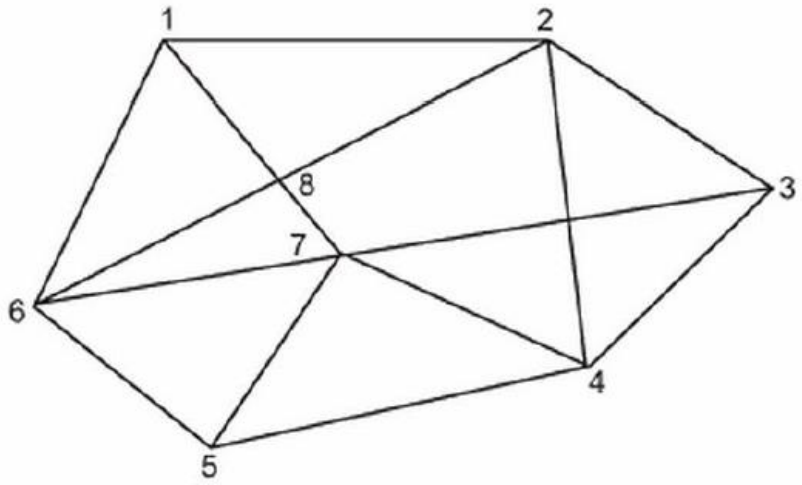
PROBLEM SOLVING & SEARCH STRATEGY
Part 2

Dr. Emad Natsheh

Traveling Salesman Problem

This problem falls in the category of path finding problems. The problem is defined as follows:

> *"Given 'n' cities connected by roads, and distances between each pair of cities. A sales person is required to travel each of the cities exactly once. We are required to find the route of salesperson so that by covering minimum distance, he can travel all the cities and come back to the city from where the journey was started".*



Traveling Salesman Problem

- The basic traveling salesman problem comprises of computing the shortest route through given a set of cities

Number of cities	Possible Routes
1	1
2	1 -2-1
3	1 -2 -3 1
	1 -3 -2 1
4	1- 2- 3- 4-1
	1- 2- 4- 3- 1
	1- 3- 2- 4- 1
	1- 3- 4- 2- 1
	1- 4- 2- 3-1
	1- 4- 3- 2- 1

Traveling Salesman Problem

- The number of routes between cities is proportional to the factorial of the (number of cities -1)
  - E.g. For three cities the number of routes will be 2*1 and for 4 well be 4! and so on.
- The number of routes increase so rapidly !!
- If it take 1 hour from a CPU to solve 30 cities it will take 30 hour for 31 cities and 330 hour for 32 cities !!!
- The proper solution of this problem is done using Neural Network
  - The neural network can solve the 10 city case just as fast as the 30 city case.

Search Technique

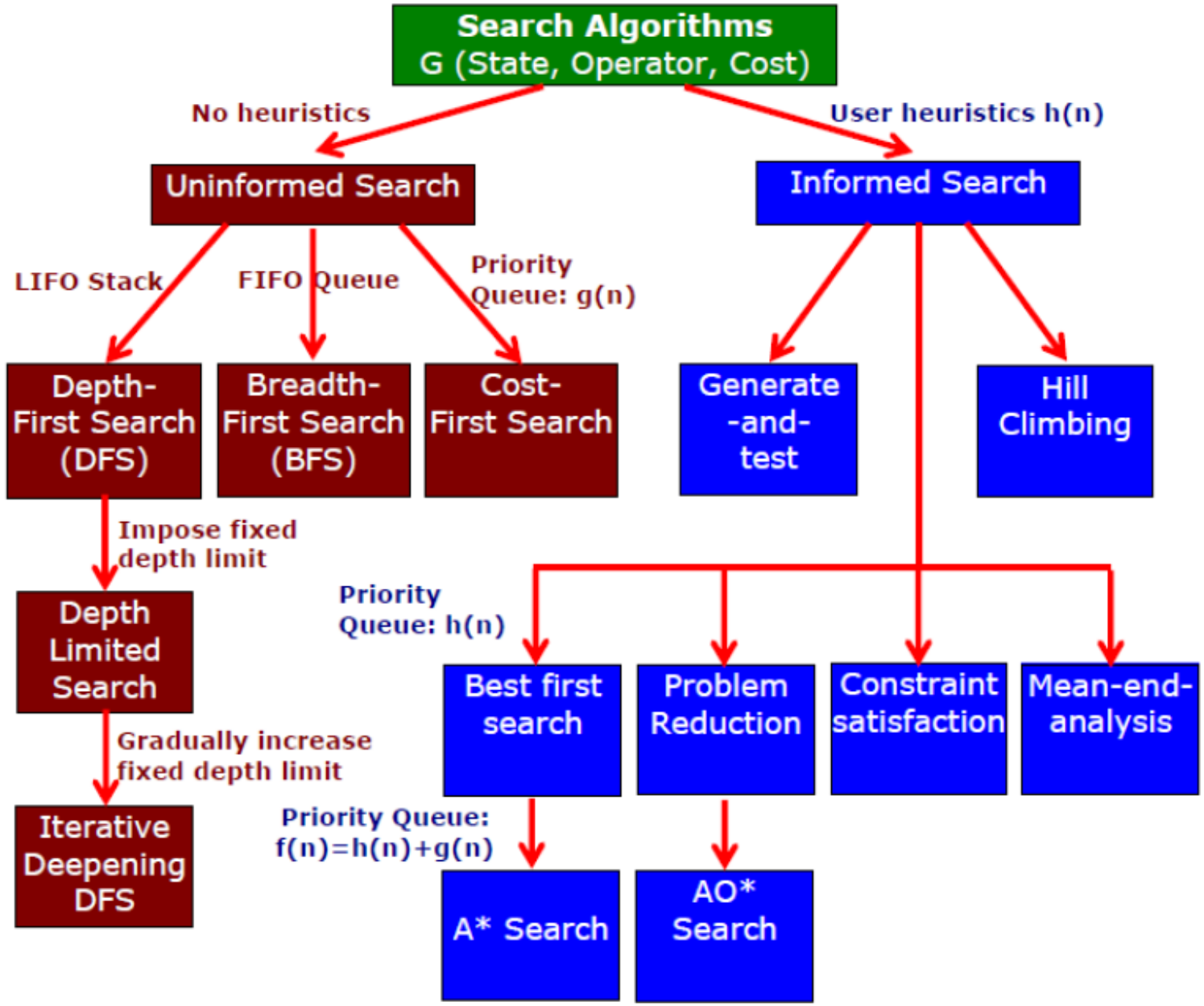
Search

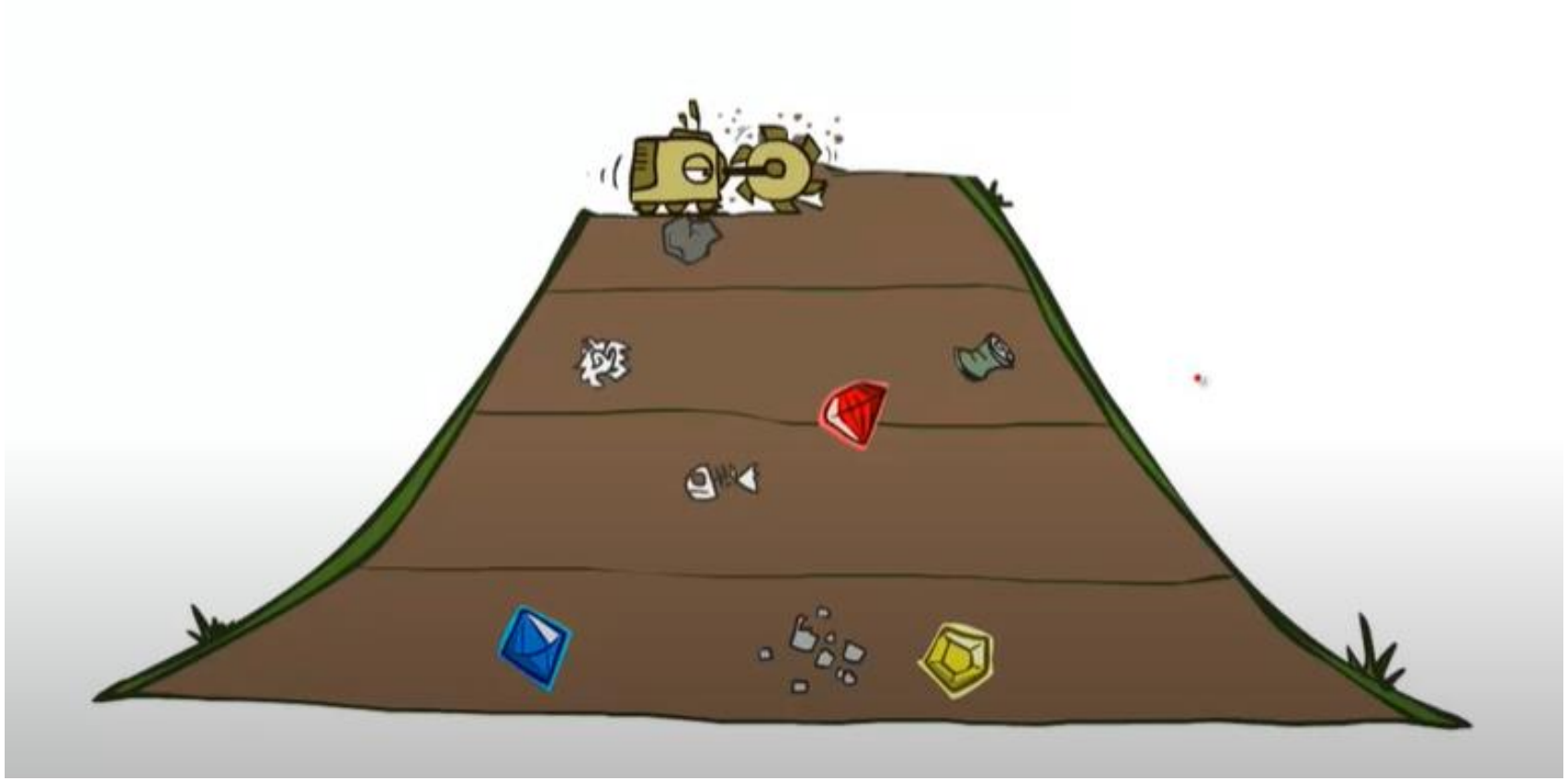
- The problem can be solved by using the rules, in combination with an appropriate control strategy, to move through the **problem space** until a path from an initial state to a goal state is found. This process is known as **search**.

