ARTIFICIAL INTELLIGENCE

LEI/3, LMA/3, MBE/1

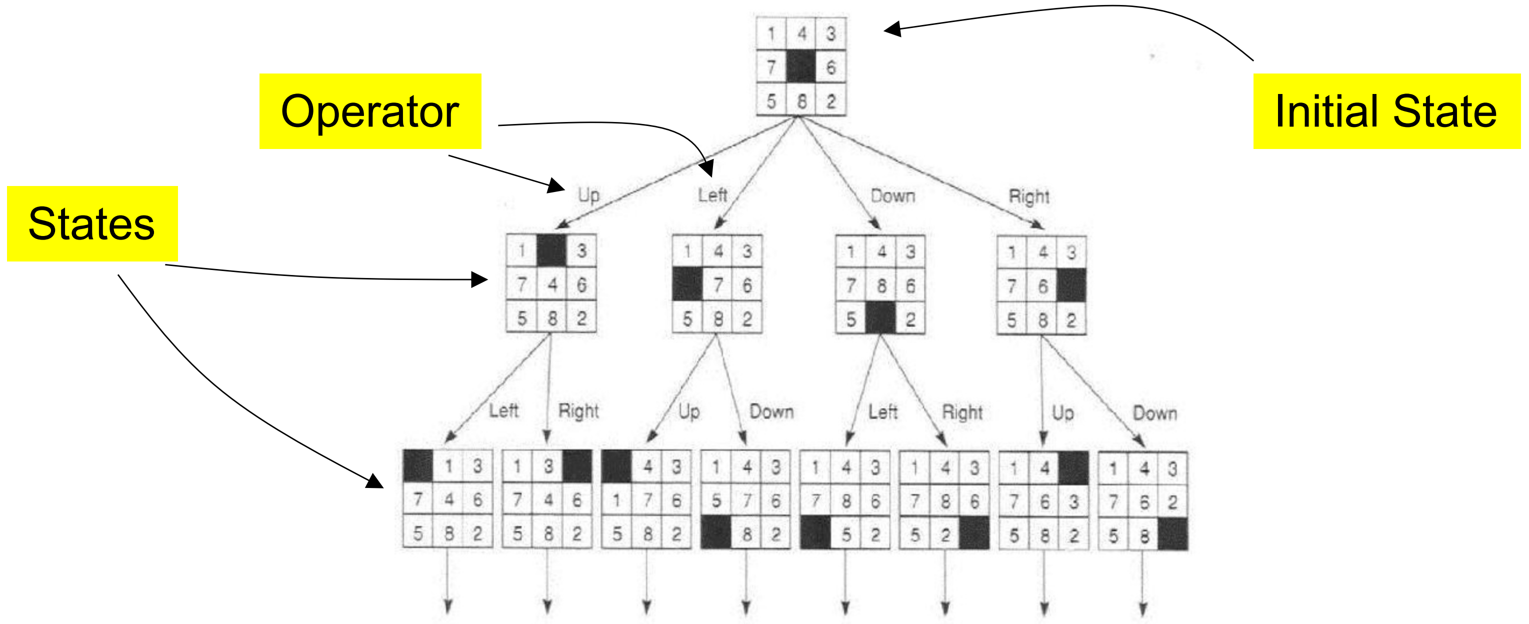
University of Beira Interior, Department of Informatics

Hugo Pedro Proença

hugomcp@di.ubi.pt, **2022/23**

State Space Search

- An important family of problems is about **Search**
  - Search can be applied/used in very heterogenous problems, such as: 1) Finding the solutions for a puzzle; 2) Finding the shortest path between two positions in a map; 3) Proof a theorem; or Finding the sequence of moves to win a *game**
    - *(*)* With game referring to any problem that involves competitors
- Formally, a Search problem is based in two parts:
  - A **State,** that contains all the information necessary to predict the effects of an action and to determine whether a state satisfies the goal. We assume that
  - A set of **Operators**, that define how a valid state can be transformed into another



