Unit 4.2 Depth First Search

Algorithms

EE3980

Apr. 16, 2020

Depth First Search

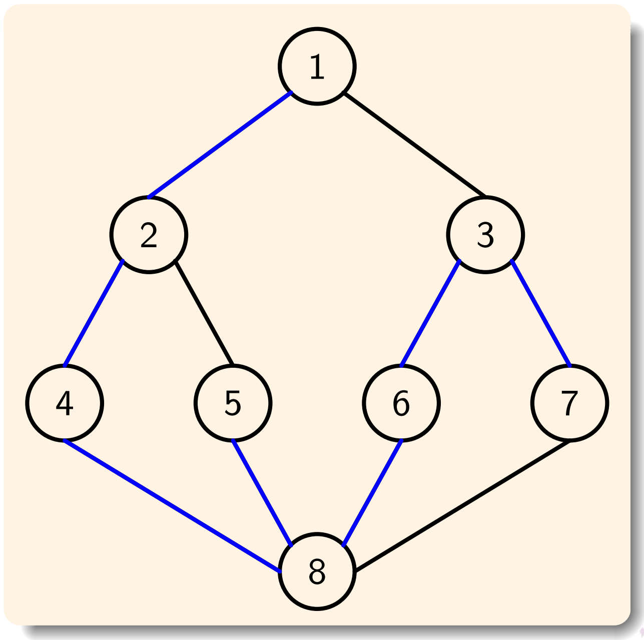
- Another popular graph traversal algorithm is

Algorithm 4.2.1. Depth First Search

```
// Depth first search starting from vertex $v$ of graph $G$.
// Input: starting node $v$
// Output: none.
Algorithm DFS($v$)
{
    $visited[v] := 1$ ; // The rest of $v$ array initialized to 0.
    for each vertex $w$ adjacent to $v$ do {
        if ($visited[w] = 0$) then DFS($w$) ;
    }
}
```

- Note shorter codes than BFS algorithm.
  - No queue is needed.

DFS Example



- DFS calling sequence

```
DFS 1
    DFS 2
        DFS 4
            DFS 8
                DFS 5
                DFS 6
                    DFS 3
                        DFS 7
```

- Larger recursion depth than Breadth first search
- Larger path lengths than BFS as well

Depth First Search – Properties

Theorem 4.2.2. DFS Reachability

Algorithm DFS visits all vertices of G reachable from v.

- Proof by induction.

Theorem 4.2.3. DFS Complexities

Let $T(n, e)$ and $S(n, e)$ be the maximum time and maximum *additional* space taken by algorithm DFS on any graph G wit n vertices and e edges.

1. $T(n, e) = \Theta(n + e)$ and $S(n, e) = \Theta(n)$ if G is represented by its adjacency lists,
2. $T(n, e) = \Theta(n^2)$ and $S(n, e) = \Theta(n)$ if G is represented by its adjacency matrix.

- The overall space time complexity is $\Theta(n + e)$ in case of adjacency list, and $\Theta(n^2)$ in case of adjacency matrix.
- The depth first search can be applied to both undirected and directed graphs.

Spanning Trees of Connected Graphs

- The DFS algorithm can be modified to find the spanning tree of a connected graph.

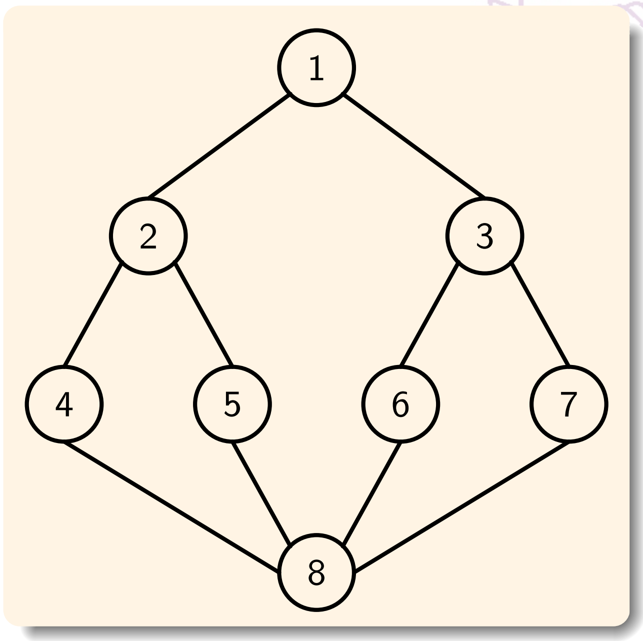
Algorithm 4.2.4. DFS to find a spanning tree

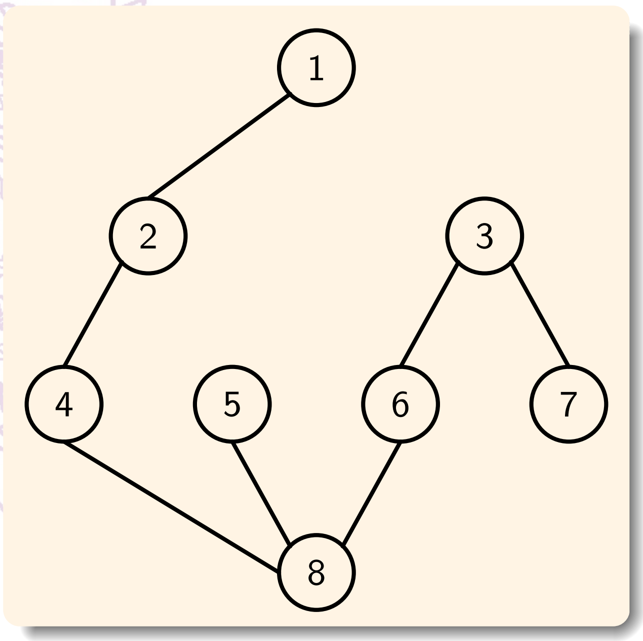
```
// Depth first search to find the spanning tree from vertex $v$.
// Input: starting node $v$
// Output: spanning tree $t$.
Algorithm DFS*($v, t$)
{
    $visited[v] := 1$;
    $t := \emptyset$; // $t$ initialized to $\emptyset$.
    for each vertex $w$ adjacent to $v$ do {
        if ($visited[w] = 0$) then {
            $t := t \cup \{(v, w)\}$; // add to spanning tree.
            DFS*($w$);
        }
    }
}
```

- On termination, t is the set of edges that forms a spanning tree of G.

DFS Spanning Tree

- The spanning tree found by Algorithm DFS* can be called DFS spanning tree.
- Example





- The time and space complexity of DFS* is the same as DFS.

