Graph Algorithms

Andreas Klappenecker

[based on slides by Prof. Welch]

Directed Graphs

Let V be a finite set and E a binary relation on V, that is, $E \subseteq V \times V$. Then the pair $G=(V,E)$ is called a directed graph.

- The elements in V are called vertices.
- The elements in E are called edges.
- Self-loops are allowed, i.e., E may contain (v,v).

Undirected Graphs

Let V be a finite set and E a subset of $\{ e \mid e \subseteq V, |e|=2 \}$. Then the pair $G=(V,E)$ is called an undirected graph.

- The elements in V are called vertices.
- The elements in E are called edges, $e=\{u,v\}$.
- Self-loops are not allowed, $e \neq \{u,u\}=\{u\}$.

Adjacency

By abuse of notation, we will write (u,v) for an edge {u,v} in an undirected graph.

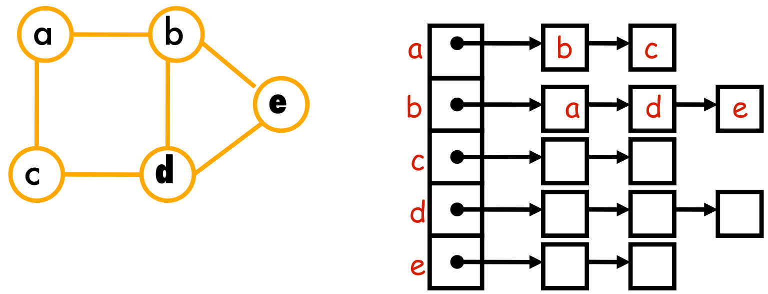
If (u,v) in E, then we say that the vertex v is adjacent to the vertex u.

For undirected graphs, adjacency is a symmetric relation.

Graph Representations

- Adjacency lists
- Adjacency matrix

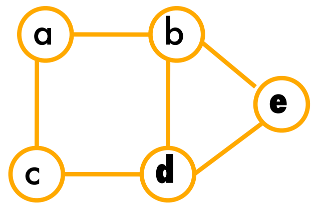
Adjacency List Representation



+ Space-efficient: just O(|V|) space for sparse graphs

- Testing adjacency is O(|V|) in the worst case

Adjacency Matrix



	a	b	c	d	e
a	0	1	1	0	0
b	1	0	0	1	1
c	1	0	0	1	0
d	0	1	1	0	1
e	0	1	0	1	0

+ Can check adjacency in constant time

- Needs $\Omega(|V|^2)$ space

Graph Traversals

Ways to traverse or search a graph such that every node is visited exactly once

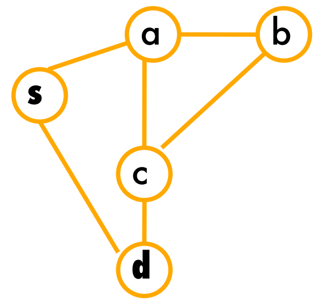
Breadth-First Search

Breadth First Search (BFS)

Input: A graph G = (V,E) and source node s in V

```
for each node v do
    mark v as unvisited
od
mark s as visited
enq(Q,s)          // first-in first-out queue Q
while Q is not empty do
    u := deq(Q)
    for each unvisited neighbor v of u do
        mark v as visited; enq(Q,v);
    od
od
```

BFS Example



Visit the nodes in the order:

s

a, d

b, c

Workout the evolution of the state of queue.

