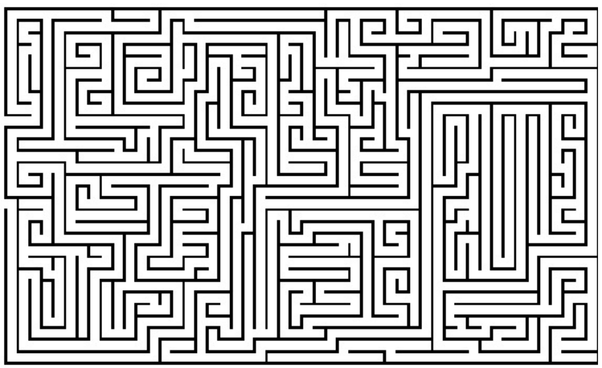
Compact representation: state transition graph G=(V,E)
(We will use "state" and "node" as interchangeable terms)

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

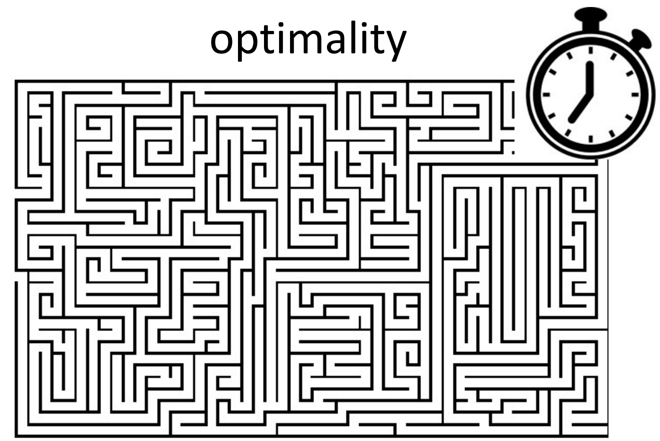


is there a path to an exit?

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

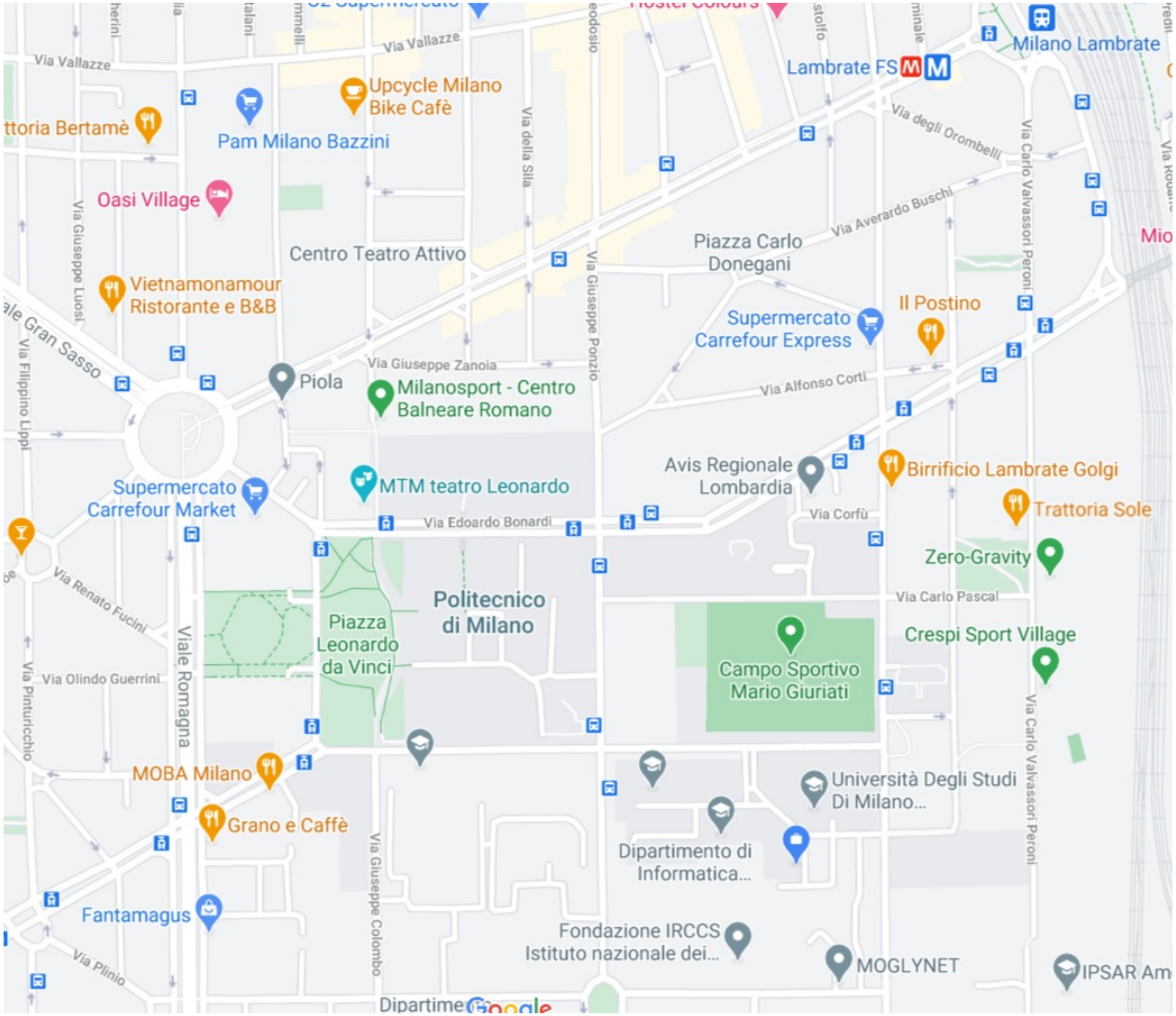
Consider a agent moving on a graph-represented environment:

- **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 agent

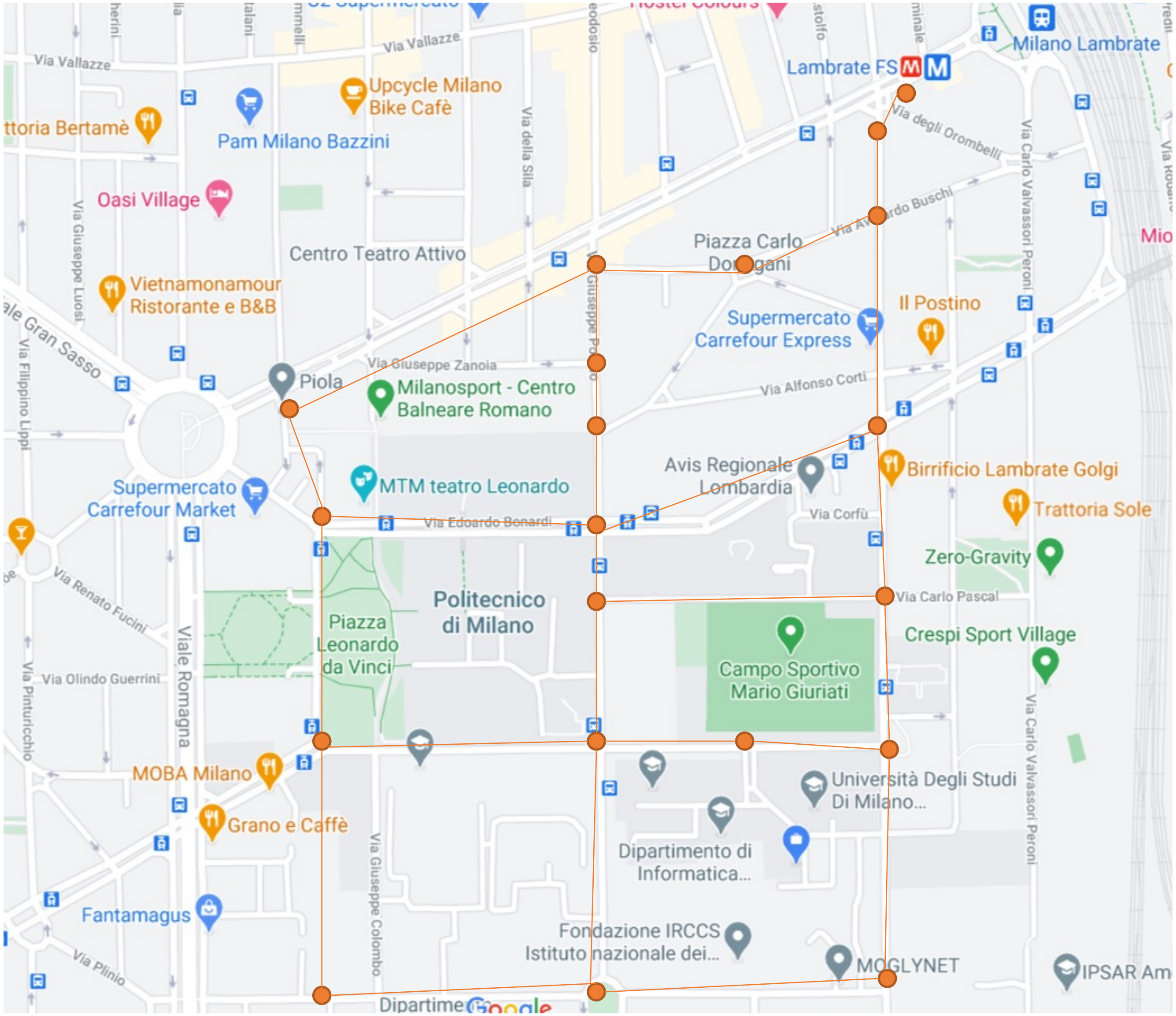
Desired solution:

- **Goal state(s)**: some location(s) to reach, …
  Find a path to the initial location to a goal one

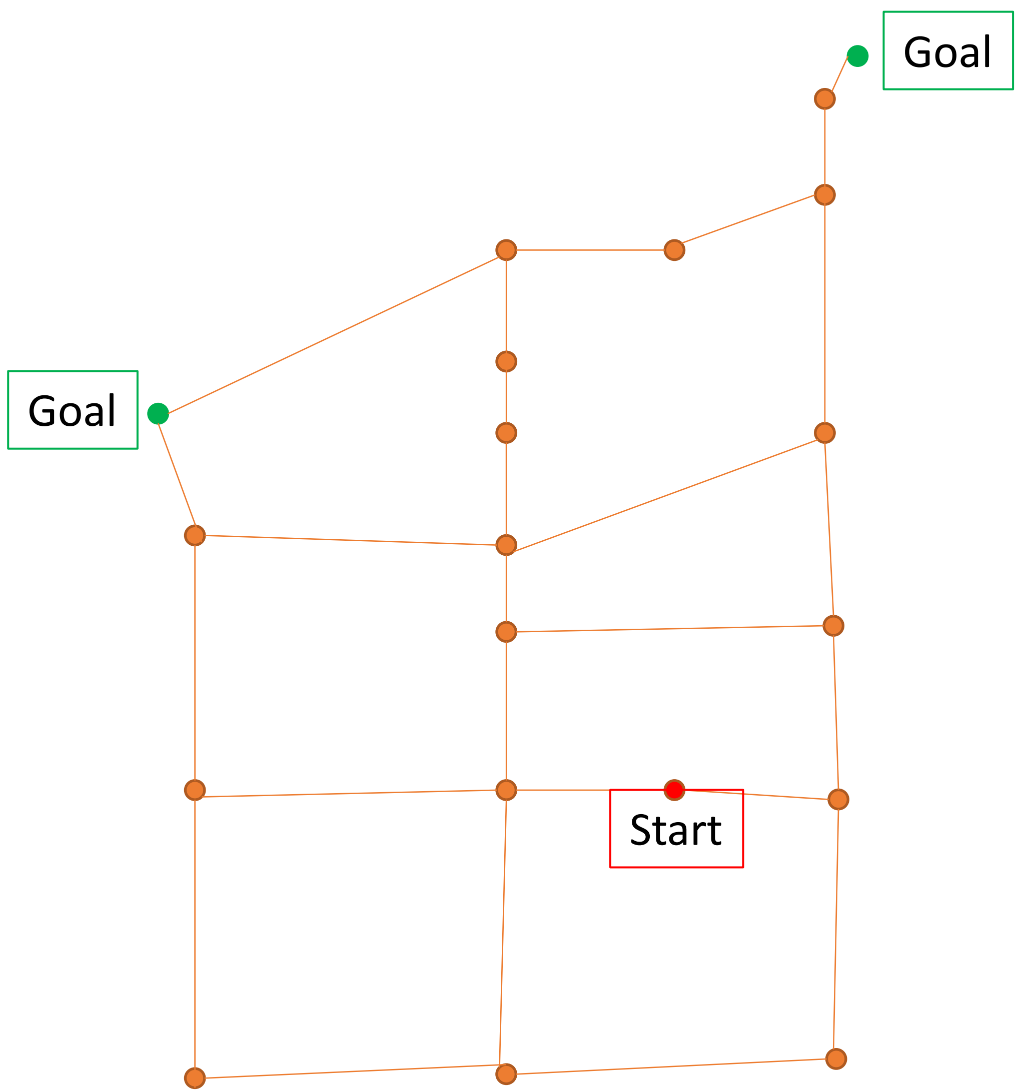
Example: going home from the CS department with METRO



Example: going home from the CS department with METRO



Example: going home from the CS department with METRO



Problem example

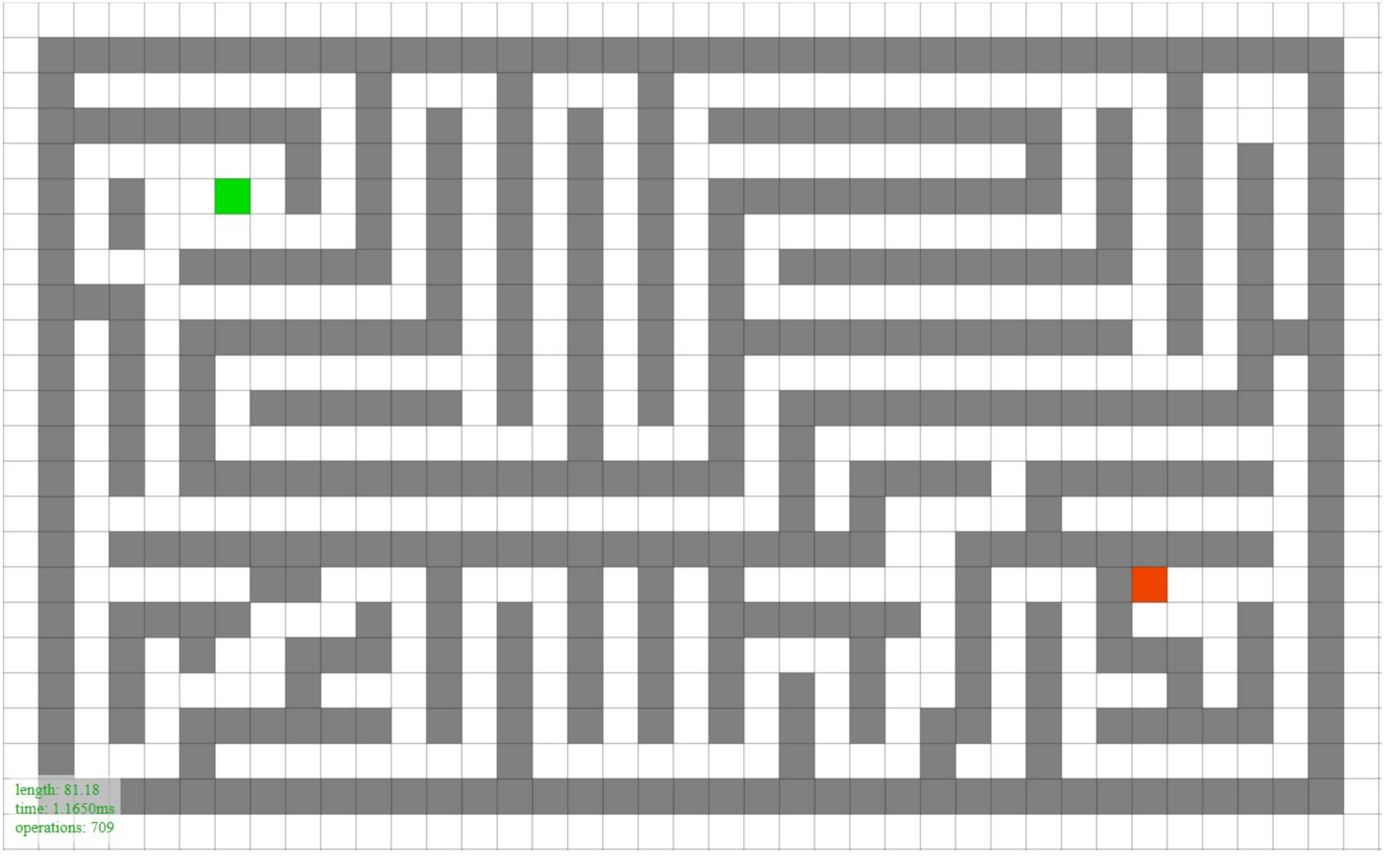
Consider a mobile robot moving on a grid environment:

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

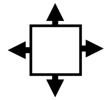
Desired solution:

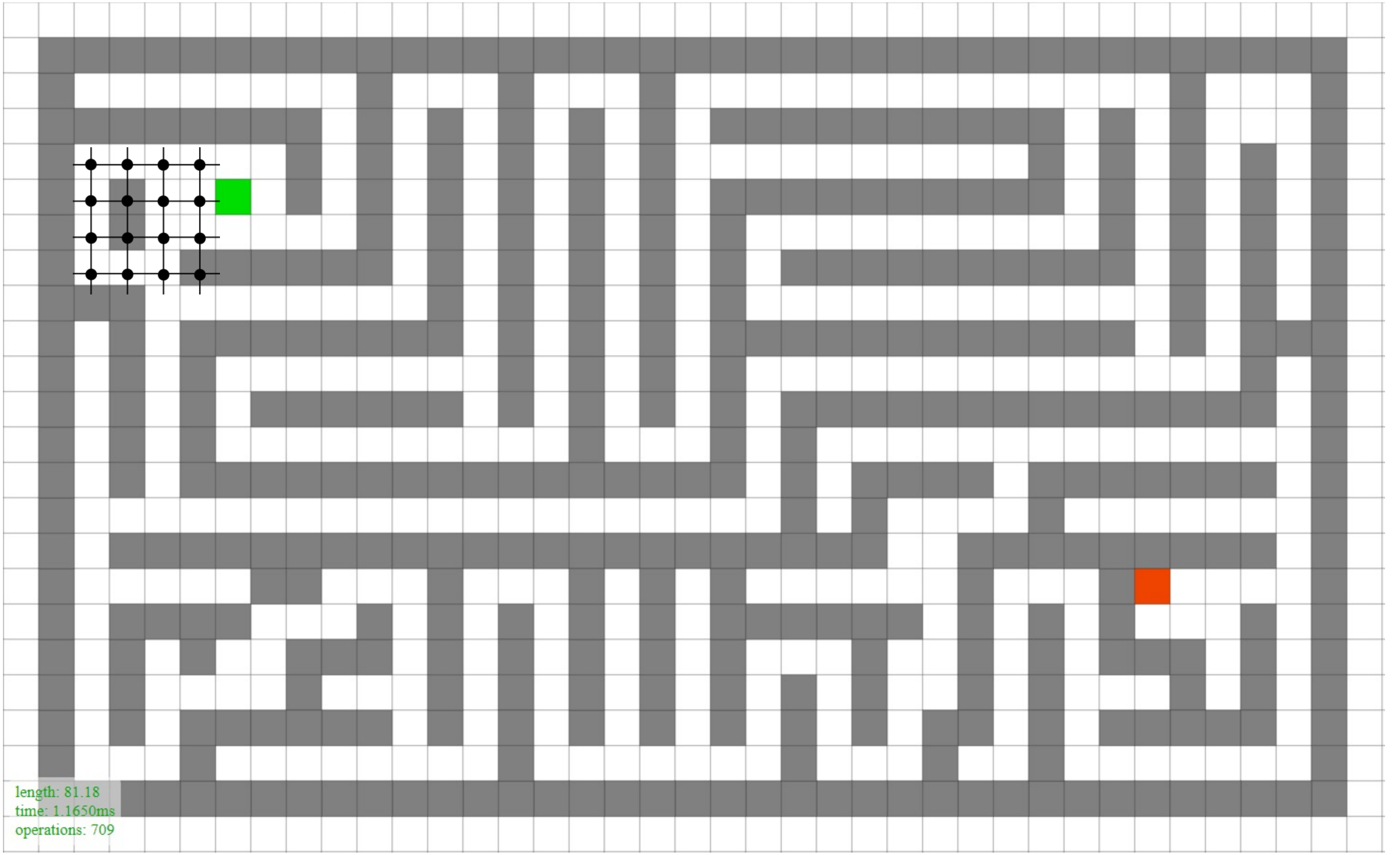
- **Goal state(s)**: some location(s) to reach
- Find a path to the initial location to a goal one

Problem Example

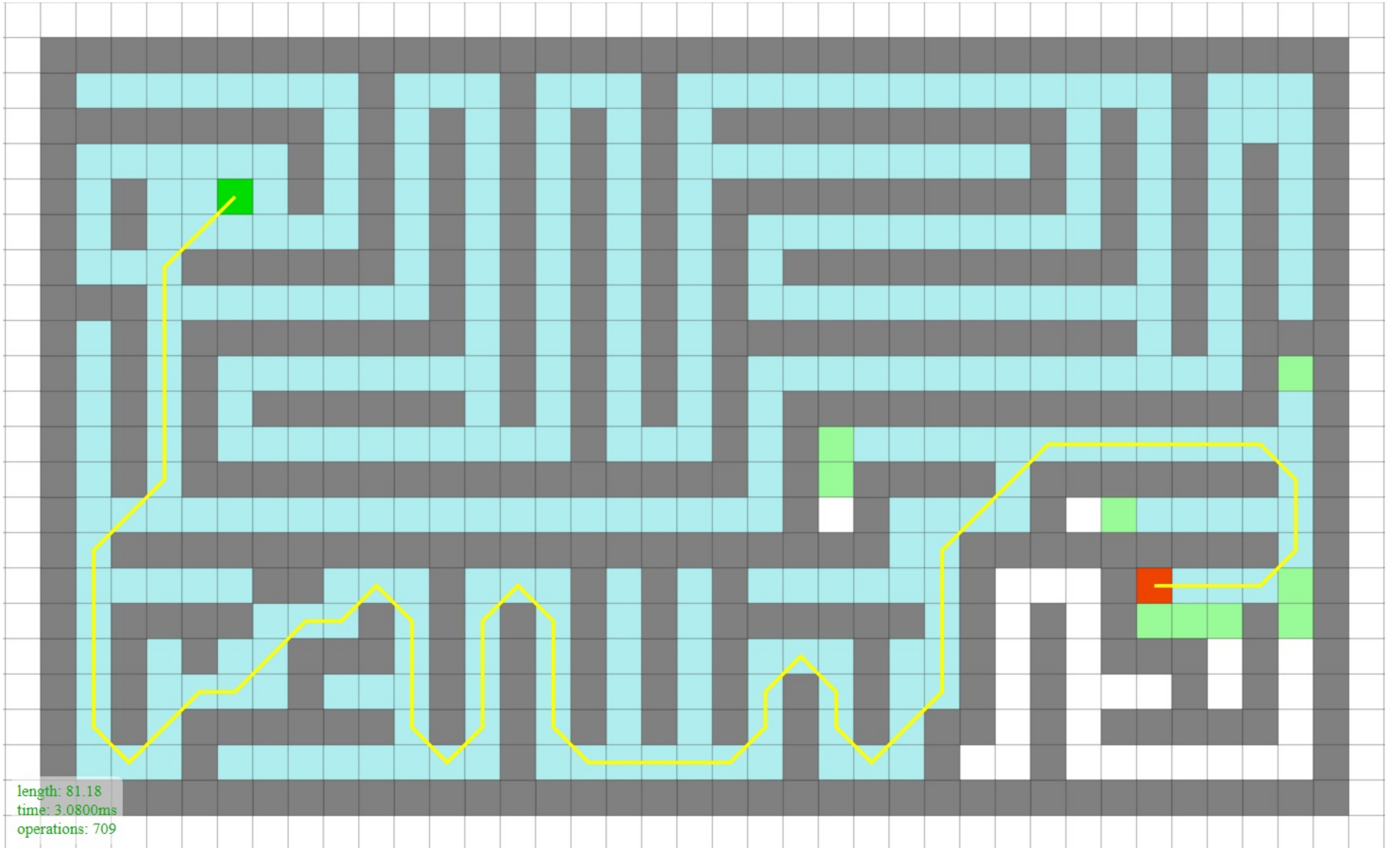


Problem Example

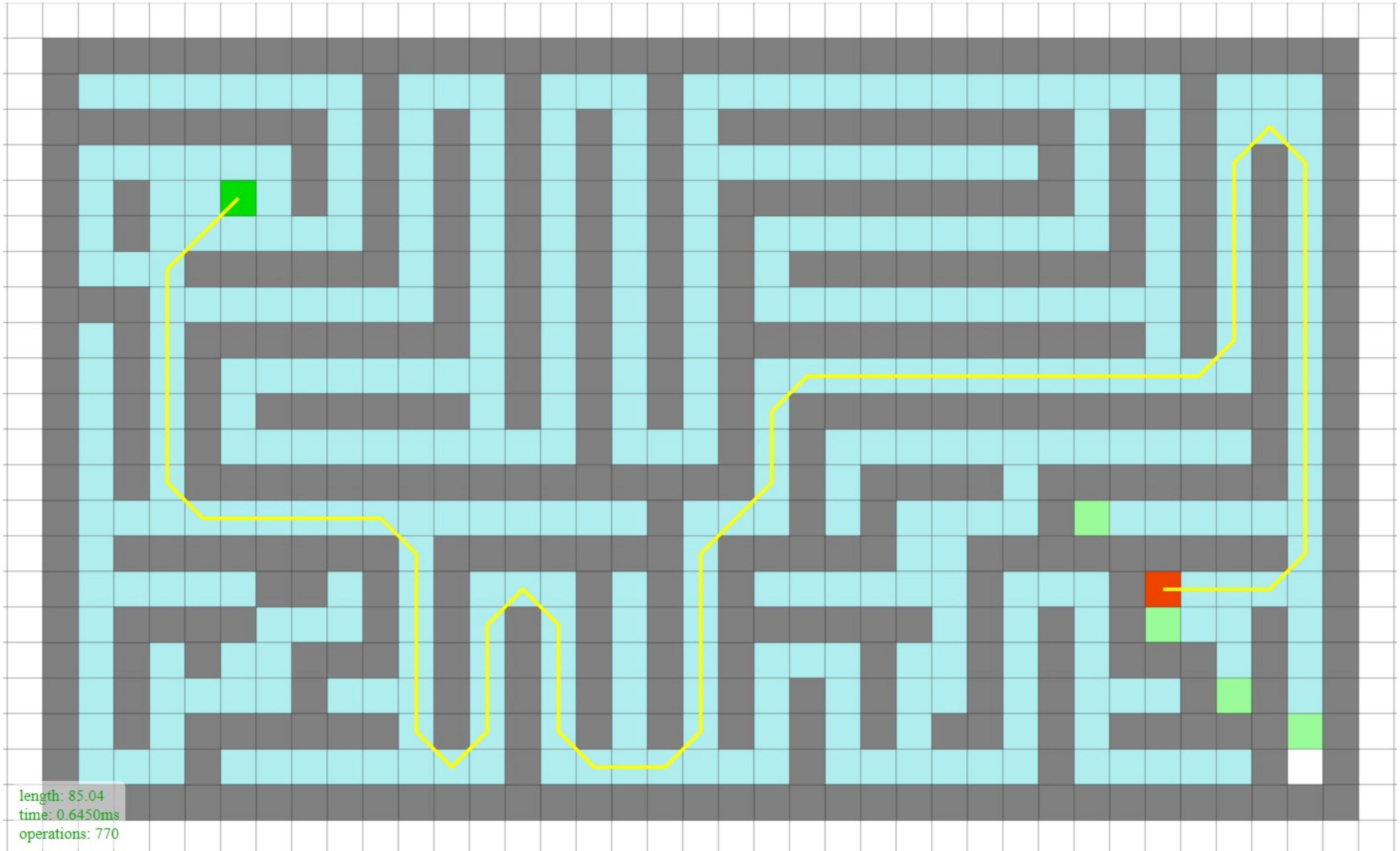




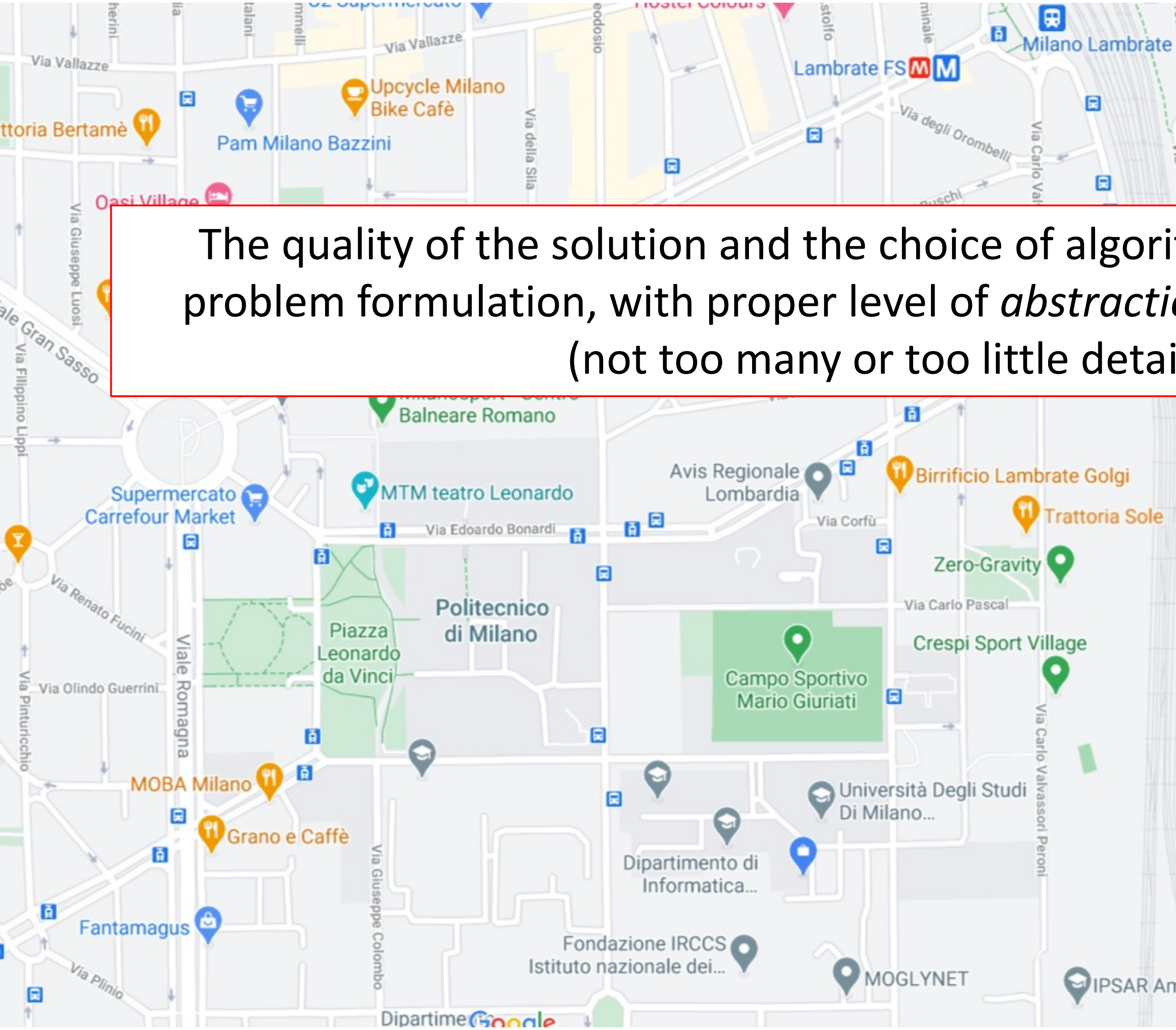
A solution



And here? Changing a few tiles, different solution

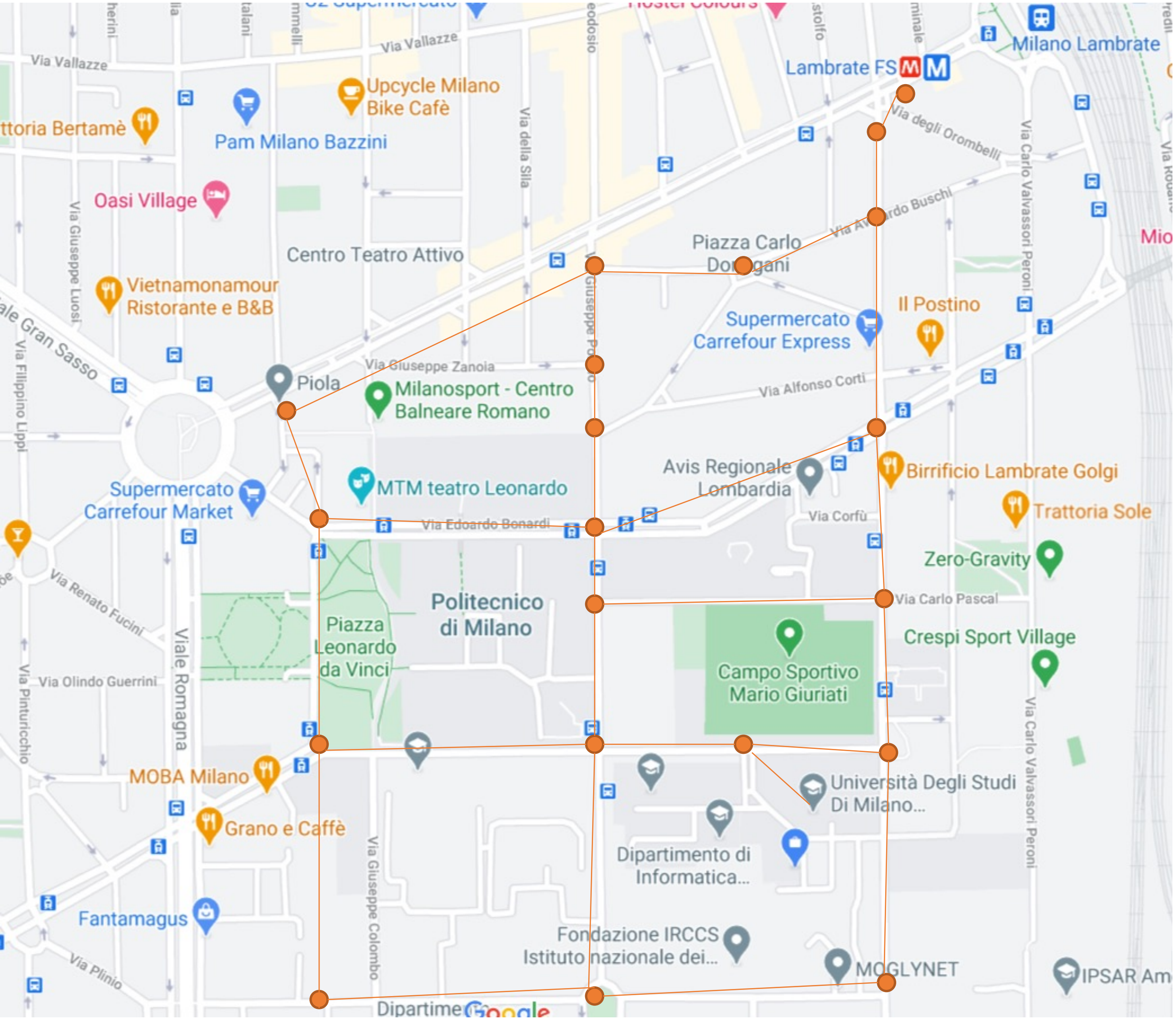


One problem, many representations



The quality of the solution and the choice of algorithms rely on a proper problem formulation, with proper level of *abstraction* needed for the task (not too many or too little details)

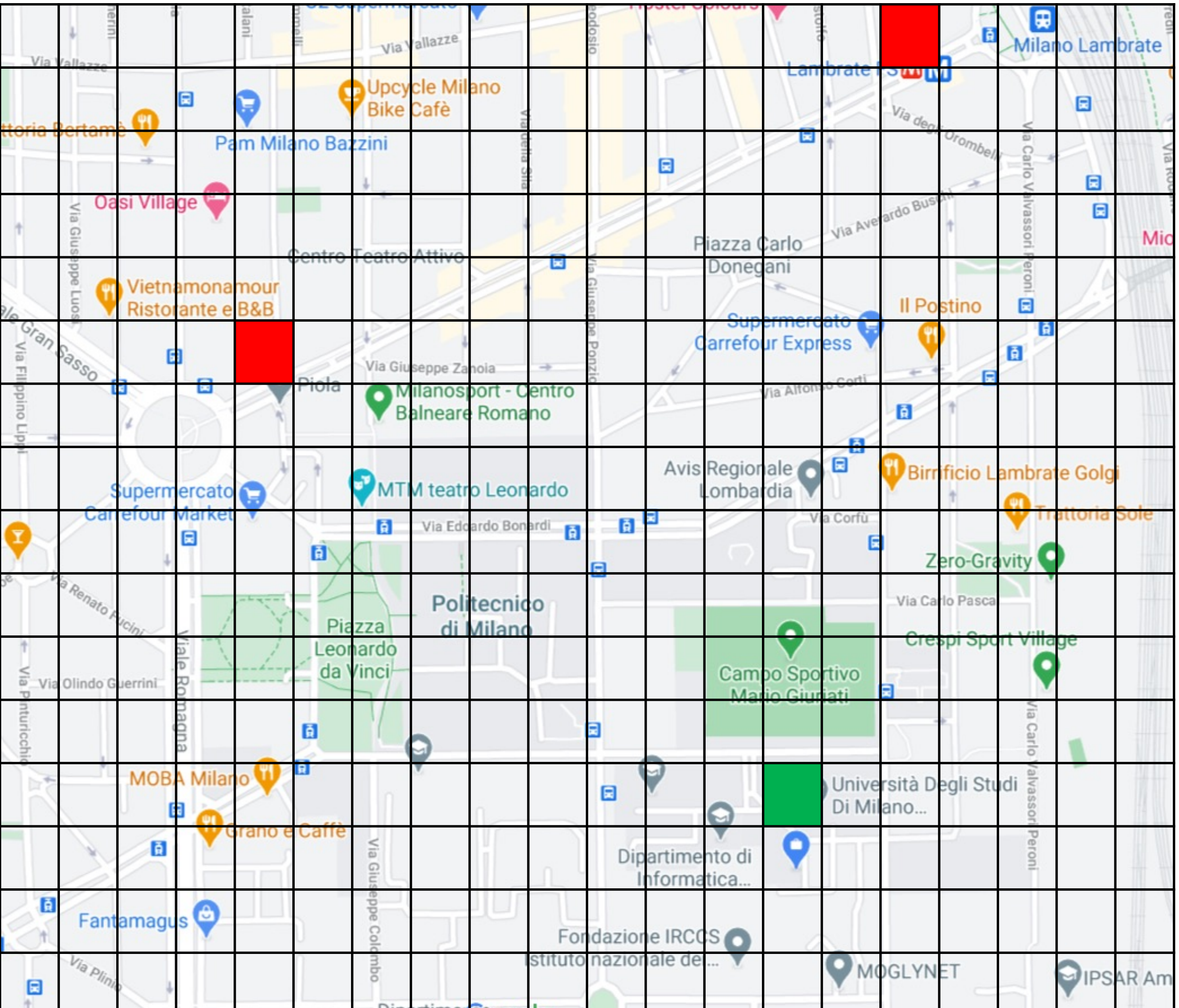
One problem, many representations



What type of representation?

