جـامعة الجـلالة
GALALA UNIVERSITY

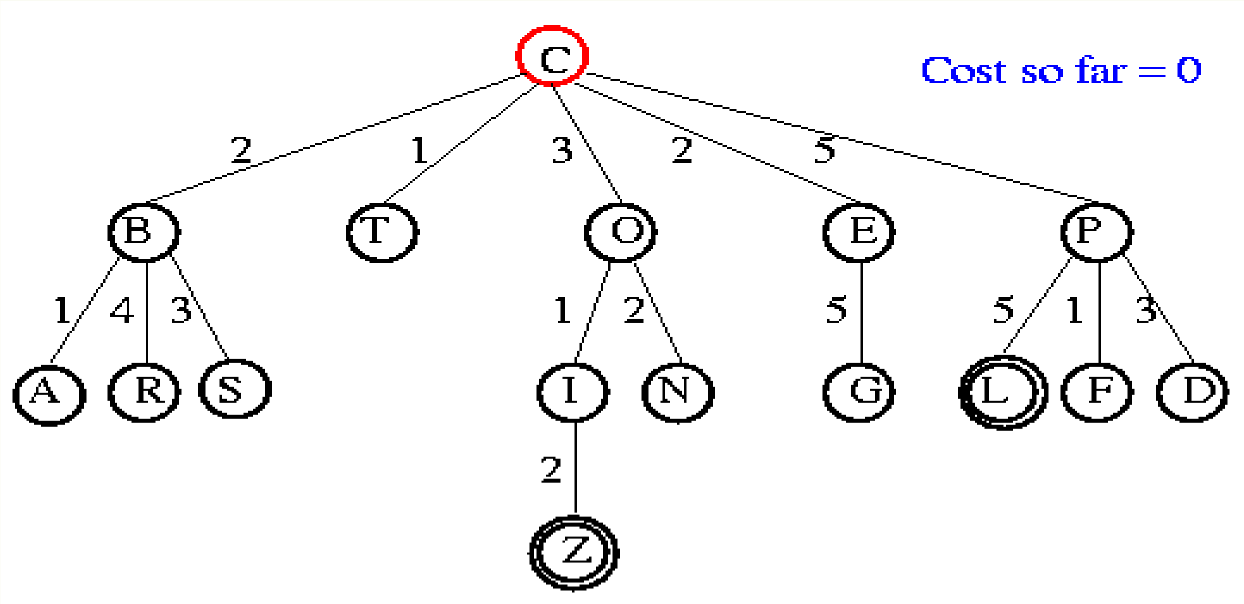
Artificial Intelligence Science Program

Chapter 3: Solving Problems by Searching

Uniform-Cost-First

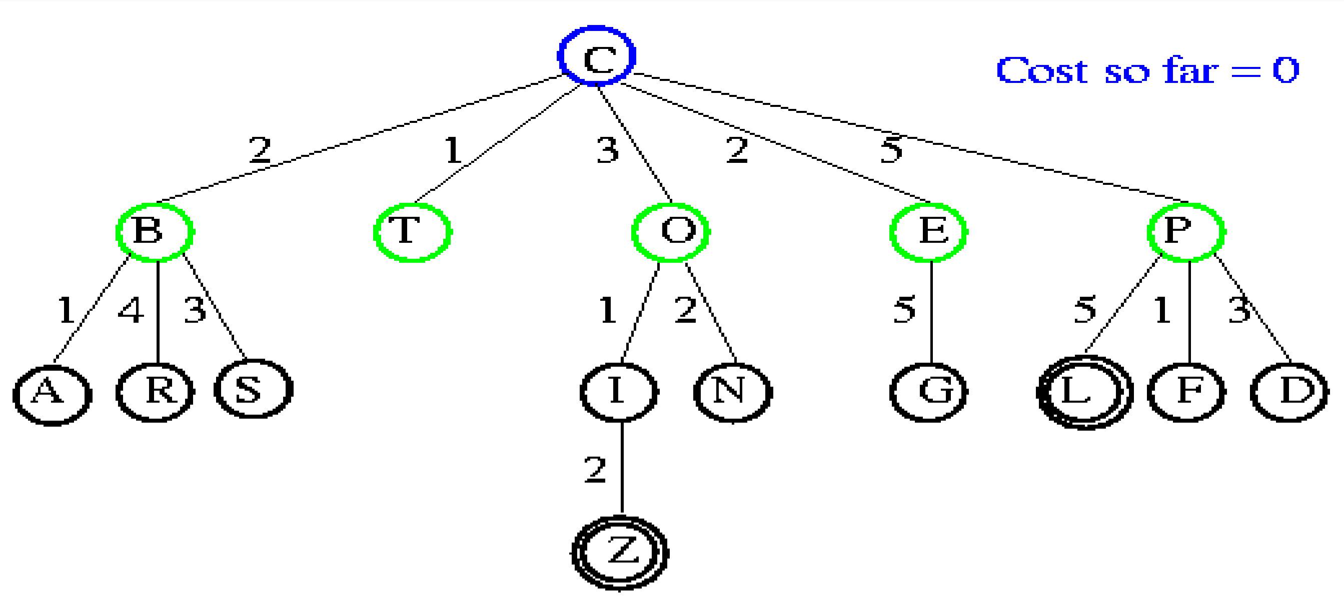
- Visits the next node which has the least total cost from the root, until a goal state is reached.
- – Similar to BREADTH-FIRST, but with an evaluation of the cost for each reachable node.
- g(n) = path cost(n) = sum of individual edge costs to reach the current node.

UCS Example



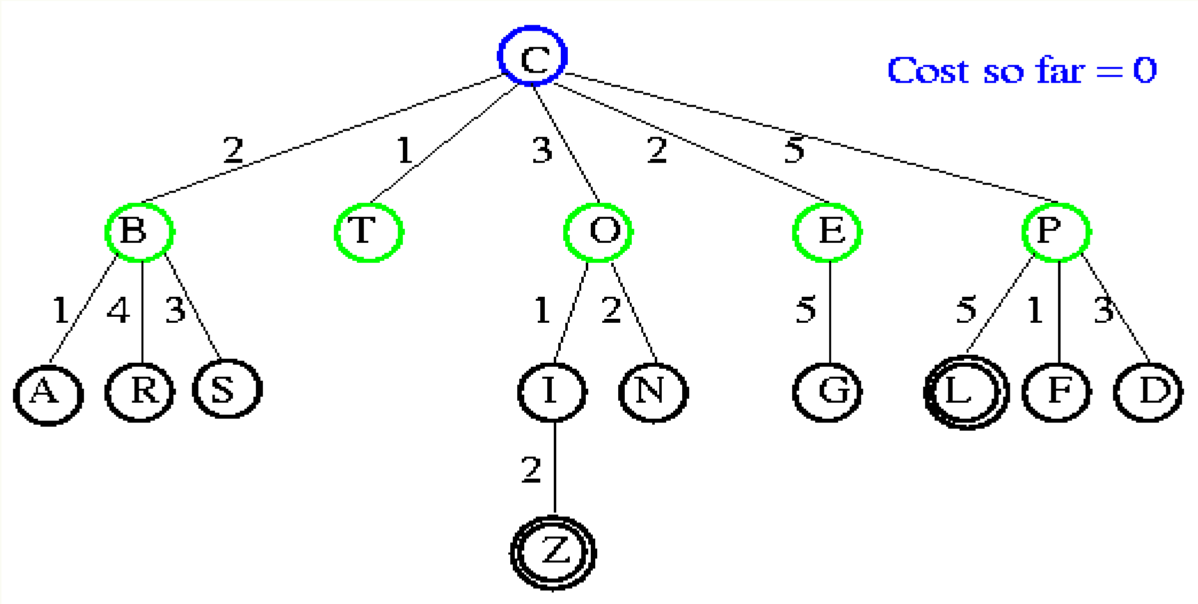
Open list: C

UCS Example



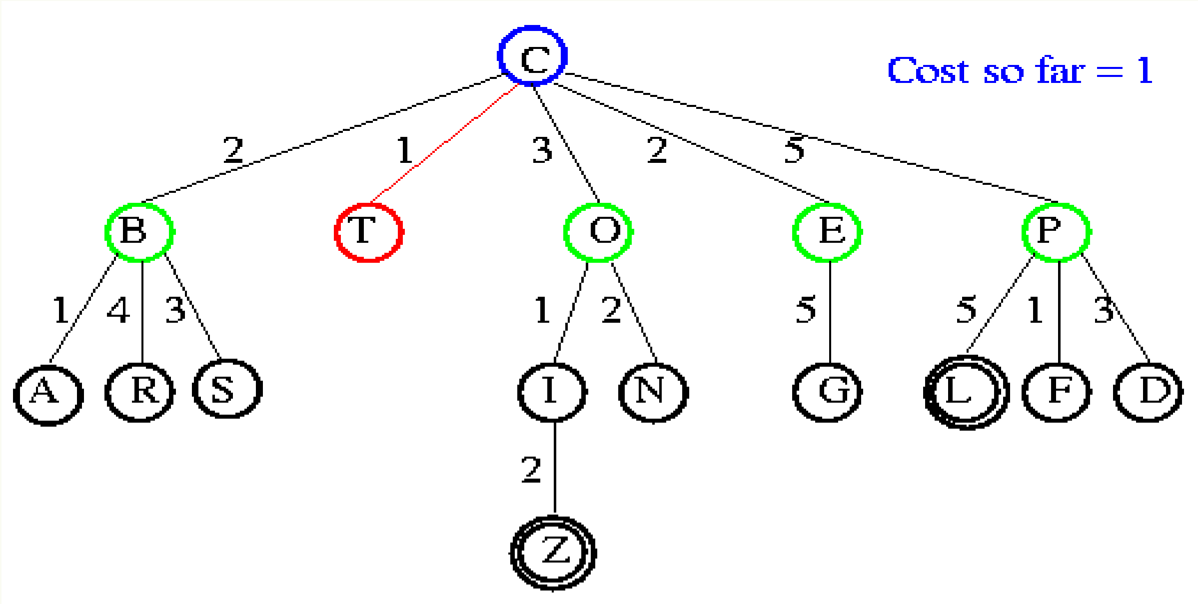
Open list: B(2) T(1) O(3) E(2) P(5)

UCS Example



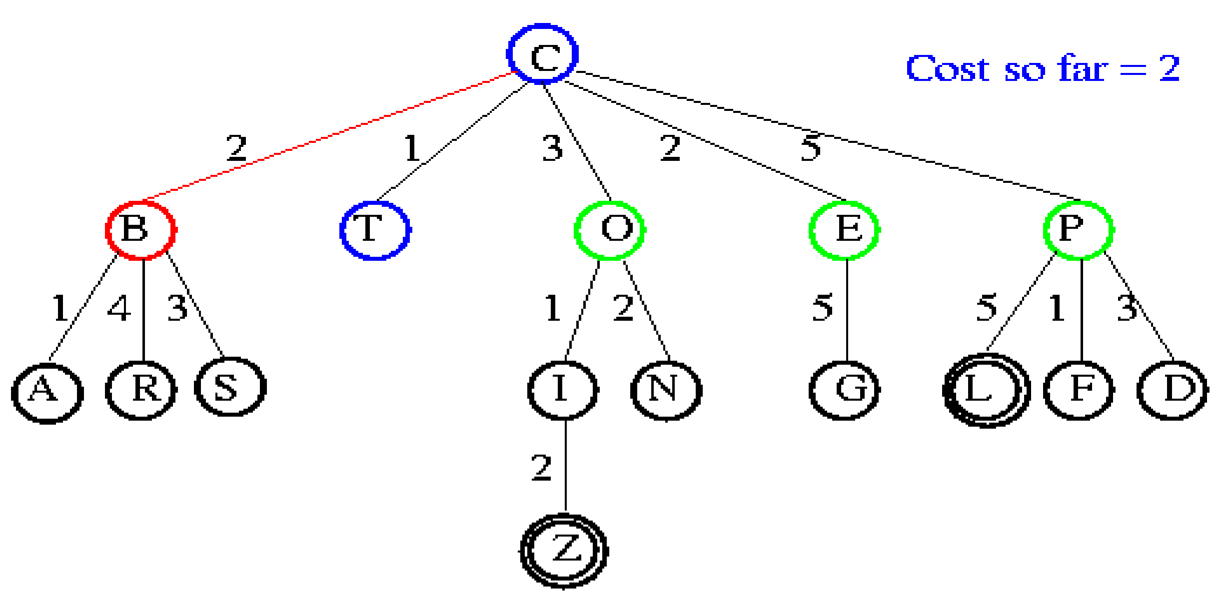
Open list: T(1) B(2) E(2) O(3) P(5)

UCS Example



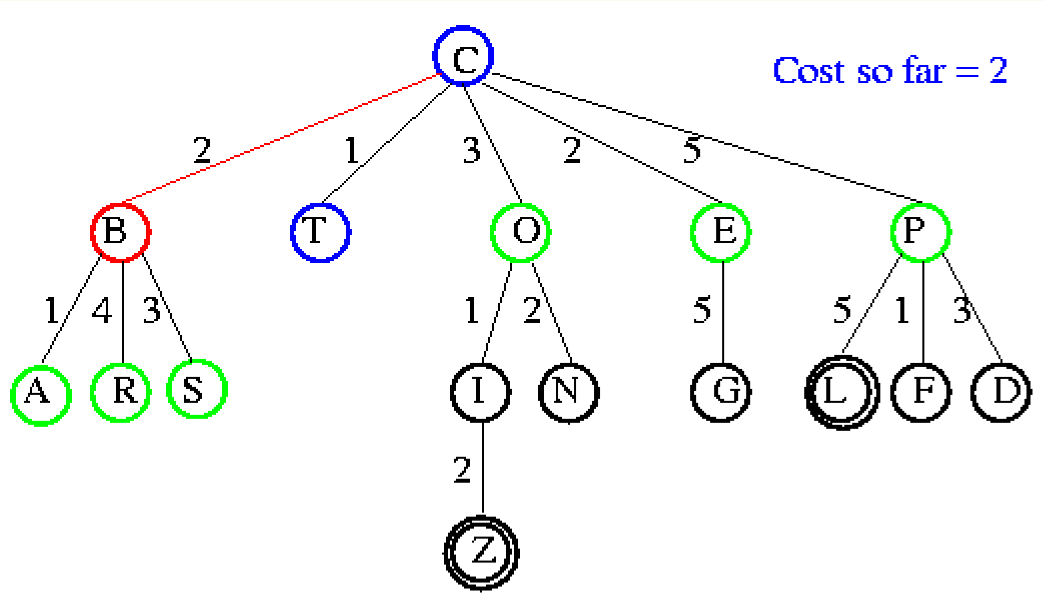
Open list: B(2) E(2) O(3) P(5)