BFS Tree

- We can make a spanning tree rooted at s by remembering the "parent" of each node

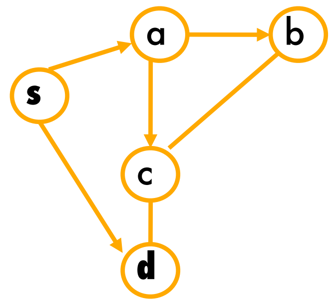
Breadth First Search #2

- Input: G = (V,E) and source s in V
- for each node v do
  - mark v as unvisited
  - parent[v] := nil
- mark s as visited
- parent[s] := s
- enq(Q,s) // FIFO queue Q

Breadth First Search #2

- while Q is not empty do
  - u := deq(Q)
  - for each unvisited neighbor v of u do
    - mark v as visited
    - parent[v] := u
    - enq(Q,v)

BFS Tree Example



BFS Trees

- BFS tree is not necessarily unique for a given graph
- Depends on the order in which neighboring nodes are processed

BFS Numbering

- During the breadth-first search, assign an integer to each node
- Indicate the distance of each node from the source s

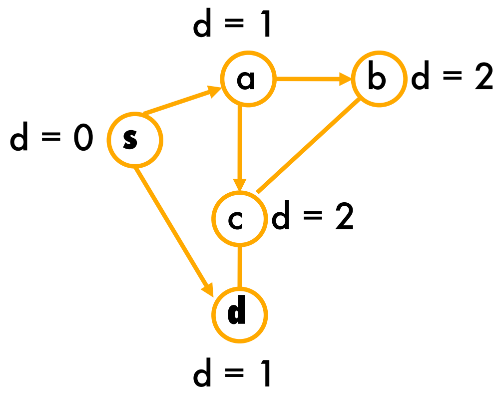
Breadth First Search #3

- Input: G = (V,E) and source s in V
- for each node v do
  - mark v as unvisited
  - parent[v] := nil
  - d[v] := infinity
- mark s as visited
- parent[s] := s
- d[s] := 0
- enq(Q,s) // FIFO queue Q

Breadth First Search #3

- while Q is not empty do
  - u := deq(Q)
  - for each unvisited neighbor v of u do
    - mark v as visited
    - parent[v] := u
    - d[v] := d[u] + 1
    - enq(Q,v)

BFS Numbering Example



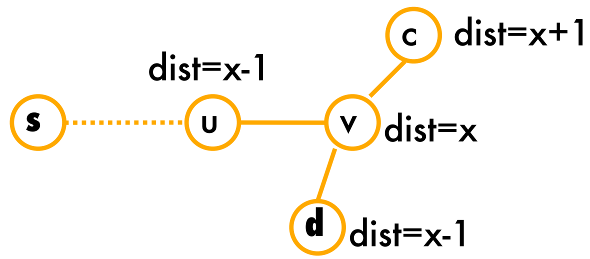
Shortest Path Tree

- Theorem: BFS algorithm
  - visits all and only nodes reachable from s
  - sets d[v] equal to the shortest path distance from s to v, for all nodes v, and
  - sets parent variables to form a shortest path tree

Proof Ideas

- Use induction on distance from s to show that the d-values are set properly.
- Basis: distance 0. d[s] is set to 0.
- Induction: Assume true for all nodes at distance x-1 and show for every node v at distance x.
- Since v is at distance x, it has at least one neighbor at distance x-1. Let u be the first of these neighbors that is enqueued.

Proof Ideas



Key property of shortest path distances:
If v has distance x,
- it must have a neighbor with distance x-1,
- no neighbor has distance less than x-1, and
- no neighbor has distance more than x+1

Proof Ideas

- Fact: When u is dequeued, v is still unvisited.
  - because of how queue operates and since d never underestimates the distance
- By induction, d[u] = x-1.
- When v is enqueued, d[v] is set to

  d[u] + 1= x

BFS Running Time

- Initialization of each node takes O(V) time
- Every node is enqueued once and dequeued once, taking O(V) time
- When a node is dequeued, all its neighbors are checked to see if they are unvisited, taking time proportional to number of neighbors of the node, and summing to O(E) over all iterations
- Total time is O(V+E)