- A very large number of AI problems are formulated as search problems.

Search Technique

- Blind (uninformed) search: is the search methodology having no additional information about states beyond that provided in the problem definitions
  - In this search total search space is looked for solution
- Heuristic (informed) search: these are the search techniques where additional information about the problem is provided in order to guide the search in a specific direction

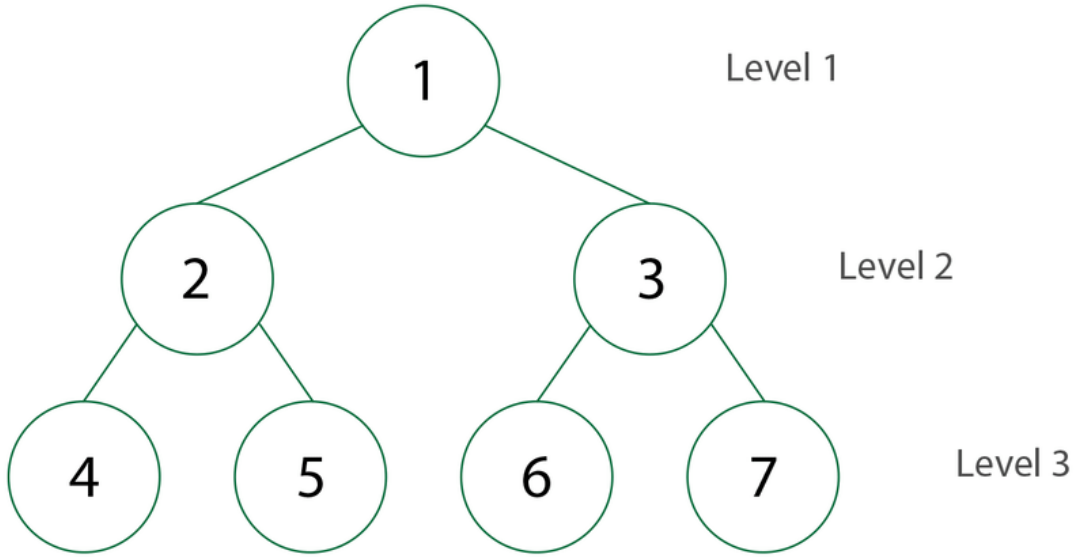

Search Algorithms
G (State, Operator, Cost)
No heuristics
User heuristics h(n)
Uninformed Search
Informed Search
LIFO Stack
FIFO Queue
Priority Queue: g(n)
Depth-First Search (DFS)
Breadth-First Search (BFS)
Cost-First Search
Generate-and-test
Hill Climbing
Impose fixed depth limit
Depth Limited Search
Gradually increase fixed depth limit
Iterative Deepening DFS
Priority Queue: h(n)
Best first search
Problem Reduction
Constraint satisfaction
Mean-end-analysis
Priority Queue: f(n)=h(n)+g(n)
A* Search
AO* Search



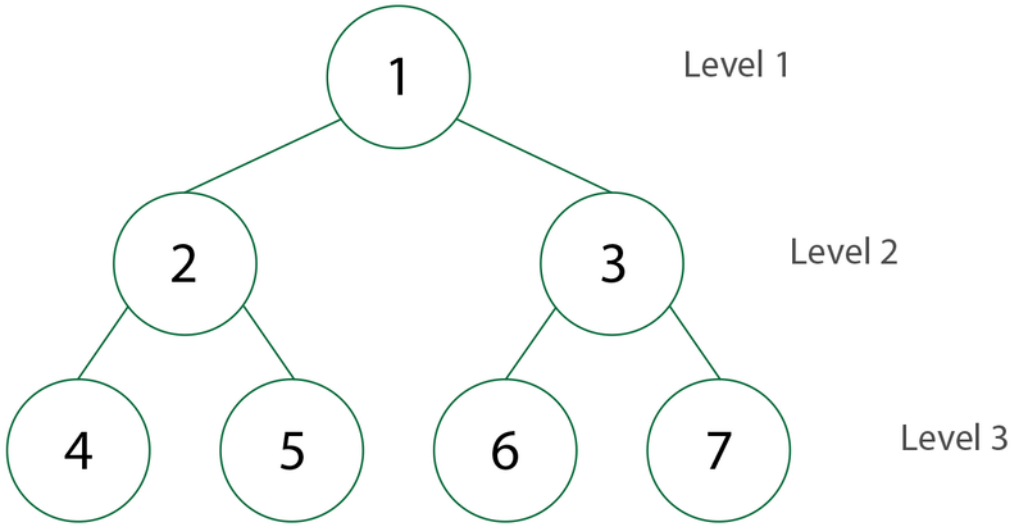
Breadth First Search

Breadth First Search

- Implementation: FIFO queue
- Strategy: expand a shallowest node first



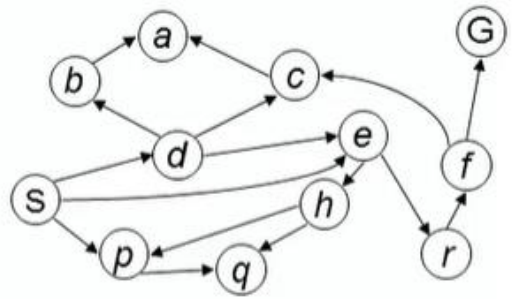
Breadth First Search (Example)

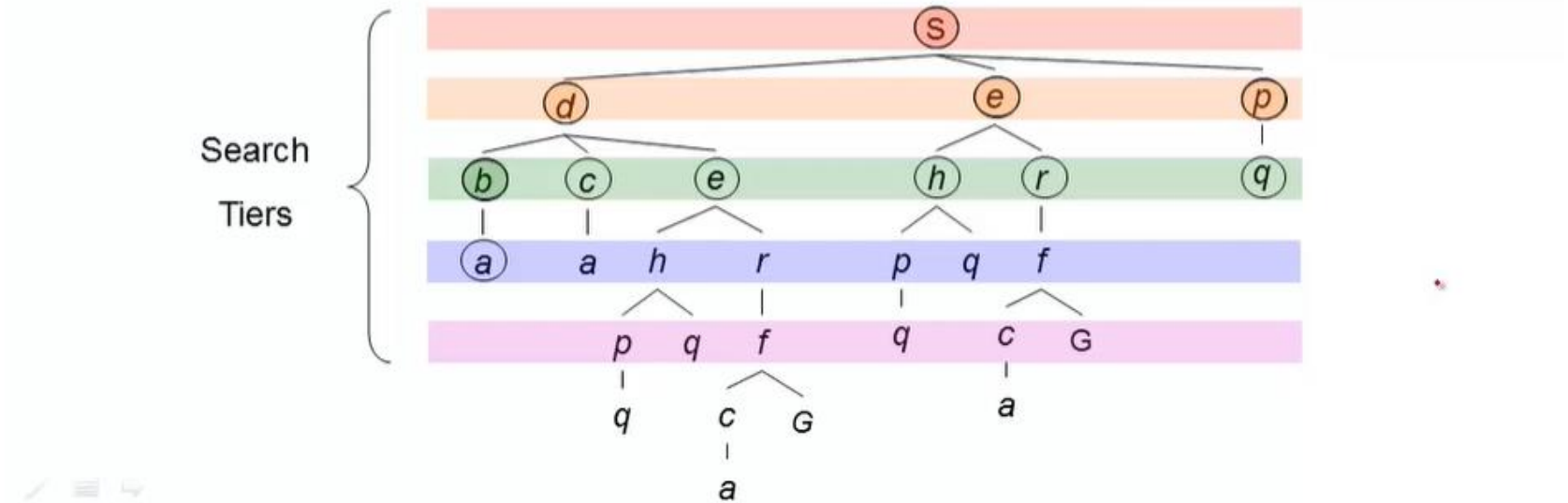


Breadth First Search (Example)

