Inf 2B: Graphs II - Applications of DFS

Kyriakos Kalorkoti

School of Informatics
University of Edinburgh

Reminder: Recursive DFS

Algorithm dfs(G)

1. Initialise Boolean array *visited* by setting all entries to FALSE
2. **for all** $v \in V$ **do**
3.   **if not** *visited*$[v]$ **then**
4.   dfsFromVertex(G, v)

Algorithm dfsFromVertex(G, v)

1. *visited*$[v] \leftarrow$ TRUE
2. **for all** w adjacent to v **do**
3.   **if not** *visited*$[w]$ **then**
4.   dfsFromVertex(G, w)

Runtime: $T(n, m) = \Theta(n + m)$, using Adjacency List representation.

Trees and Forests

Definition: A *tree* is a connected graph without any cycles (disregarding directions of edges).

Note: In computing we use *rooted* trees, i.e., a distinguished vertex is chosen as the root.

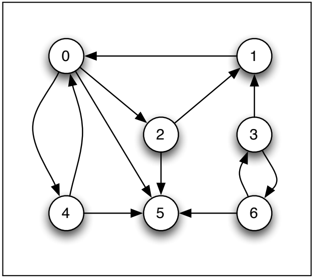
Definition: A *forest* is a collection of trees.

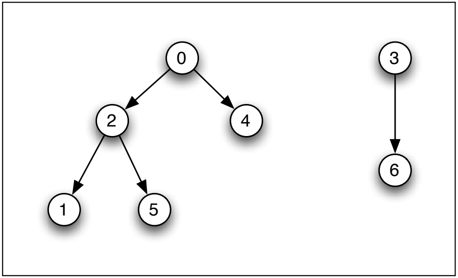
DFS Forests:

A DFS traversing a graph builds up a **forest**:

- vertices are all vertices of the graph,
- edges are those traversed during the DFS.

DFS Forests Example





Connected components of an undirected graph

$G = (V, E)$ undirected graph

Definition

- A subset C of V is *connected* if for all $v, w \in C$ there is a path from v to w (if G is directed, say *strongly connected*).
- A *connected component* of G is a maximal connected subset C of V.
  Maximal means no connected subset C' of V strictly contains C.
- G is *connected* if it only has one connected component, i.e., if for all vertices v, w there is a path from v to w.

Connected components (continued)

- Each vertex of an undirected graph is contained in exactly one connected component.
- For each vertex v of an undirected graph, the connected component that contains v is precisely the set of all vertices that are reachable from v.

For an undirected graph G, dfsFromVertex(G, v) visits exactly the vertices in the connected component of v.

Connected components (continued)

Algorithm connComp(G)

1. Initialise Boolean array *visited*
   by setting all entries to FALSE
2. **for all** $v \in V$ **do**
3.   **if** $visited[v] =$ FALSE **then**
4.   **print** "New Component"
5.   ccFromVertex(G, v)

Connected components (continued)

Algorithm ccFromVertex(G, v)

1. $visited[v] \leftarrow$ TRUE
2. **print** v
3. **for all** w adjacent to v **do**
4. **if** $visited[w] =$ FALSE **then**
5. ccFromVertex(G, w)

Topological Sorting

Example:
10 tasks to be carried out. Some of them depend on others.

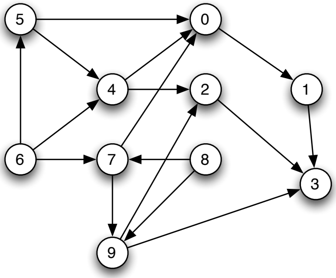
- Task 0 must be completed before Task 1 can be started.
- Task 1 and Task 2 must be done before Task 3 can start.
- Task 4 must be done before Task 0 or Task 2 can start.
- Task 5 must be done before Task 0 or Task 4 can start.
- Task 6 must be done before Task 4, 5 or 7 can start.
- Task 7 must be done before Task 0 or Task 9 can start.
- Task 8 must be done before Task 7 or Task 9 can start.
- Task 9 must be done before Task 2 or Task 3 can start.

Topological order

Definition

Let $G = (V, E)$ be a directed graph. A *topological order* of G is a total order $\prec$ of the vertex set V such that for all edges $(v, w) \in E$ we have $v \prec w$.

Example (continued)



Does this graph have a topological order?

Yes, the topological sort is:

$$8 \prec 6 \prec 7 \prec 9 \prec 5 \prec 4 \prec 2 \prec 0 \prec 1 \prec 3.$$

Topological order (continued)

A digraph that has a cycle does not have a topological order.

Definition

A *DAG* (**d**irected **a**cyclic **g**raph) is a digraph without cycles.

Theorem

A digraph has a topological order if and only if it is a DAG.

Classification of vertices during DFS

$G = (V, E)$ graph, $v \in V$. Consider dfs(G).

- v **finished** if dfsFromVertex(G, v) has been completed.

Vertices can be in the following states:

- not yet visited (call a vertex in this state *white*),
- visited, but not yet finished (*grey*).
- finished (*black*).

Classification of vertices during DFS (continued)

Lemma

Let G be a graph and v a vertex of G. Consider the moment during the execution of dfs(G) *when* dfsFromVertex(G, v) *is started. Then for all vertices w we have:*

1. *If w is* white *and reachable from v, then w will be* black *before v.*
2. *If w is* grey*, then v is reachable from w.*

Topological sorting

$G = (V, E)$ digraph. Define order on V as follows:

$$v \prec w \iff w \text{ becomes black before } v.$$

Theorem

If G is DAG then $\prec$ is a topological order.

Proof.

Suppose $(v, w) \in E$. Consider dfsFromVertex(G, v).

- If w is already *black*, then $v \prec w$.
- If w is *white*, then by Lemma part 1, w will be *black* before v. Thus $v \prec w$.
- If w is *grey*, then by Lemma part 2, v is reachable from w. So there is a path p from w to v. Path p and edge (v, w) together form a cycle. **Contradiction!** (G is acyclic ...)

□

Topological sorting (continued)

Algorithm topSort(G)

1. Initialise array *state* by setting all entries to *white*.
2. Initialise linked list L
3. **for all** $v \in V$ **do**
4.   **if** *state*$[v]$ $=$ *white* **then**
5.   sortFromVertex(G, v)
6. **print** L

Topological sorting (continued)

Algorithm sortFromVertex(G, v)

```
state[v] ← grey
for all w adjacent to v do
    if state[w] = white then
        sortFromVertex(G, w)
    else if state[w] = grey then
        print "G has a cycle"
        halt
state[v] ← black
L.insertFirst(v)
```