Depth First Search Forest

- In some applications, depth-first search is applied to graphs that the results are not a single tree but forests.
    - More often in directed graphs.
- Thus, the initial calling sequence of the depth-first search should be as following.

Algorithm 4.2.5. Calling DFS

```
    // Initialization and recursive DFS function call.
    // Input: graph $G$
    // Output: arrays $p[|V|]$: predecessor, $d[|V|]$: discover time, $f[|V|]$: finish time.
Algorithm DFS_Call($G$)
{
    for $v := 1$ to $n$ do { // Initialize to not visited and no predecessor.
        $visited[v] := 0$;
        $p[v] := 0$;
        $d[v] := 0$;
        $f[v] := 0$;
    }
    $time := 0$; // Global variable to track $time$.
    for $v := 1$ to $n$ do { // To handle forest case.
        if ($visited[v] = 0$) then DFS_d($v$);
    }
}
```

Depth First Search Structure

- The execution of the depth-first search yields a special structure as shown in the example in the preceding page.
- To more explicitly recording the structure of the depth-first search tree, the DFS(v) algorithm is modified as the following.
- Assuming $G = (V, E)$ and $|V| = n$ and $|E| = e$.
- The $visited[1:n]$ array, which is initialized to 0, has three values
    - $visited[u] = 0$ if vertex u has not been visited.
    - $visited[u] = 1$ if vertex u is being visited (visiting its successors).
    - $visited[u] = 2$ if vertex u has been visited already.
- Array $d[1:n]$ and $f[1:n]$ are added. Each $d[u]$ records the discover time, when vertex u is first being visited; And each $f[u]$ records the time when visiting vertex u is completed.
- Array $p[1:n]$ still records the predecessor of vertex u in $p[u]$.

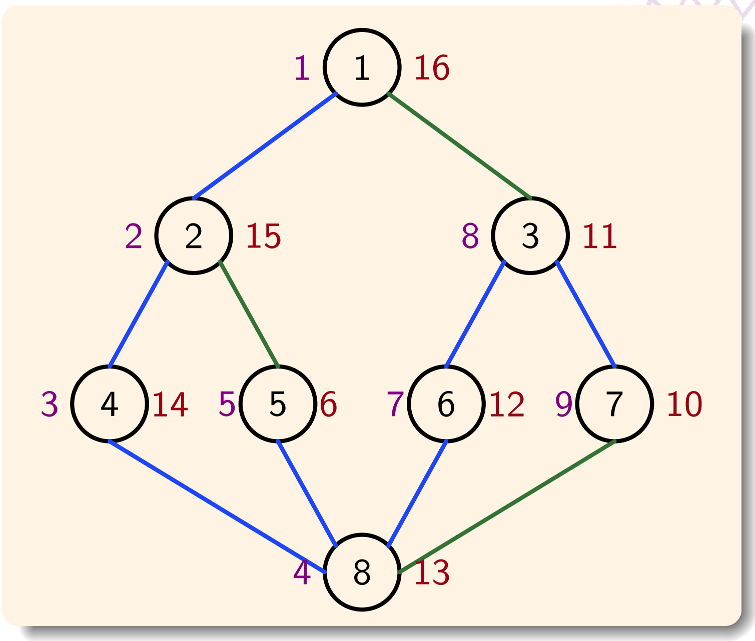
More Sophisticated Depth-first Search

Algorithm 4.2.6. More sophisticated depth-first search

```
// More sophisticated Depth first search starting from vertex $v$ of graph $G$.
// Input: starting node $v$
// Output: arrays $p[|V|]$: predecessor, $d[|V|]$: discover time, $f[|V|]$: finish time.
Algorithm DFS_d($v$)
{
    $visited[v] := 1$;
    $time := time + 1$;
    $d[v] := time$;
    for each vertex $w$ adjacent to $v$ do {
        if ($visited[w] = 0$) then {
            $p[w] := u$;
            DFS_d($w$);
        }
    }
    $visited[v] := 2$;
    $t := time + 1$;
    $f[v] := time$;
}
```

- Array *visited* has three states: 0, unvisited; 1, visiting; 2, visited.
- Array d records discover time and array f records finish time.
- Array p records the preceding vertex in the DFS path.
- *time* is a global variable to keep track of discovery and finish times.

DFS Example Revisit



- DFS_d calling sequence

```
DFS_d 1 $d[1] = 1$
  DFS_d 2 $d[2] = 2$
    DFS_d 4 $d[4] = 3$
      DFS_d 8 $d[8] = 4$
        DFS_d 5 $d[5] = 5$
        DFS_d 5 $f[5] = 6$
        DFS_d 6 $d[6] = 7$
          DFS_d 3 $d[3] = 8$
            DFS_d 7 $d[7] = 9$
            DFS_d 7 $f[7] = 10$
          DFS_d 3 $f[3] = 11$
        DFS_d 6 $f[6] = 12$
      DFS_d 8 $f[8] = 13$
    DFS_d 4 $f[4] = 14$
  DFS_d 2 $f[2] = 15$
DFS_d 1 $f[1] = 16$
```

Properties of Depth First Search Structure

Theorem 4.2.7. Parenthesis theorem

After applying depth-first search to a graph $G = (V, E)$, for any two vertices $u, v \in V$, exactly one of the following three conditions holds:

1. The intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,
2. The interval $[d[u], f[u]]$ is contained entirely with the interval $[d[v], f[v]]$ and u is a descent of v is a depth-first tree, or
3. The interval $[d[v], f[v]]$ is contained entirely with the interval $[d[u], f[u]]$ and v is a descendant of u is a depth-first tree.

- If vertex u is a descendant of of vertex v in a DFS forest, then v is visited before u, while u finished before v, thus, the interval $[d[u], f[u]]$ is contained entirely in $[d[v], f[v]]$.
- On the other hand, if v is a descendant of u the same argument proves the third case.
- If u is not a descendant nor an ancestor of v, then the traversal of the tree rooted by u is either before or after traversing tree rooted by v, therefore those two intervals are disjoint.

Properties of Depth First Search Structure, II

- An immediate corollary of the preceding theorem follows.

Corollary 4.2.8. Nesting of descendants' intervals

Vertex v is a proper descendant of vertex u if a depth-first forest for a graph $G = (V, E)$ if and only if $d[u] < d[v] < f[v] < f[u]$.

