Applications of DFS

1 The White-Gray-Black DFS

DFS is useful for simple flood-filll since it's easy to code. However, there are more complicated algorithm that uses DFS. In particular, these algorithms make use of the DFS tree. To explain what a DFS tree is, we first expand our DFS algorithm to what is known as the **White-Gray-Black DFS**. First, we mark all vertices white. When we call `dfs(u)`, we mark u to be gray. Finally, when `dfs(u)` returns, we mark u to be black. Also, we have a global counter to keep track of when each vertex is first visited.

```
bool graph[128][128];      // adjacency matrix
int dfsnum[128];           // when each vertex is first visited
int num = 0;               // global counter for dfsnum[]
int colour[128];           // 0 = white, 1 = graph, 2 = black

void dfs(int u) {
  colour[u] = 1;             // mark vertex gray
  dfsnum[u] = num++;
  for (int v = 0; v < 128; ++v) {
    if (graph[u][v] && colour[v] == 0) {
      dfs(v, u);
    }
  }
  colour[u] = 2;         // mark vertex black
}
```

In the above code, note that we only recurse if v is a white vertex. If we consider all the edges from gray to white vertices, we get what is called **DFS Tree**. The DFS tree shows how the vertices of the graph were visited in the DFS. Hence, edges that connects from gray to white vertices are called as **Tree Edges**. Edges from gray vertices to gray vertices are known as **Back Edges** since they connect one vertex to its ancestor in the DFS tree. Edges that go from gray to black vertices are either **Cross Edges** or **Forward Edges**. A cross edge is an edge that connect one DFS branch to another; a forward edge is an edge from one vertex to its descendant. In most cases, cross edges and forward edges are not really distinguished and may be classified as cross edges in general. In an undirected graph, there will be no cross or forward edges.

2 Application 1: Strongly Connected Component

In a directed graph, two vertices a and b are **strongly connected** iff there exists a path from a to b AND there exists a path from b to a. The notion of strongly connectedness is an equivalence relation:

- a vertex is strongly connected with itself
- if a is strongly connected to b, then b is strongly connected to a
- if a is strongly connected to b and b is strongly connected to c, then a is strongly connected to c

This means that we can partition an arbitrary directed graph into its strongly connected component. There are several algorithms to find the strongly connected component of a graph using DFS. We will describe **Tarjan's Algorithm** here.

2.1 Tarjan's Algorithm

The crucial observation is to note that a strongly connected component must be a subtree of the DFS tree. In other word, a component cannot be in two different parts of the tree. This means that to find the strongly connected component, we just need to find the root of the subtrees and take everything under the root to be in the same component. Furthermore, within the same strongly connected component, the root will be the vertex that has the smallest `dfsnum`. Thus, a vertex u is a root of a strongly connected component iff the smallest `dfsnum` reachable from u or any of its children is `dfsnum[u]`. When we are running the DFS, we can keep a stack of the vertices which we have visited. Once we discovered the root of a strongly connected component, we can simply pop off the vertices of the stack until the root is popped off; all these vertices are in the same strongly connected component.

```
bool graph[128][128];     // adjacency matrix
int scc[128];             // the strongly connected component of each vertex
vector<int> st;           // the stack
int num_scc = 0;          // number of strongly connected component
int dfsnum[128];          // when each vertex is first visited
int num = 0;              // global counter for dfsnum
int low[128];             // smallest dfsnum reachable from the subtree

// initialization: dfsnum and scc are initalized to -1
void SCC(int u) {
  low[u] = dfsnum[u] = num++;
  st.push_back(u);

  for (int v = 0; v < 128; ++v) {
    if (graph[u][v] && scc[v] == -1) {
      if (dfsnum[v] == -1) SCC(v);
      low[u] = min(low[u], low[v]);
    }
  }

  if (low[u] == dfsnum[u]) {
    // root of a strongly connected component
    while (scc[u] != num_scc) {
      scc[st.back()] = num_scc;
      st.pop_back();
    }
    ++num_scc;
  }
}
```

3 Application 2: Bridge Detection

In an undirected graph, a **bridge** is defined to be an edge which if removed, will separate the graph into two disconnected component. This problem can also be solved by doing a single dfs. The main observation is that an edge (u, v) cannot be a bridge if it is part of a cycle. Conversely, if (u, v) is not part of a cycle, it is a bridge. We can detect cycles in dfs by the presence of *back edges*. Thus, (u, v) is a bridge iff none of v or v's children has a back edge to u or any of u's ancestor. To detect whether any of v's children has a back edge to u's parent, we can use a similar idea above to see what is the smallest `dfsnum` reachable from the subtree rooted at v.

```
bool graph[128][128];     // adjacency matrix
int dfsnum[128];          // when each vertex is first visited
int num = 0;              // global counter for dfsnum
int low[128];             // smallest dfsnum reachable from the subtree

// initialization: dfsnum initialized to -1
void bridge_detection(int u) {
  low[u] = dfsnum[u] = num++;

  for (int v = 0; v < 128; ++v) {
    if (graph[u][v] && dfsnum[v] == -1) {
      // (u, v) is a tree edge
      bridge_detection(v);
      if (low[v] > dfsnum[u])
        output (u, v) as a bridge

      low[u] = min(low[u], low[v]);
    } else {
      // (u, v) is a back edge
      low[u] = min(low[u], dfsnum[v]);
    }
  }
}
```

4 Application 3: Articulation Vertex

In an undirected graph, an **articulation vertex** is a vertex which if removed, will separate the graph into two disconnected component. This problem is very similar bridge detection and can be implemented with relative little changes to the above code. One thing to take note is that we must handle the case of the root of the DFS tree differently.

Using the same notation above, a non-root vertex u is an articulation vertex iff there exists a child v of u in the DFS tree such that`low[v] >= dfsnum[u]`. The root of the DFS tree is an articulation vertex iff it has more than one child.