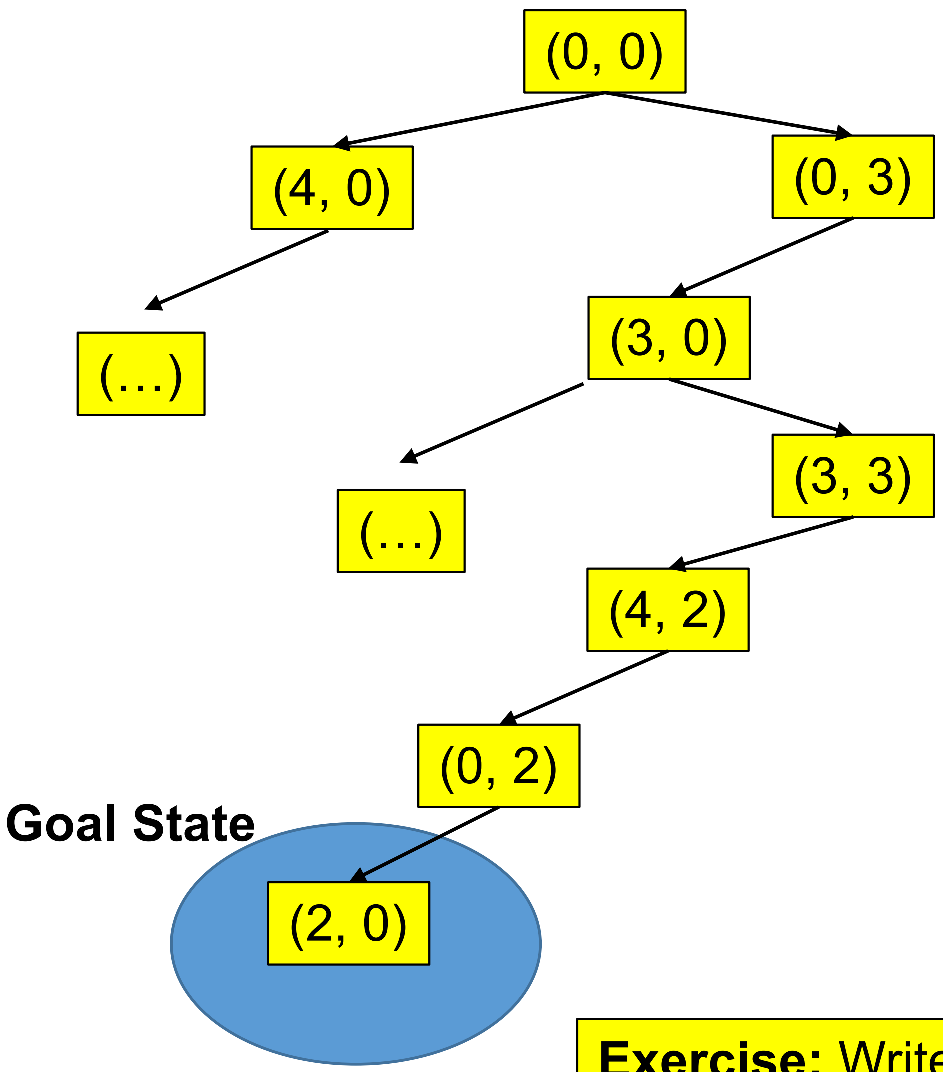
State Space Search

- Among the set of the possible states, there are two kinds that are particularly important:
  - **Start State:** The state from where the Search begins
  - **Goal State:** The state to be reached, i.e., that fully satisfies the goals of the problem.
- In this context, a **Solution** is the sequence of actions (a.k.a. **Plan**) that transforms the start state into the goal state.
- Example: **Water Jug Problem**
  - *Suppose you are given two jugs (a 4-liter and a 3-liter), with no markers. Also, there is a pump that can be used to fill a jug. Find the sequence of actions that enable to get exactly 2 liters of the 4-liter jug.*

State Space Search

- ☐ **Step 1:** Define the data structures required to represent a state.
  - ☐ In this problem, it can be as simple as (x, y), with "x" representing the liters inside the 4-liter jug, and "y" representing the liters in the 3-liter jug.
- ☐ **Step 2:** Define the allowed operations. Example:
  - ☐ (x, y) → (4, y) stands for "*Fill the first jug*"
  - ☐ (x, y) → (x, 3) stands for "*Fill the second jug*"
  - ☐ (x, y) → (0, y) stands for "*Empty the first jug on the ground*"
  - ☐ (x, y) → (x+d, y-d) stands for "*pour some water from the second to the first jug*"
  - ☐ …
- ☐ **Step 3:** Define the start state and expand it, until a solution is reached

