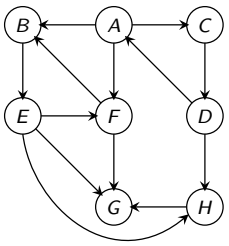
Chapter 2

DFS in Directed Graphs, Strong Connected Components, DAGs

CS 473: Fundamental Algorithms, Spring 2011
January 20, 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

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

- Vertices are $S_1, S_2, \ldots S_k$
- 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.

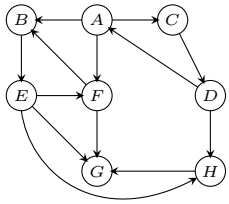


Figure 2.1: Graph G

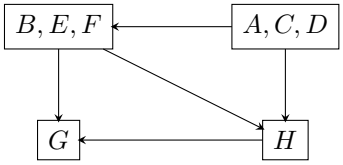


Figure 2.2: Graph of SCCs G^{SCC}

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 G^{SCC}.*

Proof: Exercise. ■

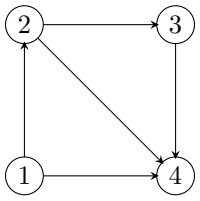
2.0.0.4 SCCs and DAGs

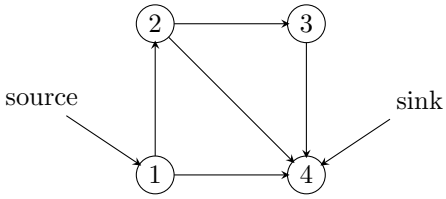
Proposition 2.0.2 *For any graph G, the graph G^{SCC} has no directed cycle.*

Proof: If G^{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. ■

Part I

Directed Acyclic Graphs





2.0.0.5 Directed Acyclic Graphs

Definition 2.0.3 *A directed graph G is a* **directed acyclic graph** *(*DAG*) if there is no directed cycle in G.*

2.0.0.6 Sources and Sinks

Definition 2.0.4

- *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.*

2.0.0.7 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 G^{rev} is a DAG.
- G is a DAG if and only each node is in its own strong connected component.

Formal proofs: exercise.

2.0.0.8 Topological Ordering/Sorting

Definition 2.0.5 *A* **topological ordering/topological sorting** *of $G = (V, E)$ is an ordering $<$ on V such that if $(u, v) \in E$ then $u < v$.*

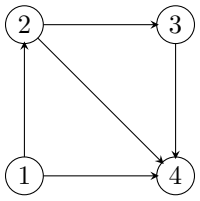


Figure 2.3: Graph G

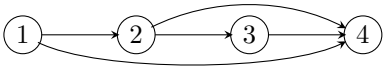


Figure 2.4: Topological Ordering of G

2.0.0.9 DAGs and Topological Sort

Lemma 2.0.6 *A directed graph G can be topologically ordered iff it is a* DAG.

Proof: Only if: Suppose G is not a DAG and has a topological ordering $<$. G has a cycle $C = u_1, u_2, \ldots, u_k, u_1$.
Then $u_1 < u_2 < \ldots < u_k < u_1$! A contradiction. ■

Proof: If: Consider the following algorithm:

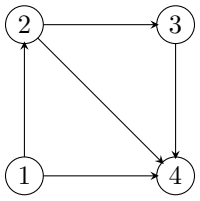
- Pick a source u, output it.
- Remove u and all edges out of u.
- Repeat until graph is empty.
- Exercise: prove this gives an ordering.

■

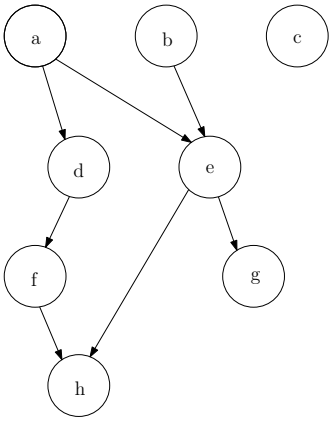
Exercise: show above algorithm can be implemented in $O(m + n)$ time.

2.0.0.10 Topological Sort: An Example

Output: 1 2 3 4



2.0.0.11 Topological Sort: Another Example



2.0.0.12 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?

2.0.1 Using DFS...

2.0.1.1 ... 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:

- Compute DFS(G)

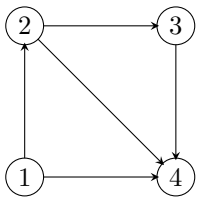
- If there is a back edge then G is not a DAG.
- Otherwise output nodes in decreasing post-visit order.

Correctness relies on the following:

Proposition 2.0.7 *G is a DAG iff there is no back-edge in **DFS**(G).*

Proposition 2.0.8 *If G is a DAG and post(v) > post(u), then (u, v) is not in G.*

2.0.1.2 Example



2.0.1.3 Back edge and Cycles

Proposition 2.0.9 *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 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. ■

2.0.1.4 DAGs and Partial Orders

Definition 2.0.10 *A **partially ordered set** is a set S along with a binary relation $\preceq$ such that $\preceq$ is*

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

Example: For numbers in the plane define $(x, y) \preceq (x', y')$ iff $x \leq x'$ and $y \leq y'$.
Observation: A *finite* partially ordered set is equivalent to a DAG.

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

2.0.1.5 Finding all SCCs of a Directed Graph

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

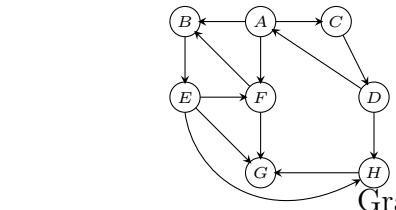
Straightforward algorithm:

```
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)
```

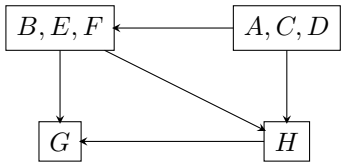
Running time: $O(n(n+m))$

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

2.0.1.6 Structure of a Directed Graph



Graph G



Graph of SCCs G^{SCC}

Proposition 2.0.11 *For a directed graph G, its meta-graph G^{SCC} is a DAG.*

2.0.1.7 Linear-time Algorithm for SCCs: Ideas

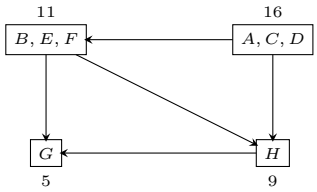
Exploit structure of meta-graph.

Algorithm

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

Justification

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



G^{SCC} with post times

2.0.1.8 Big Challenge(s)

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

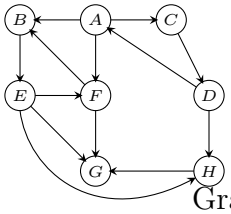
Can we obtain an *implicit* topological sort of G^{SCC} without computing G^{SCC}?

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

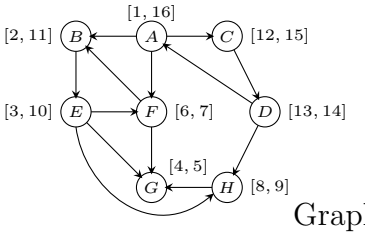
2.0.1.9 Post-visit times of SCCs

Definition 2.0.12 *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).

2.0.1.10 An Example



Graph G



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

2.0.1.11 G^{SCC} and post-visit times

Proposition 2.0.13 *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.

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

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

2.0.1.12 Topological ordering of G^{SCC}

Corollary 2.0.14 *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}!

2.0.1.13 Finding Sources

Proposition 2.0.15 *The vertex* u *with the highest post visit time belongs to a source* SCC *in* G^{SCC}

Proof :¡2-¿

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

■

2.0.1.14 Finding Sinks

Proposition 2.0.16 *The vertex* u *with highest post visit time in* **DFS**(G^{rev}) *belongs to a sink SCC of* G.

Proof :¡2-¿

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

■

2.0.1.15 Linear Time Algorithm

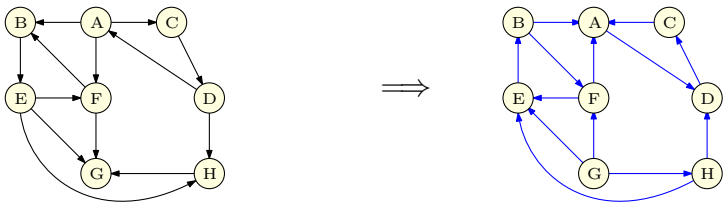
```
Do DFS(G^rev) and sort vertices in decreasing post order.
Mark all nodes as unvisited
for each u in the computed order do
    if u is not visited then
        DFS(u)
        Let S_u be the nodes reached by u
        Output S_u as a strong connected component
        Remove S_u from G
```

Analysis

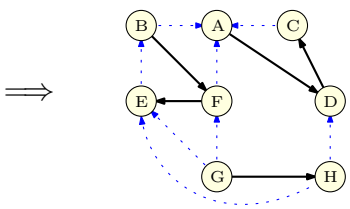
Running time is $O(n + m)$. (Exercise)

2.0.1.16 Linear Time Algorithm: An Example - Initial steps

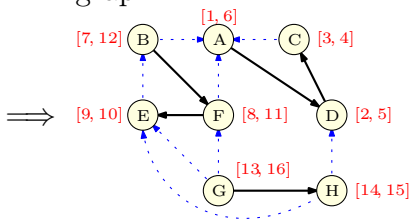
Graph G: Reverse graph G^{rev}:



DFS of reverse graph:



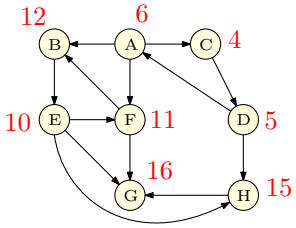
Pre/Post **DFS** numbering of reverse graph:



2.0.2 Linear Time Algorithm: An Example

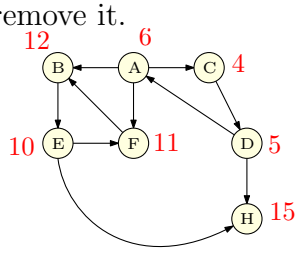
2.0.2.1 Removing connected components: 1

Original graph G with rev post numbers:



$\Longrightarrow$

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

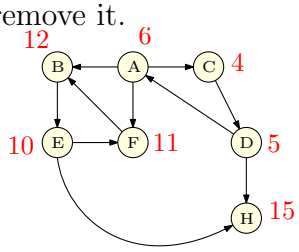


SCC computed:
$\{G\}$

2.0.3 Linear Time Algorithm: An Example

2.0.3.1 Removing connected components: 2

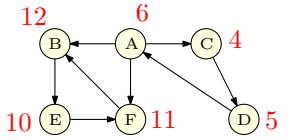
Do **DFS** from vertex G remove it.



SCC computed:
$\{G\}$

$\Longrightarrow$

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

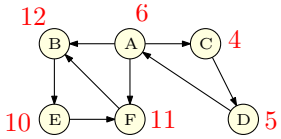


SCC computed:
$\{G\}, \{H\}$

2.0.4 Linear Time Algorithm: An Example

2.0.4.1 Removing connected components: 3

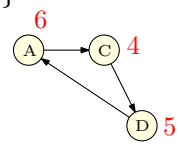
Do **DFS** from vertex H, remove it.



$\Longrightarrow$

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

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

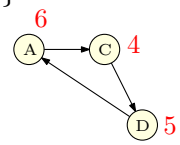


SCC computed:
$\{G\}, \{H\}, \{F, B, E\}$

2.0.5 Linear Time Algorithm: An Example

2.0.5.1 Removing connected components: 4

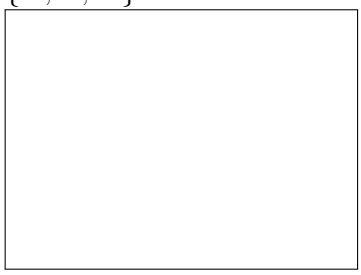
Do **DFS** from vertex F
Remove visited vertices:
$\{F, B, E\}$.



$\Longrightarrow$

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

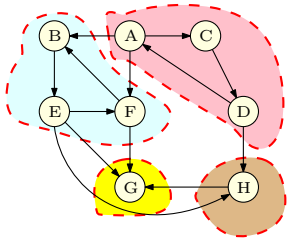
Do **DFS** from vertex A
Remove visited vertices:
$\{A, C, D\}$.



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

2.0.6 Linear Time Algorithm: An Example

2.0.6.1 Final result



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

Which is the correct answer!

2.0.6.2 Obtaining the meta-graph from strong connected components

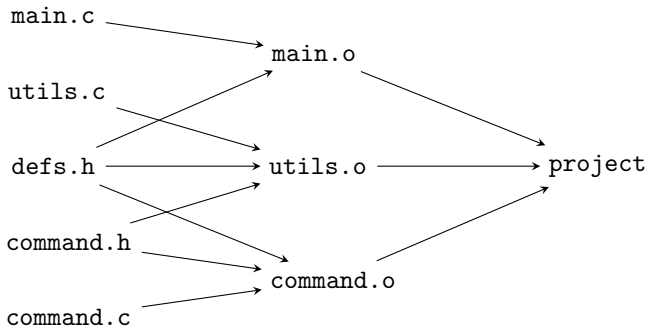
Exercise: Given all the strong connected components of a directed graph $G = (V, E)$ show that the meta-graph G^{SCC} can be obtained in $O(m + n)$ time.

2.0.6.3 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 **DFS**(G^{rev}) and let $u_1, u_2, \ldots, u_k$ be such that $\text{post}(u_i) = \text{post}(S_i) = \max_{v \in S_i} \text{post}(v)$.
- 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.
- 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`



2.0.7 `make` utility

2.0.7.1 `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

2.0.7.2 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
```

2.0.7.3 `makefile` as a Digraph

2.0.8 Computational Problems

2.0.8.1 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.

2.0.8.2 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.*

2.0.8.3 Take away Points

- 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.
- 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).