Depth-First Search

Depth-First Search

Input: $G = (V,E)$

for each node u **do**

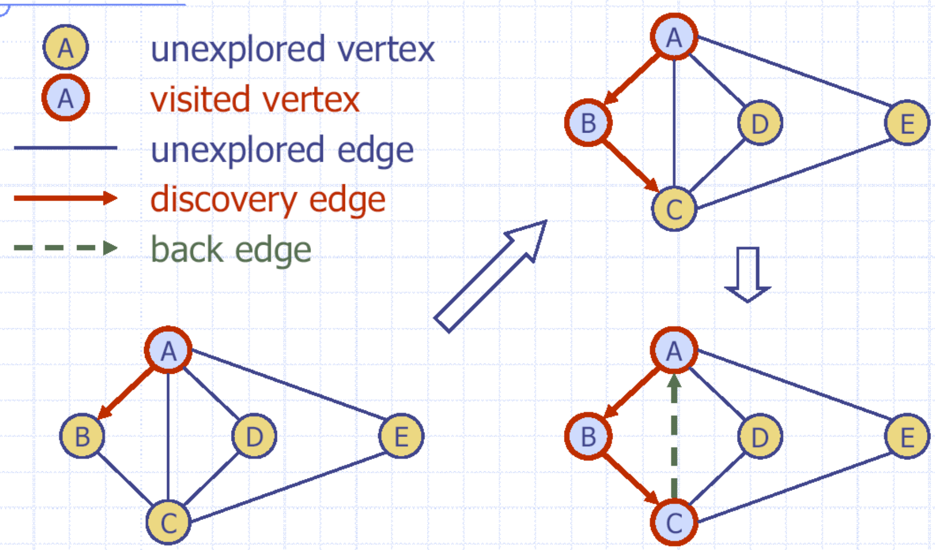
    mark u as unvisited

od;

for each unvisited node u

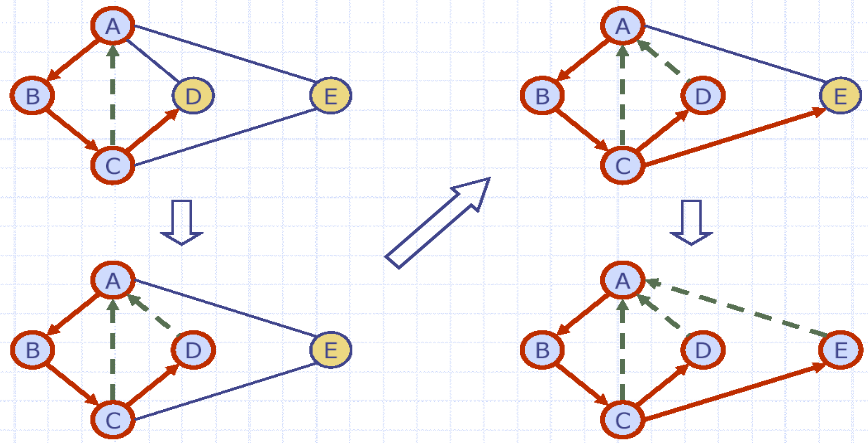
```
recursiveDFS(u):
  mark u as visited;
  for each unvisited neighbor v of u do
     recursiveDFS(v)
od
```

Example



Example taken from http://atcp07.cs.brown.edu/courses/cs016/Resource/old_lectures/DFS.pdf

Example (cont.)



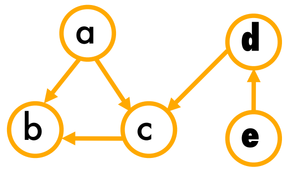
Example taken from http://atcp07.cs.brown.edu/courses/cs016/Resource/old_lectures/DFS.pdf

Disconnected Graphs

What if the graph is disconnected or is directed?

We call DFS on several nodes to visit all nodes

- purpose of second for-loop in non-recursive wrapper



DFS Forest

By keeping track of parents, we want to construct a forest resulting from the DFS traversal.

Depth-First Search #2

- Input: $G = (V,E)$
- for each node u do
  - mark u as unvisited
  - parent[u] := nil
- for each unvisited node u do
  - parent[u] := u

    // a root
  - call recursive DFS(u)

- recursiveDFS(u):
- mark u as visited
- for each unvisited neighbor v of u do
- parent[v] := u
- call recursiveDFS(v)

Further Properties of DFS

Let us keep track of some interesting information for each node.

