Graph Algorithm

IOI Training Oct. 2019

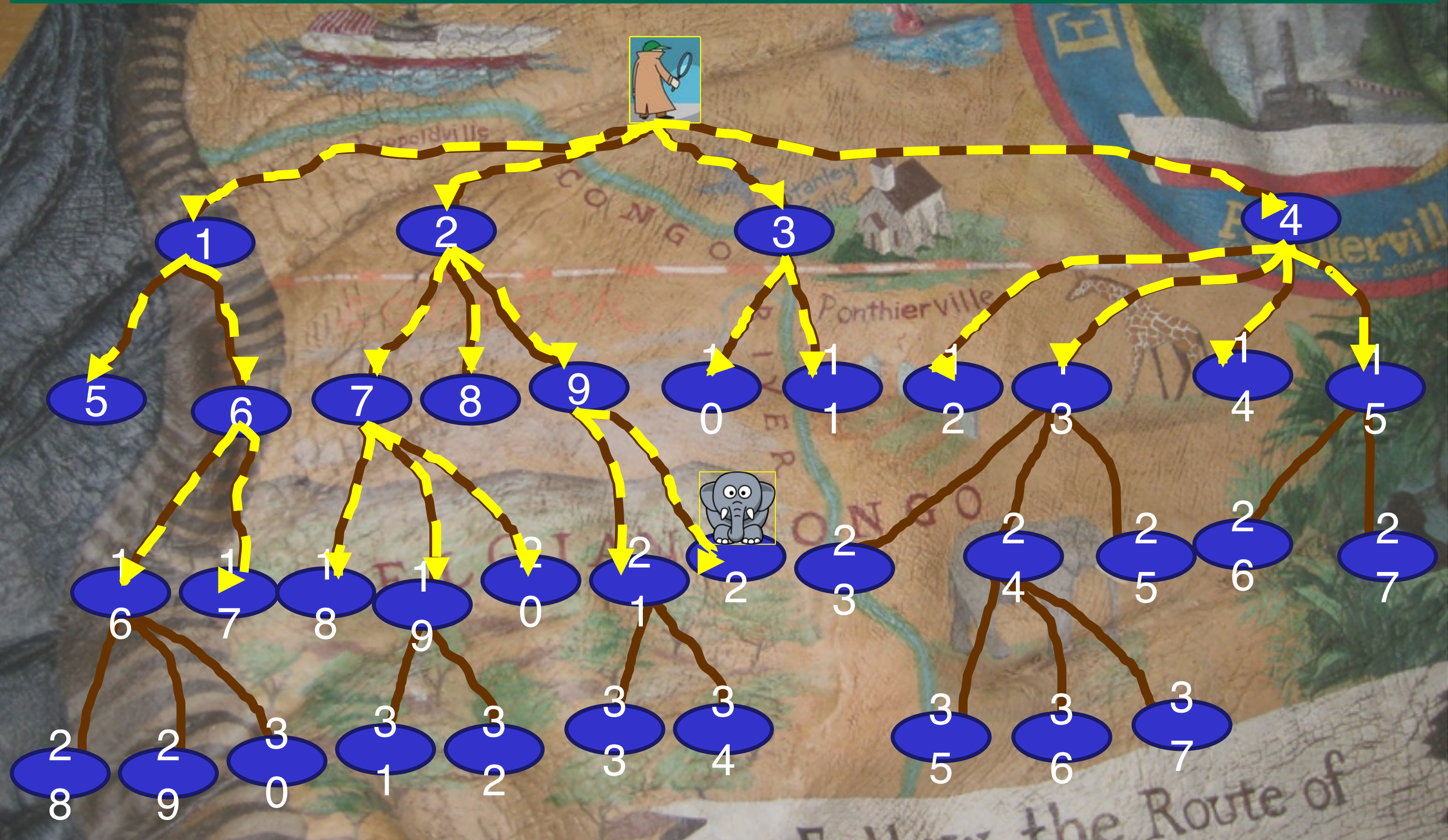
HOW DO WE KNOW IF THE GRAPH IS A CONNECTED GRAPH?

Traversals

- Visit each node – e.g., web crawler
- Have to restart traversal in each connected component, but this allows us to identify components
- Reachability in a digraph is an important issue – the transitive closure graph

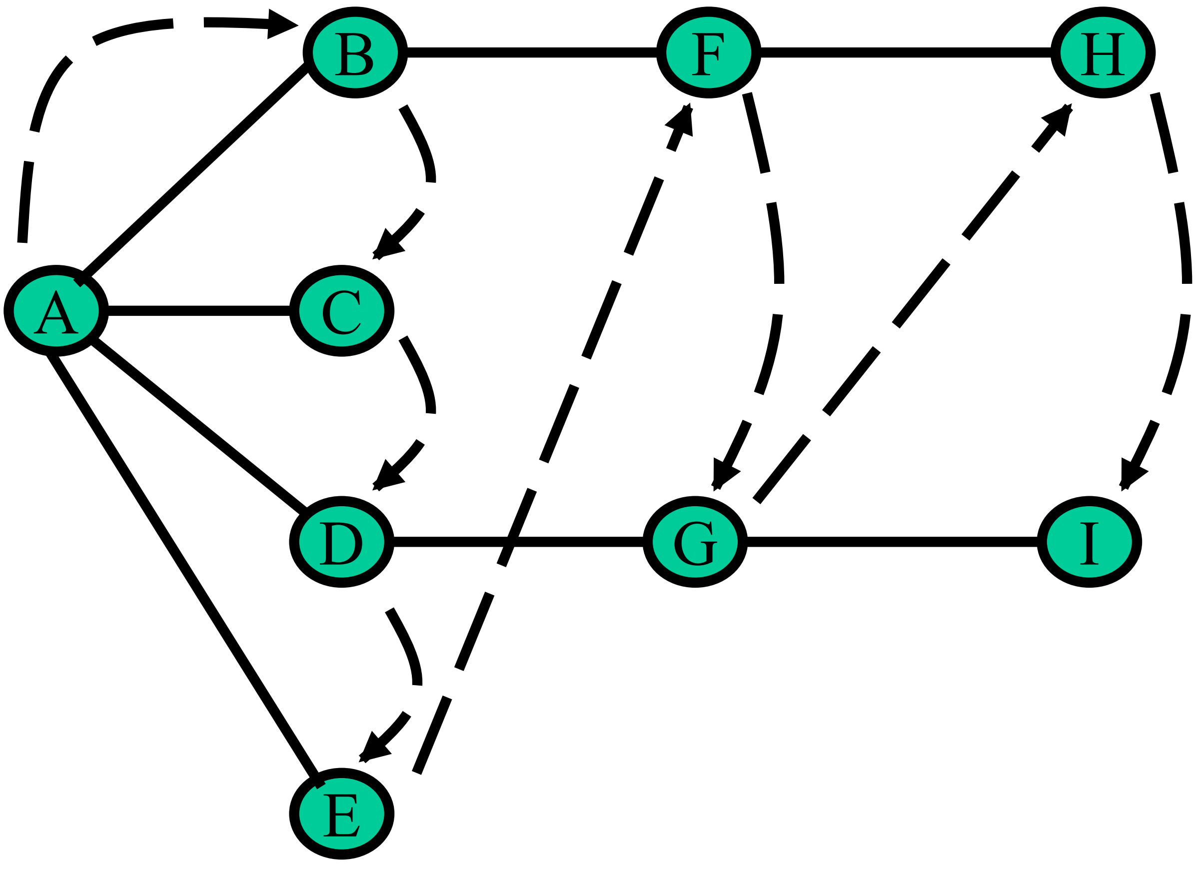
Finding the elephant Game

You are on a safari trip and want to find an elephant. The only thing you know about the jungle is your neighboring locations. There are many ways to find the elephant. You have a special ability to apparate back to where you already visited.

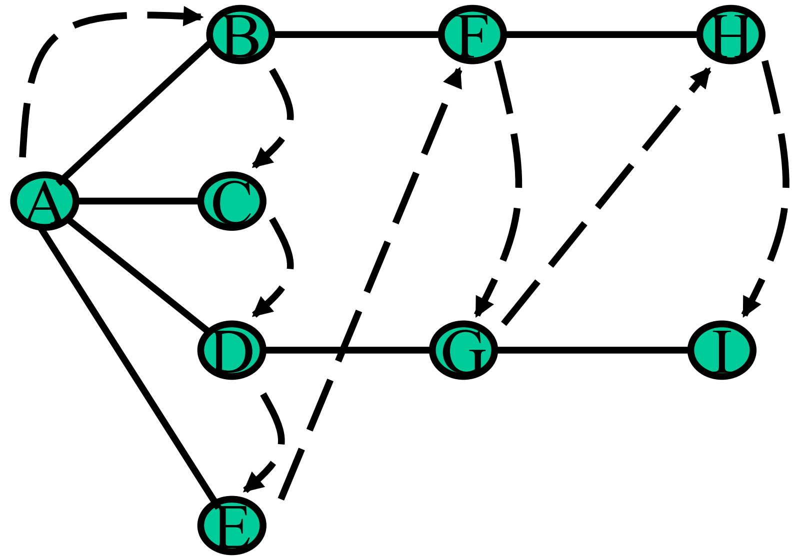


http://i14.photobucket.com/albums/a334/khellekson/disney-2008/Adventure-Jungle-Congo-Map.jpg

Breadth-first search



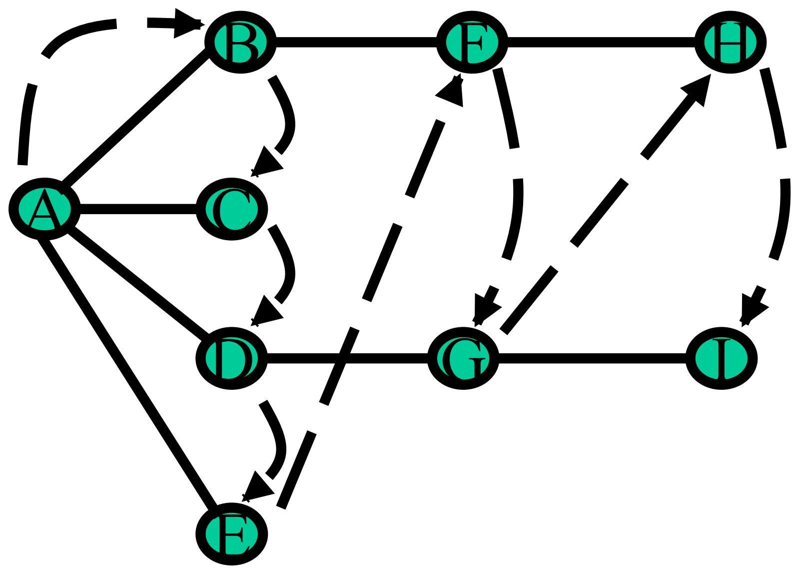
Breadth-first search (cont.)



- Use queue as our tool
- Every time we visit each vertex, insert it into the queue

Breadth-first search (cont.)

Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Remove B	CDE
Visit F	CDEF
Remove C	DEF
Remove D	EF
Visit G	EFG
. .	



First-in-First-Out

BFS Algorithm

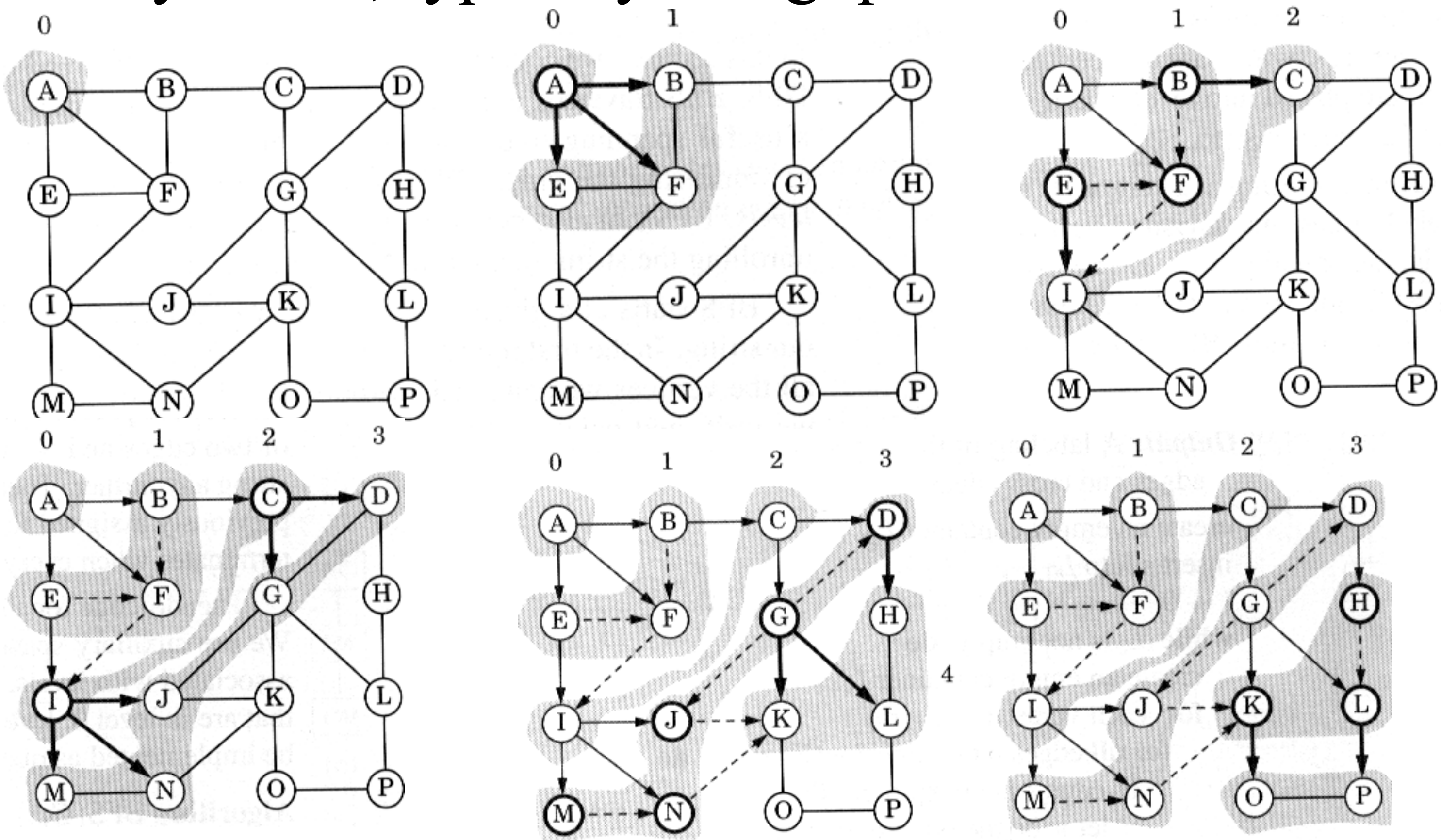
- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

```
Algorithm BFS(G)
    for all  u ? G.vertices()
        color[u] =  WHITE
        d[u] = inf.
        P[u] = nil.
    for all  v ? G.vertices()
    if v is not BLACK
        BFS(G, v)
```

```
Algorithm BFS(G, s)
    color[s]=GRAY,d[s]=0,P[s]=nil, Q={}
    Enqueue((Q, s)
    While Q.isNotEmpty()
        do u ? Dequeue(Q)
            for each v = Adj[u]
                do if color[v] = WHITE
                    then color[v] = GRAY
                        d[v] = d[u] + 1
                        P[v] = u
                        Enqueue(Q,v)
            color[u] = BLACK
```

Breadth-First Search

- By levels, typically using queues



BFS Facts

- There are discovery (tree) and cross edges

(why no back edges?) –

Once marked, don't follow again.

