CS 341: ALGORITHMS

Lecture 16: graphs – topological sort, DAG testing, strongly connected components

Slides by Trevor Brown (some material from Doug Stinson)

trevor.brown@uwaterloo.ca

(http://tbrown.pro)
DFS APPLICATION: TESTING WHETHER A GRAPH IS A DAG

A directed graph $G$ is a directed acyclic graph, or DAG, if $G$ contains no directed cycle.
Lemma 6.7

A directed graph is a DAG if and only if a depth-first search encounters no back edges.

Proof.

(⇒): Any back edge creates a directed cycle.

Back edge: points to an ancestor in the DFS forest
• Case ($\leftarrow$): Suppose $\exists$ directed cycle. Show $\exists$ back edge.

• Let $v_1, v_2, \ldots, v_k, v_1$ be a directed cycle
• WLOG let $v_1$ be earliest discovered node in the cycle

Consider edge $\{v_k, v_1\}$

Since $d[v_1] < d[v_k]$, $\{v_k, v_1\}$ must be a **back** or **cross** edge. Why?

So when $v_1$ is discovered, $v_2, \ldots, v_k$ are all white

Recall: nodes become **gray** *when discovered*

Recall: every node $v_i$ that is *white-reachable* from $v_1$ when we discover $v_1$ (call $DFSVisit(v_1)$) turns **black** before $v_1$ ($f[v_i] < f[v_1]$)

So $v_k$ must turn black **before** $v_1$, and we have $f[v_k] < f[v_1]$.

Thus, $\{v_k, v_1\}$ must be a **back edge**. QED

- **edge type**
  - tree
  - forward
  - back
  - cross

- **discovery/finish times**
  - $d[v_k] < d[v_1] < f[v_1] < f[v_k]$
When we observe an edge from $u$ to $v$, check if $v$ is gray.

Lemma 6.7

A directed graph is a DAG if and only if a depth-first search encounters no back edges.

- Search for back edges
- How to identify a back-edge?

<table>
<thead>
<tr>
<th>edge type</th>
<th>colour of $v$</th>
<th>discovery/finish times</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree</td>
<td>white</td>
<td>$d[u] &lt; d[v] &lt; f[v] &lt; f[u]$</td>
</tr>
<tr>
<td>forward</td>
<td>black</td>
<td>$d[u] &lt; d[v] &lt; f[v] &lt; f[u]$</td>
</tr>
<tr>
<td>back</td>
<td>gray</td>
<td>$d[v] &lt; d[u] &lt; f[u] &lt; f[v]$</td>
</tr>
<tr>
<td>cross</td>
<td>black</td>
<td>$d[v] &lt; f[v] &lt; d[u] &lt; f[u]$</td>
</tr>
</tbody>
</table>
Algorithm: $DFS(G)$

$DAG \leftarrow \text{true}$

for each $v \in V(G)$

\[
\begin{align*}
\text{do} & \quad \{ \text{colour}[v] \leftarrow \text{white} \\
& \qquad \{ \pi[v] \leftarrow \emptyset \\
& \quad \text{time} \leftarrow 0 \\
\text{do} & \quad \text{if } \text{colour}[v] = \text{white} \\
& \qquad \text{then } DFSvisit(v) \\
\text{return } (DAG)
\end{align*}
\]
Algorithm: $DFSvisit(v)$

$colour[v] \leftarrow \text{gray}$

$time \leftarrow time + 1$

$d[v] \leftarrow time$

comment: $d[v]$ is the discovery time for vertex $v$

for each $w \in Adj[v]$

\[
\begin{cases}
\text{if } colour[w] = \text{white} \\
\quad \text{then } \\
\quad \quad \pi[w] \leftarrow v \\
\quad \quad DFSvisit(w)
\end{cases}
\]

if $colour[w] = \text{gray}$ then $DAG \leftarrow \text{false}$

$colour[v] \leftarrow \text{black}$

$time \leftarrow time + 1$

$f[v] \leftarrow time$
Back edge found! So we set DAG = false
TOPOLOGICAL SORT

Finding node orderings that satisfy given constraints
**Example problem:** getting dressed in the morning

**Example solution:**
- **Pants before belt**
- **Socks before shoes**
- **Watch any time**

**Diagram notes:**
- Edge \( \{u, v\} \) means \( u \) must be completed before \( v \)
- Could do various things first. Which ones are possible? What do they have in common?
Topological sort

Try to order nodes linearly so there are only pointers from left to right!