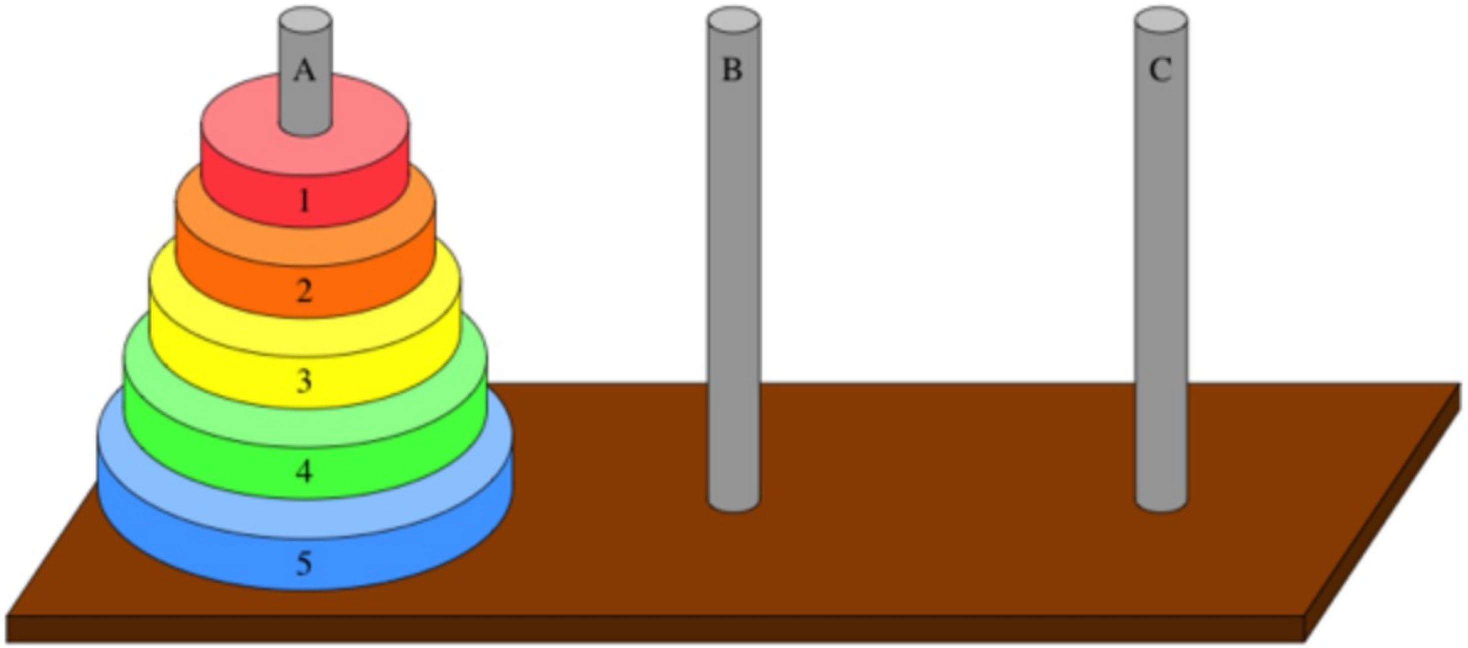
State Space Search



Exercise: Write a Python program that solves the Water Jug Problem

State Space Search – Hanoi Towers Exercise

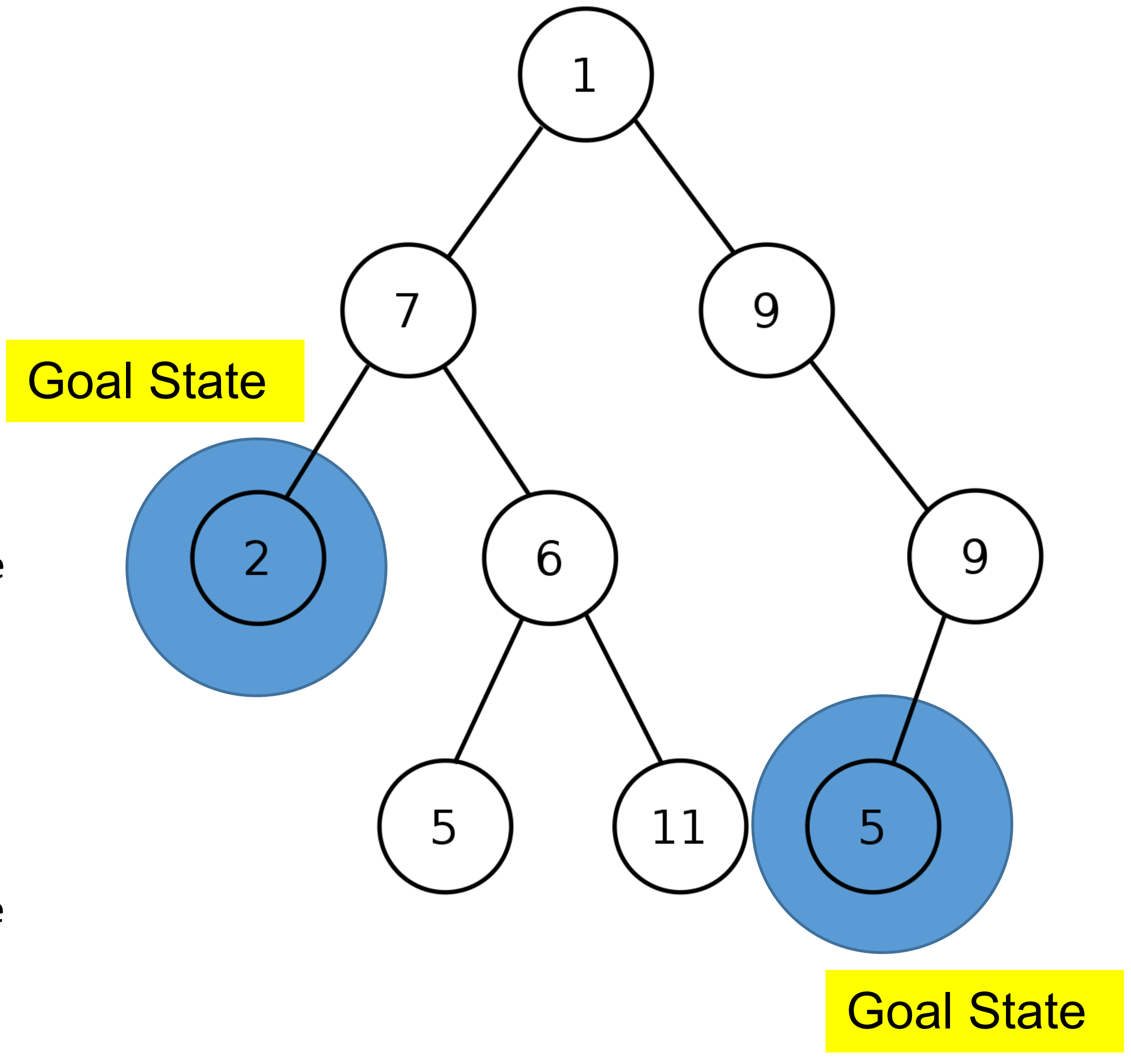
- ☐ Suppose you are given a set of three pegs and **n** disks, with each disk a different size. There are three pegs (A, B, and C), and let's identify the disks from 1 (the smallest disk), to n (the largest disk). Initially, disks are set from n → 1 in peg A. The goal is to move all disks to peg B, moving one disk at a time and without having a larger disk on top of a smaller one, at any time.
  - ☐ Formulate the problem as a State Space Search, and implement a Python script that solves it.