Strategy: expand a shallowest node first

Implementation: Fringe is a FIFO queue





BFS Analysis

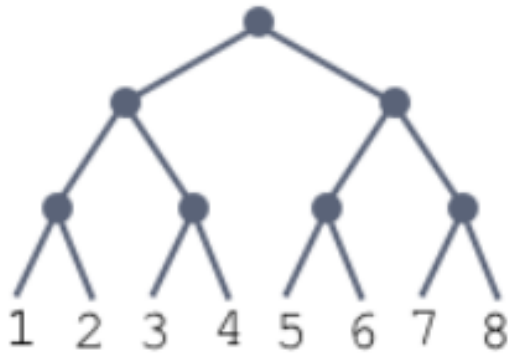
- Complete?
- Optimal?

Time vs Space Complexity

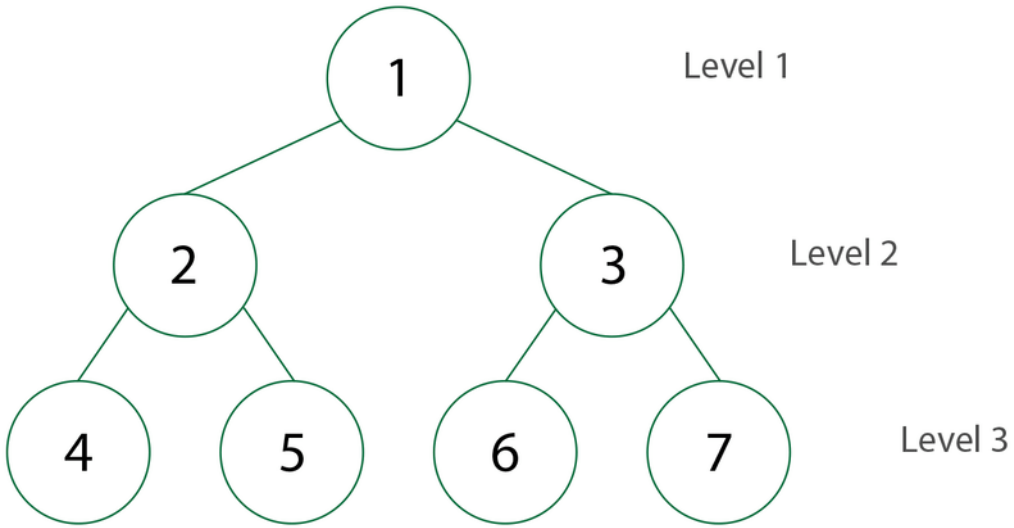
- Time Complexity:
  - It is the amount of time need to generate the nodes
- Space Complexity:
  - It is the amount of space or memory required for getting a solution
- **Big O notation**
  - Is a mathematical notation that describes the limiting behavior of a function
  - Used in Computer Science to describe the performance or complexity of an algorithm

Branching Factor

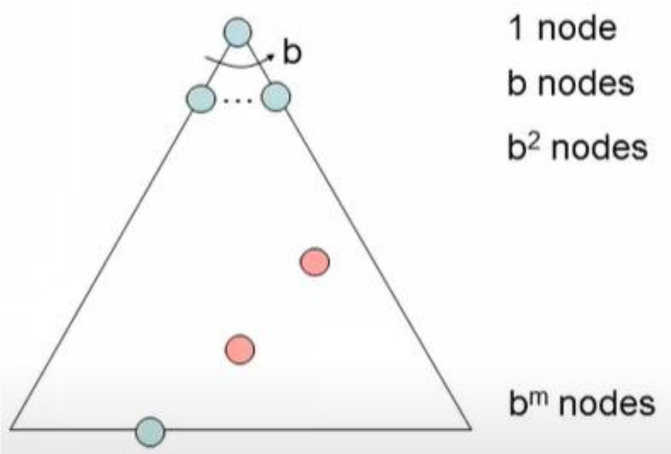
- The branching factor of a node in a tree is the number of children it has.

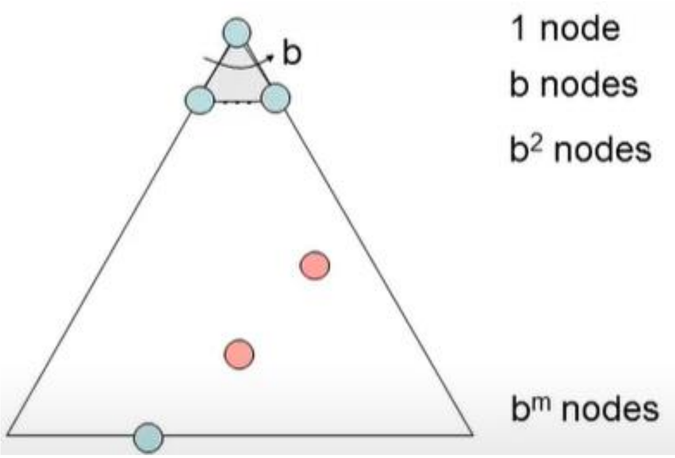


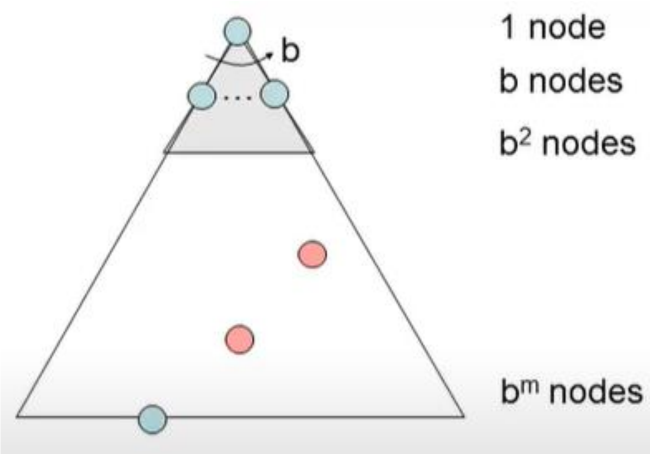
BFS Time and Space Complexity

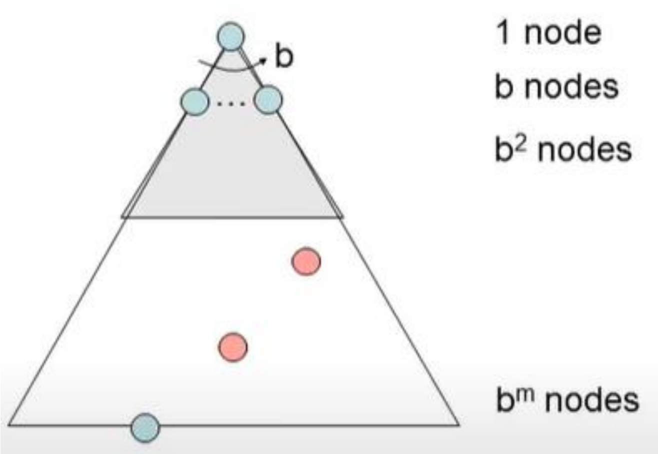


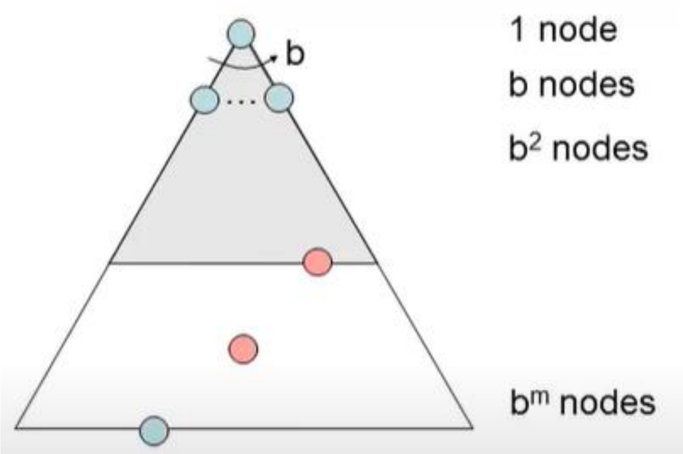
BFS Time and Space Complexity











$$1 + b + b^2 + b^3 + \cdots + b^d = O(b^d)$$

BFS Analysis

- Time Complexity → $1 + b + b^2 + b^3 + \cdots + b^d = O(b^d)$
- Space Complexity → O(b^d)
- Where b is branching factor and d is depth of solution