Theorem 4.2.9. White-path theorem

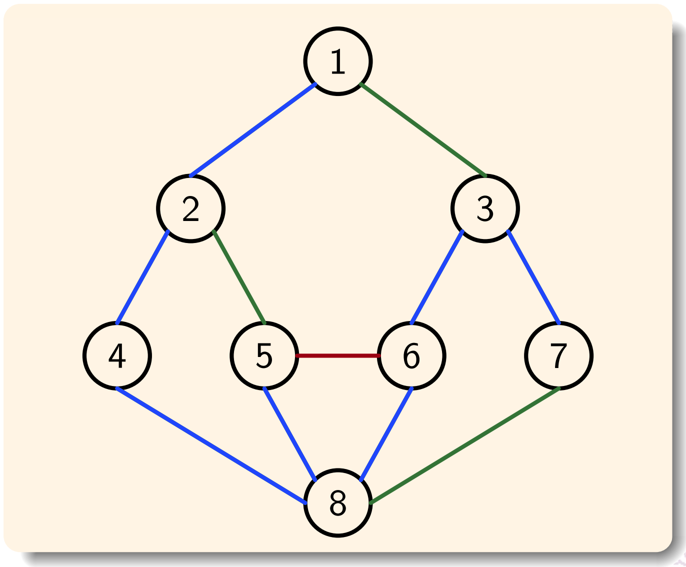
In a depth-first forest of a graph $G = (V, E)$, vertex v is a descendant of vertex u if and only if at the time $d[u]$ that u is discovered, there is a path from u to v consisting entirely of vertices with zero *visited* array values.

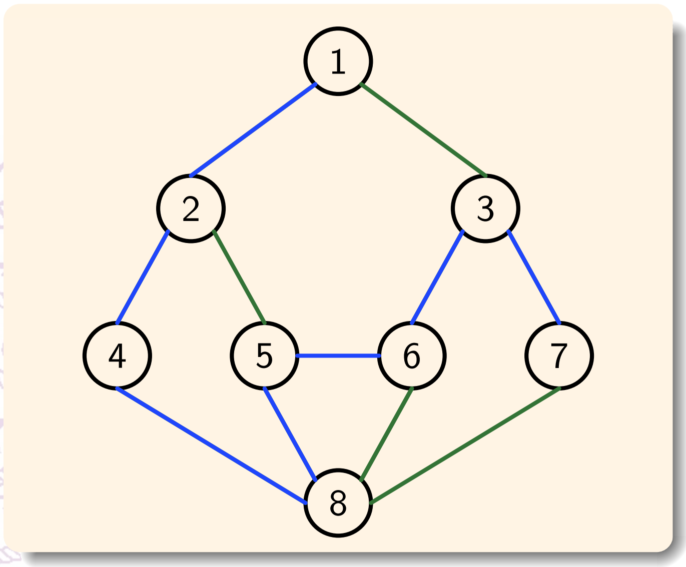
- If v is a descendant of u, then v must be visited after u hence the vertices along the path (which exists due to the descendant relationship) are all not visited ($d[\,] = 0$). On the other hand, when u is discovered and there is a path consists of vertices with zero discover time from u to v, this means that v is reachable from u and v is not visited. It follows that v is a descendant of u.

Properties of Depth First Search Structure, III

- Four types of edges can be found in a graph $G = (V, E)$ after the depth-first forest is found.
  1. Tree edges are those edges (u, v) where v is a immediate descendant of u.
  2. Back edges are those edges (u, v) connecting a vertex u to an ancestor v.
  3. Forward edges are those edges (u, v) connecting a vertex u to a descendant v.
  4. Cross edges are all other edges. They connect vertices of the same DFS tree as long as one vertex is not an ancestor of the other, or they connect vertices of different DFS trees.
- At line 4 of the DFS_d(v) algorithm (4.2.6) when an edge (v, w) is checked
  - If $visited[w] = 0$ then (v, w) is a tree edge.
  - If $visited[w] = 1$ then (v, w) is a back edge.
  - If $visited[w] = 2$ then (v, w) is a forward or cross edge.
- In an undirected graph, if (u, v) is a forward edge, then (v, u) is a back edge. But these two are the same edge.
  - The type of the edge (u, v) is defined by whichever of (u, v) or (v, u) is checked first in the DFS (v) algorithm.

Properties of Depth First Search Structure, IV





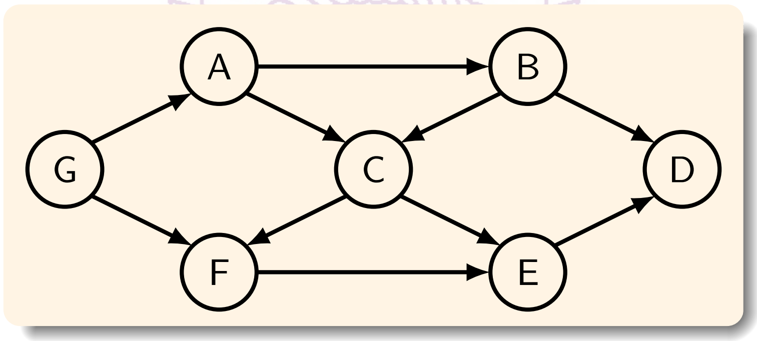
Theorem 4.2.10.

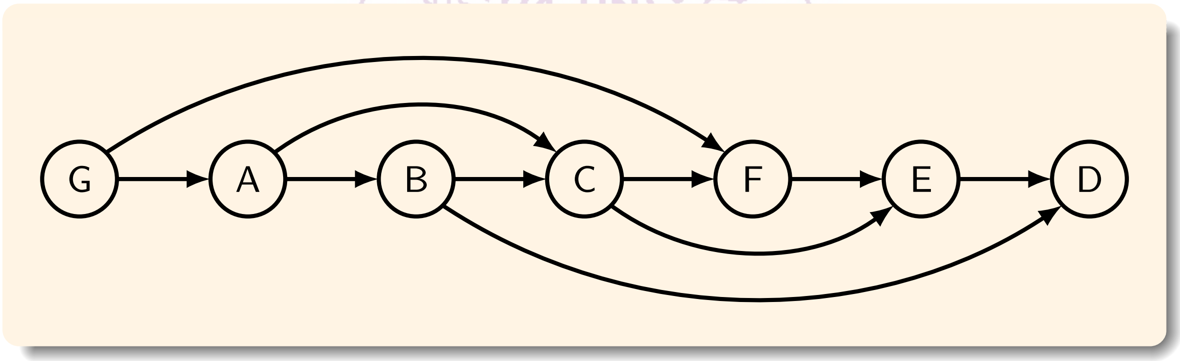
In a depth-first search of an undirected graph $G = (V, E)$, every edge of G is either a tree edge or a back edge.

- As shown in the example above, cross edges do not exist. Therefore, all edges are either tree edges or forward (backward) edges.
- Proof can also be found in textbook [Cormen], p. 610.

