Depth First Search

Depth First Search finds the lexicographical first path in the graph from a source vertex u to each vertex. Depth First Search will also find the shortest paths in a tree (because there only exists one simple path), but on general graphs this is not the case.

Description

It is very easy to describe / implement the algorithm recursively: We start the search at one vertex. After visiting a vertex, we further perform a DFS for each adjacent vertex that we haven't visited before. This way we visit all vertices that are reachable from the starting vertex.

Basic Implementation

This is the most simple implementation of Depth First Search.

```
vector<vector<int>> adj; // graph represented as an adjacency list
int n; // number of vertices

vector<bool> visited;

void dfs(int v) {
    visited[v] = true;
    for (int u : adj[v]) {
        if (!visited[u])
            dfs(u);
    }
}
```

Vertex Coloring Implementation

Sometimes, it might be useful to also compute the entry and exit times and vertex color. We will color all vertices with the color 0, if we haven't visited them, with the color 1 if we visited them, and with the color 2, if we already exited the vertex.(See Cormen's dfs algorithm)

Here is a generic implementation that additionally computes those:

```
vector<vector<int>> adj; // graph represented as an adjacency list
int n; // number of vertices

vector<int> color;

vector<int> time_in, time_out;
int dfs_timer = 0;

void dfs(int v) {
    time_in[v] = dfs_timer++;
    color[v] = 1;
    for (int u : adj[v])
        if (color[u] == 0)
            dfs(u);
    color[v] = 2;
    time_out[v] = dfs_timer++;
}
```

Classification of edges of a graph

We can classify the edges using the entry and exit time of the end nodes u and v of the edges. (u, v). These classifications are often used for problems like finding bridges and finding articulation points.
We perform a DFS and classify the encountered edges using the following rules:
If v is not visited:
Tree Edge - If v is visited after u then edge (u, v) is called a tree edge. In other words, if v is visited for the first time and u is currently being visited then (u, v) is called tree edge. These edges form a DFS tree and hence the name tree edges.
Note: Forward edges and cross edges only exist in directed graphs.

If v is visited before u: Back edges - If v is an ancestor of u, then the edge (u, v) is a back edge. v is an ancestor exactly if we already entered v, but not exited it yet. Back edges complete a cycle as there is a path from ancestor v to descendant u (in the recursion of DFS) and an edge from descendant u to ancestor v (back edge), thus a cycle is formed. Cycles can be detected using back edges.

Forward Edges - If v is a descendant of u, then edge (u, v) is a forward edge. In other words, if we already visited and exited v and $\text{entry}[u] < \text{entry}[v]$ then the edge (u, v) forms a forward edge.

Cross Edges: if v is neither an ancestor or descendant of u, then edge (u, v) is a cross edge. In other words, if we already visited and exited v and $\text{entry}[u] > \text{entry}[v]$ then (u, v) is a cross edge.

Applications of Depth First Search

1. Find any path in the graph from source vertex u to all vertices.
2. Find lexicographical first path in the graph from source u to all vertices.
3. Check if a vertex in a tree is an ancestor of some other vertex:
At the beginning and end of each search call we remember the entry and exit "time" of each vertex. Now you can find the answer for any pair of vertices (i, j) in $O(1)$: vertex i is an ancestor of vertex j if and only if $\text{entry}[i] < \text{entry}[j]$ and $\text{exit}[i] > \text{exit}[j]$.
4. Find the lowest common ancestor (LCA) of two vertices.
5. Topological sorting:
Run a series of depth first searches so as to visit each vertex exactly once in $O(n + m)$ time. The required topological ordering will be the vertices sorted in descending order of exit time.

6. Check whether a given graph is acyclic and find cycles in a graph. (As mentioned above by counting back edges in every connected components).
7. Find strongly connected components in a directed graph:
First do a topological sorting of the graph. Then transpose the graph and run another series of depth first searches in the order defined by the topological sort. For each DFS call the component created by it is a strongly connected component.
8. Find bridges in an undirected graph:
First convert the given graph into a directed graph by running a series of depth first searches and making each edge directed as we go through it, in the direction we went. Second, find the strongly connected components in this directed graph. Bridges are the edges whose ends belong to different strongly connected components.