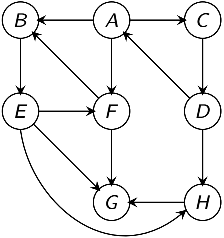
DFS in Directed Graphs, Strong Connected Components, DAGs

Lecture 2

January 20, 2011

Strong Connected Components (SCCs)



Algorithmic Problem

Find all SCCs of a given directed graph.

Previous lecture: saw an $\mathbf{O(n \cdot (n + m))}$ time algorithm.
This lecture: $\mathbf{O(n + m)}$ time algorithm.

Graph of SCCs

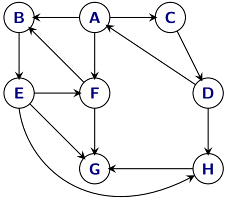


Figure: Graph $\mathbf{G}$

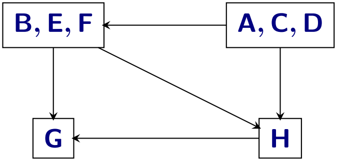


Figure: Graph of SCCs $\mathbf{G}^{\mathrm{SCC}}$

Meta-graph of SCCs

Let $\mathbf{S_1}, \mathbf{S_2}, \ldots \mathbf{S_k}$ be the SCCs of $\mathbf{G}$. The graph of SCCs is $\mathbf{G}^{\mathrm{SCC}}$

- Vertices are $\mathbf{S_1}, \mathbf{S_2}, \ldots \mathbf{S_k}$
- There is an edge $(\mathbf{S_i}, \mathbf{S_j})$ if there is some $\mathbf{u} \in \mathbf{S_i}$ and $\mathbf{v} \in \mathbf{S_j}$ such that $(\mathbf{u}, \mathbf{v})$ is an edge in $\mathbf{G}$.

Reversal and SCCs

Proposition

For any graph $\mathbf{G}$*, the graph of* SCC*s of* $\mathbf{G}^{\mathbf{rev}}$ *is the same as the reversal of* $\mathbf{G}^{\mathbf{SCC}}$*.*

Proof.

Exercise. □

SCCs and DAGs

Proposition

For any graph $\mathbf{G}$, *the graph* $\mathbf{G}^{\mathrm{SCC}}$ *has no directed cycle.*

Proof.

If $\mathbf{G}^{\mathrm{SCC}}$ has a cycle $\mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_k$ then $\mathbf{S}_1 \cup \mathbf{S}_2 \cup \cdots \cup \mathbf{S}_k$ is an SCC in $\mathbf{G}$. Formal details: exercise. $\square$

SCCs and DAGs

Proposition

For any graph $\mathbf{G}$, the graph $\mathbf{G}^{\mathrm{SCC}}$ has no directed cycle.

Proof.

If $\mathbf{G}^{\mathrm{SCC}}$ has a cycle $\mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_k$ then $\mathbf{S}_1 \cup \mathbf{S}_2 \cup \cdots \cup \mathbf{S}_k$ is an SCC in $\mathbf{G}$. Formal details: exercise. $\square$

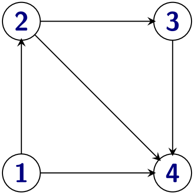
Part I

Directed Acyclic Graphs

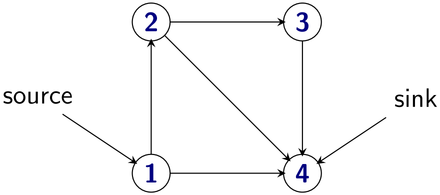
Directed Acyclic Graphs

Definition

A directed graph **G** is a **directed acyclic graph** (DAG) if there is no directed cycle in **G**.



Sources and Sinks



Definition

- A vertex **u** is a **source** if it has no in-coming edges.
- A vertex **u** is a **sink** if it has no out-going edges.

Simple DAG Properties

- Every DAG $\mathbf{G}$ has at least one source and at least one sink.
- If $\mathbf{G}$ is a DAG if and only if $\mathbf{G}^{\text{rev}}$ is a DAG.
- $\mathbf{G}$ is a DAG if and only each node is in its own strong connected component.

Formal proofs: exercise.

Simple DAG Properties

- Every DAG **G** has at least one source and at least one sink.
- If **G** is a DAG if and only if $\mathbf{G}^{\mathrm{rev}}$ is a DAG.
- **G** is a DAG if and only each node is in its own strong connected component.

Formal proofs: exercise.