We will timestamp the steps and record the

- discovery time, when the recursive call starts
- finish time, when its recursive call ends

Depth-First Search #3

- Input: $G = (V,E)$
- for each node u do
  - mark u as unvisited
  - parent[u] := nil
- time := 0
- for each unvisited node u do
  - parent[u] := u // a root
  - call recursive DFS(u)

- recursiveDFS(u):
- mark u as visited
- time++
- disc[u] := time
- for each unvisited neighbor v of u do
- parent[v] := u
- call recursiveDFS(v)
- time++
- fin[u] := time

Running Time of DFS

- initialization takes O(V) time
- second for loop in non-recursive wrapper considers each node, so O(V) iterations
- one recursive call is made for each node
- in recursive call for node u, all its neighbors are checked; total time in all recursive calls is O(E)

Nested Intervals

- Let interval for node v be [disc[v],fin[v]].
- Fact: For any two nodes, either one interval precedes the other or one is enclosed in the other

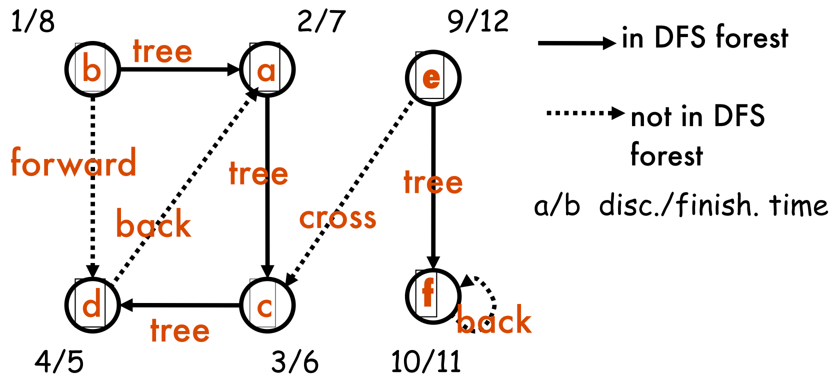
  [Reason: recursive calls are nested.]

- Corollary: v is a descendant of u in the DFS forest iff the interval of v is inside the interval of u.

Classifying Edges

- Consider edge (u,v) in directed graph G = (V,E) w.r.t. DFS forest
- tree edge: v is a child of u
- back edge: v is an ancestor of u
- forward edge: v is a descendant of u but not a child
- cross edge: none of the above

Example of Classifying Edges



tree edge: v child of u
back edge: v ancestor of u
forward edge: v descendant of u, but not child
cross edge: none of the above

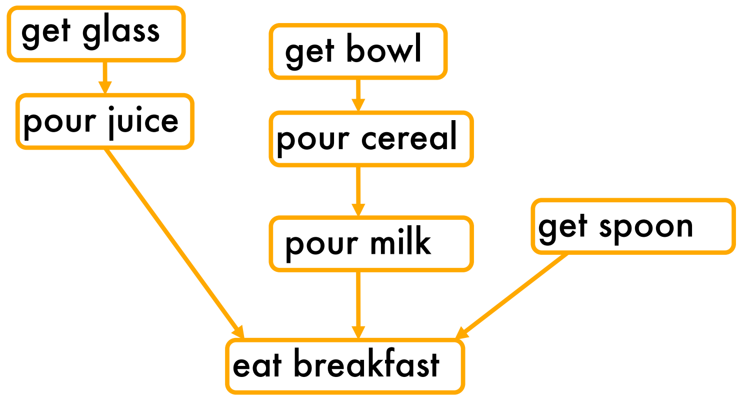
DFS Application: Topological Sort

- Given a directed acyclic graph (DAG), find a linear ordering of the nodes such that if (u,v) is an edge, then u precedes v.
- DAG indicates precedence among events:
  - events are graph nodes, edge from u to v means event u has precedence over event v
- Partial order because not all events have to be done in a certain order

