More on DFS in Directed Graphs, and Strong Connected Components, and DAGs

Lecture 3

January 27, 2015

Using DFS...

... to check for Acylicity and compute Topological Ordering

Question

Given G, is it a DAG? If it is, generate a topological sort.

DFS based algorithm:

1. Compute **DFS**(G)
2. If there is a back edge then G is not a DAG.
3. Otherwise output nodes in decreasing post-visit order.

Correctness relies on the following:

Proposition

G is a DAG iff there is no back-edge in **DFS**(G).

Proposition

If G is a DAG and $\mathrm{post}(v) > \mathrm{post}(u)$, then $(u \to v)$ is not in G.

Using DFS...

... to check for Acylicity and compute Topological Ordering

Question

Given G, is it a DAG? If it is, generate a topological sort.

DFS based algorithm:

1. Compute **DFS**(G)
2. If there is a back edge then G is not a DAG.
3. Otherwise output nodes in decreasing post-visit order.

Correctness relies on the following:

Proposition

G is a DAG iff there is no back-edge in **DFS**(G).

Proposition

If G is a DAG and $\mathrm{post}(v) > \mathrm{post}(u)$, then $(u \to v)$ is not in G.

Proof

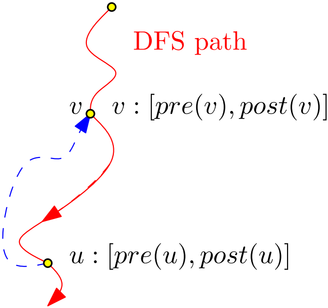
Proposition

If G is a DAG *and* $\mathrm{post}(u) < \mathrm{post}(v)$, *then* (u, v) *is not in* G.

Proof

Assume $\mathrm{post}(v) > \mathrm{post}(u)$ *and* (u, v) is an edge in G. We derive a contradiction.

1. Case 1: $[pre(u), post(u)]$ is contained in $[pre(v), post(v)]$.
2. $\implies$ u explored during $\mathbf{DFS}(v)$.
3. u descendant of v.
4. $(u, v) \in E(G)$ $\implies$ cycle in G but G is a DAG.



Proof

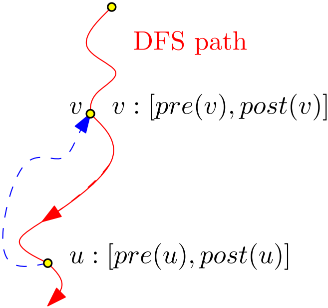
Proposition

If G is a DAG and $\mathrm{post}(u) < \mathrm{post}(v)$, then (u, v) is not in G.

Proof

Assume $\mathrm{post}(v) > \mathrm{post}(u)$ *and* (u, v) is an edge in G. We derive a contradiction.

1. Case 1: $[pre(u), post(u)]$ is contained in $[pre(v), post(v)]$.
2. $\implies$ u explored during **DFS**(v).
3. u descendant of v.
4. $(u, v) \in E(G) \implies$ cycle in G but G is a DAG.



Proof

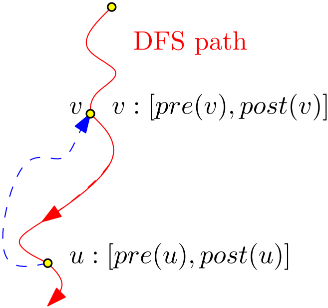
Proposition

If G is a DAG and $\mathrm{post}(u) < \mathrm{post}(v)$, then (u, v) is not in G.

Proof

Assume $\mathrm{post}(v) > \mathrm{post}(u)$ *and* (u, v) is an edge in G. We derive a contradiction.

1. Case 1: $[pre(u), post(u)]$ is contained in $[pre(v), post(v)]$.
2. $\implies$ u explored during **DFS**(v).
3. u descendant of v.
4. $(u, v) \in E(G) \implies$ cycle in G but G is a DAG.



Proof

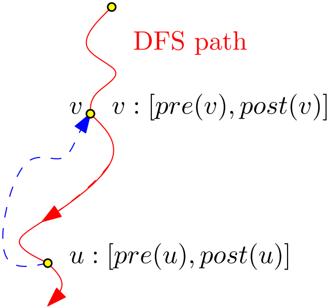
Proposition

If G is a DAG and $\mathrm{post}(u) < \mathrm{post}(v)$, then (u, v) is not in G.

Proof

Assume $\mathrm{post}(v) > \mathrm{post}(u)$ *and* (u, v) is an edge in G. We derive a contradiction.

1. Case 1: $[pre(u), post(u)]$ is contained in $[pre(v), post(v)]$.
2. $\implies$ u explored during **DFS**(v).
3. u descendant of v.
4. $(u, v) \in E(G) \implies$ cycle in G but G is a DAG.



Proof continued

Proposition

If G is a DAG *and* $\mathbf{post}(u) < \mathbf{post}(v)$*, then* (u, v) *is not in* G*.*

Proof continued...

Case 2: $[pre(u), post(u)]$ is disjoint from $[pre(v), post(v)]$.

1. By assumption: $\mathrm{post}(u) < \mathrm{post}(v)$.
2. $\implies \mathrm{pre}(u) < \mathrm{pre}(v)$
3. **DFS** visits u first and then v.
4. If $(u \rightarrow v) \in E(G)$...
5. $\implies$ **DFS** explores v during the **DFS** of u.
6. $[pre(v), post(v)] \subseteq [pre(u), post(u)]$.
7. $\implies$ contradiction.

Proof continued

Proposition

If G is a DAG *and* $\mathbf{post}(u) < \mathbf{post}(v)$, *then* (u, v) *is not in* G.

Proof continued...

Case 2: $[pre(u), post(u)]$ is disjoint from $[pre(v), post(v)]$.

1. By assumption: $\mathbf{post}(u) < \mathbf{post}(v)$.
2. $\implies \mathbf{pre}(u) < \mathbf{pre}(v)$
3. **DFS** visits u first and then v.
4. If $(u \rightarrow v) \in$ E(G)...
5. $\implies$ **DFS** explores v during the **DFS** of u.
6. $[pre(v), post(v)] \subseteq [pre(u), post(u)]$.
7. $\implies$ contradiction.

Proof continued

Proposition

If G is a DAG and $\mathrm{post}(u) < \mathrm{post}(v)$, then (u, v) is not in G.

