Algorithms
Professor John Reif

ALG 5.1

Graph Algorithms using Depth First Search:

Main Reading Selections:
CLR, Chapter 23

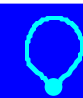
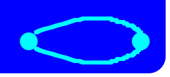
Auxillary Reading Selections:
AHU-Design, Sections 5.2-5.5
AHU-Data, Chapters 6 and 7
BB, Chapter 6

Graph Terminology

graph	G = (V,E)
vertex set	V
edge set	E pairs of vertices which are adjacent

G *directed* if edges ordered pairs (u,v)

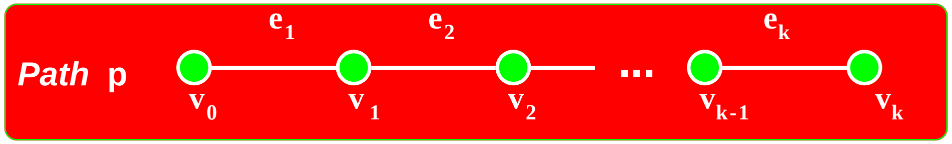
G *undirected* if edges unordered pairs {u,v}

proper graph:
no *loops* 
no *multi-edges* 

Subgraph G' of G

G' = (V', E') where

V' is a subset of V, E' is a subset of E



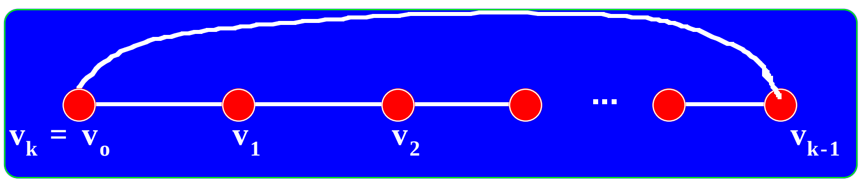
p is a sequence of vertices $v_0, v_1, \ldots, v_k$
where for $i=1, \ldots, k$, v_{i-1} is adjacent to v_i

Equivalently,

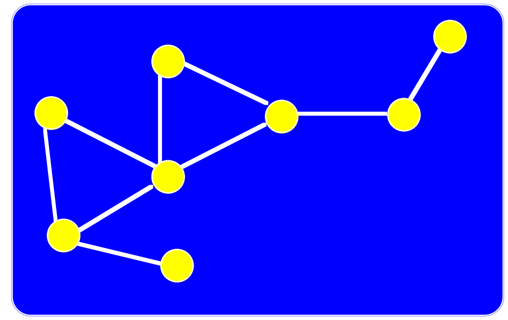
p is a sequence of edges $e_1, \ldots, e_k$
where for $i=2,\ldots,k$ edges e_{i-1}, e_i share a vertex

simple path no edge or vertex repeated, except possibly $v_0 = v_k$

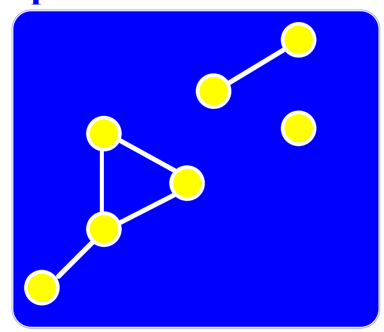
cycle is a path p with $v_0 = v_k$ where $k>1$



Connectivity of Undirected Graphs

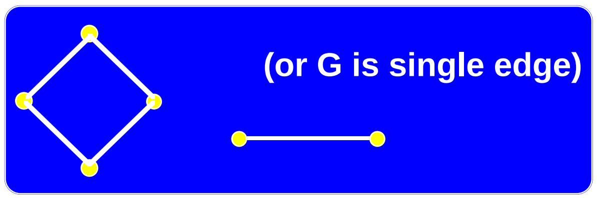


G is *connected* if $\exists$ path between each pair of vertices



else G has ≥ 2 *connected components:* maximal connected subgraphs

G is *biconnected* if $\exists$ two disjoint paths between each pair of vertices

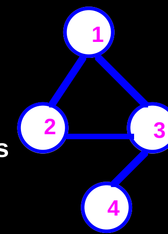


Graph Representation

graph G = (V,E)