BFS Analysis

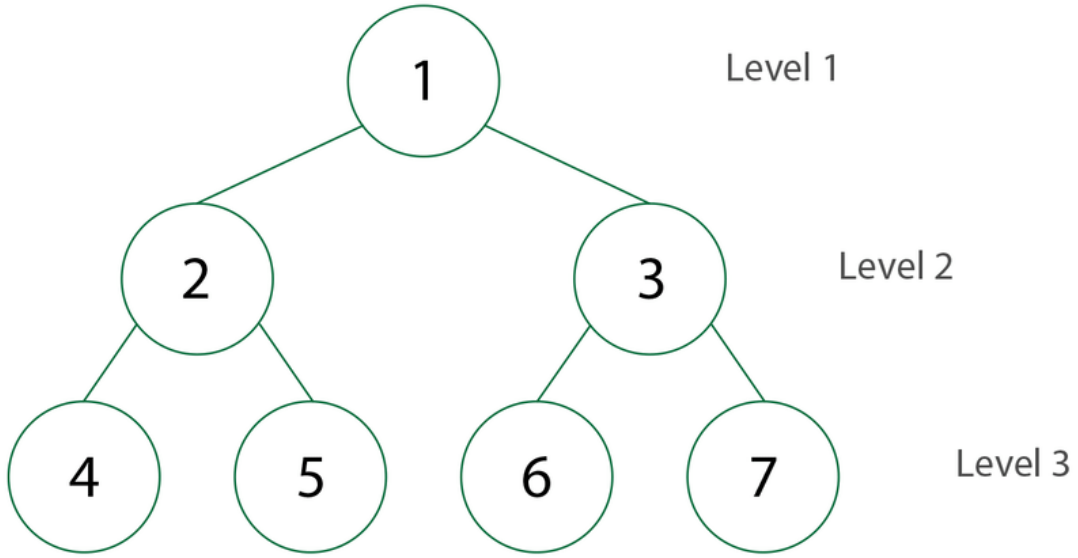
- Assume (branching factor) b =10
- CPU can expand 1 nodes/ms
- Each node size = 100 byte

d	Node expand	Time	Memory
0	1	1 ms	100 bytes
2	111	0.1 s	10 KB
4	11111	11 sec	1 MB
8	≅ 10^8	31 hour	11 GB
12	≅10^12	35 year	111 TB
14	≅10^14	3500 year	11111 TB

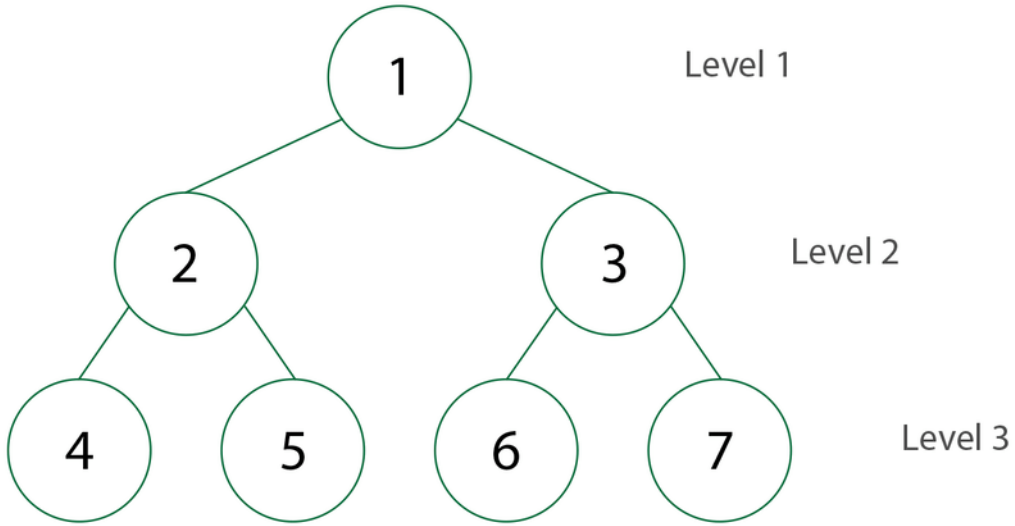
Depth First Search

Depth First Search

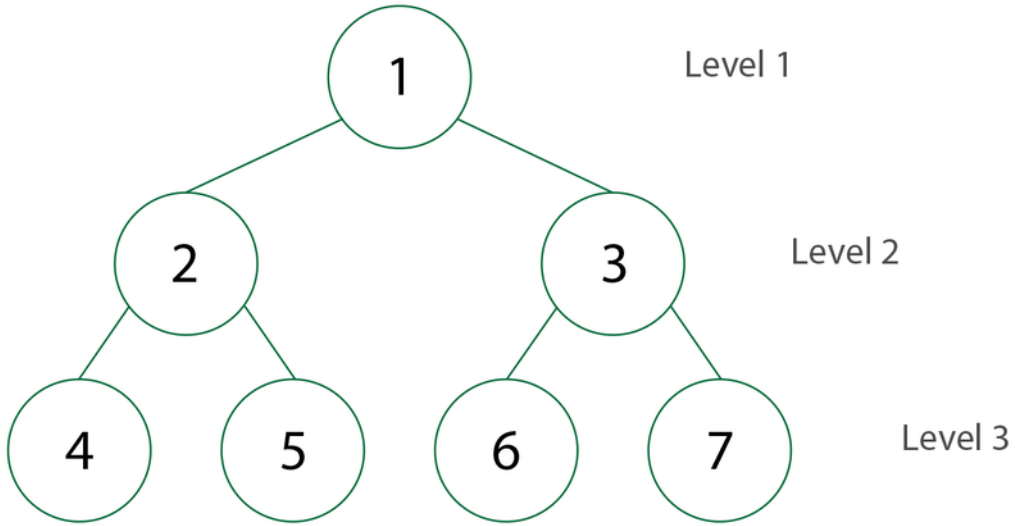
- Implementation: LIFO stack , queue
- Strategy: find the deepest solution



Depth First Search (Example- Stack)



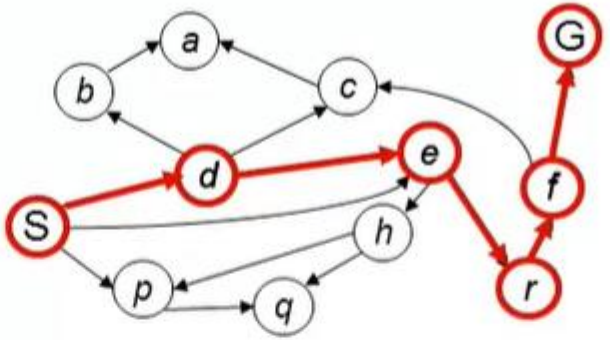
Depth First Search (Example- Queue)

