جامعة الجلالة
GALALA UNIVERSITY

Artificial Intelligence Science Program

Chapter 3: Solving Problems by Searching

Solving Problems by Searching

- How an agent can look ahead to find a sequence of actions that will achieve its goal.
- Problem-solving agent is the agent needs to plan ahead: to consider a sequence of actions that form a path to a goal state.
- The computational process agent undertakes is called **search**.
- Two types of algorithms:
    1. Informed
    2. Uninformed

7 h 17 min (578 km)

via A1 and E81

Fastest route, the usual traffic

Arad

Romania

- Get on A1 from Strada Andrei Şaguna, Strada Dorobanţi and Calea Bodrogului/DJ682F

  8 min (4.3 km)

- Get on DN68A/E673 in Judeţul Timiş from A1

  53 min (101 km)

- Get on A1 in Şoimuş from DN68A/E673 and DN7/E68

  1 h 13 min (75.7 km)

- Follow A1, DN1/DN7/E68/E81 and A1 to DN1/DN7/E68/E81 in Judeţul Sibiu. Take the DN7/DN1 exit from A1

  1 h 18 min (131 km)

- Continue to Râmnicu Vâlcea

  1 h 21 min (86.4 km)

- Continue onto DN7/E81

  Continue to follow E81

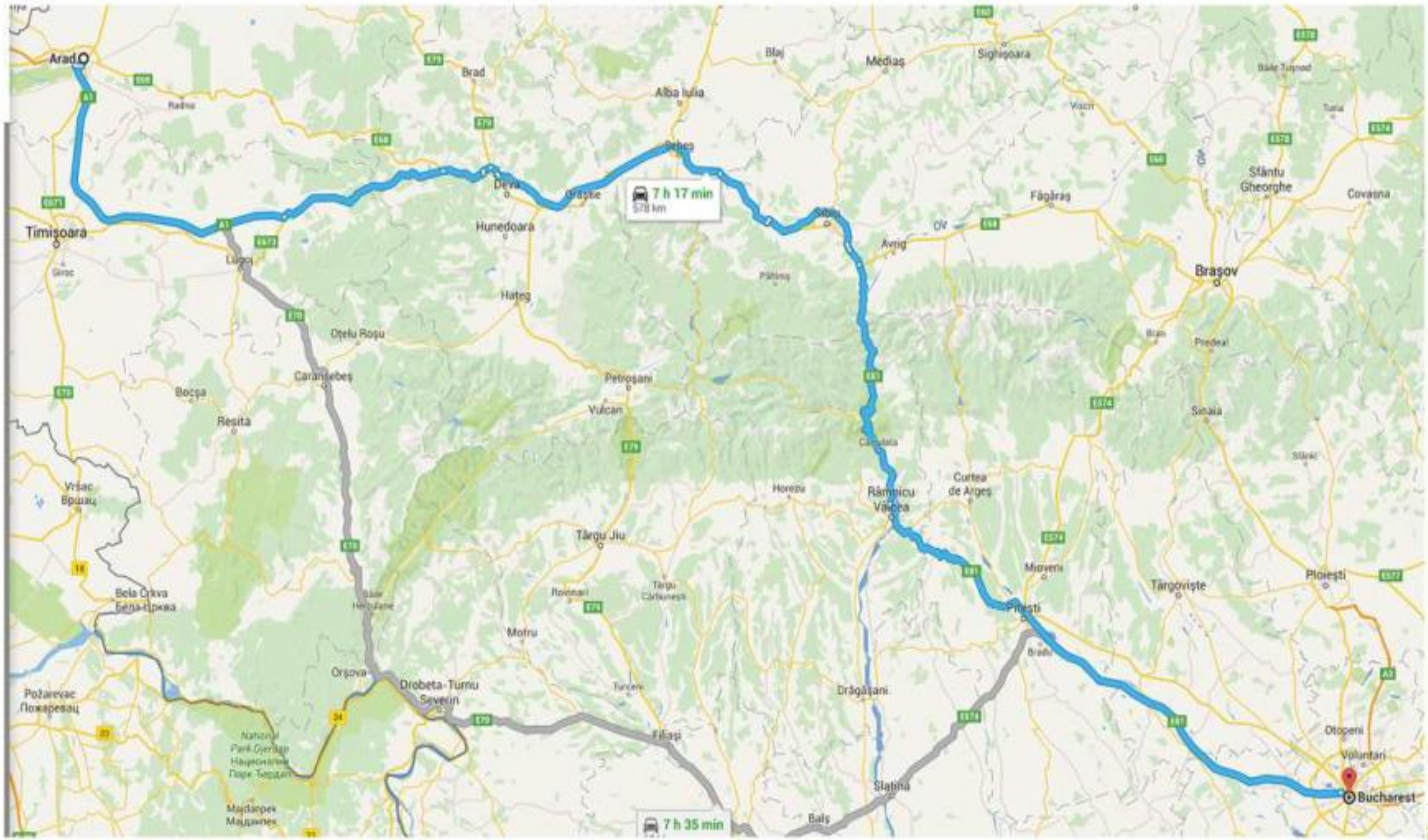
  2 h 15 min (177 km)

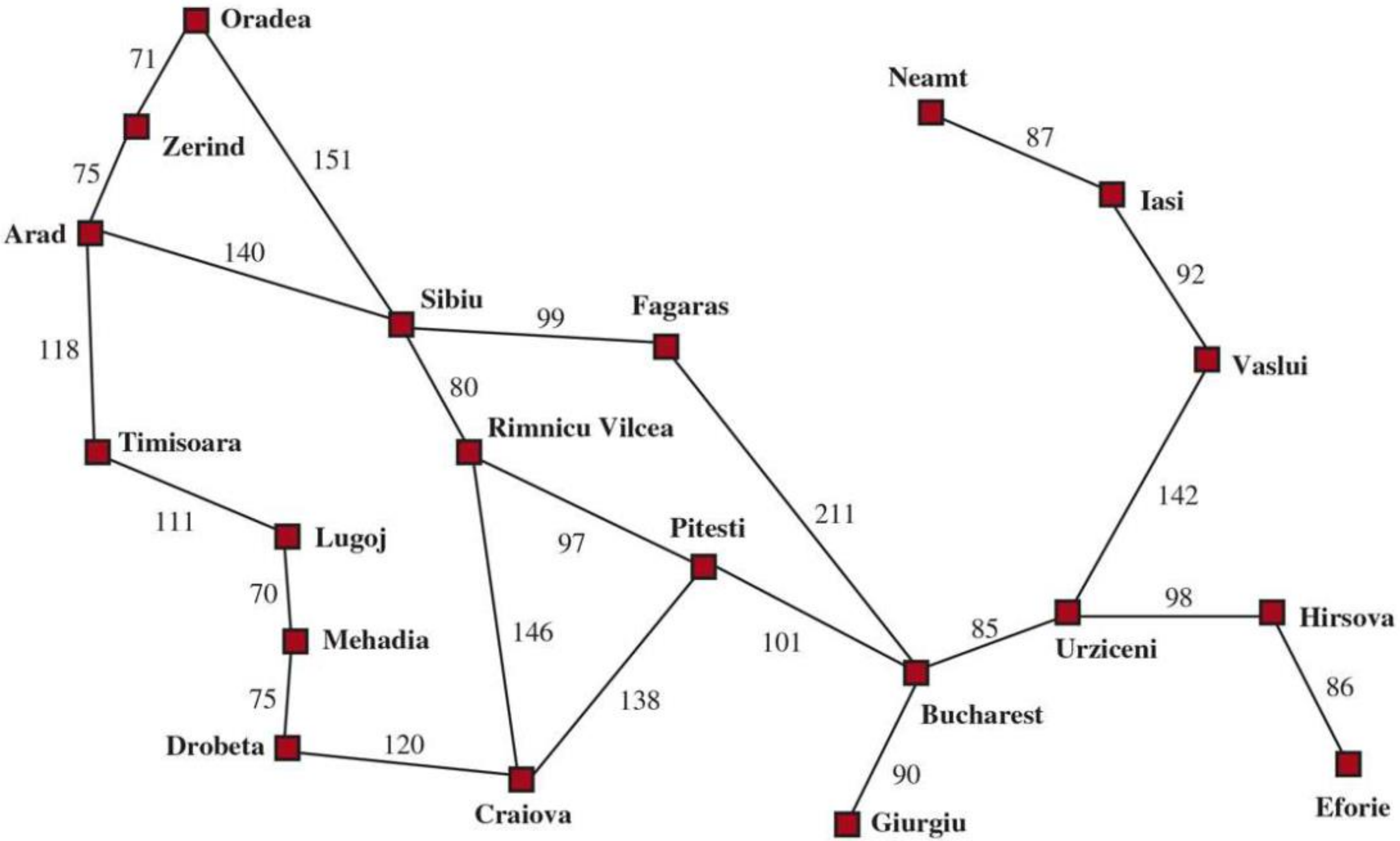
- Take Splaiul Independenţei to Bulevardul Unirii/E81

  9 min (3.5 km)

Bucharest

Romania





A simplified road map of part of Romania, with road distances in miles.

Four-phase problem-solving process

- **GOAL FORMULATION:** The agent adopts the goal of reaching Bucharest.
- **PROBLEM FORMULATION:** The agent devises a description of the states and actions necessary to reach the goal.
  - For our agent, one good model is to consider the actions of traveling from one city to an adjacent city, and therefore the only fact about the state of the world that will change due to an action is the current city.

Four-phase problem-solving process

- **SEARCH**: Before taking any action in the real world, the agent simulates sequences of actions in its model, searching until it finds a sequence of actions that reaches the goal.
  - Such a sequence is called a solution.
  - The agent might have to simulate multiple sequences that do not reach the goal, but eventually it will find a solution, or it will find that no solution is possible.
- **EXECUTION**: The agent can now execute the actions in the solution, one at a time.

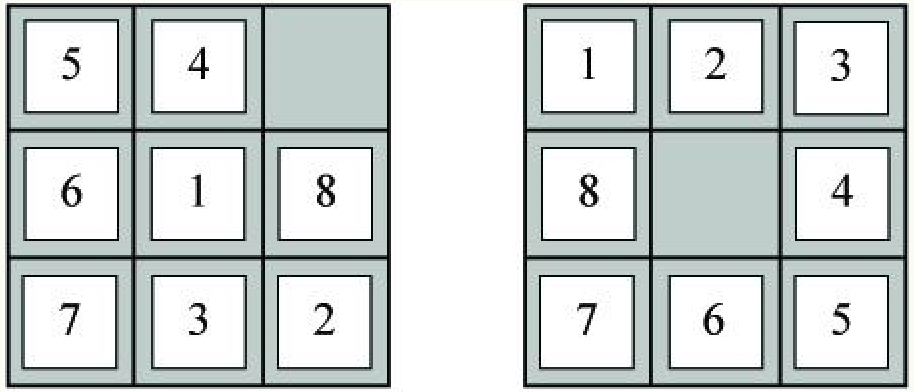
Search Space Definitions

- State
  - A description of a possible state of the world
  - Includes all features of the world that are relevant to the problem
- Initial state
  - Description of all relevant aspects of the state in which the agent starts the search
- Goal test
  - Conditions the agent is trying to meet (e.g., have $1M)
- Goal state
  - Any state which meets the goal condition
  - Thursday, have $1M, live in NYC
- Action
  - Function that maps (transitions) from one state to another

Search Space Definitions

- Problem formulation
  - Describe a general problem as a search problem
- Solution
  - Sequence of actions that transitions the world from the initial state to a goal state
- Solution cost (additive)
  - Sum of the cost of operators
  - Alternative: sum of distances, number of steps, etc.
- Search
  - Process of looking for a solution
  - Search algorithm takes problem as input and returns solution
  - We are searching through a space of possible states
- Execution
  - Process of executing sequence of actions (solution)

Example Problems – Eight Puzzle



Start State Goal State

States: tile (square) locations

Initial state: one specific tile configuration

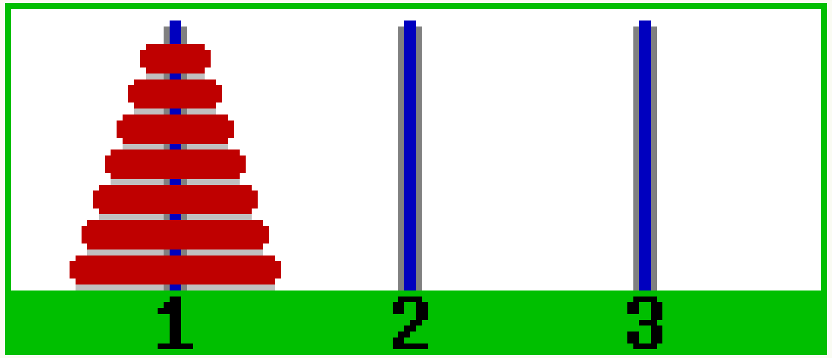
Operators: move blank tile left, right, up, or down

Goal: tiles are numbered from one to eight around the square

Path cost: cost of 1 per move (solution cost same as number of most or path length)

[Eight puzzle applet]

Example Problems – Towers of Hanoi



States: combinations of poles and disks

Operators: move disk x from pole y to pole z subject to constraints

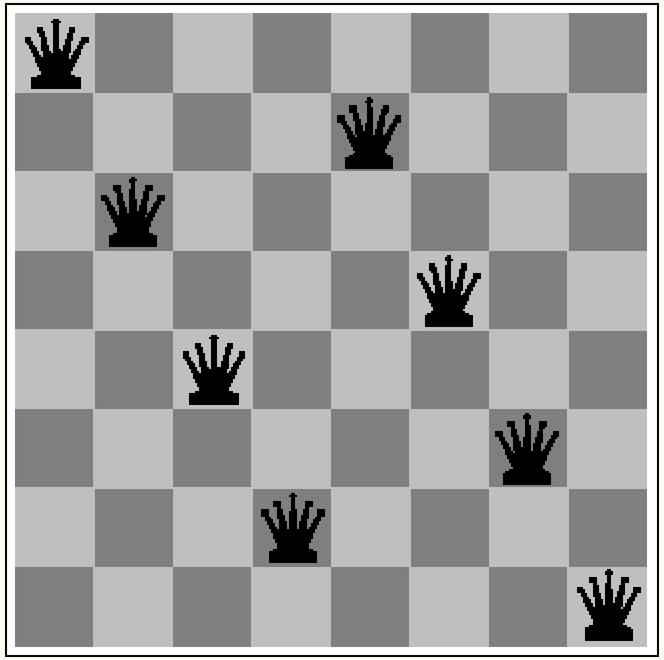
- cannot move disk on top of smaller disk
- cannot move disk if other disks on top

Goal test: disks from largest (at bottom) to smallest on goal pole

Path cost: 1 per move

[Towers of Hanoi applet]

Example Problems – Eight Queens



States: locations of 8 queens on chess board

Initial state: one specific queens configuration

Operators: move queen x to row y and column z

Goal: no queen can attack another (cannot be in same row, column, or diagonal)

Path cost: 0 per move

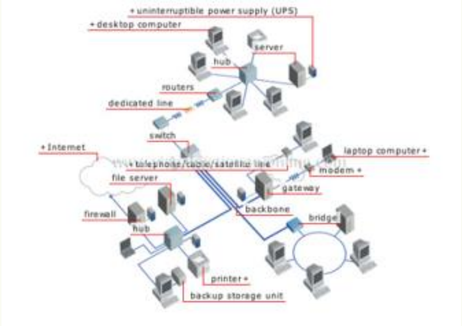
[Eight queens applet]

Route Finding Problem

- **States**
  - locations
- **Initial state**
  - starting point
- **Successor function (operators)**
  - move from one location to another
- **Goal test**
  - arrive at a certain location
- **Path cost**
  - may be quite complex • money, time, travel comfort, scenery,

Car Navigation

Airline travel planning



Routing in computer Networks

Automatic Assembly Sequencing

- **States**
  - location of components
- **Initial state**
  - no components assembled
- **Successor function (operators)**
  - place component
- **Goal test**
  - system fully assembled
- **Path cost**
  - number of moves