n = |V| = # vertices

m = |E| = # edges



Adjacency Matrix A

A is size n x n

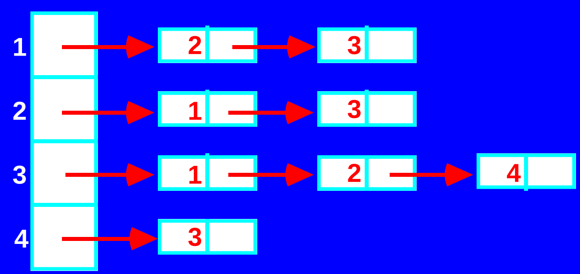
$$A(i,j) = \begin{cases} 1 & (i,j) \in E \\ 0 & \text{else} \end{cases}$$

	1	2	3	4
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1
4	0	0	1	0

space cost $n^2 - n$

Adjacency Lists Adj(1), ... , Adj (n)

Adj(v) = list of vertices adjacent to v



space cost O(n+m)

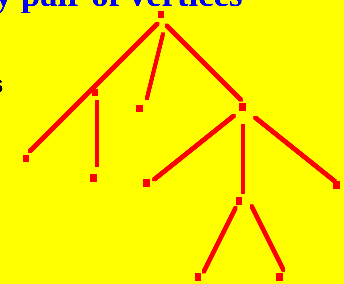
Tree

T is graph with unique path

between every pair of vertices

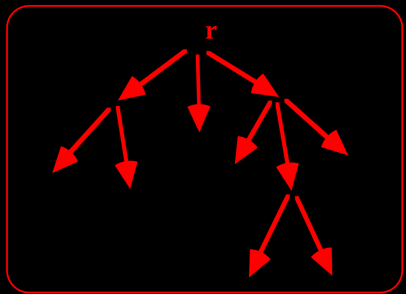
n = # vertices

n-1 = # edges



Directed Tree

T is digraph with distinguished vertex *root* r such that each vertex reachable from r by a unique path



Family Relationships:

- *ancesters*
- *descendants*
- *parent*
- *child*
- *siblings*

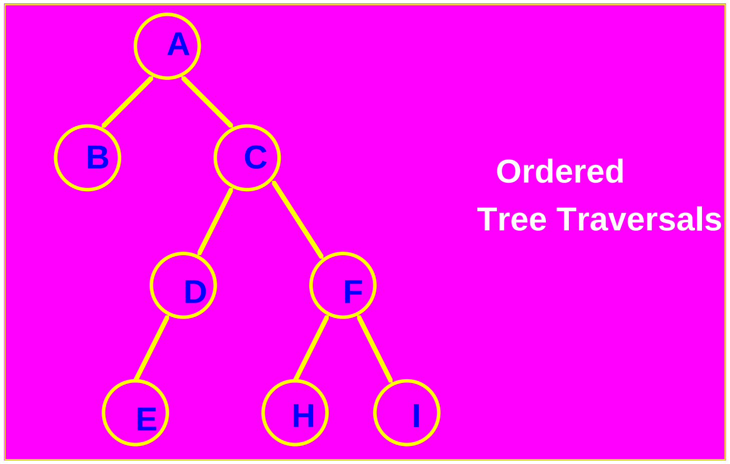
leaves have no proper descendants

Ordered Tree

is a directed tree with siblings ordered

Forest

set of Trees



Preorder **A,B,C,D,E,F,H,I**

[1] root (order vertices as pushed on stack)
[2] preorder left subtree
[3] preorder right subtree

Postorder **B,E,D,H,I,F,C,A**

[1] postorder left subtree
[2] postorder right subtree
[3] root
(order vertices as popped off stack)