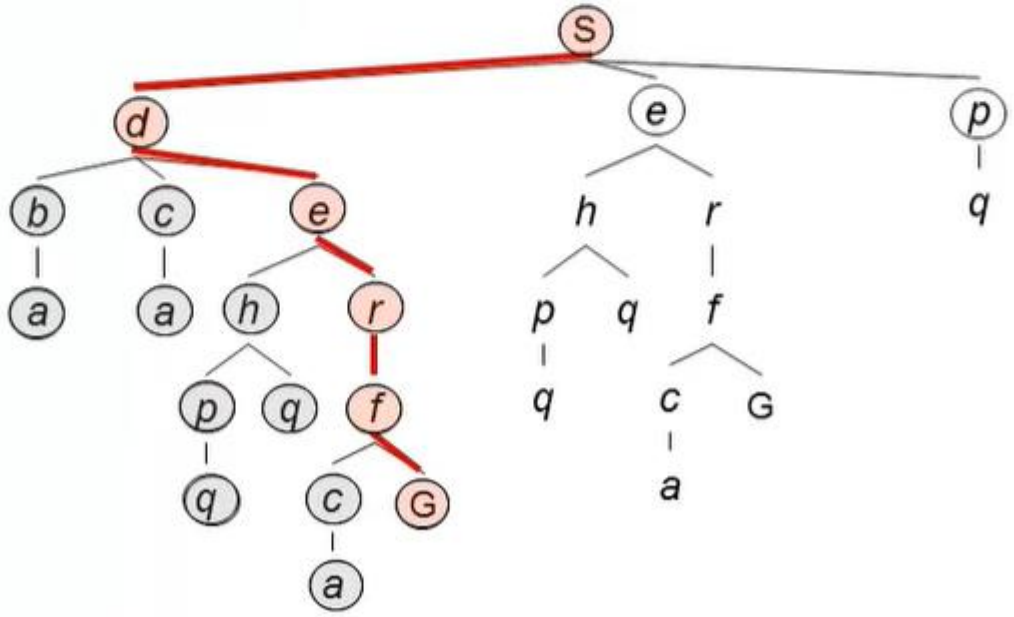


Depth First Search (Example)

Strategy: expand a deepest node first

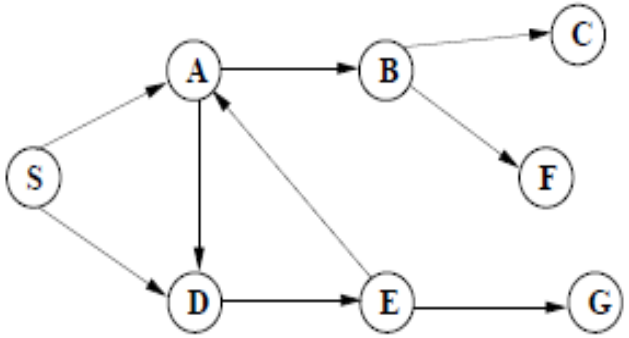
Implementation: Fringe is a LIFO stack

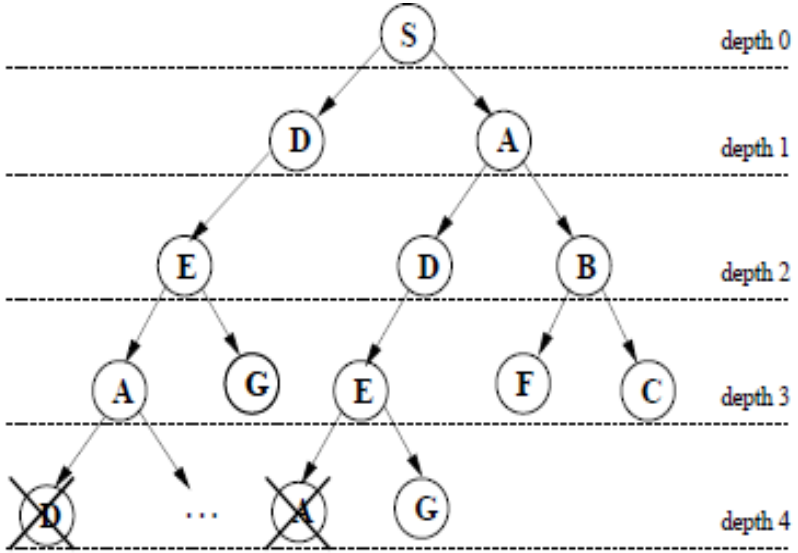




DFS Analysis

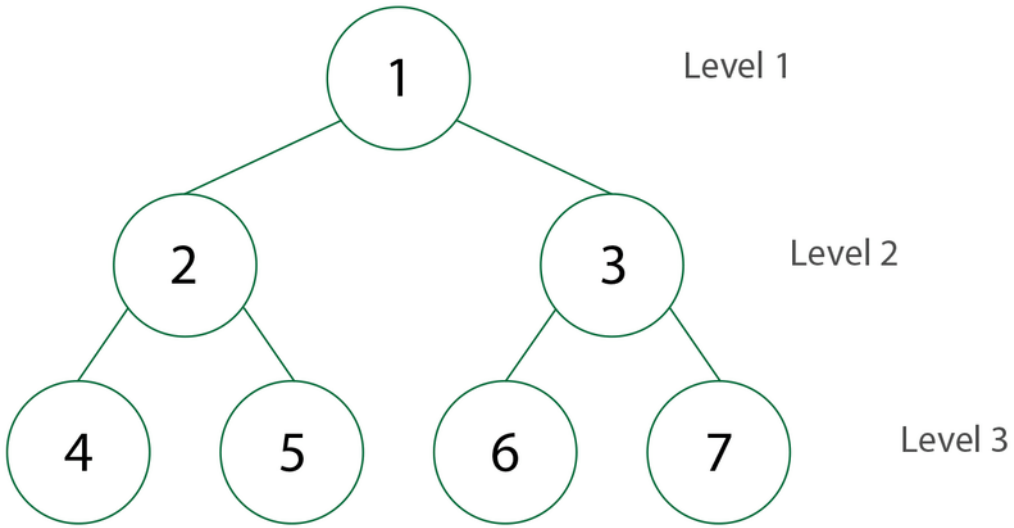
- Complete?



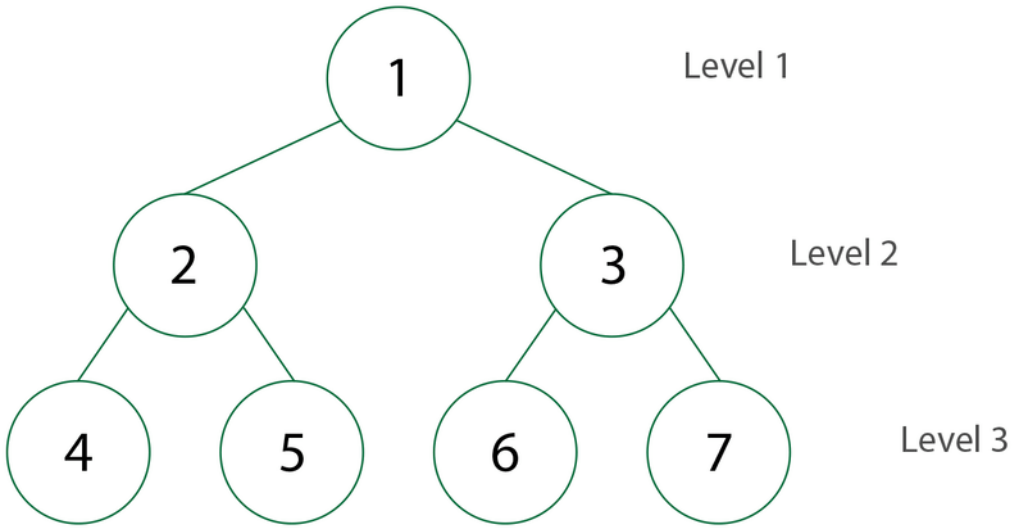


- Optimal?

DFS Time and Space Complexity (Stack)



DFS Time and Space Complexity (Queue)