Simple DAG Properties

- Every DAG $\mathbf{G}$ has at least one source and at least one sink.
- If $\mathbf{G}$ is a DAG if and only if $\mathbf{G}^{\mathbf{rev}}$ is a DAG.
- $\mathbf{G}$ is a DAG if and only each node is in its own strong connected component.

Formal proofs: exercise.

Simple DAG Properties

- Every DAG **G** has at least one source and at least one sink.
- If **G** is a DAG if and only if $\mathbf{G}^{\mathbf{rev}}$ is a DAG.
- **G** is a DAG if and only each node is in its own strong connected component.

Formal proofs: exercise.

Simple DAG Properties

- Every DAG **G** has at least one source and at least one sink.
- If **G** is a DAG if and only if $\mathbf{G}^{\mathbf{rev}}$ is a DAG.
- **G** is a DAG if and only each node is in its own strong connected component.

Formal proofs: exercise.

Topological Ordering/Sorting

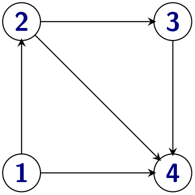


Figure: Graph $\mathbf{G}$

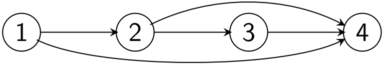


Figure: Topological Ordering of $\mathbf{G}$

Definition

A **topological ordering**/**topological sorting** of $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ is an ordering $<$ on $\mathbf{V}$ such that if $(\mathbf{u}, \mathbf{v}) \in \mathbf{E}$ then $\mathbf{u} < \mathbf{v}$.

DAGs and Topological Sort

Lemma

A directed graph $\mathbf{G}$ *can be topologically ordered iff it is a* DAG.

DAGs and Topological Sort

Lemma

A directed graph $\mathbf{G}$ can be topologically ordered iff it is a DAG.

Proof.

Only if: Suppose $\mathbf{G}$ is not a DAG and has a topological ordering $<$.
$\mathbf{G}$ has a cycle $\mathbf{C} = \mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_k, \mathbf{u}_1$.
Then $\mathbf{u}_1 < \mathbf{u}_2 < \ldots < \mathbf{u}_k < \mathbf{u}_1$! A contradiction. $\square$

DAGs and Topological Sort

Lemma

A directed graph $\mathbf{G}$ *can be topologically ordered iff it is a* DAG.

Proof.

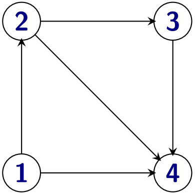
If: Consider the following algorithm:

- Pick a source $\mathbf{u}$, output it.
- Remove $\mathbf{u}$ and all edges out of $\mathbf{u}$.
- Repeat until graph is empty.
- Exercise: prove this gives an ordering.

$\square$

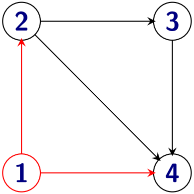
Exercise: show above algorithm can be implemented in $\mathbf{O(m+n)}$ time.

Topological Sort: An Example



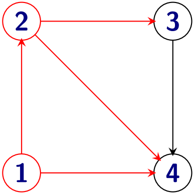
Output: 1 2 3 4

Topological Sort: An Example



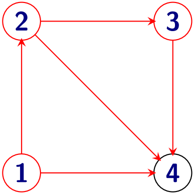
Output: 1 2 3 4

Topological Sort: An Example



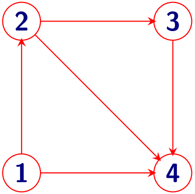
Output: 1 2 3 4

Topological Sort: An Example



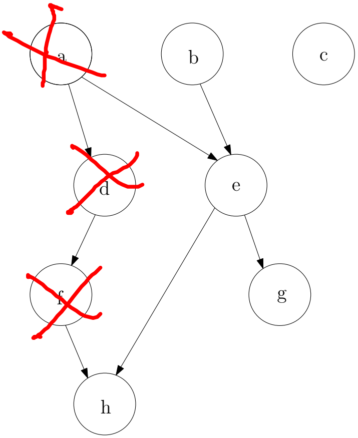
Output: 1 2 3 4

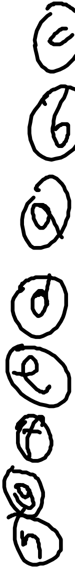
Topological Sort: An Example

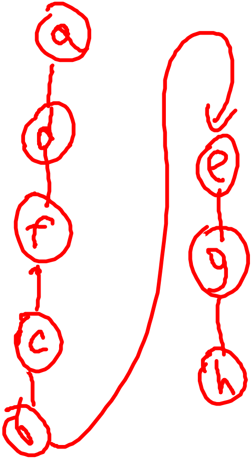


Output: 1 2 3 4

Topological Sort: Another Example







DAGs and Topological Sort

Note: A DAG $\mathbf{G}$ may have many different topological sorts.