UCS Example



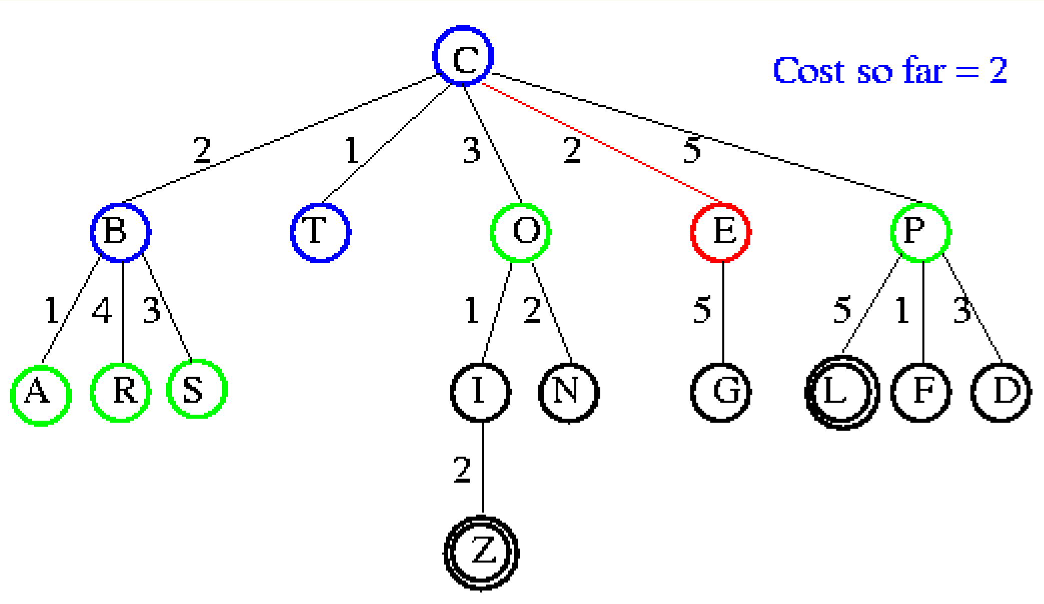
Open list: E(2) O(3) P(5)

UCS Example



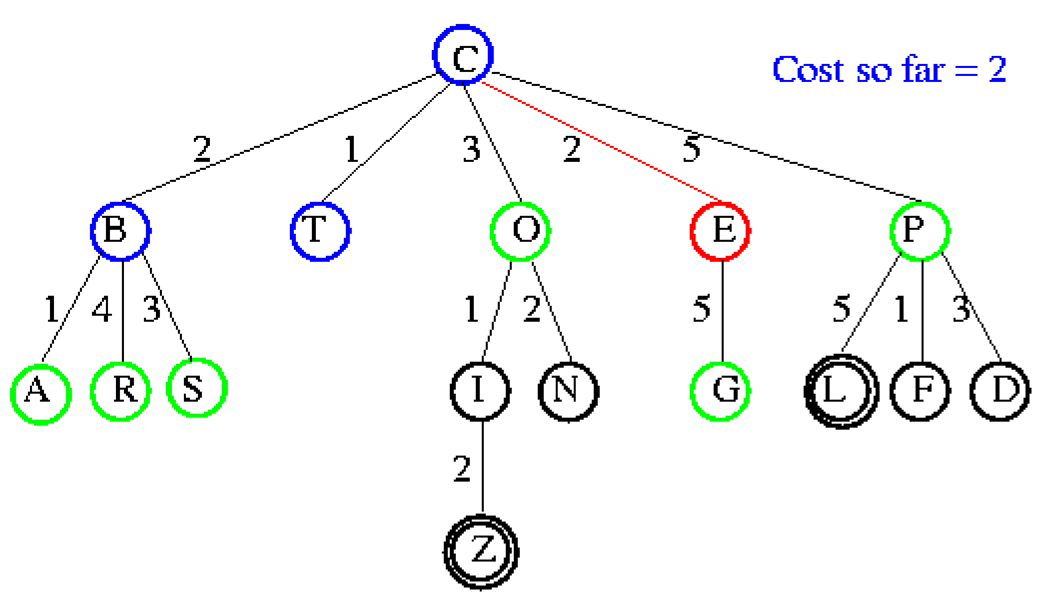
Open list: E(2) O(3) A(3) S(5) P(5) R(6)

UCS Example



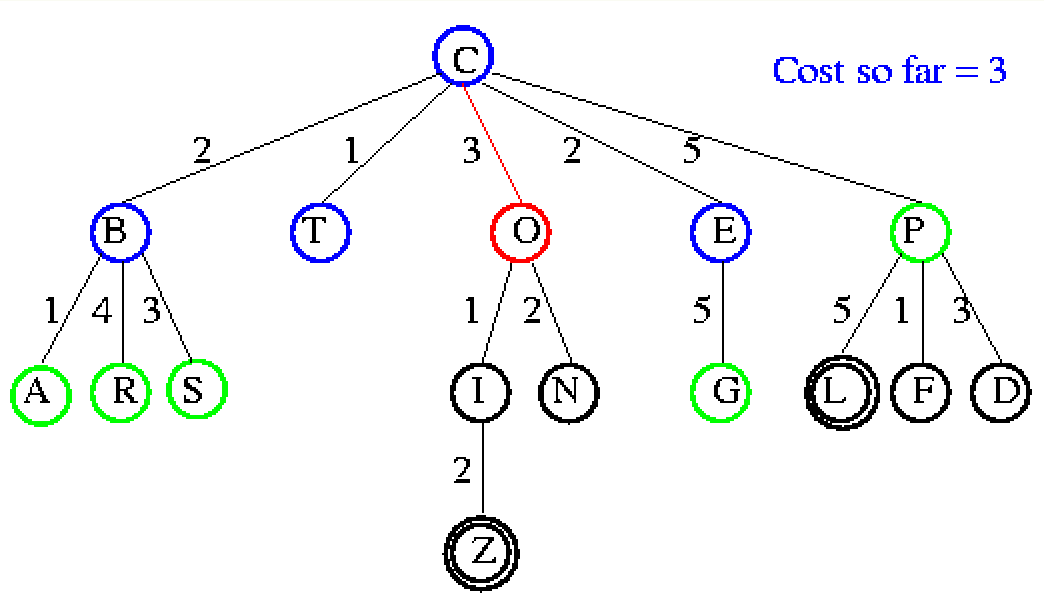
Open list: O(3) A(3) S(5) P(5) R(6)

UCS Example



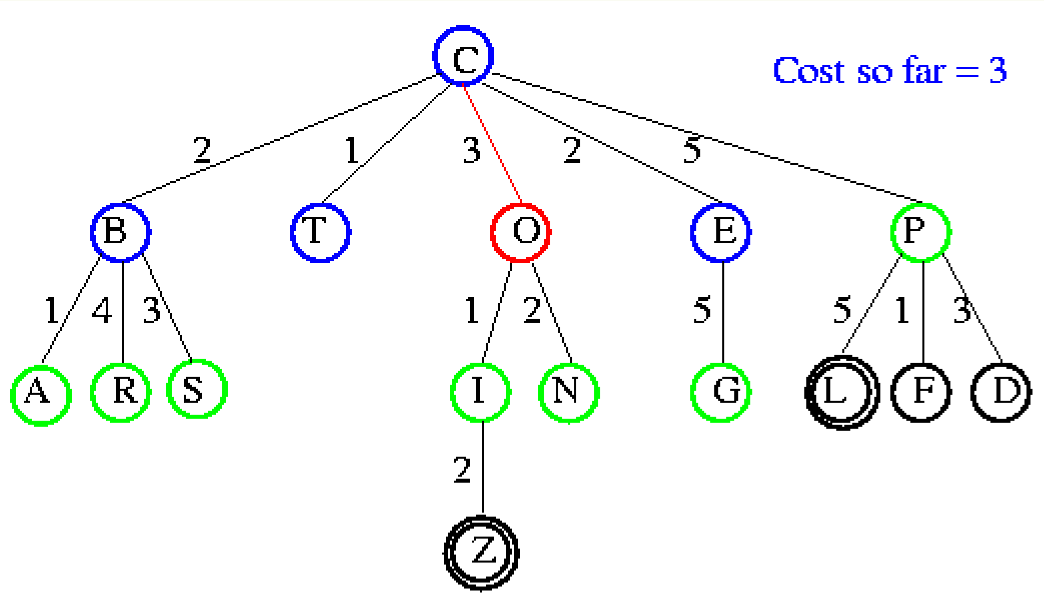
Open list: O(3) A(3) S(5) P(5) R(6) G(10)

UCS Example



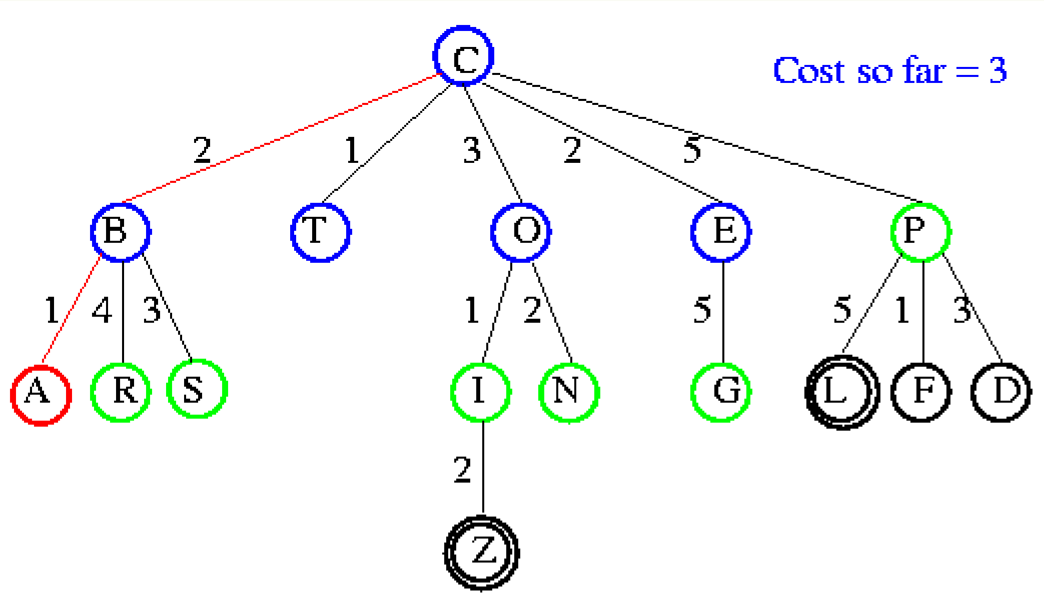
Open list: A(3) S(5) P(5) R(6) G(10)

UCS Example



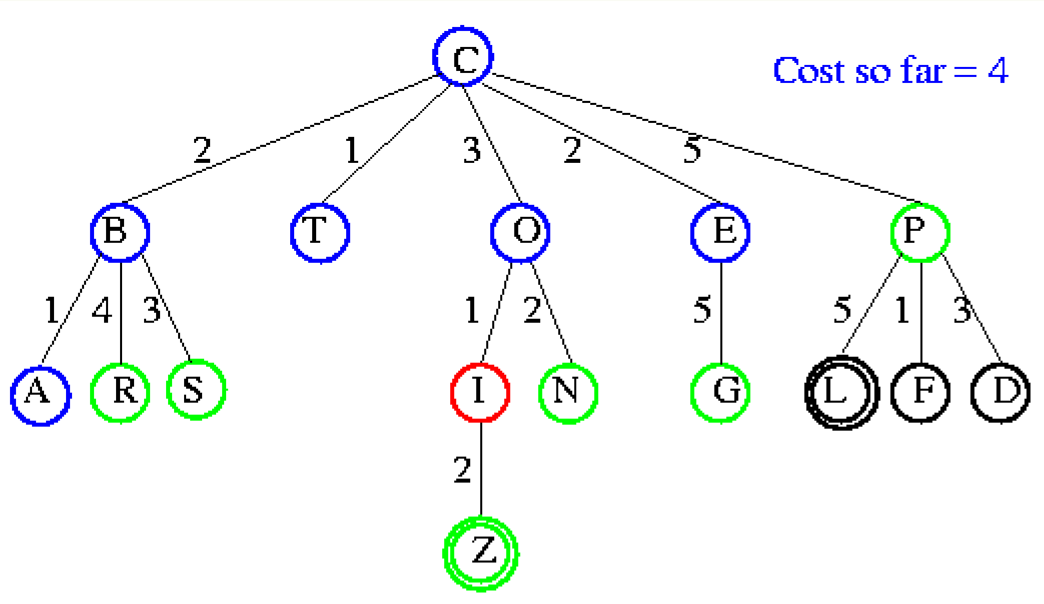
Open list: A(3) I(4) S(5) N(5) P(5) R(6) G(10)

