CS 341: ALGORITHMS

Lecture 12: graph algorithms III – DAG testing, topsort, 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.

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

- Let $v_1, v_2, ..., v_k, v_1$ be a directed cycle
- WLOG let $v_1$ be earliest discovered node in the cycle

Consider edge \{${v_k, v_1}$\}

Recall: nodes become gray when discovered

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, $v_2, ..., v_k$ are all white

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

$\Rightarrow$ 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
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?**

<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>

When we observe an edge from $u$ to $v$, check if $v$ is gray
DFS: TESTING WHETHER A GRAPH IS A DAG

**Global variables:**
- `pred[1..n] = [null, null, ..., null]`
- `colour[1..n] = [white, white, ..., white]`
- `d[1..n] = [0, 0, ..., 0] // discovery times`
- `f[1..n] = [0, 0, ..., 0] // finish times`
- `time = 0`
- `DAG = true`

**IsDAG(adj[1..n])**

```plaintext
for v = 1..n
    if colour[v] == white
        DFSVisit(adj, v)
return DAG
```

**DFSVisit(adj[1..n], v)**

```plaintext
colour[v] = gray
time = time + 1
d[v] = time
for each w in adj[v]
    if colour[w] == white
        pred[w] = v
        DFSVisit(w)
if color[w] == gray
    DAG = false
colour[v] = black
time = time + 1
f[v] = time
```
Back edge found! So we set DAG = false
TOPOLOGICAL SORT
Finding node orderings that satisfy given constraints
Edge \(\{u, v\}\) means \(u\) must be completed **before** \(v\).

Example problem: getting dressed in the morning

- Pants before belt
- Socks before shoes
- Could do various things first. Which ones are possible? What do they have in common?

Watch any time
Topological sort

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

Possible IFF graph is a DAG
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 $uv \in E$.

Graph $G$

Topological sort of $G$

$v_3 < v_2 < v_5 < v_4 < v_1 < v_6$

Edges are directed only **left-to-right** in this ordering
**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 $v_{i+1}v_i$ 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!
TOPOLOGICAL SORT VIA DFS

- We can 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 $uv$.

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, (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.

By our node ordering, $f_v > f_u$

But the lemma says for every edge $\{u, v\}$, we must have $f_v < f_u$

**Lemma 6.8**

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

**Contradiction!** Right-to-left edge cannot exist. So is is a topological ordering.
TOPOLOGICAL ORDERING VIA DFS

\[ O(n + m) \text{ w/adj. lists} \]

1. global variables:
   - pred[1..n] = [null, null, ..., null]
   - colour[1..n] = [white, white, ..., white]
   - d[1..n] = [0, 0, ..., 0] // discovery times
   - f[1..n] = [0, 0, ..., 0] // finish times
2. time = 0
3. DAG = true
4. TopologicalSort(adj[1..n])
5. \[ S = \text{new stack} \]
6. for \( v = 1 \ldots n \)
7.   if colour[v] == white
8.     DFSVisit(adj, v, S)
9. if DAG then return S
10. return null

DFSVisit(adj[1..n], v, S)
11. colour[v] = gray
12. time = time + 1
13. d[v] = time
14. for each \( w \) in adj[v]
15.   if colour[w] == white
16.     pred[w] = v
17.     DFSVisit(w)
18. if colour[w] == gray
19.   DAG = false
20. colour[v] = black
21. time = time + 1
22. f[v] = time

Push smallest finishing time first → pop largest first

Save each node when it finishes
HOME EXERCISE: RUN ON THIS GRAPH

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>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>10</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
This graph could be divided into two graphs that are each strongly connected.

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)
So, the goal is to find these (maximal) SCCs:
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
- maps; nodes = intersections, edges = roads
- Don’t want to run path finding algorithm on the entire **global** graph!
- Throw away everything except the (maximal) SCC containing source & target

Crop & find SCCs
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!
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...

DFSVisit(a) DFSVisit(h) DFSVisit(j)

Showing discovery times

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

Reverse edges

DFSVisit(a)  DFSVisit(h)  DFSVisit(j)

a,14  b,12  c,11

DFSVisit(a)  DFSVisit(e)  DFSVisit(h)

d,13  f,9  g,8

e,10

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

Problem: from h, we can reach other SCCs

Other reachable SCCs should be visited first

Then, each DFSVisit will visit exactly one SCC

(So we don't visit them again)

What if we perform DFSVisit calls in a different order in the reverse graph?
Consider component graph $C_g$ of $G$ (which we want to compute)

If we call DFSVisit in $G$ from largest to smallest finish times, we can reach **other SCCs**.

**Recall lemma:** edge $uv$ in DAG implies $f(u)>f(v)$

However, when we reverse the edges to get graph $H$ other SCCs can no longer be reached...
This is called Sharir's algorithm (sometimes Kosaraju's algorithm).