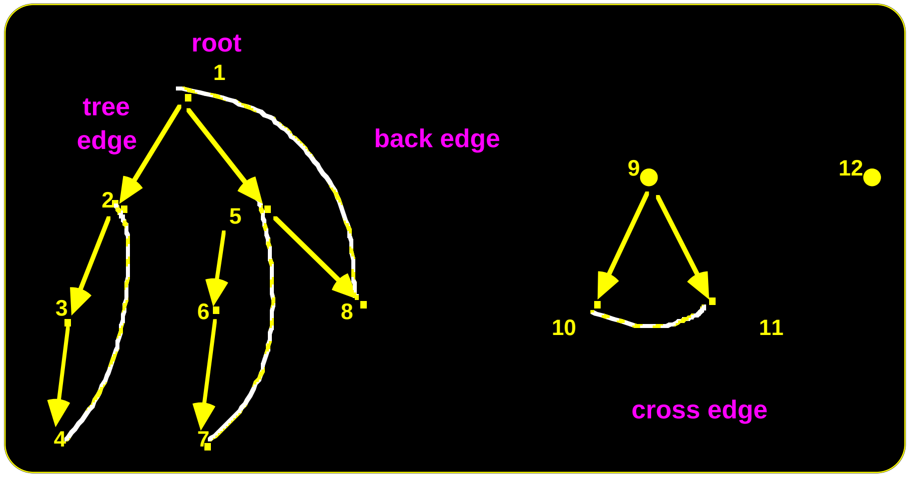
In order **B,A,E,D,C,H,F,I**

[1] inorder left subtree
[2] root
[3] inorder right subtree

T is a *spanning tree* of graph G if

(1) T is a directed tree with the same vertex set as G

(2) each edge of T is a directed version of an edge of G



an edge (u,v) of G-T is *backedge* if u is a descendant or ancester of v. else (u,v) is a *crossedge*

Spanning Forest:

forest of spanning trees of connected components of G

Tarjan's Algorithm
Depth First Search

Input graph G =(V,E) represented by adjacency lists Adj(v) for each v εV

[0] N ←0

[1] *for* all v εV *do* (number (v) ←0
children (v) ←( )) *od*

[2] *for* all v εV *do*
if number (v)=0 *then* DFS(v)

[3] *output* spanning forest defined by children

recursive procedure DFS(v)

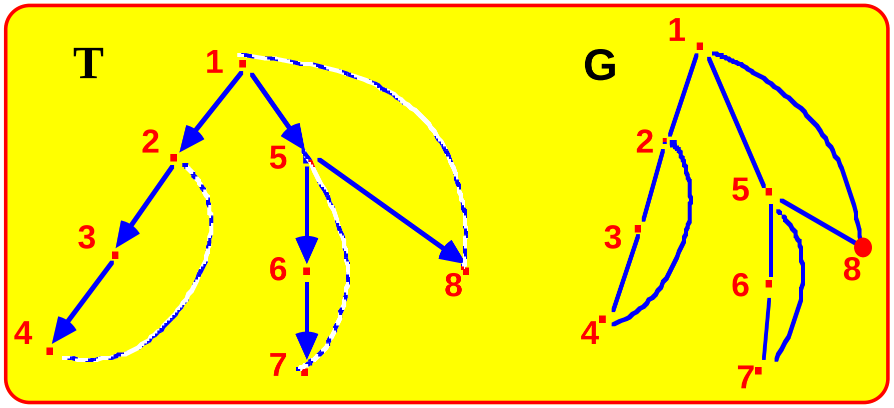
[1] N ← N+1; number (v) ← N

[2] for each u ε Adj(v) *do*
if number(u) = 0 *then*
(add u to children (v); DFS(u))

input size $n = |V|, m = |E|$

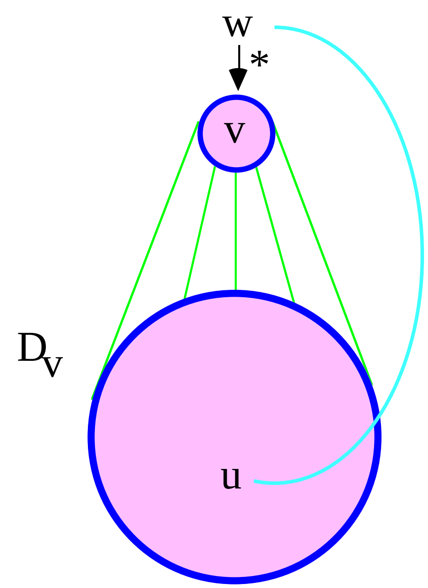
Theorem Depth First Search takes total time cost O(n+m)

proof
can associate with each edge and vertex a constant number of operations.



