**Name : Abdul Rehman Raza**

**Section : BSDS\_3A**

**Roll No : 052**

**Task 05 :**

**Documentation:(DFS) Using a Stack**

**Overview**

This document explains an iterative implementation of Depth-First Search (DFS) that visits nodes in a graph using an explicit stack. DFS is a graph traversal technique that explores as far along each branch as possible before backtracking. The iterative approach replaces the recursive call stack with an explicit data structure (stack), which makes the method suitable for environments where recursion depth is a concern.

**Purpose**

The purpose of this implementation is to traverse all reachable nodes of a directed or undirected graph starting from a specified start node, visiting each node at most once, and printing the nodes in the order they are first discovered.

**Inputs and Data Structures**

* **Graph**: A mapping from each vertex to a list of its adjacent vertices. This representation assumes that adjacency lists explicitly list neighbors for each vertex.
* **Start vertex**: The node from which DFS traversal begins.
* **Visited set**: A set that stores nodes already visited to avoid repeated processing.
* **Stack**: A list or stack container that stores nodes to be explored. The stack uses last-in, first-out order.

**Algorithm Description (High Level)**

1. Initialize an empty set called visited to record nodes that have been processed.
2. Initialize a stack and push the start vertex onto it.
3. While the stack is not empty:
   * Pop a vertex from the top of the stack.
   * If the vertex has not been marked visited:
     + Process the vertex (for example, print it or record it in an output list).
     + Mark the vertex as visited.
     + Add the vertex’s neighbors to the stack.
       - To obtain the same visitation order as the typical recursive DFS, push the neighbors in reverse order so that the left-most neighbor is processed first when popped next.
4. Repeat until the stack is empty.

This algorithm ensures that each vertex reachable from the start node is processed exactly once.

**Step-by-step Example**

Consider a graph with nodes A through F and adjacency as follows:

* A → B, C
* B → D, E
* C → F
* D → (none)
* E → F
* F → (none)

Starting from A, the traversal proceeds as follows:

1. Push A. Stack: [A]
2. Pop A, visit A, push neighbors in reverse: push C, then B. Stack: [C, B]
3. Pop B, visit B, push its neighbors in reverse: push E, then D. Stack: [C, E, D]
4. Pop D, visit D. D has no neighbors. Stack: [C, E]
5. Pop E, visit E, push neighbors in reverse: push F. Stack: [C, F]
6. Pop F, visit F. F has no neighbors. Stack: [C]
7. Pop C, visit C, push neighbors in reverse: push F, but F is already visited, so it is not added. Stack: []
8. Traversal ends.

The order of visitation produced by this approach is: A B D E F C.

**Complexity Analysis**

* **Time complexity:** O(V + E), where V is the number of vertices and E is the number of edges. Each vertex is pushed and popped at most once, and each edge is explored at most once when iterating adjacency lists.
* **Space complexity:** O(V) for the visited set and O(V) for the stack in the worst case. Additional space is required for storing the adjacency lists, which is O(V + E).

**Advantages of the Iterative Approach**

* Avoids recursion depth limits that may be encountered in languages or environments with restricted call stack size.
* Makes explicit manipulation of the traversal order easier (for example, by reversing neighbor order to match recursive behavior).
* Facilitates implementation of iterative variations or additional logic that would be more complex inside recursion.

**Practical Applications**

* Pathfinding and reachability queries in graphs.
* Topological ordering primitives (with modifications).
* Cycle detection and connectivity analysis when combined with additional logic.
* Solving puzzles and problems that require deep exploration such as maze traversal, game state trees, and backtracking algorithms.

**Edge Cases and Considerations**

* **Disconnected graph**: The algorithm as described visits only nodes reachable from the given start vertex. To traverse an entire graph that is possibly disconnected, iterate over all vertices and initiate DFS from each unvisited vertex.
* **Graph representation**: The adjacency list must include an entry for every vertex, even if that vertex has no neighbors.
* **Duplicate edges and self-loops**: The visited set protects against infinite loops from self-loops or multiple edges, but care must be taken when interpreting results.
* **Order sensitivity**: The traversal order depends on the ordering of neighbors and whether they are pushed in reverse. If a deterministic order is required, ensure adjacency lists are consistently ordered.

**Suggestions for Extension**

* Modify the routine to return a list of visited nodes rather than printing them, to facilitate unit testing and downstream processing.
* Extend the agent to detect cycles by tracking recursion ancestry or parent pointers.
* Use an explicit timestamp or discovery/finish times to enable algorithms that require entry/exit times, such as strongly connected components or bridge-finding.
* For weighted graphs, combine this traversal with other techniques (for example, use DFS as part of constructing spanning structures).

**Conclusion**

An iterative DFS using an explicit stack is a robust and practical alternative to recursive DFS. It preserves the depth-first exploration strategy while avoiding recursion limits and provides flexibility in controlling the traversal order.