Section Content:

- **Search Algorithms**

Search Algorithms

Search algorithms in AI are used to find solutions or paths in a problem space.

They are divided into two main types

- **Uninformed (blind/exhaustive) search:** use no information about problem
- **Informed (heuristic) search:** use information to guide search

Uninformed (Blind) Search

Breadth-First Search (BFS): Queue FIFO

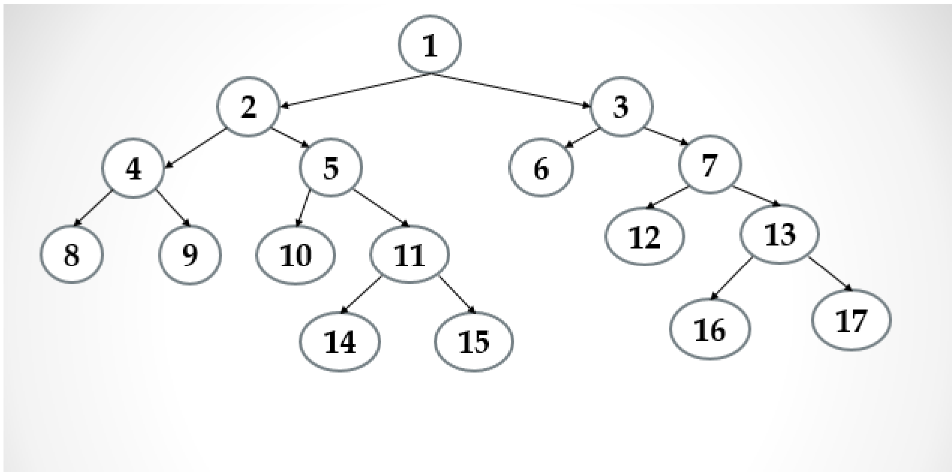
- **Explores** the search tree level by level.
- **How it works:** Starts at the root node and explores all its neighbors before moving to the next level of neighbors.

Depth-First Search (DFS): Stack LIFO

- **Explores** as far as possible along a branch before backtracking.
- **How it works:** Starts at the root node and follows one path to the deepest node before backtracking and trying the next unexplored path.

Example

- **<u>First problem</u>**



Initial: 1 goal: 11 use: BFS , DFS to find path. (in order Ascending)

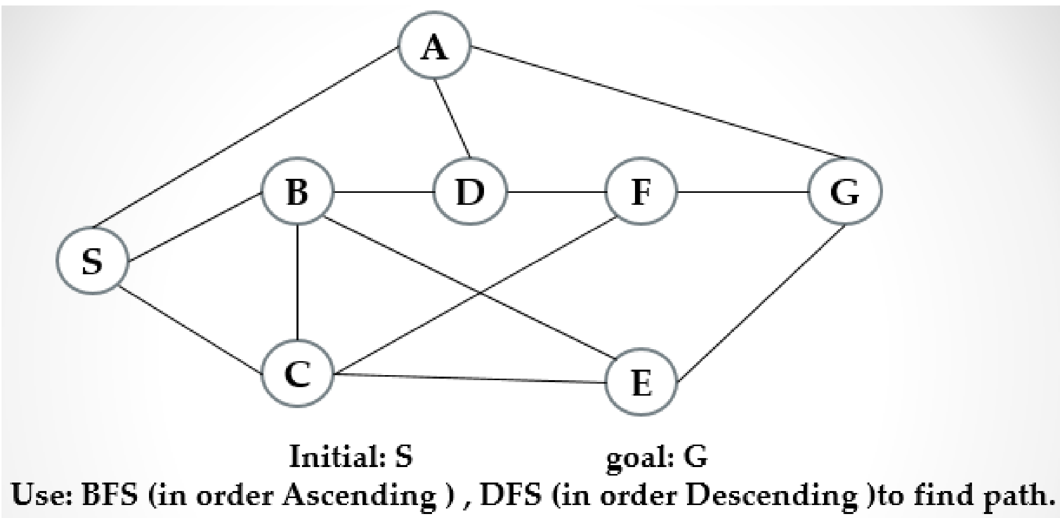
Solution use BFS

Open	Close
1	1
2,3	1,2
3,4,5	1,2,3
4,5,6,7	1,2,3,4
5,6,7,8,9	1,2,3,4,5
6,7,8,9,10,11	1,2,3,4,5,6
7,8,9,10,11	1,2,3,4,5,6,7
8,9,10,11,12,13	1,2,3,4,5,6,7,8
9,10,11,12,13	1,2,3,4,5,6,7,8,9
10,11,12,13	1,2,3,4,5,6,7,8,9,10
11,12,13	1,2,3,4,5,6,7,8,9,10,11