Sup. we preorder number a tree T
Let D_v = # of descendants of v

Lemma
u is descendant of v
iff $v \le u < v + D_v$



Lemma
If u is descendant of v
and (u,w) is back edge s.t. $w < v$
then w is a proper ancestor of v

Depth First Search Tree T

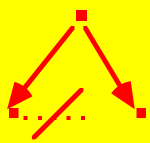
$u \rightarrow v$ iff (u,v) is tree edge of T

$u \xrightarrow{*} v$ iff u is an ancester of v

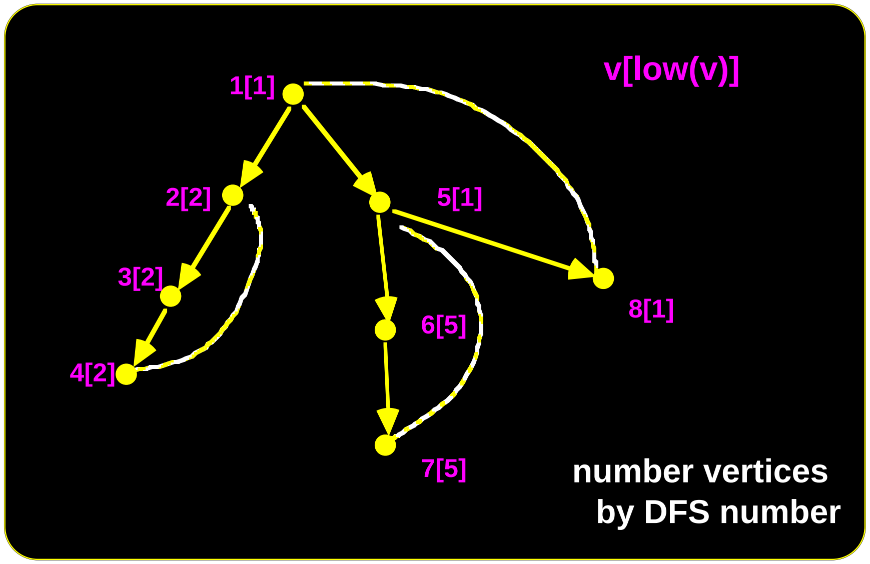
$u - - - v$ iff (u,v) is backedge if $(u,v)\ \varepsilon\ G\text{-}T$

with either $u \xrightarrow{*} v$ or $v \xrightarrow{*} u$

note DFS tree T has *no* cross edges



will number vertices by order visited in DFS (preorder of T)



example
figure gives v[low(v)]

For each vertex v,
define $low(v) = \min(\{v\} \cup \{w \mid v \xrightarrow{*} - - - w\})$

can prove by induction:

Lemma $low(v) = \min(\{v\} \cup \{low(w) \mid v \rightarrow w\} \cup \{w \mid v - - - w\})$

can easily be computed during DFS in postorder.

G is *Biconnected* iff either

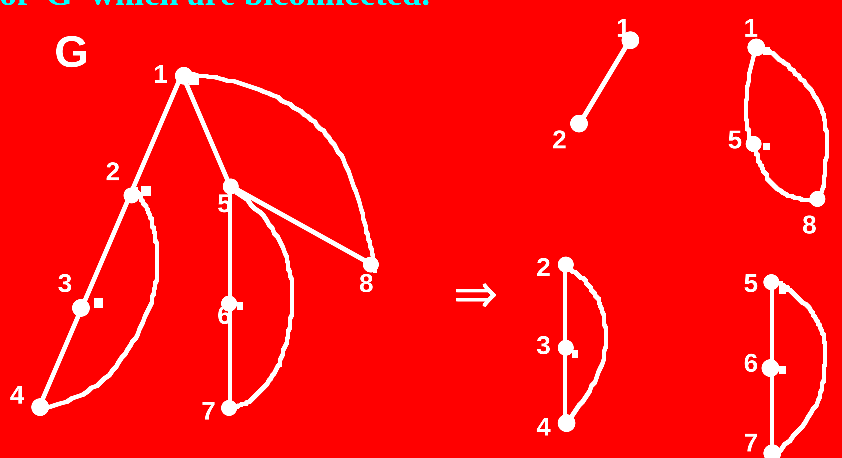
(1) G is a single edge, or

(2) for each triple of vertices u,v,w

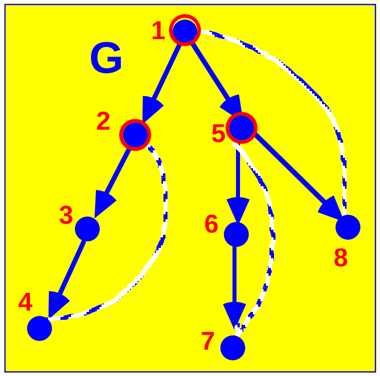
$\exists$ w-avoiding path from u to v

(equivalently: $\exists$ two disjoint paths from u to v)

Biconnected Components
maximal subgraphs
of G which are biconnected.