Topological Sort

- Given a directed acyclic graph (dag), $G = (V, E)$, a topological sort is a linear order of all the vertices such that if $\langle u, v\rangle \in E$ then u appears before v in the ordering.


$$\begin{tikzcd}
 & A \arrow[rr] \arrow[dr] & & B \arrow[dl] \arrow[dr] & \\
G \arrow[ur] \arrow[dr] & & C \arrow[dl] \arrow[dr] & & D \\
 & F \arrow[rr] & & E \arrow[ur] &
\end{tikzcd}$$


$$\begin{tikzcd}
G \arrow[r] \arrow[rrrr, bend left] & A \arrow[r] \arrow[rr, bend left] & B \arrow[r] \arrow[rrrrr, bend right] & C \arrow[r] \arrow[rr, bend right] & F \arrow[r] & E \arrow[r] & D
\end{tikzcd}$$

Topological Sort – Algorithm

- The depth-first search algorithm is ideal to solve the topological sort problem.

Algorithm 4.2.11. Topological sort

```
   // Topological sort using depth-first search algorithm.
   // Input: $v$ starting vertex
   // Output: $slist$ sorting result.
1 Algorithm top_sort($v, slist$)
2 {
3       $visited[v] := 1$;
4       for each vertex $w$ adjacent to $v$ do {
5           if $(visited[w] = 0)$ then top_sort($w$);
6       }
7       add $v$ to the head of $slist$;
8 }
```

- The modifications to the depth-first algorithm is to add the current vertex to the ordered linked list at the final time.
- After completion of the algorithm, the ordered list $slist$ is the answer.
- As in the case of the recursive depth-first search algorithm, this algorithm should be called by the following algorithm.

Topological Sort – Algorithm, II

Algorithm 4.2.12. Topological sort call

```
   // Initialization and recursive top_sort function call.
   // Input: graph $G$
   // Output: $slist$ sorted result.
 1 Algorithm topsort_Call($G, slist$)
 2 {
 3       for $v := 1$ to $n$ do $visited[v] := 0$ ;
 4       $slist :=$ NULL ;
 5       for $v := 1$ to $n$ do
 6           if ($visited[v] = 0$) then top_sort($v, slist$) ;
 7 }
```

- The top_sort($v, slist$) algorithm visits every vertex of G once and each edge is checked on line 4 of Algorithm (4.2.11) once.
- Inserting a node to the head of a linked list takes $\mathcal{O}(1)$ time.
- Thus, the computational complexity of top_sort($v, slist$) is $\Theta(n + e)$, where $n = |V|$ and $e = |E|$.

Topological Sort – Correctness

- The following lemma and theorem show the correctness of the topological sort algorithm.

Lemma 4.2.13.

A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.

- If a back edge exist then there is a cycle, and vice versa.
- Proof can also be found in textbook [Cormen], p. 614.

Theorem 4.2.14.

The top_sort algorithm (4.2.11) produces a topological sort of the directed acyclic graph given its input.

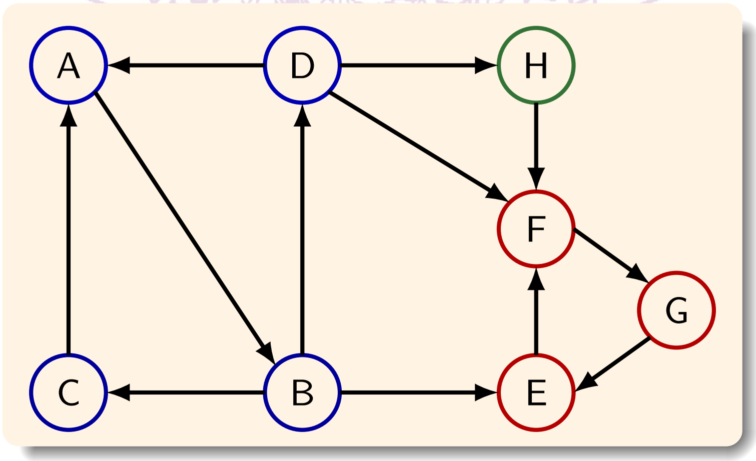
- As shown in the algorithm, if there is an edge $\langle v, w \rangle$ then v is visited first and has a smaller discover time, but w is visited later with a smaller finish time $f(w) < f(v)$. Hence v is added to the head of the list after w. This proves that v is order before w as required.
- Proof can also be found in textbook [Cormen], p. 614.

Strongly Connected Components

Definition 4.2.15.

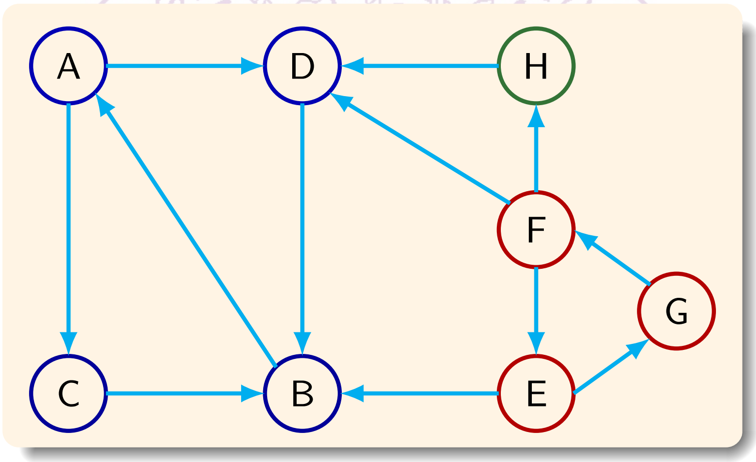
Given a directed graph $G = (V, E)$ a strongly connected component is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v they are mutual reachable; that is, vertex u is reachable from v and vice versa.

- Example. There are three strongly connected components in the graph below: $\{A, B, C, D\}$, $\{E, F, G\}$, and $\{H\}$.



Strongly Connected Components – Transpose Graph

- The algorithm to find the strongly connect graph uses the transpose graph G^T of $G = (V, E)$. $G^T = (V, E^T)$ where $E^T = \{\langle u, v\rangle | \langle v, u\rangle \in E\}$.
- E^T is E with the directions of all edges reversed.
- Given $G = (V, E)$, with $n = |V|$ and $e = |E|$ then G^T can be constructed in $\mathcal{O}(n + e)$ if G is represented using adjacency-list representation.
- G and G^T have the same strongly connected components since if $\langle u, v\rangle$ are reachable from each other in G then they are reachable from each other in G^T.