Source: https://www.khanacademy.org/computing/computer-science/algorithms/towers-of-hanoi/a/towers-of-hanoi

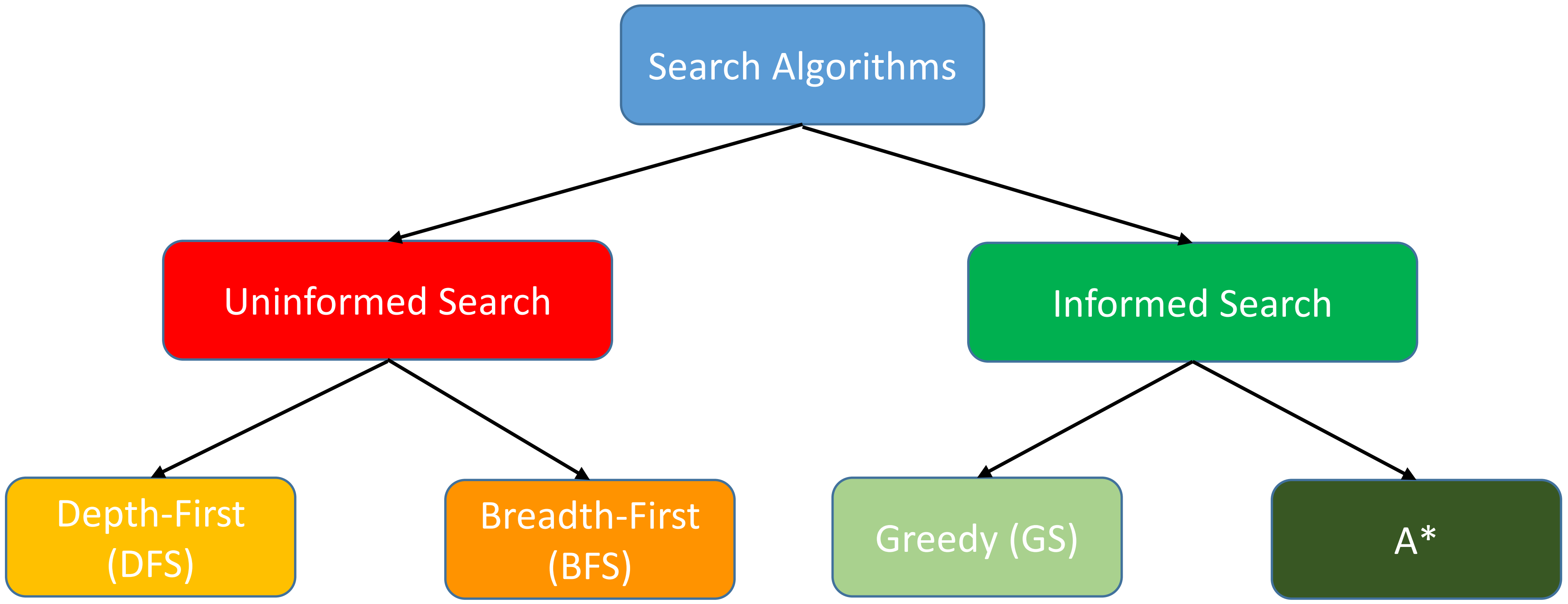
State Space Search

- In the previous examples/exercises, the goal was exclusively to find one solution (path) to the problem, without considering its **cost.**
  - Hence, the cost of a solution is the sum of all movements (i.e., operations) between the initial and the goal states.
  - If all operations have equal cost, the final cost is given by the depth in the tree where a solution is found.
- However, most of the times, a solution can be reached by many different ways, and according to a different number of steps.
  - The problem is: How can we find the optimal solution?