UCS Example



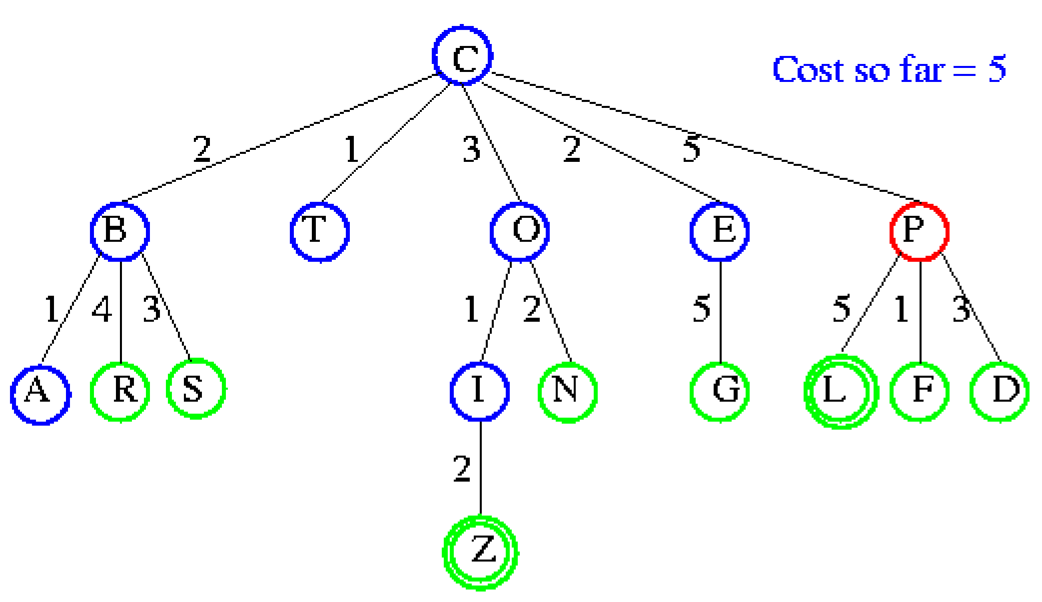
Open list: I(4) P(5) S(5) N(5) R(6) G(10)

UCS Example



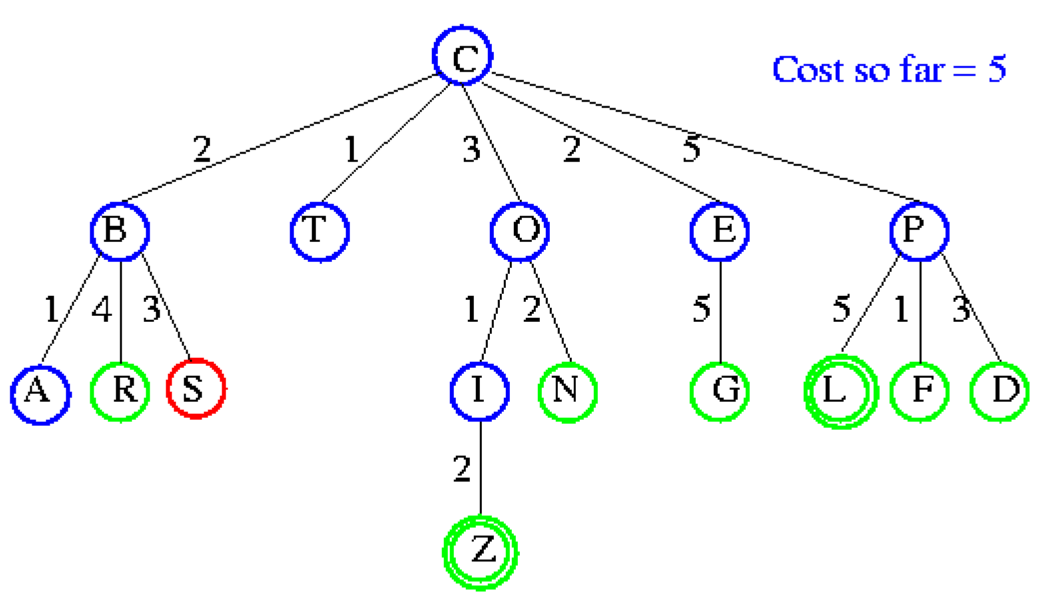
Open list: P(5) S(5) N(5) R(6) Z(6) G(10)

UCS Example



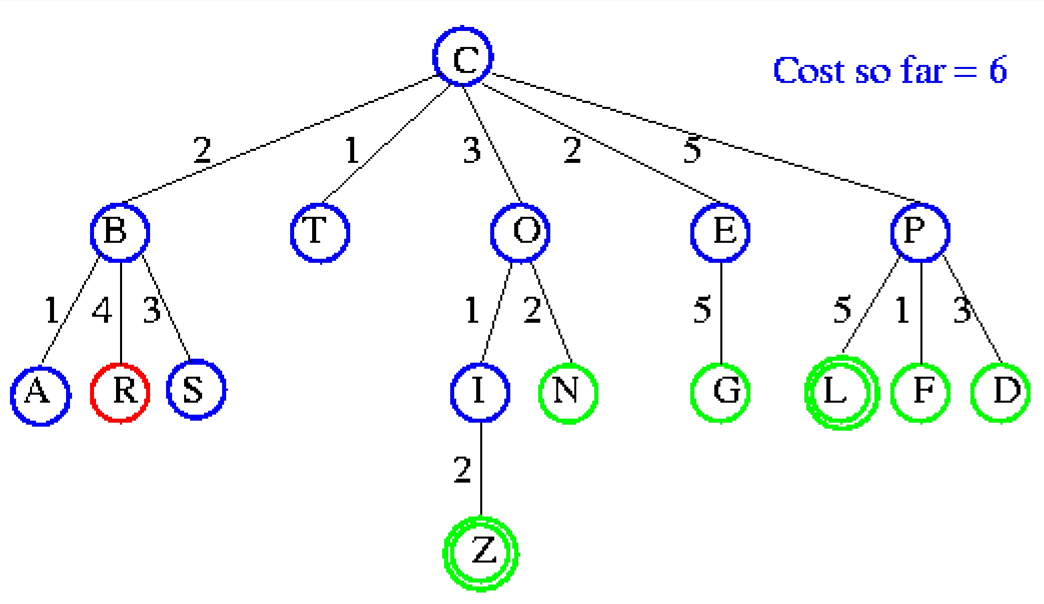
Open list: S(5) N(5) R(6) Z(6) F(6) D(8) G(10) L(10)

UCS Example



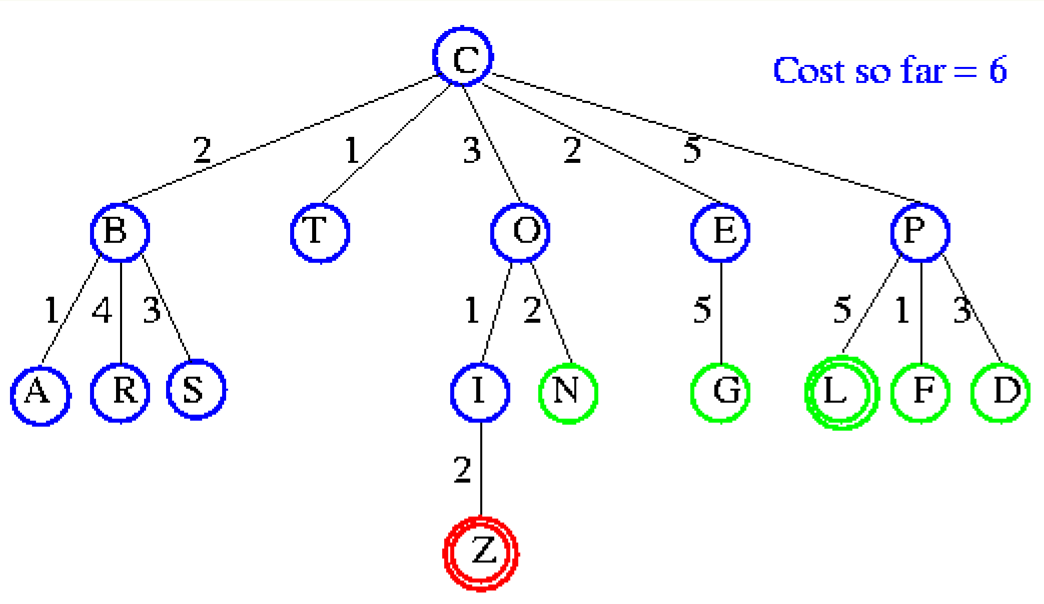
Open list: N(5) R(6) Z(6) F(6) D(8) G(10) L(10)

UCS Example



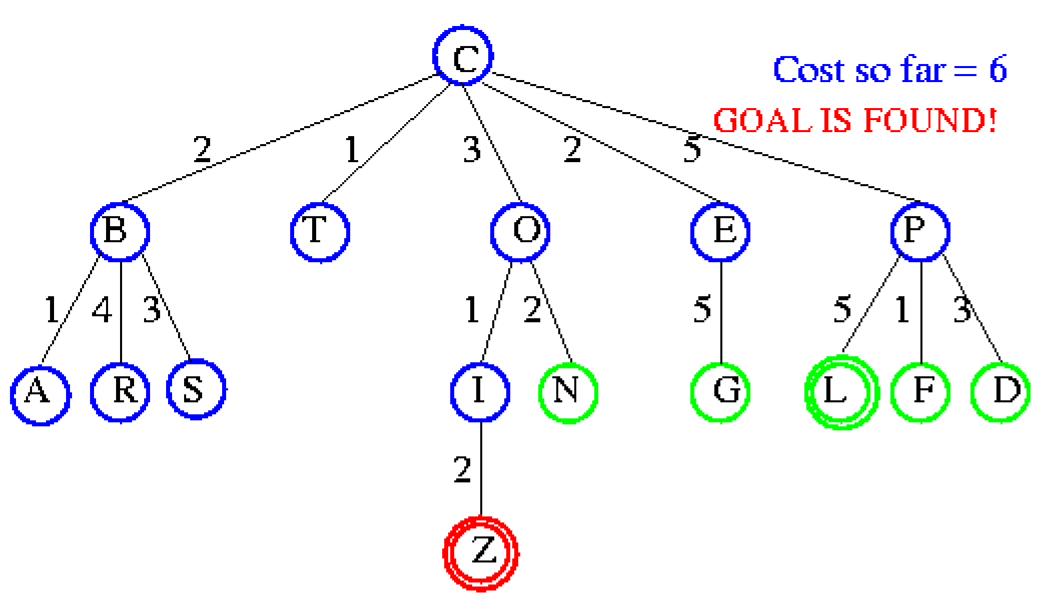
Open list: Z(6) F(6) D(8) G(10) L(10)

UCS Example



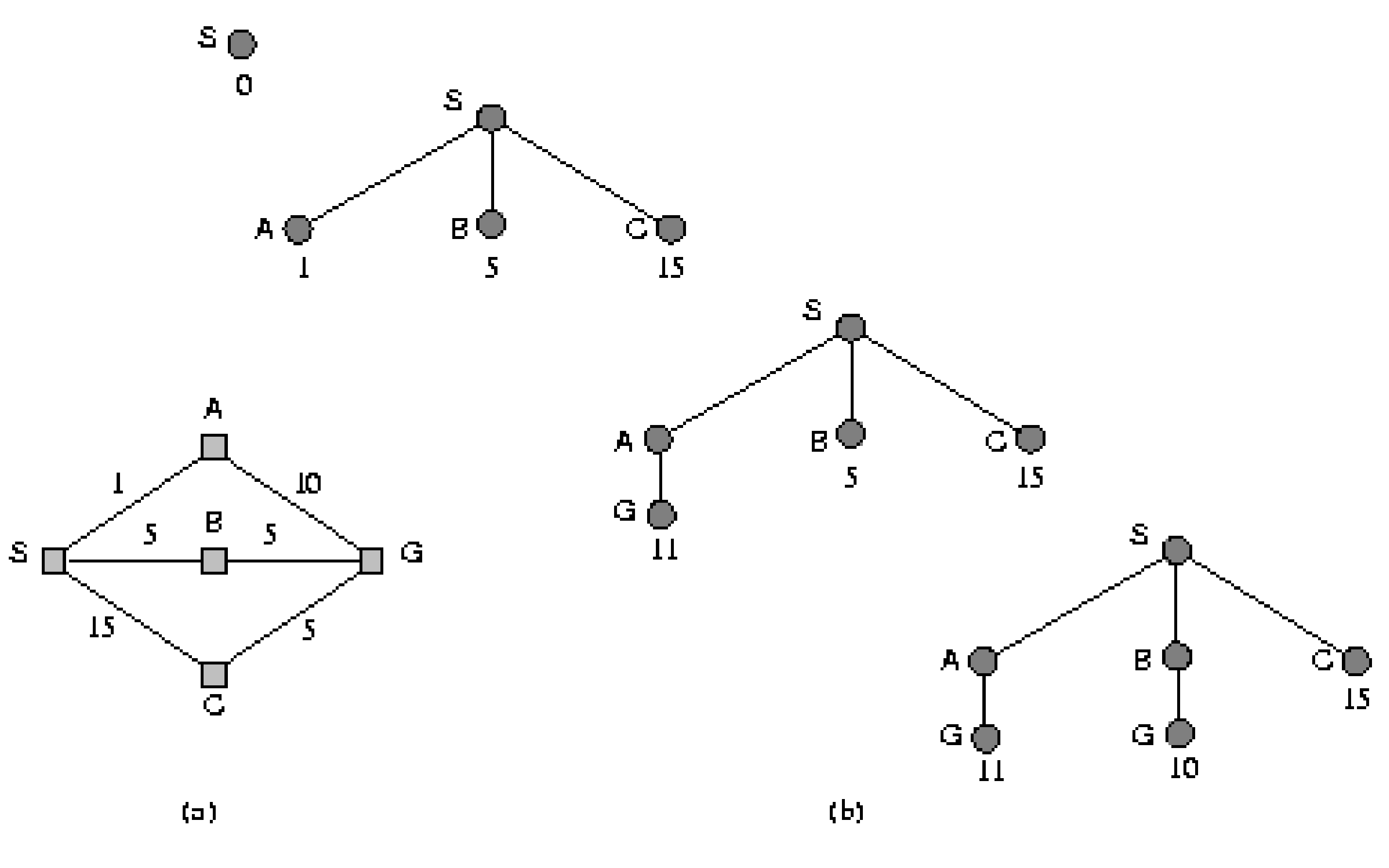
Open list: F(6) D(8) G(10) L(10)

