DFS in Directed Graphs, Strong Connected Components, and DAGs

Lecture 2
January 19, 2013
Algorithmic Problem

Find all SCCs of a given directed graph.

Previous lecture:
Saw an $O(n \cdot (n + m))$ time algorithm.
This lecture: $O(n + m)$ time algorithm.
Graph of SCCs

Let $S_1, S_2, \ldots, S_k$ be the strong connected components (i.e., SCCs) of $G$. The graph of SCCs is $G^{SCC}$

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

Meta-graph of SCCs
Proposition

For any graph $G$, the graph of SCCs of $G^{\text{rev}}$ is the same as the reversal of $G^{\text{SCC}}$.

Proof.

Exercise.
**Proposition**

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

**Proof.**

If $G^{\text{SCC}}$ has a cycle $S_1, S_2, \ldots, S_k$ then $S_1 \cup S_2 \cup \cdots \cup S_k$ should be in the same SCC in $G$. Formal details: exercise.
Part I

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

![Diagram of a directed acyclic graph with nodes 1, 2, 3, and 4 connected in a way that no cycle exists.]
### Definition

1. A vertex \( u \) is a **source** if it has no in-coming edges.
2. A vertex \( u \) is a **sink** if it has no out-going edges.
Simple DAG Properties

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

Formal proofs: exercise.
Simple DAG Properties

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

Formal proofs: exercise.
Topological Ordering/Sorting

Graph $G$

**Definition**

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

**Informal equivalent definition:**

One can order the vertices of the graph along a line (say the $x$-axis) such that all edges are from left to right.
Lemma

A directed graph \( G \) can be topologically ordered iff it is a **DAG**.

Proof.

\( \implies \): Suppose \( G \) is not a **DAG** and has a topological ordering \( \prec \). \( G \) has a cycle \( C = u_1, u_2, \ldots, u_k, u_1 \).

Then \( u_1 \prec u_2 \prec \ldots \prec u_k \prec u_1 \).

That is... \( u_1 \prec u_1 \).

A contradiction (to \( \prec \) being an order).

Not possible to topologically order the vertices.
Lemma

A directed graph $G$ can be topologically ordered iff it is a DAG.

Continued.

$\Leftarrow$: Consider the following algorithm:

1. Pick a source $u$, output it.
2. Remove $u$ and all edges out of $u$.
3. Repeat until graph is empty.
4. Exercise: prove this gives an ordering.

Exercise: show above algorithm can be implemented in $O(m + n)$ 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 $G$ may have many different topological sorts.

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

**Question:** What is a DAG with the least number of distinct topological sorts for a given number $n$ of vertices?
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 $\text{DFS}(G)$
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 $\text{DFS}(G)$.

**Proposition**

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

... to check for Acyclicitiy and compute Topological Ordering

**Question**

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

**DFS based algorithm:**

1. Compute $\text{DFS}(G)$
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 $\text{DFS}(G)$.

**Proposition**

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

Proposition

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

Proof.

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

- **Case 1:** $[\text{pre}(u), \text{post}(u)]$ is contained in $[\text{pre}(v), \text{post}(v)]$. Implies that $u$ is explored during $\text{DFS}(v)$ and hence is a descendent of $v$. Edge $(u, v)$ implies a cycle in $G$ but $G$ is assumed to be DAG!

- **Case 2:** $[\text{pre}(u), \text{post}(u)]$ is disjoint from $[\text{pre}(v), \text{post}(v)]$. This cannot happen since $v$ would be explored from $u$. 

Example

\begin{figure}
\centering
\begin{tikzpicture}
  \node[shape=circle,draw=black] (A) at (0,0) {1};
  \node[shape=circle,draw=black] (B) at (1,1) {2};
  \node[shape=circle,draw=black] (C) at (1,2) {3};
  \node[shape=circle,draw=black] (D) at (0,1) {4};

  \draw[->] (A) to (B);
  \draw[->] (A) to (D);
  \draw[->] (A) to (C);
  \draw[->] (B) to (C);
  \draw[->] (C) to (D);

\end{tikzpicture}
\end{figure}
**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 descendants 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.
Proposition

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

Proof.

If: $(u, v)$ is a back edge implies there is a cycle $C$ consisting of the path from $v$ to $u$ in $\text{DFS}$ 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 $\text{DFS}$. All other nodes in $C$ are descendants 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.
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...
A partially ordered set is a set $S$ along with a binary relation $\leq$ such that $\leq$ is

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

Example: For numbers in the plane define $(x, y) \leq (x', y')$ 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.
A partially ordered set is a set $S$ along with a binary relation $\leq$ such that $\leq$ is

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

Example: For numbers in the plane define $(x, y) \leq (x', y')$ 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.
A partially ordered set is a set $S$ along with a binary relation $\leq$ such that $\leq$ is

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

Example: For numbers in the plane define $(x, y) \leq (x', y')$ 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.
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 $G = (V, E)$, output all its strong connected components.