Searching for Solutions

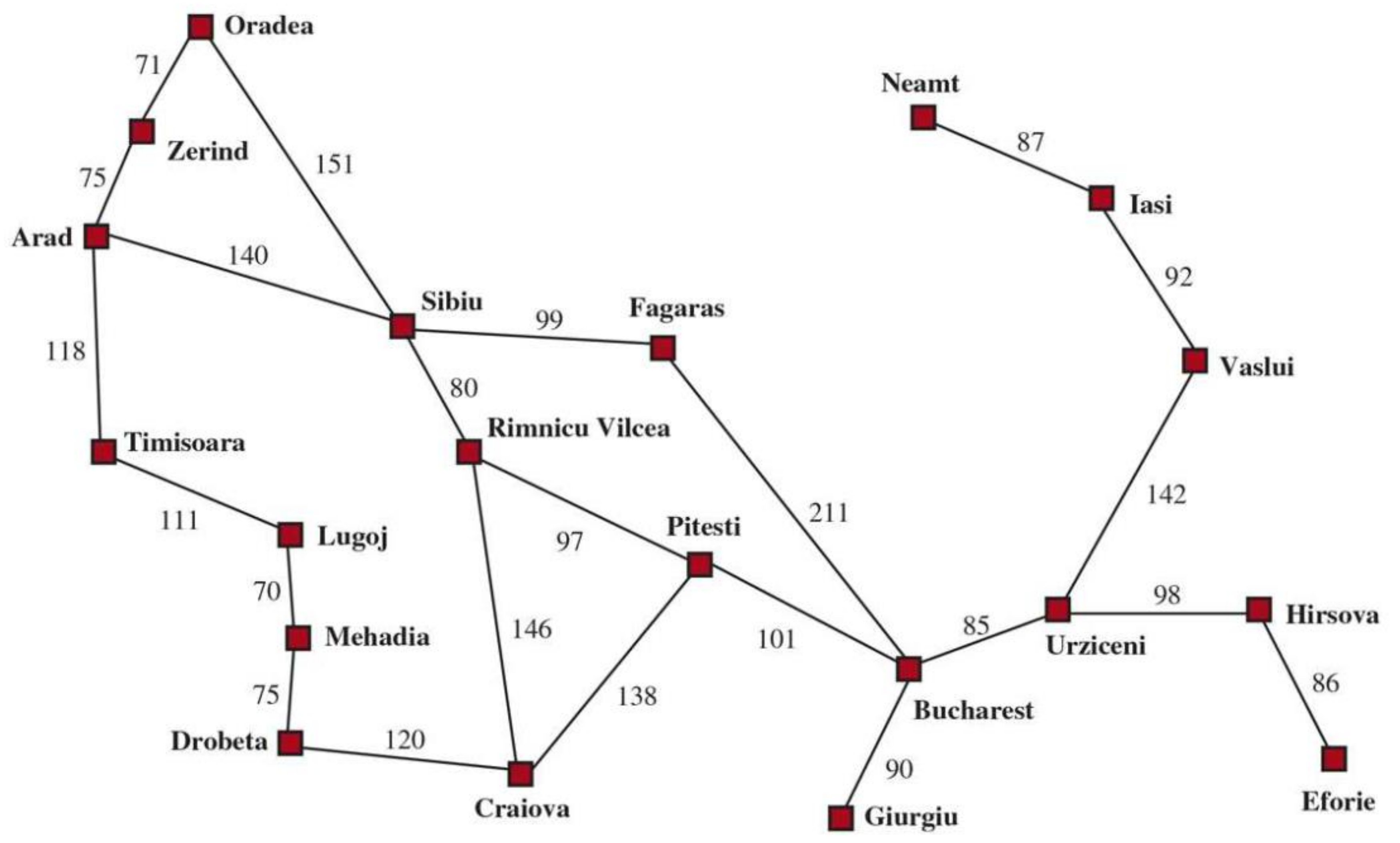
- Traversal of the search space
  - From the initial state to a goal state.
  - Legal sequence of actions as defined by successor function.
- A search tree is generated
  - Nodes are added as more states are visited

Search Algorithms

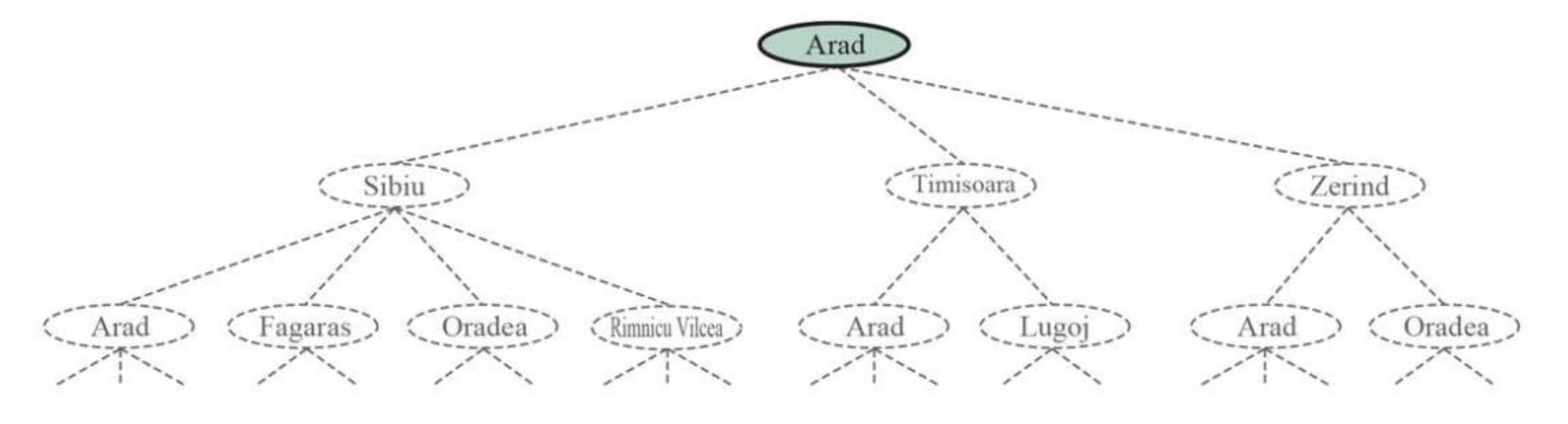
- A search algorithm takes a search problem as input and returns a solution
- Each node in the search tree corresponds to a state in the state space and the edges in the search tree correspond to actions.
- The root of the tree corresponds to the initial state of the problem.

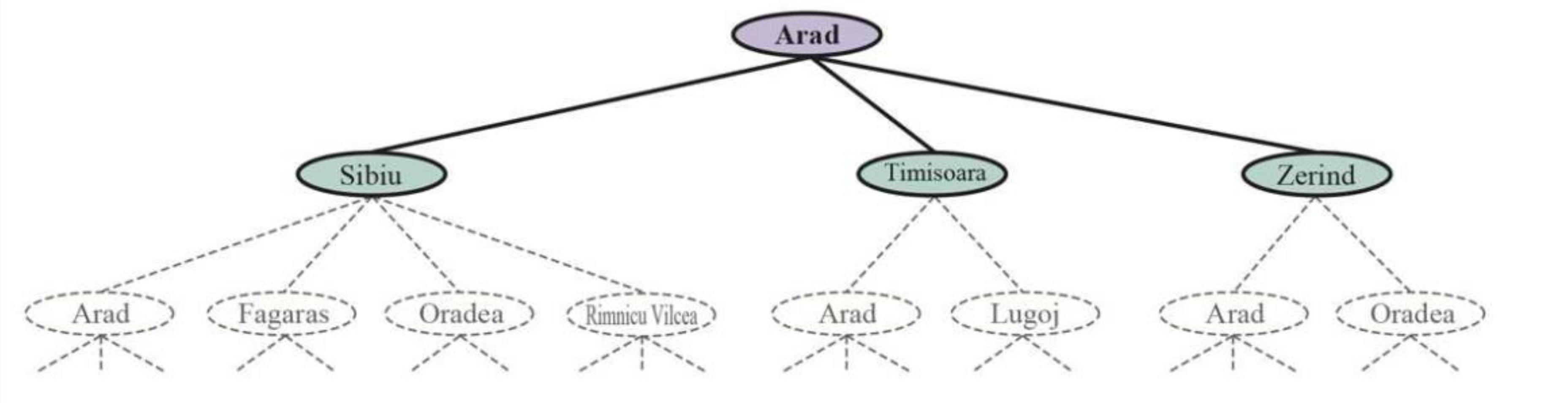
- The state space describes the set of states in the world, and the actions that allow transitions from one state to another.
- The search tree describes paths between these states, reaching towards the goal.
- The search tree may have multiple paths to any given state, but each node in the tree has a unique path back to the root (as in all trees).

Finding a path from Arad to Bucharest.

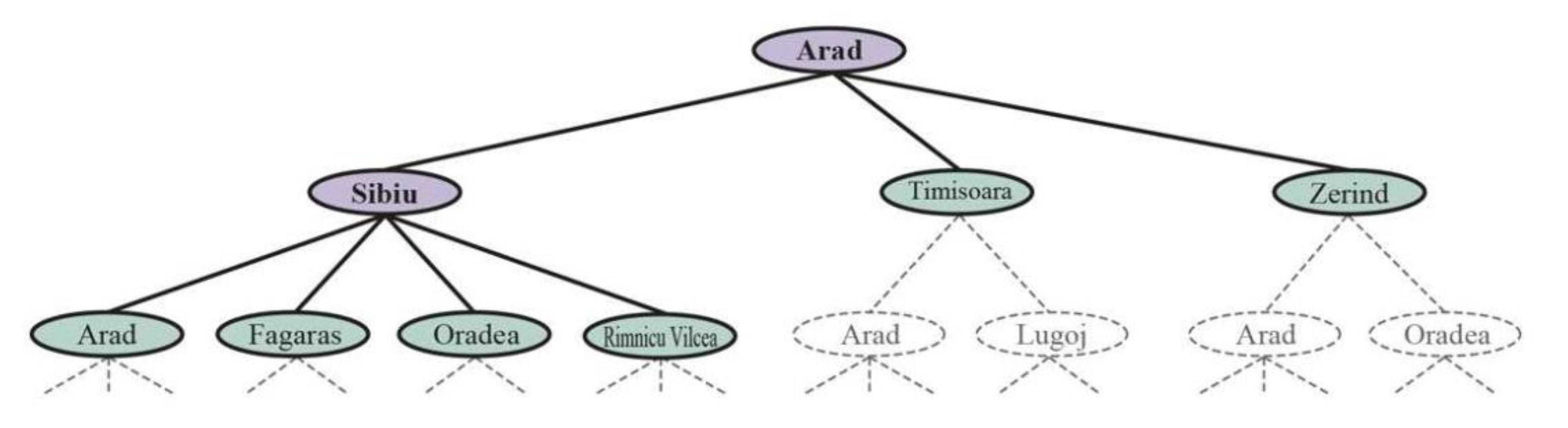


A simplified road map of part of Romania, with road distances in miles.

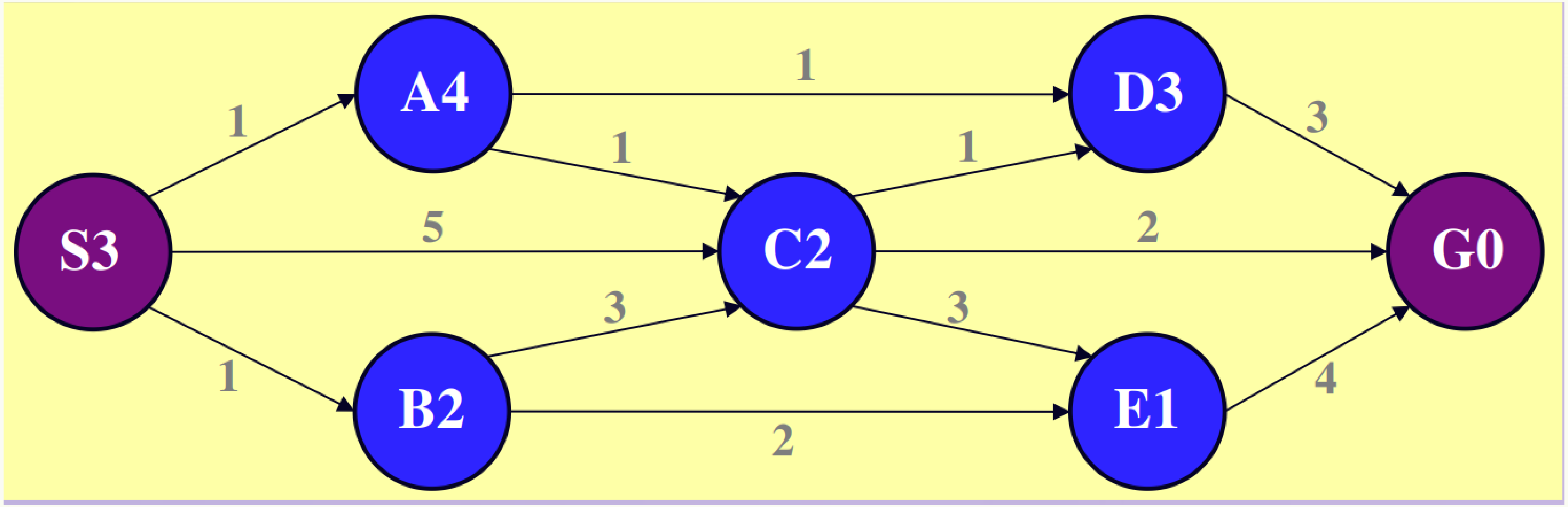

Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea

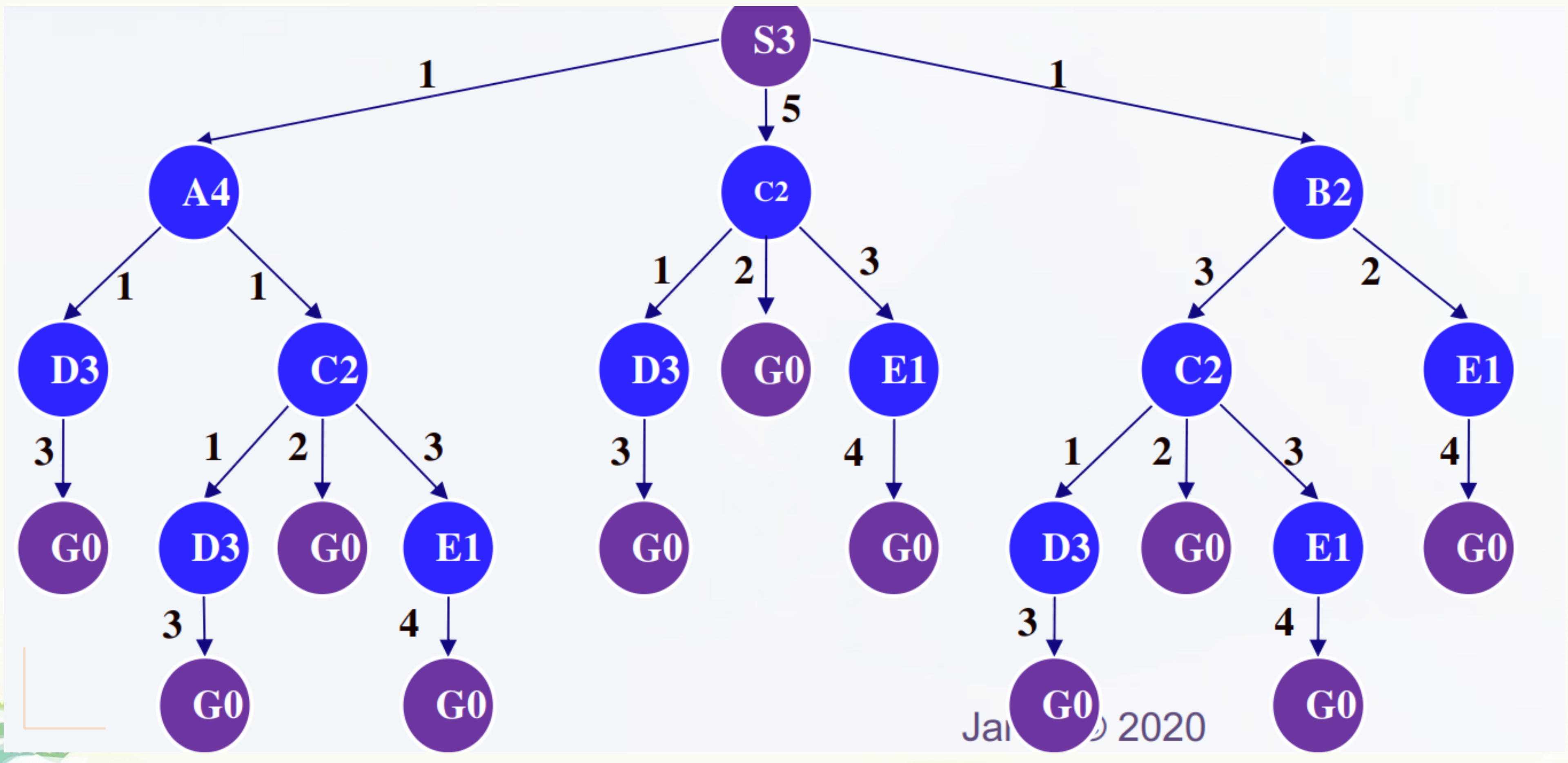


Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea



Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea


S3
A4
B2
C2
D3
E1
G0
1
1
5
1
1
3
1
2
3
3
2
4



S3
1
5
1
A4
C2
B2
1
1
1
2
3
3
2
D3
C2
D3
G0
E1
C2
E1
3
1
2
3
3
4
1
2
3
4
G0
D3
G0
E1
G0
G0
D3
G0
E1
G0
3
4
3
4
G0
G0
G0
G0

Searching Strategies

Uninformed Search	Informed Search
– breadth-first – uniform-cost search – depth-first – depth-limited search – iterative deepening – bi-directional search	– best-first search – search with heuristics – memory-bounded search – iterative improvement search

Uninformed Search (blind search)

- Number of steps, path cost unknown
- Agent knows when it reaches a goal

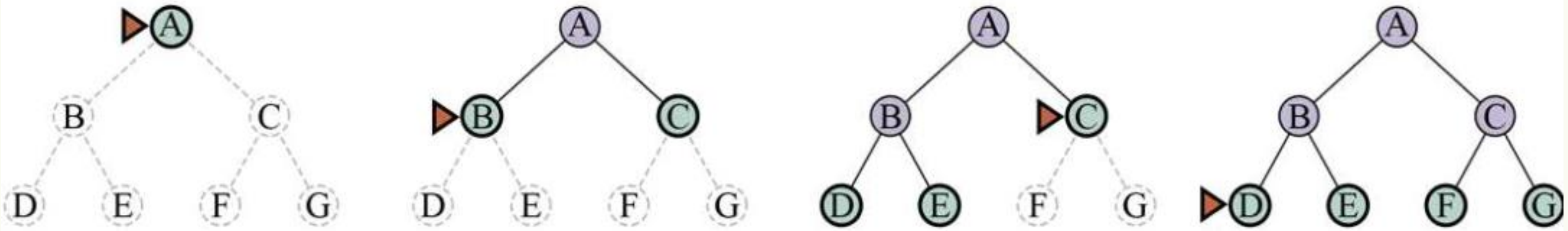
Informed Search (heuristic search)

- Agent has background information about the problem

Uninformed Search Strategies: Breadth-first search

- Which node from the frontier to expand next.
- Choose a node with minimum value of some evaluation function f(n).
- The root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.

