Introduction to Artificial Intelligence in Games

Lecture 2

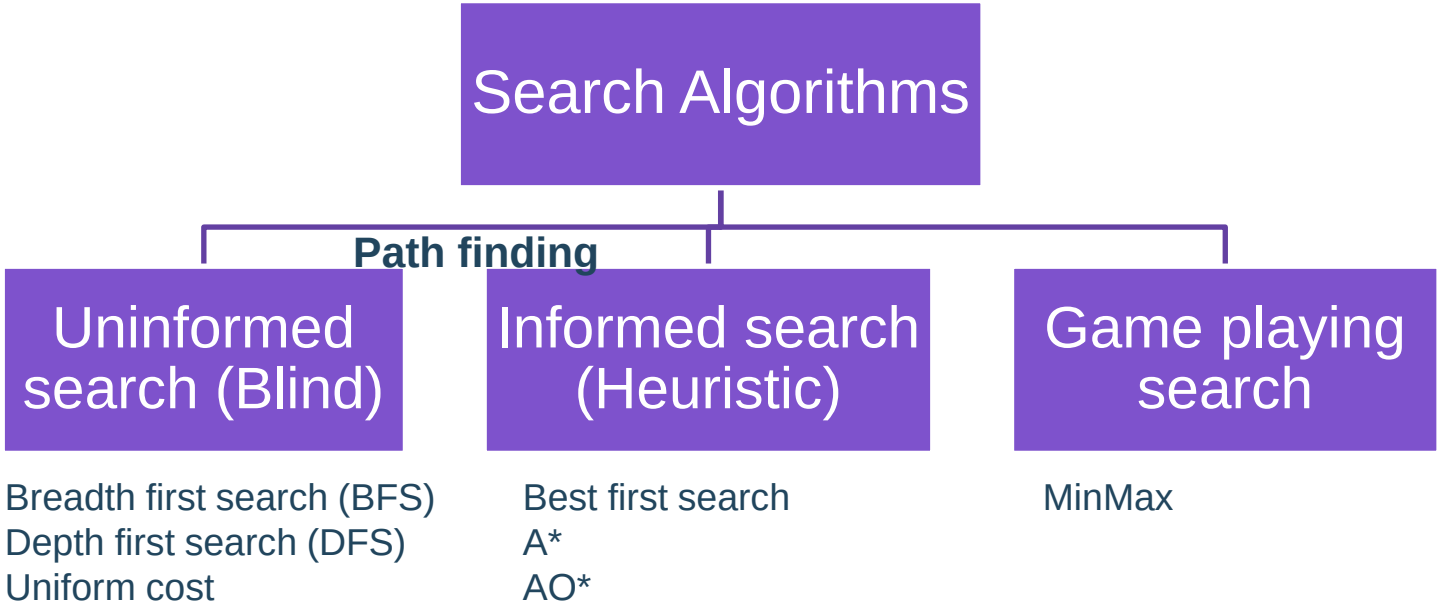
AI Search Algorithms

- AI search algorithm is a sequence of steps, which are initially unknown, to find a solution, which satisfies a number of conditions, between possible solutions.

AI Search Algorithms

Search algorithms are used to:

- Find a path from the start point to the end point (path finding)
- Find a solution for a problem or a game (game playing)



Search Problem

To solve a search problem, we need to define:

- Initial state: the first location (state) we start with.
- Goal state: the final location (state) we target.
- States (Search space): all possible states of the problem (game) to reach the solution (goal state).
- Actions: possible transitions from one state to another.
- Search tree: visual tree representation of search space, showing possible solutions from initial state.

Example: 8-puzzle

- Initial state

1	3	4
2	5	7
6	8	

- Goal state(s)

