**CS 341: ALGORITHMS**

Lecture 14: Graph Algorithms II—Topsort, DAG testing, SCC

Readings: see website

Trevor Brown
https://student.cs.uwaterloo.ca/~cs341

trevor.brown@uwaterloo.ca

---

**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.**

$(\Rightarrow)$: Any back edge creates a directed cycle.

---

**Case ($\Rightarrow$):** Suppose there is a directed cycle. Show there is a 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.

Recall: nodes become gray when discovered.

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

Thus, $v_k$ 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

---

**TURNING THE LEMMA INTO AN ALGORITHM**

**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?

---

**DFS**: Testing Whether a Graph is a DAG

1. **Global variables**
   - pred: list[null, null, ..., null]
   - colour: list[white, white, ... , white]
   - discovery: list[0, 0, ..., 0] // discovery times
   - finish: list[0, 0, ..., 0] // finish times
   - time: int
   - adj: list[null, null, ..., null]

2. $\text{DFS}(v)$
   - if colour[v] = white:
     - pred[v] = v
     - $\text{DFS}$(adj[v])
     - colour[v] = grey
   - time = time + 1
   - $f(v) = \text{time}$

3. for each $w \in \text{adj}[v]$
   - if colour[w] = white:
     - pred[w] = v
     - $\text{DFS}(w)$
   - if colour[w] = grey:
     - $\text{DAG} = \text{false}$

4. $\text{return: } \text{DAG}$

5. $\text{DFS}(adj[v])$
   - if colour[v] = white:
     - pred[v] = v
     - $\text{DFS}$(adj[v])
   - if colour[v] = grey:
     - $\text{DAG} = \text{false}$

6. $\text{return: } \text{DAG}$

---
EXAMPLE

Back edge found! So we set DAG = false

Top edge found!

DEPENDENCY GRAPH

• Edge \((u, v)\) means \(u\) must be completed before \(v\)

Example problem: getting dressed in the morning

Pants before belt

Could do various things but which ones are possible? What do they have in common?

Topological sort

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!

More realistic use case: task scheduling

FORMAL DEFINITION

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 \(uv \in E\).
USEFUL FACT

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 \( v_i \) be an arc. In the sequence \( v_1, v_2, \ldots \), consider the first repeated vertex, \( v_i = v_j \) where \( j > i \). Then \( v_i, v_{i+1}, \ldots, v_j \) is a directed cycle.

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

EXISTENCE OF A TOPOLOGICAL SORT ORDER

Theorem 6.6
A directed graph \( D \) has a topological sort if and only if it is a DAG.

Proof.
(a) Suppose \( D \) has a directed cycle \( v_1, v_2, \ldots, v_j \). Then \( v_1 < v_2 < \cdots < v_j < v_1 \), so a topological ordering does not exist.
(b) Suppose \( D \) is a DAG. The algorithm below constructs a topological ordering:

Add \( v \) to the topological order

Nodes with indeg 0 have no unsatisfied dependencies

Compute indegree for all vertices

For each node \( v \) in adj \( (u) \)

if indeg[\( v \)] = 0 then q.enqueue(\( v \))

Remove \( v \)'s out edges. If we have now satisfied all dependencies for some \( w \), add \( w \) to the queue also.

Return order

EXEMPLARY (KAHN'S ALGORITHM)

Compute indeg for all vertices

For each node \( u \) in adj \( (v) \)

if indeg[\( u \)] = 0 then q.enqueue(\( u \))

NO SUCH ORDER!

Nodes with indeg 0 have no unsatisfied dependencies

‘\( Q \) contains no nodes whose dependencies are already satisfied

‘\( Q \) always contains nodes with no unsatisfied dependencies (indeg 0)

Running time with adjacency lists?

Running time with

\( \sum \deg(v) = \Theta(n + m) \)

\( \Theta(n) \) times

\( \Theta(1) \) per check

\( \Theta(n + m) \) total work over all nodes

\( \Theta(n + m) \) total work over all iterations

Total \( \Theta(n + m) \)

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
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 $f(u) < f(v)$.

But this contradicts the node ordering! So all edges are left-to-right,
therefore this is a topological sort.

Theorem: If $D$ is a DAG, and we order vertices in reverse order of finishing time
(i.e., by largest to smallest finish 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, this contradicts the node ordering.
So all edges are left-to-right, hence this is a topological sort.

The initial calls are $DFSVisit(1)$, $DFSVisit(2)$ and $DFSVisit(3)$.

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>9</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>3</td>
</tr>
</tbody>
</table>

The topological ordering is $3, 2, 5, 4, 1, 6$ (reverse order of finishing time).

We can see strongly connected components (SCCs) in this graph.

- This graph could be divided into two graphs that are each strongly connected.

These are called strongly connected components (SCCs).

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

Maximal, so SCC
Not maximal, so not SCC

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

- So, the goal is to find these (maximal) SCCs:

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

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

FORMALLY DEFINING SCC

For two vertices x and y of G, define x ~ y if x = y, or if x ≠ y and there exist directed paths from x to y and from y to x.

The relation ~ is an equivalence relation.