State Space Search

- Even though there are many variants and strategies for searching in States Spaces, they can be broadly divided into two main families, depending whether they use additional knowledge to speed-up (i.e., by ordering the priority of states) the search process.

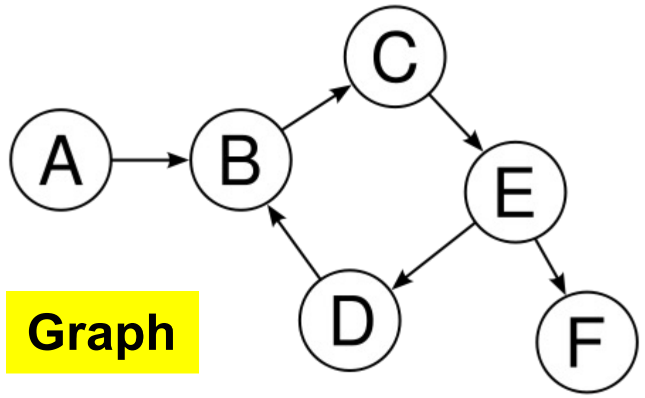


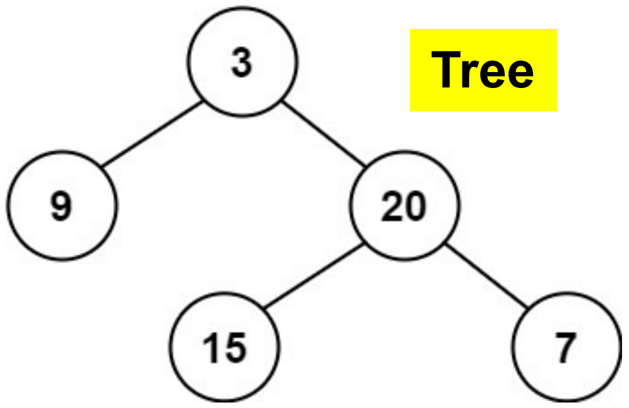
Uninformed Search

- ☐ The search algorithms in this family use no additional information on the goal node other than the one provided in the problem definition.
- ☐ The plans to reach the goal state from the start state differ only by the order and/or length of actions. Uninformed search is also called **blind search**, or **brute-force search**.
- ☐ These algorithms can only generate the successors and differentiate between the goal state and non goal state.
- ☐ There are two main strategies for performing this kind of search:
  - ☐ Depth First Search (**DFS**)
  - ☐ Breadth First Search (**BFS**)
- ☐ Keeping information about the states already explored (i.e., from where the successors were already obtained) and not, the main difference is that DFS works under the "Stack" paradigm (new nodes go directly to the first position of the "to be explored" list), while BFS assumes the "Queue" paradigm (i.e., new nodes go to the last position of the data structure).
  - ☐ Upon an obtained solution, BFS guarantees its optimality, while DFS does not.

Uninformed Search

- ☐ If the states in the solution space can be visited more than one time, a directed graph is typically used to represent the State Space. Otherwise, a n-ary tree is used.
- ☐ In a graph, more than one path can be used to move between two particular nodes, whereas in a tree, every path is unique.
- ☐ Search algorithms for graphs must pay particular attention to loops, which will be an obstacle to find a proper solution.
  - ☐ This is typically done, by keeping independent lists for the "**expanded**" and "**to_expand**" states.
  - ☐ When generating the successors of a state, they will only be inserted at the "**to_expand**" list if they are not in none of the data structures