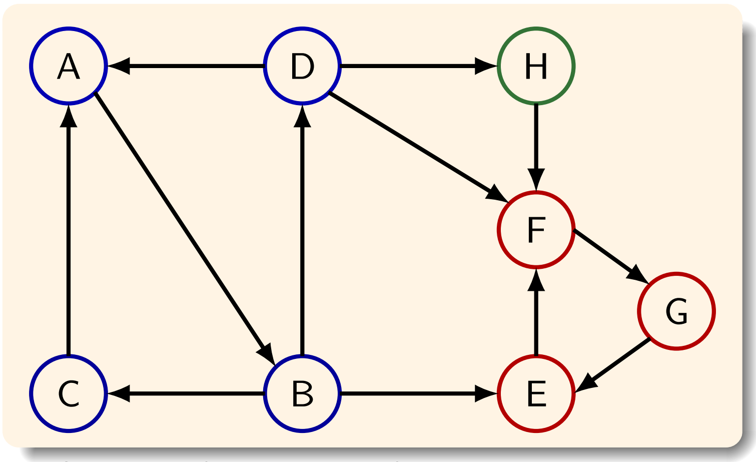
Strongly Connected Components – Algorithm

Algorithm 4.2.16. Strongly Connected Components

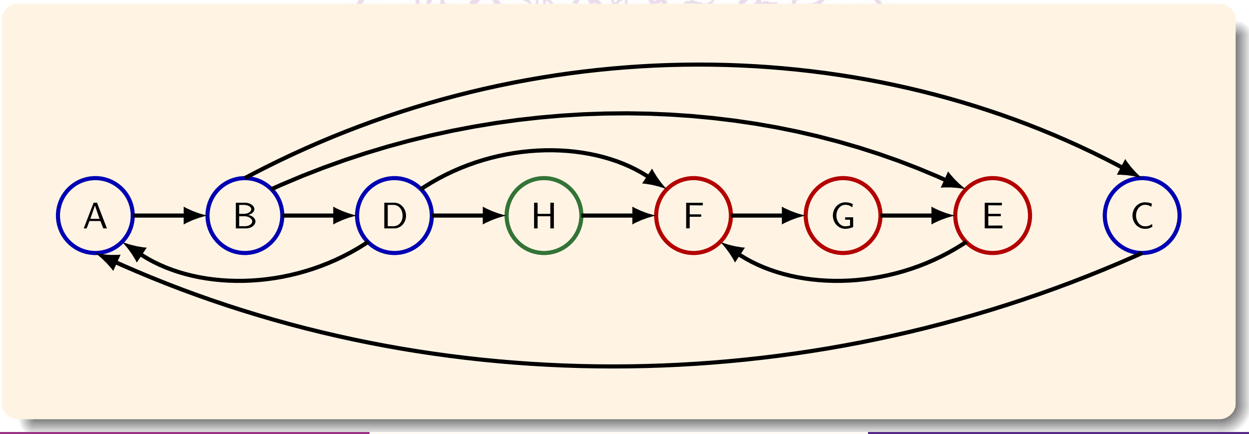
```
   // To find the strongly connected components of the graph $G = (V, E)$.
   // Input: graph $G$
   // Output: strongly connected components.
1  Algorithm SCC($G$)
2  {
3        Construct the transpose graph $G^T$;
4        DFS_Call($G$); // Perform DFS to get array $f[1 : n]$.
5        Sort $V$ of $G^T$ in order of decreasing value of $f[v]$, $v \in V$.
6        DFS_Call($G^T$); // Perform DFS on $G^T$.
7        Each tree of the resulting DFS forest is a strongly connected component.
8  }
```

- Note that lines 4 and 5 are essentially performing topological sort on the vertices, V.
- Two depth-first searches are performed in this algorithm, thus the computational complexity is $\Theta(n + e)$ assuming adjacency-list is used.

Strongly Connected Components – Example

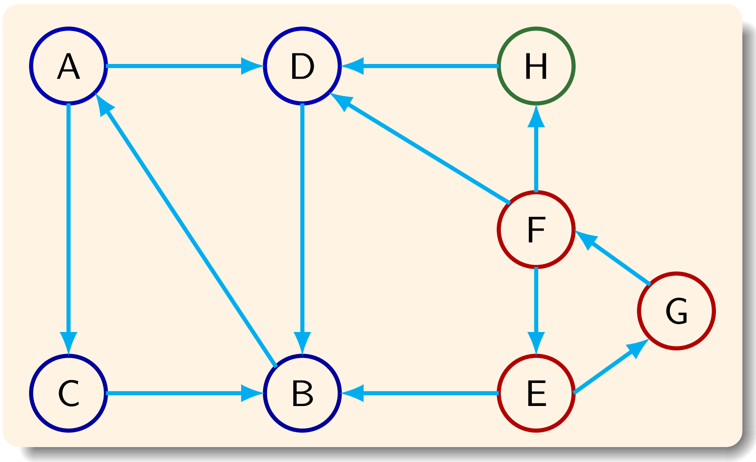


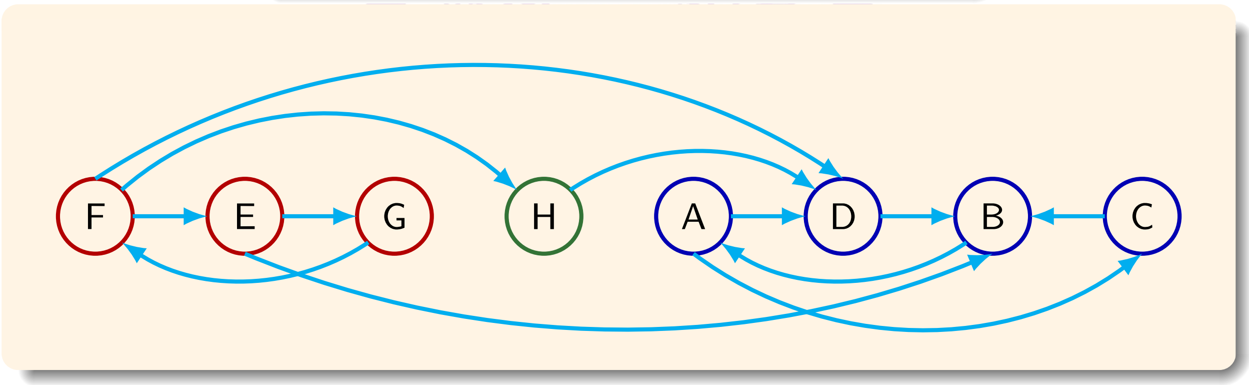
- Topological sort after the first depth-first search.



Strongly Connected Components – Example, II

- The second depth-first search is applied to G^T as the following.
- Thus, $SCC_1 = \{A, B, C, D\}$, $SCC_2 = \{H\}$, and $SCC_3 = \{E, F, G\}$.



