

# Introduction to Algorithms and Data Structures

## Lecture 15: DFS and graph structure

Mary Cryan

School of Informatics  
University of Edinburgh

# DFS (using a stack)

## Algorithm $\text{dfs}(G)$

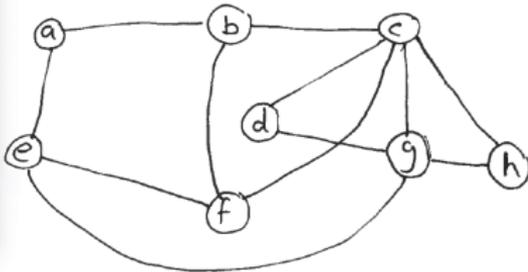
1. Initialise Boolean array  $visited$ , setting all to FALSE
2. Initialise *Stack*  $S$
3. **for all**  $v \in V$  **do**
4.     **if**  $visited[v] = \text{FALSE}$  **then**
5.          $\text{dfsFromVertex}(G, v)$

## DFS (using a stack)

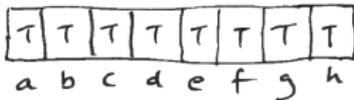
**Algorithm** dfsFromVertex( $G, v$ )

1.  $S.push(v)$
2. **while not**  $S.isEmpty()$  **do**
3.      $u \leftarrow S.pop()$
4.     **if**  $visited[u] = \text{FALSE}$  **then**
5.          $visited[u] \leftarrow \text{TRUE}$
6.         **for all**  $w$  adjacent to  $u$  **do**
7.             **if**  $visited[w] = \text{FALSE}$  **then**
8.                  $S.push(w)$

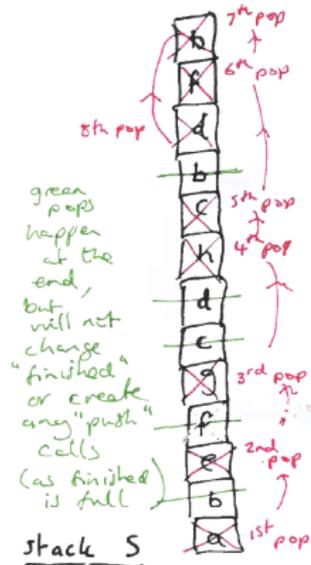
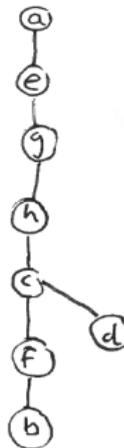
# DFS worked example



visited



dfs



# Recursive DFS (no explicit Stack)

## Algorithm $\text{dfs}(G)$

1. Initialise Boolean array  $visited$ , setting all entries to FALSE
2. **for all**  $v \in V$  **do**
3.     **if**  $visited[v] = \text{FALSE}$  **then**
4.          $\text{dfsFromVertex}(G, v)$

## Algorithm $\text{dfsFromVertex}(G, v)$

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

(We will have reversed prioritisation of the vertices adjacent to  $v$ , compared to the Stack version)

# Analysis of Recursive DFS

## Lemma

*During  $\text{dfs}(G)$ ,  $\text{dfsFromVertex}(G, v)$  is invoked exactly once for each vertex  $v$ .*

## Proof.

*At least once:*

- ▶  $\text{visited}[v]$  can only become `TRUE` when  $\text{dfsFromVertex}(G, v)$  is executed.
- ▶ If  $\text{visited}[v]$  remains `FALSE`,  $\text{dfsFromVertex}(G, v)$  will eventually be called by line 4 of  $\text{dfs}(G)$ .

*At most once:*

- ▶ First call of  $\text{dfsFromVertex}(G, v)$  sets  $\text{visited}[v]$  to `TRUE`.
- ▶ After  $\text{visited}[v]$  is `TRUE`,  $\text{dfsFromVertex}(G, v)$  is *never* called again.



(“At most once” is also true for Stack dfs, but “at least once” is not.  $\text{dfsFromVertex}$  is more to “start a component” in the Stack version)

## Analysis of recursive DFS (cont'd)

### Lemma

*For a directed graph,  $\sum_{v \in V} \text{out-degree}(v) = m$ .*

*For an undirected graph,  $\sum_{v \in V} \text{deg}(v) = 2m$ .*

### Proof.

Every edge is counted exactly once on both sides of the equation (for directed).

For the undirected case, every edge is counted twice on the lhs. □

# Analysis of recursive DFS

$G = (V, E)$  graph with  $n$  vertices and  $m$  edges

**Algorithm** dfs( $G$ )

1. Initialise Boolean array *visited*, setting all to FALSE
2. **for all**  $v \in V$  **do**
3.     **if** *visited*[ $v$ ] = FALSE **then**
4.         dfsFromVertex( $G, v$ )

▶ dfs( $G$ ): Ignoring calls to dfsFromVertex, total time  $\Theta(n)$

▶ dfsFromVertex( $v$ ) is called at most once for every vertex  $v$ , and does  $\Theta(\text{out-degree}(v))$  work, excluding recursive calls.