The intersection of two biconnected components consists of at most *one* vertex, called an *Articulation Point.*



Example 1, 2, 5 are articulation points

If can *find articulation points*
then can *compute biconnected components:*

Method during DFS, use auxillary stack to store visited edges. Each time we complete the DFS of a tree child of an articulation point, pop all stacked edges currently in stack (these form a biconnected component) up to that tree edge.

Characterization

Theorem

a is an *articulation point* iff either

(1) a is root with ≥ 2 tree children

or

(2) a is not root but a has a tree child v with low(v) $\geq a$

(*note* easy to check given low computed)

proof

The conditions are *sufficient* since any a-avoiding path from v remains in the subtree T_v rooted at v, if v is a child of a

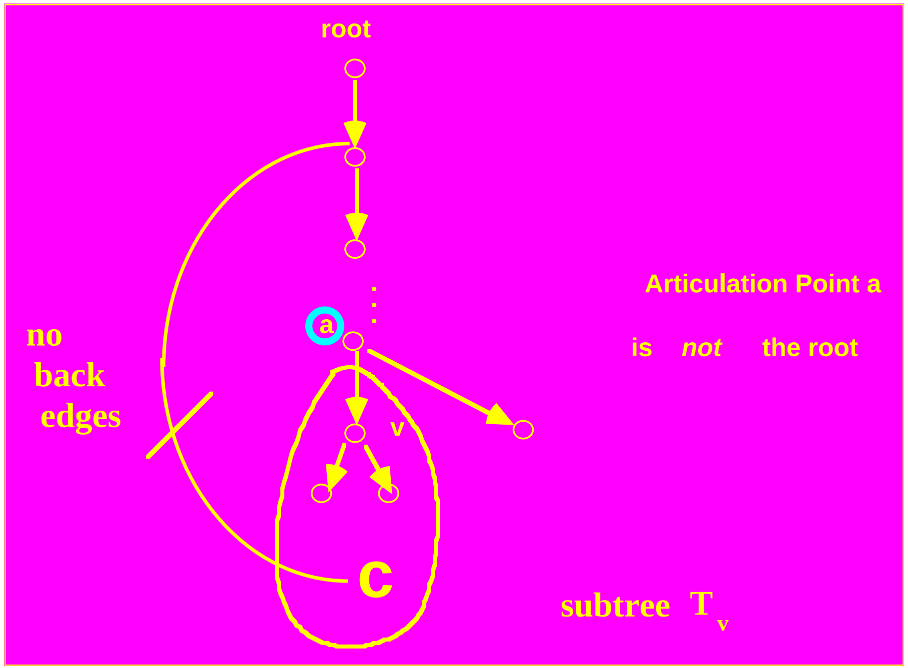
To show condition *necessary*, assume a is an articulation point.

Case(1)

If a is a root and is articulation point, a must have ≥ 2 tree edges to two distinct biconnected components.

Case(2) If a is *not* root, consider graph G- $\{a\}$ which must have a connected component C consisting of only descendants of a, and with no backedge from C to an ancestor of v. Hence a has a tree child v in C and low (v) $\geq a$

Case (2)



Theorem

The Biconnected Components of G = (V,E) can be computed in time O(|V|+|E|) using a RAM

proof

Summary of Algorithm:

