

EXPERIMENT NO.: 04

AIM: To implement Depth First Search and Breadth First Search algorithm in AI.

OBJECTIVES: A] To explore and traverse through a graph in a way that prioritizes deeper paths, aiming to reach the deepest nodes of the graph before backtracking, with the goal of finding a specific node or discovering the structure of the graph.

B] To traverse through a graph in the smallest number of iterations.

THEORY:

Depth First Search (DFS) is a fundamental graph traversal algorithm that explores as far as possible along a branch before backtracking. It is used in a variety of applications and has its own set of advantages and disadvantages.

Explanation:

DFS is a recursive algorithm that starts at the root node and explores as far as possible along each branch before backtracking. It follows the idea of depth-first traversal, meaning it goes as deep as possible into the graph before exploring other branches. The algorithm is typically implemented using a stack or recursion to keep track of nodes to visit.

Need:

Graph Traversal: DFS is used to visit all nodes in a graph, which is essential in various graph-based problems such as finding paths, cycles, or connected components.

Maze Solving: DFS can be used to solve mazes by exploring paths and backtracking when a dead end is reached.

Topological Sorting: It's used for topological sorting in directed acyclic graphs, an essential step in scheduling tasks or dependencies.

Searching: DFS can be employed in searching and puzzle-solving problems like the N-Queens puzzle or the Eight Puzzle.

Advantages:

Simplicity: DFS is relatively easy to implement, both recursively and iteratively.

Memory Efficiency: It typically uses less memory compared to Breadth-First Search (BFS) as it only requires enough memory to store the path from the root to the current node.

Use in Pathfinding: In certain situations, DFS can be more efficient for finding specific paths, particularly if the goal is deep within the graph.

Disadvantages:

Completeness: DFS does not guarantee finding the shortest path if one exists. It may get stuck exploring deeper paths while a shallower, shorter path remains unexplored.

Non-optimal Solutions: In certain cases, it may return suboptimal solutions or take longer to reach a solution due to its depth-first nature.

Stack Overflow: Recursive DFS may run into stack overflow errors when working with very deep graphs, and iterative DFS using a stack may require additional memory.

Applications:

Maze Solving: DFS is used to solve mazes and find the way from the start to the exit.

Graph Connectivity: It is used to find connected components in a graph.

Topological Sorting: In project scheduling, task dependencies, and data flow analysis.

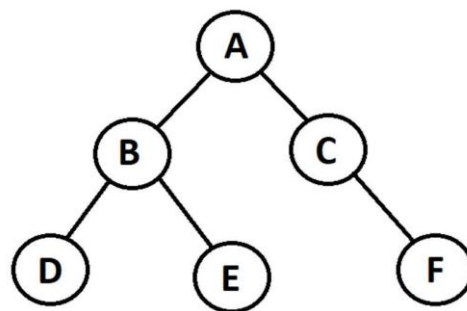
Game Development: Used for game state exploration and AI decision-making.

Compiler Design: DFS is used in parsing and code optimization.

Network Routing: In finding routes in computer networks.

AI and Machine Learning: DFS can be used to explore decision trees and search for optimal strategies.

Example explanation:



Start at node 'A' because 'A' is the starting node.

Mark 'A' as visited and print 'A'.

Explore its neighbours, 'B' and 'C'. Choose the leftmost neighbour first ('B').

Start at node 'A' because 'A' is the starting node.

Mark 'A' as visited and print 'A'.

Explore its neighbours, 'B' and 'C'. Choose the le most neighbour first ('B').

Visited: {A}

Stack: ['B']

Output: A

Start at 'B'. Mark 'B' as visited and print 'B'.

Explore its neighbours, 'D' and 'E'. Choose the le most neighbour first ('D').

Visited: {A, B}

Stack: ['D']

Output: A -> B

Start at 'D'. Mark 'D' as visited and print 'D'. 'D' has no unvisited neighbours, so backtrack to 'B'.

Visited: {A, B, D}

Stack: []

Output: A -> B -> D

Continue exploring 'B' - now choose 'E'. Mark 'E' as visited and print 'E'. 'E' has an unvisited neighbour 'F', so go to 'F'.