Precedence Example

- Tasks that have to be done to eat breakfast:
  - get glass, pour juice, get bowl, pour cereal, pour milk, get spoon, eat.
- Certain events must happen in a certain order (ex: get bowl before pouring milk)
- For other events, it doesn't matter (ex: get bowl and get spoon)

Precedence Example



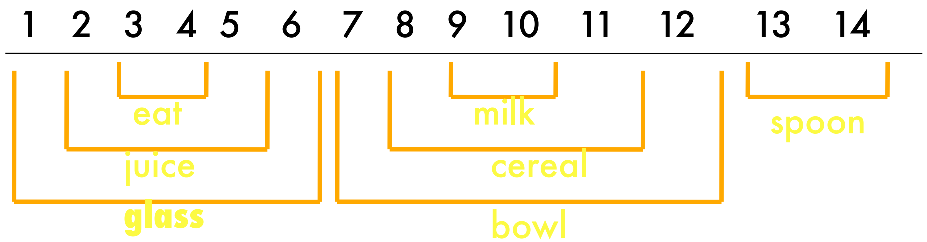
Order: glass, juice, bowl, cereal, milk, spoon, eat.

Why Acyclic?

- Why must a directed graph be acyclic for the topological sort problem?
- Otherwise, no way to order events linearly without violating a precedence constraint.

Idea for Topological Sort Alg.

- What does DFS do on a DAG?



consider reverse order of finishing times:
spoon, bowl, cereal, milk, glass, juice, eat

Topological Sort Algorithm

input: DAG G = (V,E)

1. call DFS on G to compute finish[v] for all nodes v
2. when each node's recursive call finishes, insert it on the front of a linked list
3. return the linked list

Correctness of T.S. Algorithm

- Show that if (u,v) is an edge, then v finishes before u.

Case 1: v is finished when u is discovered. Then v finishes before u finishes.

Case 2: v is not yet discovered when u is discovered.

Claim: v will become a descendant of u and thus v will finish before u finishes.

Case 3: v is discovered but not yet finished

Correctness of T.S. Algorithm

- v is discovered but not yet finished when u is discovered.
- Then u is a descendant of v.
- But that would make (u,v) a back edge and a DAG cannot have a back edge (the back edge would form a cycle).
- Thus Case 3 is not possible.

DFS Application: Strongly Connected Components

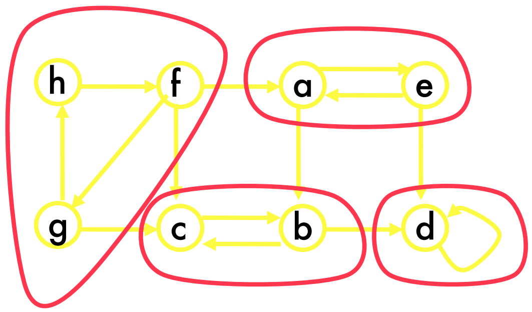
- Consider a directed graph.
- A strongly connected component (SCC) of the graph is a maximal set of nodes with a (directed) path between every pair of nodes
- Problem: Find all the SCCs of the graph.

What Are SCCs Good For?

- packaging software modules
- construct directed graph of which modules call which other modules
- A SCC is a set of mutually interacting modules
- pack together those in the same SCC

from http://www.cs.princeton.edu/courses/archive/fall07/cos226/lectures.html

SCC Example



four SCCs

How Can DFS Help?