- Tree edges form spanning tree
- Tree edges are paths, minimal in length
- Cross edges differ by at most one in level
- Try writing the code to do a BFS

Algorithms based on BFS

- Test for connectivity
- compute spanning forest
- compute connected components
- find shortest path between two points (in number of links)
- compute a cycle in graph, or report none
- (have cross edges)
- Good for shortest path information, while DFS better for complex connectivity questions

ตัวแปรเก็บสถานะการเยี่ยมโหนด

- ตัวแปรสถานะการเยี่ยมโหนดมักถูกใช้เพื่อป้องกันไม่ให้เราทำเรื่องเดิมซ้ำไม่รู้จบ (ที่จริงเราเอาไว้บันทึกว่าโปรแกรมคำนวณโหนดใดไปแล้วนั่นเอง)
- เนื่องจากเราบันทึกทุกโหนด และโหนดมีเป็นจำนวนมาก
  → ตัวแปรนี้จึงถูกจัดทำในรูปของอาเรย์

```
boolean[] arVisit;

private void prepareSpace() {
  ......
  arVisit = new boolean[9];
}

private void insertData() {
   ......
   Arrays.fill(arVisit, false);
}
```

ตัวแปรเก็บสถานการณ์เยี่ยมโหนด สามารถใช้บูลีนเป็นตัวเก็บข้อมูลได้

ใช้ตัวอย่างเดิมมี 9 โหนด

กำหนดให้ค่าสถานะเริ่มต้นเป็น false คือยังไม่ถูกเยี่ยม (คำสั่งนี้สามารถเติมค่าในอาเรย์ได้เร็วมาก มาจาก java.util)

BFS กับการตรวจหาการไปถึงกันได้ของโหนด

```
int[] arIndex; // Keep pending nodes

public void listReachableNodes(final int src) {
  // Reset visit status
  Arrays.fill(arVisit, false);

  // Init the list of pending nodes
  int pending = src;
  int index0 = 0;    // Start index
  int index1 = 0;    // End index
  arIndex = new int[9];
  arIndex[index1] = pending;
  arVisit[pending] = true;
  ++index1;
  ......
}
```

เอาไว้ใช้บันทึกว่าโหนดไหนบ้างที่ถูกนำมาพิจารณาในการค้นหาครั้งนี้
โหนดไหนที่ถูกนำมาพิจารณาก็คือโหนดที่เข้าถึงได้จากโหนดที่กำหนด

เก็บอินเด็กซ์โหนดที่กำลังจะพิจารณา

เอาไว้เก็บขอบเขตอินเด็กซ์ของโหนดที่
ยังไม่ได้พิจารณา (ค่าเปลี่ยนไปเรื่อย ๆ)

Ref: Aj. Pinyo Taeprasartsit, Silapakorn

BFS กับการตรวจหาการไปถึงกันได้ของโหนด (2)

```
public void listReachableNodes(final int src) {
  ......

  while(index0 < index1) {
    // Explore neighboring nodes
    int nNeighbors = arNode[pending].length;
    for(int i = 0; i < nNeighbors; ++i) {
       int id = arNode[pending][i];
       if(arVisit[id] == false) {
         arIndex[index1] = id;
         ++index1;
         arVisit[id] = true;
       }
    }
    ++index0;
    pending = arIndex[index0];
  }
  ......
}
```

Ref: Aj. Pinyo Taeprasartsit, Silapakorn

BFS กับการตรวจหาการไปถึงกันได้ของโหนด (3)

ส่วนนี้แสดงให้เห็นว่าอาเรย์ที่สร้างขึ้นมาเอาไปใช้ระบุโหนดที่เข้าถึงได้ได้อย่างไร

```
public void listReachableNodes(final int src) {
  ……
  ……

  // List reachable nodes
  System.out.println("Source node = " + src);
  for(int i = 0; i < index1; ++i) {
    System.out.println(arIndex[i]);
  }
}
```

คำถามชวนคิด

ถ้าอยากรู้ว่ากราฟแบบไม่มีทิศทางถูกแบ่งออกเป็นกี่ส่วน ควรจะทำอย่างไร ?
(ส่วนเดียวกันคือโหนดที่เชื่อมต่อถึงกันไปมาได้หมด)

Ref: Aj. Pinyo Taeprasartsit, Silapakorn

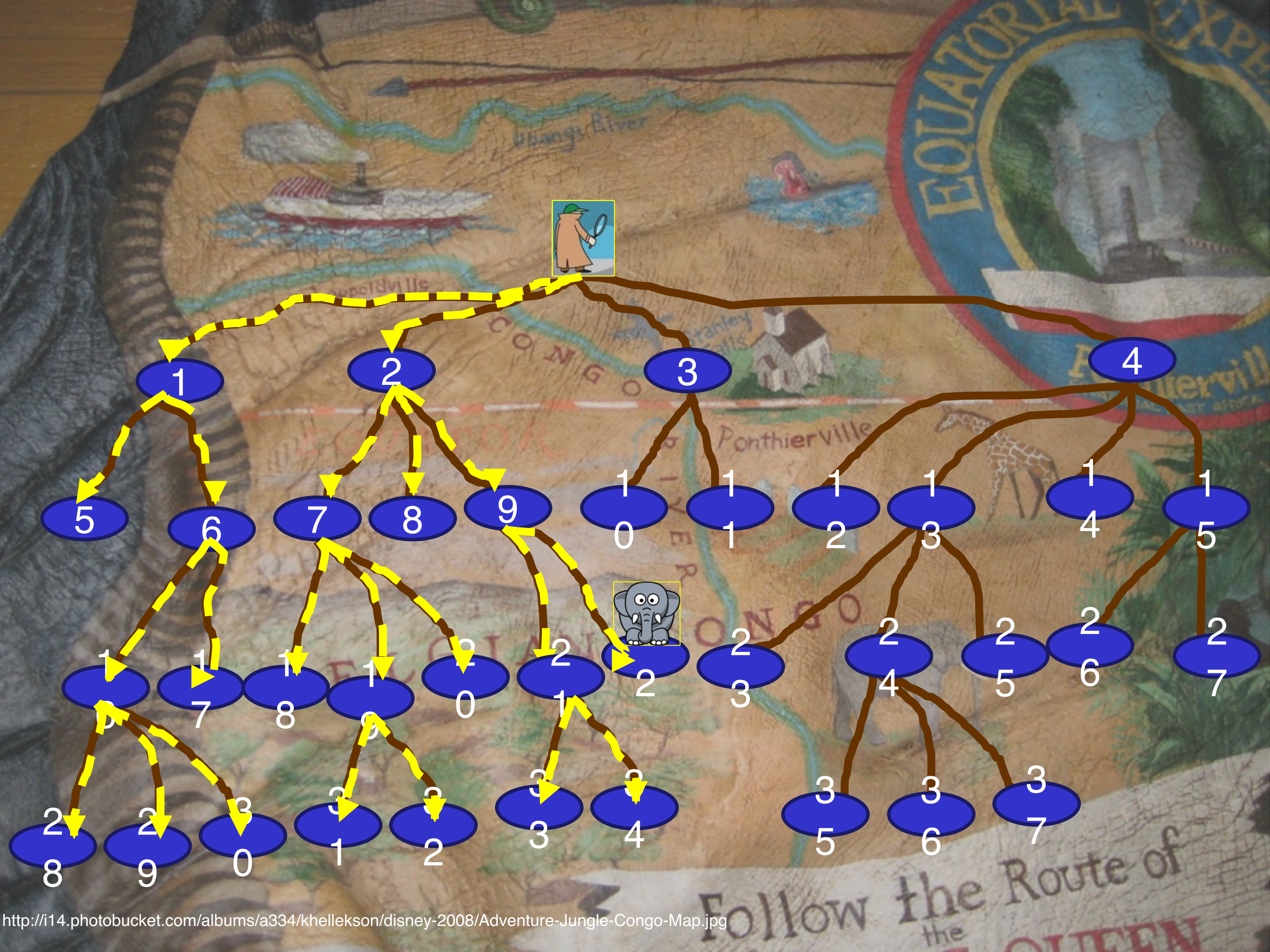
Complexity

- For each node in a digraph, how would you see which other nodes are reachable from that node?
- What would the complexity be?

Depth-First Search

- For each incident edge to a vertex
  - If opposite (other) vertex is unvisited
    - Label edge as "discovery"
    - Recur on the opposite vertex
  - Else label edge as "back"

Note: with m edges, O(m) using an adjacency list, but not using an adjacency matrix



http://i14.photobucket.com/albums/a334/khellekson/disney-2008/Adventure-Jungle-Congo-Map.jpg

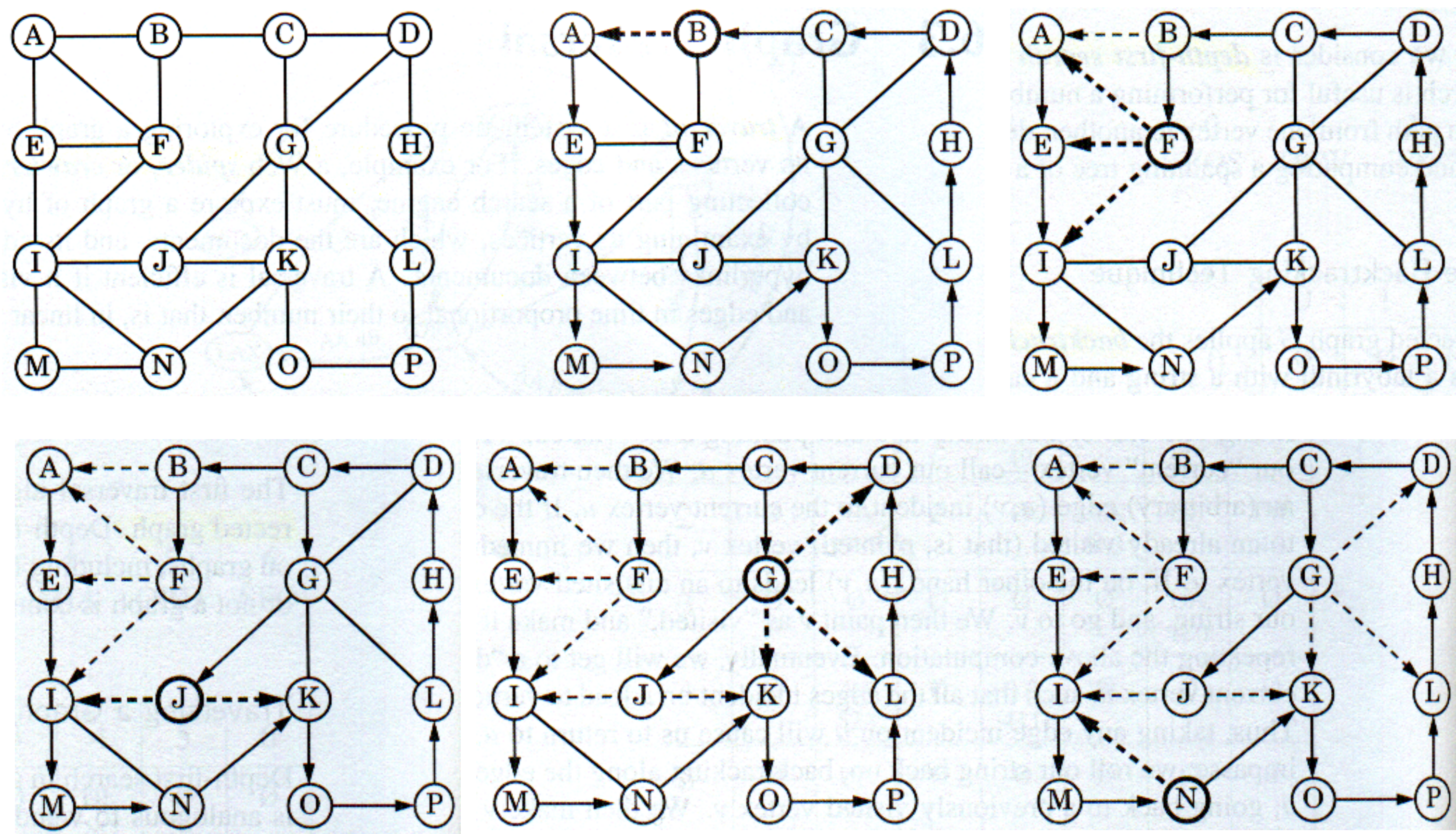
DFS Algorithm

- The algorithm uses a mechanism for setting and getting “labels” of vertices and edges

```
Algorithm DFS(G)
  for all  u ? G.vertices()
    color[u] = WHITE
    P[u] = nil
  time = 0
  for all  v ? G.vertices()
    if  color[v] = WHITE
          DFS(G, v)
```

```
Algorithm DFS(G, v)
  color[v] = Gray
  time = time + 1
  d[v] = time
   for all  u ? Adj (v)
     do if  color[u] = WHITE
        then P[u] = v
             DFS(G,u)
  color[v] = BLACK
  time = time + 1
  f[v] = time
```

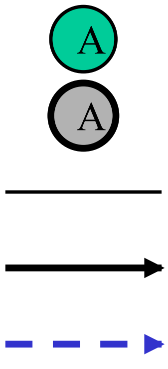
Depth-First Traversal



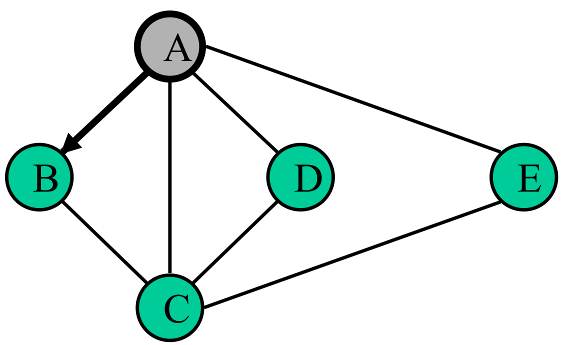
What is meant by depth first search?

