

Directed Graphs and DFS

Lecture 16

October 23, 2018

Topics

- Structure of directed graphs
- **DFS** and its properties
- One application of **DFS** to obtain fast algorithms

Strong Connected Components (SCCs)

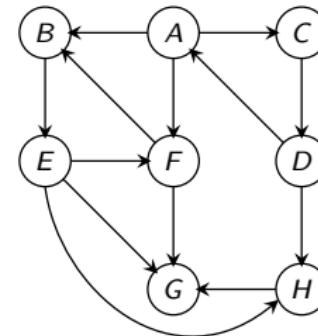
Algorithmic Problem

Find all **SCCs** of a given directed graph.

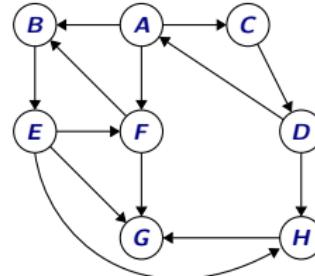
Previous lecture:

Saw an $O(n \cdot (n + m))$ time algorithm.

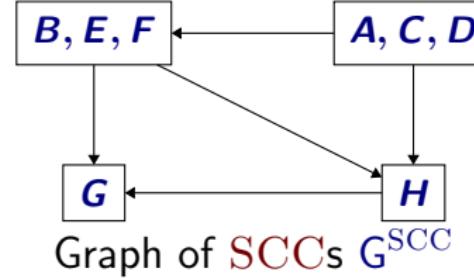
This lecture: sketch of a $O(n + m)$ time algorithm.



Graph of SCCs



Graph G



Graph of SCCs G^{SCC}

Meta-graph of SCCs

Let S_1, S_2, \dots, 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, \dots, 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 .
- G^{SCC} cannot have cycles (acyclic)

do not have strongly connected components

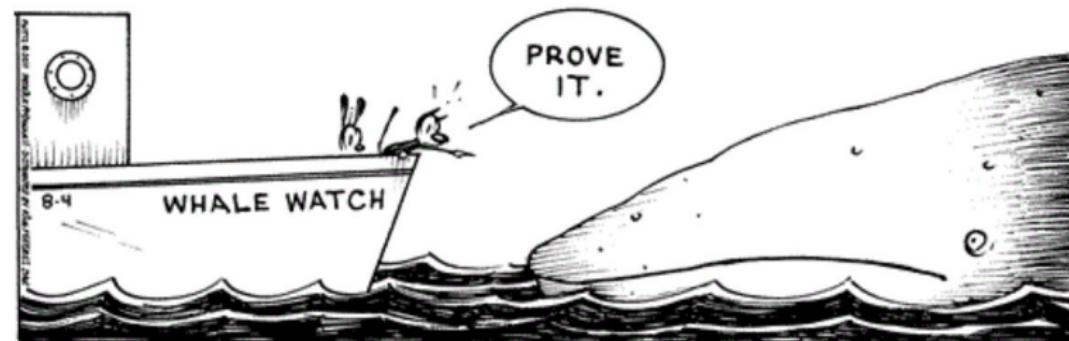
Proposition

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

Proof.

Exercise. □

MUTTS by Patrick McDonnell | 08/04/11



Proposition

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

Proof.

If G^{SCC} has a cycle S_1, S_2, \dots, S_k then $S_1 \cup S_2 \cup \dots \cup S_k$ should be in the same SCC in G . Formal details: exercise. \square

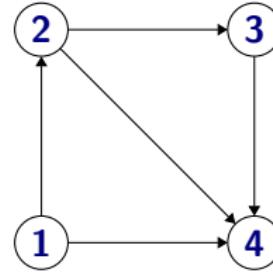
Part I

Directed Acyclic Graphs

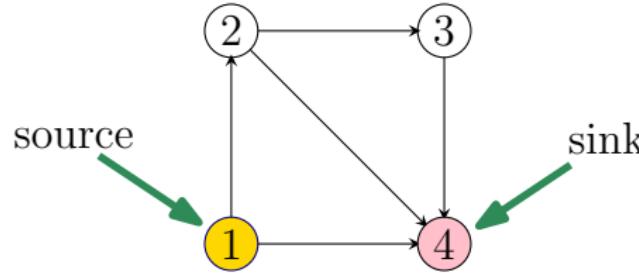
Directed Acyclic Graphs

Definition

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



Sources and Sinks



Definition

- 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.

Proposition

Every DAG G has at least one source and at least one sink.

$$\textcircled{0} \rightarrow \textcircled{0}$$

$$\textcircled{0}$$

Proposition

Every DAG G has at least one source and at least one sink.

Proof.

Let $P = v_1, v_2, \dots, v_k$ be a longest path in G . Claim that v_1 is a source and v_k is a sink. Suppose not. Then v_1 has an incoming edge which either creates a cycle or a longer path both of which are contradictions. Similarly if v_k has an outgoing edge. \square

Proposition

Every DAG G has at least one source and at least one sink.