IFF there is a (directed) cycle!

Might not be possible! How can this happen?

Try to order nodes linearly so there are only pointers from left to right!
Goal: output a serial order of tasks that can be run without worrying about dependencies (because all dependencies are already satisfied by the time a task is run).

(Nodes are numbered according to one such order.)

Can even schedule tasks to run in parallel! Can do 1||2 then 3||13 then 4||5||15 etc.
A directed graph $G = (V, E)$ has a **topological ordering**, or **topological sort**, if there is a linear ordering $<$ of all the vertices in $V$ such that $u < v$ whenever $uv \in E$.
Lemma 6.5

A DAG contains a vertex of indegree 0.

Proof.

Suppose we have a directed graph in which every vertex has positive indegree. Let \( v_1 \) be any vertex. For every \( i \geq 1 \), let \( v_{i+1}v_i \) be an arc. In the sequence \( v_1, v_2, v_3, \ldots \), consider the first repeated vertex, \( v_i = v_j \) where \( j > i \). Then \( v_j, v_{j-1}, \ldots, v_i, v_j \) is a directed cycle.

One of these must be repeated. So there is a cycle!
Theorem 6.6

A directed graph $D$ has a topological sort if and only if it is a DAG.

Proof.

$(\Rightarrow)$: Suppose $D$ has a directed cycle $v_1, v_2, \ldots, v_j, v_1$. Then $v_1 < v_2 < \cdots < v_j < v_1$, so a topological ordering does not exist.

$(\Leftarrow)$: Suppose $D$ is a DAG. Then the algorithm below constructs a topological ordering.
We call a node with indegree 0 a **source**. So this step is enqueueing all source nodes.

Source nodes have no unsatisfied dependencies (are ready to be added to the topological sort).

If we create a new source, enqueue it.
EXAMPLE (Kahn’s Algorithm)

Compute **indegree** for all vertices

For each **u** in **V**
   For each **w** in **adj(u)**
      **w.deg** = **w.deg** + 1

Sources go into the queue

Until **Q** is empty: pop, output that element, decrement its neighbours, enqueue new sources
Algorithm: *Kahn(D)*

- **compute** \( \text{deg}^- (v) \) for all vertices \( v \)
- **for all** \( v \) such that \( \text{deg}^- (v) = 0 \), **insert** \( v \) into a queue \( Q \)
- **for** \( i \leftarrow 1 \) **to** \( n \)
  - **if** \( Q \) is empty
    - **then return** \( () \)
  - **else**
    - **let** \( v \) be the first vertex in \( Q \)
    - **remove** \( v \) from \( Q \) and **output** \( v \)
    - **for all** \( w \) in \( \text{Adj}(v) \)
      - **do**
        - **if** \( \text{deg}^- (w) = 0 \)
          - **then** **insert** \( w \) into \( Q \)
        - \( \text{deg}^- (w) \leftarrow \text{deg}^- (w) - 1 \)

**Running time with adjacency lists?**

- Total \( O(n + m) \)
- **iterations** \( O(n) \)
- **per check** \( O(1) \)
- **top()** \( O(1) \)
- **dequeue()** \( O(1) \)
- \( \sum \text{deg}(w) = m \)
- **inside loop** \( O(1) \)

**Total** \( O(n + m) \)
TOPOLOGICAL SORT VIA DFS

- We can also implement topological sort by using **DFS**!
- The **finishing times** of nodes help us
- Understanding this algo will be **key** for understanding **strongly connected components**
Lemma 6.8

Suppose $D$ is a DAG. Then $f[v] < f[u]$ for every arc $uv$.

