Chapter 2

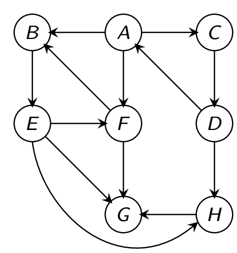
DFS in Directed Graphs, Strong Connected Components, and DAGs

CS 473: Fundamental Algorithms, Fall 2011
August 25, 2011

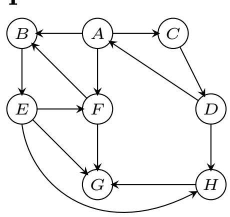
2.0.0.1 Strong Connected Components (SCCs)

Algorithmic Problem
Find all SCCs of a given directed graph.

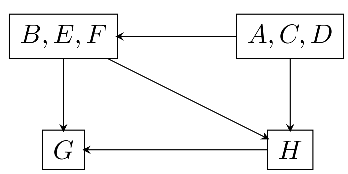
Previous lecture:
Saw an $O(n \cdot (n+m))$ time algorithm.
This lecture: $O(n+m)$ time algorithm.



2.0.0.2 Graph of SCCs



Graph G



Graph of SCCs G^{SCC}

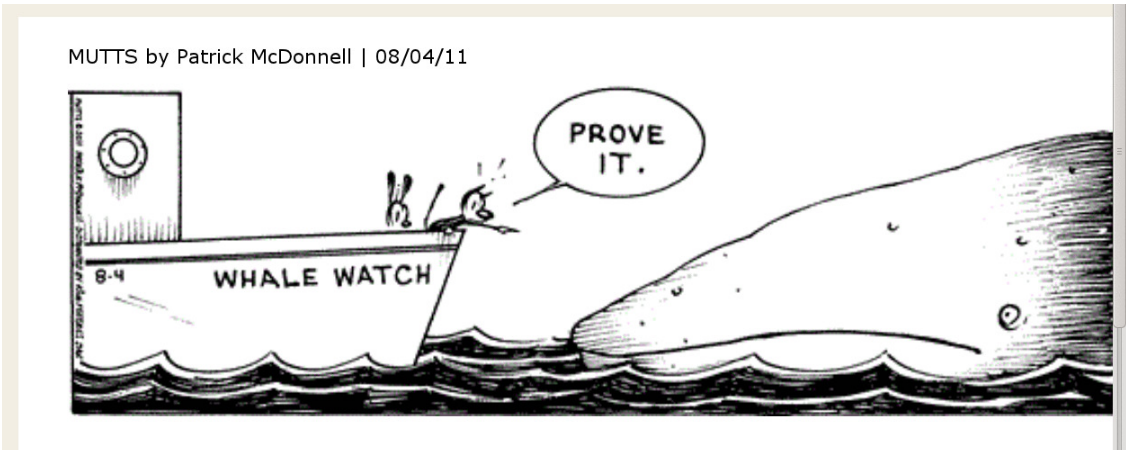
Meta-graph of SCCs
Let $S_1, S_2, \ldots S_k$ be the strong connected components (i.e., SCCs) of G. The graph of SCCs is G^{SCC}

(A) Vertices are $S_1, S_2, \ldots S_k$
(B) 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.

2.0.0.3 Reversal and SCCs

Proposition 2.0.1 *For any graph* G*, the graph of* SCC*s of* G^{rev} *is the same as the reversal of* $\mathsf{G}^{\mathrm{SCC}}$*.*

Proof: Exercise. ∎



2.0.0.4 SCCs and DAGs

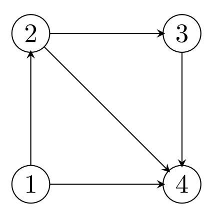
Proposition 2.0.2 *For any graph* G*, the graph* $\mathsf{G}^{\mathrm{SCC}}$ *has no directed cycle.*

Proof: If $\mathsf{G}^{\mathrm{SCC}}$ has a cycle $S_1, S_2, \ldots, S_k$ then $S_1 \cup S_2 \cup \cdots \cup S_k$ is an SCC in G. Formal details: exercise. ∎