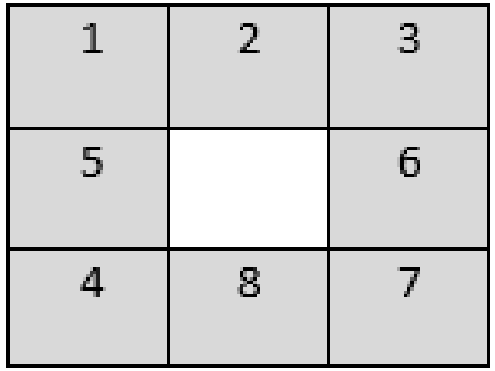
1	2	3
4	5	6
7	8	

1	2	3
8		4
7	6	5

	1	2
5	4	3
8	7	6

Example: 8-puzzle

- Actions: it depends on the current state



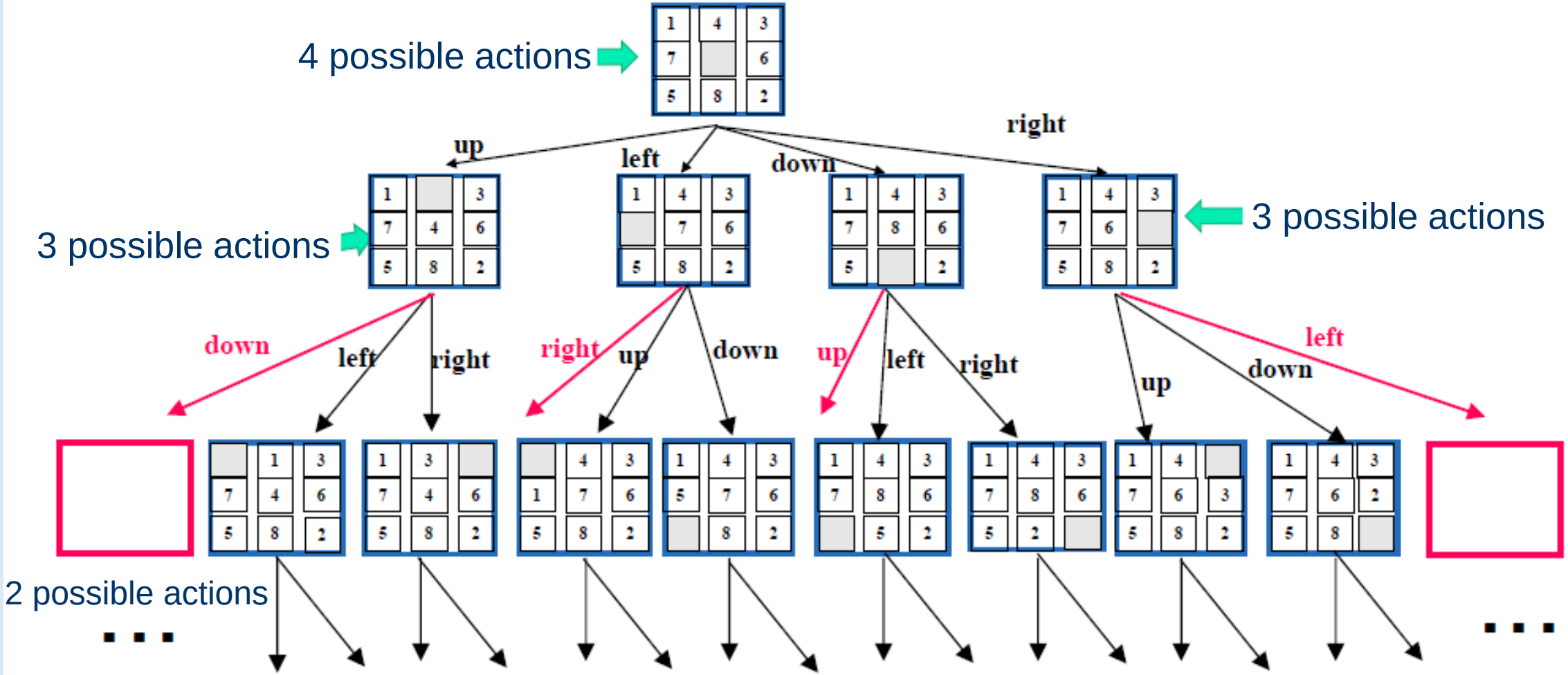
4 possible actions

3 possible actions

2 possible actions

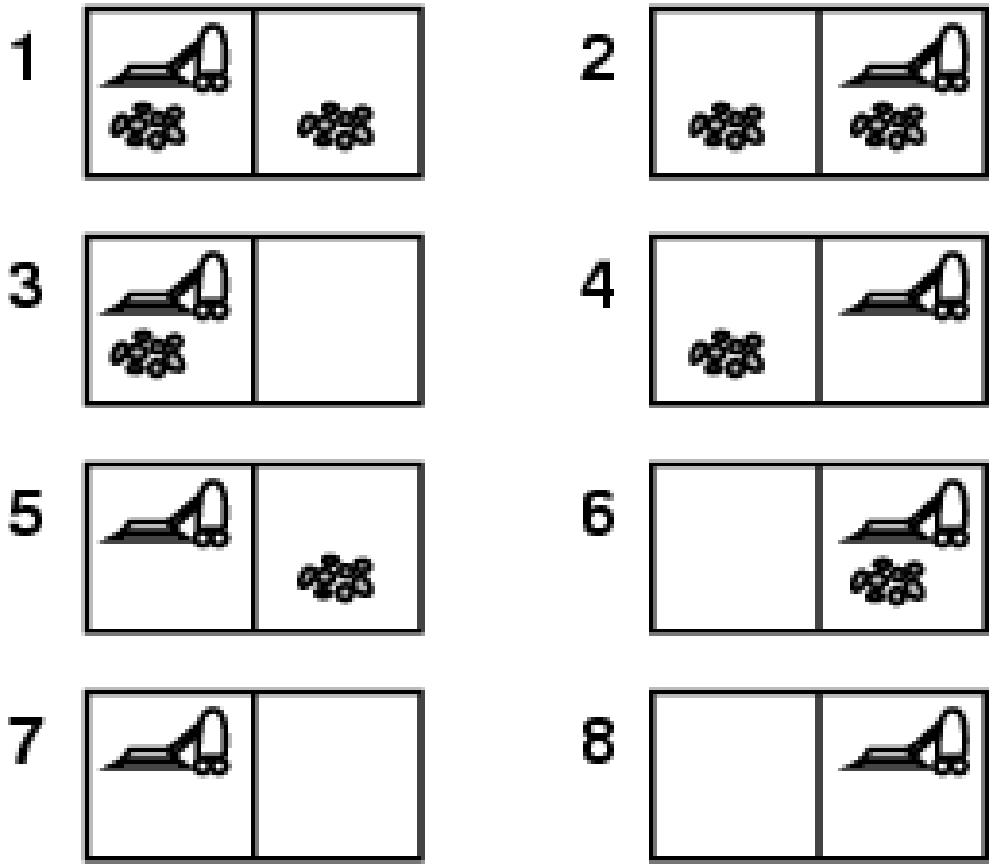
Example: 8-puzzle

- States: all different states of the game to reach the goal state



Example: Cleaning Robot

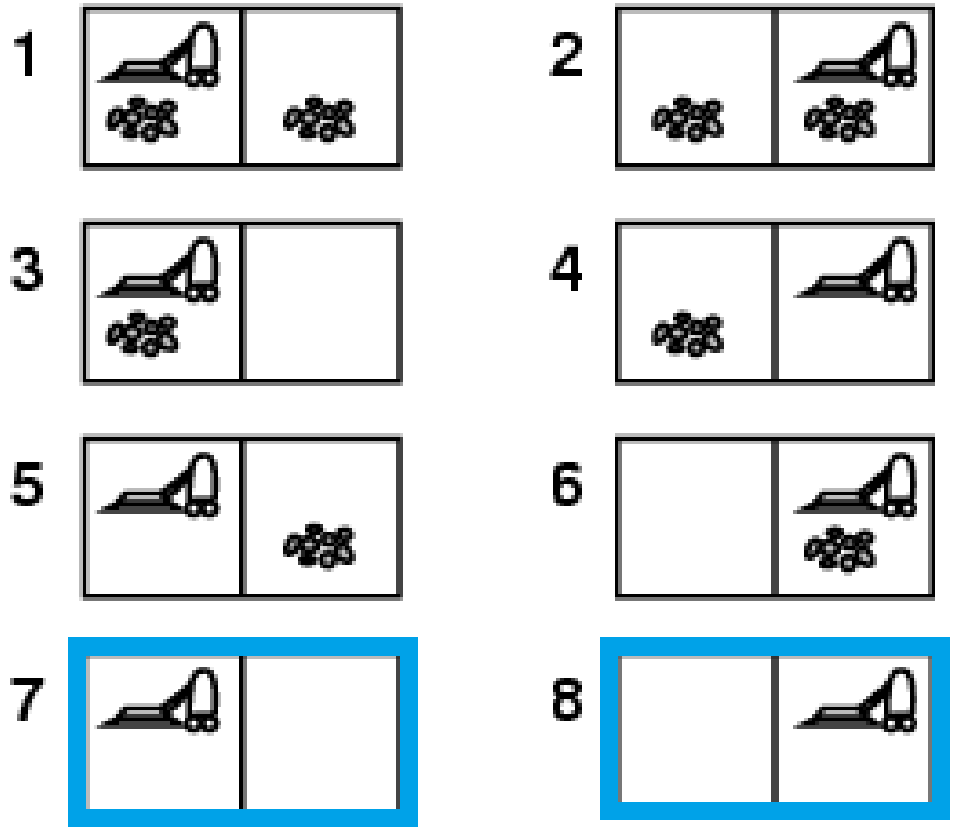
- There are two rooms: right and left.
- Room can be dusty or clean.
- The robot can be in the right or the left room.
- Three possible actions: Go right, Go left, or Suck.
- 8 possible states.



Example: Cleaning Robot

Problem analyzing

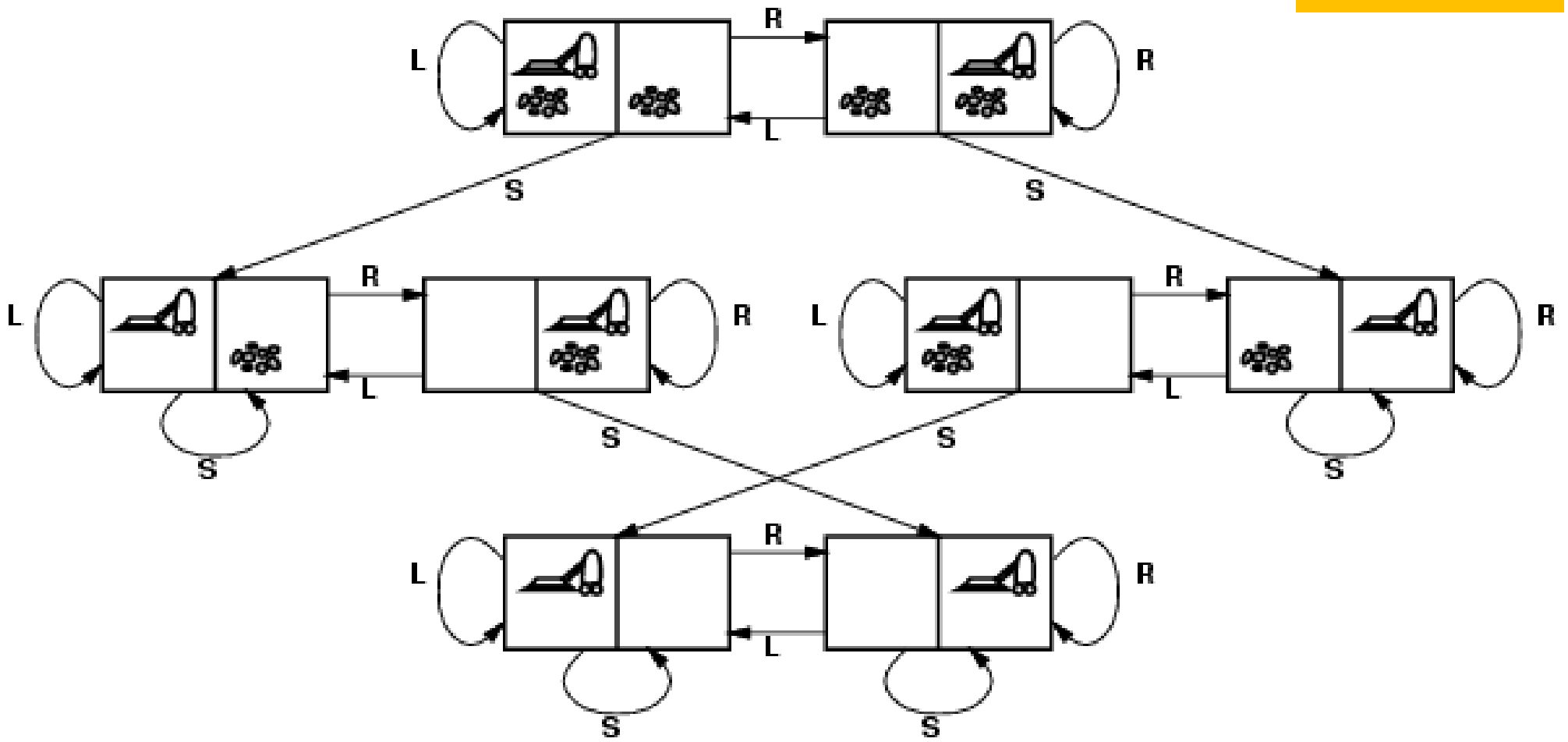
- Initial state: one from the 8 states.
- Actions: Right (R), Left (L), Suck (S)
- Goal state: the room is clean (the robot is at state 7 or 8).