Proof continued...

Case 2: $[pre(u), post(u)]$ is disjoint from $[pre(v), post(v)]$.

1. By assumption: $\mathrm{post}(u) < \mathrm{post}(v)$.
2. $\implies \mathrm{pre}(u) < \mathrm{pre}(v)$
3. DFS visits u first and then v.
4. If $(u \to v) \in$ E(G)...
5. $\implies$ DFS explores v during the DFS of u.
6. $[pre(v), post(v)] \subseteq [pre(u), post(u)]$.
7. $\implies$ contradiction.

Proof continued

Proposition

If G is a DAG and $\mathrm{post}(u) < \mathrm{post}(v)$, then (u, v) is not in G.

Proof continued...

Case 2: $[pre(u), post(u)]$ is disjoint from $[pre(v), post(v)]$.

1. By assumption: $\mathrm{post}(u) < \mathrm{post}(v)$.
2. $\implies \mathrm{pre}(u) < \mathrm{pre}(v)$
3. **DFS** visits u first and then v.
4. If $(u \to v) \in E(G)$...
5. $\implies$ **DFS** explores v during the **DFS** of u.
6. $[pre(v), post(v)] \subseteq [pre(u), post(u)]$.
7. $\implies$ contradiction.

Proof continued

Proposition

If G is a DAG and $\mathrm{post}(u) < \mathrm{post}(v)$, then (u, v) is not in G.

Proof continued...

Case 2: $[pre(u), post(u)]$ is disjoint from $[pre(v), post(v)]$.

1. By assumption: $\mathrm{post}(u) < \mathrm{post}(v)$.
2. $\implies \mathrm{pre}(u) < \mathrm{pre}(v)$
3. **DFS** visits u first and then v.
4. If $(u \to v) \in E(G)$...
5. $\implies$ **DFS** explores v during the **DFS** of u.
6. $[pre(v), post(v)] \subseteq [pre(u), post(u)]$.
7. $\implies$ contradiction.

Proof continued

Proposition

If G is a DAG *and* $\mathrm{post}(u) < \mathrm{post}(v)$*, then* (u, v) *is not in* G*.*

Proof continued...

Case 2: $[pre(u), post(u)]$ is disjoint from $[pre(v), post(v)]$.

1. By assumption: $\mathrm{post}(u) < \mathrm{post}(v)$.
2. $\implies \mathrm{pre}(u) < \mathrm{pre}(v)$
3. **DFS** visits u first and then v.
4. If $(u \to v) \in E(G)$...
5. $\implies$ **DFS** explores v during the **DFS** of u.
6. $[pre(v), post(v)] \subseteq [pre(u), post(u)]$.
7. $\implies$ contradiction.

Proof continued

Proposition

If G is a DAG and $\mathrm{post}(u) < \mathrm{post}(v)$, then (u, v) is not in G.

Proof continued...

Case 2: $[pre(u), post(u)]$ is disjoint from $[pre(v), post(v)]$.

1. By assumption: $\mathrm{post}(u) < \mathrm{post}(v)$.
2. $\implies \mathrm{pre}(u) < \mathrm{pre}(v)$
3. **DFS** visits u first and then v.
4. If $(u \rightarrow v) \in \mathrm{E(G)}$...
5. $\implies$ **DFS** explores v during the **DFS** of u.
6. $[pre(v), post(v)] \subseteq [pre(u), post(u)]$.
7. $\implies$ contradiction.

Proof continued

Proposition

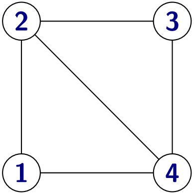
If G is a DAG and $\mathrm{post}(u) < \mathrm{post}(v)$, then (u, v) is not in G.

Proof continued...

Case 2: $[pre(u), post(u)]$ is disjoint from $[pre(v), post(v)]$.

1. By assumption: $\mathrm{post}(u) < \mathrm{post}(v)$.
2. $\implies \mathrm{pre}(u) < \mathrm{pre}(v)$
3. **DFS** visits u first and then v.
4. If $(u \to v) \in E(G)$...
5. $\implies$ **DFS** explores v during the **DFS** of u.
6. $[pre(v), post(v)] \subseteq [pre(u), post(u)]$.
7. $\implies$ contradiction.

Example



Back edge and Cycles

Proposition

*G has a cycle iff there is a back-edge in **DFS**(G).*

Proof.

1. If: (u, v) is a back edge $\implies$ there is a cycle C in G: $C =$ path from v to u in **DFS** tree + edge $(u \to v)$.
2. Only if: Suppose there is a cycle $C = v_1 \to v_2 \to \ldots \to v_k \to v_1$.
   1. Let v_i be first node in C visited in **DFS**.
   2. All other nodes in C are descendants of v_i since they are reachable from v_i.
   3. Therefore, (v_{i-1}, v_i) (or (v_k, v_1) if $i = 1$) is a back edge.

Back edge and Cycles

Proposition

G has a cycle iff there is a back-edge in **DFS**(G).

Proof.

1. If: (u, v) is a back edge $\implies$ there is a cycle C in G:
   $C =$ path from v to u in **DFS** tree + edge $(u \to v)$.
2. Only if: Suppose there is a cycle
   $C = v_1 \to v_2 \to \ldots \to v_k \to v_1$.
   1. Let v_i be first node in C visited in **DFS**.
   2. All other nodes in C are descendants of v_i since they are reachable from v_i.
   3. Therefore, (v_{i-1}, v_i) (or (v_k, v_1) if $i = 1$) is a back edge.

□

Back edge and Cycles

Proposition

G has a cycle iff there is a back-edge in **DFS**(G).

Proof.

1. If: (u, v) is a back edge $\implies$ there is a cycle C in G: $C =$ path from v to u in **DFS** tree + edge $(u \to v)$.
2. Only if: Suppose there is a cycle $C = v_1 \to v_2 \to \ldots \to v_k \to v_1$.
   1. Let v_i be first node in C visited in **DFS**.
   2. All other nodes in C are descendants of v_i since they are reachable from v_i.
   3. Therefore, (v_{i-1}, v_i) (or (v_k, v_1) if $i = 1$) is a back edge.

Back edge and Cycles

Proposition

G has a cycle iff there is a back-edge in **DFS**(G).

Proof.