- With which granularity?
- Shall I represent other nearby stations (Loreto, Udine?)
- Shall I represent also the bus stops?
- Trams?
- Main central stations?
- All Milan city map?
- Shall I represent all crossings and traffic lights?
- How about directions inside the campus?
- How about directions inside the building?

One problem, many representations



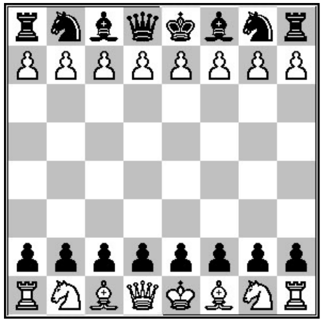
What type of representation?

- Grid map?
- How big the grid?
- Which distance?
  - Euclidean
  - Manhattan
  - ?
- Shall I represent all crossings and traffic lights?
- How about directions inside the campus? (different grid size?)
- How about directions inside the building? (smaller?)

;

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:



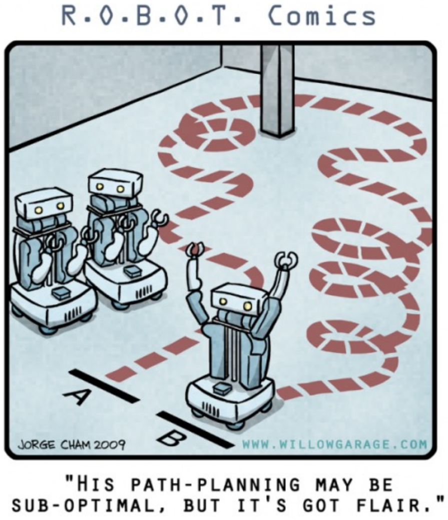
  - **Chess board**: approx. 10^{47} states
  - We can specify the initial state and the transition function in some compact form (e.g., set of rules to generate next states)
  - The planning problem “unfolds” as search progresses

- We need an efficient procedure for *goal checking*

General features of search algorithms

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

- feasibility: when a goal node is visited the path that led to that node is returned
- 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



Given a state and the path followed to get there, the next node to explore is chosen using a *state strategy*

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

  - Completeness:
    If there is a solution, is the algorithm guaranteed to find it?
    - Systematic:
      If the state space is finite, will the algorithm visit all reachable state
      (so finding a solution if a solution exists?)
  - Optimality: does the strategy find an optimal solution?
  - Space complexity:
    How much memory is needed to find a solution?
  - Time complexity?
    How long does it takes?

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

Soundness

- Optimality: *does the returned solution lead to a goal with minimum cost?*

Maybe we are not always looking for the optimal solution…

…for some problems, we may look for other features

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

- Example:
    - Feasibility: *does the returned solution lead to a goal?*
    - Optimality: *does the returned solution lead to a goal with minimum cost?*

    (We may need other features from the algorithm e.g., approximation)

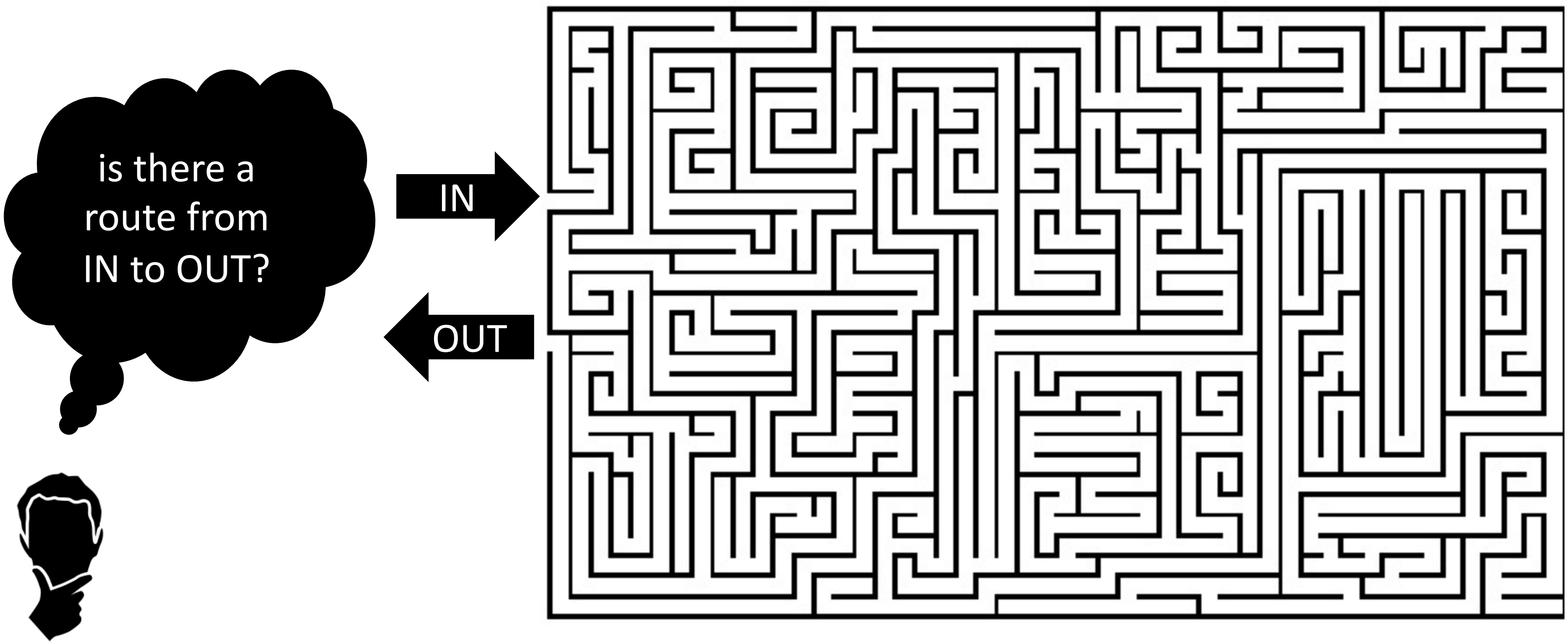
Completeness and the systematic property

- If a solution exists, does the algorithm find it?

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

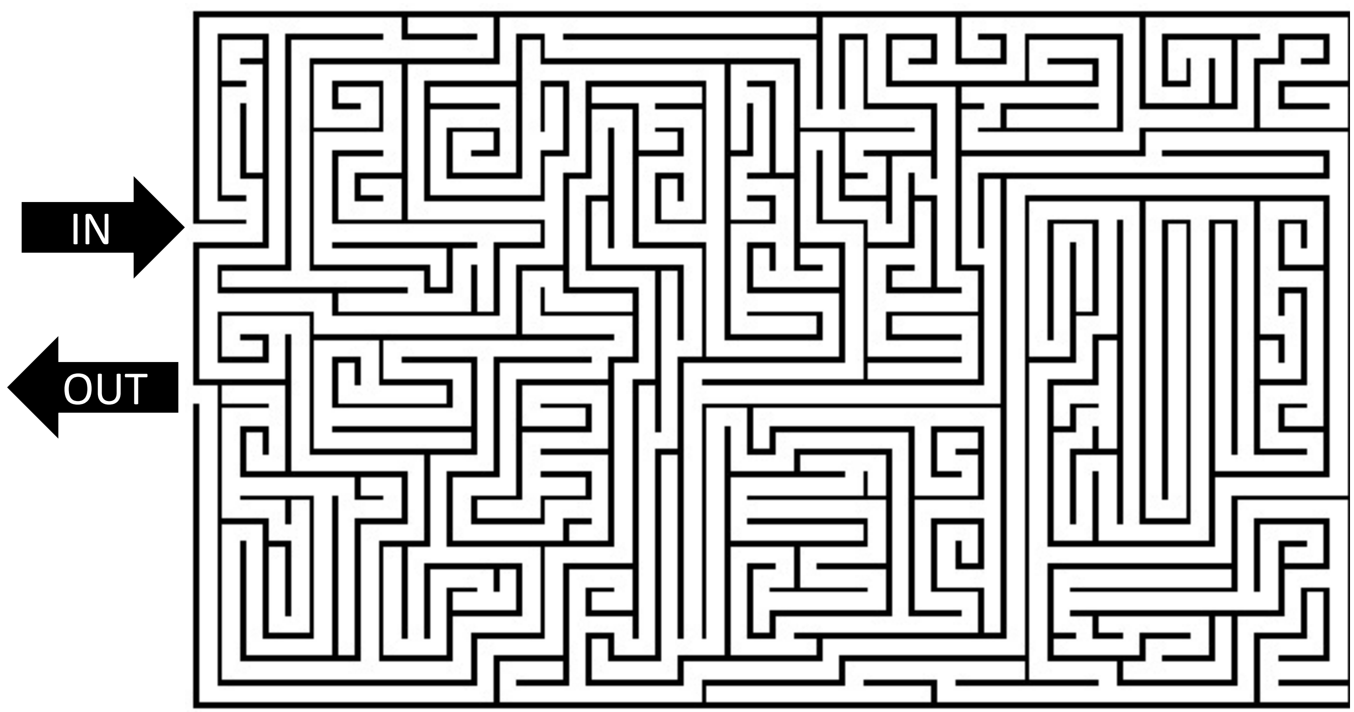
- If the state-space is finite, ensuring that no redundant exploration occurs is sufficient to make the search systematic.
- If the state space is infinite, we can ask if the search is systematic:
  - If there is a solution, the search algorithm must report it in finite time
  - if the answer is no solution, 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 – all reachable vertex is explored - (this definition is sound under the assumption of countable state space)

Visual example



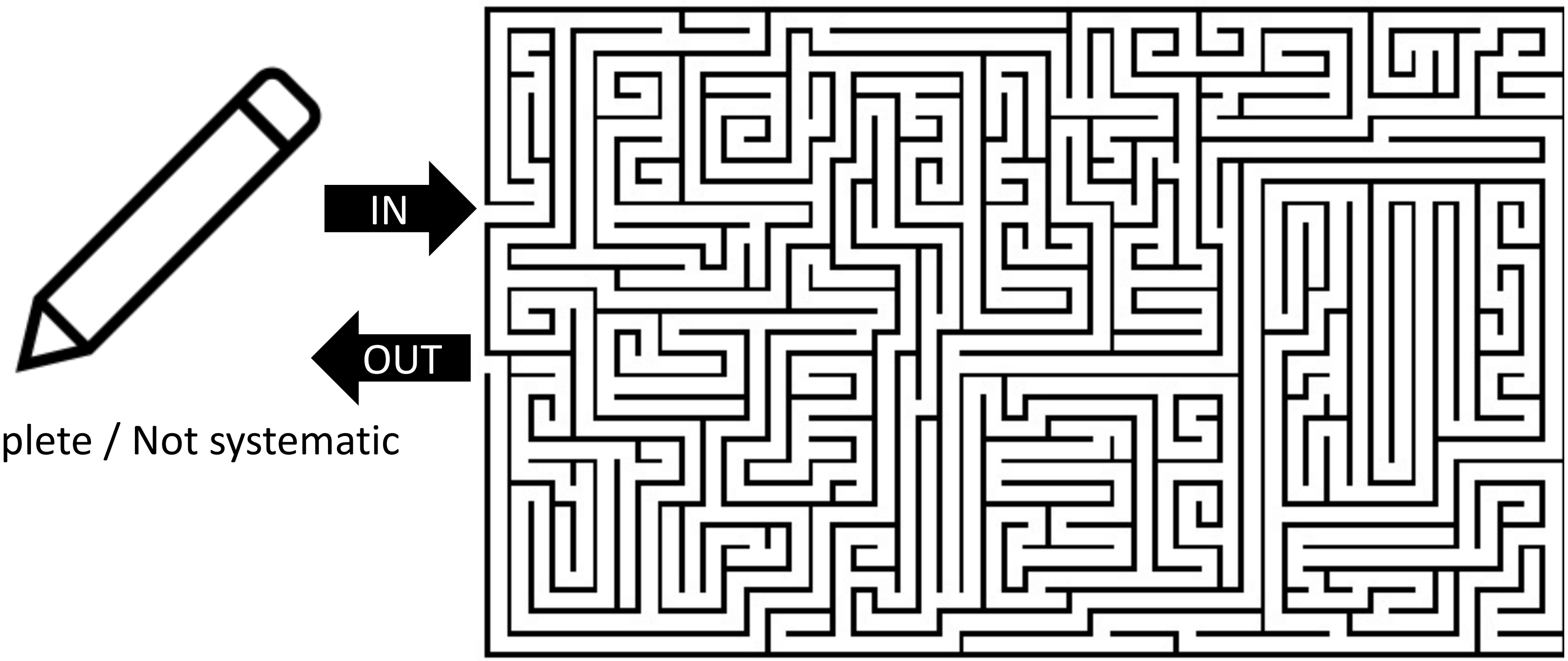
Visual example

Complete / Systematic



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

Visual example

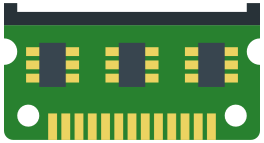


Not complete / Not systematic

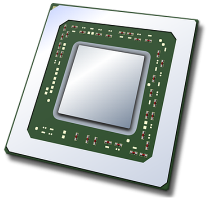
- Searching along a **single** trajectory, eventually gets stuck in a dead end (or find a solution if we are lucky)

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)?



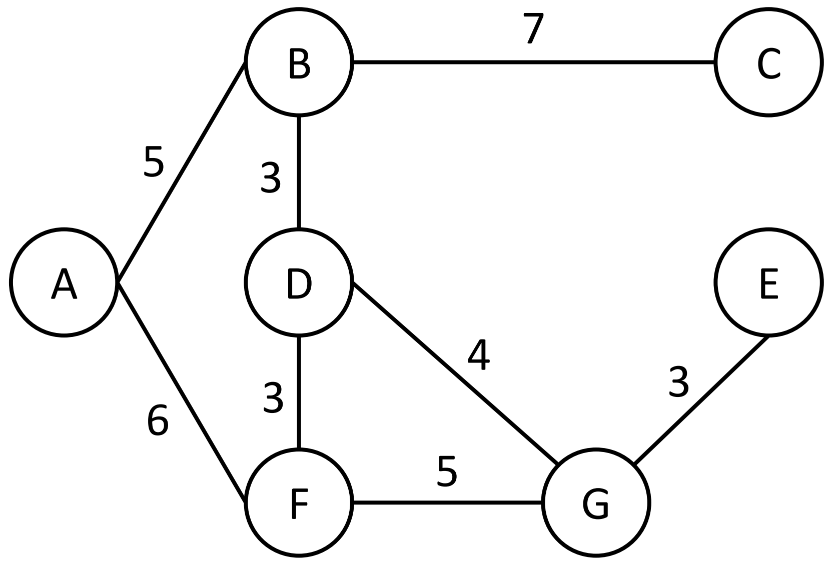
- Asymptotic trend:
  - We measure complexity with a function $f(n)$ of the input size
  - For analysis purposes, the "Big O" notation is convenient:

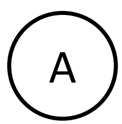
$$\text{A function } f(n) \text{ is } O(g(n)) \text{ if } \exists k > 0, n_0 \text{ such that } f(n) \leq k g(n) \text{ for } n > n_0$$

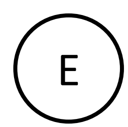
- An algorithm that is $O(n^2)$ is better than one that is $O(n^5)$
- If $g(n)$ is an exponential, the algorithm is not efficient

Running example

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

State-transition graph: 

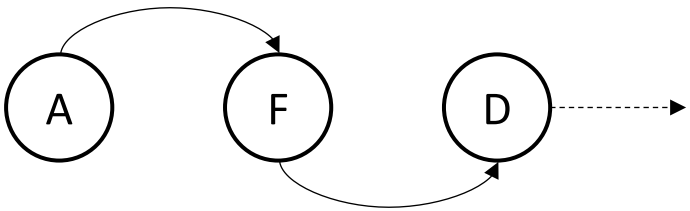
Initial state: 

Desired solution: any path to goal state 

- 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

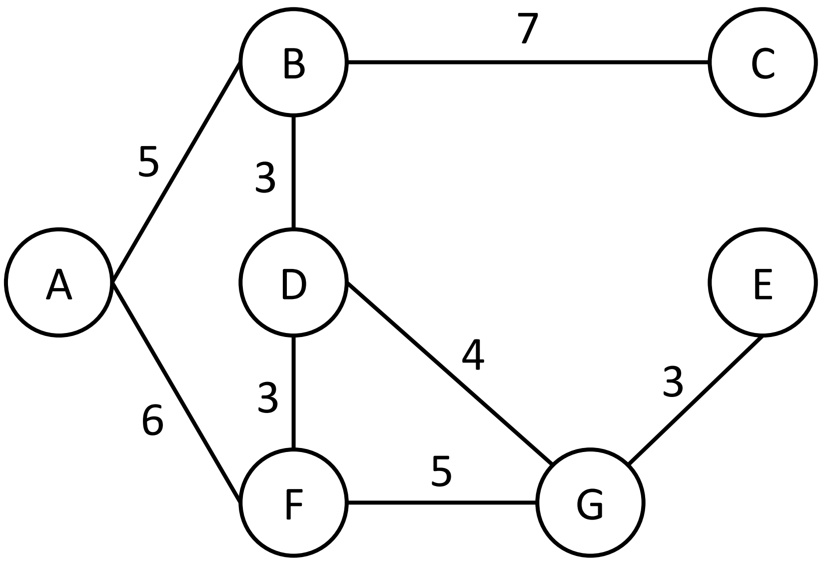
- The different search algorithms are substantially characterized by the answer they provide to the following question:



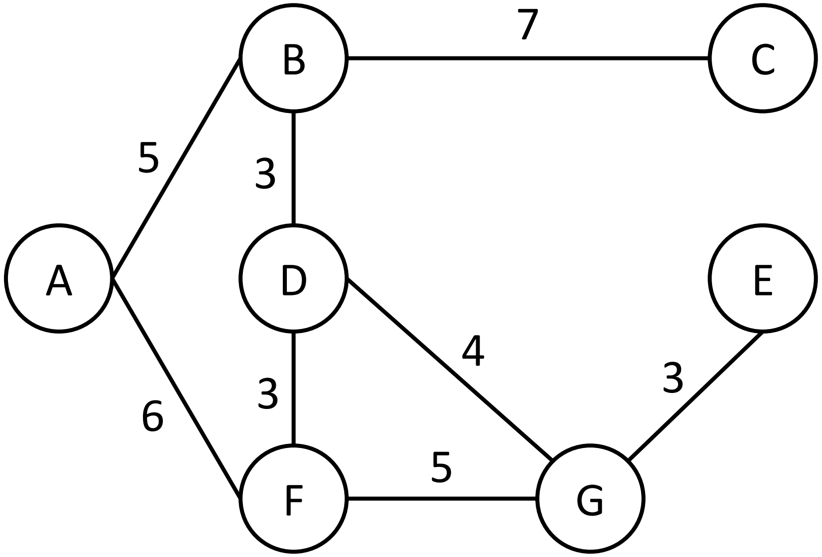
Given what I searched so far,
where to search next?
(search strategy)

- 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)

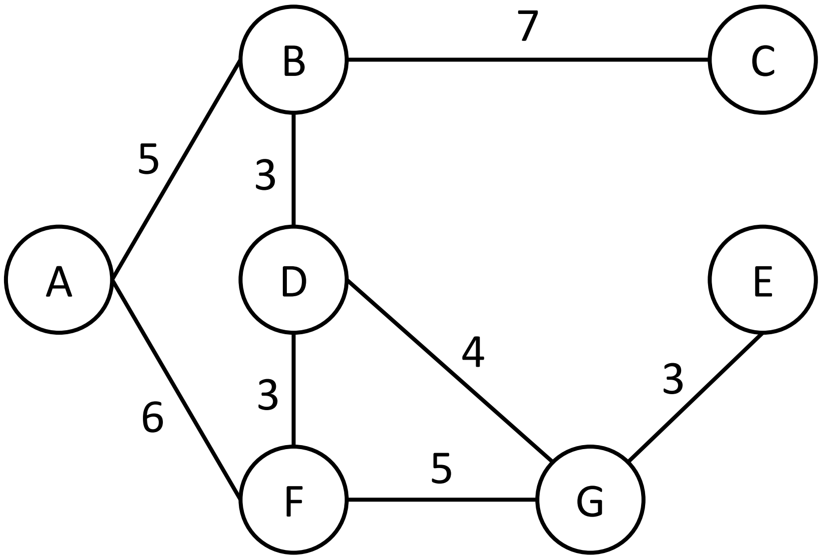


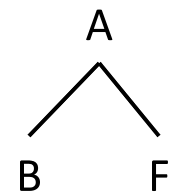
Depth-First Search (DFS)



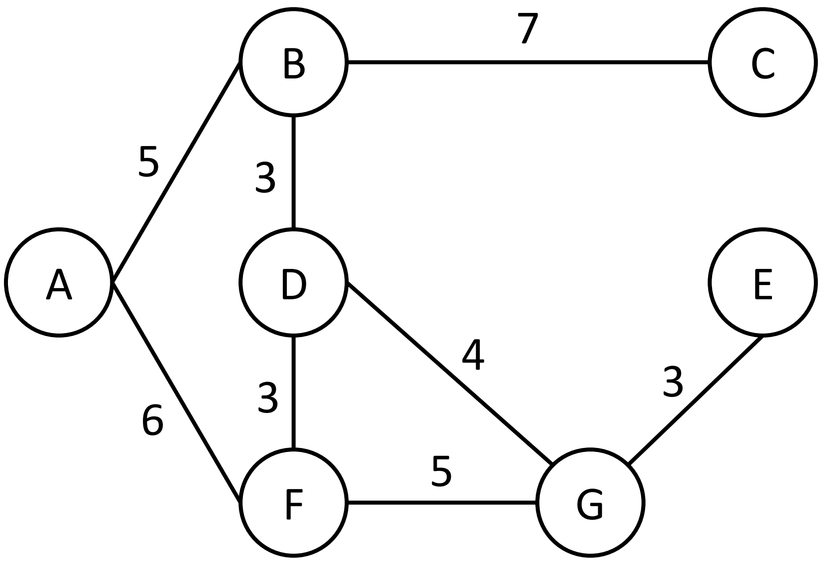
A

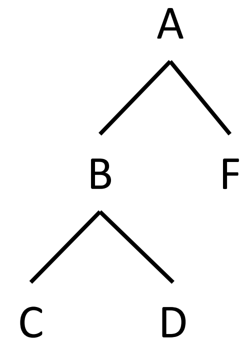
Depth-First Search (DFS)



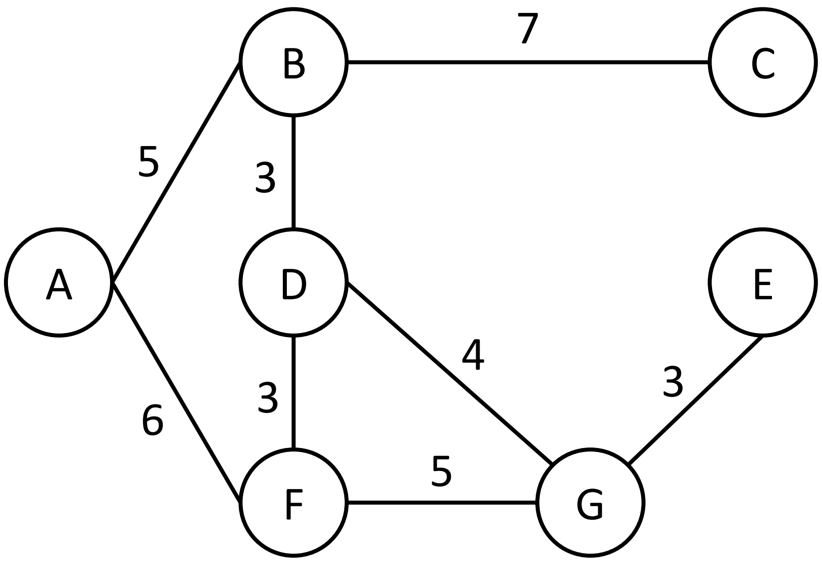


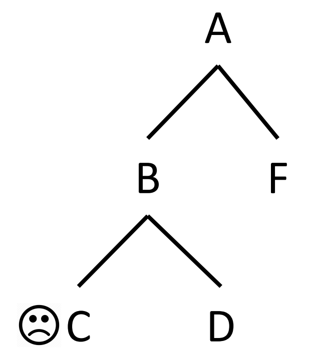
Depth-First Search (DFS)



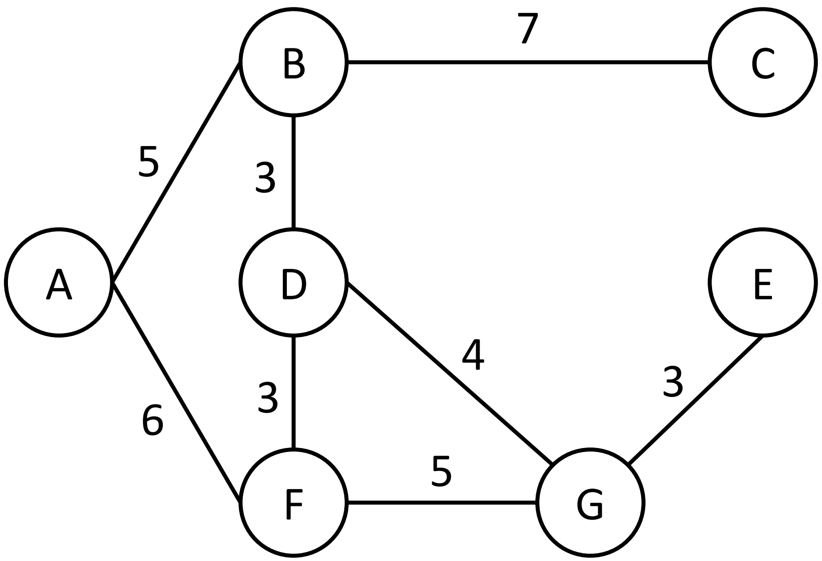


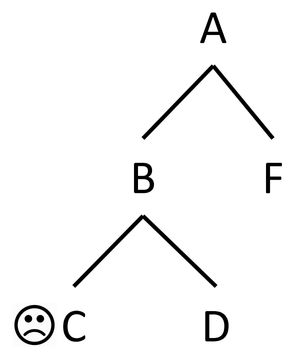
Depth-First Search (DFS)





Depth-First Search (DFS)



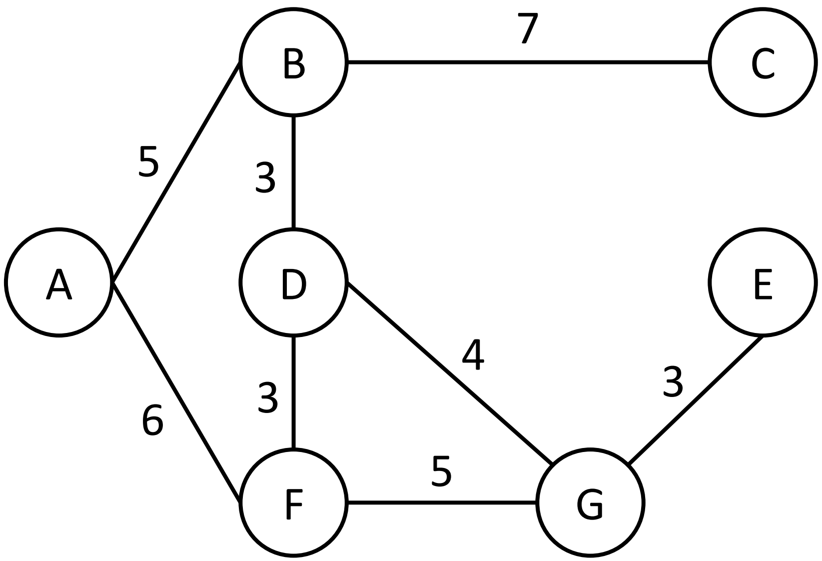


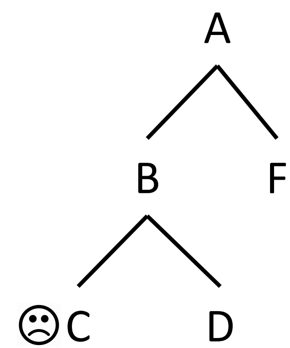
- A Depth-First Search (DFS) chooses the deepest node in the search tree (How to break ties? For now lexicographic order)

- 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)

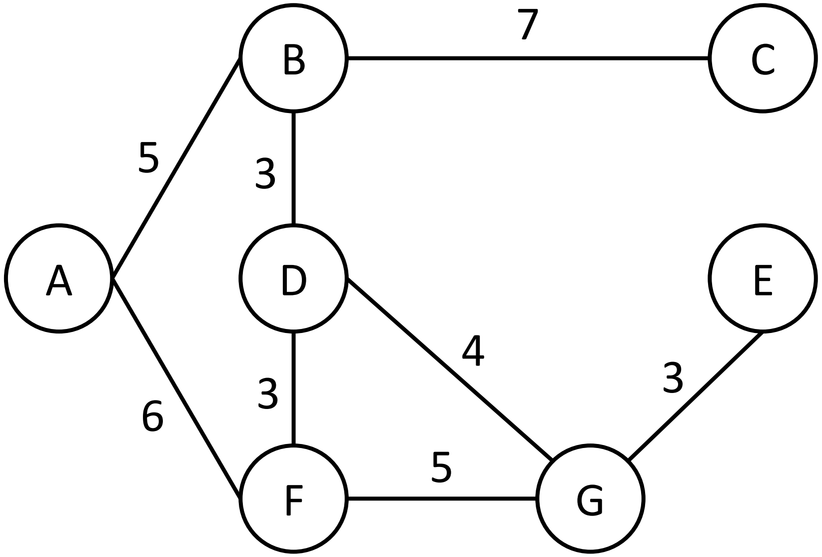


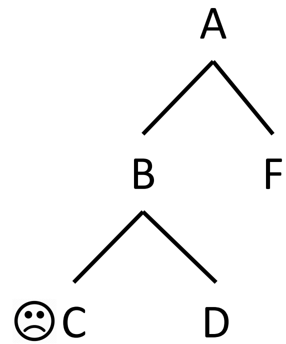


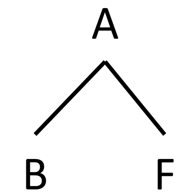
A

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





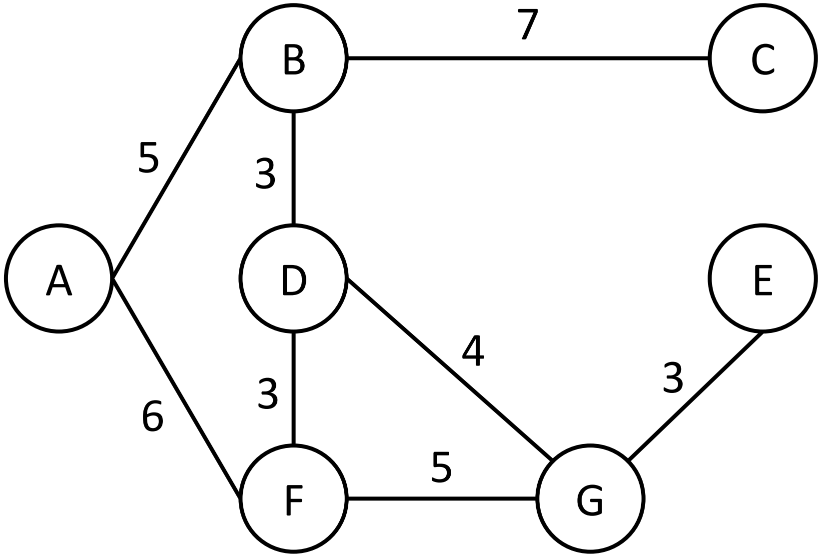


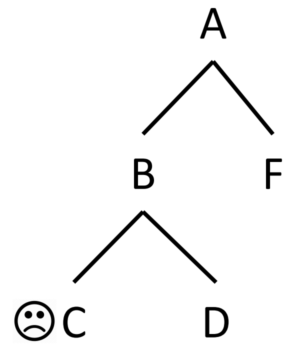
- A Depth-First Search (DFS) chooses the deepest node in the search tree (How to break ties? For now lexicographic order)

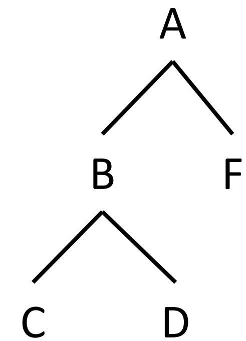
- A dead end stopped the search, DFS seems not complete. Can we fix this?

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

Depth-First Search (DFS)





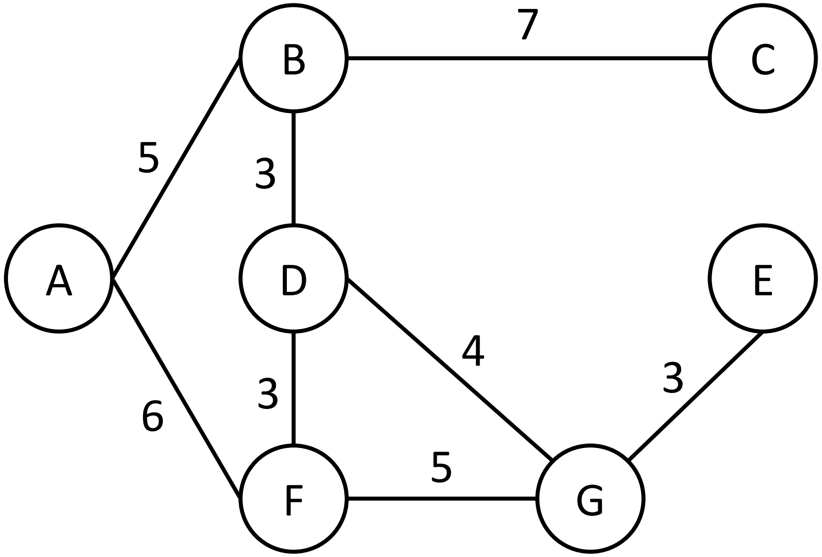


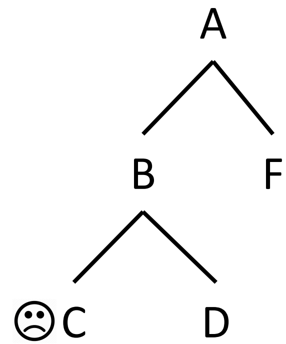
- A Depth-First Search (DFS) chooses the deepest node in the search tree (How to break ties? For now lexicographic order)

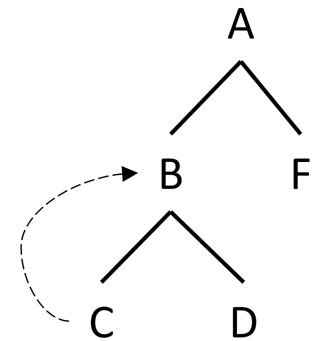
- A dead end stopped the search, DFS seems not complete. Can we fix this?