- Suppose we run DFS on the directed graph.
- All nodes in the same SCC are in the same DFS tree.
- But there might be several different SCCs in the same DFS tree.
  - Example: start DFS from node h in previous graph

Main Idea of SCC Algorithm

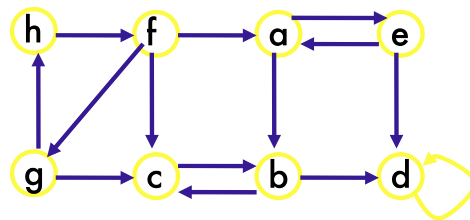
- DFS tells us which nodes are reachable from the roots of the individual trees
- Also need information in the "other direction": is the root reachable from its descendants?
- Run DFS again on the "transpose" graph (reverse the directions of the edges)

SCC Algorithm

input: directed graph $G = (V,E)$

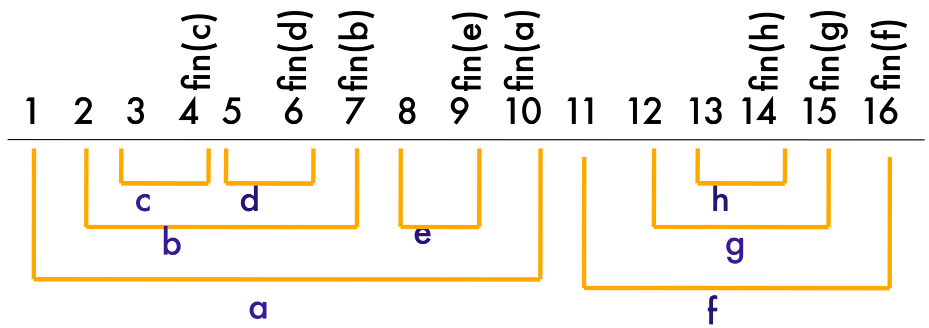
1. call DFS(G) to compute finishing times
2. compute G^T // transpose graph
3. call DFS(G^T), considering nodes in decreasing order of finishing times
4. each tree from Step 3 is a separate SCC of G

SCC Algorithm Example



input graph - run DFS

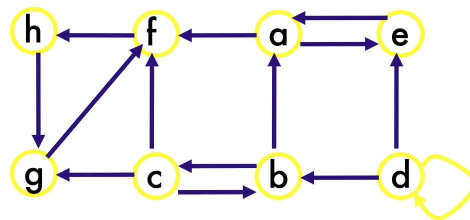
After Step 1



Order of nodes for Step 3: f, g, h, a, e, b, d, c

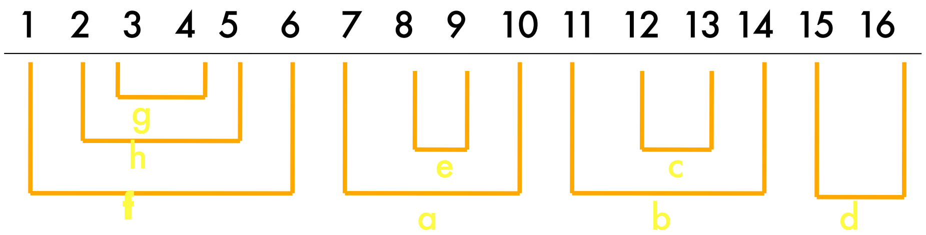
f reaches g reaches h; a reaches b, e; b reaches c,d

After Step 2



transposed input graph - run DFS with specified order of nodes: f, g, h, a, e, b, d, c
f can be reached from h, h can be reached from g, ...

After Step 3



SCCs are {f,h,g} and {a,e} and {b,c} and {d}.

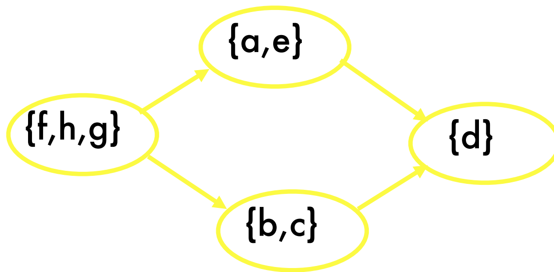
Running Time of SCC Algorithm

- Step 1: O(V+E) to run DFS
- Step 2: O(V+E) to construct transpose graph, assuming adjacency list rep.
- Step 3: O(V+E) to run DFS again
- Step 4: O(V) to output result
- Total: O(V+E)

Correctness of SCC Algorithm

- Proof uses concept of component graph, G^{SCC}, of G.
- Nodes are the SCCs of G; call them C_1, C_2, …, C_k
- Put an edge from C_i to C_j iff G has an edge from a node in C_i to a node in C_j

Example of Component Graph



based on example graph from before

Facts About Component Graph

- Claim: G^{SCC} is a directed acyclic graph.
- Why?
- Suppose there is a cycle in G^{SCC} such that component C_i is reachable from component C_j and vice versa.
- Then C_i and C_j would not be separate SCCs.