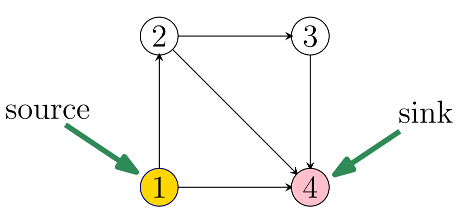
2.1 Directed Acyclic Graphs

2.1.0.5 Directed Acyclic Graphs

Definition 2.1.1 *A directed graph* G *is a* **directed acyclic graph** *(*DAG*) if there is no directed cycle in* G*.*



2.1.0.6 Sources and Sinks

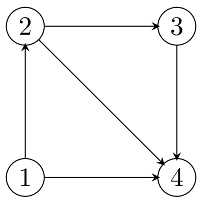


Definition 2.1.2 *(A) A vertex* u *is a* **source** *if it has no in-coming edges.*
(B) A vertex u *is a* **sink** *if it has no out-going edges.*

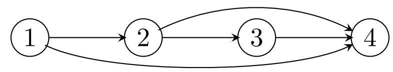
2.1.0.7 Simple DAG Properties

(A) Every DAG G has at least one source and at least one sink.
(B) If G is a DAG if and only if G^{rev} is a DAG.
(C) G is a DAG if and only each node is in its own strong connected component.
Formal proofs: exercise.

2.1.0.8 Topological Ordering/Sorting



Graph G



Topological Ordering of G

Definition 2.1.3 *A* **topological ordering/topological sorting** *of $G = (V, E)$ is an ordering $<$ 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.

2.1.0.9 DAGs and Topological Sort

Lemma 2.1.4 *A directed graph G can be topologically ordered iff it is a DAG.*

Proof: $\Longrightarrow$: 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. ■

2.1.0.10 DAGs and Topological Sort

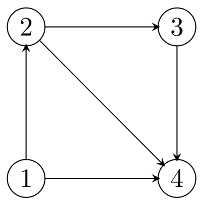
Lemma 2.1.5 *A directed graph G can be topologically ordered iff it is a DAG.*

Proof:[Continued] $\Leftarrow$: Consider the following algorithm:
(A) Pick a source u, output it.
(B) Remove u and all edges out of u.
(C) Repeat until graph is empty.
(D) Exercise: prove this gives an ordering.

■

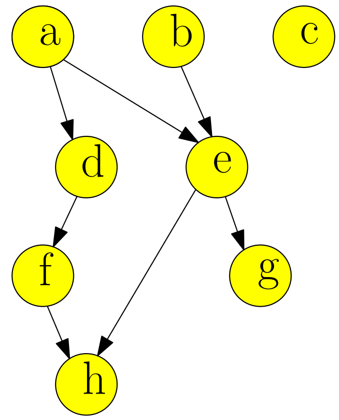
Exercise: show above algorithm can be implemented in $O(m + n)$ time.

2.1.0.11 Topological Sort: An Example



Output: 1 2 3 4

2.1.0.12 Topological Sort: Another Example



2.1.0.13 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?

2.1.1 Using DFS...

2.1.1.1 ... 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:
(A) Compute **DFS**(G)
(B) If there is a back edge then G is not a DAG.
(C) Otherwise output nodes in decreasing post-visit order.
Correctness relies on the following:

Proposition 2.1.6 *G is a DAG iff there is no back-edge in* **DFS**(G).

Proposition 2.1.7 *If G is a DAG and* $\text{post}(v) > \text{post}(u)$, *then* (u, v) *is not in G.*

Proof: There are several possibilities:
(A) $[\text{pre}(v), \text{post}(v)]$ comes after $[\text{pre}(u), \text{post}(u)]$ and they are disjoint. But then, u was visited first by the **DFS**, if $(u, v) \in E(G)$ then **DFS** will visit v during the recursive call on u. But then, $\text{post}(v) < \text{post}(u)$. A contradiction.