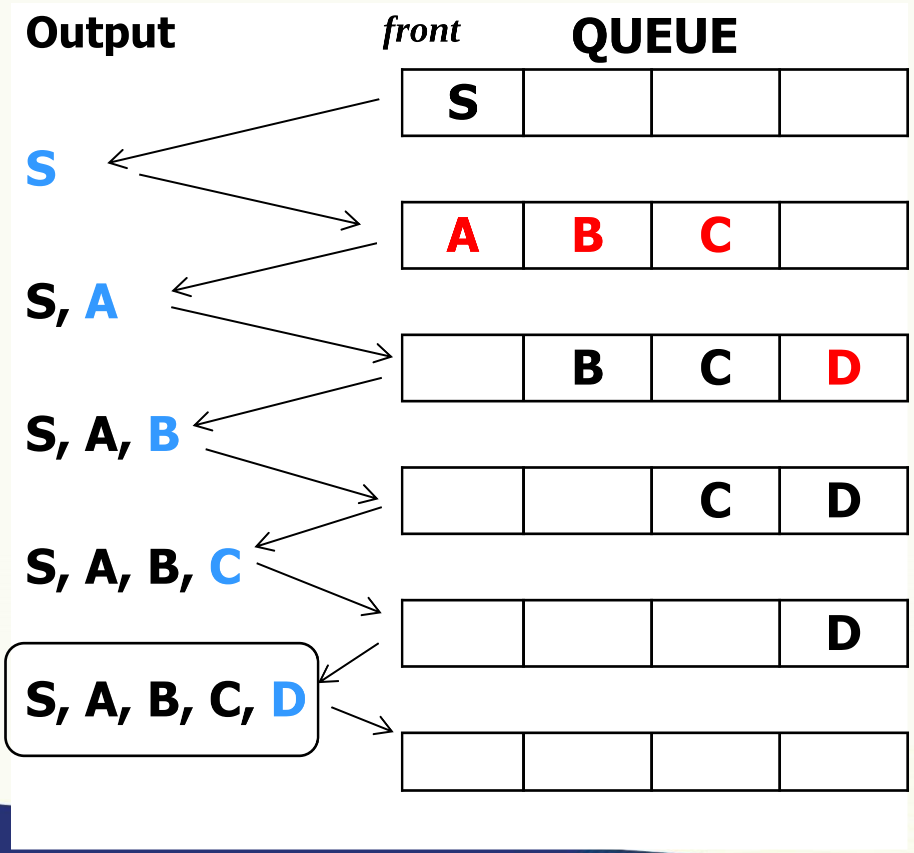
Breadth-first search



Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

Answer

- Enqueue the starting node **S** and mark it as visited.



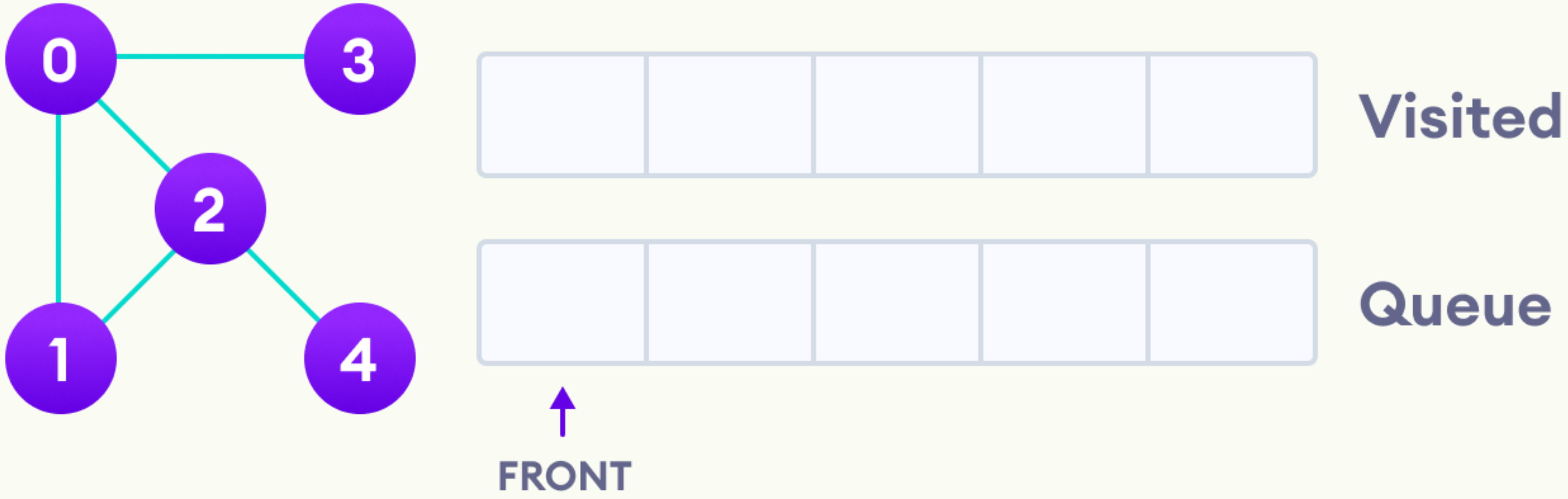
Nodes adjacent to S: A,B,C
(alphabetical order)