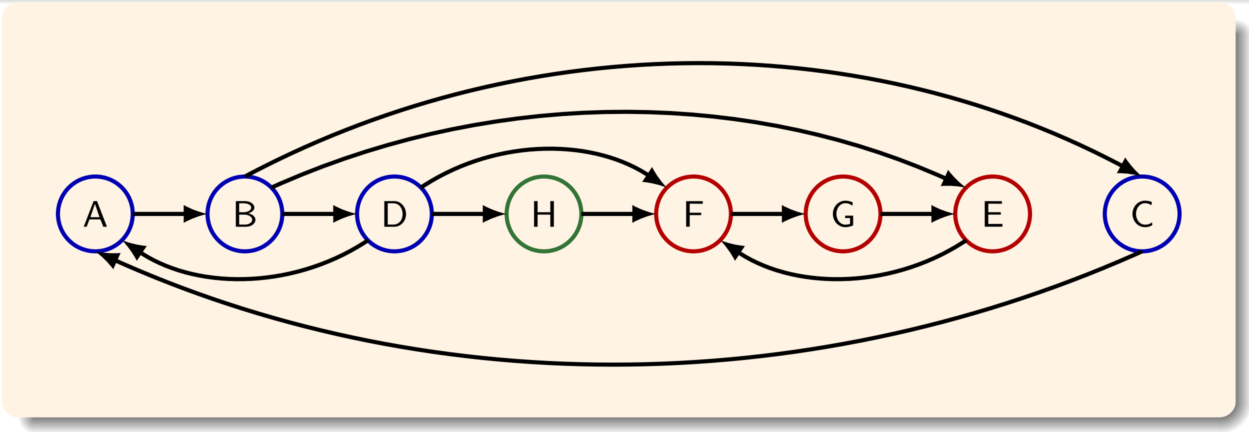


Strongly Connected Components – Properties

Lemma 4.2.17.

Let C and C' be two distinct strongly connect components in a directed graph $G = (V, E)$. If $u, v \in C$, $u', v' \in C'$ and there is a path from u to u' then G cannot contain a path from v' to v.

Proof. If G contains a path from v' to v. Since u, v are reachable from each other, v' can then reach u; while u can already reach u' and hence v', u and v' are then in the same strongly connected component. This contradicts to the assume that they are in distinct strongly connect components. □



Strongly Connected Components – Properties

Definition 4.2.18.

Given a directed graph $G = (V, E)$, then after applying the depth-first search on G, we have $1 \le f(v) \le 2 \times |V|$, which is the finish time for each vertex, v, if $C \subseteq V$ define

$$f(C) = \max_{v \in C} f(v), \tag{4.2.1}$$

$f(C)$ is the largest finish time for all the vertices in C.

Lemma 4.2.19.

Let C and C' be two distinct strongly connect components in a directed graph $G = (V, E)$. Suppose there is an edge $\langle u, v \rangle \in E$, where $u \in C$ and $v \in C'$. Then, $f(C) > f(C')$.

- If a vertex $w \in C$ is visited first, since there an edge $\langle u, v \rangle$, $u \in C$, $v \in C'$, then all vertices in C' are descendants of w. Therefore $f(C) \ge f(w) > f(C')$.
- On the other hand, if $w \in C'$ is visited first, then none of the vertices in C can be traverse by a single `top_sort`$(w, slist)$ call. Instead, they can only be visited afterward, line 6 of `topsort_Call`. Hence, $f(C) > f(C')$.
- Proof can also be found in textbook [Cormen], p. 618.

Strongly Connected Components – Properties, II

Lemma 4.2.20.

Let C and C' be two distinct strongly connect components in a directed graph $G = (V, E)$. Suppose there is an edge $\langle u, v \rangle \in E^T$, where $u \in C$ and $v \in C'$. Then, $f(C) < f(C')$.

Proof. Since $\langle u, v \rangle \in E^T$, $\langle v, u \rangle \in E$. Because the strongly connect components of G and G^T are the same, from Lemma (4.2.19) we have $f(C) < f(C')$. $\square$

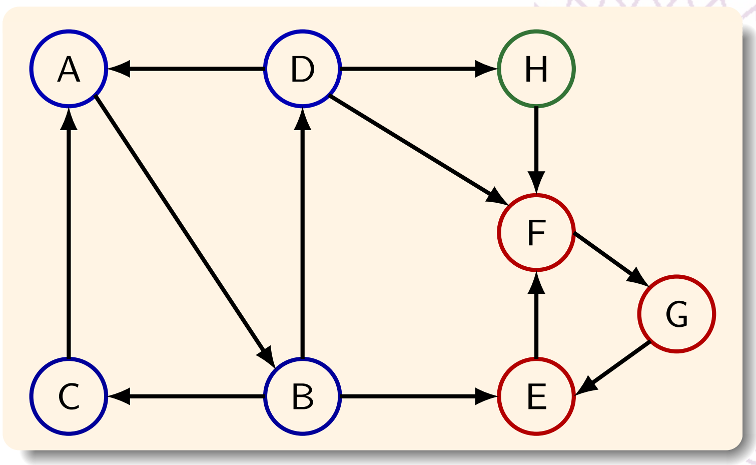
- From the preceding lemmas, we have the following theorem.

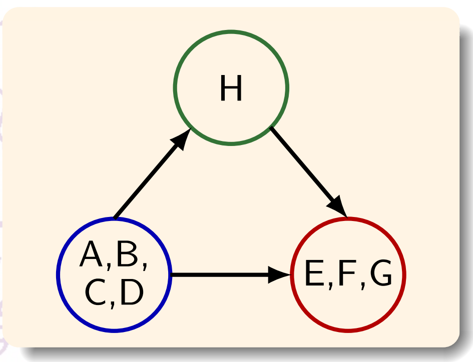
Theorem 4.2.21.

The Algorithm (4.2.16) correctly computes the strongly connected components of the directed graph G given as an input.

- Proof can also be found in textbook [Cormen], pp. 619-620.

Strongly Connected Components – Properties, III





- Vertices of a strongly connected components can be collapsed into a single node to form a component graph, G^{SCC}, as shown.

Biconnected Graphs

- In this subsection, only undirected graphs are studied.

Definition 4.2.22.

A vertex v of a connected graph G is an articulation point if and only if the deletion of vertex v together with all edges connecting to v disconnects G into two or more nonempty components.

Definition 4.2.23.

A graph G is biconnected if and only if it contains no articulation points.