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

Depth-First Search (DFS)

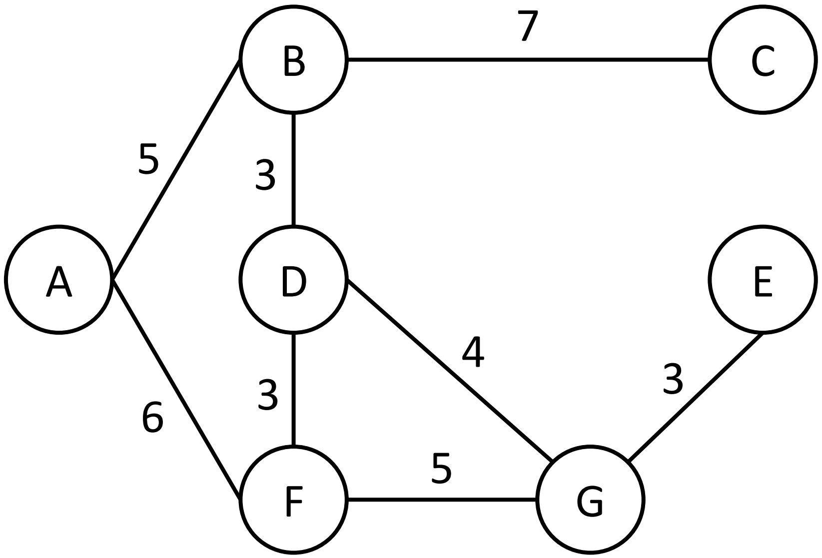


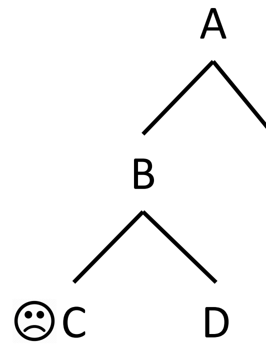


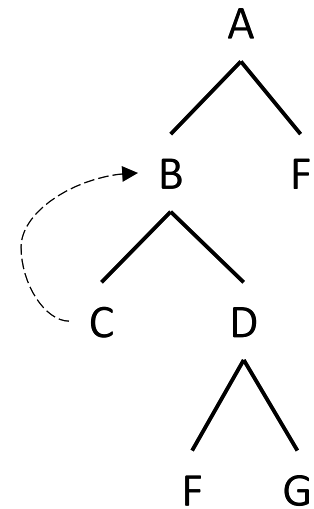


- A Depth-First Search (DFS) chooses the deepest node in the search tree (How to break ties? For now lexicographic order)
- A dead end stopped the search, DFS seems not complete. Can we fix this?
- Let’s endow our DFS with **backtracking**: a way to reconsider previously evaluated decisions

Depth-First Search (DFS)





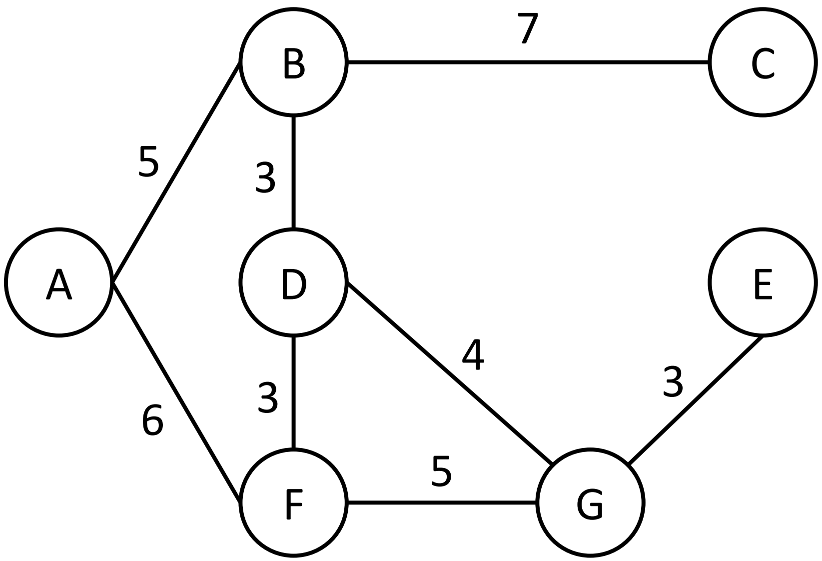


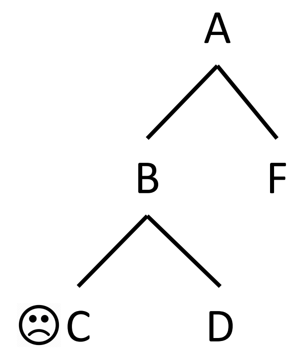
- A Depth-First Search (DFS) chooses the deepest node in the search tree (How to break ties? For now lexicographic order)

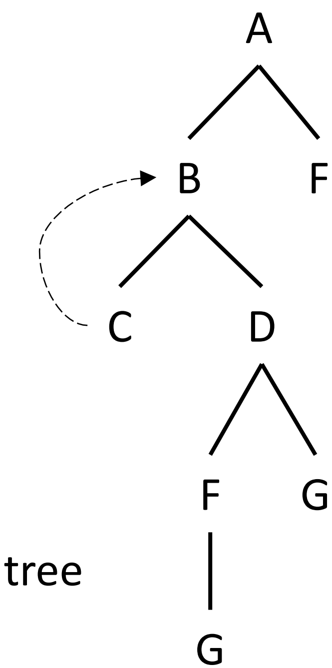
- A dead end stopped the search, DFS seems not complete. Can we fix this?

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

Depth-First Search (DFS)





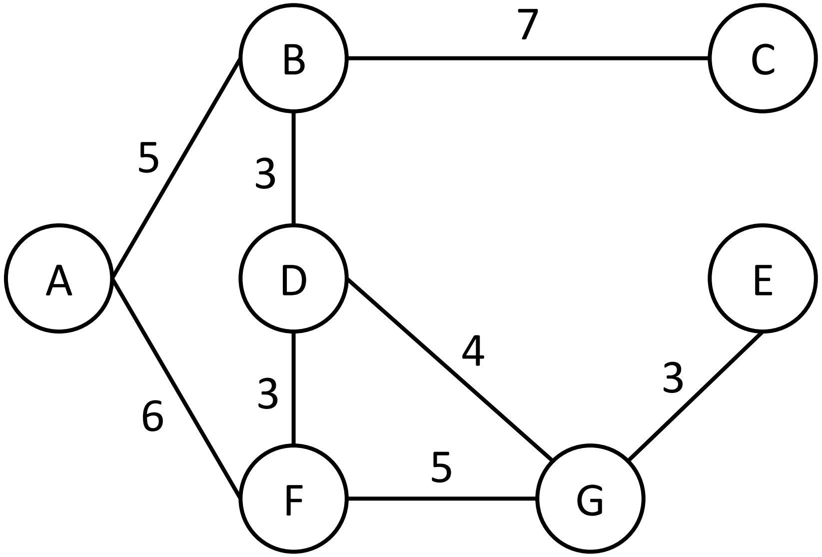


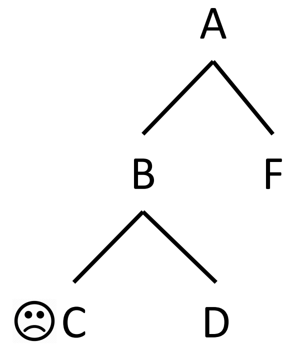
- A Depth-First Search (DFS) chooses the deepest node in the search tree (How to break ties? For now lexicographic order)

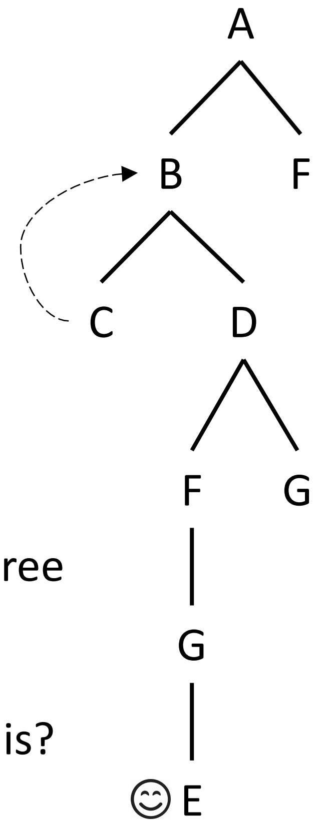
- A dead end stopped the search, DFS seems not complete. Can we fix this?

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

Depth-First Search (DFS)







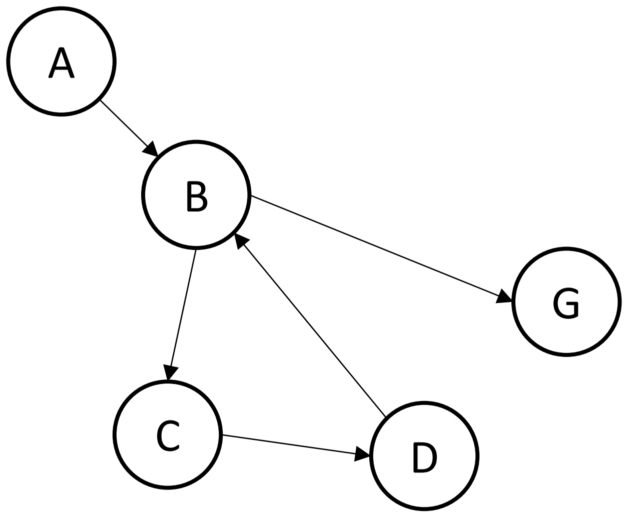
- A Depth-First Search (DFS) chooses the deepest node in the search tree (How to break ties? For now lexicographic order)

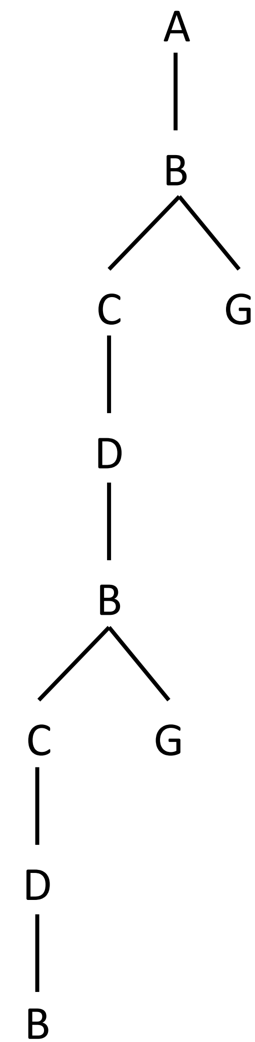
- A dead end stopped the search, DFS seems not complete. Can we fix this?

- 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) and Loops



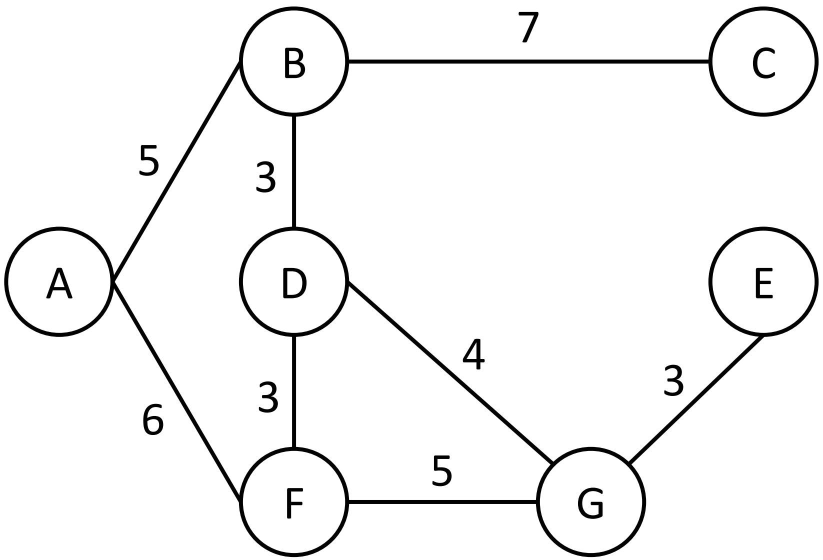


- DFS with loops – non systematic / complete
- We are **avoiding loops** on the same branch (loops are redundant paths)

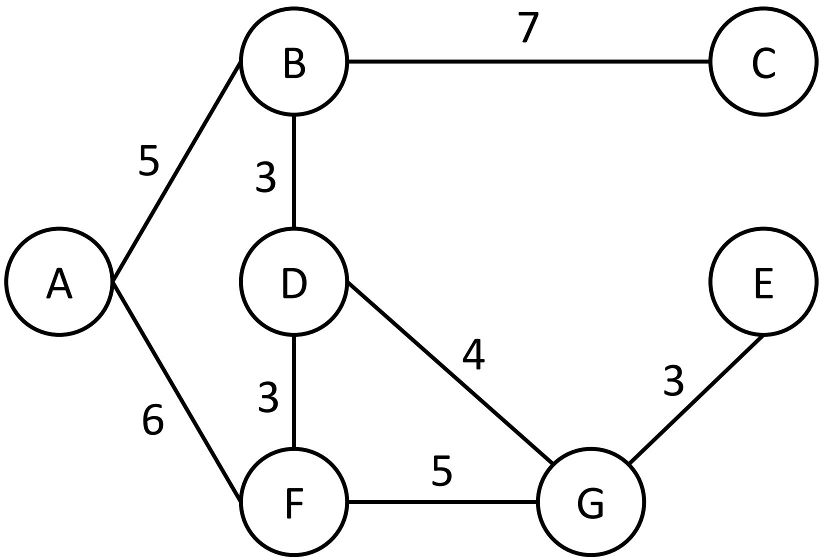
Depth-First Search (DFS)

- DFS with loops removal and BT is sound and complete (for finite spaces)
- 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)

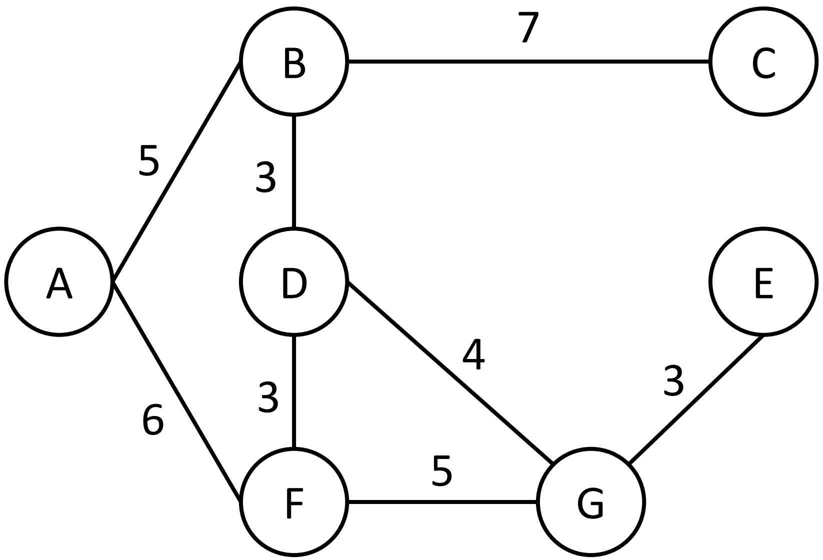


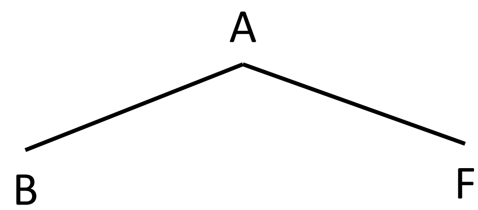
Breadth-First Search (BFS)



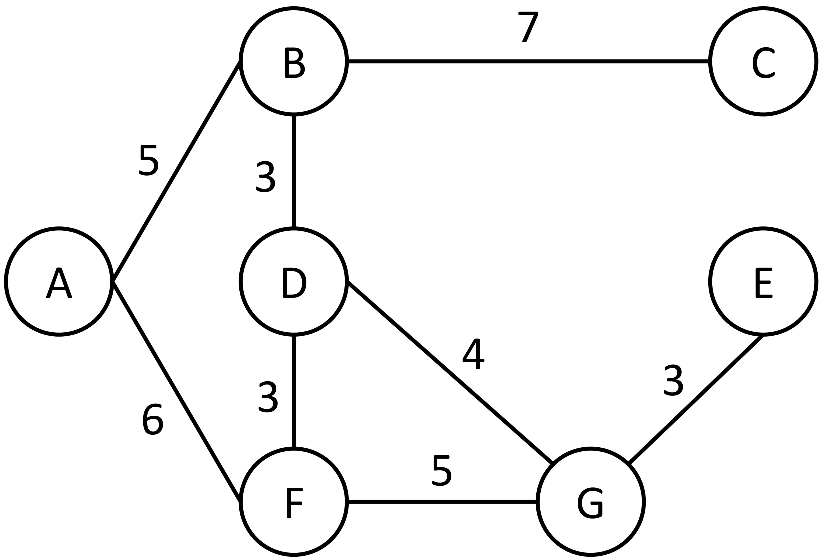
A

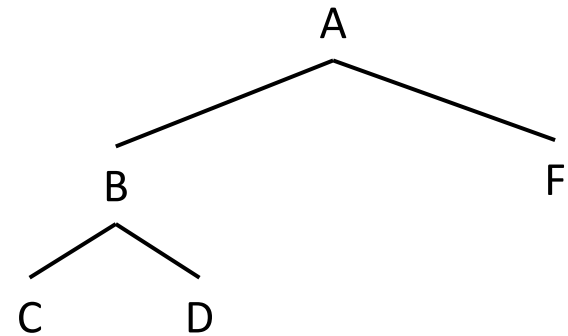
Breadth-First Search (BFS)



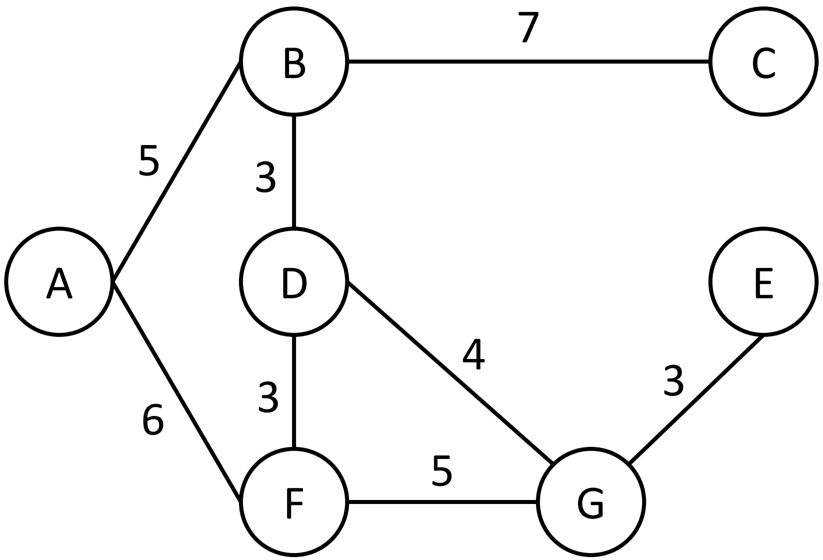


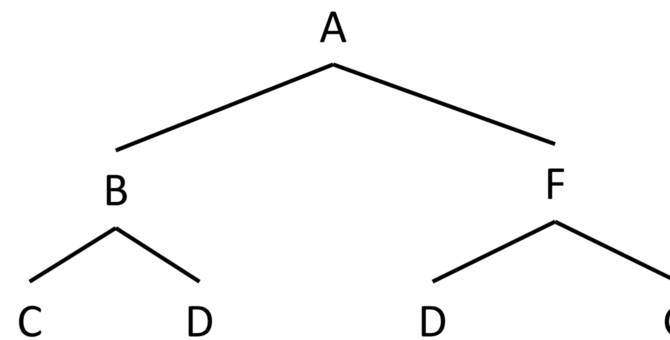
Breadth-First Search (BFS)



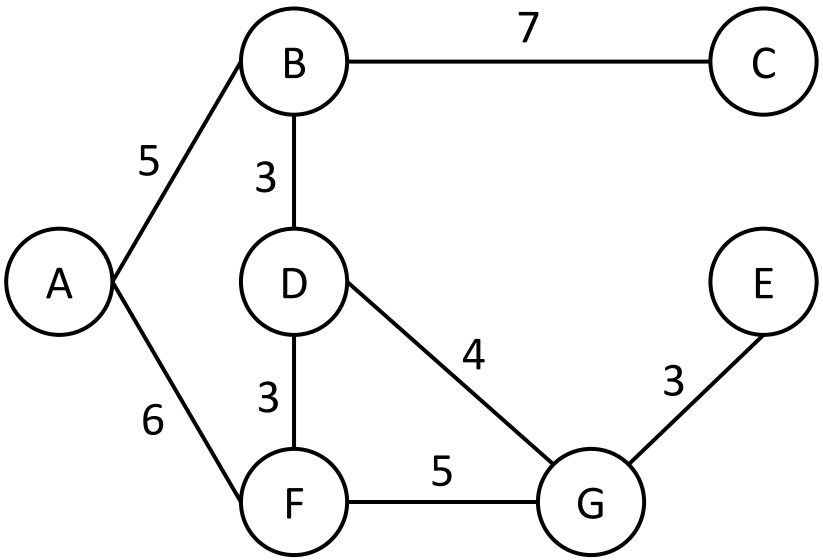


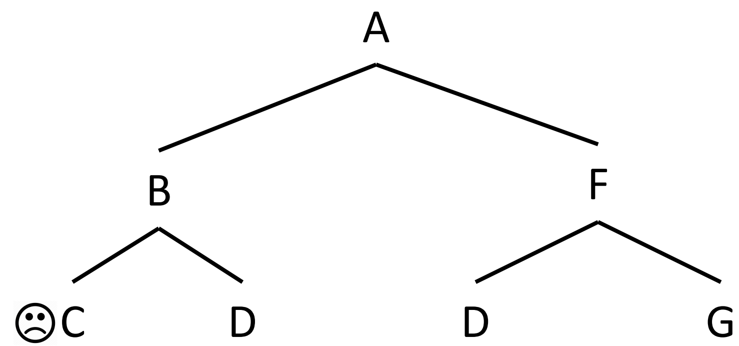
Breadth-First Search (BFS)



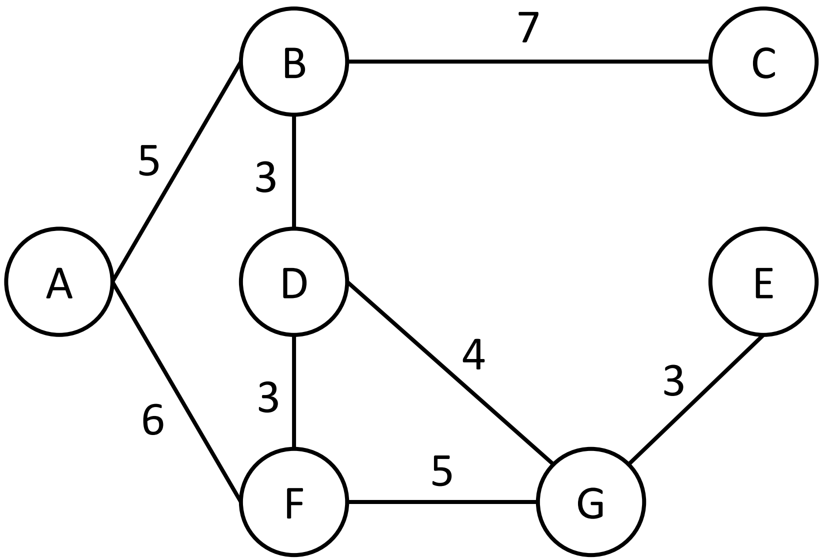


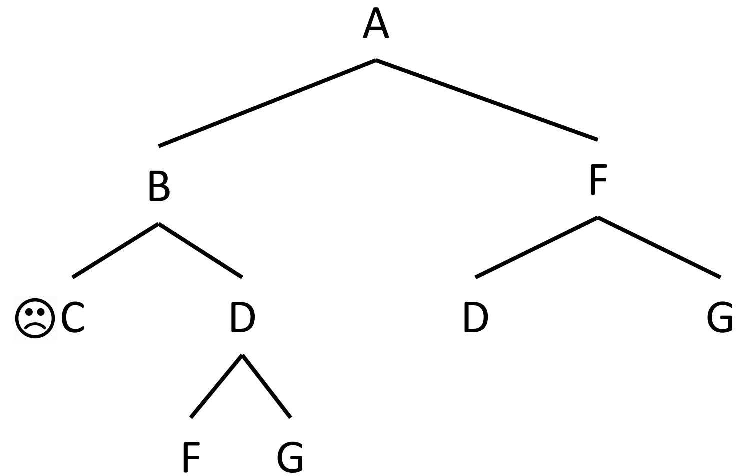
Breadth-First Search (BFS)



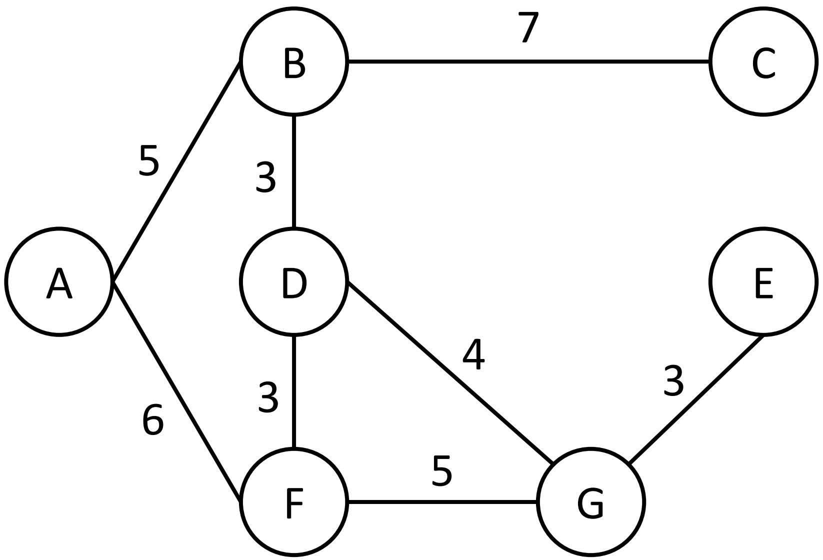


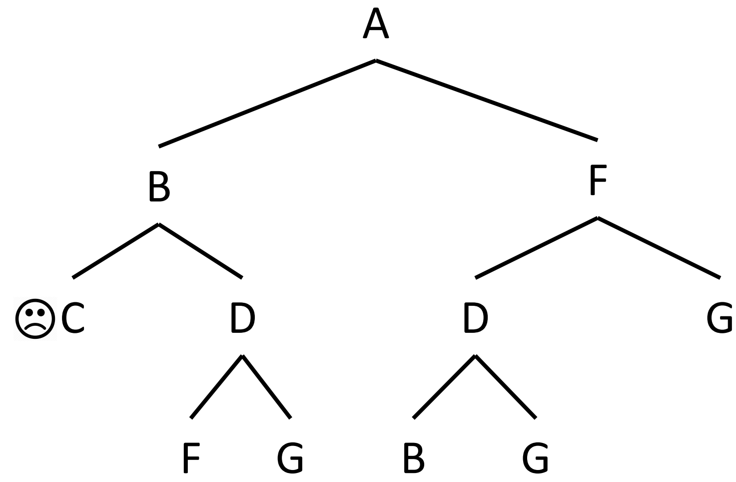
Breadth-First Search (BFS)



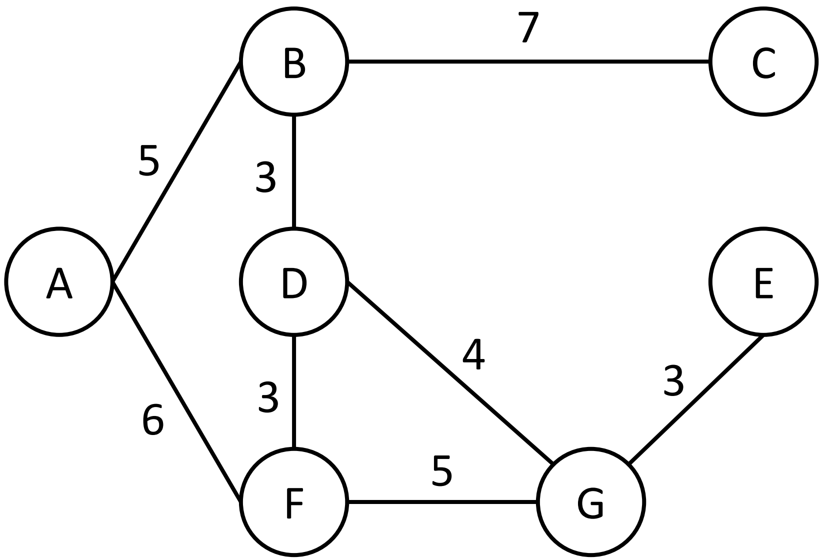


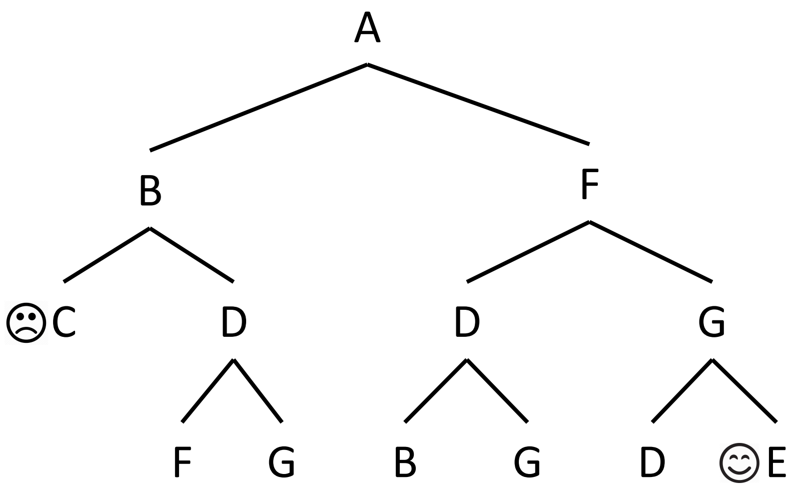
Breadth-First Search (BFS)





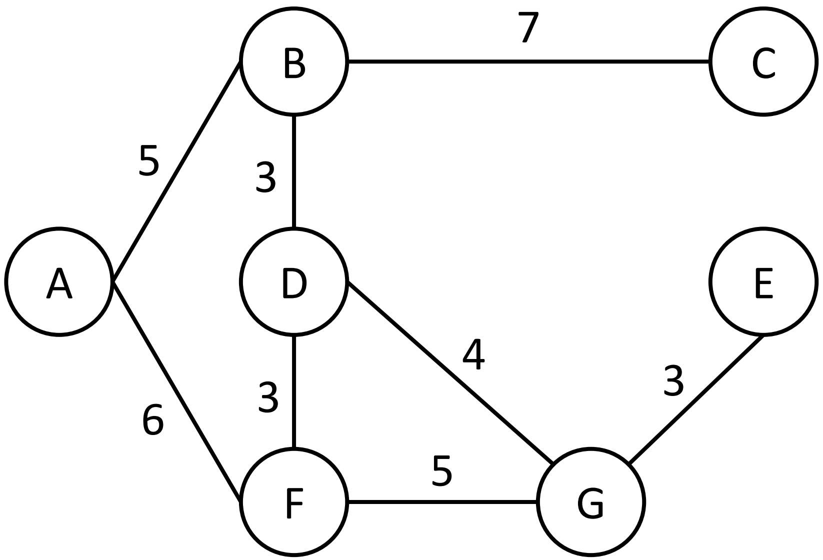
Breadth-First Search (BFS)

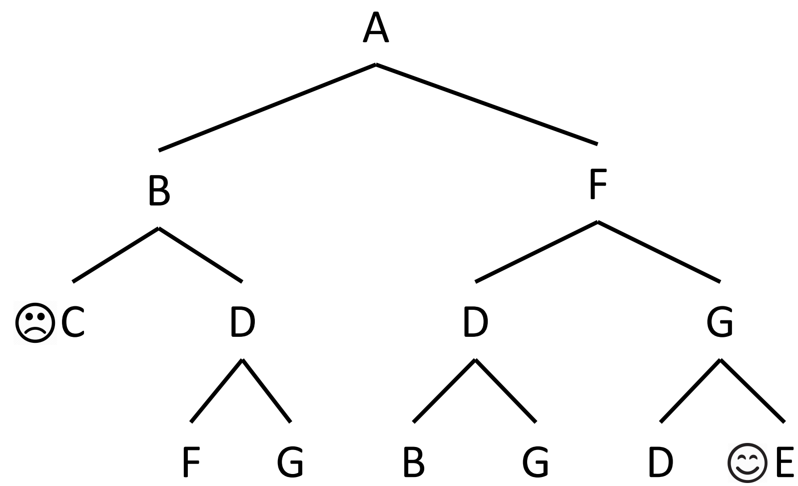




Solution: (A->F->G->E)

Breadth-First Search (BFS)





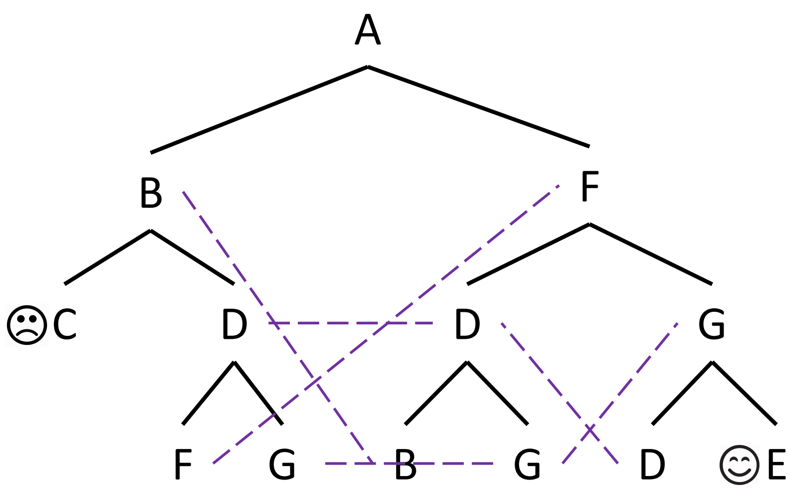
Solution: (A->F->G->E)

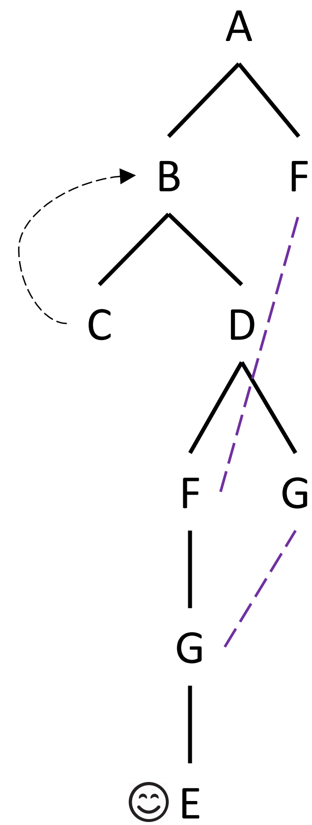
- A Breadth-First Search (BFS) chooses the shallowest node, thus exploring in a level by level fashion
- It has a more conservative behavior and does not need to reconsider decisions
- Call q the depth of the shallowest solution (in general $q \leq d$)
- Space complexity: $O(b^q)$
- Time complexity: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

Redundant paths

- Both DFS and BFS visited some nodes multiple times (avoiding loops prevents this to happen only within the same branch)

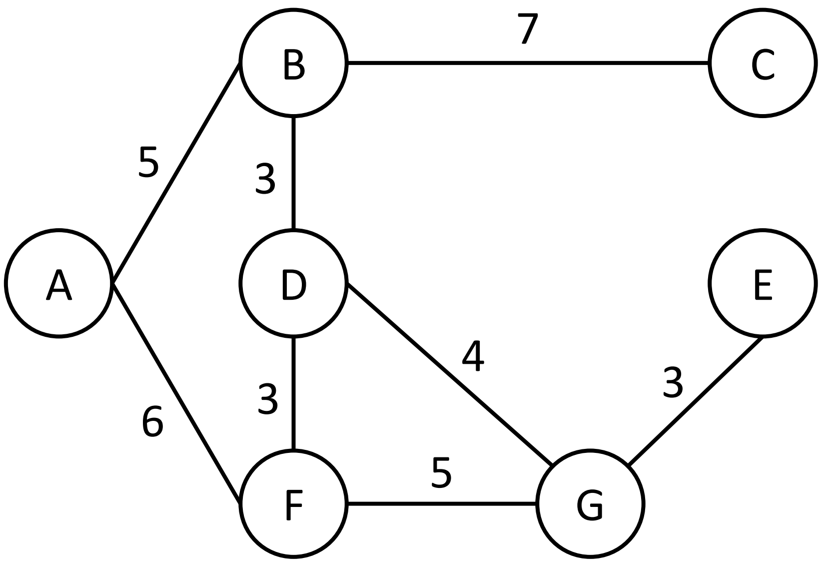
- In general, this does not seem very efficient. Why?