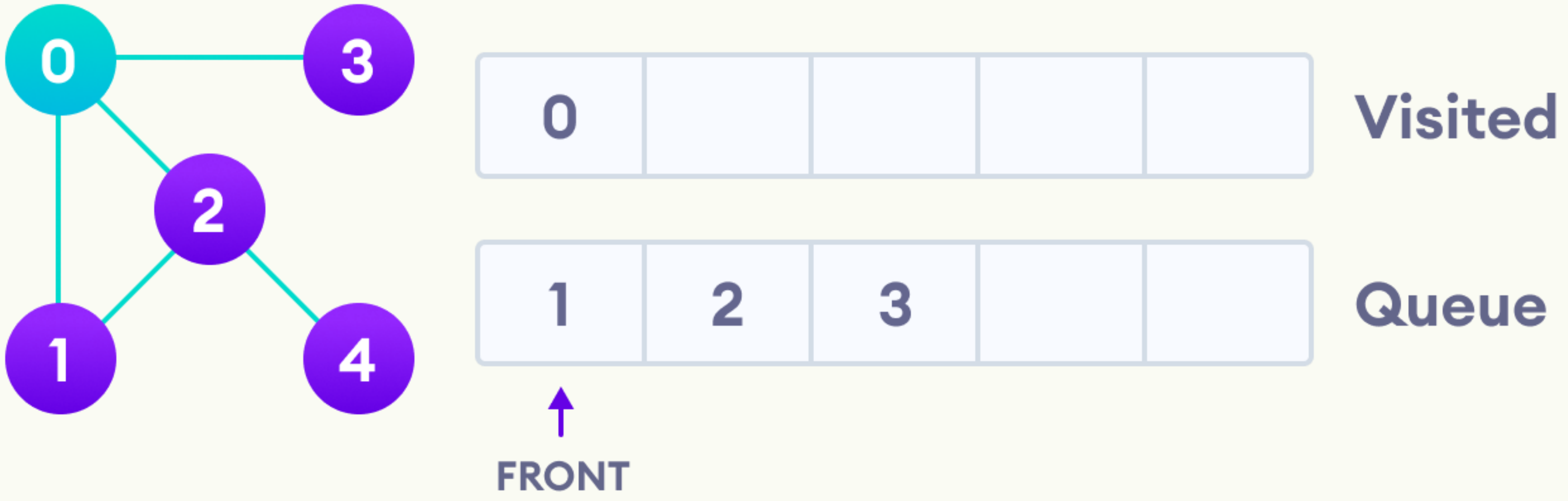
D is adjacent to A

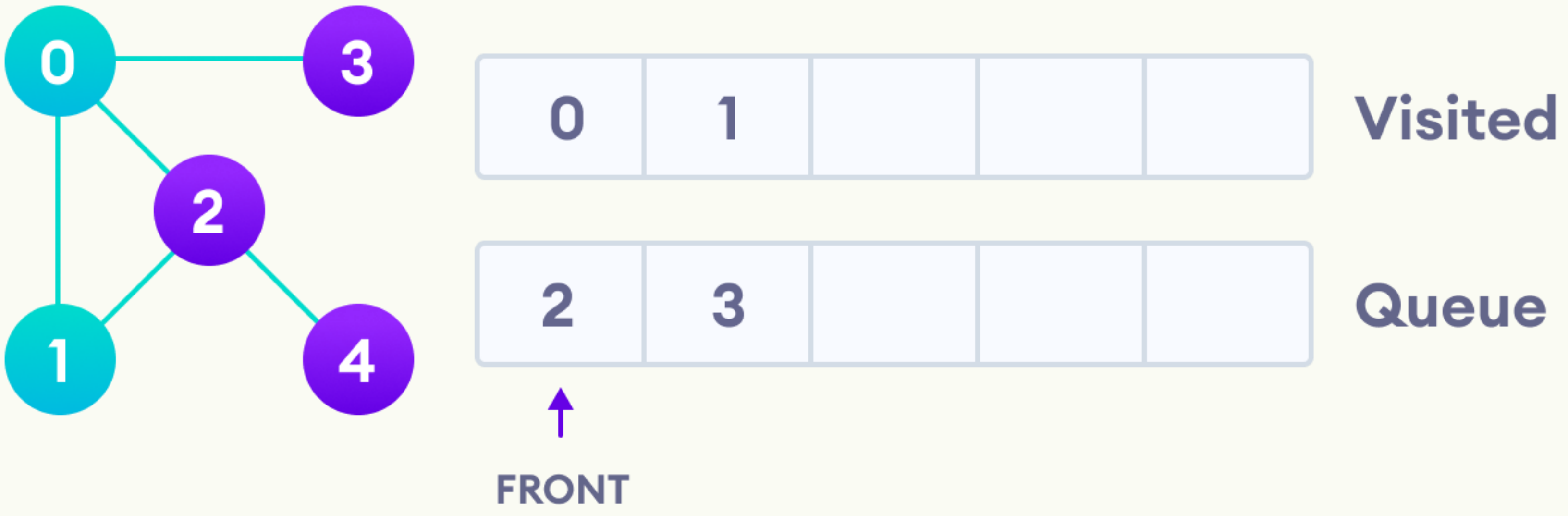
No adjacent to B

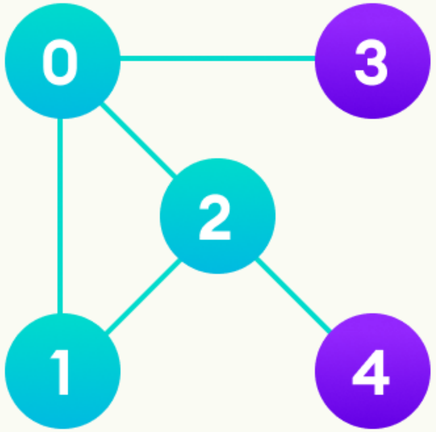
No adjacent to C

No adjacent to D


0
3
2
1
4
Visited
Queue
FRONT

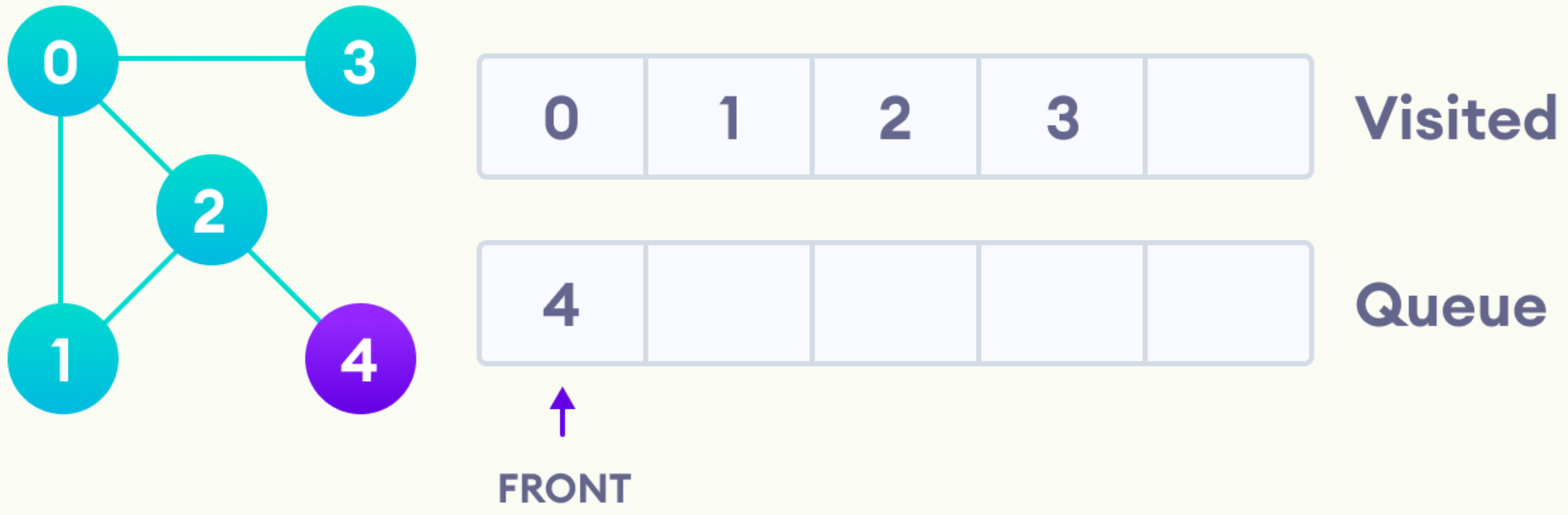

0
3
2
1
4
0
Visited
1
2
3
Queue
FRONT

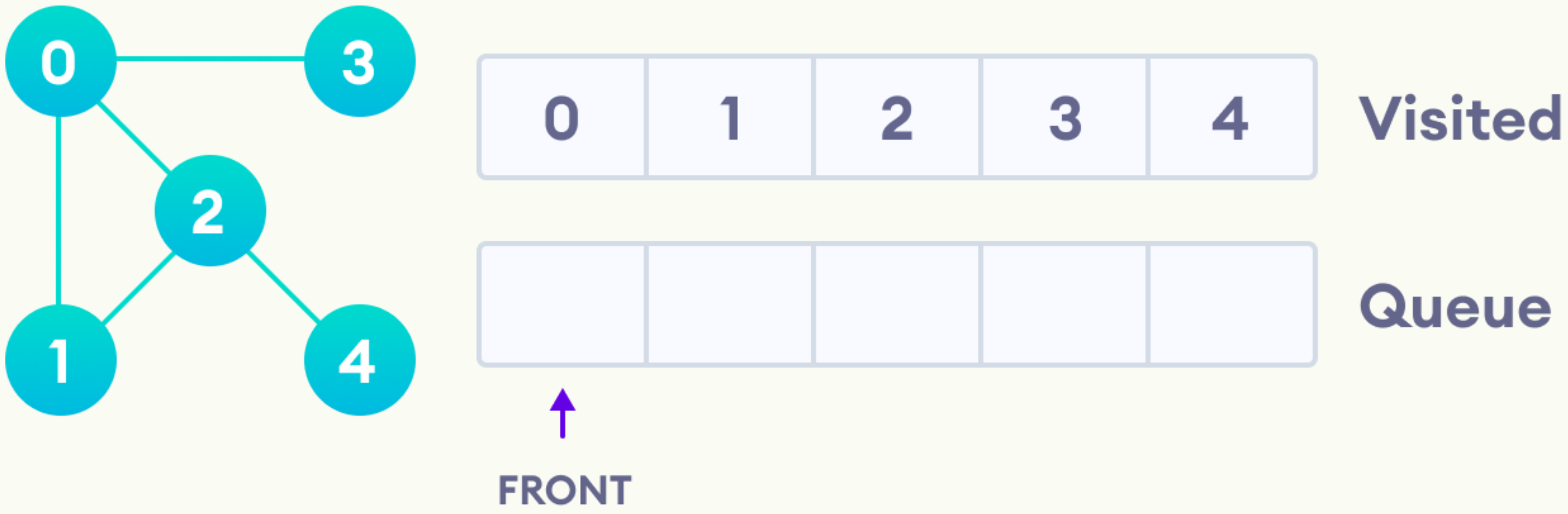

0
3
2
1
4
0 1
Visited
2 3
Queue
FRONT



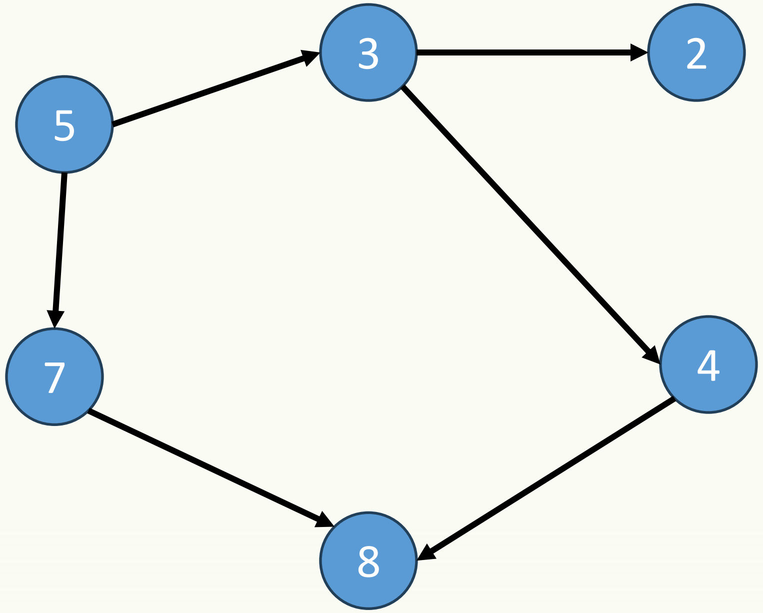
0	1	2			Visited
3	4				Queue

↑ FRONT


0
3
2
1
4
0
1
2
3
Visited
4
Queue
FRONT


0
3
2
1
4
0 1 2 3 4
Visited
Queue
FRONT

Write python code for BFS to traversal the following network



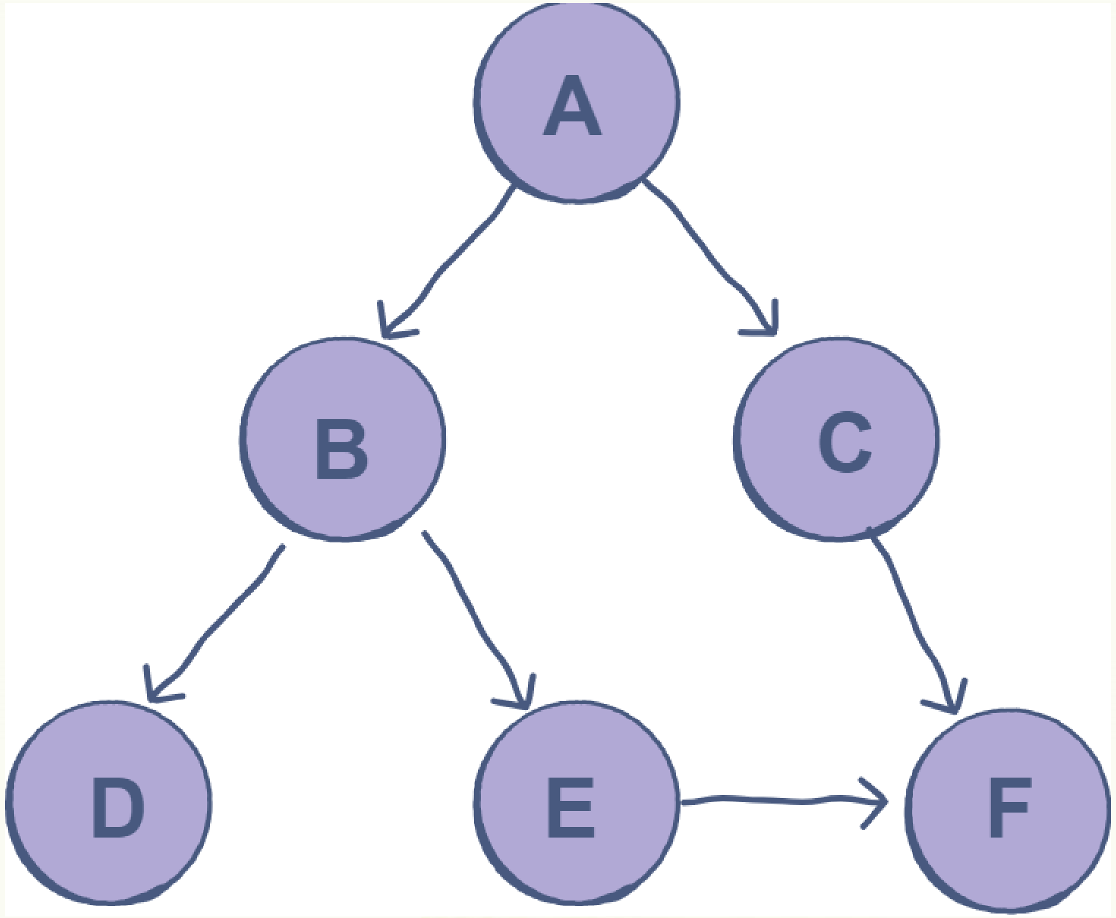
Code

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
visited = [] # List for visited nodes.
queue = []     #Initialize a queue
```

```
def bfs(visited, graph, node):  # Function for BFS
    visited.append(node)  # Mark the current node as visited
    queue.append(node)  # Add the current node to the queue
    while queue:  # Iterate until the queue is empty
        m = queue.pop(0)  # Retrieve and remove the first element from the queue
        print(m, end=" ")  # Print the value of the current node
        for neighbour in graph[m]:  # Iterate over the neighbors of the current node
            if neighbour not in visited:  # Check if the neighbor has not been visited
                visited.append(neighbour)  # Mark the neighbor as visited
                queue.append(neighbour)  # Add the neighbor to the queue

# Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')  # Function call with the initial visited list, graph, and
starting node
```

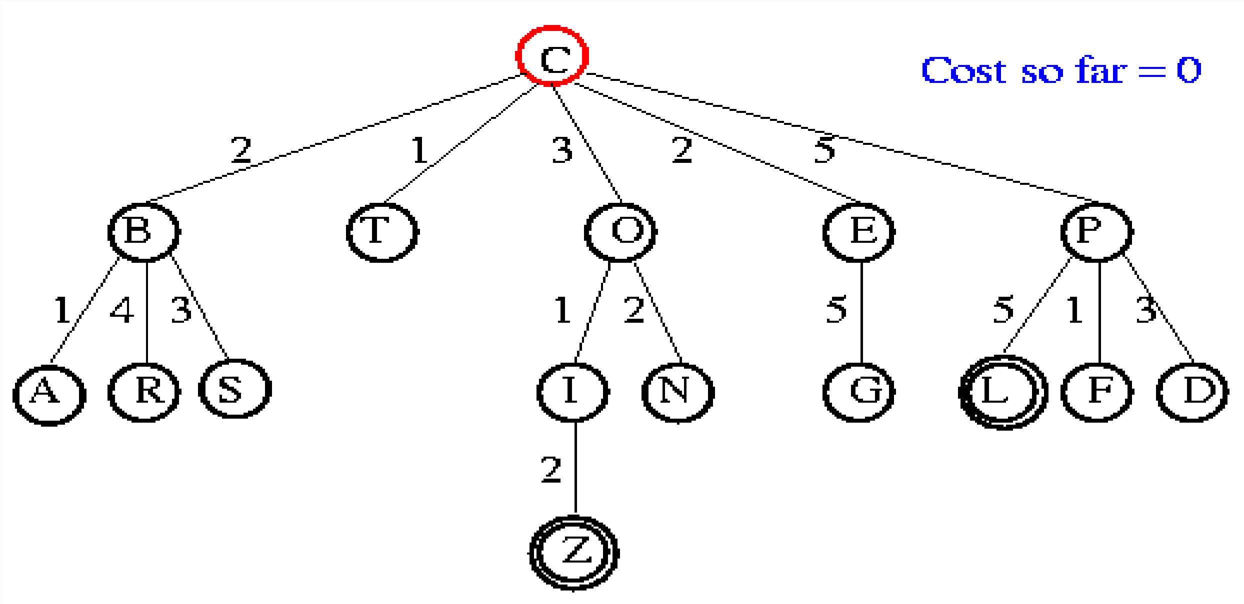
Write BFS code for this tree



Uniform-Cost-First

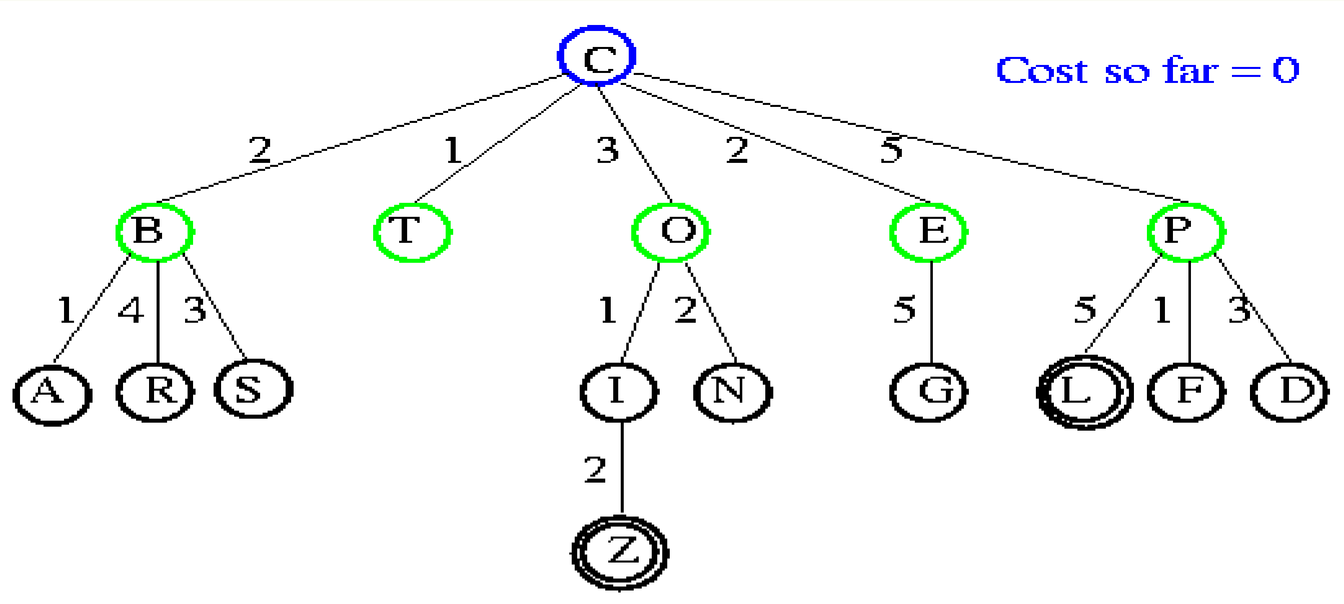
- Visits the next node which has the least total cost from the root, until a goal state is reached.
- – Similar to BREADTH-FIRST, but with an evaluation of the cost for each reachable node.
- g(n) = path cost(n) = sum of individual edge costs to reach the current node.

UCS Example



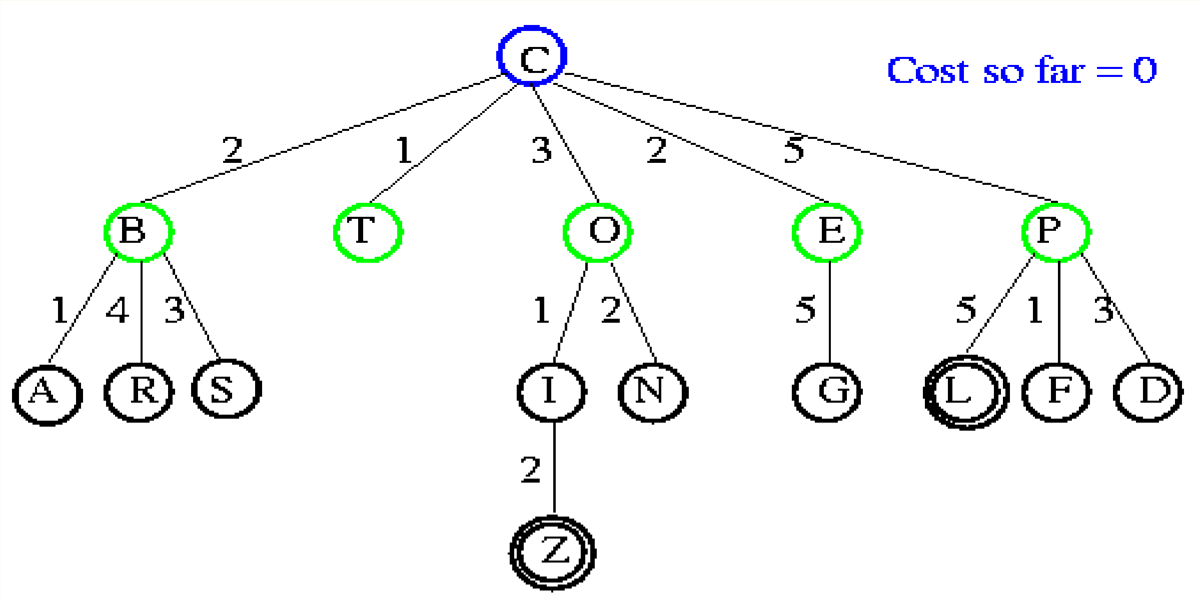
Open list: C

UCS Example



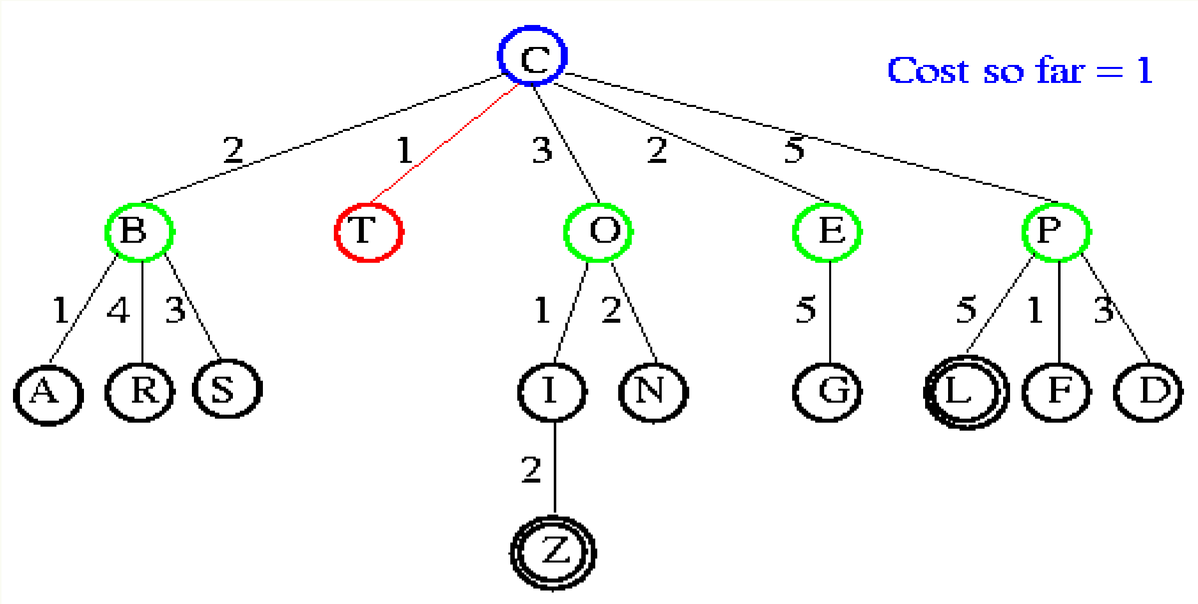
Open list: B(2) T(1) O(3) E(2) P(5)

UCS Example



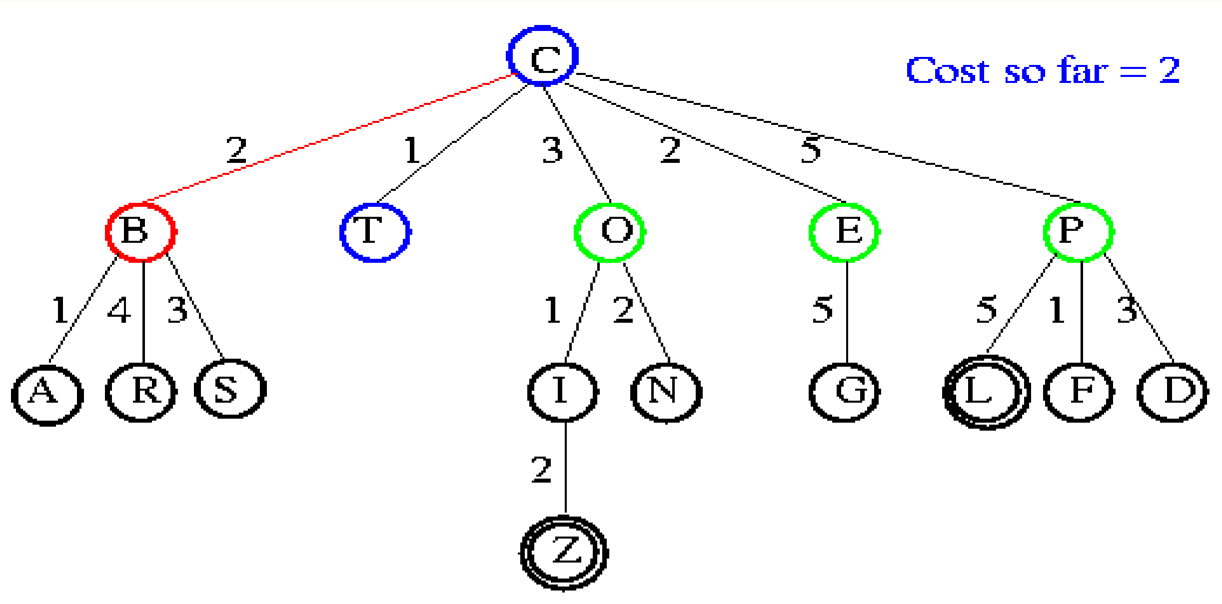
Open list: T(1) B(2) E(2) O(3) P(5)

UCS Example



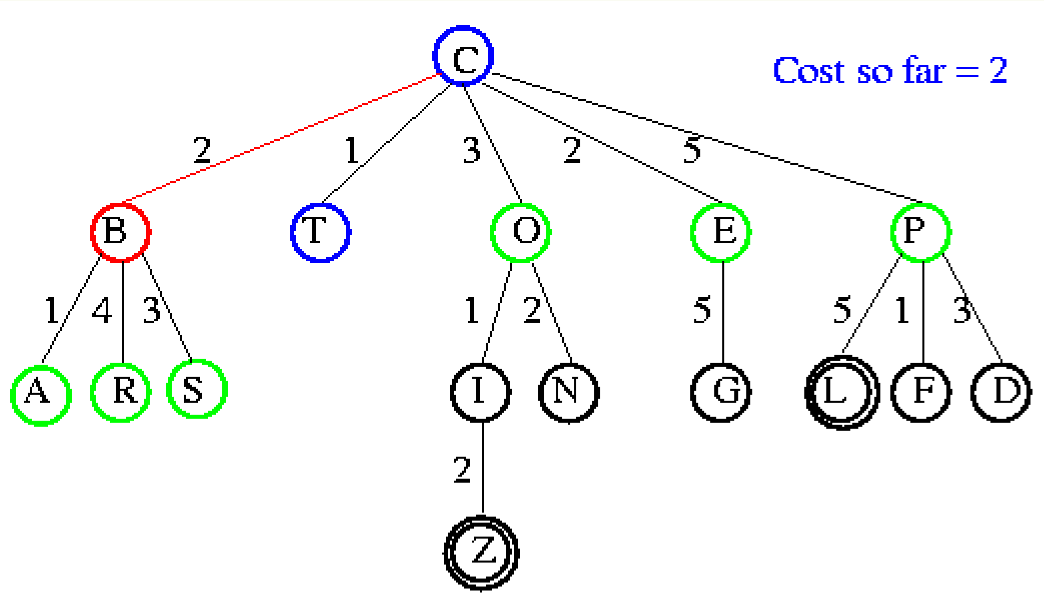
Open list: B(2) E(2) O(3) P(5)

UCS Example



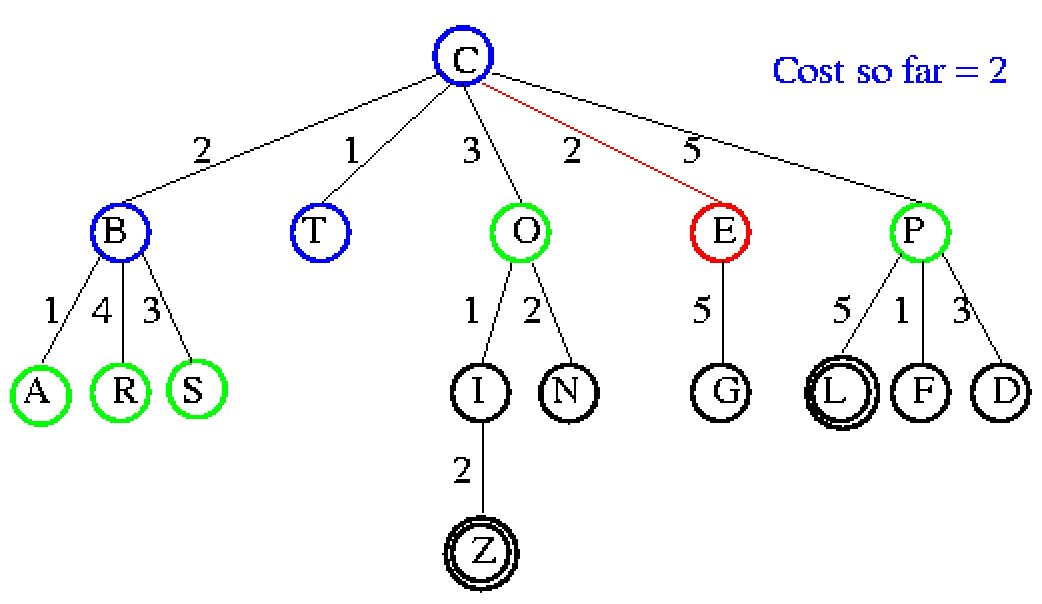
Open list: E(2) O(3) P(5)

UCS Example



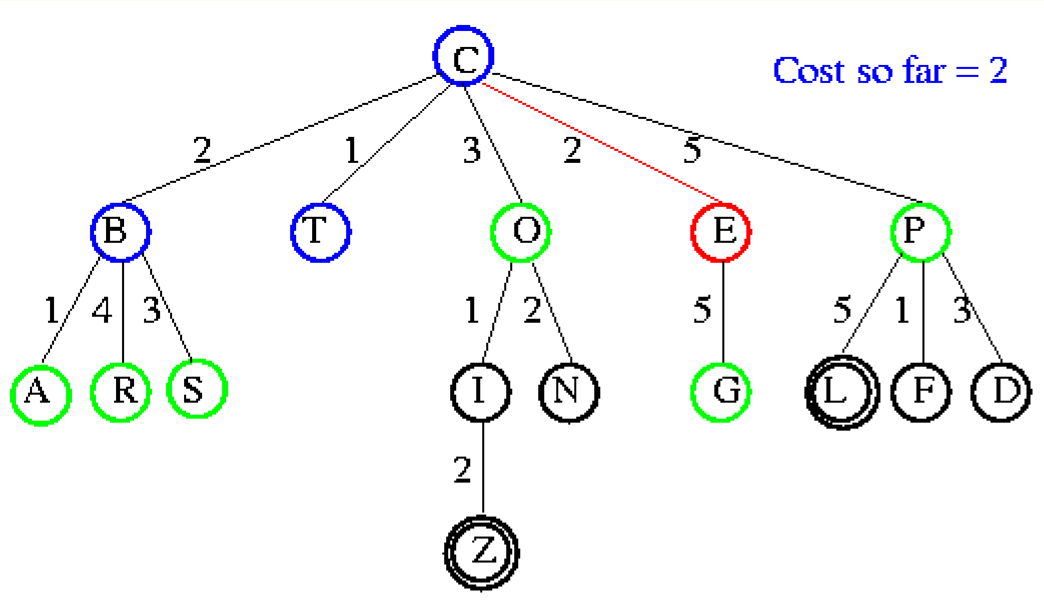
Open list: E(2) O(3) A(3) S(5) P(5) R(6)

UCS Example



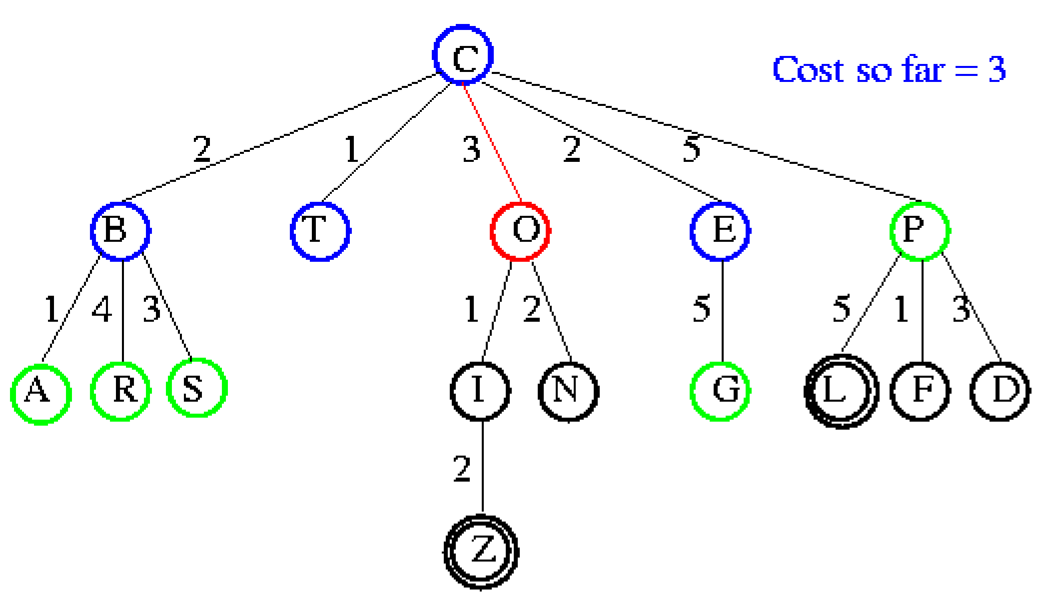
Open list: O(3) A(3) S(5) P(5) R(6)

UCS Example



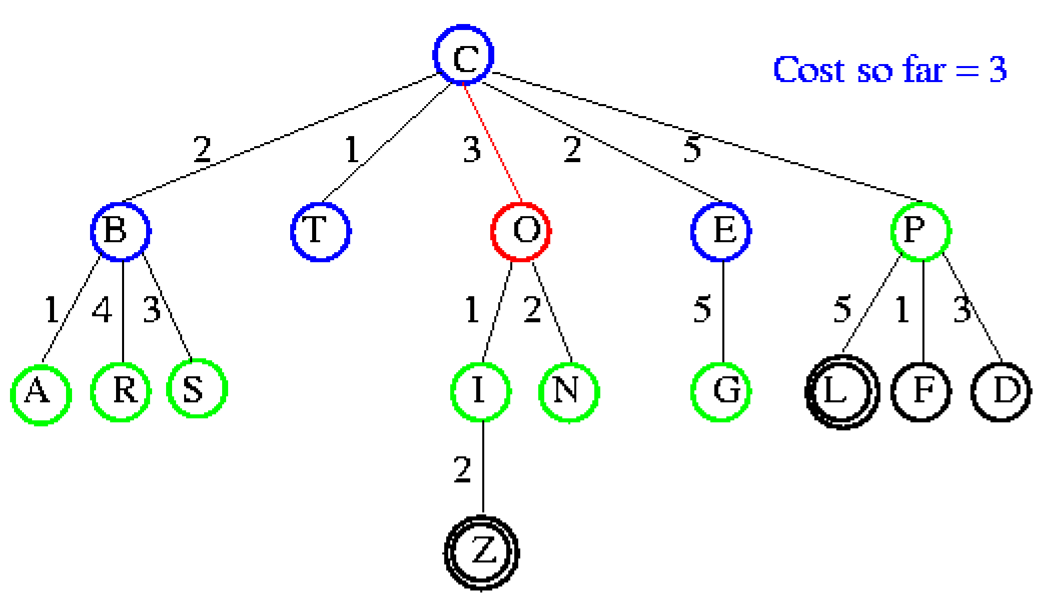
Open list: O(3) A(3) S(5) P(5) R(6) G(10)

UCS Example



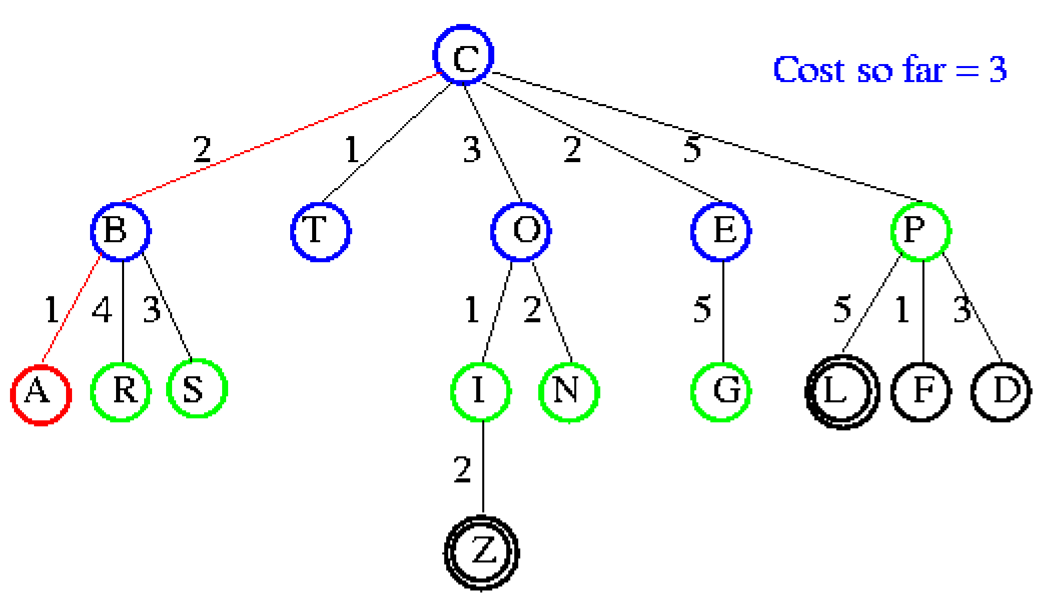
Open list: A(3) S(5) P(5) R(6) G(10)

UCS Example



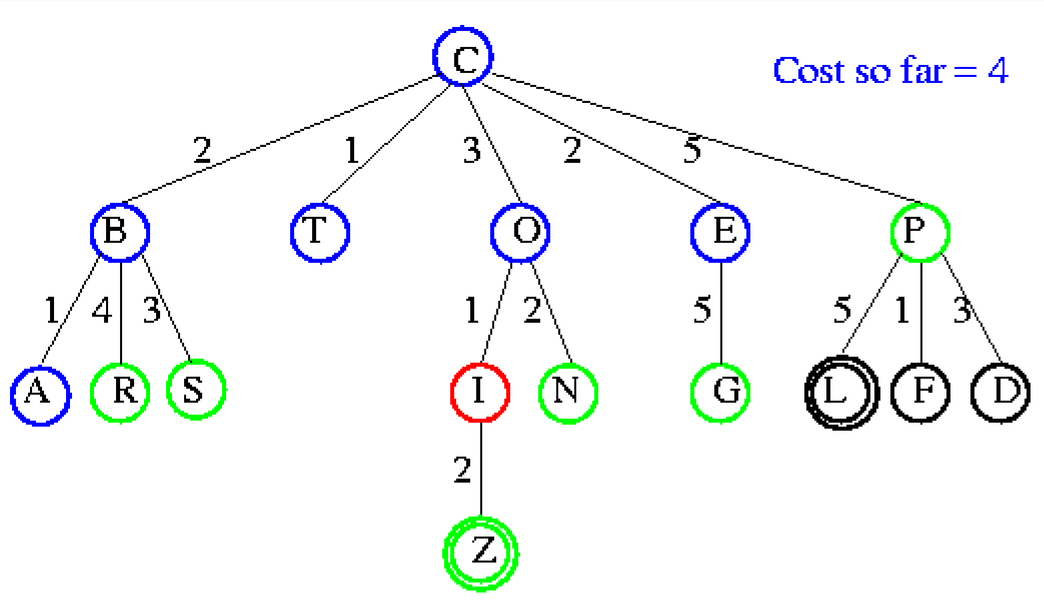
Open list: A(3) I(4) S(5) N(5) P(5) R(6) G(10)

UCS Example



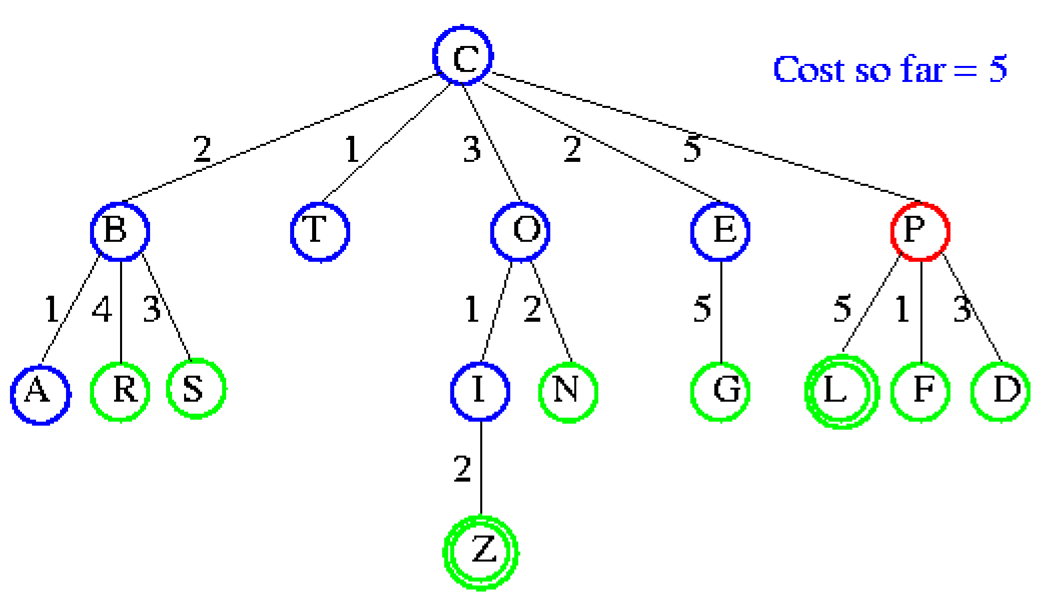
Open list: I(4) P(5) S(5) N(5) R(6) G(10)

UCS Example



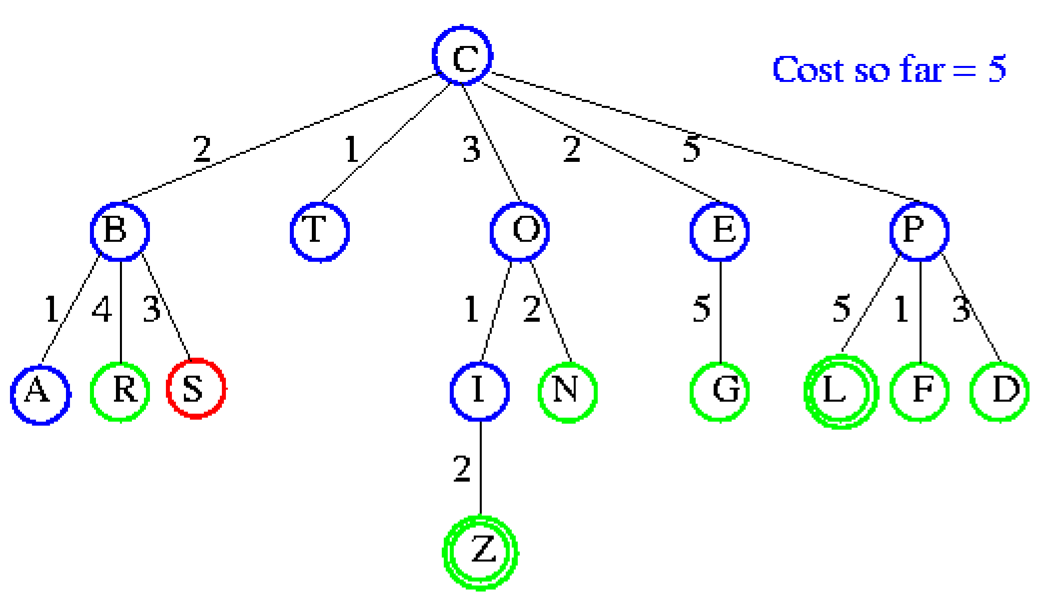
Open list: P(5) S(5) N(5) R(6) Z(6) G(10)

UCS Example



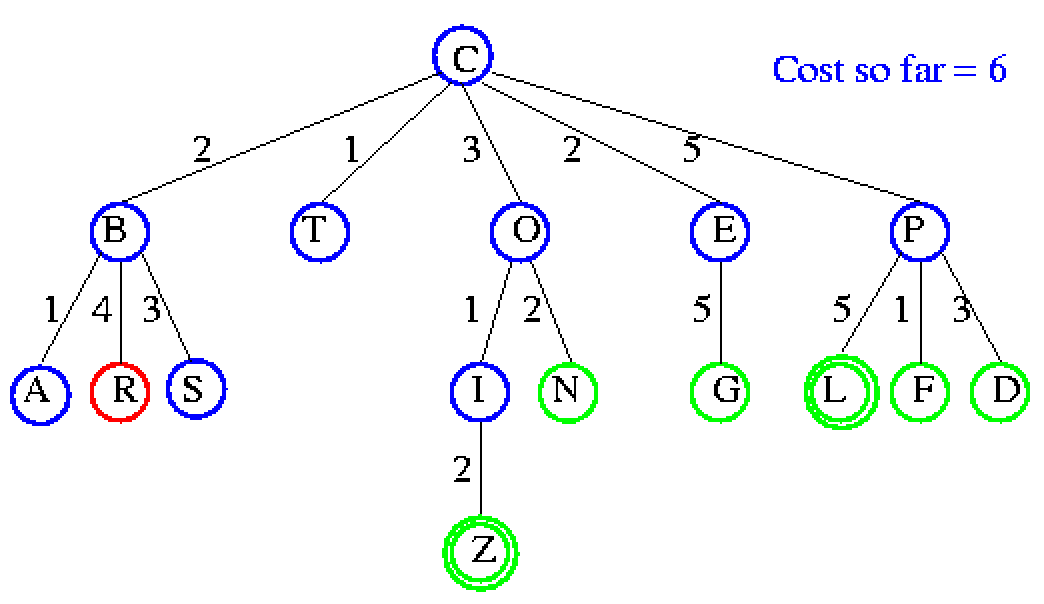
Open list: S(5) N(5) R(6) Z(6) F(6) D(8) G(10) L(10)

UCS Example



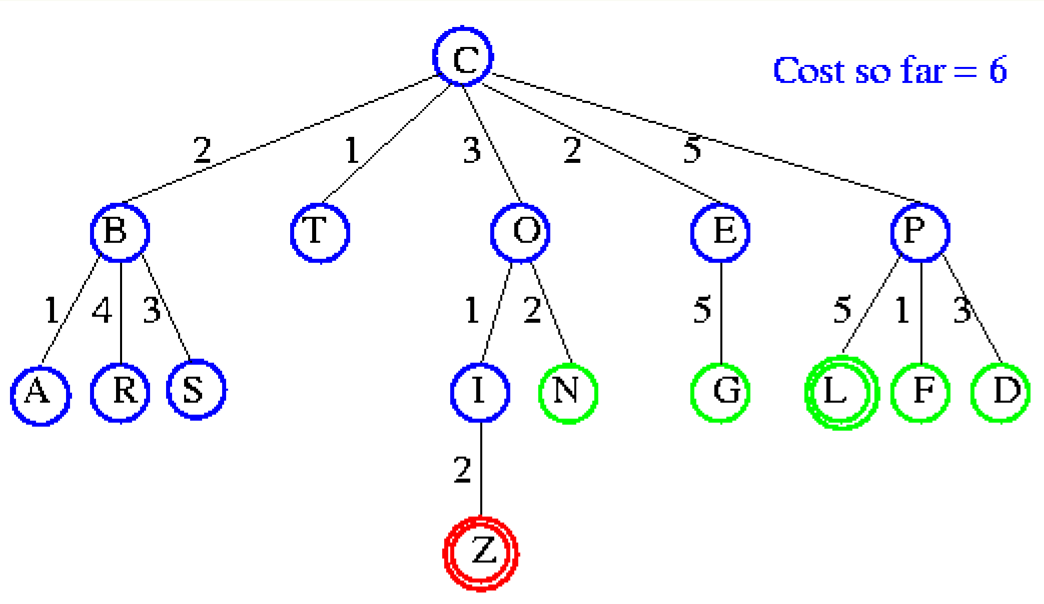
Open list: N(5) R(6) Z(6) F(6) D(8) G(10) L(10)

UCS Example