Example: Cleaning Robot

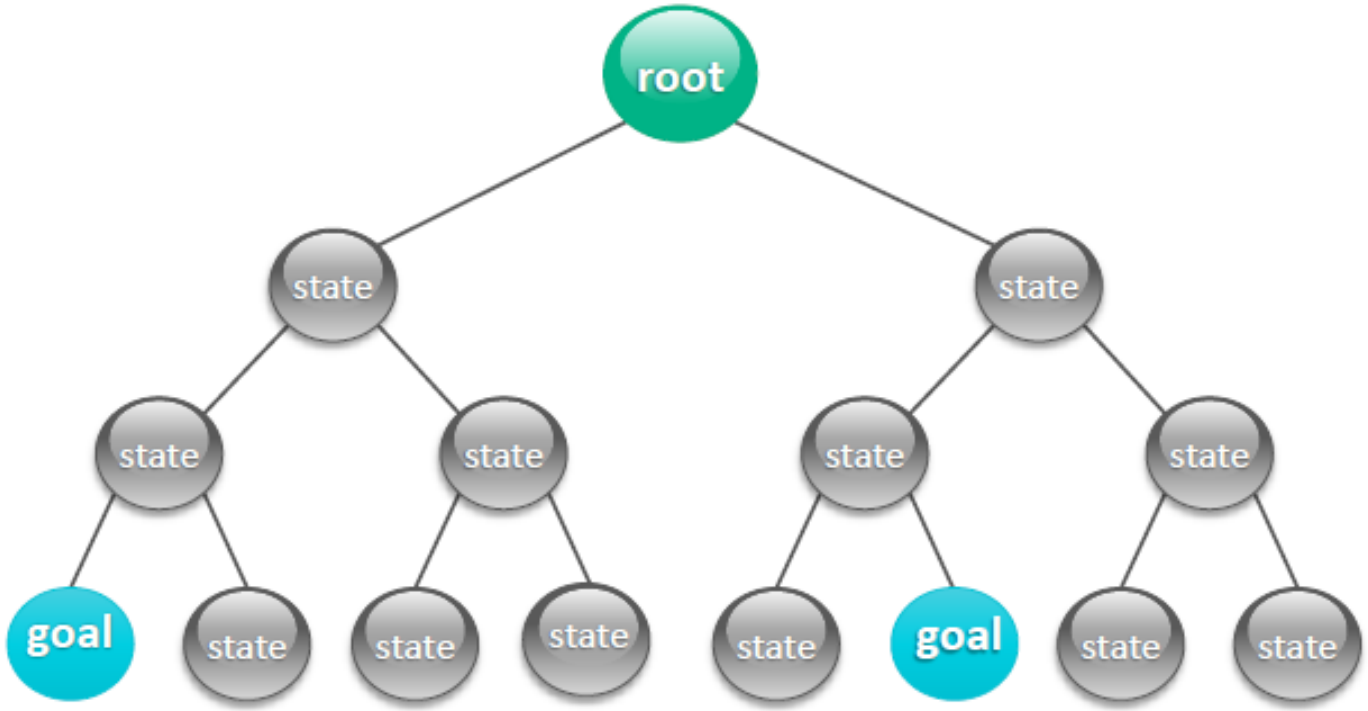
Possible states and actions

Search Tree



Search Tree

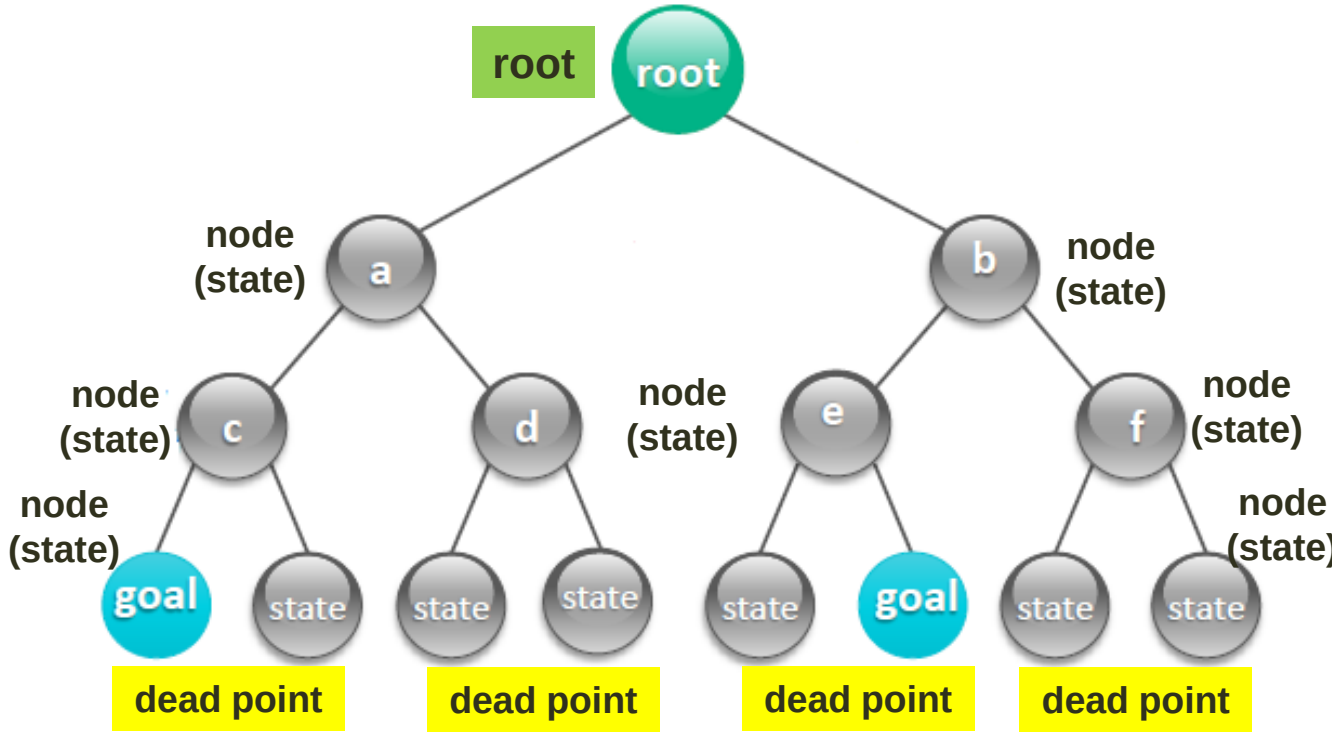
- Search tree is a tree representation of different states and actions (transitions), showing possible solutions from initial state.
- It is constructed during the search process.
- Search tree makes the search process easier for finding possible solutions using the search algorithms.
- However, some complicated problems can’t be described using search tree.



Search Tree

Terms

- **Nodes**: points in hierarchal levels, each point is a state.
- **Search space:** all nodes in the search tree.
- **Root**: the root of the tree, which is the initial state we start from.
- **Parent**: the node that has branching nodes towards the below level. The branching nodes are called **child**.
- **Dead point (leaves)**: the node that has no childs.

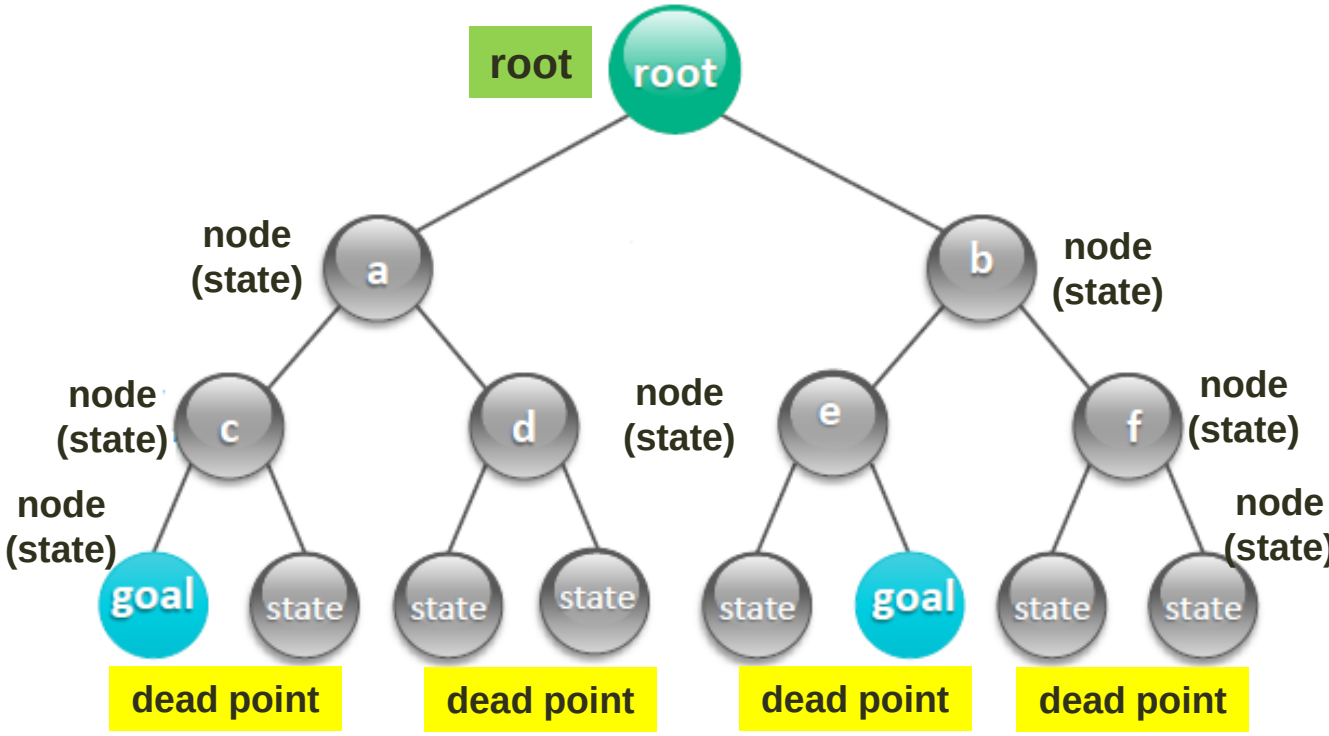


root is a parent for a and b. a is a parent for c and d, and so on.

Search Tree

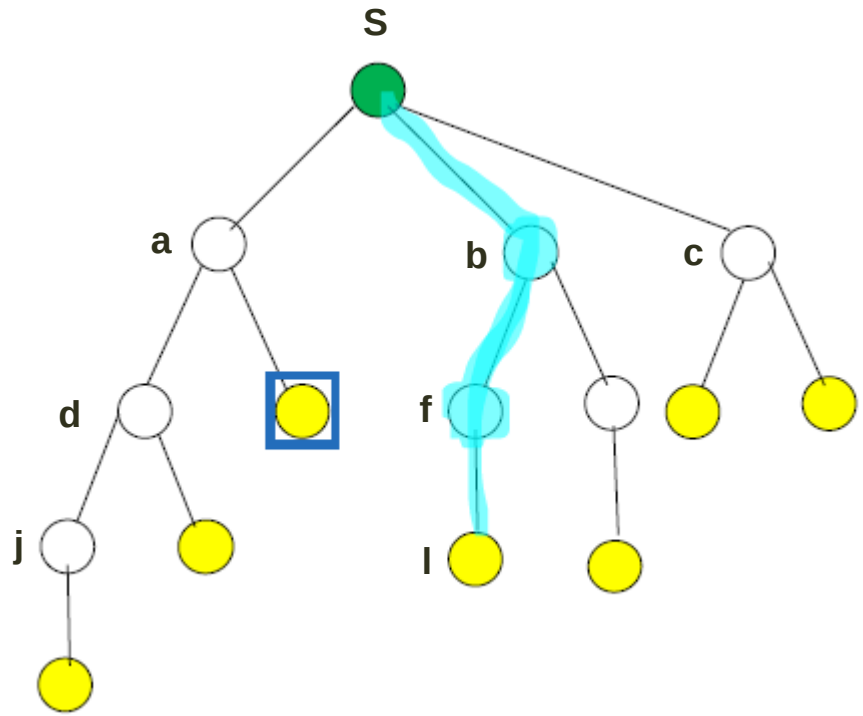
Terms (Cont.)

- **Node expansion**: generating new node related to previous nodes.
- **Path cost:** the cost, from initial to goal state. The cost can be related to speed (more speed means low cost), distance (long distance means high cost), time (more time means high cost), and so on.
- **Depth**: number of steps along the path from initial state.

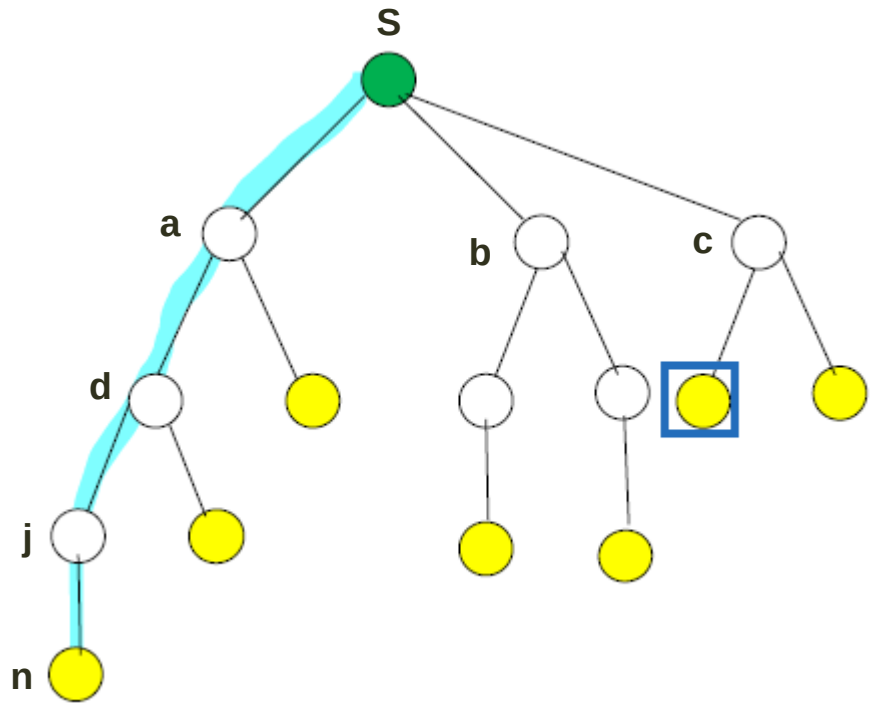


Tree Path

Tree path is a collection of nodes from the root to any leaf of the tree.



