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.

– DAG testing, topsort, SCC

• Case (⇐): Suppose ∃ directed cycle.
  Show ∃ back edge.
  • Let v₁, v₂, ..., vₖ, v₁ be a directed cycle
  • WLOG let v₁ be earliest discovered node in the cycle
  • Recall: nodes become gray when discovered
    • So when v₁ is discovered, v₂, ..., vₖ are all white
  • Recall: every node vᵢ that is white-reachable
    from v₁ when we discover v₁ (call DFSVisit(v₁)) turns black before v₁
  • Thus, (vₖ, v₁) must be a back edge. QED

TURNING THE LEMMA INTO AN ALGORITHM

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

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

DFS: TESTING WHETHER A GRAPH IS A DAG

edge type | discovery/dismiss times
--- | ---
tree | time [v] = grey
tree | time = time + 1
cross | d[v] = time
cross | time = time + 1
cross | if colour[v] = grey
     | if colour[v] = white
     | pred[v] = v
     | DFSVisit(adj[v])
back | if colour[v] = grey
     | if colour[v] = white
     | pred[v] = v
     | DFSVisit(adj[v])
TOPOLOGICAL SORT
Finding node orderings that satisfy given constraints

DEPENDENCY GRAPH
- Edge \((u, v)\) means \(u\) must be completed before \(v\)

EXAMPLE
Back edge found! So we set DAG = false

DEPENDENCY GRAPH
- Edge \((u, v)\) means \(u\) must be completed before \(v\)

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

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 \(u, v \in E\).
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

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

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

By our node ordering, $f_u < f_v$. But the lemma says for every edge $(u,v)$, we must have $f_v < f_u$.

Contradiction: Right-to-left edge cannot exist. So is a topological ordering.

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!

TOPOLOGICAL ORDERING VIA DFS

Push smallest finishing time list to pop largest first

Home exercise: Run on this graph

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

The discovery/finish times are as follows:

\[
\begin{array}{c|c|c|c|c|c|c}
  \hline
  1 & 1 & 5 & 2 & 3 & 5 \\
  2 & 4 & 7 & 3 & 6 & 9 \\
  3 & 11 & 12 & 4 & 8 & 3 \\
  4 & & & & & \\
  5 & & & & & \\
\end{array}
\]

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).

It could also be divided into three graphs...

Maximal SCC Not maximal

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

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

Finding all cyclic dependencies in code

Can find single cycle with an easier DFS-based algorithm

But it’s 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

APPLICATIONS OF SCCS AND COMPONENT GRAPHS

• Component graph is a DAG!
  • And an edge between two nodes iff there is an edge between the corresponding 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

APPLICATIONS OF SCCS AND COMPONENT GRAPHS

• Consider this graph
  • These are its SCCs
  • The following is its component graph
  • It has one node for each SCC

COMPONENT GRAPH

• Crop & find SCCs
  • e.g. h,l
  • a,b,c,d

• Contain all SCC
  • e.g. b,c,d,h
This is called in DAG $i$ reverse graph $m$ other SCCs as well.

What if we run DFS, then reverse all edges, then run DFS from largest to smallest $DFSVist$ (which we want to compute) $\mathcal{O}(n)$.

Consider component graph $c$, of $c$, (which we want to compute) $DFSVist$.

If we call $DFSVist$ in $c$, from largest to smallest finish times, we can reach other SCCs.

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

Recall lemma: edge $uv$ in $G$ implies $(f(u),f(v))$.

This paper first introduced it.
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

- 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 \( C_i \) are white reachable from \( u \), so they are descendants in the DFS forest and finish before \( u \).
- So \( f(C_i) = f(u) > f(C_j) \).

- 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 \( |d(C_i)| < |d(C_j)| \):
- 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 \( d(C_i) < f(C_i) < d(C_j) < f(C_j) \). QED.

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 \( C' \) be any other SCC.
- Since we call SCCVisit on nodes starting from the largest finish time,
  - We know \( f(C) > f(C') \).

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

- 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 \( |d(C_i)| < |d(C_j)| \):
- 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 \( d(C_i) < f(C_i) < d(C_j) < f(C_j) \). QED.

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\((u)\) = SCC for all nodes in the SCC.
- So each top-level call explores one SCC.
- And larger finish time means already explored.
- If \( G \) 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 \( C \).

COMPETING THE PROOF
**EXISTENCE OF A TOPOLOGICAL SORT ORDER**

**Theorem 4.6**
A directed graph \( D \) has a topological sort if and only if it is a DAG.

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

\( (\Leftarrow) \): Suppose \( D \) is a DAG. Then the algorithm below constructs a topological ordering:

**Algorithm:**
1. Compute indegree for all vertices.
2. For each vertex \( u \) in \( D \), add \( u \) to \( Q \).
3. While \( Q \) is non-empty, output an element from \( Q \), decrement its neighbours, and add any new indeg 0’s to \( Q \).
4. Output the resulting order.

**Running time with adjacency lists**

\[
O(n + m)
\]

\[
O(n + m) \text{ total work over all iterations}
\]

\[
O(1) \text{ per check}
\]

\[
O(deg(v)) \text{ per iteration}
\]

\[
O(deg(v)) \text{ total work over all nodes}
\]

**BONUS SLIDES**
**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?

Output depends where first DFS starts...

- If first DFS starts at c, then...

DFSVisit(b) would reach two SCCs.