2.2 UNIFORMED SEARCH

UNIFORMED SEARCH IN BFS:

•Breadth-First Search (BFS) is an uninformed search algorithm that explores a graph or tree level by level, starting from the root node.

•It systematically explores all the neighboring nodes at the present depth before moving on to the nodes at the next level.

•BFS is known for finding the shortest path in an unweighted graph and is often implemented using a FIFO queue.

How it Works:

1.Initialization: Start with a queue containing the root node.

2.Iteration: While the queue is not empty: o Dequeue a node from the front of the queue. o If the dequeued node is the goal, return the solution.

o Otherwise, enqueue all the unvisited neighbors of the dequeued node.

3. Termination: The algorithm terminates when the goal node is found or the queue becomes empty (meaning no solution exists). Example:

Imagine a maze represented as a graph. BFS would start at the entrance, explore all the paths leading one step away from the entrance, then all the paths two steps away, and so on, until it finds the exit.

Advantages:

•Guaranteed shortest path in unweighted graphs.

•Complete algorithm.

•Relatively simple to implement.

Disadvantages:

•Can be memory intensive: It needs to store all nodes at each level, which can be a problem for large search spaces.

•Can be slow if the solution is far from the starting point.

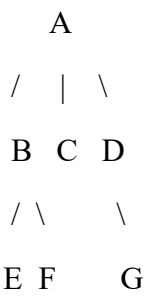
Applications:

•Finding the shortest path in unweighted graphs.

•Web crawling.

BFS – Example Diagram

Start at A, Goal is G



Step-by-step:

Queue: [A]

→ Expand A → Add B, C, D → Queue: [B, C, D]

→ Expand B → Add E, F → Queue: [C, D, E, F]

→ Expand C → No new nodes → Queue: [D, E, F]

→ Expand D → Add G → Queue: [E, F, G]

→ Expand E → No goal → Queue: [F, G]

→ Expand F → No goal → Queue: [G]

→ Expand G → Goal found

Python Program for bfs:

```
from collections import deque
# Graph represented as an adjacency list
graph = {
```

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

def bfs(start):
    visited = set()         # To keep track of visited nodes
    queue = deque([start])  # Initialize queue with the start node

    while queue:
        node = queue.popleft()  # Get the first node in the queue
        if node not in visited:
            print(node, end=' ')  # Visit the node
            visited.add(node)     # Mark it as visited
            queue.extend(graph[node])  # Add neighbors to the queue
# Run BFS starting from node 'A'
bfs('A')
```

Output:

A B C D E F

Key Characteristics:

•Uninformed Search:

BFS, like other uninformed search algorithms, doesn't use any domain-specific knowledge or heuristics to guide its search. It explores all possibilities without any prior information about the goal.

•Level-by-Level Exploration:

BFS explores the search space level by level. It visits all the nodes at a particular depth before moving on to the next level.

•FIFO Queue:

BFS uses a queue data structure (specifically, a First-In, First-Out queue) to manage the order in which nodes are explored. Nodes are added to the back of the queue and removed from the front.

•Completeness and Optimality:

BFS is complete, meaning that if a solution exists, BFS will find it. It is also optimal for finding the shortest path in unweighted graphs.

•Time and Space Complexity:

In the worst case, BFS has a time complexity of O(b^(d+1)), where 'b' is the branching factor (number of children per node) and 'd' is the depth of the solution.

The space complexity is also O(b^(d+1)) because it needs to store all the nodes at the current level.

UNIFORMED SEARCH IN DFS

•Depth-First Search (DFS) is an uninformed search algorithm that explores a graph or tree by going as deep as possible along each branch before backtracking.

•It uses a stack (either explicitly or implicitly through recursion) to keep track of the path and prioritize exploring deeper nodes.

•DFS is often used when the solution is likely to be found deep within the search space or when memory is a concern, as it typically uses less memory than Breadth-First Search.

How it Works:

1.Initialization: Start with an empty stack (or recursive call stack) and push the starting node onto the stack.

2.Iteration: Pop a node from the stack. o If the node is the goal, return the path

(solution).

Otherwise, push all unvisited neighbors of the node onto the stack.

3. Repeat: Continue the iteration until the stack is empty or the goal is found. Example:

Imagine a tree where you want to find a specific node (the goal). DFS would explore the left-most branch as deeply as possible before moving to the right branch. If it hits a dead end, it will backtrack to the previous node and explore another unexplored branch.

Applications:

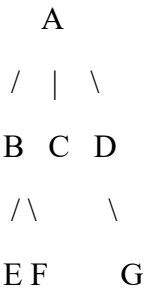
•Finding a path in a maze: DFS can be used to explore a maze, but it might not find the shortest path.

•Topological sorting: DFS can be used to determine the order of tasks in a project with dependencies.

•Cycle detection in graphs: DFS can help identify cycles in graphs.

DFS – Example Tree Diagram

Let's say we want to go from A to G:



Goal: G

🔷 DFS – Traversal Steps

🔶 Stack Process (LIFO):

1.Start at A → Stack: [A]

2.Expand A → Push D, C, B → Stack: [D, C, B]

3.Pop B → Push F, E → Stack: [D, C, F, E]

4.Pop E → Not goal

5.Pop F → Not goal

6.Pop C → Not goal

7.Pop D → Push G → Stack: [G]

8.Pop G → Goal Found

🔷 Python Program for DFS (Uniformed Search)

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

def dfs(node, visited=None):
    if visited is None:
        visited = set()  # Set to keep track of visited nodes

    if node not in visited:
        print(node, end=' ')  # Visit the node
        visited.add(node)     # Mark it as visited

        # Recursively visit all neighbors
        for neighbor in graph[node]:
            dfs(neighbor, visited)
```

```
# Run DFS starting from node 'A'
dfs('A')
```

