Depth First Search Algorithm

Depth-first search is a systematic way to find all the vertices reachable from a source vertex, s. Historically, depth-first was first stated formally hundreds of years ago as a method for traversing mazes. Like breadth-first search, DFS traverse a connected component of a given graph and defines a spanning tree. The basic idea of depth-first search is this: It methodically explore every edge. We start over from different vertices as necessary. As soon as we discover a vertex, DFS starts exploring from it (unlike BFS, which puts a vertex on a queue so that it explores from it later).

Strategy of DFS Algorithm

Depth-first search selects a source vertex s in the graph and paint it as "visited." Now the vertex s becomes our current vertex. Then, we traverse the graph by considering an arbitrary edge (u, v) from the current vertex u. If the edge (u, v) takes us to a painted vertex v, then we back down to the vertex u. On the other hand, if edge (u, v) takes us to an unpainted vertex, then we paint the vertex v and make it our current vertex, and repeat the above computation. Sooner or later, we will get to a “dead end,” meaning all the edges from our current vertex u takes us to painted vertices. This is a deadlock.

To get out of this, we back down along the edge that brought us here to vertex u and go back to a previously painted vertex v. We again make the vertex v our current vertex and start repeating the above computation for any edge that we missed earlier. If all of v's edges take us to painted vertices, then we again back down to the vertex we came from to get to vertex v, and repeat the computation at that vertex. Thus, we continue to back down the path that we have traced so far until we find a vertex that has yet unexplored edges, at which point we take one such edge and continue the traversal. When the depth-first search has backtracked all the way back to the original source vertex, s, it has built a DFS tree of all vertices reachable from that source. If there still undiscovered vertices in the graph then it selects one of them as the source for another DFS tree. The result is a forest of DFS-trees.

Note that the edges lead to new vertices are called discovery or tree edges and the edges lead to already visited (painted) vertices are called back edges.

Like BFS, to keep track of progress depth-first-search colors each vertex. Each vertex of the graph is in one of three states:

1. Undiscovered;
2. Discovered but not finished (not done exploring from it); and
3. Finished (have found everything reachable from it) i.e. fully explored.

The state of a vertex, u, is stored in a color variable as follows:

1. color[u] = White - for the "undiscovered" state,
2. color[u] = Gray - for the "discovered but not finished" state, and
3. color[u] = Black - for the "finished" state.

Like BFS, depth-first search uses π[v] to record the parent of vertex v. We have π[v] = NIL if and only if vertex v is the root of a depth-first tree.

DFS time-stamps each vertex when its color is changed.

1. When vertex v is changed from white to gray the time is recorded in d[v].
2. When vertex v is changed from gray to black the time is recorded in f[v].

The discovery and the finish times are unique integers, where for each vertex the finish time is always after the discovery time. That is, each time-stamp is an unique integer in the range of 1 to 2|V| and for each vertex v, d[v] < f[v]. In other words, the following inequalities hold:

$1 \leq d[v] < f[v] \leq 2|V|$

Depth-First Search Algorithms :

The DFS forms a depth-first forest comprised of more than one depth-first trees. Each tree is made of edges (u, v) such that u is gray and v is white when edge (u, v) is explored. The following pseudocode for DFS uses a global timestamp time.

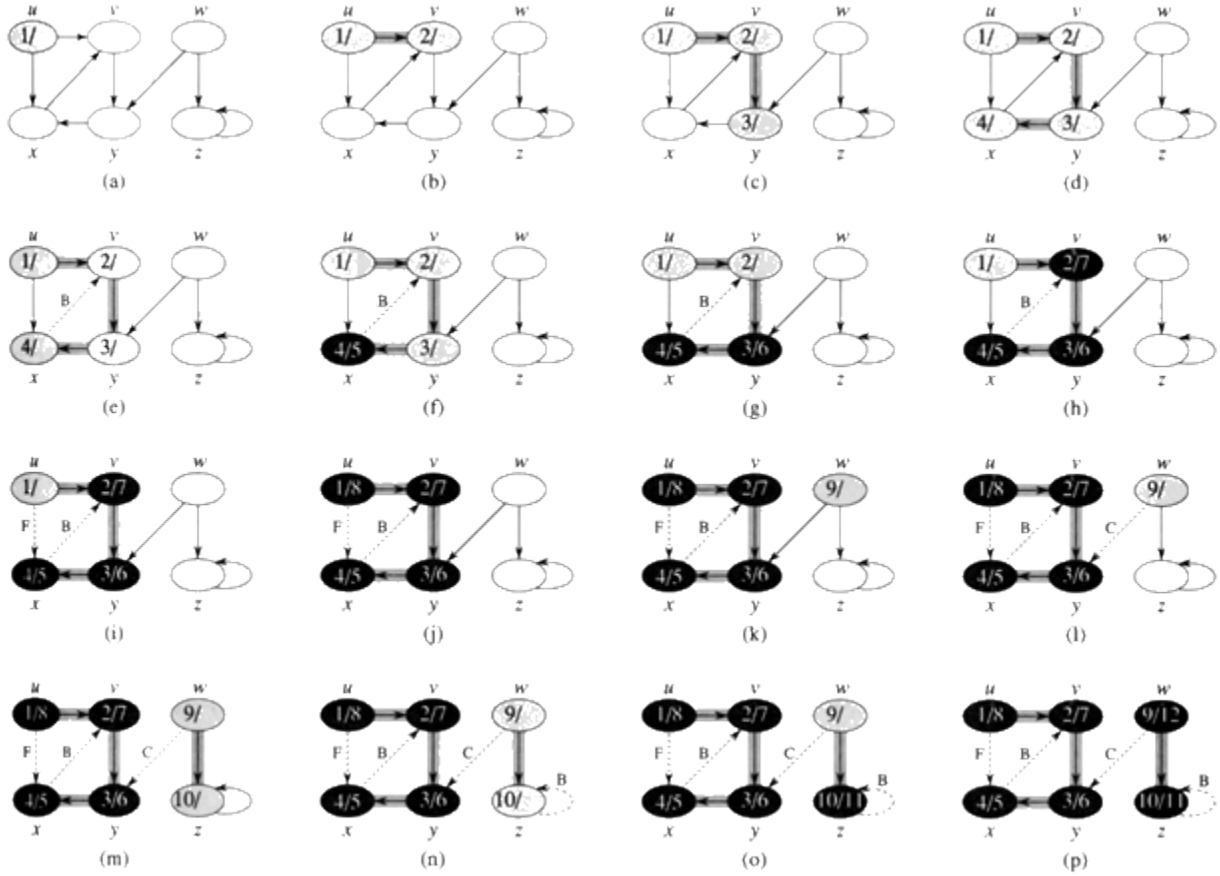
```
DFS (V, E) {
for each vertex u in V[G]
do color[u] ← WHITE
π[u] ← NIL
time ← 0
for each vertex u in V[G]
do if color[u] ← WHITE
then DFS-Visit(u)     /*build a new DFS-tree from u*/
}



DFS-Visit(u) {
color[u] ← GRAY     /*discover u*/
time ← time + 1
d[u] ← time
for each vertex v adjacent to u     /* explore (u, v) */
do if color[v] ← WHITE
then π[v] ← u
DFS-Visit(v)
color[u] ← BLACK
time ← time + 1
f[u] ← time    /*we are done with u */
```

```
}
```