DFS Analysis

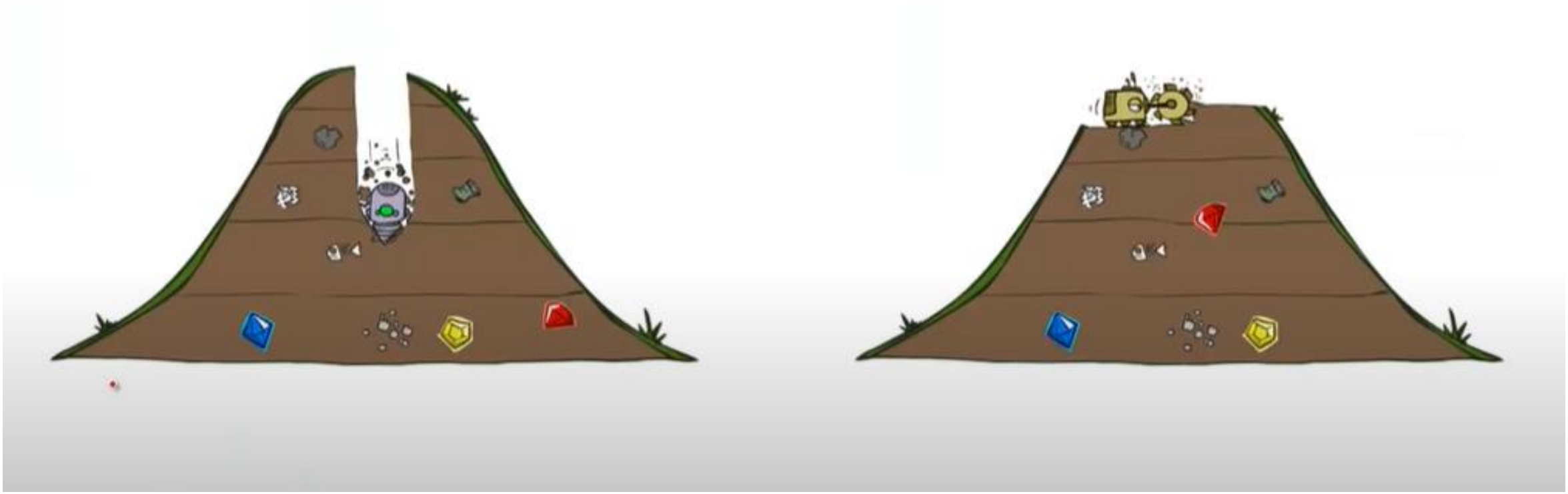
- Time Complexity → O(b^m)
- Stack
  - Space Complexity → O(m)
- Queue
  - Space Complexity → O(bm)

- Where b is branching factor and m is maximum depth

Quiz: DFS vs BFS

- When will BFS outperform DFS?

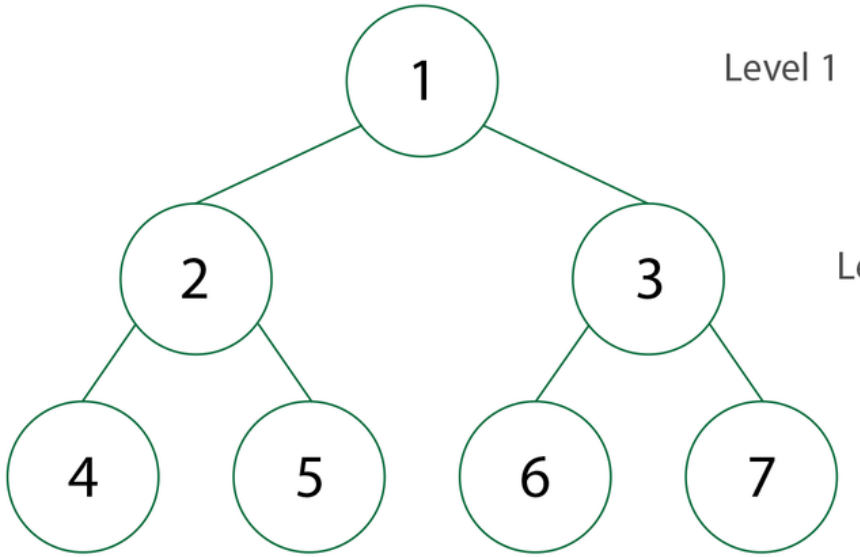
- When will DFS outperform BFS?



Iterative Deepening Depth First Search

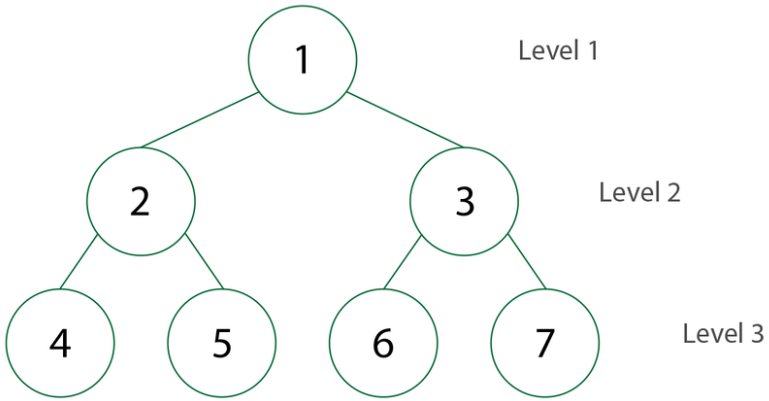
Iterative Deepening Depth First Search

- Its depth-limited version of depth-first search is run repeatedly with increasing depth limits until the goal is found.
- Find the shallowest solution

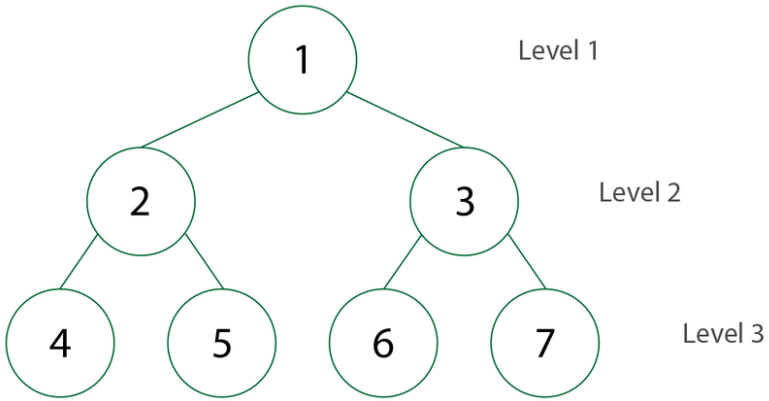


- Run a DFS with depth limit 1. If no solution...
- Run a DFS with depth limit 2. If no solution...
- Run a DFS with depth limit 3. .....

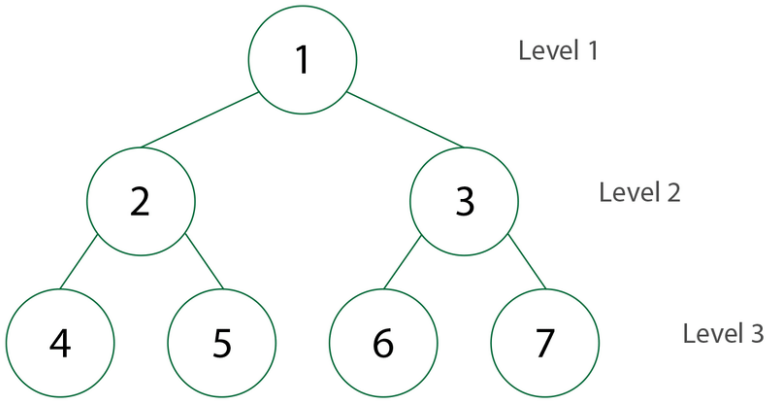
IDDFS(Example depth=0)



IDDFS(Example depth=1)



IDDFS(Example depth=2)



IDDFS Pseudocode

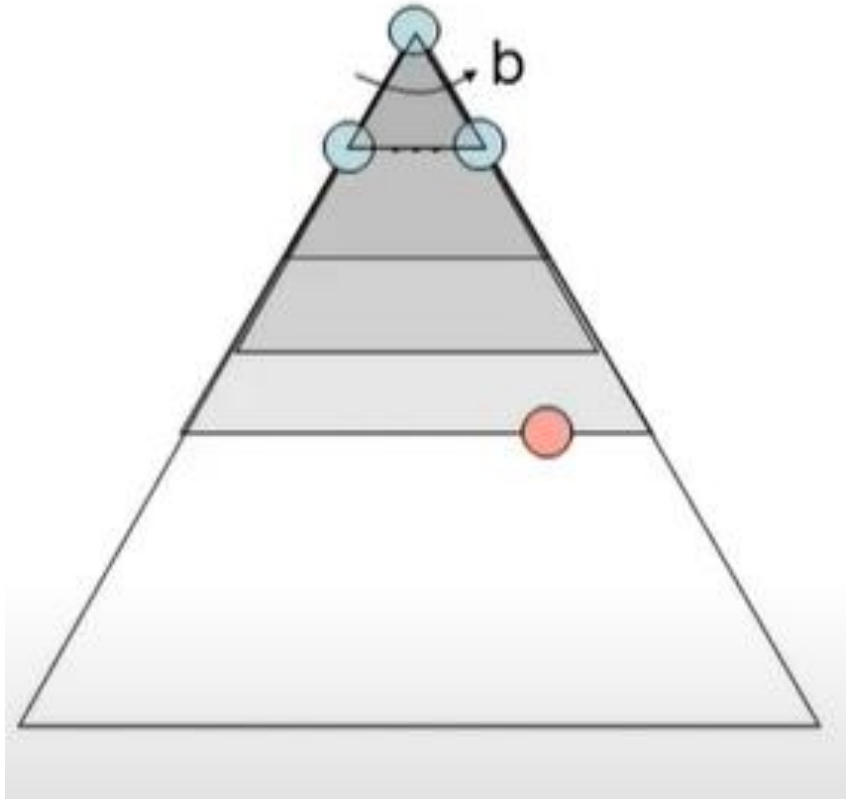
```
function IDDFS(root) is
    for depth from 0 to ∞ do
        found, remaining ← DLS(root, depth)
        if found ≠ null then
            return found
        else if not remaining then
            return null

function DLS(node, depth) is
    if depth = 0 then
        if node is a goal then
            return (node, true)
        else
            return (null, true)    (Not found, but may have children)

    else if depth > 0 then
        any_remaining ← false
        foreach child of node do
            found, remaining ← DLS(child, depth−1)
            if found ≠ null then
                return (found, true)
            if remaining then
                any_remaining ← true    (At least one node found at depth, let IDDFS deepen)
        return (null, any_remaining)
```

IDDFS Analysis

- Complete?