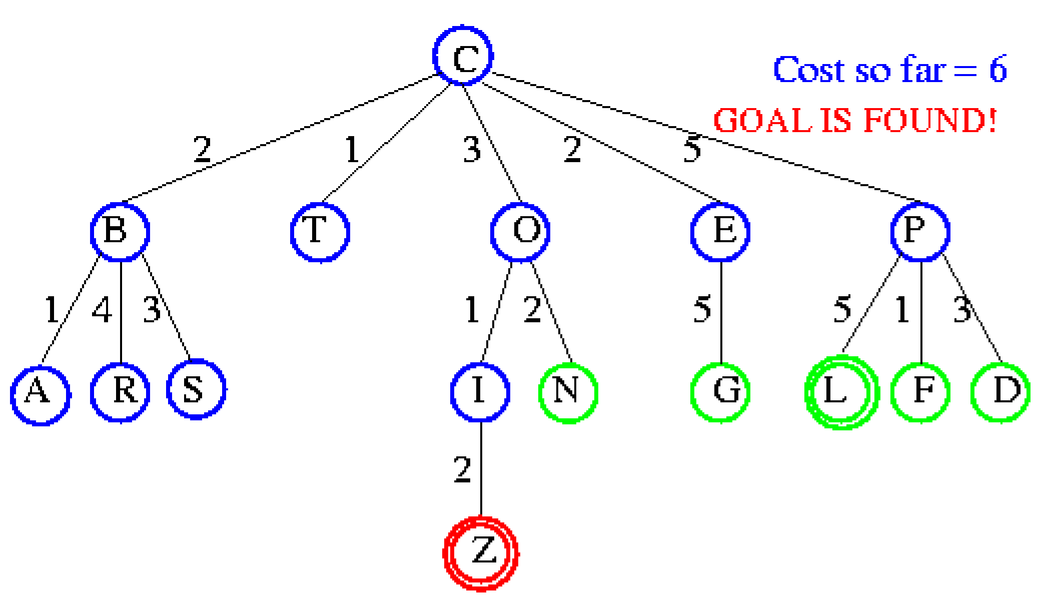
Open list: Z(6) F(6) D(8) G(10) L(10)

UCS Example

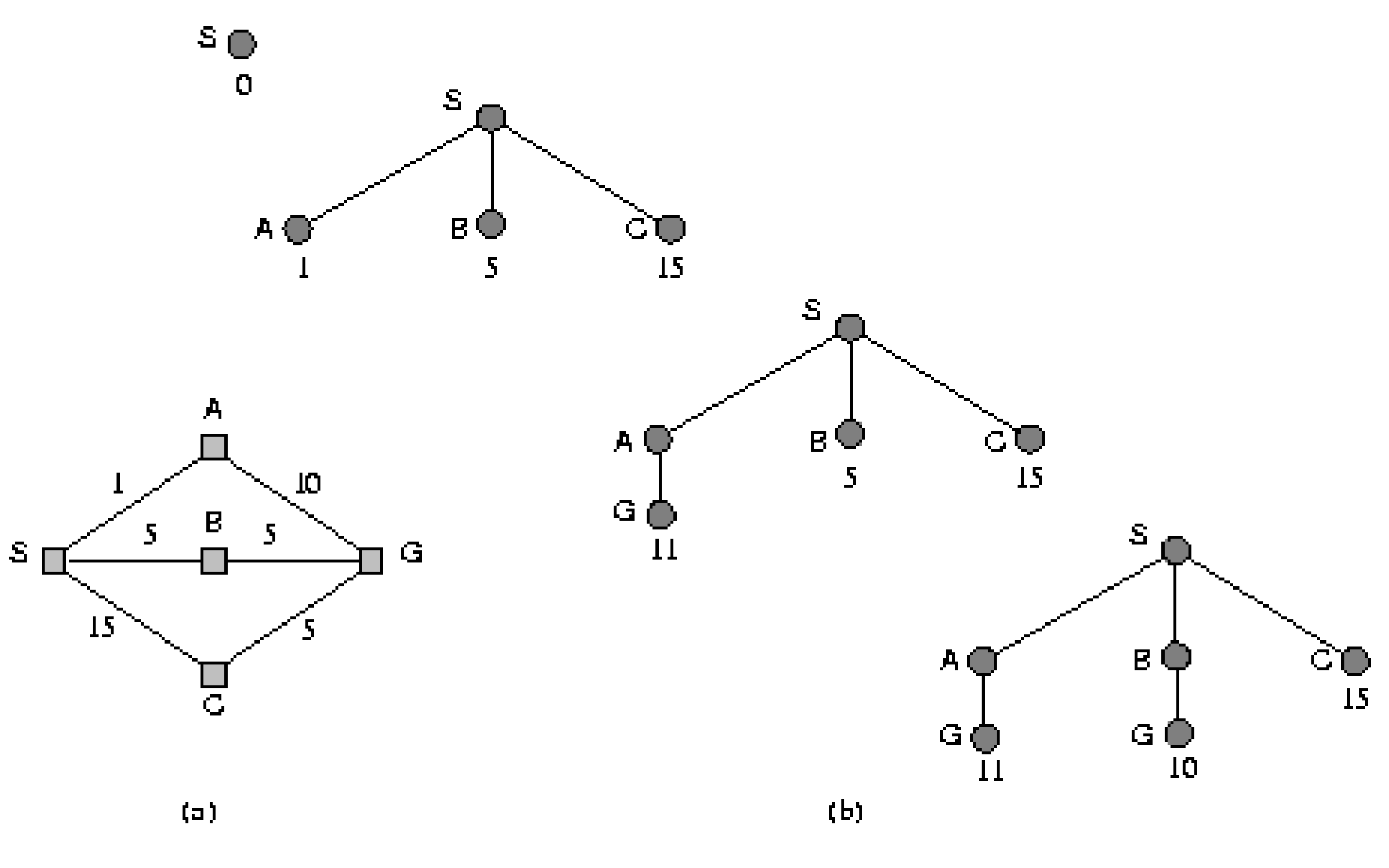


Open list: F(6) D(8) G(10) L(10)

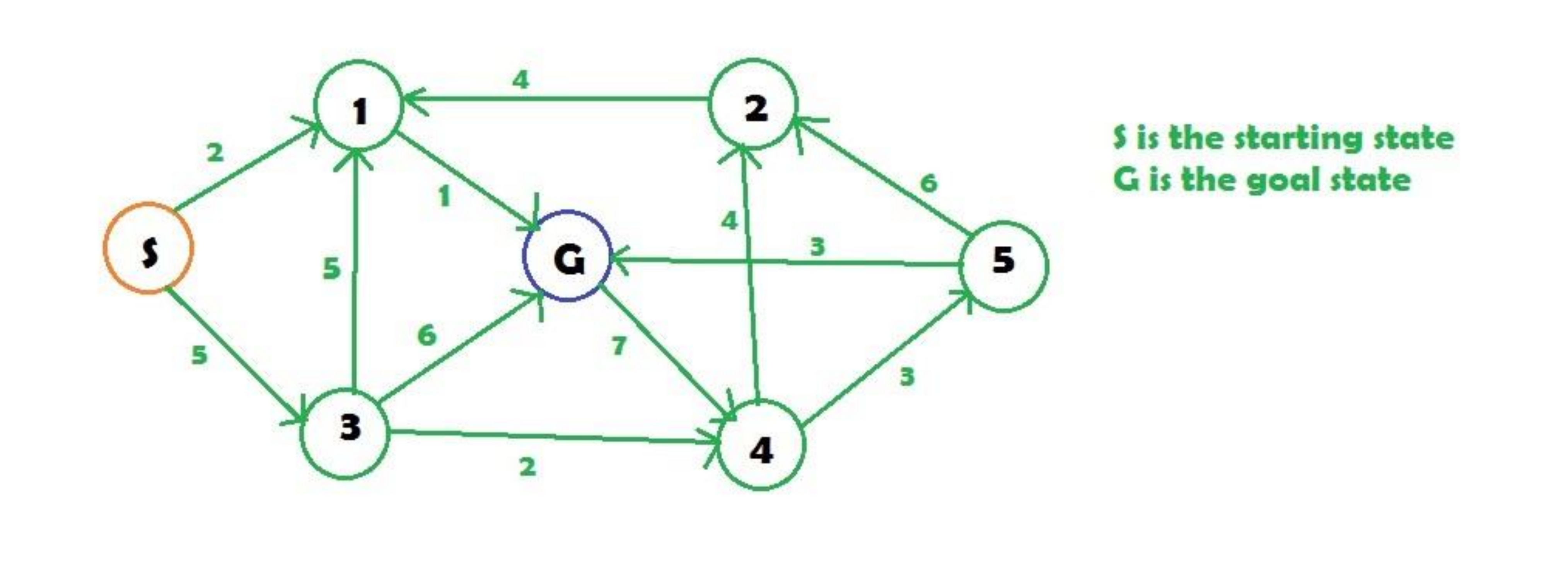
UCS Example


C
B
T
O
E
P
A
R
S
I
N
G
L
F
D
Z
2
1
3
2
5
1
4
3
1
2
5
5
1
3
2
Cost so far = 6
GOAL IS FOUND!

UCS Example

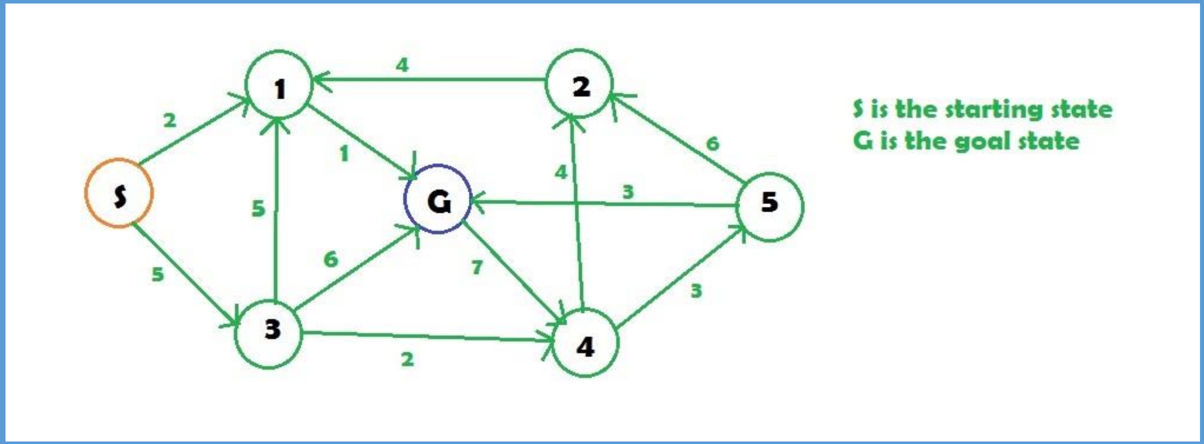


UCS Example



1
2
S
G
5
3
4
4
2
1
5
6
5
6
7
4
3
3
2
S is the starting state
G is the goal state

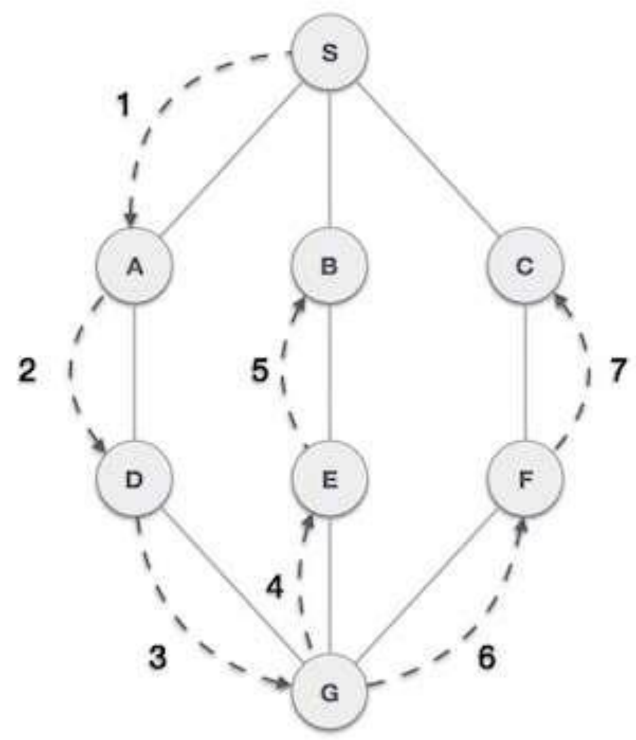
```python
# main function
if __name__ == '__main__':
                # create the graph
                graph,cost = [[] for i in range(8)],{}
                # add edge
                graph[0].append(1)
                graph[0].append(3)
                graph[3].append(1)
                graph[3].append(6)
                graph[3].append(4)
                graph[1].append(6)
                graph[4].append(2)
                graph[4].append(5)
                graph[2].append(1)
                graph[5].append(2)
                graph[5].append(6)
                graph[6].append(4)
                # add the cost
                cost[(0, 1)] = 2
                cost[(0, 3)] = 5
                cost[(1, 6)] = 1
                cost[(3, 1)] = 5
                cost[(3, 6)] = 6
                cost[(3, 4)] = 2
                cost[(2, 1)] = 4
                cost[(4, 2)] = 4
                cost[(4, 5)] = 3
                cost[(5, 2)] = 6
                cost[(5, 6)] = 3
                cost[(6, 4)] = 7
                # goal state
                goal = []
                # set the goal
                # there can be multiple goal states
                goal.append(6)
                # get the answer
                answer = uniform_cost_search(goal, 0)
                # print the answer
                print("Minimum cost from 0 to 6 is = ",answer[0])
```