Visited: {A, B, D, E}

Stack: ['F']

Output: A -> B -> D -> E

Start at 'F'. Mark 'F' as visited and print 'F'. 'F' has no unvisited neighbours, so backtrack to 'E'.

Visited: {A, B, D, E, F}

Stack: []

Output: A -> B -> D -> E -> F

Since 'E' has been completely explored, backtrack to 'B'.

Visited: {A, B, D, E, F}

Stack: ['C']

Output: A -> B -> D -> E -> F

Continue exploring 'B' and now explore 'C'. Mark 'C' as visited and print 'C'.

Visited: {A, B, D, E, F, C}

Stack: []

Output: A -> B -> D -> E -> F -> C

'C' has no unvisited neighbours, so we've now fully explored the graph. The DFS is complete.

The final DFS traversal of the graph starting from 'A' is:

A -> B -> D -> E -> F -> C

This represents the order in which the nodes are visited in a depth-first search.

BFS

Breadth-First Search (BFS) is an algorithm used for traversing graphs or trees. Traversing means visiting each node of the graph. Breadth-First Search is a recursive algorithm to search all the vertices of a graph or a tree. BFS in python can be implemented by using data structures like a dictionary and lists. Breadth-First Search in tree and graph is almost the same. The only difference is that the graph may contain cycles, so we may traverse to the same node again.

The architecture of BFS algorithm

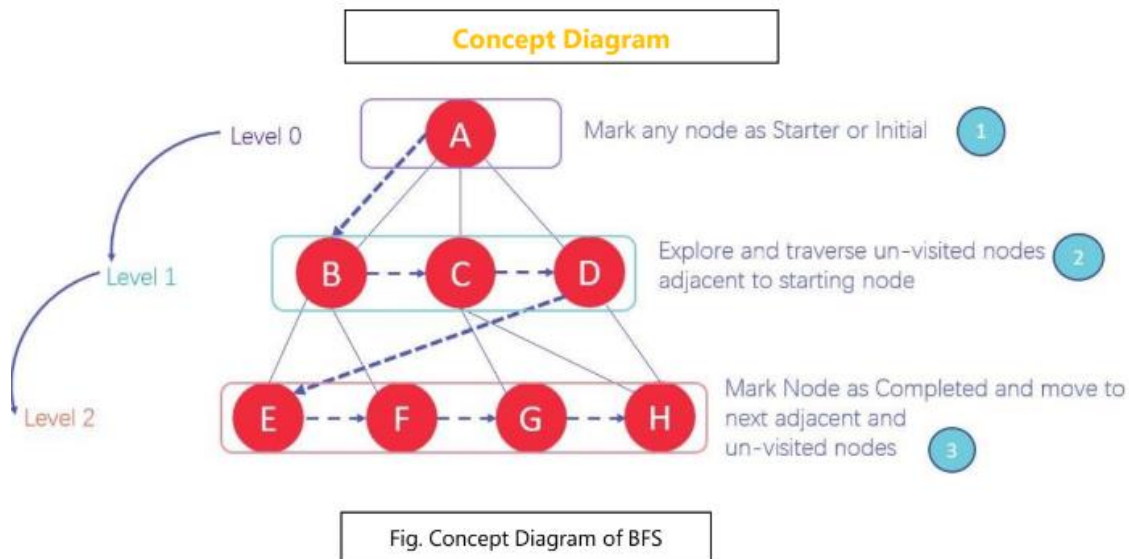


Fig. Concept Diagram of BFS

In the various levels of the data, you can mark any node as the starting or initial node to begin traversing. The BFS will visit the node and mark it as visited and places it in the queue.

Now the BFS will visit the nearest and un-visited nodes and marks them. These values are also added to the queue. The queue works on the FIFO model.

In a similar manner, the remaining nearest and un-visited nodes on the graph are analysed marked and added to the queue. These items are deleted from the queue as receive and printed as the result.

