5. Graph Traversal: Depth-First Search (DFS)

Depth-First Search (DFS) is a traversal algorithm that explores the graph by going as "**deep**" as possible down one path before backtracking.

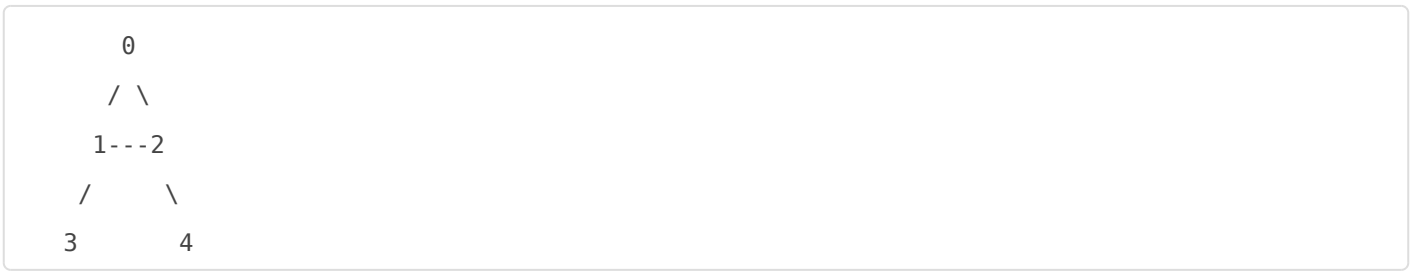
5.1 DFS Analogy and Illustration

Imagine exploring a **maze** with only one path.

- You pick a direction (an **edge**) and follow it.
- You keep going, taking the **first available turn** at each junction (visiting the first unvisited neighbor).
- You continue until you hit a **dead end** (a node with no unvisited neighbors).
- At the dead end, you **backtrack** to the previous junction and **try a different path**.

DFS works this way. It is excellent for detecting cycles, topological sorting, and solving puzzles.

Example: We want to trace BFS starting from node `0` on the following graph:



The traversal order will be `0, 1, 2, 4, 3`.

5.2 The Role of `std::stack`

The LIFO (Last-In, First-Out) nature of a stack is perfect for DFS.

- **Recursive DFS:** The system's call stack is used implicitly. When you make a **recursive call** `DFSUtil(neighbor)`, you **"push" the new function** onto the stack. When you hit a dead end, the function returns, **"popping" itself off** the stack and automatically "backtracking" to the previous node.

- **Iterative DFS:** We manually use a `std::stack`. We **push a node**, then **push its neighbors**. The last neighbor pushed is on **top**, so it will be the first one we explore next, perfectly mimicking the "go deep" behavior of recursion.

5.3 DFS Implementation

5.3.1 Iterative Approach

This approach uses an **explicit** `std::stack` and avoids recursion.

```
void DFS(int s) {
    // 1. Create a visited array, initialized to all false
    std::vector<bool> visited(V, false);

    // 2. Create a Stack for DFS
    std::stack<int> stack;

    // 3. Push the source vertex onto the stack
    stack.push(s);

    std::cout << "DFS Traversal (Iterative): ";

    // 4. Loop as long as the stack is not empty
    while (!stack.empty()) {
        // 5. Pop a vertex from the top of the stack
        int u = stack.top();
        stack.pop();

        // 6. If it hasn't been visited yet:
        if (!visited[u]) {
            visited[u] = true;
            std::cout << u << " ";
        }

        // 7. Get all neighbors of 'u'
        for (auto& neighbor : adj[u]) {
            // 8. If the neighbor is unvisited, push it
            if (!visited[neighbor]) {
                stack.push(neighbor);
```

```
            }
        }
    }
    std::cout << std::endl;
}
```

Code Explanation:

- Initialize a `visited` array (all `false`) and an empty `stack`.
- Push the starting node `s` onto the stack.
- While the stack is not empty:
  - Pop a node `u` from the stack.
  - If `u` has not been visited:
    - Mark `u` as visited and print it.
    - Iterate through all neighbors of `u`.
    - If a neighbor has not been visited, push it onto the stack (to be processed next).

5.3.2 Recursive Approach

This is the most common and intuitive way to implement DFS. It uses a **helper function**.

```
// Private helper function for recursive DFS
void DFSUtil(int u, std::vector<bool>& visited) {
    // 1. Mark the current node as visited and print it
    visited[u] = true;
    std::cout << u << " ";

    // 2. Iterate over all neighbors
    for (auto& neighbor : adj[u]) {
        // 3. If a neighbor is unvisited:
        if (!visited[neighbor]) {
            // 4. Make a recursive call to "go deeper"
            DFSUtil(neighbor, visited);
        }
    }
    // 5. When all neighbors are visited, the function returns.
    //    This is the implicit "backtracking" step.
}

// Public wrapper function to start the DFS
```

```
void DFSRecursive(int s) {
    // 1. Create a visited array
    std::vector<bool> visited(V, false);
    std::cout << "DFS Traversal (Recursive): ";

    // 2. Call the helper function to start the recursion
    DFSUtil(s, visited);
    std::cout << std::endl;
}
```

Code Explanation:

- The `DFSRecursive` function initializes the `visited` array.
- It calls the helper function `DFSUtil` with the starting node `s`.
- Inside `DFSUtil` (the recursive part):
  - Marks the current node `u` as visited and prints it.
  - Iterates through all neighbors of `u`.
  - If a neighbor has not been visited, it immediately calls `DFSUtil` on that neighbor, "**going deeper**."
  - When a node has no unvisited neighbors, its function call finishes and "**backtracks**" to the previous node's loop.

5.3.3 Comparison: Iterative vs. Recursive

Both recursive and iterative DFS **accomplish the same goal**, but they differ in implementation and performance characteristics.

Aspect	Recursive DFS	Iterative DFS
Logic	Intuitive and clean. Backtracking is handled **implicitly** by function returns.	Backtracking is handled **explicitly** using a `std::stack`.
Data Structure	Uses the **Call Stack** (managed by the system).	Uses an explicit `std::stack` (managed by the programmer on the heap).
Memory Usage	Can cause a **Stack Overflow** error if the graph is very deep (long paths).	Safer for very deep graphs, as heap memory is much larger than stack memory.
Code Simplicity	Often shorter and easier to write and understand the core "go deep" logic.	Can be more complex to write, but the process is more transparent.

In summary:

- Use **Recursive DFS** for simplicity and clarity, especially when the graph depth is **known to be manageable**.

- Use **Iterative DFS** when you need to avoid stack overflow on **very deep graphs** or when you need **finer control over the traversal**.

Revision #3
Created 2025-11-11 11:28:24 UTC by RE
Updated 2025-11-11 11:33:59 UTC by RE