[0] initialize a STACK to *empty*

During* a DFS traversal *do

[1] add visited edge to STACK

[2] compute low of visited vertex v using Lemma

[3] test if v is an articulation point

[4] if so, *for* each u ε children(v) in order where low $(u) \geq v$

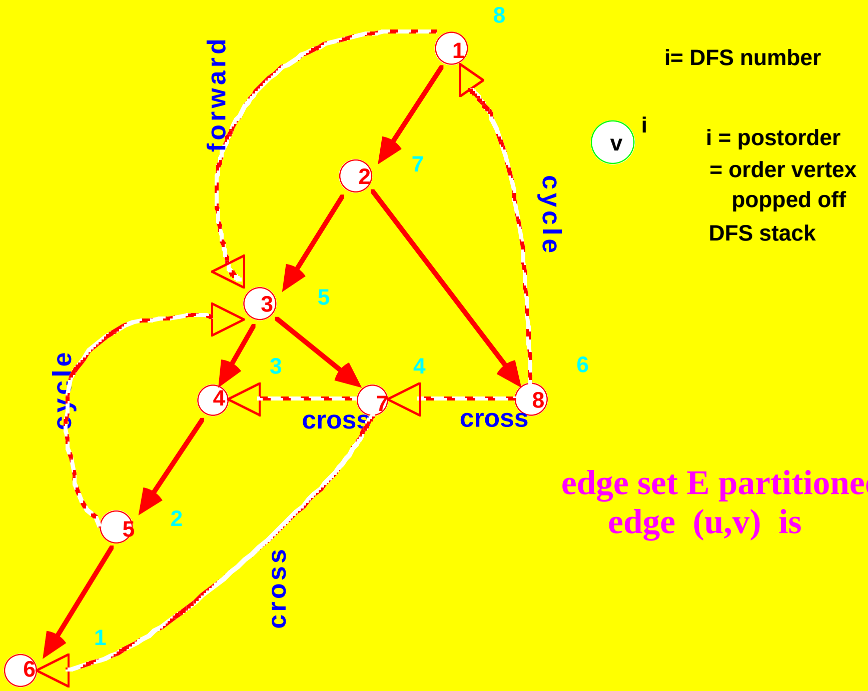
***do* pop all edges in STACK upto and including tree edge (v,u)**
***output* these edges as a biconnected component of G**
od

Time Bounds:

Each edge and vertex can be associated with 0(1) operations. So time O(|V|+|E|).

Depth first search tree T of Directed Graph

G = (V,E)



edge set E partitioned:
edge (u,v) is

- ***tree edge*** **if** $u \rightarrow v$ **in T**
- ***cycle edge*** **if** $v \xrightarrow{*} u$
- ***forward edge*** **if** $(u,v) \notin T$ **but** $u \xrightarrow{*} v$ **in T**
- ***cross edge*** **otherwise**

Digraph G = (V,E) is *acylic* if it has no cycles

Topological Order

$V = \{v_1, \ldots, v_n\}$ satisfies
$(v_i, v_j) \in E \Rightarrow i < j$

Lemma
G is acylic iff ∃ no cycle edge

proof

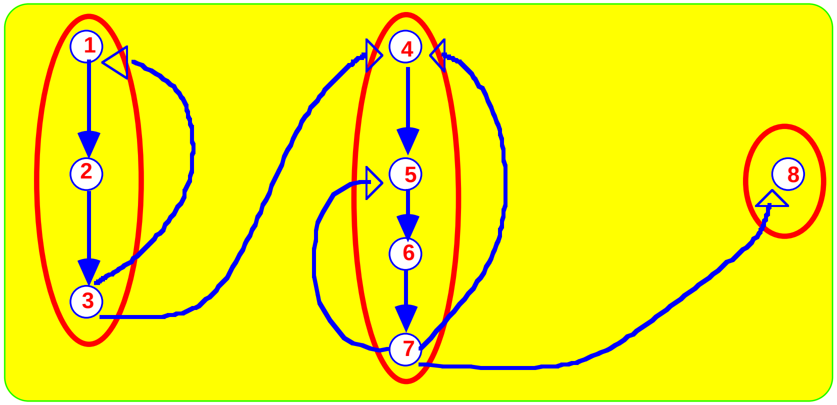
Suppose (u,v) ε E is a cycle edge, so $v \xrightarrow{*} u$. But let $e_1, \ldots, e_k$ be the tree edges from v to u. Then (u,v), $e_1, \ldots, e_k$ is a cycle.

Next suppose there is no cycle edge.

Then order vertices in postorder of DFS spanning forest (i.e., in order vertices are popped off DFS stack). This is a *reverse topological order* of G. So G can have no cycles.