This paper first introduced it.
Running Sharir’s Algorithm

Phase 1: DFS to get finish times

Phase 2: DFSVisit reverse graph by reverse finish times

DFSVisit(j)  DFSVisit(h)  DFSVisit(a)  DFSVisit(e)

scc is shown

scc = 4
**TIME COMPLEXITY?**

```
1 SCC(adj[1..n])
2     DFS(adj)
3     let order[1..n] = node labels sorted by largest to smallest finish time
4     reverse all edges in adj
5     colour[1..n] = [white, ..., white]
6     comp[1..n] = [0, ..., 0]
7     for i = 1..n
8         v = order[i]
9         if colour[v] == white
10            scc = scc + 1
11            SCCVisit(adj, v, scc, colour, comp)
12 return comp

SCCVisit(adj[1..n], v, scc, colour, comp)
13     colour[v] = gray
14     comp[v] = scc
15     for each w in adj[v]
16         if colour[w] == white
17             SCCVisit(w)
18     colour[v] = black
```

- **$O(n + m)$** can be returned as part of the DFS with no added runtime.
- Finish times **increase** as we set them, so just use a stack...
- Total of **$O(n + m)$** work over all $n$ iterations of the $i$ loop.

**Total $O(n + m)$**

(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 $d[C] = \min\{d[v] : v \in C\}$ and $f[C] = \max\{f[v] : v \in C\}$.
Lemma: if $C_i, C_j$ are SCCs and there is an edge $C_i \rightarrow C_j$ 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 $C_i \cup C_j$ are white-reachable from $u$, so they are descendants in the DFS forest and finish before $u$
- So $f[C_i] = f[u] > f[C_j]$
A KEY LEMMA

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

- **Proof. Case 2 ($d[C_j] < d[C_i]$):**
  - 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.
COMPLETING THE PROOF

- Suppose we have performed DFS to get our finish times, and we are about to perform SCC Visits on the reverse graph.
- **We prove each top-level SCC Visit call visits precisely one SCC.**
- Consider the first top-level SCC Visit $\text{SCCVisit}(u)$.
- Let $C$ be the SCC containing $u$ and $C'$ be any other SCC.
- Since we call SCC Visit on nodes starting from the largest finish time,
  - We know $f(C) > f(C')$.
COMPLETING THE PROOF

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

... and sets comp[v] = scc for all nodes in the SCC

So each top-level call explores one SCC... and larger finish time means already explored!

In $G$, edges go from larger to smaller finish times. In $H$, edges go from smaller to larger.

Similar argument for subsequent top-level calls to SCCVisit.

So SCCVisit($u$) visits exactly the nodes in $C$
IF WE HAVE TIME

topological sort without relying on DFS
**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.**

($\Rightarrow$): 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.
Kahn(adj[1..n])

1. indeg[1..n] = [0, ..., 0]
2. for each edge (u,v) in adj
   indeg[v] = indeg[v] + 1
3. order = new list
4. q = new queue containing {v : indeg[v] == 0}
5. for i = 1..n
6.   if q.empty() return null
7.   v = q.dequeue()
8.   order.append(v)
9.   for each w in adj[v]
   10.      indeg[w] = indeg[w] - 1
   11.      if indeg[w] == 0 then q.enqueue(w)
12. return order

\[\text{indeg}[v] = \text{# of edges pointing into node } v\]

\[\text{indeg } v = \text{number of unsatisfied constraints on } v\]

Nodes with \text{indeg } 0 have no unsatisfied dependencies.

So this step is enqueuing nodes whose dependencies are already satisfied.

\(q\) always contains nodes with no unsatisfied dependencies (indeg 0).

No such order!

Add \(v\) to the topological order.

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

Compute **indegree** for all vertices

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

vertices with indeg 0 go into the queue

Until Q is empty: pop, output that element, decrement its neighbours, enqueue new indeg 0’s

Queue Q

```
3 2 5 4 1 6
```

Output
Kahn(adj[1..n])

indeg[1..n] = [0, ..., 0]
for each edge (u,v) in adj
    indeg[v] = indeg[v] + 1

order = new list
q = new queue containing {v : indeg[v] == 0}
for i = 1..n
    if q.empty() return null
    v = q.dequeue()
    order.append(v)
    for each w in adj[v]
        indeg[w] = indeg[w] - 1
        if indeg[w] == 0 then q.enqueue(w)

return order

Running time with adjacency lists?

\( O(n) \)

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

\( O(n) \) iterations

\( O(1) \) per check

\( \sum_{v \in V} \deg(v) \in O(n + m) \)

Total work over all nodes \( v \)

\( O(n + m) \)
SCC: HOW ABOUT A DIFFERENT ORDERING?

- Rather than doing DFS in the reverse graph in order of decreasing finish times
- Why not do DFS in the original graph in order of increasing finish times?
- Exercise: does this work?
SCC: HOW ABOUT A DIFFERENT ORDERING?

- Why not do DFS in the original graph in order of increasing finish times?

Doesn’t work!
Output depends where first DFS starts…
If first DFS starts at c, then…

DFSVisit(b) would reach two SCCs.