1. If: (u, v) is a back edge $\implies$ there is a cycle C in G: $C =$ path from v to u in **DFS** tree $+$ edge $(u \to v)$.
2. Only if: Suppose there is a cycle $C = v_1 \to v_2 \to \ldots \to v_k \to v_1$.
   1. Let v_i be first node in C visited in **DFS**.
   2. All other nodes in C are descendants of v_i since they are reachable from v_i.
   3. Therefore, (v_{i-1}, v_i) (or (v_k, v_1) if $i = 1$) is a back edge.

Back edge and Cycles

Proposition

*G has a cycle iff there is a back-edge in **DFS**(G).*

Proof.

1. If: (u, v) is a back edge $\implies$ there is a cycle C in G: $C =$ path from v to u in **DFS** tree + edge $(u \to v)$.
2. Only if: Suppose there is a cycle $C = v_1 \to v_2 \to \ldots \to v_k \to v_1$.
   1. Let v_i be first node in C visited in **DFS**.
   2. All other nodes in C are descendants of v_i since they are reachable from v_i.
   3. Therefore, (v_{i-1}, v_i) (or (v_k, v_1) if $i = 1$) is a back edge.

□

Back edge and Cycles

Proposition

*G has a cycle iff there is a back-edge in **DFS**(G).*

Proof.

1. If: (u, v) is a back edge $\implies$ there is a cycle C in G: $C =$ path from v to u in **DFS** tree $+$ edge $(u \to v)$.
2. Only if: Suppose there is a cycle $C = v_1 \to v_2 \to \ldots \to v_k \to v_1$.
    1. Let v_i be first node in C visited in **DFS**.
    2. All other nodes in C are descendants of v_i since they are reachable from v_i.
    3. Therefore, (v_{i-1}, v_i) (or (v_k, v_1) if $i = 1$) is a back edge.

Back edge and Cycles

Proposition

G has a cycle iff there is a back-edge in **DFS**(G).

Proof.

1. If: (u, v) is a back edge $\implies$ there is a cycle C in G:
   $C =$ path from v to u in **DFS** tree $+$ edge $(u \to v)$.
2. Only if: Suppose there is a cycle
   $C = v_1 \to v_2 \to \ldots \to v_k \to v_1$.
   1. Let v_i be first node in C visited in **DFS**.
   2. All other nodes in C are descendants of v_i since they are reachable from v_i.
   3. Therefore, (v_{i-1}, v_i) (or (v_k, v_1) if $i = 1$) is a back edge.

□

Topological sorting of a DAG

Input: DAG G. With n vertices and m edges.

$O(n + m)$ algorithms for topological sorting

(A) Put source s of G as first in the order, remove s, and repeat. (Implementation not trivial.)

(B) Do **DFS** of G.
Compute post numbers.
Sort vertices by decreasing post number.

Topological sorting of a DAG

Input: DAG G. With n vertices and m edges.

$O(n + m)$ algorithms for topological sorting

(A) Put source s of G as first in the order, remove s, and repeat. (Implementation not trivial.)

(B) Do **DFS** of G.
Compute post numbers.
Sort vertices by decreasing post number.

Question

How to avoid sorting?

Topological sorting of a DAG

Input: DAG G. With n vertices and m edges.

$O(n + m)$ algorithms for topological sorting

(A) Put source s of G as first in the order, remove s, and repeat. (Implementation not trivial.)

(B) Do **DFS** of G.
Compute post numbers.
Sort vertices by decreasing post number.

Question

How to avoid sorting?
No need to sort - post numbering algorithm can output vertices...

DAGs and Partial Orders

Definition

A **partially ordered set** is a set S along with a binary relation $\preceq$ such that $\preceq$ is

1. **reflexive** ($a \preceq a$ for all $a \in V$),
2. **anti-symmetric** ($a \preceq b$ and $a \neq b$ implies $b \not\preceq a$), and
3. **transitive** ($a \preceq b$ and $b \preceq c$ implies $a \preceq c$).

Example: For numbers in the plane define $(x, y) \preceq (x', y')$ iff $x \leq x'$ and $y \leq y'$.

Observation: A *finite* partially ordered set is equivalent to a DAG. (No equal elements.)

Observation: A topological sort of a DAG corresponds to a complete (or total) ordering of the underlying partial order.

DAGs and Partial Orders

Definition

A **partially ordered set** is a set S along with a binary relation $\preceq$ such that $\preceq$ is

1. **reflexive** ($a \preceq a$ for all $a \in V$),
2. **anti-symmetric** ($a \preceq b$ and $a \neq b$ implies $b \not\preceq a$), and
3. **transitive** ($a \preceq b$ and $b \preceq c$ implies $a \preceq c$).

Example: For numbers in the plane define $(x, y) \preceq (x', y')$ iff $x \leq x'$ and $y \leq y'$.

Observation: A *finite* partially ordered set is equivalent to a DAG. (No equal elements.)

Observation: A topological sort of a DAG corresponds to a complete (or total) ordering of the underlying partial order.

DAGs and Partial Orders

Definition

A **partially ordered set** is a set S along with a binary relation $\preceq$ such that $\preceq$ is

1. **reflexive** ($a \preceq a$ for all $a \in V$),
2. **anti-symmetric** ($a \preceq b$ and $a \neq b$ implies $b \not\preceq a$), and
3. **transitive** ($a \preceq b$ and $b \preceq c$ implies $a \preceq c$).

Example: For numbers in the plane define $(x, y) \preceq (x', y')$ iff $x \leq x'$ and $y \leq y'$.

Observation: A *finite* partially ordered set is equivalent to a DAG. (No equal elements.)

Observation: A topological sort of a DAG corresponds to a complete (or total) ordering of the underlying partial order.

What's DAG but a sweet old fashioned notion

Who needs a DAG...

Example

1. V: set of n products (say, n different types of tablets).
2. Want to buy one of them, so you do market research...
3. Online reviews compare only pairs of them.
   ...Not everything compared to everything.
4. Given this partial information:
   1. Decide what is the best product.
   2. Decide what is the ordering of products from best to worst.
   3. ...

What DAGs got to do with it?