UCS Example

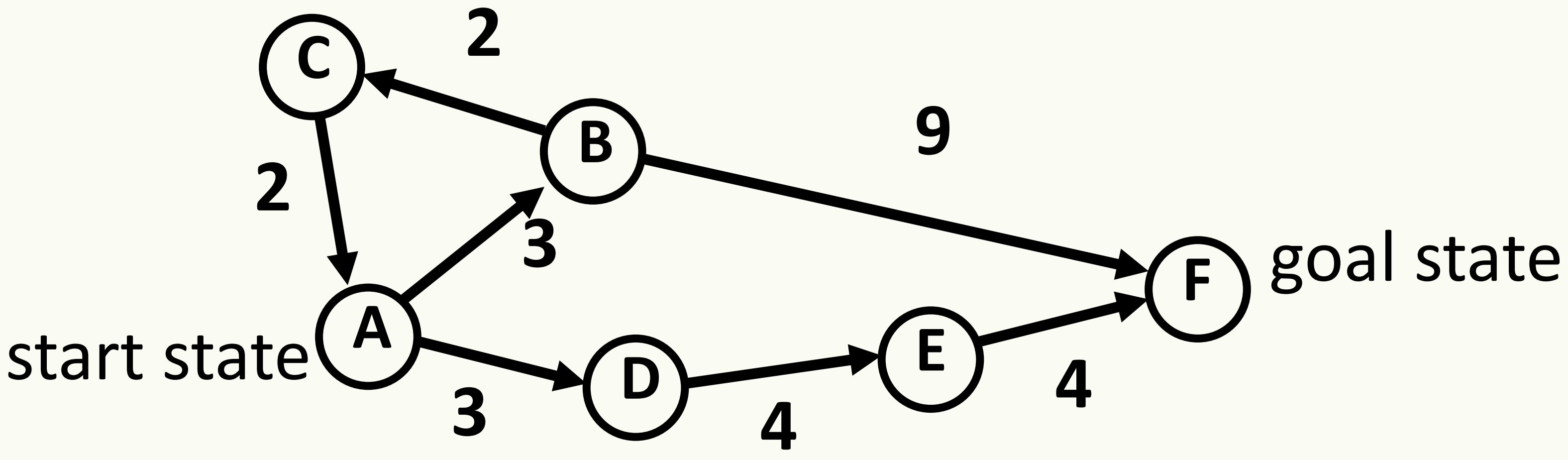


Cost so far = 6
GOAL IS FOUND!
C
2
1
3
2
5
B
T
O
E
P
1
4
3
1
2
5
5
1
3
A
R
S
I
N
G
L
F
D
2
Z

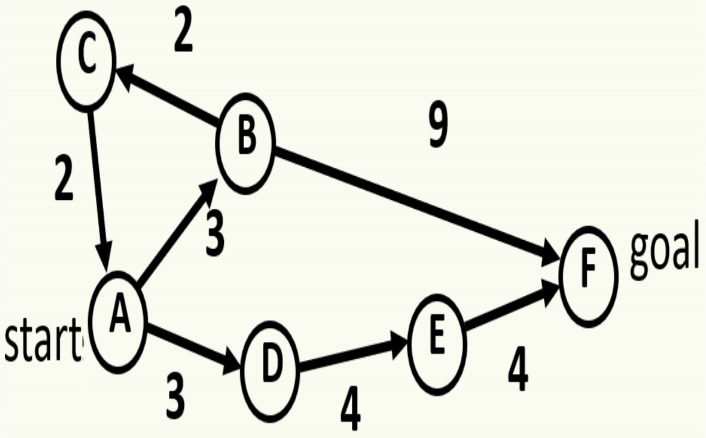
UCS Example

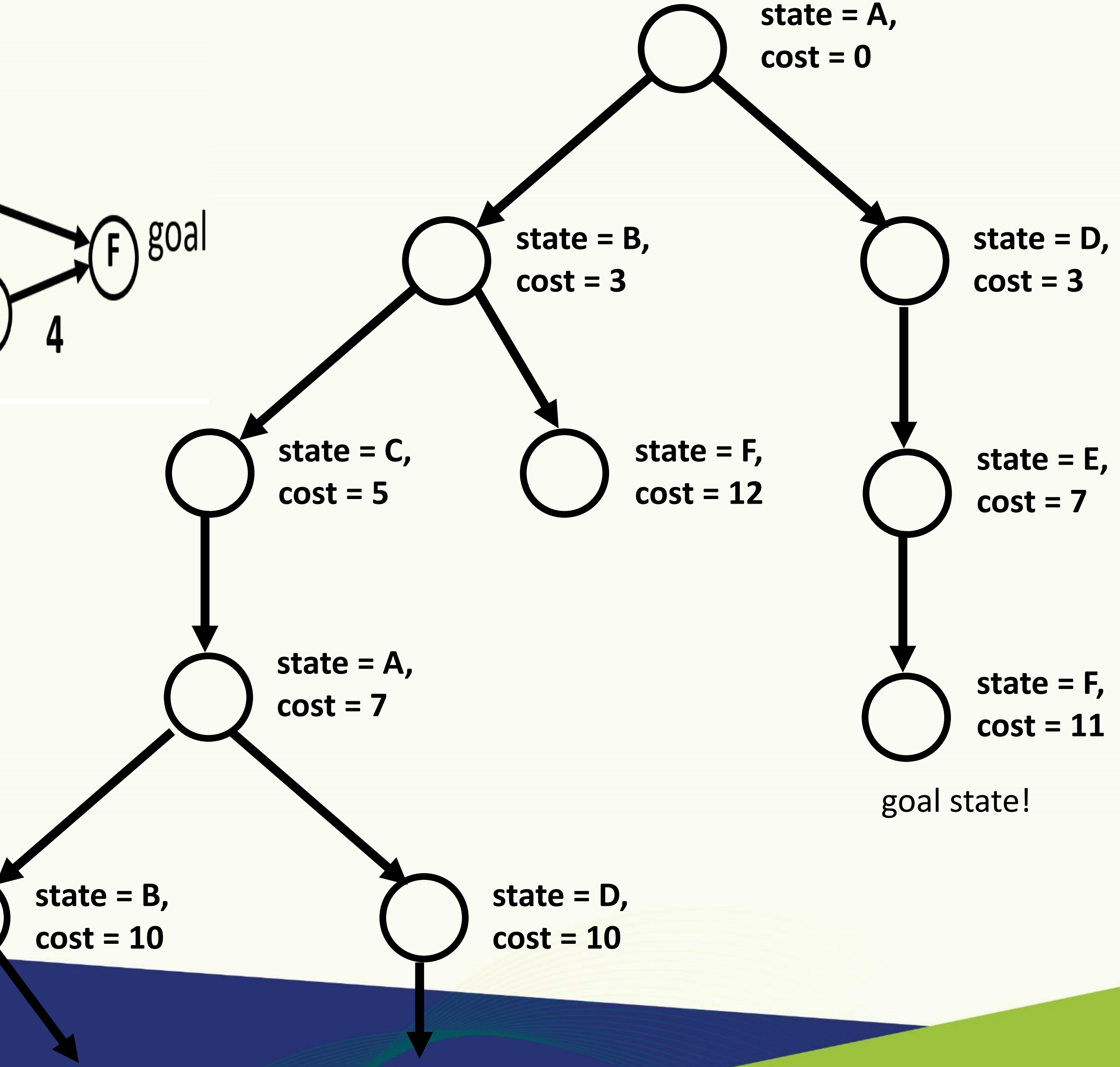


A simple example: traveling on a graph



Uniform cost search

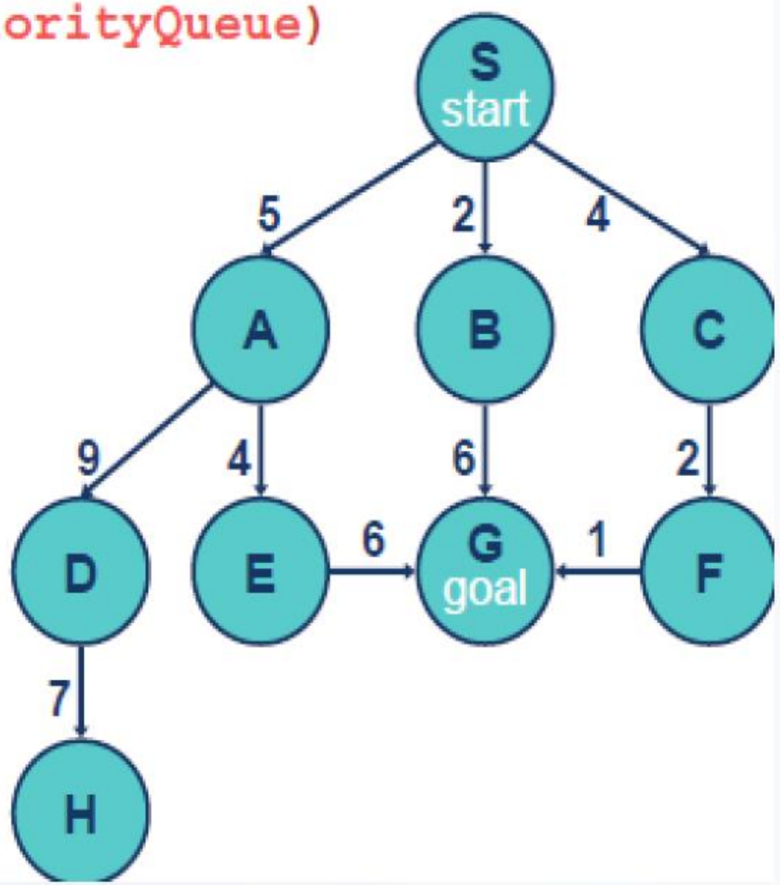




```
generalSearch(problem, priorityQueue)
```

# of nodes tested: 0, expanded: 0

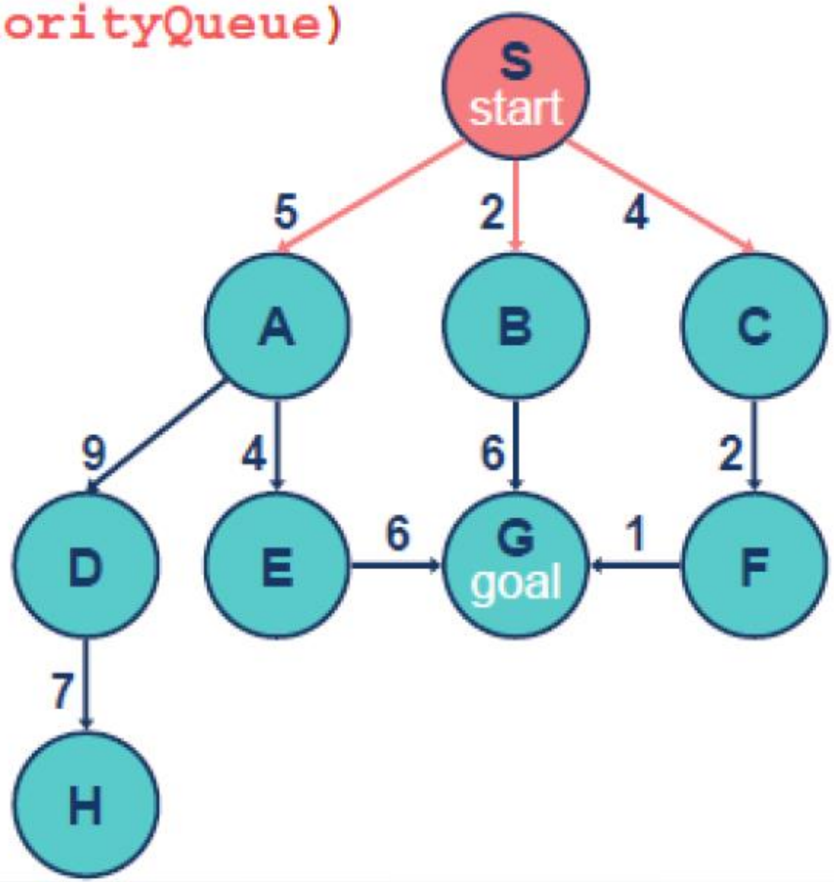
expnd. node	nodes list
	{S}



```
generalSearch(problem, priorityQueue)
```

\# of nodes tested: 1, expanded: 1

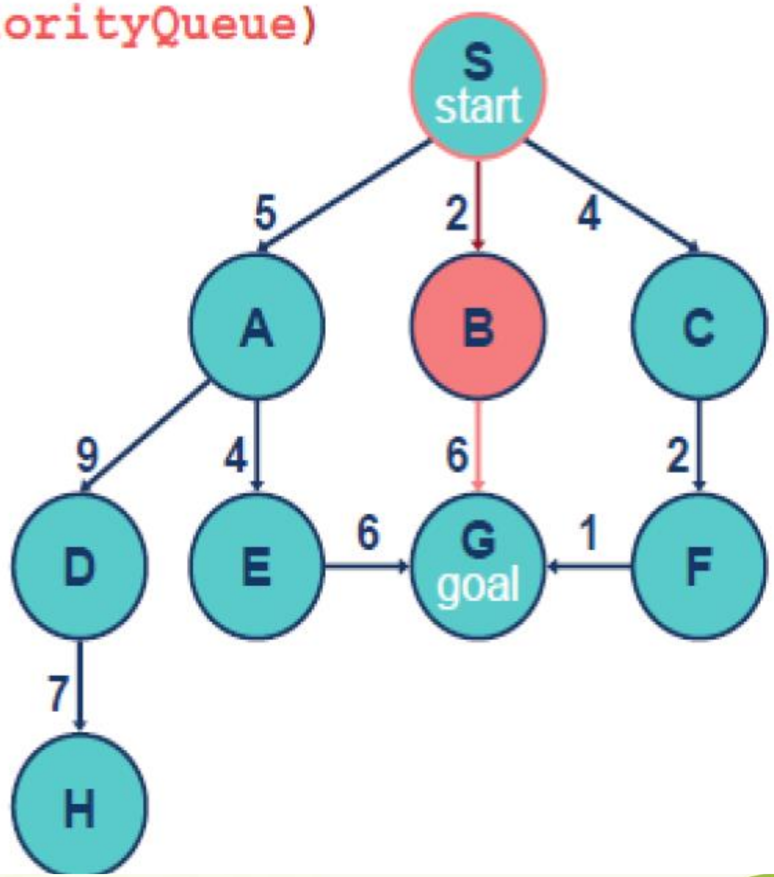
expnd. node	nodes list
	{S:0}
S not goal	{B:2,C:4,A:5}



```
generalSearch(problem, priorityQueue)
```