Question: What is a DAG with the most number of distinct topological sorts for a given number $\mathbf{n}$ of vertices?

n singletons n! orderings.

Question: What is a DAG with the least number of distinct topological sorts for a given number $\mathbf{n}$ of vertices?

1 linked list

1 ordering

$1 \to 2 \to 3 \to \cdots \to n$

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:

- Compute **DFS(G)**
- If there is a back edge then **G** is not a DAG.
- 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* $post(v) > post(u)$, *then* $(\mathbf{u}, \mathbf{v})$ *is not in* **G**.

Using DFS...

... to check for Acylicity and compute Topological Ordering

Question

Given $\mathbf{G}$, is it a DAG? If it is, generate a topological sort.

DFS based algorithm:

- Compute **DFS(G)**
- If there is a back edge then $\mathbf{G}$ is not a DAG.
- Otherwise output nodes in decreasing post-visit order.

Correctness relies on the following:

Proposition

*$\mathbf{G}$ is a DAG iff there is no back-edge in **DFS(G)**.*

Proposition

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

Using DFS...

... to check for Acylicity and compute Topological Ordering

Question

Given $\mathbf{G}$, is it a DAG? If it is, generate a topological sort.

DFS based algorithm:

- Compute **DFS**$(\mathbf{G})$
- If there is a back edge then $\mathbf{G}$ is not a DAG.
- Otherwise output nodes in decreasing post-visit order.

Correctness relies on the following:

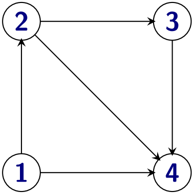
Proposition

$\mathbf{G}$ *is a* DAG *iff there is no back-edge in* **DFS**$(\mathbf{G})$.

Proposition

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

Example



1 [1, 8]

2 [2, 7]

3 [3, 6]

4 [4, 5]

Back edge and Cycles

Proposition

$\mathbf{G}$ has a cycle iff there is a back-edge in **DFS**$(\mathbf{G})$.

Proof.

If: $(\mathbf{u}, \mathbf{v})$ is a back edge implies there is a cycle $\mathbf{C}$ consisting of the path from $\mathbf{v}$ to $\mathbf{u}$ in **DFS** search tree and the edge $(\mathbf{u}, \mathbf{v})$.

Only if: Suppose there is a cycle $\mathbf{C} = \mathbf{v}_1 \rightarrow \mathbf{v}_2 \rightarrow \ldots \rightarrow \mathbf{v}_k \rightarrow \mathbf{v}_1$.
Let $\mathbf{v}_i$ be first node in $\mathbf{C}$ visited in **DFS**.
All other nodes in $\mathbf{C}$ are descendents of $\mathbf{v}_i$ since they are reachable from $\mathbf{v}_i$.
Therefore, $(\mathbf{v}_{i-1}, \mathbf{v}_i)$ (or $(\mathbf{v}_k, \mathbf{v}_1)$ if $\mathbf{i} = 1$) is a back edge. □

Back edge and Cycles

Proposition

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

Proof.

If: (u, v) is a back edge implies there is a cycle C consisting of the path from v to u in **DFS** search tree and the edge (u, v).

Only if: Suppose there is a cycle $C = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \rightarrow v_1$.
Let v_i be first node in C visited in **DFS**.
All other nodes in C are descendents of v_i since they are reachable from v_i.
Therefore, (v_{i-1}, v_i) (or (v_k, v_1) if $i = 1$) is a back edge. $\square$

Proposition

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

Proof.

Assume post(v) $>$ post(u) *and* $(\mathbf{u}, \mathbf{v})$ is an edge in $\mathbf{G}$. We derive a contradiction. One of two cases holds from **DFS** property.

- Case 1: $[\mathbf{pre(u)}, \mathbf{post(u)}]$ is contained in $[\mathbf{pre(v)}, \mathbf{post(v)}]$. Implies that $(\mathbf{u}, \mathbf{v})$ is a back edge but a DAG has no back edges!
- Case 2: $[\mathbf{pre(u)}, \mathbf{post(u)}]$ is disjoint from $[\mathbf{pre(v)}, \mathbf{post(v)}]$. This cannot happen since $\mathbf{v}$ would be explored from $\mathbf{u}$.

$\square$

DAGs and Partial Orders

Definition

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

(i) reflexive ($\mathbf{a} \preceq \mathbf{a}$ for all $\mathbf{a} \in \mathbf{V}$),

(ii) anti-symmetric ($\mathbf{a} \preceq \mathbf{b}$ and $\mathbf{a} \neq \mathbf{b}$ implies $\mathbf{b} \not\preceq \mathbf{a}$), and

(iii) transitive ($\mathbf{a} \preceq \mathbf{b}$ and $\mathbf{b} \preceq \mathbf{c}$ implies $\mathbf{a} \preceq \mathbf{c}$).

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

Observation: A *finite* partially ordered set is equivalent to a DAG.

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 $\mathbf{S}$ along with a binary relation $\preceq$ such that $\preceq$ is

(i) reflexive ($\mathbf{a} \preceq \mathbf{a}$ for all $\mathbf{a} \in \mathbf{V}$),

(ii) anti-symmetric ($\mathbf{a} \preceq \mathbf{b}$ and $\mathbf{a} \neq \mathbf{b}$ implies $\mathbf{b} \not\preceq \mathbf{a}$), and

(iii) transitive ($\mathbf{a} \preceq \mathbf{b}$ and $\mathbf{b} \preceq \mathbf{c}$ implies $\mathbf{a} \preceq \mathbf{c}$).

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

Observation: A *finite* partially ordered set is equivalent to a DAG.

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 $\mathbf{S}$ along with a binary relation $\preceq$ such that $\preceq$ is

(i) reflexive ($\mathbf{a} \preceq \mathbf{a}$ for all $\mathbf{a} \in \mathbf{V}$),

(ii) anti-symmetric ($\mathbf{a} \preceq \mathbf{b}$ and $\mathbf{a} \neq \mathbf{b}$ implies $\mathbf{b} \not\preceq \mathbf{a}$), and

(iii) transitive ($\mathbf{a} \preceq \mathbf{b}$ and $\mathbf{b} \preceq \mathbf{c}$ implies $\mathbf{a} \preceq \mathbf{c}$).

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

Observation: A *finite* partially ordered set is equivalent to a DAG.

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

Part II

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

Finding all SCCs of a Directed Graph

Problem

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

Straightforward algorithm:

```
For each vertex u ∈ V do
    find SCC(G, u) the strong component containing u as follows:
        Obtain rch(G, u) using DFS(G, u)
        Obtain rch(G^rev, u) using DFS(G^rev, u)
        Output SCC(G, u) = rch(G, u) ∩ rch(G^rev, u)
```

Running time: $\mathbf{O(n(n+m))}$

Is there an $\mathbf{O(n+m)}$ time algorithm?

Finding all SCCs of a Directed Graph

Problem

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

Straightforward algorithm:

```
For each vertex u ∈ V do
    find SCC(G, u) the strong component containing u as follows:
        Obtain rch(G, u) using DFS(G, u)
        Obtain rch(G^rev, u) using DFS(G^rev, u)
        Output SCC(G, u) = rch(G, u) ∩ rch(G^rev, u)
```

Running time: $\mathbf{O}(\mathbf{n}(\mathbf{n}+\mathbf{m}))$

Is there an $\mathbf{O}(\mathbf{n}+\mathbf{m})$ time algorithm?

Finding all SCCs of a Directed Graph

Problem

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

Straightforward algorithm:

```
For each vertex u ∈ V do
    find SCC(G, u) the strong component containing u as follows:
        Obtain rch(G, u) using DFS(G, u)
        Obtain rch(G^rev, u) using DFS(G^rev, u)
        Output SCC(G, u) = rch(G, u) ∩ rch(G^rev, u)
```

Running time: $\mathbf{O(n(n+m))}$

Is there an $\mathbf{O(n+m)}$ time algorithm?

Structure of a Directed Graph

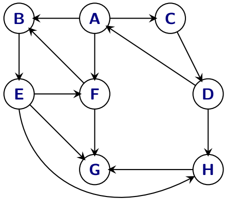


Figure: Graph $\mathbf{G}$

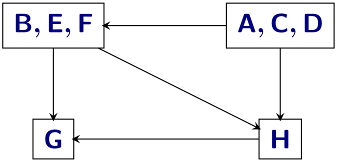


Figure: Graph of SCCs $\mathbf{G}^{\mathrm{SCC}}$

Proposition

For a directed graph $\mathbf{G}$, *its meta-graph* $\mathbf{G}^{\mathrm{SCC}}$ *is a* DAG.

Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph.

Algorithm

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

Justification

- **DFS**$(\mathbf{u})$ only visits vertices (and edges) in $\mathrm{SCC}(\mathbf{u})$
- **DFS**$(\mathbf{u})$ takes time proportional to size of $\mathrm{SCC}(\mathbf{u})$
- Therefore, total time $\mathbf{O(n+m)}$!

Big Challenge(s)

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

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

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

Big Challenge(s)

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

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

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

Big Challenge(s)

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

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

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

Post-visit times of SCCs

Definition

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

An Example

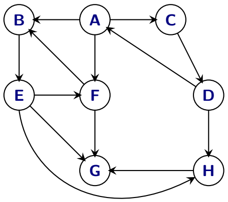


Figure: Graph **G**

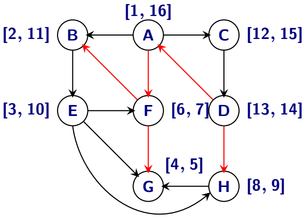


Figure: Graph with pre-post times for **DFS** (A); black edges in tree

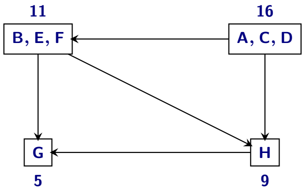


Figure: $\mathbf{G}^{\mathrm{SCC}}$ with post times

$\mathbf{G}^{\mathrm{SCC}}$ and post-visit times

Proposition

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

Proof.

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

- If $\mathbf{u} \in \mathbf{S}$ then all of $\mathbf{S'}$ will be explored before $\mathbf{DFS}(\mathbf{u})$ completes.
- If $\mathbf{u} \in \mathbf{S'}$ then all of $\mathbf{S'}$ will be explored before any of $\mathbf{S}$.

$\square$

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

G^{SCC} 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.

- If $u \in S$ then all of S' will be explored before $\mathbf{DFS}(u)$ completes.
- If $u \in S'$ then all of S' will be explored before any of S.

$\square$

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 $u' \in S'$, $\mathrm{post}(u) > \mathrm{post}(u')$.

$\mathbf{G}^{\mathrm{SCC}}$ and post-visit times

Proposition

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

Proof.

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

- If $\mathbf{u} \in \mathbf{S}$ then all of $\mathbf{S'}$ will be explored before **DFS**$(\mathbf{u})$ completes.
- If $\mathbf{u} \in \mathbf{S'}$ then all of $\mathbf{S'}$ will be explored before any of $\mathbf{S}$.

$\square$

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

G^{SCC} 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.

- If $u \in S$ then all of S' will be explored before $\mathrm{DFS}(u)$ completes.
- If $u \in S'$ then all of S' will be explored before any of S.

$\square$

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 $u' \in S'$, $\mathrm{post}(u) > \mathrm{post}(u')$.

$\mathbf{G}^{\mathrm{SCC}}$ and post-visit times

Proposition

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

Proof.

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

- If $\mathbf{u} \in \mathbf{S}$ then all of $\mathbf{S'}$ will be explored before $\mathbf{DFS}(\mathbf{u})$ completes.
- If $\mathbf{u} \in \mathbf{S'}$ then all of $\mathbf{S'}$ will be explored before any of $\mathbf{S}$.

$\square$

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

$\mathbf{G}^{\mathrm{SCC}}$ and post-visit times

Proposition

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

Proof.

Let $\mathbf{u}$ be first vertex in $\mathsf{S} \cup \mathsf{S}'$ that is visited.

- If $\mathbf{u} \in \mathsf{S}$ then all of S' will be explored before **DFS**$(\mathbf{u})$ completes.
- If $\mathbf{u} \in \mathsf{S}'$ then all of S' will be explored before any of S.

$\square$

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

Topological ordering of $\mathbf{G}^{\mathrm{SCC}}$

Corollary

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

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

So...

$\mathbf{DFS}(\mathbf{G})$ gives some information on topological ordering of $\mathbf{G}^{\mathrm{SCC}}$!

Topological ordering of $\mathbf{G}^{\mathrm{SCC}}$

Corollary

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

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

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

An Example

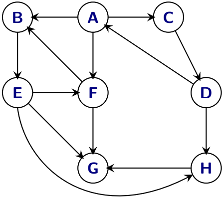


Figure: Graph **G**

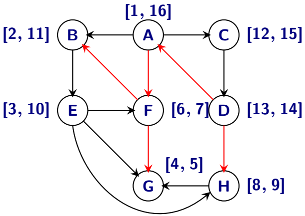


Figure: Graph with pre-post times for **DFS** (A); black edges in tree

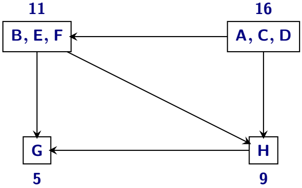


Figure: $\mathbf{G}^{\mathrm{SCC}}$ with post times

Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph.

Algorithm

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

Justification

- **DFS**$(\mathbf{u})$ only visits vertices (and edges) in $\mathrm{SCC}(\mathbf{u})$
- **DFS**$(\mathbf{u})$ takes time proportional to size of $\mathrm{SCC}(\mathbf{u})$
- Therefore, total time $\mathbf{O(n + m)}$!

Big Challenge(s)

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

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

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

Big Challenge(s)

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

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

Answer: **DFS**$(\mathbf{G})$ gives some information!

Big Challenge(s)

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

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

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

Finding Sources

Proposition

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

Proof.

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

Finding Sources

Proposition

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

Proof.

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

$\square$

Finding Sinks

Proposition

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

Proof.

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

Finding Sinks

Proposition

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

Proof.

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

Linear Time Algorithm

```
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

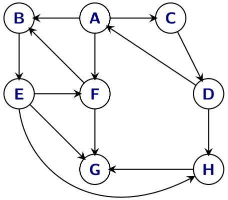


Figure: Graph **G**

Figure: $\mathbf{G}^{\mathrm{rev}}$ with pre-post times. Red edges not traversed in **DFS**

Order of second **DFS**: **DFS**$(\mathbf{G}) = \{\mathbf{G}\}$; **DFS**$(\mathbf{H}) = \{\mathbf{H}\}$; **DFS**$(\mathbf{B}) = \{\mathbf{B}, \mathbf{E}, \mathbf{F}\}$; **DFS**$(\mathbf{A}) = \{\mathbf{A}, \mathbf{C}, \mathbf{D}\}$.

Linear Time Algorithm: An Example

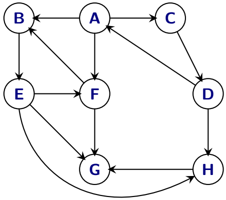


Figure: Graph **G**

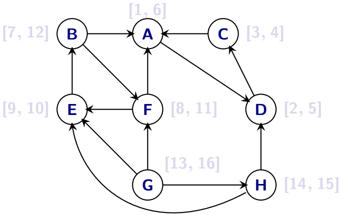


Figure: $\mathbf{G}^{\mathbf{rev}}$ with pre-post times. Red edges not traversed in **DFS**

Order of second **DFS**: **DFS**$(\mathsf{G}) = \{\mathsf{G}\}$; **DFS**$(\mathsf{H}) = \{\mathsf{H}\}$; **DFS**$(\mathsf{B}) = \{\mathsf{B}, \mathsf{E}, \mathsf{F}\}$; **DFS**$(\mathsf{A}) = \{\mathsf{A}, \mathsf{C}, \mathsf{D}\}$.

Linear Time Algorithm: An Example

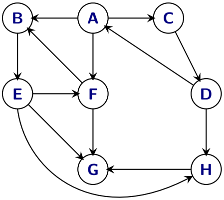


Figure: Graph **G**

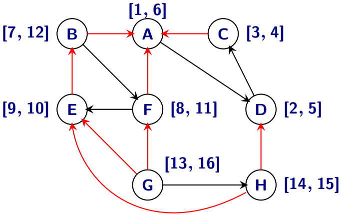


Figure: $\mathbf{G}^{\mathrm{rev}}$ with pre-post times. Red edges not traversed in **DFS**

Order of second **DFS**: $\mathbf{DFS}(\mathbf{G}) = \{\mathbf{G}\}$; $\mathbf{DFS}(\mathbf{H}) = \{\mathbf{H}\}$; $\mathbf{DFS}(\mathbf{B}) = \{\mathbf{B}, \mathbf{E}, \mathbf{F}\}$; $\mathbf{DFS}(\mathbf{A}) = \{\mathbf{A}, \mathbf{C}, \mathbf{D}\}$.

Linear Time Algorithm: An Example

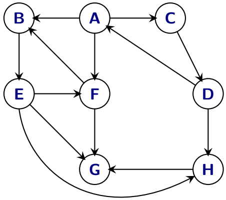


Figure: Graph **G**

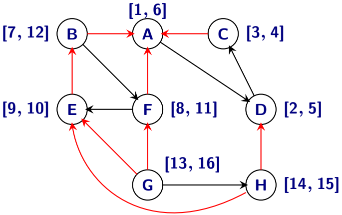


Figure: $\mathbf{G}^{\mathbf{rev}}$ with pre-post times. Red edges not traversed in **DFS**

Order of second **DFS**: $\mathbf{DFS(G)} = \{\mathbf{G}\}$; $\mathbf{DFS(H)} = \{\mathbf{H}\}$; $\mathbf{DFS(B)} = \{\mathbf{B}, \mathbf{E}, \mathbf{F}\}$; $\mathbf{DFS(A)} = \{\mathbf{A}, \mathbf{C}, \mathbf{D}\}$.

Linear Time Algorithm: An Example

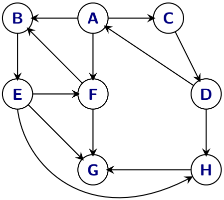


Figure: Graph **G**

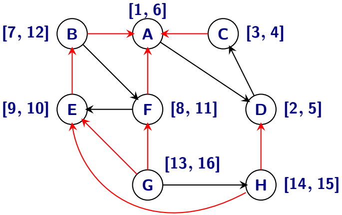


Figure: $\mathbf{G^{rev}}$ with pre-post times. Red edges not traversed in **DFS**

Order of second **DFS**: $\mathbf{DFS(G) = \{G\}}$; $\mathbf{DFS(H) = \{H\}}$; $\mathbf{DFS(B) = \{B, E, F\}}$; $\mathbf{DFS(A) = \{A, C, D\}}$.

Linear Time Algorithm: An Example

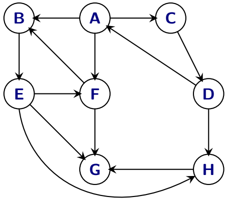


Figure: Graph **G**

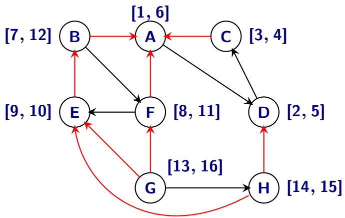


Figure: $\mathbf{G}^{\mathbf{rev}}$ with pre-post times. Red edges not traversed in **DFS**

Order of second **DFS**: $\mathbf{DFS(G) = \{G\}}$; $\mathbf{DFS(H) = \{H\}}$; $\mathbf{DFS(B) = \{B, E, F\}}$; $\mathbf{DFS(A) = \{A, C, D\}}$.

Linear Time Algorithm: An Example

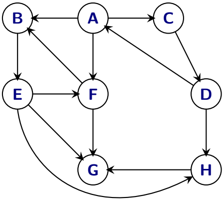


Figure: Graph $\mathbf{G}$

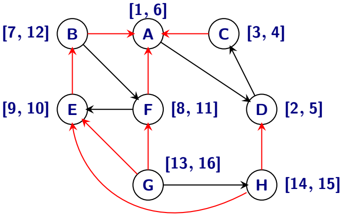


Figure: $\mathbf{G}^{\mathbf{rev}}$ with pre-post times. Red edges not traversed in **DFS**

Order of second **DFS**: $\mathbf{DFS(G) = \{G\}}$; $\mathbf{DFS(H) = \{H\}}$; $\mathbf{DFS(B) = \{B, E, F\}}$; $\mathbf{DFS(A) = \{A, C, D\}}$.

Obtaining the meta-graph from strong connected components

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

Correctness: more details

- let $S_1, S_2, \ldots, S_k$ be strong components in G
- Strong components of G^{rev} and G are same and meta-graph of G is reverse of meta-graph of G^{rev}.
- consider DFG(G^{rev}) and let $u_1, u_2, \ldots, u_k$ be such that post(u_i) = post(S_i) = $\max_{v \in S_i} \mathrm{post}(v)$.
- Assume without loss of generality that $\mathrm{post}(u_k) > \mathrm{post}(u_{k-1}) \geq \ldots \geq \mathrm{post}(u_1)$ (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.
- u_k has highest post number and **DFS** (u_k) will explore all of S_k which is a sink component in G.
- 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

- let $S_1, S_2, \ldots, S_k$ be strong components in G
- Strong components of G^{rev} and G are same and meta-graph of G is reverse of meta-graph of G^{rev}.
- consider DFG(G^{rev}) and let $u_1, u_2, \ldots, u_k$ be such that post(u_i) = post(S_i) = $\max_{v \in S_i} \mathrm{post}(v)$.
- Assume without loss of generality that $\mathrm{post}(u_k) > \mathrm{post}(u_{k-1}) \geq \ldots \geq \mathrm{post}(u_1)$ (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.
- u_k has highest post number and **DFS** (u_k) will explore all of S_k which is a sink component in G.
- 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

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

Correctness: more details

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

Correctness: more details

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

Correctness: more details

- let $S_1, S_2, \ldots, S_k$ be strong components in G
- Strong components of G^{rev} and G are same and meta-graph of G is reverse of meta-graph of G^{rev}.
- consider DFG(G^{rev}) and let $u_1, u_2, \ldots, u_k$ be such that post(u_i) = post(S_i) = $\max_{v \in S_i} \text{post}(v)$.
- Assume without loss of generality that $\text{post}(u_k) > \text{post}(u_{k-1}) \geq \ldots \geq \text{post}(u_1)$ (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.
- u_k has highest post number and **DFS** (u_k) will explore all of S_k which is a sink component in G.
- 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 III

An Application to `make`

make Utility [Feldman]

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

`make` Utility [Feldman]

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

make Utility [Feldman]

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

make Utility [Feldman]

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