Overall time:

$$\begin{aligned} T(n, m) &= \Theta(n) + \sum_{v \in V} \Theta(\text{out-degree}(v)) \\ &= \Theta\left(n + \sum_{v \in V} \text{out-degree}(v)\right) \\ &= \Theta(n + m) \end{aligned}$$

## Adjacency List or Adjacency Matrix?

We said each call to `dfsFromVertex( $v$ )` takes  $\Theta(\text{out-degree}(v))$  time (excluding recursive calls).

**Algorithm** `dfsFromVertex( $G, v$ )`

1. `visited[v] ← TRUE`
2. **for all**  $w$  adjacent to  $v$  **do**
3.     **if** `visited[w] = FALSE` **then**
4.         `dfsFromVertex( $G, w$ )`

If we are iterating over “all  $w$  adjacent to  $v$ ” in  $\Theta(\text{out-degree}(v))$  time, then we **must** be using an Adjacency list structure.

# Analysis of Stack DFS

Compare the two  $\text{dfsFromVertex}(G, v)$  methods:

**Algorithm**  $\text{dfsFromVertex}(G, v)$

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

**Algorithm**  $\text{dfsFromVertex}(G, v)$

1.  $S.\text{push}(v)$
2. **while not**  $S.\text{isEmpty}()$  **do**
3.      $u \leftarrow S.\text{pop}()$
4.     **if**  $\text{visited}[u] = \text{FALSE}$  **then**
5.          $\text{visited}[u] \leftarrow \text{TRUE}$
6.     **for all**  $w$  adjacent to  $u$  **do**
7.         **if**  $\text{visited}[w] = \text{FALSE}$  **then**
8.              $S.\text{push}(w)$

|  $\text{visited}[w] \leftarrow \text{TRUE}$  |  $\leftrightarrow$  |  $u \leftarrow S.\text{pop}(); \text{visited}[u] \leftarrow \text{TRUE};$  |

Recursive: marks  $v$  as “visited”, *then* calls  $\text{dfsFromVertex}$  for unvisited adjacent vertices

Iterative: “pops”  $v$  off top to mark as “visited” and explore/push adjacent vertices.

However, the number of Stack operations for  $v$  is bounded in terms of *the number of edges into*  $v \Rightarrow$  the overall runtime for our original dfs is still  $\Theta(n + m)$ .

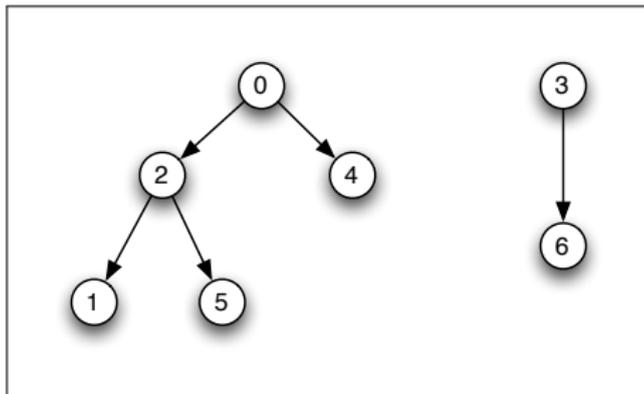
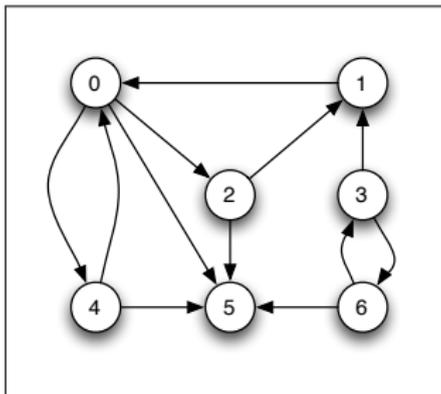
## DFS Forests

A DFS traversing a graph builds up a *forest* whose vertices are all vertices of the graph and whose edges are all vertices traversed during the DFS.

### Definition

(in recursive DFS) a vertex  $w$  is a *child* of a vertex  $v$  in the DFS forest if  $\text{dfsFromVertex}(G, w)$  is called from  $\text{dfsFromVertex}(G, v)$ .

## DFS Forests Example



On directed graphs, the connected components (trees) might vary depending on the order in which we consider vertices at the top-level of dfs.

# 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.

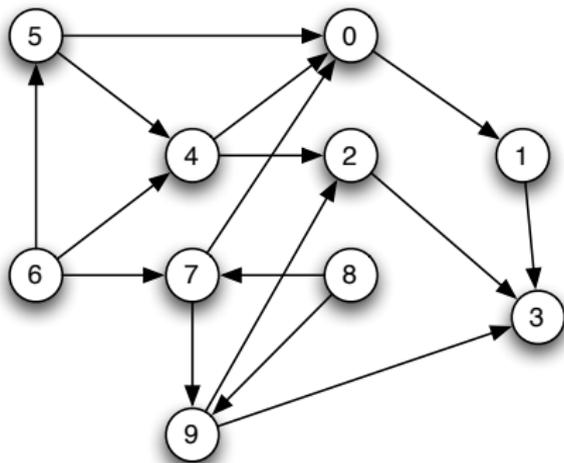
# Topological order

## 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  $v \prec w$ .

(in some fields this is called a *linear extension*)

## Tasks as a (directed) graph



Does this graph have a topological order?

Yes. One topological sort is:

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

## Topological order (cont'd)

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 recursive DFS

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

- ▶  $v$  is **finished** if  $\text{dfsFromVertex}(G, v)$  has been completed.

Vertices can be in the following states:

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

(note these colours are *explicitly* marked in version of DFS by [CLRS] 22.3)

## Classification of vertices during recursive DFS (cont'd)

### Lemma

*Let  $G$  be a graph and  $v$  a vertex of  $G$ . Consider the moment during the execution of  $\text{dfs}(G)$  when  $\text{dfsFromVertex}(G, v)$  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 a DAG then  $\prec$  is a topological order.

## Proof.

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

- ▶ If  $w$  is already *black*, then  $v \prec w$  (and this is what we want).
- ▶ 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 implemented

**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 implemented

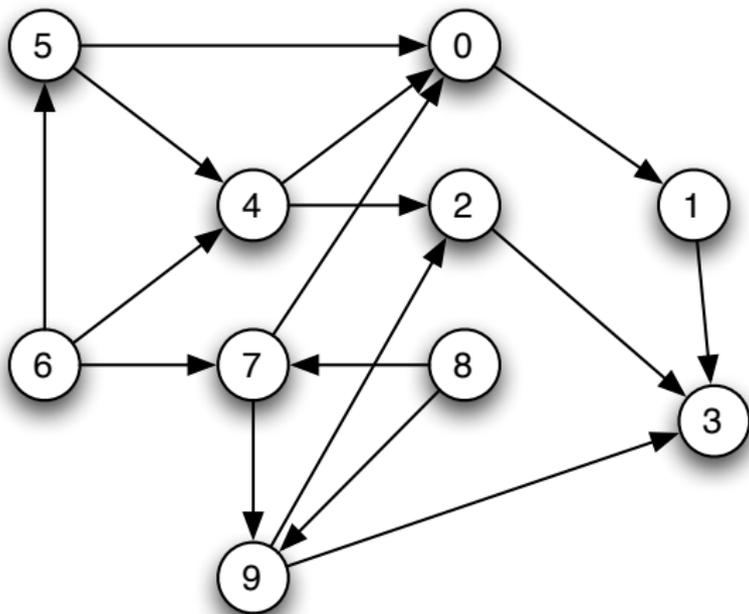
**Algorithm** sortFromVertex( $G, v$ )

1.  $state[v] \leftarrow grey$
2. **for all**  $w$  adjacent to  $v$  **do**
3.     **if**  $state[w] = white$  **then**
4.         sortFromVertex( $G, w$ )
5.     **else if**  $state[w] = grey$  **then**
6.         **print** “ $G$  has a cycle”
7.         **halt**
8.  $state[v] \leftarrow black$
9.  $L.insertFirst(v)$

Difference from dfs itself - the order the vertices get added to the list.

Running-time is again  $\Theta(n + m)$

## Example



Use the algorithm `topSort` to compute a topological sort of this graph.

# 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 **maximum connected subset**  $C$  of  $V$ . (no connected subset  $C'$  of  $V$  strictly contains  $C$ ).
- ▶  $G$  is *connected* if it only has one connected component, that is, if for all vertices  $v, w$  there is a path from  $v$  to  $w$ .

## Connected components - undirected (cont'd)

- ▶ 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  $v$ .

And the same is true for `bfsFromVertex( $G, v$ )` (either will do!)

## Connected components - undirected (cont'd)

**Algorithm** connComp( $G$ )

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

**Algorithm** ccFromVertex( $G, v$ )

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

# Reading

From [CLRS]:

- ▶ Depth-first search - Section 22.3
- ▶ Computing topological sort - Section 22.4

From “Algorithms Illuminated”:

- ▶ sections 8.3, 8.4, 8.5

Hope you get a break over the holidays!