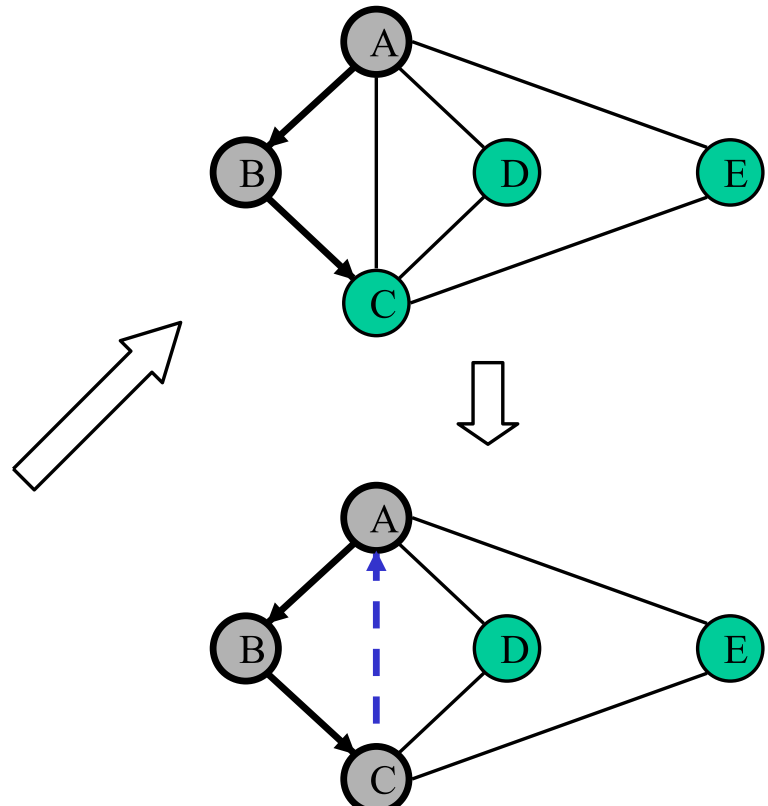
- Go deeper rather than broader
- Requires recursion or stack
- Used as a general means of traversal

Example

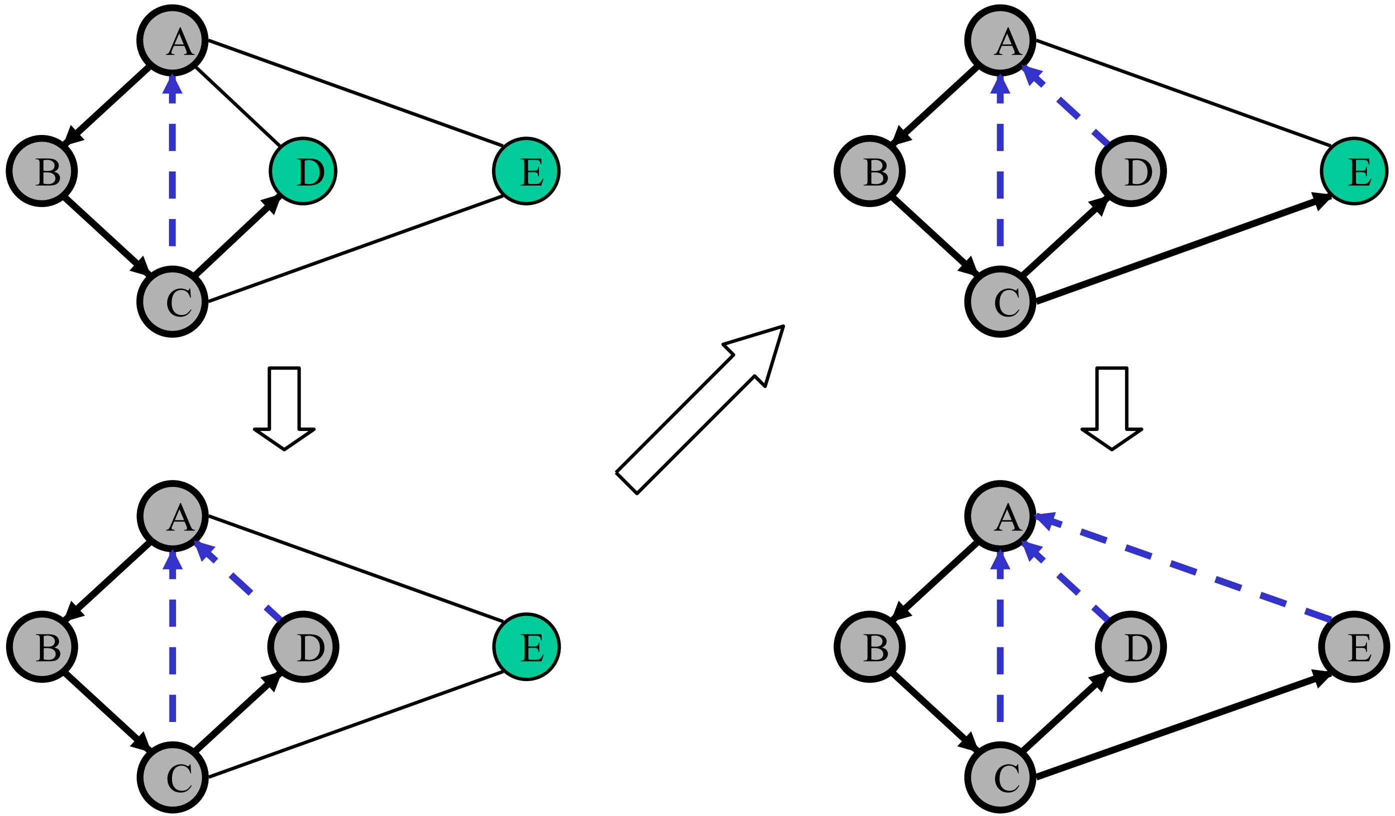


unexplored vertex
visited vertex
unexplored edge
discovery edge
back edge





Example (cont.)



DFS กับการตรวจหาการไปถึงกันได้ของโหนด

- ต่อให้เป็น DFS ก็ต้องใช้ตัวแปรเก็บสถานะเพื่อป้องกันการทำงานซ้ำ

  → ตัวแปรเก็บสถานะการเยี่ยมโหนดเป็นของคู่ชีพการจัดการกราฟจริง ๆ

```
int[] arIndex; // Keep pending nodes
int index;      // Keep current cursor of arIndex

public void listReachableNodes(final int src) {
  // Reset visit status
  Arrays.fill(arVisit, false);

  // Init the list of pending nodes
  arIndex = new int[9];
  index = 0;
  arIndex[index] = src;
  arVisit[src] = true;
  ++index;
  dfs(src);
  ......
}
```

Ref: Aj. Pinyo Taeprasartsit, Silapakorn

ส่วนหลักของ DFS

```
private void dfs(final int src) {
  int nNeighbors = arNode[src].length;
  for(int i = 0; i < nNeighbors; ++i) {
    int id = arNode[src][i];
    if(arVisit[id] == false) {
      arIndex[index] = id;
      arVisit[id] = true;
      ++index;
      dfs(id);
    }
  }
}
```

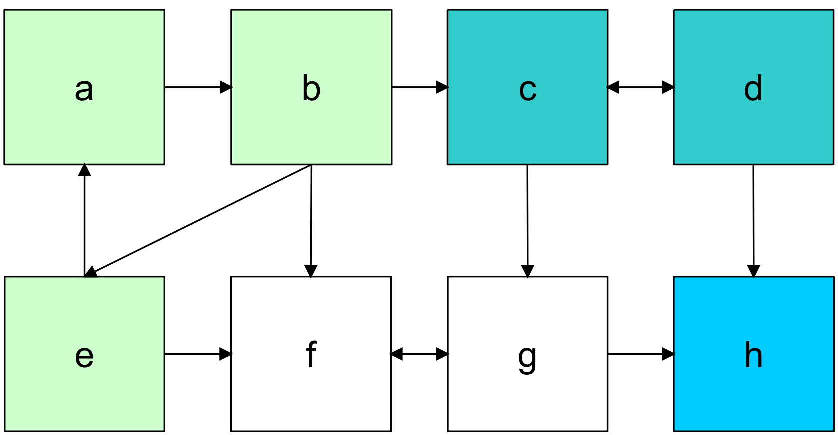
พอพบโหนดใหม่ เราก็รีบเจาะลึกค้นหามันต่อไปทันที ไม่รอดูเพื่อนบ้านคนอื่นของโหนด src ปัจจุบัน

- ถ้าไม่ทำเป็นแบบรีเคอซีฟเรื่องจะยุ่งขึ้นเล็กน้อย เพราะโหนดที่เราจดจ่ออยู่จะเปลี่ยนไปเร็วมากทำให้เราต้อง (1) อ่านลิสต์เพื่อนบ้านใหม่เมื่อย้อนกลับมาซ้ำ ๆ หรือ (2) ต้องคอยจำค่าว่าอ่านลิสต์เพื่อนบ้านแต่ละคนไปถึงไหนแล้ว และมีรายละเอียดจุกจิกอื่น ๆ ด้วย

Ref: Aj. Pinyo Taeprasartsit, Silapakorn

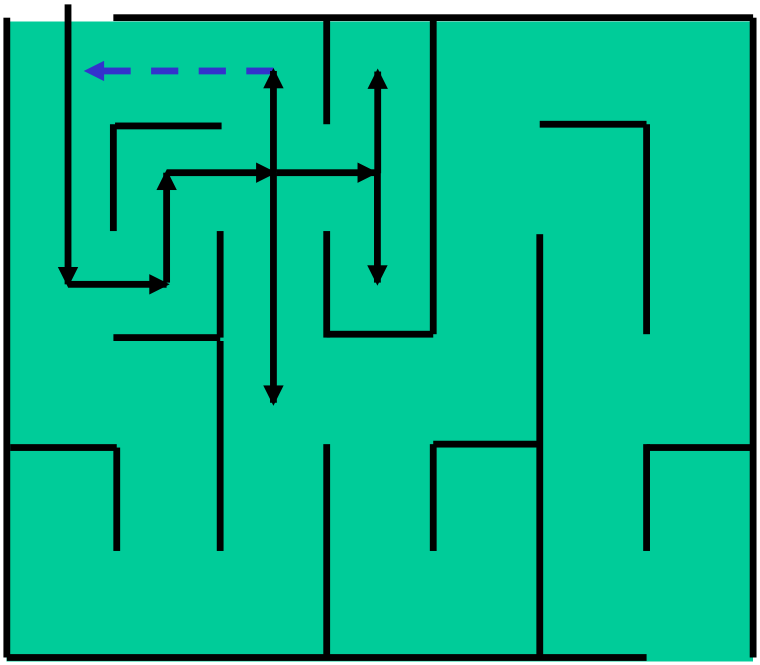
Perform BFS and DFS on this graph

- Starting from node: “a”
- If there are more than one possible choice at the same level, explore the node with lower alphabetical order first (“c” before “e”)
- What are the corresponding BFS-/DFS- Trees?



DFS and Maze Traversal

- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



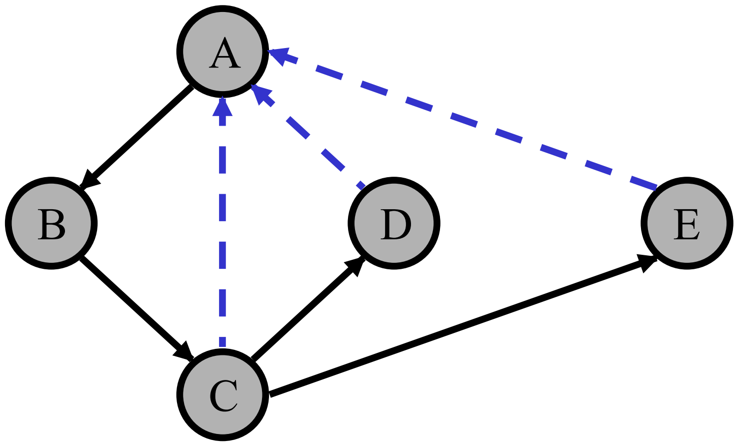
Properties of DFS

Property 1

DFS(G, v) visits all the vertices and edges in the connected component of v

Property 2

The discovery edges labeled by DFS(G, v) form a spanning tree of the connected component of v



What is the complexity?