# of nodes tested: 2, expanded: 2

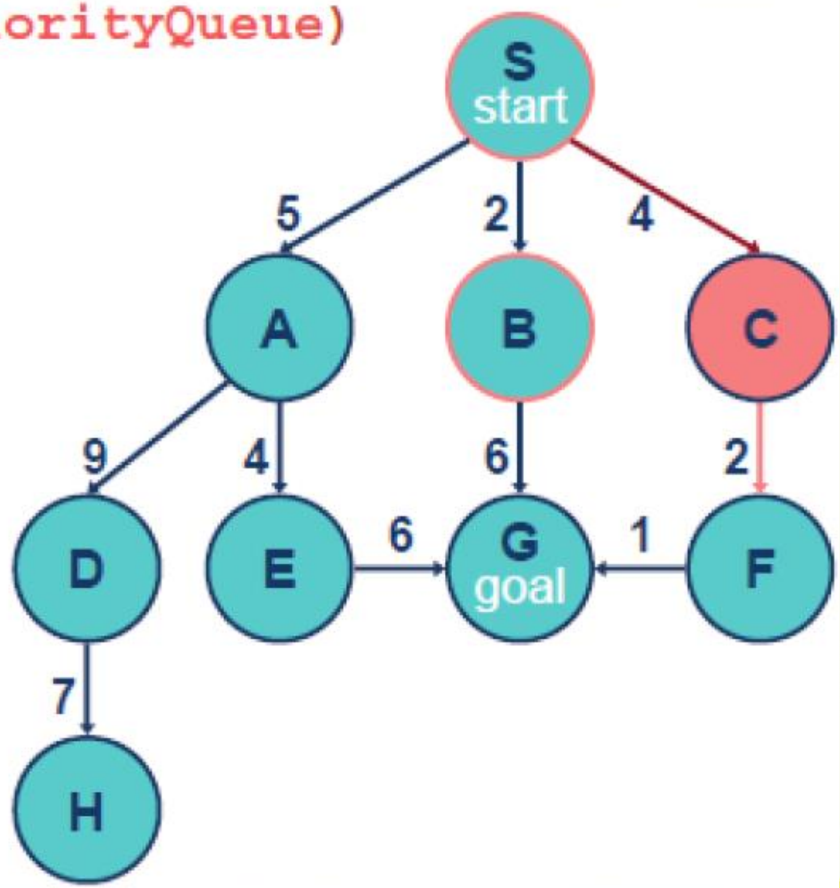
expnd. node	nodes list
	{S}
S	{B:2,C:4,A:5}
B not goal	{C:4,A:5,G:2+6}



generalSearch(problem, priorityQueue)

# of nodes tested: 3, expanded: 3

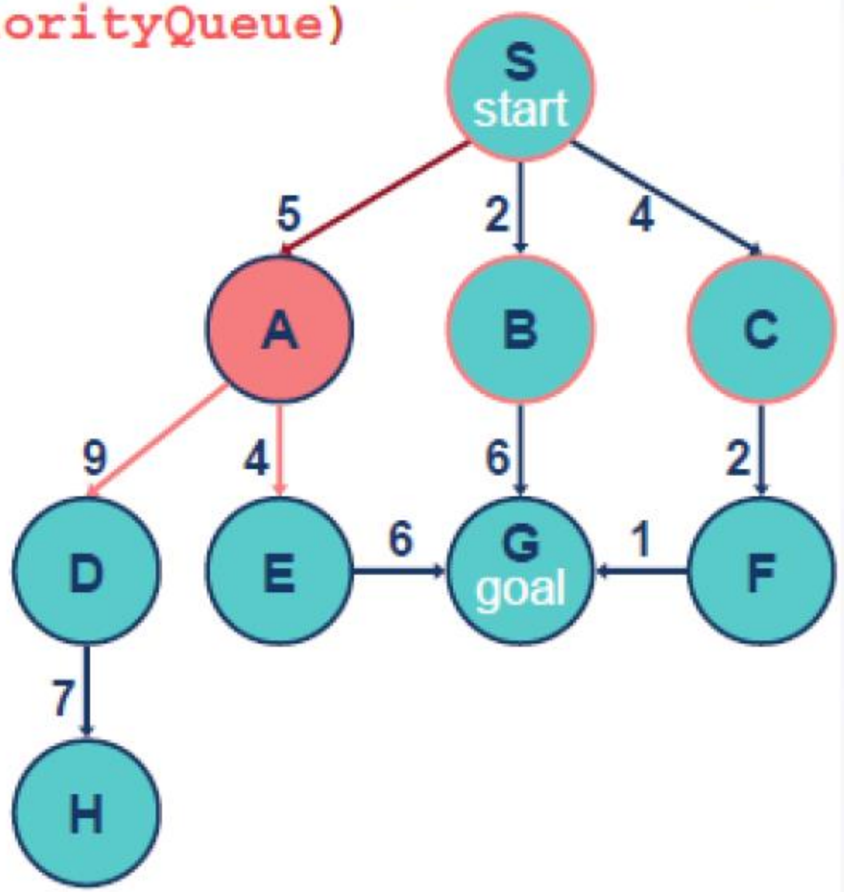
expnd. node	nodes list
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C not goal	{A:5,F:4+2,G:8}



`generalSearch(problem, priorityQueue)`

# of nodes tested: 4, expanded: 4

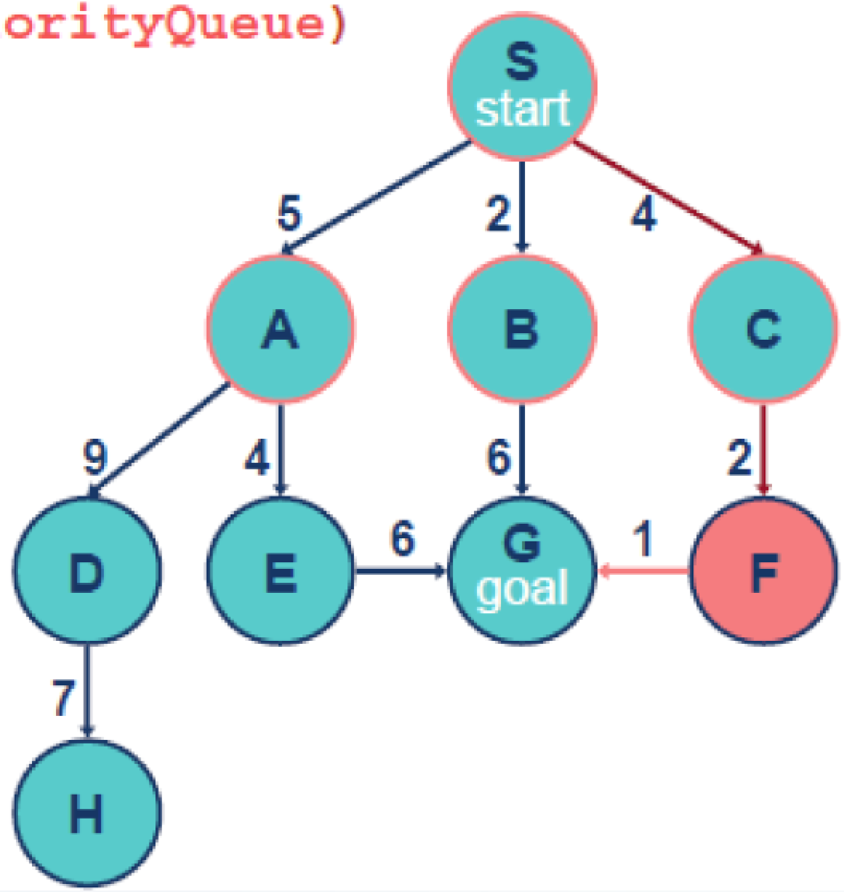
expnd. node	nodes list
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A not goal	{F:6,G:8,E:5+4, D:5+9}



```
generalSearch(problem, priorityQueue)
```

# of nodes tested: 5, expanded: 5

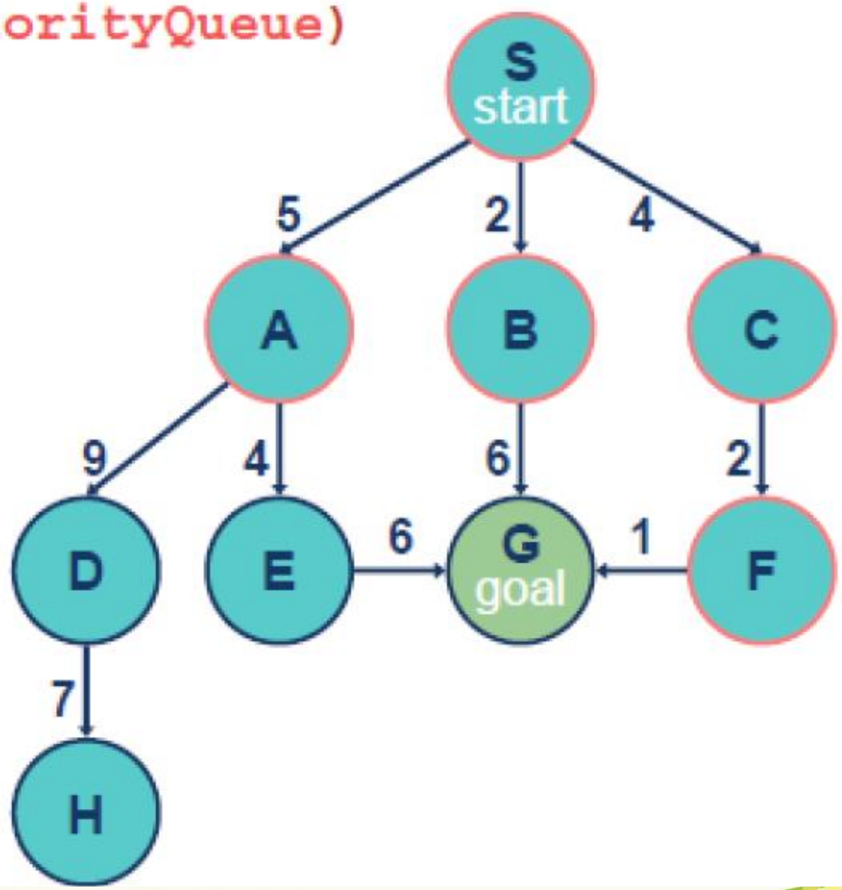
expnd. node	nodes list
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F not goal	{G:4+2+1,G:8,E:9, D:14}



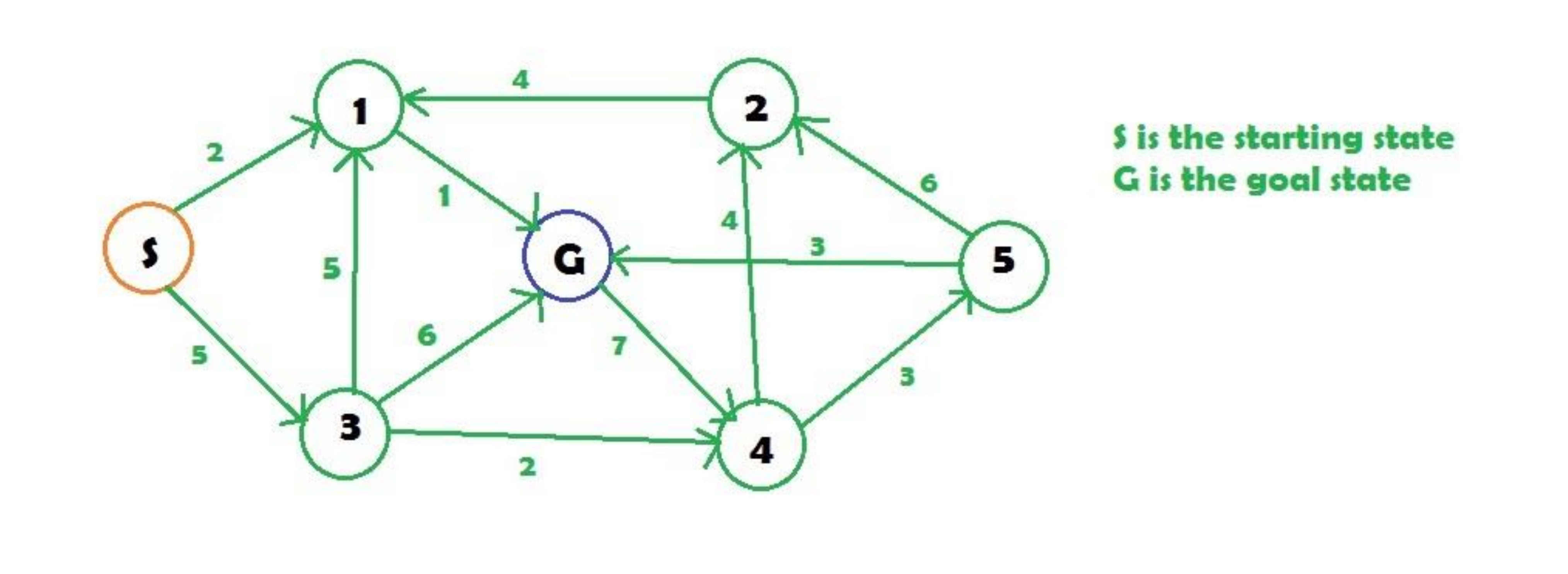
```
generalSearch(problem, priorityQueue)
```