Or why we should care about DAGs

1. DAGs enable us to represent partial ordering information we have about some set (very common situation in the real world).
2. Questions about DAGs:
   1. Is a graph G a DAG?
      $\iff$
      Is the partial ordering information we have so far is consistent?
   2. Compute a topological ordering of a DAG.
      $\iff$
      Find an a consistent ordering that agrees with our partial information.
   3. Find comparisons to do so DAG has a unique topological sort.
      $\iff$
      Which elements to compare so that we have a consistent ordering of the items.

What DAGs got to do with it?

Or why we should care about DAGs

1. DAGs enable us to represent partial ordering information we have about some set (very common situation in the real world).
2. Questions about DAGs:
   1. Is a graph G a DAG?

      $\Longleftrightarrow$

      Is the partial ordering information we have so far is consistent?
   2. Compute a topological ordering of a DAG.

      $\Longleftrightarrow$

      Find an a consistent ordering that agrees with our partial information.
   3. Find comparisons to do so DAG has a unique topological sort.

      $\Longleftrightarrow$

      Which elements to compare so that we have a consistent ordering of the items.

What DAGs got to do with it?

Or why we should care about DAGs

1. DAGs enable us to represent partial ordering information we have about some set (very common situation in the real world).
2. Questions about DAGs:
    1. Is a graph G a DAG?

       $\iff$

       Is the partial ordering information we have so far is consistent?
    2. Compute a topological ordering of a DAG.

       $\iff$

       Find an a consistent ordering that agrees with our partial information.
    3. Find comparisons to do so DAG has a unique topological sort.

       $\iff$

       Which elements to compare so that we have a consistent ordering of the items.

What DAGs got to do with it?

Or why we should care about DAGs

1. DAGs enable us to represent partial ordering information we have about some set (very common situation in the real world).
2. Questions about DAGs:
   1. Is a graph G a DAG?

      $\iff$

      Is the partial ordering information we have so far is consistent?
   2. Compute a topological ordering of a DAG.

      $\iff$

      Find an a consistent ordering that agrees with our partial information.
   3. Find comparisons to do so DAG has a unique topological sort.

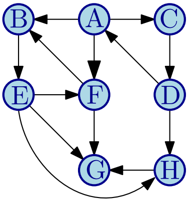
      $\iff$

      Which elements to compare so that we have a consistent ordering of the items.

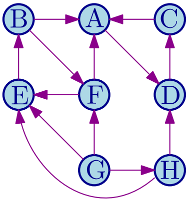
Part I

Linear time algorithm for finding all strong connected components of a directed graph

Reminder I: Graph G and its reverse graph $G^{\mathbf{rev}}$



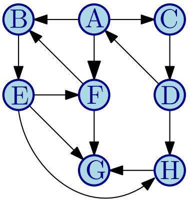
Graph G



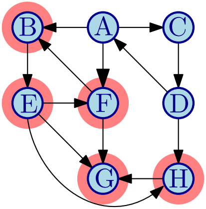
Reverse graph $G^{\mathbf{rev}}$

Reminder II: Graph G a vertex F

.. and its reachable set **rch(G, F)**



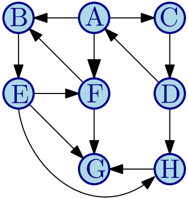
Graph G



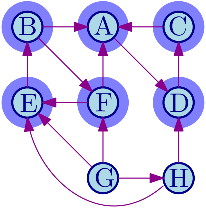
Reachable set of vertices from ***F***

Reminder III: Graph G a vertex **F**

.. and the set of vertices that can reach it in G: **rch**$(G^{\mathbf{rev}}, \mathbf{F})$



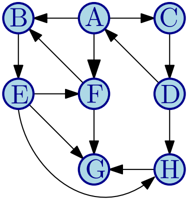
Graph G



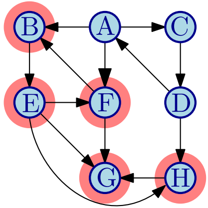
Set of vertices that can reach $\boldsymbol{F}$, computed via **DFS** in the reverse graph $\boldsymbol{G}^{\mathbf{rev}}$.

Reminder IV: Graph G a vertex F and...

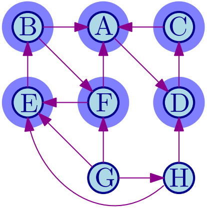
its strong connected component in G: $\mathrm{SCC}(G, F)$



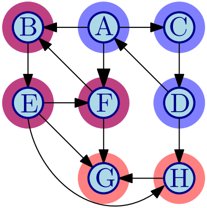
Graph G



$\mathsf{rch}(G, F)$

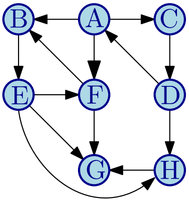


$\mathsf{rch}(G^{\mathrm{rev}}, F)$

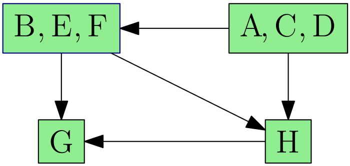


$$\mathrm{SCC}(G, F) = \mathsf{rch}(G, F) \cap \mathsf{rch}(G^{\mathrm{rev}}, F)$$

Reminder II: Strong connected components (SCC)



Graph G



Graph of SCCs G^{SCC}

Finding all SCCs of a Directed Graph

Problem

Given a directed graph $G = (V, E)$, output *all* its strong connected components.

Straightforward algorithm:

```
Mark all vertices in V as not visited.
for each vertex u ∈ V not visited yet do
    find SCC(G, u) the strong component of u:
        Compute rch(G, u) using DFS(G, u)
        Compute rch(G^rev, u) using DFS(G^rev, u)
        SCC(G, u) ⇐ rch(G, u) ∩ rch(G^rev, u)
        ∀u ∈ SCC(G, u): Mark u as visited.
```

Running time: $O(n(n+m))$

Is there an $O(n+m)$ time algorithm?

Finding all SCCs of a Directed Graph

Problem

Given a directed graph $G = (V, E)$, output *all* its strong connected components.

Straightforward algorithm:

```
Mark all vertices in V as not visited.
for each vertex u ∈ V not visited yet do
    find SCC(G, u) the strong component of u:
        Compute rch(G, u) using DFS(G, u)
        Compute rch(G^rev, u) using DFS(G^rev, u)
        SCC(G, u) ⇐ rch(G, u) ∩ rch(G^rev, u)
        ∀u ∈ SCC(G, u): Mark u as visited.
```

Running time: $O(n(n + m))$

Is there an $O(n + m)$ time algorithm?