<table>
<thead>
<tr>
<th>edge type</th>
<th>colour of $v$</th>
<th>discovery/finish times</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree</td>
<td>white</td>
<td>$d[u] &lt; d[v] &lt; f[v] &lt; f[u]$</td>
</tr>
<tr>
<td>forward</td>
<td>black</td>
<td>$d[u] &lt; d[v] &lt; f[v] &lt; f[u]$</td>
</tr>
<tr>
<td>back</td>
<td>gray</td>
<td>$d[u] &lt; d[v] &lt; f[v] &lt; f[u]$</td>
</tr>
<tr>
<td>cross</td>
<td>black</td>
<td>$d[v] &lt; f[v] &lt; d[u] &lt; f[u]$</td>
</tr>
</tbody>
</table>

Recall from DAG-testing: there are no back edges in a DAG.

Theorem: if $D$ is a DAG, and we order vertices in reverse order of finishing time, then we get a topological ordering!

To see why, suppose $D$ is a DAG and we order nodes in this way, so $f_{v_1} > f_{v_2} > \cdots > f_{v_{n-1}} > f_{v_n}$.

For contradiction, suppose a right-to-left edge $\{u, v\}$ exists.

Since edge $\{u, v\}$ exists, the lemma implies $f_v < f_u$.

But this contradicts the node ordering!

So all edges are left-to-right, hence this is a topological sort.
Algorithm: $DFS(G)$

1. InitializeStack($S$)
2. $DAG \leftarrow true$
3. For each $v \in V(G)$
   - $colour[v] \leftarrow white$
   - $\pi[v] \leftarrow \emptyset$
   - $time \leftarrow 0$
4. For each $v \in V(G)$
   - If $colour[v] = white$
     - Then $DFSvisit(v)$
5. If $DAG$ then return $(S)$ else return $(DAG)$
Algorithm: DFSvisit(v)

\[ colour[v] \leftarrow \text{gray} \]
\[ time \leftarrow time + 1 \]
\[ d[v] \leftarrow time \]

comment: \( d[v] \) is the discovery time for vertex \( v \)

for each \( w \in Adj[v] \)

\[
\begin{cases} 
\text{if} \ colour[w] = \text{white} \\
\quad \text{then} \quad \{ \\
\quad \quad \pi[w] \leftarrow v \\
\quad \quad \text{DFSvisit}(w) \\
\quad \}
\text{if} \ colour[w] = \text{gray} \quad \text{then} \quad \text{DAG} \leftarrow \text{false}
\end{cases}
\]

\[ colour[v] \leftarrow \text{black} \]

\[ \text{Push}(S, v) \]
\[ time \leftarrow time + 1 \]
\[ f[v] \leftarrow time \]

Running time \( O(n + m) \) with adjacency lists

We are pushing smallest finishing times first into a stack, so when we pop them out, we will get largest finishing time first

Save each node when it finishes
The initial calls are \(\text{DFSvisit}(1)\), \(\text{DFSvisit}(2)\) and \(\text{DFSvisit}(3)\).

The discovery/finish times are as follows:

<table>
<thead>
<tr>
<th>(v)</th>
<th>(d[v])</th>
<th>(f[v])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(v)</th>
<th>(d[v])</th>
<th>(f[v])</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

The topological ordering is 3, 2, 5, 4, 1, 6 (reverse order of finishing time).
I renamed "my documents" on your computer to "our documents".

You haven't texted me in 1 minute and 42 seconds. Why are you ignoring me?

Strongly connected components.
These are called strongly connected components (SCCs)
STRONGLY CONNECTED COMPONENTS

- It could also be divided into **three graphs**...

- But we want our SCCs to be **maximal** (as large as possible)
STRONGLY CONNECTED COMPONENTS

• So, the goal is to find these (maximal) SCCs:
FORMAL DEFINITIONS

For two vertices $x$ and $y$ of $G$, define $x \sim y$ if $x = y$; or if $x \neq y$ and there exist directed paths from $x$ to $y$ and from $y$ to $x$.

The relation $\sim$ is an **equivalence relation**.

The **strongly connected components** of $G$ are the equivalence classes of vertices defined by the relation $\sim$.

A strongly connected component of a digraph $G$ is a maximal strongly connected subgraph of $G$.

Note: a connected component can contain just a **single node**.