Solution use DFS

Open	Close
1	1
2,3	1,2
4,5,3	1,2,4
8,9,5,3	1,2,4,8
9,5,3	1,2,4,8,9
5,3	1,2,4,8,9,5
10,11,3	1,2,4,8,9,5,10
11,3	1,2,4,8,9,5,10,11

- **<u>Second problem</u>**



Solution use BFS

Open	Close
S	S
A,B,C	S,A
B,C,D,G	S,A,B
C,D,G,E	S,A,B,C
D,G,E,F	S,A,B,C,D
G,E,F	S,A,B,C,D,G

Solution use DFS

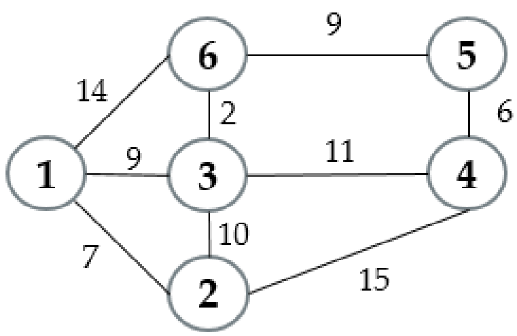
Open	Close
S	S
C,B,A	S,C
F,E,C,B,A	S,C,F
G,D,E,C,B,A	S,C,F,G

Uniform Cost Search (UCS):

- **Explores** nodes based on the lowest path cost.
- **How it works:** Similar to BFS, but prioritizes nodes with the lowest cumulative cost from the start node.