Need OF BFS Algorithm

There are numerous reasons to utilize the BFS Algorithm to use as searching for your dataset. Some of the most vital aspects that make this algorithm your first choice are:

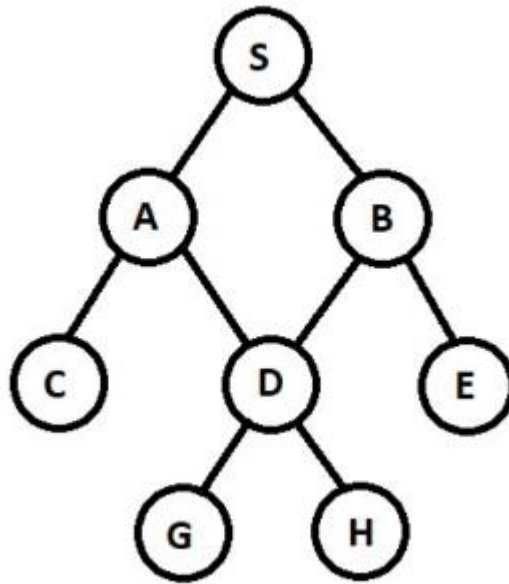
- BFS is useful for analysing the nodes in a graph and constructing the shortest path of traversing through these.
- BFS can traverse through a graph in the smallest number of iterations.
- The architecture of the BFS algorithm is simple and robust.
- The result of the BFS algorithm holds a high level of accuracy in comparison to other algorithms.
- BFS iterations are seamless, and there is no possibility of this algorithm getting caught up in an infinite loop problem.

BFS used in

- In GPS navigation, it helps in finding the shortest path available from one point to another.
- In pathfinding algorithms
- Cycle detection in an undirected graph
- In minimum spanning tree

- To build index by search index
- In Ford-Fulkerson algorithm to find maximum flow in a network.

Example explanation



Initialization:

Visited Queue: []

Waiting Queue: [S]

Step 1: Process Node 'S'

Remove 'S' from the waiting queue.

Add 'S' to the visited queue.

Enqueue 'A' and 'B' into the waiting queue.

Visited Queue: [S] Waiting Queue: [A, B]

Step 2: Process Node 'A'

Remove 'A' from the waiting queue.

Add 'A' to the visited queue.

Enqueue 'C' and 'D' into the waiting queue.

Visited Queue: [S, A] Waiting Queue: [B, C, D]

Step 3: Process Node 'B'

Remove 'B' from the waiting queue.

Add 'B' to the visited queue.

Enqueue 'D' and 'E' into the waiting queue.

Visited Queue: [S, A, B] Waiting Queue: [C, D, D, E]

Step 4: Process Node 'C'

Remove 'C' from the waiting queue.

Add 'C' to the visited queue.

'C' has no unvisited neighbors.

Visited Queue: [S, A, B, C] Waiting Queue: [D, D, E]

Step 5: Process Node 'D'

Remove 'D' from the waiting queue.

Add 'D' to the visited queue.

Enqueue 'G' and 'H' into the waiting queue.

Visited Queue: [S, A, B, C, D] Waiting Queue: [D, E, G, H]

Step 6: Process Node 'D'

Class and Branch: TE AI-DS
Student Roll no.: 41
Student Name: Sharvil M. Palvekar

DOP: 03-09-2024

Remove 'D' from the waiting queue. (Note: This is the duplicate entry.) 'D' has already been visited.

Visited Queue: [S, A, B, C, D] Waiting Queue: [E, G, H]

Step 7: Process Node 'E'

Remove 'E' from the waiting queue.

Add 'E' to the visited queue.

'E' has no unvisited neighbors.

Visited Queue: [S, A, B, C, D, E]

Waiting Queue: [G, H]

Step 8: Process Node 'G'

Remove 'G' from the waiting queue.

Add 'G' to the visited queue.

'G' has no unvisited neighbors.

Visited Queue: [S, A, B, C, D, E, G] Waiting Queue: [H]

Step 9: Process Node 'H'

Remove 'H' from the waiting queue.

Add 'H' to the visited queue.

'H' has no unvisited neighbors.

Visited Queue: [S, A, B, C, D, E, G, H] Waiting Queue: []

BFS traversal order: S, A, B, C, D, E, G, H

The traversal explores nodes level by level. Each node's unvisited neighbors are added to the waiting queue, and nodes are visited in the order they were added to the waiting queue. The visited queue stores the nodes that have already been visited.

PROGRAM:

1) DFS :

Defining the graph using an adjacency list

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

Set to keep track of visited nodes

```
visited = set()
```

Function to perform Depth-First Search

```
def dfs(visited, graph, node):
```

```
    if node not in visited:
```

```
        print(node) # Print the current node
```

```
        visited.add(node) # Mark the node as visited
```

```
        for neighbor in graph[node]: # Recursively visit the neighbors
```

```
            dfs(visited, graph, neighbor)
```

Main block to start DFS from node 'A'

```
print("Following is the Depth-First Search:")
```

```
dfs(visited, graph, 'A')
```

2) BFS :

```
# Defining the graph using an adjacency list
```

```
graph = {  
    'S': ['A', 'B'],  
    'A': ['C', 'D'],  
    'B': ['D', 'E'],  
    'C': [],  
    'D': ['G', 'H'],  
    'E': [],  
    'G': [],  
    'H': []  
}
```

```
# List to keep track of visited nodes
```

```
visited = []
```

```
# List to simulate the queue for BFS
```

```
queue = []
```

```
# Function to perform Breadth-First Search
```

```
def bfs(visited, graph, node):
```

```
    visited.append(node)
```

```
    queue.append(node)
```

```
    while queue:
```

```
        current_node = queue.pop(0) # Dequeue a node
```

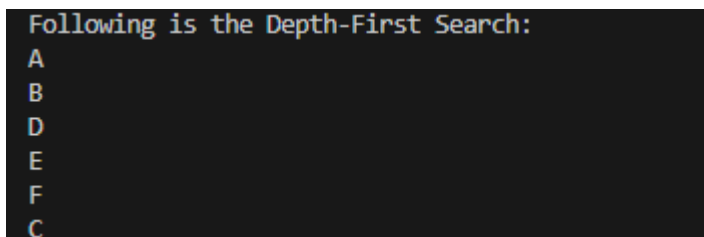
```
print(current_node, end=" ") # Print the current node

# Get all the adjacent nodes (neighbors)
for neighbor in graph[current_node]:
    if neighbor not in visited: # If the neighbor hasn't been visited
        visited.append(neighbor)
        queue.append(neighbor)

# Main block to start BFS from node 'S'
print("Breadth-First Search:")
bfs(visited, graph, 'S')
```

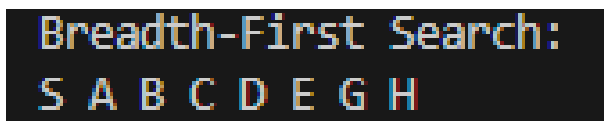
OUTPUT:

1) DFS:-

A screenshot of a terminal window showing the output of a Depth-First Search (DFS) algorithm. The text is as follows:

```
Following is the Depth-First Search:
A
B
D
E
F
C
```

2) BFS:-

A screenshot of a terminal window showing the output of a Breadth-First Search (BFS) algorithm. The text is as follows:

```
Breadth-First Search:
S A B C D E G H
```

OBSERVATION AND LEARNING:

- DFS (Depth-First Search) explores nodes by diving deep into each branch before backtracking, useful for problems requiring exploration of all paths, like mazes or puzzles.
- BFS (Breadth-First Search) traverses level by level, ensuring all neighbours are visited before moving deeper, making it ideal for finding the shortest path in unweighted graphs.
- Both algorithms utilize different data structures: DFS uses a stack (implicitly through recursion), while BFS uses a queue.
- DFS is more memory-efficient on deep or large graphs, whereas BFS ensures the most optimal solution in problems like shortest path finding.

- Understanding these traversal methods is essential for problem-solving in areas such as network analysis, pathfinding, and AI.

CONCLUSION:

DFS and BFS are fundamental graph traversal algorithms, each suited to specific types of problems. DFS is effective for exploring deeper paths and is ideal when the entire search space needs to be visited, while BFS is optimal for finding the shortest path in unweighted graphs due to its level-wise exploration. Mastering both algorithms enhances problem-solving skills, especially in domains like AI, networking, and optimization, where traversing and analyzing complex structures is key.