Finding all SCCs of a Directed Graph

Problem

Given a directed graph $G = (V, E)$, output *all* its strong connected components.

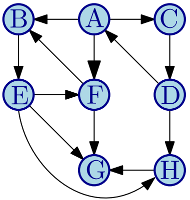
Straightforward algorithm:

```
Mark all vertices in V as not visited.
for each vertex u ∈ V not visited yet do
    find SCC(G, u) the strong component of u:
        Compute rch(G, u) using DFS(G, u)
        Compute rch(G^rev, u) using DFS(G^rev, u)
        SCC(G, u) ⇐ rch(G, u) ∩ rch(G^rev, u)
        ∀u ∈ SCC(G, u): Mark u as visited.
```

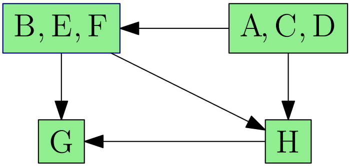
Running time: $O(n(n + m))$

Is there an $O(n + m)$ time algorithm?

Structure of a Directed Graph



Graph G



Graph of SCCs G^{SCC}

Reminder

G^{SCC} is created by collapsing every strong connected component to a single vertex.

Proposition

For a directed graph G, its meta-graph G^{SCC} is a DAG.

Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph...

Wishful Thinking Algorithm

1. Let u be a vertex in a *sink* SCC of G^{SCC}
2. Do **DFS**(u) to compute $\mathrm{SCC}(u)$
3. Remove $\mathrm{SCC}(u)$ and repeat

Justification

1. **DFS**(u) only visits vertices (and edges) in $\mathrm{SCC}(u)$
2.
3.
4.

Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph...

Wishful Thinking Algorithm

1. Let u be a vertex in a *sink* SCC of G^{SCC}
2. Do **DFS**(u) to compute $SCC(u)$
3. Remove $SCC(u)$ and repeat

Justification

1. **DFS**(u) only visits vertices (and edges) in $SCC(u)$
2.
3.
4.

Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph...

Wishful Thinking Algorithm

1. Let u be a vertex in a *sink* SCC of G^{SCC}
2. Do **DFS**(u) to compute $\mathrm{SCC}(u)$
3. Remove $\mathrm{SCC}(u)$ and repeat

Justification

1. **DFS**(u) only visits vertices (and edges) in $\mathrm{SCC}(u)$
2. ... since there are no edges coming out a sink!
3.
4.

Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph...

Wishful Thinking Algorithm

1. Let u be a vertex in a *sink* SCC of G^{SCC}
2. Do **DFS**(u) to compute $\mathrm{SCC}(u)$
3. Remove $\mathrm{SCC}(u)$ and repeat

Justification

1. **DFS**(u) only visits vertices (and edges) in $\mathrm{SCC}(u)$
2. ... since there are no edges coming out a sink!
3. **DFS**(u) takes time proportional to size of $\mathrm{SCC}(u)$
4.

Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph...

Wishful Thinking Algorithm

1. Let u be a vertex in a *sink* SCC of G^{SCC}
2. Do **DFS**(u) to compute $SCC(u)$
3. Remove $SCC(u)$ and repeat

Justification

1. **DFS**(u) only visits vertices (and edges) in $SCC(u)$
2. ... since there are no edges coming out a sink!
3. **DFS**(u) takes time proportional to size of $SCC(u)$
4. Therefore, total time $O(n + m)$!

Big Challenge(s)

How do we find a vertex in a sink SCC of $\mathrm{G}^{\mathrm{SCC}}$?

Can we obtain an *implicit* topological sort of $\mathrm{G}^{\mathrm{SCC}}$ without computing $\mathrm{G}^{\mathrm{SCC}}$?

Answer: **DFS**(G) gives some information!

Big Challenge(s)

How do we find a vertex in a sink SCC of G^{SCC}?

Can we obtain an *implicit* topological sort of G^{SCC} without computing G^{SCC}?

Answer: **DFS**(G) gives some information!

Big Challenge(s)

How do we find a vertex in a sink SCC of G^{SCC}?

Can we obtain an *implicit* topological sort of G^{SCC} without computing G^{SCC}?

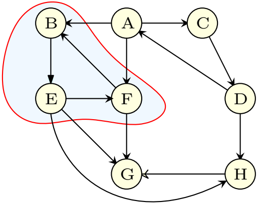
Answer: **DFS**(G) gives some information!

Post-visit times of SCCs

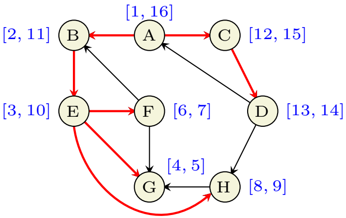
Definition

Given G and a SCC S of G, define $\text{post}(S) = \max_{u \in S} \text{post}(u)$ where post numbers are with respect to some **DFS**(G).

An Example



Graph G



Graph with pre-post times for **DFS**(*A*); black edges in tree

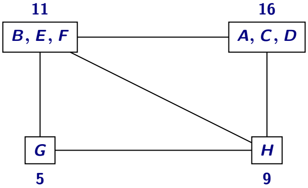


Figure: G^{SCC} with post times

Graph of strong connected components

... and post-visit times

Proposition

If S and S' are SCCs in G and (S, S') is an edge in G^{SCC} then $\mathrm{post}(S) > \mathrm{post}(S')$.

Proof.

Let u be first vertex in $S \cup S'$ that is visited.

1. If $u \in S$ then all of S' will be explored before **DFS**(u) completes.
2. If $u \in S'$ then all of S' will be explored before any of S.

□

A False Statement: If S and S' are SCCs in G and (S, S') is an edge in G^{SCC} then for *every* $u \in S$ and $u' \in S'$, $\mathrm{post}(u) > \mathrm{post}(u')$.

Graph of strong connected components

... and post-visit times

Proposition

If S and S' are SCCs in G and (S, S') is an edge in G^{SCC} then $\mathrm{post}(S) > \mathrm{post}(S')$.

Proof.

Let u be first vertex in $S \cup S'$ that is visited.

1. If $u \in S$ then all of S' will be explored before **DFS**(u) completes.
2. If $u \in S'$ then all of S' will be explored before any of S.

