Artificial Intelligence

BS (CS) _SP_2024

# Lab_05 Manual

## Learning Objectives:

1. Uninformed searches
2. Breadth-First-Search(BFS)
3. Depth-First-Search(DFS)
4. Tree Search

# Lab Manual
## Uninformed Searches

## Uninformed Search:

Uninformed search is a class of general-purpose search algorithms.Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree, so it is also called **blind** search.They operate in a brute force, meaning they try out every part of search space blindly.

### Types:

1. Breadth-first Search
2. Depth-first Search
3. Depth-limited Search
4. Iterative deepening depth-first search
5. Uniform cost search

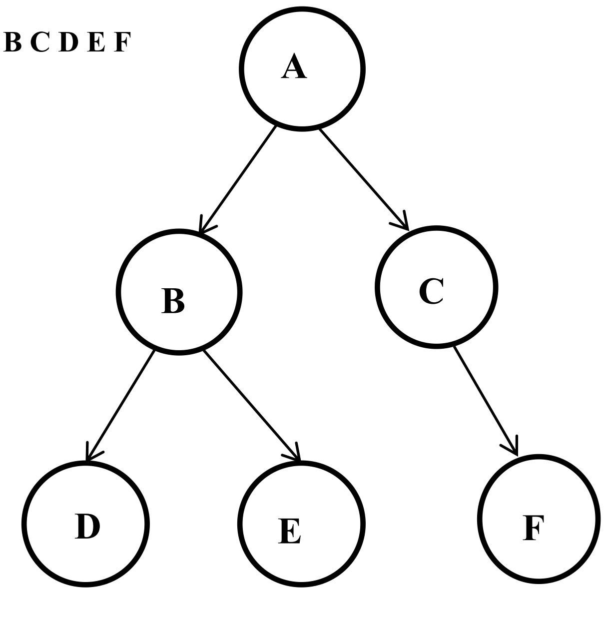
### Breadth-First-Search:

- BFS explores all the neighbor nodes at the present depth before moving on to the nodes at the next depth level.
- It guarantees the shortest path from the starting node to the goal node in an unweighted graph.
- It uses a queue data structure to keep track of the nodes to be explored next.

**Example:**

**BFS Traversal: A B C D E F**

### Code:



```python
from collections import deque

graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            queue.extend(graph[node])

print("BFS Traversal:")
bfs(graph, 'A')
```

```
BFS Traversal:
A B C D E F
```

### Example: to find shortest path in BFS

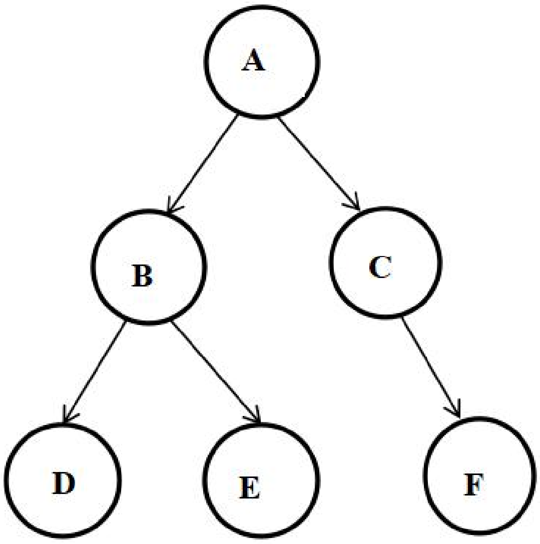
```python
from collections import deque

def shortest_path_bfs(graph, start, target):
    # Initialize a queue for BFS traversal
    queue = deque([(start, [start])])
    visited = set([start])

    # Perform BFS traversal
    while queue:
        node, path = queue.popleft()
        if node == target:
            return path
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, path + [neighbor]))

# Example graph represented as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

# Find the shortest path from node 'A' to node 'F' using BFS
shortest_path = shortest_path_bfs(graph, 'A', 'F')
print("Shortest Path (BFS):", shortest_path)
```

## Depth-First-Search:

- DFS explores as far as possible along each branch before backtracking.
- It may not find the shortest path to the goal node.
- It uses a stack data structure (or recursion) to keep track of the nodes to be explored next.

**Example:**

DFS Traversal: A B D E F C




## Code:

```
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

def dfs(graph, node, visited=None):
    if visited is None:
        visited = set()
    visited.add(node)
    print(node, end=" ")
    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

print("DFS Traversal:")
dfs(graph, 'A')
```

```
DFS Traversal:
A B D E F C
```

**Example: to find shortest path in DFS**

```python
def dfs_paths(graph, start, target, path=None):
    if path is None:
        path = [start]
    if start == target:
        return [path]
    paths = []
    for neighbor in graph[start]:
        if neighbor not in path:
            new_path = dfs Loading... ph, neighbor, target, path + [neighbor])
            paths.extend(new_path)
    return paths

# Example
all_paths = dfs_paths(graph, 'A', 'F')
# Find the shortest path among all paths found by DFS
shortest_path_dfs = min(all_paths, key=len)
print("Shortest Path (DFS):", shortest_path_dfs)
```

```
Shortest Path (DFS): ['A', 'C', 'F']
```

## Tree-Search:

- Tree search is a general framework that combines a search strategy (like BFS or DFS) with a data structure to explore the search space.
- It can be used with informed search algorithms by incorporating heuristics to guide the search.
- The choice of search strategy and data structure determines the efficiency and effectiveness of the search algorithm.

**Example:**
Now, let's illustrate BFS, DFS, and Tree Search with an example in Python:

```python
# Example Graph represented as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
```

```python
# Breadth-First Search (BFS)
def bfs(graph, start):
    visited = set()
    queue = [start]
    while queue:
        node = queue.pop(0)
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            queue.extend(graph[node])

# Depth-First Search (DFS)
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=" ")
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

# Example usage
print("BFS:")
bfs(graph, 'A')
print("\nDFS:")
dfs(graph, 'A')
```

**Output:**

```
BFS:
A B C D E F
DFS:
A B D E F C
```

**Example: to find shortest path using tree search:**

```python
from collections import deque

class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []

def shortest_path_tree_search(root, target):
    # Initialize a queue for BFS traversal
    queue = deque([(root, [root])])

    # Perform BFS traversal
    while queue:
        node, path = queue.popleft()
        if node.value == target:
            return path
        for child in node.children:
            queue.append((child, path + [child.value]))

# Example tree structure
#         A
#       / | \
#      B  C  D
#     / \    |
#    E   F   G
root = TreeNode('A')
root.children = [TreeNode('B'), TreeNode('C'), TreeNode('D')]
root.children[0].children = [TreeNode('E'), TreeNode('F')]
root.children[2].children = [TreeNode('G')]
```

```python
# Find the shortest path from node 'A' to node 'G' using tree search (BFS)
shortest_path = shortest_path_tree_search(root, 'G')
print("Shortest Path (Tree Search):", shortest_path)
```

```
Shortest Path (Tree Search): [<__main__.TreeNode object at 0x7bca43309420>, 'D', 'G']
```