- Assign DFS Numbers
- Assign Low
- Examine nodes for articulation point

If we don't know which is greater n (the number of nodes) or m (the number of edges), we show it as $O(n+m)$

Complexity of DFS

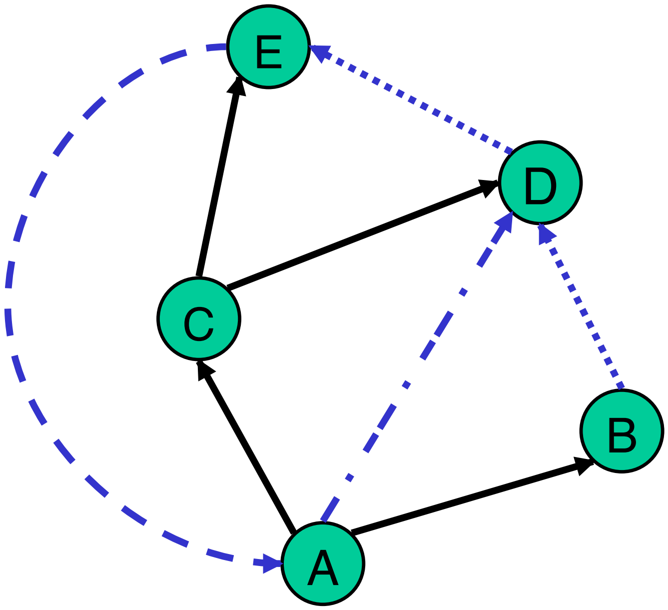
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure

Digraph Facts

- Directed DFS gives directed paths from root to each reachable vertex
- Used for O(n(n+m)) algorithm [dfs is O(n+m), these algorithms use n dfs searches]
  - Find all induced subgraphs (from each vertex, v, find subgraph reachable from v)
  - Test for strong connectivity
  - Compute the transitive closure

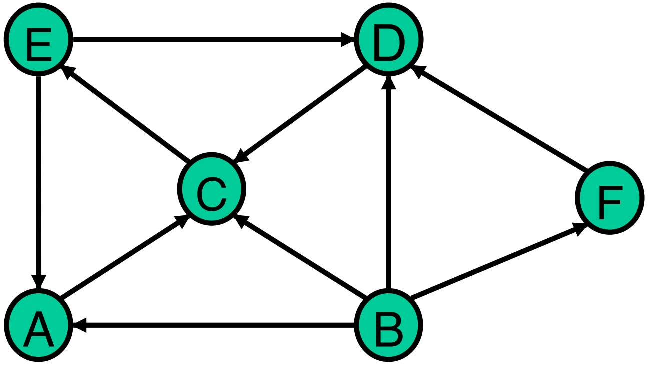
Directed DFS

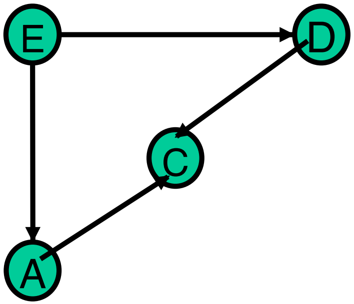
- We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction
- In the directed DFS algorithm, we have four types of edges
  - discovery edges
  - back edges (to ancestor)
  - forward edges (to descendant)
  - cross edges (to other)
- A directed DFS starting at avertex s determines the vertices reachable from s

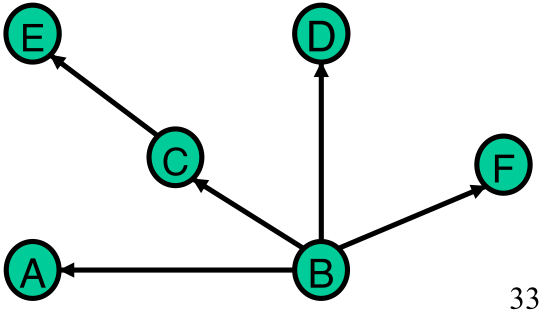


Reachability

- DFS tree rooted at v: vertices reachable from v via directed paths







ระเบิดกำแพงเขาวงกต (TOI 2012)

มีระเบิดหนึ่งลูก ต้องใช้ระเบิดกำแพงในจุดที่ทำให้เกิดทางที่สั้นที่สุดระหว่างจุดเริ่มต้นไปถึงทางออก

นักผจญภัยเดินได้เฉพาะบนช่องเลข 1 ในแนวตั้งฉาก จุดเริ่มต้นคือวงรีแดง ทางออกคือสามเหลี่ยมเลข 1 แต่มีกำแพงเลข 0 ขวางไว้ ให้หาว่าจะวางระเบิดซึ่งระเบิดช่องเลข 0 ได้แค่ช่องเดียวอย่างไรจึงจะได้ทางที่สั้นที่สุด

0	0	1	1	0	0	0	0
1	0	1	1	0	1	1	1
1	0	1	1	1	0	0	1
1	1	0	0	1	0	0	1
0	0	1	1	0	1	1	1

Ref: Aj. Pinyo Taeprasartsit, Silapakorn

โจทย์ทดสอบระบบ ACM ภาคกลางปี 2012

ข้อ Island Survey: นับว่ามีเลข 1 ที่เชื่อมต่อกันเป็นพื้นที่แยกกันกี่พื้นที่

0	0	0	0	0	0	0	0	0
0	0	0	**1**	**1**	**1**	0	**1**	0
0	**1**	**1**	0	**1**	**1**	0	**1**	0
0	0	**1**	**1**	0	0	0	**1**	0
0	0	0	0	0	0	0	0	0

กำหนดให้การเชื่อมต่อกันของพื้นที่เป็นไปได้ 8 ทิศทาง ดังนั้นพื้นที่หมายเลข 1 ที่เชื่อมต่อกัน จะแยกเป็นพื้นที่ได้ทั้งหมด 2 พื้นที่

- ข้อกำหนดเกี่ยวกับการเชื่อมต่อทำให้เราไม่จำเป็นต้องให้ข้อมูลเส้นเชื่อมแยกต่างหาก แต่คิดได้จากตำแหน่งที่กำลังพิจารณาแต่ละตำแหน่งเลย
- กราฟถูกมองได้เป็นอาเรย์สองมิติ และการสำรวจการเชื่อมต่อสามารถทำได้โดยการใช้ตัวเก็บสถานะการเยี่ยมโหนดที่เป็นอาเรย์สองมิติ

Ref: Aj. Pinyo Taeprasartsit, Silapakorn

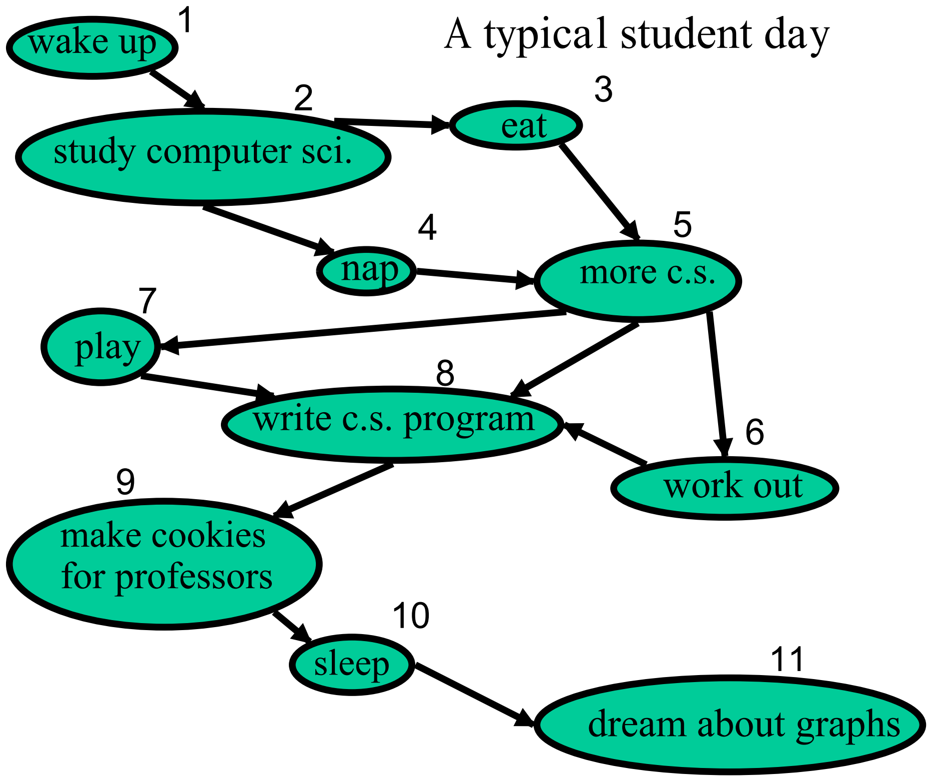
TOPOLOGICAL SORT

Topological sort

- We have a set of tasks and a set of dependencies (precedence constraints) of form "task A must be done before task B"
- Topological sort: An ordering of the tasks that conforms with the given dependencies
- Goal: Find a topological sort of the tasks or decide that there is no such ordering

Topological Sorting

- Number vertices, so that (u,v) in E implies u < v



Topological sort more formally

- Suppose that in a directed graph $G = (V, E)$ vertices

  V represent tasks, and each edge $(u, v) \in E$ means

  that task u must be done before task v

- What is an ordering of vertices 1, ..., $|V|$ such that for every edge (u, v), u appears before v in the ordering?

- Such an ordering is called a topological sort of G
- Note: there can be multiple topological sorts of G

Topological sort more formally

- Is it possible to execute all the tasks in G in an order that respects all the precedence requirements given by the graph edges?
- The answer is "yes" if and only if the directed graph G has no cycle!

  (otherwise we have a deadlock)
- Such a G is called a Directed Acyclic Graph, or just a DAG

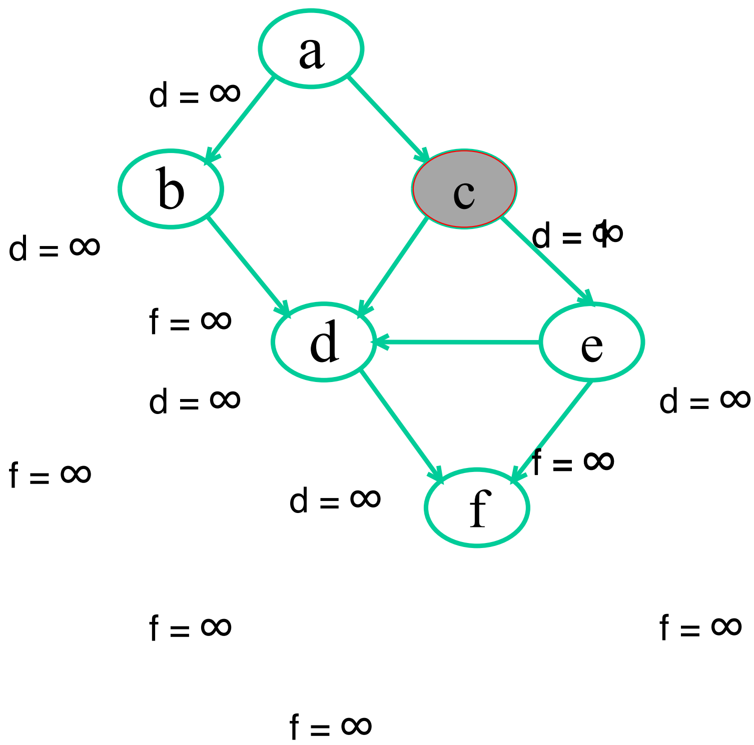
Algorithm for TS

- TOPOLOGICAL-SORT(G):
  1) call DFS(G) to compute finishing times f[v] for each vertex v
  2) as each vertex is finished, insert it onto the front of a linked list
  3) return the linked list of vertices
- Note that the result is just a list of vertices in order of decreasing finish times f[]

Topological sort

1) Call DFS(G) to compute the finishing times f[v]

Time =



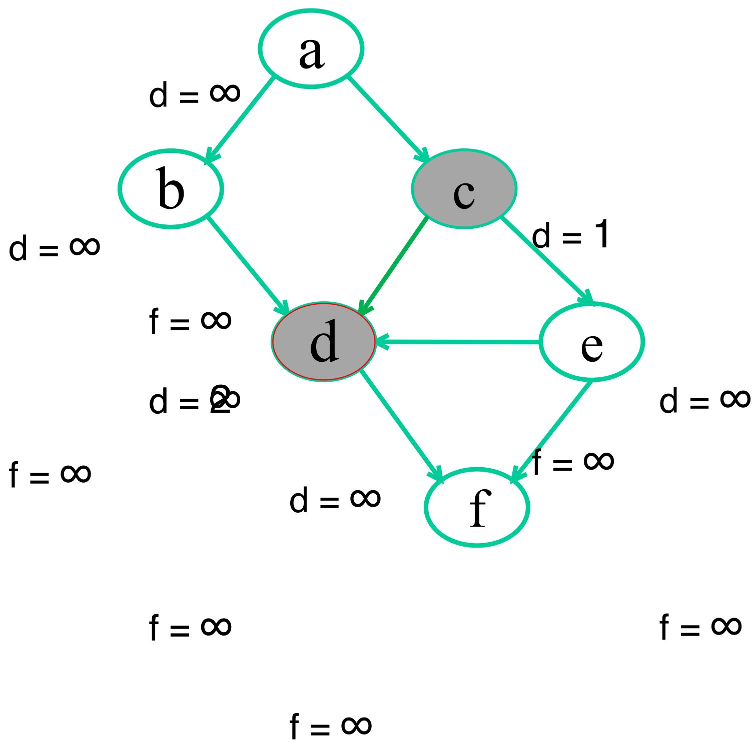
Let's say we start the DFS from the vertex c