Facts About Component Graph

- Consider any component C during Step 1 (running DFS on G)
- Let $d(C)$ be earliest discovery time of any node in C
- Let $f(C)$ be latest finishing time of any node in C
- Lemma: If there is an edge in G^{SCC} from component C' to component C, then

  $$f(C') > f(C).$$

Proof of Lemma

- Case 1: $d(C') < d(C)$.
- Suppose x is first node discovered in C'.
- By the way DFS works, all nodes in C' and C become descendants of x.
- Then x is last node in C' to finish and finishes after all nodes in C.
- Thus $f(C') > f(C)$.

Proof of Lemma

- Case 2: $d(C') > d(C)$.
- Suppose y is first node discovered in C.
- By the way DFS works, all nodes in C become descendants of y.
- Then y is last node in C to finish.
- Since $C' \to C$, no node in C' is reachable from y, so y finishes before any node in C' is discovered.

SCC Algorithm is Correct

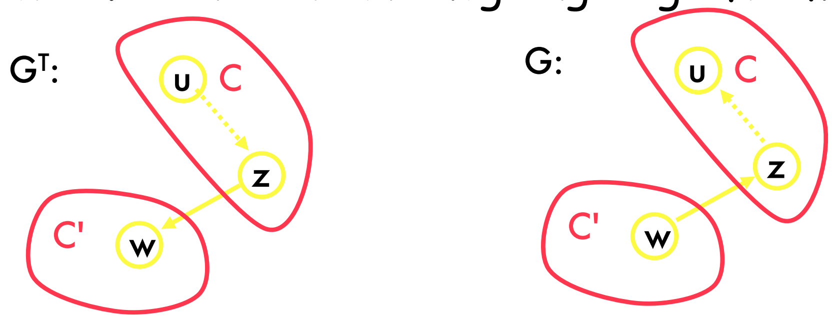
- Prove this theorem by induction on number of trees found in Step 3 (calling DFS on G^T).
- Hypothesis is that the first k trees found constitute k SCCs of G.
- Basis: k = 0. No work to do !

SCC Algorithm is Correct

- Induction: Assume the first k trees constructed in Step 3 correspond to k SCCs, and consider the (k+1)st tree.
- Let u be the root of the (k+1)st tree.
- u is part of some SCC, call it C.
- By the inductive hypothesis, C is not one of the k SCCs already found and all nodes in C are unvisited when u is discovered.
  - By the way DFS works, all nodes in C become part of u's tree

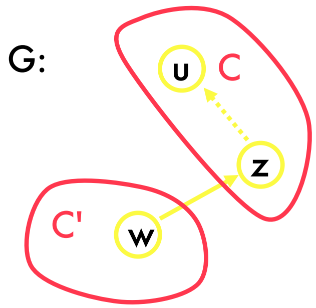
SCC Algorithm is Correct

- Show only nodes in C become part of u's tree. Consider an outgoing edge from C.



SCC Algorithm is Correct

- By lemma, in Step 1 the last node in C' finishes after the last node in C finishes.
- Thus in Step 3, some node in C' is discovered before any node in C is discovered.
- Thus all of C', including w, is already visited before u's DFS tree starts



Conclusion

- The proof that the algorithm does indeed find the strongly connected components is rather typical.
- The main ideas are quite simple:
  - the DFS forest of G specifies which nodes can be reached from their roots
  - the DFS forest of G^{t} specifies from where the root can be reached.
- You need to have a good grasp of the algorithm before you can attempt to prove it correct. The formalization of the proof can be difficult.