- Optimal?

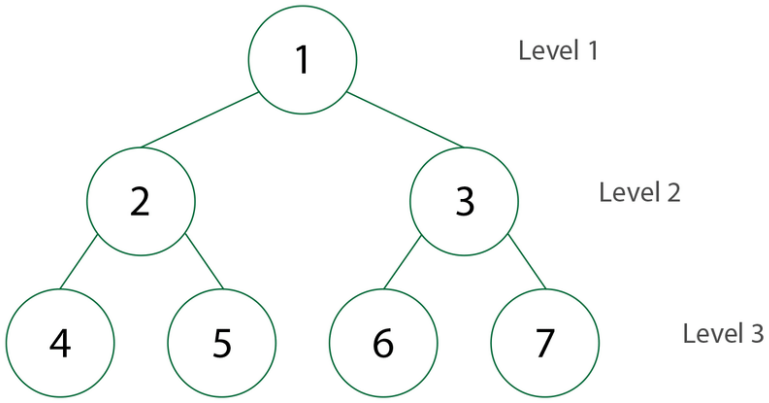
IDDFS Time Complexity



$$(d+1) + d(b) + (d-1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + b^d$$

$$= \sum_{i=0}^{d} (d+1-i)\, b^i$$

IDDFS Space Complexity



IDDFS Analysis

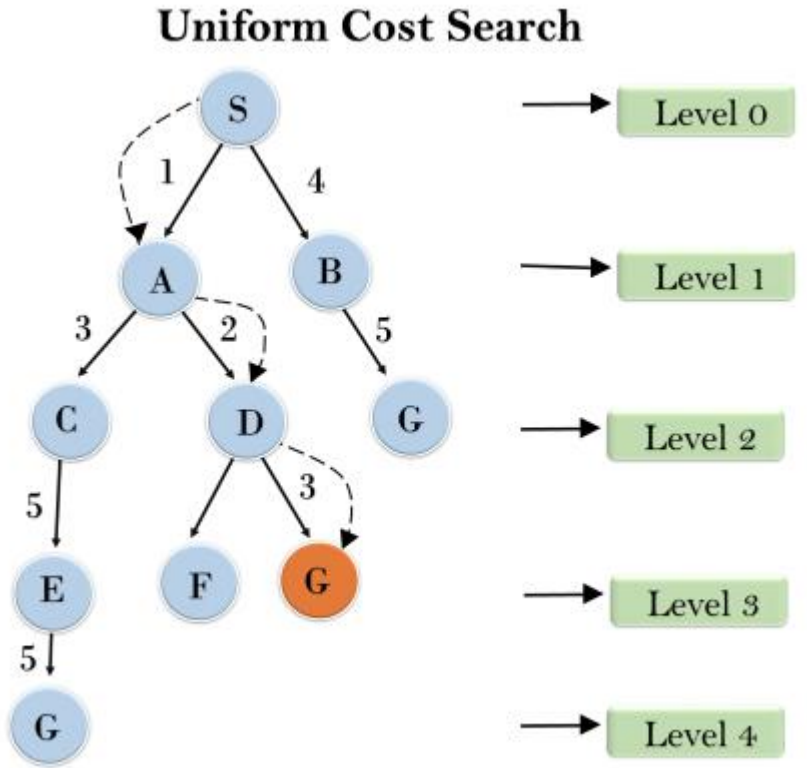
- Time Complexity → O(b^d)
- Stack
  - Space Complexity → O(d)
- Queue
  - Space Complexity → O(bd)

- Where b is branching factor and d is depth of solution

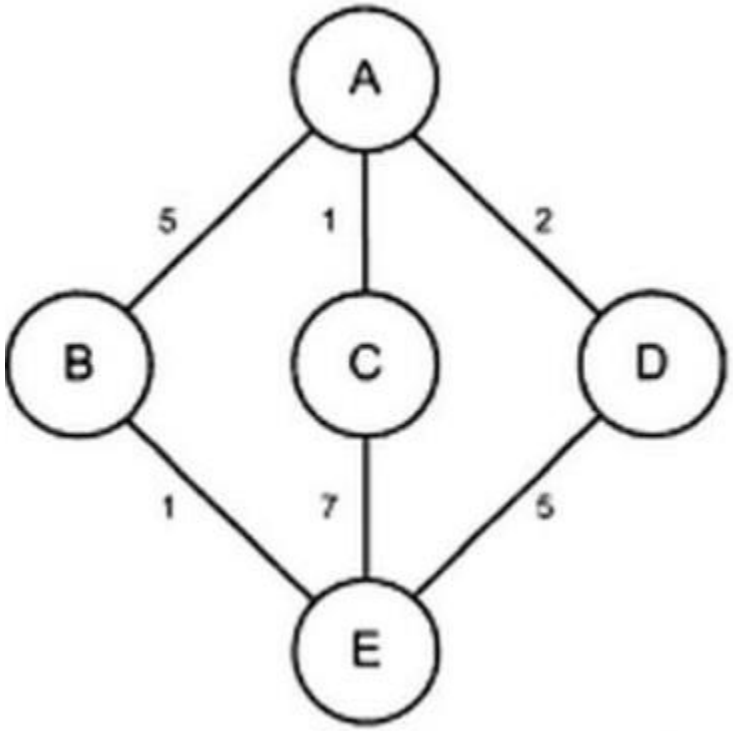
Uniform Cost Search

UCS (Branch and Bound Search)

- Implementation: priority queue (sort by cost function **g(n)**)
- **g(n)** is cost from root node to current node n
- Uniform cost search vs Dijkstra !!!!

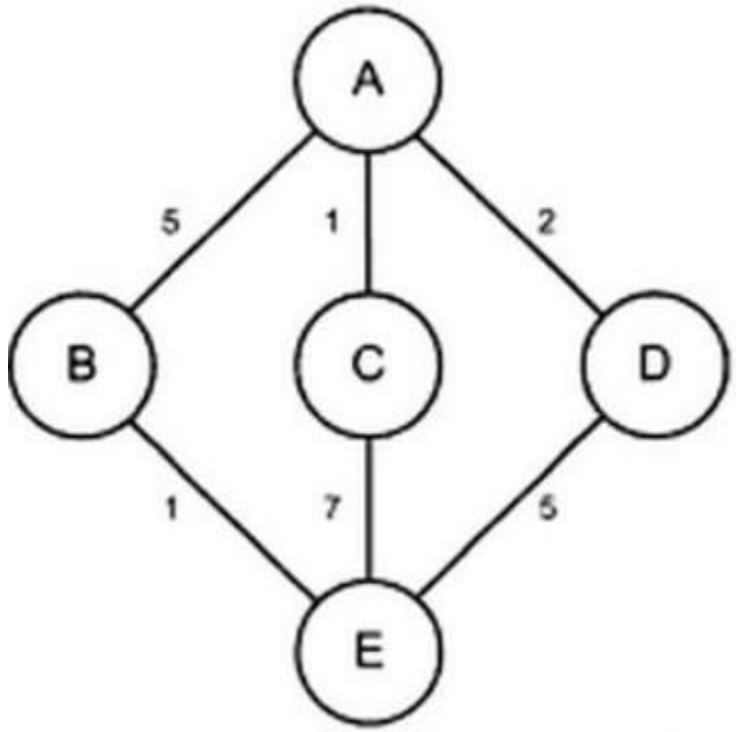


UCS (Example Table)