The strongly connected components of G are the equivalence classes of vertices defined by the relation ~.

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

COMPONENT GRAPH

Consider this graph

These are its SCCs

Can there be a cycle in the component graph?

Not if there are paths both ways between components—they are actually the same SCC.

Component graph is a DAG!

The following is its component graph

It has one node for each SCC.

And an edge between two nodes IF there is an edge between the corresponding SCCs.

BRAINSTORMING AN ALGORITHM

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

The will definitely visit every node in a’s SCC.

And in fact it might visit other SCCs as well…

Showing discovery times

Showing finish times

Showing discovery times
**SCC ALGORITHM**

This is called Shao's algorithm (sometimes Kosaraju's algorithm).

<table>
<thead>
<tr>
<th>SCC Algorithm</th>
<th>Running Shao's Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: SCC(adj[1..n][1..n])</td>
<td>Phase 1: DFS to get finish times</td>
</tr>
<tr>
<td>2: DFS(adj)</td>
<td>Phase 2: DFSVisit reverse graph by reverse finish times</td>
</tr>
<tr>
<td>3: let order[1..n] = node labels sorted by largest to smallest finish time</td>
<td>order[1..n]</td>
</tr>
<tr>
<td>4: reverse all edges in adj</td>
<td>reverse all edges in adj</td>
</tr>
<tr>
<td>5: colour[1..n] = {white, white}</td>
<td>colour[1..n] = {white, white}</td>
</tr>
<tr>
<td>6: comp[1..n] = {0, ..., 0}</td>
<td>comp[1..n] = {0, ..., 0}</td>
</tr>
<tr>
<td>7: for i = 1 to n</td>
<td>for each v in adj[v]</td>
</tr>
<tr>
<td>8: if colour[v] = white</td>
<td>if colour[v] = white</td>
</tr>
<tr>
<td>9: acc = acc + 1</td>
<td>accVisit(adj), v, acc, colour, comp</td>
</tr>
<tr>
<td>10: colour[v] = grey</td>
<td>colour[v] = grey</td>
</tr>
<tr>
<td>11:</td>
<td>comp[v] = acc</td>
</tr>
<tr>
<td>12: accVisit(adj), v, acc, colour, comp</td>
<td>for each v in adj[v]</td>
</tr>
<tr>
<td>13:</td>
<td>if colour[v] = white</td>
</tr>
<tr>
<td>14:</td>
<td>accVisit(adj), v, acc, colour, comp</td>
</tr>
<tr>
<td>15:</td>
<td>colour[v] = black</td>
</tr>
<tr>
<td>16: return comp</td>
<td>acc = acc</td>
</tr>
</tbody>
</table>

**TIME COMPLEXITY?**

Can be returned as part of the DFS with no added runtime.

- Each edge is inspected once, each visited node is visited once, constant work per visited node/inspected edge.
- Total of \(O(n + m)\) work over all n iterations of the \(i\) loop (each edge is inspected once, each node is visited once, constant work per visited node/inspected edge).

**CORRECTNESS**

- Want to prove that each top-level call to SCCVisit explores exactly the nodes in one SCC.
- Proof hinges on a key lemma that talks about the finish times of SCCs in the component graph.
- To talk about finish times of SCCs, we need a definition...
A KEY DEFINITION

- For a strongly connected component \( C \), let \( d(C) = \min\{d(u) \mid u \in C\} \) and \( f(C) = \max\{f(u) \mid u \in C\} \).

\[
d(C) = 1 \quad d(C) = 5 \quad d(C) = 15 \quad d(C) = 19
\]

\[
f(C) = 24 \quad f(C) = 18 \quad f(C) = 10 \quad f(C) = 14
\]

\[
d(C) = 19 \quad d(C) = 20 \quad d(C) = 19 \quad d(C) = 19
\]

\[
f(C) = 18 \quad f(C) = 18 \quad f(C) = 10 \quad f(C) = 14
\]

A KEY LEMMA

- **Lemma**: If \( C_i, C_j \) are SCCs and there is an edge \( C_i \to C_j \) in \( G \), 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 \( C_j \cup C_i \) are white-reachable from \( u \), so they are descendants in the DFS forest and finish before \( u \).
  - So \( f(C_i) = f(u) > f(C_j) \).

A KEY LEMMA

- **Lemma**: If \( C_i, C_j \) are SCCs and there is an edge \( C_i \to C_j \) in \( G \), 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 \( C_j \to C_i \).
  - 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

COMPLETING THE PROOF

- We know \( f(C) > f(C') \).
- By Lemma: if there were an edge \( C' \to C \) in \( G \), then we would have \( f(C') > f(C) \).
- So there is no edge \( C' \to C \) in \( G \).
- And hence no edge \( C \to C' \) in \( H \).
- So, SCCVisit(\( u \)) in \( H \) cannot visit \( C' \).

**component graph \( C_i \) of \( C \)**

**component graph \( C_j \) of \( H \)**

**component graph \( C_j \) of \( H \)**

**component graph \( C_j \) of \( H \)**

[Diagram showing component graph relationships and discovery times]