Problem Statement

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking. In this project we implemented some of the major applications of DFS which are

1) Finding **Strongly Connected Components (SCC)** of a graph.A directed graph is strongly connected if there is a path between all pairs of vertices. A SCC of a directed graph is a maximal strongly connected subgraph. Our aim is to find all the **Strongly Connected Components (SCC)** in a given directed graph.

2) **Detecting Cycle in a graph**. A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.

3) **Path Finding**. We can use DFS to find a path between two given vertices.

4) **Solving a Maze**. DFS can be used to find the solution to a maze by storing the nodes on the current path.

Strongly Connected Components (SCC)

0.1 Approach

The algorithm used here is **Kosaraju's algorithm** which makes two DFS calls on the graph. DFS of a graph G, either produces a single tree if all vertices are reachable from the DFS starting point or a forest if not. This means DFS of a graph with only one SCC always produces a tree. Whether DFS produces a tree or a forest entirely depends on the chosen starting point. To find the SCC's, we have to process vertices from sink to the source. But how do we find this sequence of picking vertices as starting point of the DFS. However, if we do a DFS of graph and store vertices according to their finish times, we make sure that the finish time of a vertex that connects to other SCCs (other that its own SCC), will always be greater than finish time of vertices in the other SCC. So to use this property, we do DFS traversal of complete graph and push every finished vertex to a stack.

In the next step we reverse the graph G i.e. the edges in the graph gets reversed. It means that the source vertices become the sink vertices and vice versa. Now when we do a second DFS on G^T (transpose of the graph), starting with one of the SCC (with the vertices in the stack), because of the above mentioned property, there will be no edges from one SCC to another SCC in G^T. Therefore, DFS will visit only vertices in the present SCC. This process continues for the vertives in the other SCC thereby giving us all the SCC's.

0.2 Algorithm

1) Create an empty stack S and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack.
2) Reverse directions of all arcs to obtain the transpose graph.
3) One by one pop a vertex from S while S is not empty. Let the popped vertex be v. Take v as source and do DFS (call DFSUtil(v)). The DFS starting from v prints strongly connected component of v.

0.3 Complexity

1) If we use **adjacency list** to describe the graph, then the algorithm takes $\Theta(V + E)$ (linear) time since the algorithm must go through all vertices and edges.

2) If the graph is represented by adjacency matrix, then the algorithm takes $\mathcal{O}(V^2)$ time.

0.4 Code

The algorithm is implemented in C++. For the adjacency list implementation, run the file **SCC_adjlist.cpp** using the command **g++ SCC_adjlist.cpp -o test** to compile and **./test** to run. For the adjacency matrix implementation, run the file **SCC_adjmatrix.cpp** using the command **g++ SCC_adjmatrix.cpp -o test** to compile and **./test** to run.

Detecting a Cycle in a Graph

0.5 Approach

The main requirement for a graph to have a cycle is to have a back edge. For that first we do a DFS on the given graph. Now for a particular visited vertex "v", if there exists an adjacent vertex "w" which is already visited and not it's parent then it's a back edge and therefore there exists a cycle. If such vertex doesn't exist then there is no cycle.

0.6 Algorithm

1) Start with any vertex u.
2) If the vertex v = adj[u] is visited and parent[v] != u, then there is a cycle.
3) Else mark the vertex v visited and assign parent[v] = u.
4) Go to Step 2.

0.7 Complexity

If we use **adjacency list** to describe the graph, then the algorithm takes $\mathcal{O}(V + E)$ (linear) time since the algorithm must go through all vertices and edges.

0.8 Code

The algorithm is implemented in C++. Run the file **find_cycle.cpp** using the command **g++ find_cycle.cpp -o test** to compile and **./test** to run.

Path Finding between any two given vertices

0.9 Approach

DFS can also be used to check whether there exists a path between any two vertices u and v. For this, start with any of the vertex as the source vertex and do a DFS. If we visit the other vertex in the DFS traversal then it means there exists a path between the given vertices else it doesn't.

0.10 Algorithm

Let the given vertices be u and v.
1) Call DFS(u) with u as the source (start) vertex.
2) Keep track of all the vertices between u and all the encountered vertices.
3) If the vertex v is visited during the DFS traversal then there is a path else no path.

0.11 Complexity

If we use **adjacency list** to describe the graph, then the algorithm takes $\mathcal{O}(V + E)$ (linear) time since the algorithm must go through all vertices and edges.

0.12 Code

The algorithm is implemented in C++. Run the file **find_path.cpp** using the command **g++ find_path.cpp -o test** to compile and **./test** to run.

Solving a MAZE

0.13 Approach

The path finding application can be easily used to solve a maze where we include all the vertices in the path between source and destination. Here we read the maze from the *input_graph.txt* file where the vertices of the graph are represented by 0's and 1's. If there exists two 1's adjacent to each other then there is a path between them. The number of vertices is given in the first line. The entry and the exit vertices are given in the last line.Now that we have the maze as a graph then we simply apply our path finding algorithm on this graph giving us the path between the entry vertex and the exit vertex.

0.14 Code

The algorithm is implemented in C++. Run the file **maze_solve.cpp** using the command **g++ maze_solve.cpp -o test** to compile and **./test** to run.

Testcases

Please do refer Testcases section below for some of the testcases and their results for the above mentioned implementations.

Summary

From the above implementations it is quite evident how DFS is useful in many practical scenarios. Apart from the above mentioned applications, DFS has a lot more applications in various fields.