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.

Case (=): Suppose ∃ directed cycle. Show ∃ back edge.

1. Let v₁, v₂, ..., vᵦ, v₁ be a directed cycle
2. WLOG let v₁ be earliest discovered node in the cycle
3. Let v₁, v₂, ..., vᵦ, v₁ be a directed cycle
4. Since d(v₁) < d(vᵦ), (vᵦ, v₁) must be a back or cross edge
5. Why?

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?

When we observe an edge from u to v, check if v is gray

Tree edge:

tree:
back:
cross:

When we observe an edge from u to v, check if v is gray

edge type
colour of v
discovery/finish times
tree
black
d[v] < d[u] < f[u] < f[v]
back
gray
d[v] < f[u] < d[u] < f[v]
cross
black
d[v] < f[u] < d[u] < f[v]
**EXAMPLE**

Back edge found! So we set DAG = false.

**TOPOLOGICAL SORT**

Finding node orderings that satisfy given constraints.

**DEPENDENCY GRAPH**

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

Example problem: getting dressed in the morning.

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

Possible IFF graph is a DAG.

**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 \((u, v) \in E\).

**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 \(j \geq 1\), let \(v_j + 1\) 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 \(v_i, v_{i+1}, \ldots, v_j\) is a directed 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.

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 $f_u > f_v > \ldots > f_{n-1} > f_n$. But the lemma says for every edge $(u, v)$, we must have $f_u < f_v$. Contradiction! Right-to-left edge cannot exist. So is is a topological ordering.

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>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

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

HOME EXERCISE: RUN ON THIS GRAPH

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>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

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

These are called strongly connected components (SCCs).

---

STRONGLY CONNECTED COMPONENTS

- But we want our SCCs to be **maximal** (as large as possible).

---

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
BRAINSTORMING AN ALGORITHM

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

...other SCCs can no longer be reached...

What if we run DFS, then reverse all edges, then run DFS?

...is shown

To identify SCC

This is called Shapir’s algorithm (sometimes Kosaraju’s algorithm).

This paper first introduced.

Consider component graph C of G which we want to compute.

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

However, when we reverse the edges to get graph G reversed other SCCs can no longer be reached...

SCC ALGORITHM

Running Shapir’s Algorithm

Phase 1: DFS to get finish times

Phase 2: DFSVisit reverse graph by reverse finish times

TIME COMPLEXITY?

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

Finish times increase as we set them, so just use a stack...

Each edge is inspected once, runtime complexity is O(n+m).
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 LEMMA
- Lemma: if 𝒖, 𝒗 are SCCs and there is an edge 𝒖 → 𝒗 in 𝐺, then 𝑓(𝒖) > 𝑓(𝒗)
- Proof. Case 1 (𝒅(𝒖) < 𝒃(𝒗)): let 𝑑(𝒖) = min{𝒅(𝒗) : 𝒖 ∈ 𝐶}
- Case 2 (𝒅(𝒖) ≤ 𝒃(𝒗)): let 𝑑(𝒖) = max{𝒅(𝒗) : 𝒖 ∈ 𝐶}

A KEY DEFINITION
- For a strongly connected component 𝐶, let 𝑑(𝐶) = min{𝒅(𝒗) : 𝒖 ∈ 𝐶} and 𝒇(𝐶) = max{𝑓(𝒗) : 𝒖 ∈ 𝐶}
- So, 𝐶 is a SCC if 𝑑(𝐶) ≤ 𝒃(𝐶)

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

A KEY DEFINITION
- For a strongly connected component 𝐶, let 𝑑(𝐶) = min{𝒅(𝒗) : 𝒖 ∈ 𝐶} and 𝒇(𝐶) = max{𝑓(𝒗) : 𝒖 ∈ 𝐶}

COMPLETING THE PROOF
- We know 𝑓(𝒖) > 𝑓(𝒑)
- By Lemma: if there were an edge 𝒑 → 𝒖 in 𝐺, then we would have 𝑓(𝒑) > 𝑓(𝒖)
- So there is no edge 𝒑 → 𝒖 in 𝐺
- and hence no edge 𝒑 → 𝒖 in 𝐻
- So, SCCVisit(𝒖) in 𝐻 cannot visit 𝒑

A KEY LEMMA
- Lemma: if 𝒖, 𝒗 are SCCs and there is an edge 𝒖 → 𝒗 in 𝐺, then 𝑓(𝒖) > 𝑓(𝒗)
- Proof. Case 1 (𝒅(𝒖) < 𝒃(𝒗)): let 𝑑(𝒖) = min{𝒅(𝒗) : 𝒖 ∈ 𝐶}
- Case 2 (𝒅(𝒖) ≤ 𝒃(𝒗)): let 𝑑(𝒖) = max{𝒅(𝒗) : 𝒖 ∈ 𝐶}

A KEY DEFINITION
- For a strongly connected component 𝐶, let 𝑑(𝐶) = min{𝒅(𝒗) : 𝒖 ∈ 𝐶} and 𝒇(𝐶) = max{𝑓(𝒗) : 𝒖 ∈ 𝐶}
- So, 𝐶 is a SCC if 𝑑(𝐶) ≤ 𝒃(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:*

1. Suppose $D$ has a directed cycle $(v_1, v_2, \ldots, v_k)$. Then $v_1 < v_2 < \cdots < v_k < v_1$, so a topological ordering does not exist.
2. Suppose $D$ is a DAG. Then the algorithm below constructs a topological ordering.

---

**Kahn's Algorithm**

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

---

**Running Time with adjacency lists**

- $O(n)$
- $O(n + m)$ total work over all iterations
- $O(1)$ per check
- $O(deg(v))$ per iteration
- Total work over all nodes $v$

---

**BONUS SLIDES**

---

**EXAMPLE (KAHN'S ALGORITHM)**

1. Compute $\text{indegree}$ for all vertices
2. For each node $v$ in adj $w \in \text{adj}(v)$
3. $\text{deg}(v) = \text{deg}(v) + 1$
4. for each $w$ in adj $v$
5.  indeg[w] = indeg[w] - 1
6. if indeg[w] == 0 then q.enqueue(w)
7. order = new list
8. q = new queue containing $(v : indeg[v] == 0)$
9. for i = 1..n
10. if q empty() return null
11. v = q.dequeue()
12. order.append(v)
13. for each w in adj[v]
14.  indeg[w] = indeg[w] - 1
15. if indeg[w] == 0 then q.enqueue(w)
16. return order
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?

![Diagram showing DFS visit order and SCCs]

Doesn't work!

Output depends where first DFS starts...

DFSVisit(b) would reach two SCCs.