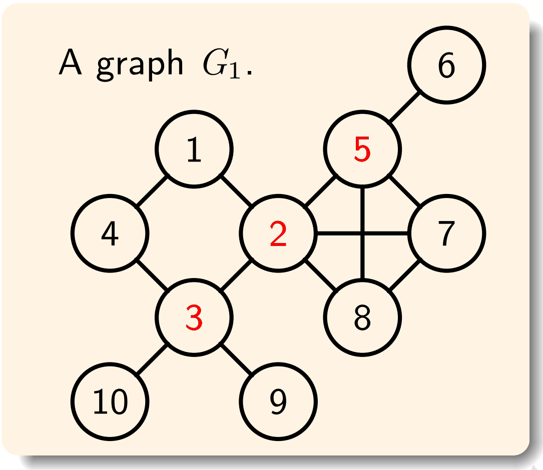
Definition 4.2.24.

$G' = (V', E')$ is a maximal biconnected components of $G = (V, E)$ if there is no biconnected subgraph $G'' = (V'', E'')$ such that $V' \subseteq V''$ and $E' \subset E''$.

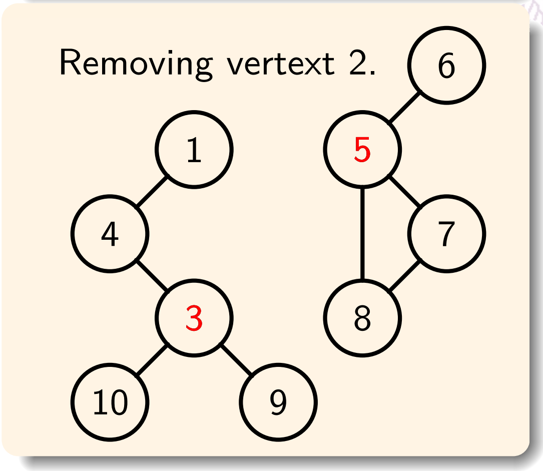
- A communication network is more robust if the underlying graph is biconnected.

Biconnected Graphs — Examples

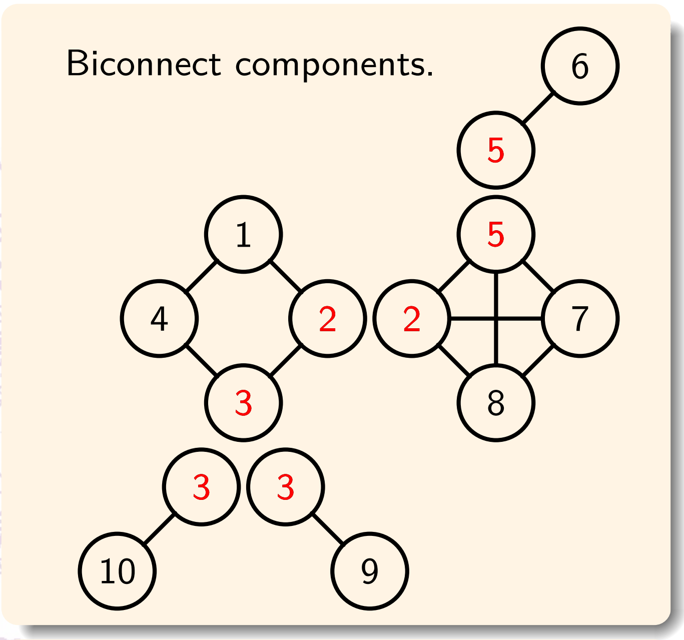
A graph G_1.



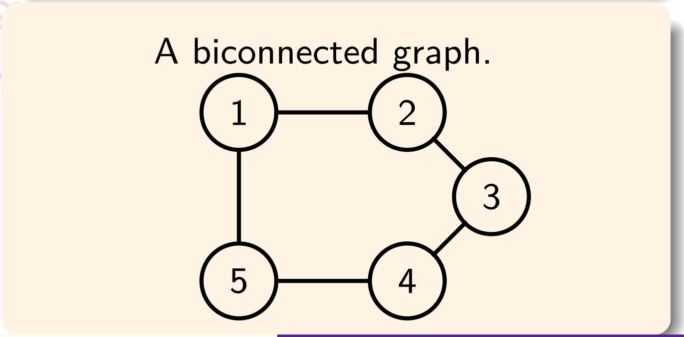
Removing vertext 2.



Biconnect components.



A biconnected graph.



Biconnected Components

Lemma 4.2.25.

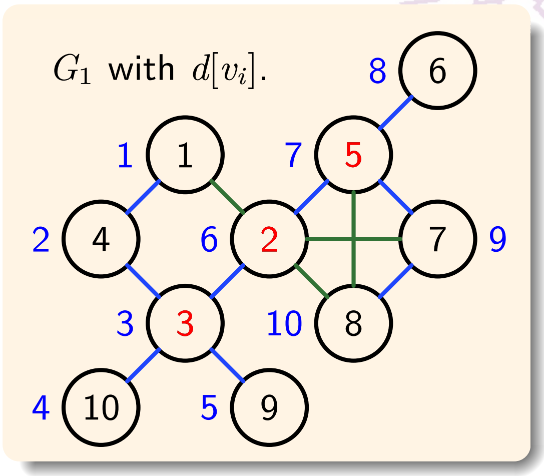
Two biconnected components can have at most one vertex in common and this vertex is an articulation point.

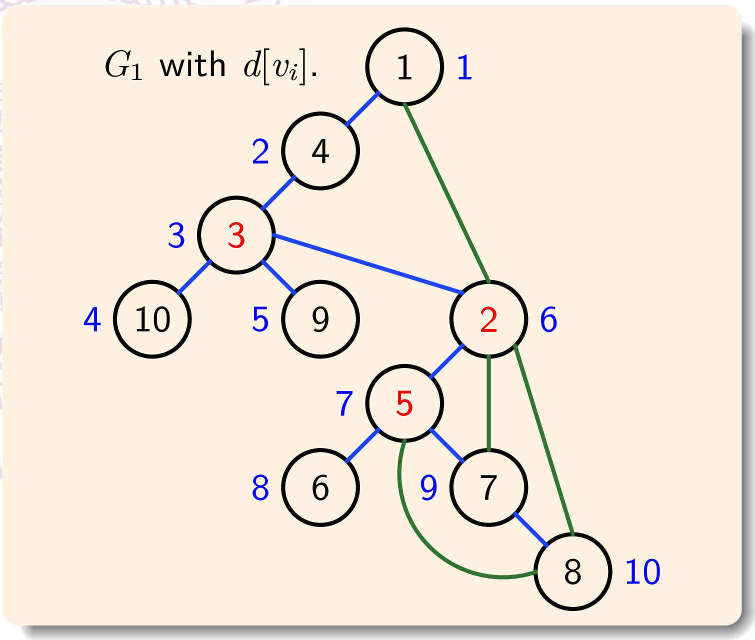
- There is no edge between two biconnected components.
- For the graph G_1,
  - Adding edges $(4, 10)$ and $(10, 9)$, eliminates articulation point 3.
  - Adding edge $(1, 5)$ eliminates articulation point 2.
  - Adding edge $(6, 7)$ eliminates articulation point 5.
- The following algorithm eliminates all articulation points

```
for each articulation point $a$ do {
    Let $B_1, B_2, \cdots, B_k$ be the biconnected components containing vertex $a$;
    Let $v_i$, $v_i \neq a$, be a vertex in $B_i$, $1 \leq i \leq k$;
    Add to $G$ edges $(v_i, v_{i+1})$, $1 \leq i < k$;
}
```

DFS and Biconnected Components

- Depth first search is useful in identifying the articulation points in a graph.
- Perform a depth first search on G and let $d[v_i]$ of a vertex, v_i, be the number corresponds to the order of the depth first tree visits the vertex.