Path: S → b → f → l



Path: S → a → d → j → n

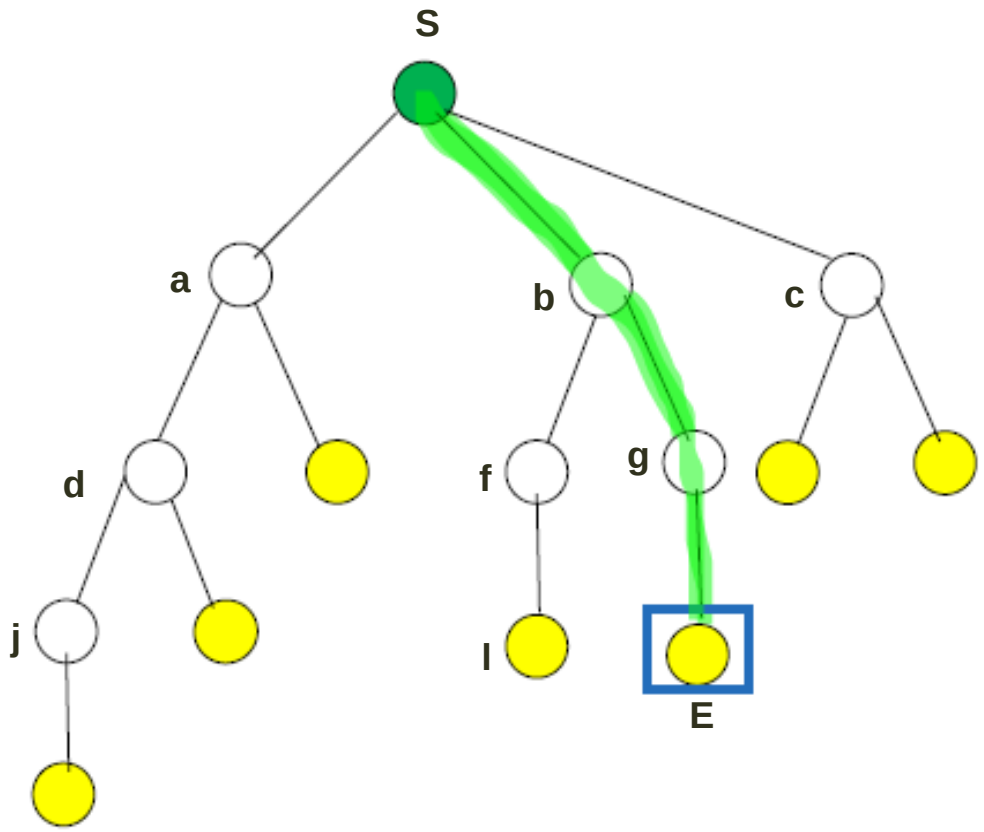
Root (initial state)

Dead point (leaf)

Goal state

Solution Path

It is a collection of nodes from the root to the goal state.



Solution path: S → b → g → E

Root (initial state)

Dead point (leaf)

Goal state

Measuring Searching Performance

- The output from a searching algorithm is either FAILURE or SOLUTION.
- Searching algorithms are evaluated along the following dimensions:
  - Completeness: does it always find a solution if one exists?
  - Optimality: does it always find the shortest path solution? (optimal solution is the solution with minimal cost)
  - Time complexity: how long?
  - Space complexity: how much memory? (maximum number of nodes in memory)
- Time and space complexity are measured in terms of:
  - b: maximum branching factor of the search tree
  - d: depth of the shortest path solution
  - m: maximum depth of the state space (may be ∞)

Generic Search Algorithm

- Fringe = list of nodes generated but not expanded (open nodes).

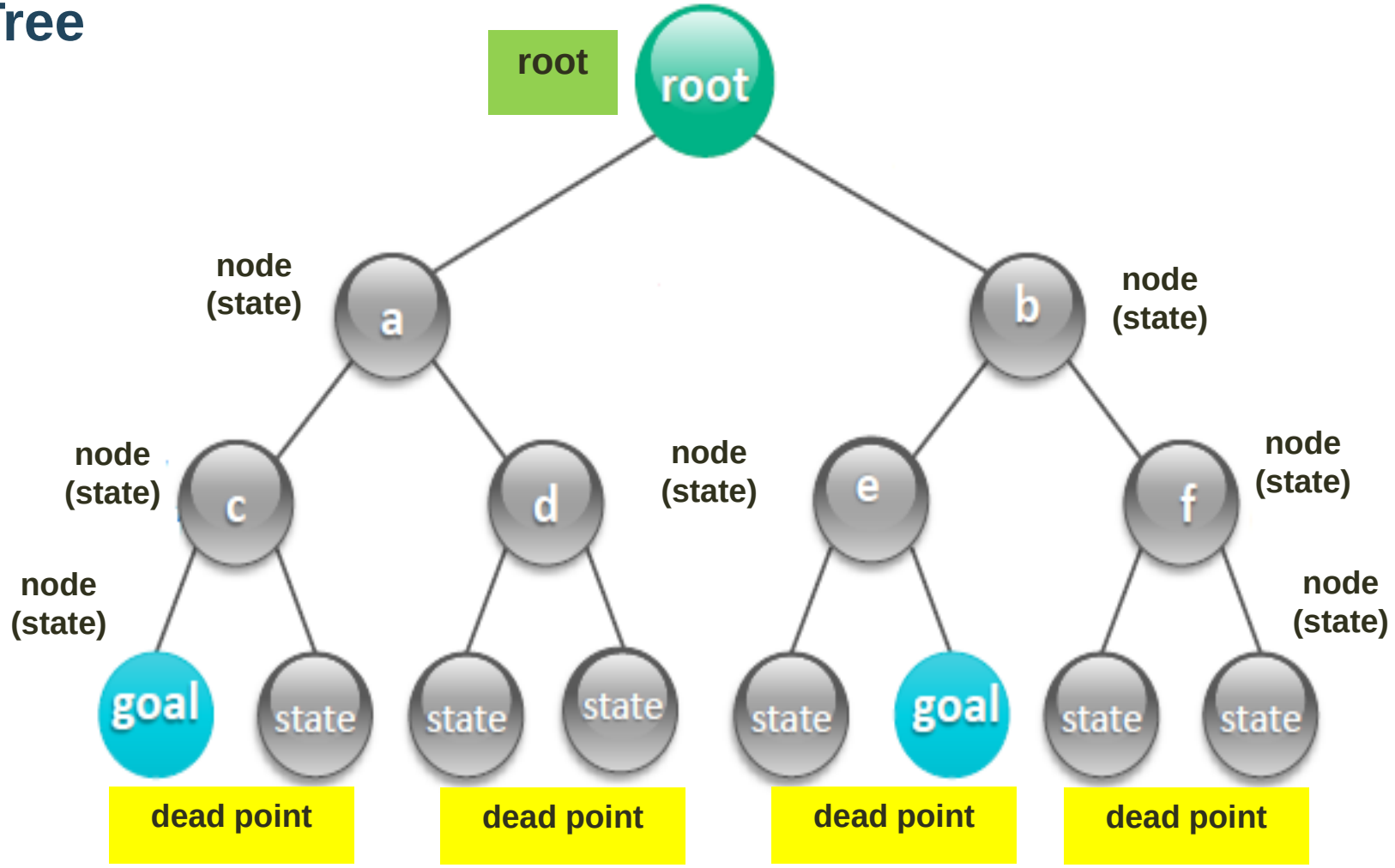
Steps:

- fringe := {open nodes based on initial state}
- loop:
  - if fringe empty, declare failure and exit
  - choose and remove a node v from fringe
  - check if v's state is a goal state; if so, declare success
  - if not, expand v, insert resulting nodes into fringe

Key question in search: Which of the generated nodes do we expand next?

Searching Techniques

Search Tree



Searching (Path finding) Techniques

Uninformed Search

- **Blind search** → traversing the search space until the goal nodes is found (might be doing exhaustive search).
- *Techniques* : **Breadth First Uniform Cost ,Depth first, Iterative Deepening search**.
- Guarantees solution.

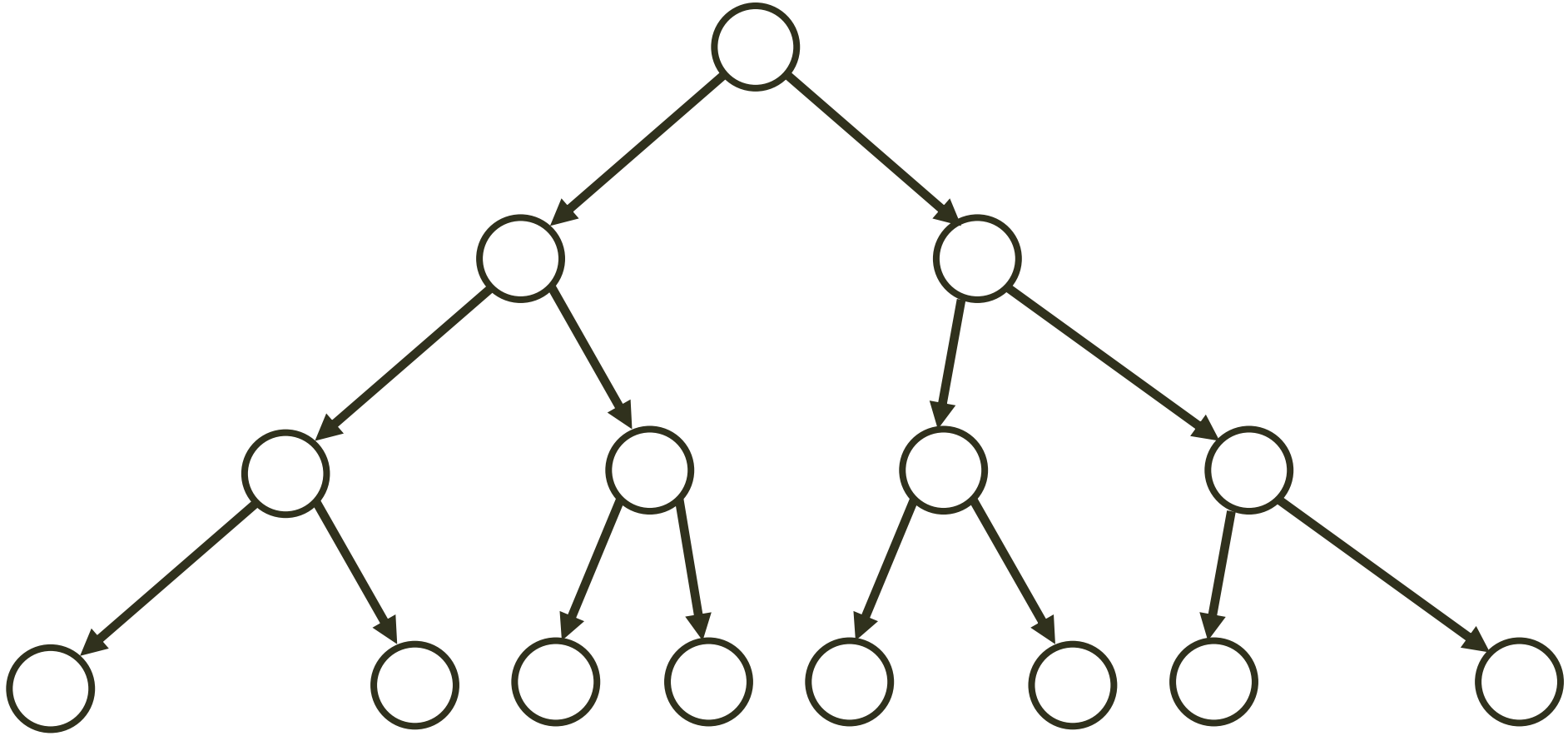
Informed Search

- **Heuristic search** → search process takes place by traversing search space with applied rules (information).
- *Techniques*: **Greedy Best First Search, A* Algorithm**
- There is no guarantee that solution is found.