Output:

A B D E F C

Key Characteristics of DFS:

•Uninformed:

It does not use any extra information (heuristics) to guide the search. It only relies on the structure of the graph or tree.

•Backtracking:

When a dead end is reached (no unvisited neighbors), the algorithm backtracks to the most recent branching point and explores another path.

•Stack Data Structure:

DFS uses a stack to store the nodes to be visited. The last node added to the stack is the first one to be explored (LIFO - Last In, First Out).

•Completeness:

In a finite state space, DFS is complete, meaning it will eventually find a solution if one exists.

However, in infinite state spaces, it may not be complete as it can get stuck in infinite loops.

•Time Complexity:

In the worst case, the time complexity is O(b^m), where 'b' is the branching factor (average number of children per node) and 'm' is the maximum depth of the search tree.

•Space Complexity:

The space complexity is typically O(bm), as it stores the nodes on the current path.

•Not Optimal:

DFS does not guarantee finding the shortest path to the goal.

INTERACTIVE DEEPENING DFS:

•Iterative Deepening Search (IDS) is an uninformed search algorithm that combines the benefits of Depth-First Search (DFS) and Breadth-First Search (BFS) while mitigating their drawbacks.

•It works by repeatedly performing depth-limited search with increasing depth limits until the goal is found.

How it Works:

1.Initialize: Set the depth limit to 0.

2.Depth-Limited Search: Perform a depth-first search with the current depth limit.

3.Check for Goal: If the goal node is found, terminate the search.

4.Increment Depth: If the goal is not found, increment the depth limit and repeat steps

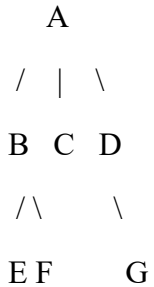
2-3.

Example:

Consider a search tree where the goal node is at depth 3. IDS would first perform a depth-first search up to depth 1, then up to depth 2, and finally up to depth 3, at which point the goal node would be found.

IDDFS Tree Diagram Example:

Let's use this example tree to find node G:



Goal: G

Depth Levels:

•Depth 0 → Only A

•Depth 1 → A → B, C, D

•Depth 2 → B → E, F and D → G

IDDFS Traversal Step-by-Step:

It performs DFS with depth limits:

1. Depth = 0: A → Not goal

2. Depth = 1: A → B, C, D → Not goal 3. Depth = 2: o A → B → E o B → F

o D → G Goal

Found

Python Program for IDDFS

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

# Depth-Limited DFS
def dls(node, goal, depth, visited):
   if depth == 0 and node == goal:
      return True
   if depth > 0:
```

```
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                if dls(neighbor, goal, depth - 1, visited):
                    return True
        visited.remove(node)
    return False

# Iterative Deepening DFS
def iddfs(start, goal, max_depth):
    for depth in range(max_depth + 1):
        visited = set()
        if dls(start, goal, depth, visited):
            print(f"Goal {goal} found at depth {depth}")
            return
    print(f"Goal {goal} not found up to depth {max_depth}")

# Example: search for 'F' starting from 'A'
iddfs('A', 'F', 5)
```

Output:

Goal F found at depth 2

Key Features:

•Uninformed:

Like other uninformed search algorithms, IDS doesn't require any prior knowledge or heuristic information about the search space or the goal.

•Iterative Approach:

IDS performs a series of depth-limited searches, starting with a depth limit of 1 and incrementing it in each iteration.

•Combines DFS and BFS:

It leverages the memory efficiency of DFS by exploring one path at a time, and the completeness of BFS by guaranteeing the shortest path is found if it exists.

•Completeness and Optimality:

If a solution exists, and the cost function is non-decreasing with depth, IDS is complete (it will find a solution if one exists) and optimal (it will find the shortest path).

•Time and Space Complexity:

The time complexity is O(b^d), where b is the branching factor and d is the depth of the solution. The space complexity is O(bd), making it more space-efficient than BFS.