# of nodes tested: 6, expanded: 5

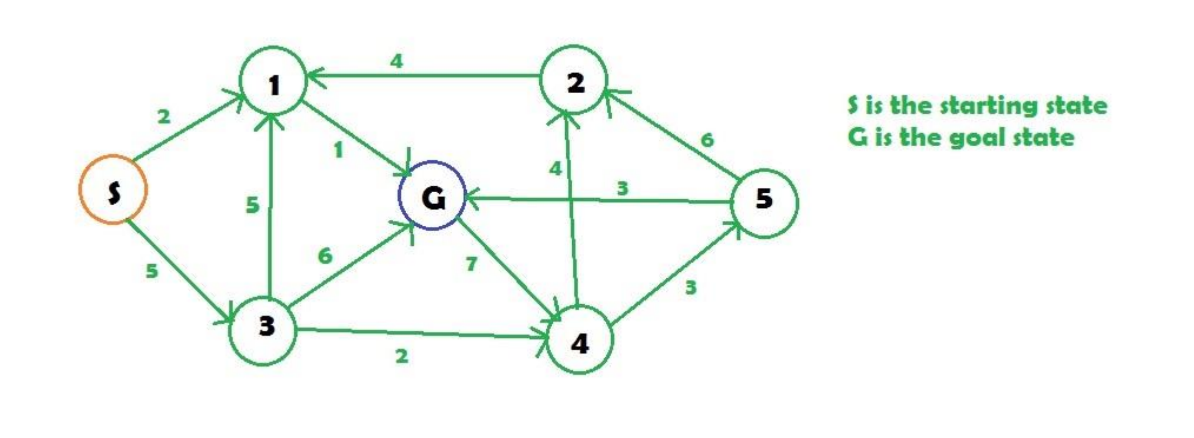
expnd. node	nodes list
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F	{G:7,G:8,E:9,D:14}
G goal	{G:8,E:9,D:14} no expand



UCS Example

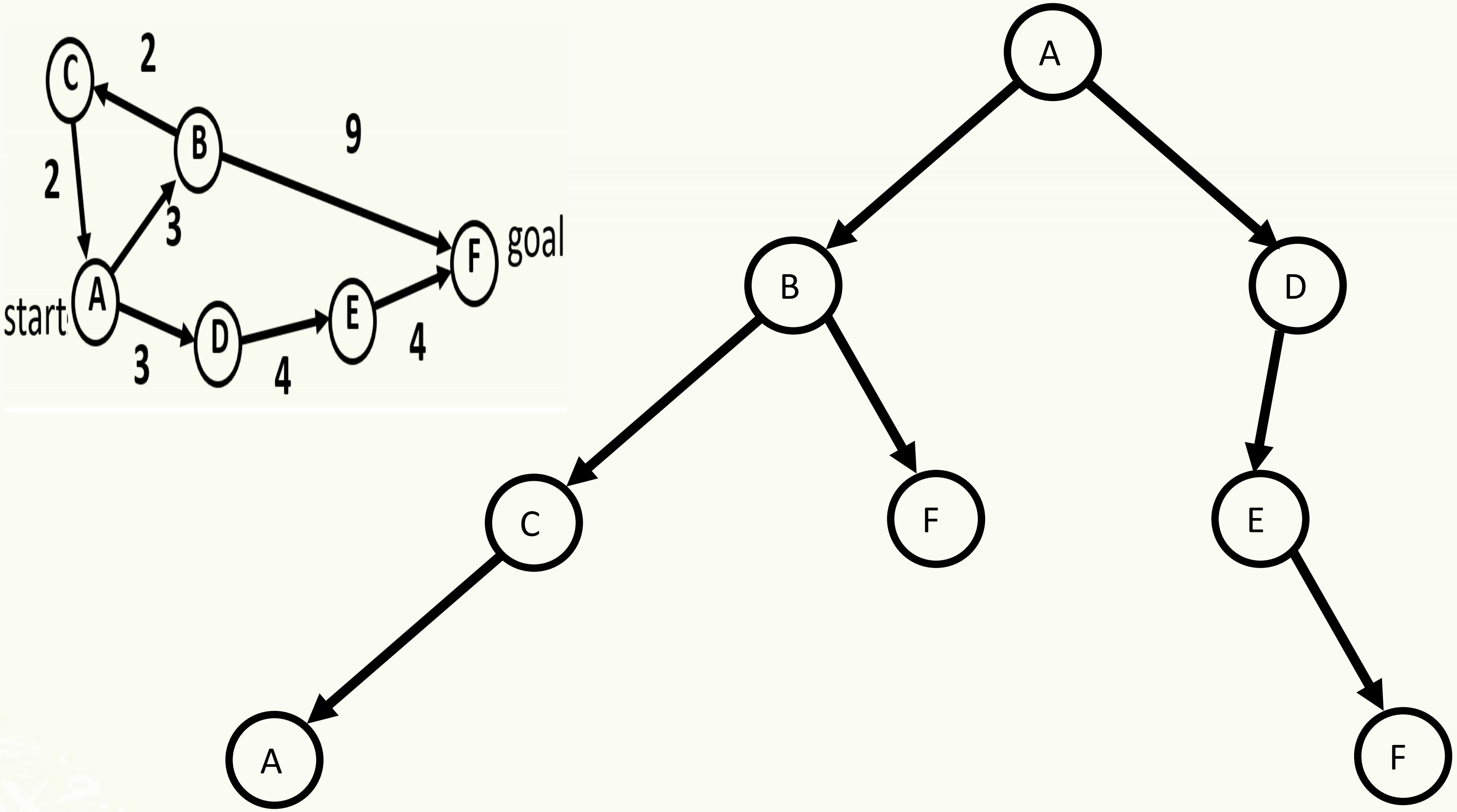

S
1
2
3
4
5
G
2
5
4
1
5
6
2
7
4
3
6
3
S is the starting state
G is the goal state

```
# 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])
```



```
# 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)):
    answer
# check for 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
```

Breadth-first search



Breadth-First vs. Uniform-Cost

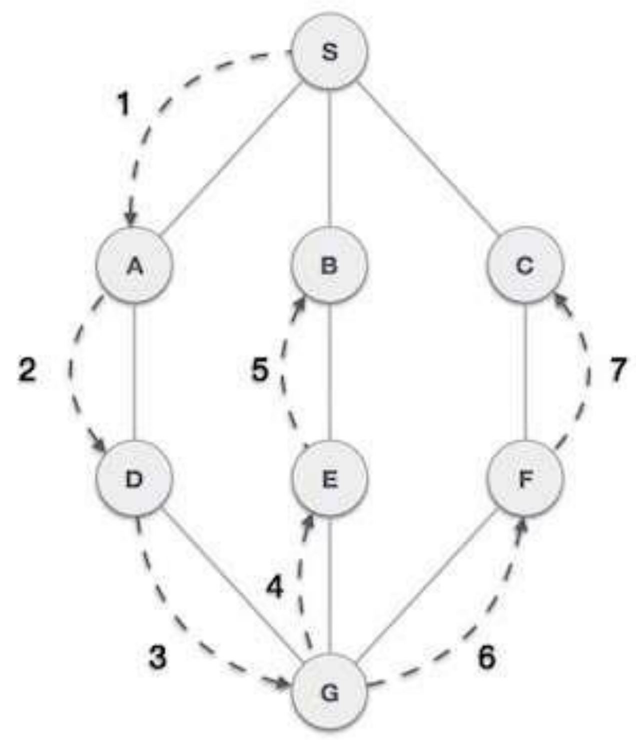
- Breadth-first search (BFS) is a special case of uniform-cost search when all edge costs are positive and identical.
- Breadth-first always expands the shallowest node
- Uniform-cost considers the overall path cost
    - Optimal for any (reasonable) cost function

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**.

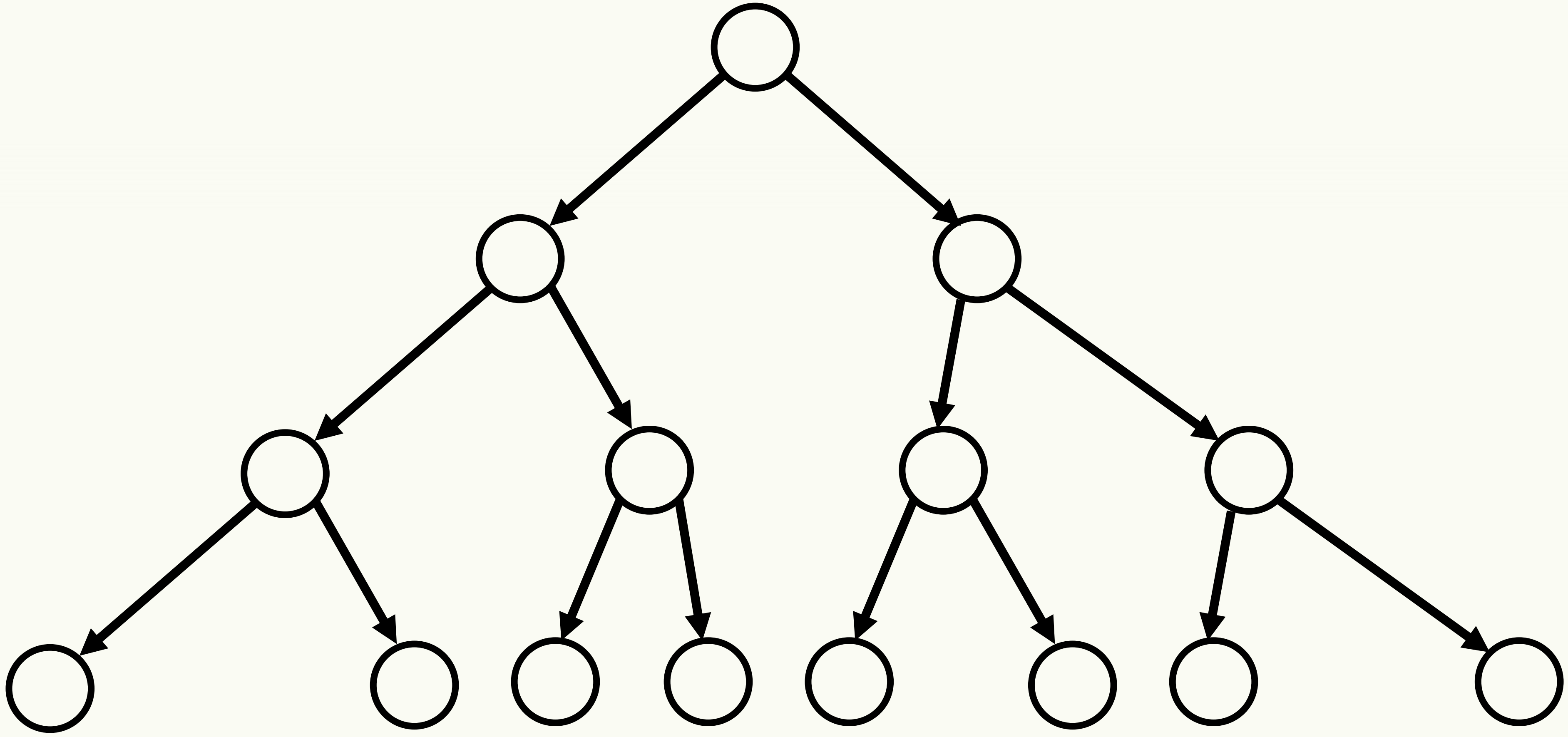
- It is implemented using a **stack** to return to the previous vertex to start a search, when a dead end is reached.



The DSF algorithm follows as:

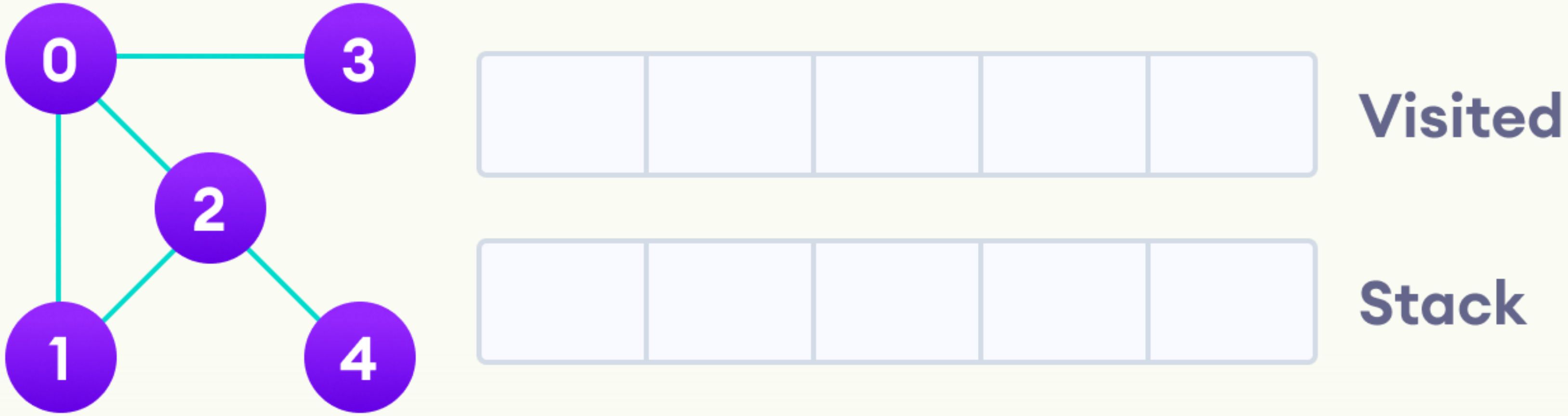
1. We will start by putting any one of the graph's vertex on top of the stack.
2. After that take the top item of the stack and add it to the visited list of the vertex.
3. Next, create a list of that adjacent node of the vertex. Add the ones which aren't in the visited list of vertexes to the top of the stack.
4. Lastly, keep repeating steps 2 and 3 until the stack is empty.