□

A False Statement: If S and S' are SCCs in G and (S, S') is an edge in $\mathrm{G}^{\mathrm{SCC}}$ then for *every* $u \in S$ and $u' \in S'$, $\mathrm{post}(u) > \mathrm{post}(u')$.

Topological ordering of the strong components

Corollary

Ordering SCC*s in decreasing order of* $\mathbf{post}(S)$ *gives a topological ordering of* G^{SCC}

Recall: for a DAG, ordering nodes in decreasing post-visit order gives a topological sort.

So...
DFS(G) gives some information on topological ordering of G^{SCC}!

Topological ordering of the strong components

Corollary

Ordering SCCs in decreasing order of $\mathrm{post}(S)$ gives a topological ordering of G^{SCC}

Recall: for a DAG, ordering nodes in decreasing post-visit order gives a topological sort.

So...
DFS(G) gives some information on topological ordering of G^{SCC}!

Finding Sources

Proposition

The vertex u with the highest post visit time belongs to a source SCC *in* G^{SCC}

Proof.

1. $\mathrm{post}(\mathrm{SCC}(u)) = \mathrm{post}(u)$
2. Thus, $\mathrm{post}(\mathrm{SCC}(u))$ is highest and will be output first in topological ordering of G^{SCC}.

□

Finding Sources

Proposition

The vertex u *with the highest post visit time belongs to a source* SCC *in* G^{SCC}

Proof.

1. $\mathbf{post}(\mathbf{SCC}(u)) = \mathbf{post}(u)$
2. Thus, $\mathbf{post}(\mathbf{SCC}(u))$ is highest and will be output first in topological ordering of $G^{\mathbf{SCC}}$.

□

Finding Sinks

Proposition

The vertex u with highest post visit time in $\mathsf{DFS}(G^{\mathrm{rev}})$ *belongs to a sink SCC of G.*

Proof.

1. u belongs to source SCC of G^{rev}
2. Since graph of SCCs of G^{rev} is the reverse of G^{SCC}, $\mathrm{SCC}(u)$ is sink SCC of G. □

Finding Sinks

Proposition

The vertex u with highest post visit time in **DFS**(G^{rev}) *belongs to a sink SCC of* G*.*

Proof.

1. u belongs to source SCC of G^{rev}
2. Since graph of SCCs of G^{rev} is the reverse of G^{SCC}, $\mathrm{SCC}(u)$ is sink SCC of G. □

Linear Time Algorithm

...for computing the strong connected components in **G**

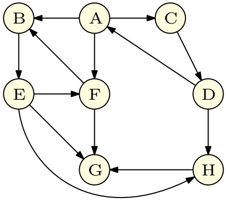
```
do DFS(G^rev) and sort vertices in decreasing post order.
Mark all nodes as unvisited
for each u in the computed order do
    if u is not visited then
        DFS(u)
        Let S_u be the nodes reached by u
        Output S_u as a strong connected component
        Remove S_u from G
```

Analysis

Running time is $O(n + m)$. (Exercise)

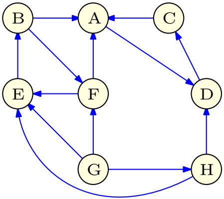
Linear Time Algorithm: An Example - Initial steps

Graph G:



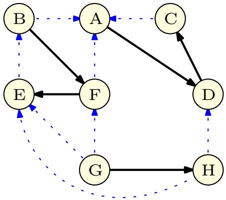
⟹

Reverse graph G^{rev}:



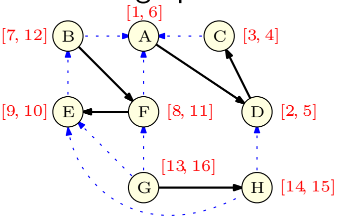
⟹

DFS of reverse graph:



⟹

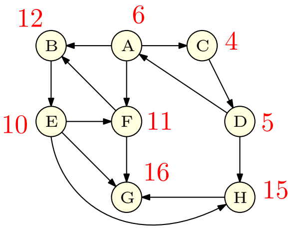
Pre/Post **DFS** numbering of reverse graph:



Linear Time Algorithm: An Example

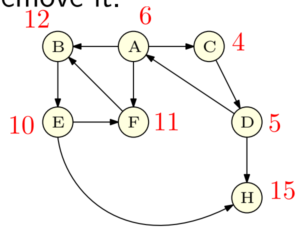
Removing connected components: 1

Original graph G with rev post numbers:



⟹

Do **DFS** from vertex G remove it.

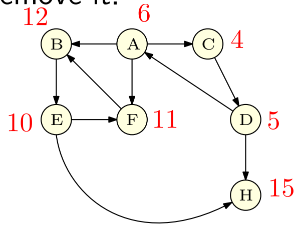


SCC computed:
$\{G\}$

Linear Time Algorithm: An Example

Removing connected components: 2

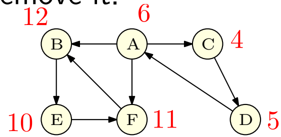
Do **DFS** from vertex G remove it.



SCC computed:
$\{G\}$

$\Longrightarrow$

Do **DFS** from vertex H, remove it.

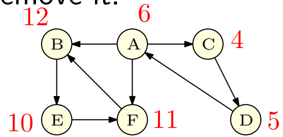


SCC computed:
$\{G\}, \{H\}$

Linear Time Algorithm: An Example

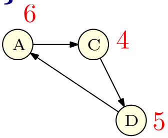
Removing connected components: 3

Do **DFS** from vertex H, remove it.



$\Longrightarrow$

Do **DFS** from vertex B
Remove visited vertices:
$\{F, B, E\}$.



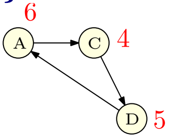
SCC computed:
$\{G\}, \{H\}$

SCC computed:
$\{G\}, \{H\}, \{F, B, E\}$

Linear Time Algorithm: An Example

Removing connected components: 4

Do **DFS** from vertex F
Remove visited vertices:
$\{F, B, E\}$.



SCC computed:
$\{G\}, \{H\}, \{F, B, E\}$

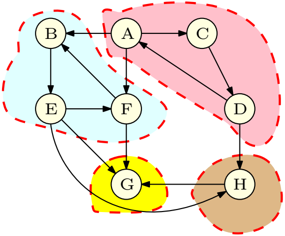
$\Longrightarrow$

Do **DFS** from vertex A
Remove visited vertices:
$\{A, C, D\}$.

