Depth-First Traversal

Depth-First Traversal is another algorithm for traversing a graph. Called depth-first because it travels "deeper" in the graph whenever possible.

Edges are explored out of the most recently discovered vertex $v$ that still has unexplored edges. When all of $v$'s edges have been explored, the search "backtracks" to explore the edges incident on the vertex from which $v$ was discovered.

We will use an algorithm with a stack, $S$, to manage the set of nodes.

**Depth-First Search**

DFS algorithm maintains the following information for each vertex $u$:
- $u.c$ (white, gray, or black) : indicates status
  - white = not discovered yet
  - gray = discovered, but not finished
  - black = finished
- $u.d$ : discovery time of node $u$
- $u.f$ : finishing time of node $u$
- $u.\pi$ : predecessor of $u$ in Depth-First tree

**DFS node**

Each node has fields for predecessor ($\pi$), discovery time ($d$), finish time ($f$) and color ($c$). Each node also has an associated adjacency list with pointers to neighboring nodes.

```
<table>
<thead>
<tr>
<th>d</th>
<th>\pi</th>
<th>Adjacency List</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>\pi</td>
<td>Adjacency List</td>
</tr>
</tbody>
</table>
```

**Depth-First Search Using a Stack**

DFS ($G$, $s$)
1. time = 0
2. while $S$ is not empty
3. $u = S$.peek()
4. if $u.c$ == WHITE
5. $u.c$ = GRAY
6. $u.d$ = time
7. time = time + 1
8. for all white neighbors $v$ of $u$
9. $v.c$ = $u$
10. $S$.push($v$)
11. else if $u.c$ == GRAY
12. $S$.pop()
13. $u.c$ = BLACK
14. $u.f$ = time
15. time = time + 1
16. else // $u$ is BLACK
17. $S$.pop()
18. end while

**Depth-First Search (recursive version)**

DFS ($G$)
1. for each $w \in G$
2. if $w.c ==$ white
3. DFS-Visit ($G$, $w$)

DFS-Visit ($G$, $u$)
1. $u.c$ = gray
2. $u.d$ = time
3. time = time + 1
4. for each $v$ adjacent to $u$
5. if $v.c ==$ white
6. $v.\pi$ = $u$
7. DFS-Visit ($G$, $v$)
8. end if
9. end for
10. $u.c$ = black
11. $u.f$ = time
12. time = time + 1

Complexity is based on number of edges $|E|

Complexity (Adjacency List)
- check all edges adjacent to each node from both directions - $O(E)$ time
- total = $O(V + E) = O(V^2)$ (w.c.)

Note: If $G = (V, E)$ is not connected, then DFS will still visit the entire graph with the additional code above.
public class DepthFirstSearch {
    private boolean[] marked;
    private int[] edgeTo;
    private int count = 0;
    public DepthFirstSearch(Graph G, int s) {
        marked = new boolean[G.V()];
        dfs(G, s);
    }
    private void dfs(Graph G, int v) {
        marked[v] = true;
        count++;
        for each w adjacent to v
            if (!marked[w])
                edgeTo[w] = v;
                dfs(G, w);
    }
}

Enumerating shortest path, s→v

pathTo(v):
1. if (!marked[v]) return false
2. Stack<Integer> path = new Stack<Integer>()
3. for (int x = v; x != s; x = edgeTo[x])
4. path.push(x)
5. path.push(s)
6. return path

When pathTo finishes, the stack path will contain the dfs path from s to v and they can be popped off the stack in order.

Proposition B1: DFS marks all vertices connected to a given source in time proportional to the sum of their degrees.

Informal proof:
Every marked vertex is connected by a path to s since the algorithm finds vertices only by following edges. Now suppose that some unmarked vertex w is connected to s. Since s itself is marked, any path from s to w must have at least one edge from the set of marked to unmarked nodes. Since s is marked, any path from s to w must have at least one edge from the set of marked to unmarked vertices, say v-x. But the algorithm would have discovered x after marking v, so no such edge can exist, a contradiction. The time bound follows because marking ensures that each vertex is visited once (taking time prop to its degree to check marks

Analysis of Depth-First Search

Proposition B2: DFS allows us to provide clients with a path from a given source to any marked vertex in time proportional to the path length.

Informal proof:
By induction on the number of vertices visited, it follows that the the edgeTo array represents a tree rooted at the source. The pathTo method builds the path in time proportional to its length.

Example DFS Traversal

Depth-first Search Forest

Tree edges are solid lines and dashed lines are back edges.
Breadth-first Search Forest

Tree edges are solid lines and dashed lines are cross edges.

DFS Tree

DFS builds a depth-first tree whose edges can be traced from any node to s using the π values at each node. Or by using the edgeTo[] array in the non-object-oriented version.

The DFS algorithm defines a depth-first forest Gπ.

Topological Sort - Application of DFS

Complexity (Adjacency List Representation) - O(V + E)

Topologically sorted vertices are ordered in reverse order of their finishing times. An application of this type of sorting algorithm is to indicate precedence among ordered events represented in a DAG.

Classification of DFS Edges in a Directed Graph

1. Tree edges: Edges included in depth-first forest. Edge (u,v) is a tree edge if v was first discovered by edge (u,v).
2. Back edges: Edge (u,v) connects a vertex u to an ancestor (non-parent) v in a depth-first tree.
3. Forward edges: Edge (u,v) connects a vertex u to a descendant (non-child) v in a depth-first tree.
4. Cross edges: All other edges, i.e., between sibling nodes (e.g., nodes on different branches) of the same depth-first tree or between nodes in different depth-first trees.

Finding Strongly Connected Components of a Digraph

A digraph is strongly connected if, for any distinct pair of vertices u and v there exists a directed path from u to v and a directed path from v to u. In general, a digraph's vertices can be partitioned into disjoint maximal subsets of vertices that are mutually accessible via directed paths of the digraph; these subsets are called strongly connected components.

Finding Strongly Connected Components of a Digraph

The strongly connected components GSCC are exactly the subsets of vertices in each DFS tree obtained during step 3, the last traversal.

Time complexity of this algorithm?