Design and Analysis of Algorithm

Assignment -02

Submitted by: **Submitted to :**

Laiba Shakoor BSEF21M021 Sir Nadeem Akhtar

Breadth First Traversal for Graphs

BFS is an algorithm for traversing or searching tree or graph data structures. It starts at the root (or an arbitrary node) and explores all the neighbor nodes at the present depth prior to moving on to nodes at the next depth level. BFS uses a queue to keep track of the nodes to be explored.

Data Structure Used:

- BFS uses a **queue** to manage the nodes to be explored next. A queue operates in a First-In-First-Out (FIFO) manner, meaning the first node added to the queue is the first one to be processed.

Examples

1. **Shortest Path in an Unweighted Graph**:
   Finding the shortest route between two subway stations in a transit system where all connections are equally spaced.
2. **Level-Order Traversal in a Binary Tree:**
   Printing all employees in a company level by level, starting with the CEO (root) and moving down through the management hierarchy.
3. **Finding Connected Components in a Graph:**
   Analyzing social networks to identify groups of friends who are all directly or indirectly connected.

Pseudocode:

```
BFS(Graph, startNode):
    create a queue Q
    mark startNode as visited and enqueue startNode into Q
    while Q is not empty:
        node = Q.dequeue()
        process(node)
        for each neighbor of node:
            if neighbor is not visited:
                mark neighbor as visited
                enqueue neighbor into Q
```

C++ Implementation

```
#include <iostream>
#include <queue>
#include <vector>

using namespace std;

//Function for BFS
void BFS(int startNode, vector<vector<int>>& adjList, vector<bool>&
visited) {
   queue<int> q;
   q.push(startNode);
   visited[startNode] = true;

   while (!q.empty()) {
      int node = q.front();
      q.pop();
      cout << node << " "; // Process node

      for (int neighbor : adjList[node]) {
         if (!visited[neighbor]) {
            visited[neighbor] = true;
            q.push(neighbor);
         }
      }
   }
}

//Main
int main() {
   int n = 5; // Number of nodes
   vector<vector<int>> adjList(n);
   adjList[0] = {1, 2};
   adjList[1] = {0, 3, 4};
   adjList[2] = {0};
   adjList[3] = {1};
   adjList[4] = {1};

   vector<bool> visited(n, false);
   BFS(0, adjList, visited);

   return 0;
}
```

Depth First Traversal for Graphs

DFS is an algorithm for traversing or searching tree or graph data structures. It starts at the root (or an arbitrary node) and explores as far as possible along each branch before backtracking. DFS uses a stack (which can be implemented using recursion) to keep track of the path.

Data Structure Used:

- DFS uses a **stack** to manage the nodes to be explored next. In the recursive implementation, this stack is implicit and handled by the system's call stack.

Examples

1. **Path Finding in a Maze:**

   Navigating a puzzle game where you must explore all potential paths to reach the goal.

2. **Topological Sorting in a Directed Acyclic Graph (DAG):**

   Determining the order in which to complete tasks given dependencies between them, such as compiling files where some files depend on others.

3. **Cycle Detection in a Graph:**

   Checking if there is a circular dependency between tasks in a project management tool.

Pseudocode:

```
DFS(Graph, node):
    mark node as visited
    process(node)
    for each neighbor of node:
        if neighbor is not visited:
            DFS(Graph, neighbor)
```

C++ Implementation

```
#include <iostream>
#include <vector>

using namespace std;

void DFS(int node, vector<vector<int>>& adjList, vector<bool>& visited) {
    visited[node] = true;
    cout << node << " "; // Process node

    for (int neighbor : adjList[node]) {
        if (!visited[neighbor]) {
            DFS(neighbor, adjList, visited);
        }
    }
}

int main() {
       int n = 5; // Number of nodes
      vector<vector<int>> adjList(n);
      adjList[0] = {1, 2};
      adjList[1] = {0, 3, 4};
      adjList[2] = {0};
      adjList[3] = {1};
      adjList[4] = {1};
      vector<bool> visited(n, false);
      DFS(0, adjList, visited);
     return 0;
}
```