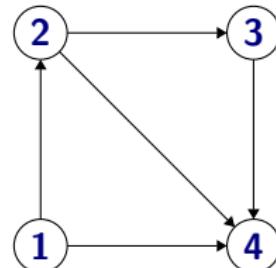
Proof.

Let $P = v_1, v_2, \dots, v_k$ be a longest path in G . Claim that v_1 is a source and v_k is a sink. Suppose not. Then v_1 has an incoming edge which either creates a cycle or a longer path both of which are contradictions. Similarly if v_k has an outgoing edge. \square

- ① G is a DAG if and only if G^{rev} is a DAG.
- ② G is a DAG if and only each node is in its own strong connected component.

Formal proofs: exercise.

Topological Ordering/Sorting



Graph G



Topological Ordering of G

Low number vertices

to high number vertices

Definition

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

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.

Need to show both directions. 

Lemma

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

Proof.

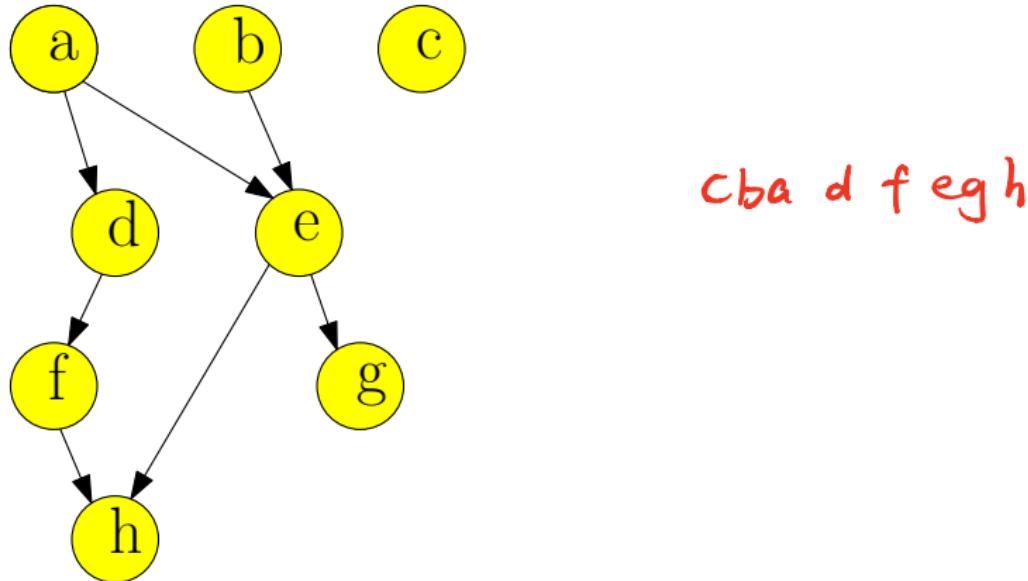
Consider the following algorithm:

- ① Pick a source u , output it.
- ② Remove u and all edges out of u . \Rightarrow it is safe since no edges coming back to u
- ③ Repeat until graph is empty.

Exercise: prove this gives toplogical sort. □

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

Topological Sort: Example



Lemma

A directed graph G can be topologically ordered only if it is a DAG.

Proof.

Suppose G is not a DAG and has a topological ordering \prec . G has a cycle $C = u_1, u_2, \dots, u_k, u_1$.

Then $u_1 \prec u_2 \prec \dots \prec u_k \prec u_1$!

That is... $u_1 \prec u_1$.

A contradiction (to \prec being an order).

Not possible to topologically order the vertices. □

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?

No edges

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

a path

Cycles in graphs

Question: Given an *undirected* graph how do we check whether it has a cycle and output one if it has one?

Spanning tree has nodes
not contained inside it

Question: Given an *directed* graph how do we check whether it has a cycle and output one if it has one?

No source in the graph

To Remember: Structure of Graphs

Undirected graph: connected components of $G = (V, E)$ partition V and can be computed in $O(m + n)$ time.

Directed graph: the meta-graph G^{SCC} of G can be computed in $O(m + n)$ time. G^{SCC} gives information on the partition of V into strong connected components and how they form a DAG structure.

Above structural decomposition will be useful in several algorithms

Part II

Depth First Search (DFS)

Depth First Search

DFS is a special case of Basic Search but is a versatile graph exploration strategy. John Hopcroft and Bob Tarjan (Turing Award winners) demonstrated the power of **DFS** to understand graph structure. **DFS** can be used to obtain linear time ($O(m + n)$) algorithms for

- ① Finding cut-edges and cut-vertices of undirected graphs
- ② Finding strong connected components of directed graphs
- ③ Linear time algorithm for testing whether a graph is planar

Many other applications as well.

*discovering the
structure of the
graph*

DFS in Undirected Graphs

Recursive version. Easier to understand some properties.