Next we discover the vertex d

Topological sort

1) Call DFS(G) to compute the finishing times f[v]

Time =



Let's say we start the DFS from the vertex c

Next we discover the vertex d

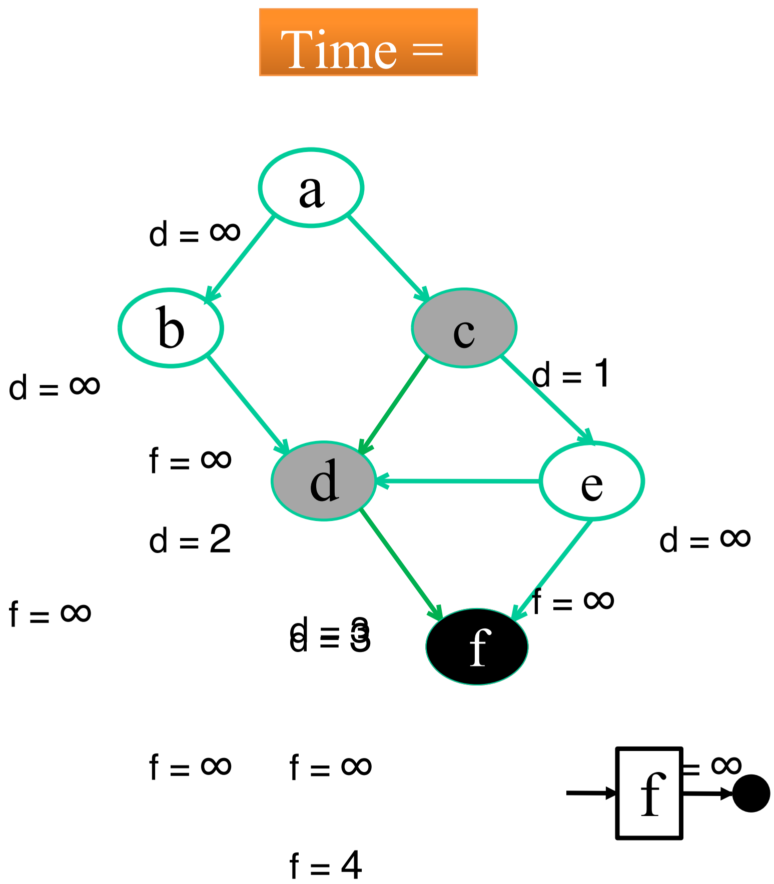
Topological sort

1) Call DFS(G) to compute the finishing times f[v]

2) as each vertex is finished, insert it onto the front of a linked list

Next we discover the vertex f

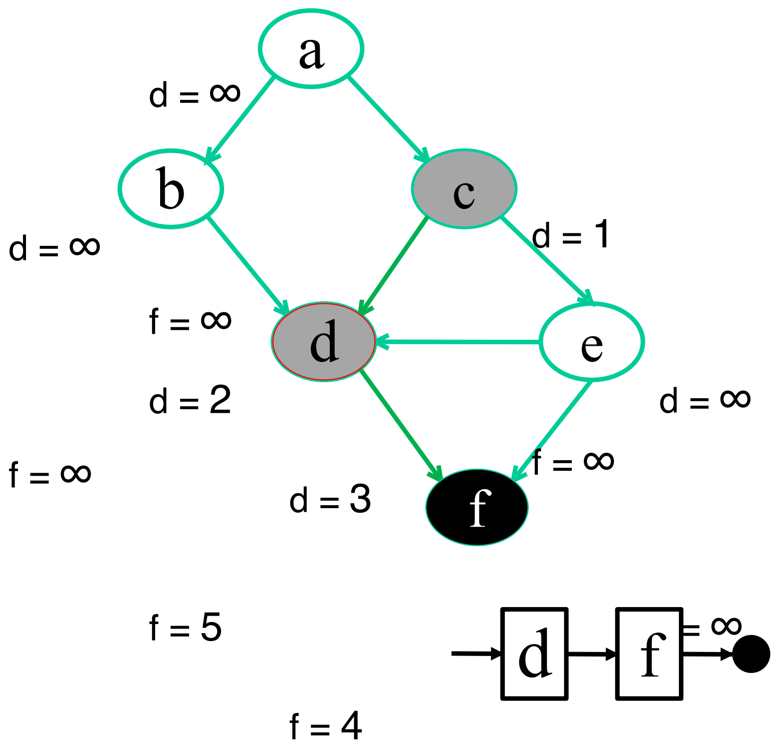
f is done, move back to d



Topological sort

1) Call DFS(G) to compute the finishing times f[v]

Time =



Let's say we start the DFS from the vertex c

Next we discover the vertex d

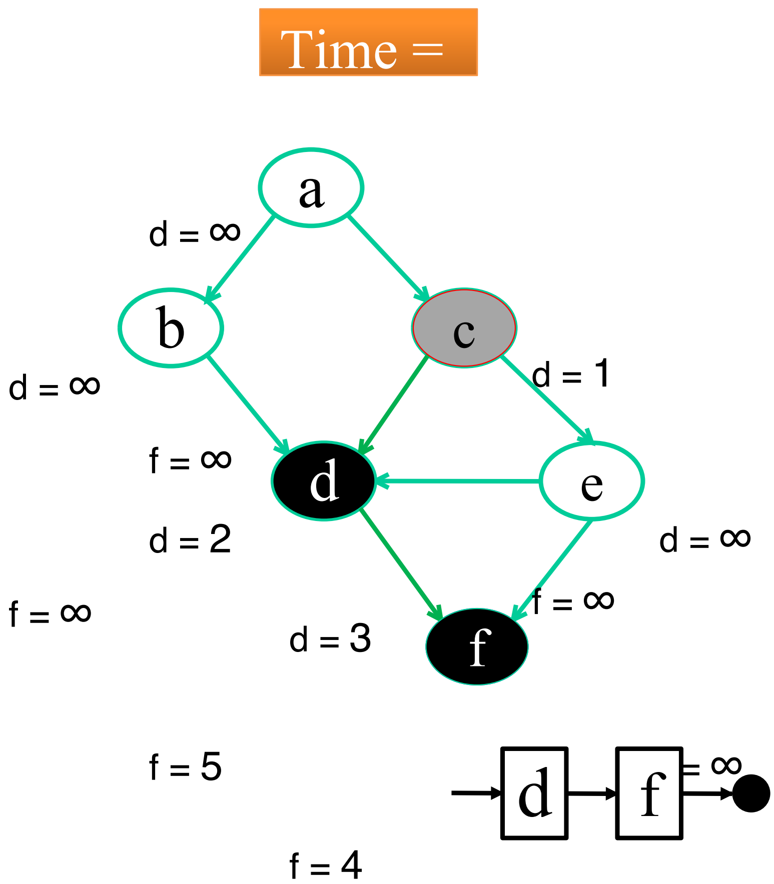
Next we discover the vertex f

f is done, move back to d

d is done, move back to c

Topological sort

1) Call DFS(G) to compute the finishing times f[v]



Let’s say we start the DFS from the vertex c

Next we discover the vertex d

Next we discover the vertex f

f is done, move back to d

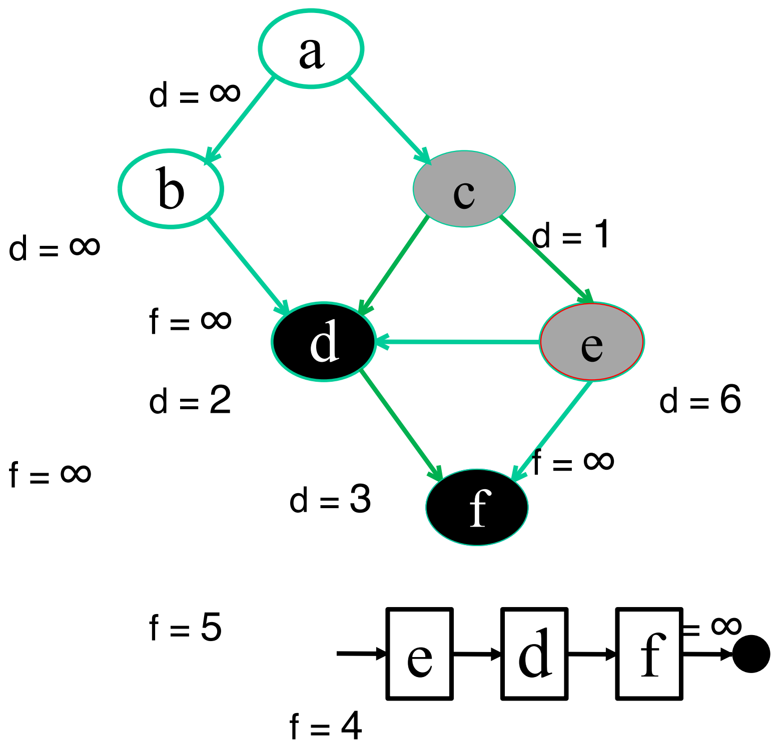
d is done, move back to c

Next we discover the vertex e

Topological sort

1) Call DFS(G) to compute the finishing times f[v]

Time =



Let’s say we start the DFS from the vertex c

Next we discover the vertex d

Both edges from e are cross edges

d is done, move back to c

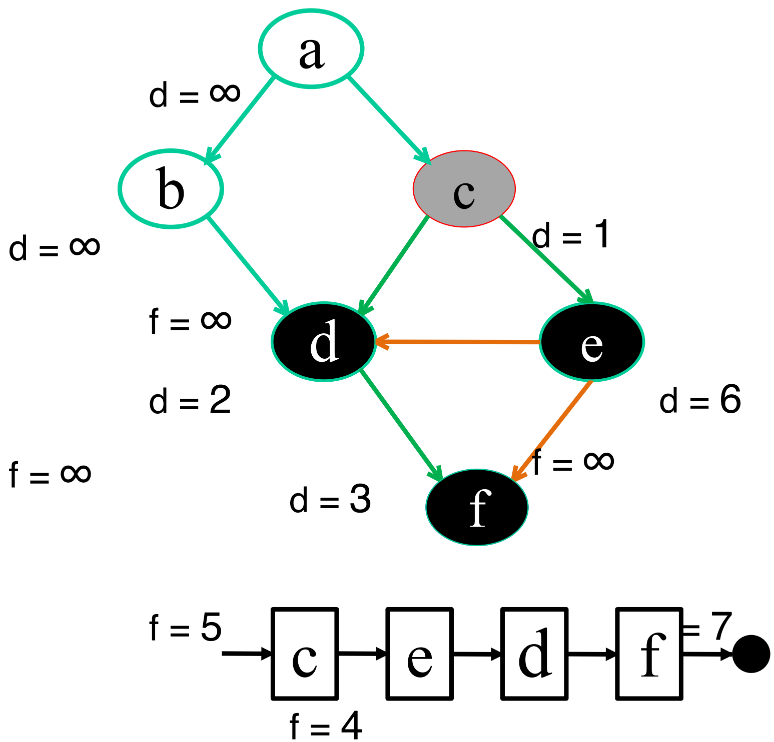
Next we discover the vertex e

e is done, move back to c

Topological sort

1) Call DFS(G) to compute the finishing times f[v]

Time =



Let's say we start the DFS from the vertex c

Just a note: If there was (c,f) edge in the graph, it would be classified as a forward edge (in this particular DFS run)

d is done, move back to c

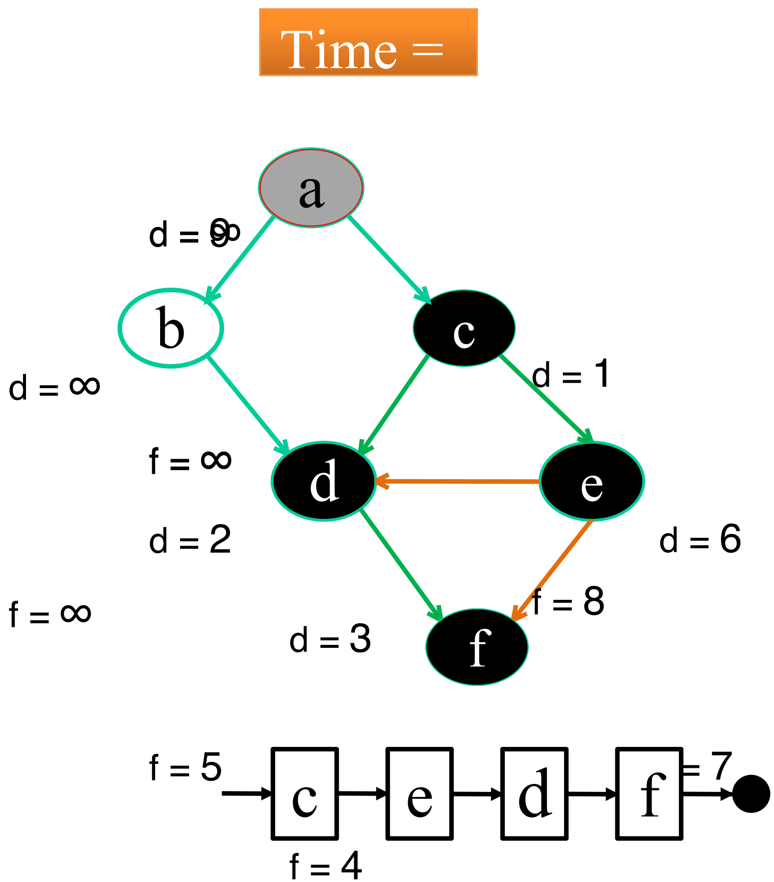
Next we discover the vertex e

e is done, move back to c

c is done as well

Topological sort

1) Call DFS(G) to compute the finishing times f[v]