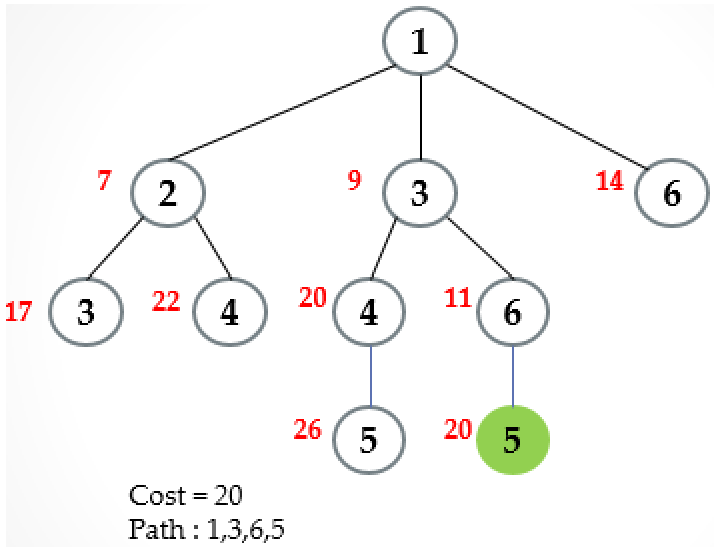
Example

- **First problem**

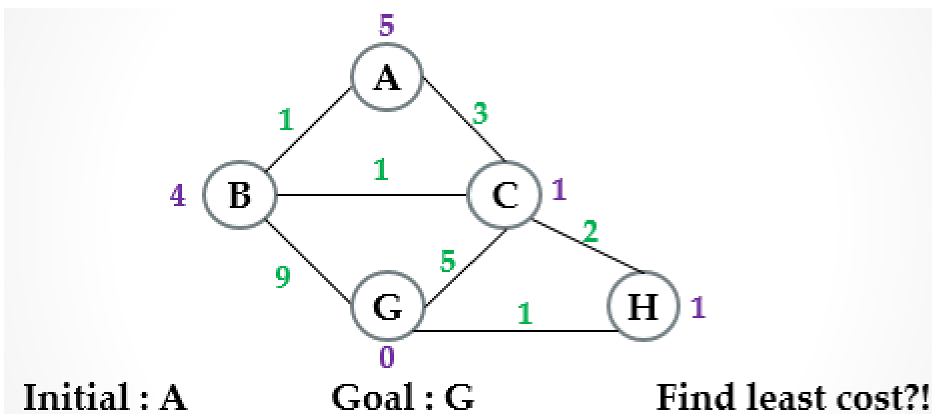


Initial : 1 **Goal : 5** **Find least cost?!**

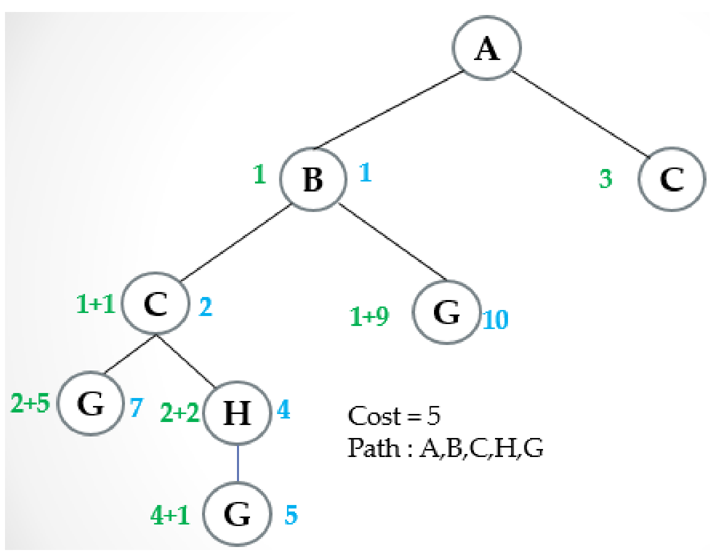
Solution



- **Second problem**



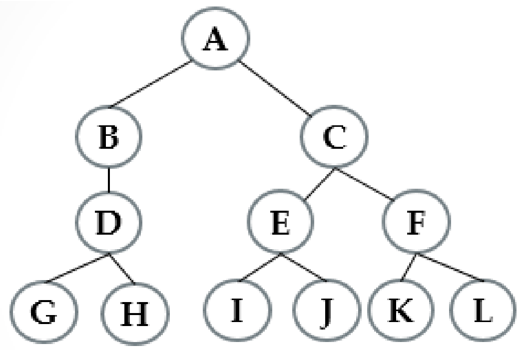
Solution



Iterative Deeping Depth first search

Example

- **First problem**



Initial : A   Goal : J

Solution

0

Open	Close
A	A

1

A	A
B,C	A,B
C	A,B,C

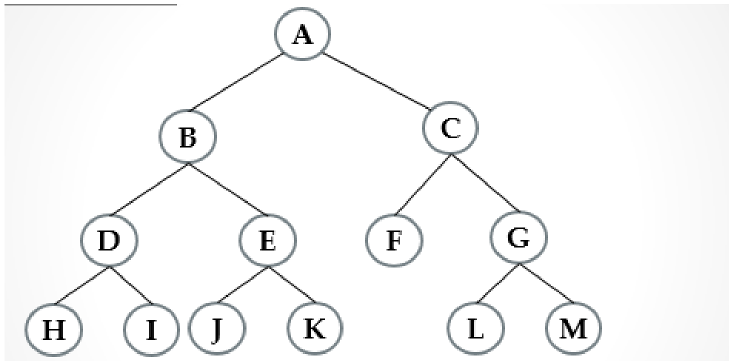
2

A	A
B,C	A,B
D,C	A,B,D
C	A,B,D,C
E,F	A,B,D,C,E
F	A,B,D,C,E,F

3

Open	Close
A	A
B,C	A,B
D,C	A,B,D
G,H,C	A,B,D,G
H,C	A,B,D,G,H
C	A,B,D,G,H,C
E,F	A,B,D,G,H,C,E
I,J,F	A,B,D,G,H,C,E,I
J,F	A,B,D,G,H,C,E,I,J

• Second problem



Initial : A Goal : G,L Find least cost?!

