

Search Methods

Unit No 2

Dr. Praveen Barapatre

Breadth First search (BFS)

- **Breadth-First Search (BFS)** is a graph traversal algorithm that explores all nodes at a given depth before moving to the next level. It's often used to find the shortest path between two nodes in a graph where the edges have uniform weights (like a maze or a network).

Algorithm Steps:

Initialization:

1. Mark all nodes as unvisited.
2. Choose a starting node and mark it as visited.
3. Create a queue to store nodes for exploration.
4. Enqueue the starting node.

Algorithm Steps:

Traversal:

1. While the queue is not empty:
 1. Dequeue a node from the front of the queue.
 2. Process the node (e.g., print its value or check if it's the target).
 3. For each unvisited neighbor of the dequeued node:
 1. Mark the neighbor as visited.
 2. Enqueue the neighbor.

Pseudocode:

```
function BFS(graph, start):  
    queue = []  
    visited = set()  
    queue.append(start)  
    visited.add(start)  
    while queue:  
        node = queue.pop(0) # Dequeue the front node  
        # Process the node (e.g., print its value or check if it's the target)  
        for neighbor in graph[node]:  
            if neighbor not in visited:  
                visited.add(neighbor)  
                queue.append(neighbor)
```

Applications

- **Shortest path in unweighted graphs:** BFS finds the path with the fewest edges between two nodes.
- **Connected components:** BFS can identify isolated groups within a graph.
- **Level-order traversal of trees:** BFS can traverse a tree level by level.
- **Page ranking algorithms:** BFS is used in some page ranking algorithms to explore web pages.

Video

<https://www.youtube.com/watch?v=oDqjPvD54Ss>

Comparison Between BFS and DFS

| Feature | BFS | DFS |
|-----------------------------------|---|--|
| Traversal Order | Level-by-level (breadth-wise) | Depth-wise (branch-by-branch) |
| Data Structure | Queue | Stack |
| Time Complexity | $O(V)$ | $O(V)$ |
| Space Complexity | $O(V)$ | $O(V)$ |
| Shortest Path (Unweighted Graphs) | Finds the shortest path (in terms of edges) | May not find the shortest path |
| Applications | Shortest paths in unweighted graphs, connected components, level-order traversal of trees | Topological sorting, cycle detection, maze solving |

Key Differences

- **Traversal Order:**

- BFS explores nodes level by level, moving from left to right in each level.
- DFS explores nodes depth-wise, going as deep as possible along a path before backtracking.

- **Data Structure:**

- BFS uses a queue to store nodes for exploration.
- DFS uses a stack (or recursion) to store nodes for exploration.

- **Shortest Path:**

- BFS is guaranteed to find the shortest path in unweighted graphs.
- DFS may not find the shortest path, as it can explore a deeper path that may not be optimal.

When to Use Which

- **BFS:**

- When you need to find the shortest path in an unweighted graph.
- For level-order traversal of trees.
- To find connected components in a graph.

- **DFS:**

- When you need to explore all possible paths in a graph.
- For topological sorting or cycle detection.
- For maze solving or game-tree search.

Quality of Solution

Correctness:

- Does the solution accurately address the problem?
- Are there any errors or bugs in the implementation?
- Does it produce the expected output for various inputs?

Efficiency:

- How fast does the solution run?
- Does it use efficient algorithms and data structures?
- Can the performance be improved?

Readability:

- Is the code well-formatted and easy to understand?
- Are variable and function names meaningful?
- Are comments used to explain complex logic?

Maintainability:

- Is the code modular and easy to modify?
- Can changes be made without introducing errors?
- Is the solution scalable to handle future requirements?

Testability:

- Is the solution well-tested with a variety of inputs?
- Are there unit tests and integration tests in place?

Elegance:

- Is the solution concise and elegant?
- Does it avoid unnecessary complexity?
- Is it a pleasure to read and understand?

Depth Bounded DFS

- Depth-Bounded DFS (DBFS) is a variant of Depth-First Search (DFS) that limits the depth of the search to a specified bound. This can be useful in situations where the search space is very large and exploring all possible paths would be computationally expensive or impractical.
- Set a depth limit, which determines the maximum depth that the search can reach.
- If the search reaches the depth limit, it backtracks to the previous node and continues the search from there.

Pseudocode:

```
function DBFS(graph, start, depth_limit):  
    visited = set()  
    stack = [(start, 0)] # (node, depth)  
    while stack:  
        node, depth = stack.pop()  
        if depth > depth_limit:  
            continue # Ignore nodes beyond the depth limit  
        if node not in visited:  
            visited.add(node)  
            # Process the node (e.g., print its value or check if it's the target)  
            for neighbor in graph[node]:  
                stack.append((neighbor, depth + 1))
```

Applications

- **Game Tree Search:** In games like chess or Go, DBFS can be used to explore the game tree to a certain depth to evaluate possible moves.
- **Constraint Satisfaction Problems:** DBFS can be used to solve problems with constraints, such as Sudoku or the N-Queens problem, by limiting the search to a certain depth.
- **Pathfinding:** DBFS can be used to find paths in a graph with a limited number of steps.

Advantages:

- Can be more efficient than DFS for large search spaces.
- Can be used to find solutions within a specified time or resource constraint.

Disadvantages:

- May not find the optimal solution if the depth limit is too low.
- Can be sensitive to the choice of depth limit.

Depth-First Iterative Deepening (DFID)

- **Depth-First Iterative Deepening (DFID)** is a search algorithm that combines the benefits of depth-first search (DFS) and breadth-first search (BFS). It's particularly useful for problems where the solution depth is unknown or may vary significantly.

How DFID Works

- 1. Start with a depth limit of 0.**
- 2. Perform a depth-first search (DFS) up to the current depth limit.**
- 3. If a solution is found, return it.**
- 4. If no solution is found, increase the depth limit by 1 and repeat steps 2-3.**

- Efficiency: It's more efficient than BFS in terms of space complexity, as it only needs to store the current path.
- Completeness: It's complete, meaning it will always find a solution if one exists.
- Time Complexity: Its time complexity is similar to BFS in the worst case, but it can be more efficient in practice, especially if the solution depth is relatively small.
- Space Complexity: Its space complexity is $O(d)$, where d is the depth of the solution.

Advantages of DFID

- **Efficient Space Usage:** Compared to BFS, DFID requires less memory.
- **Completeness:** It guarantees to find a solution if one exists.
- **Can Handle Varying Solution Depths:** It's well-suited for problems where the solution depth is unknown or can vary.

Disadvantages of DFID

- **Re-Explores Nodes:** It can re-explore nodes multiple times, which can be inefficient in some cases.

Applications of DFID

- **Game-Playing:** DFID is often used in game-playing algorithms, such as those for chess and Go, to explore the search space efficiently.
- **Planning:** It can be applied to planning problems where the goal state is unknown or the path to the goal may vary in length.
- **Constraint Satisfaction Problems:** DFID can be used to solve constraint satisfaction problems, such as Sudoku and Minesweeper.

Depth Bounded DFS Video

<https://www.youtube.com/watch?v=P7WQUBLKDmo>

Depth First Iterative Deepening Video

<https://www.youtube.com/watch?v=BK8cEWKHCKY>