Let's now call DFS visit from the vertex a

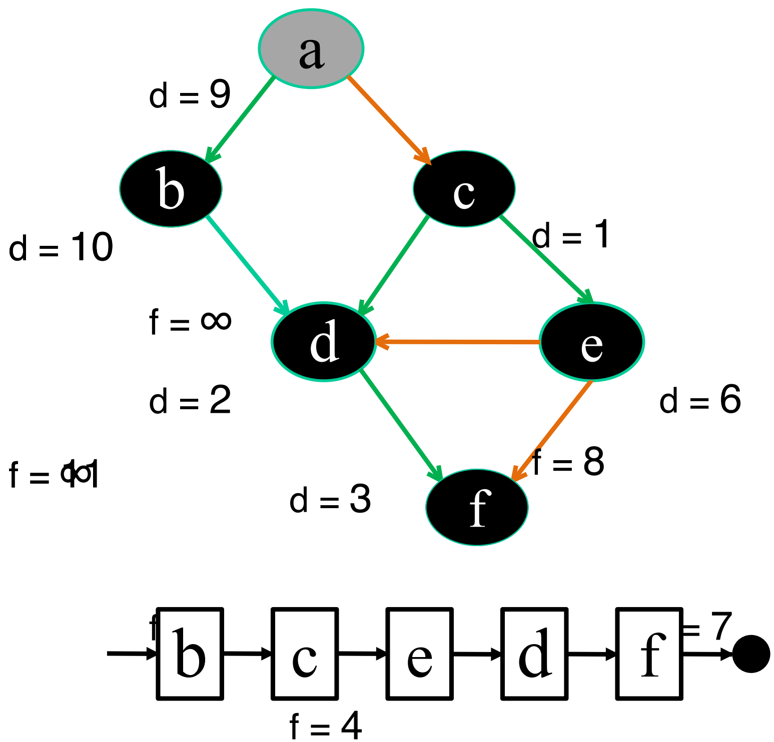
Next we discover the vertex c, but c was already processed => (a,c) is a cross edge

Next we discover the vertex b

Topological sort

1) Call DFS(G) to compute the finishing times f[v]

Time =



Let's now call DFS visit from the vertex a

Next we discover the vertex c, but c was already processed => (a,c) is a cross edge

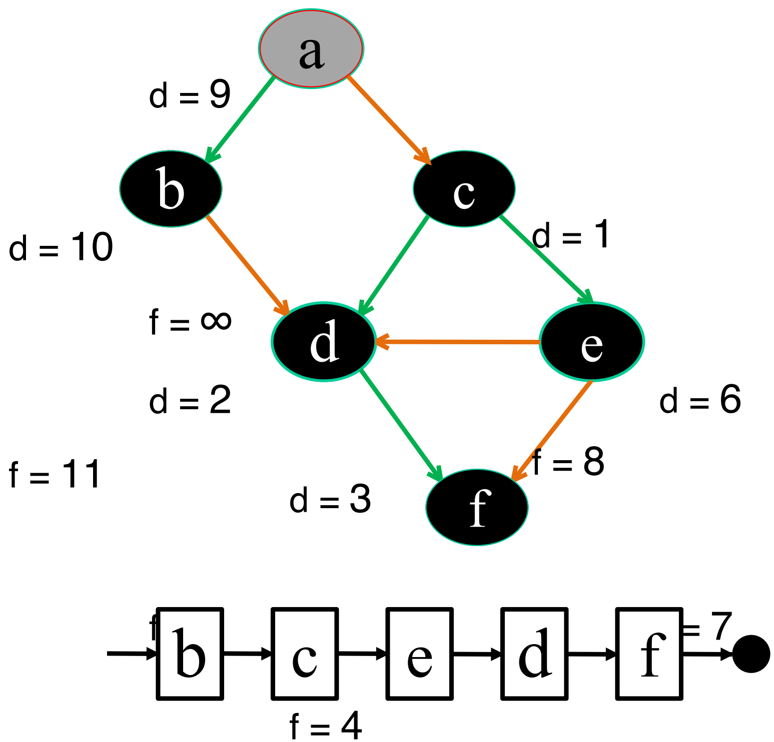
Next we discover the vertex b

b is done as (b,d) is a cross edge => now move back to c

Topological sort

1) Call DFS(G) to compute the finishing times f[v]

Time =



Let's now call DFS visit from the vertex a

Next we discover the vertex c, but c was already processed => (a,c) is a cross edge

Next we discover the vertex b

b is done as (b,d) is a cross edge => now move back to c

a is done as well

Topological sort

1) Call DFS(G) to compute the finishing times f[v]

Time =

WE HAVE THE RESULT!

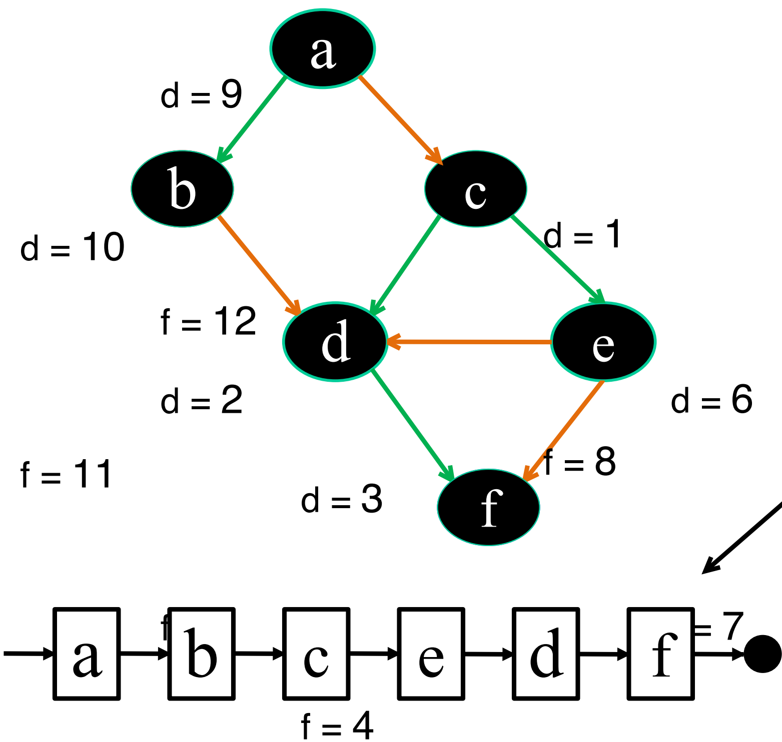
3) return the linked list of vertices

=> (a,c) is a cross edge

Next we discover the vertex b

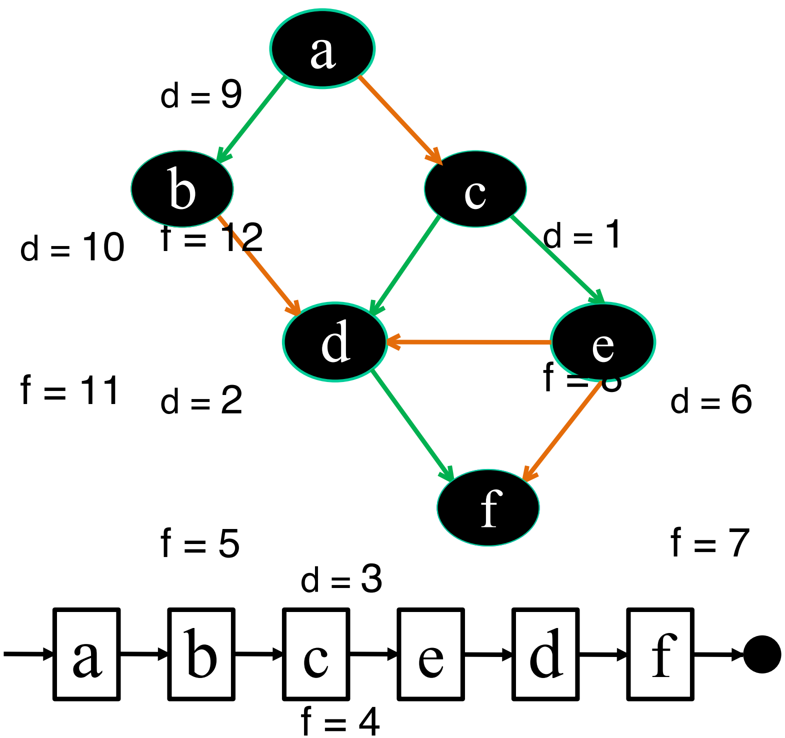
b is done as (b,d) is a cross edge => now move back to c

a is done as well



Topological sort

Time =

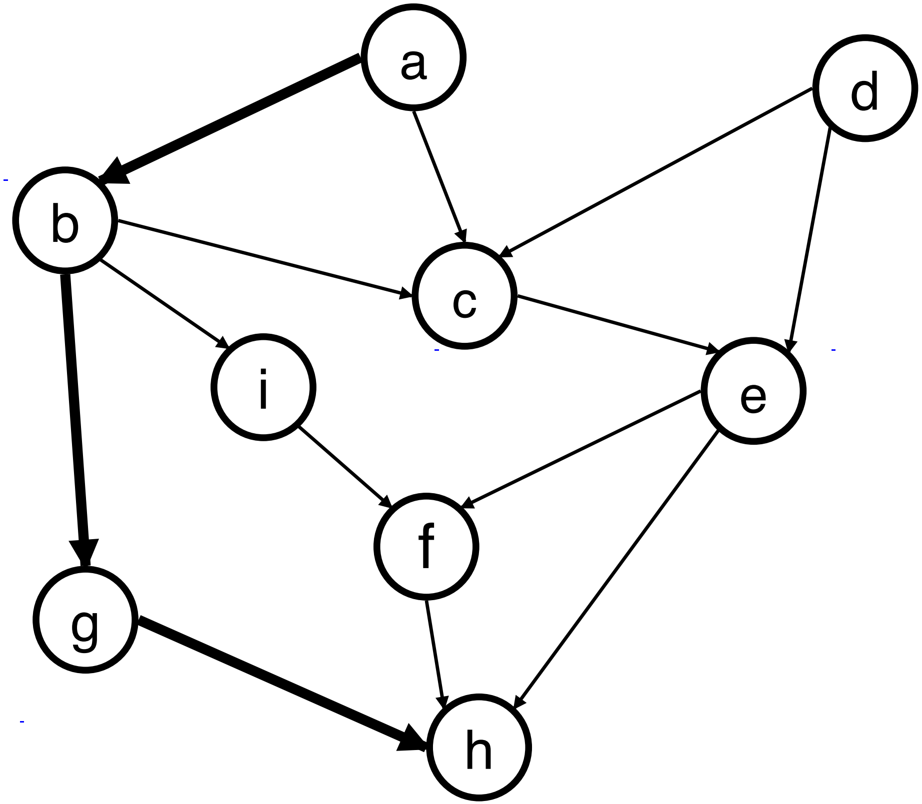


The linked list is sorted in decreasing order of finishing times f[]

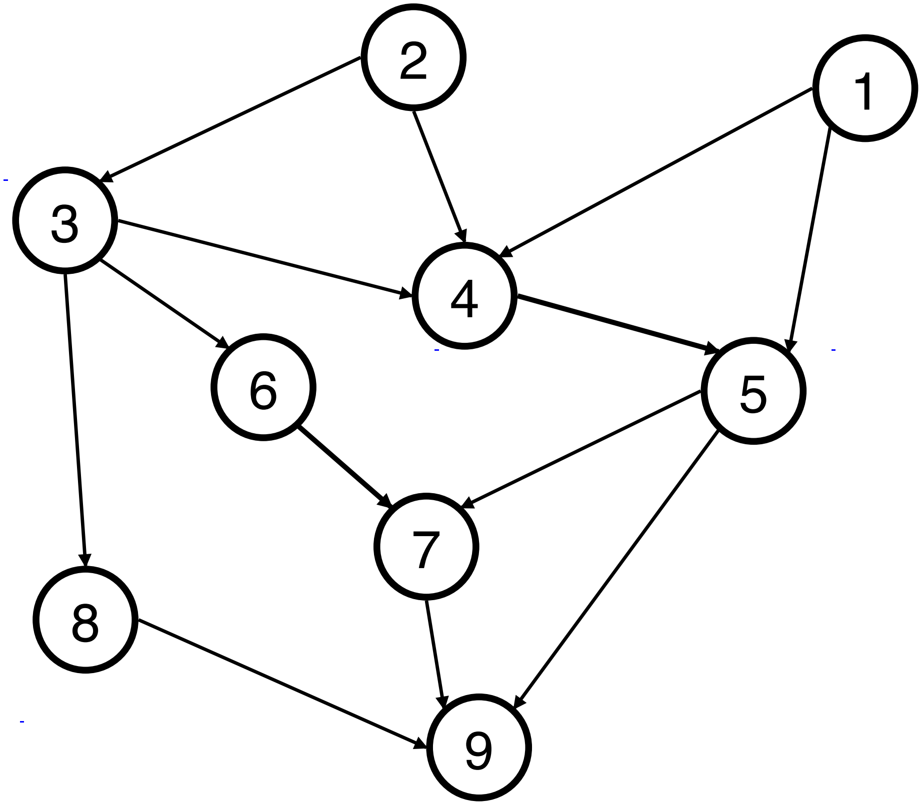
Try yourself with different vertex order for DFS visit

Note: If you redraw the graph so that all vertices are in a line ordered by a valid topological sort, then all edges point „from left to right“

Topological Sorting Exercise



Topological Sorting Exercise



Time complexity of TS(G)

- Running time of topological sort:

$$\Theta(n + m)$$
$$\text{where } n=|V| \text{ and } m=|E|$$

- Why? Depth first search takes $\Theta(n + m)$ time in the worst case, and inserting into the front of a linked list takes $\Theta(1)$ time