Straightforward algorithm:
Mark all vertices in $V$ as not visited.
for each vertex $u \in V$ 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 $DFS(G^{rev}, u)$
$SCC(G, u) \leftarrow rch(G, u) \cap rch(G^{rev}, u)$
$\forall u \in SCC(G, u)$: 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 $u \in V$ 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 $DFS(G^{rev}, u)$

$SCC(G, u) \leftarrow rch(G, u) \cap rch(G^{rev}, u)$

$\forall u \in SCC(G, u)$: 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 $u \in V$ not visited yet do

find $\text{SCC}(G, u)$ the strong component of $u$:

Compute $\text{rch}(G, u)$ using $\text{DFS}(G, u)$

Compute $\text{rch}(G^{\text{rev}}, u)$ using $\text{DFS}(G^{\text{rev}}, u)$

$\text{SCC}(G, u) \leftarrow \text{rch}(G, u) \cap \text{rch}(G^{\text{rev}}, u)$

$\forall u \in \text{SCC}(G, u)$: 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 $\text{DFS}(u)$ to compute $\text{SCC}(u)$
3. Remove $\text{SCC}(u)$ and repeat

Justification

1. $\text{DFS}(u)$ only visits vertices (and edges) in $\text{SCC}(u)$
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 $\text{DFS}(u)$ to compute $\text{SCC}(u)$
3. Remove $\text{SCC}(u)$ and repeat

**Justification**

1. $\text{DFS}(u)$ only visits vertices (and edges) in $\text{SCC}(u)$
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 \( G^{\text{SCC}} \)
2. Do \( \text{DFS}(u) \) to compute \( \text{SCC}(u) \)
3. Remove \( \text{SCC}(u) \) and repeat

Justification

1. \( \text{DFS}(u) \) only visits vertices (and edges) in \( \text{SCC}(u) \)
2. ... since there are no edges coming out a sink!
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 $\text{DFS}(u)$ to compute $\text{SCC}(u)$
3. Remove $\text{SCC}(u)$ and repeat

### Justification

1. $\text{DFS}(u)$ only visits vertices (and edges) in $\text{SCC}(u)$
2. ... since there are no edges coming out a sink!
3. $\text{DFS}(u)$ takes time proportional to size of $\text{SCC}(u)$
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 \( O(n + m) \)!
How do we find a vertex in a sink \(\text{SCC}\) of \(G^{\text{SCC}}\)?

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

**Answer:** \(\text{DFS}(G)\) gives some information!
Big Challenge(s)

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

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

Answer: $\text{DFS}(G)$ gives some information!
How do we find a vertex in a sink $\text{SCC}$ of $G^{\text{SCC}}$?

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

Answer: $\text{DFS}(G)$ 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 $\text{post}$ numbers are with respect to some $\text{DFS}(G)$.
Graph $G$

Graph with pre-post times for $\text{DFS}(A)$; black edges in tree

Figure: $G^\text{SCC}$ with post times
Proposition

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

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 $\text{DFS}(u)$ 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 $G^\text{SCC}$ then for every $u \in S$ and $u' \in S'$, $\text{post}(u) > \text{post}(u')$. 
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* $post(S) > post(S')$.

**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 $u' \in S'$, $post(u) > post(u')$. 
Topological ordering of the strong components

Corollary

Ordering SCCs in decreasing order of 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 $G^{SCC}$!
Corollary

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

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

So...
\( \text{DFS}(G) \) gives some information on topological ordering of \( G^{\text{SCC}} \)!
**Finding Sources**

### Proposition

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

### Proof.

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

Proposition

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

Proof.

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

**Proposition**

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

**Proof.**

1. $u$ belongs to source SCC of $G^{\text{rev}}$

2. Since graph of SCCs of $G^{\text{rev}}$ is the reverse of $G^{\text{SCC}}$, $\text{SCC}(u)$ is sink SCC of $G$.  

□
Linear Time Algorithm

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

do $\text{DFS}(G^{\text{rev}})$ 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
    $\text{DFS}(u)$
    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 $G$:

$G$

Reverse graph $G^{\text{rev}}$:

Pre/Post DFS numbering of reverse graph:

$[7, 12] [1, 6] [9, 10] [8, 11] [2, 5] [13, 16] [14, 15]$

DFS 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.

$\text{SCC computed: } \{G\}$
Linear Time Algorithm: An Example

Removing connected components: 2

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

SCC computed:
{G}

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.

Do DFS from vertex $B$, remove it.

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

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

SCC computed: 
$$\{G\}, \{H\}, \{F, B, E\}, \{A, C, D\}$$
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 \( \text{post}(u_i) = \text{post}(S_i) = \max_{v \in S_i} \text{post}(v) \).
4. 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 \( G \).
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 III

An Application to make
Unix utility for automatically building large software applications

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

main.c
  \rightarrow main.o
  \leftarrow utils.c
  \rightarrow utils.o
  \rightarrow project
  \leftarrow defs.h
  \rightarrow utils.o
  \leftarrow command.h
  \rightarrow command.o
  \leftarrow command.c
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).