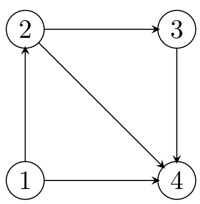
(B) $[\text{pre}(v), \text{post}(v)] \subseteq [\text{pre}(u), \text{post}(u)]$: impossible as $\text{post}(v) > \text{post}(u)$.

(C) $[\text{pre}(u), \text{post}(u)] \subseteq [\text{pre}(v), \text{post}(v)]$. But then **DFS** visited v, and then visited u. Namely there is a path in G from v to u. But then if $(u, v) \in E(G)$ then there would be a cycle in G, and it would not be a DAG. Contradiction.

(D) No other possibility - since "lifetime" intervals of **DFS** are either disjoint or contained in each other.

■

2.1.1.2 Example



2.1.1.3 Back edge and Cycles

Proposition 2.1.8 *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 $C = v_1 \to v_2 \to \ldots \to v_k \to v_1$.

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. ■

2.1.1.4 DAGs and Partial Orders

Definition 2.1.9 *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 $b \npreceq a$), 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.

2.1.2 What's DAG but a sweet old fashioned notion

2.1.2.1 Who needs a DAG...

Example

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

2.1.3 What DAGs got to do with it?

2.1.3.1 Or why we should care about DAGs

(A) DAGs enable us to represent partial ordering information we have about some set (very common situation in the real world).
(B) Questions about DAGs:
  (A) Is a graph G a DAG?
  $\iff$
  Is the partial ordering information we have so far is consistent?
  (B) Compute a topological ordering of a DAG.
  $\iff$
  Find an a consistent ordering that agrees with our partial information.
  (C) 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.

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

2.2.0.2 Finding all SCCs of a Directed Graph

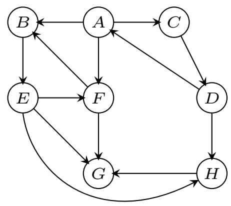
Problem
Given a directed graph $G = (V, E)$, output *all* its strong connected components.

Straightforward algorithm:

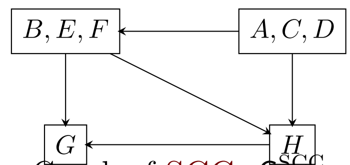
```
Mark all vertices in V as not visited.
for each vertex u ∈ V not visited yet do
    find SCC(G, u) the strong component of u:
        Compute rch(G, u) using DFS(G, u)
        Compute rch(G^rev, u) using DFS(G^rev, u)
        SCC(G, u) ⇐ rch(G, u) ∩ rch(G^rev, u)
        ∀u ∈ SCC(G, u): Mark u as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

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

2.2.0.3 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 2.2.1 *For a directed graph G, its meta-graph* G^{SCC} *is a DAG.*

2.2.1 Linear-time Algorithm for SCCs: Ideas

2.2.1.1 Exploit structure of meta-graph...

Wishful Thinking Algorithm

(A) Let u be a vertex in a sink SCC of G^{SCC}
(B) Do **DFS**(u) to compute SCC(u)
(C) Remove SCC(u) and repeat

Justification

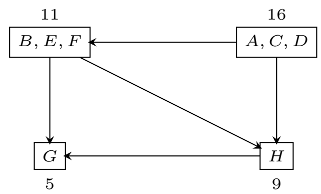
(A) **DFS**(u) only visits vertices (and edges) in SCC(u)
(B) **DFS** done only in G (not in G^{rev}) to compute u strong connected component (SCC). [Magic!]
(C) **DFS**(u) takes time proportional to size of SCC(u)
(D) Therefore, total time $O(n+m)$!

2.2.1.2 Big Challenge(s)

How do we find a vertex in the 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!

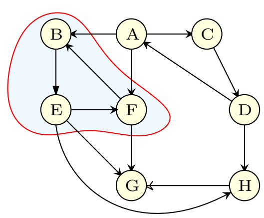


G^{SCC} with post times

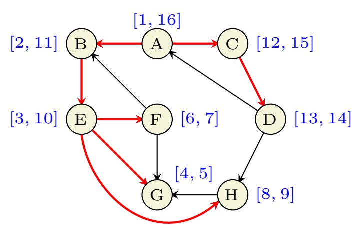
2.2.1.3 Post-visit times of SCCs

Definition 2.2.2 *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)*.*

2.2.1.4 An Example



Graph G



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

2.2.2 Graph of strong connected components

2.2.2.1 ... and post-visit times

Proposition 2.2.3 *If* S *and* S' *are* SCC*s in* G *and* (S, S') *is an edge in* G^{SCC} *then* $\text{post}(S) > \text{post}(S')$*.*

Proof: Let u be first vertex in $S \cup S'$ that is visited.
(A) If $u \in S$ then all of S' will be explored before **DFS**(u) completes.
(B) 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 $u' \in S'$, $\text{post}(u) > \text{post}(u')$.

2.2.2.2 Topological ordering of the strong components

Corollary 2.2.4 *Ordering* SCC*s in decreasing order of* $\text{post}(S)$ *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}!

2.2.2.3 Finding Sources

Proposition 2.2.5 *The vertex u with the highest post visit time belongs to a source* SCC *in* G^{SCC}

Proof:¡2-¿
(A) $\mathrm{post}(\mathrm{SCC}(u)) = \mathrm{post}(u)$
(B) Thus, $\mathrm{post}(\mathrm{SCC}(u))$ is highest and will be output first in topological ordering of G^{SCC}.

■

2.2.2.4 Finding Sinks

Proposition 2.2.6 *The vertex u with highest post visit time in* **DFS**(G^{rev}) *belongs to a sink SCC of* G.

Proof:¡2-¿
(A) u belongs to source SCC of G^{rev}
(B) Since graph of SCCs of G^{rev} is the reverse of $\mathsf{G}^{\mathrm{SCC}}$, $\mathrm{SCC}(u)$ is sink SCC of G.

■

2.2.3 Linear Time Algorithm

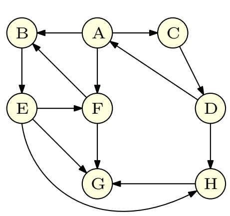
2.2.3.1 ...for computing the strong connected components in G

```
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)

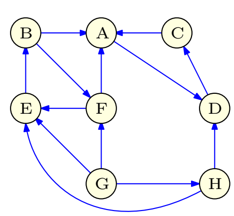
2.2.3.2 Linear Time Algorithm: An Example - Initial steps

Graph G:



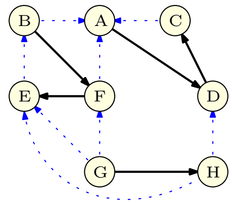
$\Longrightarrow$

Reverse graph G^{rev}:



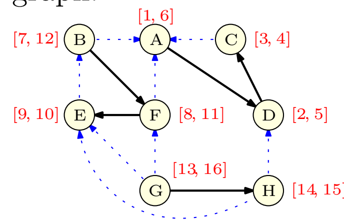
DFS of reverse graph:

$\Longrightarrow$



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

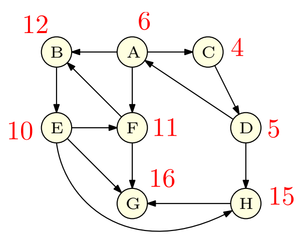
$\Longrightarrow$



2.2.4 Linear Time Algorithm: An Example

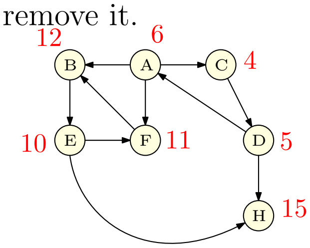
2.2.4.1 Removing connected components: 1

Original graph G with rev post numbers:



$\Longrightarrow$

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

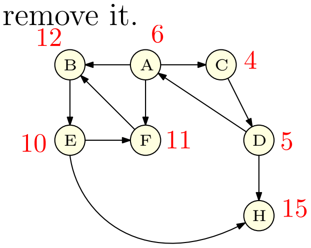


SCC computed:
$\{G\}$

2.2.5 Linear Time Algorithm: An Example

2.2.5.1 Removing connected components: 2

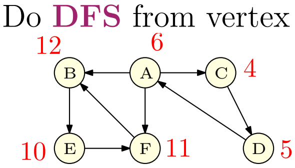
Do **DFS** from vertex G remove it.