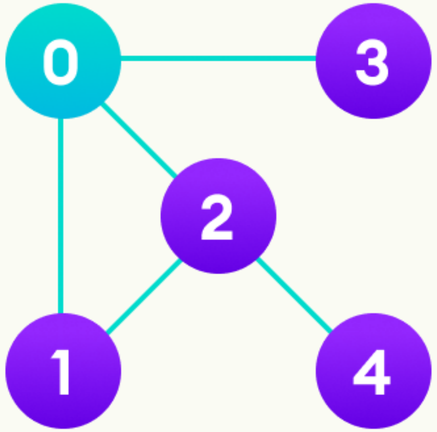
Depth-first search



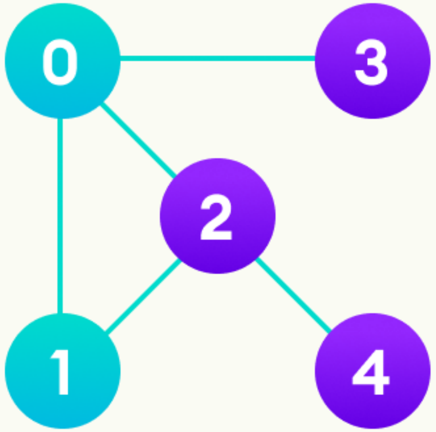
Example

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

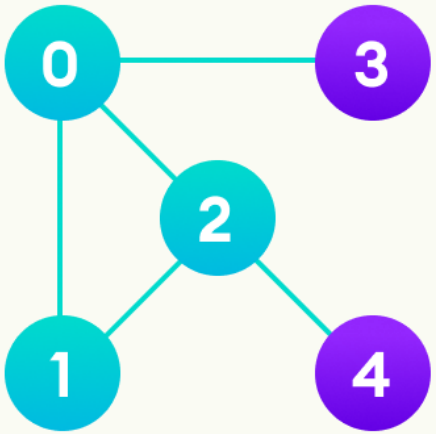




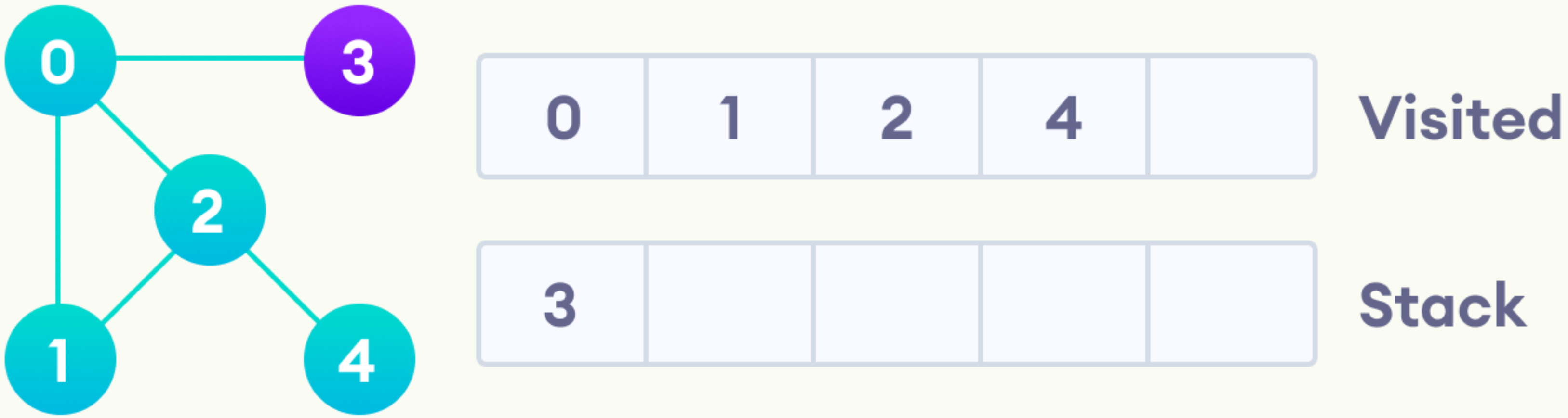
0					Visited
1	2	3			Stack

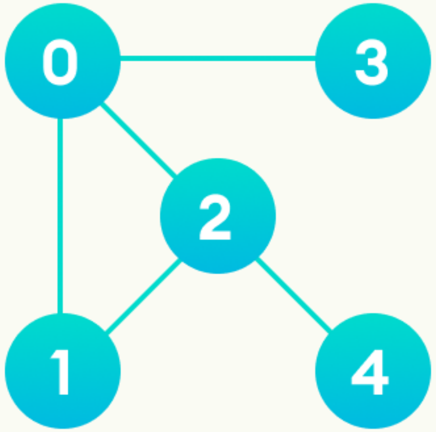


0	1				Visited
2	3				Stack

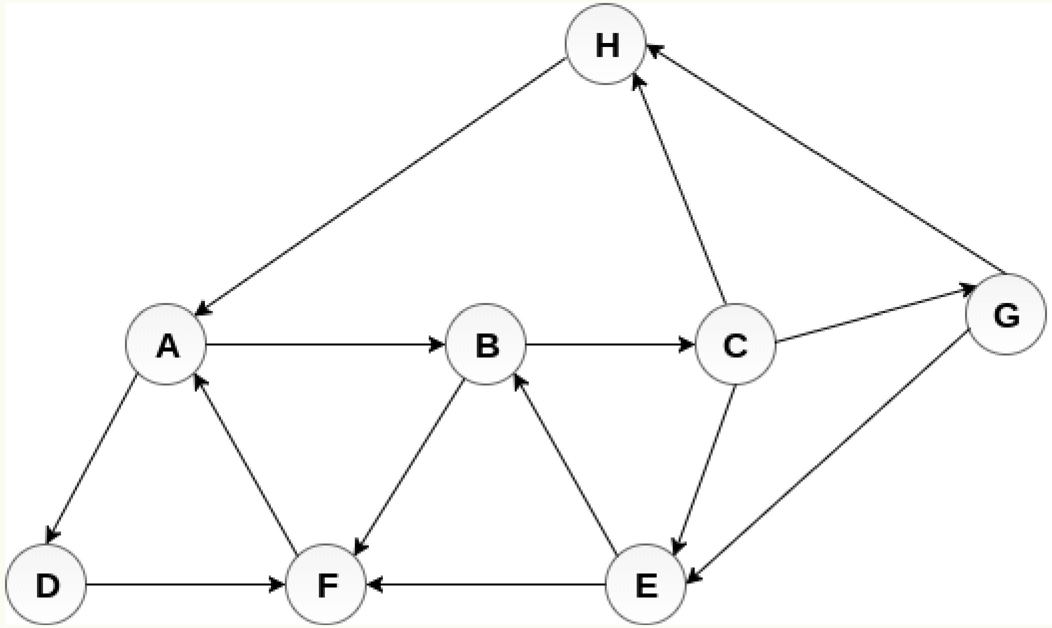


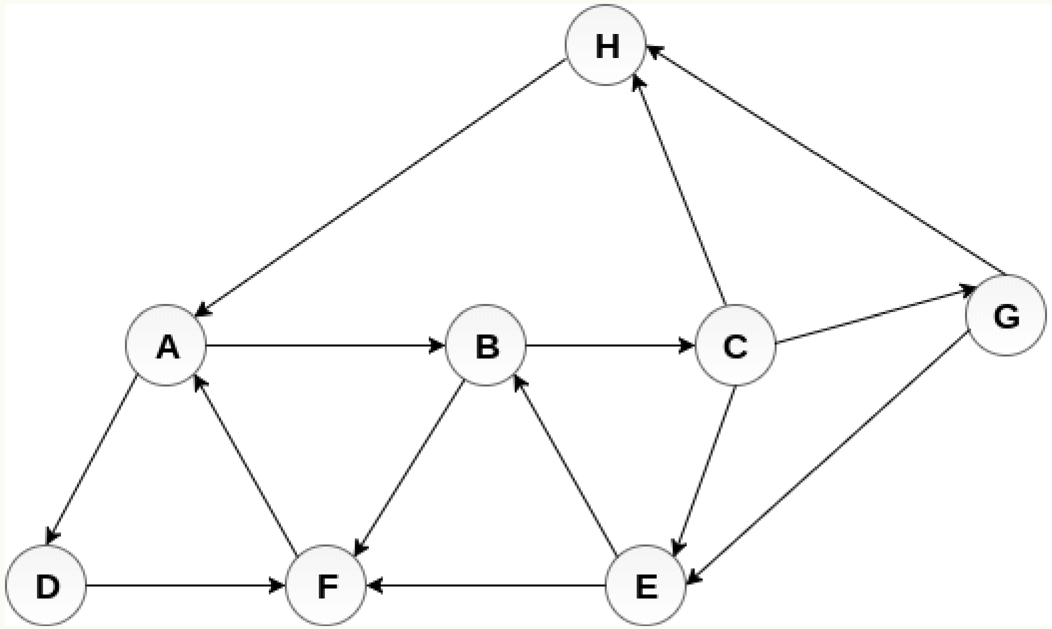
0	1	2			Visited
4	3				Stack


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



0	1	2	4	3	**Visited**
					Stack


H
A
B
C
G
D
F
E



Adjacency Lists

A : B, D

B : C, F

C : E, G, H

G : E, H

E : B, F

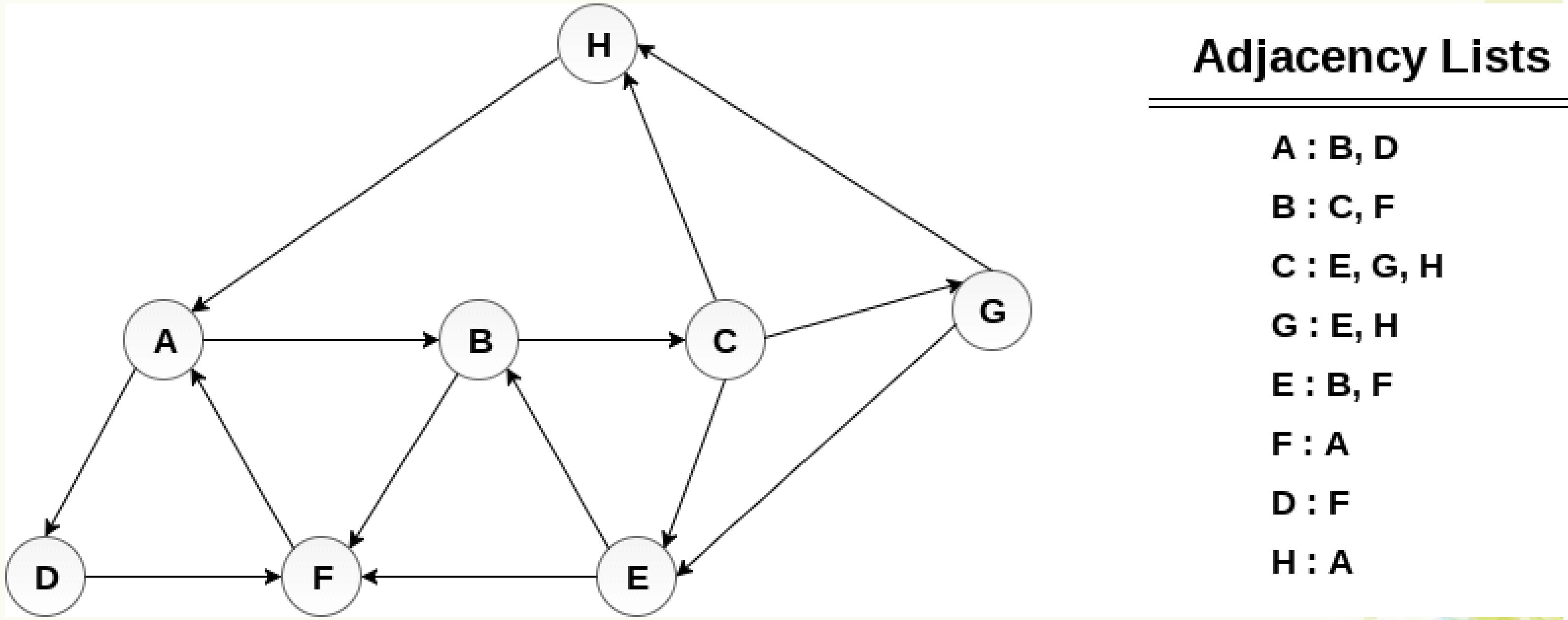
F : A

D : F

H : A

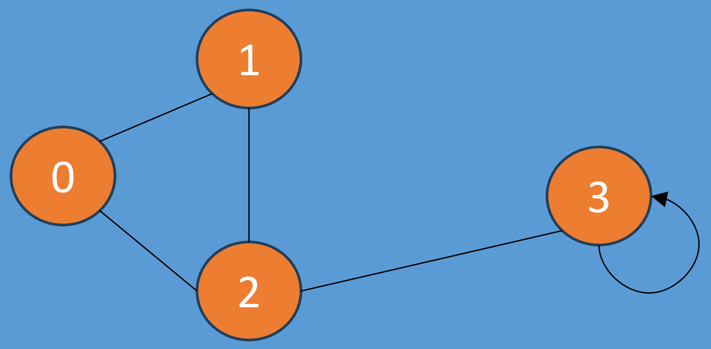
How work DFS

- STACK : H
- STACK : A
- Stack : B, D
- Stack : B, F
- Stack : B
- Stack : C
- Stack : E, G
- Stack : E
-



```python
# Python3 program to print DFS traversal# from a given graph
from collections import defaultdict
# This class represents a directed graph using adjacency list representation
class Graph:
	# Constructor
	def __init__(self):
		# Default dictionary to store graph
		self.graph = defaultdict(list)
	# Function to add an edge to graph
	def addEdge(self, u, v):
		self.graph[u].append(v)
	# A function used by DFS
	def DFSUtil(self, v, visited):
		# Mark the current node as visited and print it
		visited.add(v)
		print(v, end=' ')
		# Recur for all the vertices adjacent to this vertex
		for neighbour in self.graph[v]:
			if neighbour not in visited:
				self.DFSUtil(neighbour, visited)
	# The function to do DFS traversal. It uses
	# recursive DFSUtil()
	def DFS(self, v):
		# Create a set to store visited vertices
		visited = set()
		# Call the recursive helper function to print DFS traversal
		self.DFSUtil(v, visited)
```

```python
# Driver's code
if __name__ == "__main__":
	g = Graph()
	g.addEdge(0, 1)
	g.addEdge(0, 2)
	g.addEdge(1, 2)
	g.addEdge(2, 0)
	g.addEdge(2, 3)
	g.addEdge(3, 3)
	print("Following is DFS (starting f 2)")
	# Function call
	g.DFS(2)
```



Depth-First vs. Breadth-First

- Depth-first goes off into one branch until it reaches a leaf node
  - Not good if the goal is on another branch
  - Uses much less space than breadth-first
- Breadth-first is more careful by checking all alternatives
  - Very memory-intensive
  - For a large tree, breadth-first search memory requirements maybe excessive
  - For a large tree, a depth-first search may take an excessively long time to find even a very nearby goal node.