SCC computed:
$\{G\}, \{H\}, \{F, B, E\}, \{A, C, D\}$

Linear Time Algorithm: An Example

Final result



SCC computed:

$\{G\}, \{H\}, \{F, B, E\}, \{A, C, D\}$

Which is the correct answer!

Obtaining the meta-graph...

Once the strong connected components are computed.

Exercise:

Given all the strong connected components of a directed graph $G = (V, E)$ show that the meta-graph G^{SCC} can be obtained in $O(m + n)$ time.

Correctness: more details

1. let $S_1, S_2, \ldots, S_k$ be strong components in G
2. Strong components of G^{rev} and G are same and meta-graph of G is reverse of meta-graph of G^{rev}.
3. consider **DFS**(G^{rev}) and let $u_1, u_2, \ldots, u_k$ be such that $\mathrm{post}(u_i) = \mathrm{post}(S_i) = \max_{v \in S_i} \mathrm{post}(v)$.
4. Assume without loss of generality that $post(u_k) > post(u_{k-1}) \geq \ldots \geq post(u_1)$ (renumber otherwise). Then $S_k, S_{k-1}, \ldots, S_1$ is a topological sort of meta-graph of G^{rev} and hence $S_1, S_2, \ldots, S_k$ is a topological sort of the meta-graph of G.
5. u_k has highest post number and **DFS**(u_k) will explore all of S_k which is a sink component in G.
6. After S_k is removed u_{k-1} has highest post number and **DFS**(u_{k-1}) will explore all of S_{k-1} which is a sink component in remaining graph $G - S_k$. Formal proof by induction.

Correctness: more details

1. let $S_1, S_2, \ldots, S_k$ be strong components in G
2. Strong components of G^{rev} and G are same and meta-graph of G is reverse of meta-graph of G^{rev}.
3. consider **DFS**(G^{rev}) and let $u_1, u_2, \ldots, u_k$ be such that $\mathrm{post}(u_i) = \mathrm{post}(S_i) = \max_{v \in S_i} \mathrm{post}(v)$.
4. Assume without loss of generality that $post(u_k) > post(u_{k-1}) \geq \ldots \geq post(u_1)$ (renumber otherwise). Then $S_k, S_{k-1}, \ldots, S_1$ is a topological sort of meta-graph of G^{rev} and hence $S_1, S_2, \ldots, S_k$ is a topological sort of the meta-graph of G.
5. u_k has highest post number and **DFS**(u_k) will explore all of S_k which is a sink component in G.
6. After S_k is removed u_{k-1} has highest post number and **DFS**(u_{k-1}) will explore all of S_{k-1} which is a sink component in remaining graph $G - S_k$. Formal proof by induction.

Correctness: more details

1. let $S_1, S_2, \ldots, S_k$ be strong components in G
2. Strong components of G^{rev} and G are same and meta-graph of G is reverse of meta-graph of G^{rev}.
3. consider **DFS**(G^{rev}) and let $u_1, u_2, \ldots, u_k$ be such that $\mathrm{post}(u_i) = \mathrm{post}(S_i) = \max_{v \in S_i} \mathrm{post}(v)$.
4. Assume without loss of generality that $post(u_k) > post(u_{k-1}) \geq \ldots \geq post(u_1)$ (renumber otherwise). Then $S_k, S_{k-1}, \ldots, S_1$ is a topological sort of meta-graph of G^{rev} and hence $S_1, S_2, \ldots, S_k$ is a topological sort of the meta-graph of G.
5. u_k has highest post number and **DFS**(u_k) will explore all of S_k which is a sink component in G.
6. After S_k is removed u_{k-1} has highest post number and **DFS**(u_{k-1}) will explore all of S_{k-1} which is a sink component in remaining graph $G - S_k$. Formal proof by induction.

Correctness: more details

1. let $S_1, S_2, \ldots, S_k$ be strong components in G
2. Strong components of G^{rev} and G are same and meta-graph of G is reverse of meta-graph of G^{rev}.
3. consider **DFS**(G^{rev}) and let $u_1, u_2, \ldots, u_k$ be such that $\mathrm{post}(u_i) = \mathrm{post}(S_i) = \max_{v \in S_i} \mathrm{post}(v)$.
4. Assume without loss of generality that $post(u_k) > post(u_{k-1}) \geq \ldots \geq post(u_1)$ (renumber otherwise). Then $S_k, S_{k-1}, \ldots, S_1$ is a topological sort of meta-graph of G^{rev} and hence $S_1, S_2, \ldots, S_k$ is a topological sort of the meta-graph of G.
5. u_k has highest post number and **DFS**(u_k) will explore all of S_k which is a sink component in G.
6. After S_k is removed u_{k-1} has highest post number and **DFS**(u_{k-1}) will explore all of S_{k-1} which is a sink component in remaining graph $G - S_k$. Formal proof by induction.

Correctness: more details

1. let $S_1, S_2, \ldots, S_k$ be strong components in G
2. Strong components of G^{rev} and G are same and meta-graph of G is reverse of meta-graph of G^{rev}.
3. consider **DFS**(G^{rev}) and let $u_1, u_2, \ldots, u_k$ be such that $\mathrm{post}(u_i) = \mathrm{post}(S_i) = \max_{v \in S_i} \mathrm{post}(v)$.
4. Assume without loss of generality that $post(u_k) > post(u_{k-1}) \geq \ldots \geq post(u_1)$ (renumber otherwise). Then $S_k, S_{k-1}, \ldots, S_1$ is a topological sort of meta-graph of G^{rev} and hence $S_1, S_2, \ldots, S_k$ is a topological sort of the meta-graph of G.
5. u_k has highest post number and **DFS**(u_k) will explore all of S_k which is a sink component in G.
6. After S_k is removed u_{k-1} has highest post number and **DFS**(u_{k-1}) will explore all of S_{k-1} which is a sink component in remaining graph $G - S_k$. Formal proof by induction.