DFS(G)

```
for all  $u \in V(G)$  do
    Mark  $u$  as unvisited
    Set pred( $u$ ) to null
 $T$  is set to  $\emptyset$ 
while  $\exists$  unvisited  $u$  do
    DFS( $u$ )
Output  $T$ 
```

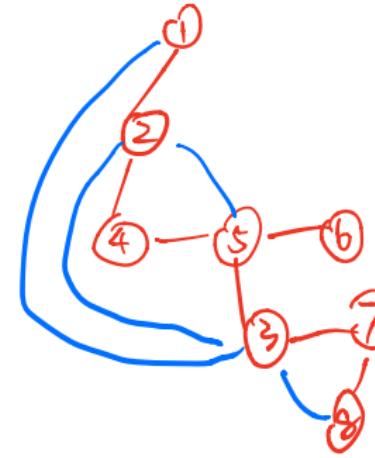
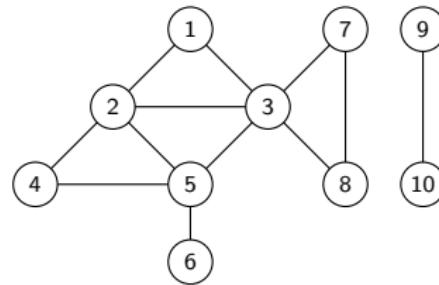
DFS(u)

```
Mark  $u$  as visited
for each  $uv$  in  $Out(u)$  do
    if  $v$  is not visited then
        add edge  $uv$  to  $T$ 
        set pred( $v$ ) to  $u$ 
        DFS( $v$ )
```

Implemented using a global array **Visited** for all recursive calls.

T is the search tree/forest.

Example



Edges classified into two types: $uv \in E$ is a

① tree edge: belongs to T

② non-tree edge: does not belong to T

from descendants to parent

Proposition

- ① T is a forest
- ② connected components of T are same as those of G .
- ③ If $uv \in E$ is a non-tree edge then, in T , either:
 - ① u is an ancestor of v , or
 - ② v is an ancestor of u .

Question: Why are there no cross-edges?

DFS with Visit Times

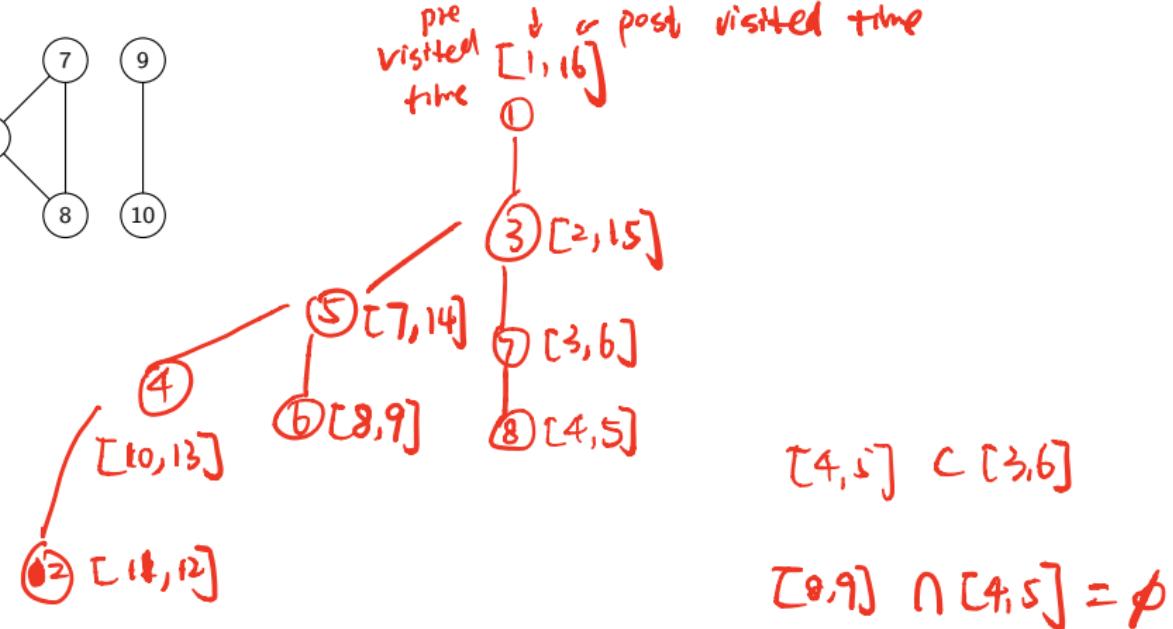
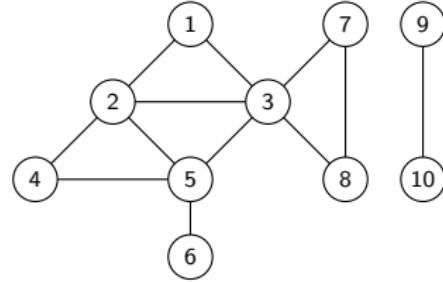
Keep track of when nodes are visited.

```
DFS( $G$ )
  for all  $u \in V(G)$  do
    Mark  $u$  as unvisited
     $T$  is set to  $\emptyset$ 
     $time = 0$ 
  while  $\exists$ unvisited  $u$  do
    DFS( $u$ )
  Output  $T$ 
```

```
DFS( $u$ )
  Mark  $u$  as visited
   $pre(u) = ++time$ 
  for each  $uv$  in  $Out(u)$  do
    if  $v$  is not marked then
      add edge  $uv$  to  $T$ 
      DFS( $v$ )
   $post(u) = ++time$ 
```

first visited \rightarrow last visited

Example



pre and post numbers

Node u is **active** in time interval $[\text{pre}(u), \text{post}(u)]$

Proposition

For any two nodes u and v , the two intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ are disjoint or one is contained in the other.