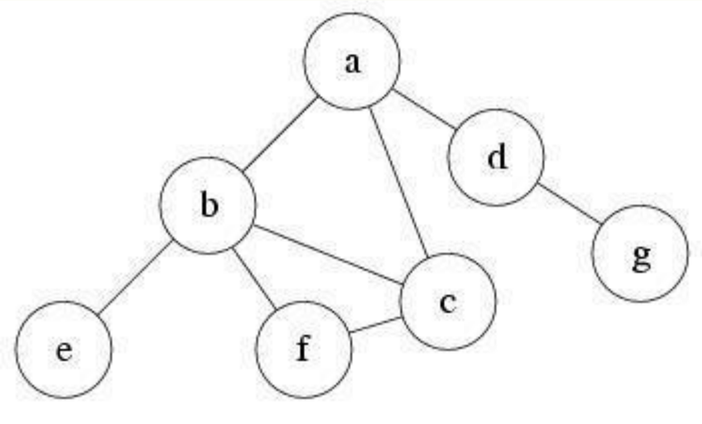
Depth-Limited Search

- Similar to depth-first, but with a limit
  - i.e., nodes at depth l have no successors
  - Overcomes problems with infinite paths
  - Sometimes a depth limit can be estimated from the problem description
- In other cases, a good depth limit is only known when the problem is solved

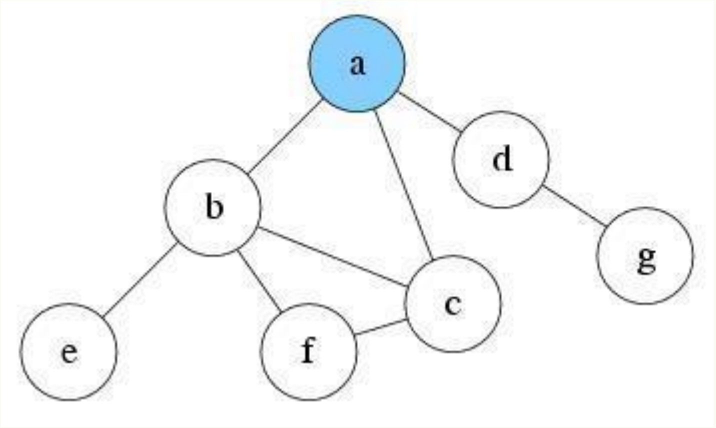
– must keep track of the depth

- Same as DFS, we use the stack data structure S1 to record the node we've explored. Suppose the source node is node a.

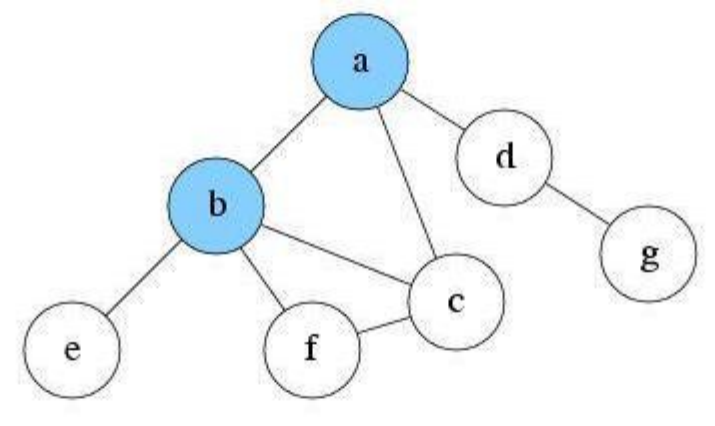
what will happen if we use DLS with L=1.



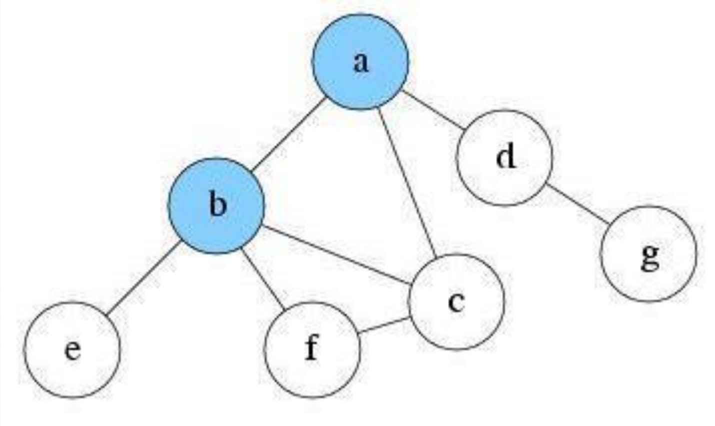
- **S1:**
- At first, the only reachable node is a. So push it into S1 and mark as visited. Current level is 0.



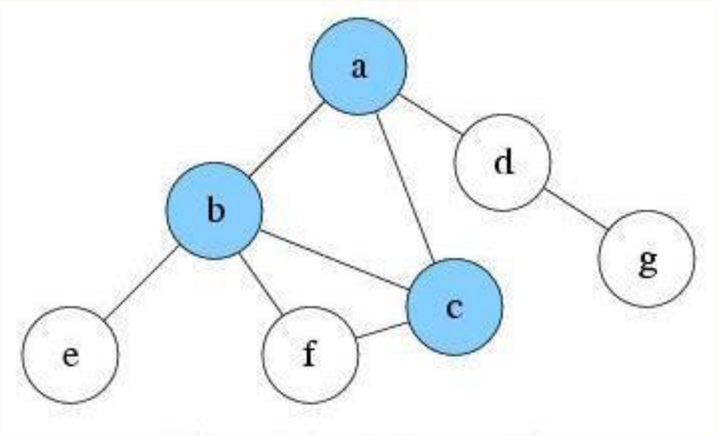
- **S1: a**
- After exploring a, now there are three nodes reachable: node b, c and d. Suppose we pick node b to explore first. Push b into S1 and mark it as visited. Current level is 1.



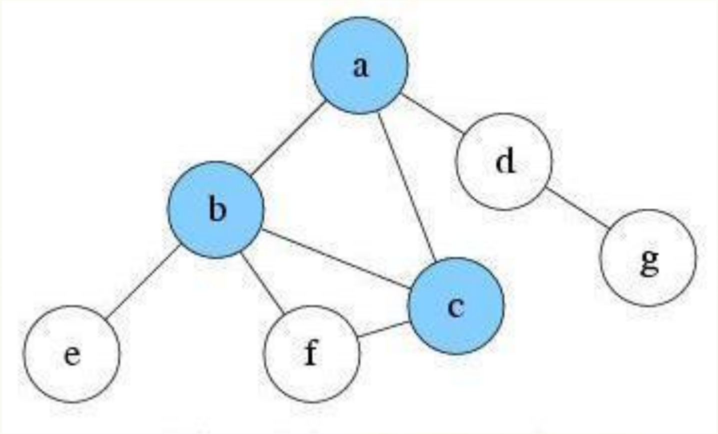
- **S1: b, a**
- Since current level is already the max depth L. Node b will be treated as having no successor. So, there is nothing reachable. Pop b from S1. Current level is 0.



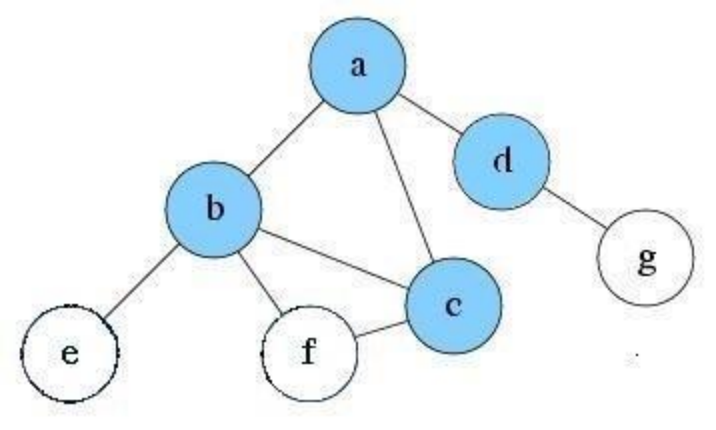
- **S1: a**
- Explore a again. There are two unvisited nodes c and d that are reachable. Suppose we pick node c to explore first. Push c into S1 and mark it as visited. Current level is 1.



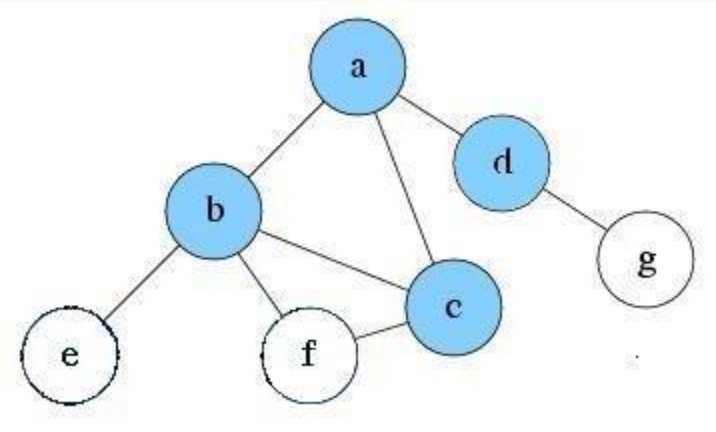
- **S1: c, a**
- Since current level is already the max depth L. Node c will be treated as having no successor. So there is nothing reachable. Pop c from S1. Current level is 0.



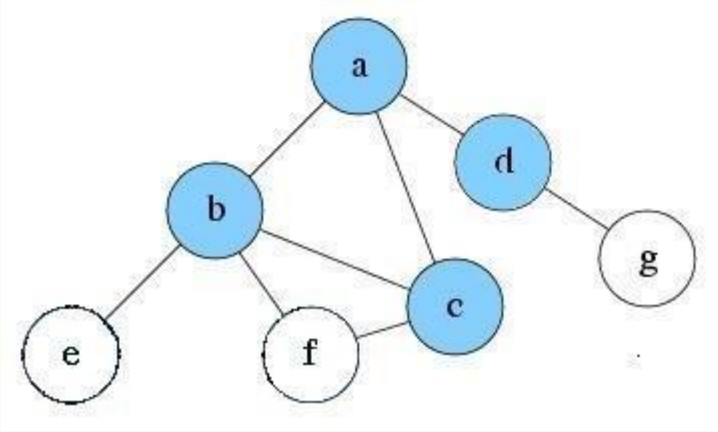
- **S1: a**
- Explore a again. There is only one unvisited node d reachable. Push d into S1 and mark it as visited. Current level is 1.



- **S1: d, a**
- Explore d and find no new node is reachable. Pop d from S1. Current level is 0.

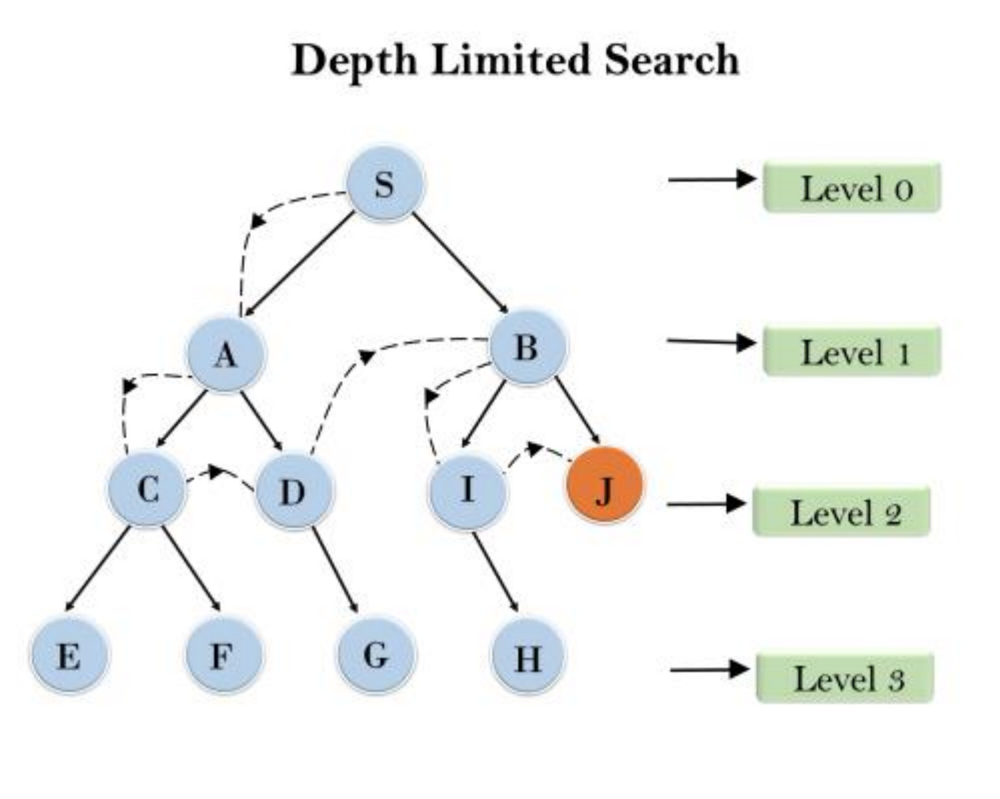


- **S1: a**
- Explore a again. No new reachable node. Pop a from S1



- **S1: d, a**
- Explore d and find no new node is reachable. Pop d from S1. Current level is 0.

Example


Depth Limited Search
S
A
B
C
D
I
J
E
F
G
H
Level 0
Level 1
Level 2
Level 3

Homework

- Iterative deepening depth-first Search:
- Bidirectional Search Algorithm: