Algorithm dfs(G)

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

1. visited[v] ← TRUE
2. S.push(v)
3. while not S.isEmpty() do
4.    u ← S.pop()
5.    for all w adjacent to u do
6.      if visited[w] = FALSE then
7.        visited[w] ← TRUE
8.        S.push(w)
DFS worked example
Recursive DFS (no explicit Stack)

**Algorithm** \( \text{dfs}(G) \)

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

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

1. \text{visited}[v] \leftarrow \text{TRUE} 
2. \textbf{for all} \ w \ \text{adjacent to} \ v \ \textbf{do} 
3. \hspace{1em} \textbf{if} \ \text{visited}[w] = \text{FALSE} \ \textbf{then} 
4. \hspace{2em} \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 dfs(G), dfsFromVertex(G, v) is invoked exactly once for each vertex v.

Proof.

At least once:

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

At most once:

- First call of dfsFromVertex(G, v) sets visited[v] to TRUE.
- After visited[v] is TRUE, dfsFromVertex(G, v) is never called again.

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

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

\( \text{dfs}(G) \): Ignoring calls to \( \text{dfsFromVertex} \), total time \( \Theta(n) \)

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

Overall time:

\[
T(n, m) = \Theta(n) + \sum_{v \in V} \Theta(\text{out-degree}(v)) \\
= \Theta\left(n + \sum_{v \in V} \text{out-degree}(v)\right) \\
= \Theta(n + m)
\]
Adjacency List or Adjacency Matrix?

We said each call to dfsFromVertex(ν) takes Θ(out-degree(ν)) time (excluding recursive calls).

Algorithm dfsFromVertex(G, ν)

1. \textit{visited}[ν] ← TRUE
2. \textbf{for all} w adjacent to ν \textbf{do}
3. \hspace{1em} \textbf{if} visited[w] = \text{FALSE} \textbf{then}
4. \hspace{2em} dfsFromVertex(G, w)

If we are iterating over “all w adjacent to ν” in Θ(out-degree(ν)) time, then we must be using an Adjacency list structure.
Analysis of Stack DFS

Compare the two dfsFromVertex\((G, v)\) methods:

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

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

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

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

| \(visited[w] \leftarrow \text{TRUE}\) | \(\leftarrow\) | \(visited[w] \leftarrow \text{TRUE}; S\).push\((w)\) |

Recursive: marks \(v\) as “visited”, then calls dfsFromVertex for unvisited adjacent vertices

Iterative: marks \(v\) as “visited”, “pops” top to “push” all adjacent vertices … iterates.

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
A vertex \( w \) is a child of a vertex \( v \) in the DFS forest if \( \text{dfsFromVertex}(G, w) \) is called from \( \text{dfsFromVertex}(G, v) \).
Q2 of tutorial sheet 5 considers how the connected components can 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)
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* (directed acyclic graph) 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**).
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 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 . . . )
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. \(\text{state}[v] \leftarrow \text{grey} \)
2. for all \(w\) adjacent to \(v\) do
3.     if \(\text{state}[w] = \text{white}\) then
4.         sortFromVertex\((G, w)\)
5.     else if \(\text{state}[w] = \text{grey}\) then
6.         print “\(G\) has a cycle”
7.         halt
8.     \(\text{state}[v] \leftarrow \text{black} \)
9. \(L.\text{insertFirst}(v)\)

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

Running-time is again \(\Theta(n + m)\)
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 \( w \).
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 \( G \), \( \text{dfsFromVertex}(G, v) \) visits exactly the vertices in the connected component of \( v \).

And the same is true for \( \text{bfsFromVertex}(G, v) \) (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 ∈ V do
3. if visited[ v ] = FALSE then
4. print “New Component”
5. ccFromVertex( G, v )

Algorithm\ ccFromVertex( G, v )

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

From [CLRS] as usual:

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

Hope you get a break over the holidays!

And “see” you in 2023.