Solution

0

Open	Close
A	A

1

A	A
B,C	A,B
C	A,B,C

2

A	A
B,C	A,B
D,E,C	A,B,D
E,C	A,B,D,E
C	A,B,D,E,C
F,G	A,B,D,E,C,F
G	A,B,D,E,C,F,G

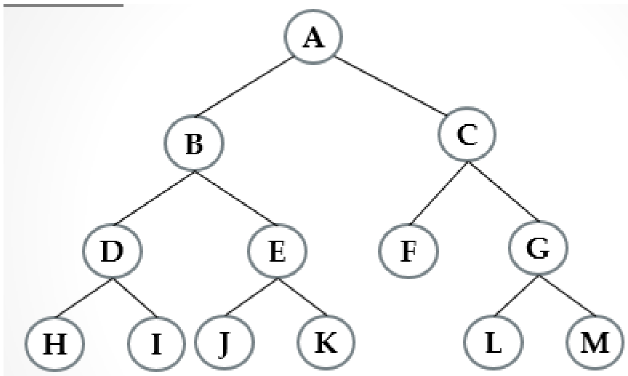
3

Open	Close
A	A
B,C	A,B
D,E,C	A,B,D
H,I,E,C	A,B,D,H
I,E,C	A,B,D,H,I
E,C	A,B,D,H,I,E
J,K,C	A,B,D,H,I,E,J
K,C	A,B,D,H,I,E,J,K
C	A,B,D,H,I,E,J,K,C
F,G	A,B,D,H,I,E,J,K,C,F
G	A,B,D,H,I,E,J,K,C,F,G
L,M	A,B,D,H,I,E,J,K,C,F,G,L

Limit Depth first search

Example

- **First problem**



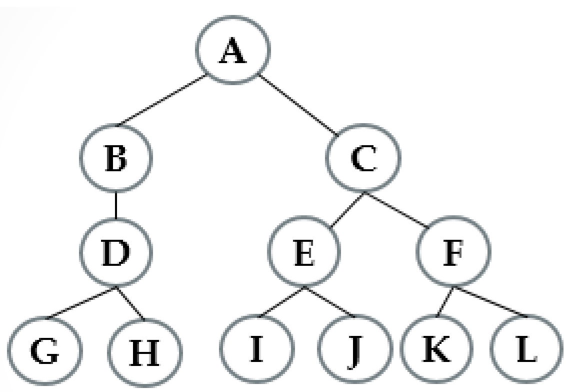
Initial : A Goal : G,L cut off/limit = 2

Solution

Open	Close
A	A
B,C	A,B
D,E,C	A,B,D
E,C	A,B,D,E
C	A,B,D,E,C
F,G	A,B,D,E,C,F
G	A,B,D,E,C,F,G

Path(G): A,B,D,E,C,F,G
Path(L): Goal Not found

- **Second problem**



Initial : A Goal : F,J limit=3

Solution

Open	Close
A	A
B,C	A,B
D,E,C	A,B,D
E,C	A,B,D,E
C	A,B,D,E,C
F,G	A,B,D,E,C,F
G	A,B,D,E,C,F,G

Path(G): A,B,D,E,C,F,G
Path(L): Goal Not found

Code with python

Breadth-First Search (BFS): Queue FIFO

```python
# Import deque for an efficient queue implementation
from collections import deque
def bfs(graph, start_node, goal):
    # Initialize the queue with the starting node
    queue = deque([start_node])
    # Set to track visited nodes to avoid revisiting them
    visited = set([start_node])
    # Continue while there are nodes in the queue
    while queue:
        # Dequeue a node from the front of the queue
        node = queue.popleft()
        # Process the current node (here we simply print it)
        print(node, end=' ')
        if node == goal:
            break
        # Iterate over all the neighboring nodes of the current node
        for neighbor in graph[node]:
            # If the neighbor has not been visited
            if neighbor not in visited:
                queue.append(neighbor) # Add to queue to explore later
                visited.add(neighbor)  # Mark as visited to avoid revisiting
graph = {
    'A': ['B', 'C'],  # Node A connects to B and C
    'B': ['D', 'E'],  # Node B connects to D and E
    'C': ['F'],       # Node C connects to F
    'D': [],          # Node D has no neighbors
    'E': ['F'],       # Node E connects to F
    'F': []           # Node F has no neighbors
}
```

```python
# Start BFS traversal from node 'A'
bfs(graph, 'A', 'F')
```