SCC computed:
$\{G\}$

$\Longrightarrow$

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

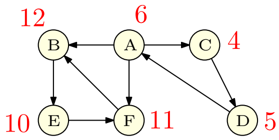


SCC computed:
$\{G\}, \{H\}$

2.2.6 Linear Time Algorithm: An Example

2.2.6.1 Removing connected components: 3

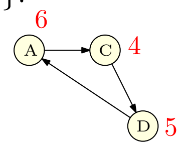
Do **DFS** from vertex H, remove it.



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

$\Longrightarrow$

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

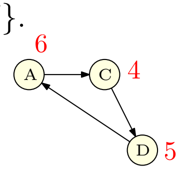


SCC computed:
$\{G\}, \{H\}, \{F, B, E\}$

2.2.7 Linear Time Algorithm: An Example

2.2.7.1 Removing connected components: 4

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



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

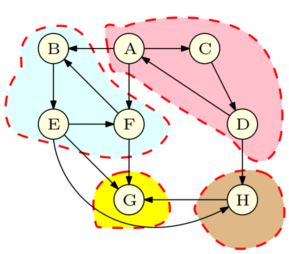
$\Longrightarrow$

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

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

2.2.8 Linear Time Algorithm: An Example

2.2.8.1 Final result



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

Which is the correct answer!

2.2.9 Obtaining the meta-graph...

2.2.9.1 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 $\mathsf{G}^{\mathrm{SCC}}$ can be obtained in $O(m + n)$ time.

2.2.9.2 Correctness: more details

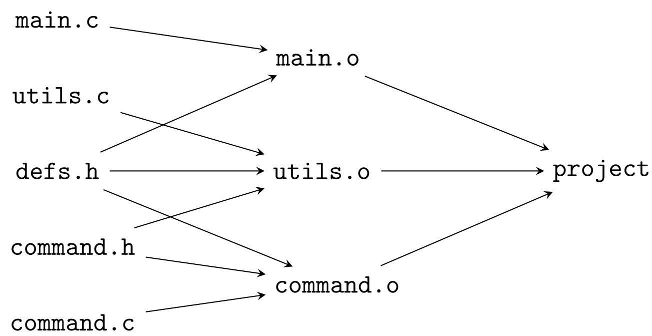
(A) let $S_1, S_2, \ldots, S_k$ be strong components in G
(B) Strong components of G^{rev} and G are same and meta-graph of G is reverse of meta-graph of G^{rev}.
(C) consider **DFS**(G^{rev}) and let $u_1, u_2, \ldots, u_k$ be such that $\mathrm{post}(u_i) = \mathrm{post}(S_i) = \max_{v \in S_i} \mathrm{post}(v)$.
(D) 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.
(E) u_k has highest post number and **DFS**(u_k) will explore all of S_k which is a sink component in G.
(F) 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.

2.3 An Application to `make`

2.3.1 `make` utility

2.3.1.1 `make` Utility [Feldman]

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



2.3.1.2 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
```

2.3.1.3 `makefile` as a Digraph

2.3.2 Computational Problems

2.3.2.1 Computational Problems for `make`

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

2.3.2.2 Algorithms for `make`

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

2.3.2.3 Take away Points

(A) Given a directed graph G, its SCCs and the associated acyclic meta-graph $\mathsf{G}^{\mathrm{SCC}}$ give a structural decomposition of G that should be kept in mind.

(B) There is a **DFS** based linear time algorithm to compute all the SCCs and the meta-graph. Properties of **DFS** crucial for the algorithm.

(C) 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).