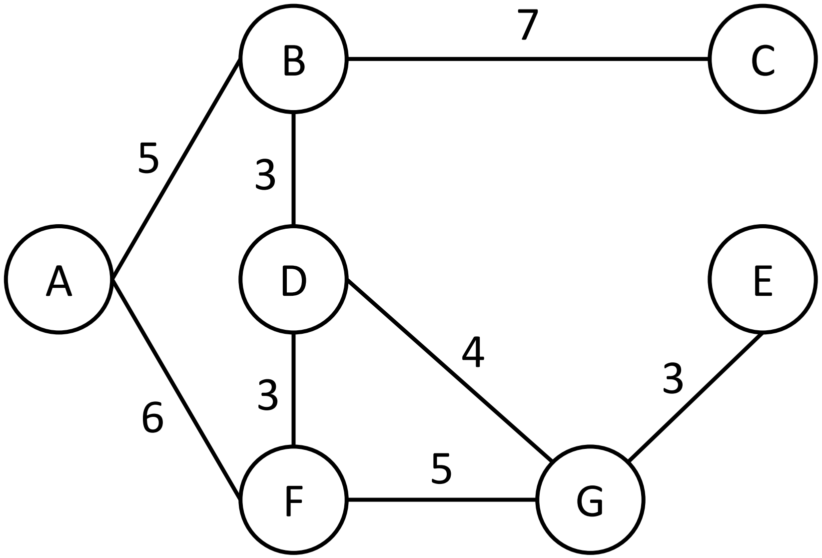


- Idea: discard a newly generated node if already present somewhere on the tree, we can do this with an **enqueued list**

DFS with Enqueued List

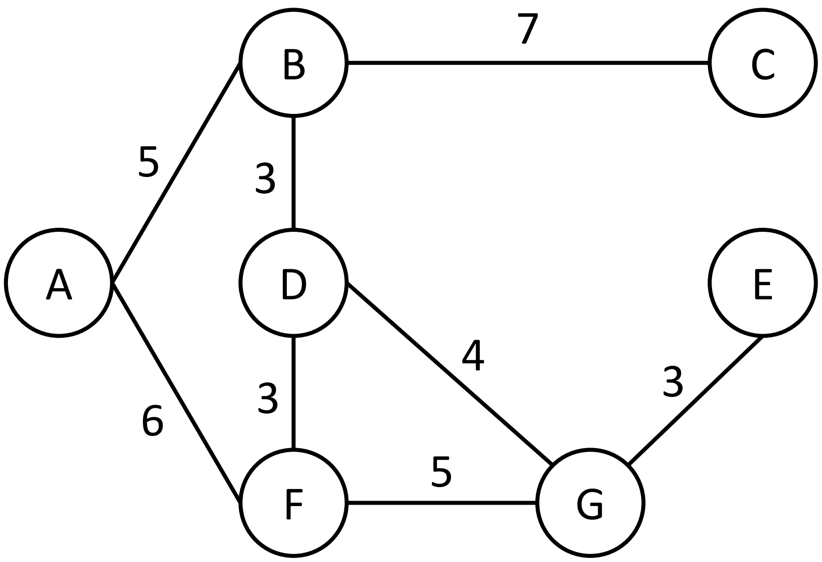


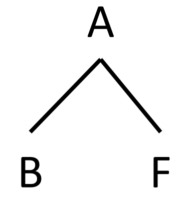
DFS with Enqueued List



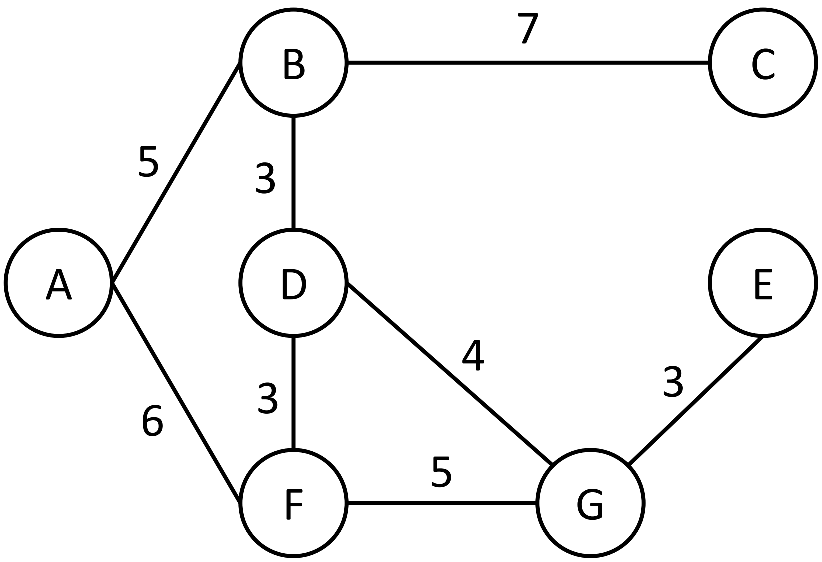
A

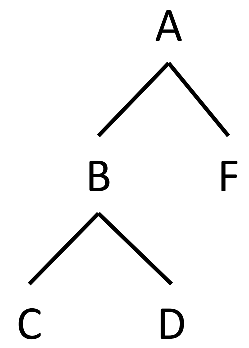
DFS with Enqueued List



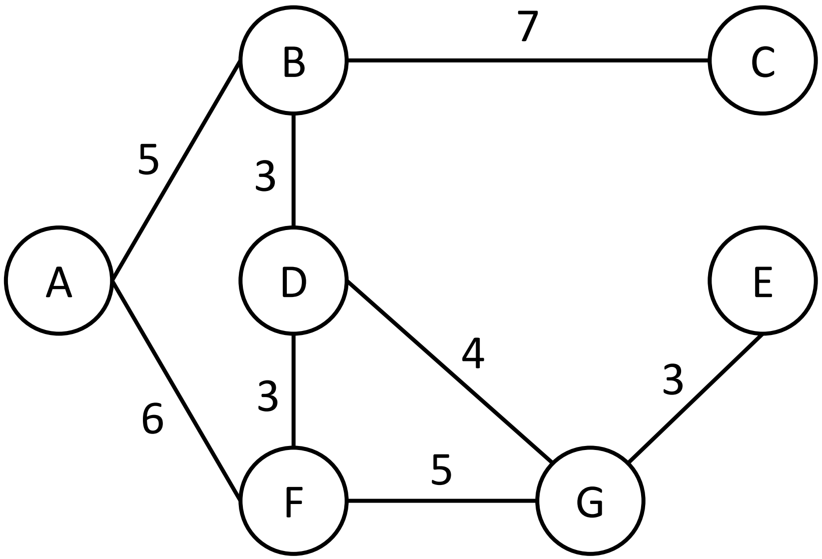


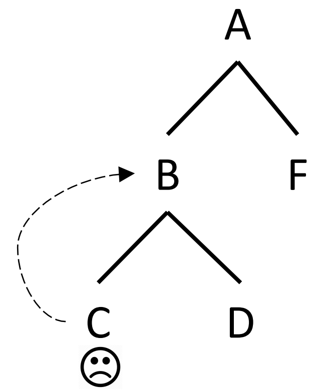
DFS with Enqueued List



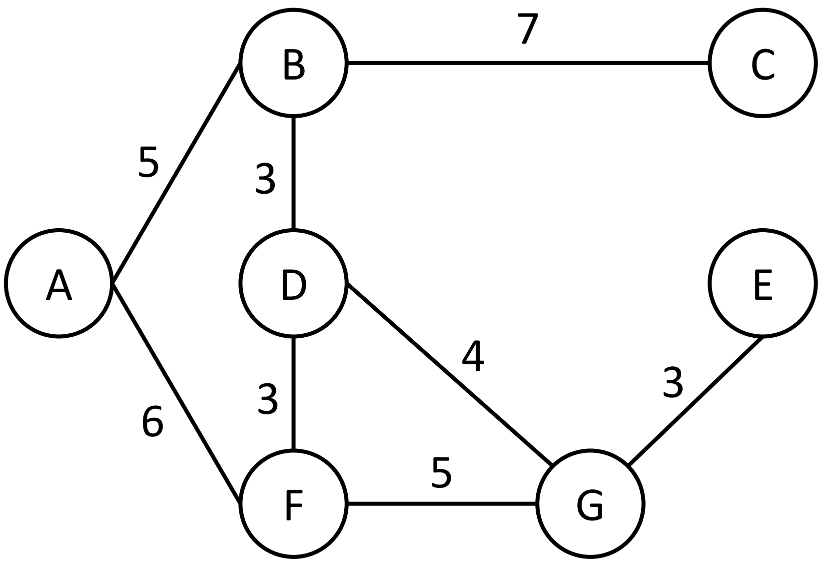


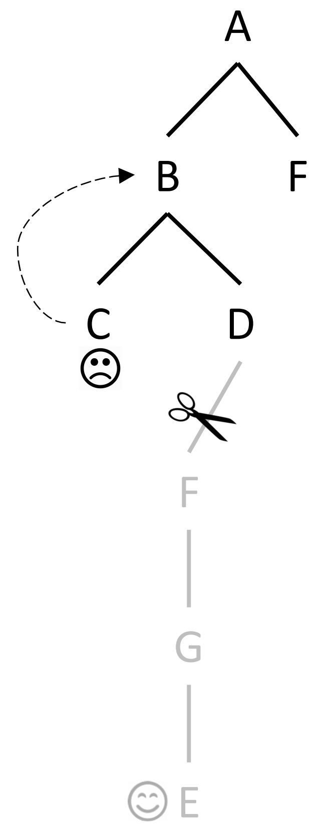
DFS with Enqueued List





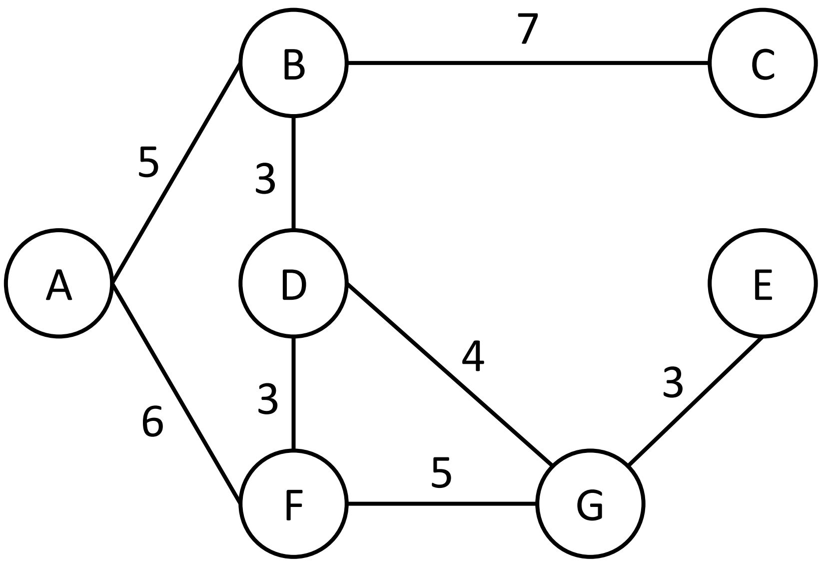
DFS with Enqueued List

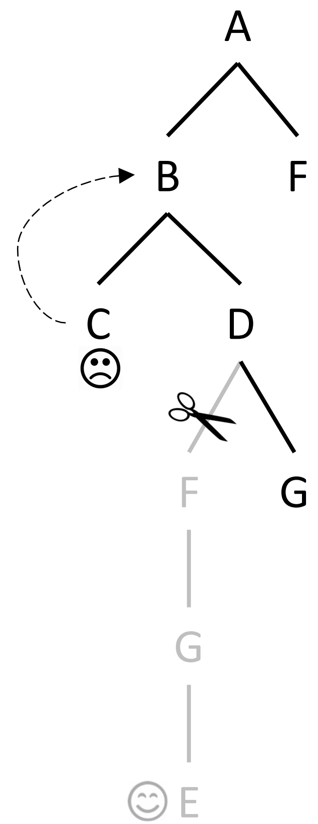




- Node F ha already been "enqueued" on the tree, by discarding it we *prune* a branch of the tree

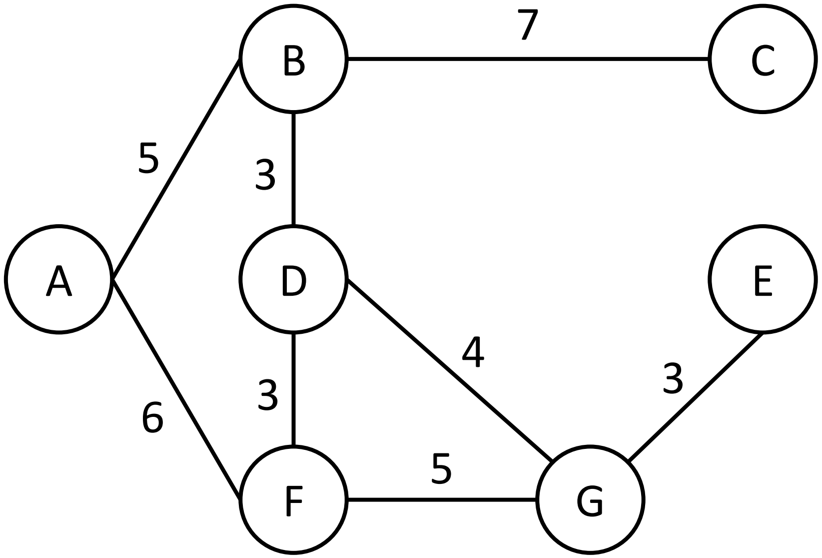
DFS with Enqueued List

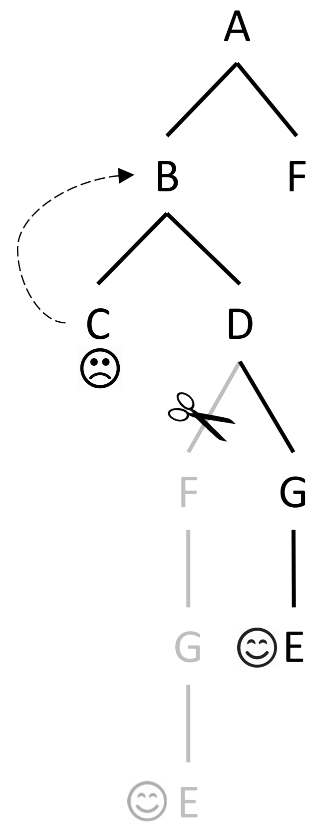




- Node F ha already been "enqueued" on the tree, by discarding it we *prune* a branch of the tree

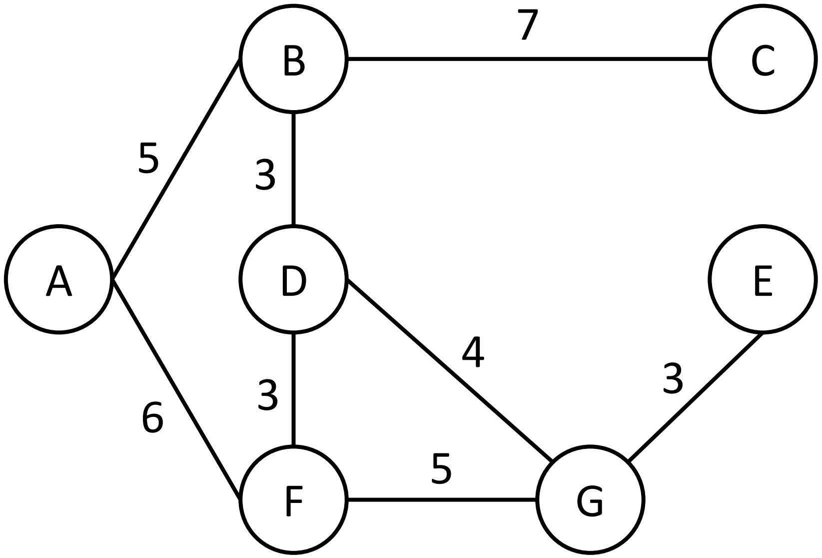
DFS with Enqueued List



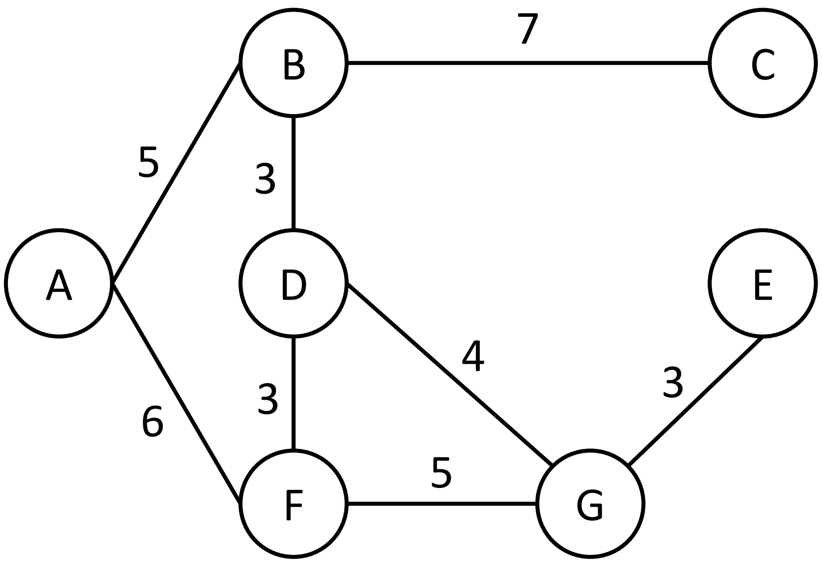


- Node F ha already been “enqueued” on the tree, by discarding it we *prune* a branch of the tree

BFS with Enqueued List

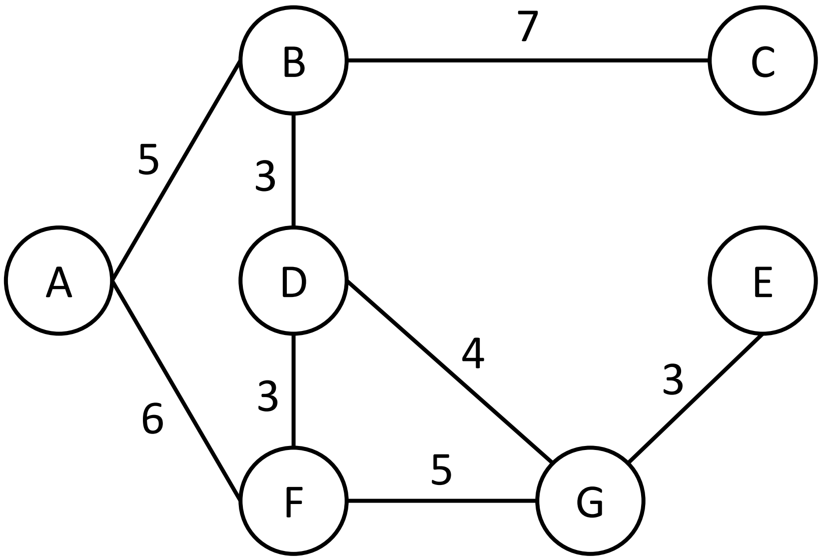


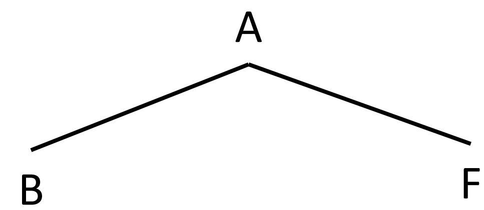
BFS with Enqueued List



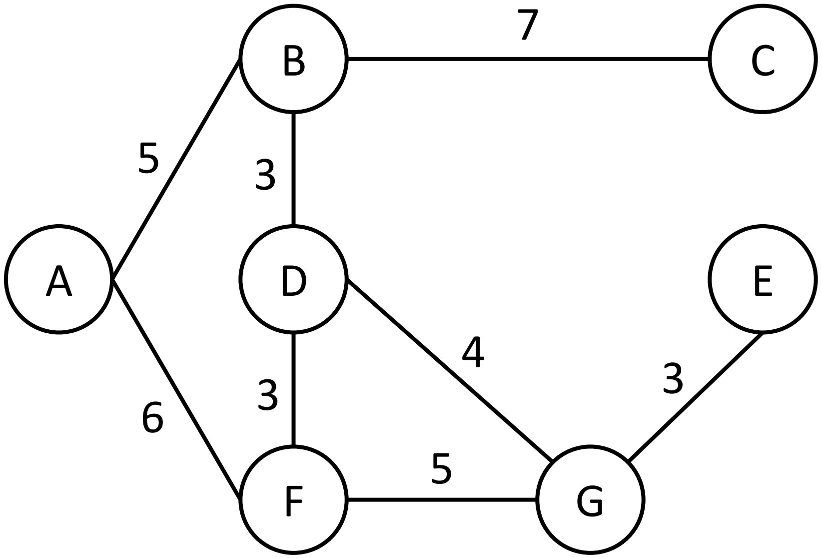
A

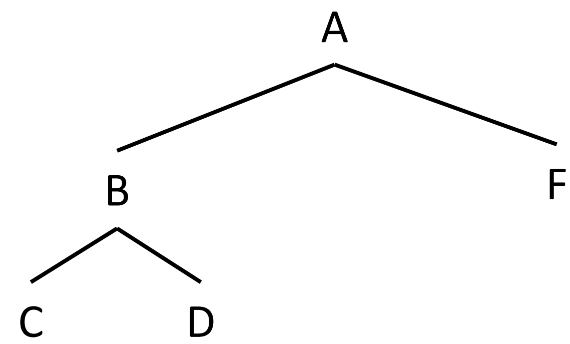
BFS with Enqueued List



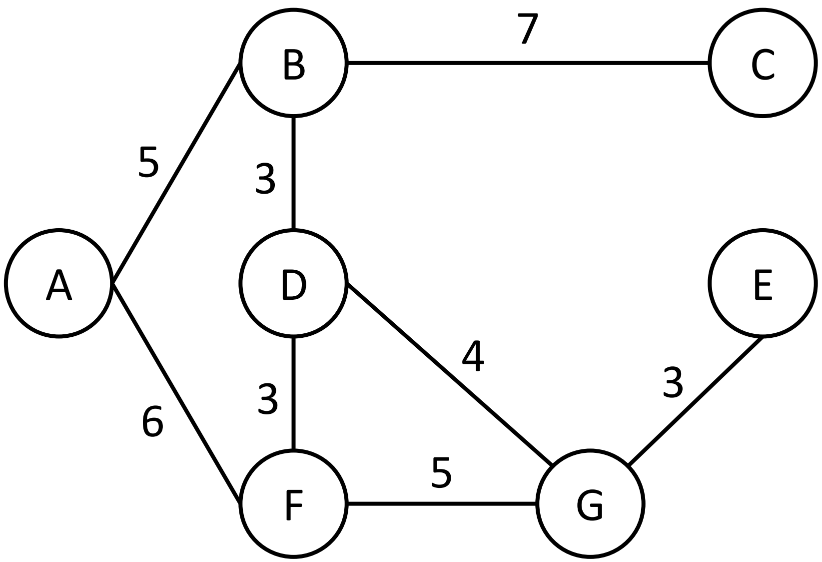


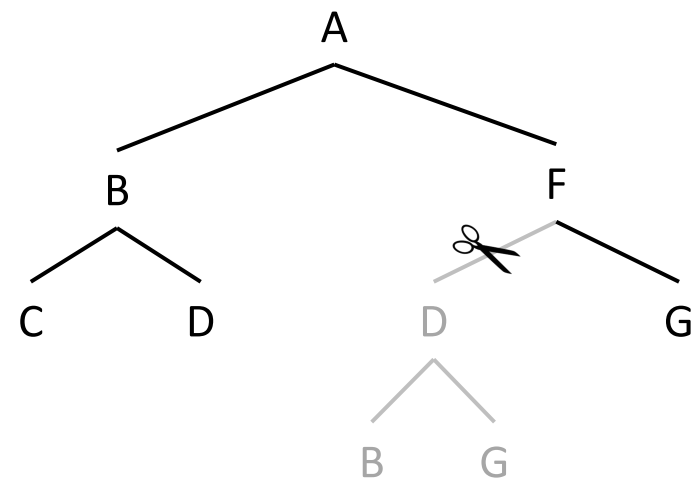
BFS with Enqueued List



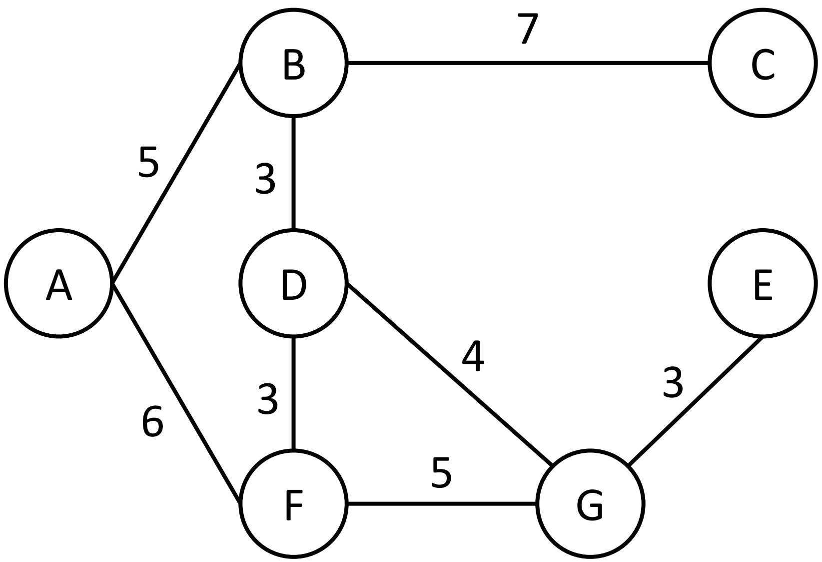


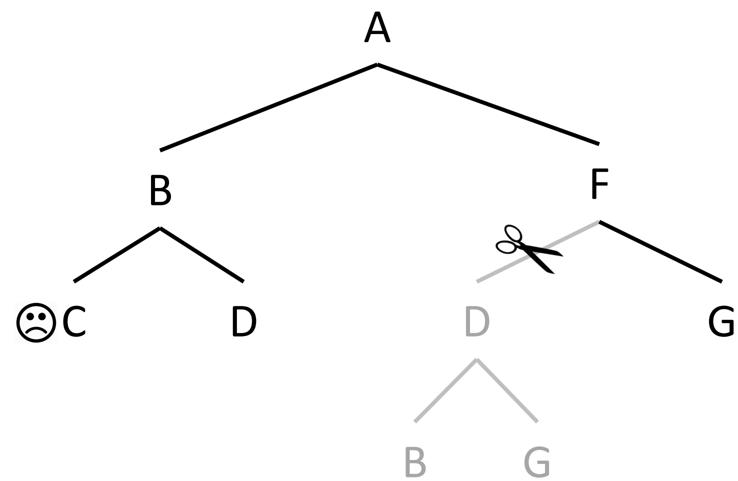
BFS with Enqueued List



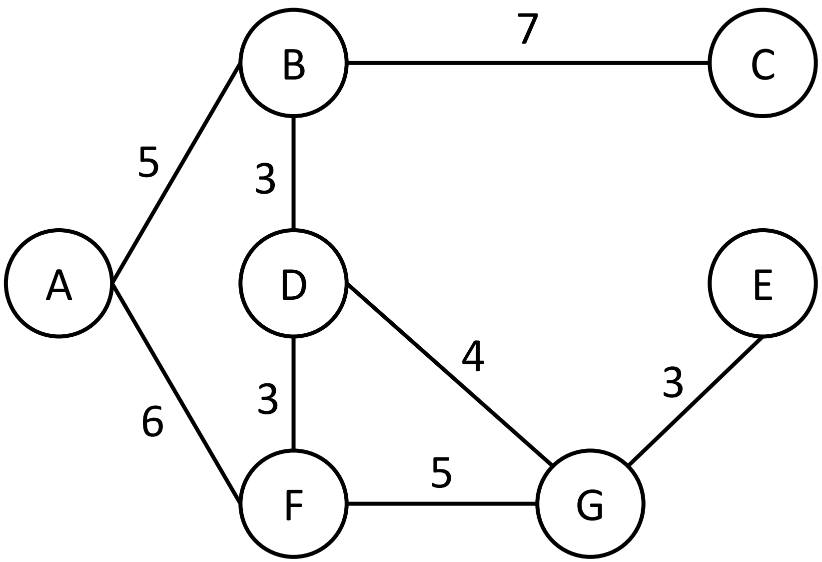


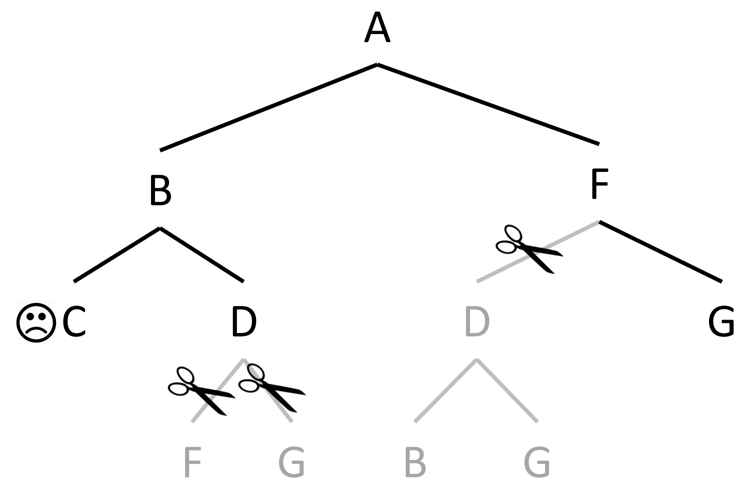
BFS with Enqueued List



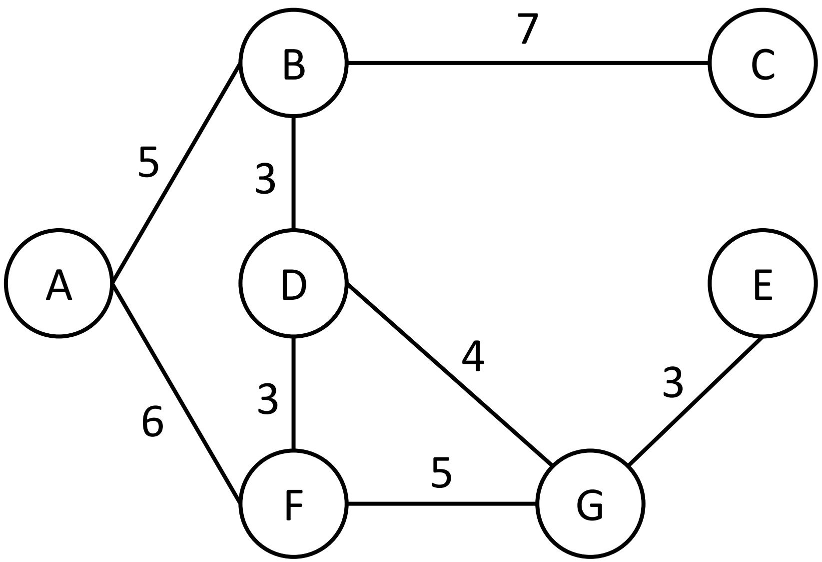


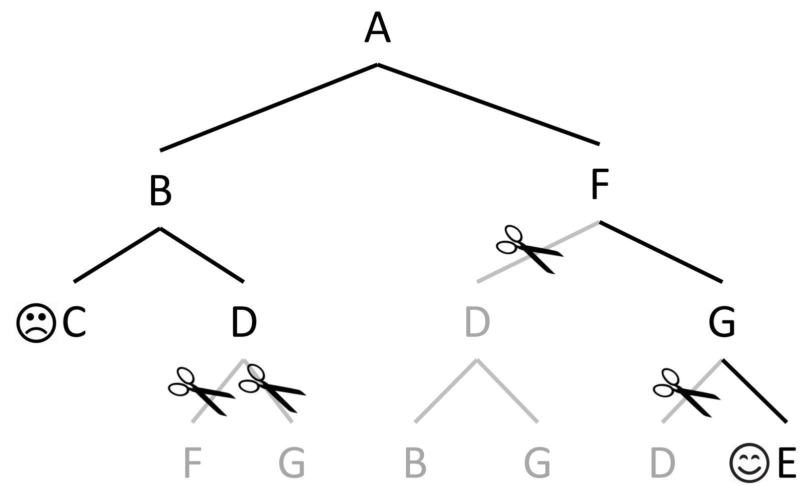
BFS with Enqueued List





BFS with Enqueued List

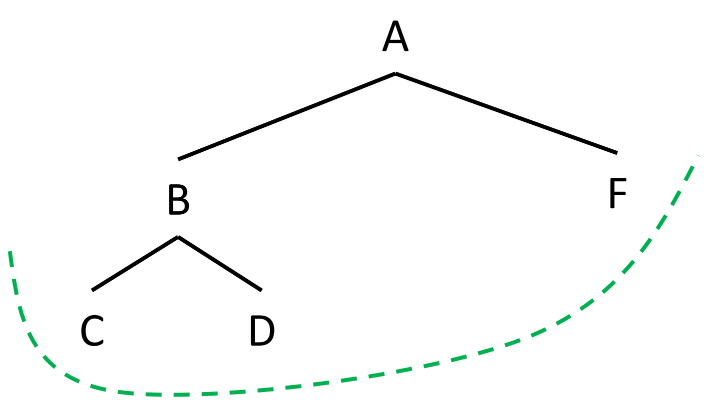


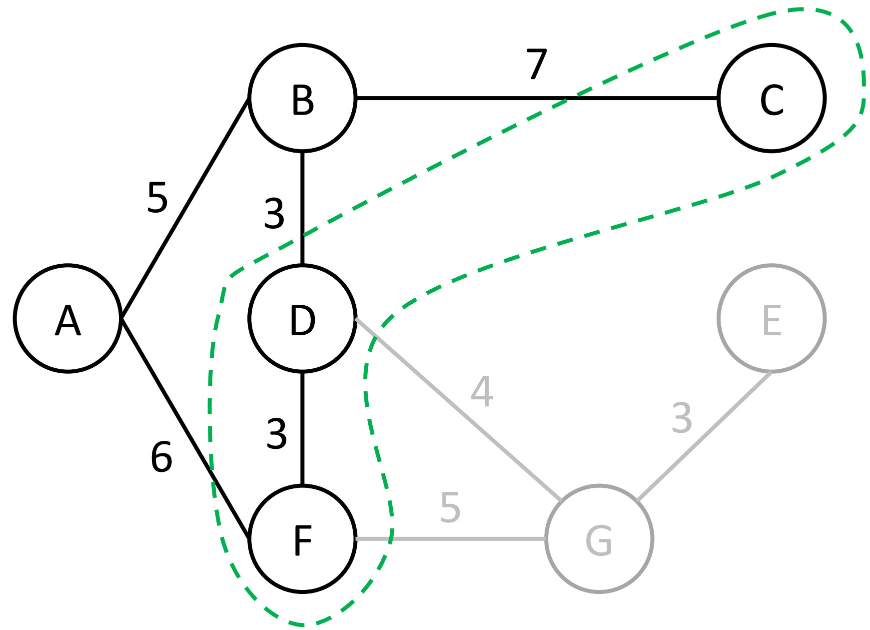


Implementation

- 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)

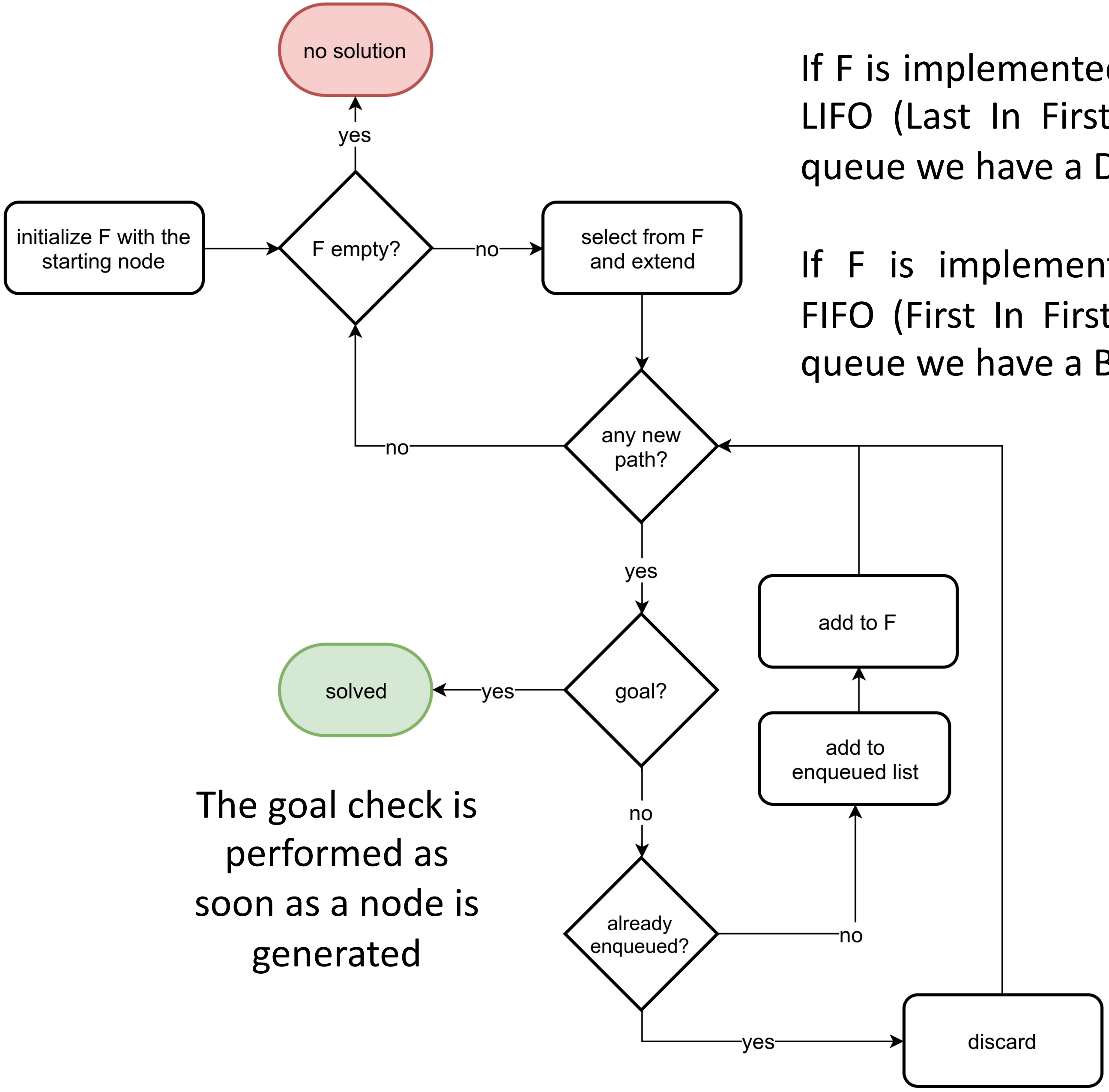
- The frontier F contains the terminal nodes of all the paths currently under exploration on the tree





- 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

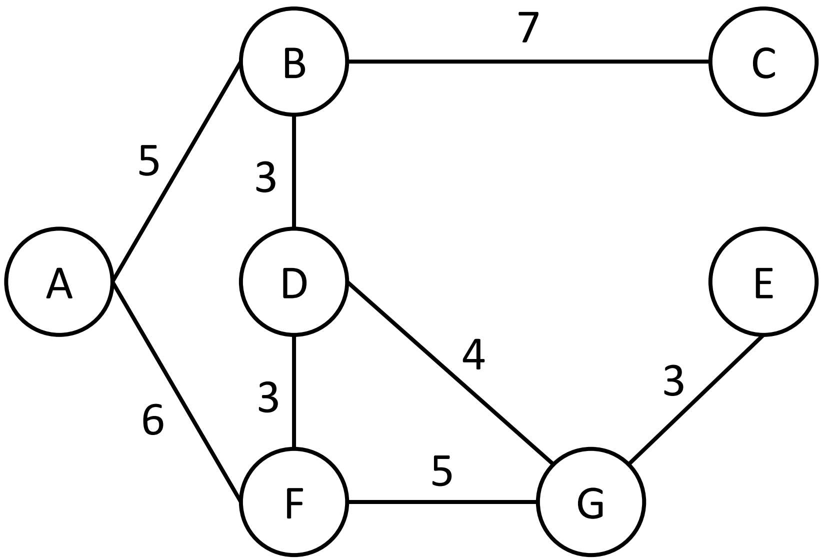


If F is implemented as a LIFO (Last In First Out) queue we have a DFS

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

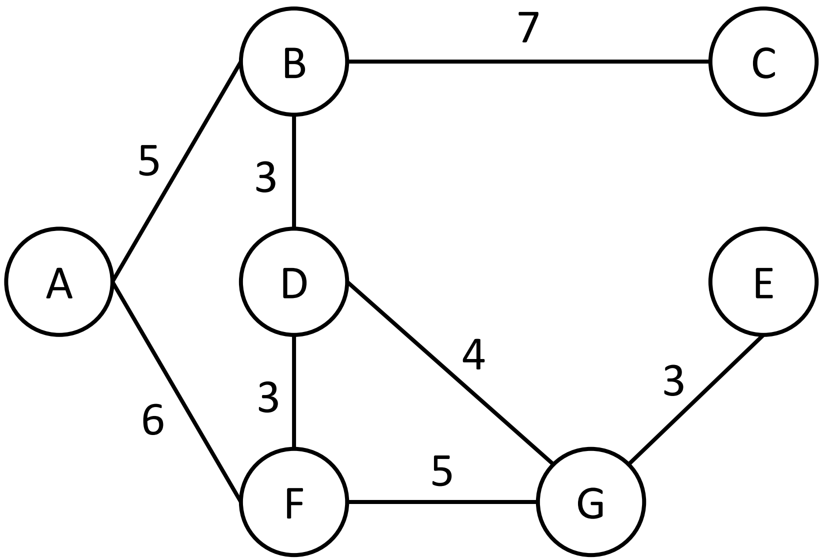
Depth-limited Search



- Variant of DFS, trying to solve issues in “deep” or infinite state space
- Idea: limit the max number of depth search to a level l
- Nodes at level l are treated as if they have no successor
- Call q the depth of the shallowest solution, how do we set l?
- What if we choose $l > d$? Non-optimal

- Time complexity: $O(b^l)$
- Space complexity: $O(bl)$

Iterative-deepening DFS



- Variant of DFS and similar to depth-limited search
- Idea: limit the max number of depth search to a level l, increasing l
- Nodes at level l are treated as if they have no successor
- We start with $l = 0$, if no solution is found increase $l = l + 1$ until a solution is found
- Complete in finite spaces

- Space complexity: $O(b^q)$
- Time complexity: $O(bq)$

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?