```python
# Python3 implementation of above approach
# returns the minimum cost in a vector( if
# there are multiple goal states)
def uniform_cost_search(goal, start):
        # minimum cost upto
        # goal state from starting
        global graph,cost
        answer = []
        # create a priority queue
        queue = []
        # set the answer vector to max value
        for i in range(len(goal)):
                answer.append(10**8)
        # insert the starting index
        queue.append([0, start])
        # map to store visited node
        visited = {}
        # count
        count = 0
        # while the queue is not empty
        while (len(queue) > 0):
          # get the top element of the
          queue = sorted(queue)
          p = queue[-1]
          # pop the element
          del queue[-1]
          # get the original value
          p[0] *= -1
                        # get the position
                        index = goal.index(p[1])
                        # if a new goal is reached
                        if (answer[index] == 10**8):
                                count += 1
                        # if the cost is less
                        if (answer[index] > p[0]):
                                answer[index] = p[0]
                        # pop the element
                        del queue[-1]
                        queue = sorted(queue)
                        if (count == len(goal)):
                                return answer
                # check for the non visited nodes and which are adjacent to present
node
                if (p[1] not in visited):
                        for i in range(len(graph[p[1]])):
                                # value is multiplied by -1 so that
                                # least priority is at the top
                                queue.append( [(p[0] + cost[(p[1], graph[p[1]][i])])* -1,
graph[p[1]][i]])
                # mark as visited
                visited[p[1]] = 1
        return answer
```

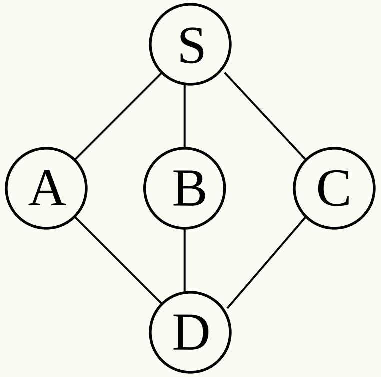
Depth-First Traversal