- Tree edges are shown in blue; and back edges in green.

DFS and Biconnected Components — Properties

Lemma 4.2.26.

If (u, v) is any edge in G, then relative to the depth first spanning tree t, either u is an ancestor of v or v is an ancestor of u. So, there are no cross edges relative to a depth first spanning tree. (Note that (u, v) is a cross edge relative to t if and only if u is not an ancestor of v and v is not an ancestor of u).

Lemma 4.2.27.

The root node of a depth first spanning tree is an articulation point if and only if it has at least two children. Furthermore, if u is any other vertex, then it is not an articulation point if and only if from any child w of u it is possible to reach an ancestor of u using only a path made up of descendants of w and a back edge.

- Define

$$L[u] = \min\Big\{ d[u],\ \ \min\{L[w] \mid w \text{ is a child of } u\},\ \min\{d[w] \mid (u,w) \text{is a back edge}\}\Big\}. \tag{4.2.2}$$

then if u is not a root then u is an articulation point if and only if u has child w such that $L[w] \geq d[u]$.

Articulation Point Algorithm

- The following algorithm calculates $L[u]$.
    - Array d is initialized to 0, and global variable num initialized to 1.

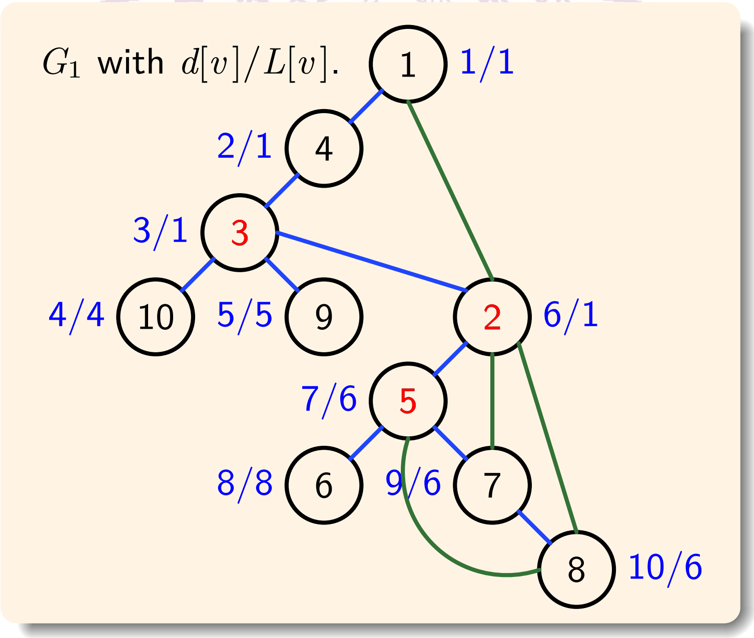
Algorithm 4.2.28. Calculate $L[u]$

```
   // Given a graph to calculate $L[u]$.
   // Input: $u$ and its parent $v$
   // Output: $L[u]$.
 1 Algorithm Art($u$, $v$)
 2 {
 3       $d[u] := num$;
 4       $L[u] := num$;
 5       $num := num + 1$;
 6       for each vertex $w$ adjacent to $u$ do {
 7            if ($d[w] = 0$) then {
 8                 Art($w$, $u$);
 9                 $L[u] :=$ min($L[u]$, $L[w]$);
10            }
11            else if ($w \neq v$) then $L[u] :=$ min($L[u]$, $d[w]$);
12       }
13 }
```

- $L[w]$ is the smallest $d[v]$ that v is connected to w.
    - Thus if $L[w] < d[u]$ there is an edge connecting w and an ancestor of u, and eliminating node u does not isolate vertex w.
    - On the other hand, if $L[w] \geq d[u]$ then removing vertex u disconnects w from the tree, in case that u is not the root.

Articulation Point Example

- Once array L is found, the articulation points can be identified in $\mathcal{O}(n)$ time, where n is the number of vertices in G.
- The algorithm Art has the time complexity of $\mathcal{O}(n + e)$, where e is the number of edges in G.
- The time complexity of find all the articulation points is $\mathcal{O}(n + e)$.



Finding All Biconnected Components

Algorithm 4.2.29. Finding all biconnected components

```
   // Given a graph $G$ find all biconnected components.
   // Input: $u$ and its parent $v$
   // Output: All biconnected components.
Algorithm BiComp($u$, $v$)
{
    $d[u] := num$ ; // init $d[u]$
    $L[u] := num$ ; // init $L[u]$
    $num := num + 1$ ; // tracking discover time
    for each vertex $w$ adjacent to $u$ do { // all edges
        if (($w \neq v$) and ($d[w] < d[u]$)) then Push(($u$, $w$)) to stack $S$ ; // save edge to stack
        if ($d[w] = 0$) then { // w not visited
            BiComp($w$, $u$) ; // visit w first
            if ($L[w] \geq d[u]$) then { // $u$ is an articulation point
                write ("$New\ bicomponent$ : ") ; // print out biconnected components
                repeat { //                          using stack
                    $(x, y)$ := Pop() from stack $S$ ;
                    write $(x, y)$ ;
                } until ((($x, y$) = ($u, w$)) or (($x, y$) = ($w, u$))) ;
            }
            $L[u]$ := min($L[u]$, $L[w]$) ; // Update $L[u]$
        }
        else if ($w \neq v$) then $L[u]$ := min($L[u]$, $d[w]$) ; // update $L[u]$
    }
}
```

Finding All Biconnected Components — Correctness

Theorem 4.2.30.

Algorithm (4.2.29) correctly generates the biconnected components of the connected graph G when G has at least two vertices.

- Proof please see textbook [Horowitz], pp. 355-356.
- In discussing articulation points and biconnected graphs, it is assumed no cross edges in the spanning tree. However, since breadth first spanning trees can have cross edges, thus algorithm `Art` cannot be adapted to `BFS`.

Summary

- Depth first search
- Topological sort
- Strongly connected components
- Biconnected graphs