Toposort Implementation

ส่วนที่ดัดแปลงมาจาก dfs

```
void dfs2(int src) {
  arVisit[src] = true;

  final int nNeighbors = arNode[src].length;
  for(int i = 0; i < nNeighbors; ++i) {
    int id = arNode[src][i];
    if(arVisit[id] == false) {
      // No node insertion here, keep it for later.
      dfs2(id);
    }
  }

  //System.out.println("Insert " + src);
  arOrder[index] = src;
  ++index;
}
```

Ref: Aj. Pinyo Taeprasartsit, Silapakorn

Toposort Implementation (2)

ส่วนเตรียมตัวเรียกการใช้งาน

```
public void listTopoOrder() {
  // Reset visit status
  Arrays.fill(arVisit, false);

  // Init the list of pending nodes
  arOrder = new int[8]; // This example has Nodes 0 to 7.
  index = 0;
  for(int src = 0; src < 8; ++src) {
    if(arVisit[src] == false)
       dfs2(src);
    }

    // Write topological order
    System.out.println("Topological order:");
    for(int i = index - 1; i >= 0; --i) {
      System.out.println(arOrder[i]);
    }
}
```

เนื่องจากใช้อาเรย์มาเก็บลำดับ ไม่ได้ใช้แสต็ค ลำดับจึงย้อนหลัง

Ref: Aj. Pinyo Taeprasartsit, Silapakorn

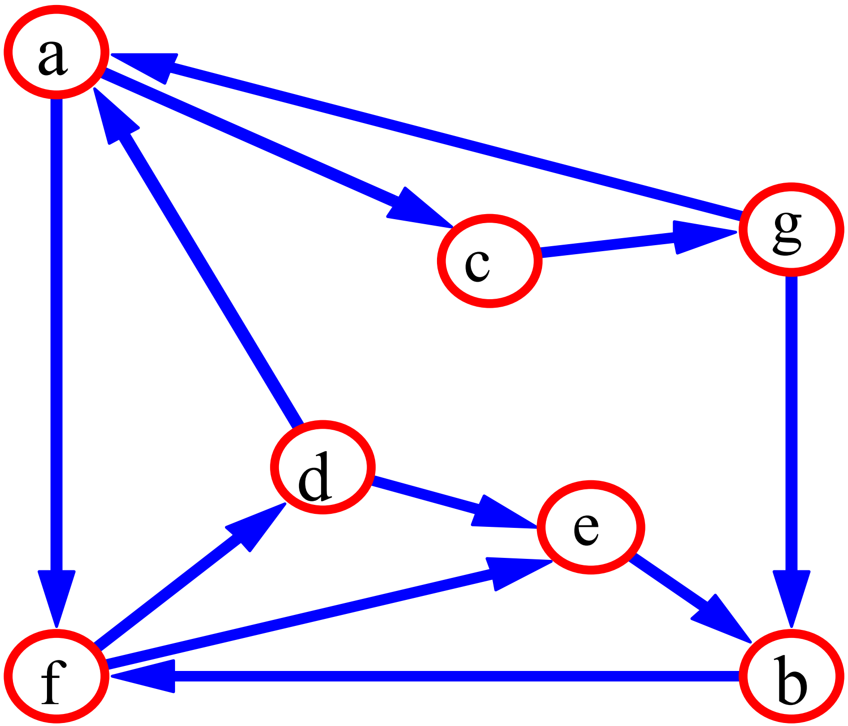
ประยุกต์ใช้กับ Job Scheduling

- สมมติว่าข้อจำกัดมีเพียงว่าต้องทำงานที่จำเป็นก่อนหน้าให้เสร็จก่อน
  - งานขั้นตอนงานทุกอย่างใช้เวลาเท่ากัน
  - งานที่ไม่ขึ้นต่อกันทำพร้อมกันกี่อันก็ได้ ขอแค่ขั้นตอนก่อนหน้าเสร็จไปแล้ว
  - อยากตอบให้ได้ว่างานจะเสร็จเร็วที่สุดได้ต้องทำกี่ขั้น และควรจัดงานอย่างไร

Ref: Aj. Pinyo Taeprasartsit, Silapakorn

Strongly Connected Directed graphs

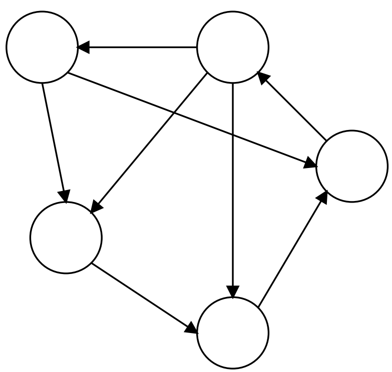
- Every pair of vertices are reachable from each other



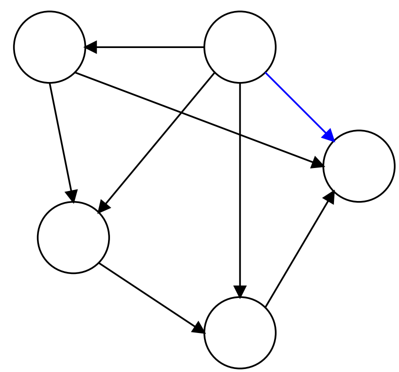
Strongly-Connected

Graph G is strongly connected if, for every u and v in V, there is some path from u to v and some path from v to u.

Strongly Connected

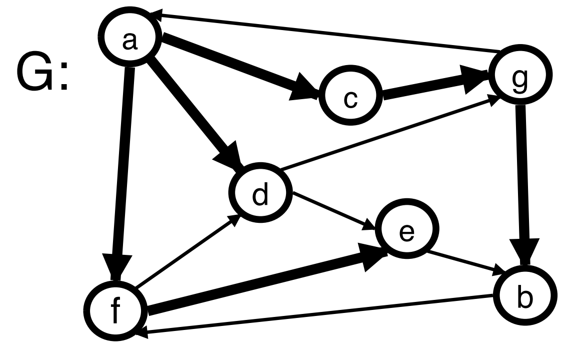


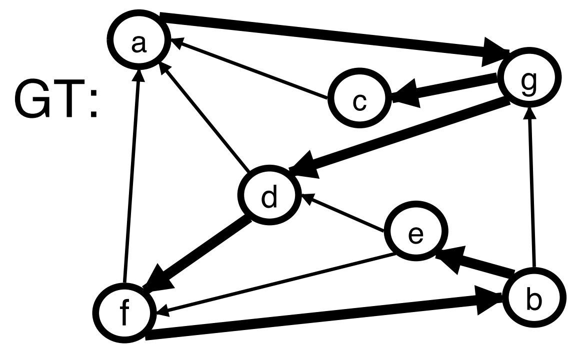
Not Strongly Connected



Strong Connectivity Algorithm

- Pick a vertex v in G.
- Perform a DFS from v in G.
  - If there's a w not visited, return not strongly connected
- Let GT be G with edges reversed.
- Perform a DFS from v in GT.
  - If there's a w not visited, return not strongly connected
  - Else, return strongly connected



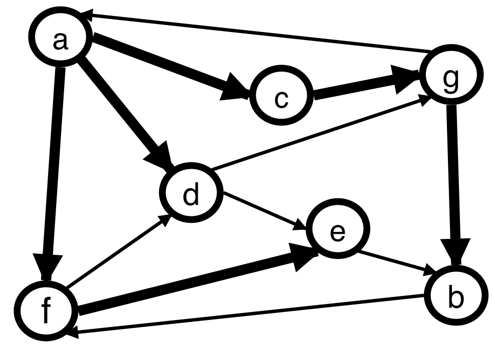


- Running time: O(n+m).

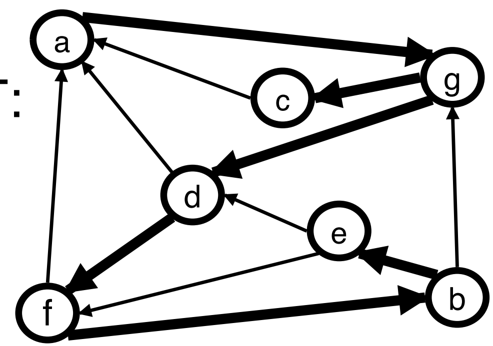
Strong Connectivity Algorithm

- Pick a vertex v in G.
- Perform a DFS from v in G.
  - If there's a w not visited, return not strongly connected
- Let GT be G with edges reversed.
- Perform a DFS from v in GT.
  - If there's a w not visited, return not strongly connected
  - Else, return strongly connected
- Running time: O(n+m).

G:



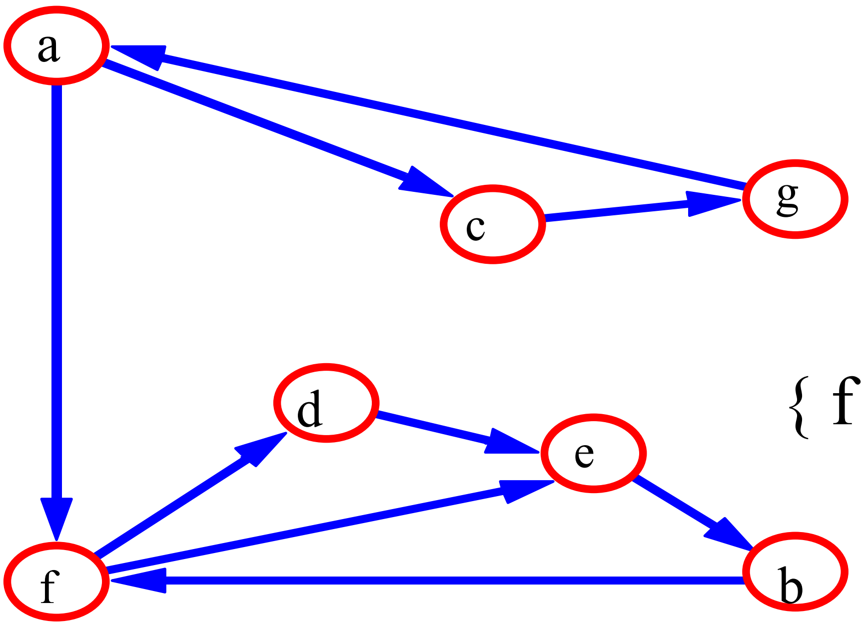
GT:



Strongly-Connected Components

A strongly connected component of a graph is a maximal subset of nodes (along with their associated edges) that is strongly connected. Nodes share a strongly connected component if they are inter-reachable.

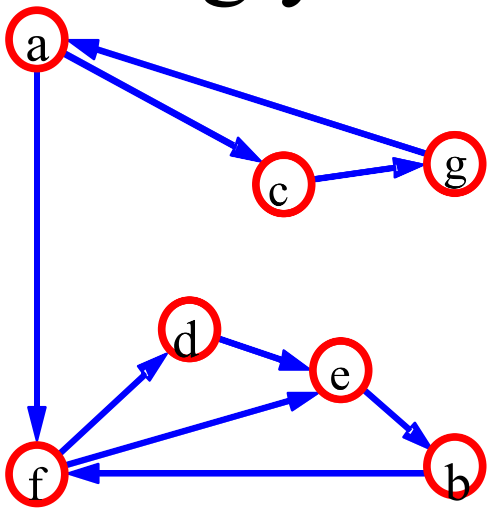
Strongly Connected Components

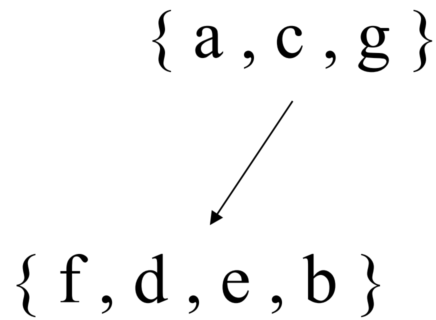


{ a , c , g }

{ f , d , e , b }

Reduced Component Graph of Strongly Connected Components





- Component graph GSCC=(VSCC, ESCC): one vertex for each component
  - (u, v) ∈ESCC if there exists at least one directed edge from the corresponding components

Graph of Strongly Connected Components

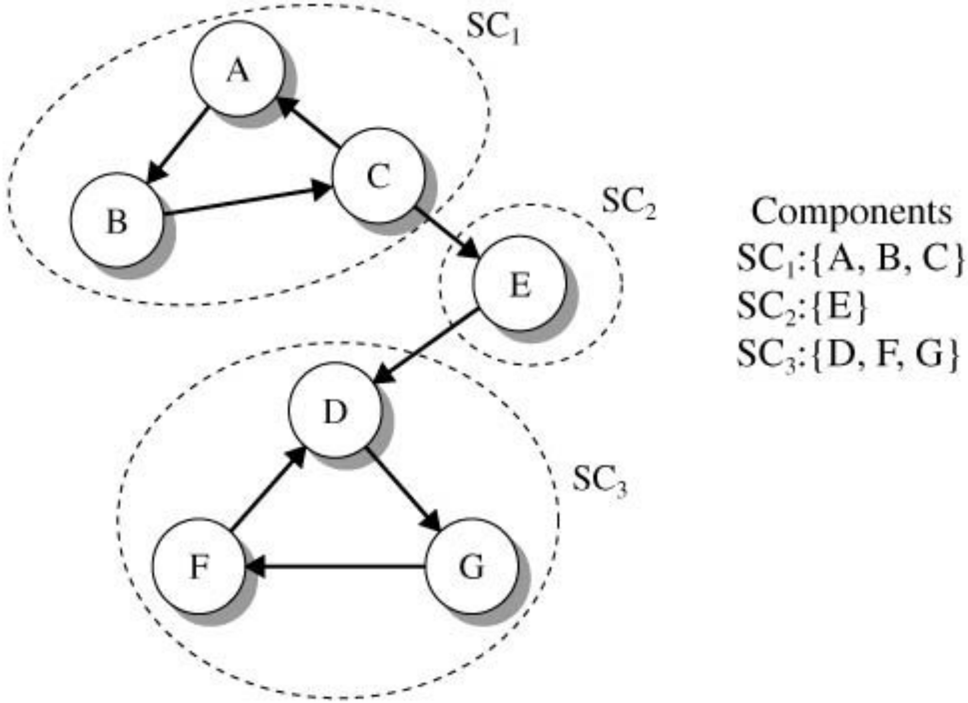
- Theorem: the Component graph GSCC=(VSCC , ESCC) is a directed-acyclic-graph (DAG)
  - Each component is maximal in the sense that no other vertices can be added to it. If GSCC=(VSCC, ESCC) is not a DAG, then one can merge components on along a circle of GSCC
- Therefore, GSCC has a topological ordering

Finding Strongly-Connected Components

- Input: A directed graph $G = (V,E)$
- Output: a partition of V into disjoint sets so that each set defines a strongly connected component of G
- How should we compute the partition?