Depth-First Search (DFS): Stack LIFO

```python
def dfs(graph, start_node, goal):
    # Initialize the stack with the starting node
    stack = [start_node]
    # Set to track visited nodes to avoid revisiting them
    visited = set([start_node])
    # Continue while there are nodes in the stack
    while stack:
        # Pop a node from the top of the stack (LIFO behavior)
        node = stack.pop()
        # Process the current node (here we simply print it)
        print(node, end=' ')
        if node == goal:
            break
        # Iterate over the neighboring nodes of the current node in reverse order
        # This ensures that nodes are visited in the correct order
        for neighbor in reversed(graph[node]):
            # If the neighbor has not been visited
            if neighbor not in visited:
                stack.append(neighbor)  # Add to stack to explore later
                visited.add(neighbor)   # Mark as visited to avoid revisiting
# Example graph represented as an adjacency list
graph = {
    'A': ['B', 'C'],  # Node A connects to B and C
    'B': ['D', 'E'],  # Node B connects to D and E
    'C': ['F'],       # Node C connects to F
    'D': [],          # Node D has no neighbors
    'E': ['F'],       # Node E connects to F
    'F': []           # Node F has no neighbors
}
# Start DFS traversal from node 'A'
dfs(graph, 'A', 'F')
```

Uniform Cost Search (UCS):

```
import heapq  # Import heapq to use a priority queue
def ucs(graph, start_node, goal_node):
    # Priority queue (min-heap) initialized with the start node and its cost (0)
    priority_queue = [(0, start_node)]  # (cost, node)
    # Dictionary to store the minimum cost to reach each node
    visited = {start_node: 0}
    while priority_queue:
        # Pop the node with the lowest cost from the priority queue
        current_cost, current_node = heapq.heappop(priority_queue)
        # If the goal is reached, return the cost
        if current_node == goal_node:
            return current_cost
        # Explore neighbors of the current node
        for neighbor, cost in graph[current_node]:
            new_cost = current_cost + cost
            # If the new path to the neighbor is cheaper, update the path and
push to the queue
            if neighbor not in visited or new_cost < visited[neighbor]:
                visited[neighbor] = new_cost  # Record the cheaper cost
# Add to priority queue
                heapq.heappush(priority_queue, (new_cost, neighbor))
    # If the goal is not reachable, return infinity
    return float('inf')
# Example graph represented as an adjacency list (node, cost)
graph = {
    'A': [('B', 1), ('C', 4)],  # Node A connects to B (cost 1) and C (cost 4)
    'B': [('D', 2), ('E', 5)],  # Node B connects to D (cost 2) and E (cost 5)
    'C': [('F', 1)],            # Node C connects to F (cost 1)
    'D': [],                    # Node D has no neighbors
    'E': [('F', 3)],            # Node E connects to F (cost 3)
    'F': []                     # Node F has no neighbors
}
# Start UCS from node 'A' to node 'F'
cost = ucs(graph, 'A', 'F')
print("Minimum cost from A to F:", cost)
```

Iterative Deeping Depth first search

```
graph = {
    'A': ['B', 'C', 'D'],  # Node A connects to B, C, and D
    'B': ['E', 'F'],       # Node B connects to E and F
    'C': ['G'],            # Node C connects to G
    'D': [],               # Node D has no neighbors
    'E': [],               # Node E has no neighbors
    'F': []                # Node F has no neighbors
}
# Depth-Limited DFS function
def DFS(currentNode, destination, graph, maxDepth, path):
    # Add the current node to the path
    path.append(currentNode)
    # Print the current path to show progress
    print(path)
    # Check if we have reached the destination
    if currentNode == destination:
        return True  # Destination found
    # Check if maxDepth has been reached
    if maxDepth <= 0:
        return False  # Stop exploring further as the depth limit is reached
    # Recursively explore each neighbor of the current node
    for node in graph[currentNode]:
        # Recur with reduced maxDepth
        if DFS(node, destination, graph, maxDepth - 1, path):
            return True  # If destination is found in the recursion, return True
    # If destination is not found, return False
    return False
# Iterative Deepening Depth-First Search (IDDFS) function
def iterativeDDFS(currentNode, destination, graph, maxDepth):
    # Iteratively increase depth from 0 to maxDepth
    for i in range(maxDepth):
        # Print the current depth level being explored
        print("Iterative Depth Level:", i)
        # Initialize a fresh path for each depth
        path = []
        # Call DFS with the current depth limit (i)
        if DFS(currentNode, destination, graph, i, path):
```

```
            return True  # If DFS finds the destination, return True
    # If destination is not found after exploring all depths, return False
    return False
# Calling the iterative deepening DFS to find a path from 'A' to 'F'
if not iterativeDDFS('A', 'F', graph, 4):
    print("Path is not available")  # If no path is found within the max depth
else:
    print("A path exists")  # If a path to the destination is found
```