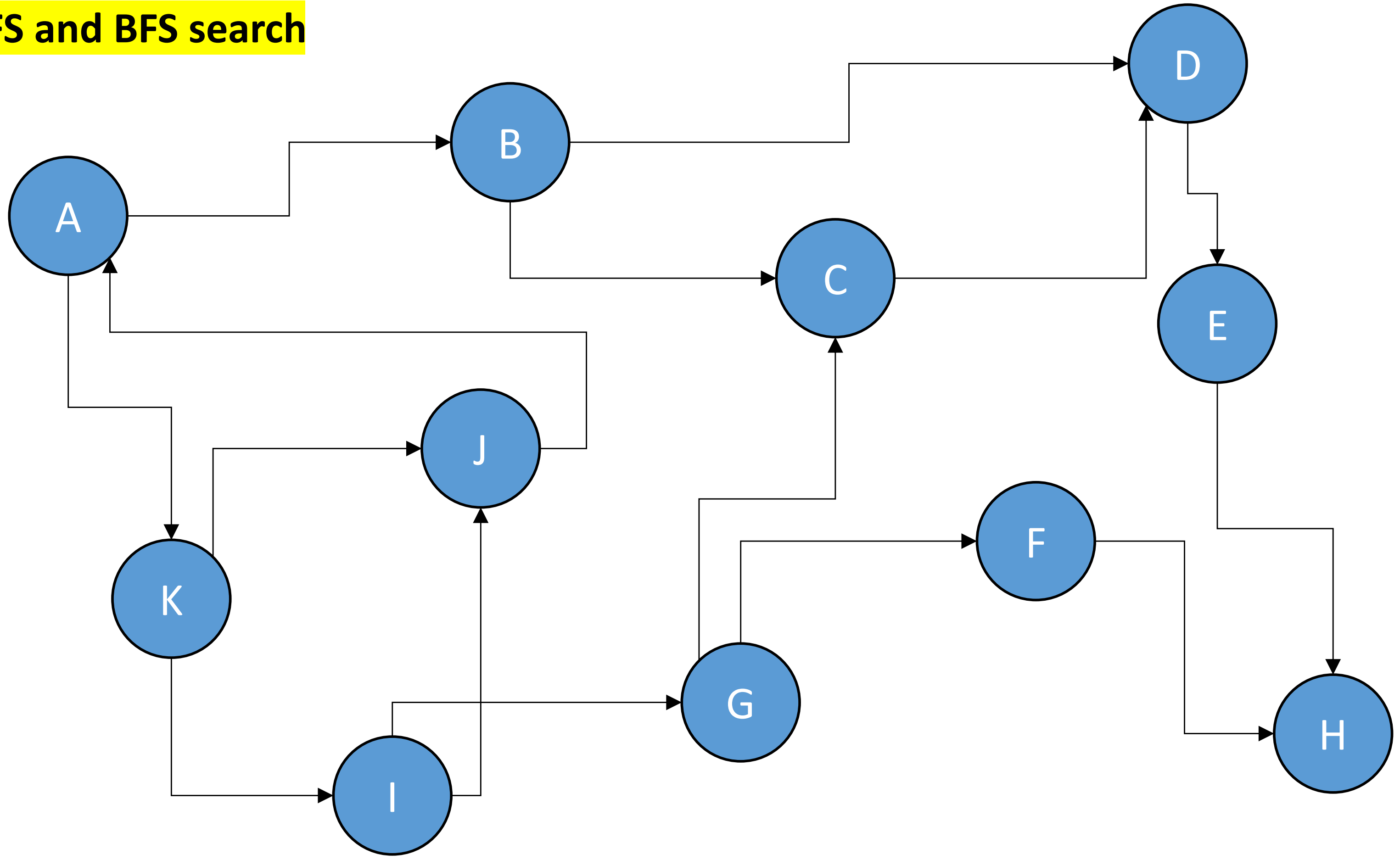
Uninformed Search – DFS and BFS

- Depth-first search and Breadth-first search (DFS and BFS) are algorithms for traversing or searching a tree or graph data structures.
  - **DFS** starts at the initial state node and explores as far as possible along each branch before backtracking. It uses last in- first-out strategy and hence it is implemented using **a Stack**.
  - **BFS** starts at the initial node and only evaluates nodes in a level when all nodes at the previous levels were already explored. Uses first in- first-out and is implemented using **a Queue.**
- Algorithm:

```
1. cur_state = initial_state
2. to_explore = []
3. explored = []
4. push(to_explore, cur_state)
5. WHILE to_explore NOT EMPTY
   1. cur = pop(to_explore)
   2. if cur == goal_state
          Return cur
   3. suc = find_sucessors(cur)
   4. for s in suc:
      1. if s NOT IN explored AND s NOT IN to_explore
            push(to_explore, s)
   5. push(explored, cur)
6. return None
```

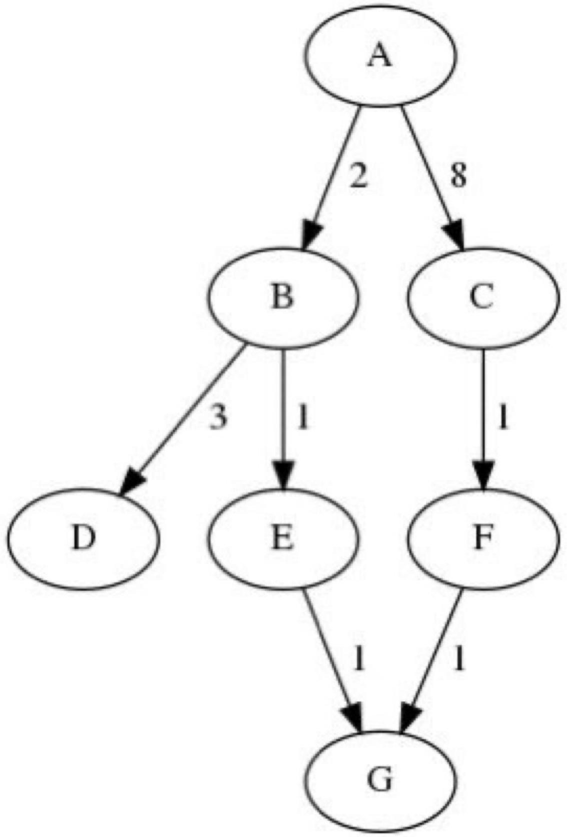
Uninformed Search – DFS and BFS

- **Exercise:** Consider the following Graph. Assuming that all transitions between states have equal cost, find the order by which states will be visited, in order to move from ”A” → “H”, using…
  - DFS and BFS search