Uninformed search

- Given a state, we only know whether it is a goal state or not
- Cannot say one nongoal state looks better than another nongoal state
- Can only traverse state space blindly in hope of somehow hitting a goal state at some point
  - Also called blind search
  - Blind does **not** imply unsystematic!

Breadth First Search

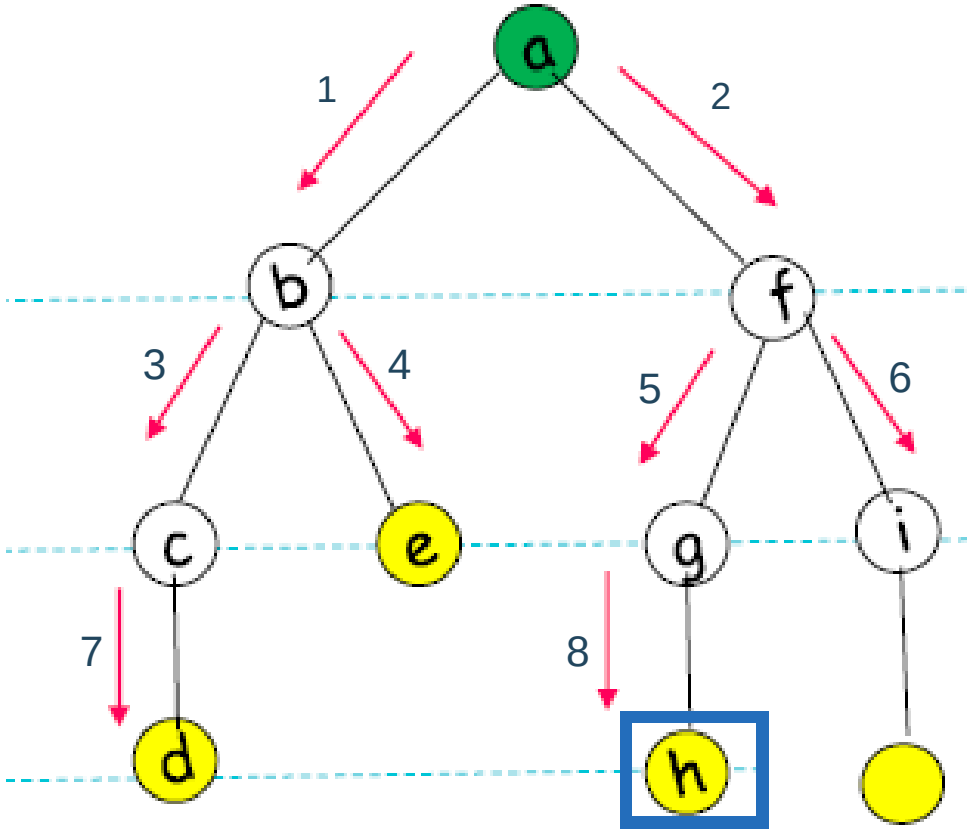


Breadth First Search (BFS)

- Nodes are expanded in the same order in which they are generated
  - Fringe can be maintained as a First-In-First-Out (FIFO) queue
- BFS is complete: if a solution exists, BFS will find it
- BFS finds a solution that guaranteed to be the shortest path possible
- Is BFS optimal?
  - Optimal means that the algorithm finds the **best** solution
  - The **best** here can mean the solution is either the shortest, fastest, cheapest, …
  - BFS is optimal if (1) the path cost is concerned with the shortest path, (2) the actions of a single node have the same cost, and (3) costs increase by depth.
  - Therefore, BFS is not necessarily an optimal solution

Breadth First Search (BFS)

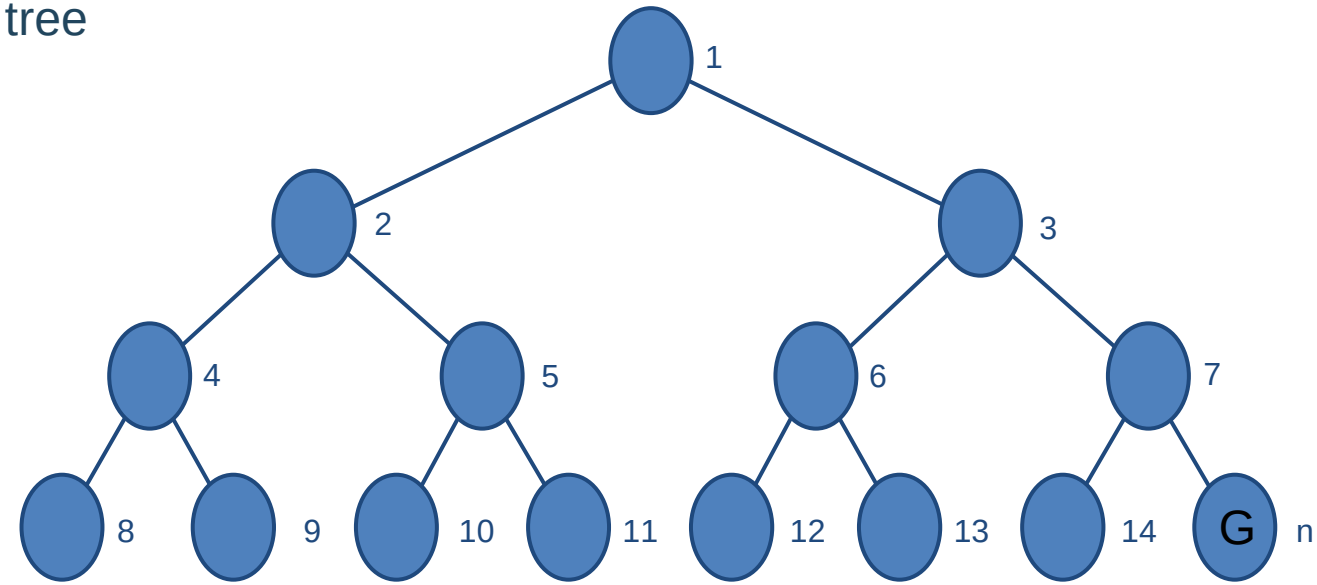
Example



Solution: a → f → g → h

Breadth First Search (BFS)

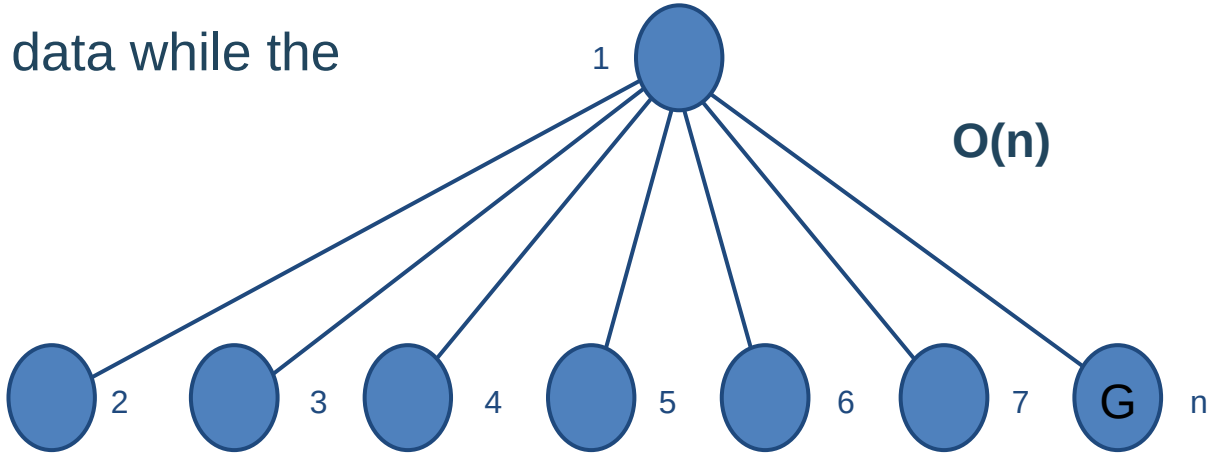
- Time Complexity
  - It is the amount of time taken by an algorithm to run, as a function of the input length.
  - It depends on how many nodes in the tree
  - assume (worst case) that there is 1 goal leaf at the RHS
  - So BFS will expand all nodes
    = **O(n)**
  - where n is the number of nodes in the tree



Breadth First Search (BFS)

Algorithms require memory to hold temporary data while the program is in execution.

- Space Complexity
  - It is the amount of space or memory required for getting a solution
  - It depends on how many nodes can be in the queue (worst-case)

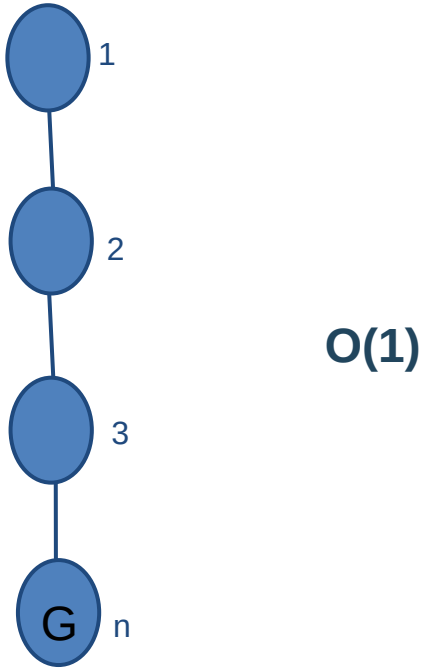


Worst case:

- Worst case would be storing (n - 1) nodes where all but the root node are located at the first level.
- So BFS will store (n-1) nodes,
  so the space complexity is O(n)

Best case:

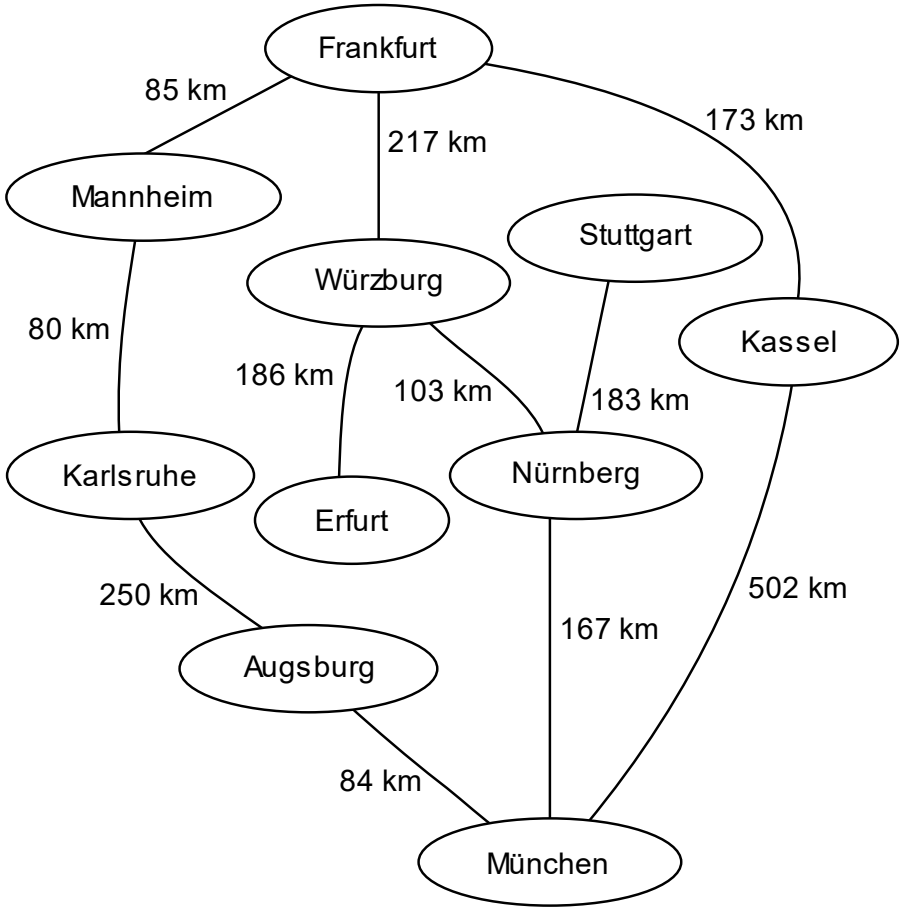
- Best case is when the tree contains only 1 element at each level.
- So BFS will store 1 node
  so the space complexity is O(1), which means it is constant regardless of any other parameter.