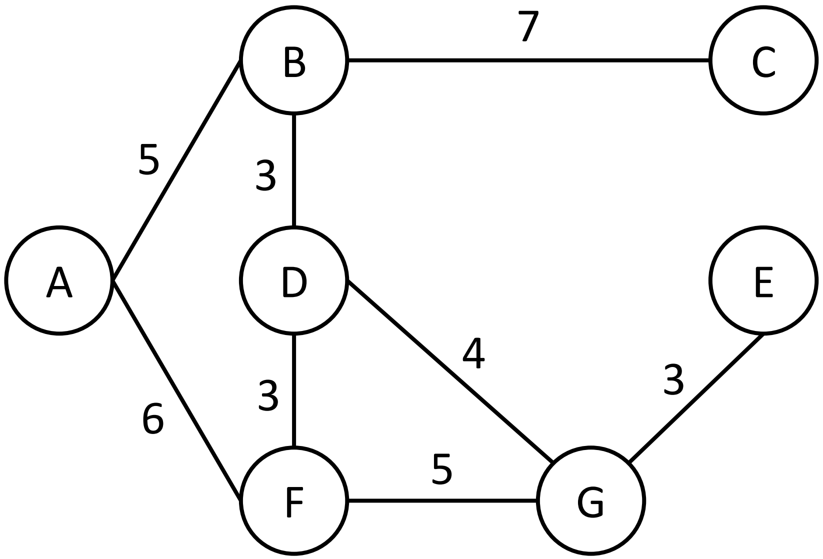
- We generalize the idea of BFS to that of Uniform Cost Search (UCS)

- 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)

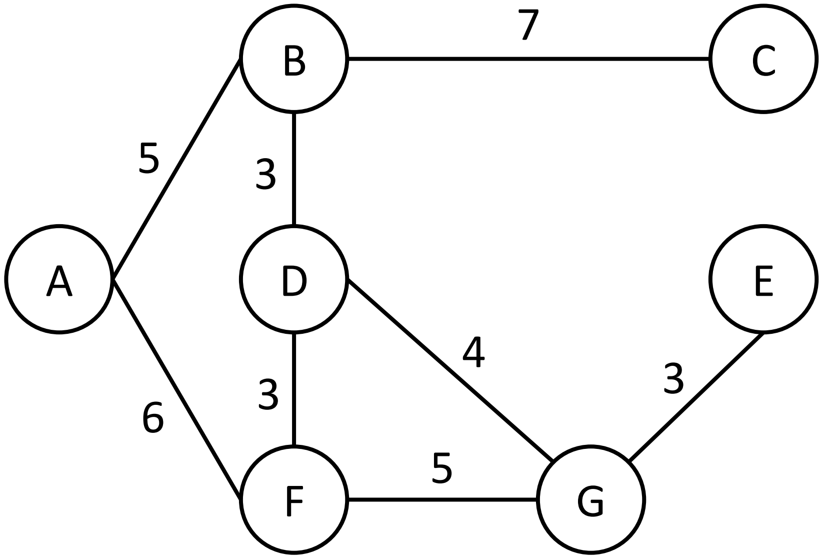
- Cost accumulated on a path from the start node to v: $g(v)$ (we should include a dependency on the path, but it will always be clear from the context)

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

Uniform Cost Search (UCS)

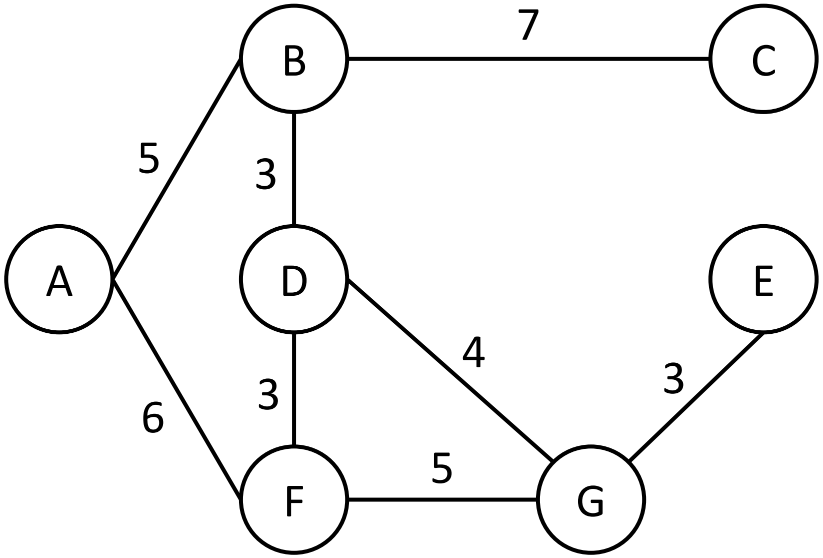


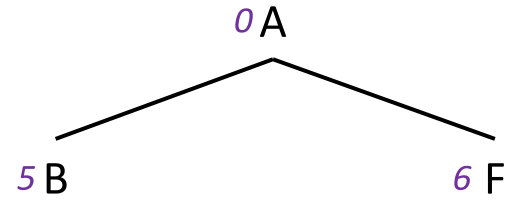
Uniform Cost Search (UCS)



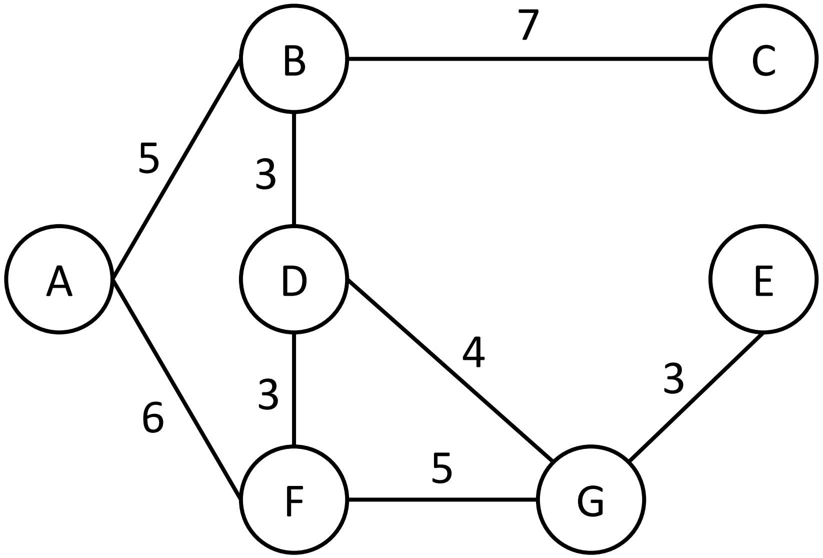
0 A

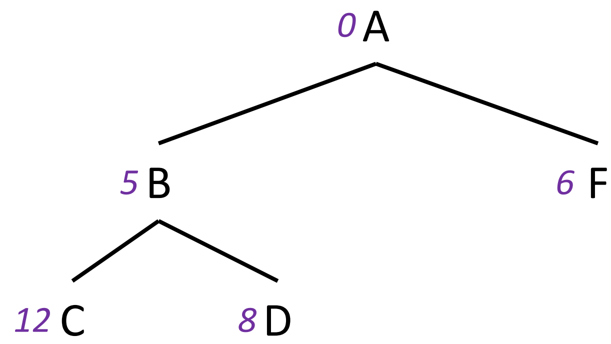
Uniform Cost Search (UCS)



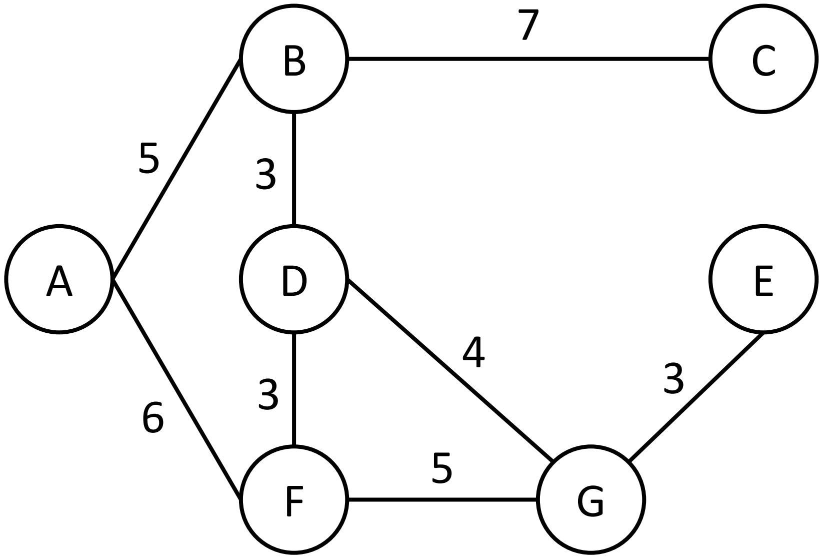


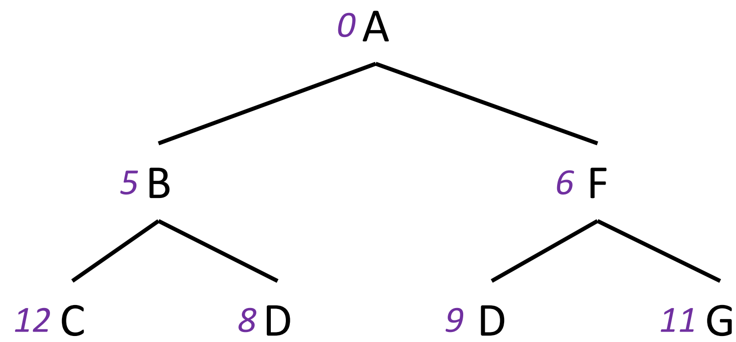
Uniform Cost Search (UCS)



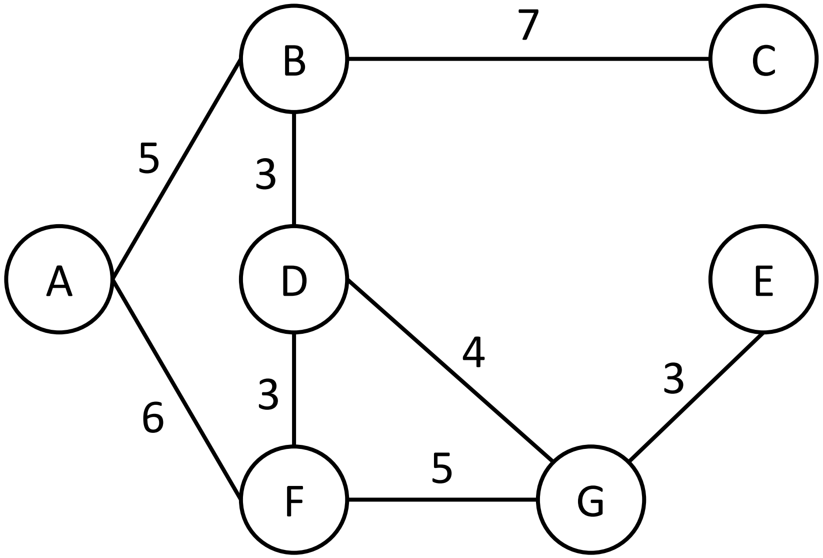


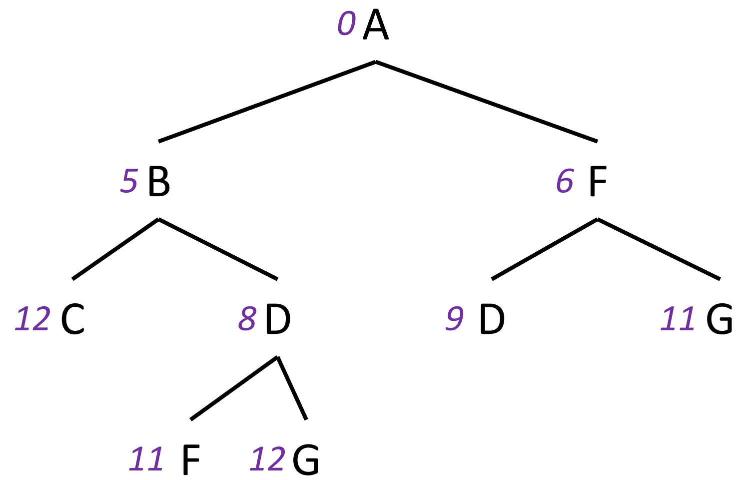
Uniform Cost Search (UCS)



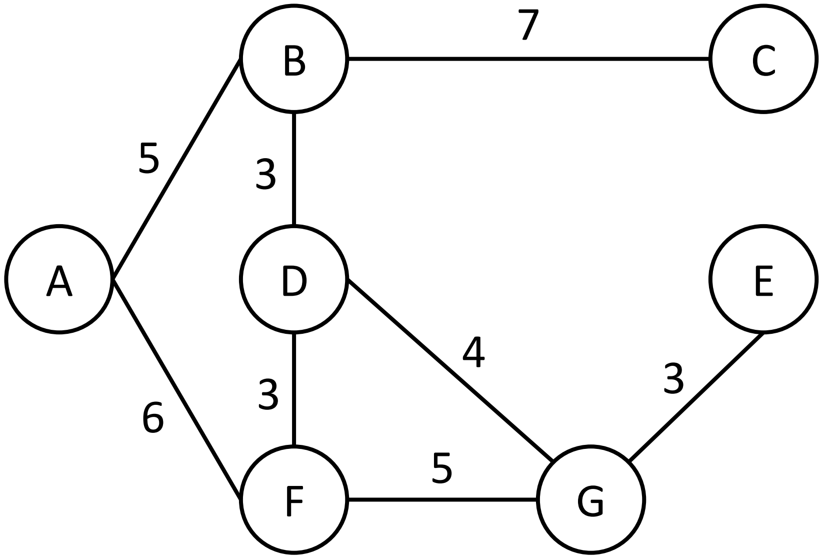


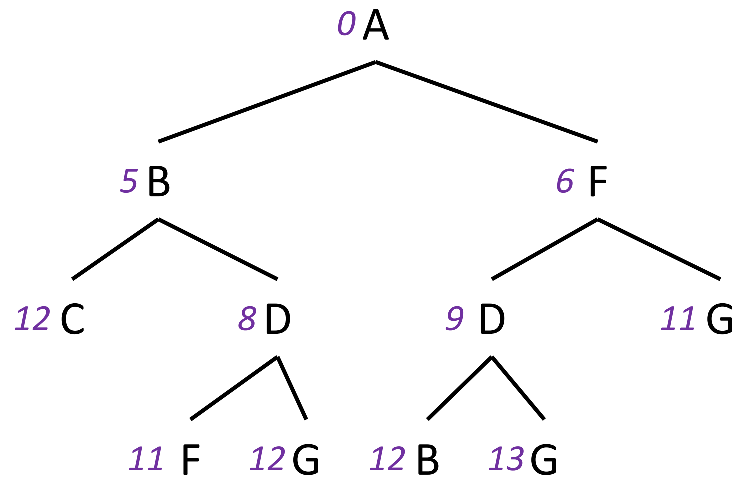
Uniform Cost Search (UCS)



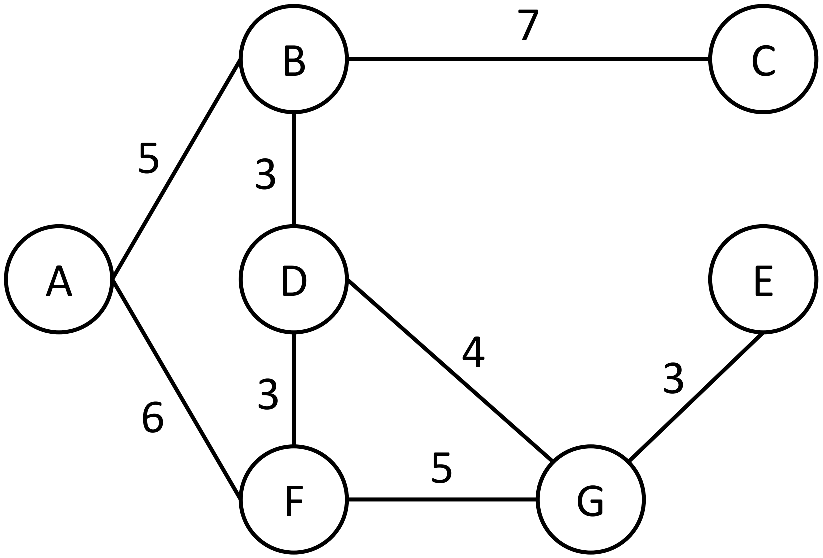


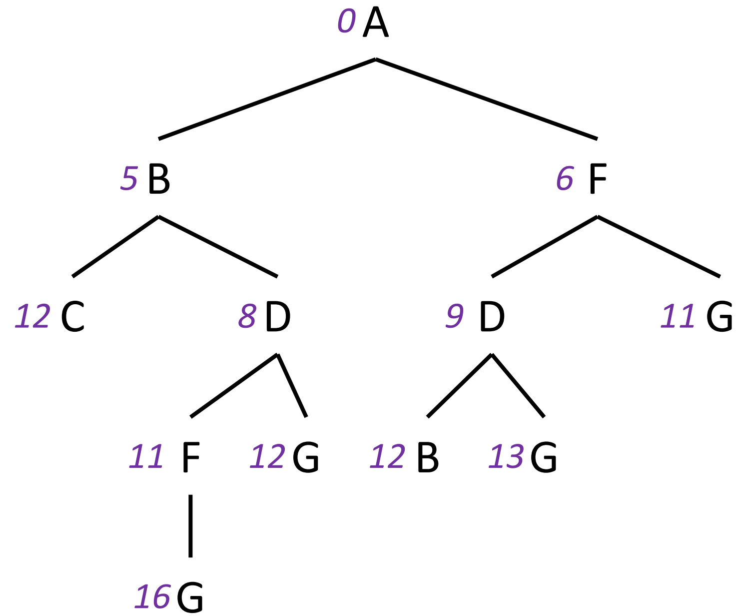
Uniform Cost Search (UCS)



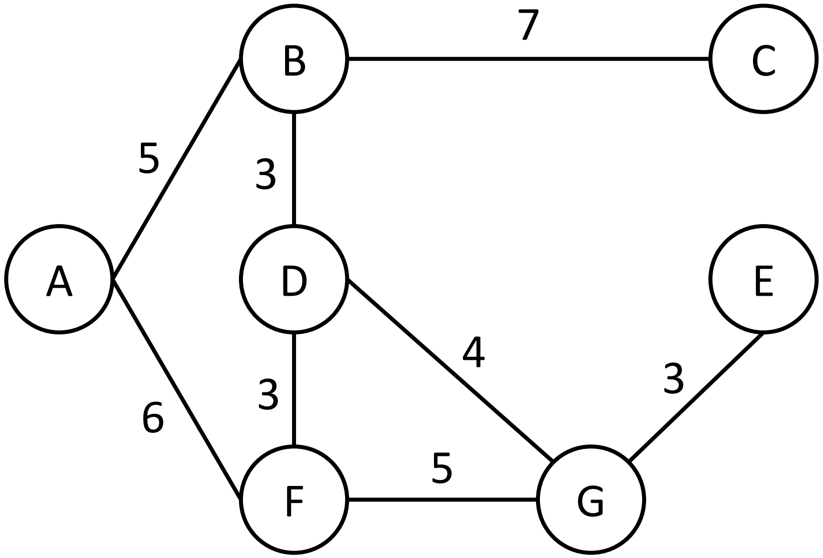


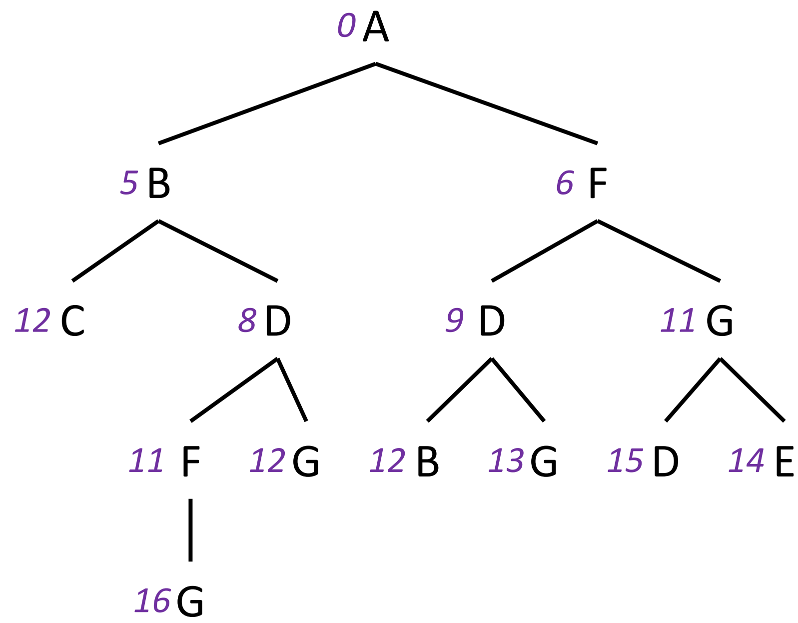
Uniform Cost Search (UCS)



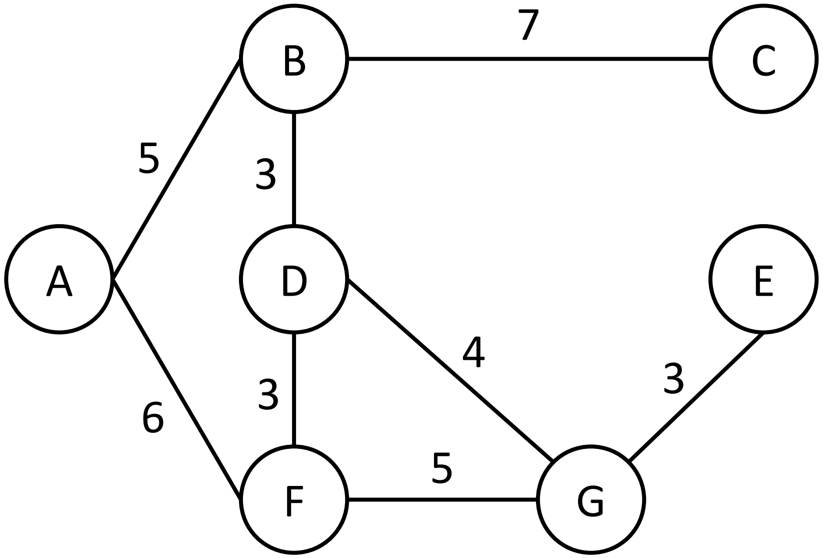


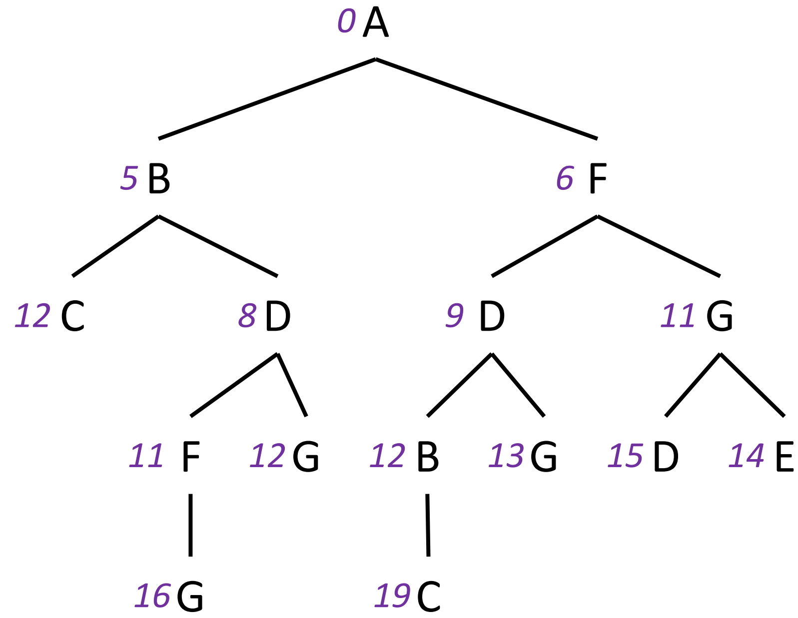
Uniform Cost Search (UCS)



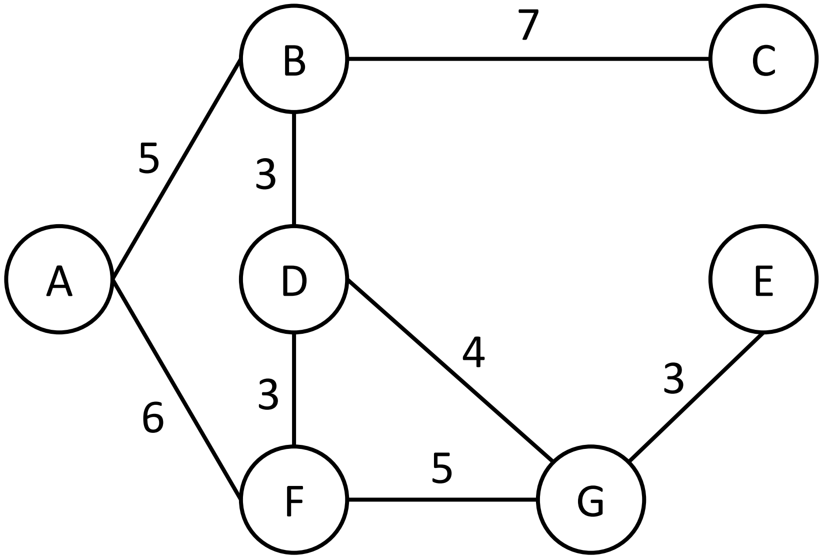


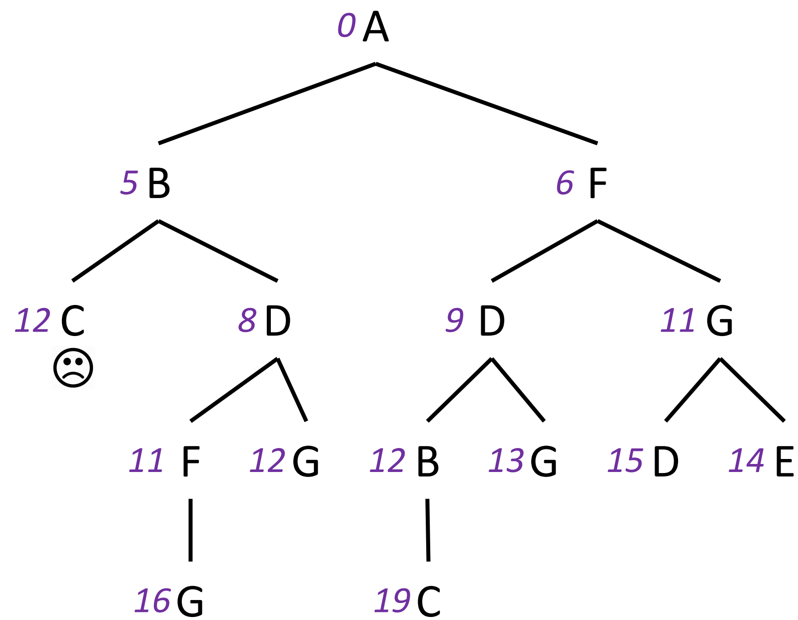
Uniform Cost Search (UCS)



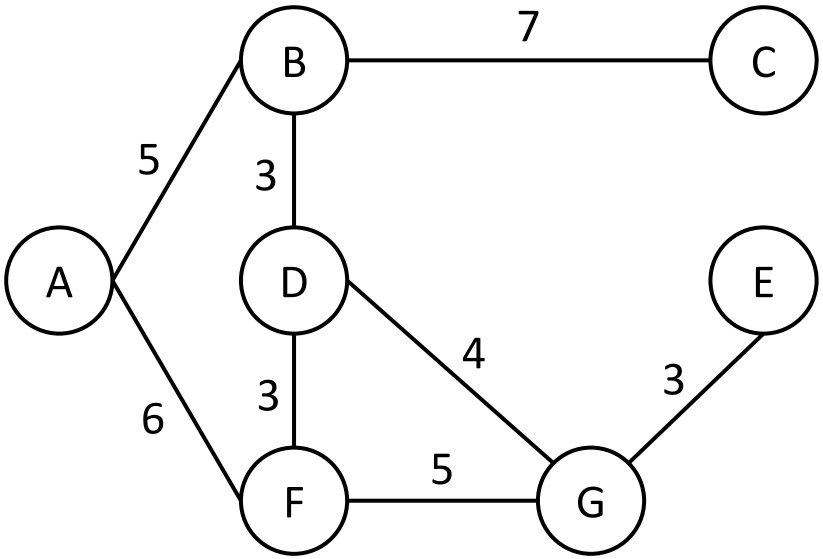


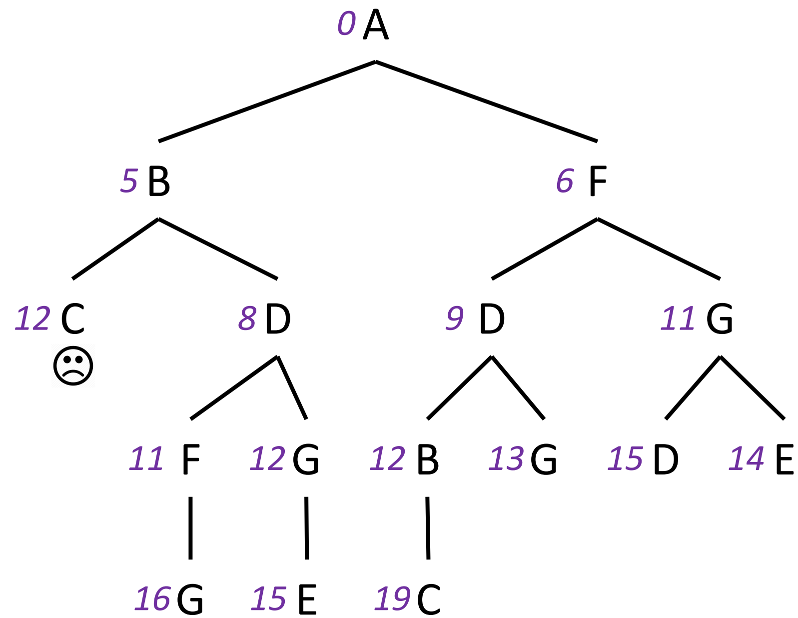
Uniform Cost Search (UCS)



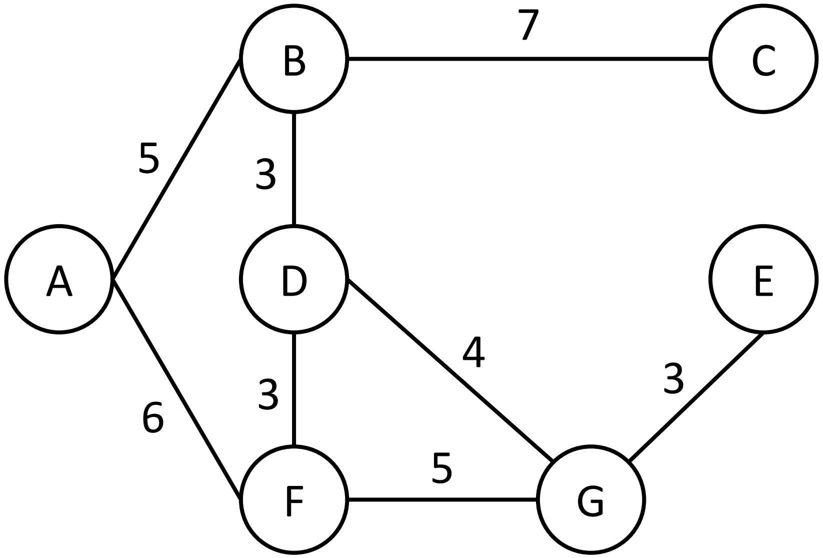


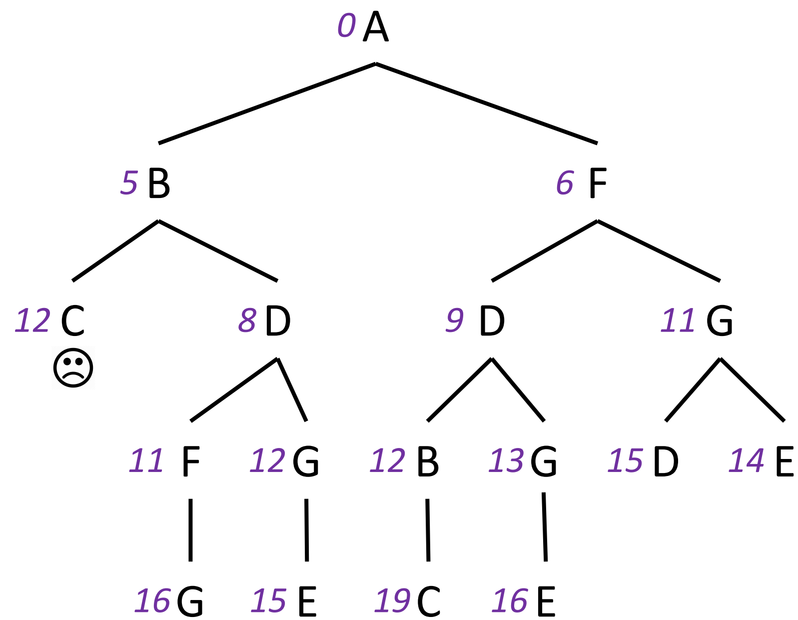
Uniform Cost Search (UCS)



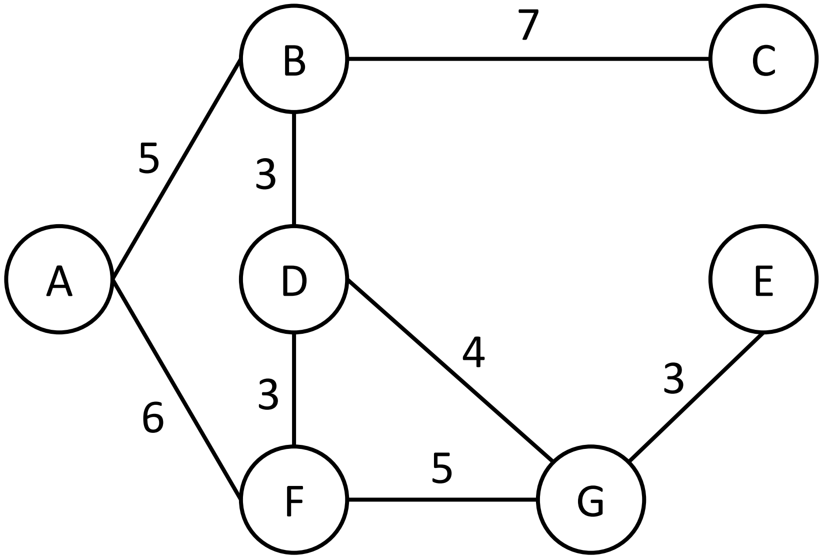


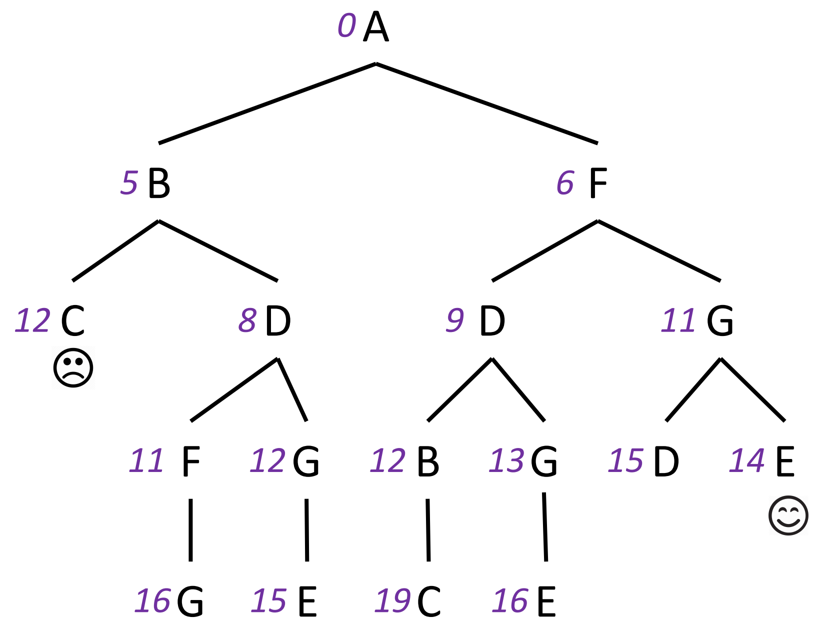
Uniform Cost Search (UCS)



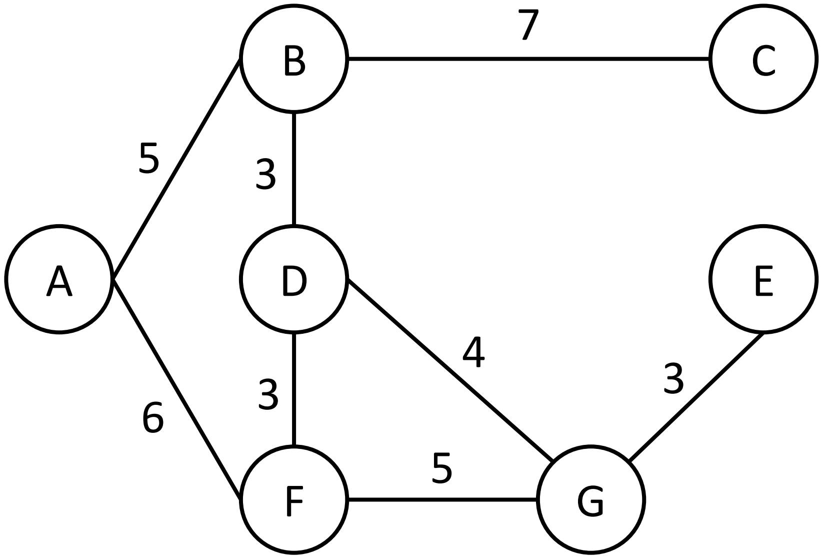


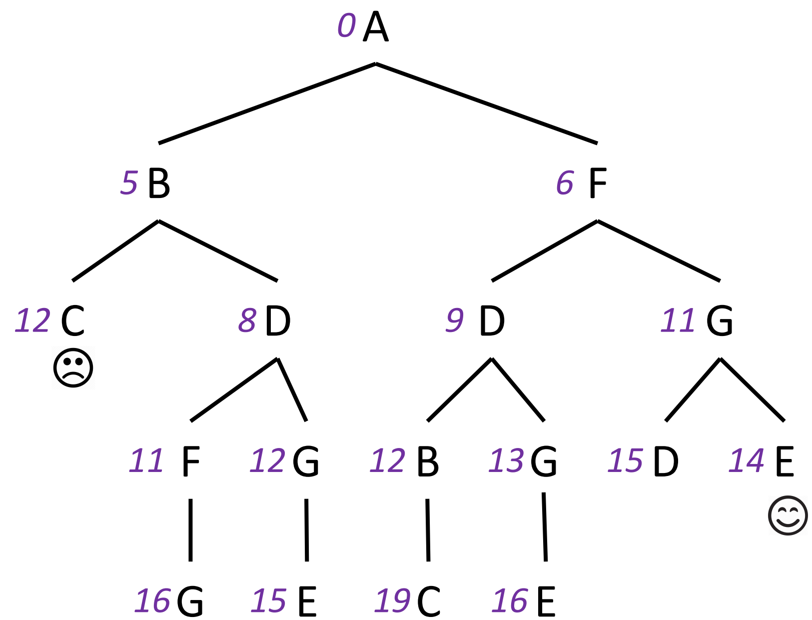
Uniform Cost Search (UCS)





Uniform Cost Search (UCS)



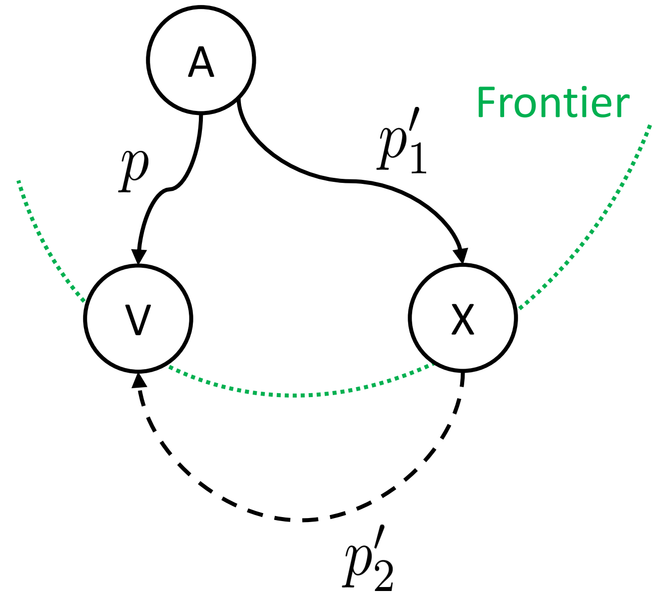


- Have we found the optimal path to the goal? In this problem instance, we can answer *yes* by inspecting the graph
- How about larger instances? Can we prove optimality?
- Actually, we can prove a stronger claim: every time UCS selects **for the first time** a node for expansion, the associated path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

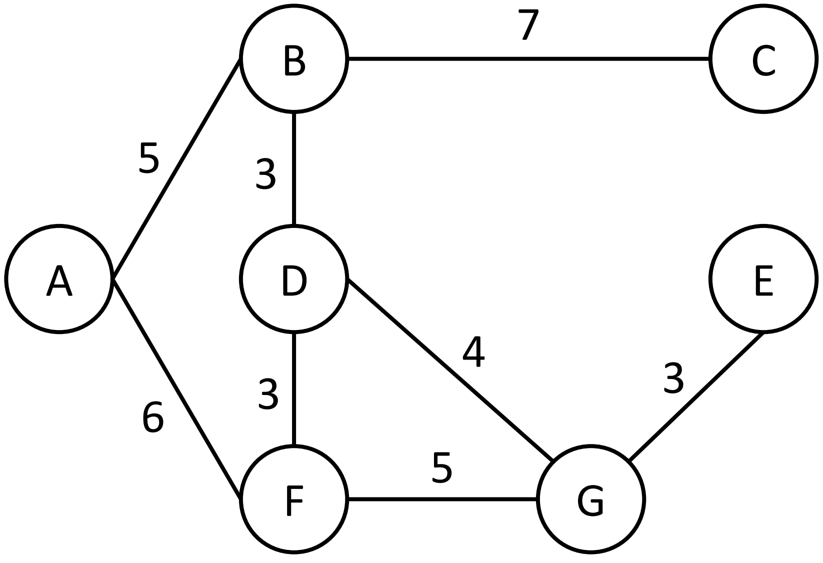
If when we select for the first time we discover the optimal path, there is no reason to select the same node a second time: **extended list**

Every time we select a node for extension:

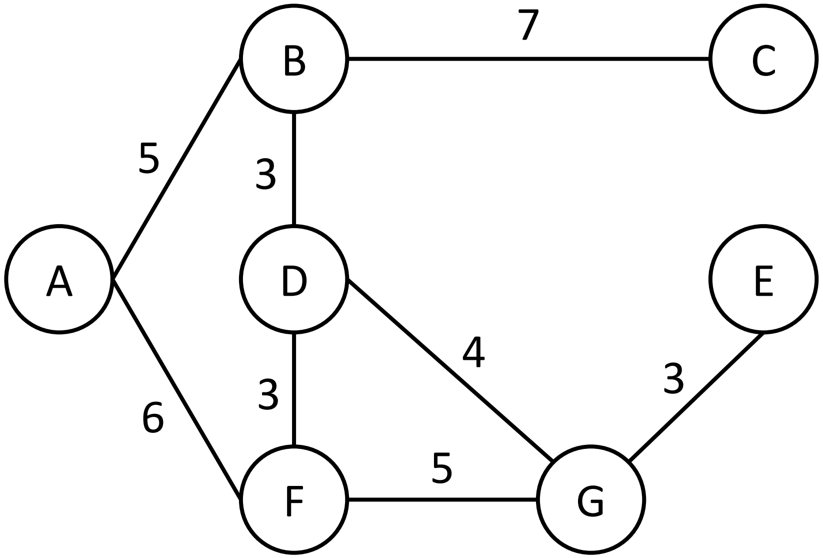
- If the node is already in the extended list we discard it
- Otherwise we extend it and we put it the extended list

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

UCS with extended list

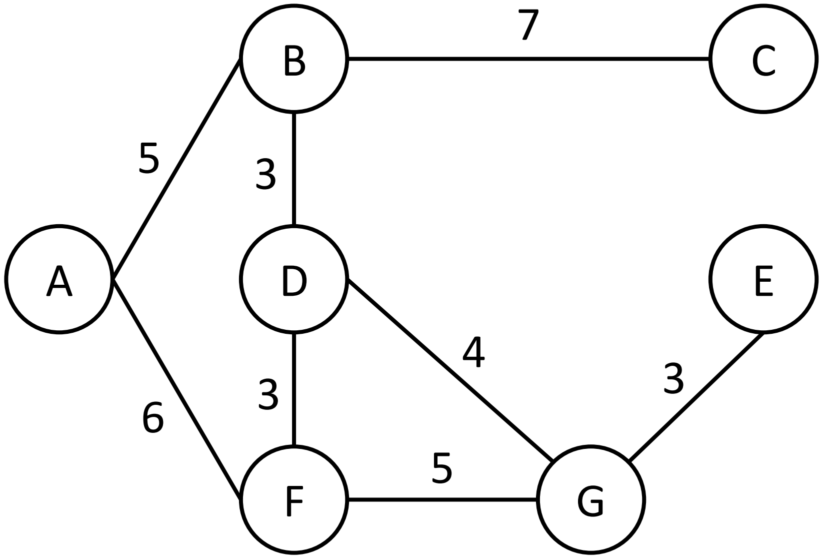


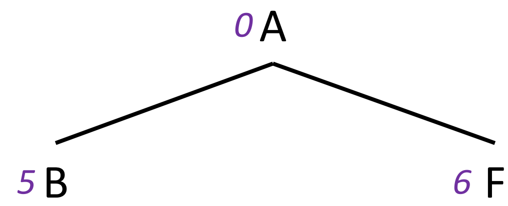
UCS with extended list



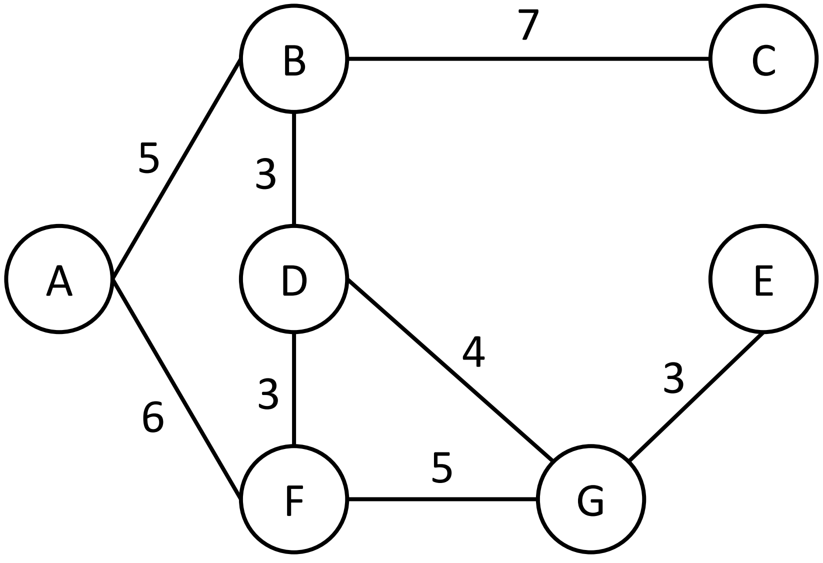
0 A

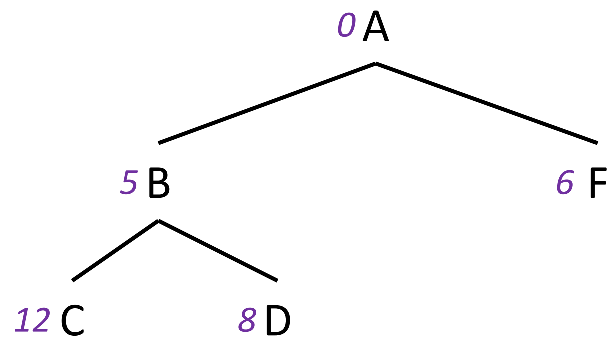
UCS with extended list



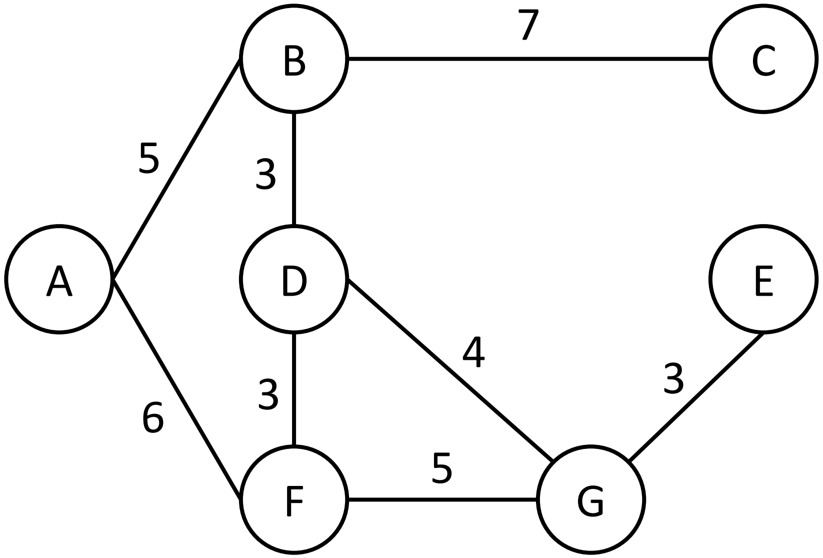


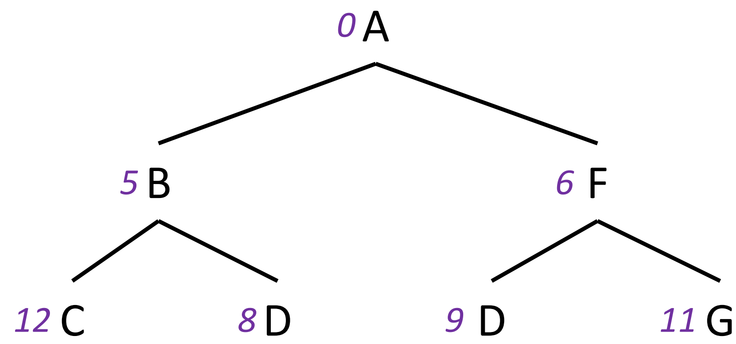
UCS with extended list



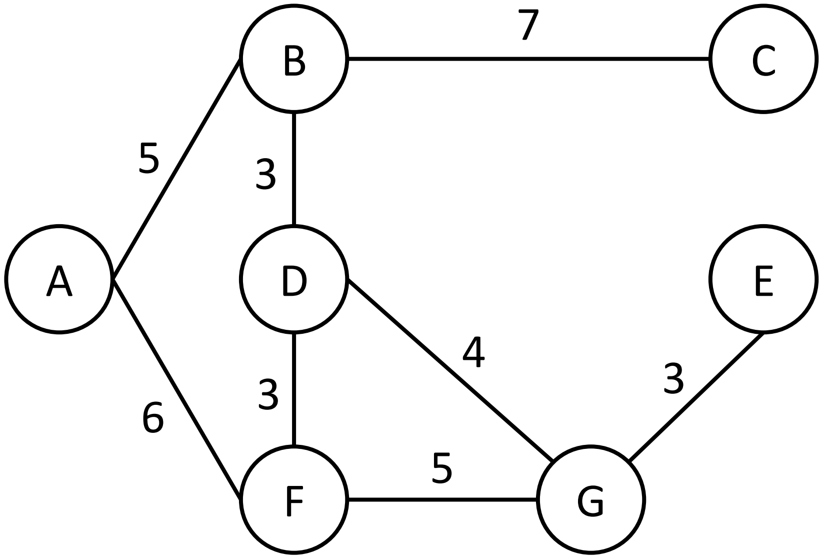


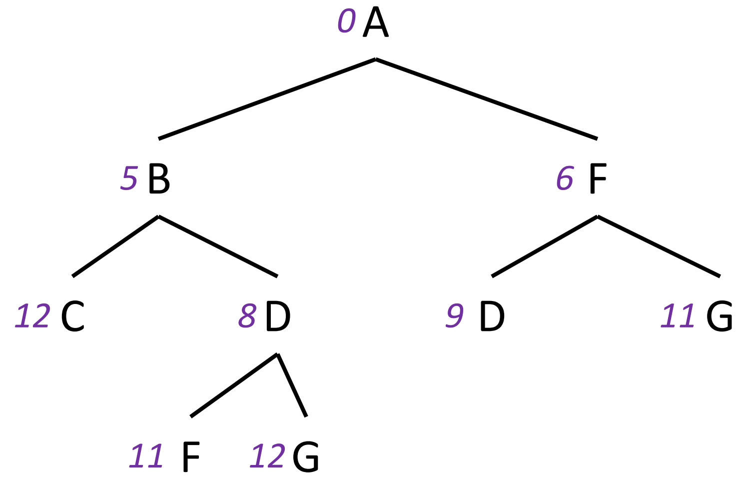
UCS with extended list



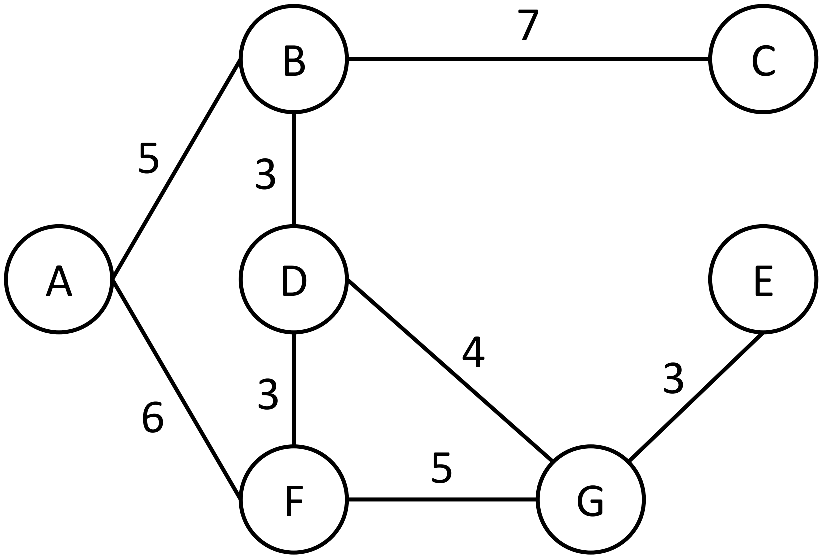


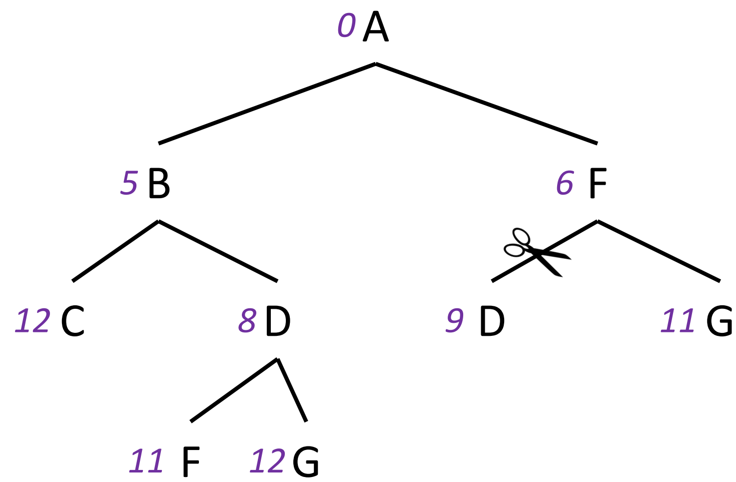
UCS with extended list



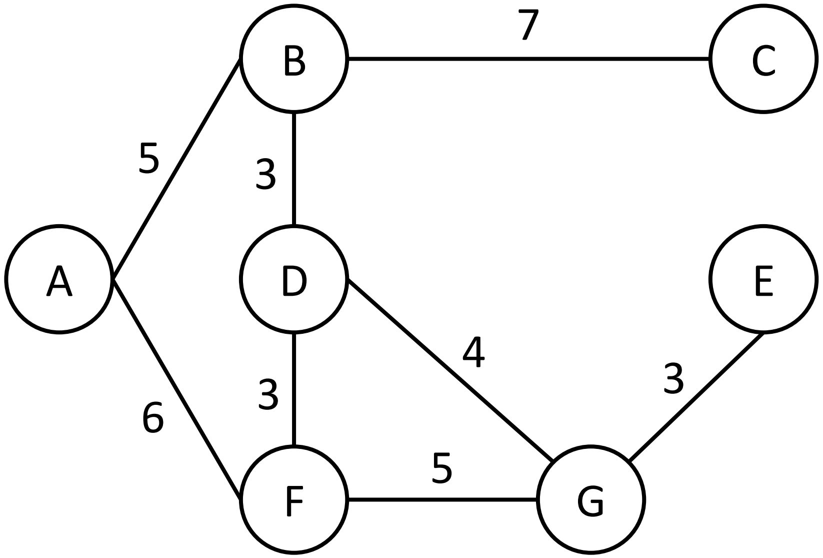


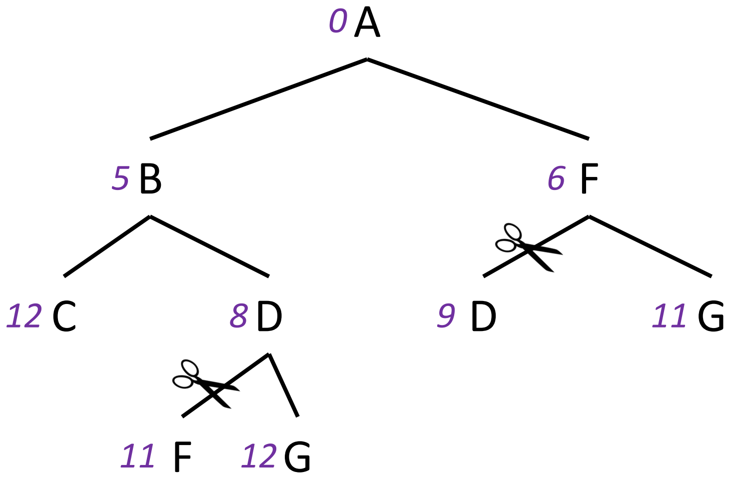
UCS with extended list



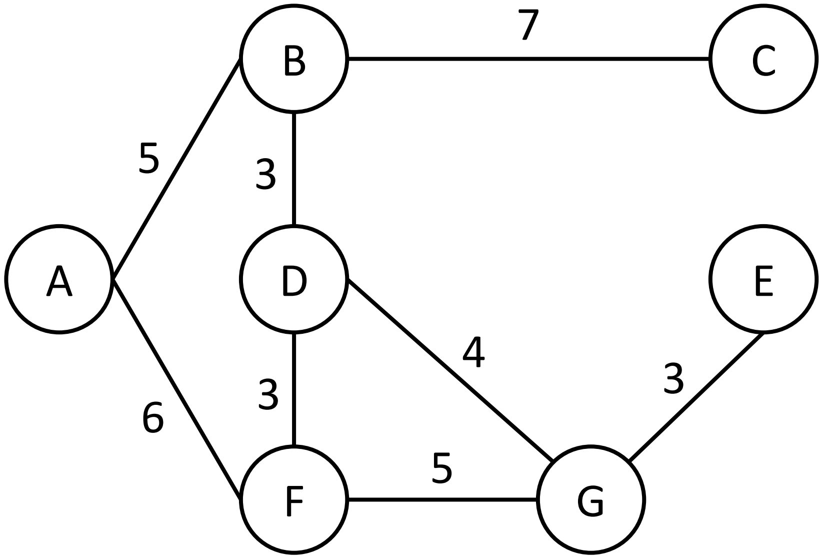


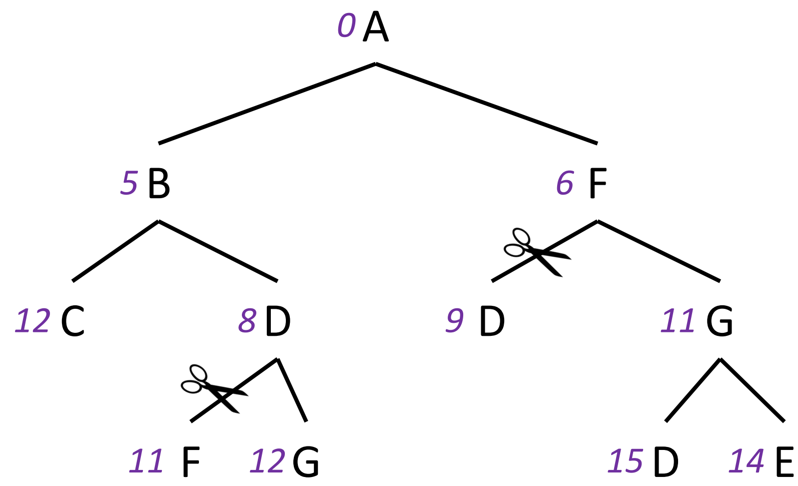
UCS with extended list



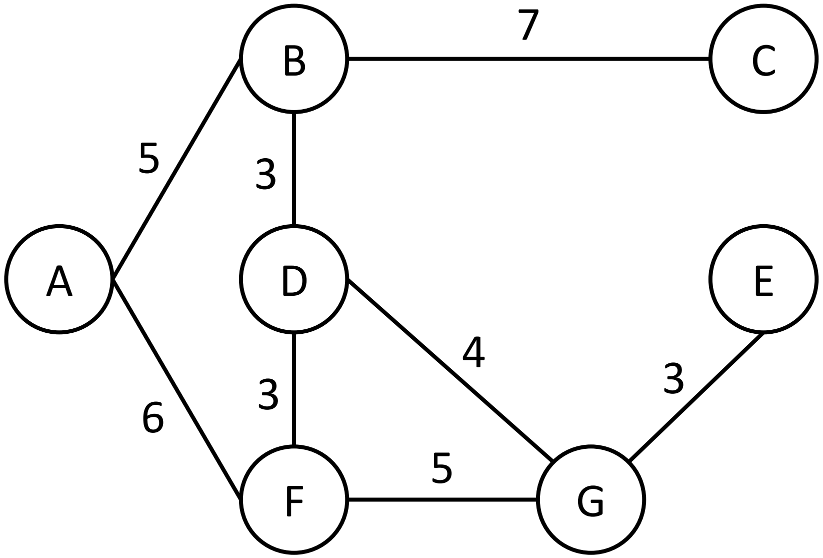


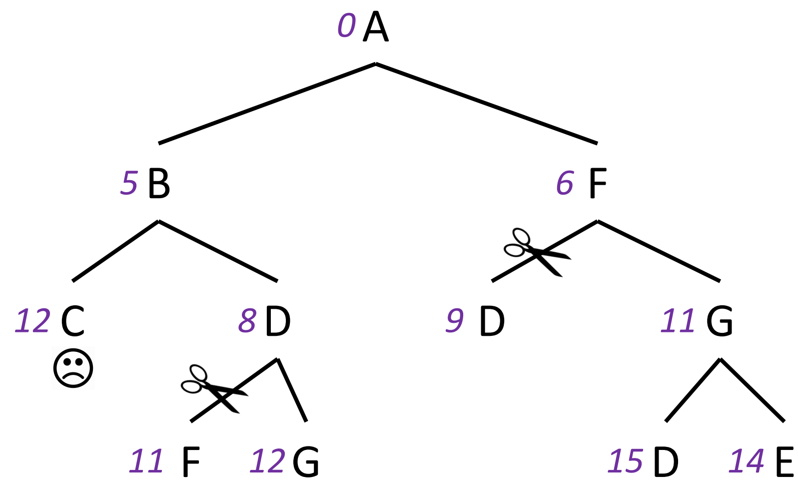
UCS with extended list



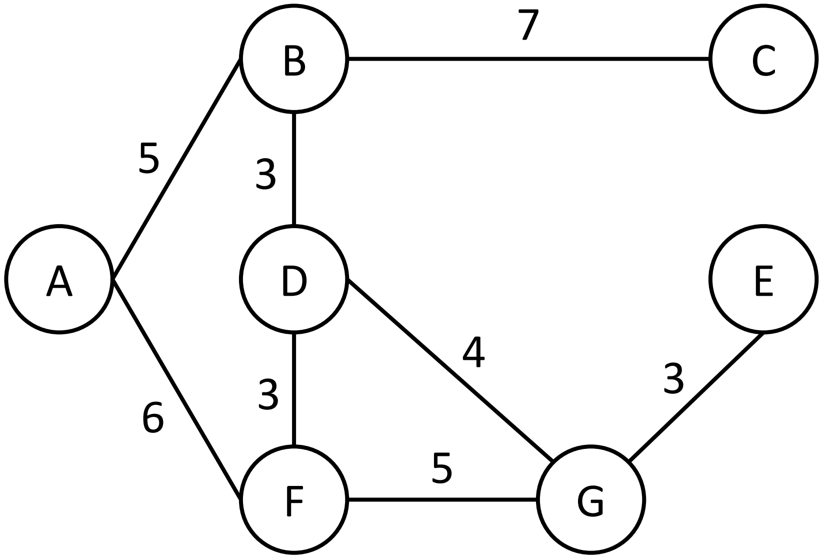


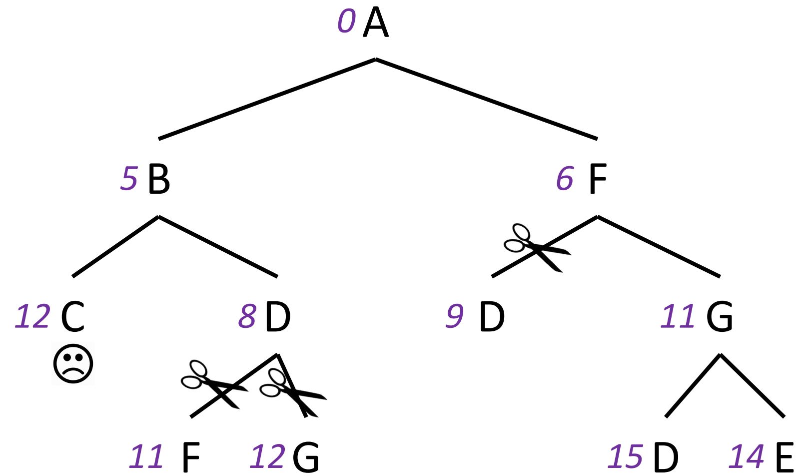
UCS with extended list



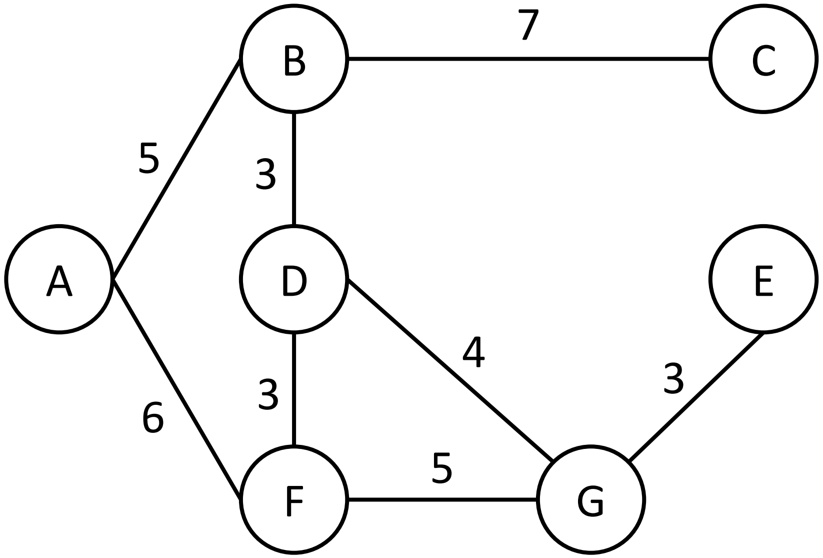


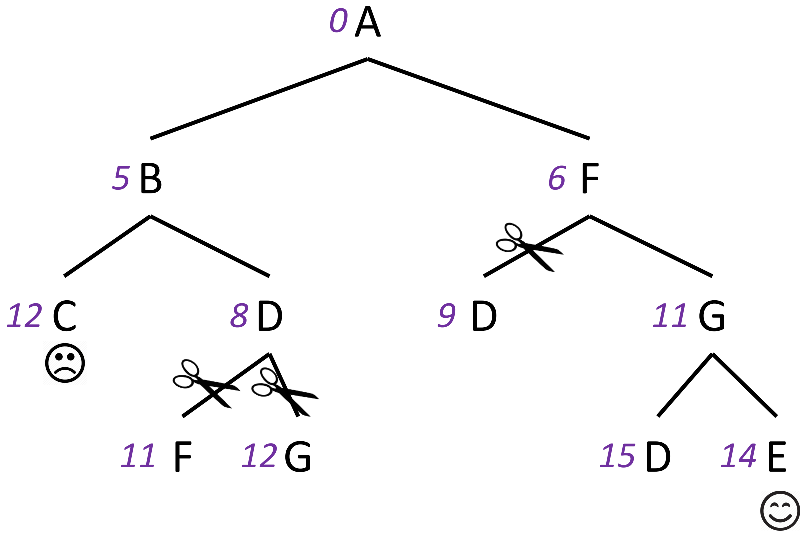
UCS with extended list





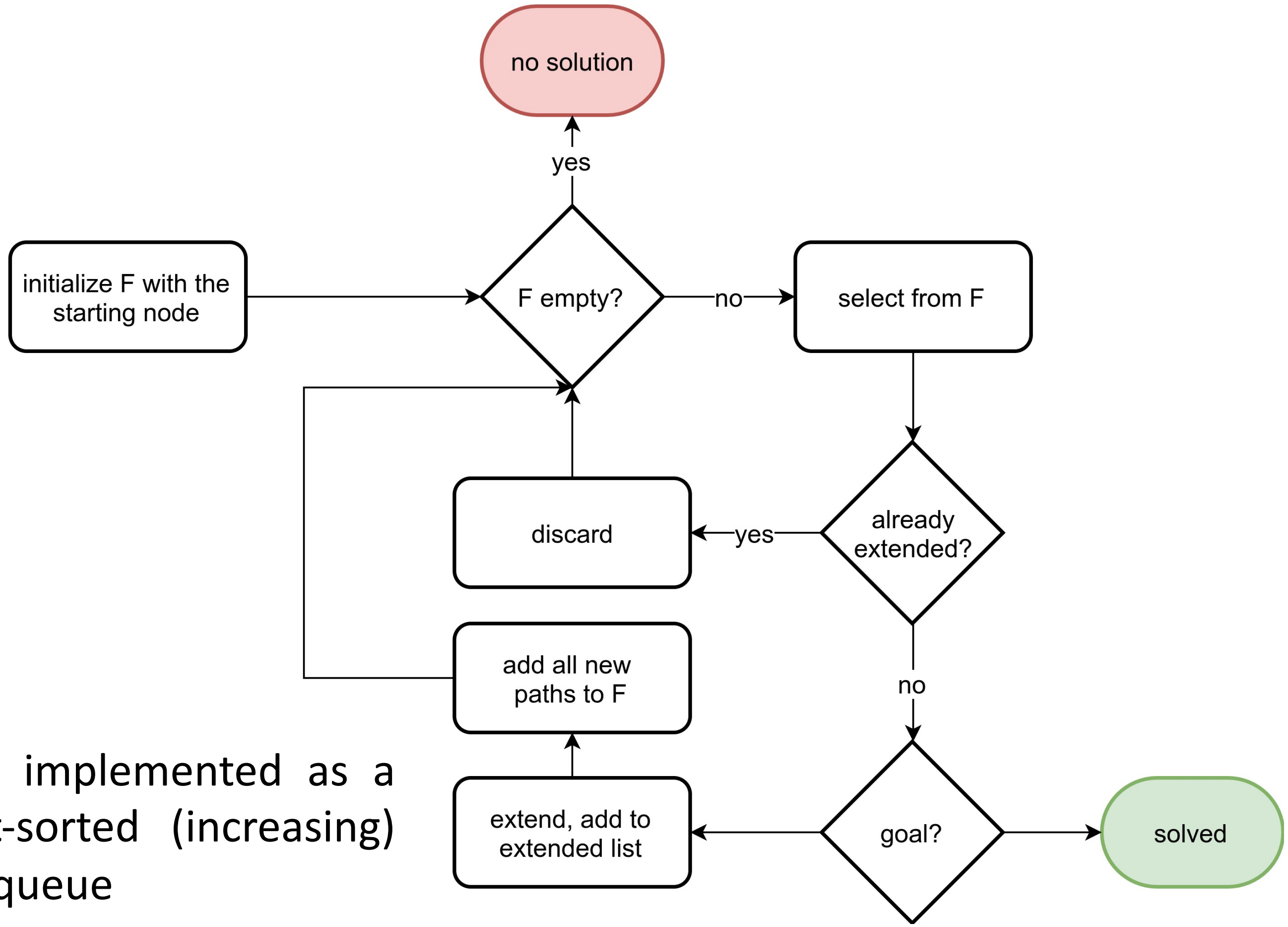
UCS with extended list





- Thanks to the extended list we can prune two branches

Implementation



F is implemented as a cost-sorted (increasing) list queue

The goal check is done when the node is selected (not when is generated)

- Question: is this search informed?

Summing up

b branching factor,
q depth of the shallowest solution,
m maximum depth of search tree,
l depth limit

Criterion	BFS	UCS	DFS	Limited DFS	Iterative DFS
Complete?	Yes (if b finite)	Yes (if b finite and cost positive)	No (only for finite spaces)	No ($l > q$)	Yes (if b finite)
Time com.	$O(b^q)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^q)$
Space com.	$O(b^q)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bq)$
Optimal?	Yes (identical costs)	Yes	No	No	Yes (identical costs)

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

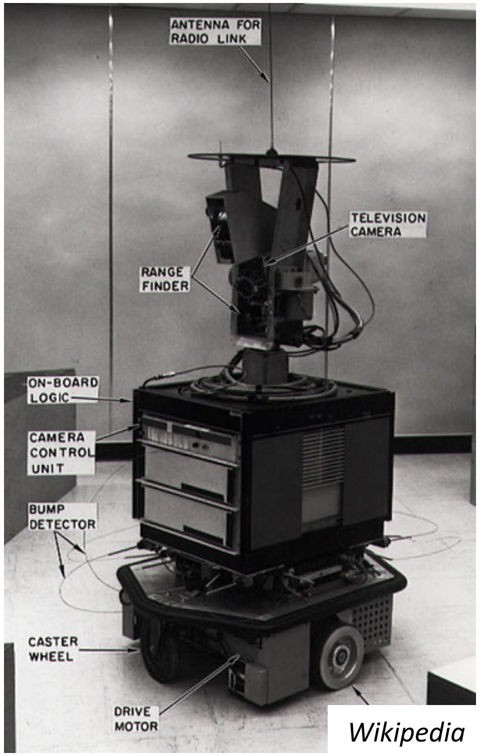
- Both search methods break ties in lexicographical order, but it seems reasonable to do that in favor of nodes that are believed to be closer to the goal

- **Hill climbing**
  - A DFS where ties are broken in favor the node with smallest h

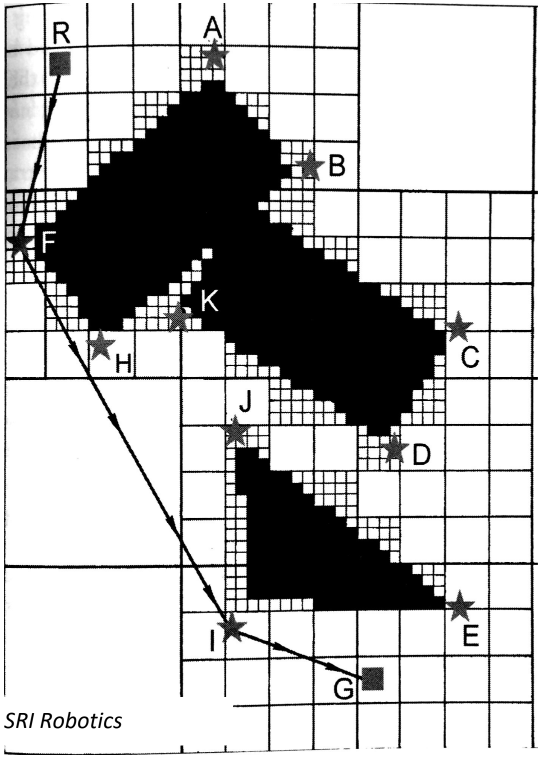
- **Beam** (of width w)
  - A BFS where at each level we keep the first w nodes in increasing order of h

A*

- The informed version of UCS is called A*
- Very popular search algorithm
- It was born in the early days of mobile robotics when, in 1968, Nilsson, Hart, and Raphael had to face a practical problem with Shakey (one of the ancestors of today's mobile robots)



Wikipedia



SRI Robotics

A*

- The idea behind A* is simple: perform a UCS, but instead of considering accumulated costs consider the following:

Heuristic
("cost-to-go")

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

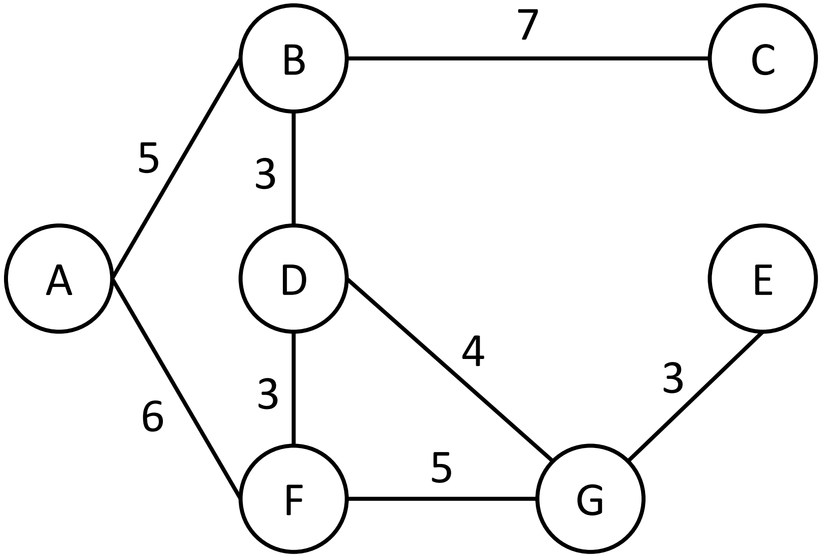
Cost accumulated
on the path to n
("cost-to-come")

- To guarantee that the search is sound and complete we need to require that the heuristic is **admissible**: it is an optimistic estimate or, more formally:

$$h(n) \leq \text{ Cost of the minimum path from n to the goal}$$

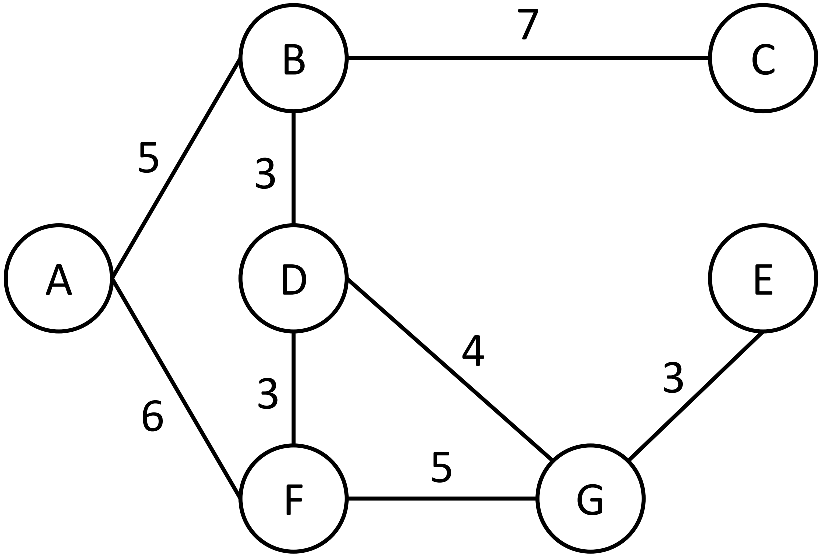
- If the heuristic is not admissible we might discard a path that could actually turn out to be better that the best candidate found so far

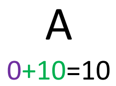
A*



node v	$h(v)$
A	10
B	7
C	1
D	3
E	0
F	7
G	2

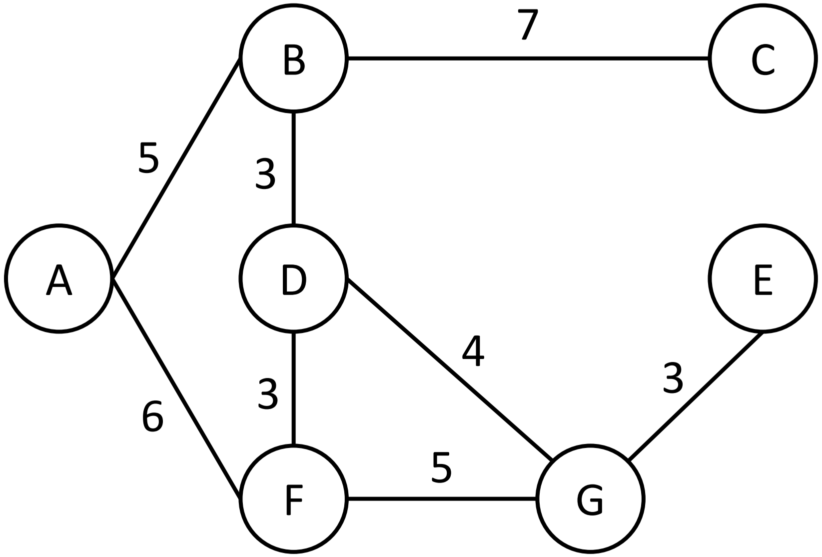
A*

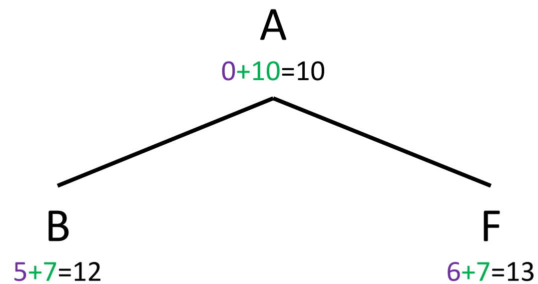




node v	$h(v)$
A	10
B	7
C	1
D	3
E	0
F	7
G	2

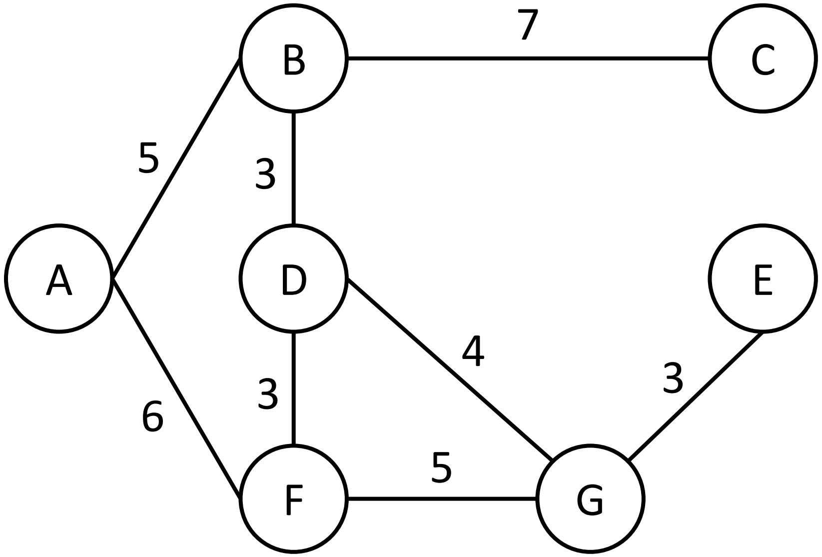
A*

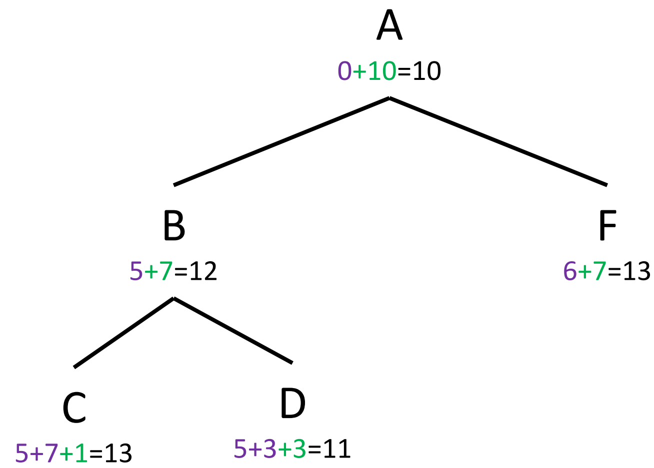




node v	$h(v)$
A	10
B	7
C	1
D	3
E	0
F	7
G	2

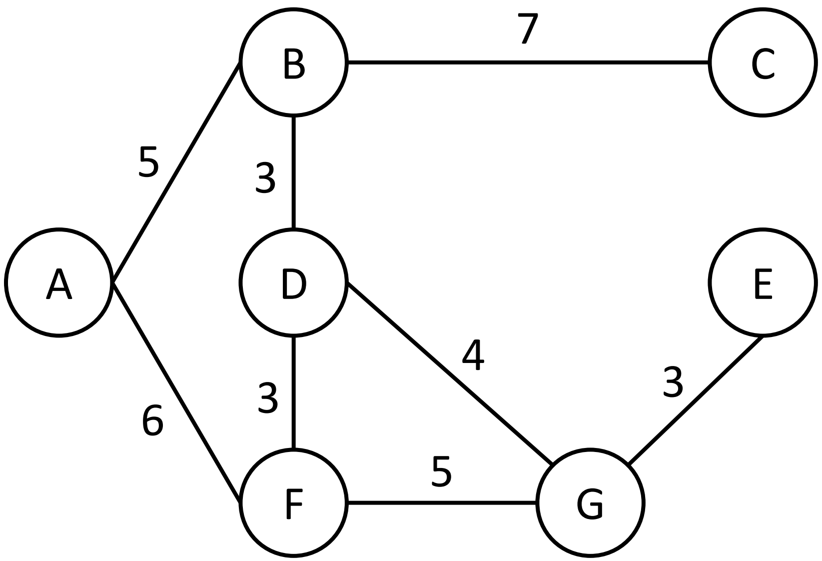
A*

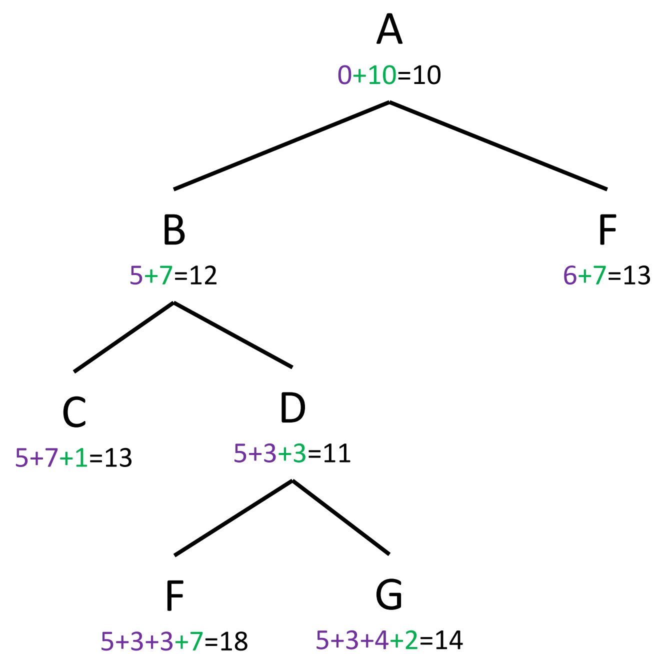




node v	$h(v)$
A	10
B	7
C	1
D	3
E	0
F	7
G	2

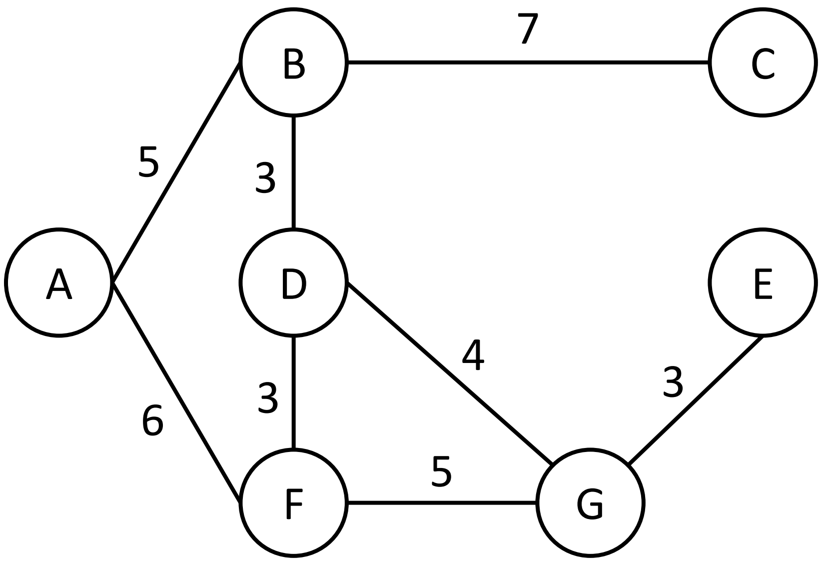
A*

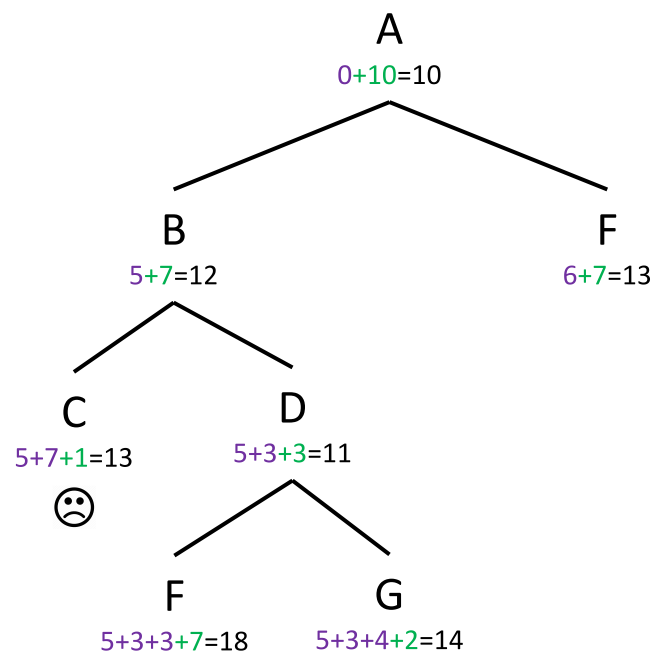




node v	$h(v)$
A	10
B	7
C	1
D	3
E	0
F	7
G	2

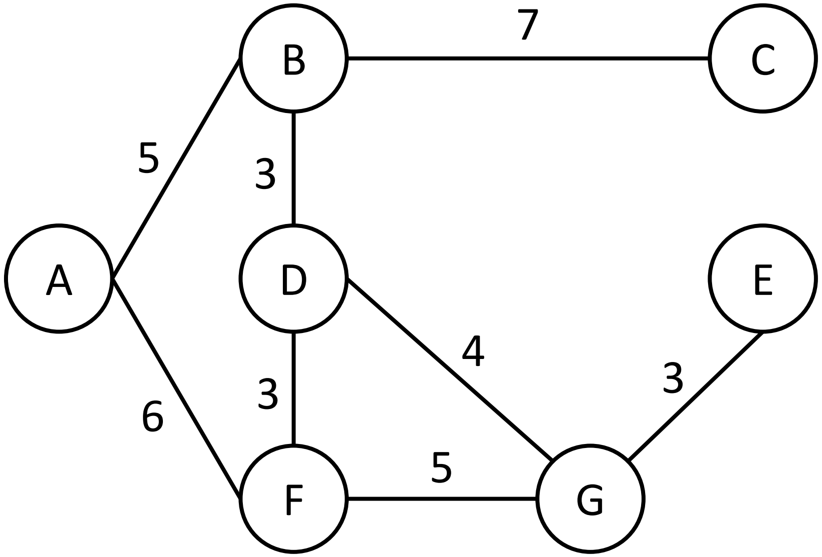
A*

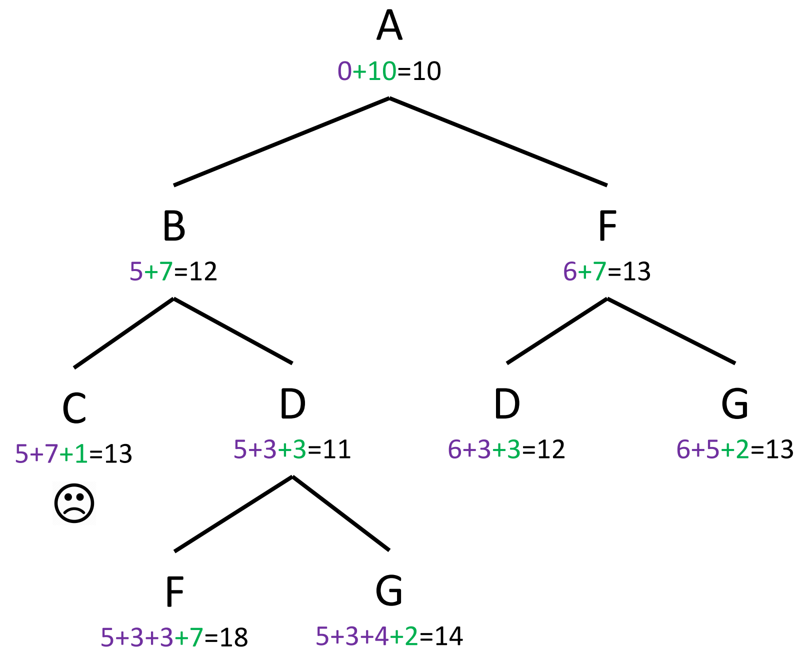




node v	$h(v)$
A	10
B	7
C	1
D	3
E	0
F	7
G	2

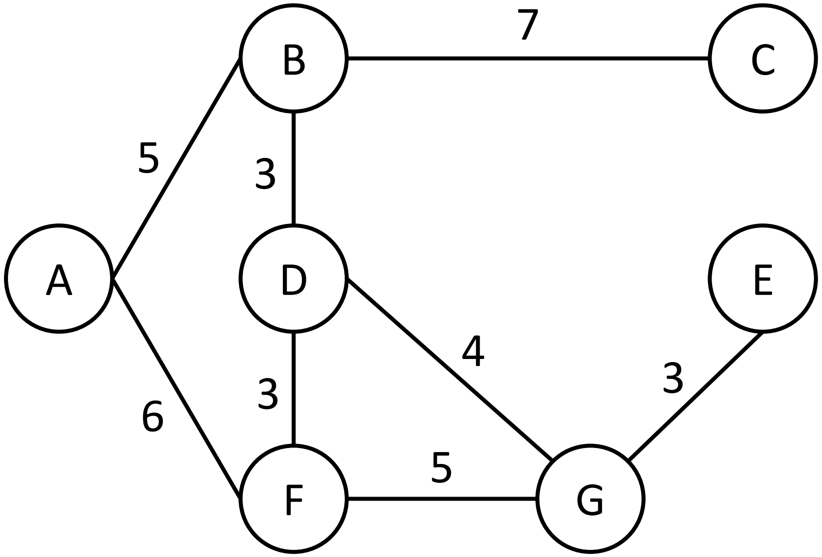
A*

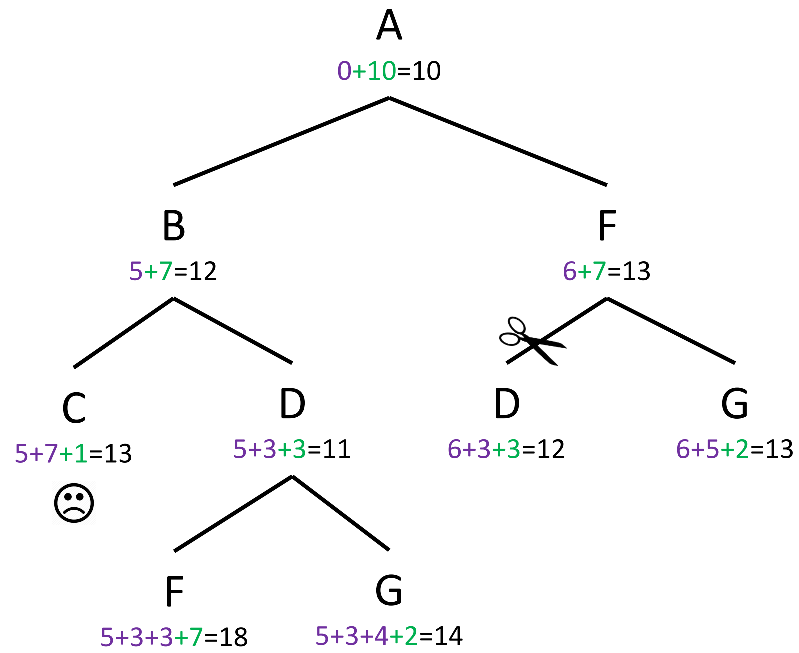




node v	$h(v)$
A	10
B	7
C	1
D	3
E	0
F	7
G	2

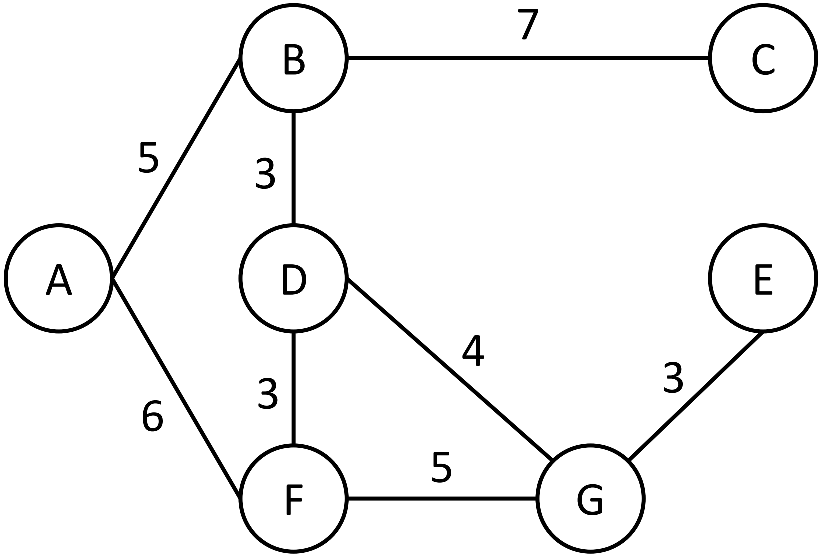
A*

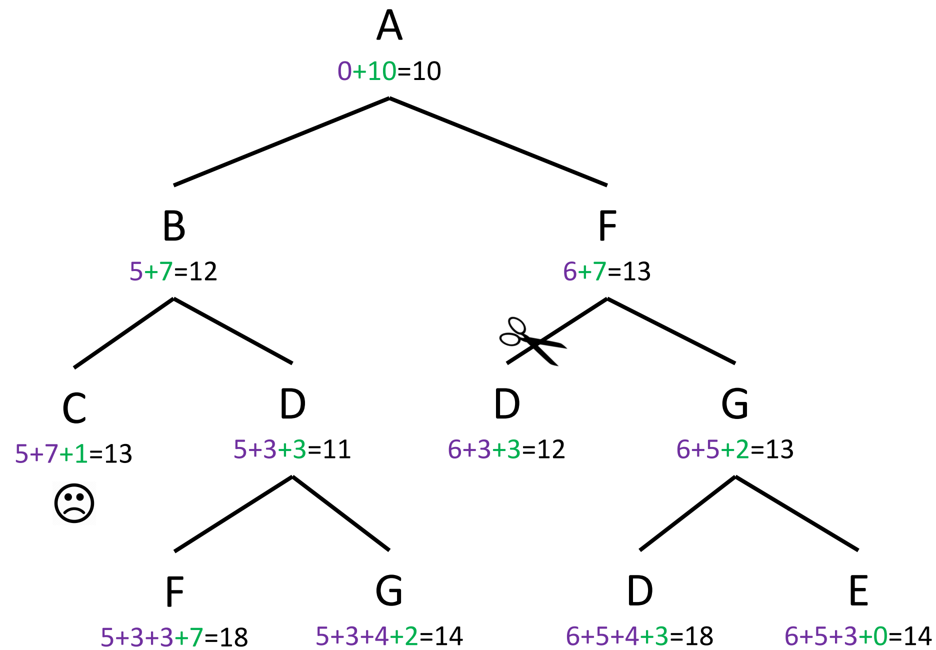




node v	$h(v)$
A	10
B	7
C	1
D	3
E	0
F	7
G	2

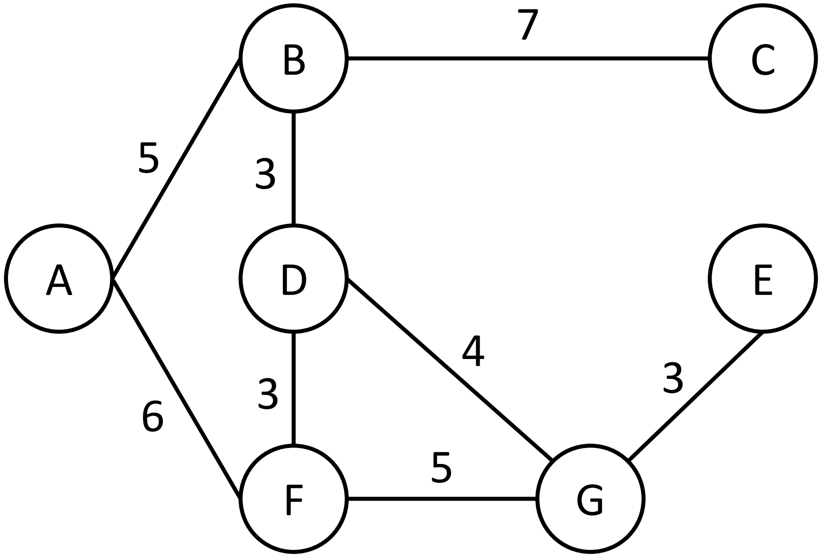
A*



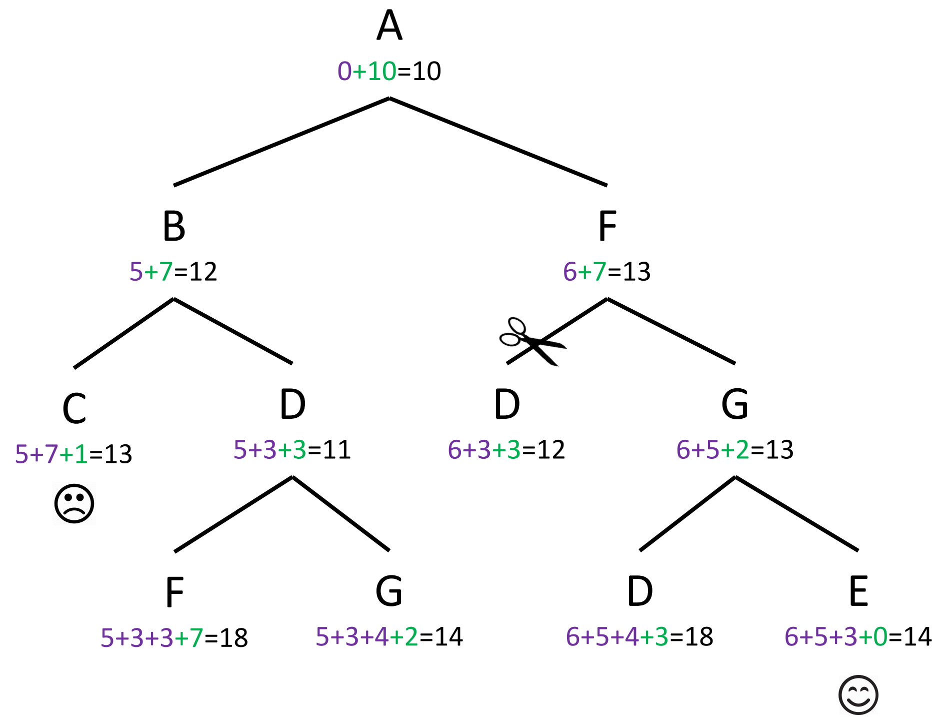


node v	$h(v)$
A	10
B	7
C	1
D	3
E	0
F	7
G	2

A*



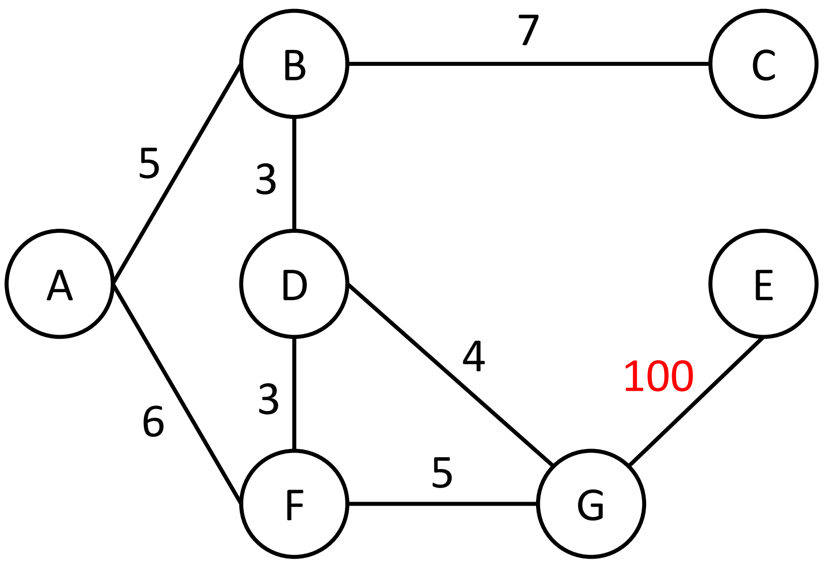
node v	$h(v)$
A	10
B	7
C	1
D	3
E	0
F	7
G	2



A*

- Problem: if we work with an extended list, admissibility is not enough!

- Let’s consider this “pathological" instance:

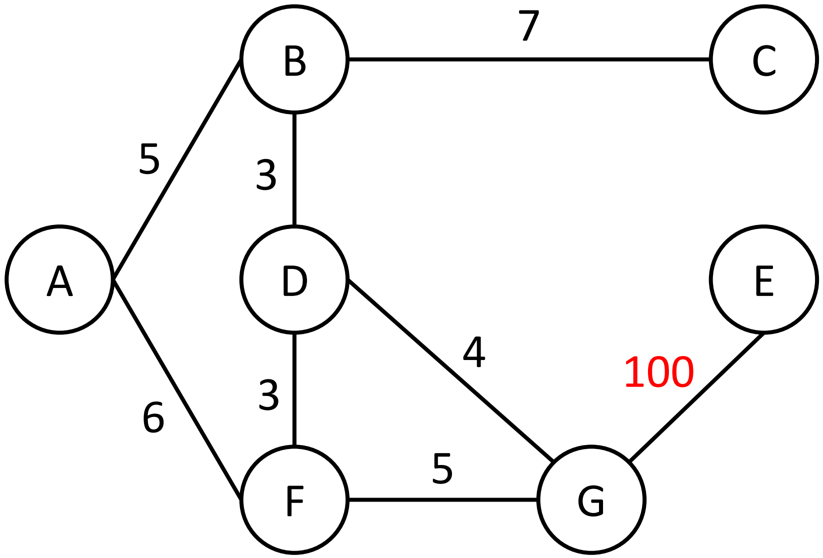


node v	$h(v)$
A	10
B	0
C	1
D	0
E	0
F	100
G	0

A*

- Problem: if we work with an extended list, admissibility is not enough!

- Let's consider this "pathological" instance:

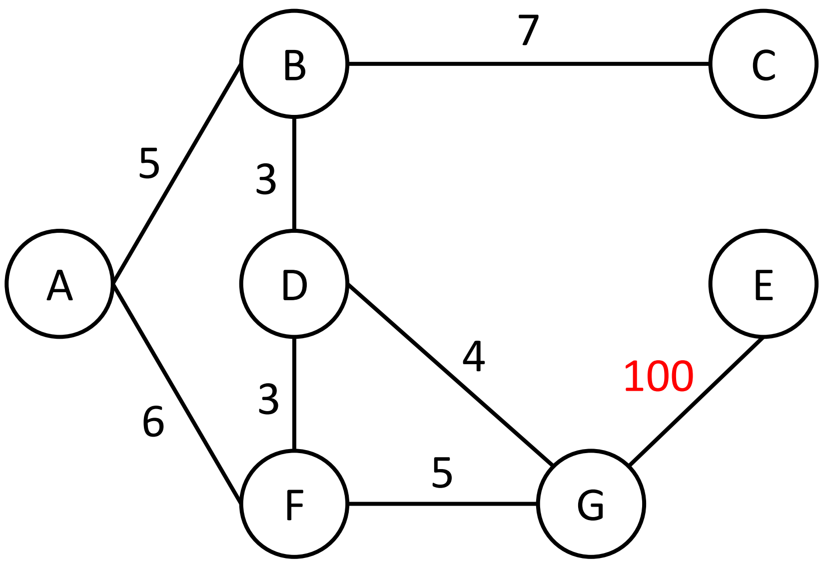


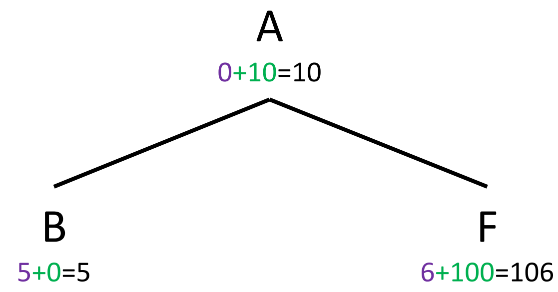
A
0+10=10

node v	$h(v)$
A	10
B	0
C	1
D	0
E	0
F	100
G	0

A*

- Problem: if we work with an extended list, admissibility is not enough!
- Let's consider this "pathological" instance:



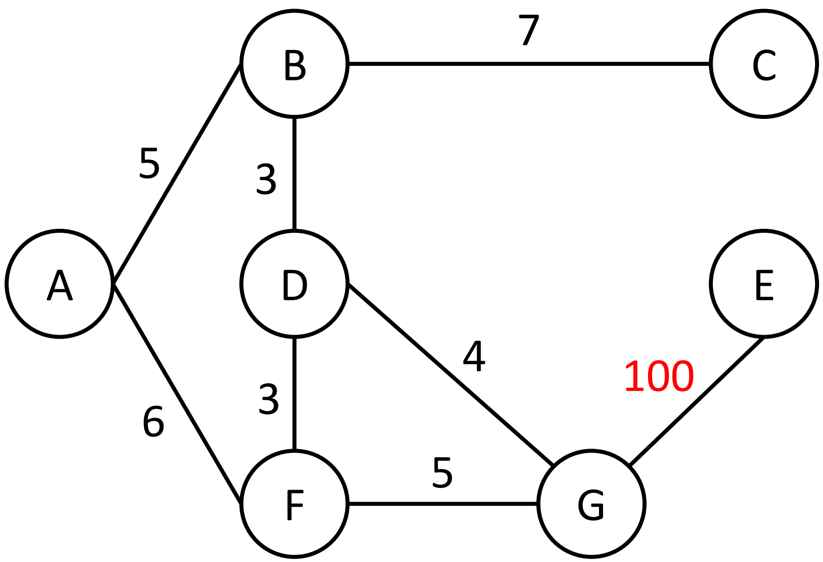


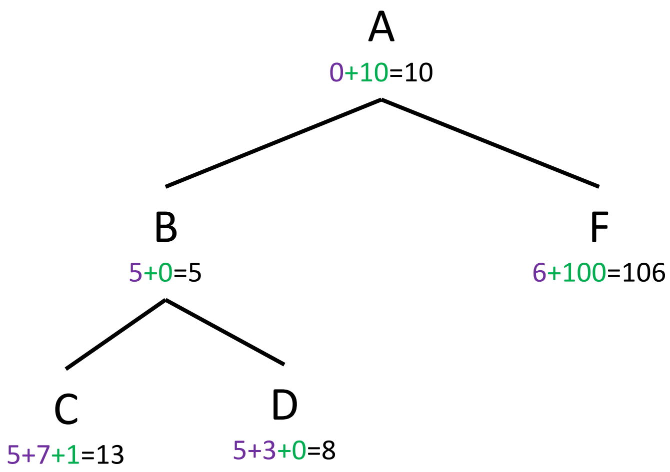
node v	$h(v)$
A	10
B	0
C	1
D	0
E	0
F	100
G	0

A*

- Problem: if we work with an extended list, admissibility is not enough!

- Let's consider this "pathological" instance:



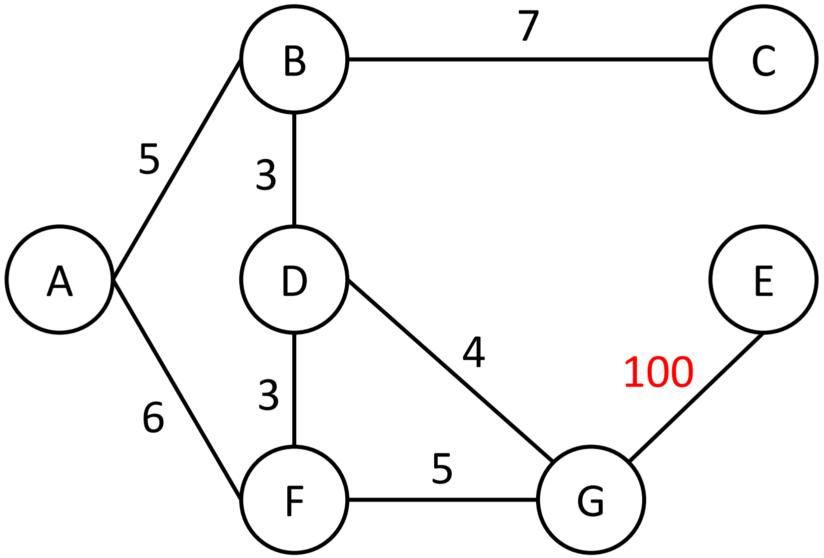


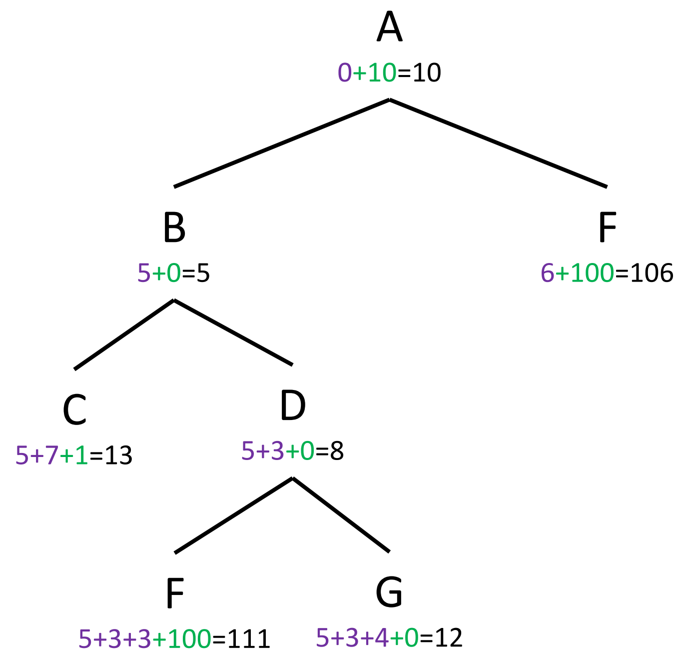
node v	$h(v)$
A	10
B	0
C	1
D	0
E	0
F	100
G	0

A*

- Problem: if we work with an extended list, admissibility is not enough!