Limit Depth first search

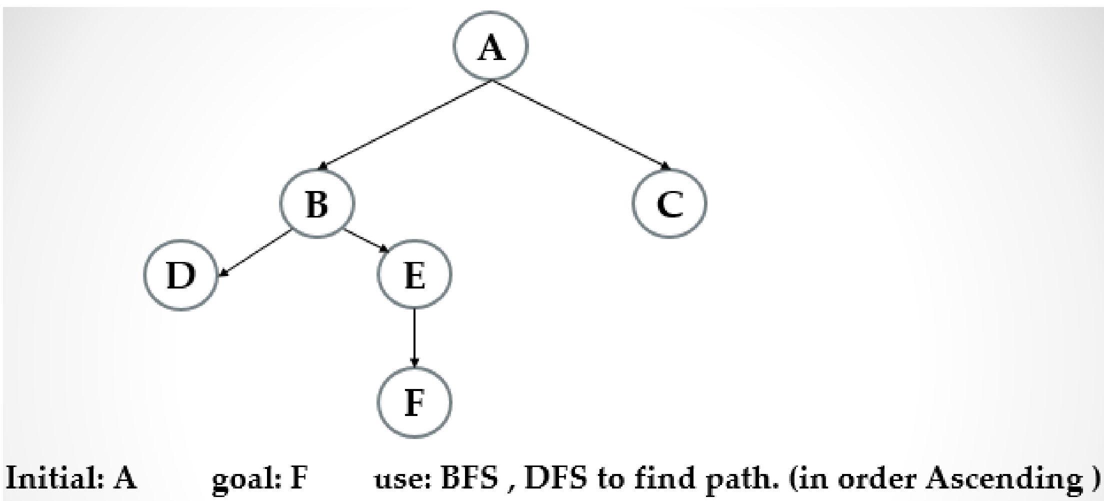
```
graph = {
    'A': ['B', 'C', 'D'],  # Node A connects to B, C, and D
    'B': ['E', 'F'],       # Node B connects to E and F
    'C': ['G'],            # Node C connects to G
    'D': [],               # Node D has no neighbors
    'E': [],               # Node E has no neighbors
    'F': []                # Node F has no neighbors
}
# Depth-Limited Search (DLS) function
def DLS(start, goal, path, Level, maxD):
    # Add the current node to the path
    path.append(start)
    # Check if the current node is the goal node
    if start == goal:
        return path  # If goal is found, return the path
    # Check if the maximum depth limit has been reached
    if Level == maxD:
        return False  # If depth limit is reached, stop searching further
    # Recursively explore each child node
    for child in graph[start]:
        # Recur with an incremented level (depth)
        if DLS(child, goal, path, Level + 1, maxD):
            return path  # If goal is found in the recursion, return the path
    # If the goal is not found, backtrack by returning False
    return False
# Starting node
start = 'A'
# Taking user input for the goal node
goal = input('Enter the goal node: ')
# Taking user input for the maximum depth limit
maxD = int(input("Enter the maximum depth limit: "))
# Empty list to store the path from start to goal
path = list()
```

```python
# Call the Depth-Limited Search function with initial depth (Level = 0)
res = DLS(start, goal, path, 0, maxD)


# Check the result of the search
if res:
    print("Path to goal node available")
    print("Path:", path)  # Print the path if the goal node is found
else:
    print("No path available for the goal node in given depth limit")
```

Assignment

- **First**



- **Second:** uniform cost