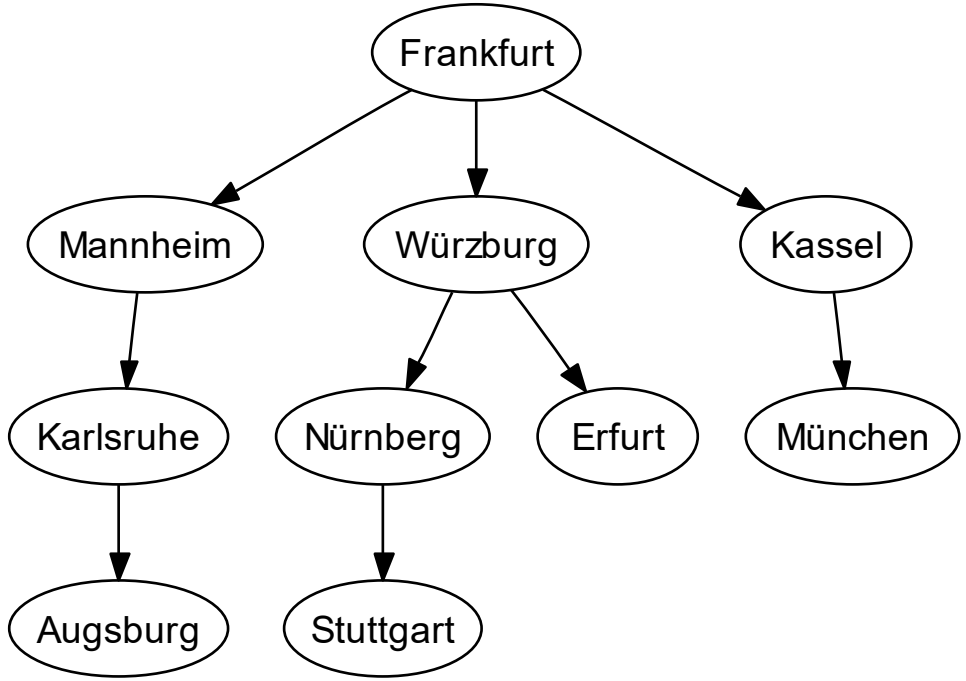


Breadth First Search (BFS)

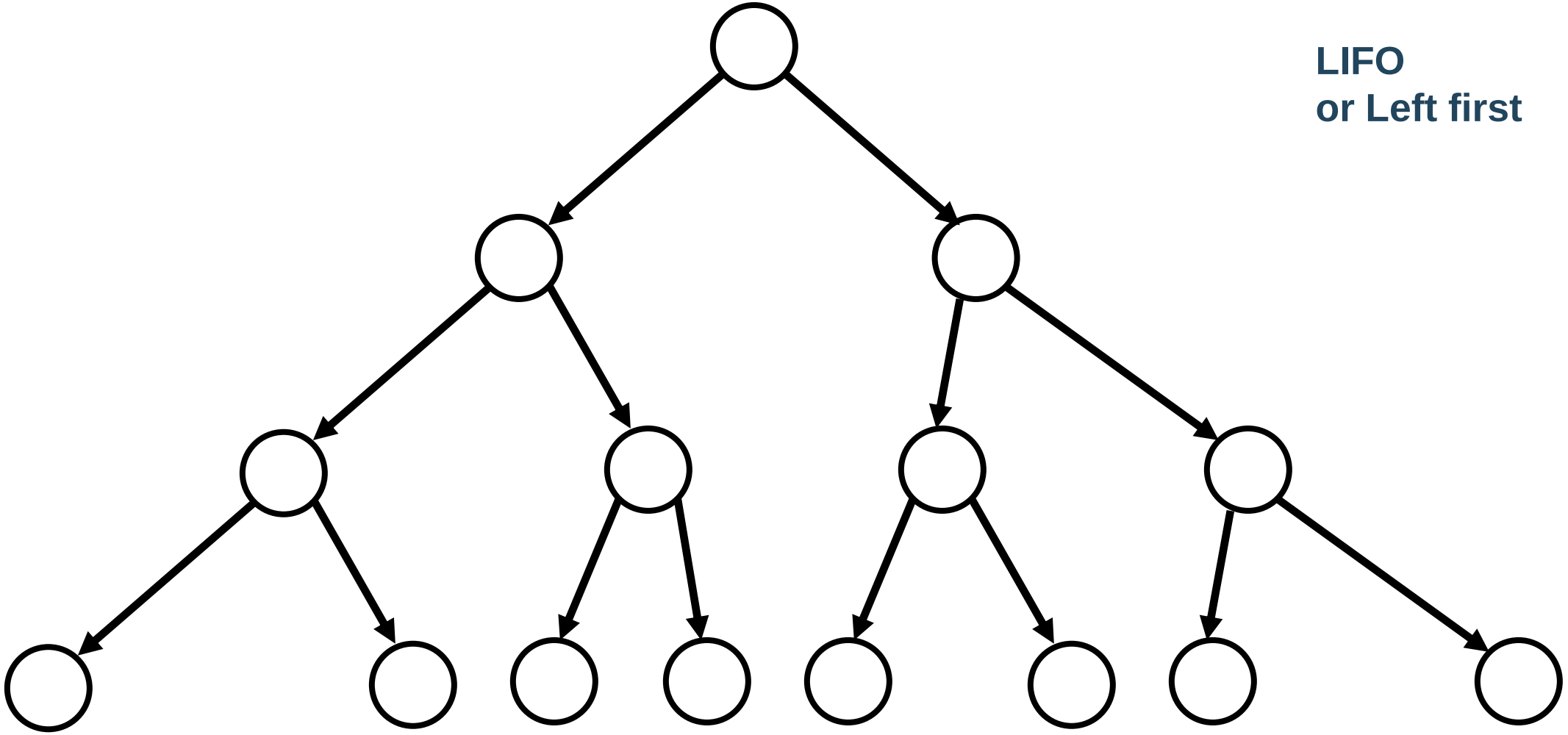
Example

The following is an example of the breadth-first tree obtained by running a BFS on German cities starting from Frankfurt:





Depth First Search (DFS)



LIFO
or Left first

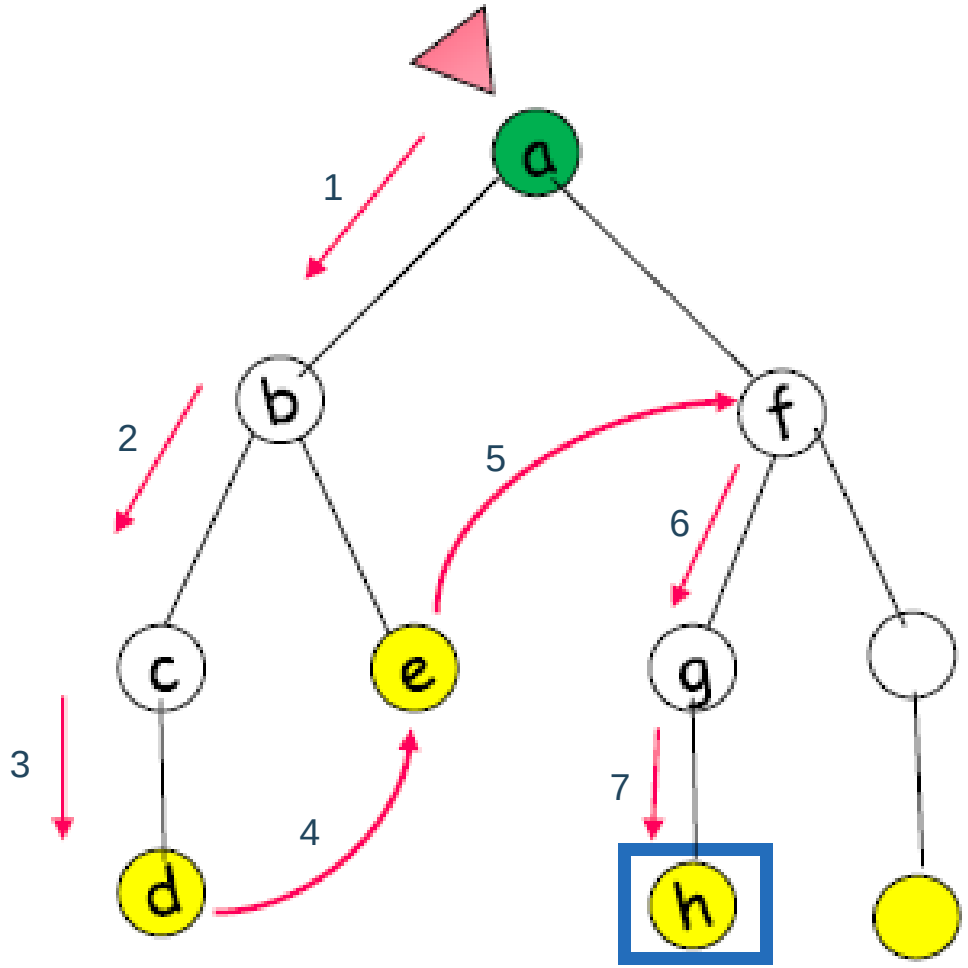
Depth First Search (DFS)

- Depth-First Search (DFS) begins at the root node and search each branch to it maximum depth till a solution is found
- If greatest depth is reached with not solution, we backtrack till we find an unexplored branch to follow

- Completeness:
  - If cycles (loops) are presented in the graph, DFS will follow these cycles indefinitively.
  - If there are no cycles, the algorithm is complete.
  - Cycles effects can be limited by imposing a maximal depth of search.
- Optimality:
  - The first solution is found and not the shortest path to a solution.
  - If we take the closeness as preference, DFS will not optimal.
- The algorithm can be implemented with a Last In First Out (LIFO) stack