- Let's consider this "pathological" instance:



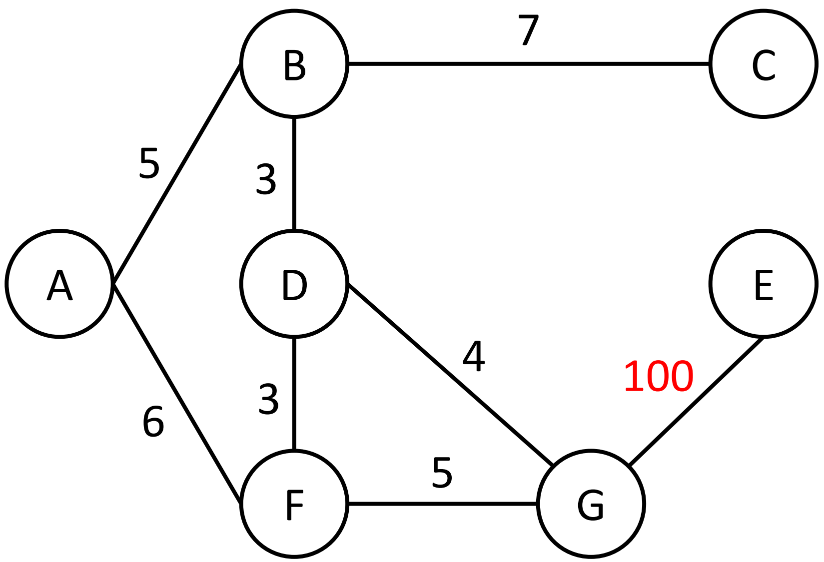


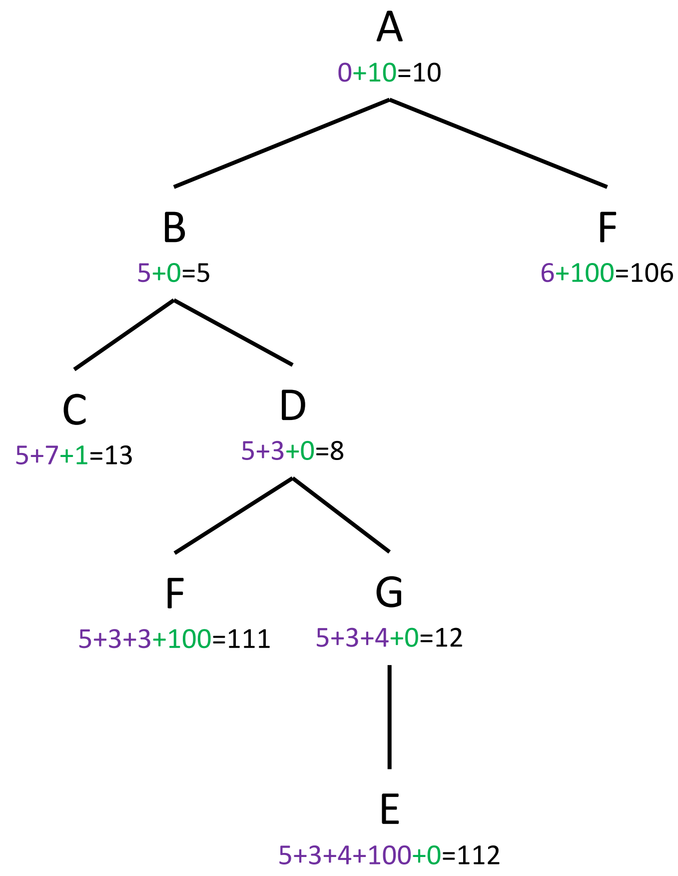
node v	$h(v)$
A	10
B	0
C	1
D	0
E	0
F	100
G	0

A*

- Problem: if we work with an extended list, admissibility is not enough!

- Let's consider this "pathological" instance:

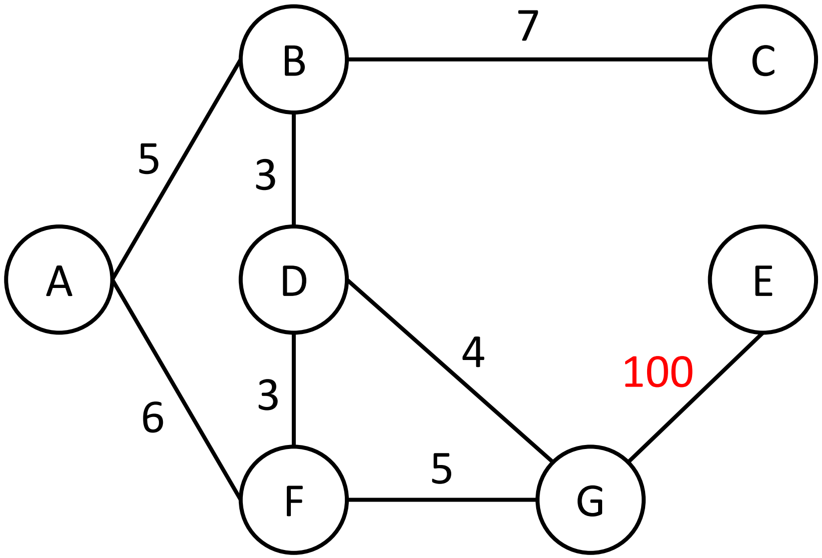




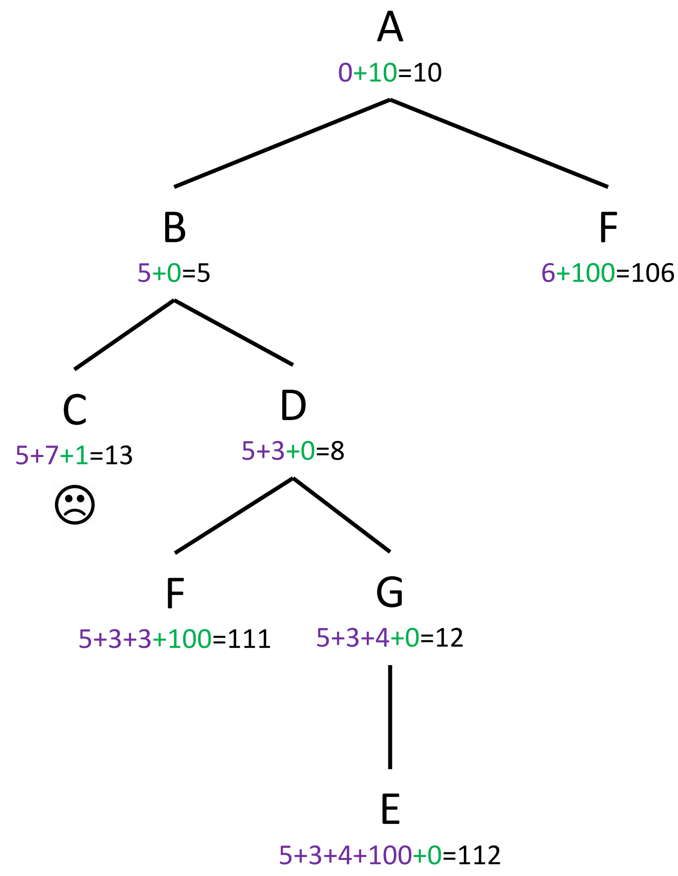
node v	$h(v)$
A	10
B	0
C	1
D	0
E	0
F	100
G	0

A*

- Problem: if we work with an extended list, admissibility is not enough!
- Let's consider this "pathological" instance:

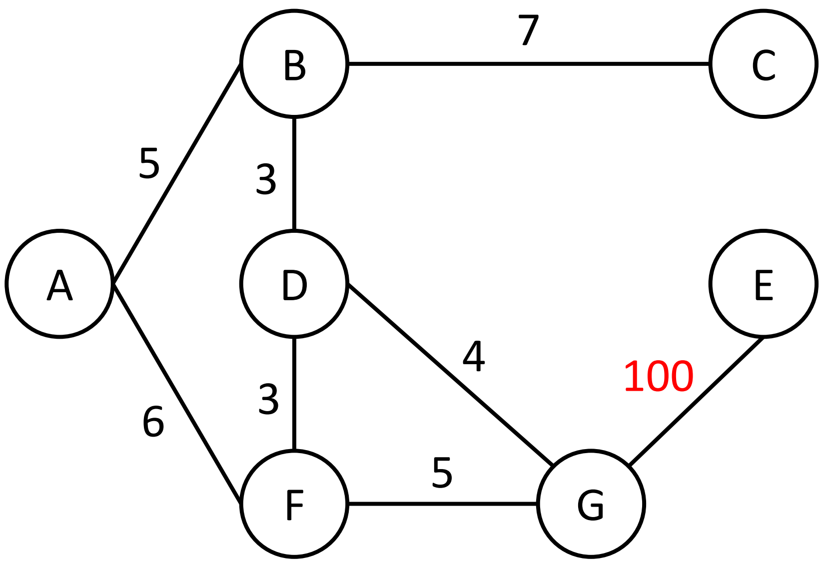


node v	$h(v)$
A	10
B	0
C	1
D	0
E	0
F	100
G	0

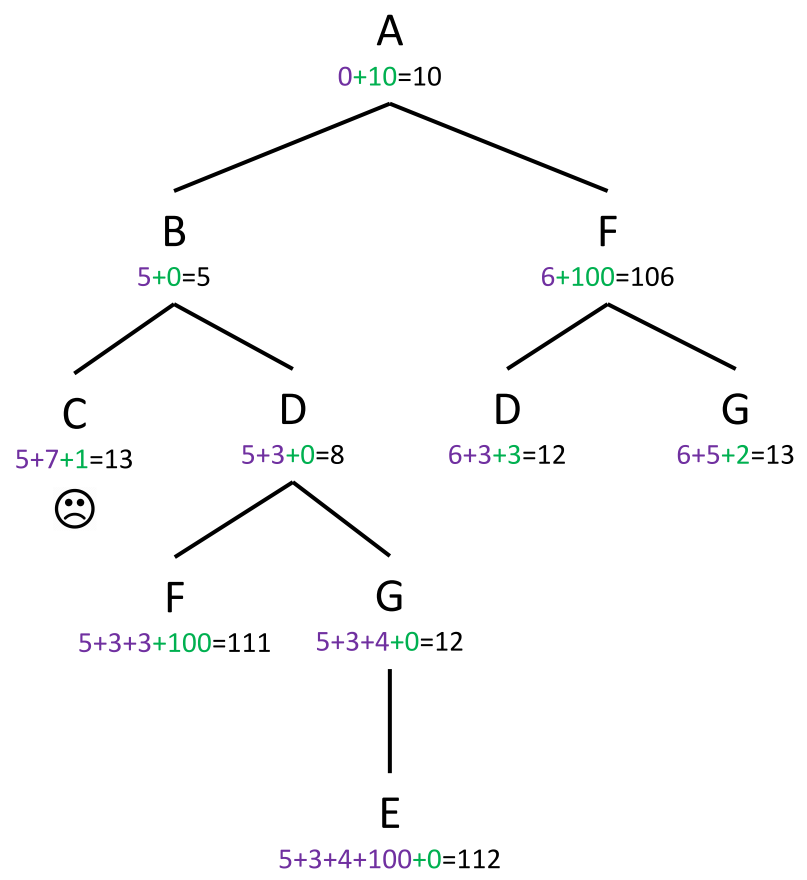


A*

- Problem: if we work with an extended list, admissibility is not enough!
- Let’s consider this “pathological" instance:



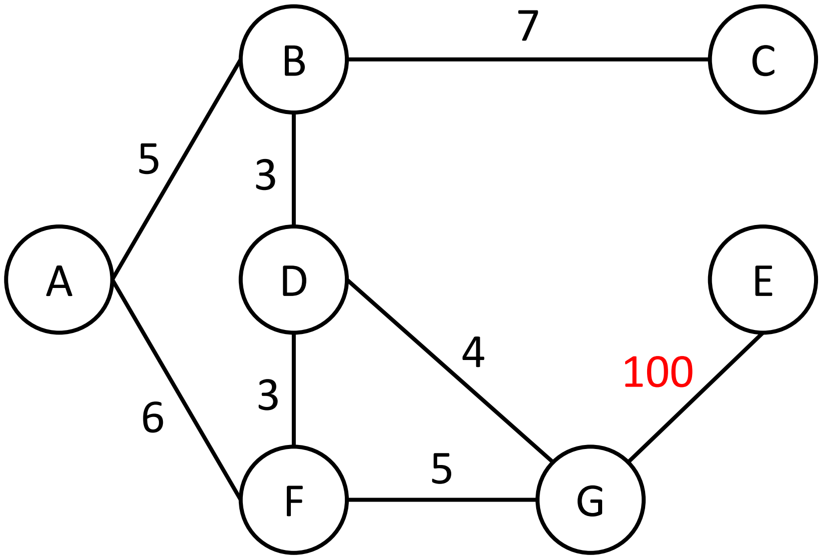
node v	$h(v)$
A	10
B	0
C	1
D	0
E	0
F	100
G	0



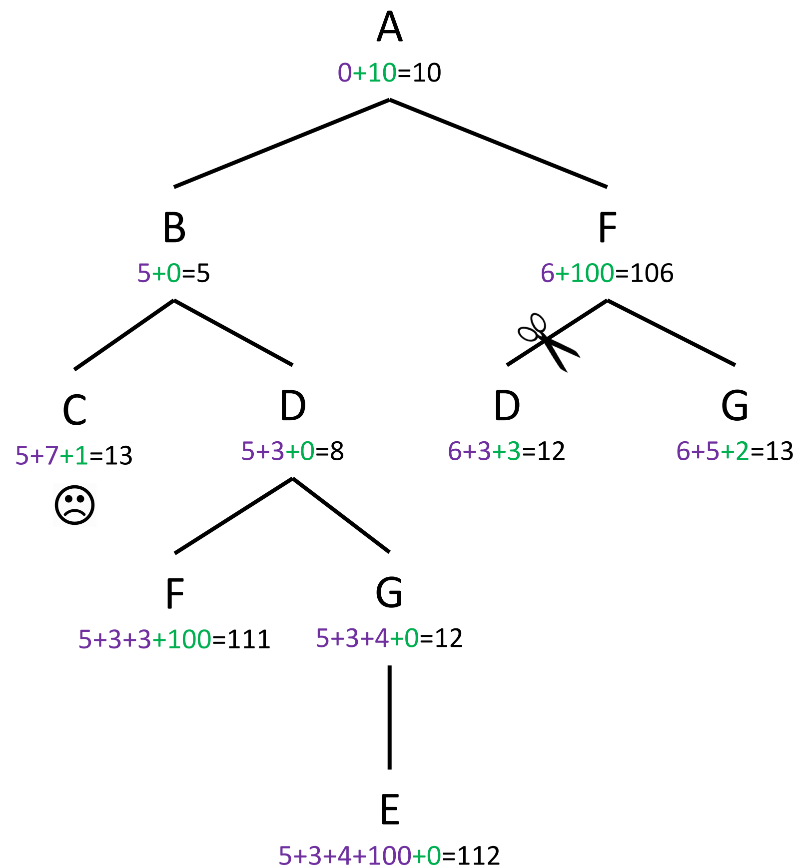
A*

- Problem: if we work with an extended list, admissibility is not enough!

- Let's consider this "pathological" instance:

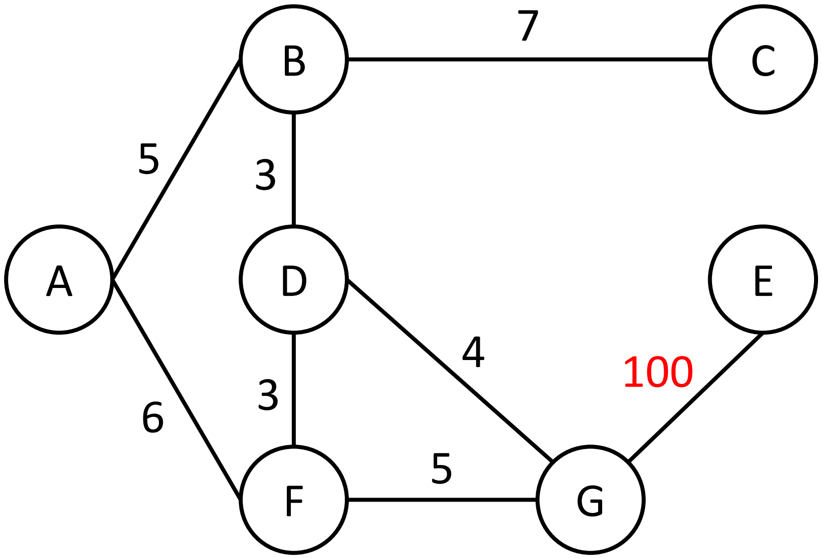


node v	$h(v)$
A	10
B	0
C	1
D	0
E	0
F	100
G	0

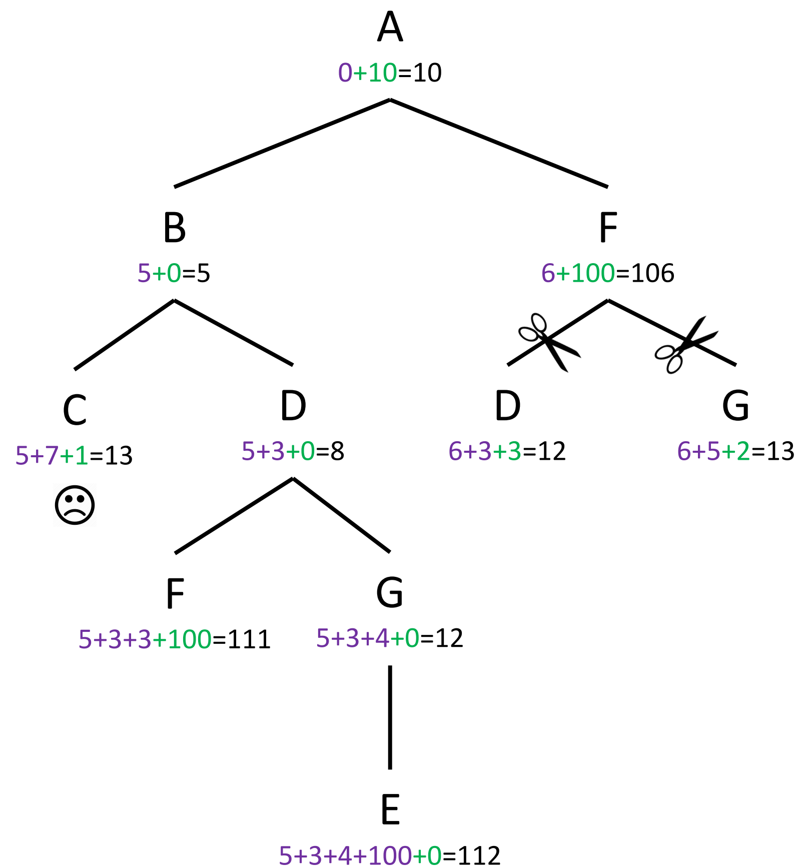


A*

- Problem: if we work with an extended list, admissibility is not enough!
- Let’s consider this “pathological" instance:



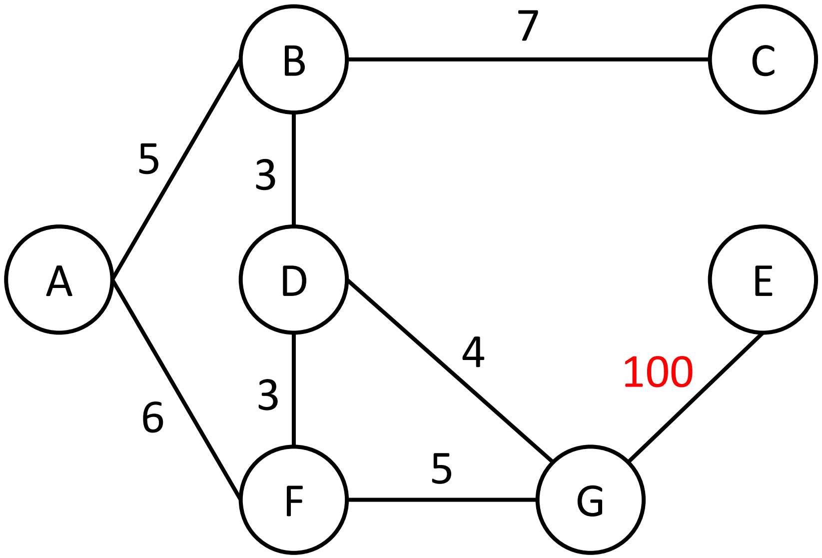
node v	$h(v)$
A	10
B	0
C	1
D	0
E	0
F	100
G	0



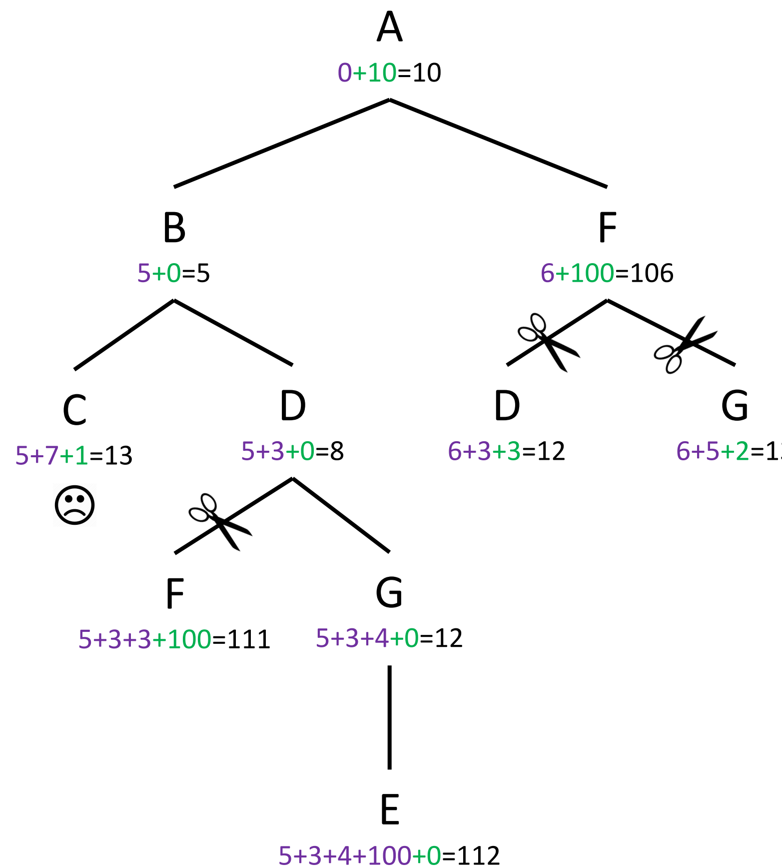
A*

- Problem: if we work with an extended list, admissibility is not enough!

- Let's consider this "pathological" instance:

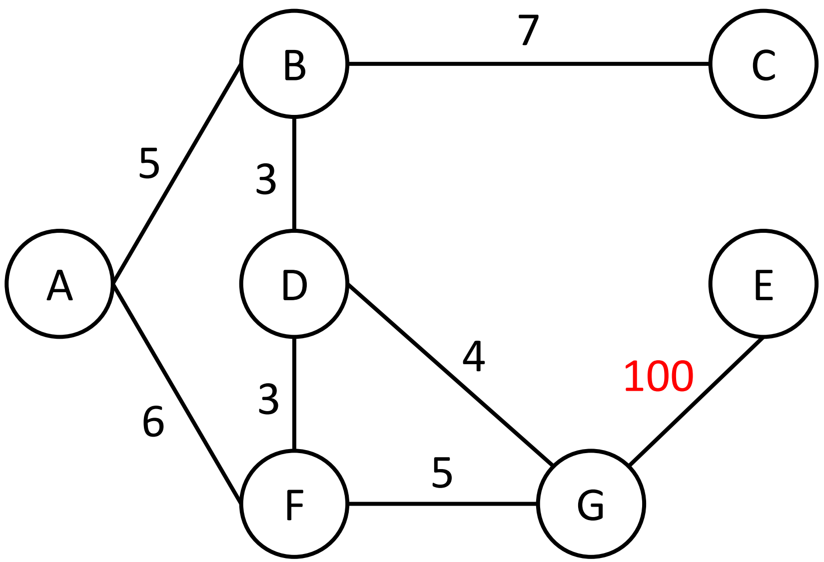


node v	$h(v)$
A	10
B	0
C	1
D	0
E	0
F	100
G	0

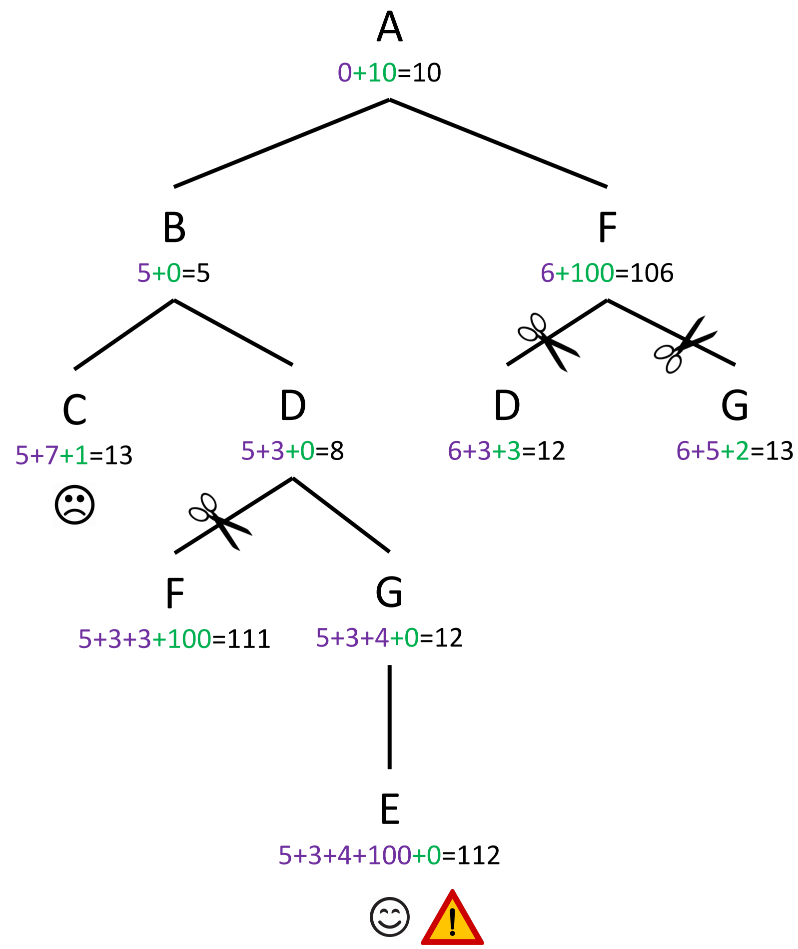


A*

- Problem: if we work with an extended list, admissibility is not enough!
- Let's consider this "pathological" instance:

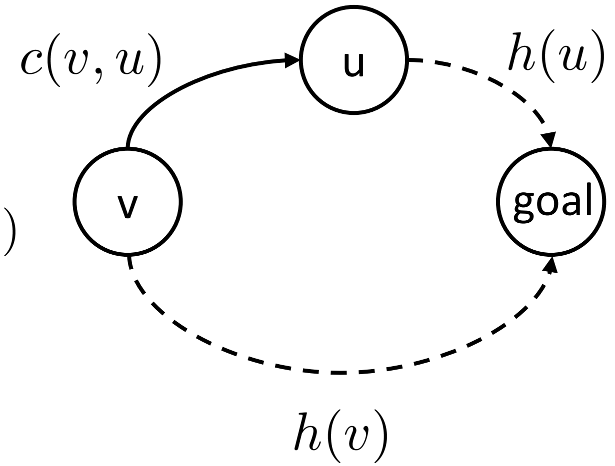


node v	$h(v)$
A	10
B	0
C	1
D	0
E	0
F	100
G	0



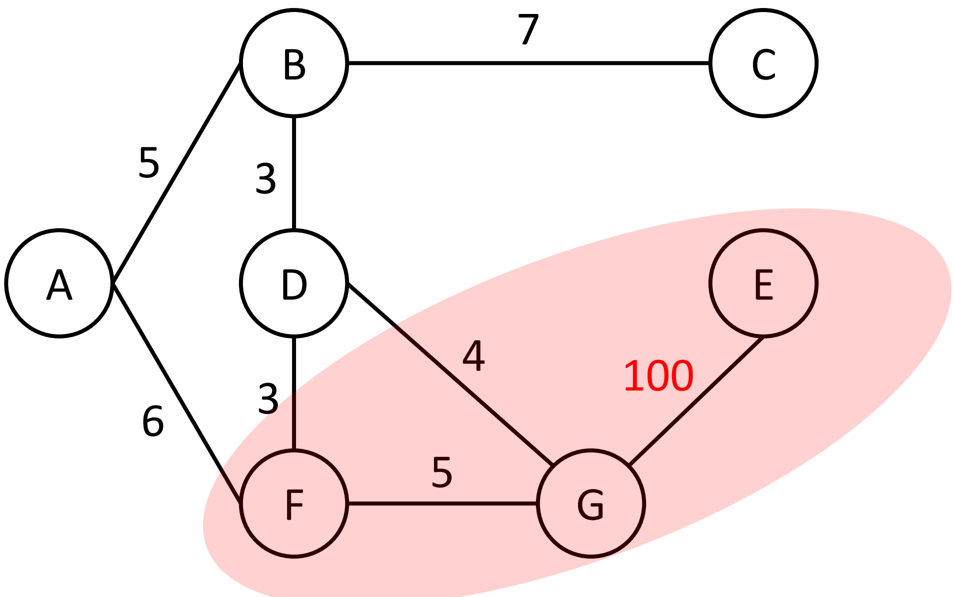
A*

- We need to require a stronger property: **consistency**
- For any connected nodes u and v: $h(v) \leq c(v,u) + h(u)$



- It's a sort of triangle inequality, let's reconsider our pathological instance:

node v	$h(v)$
A	10
B	0
C	1
D	0
E	0
F	100
G	0



Optimality of A*

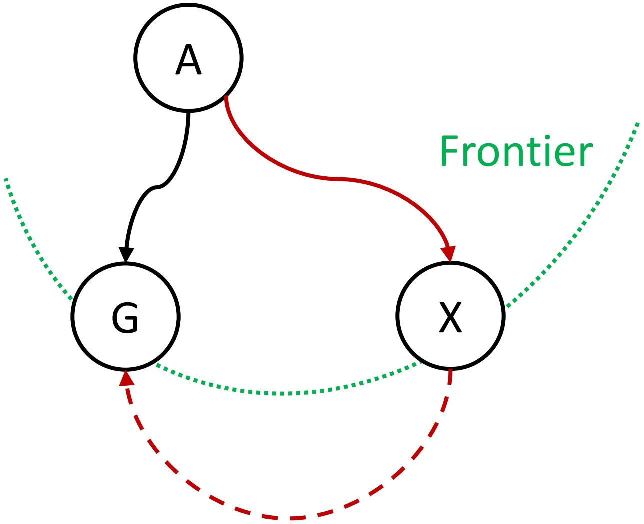
$$f(v) = g(v) + h(v)$$

$$f(u) = \overline{g(u)} + h(u) = \overline{g(v) + c(v,u)} + h(u) \geq g(v) + h(v)$$

consistency

$f(u) \geq f(v)$ ⟶ f is non-decreasing along any search trajectory

Hypotheses:
1. A* selects from the frontier a node G that has been generated through a path p
2. p is not the optimal path to G



Given 2 and the frontier separation property, we know that there must exist a node X on the frontier that is on a better path to G

f is non-decreasing: $f(G) \geq f(X)$

A* selected G: $f(G) < f(X)$

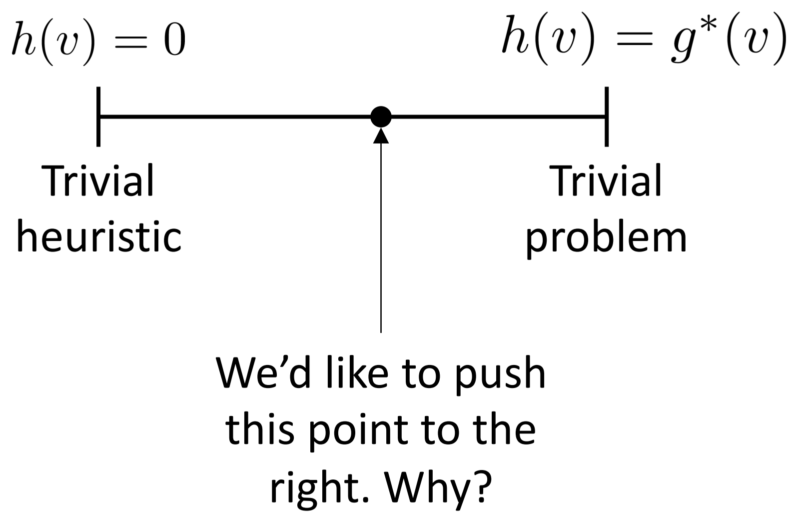
When A selects a node for expansion, it discovers the optimal path to that node*

Building good heuristics

- The “larger heuristics are better” principle is not a methodology to define a good heuristic
- Such a task, seems to be rather complex: heuristics deeply leverage the inner structure of a problem and have to satisfy a number of constraints (admissibility, consistency, efficiency) whose guarantee is not straightforward
- When we adopted the straight-line distance in our route finding examples, we were sure it was a good heuristic
- Would it be possible to generalize what we did with the straight-line distance to define a method to *compute* heuristics for a problem?
- Good news: the answer is yes

Evaluating heuristics

- How to evaluate if an heuristic is good?



- A* will expand all nodes v such that: $f(v) < g^*(goal) \longrightarrow h(v) < g^*(goal) - g(v)$
- If, for any node v $h_1(v) \leq h_2(v)$

  then A* with h_2 will not expand more nodes than A* with h_1, in general h_2 is better (provided that is consistent and can be computed by an efficient algorithm)

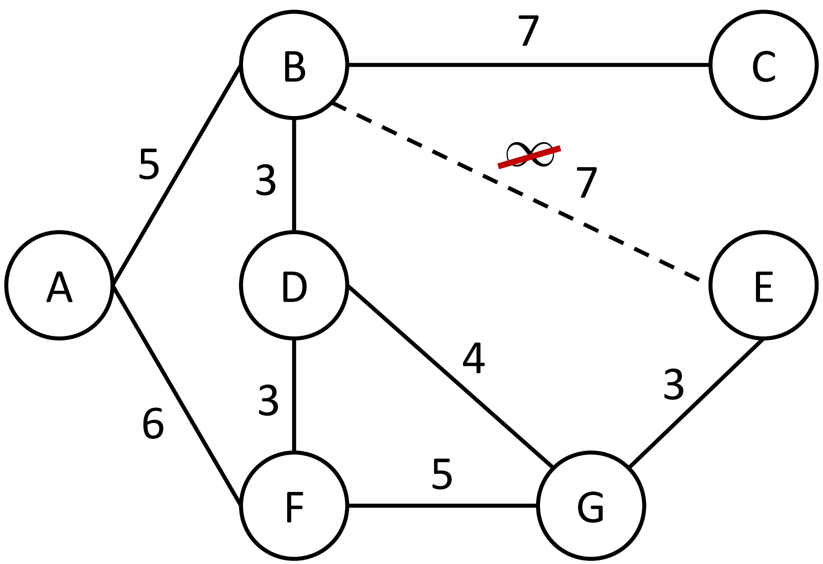
- If we have two consistent heuristics h_1 and h_2 we can define

$$h_3(v) = \max\{h_2(v), h_1(v)\}$$

Relaxed problems

- Given a problem P, a relaxation of P is an easier version of P where some constraints have been dropped

P Original problem —Removing constraints→ $\hat{P}$ Relaxed problem



$$\hat{g}(v,u) \leq g(v,u)$$

Costs in the relaxation

Costs in the original problem

- In our route finding problems removing the constraint that movements should be over roads (links) means that some costs pass from an infinite value to a finite one (the straight-line distance)

Relaxed problems

- Idea:

Define a relaxation of P: $\hat{P}$ → Apply A* to every node and get $\hat{h}^*(v)$ → Set $h(v) = \hat{h}^*(v)$ in the original problem and run A*

- We can easily define a problem relaxation, it's just matter of removing constraints/rewriting costs

- But what happens to soundness and completeness of A*?

$\hat{h}^*(v) \leq \hat{g}(v,u) + \hat{h}^*(u)$ Path costs are optimal

$h(v) \leq \hat{g}(v,u) + h(u)$ From our idea

$\hat{g}(v,u) \leq g(v,u)$ From the definition of relaxation

$h(v) \leq g(v,u) + h(u)$ **h is consistent**

References

- Russel S., Norvig P., Artificial Intelligence, a Modern Approach, III ED
- LaValle, SM., Planning Algorithms http://lavalle.pl/planning/
- https://qiao.github.io/PathFinding.js/visual/
- https://www.redblobgames.com/pathfinding/a-star/introduction.html

Sistemi Intelligenti Avanzati
Corso di Laurea in Informatica, A.A. 2021-2022
Università degli Studi di Milan

Matteo Luperto
Dipartimento di Informatica
matteo.luperto@unimi.it