Correctness: more details

1. let $S_1, S_2, \ldots, S_k$ be strong components in G
2. Strong components of G^{rev} and G are same and meta-graph of G is reverse of meta-graph of G^{rev}.
3. consider **DFS**(G^{rev}) and let $u_1, u_2, \ldots, u_k$ be such that $\mathrm{post}(u_i) = \mathrm{post}(S_i) = \max_{v \in S_i} \mathrm{post}(v)$.
4. Assume without loss of generality that $post(u_k) > post(u_{k-1}) \geq \ldots \geq post(u_1)$ (renumber otherwise). Then $S_k, S_{k-1}, \ldots, S_1$ is a topological sort of meta-graph of G^{rev} and hence $S_1, S_2, \ldots, S_k$ is a topological sort of the meta-graph of G.
5. u_k has highest post number and **DFS**(u_k) will explore all of S_k which is a sink component in G.
6. After S_k is removed u_{k-1} has highest post number and **DFS**(u_{k-1}) will explore all of S_{k-1} which is a sink component in remaining graph $G - S_k$. Formal proof by induction.

Part II

An Application to `make`

make Utility [Feldman]

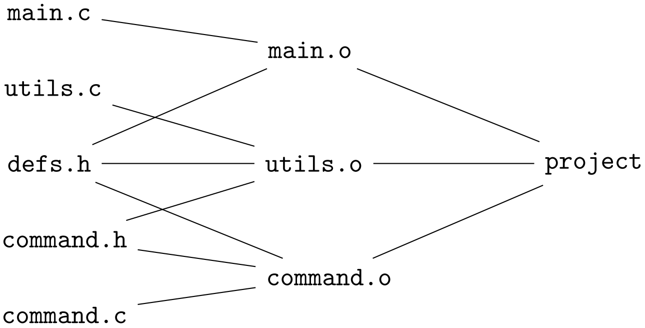
1. Unix utility for automatically building large software applications
2. A makefile specifies
   1. Object files to be created,
   2. Source/object files to be used in creation, and
   3. How to create them

An Example makefile

```
project:  main.o utils.o command.o
    cc -o project main.o utils.o command.o

main.o:  main.c defs.h
    cc -c main.c
utils.o:  utils.c defs.h command.h
    cc -c utils.c
command.o:  command.c defs.h command.h
    cc -c command.c
```

makefile as a Digraph



Computational Problems for `make`

1. Is the `makefile` reasonable?
2. If it is reasonable, in what order should the object files be created?
3. If it is not reasonable, provide helpful debugging information.
4. If some file is modified, find the fewest compilations needed to make application consistent.

Algorithms for `make`

1. Is the `makefile` reasonable? Is G a DAG?
2. If it is reasonable, in what order should the object files be created? Find a topological sort of a DAG.
3. If it is not reasonable, provide helpful debugging information. Output a cycle. More generally, output all strong connected components.
4. If some file is modified, find the fewest compilations needed to make application consistent.
    1. Find all vertices reachable (using **DFS**/**BFS**) from modified files in directed graph, and recompile them in proper order. Verify that one can find the files to recompile and the ordering in linear time.

Take away Points

1. Given a directed graph G, its SCCs and the associated acyclic meta-graph G^{SCC} give a structural decomposition of G that should be kept in mind.
2. There is a **DFS** based linear time algorithm to compute all the SCCs and the meta-graph. Properties of **DFS** crucial for the algorithm.
3. DAGs arise in many application and topological sort is a key property in algorithm design. Linear time algorithms to compute a topological sort (there can be many possible orderings so not unique).

Part III

Not for lecture - why do we have to use the reverse graph in computing the SCC?

Finding a sink via post numbers in a DAG

Lemma

Let G be a DAG, and consider the vertex u in G that minimizes $\mathrm{post}(u)$. Then u is a sink of G.

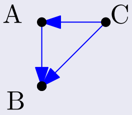
Proof.

The minimum $\mathrm{post}(\cdot)$ is assigned the first time **DFS** returns for its recursion. Let $\pi = v_1, v_2, \ldots, v_k = u$ be the sequence of vertices visited by the **DFS** at this point. Clearly, u (i.e., v_k), can not have an edge going into $v_1, \ldots, v_{k-1}$ since this would violates the assumption that there are no cycles. Similarly, u can not have an outgoing edge going into a vertex $z \in \mathsf{V}(\mathsf{G}) \setminus \{v_1, \ldots, v_k\}$, since the **DFS** would have continued into z, and u would not have been the first vertex to get assigned a post number. We conclude that u has no outgoing edges, and it is thus a sink. □

Counterexample: Finding a source via min post numbers in a DAG

Counter example

Let G be a DAG, and consider the vertex u in G that minimizes $\mathrm{post}(u)$ is a source. This is FALSE.



the **DFS** numbering might be:
A:[1,4]
B:[2,3]
C:[5,6]

But clearly B is not a source.

Finding a source via post numbers in a DAG

Lemma

Let G be a DAG, and consider the vertex u in G that maximizes $\mathrm{post}(u)$. Then u is a source of G.

Proof: Exercise (And should already be in the slides.)

Meta graph computing the sink..

We proved:

Lemma

Consider the graph G^{SCC}, *with every CC* $\boldsymbol{S} \in V\left(G^{\mathrm{SCC}}\right)$ *numbered by* $\mathbf{post}(\boldsymbol{S})$. *Then:*

$$\forall\,(\boldsymbol{S} \rightarrow \boldsymbol{T}) \in E(G^{\mathrm{SCC}}) \qquad \mathbf{post}(\boldsymbol{S}) > \mathbf{post}(\boldsymbol{T}).$$

1. So, the SCC realizing **min** $\mathbf{post}(\boldsymbol{S})$ is indeed a sink of $\mathrm{G}^{\mathrm{SCC}}$.
2. But how to compute this? Not clear at all.

Meta graph computing a source is easy!

1. The SCC realizing $\max \mathrm{post}(S)$ is a source of G^{SCC}.
2. Furthermore, computing

$$\max_{S \in V(G^{SCC})} \mathrm{post}(S) = \max_{S \in V(G^{SCC})} \max_{v \in S} \mathrm{post}(v) = \max_{v \in V(G)} \mathrm{post}(v).$$

   is easy!
3. So computing a source in the meta-graph is easy from the post numbering.
4. But the algorithm needs a sink of the meta graph. Thus, we compute a vertex in the source SCC of the meta-graph of $(G^{\mathbf{rev}})^{SCC} = (G^{SCC})^{\mathbf{rev}}$.

Notes

Notes

Notes

Notes