Depth First Search (DFS)

Example

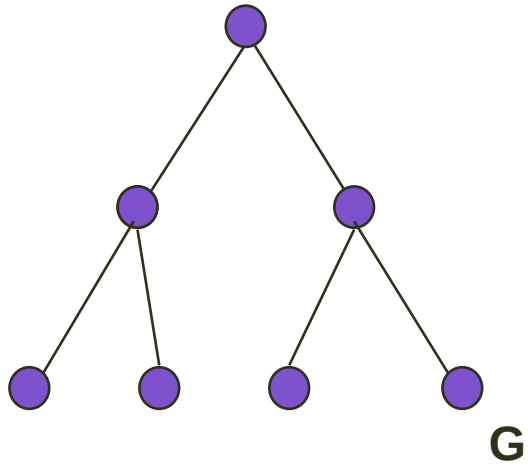


Solution: a → f → g → h

Depth First Search (DFS)

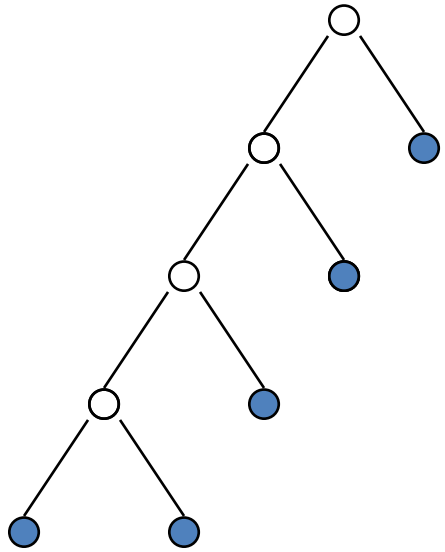
- Time Complexity
  - assume (worst case) that there is 1 goal leaf at the RHS
  - so DFS will expand all nodes
  - Time complexity is the same magnitude as BFS

  O(n)
  - where n is the number of nodes in the tree



Depth First Search (DFS)

- Space Complexity
  - It is the amount of space or memory required for getting a solution
  - It depends on how many nodes can be in the queue (worst-case)



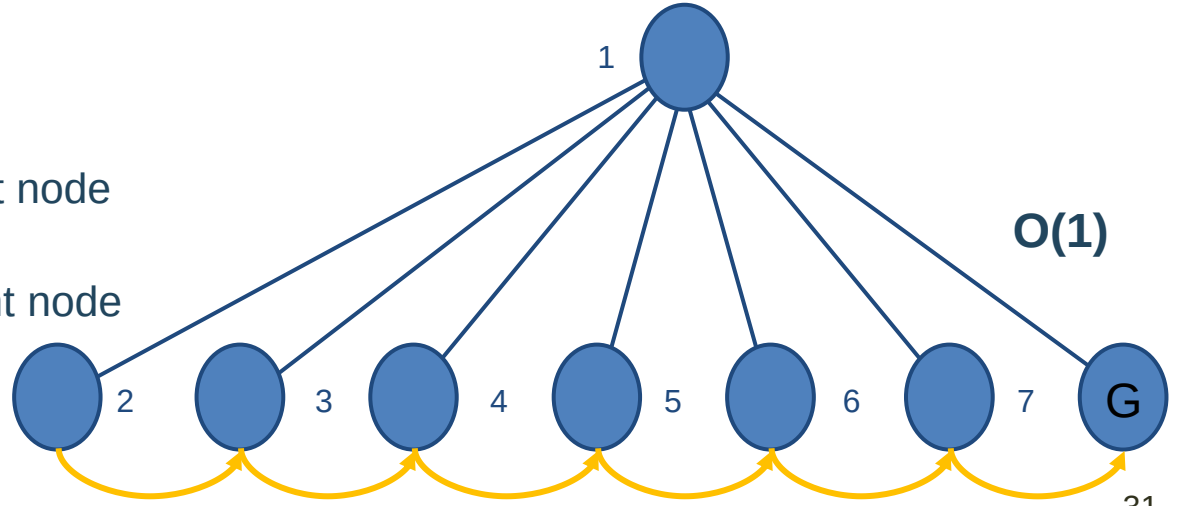
O(d)

Worst case:

- the stack will contain d open nodes, where d is the maximum depth of the tree
- so the space complexity is O(d)

Best case:

- Best case is when the tree contains all but the root node located at the first level.
- So DFS will store the root node besides the current node **so, the space complexity is O(1)**
- The best case for DFS is the worst case of BFS

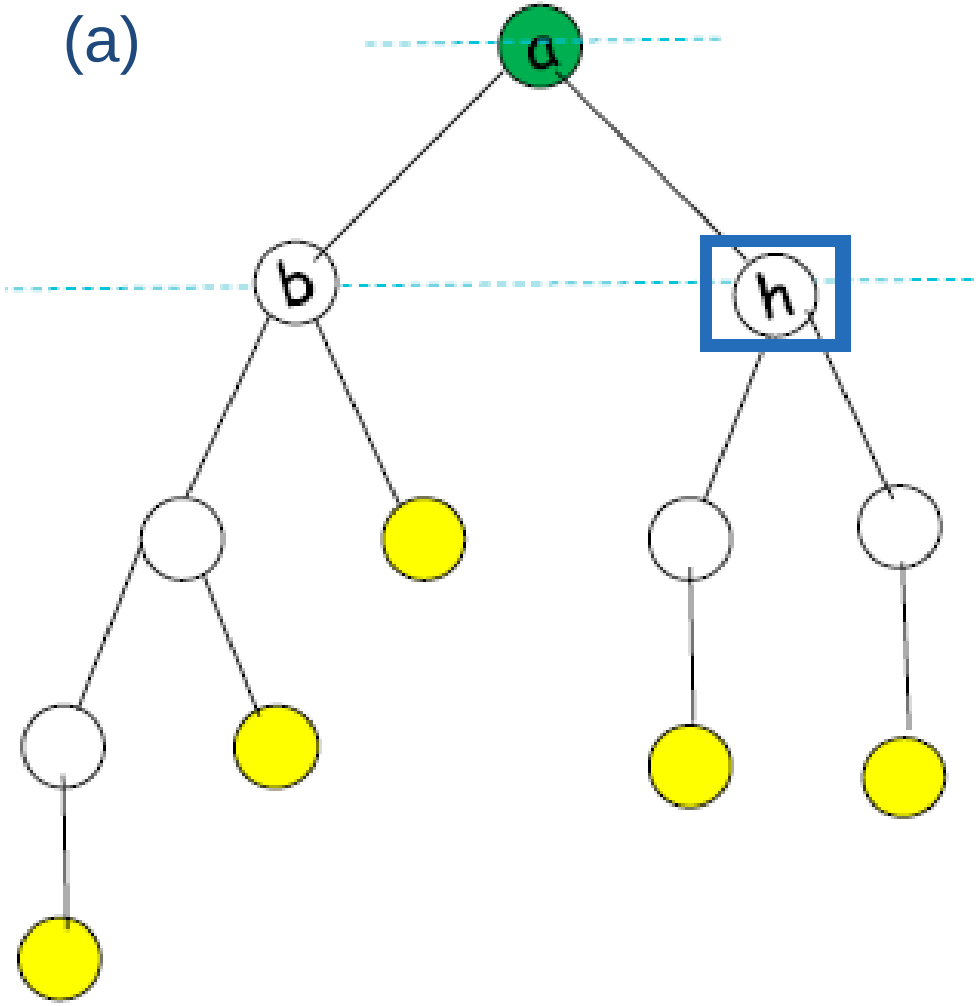


O(1)

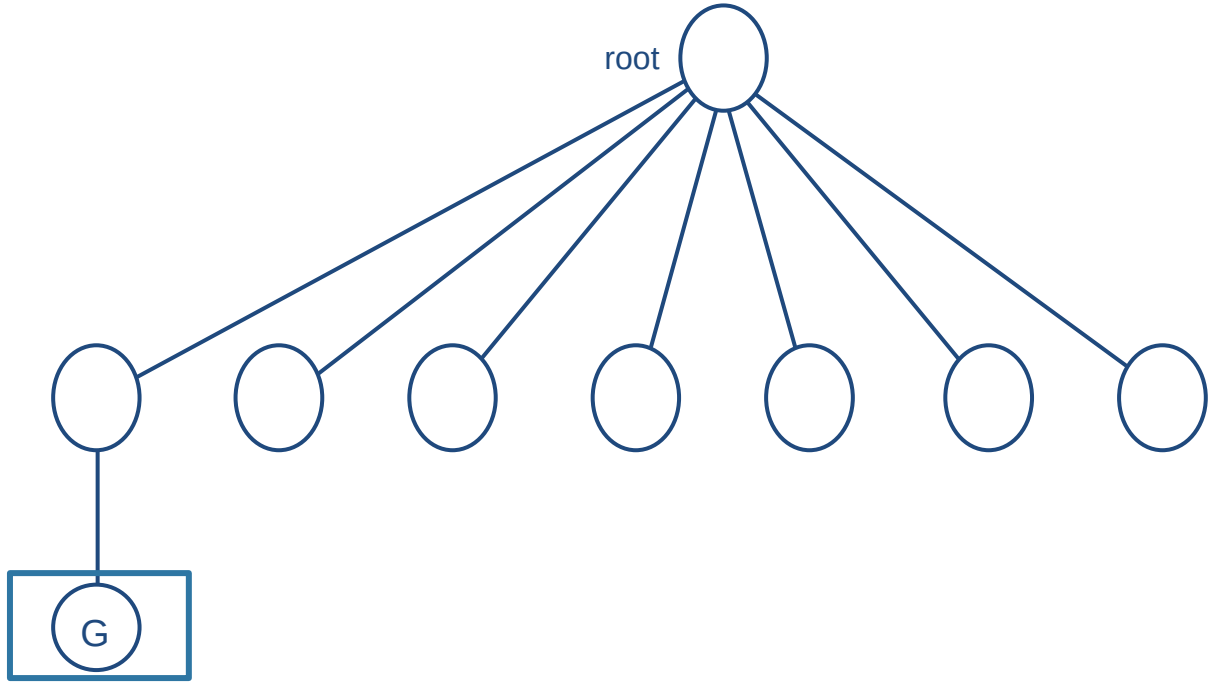
BFS or DFS

Select the best algorithm to find the solution for the following:

(a)