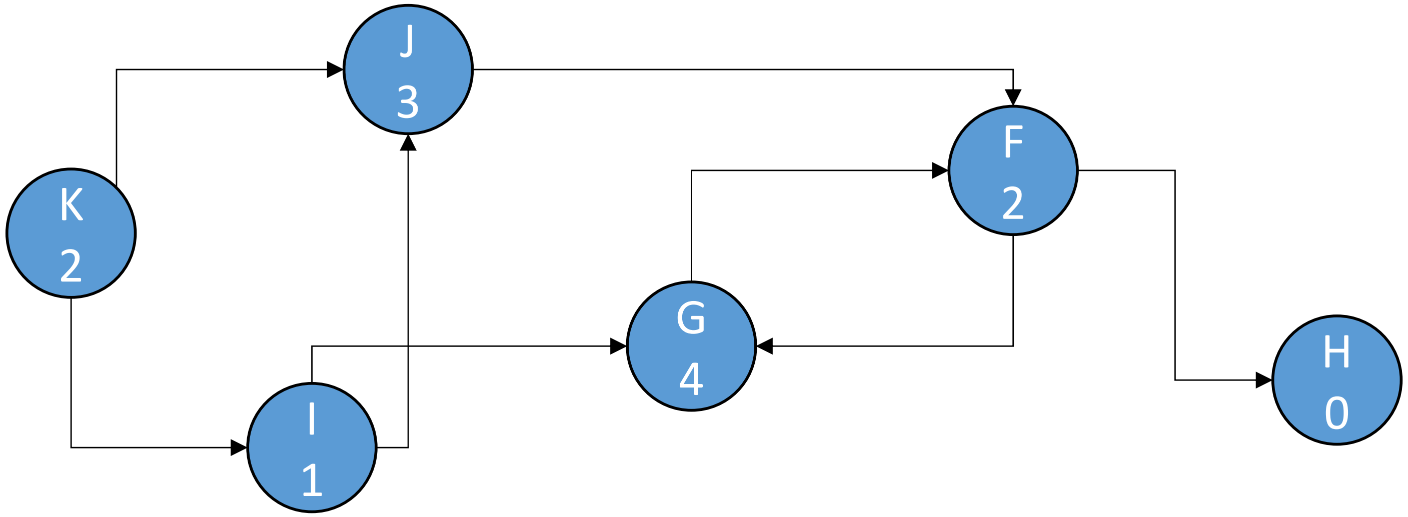
Uninformed Search – Uniform Search (US)

- In this variant, the costs from moving between states come into play. Assuming that we have information about the cost for reaching a state (typically, the sum of the costs from all transitions since the initial state), each state will be inserted at a specific position of the "to_explore" data structure, depending of its cost, with the cheaper states being at the initial positions of the list.
- In practice, it is equivalent to BFS, provided all transitions have the same cost.
- **Exercise:** Find the order by which the nodes will be visited in the tree, using US search.



Informed Search – Greedy Search

- In this family of algorithms, we have information about the goal state, and – using a **heuristic h()**– we are able to estimate how far we are from the goal state.
  - For example – Manhattan distance, Euclidean distance, etc. (here, the lesser the distance, the closer the goal). In other cases, similarity-based metrics can be used in the exact same way (i.e., higher values, higher similarity).
- The basic strategy is called **Greedy Search (GS)** and we always expand (explore) the state that is closer to the goal.
- **Exercise:** Considering the h() values below each state, find the order by which states will explored between K→H, using the GS algorithm.



Informed Search – A*

- This variant combines the strengths of uniform-cost search and greedy search. In this search, the heuristic is the summation of the cost to reach the current state, denoted by *g(x)*, and the value of the heuristic used in **GS**, denoted by *h(x)*. The summed cost is denoted by *f(x)*.
- Hence, the nodes will be visited (explored) according to:

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

- ***h(x)*** is called the **forward cost** and is an estimate of the distance of the current state from the goal state.
- ***g(x)*** is called the **backward cost** and is the cumulative cost of a state from the root state.
- Importantly, A* search is optimal when for all nodes, the forward cost for a node h(x) underestimates the actual cost h*(x) to reach the goal. This property of A* heuristic is called **admissibility**.