- DFS begins at some arbitrary vertex, exploring *as far as* possible down a branch before backtracking.
- For example, in the figure shown: DFS traverses **S, A, D, G, E, B** before backtracking to **E** to **G** and then visiting **F** then **C**.
- Backtracking is implemented using a **stack** to return to the previous vertex to start a search, when a dead end is reached.



Example

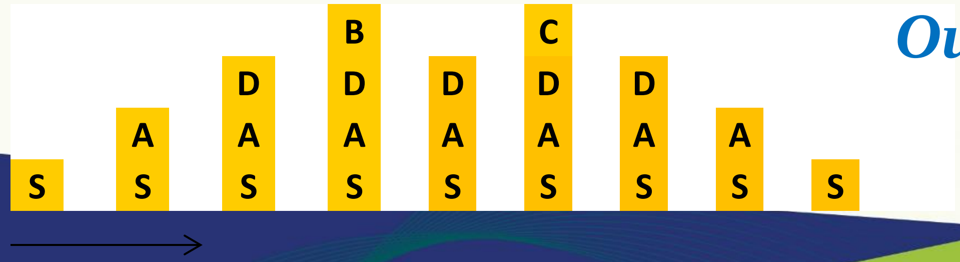
- Apply DFS algorithm to the following graph starting from node **S**. Show the contents of the stack.

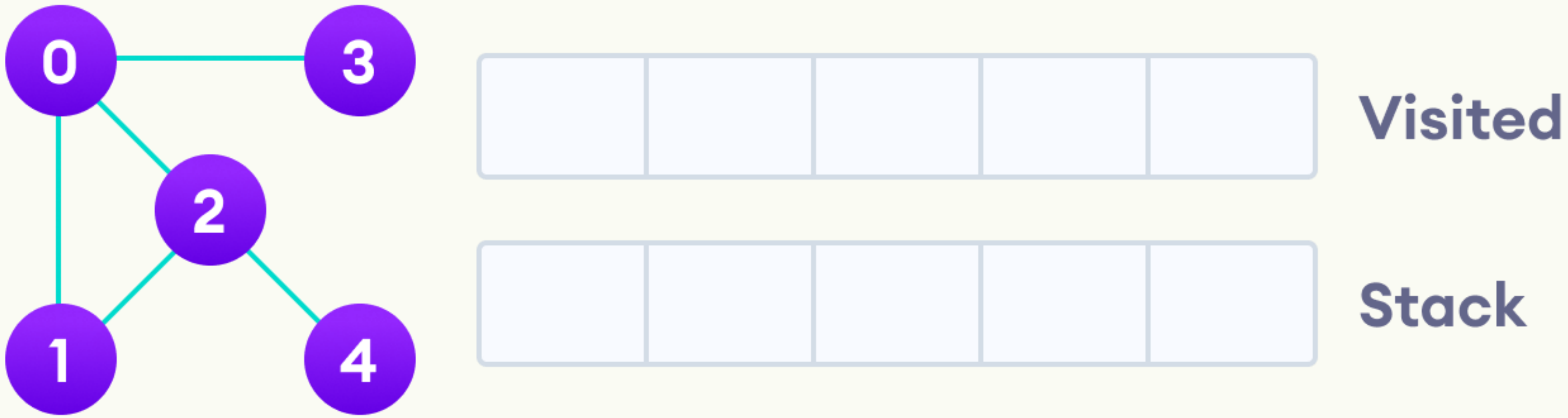


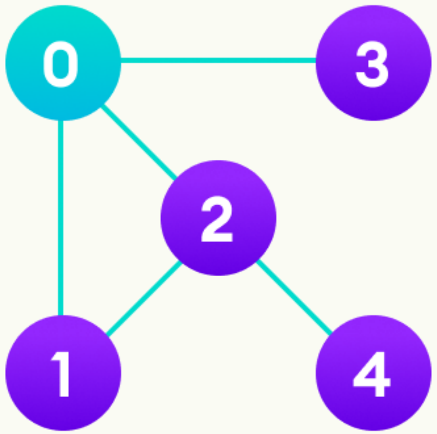
Stack is *EMPTY:* *End of Algorithm*

Output: *{S, A, D, B, C}.*

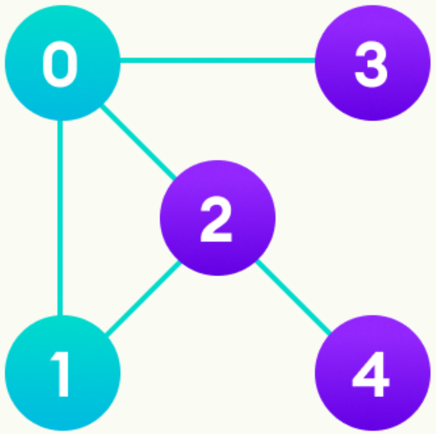
stack



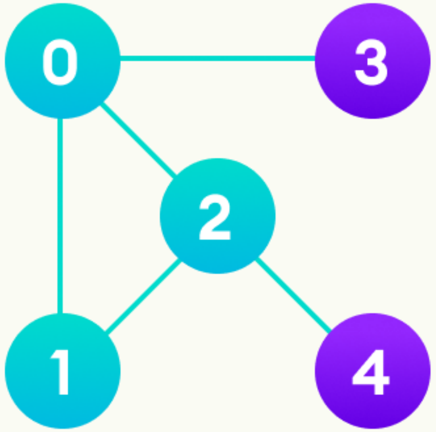

0
3
2
1
4
Visited
Stack



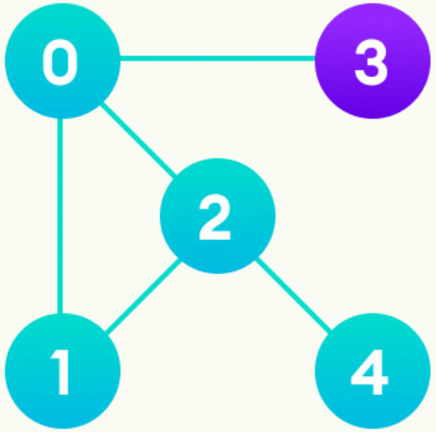
0					Visited
1	2	3			Stack



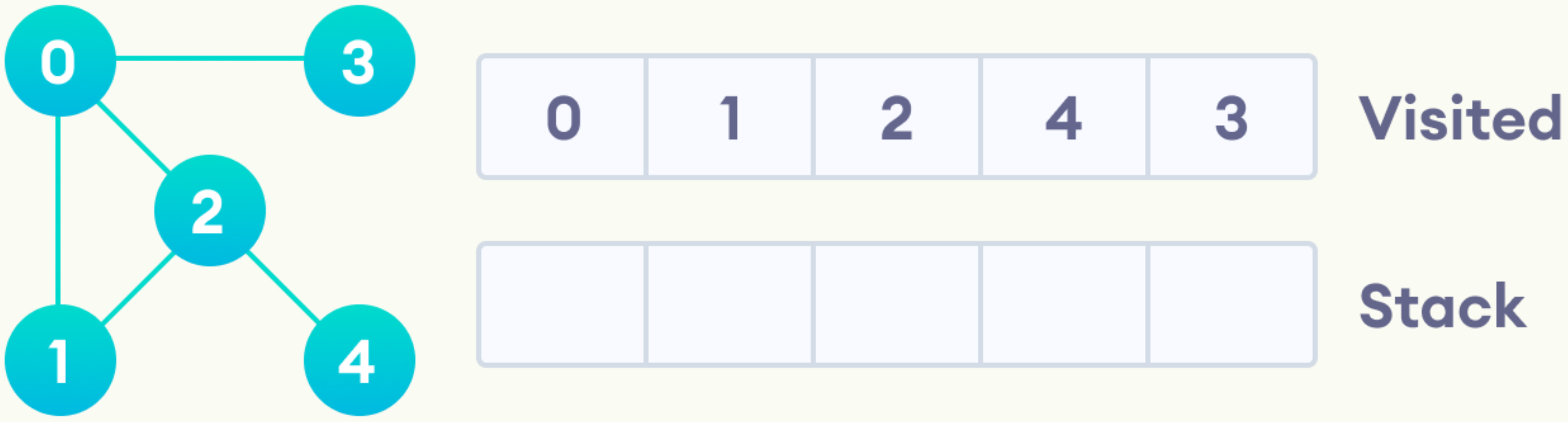
0	1				Visited
2	3				Stack

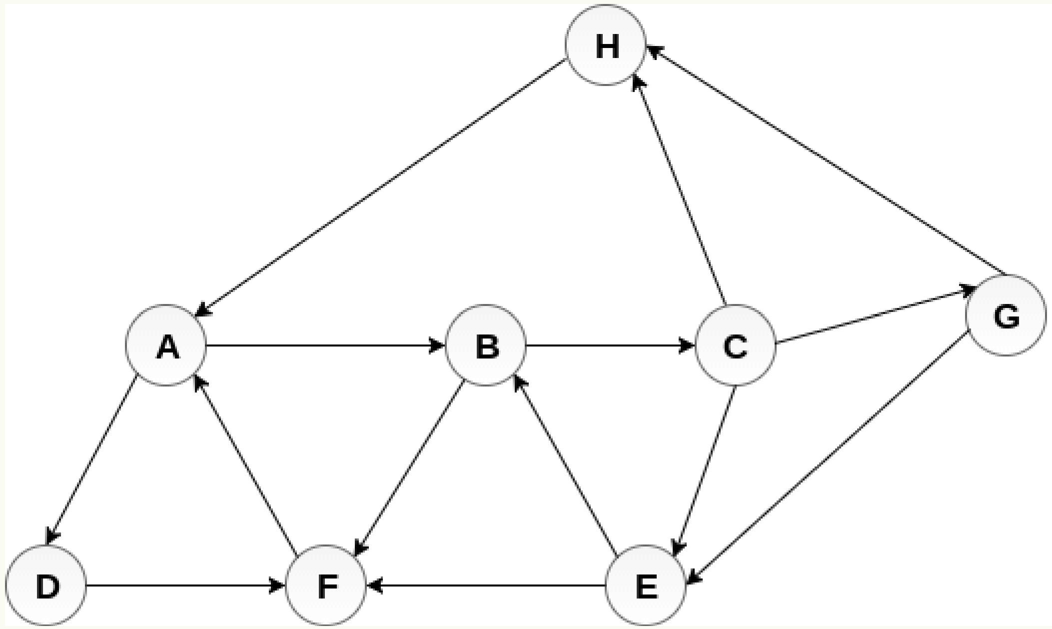


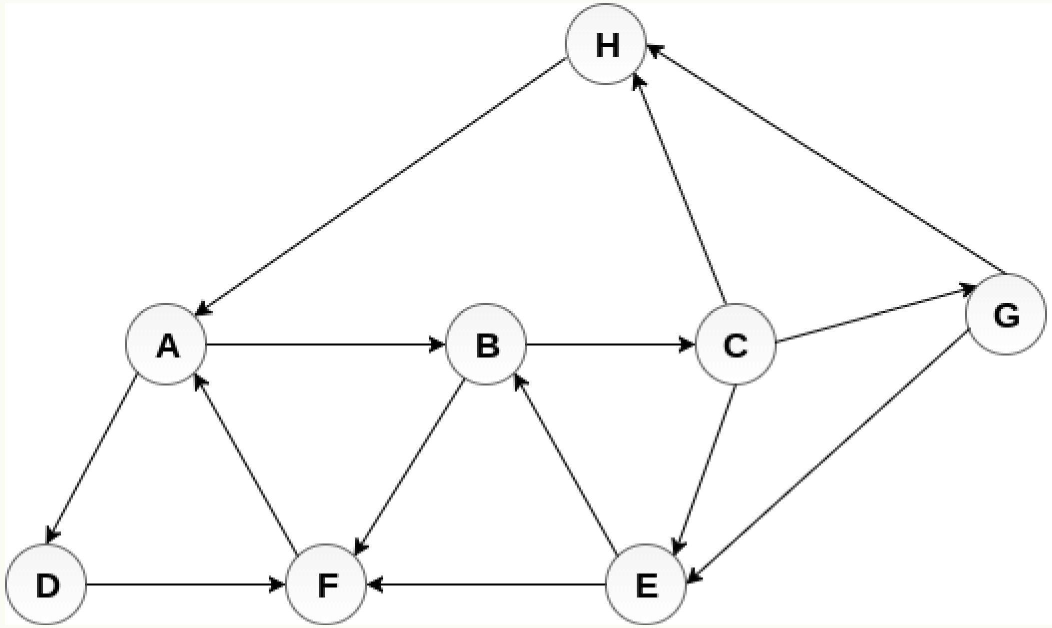
0	1	2			Visited
4	3				Stack



0	1	2	4		Visited
3					Stack

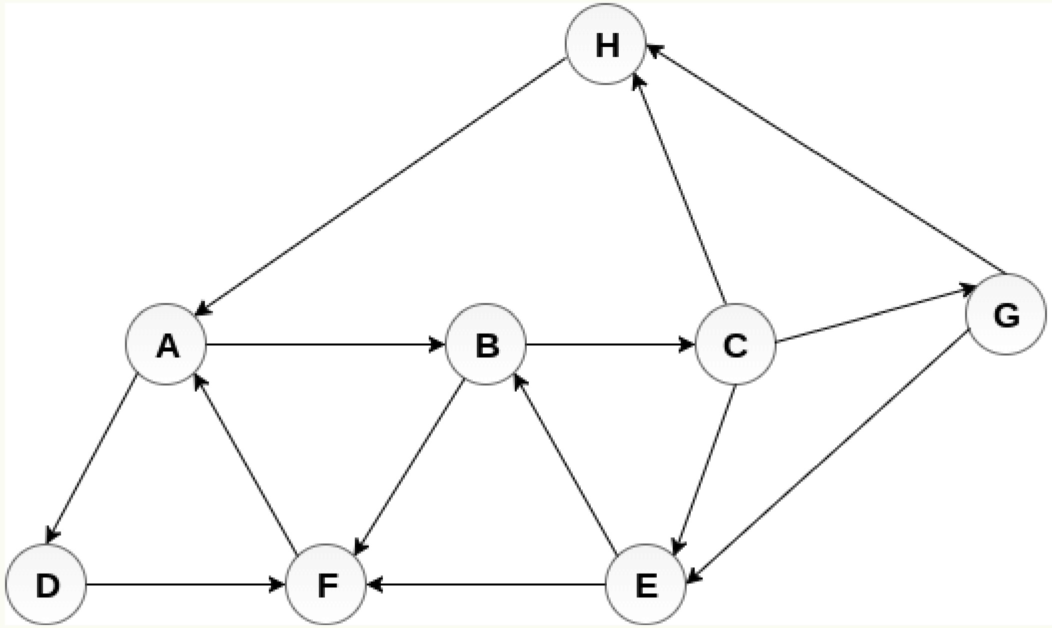

0
3
2
1
4
0 1 2 4 3
Visited
Stack


H
A
B
C
G
D
F
E



Adjacency Lists

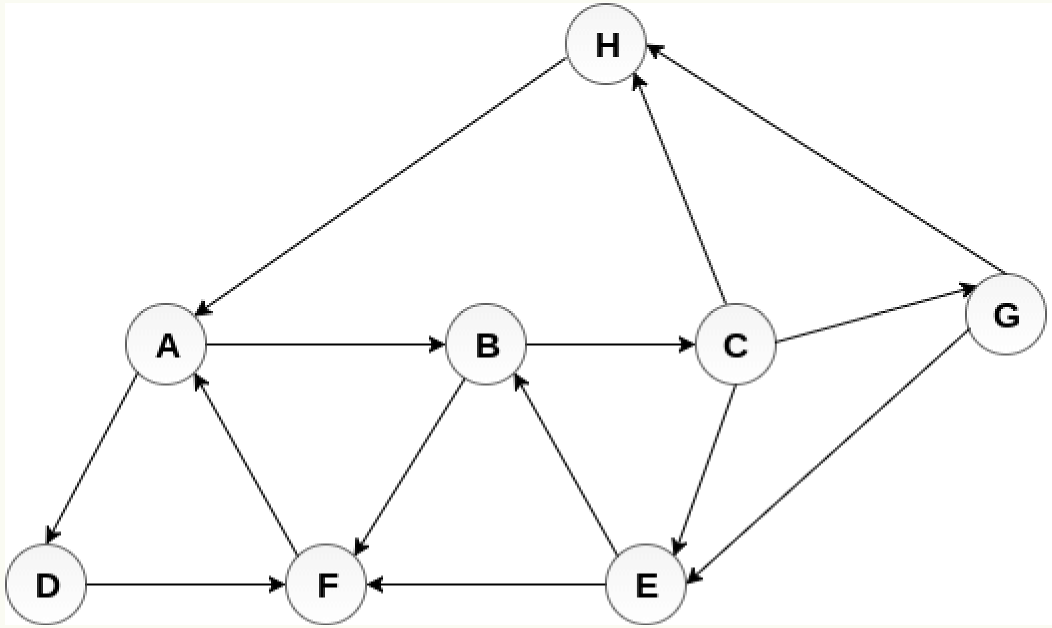
A : B, D



Adjacency Lists

A : B, D

B : C, F

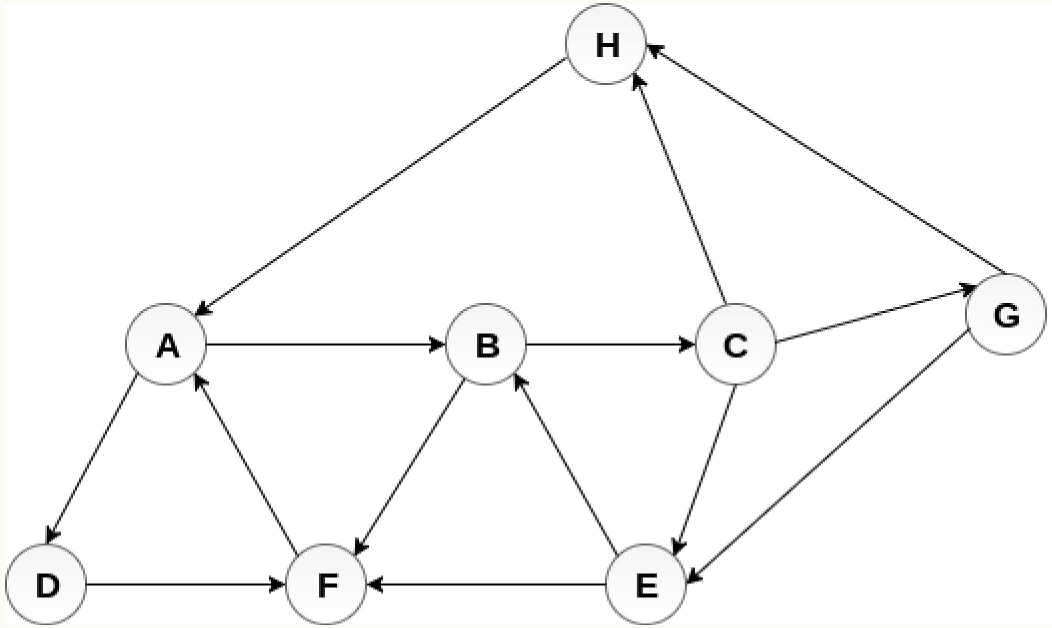


Adjacency Lists

A : B, D

B : C, F

C : E, G, H



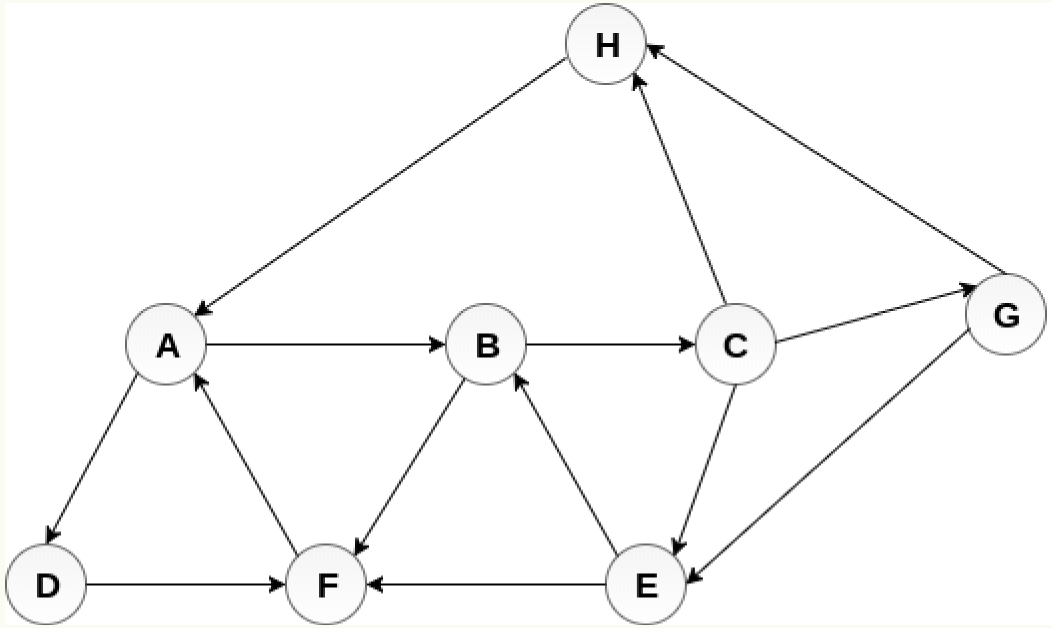
Adjacency Lists

A : B, D

B : C, F

C : E, G, H

G : E, H



Adjacency Lists

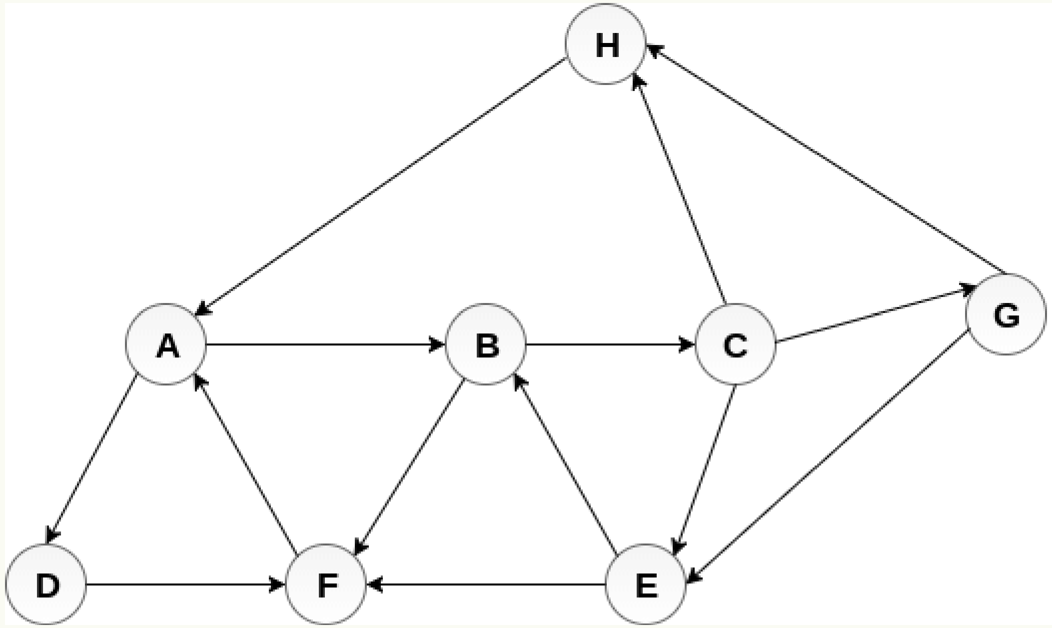
A : B, D

B : C, F

C : E, G, H

G : E, H

E : B, F



Adjacency Lists

A : B, D

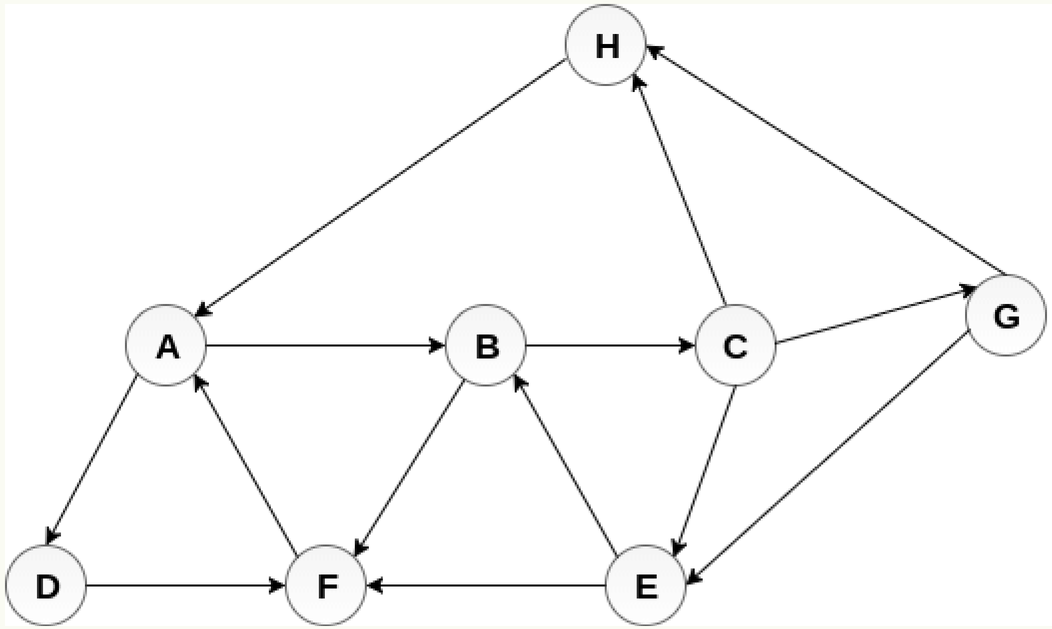
B : C, F

C : E, G, H

G : E, H

E : B, F

F : A



Adjacency Lists

A : B, D

B : C, F

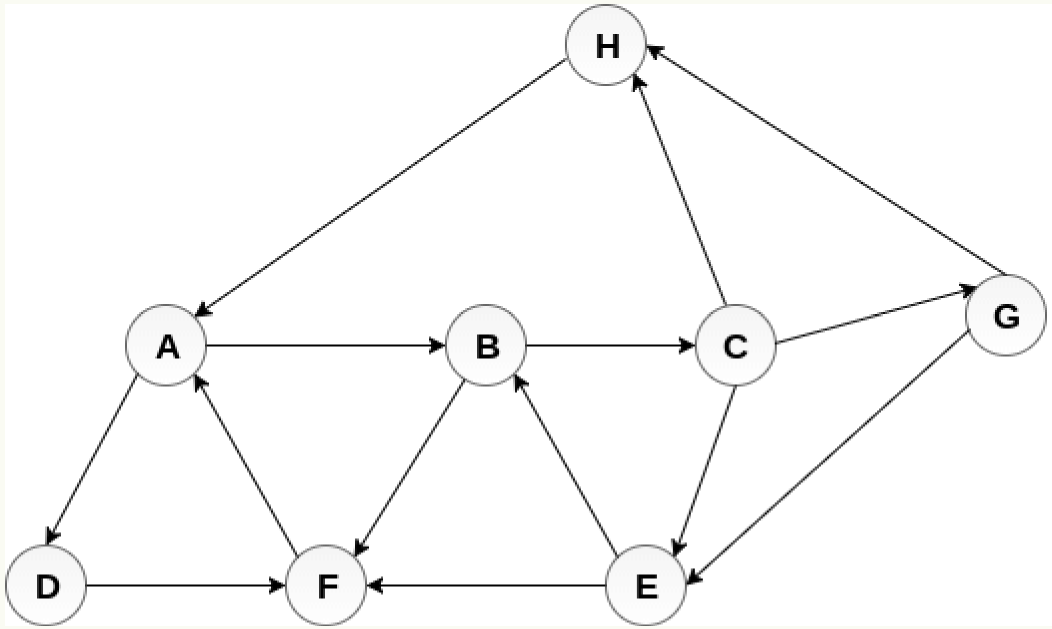
C : E, G, H

G : E, H

E : B, F

F : A

D : F



Adjacency Lists

A : B, D

B : C, F

C : E, G, H

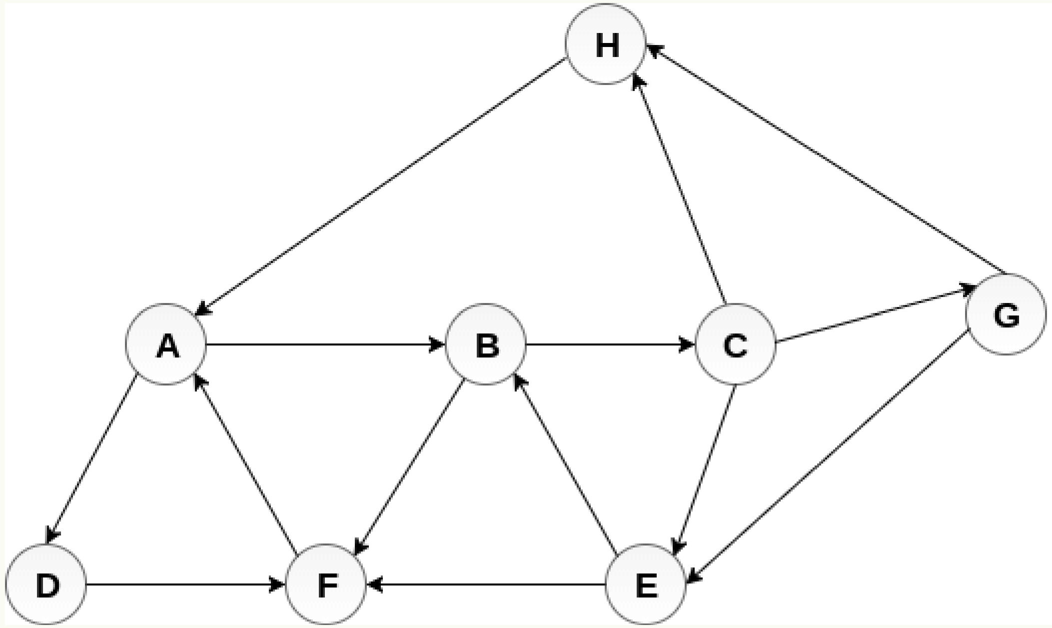
G : E, H

E : B, F

F : A

D : F

H : A



Adjacency Lists

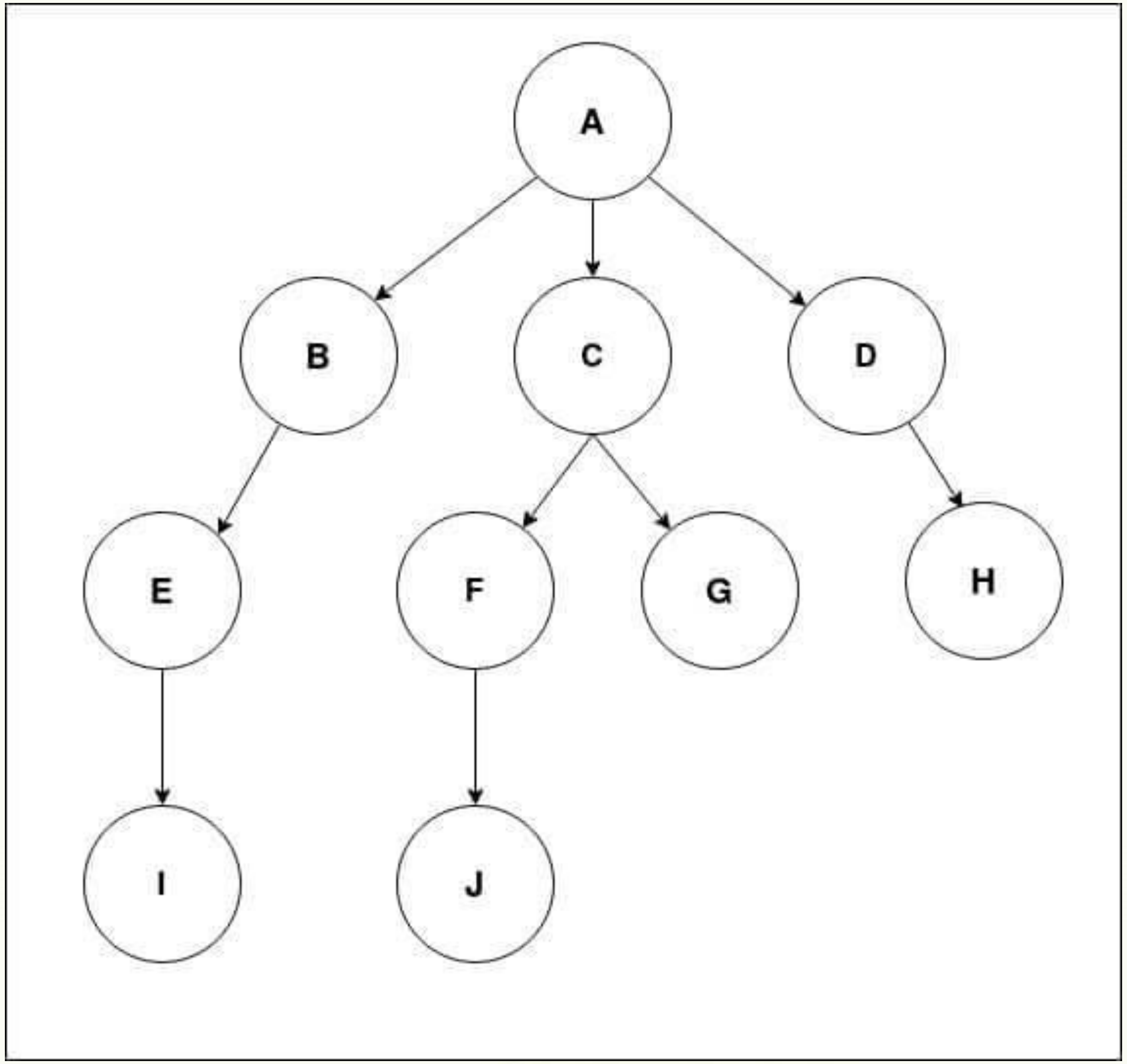
A : B, D

B : C, F

C : E, G, H

G : E, H

E : B, F


A
B
C
D
E
F
G
H
I
J