pre and post numbers

Node u is **active** in time interval $[\text{pre}(u), \text{post}(u)]$

Proposition

For any two nodes u and v , the two intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ are disjoint or one is contained in the other.

Proof.

pre and post numbers

Node u is **active** in time interval $[\text{pre}(u), \text{post}(u)]$

Proposition

For any two nodes u and v , the two intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ are disjoint or one is contained in the other.

Proof.

- Assume without loss of generality that $\text{pre}(u) < \text{pre}(v)$. Then v visited after u .

pre and post numbers

Node u is **active** in time interval $[\text{pre}(u), \text{post}(u)]$

Proposition

For any two nodes u and v , the two intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ are disjoint or one is contained in the other.

Proof.

- Assume without loss of generality that $\text{pre}(u) < \text{pre}(v)$. Then v visited after u .
- If $\text{DFS}(v)$ invoked before $\text{DFS}(u)$ finished,
 $\text{post}(v) < \text{post}(u)$.

pre and post numbers

Node u is **active** in time interval $[\text{pre}(u), \text{post}(u)]$

Proposition

For any two nodes u and v , the two intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ are disjoint or one is contained in the other.

Proof.

- Assume without loss of generality that $\text{pre}(u) < \text{pre}(v)$. Then v visited after u .
- If $\text{DFS}(v)$ invoked before $\text{DFS}(u)$ finished, $\text{post}(v) < \text{post}(u)$.
- If $\text{DFS}(v)$ invoked after $\text{DFS}(u)$ finished, $\text{pre}(v) > \text{post}(u)$. \square

pre and post numbers

Node u is **active** in time interval $[\text{pre}(u), \text{post}(u)]$

Proposition

For any two nodes u and v , the two intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ are disjoint or one is contained in the other.

Proof.

- Assume without loss of generality that $\text{pre}(u) < \text{pre}(v)$. Then v visited after u .
- If $\text{DFS}(v)$ invoked before $\text{DFS}(u)$ finished, $\text{post}(v) < \text{post}(u)$.
- If $\text{DFS}(v)$ invoked after $\text{DFS}(u)$ finished, $\text{pre}(v) > \text{post}(u)$. \square

pre and post numbers useful in several applications of **DFS**

DFS in Directed Graphs

DFS(G)

Mark all nodes u as unvisited

T is set to \emptyset

$time = 0$

while there is an unvisited node u **do**

DFS(u)

Output T

DFS(u)

Mark u as visited

$pre(u) = ++time$

for each edge (u, v) in $Out(u)$ **do**

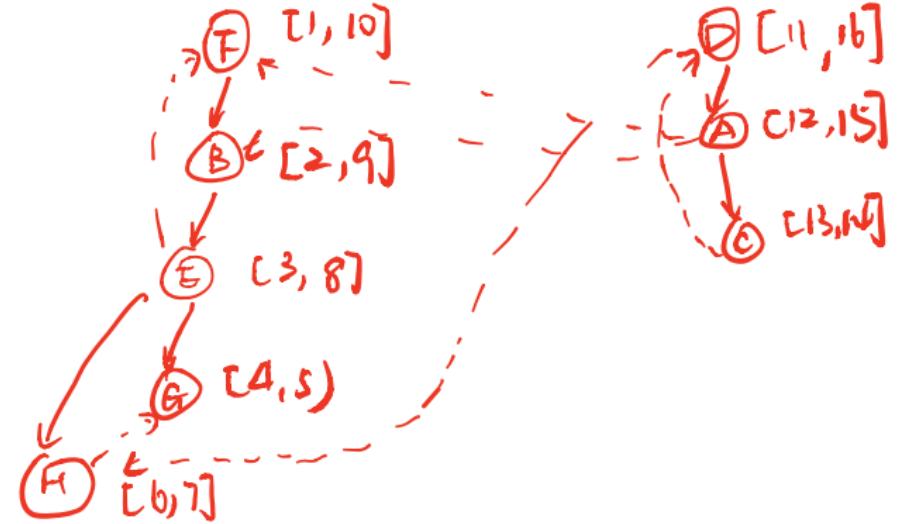
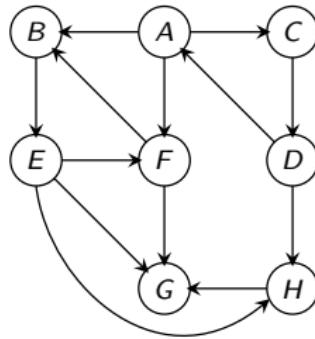
if v is not visited

 add edge (u, v) to T

DFS(v)

$post(u) = ++time$

Example



DFS Properties

Generalizing ideas from undirected graphs:

- ① **DFS(G)** takes $O(m + n)$ time.

DFS Properties

Generalizing ideas from undirected graphs:

- ① $\text{DFS}(G)$ takes $O(m + n)$ time.
- ② Edges added form a *branching*: a forest of out-trees. Output of $\text{DFS}(G)$ depends on the order in which vertices are considered.

DFS Properties

Generalizing ideas from undirected graphs:

- ① $\text{DFS}(G)$ takes $\underline{O(m + n)}$ time.
- ② Edges added form a *branching*: a forest of out-trees. Output of $\text{DFS}(G)$ depends on the order in which vertices are considered.
- ③ If u is the first vertex considered by $\text{DFS}(G)$ then $\text{DFS}(u)$ outputs a directed out-tree T rooted at u and a vertex v is in T if and only if $v \in \text{rch}(u)$

DFS Properties

Generalizing ideas from undirected graphs:

- ① $\text{DFS}(G)$ takes $O(m + n)$ time.
- ② Edges added form a *branching*: a forest of out-trees. Output of $\text{DFS}(G)$ depends on the order in which vertices are considered.
- ③ If u is the first vertex considered by $\text{DFS}(G)$ then $\text{DFS}(u)$ outputs a directed out-tree T rooted at u and a vertex v is in T if and only if $v \in \text{rch}(u)$
- ④ For any two vertices x, y the intervals $[\text{pre}(x), \text{post}(x)]$ and $[\text{pre}(y), \text{post}(y)]$ are either disjoint or one is contained in the other.

DFS Properties

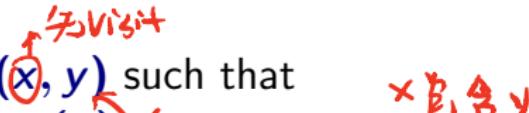
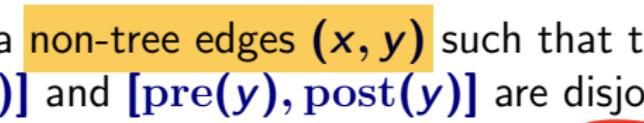
Generalizing ideas from undirected graphs:

- ① $\text{DFS}(G)$ takes $O(m + n)$ time.
- ② Edges added form a *branching*: a forest of out-trees. Output of $\text{DFS}(G)$ depends on the order in which vertices are considered.
- ③ If u is the first vertex considered by $\text{DFS}(G)$ then $\text{DFS}(u)$ outputs a directed out-tree T rooted at u and a vertex v is in T if and only if $v \in \text{rch}(u)$
- ④ For any two vertices x, y the intervals $[\text{pre}(x), \text{post}(x)]$ and $[\text{pre}(y), \text{post}(y)]$ are either disjoint or one is contained in the other.

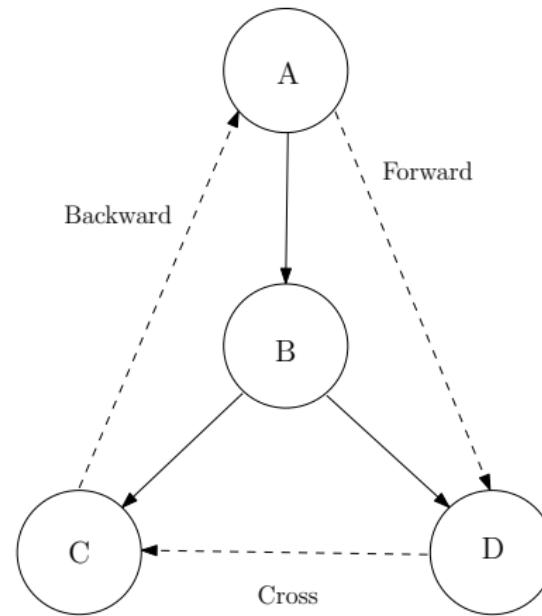
Note: Not obvious whether $\text{DFS}(G)$ is useful in dir graphs but it is.

DFS Tree

Edges of G can be classified with respect to the **DFS** tree T as:

- ① **Tree edges** that belong to T
- ② A **forward edge** is a non-tree edges (x, y) such that $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$. 
- ③ A **backward edge** is a non-tree edge (y, x) such that $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$. 
- ④ A **cross edge** is a non-tree edges (x, y) such that the intervals $[\text{pre}(x), \text{post}(x)]$ and $[\text{pre}(y), \text{post}(y)]$ are disjoint. 

Types of Edges



Cycles in graphs

Question: Given an *undirected* graph how do we check whether it has a cycle and output one if it has one?

non-tree edges

Question: Given an *directed* graph how do we check whether it has a cycle and output one if it has one?

doesn't have a source and a sink

Using DFS...

... to check for Acyclicity and compute Topological Ordering

Question

Given G , is it a DAG? If it is, generate a topological sort. Else output a cycle C .

Using DFS...

... to check for Acyclicity and compute Topological Ordering

Question

Given G , is it a DAG? If it is, generate a topological sort. Else output a cycle C .

DFS based algorithm:

- ① Compute $\text{DFS}(G)$
- ② If there is a back edge $e = (v, u)$ then G is not a DAG. Output cycle C formed by path from u to v in T plus edge (v, u) .
- ③ Otherwise output nodes in decreasing post-visit order. Note: no need to sort, $\text{DFS}(G)$ can output nodes in this order.

Algorithm runs in $O(n + m)$ time.

Using DFS...

... to check for Acyclicity and compute Topological Ordering

Question

Given G , is it a DAG? If it is, generate a topological sort. Else output a cycle C .

DFS based algorithm:

- ① Compute $\text{DFS}(G)$
- ② If there is a back edge $e = (v, u)$ then G is not a DAG. Output cycle C formed by path from u to v in T plus edge (v, u) .
- ③ Otherwise output nodes in decreasing post-visit order. Note: no need to sort, $\text{DFS}(G)$ can output nodes in this order.

Algorithm runs in $O(n + m)$ time.

Correctness is not so obvious. See next two propositions.

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 DFS search tree and the edge (u, v) .

Only if: Suppose there is a cycle $C = v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k \rightarrow v_1$.

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

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 $[\text{pre}(v), \text{post}(v)]$.

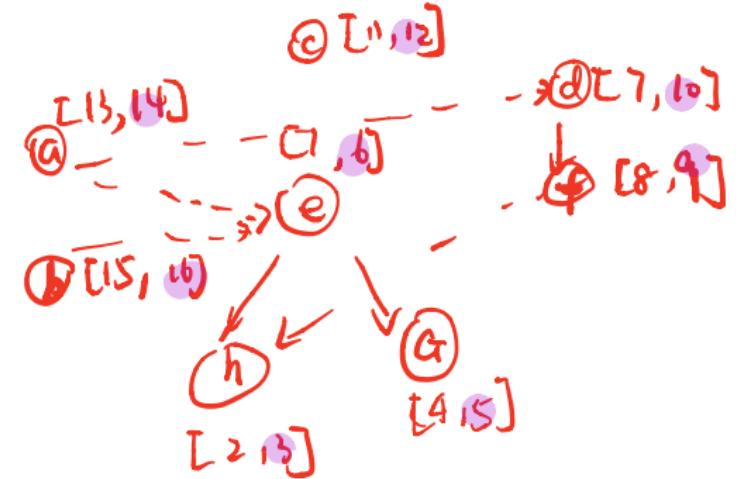
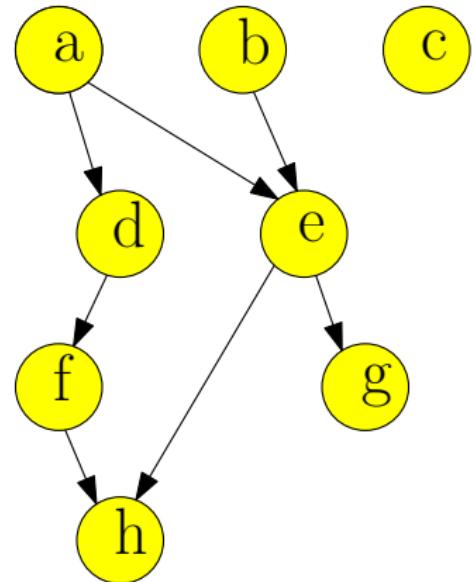
Implies that u is explored during $\text{DFS}(v)$ and hence is a descendent of v . Edge (u, v) implies a cycle in G but G is assumed to be DAG! ← back edge

- 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 . → (u, v) cannot exist, contradicts



Example



post order : b, a, c, d, f, e , g , h (\leftarrow post order number)

Part III

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.

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  $\text{DFS}(G, u)$  (int)  
        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.
```

TB 所有 $\text{SCC}(G, u)$ 中的
Mark $\#$ visited

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

Finding all SCCs of a Directed Graph

Problem

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

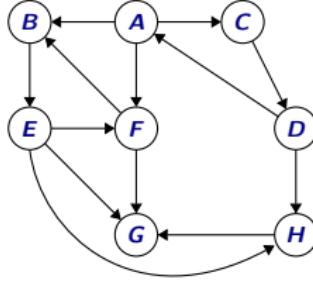
Straightforward algorithm:

```
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.
```

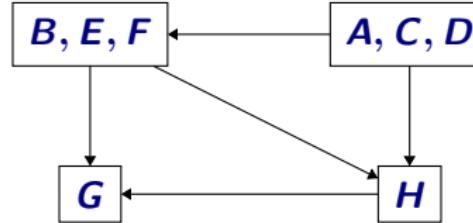
Running time: $O(n(n + m))$

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

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

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

Justification

- ➊ $\text{DFS}(u)$ only visits vertices (and edges) in $\text{SCC}(u)$
- ➋ ... since there are no edges coming out a sink!
- ➌
- ➍

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 $\text{SCC}(u)$
- ④

Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph...

Wishful Thinking Algorithm

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

Justification

- ➊ $\text{DFS}(u)$ only visits vertices (and edges) in $\text{SCC}(u)$
... since there are no edges coming out a sink!
- ➋ $\text{DFS}(u)$ takes time proportional to size of $\text{SCC}(u)$
- ➌ Therefore, total time $O(n + m)$!

Big Challenge(s)

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

Big Challenge(s)

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

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

Big Challenge(s)

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

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

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

Linear Time Algorithm

...for computing the strong connected components in G

Find the information of the sinks (for finding
↓
of
 G)
SCC)

do $\text{DFS}(G^{\text{rev}})$ and output 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

=

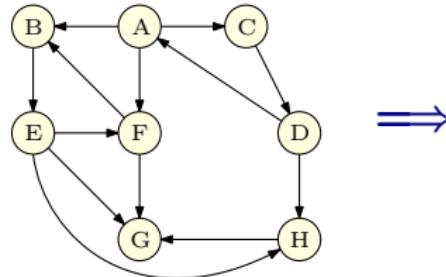
find sources
in G^{rev}

Theorem

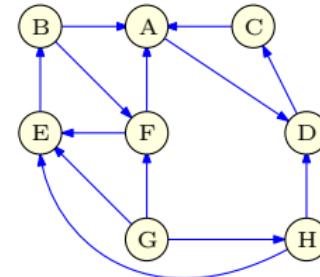
Algorithm runs in time $O(m + n)$ and correctly outputs all the SCCs of G .

Linear Time Algorithm: An Example - Initial steps

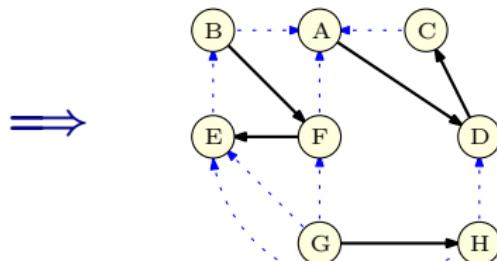
Graph G :



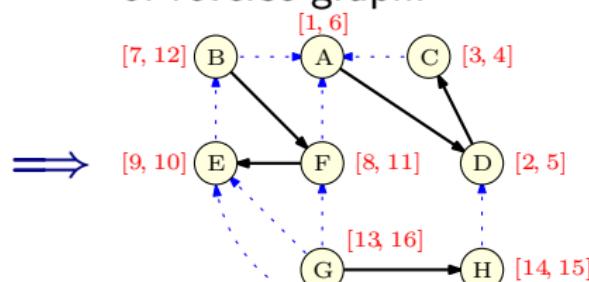
Reverse graph G^{rev} :



DFS of reverse graph:



Pre/Post DFS numbering
of reverse graph:

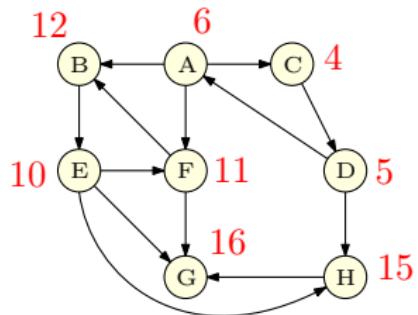


GH(B F E)ADC

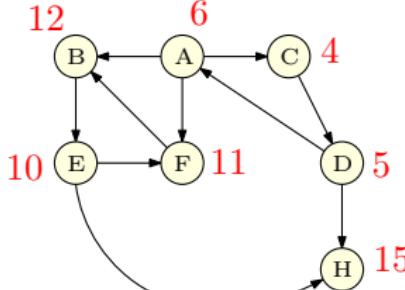
Linear Time Algorithm: An Example

Removing connected components: 1

Original graph G with rev post numbers:



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

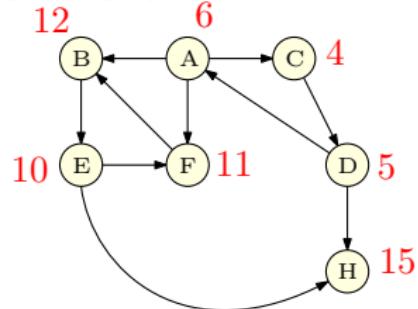


SCC computed:
 $\{G\}$

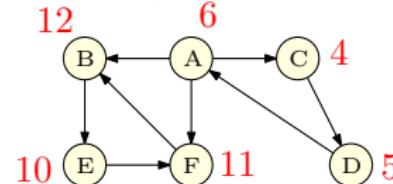
Linear Time Algorithm: An Example

Removing connected components: 2

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



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



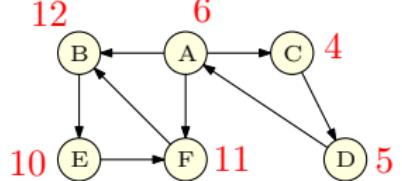
SCC computed:
 $\{G\}$

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

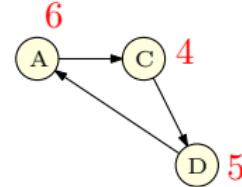
Linear Time Algorithm: An Example

Removing connected components: 3

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



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



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

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

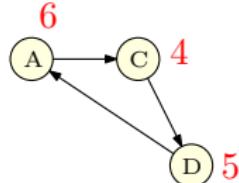
Linear Time Algorithm: An Example

Removing connected components: 4

Do **DFS** from vertex **F**

Remove visited vertices:

$\{F, B, E\}$.



Do **DFS** from vertex **A**

Remove visited vertices:

$\{A, C, D\}$.



SCC computed:

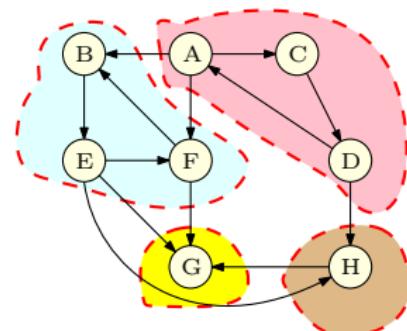
$\{G\}, \{H\}, \{F, B, E\}$

SCC computed:

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

Linear Time Algorithm: An Example

Final result



SCC computed:

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

Which is the correct answer!

Obtaining the meta-graph...

Once the strong connected components are computed.

Exercise:

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

Solving Problems on Directed Graphs

A template for a class of problems on directed graphs:

- Is the problem solvable when G is strongly connected?
- Is the problem solvable when G is a DAG?
- If the above two are feasible then is the problem solvable in a general directed graph G by considering the meta graph G^{SCC} ?

Part IV

An Application to make

Make/Makefile

- (A) I know what make/makefile is. ✓ need to make sure dependency is good
- (B) I do NOT know what make/makefile is.

make Utility [Feldman]

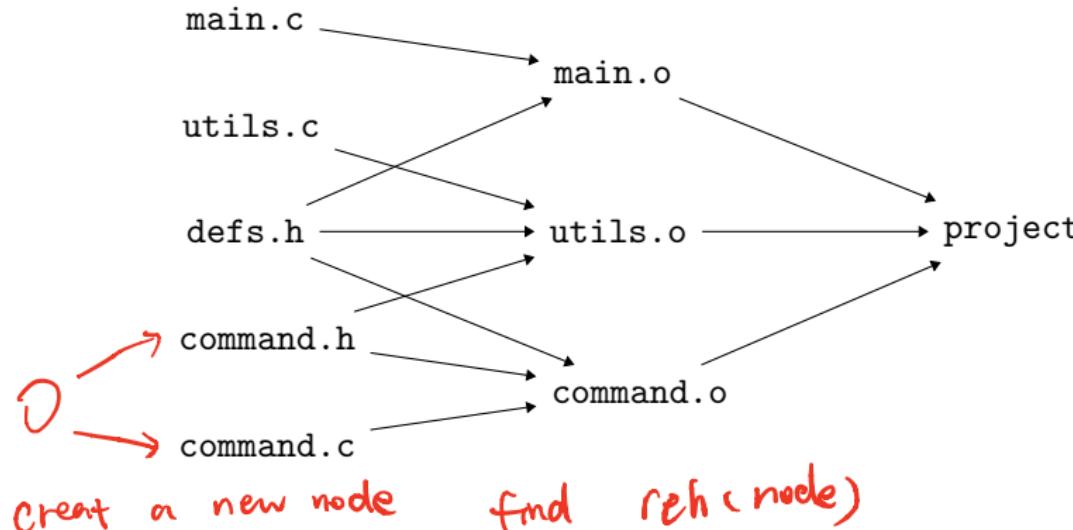
- ➊ Unix utility for automatically building large software applications
- ➋ A makefile specifies
 - ➌ Object files to be created,
 - ➍ Source/object files to be used in creation, and
 - ➎ How to create them

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

- ➊ Is the makefile reasonable?
- ➋ If it is reasonable, in what order should the object files be created? *Topological Sort*
- ➌ If it is not reasonable, provide helpful debugging information. *(cycle)*
- ➍ If some file is modified, find the fewest compilations needed to make application consistent. *(rehfile)*

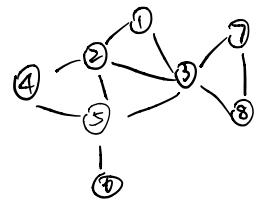
Algorithms for make

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

BFS:



① L_0

②,3 L_1

④,5,7,8 L_2

⋮
⑥ L_d

L_s