Example: In the following figure, the solid edge represents discovery or tree edge and the dashed edge shows the back edge. Furthermore, each vertex has two time stamps: the first time-stamp records when vertex is first discovered and second time-stamp records when the search finishes examining adjacency list of vertex.



Analysis :

The analysis is similar to that of BFS analysis. The DFS-Visit is called (from DFS or from itself) once for each vertex in V[G] since each vertex is changed from white to gray once. The for-loop in DFS-Visit is executed a total of |E| times for a directed graph or 2|E| times for an undirected graph since each edge is explored once. Moreover, initialization takes Θ(|V|) time. Therefore, the running time of DFS is Θ(V + E).

Note that its Θ, not just O, since guaranteed to examine every vertex and edge.

Consider vertex u and vertex v in V[G] after a DFS. Suppose vertex v in some DFS-tree. Then we have d[u] < d[v] < f[v] < f[u] because of the following reasons:

1. Vertex u was discovered before vertex v; and
2. Vertex v was fully explored before vertex u was fully explored.

Note that converse also holds: if d[u] < d[v] < f[v] < f[u] then vertex v is in the same DFS-tree and a vertex v is a descendent of vertex u.

Suppose vertex u and vertex v are in different DFS-trees or suppose vertex u and vertex v are in the same DFS-tree but neither vertex is the descendent of the other. Then one vertex was discovered and fully explored before the other was discovered i.e., f[u] < d[v] or f[v] < d[u].

Algorithms based on DFS

Based upon DFS, there are O(V + E)-time algorithms for the following problems:

• Testing whether graph is connected.

• Computing a spanning forest of G.

• Computing the connected components of G.

• Computing a path between two vertices of G or reporting that no such path exists.

• Computing a cycle in G or reporting that no such cycle exists.

Application

As an application of DFS lets determine whether or not an undirected graph contains a cycle. It is not difficult to see that the algorithm for this problem would be very similar to DFS(G) except that when the adjacent edge is already a GRAY edge than a cycle is detected. While doing this the algorithm also takes care that it is not detecting a cycle when the GRAY edge is actually a tree edge from a ancestor to a descendent.

```
ALGORITHM DFS_DETECT_CYCLES [G] {
for each vertex u in V[G]
do color[u] ← WHITE
predecessor[u] ← NIL
time ← 0
for each vertex u in V[G]
do if color[u] ← WHITE
DFS_visit(u)
}
```

The sub-algorithm DFS_visit(u) is as follows:

```
DFS_visit(u) {
color(u) ← GRAY
d[u] ← time ← time + 1
for each v adjacent to u
do if color[v] ← GRAY and Predecessor[u] ≠ v
return "cycle exists"
if color[v] ← WHITE
do predecessor[v] ← u
recursively DFS_visit(v)
color[u] ← BLACK
```

```
f[u] ← time ← time + 1
}
```

Correctness - To see why this algorithm works suppose the node to visited v is a gray node, then there are two possibilities. The first possibility is that the node v is a parent node of u and we are going back the tree edge which we traversed while visiting u after visiting v. In that case it is not a cycle. The second possibility is that v has already been encountered once during DFS_visit and what we are traversing now will be back edge and hence a cycle is detected.

Time Complexity - The maximum number of possible edges in the graph G if it does not have cycle is |V| − 1. If G has a cycles, then the number of edges exceeds this number. Hence, the algorithm will detects a cycle at the most at the Vth edge if not before it. Therefore, the algorithm will run in O(V) time.

Source:

http://www.learnalgorithms.in/#