Example: a node with no out-edges.
Consider this graph. These are its SCCs. The following is its component graph. It has one node for each SCC. And an edge between two nodes IFF there is an edge between the corresponding SCCs. 

Can there be a cycle in the component graph? No! If there are paths both ways between components, they are actually the same SCC. Component graph is a DAG!
APPLICATIONS OF SCCs AND COMPONENT GRAPHS

• Finding all cyclic dependencies in code
• Can find single cycle with an easier DFS-based algorithm
• But it is nicer to find all cycles at once, so you don’t have to fix one to expose another
APPLICATIONS OF SCCs AND COMPONENT GRAPHS

- **Data filtering** before running other algorithms
- Consider Google maps; nodes = intersections, edges = roads
- Don’t want to run path finding algorithm on the entire **global** graph!
- First restrict execution to a rectangle
- Then throw away everything except the (maximal) SCC containing source & target
BRAINSTORMING AN ALGORITHM

• What if we run DFS, then reverse all edges, then run DFS (like checking whether an entire graph is strongly connected?)

This will definitely visit every node in a’s SCC

And in fact it might visit other SCCs as well…

<table>
<thead>
<tr>
<th>DFSVisit(a)</th>
<th>DFSVisit(h)</th>
<th>DFSVisit(j)</th>
</tr>
</thead>
</table>

Showing discovery times

Showing finish times
What if we run DFS, then reverse all edges, then run DFS?

We fail to identify SCC \{ h, i \}

Problem: from h, we can reach other SCCs

What if we perform DFSVisit calls in a different order?

Other reachable SCCs should be visited first

Then, each DFSVisit will visit exactly one SCC
What if we DFS visit $H$ according to a topological order in $C_G$ (some edges to unfinished SCCs)

In $C_H$: all edges to finished SCCs

Consider **component graph** $C_G$ of $G$ (which we want to compute)

**Idea:** in a DAG, reverse finish time would be a topological sort!

*Yes!* We will prove this…

$G$ might not be a DAG… but $C_G$ is! Does reverse finish order in $G$ give us a topological sort of $C_G$?
**PROVING DECREASING FINISH TIMES INDUCE A TOPOLOGICAL ORDER ON THE COMPONENT GRAPH**

- **Definition:** For a strongly connected component $C$, let $d[C] = \min\{d[v] : v \in C\}$ and $f[C] = \max\{f[v] : v \in C\}$

- **Lemma:** If $C_i, C_j$ are SCCs and there is an edge $C_i \rightarrow C_j$, then $f[C_i] > f[C_j]$

- **Proof.** Case 1 ($d[C_i] < d[C_j]$):
  - Let $u$ be the earliest discovered node in $C_i$
  - All nodes in $C_i \cup C_j$ are white-reachable from $u$, so they are *descendants in the DFS forest* and finish before $u$
  - So $f[C_i] > f[C_j]$

$u$ = earliest discovered node in here
PROVING DECREASING FINISH TIMES INDUCE A TOPOLOGICAL ORDER ON THE COMPONENT GRAPH

• **Definition:** For a strongly connected component $C$, let $d[C] = \min\{d[v] : v \in C\}$ and $f[C] = \max\{f[v] : v \in C\}$

• **Lemma:** if $C_i, C_j$ are SCCs and there is an edge $C_i \rightarrow C_j$, then $f[C_i] > f[C_j]$

• **Proof.** Case 2 ($d[C_i] > d[C_j]$):
  - Since component graph is a DAG, there is **no path** $C_j \rightarrow C_i$
  - Thus, **no nodes** in $C_i$ are reachable from $C_j$
  - So we discover $C_j$ and finish $C_j$ **without** discovering $C_i$
  - Therefore $d[C_j] < f[C_j] < d[C_i] < f[C_i]$. QED
CONSEQUENCE OF THE LEMMA

• So, if we perform $DFSVisit(u)$ on nodes from **largest to smallest** finishing time, any **other SCCs** reachable from the current SCC must already be **finished/black**

• So each $DFSVisit(u)$ call will **explore precisely one SCC**
USING THE LEMMA TO BUILD AN ALGORITHM

• Algorithm:
  • \((v_{i_1}, v_{i_2}, \ldots, v_{i_n}) := DFS\_topsort (G)\)
  • \(H := \text{construct by reversing each edge in } G\)
  • return := DFS\_SCC(H, (v_{i_1}, v_{i_2}, \ldots, v_{i_n}))

This is called Sharir’s algorithm (sometimes Kosaraju’s algorithm). This paper first introduced it.