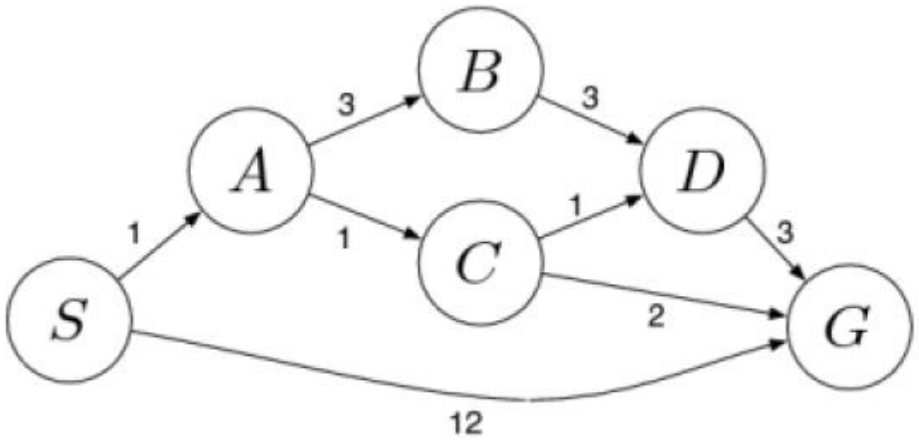


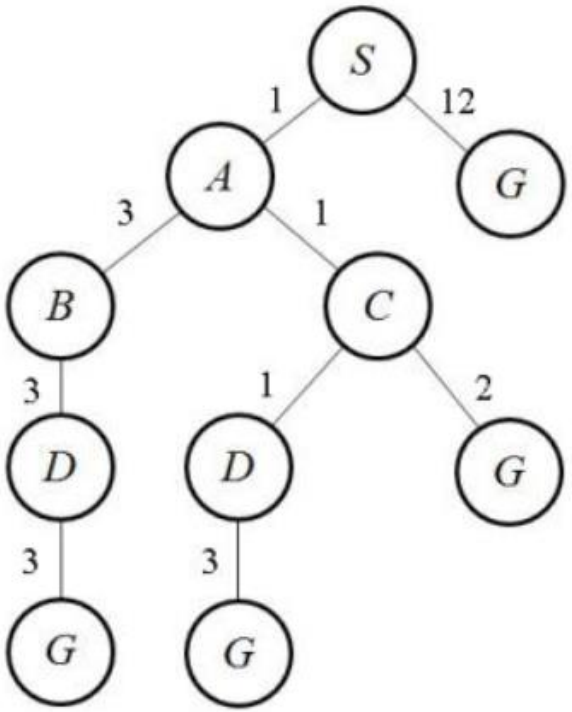
Visited Nodes (Close)	Priority Queue (Open)
-	A

UCS (Example Tree)

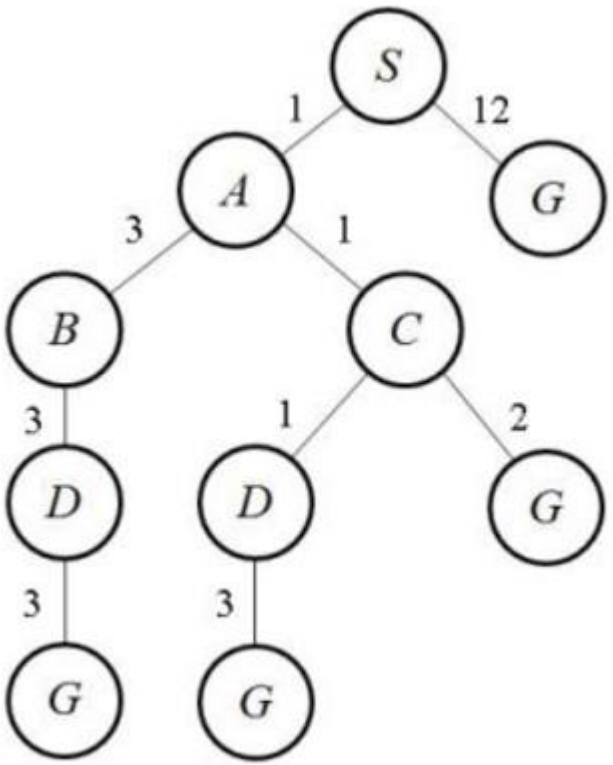


Class Exercise



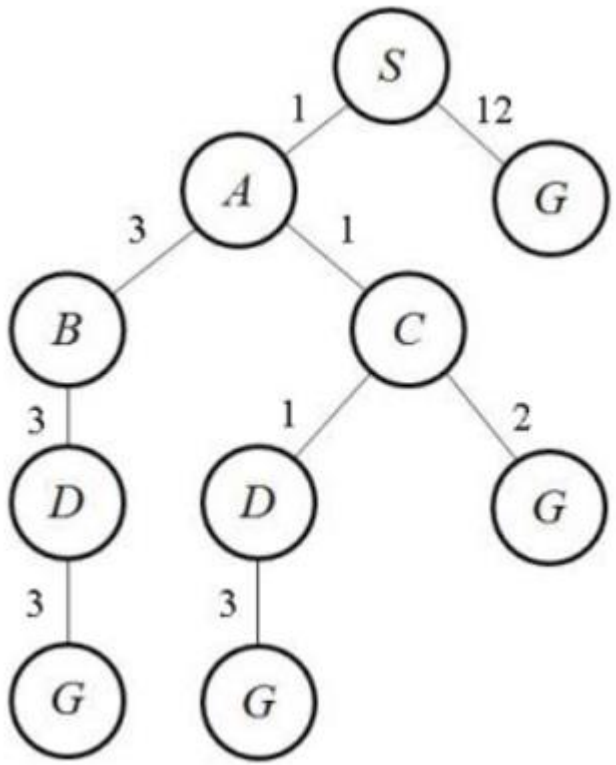


Class Exercise (Solution Table)



Visited Nodes (Close)	Priority Queue (Open)
-	S

Class Exercise (Solution Tree)



UCS Pseudocode

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier ← a priority queue ordered by PATH-COST, with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            frontier ← INSERT(child, frontier)
```

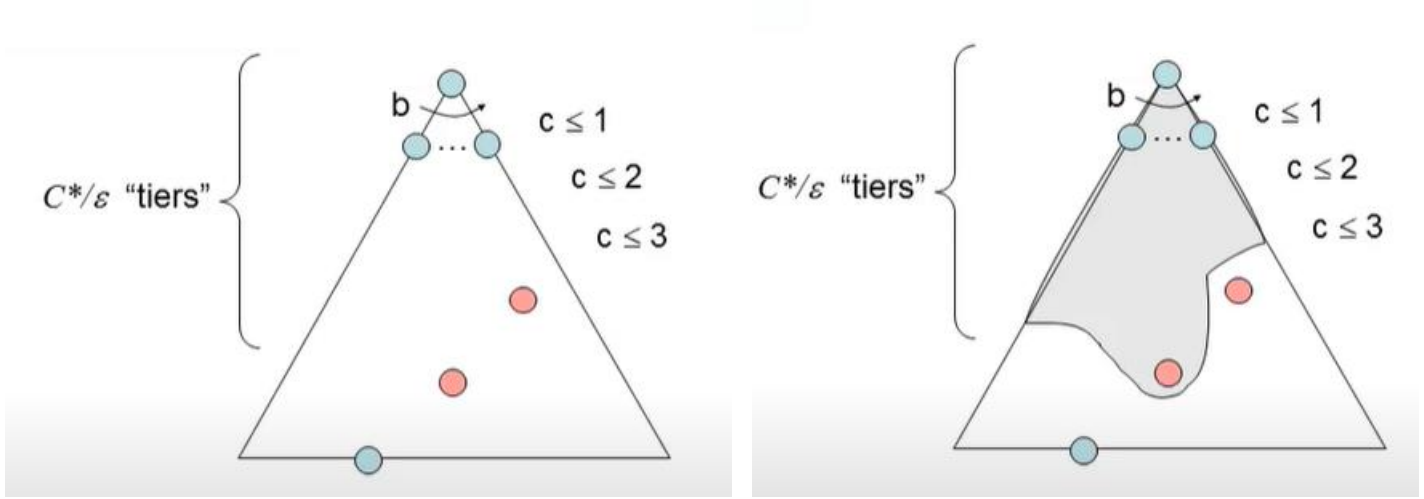
UCS Analysis

- Complete?

- Optimal?

UCS Analysis

- Time Complexity
  - $O(b^d)$
  - $O(b^{C*/\varepsilon})$
- Space Complexity
  - $O(b^d)$
  - $O(b^{C*/\varepsilon})$



UCS and 8 puzzle