`make` Utility [Feldman]

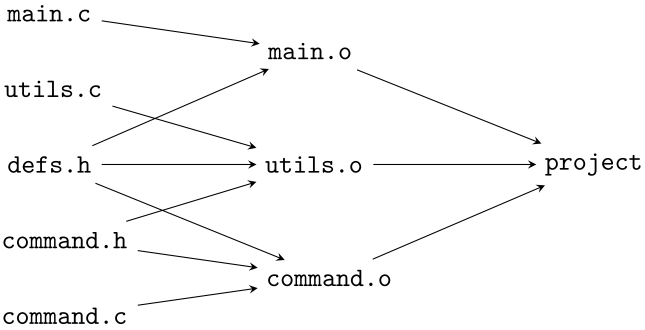
- Unix utility for automatically building large software applications
- A makefile specifies
  - Object files to be created,
  - Source/object files to be used in creation, and
  - How to create them

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`

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

Computational Problems for `make`

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

Computational Problems for `make`

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

Computational Problems for `make`

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

Algorithms for `make`

- Is the `makefile` reasonable? Is **G** a DAG?
- If it is reasonable, in what order should the object files be created? Find a topological sort of a DAG.
- If it is not reasonable, provide helpful debugging information. Output a cycle. More generally, output all strong connected components.
- If some file is modified, find the fewest compilations needed to make application consistent.
  - 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.

Algorithms for `make`

- Is the `makefile` reasonable? Is **G** a DAG?
- If it is reasonable, in what order should the object files be created? Find a topological sort of a DAG.
- If it is not reasonable, provide helpful debugging information. Output a cycle. More generally, output all strong connected components.
- If some file is modified, find the fewest compilations needed to make application consistent.
    - 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.

Algorithms for `make`

- Is the `makefile` reasonable? Is **G** a DAG?
- If it is reasonable, in what order should the object files be created? Find a topological sort of a DAG.
- If it is not reasonable, provide helpful debugging information. Output a cycle. More generally, output all strong connected components.
- If some file is modified, find the fewest compilations needed to make application consistent.
  - 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.

Algorithms for `make`

- Is the `makefile` reasonable? Is **G** a DAG?
- If it is reasonable, in what order should the object files be created? Find a topological sort of a DAG.
- If it is not reasonable, provide helpful debugging information. Output a cycle. More generally, output all strong connected components.
- If some file is modified, find the fewest compilations needed to make application consistent.
  - 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.

Algorithms for `make`

- Is the `makefile` reasonable? Is **G** a DAG?
- If it is reasonable, in what order should the object files be created? Find a topological sort of a DAG.
- If it is not reasonable, provide helpful debugging information. Output a cycle. More generally, output all strong connected components.
- If some file is modified, find the fewest compilations needed to make application consistent.
  - 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

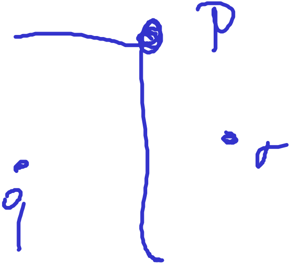
- Given a directed graph $\mathbf{G}$, its SCCs and the associated acyclic meta-graph $\mathbf{G}^{\mathrm{SCC}}$ give a structural decomposition of $\mathbf{G}$ that should be kept in mind.
- There is a **DFS** based linear time algorithm to compute all the SCCs and the meta-graph. Properties of **DFS** crucial for the algorithm.
- 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).

Notes

Notes

Notes

Notes



$q \prec p$

$q \prec r$

$\boxed{p \quad r}$