Topological sort algorithm we saw earlier. Returns nodes ordered by finish time from largest to smallest.

Calls DFSVisit on nodes in topological order, and gives each node an SCC number in a component[] array, which is then returned.
Assume that \( f[v_{i_1}] > f[v_{i_2}] > \cdots > f[v_{i_n}] \).

**Algorithm:** \( DFS(H, (v_{i_1}, v_{i_2}, ..., v_{i_n})) \)

```plaintext
for j ← 1 to n
    do colour[v_{i_j}] ← white

sec ← 0

for j ← 1 to n
    do if colour[v_{i_j}] = white
        then sec ← sec + 1
            DFSvisit(H, v_{i_j}, sec)

return (comp)
```

**Algorithm:** \( DFSvisit(H, v, sec) \)

```plaintext
colour[v] ← gray
comp[v] ← sec

for each w ∈ Adj[v]
    do if colour[w] = white
        then DFSvisit(H, w, sec)

colour[v] ← black
```

**PSEUDOCODE**
Running Sharir’s Algorithm

Phase 1: DFS topological sort

Phase 2: DFSVisit reverse graph by reverse finish times

DFSVisit(j)  DFSVisit(h)  DFSVisit(e)  DFSVisit(a)  $scc = 4$

$scc$ is shown
PSEUDOCODE

Assume that \( f[v_1] > f[v_2] > \cdots > f[v_n] \).

**Algorithm:** \( DFS(H, (v_1, v_2, \ldots, v_n)) \)

for \( j \leftarrow 1 \) to \( n \)
    do \( colour[v_i] \leftarrow \text{gray} \)

\( \text{comp} \leftarrow 0 \)

for \( j \leftarrow 1 \) to \( n \)
    do \( \)
        if \( \text{colour}[v_i] = \text{white} \)
            then \( \)
                \( \text{comp} \leftarrow \text{comp} + 1 \)
                \( DFSvisit(H, v_i, \text{comp}) \)

return \( (\text{comp}) \)

**Algorithm:** \( DFSvisit(H, v, \text{sec}) \)

\( \text{colour}[v] \leftarrow \text{gray} \)

\( \text{comp}[v] \leftarrow \text{sec} \)

for each \( w \in \text{Adj}[v] \)
    do \( \)
        if \( \text{colour}[w] = \text{white} \)
            then \( DFSvisit(H, w, \text{sec}) \)

\( \text{colour}[v] \leftarrow \text{black} \)

Complexity? \( O(n + m) \)
Proof of Correctness of Sharir’s Algorithm

First, note that $G$ and $H$ have the same strongly connected components. Let $u = v_{i_1}$ be the first vertex visited in step 3. Let $C$ be the s.c.c. containing $u$ and let $C'$ be any other s.c.c. $f(C') > f(C')$, so there is no edge from $C'$ to $C$ in $G$ (by the Lemma). Therefore there is no edge from $C$ to $C'$ in $H$. Hence no vertex in $C'$ is reachable from $u$ in $H$.

Therefore, $DFSvisit(u)$ explores the vertices in $C$ (and only those vertices); this forms one DFS tree in $H$.

Next, $DFSvisit(v_{i_2})$ explores the vertices in the s.c.c. containing $v_{i_2}$, etc. Every time we make an initial call to $DFSvisit$, we are exploring a new s.c.c.

We increment $scc$, which is used to label the various s.c.c.

$comp[v]$ denotes the label of the s.c.c. containing $v$. 

I did not go through this slide in class, and you do not need to know it. But if you are curious, here is a proof for the algorithm.
FREQUENTLY ASKED QUESTION

Could we simply do the DFSVisit calls for the second DFS in the original graph G, in order from smallest to largest finishing time?
DFSVisit(5) would reach two SCCs.

No!

Depends where first DFS starts…

If first DFS starts at a, then…