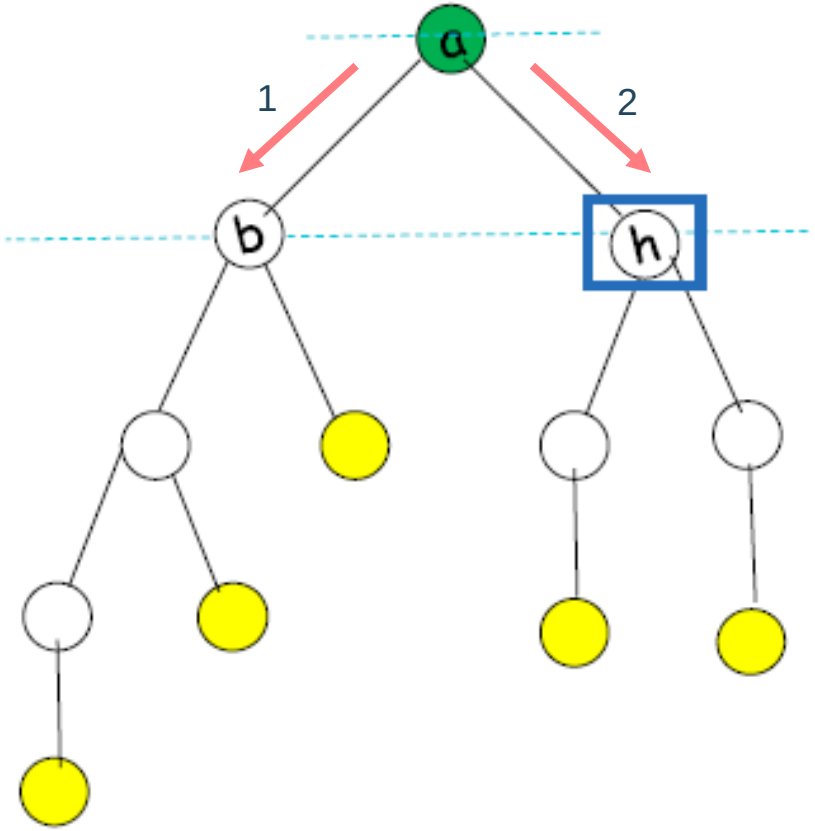
(b)

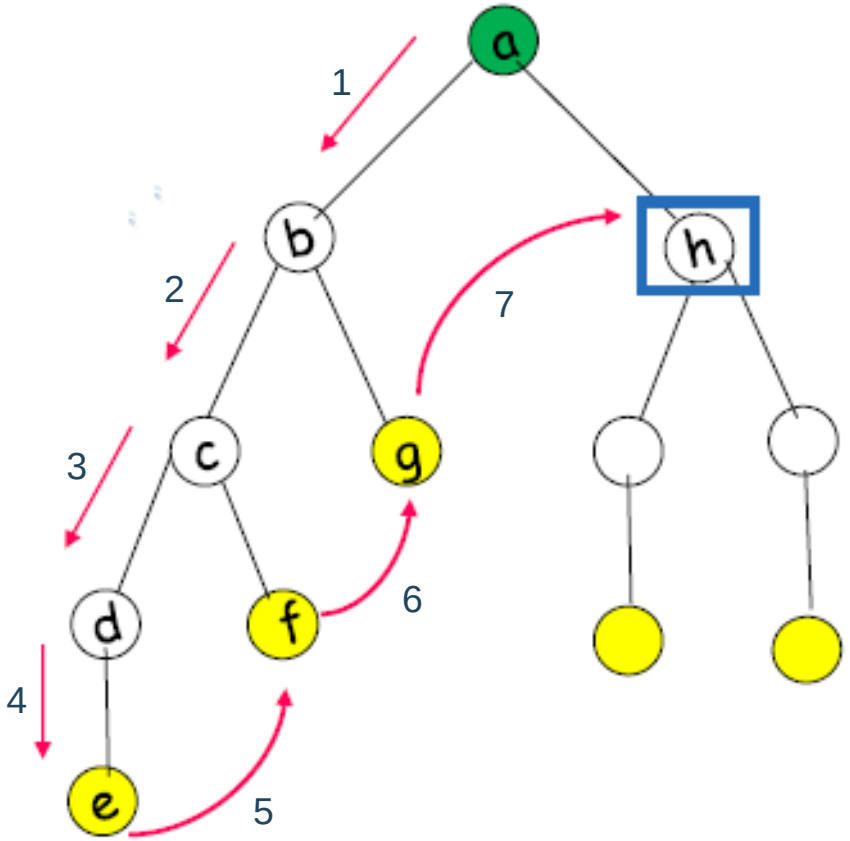


BFS or DFS

Solution

(a)





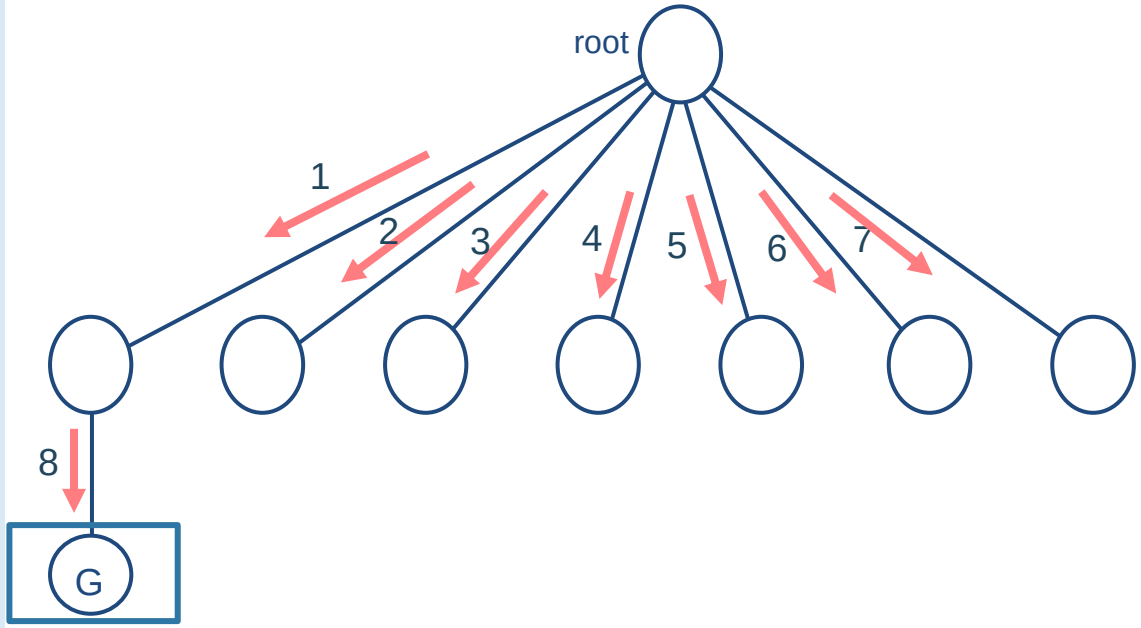
BFS finds the solution faster

DFS

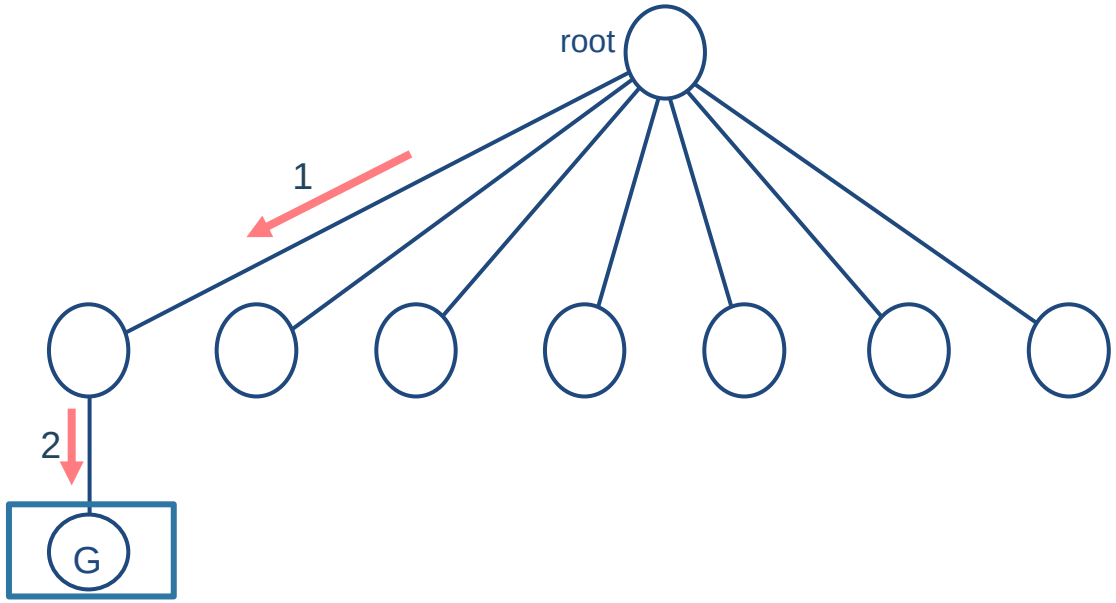
BFS or DFS

Solution

(b)



BFS



DFS finds the solution faster

Variations of DFS

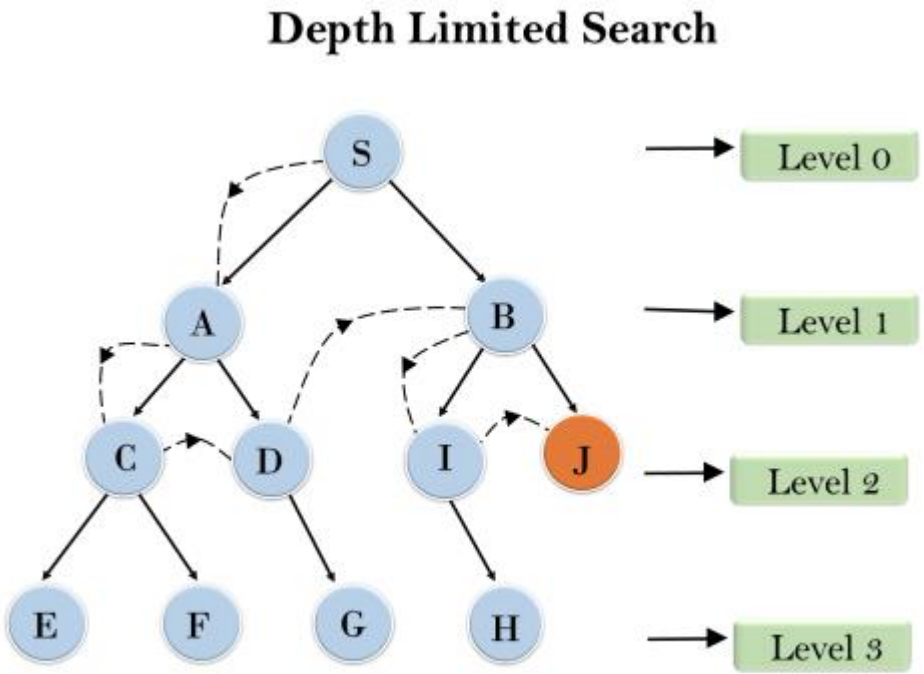
- **Depth Limited Search (DLS):**
  - just like DFS, except never go deeper than some depth d.
  - It can solve the problem of infinite path in the DFS.
  - In this algorithm, the node at the depth limit will be treated as a dead node.

Advantages:

Depth-limited search is Memory efficient.

Disadvantages:

- It also has a disadvantage of incompleteness. It is complete if the solution is above the depth-limit.
- It may not be optimal if the problem has more than one solution.




Any Questions