$$0 \leq h(x) \leq h^* (x)$$

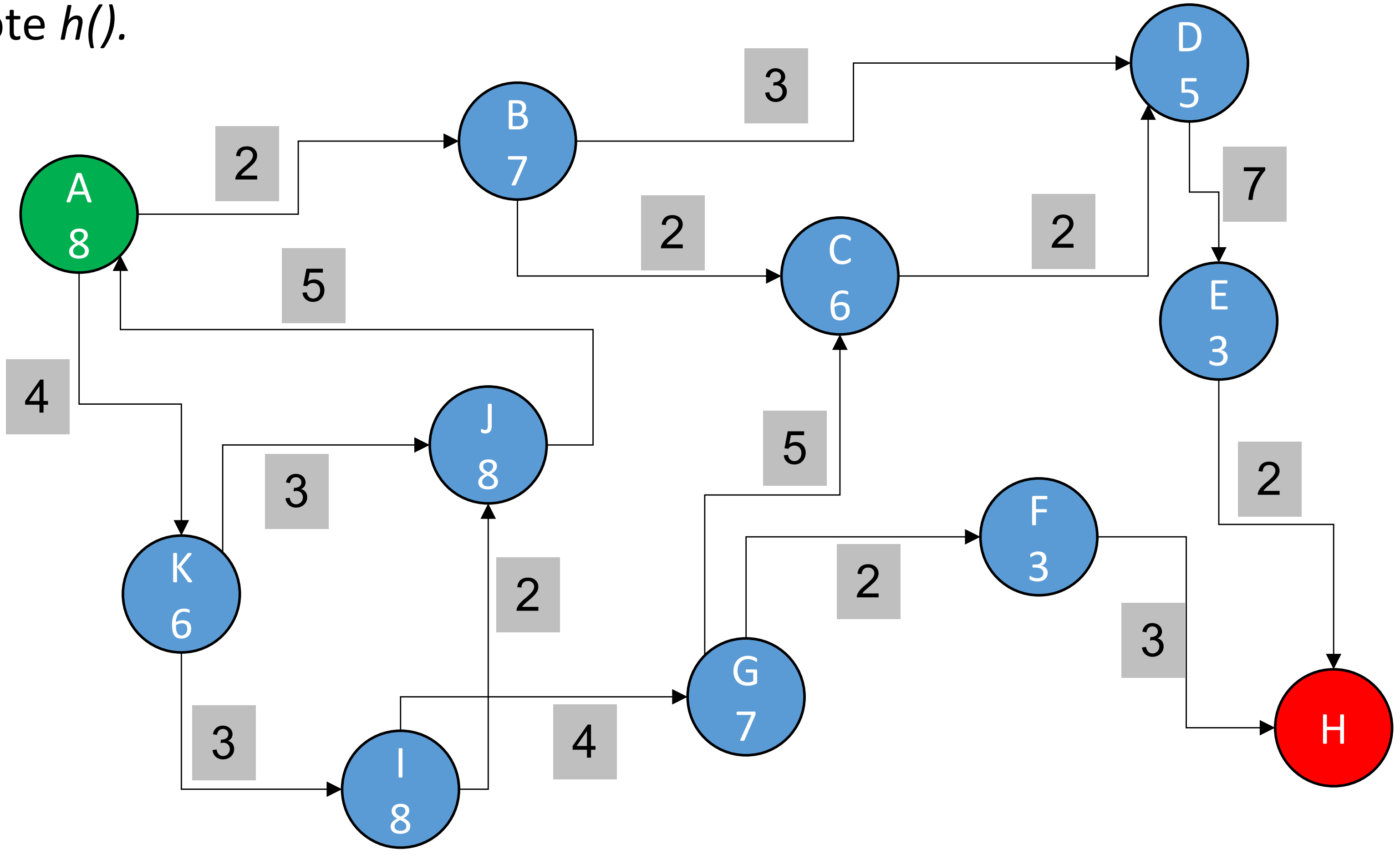
Informed Search – A*

- In terms of pseudo-code, with respect to the uniformed search algorithms, the difference is that the insertion in that the “to_explore” set is sorted according to the values of *f()*, in ascending order.
- Algorithm:

```
1.  cur_state = initial_state
2.  to_explore = []
3.  explored = []
4.  push(to_explore, cur_state)
5.  WHILE to_explore NOT EMPTY
    1.  cur = pop(to_explore)
    2.  if cur == goal_state
            Return cur
    3.  suc = find_sucessors(cur)
    4.  for s in suc:
        1.  if s NOT IN explored AND s NOT IN to_explore
                to_explore = insert(to_explore, s, f(s))
    5.  push(explored, cur)
6.  return None
```

Informed Search – A*

- Exercise: Find the order by which states will be visited (from nodes A→H), according to A*, assuming that the values near the edges denote the corresponding transition costs and the values inside each node denote *h()*.



A
8
B
7
C
6
D
5
E
3
F
3
G
7
H
I
8
J
8
K
6
2
3
2
2
7
5
4
3
5
2
2
2
3
3
4