Note: Gives an O(|V|+|E|) algorithm for computing Topological Ordering of an acyclic graph G = (V,E), (Knuth).

Directed Graph G = (V,E)



Strong Component

maximum set vertices S of V such that $\forall$ u,v ε S $\exists$ cycle containing u,v

Collapsed Graph

G* derived by collapsing each strong component into a single vertex.

note G* is acyclic.

(due to Kosaraju)

Algorithm
Strong Components

Input **digraph G**

[1] Perform DFS on G. Renumber vertices by postorder.

[2] Let G^- by digraph derived from G by reversing direction of each edge.

[3] Perform DFS on G^-, starting at highest numbered vertex.

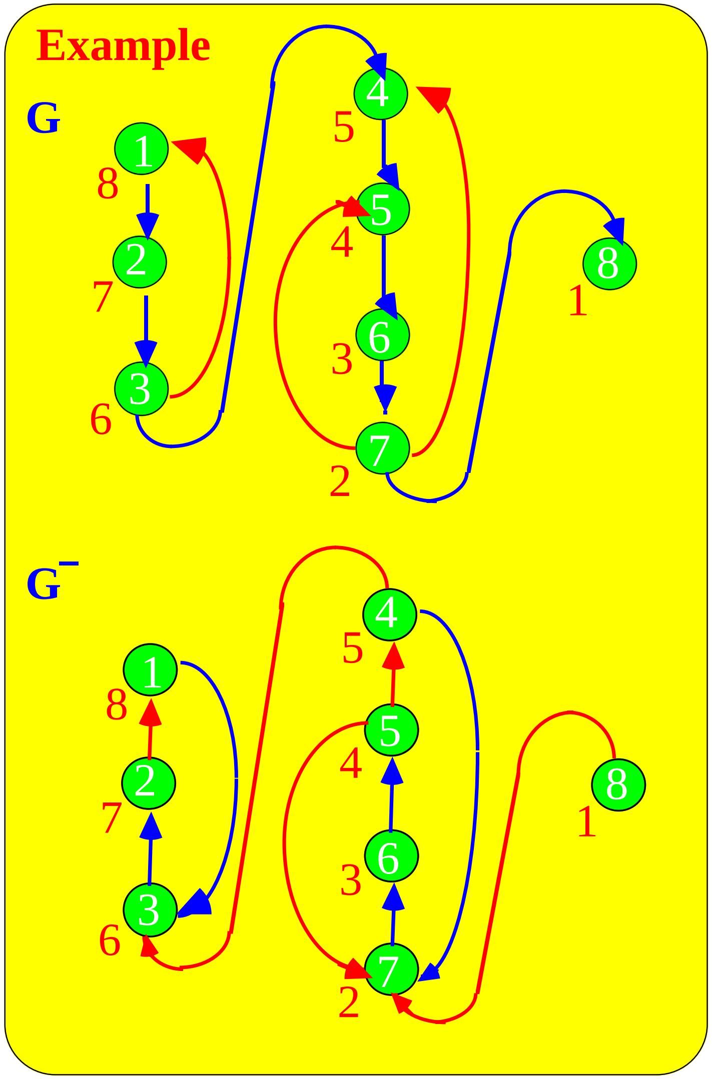
Output resulting DFS tree of G^- as a strongly connected component of G.

[4] repeat [3], starting at highest numbered vertex not so for visited (halt when all vertices visited)

Time Bounds

each DFS costs time O(|V|+|E|)

$\Rightarrow$ total time O(|V|+|E|).



Theorem

The Algorithm outputs the strong components of G.

proof

We must show these are exactly the vertices in each DFS spanning forest of G^{-}

Suppose

v,w in the same strong component and DFS search in G^{-} starts at vertex r and reaches v. Then w will also be reached. So v,w are output together in same spanning tree of G^{-}.

Suppose

v,w output in same spanning tree of G^{-}. Let r be the root of that spanning tree. Then $\exists$ paths in G^{-} from r to each of v and w. So there exists paths in G to r from each of v and w. Suppose no path in G to r from v. Then since r has a higher postorder than v, there is no path in G from v to r, a contradiction. Hence $\exists$ path in G from r to v, and similar argument gives path from r to w. Hence, v and w are in a cycle of G, so must be in the same strong component.