Strongly Connected Components

Any graph can be partitioned into a unique set of strong components.

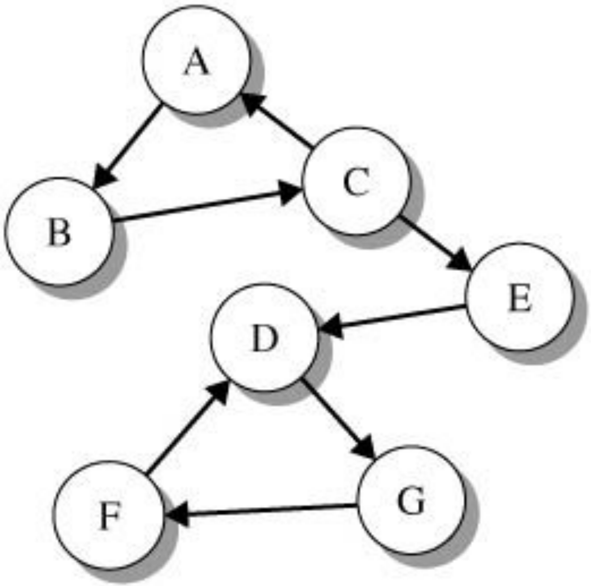


Strongly connected components in a directed graph.

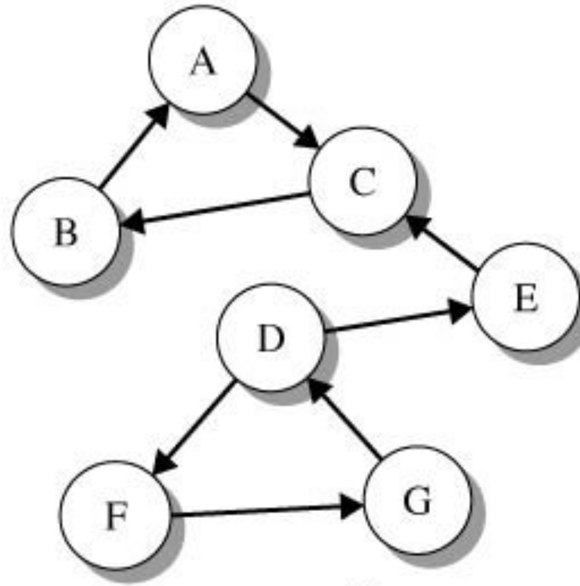
Strongly Connected Components (continued)

- Execute the depth-first search dfs() for the graph G which creates the list dfsList consisting of the vertices in G in the reverse order of their finishing times.
- Generate the transpose graph G^T.
- Using the order of vertices in dfsList, make repeated calls to dfs() for vertices in G^T. The list returned by each call is a strongly connected component of G.

Strongly Connected Components (continued)



Graph G



Graph G^T

Strongly Connected Components (continued)

dfsList: [A, B, C, E, D, G, F]

Using the order of vertices in dfsList, make successive calls to dfs() for graph GT

Vertex A: dfs(A) returns the list [A, C, B] of vertices reachable from A in GT.

Vertex E: The next unvisited vertex in dfsList is E. Calling dfs(E) returns the list [E].

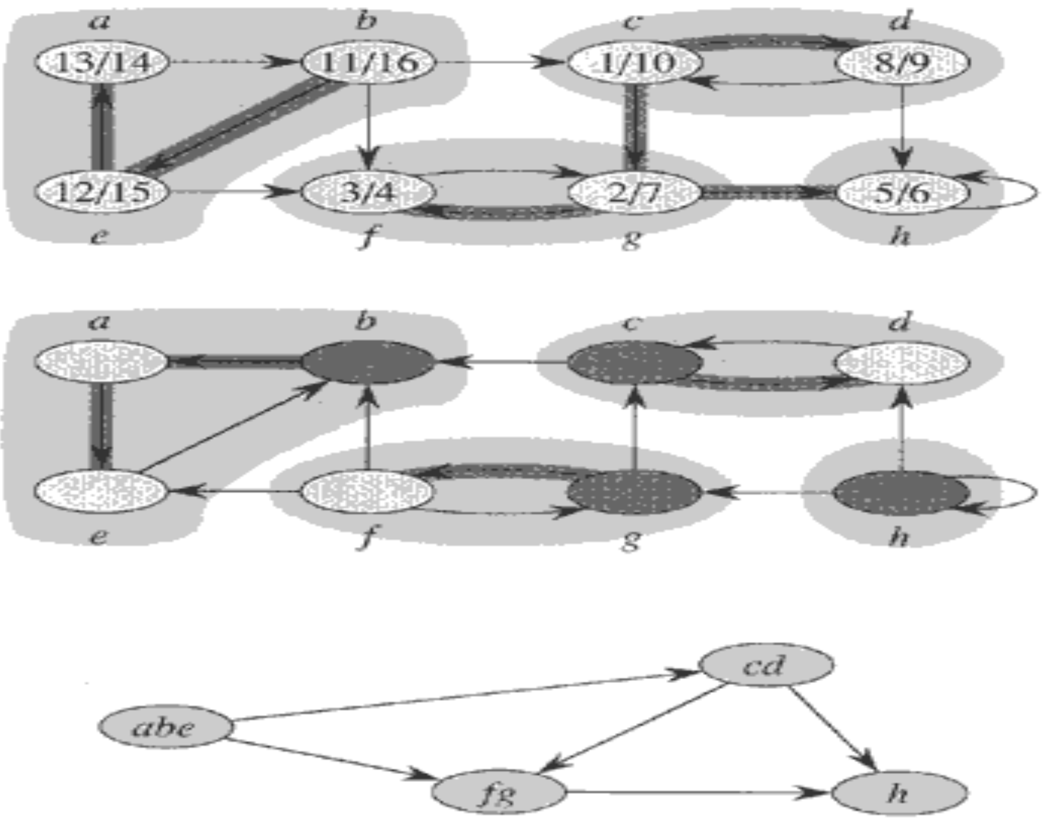
Vertex D: The next unvisited vertex in dfsList is D; dfs(D) returns the list [D, F, G] whose elements form the last strongly connected component..

Strongly Connected Components

(a)

(b)

(c)

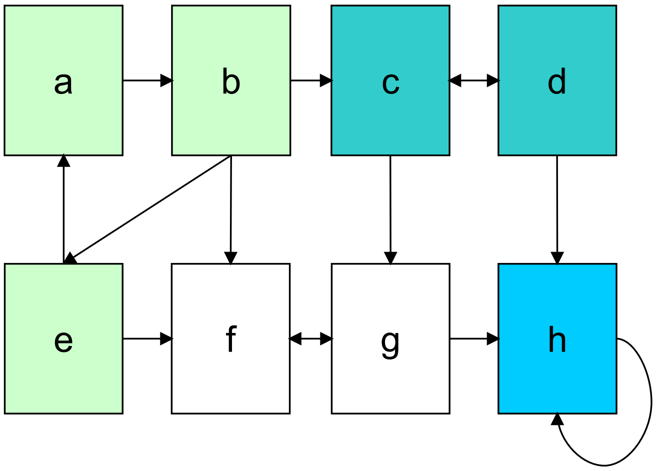


Running Time of strongComponents()

- Recall that the depth-first search has running time O(V+E), and the computation for GT is also O(V+E). It follows that the running time for the algorithm to compute the strong components is O(V+E).

Exercise

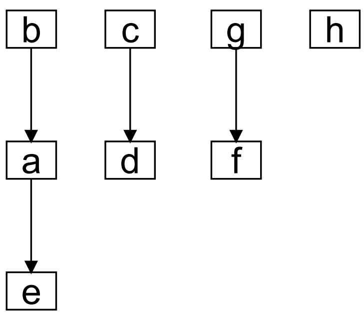
Find the strongly-connected components of the following graph

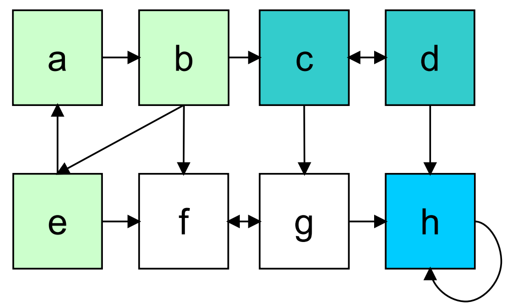


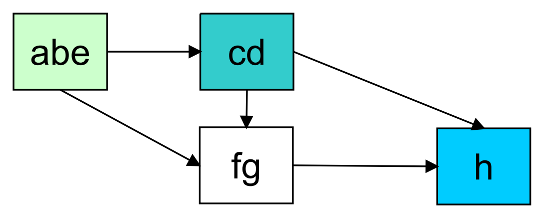
Strongly-Connected Components

These are the 4 trees that result, yielding the strongly connected components.

Finally, merge the nodes of any given tree into a super-node, and draw links between them, showing the resultant acyclic component graph.



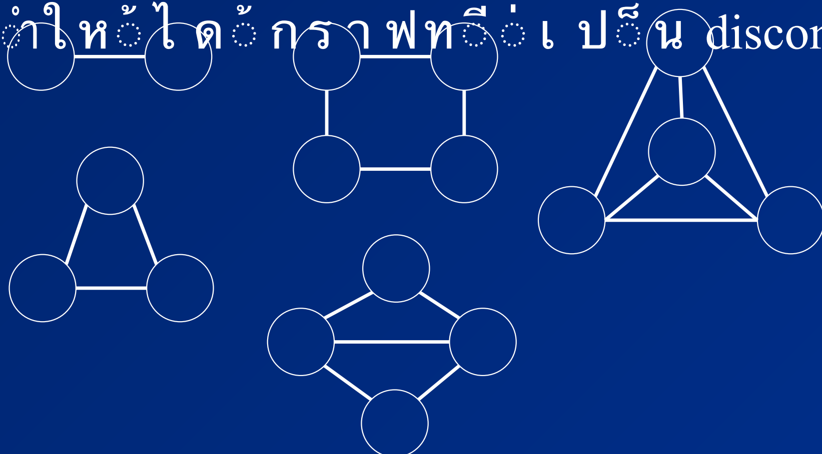


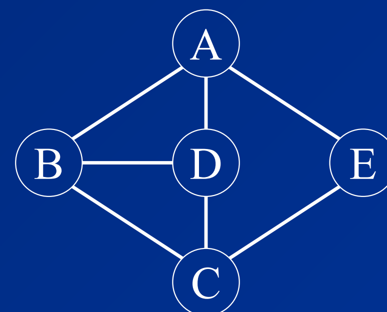


Component Graph

9.6.2. Biconnectivity

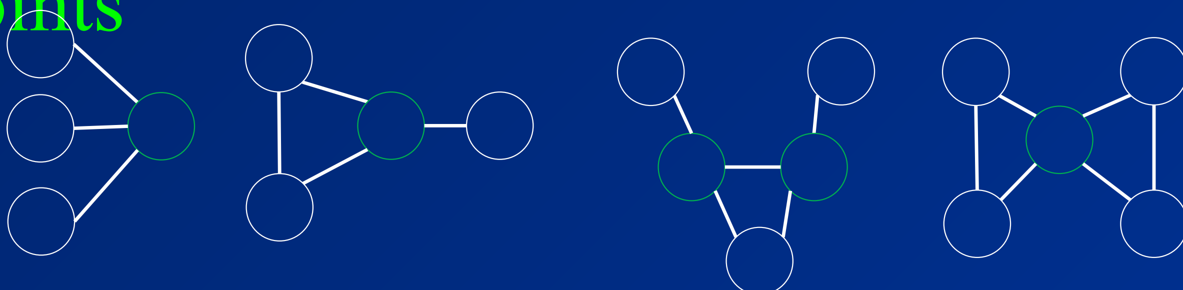
connected undirected graph เป็น **biconnected** ถ้าไม่มี vertices ที่เมื่อย้ายมันออกจากกราฟแล้ว ทำให้ได้กราฟที่เป็น disconnects





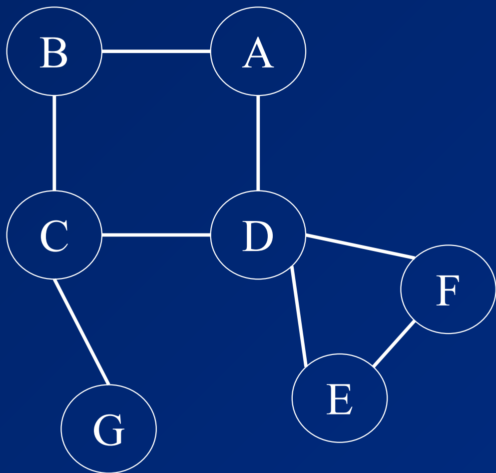
กราฟในรูปเป็น biconnected

กราฟที่ไม่เป็น biconnected จะมี vertices ที่เมื่อย้ายออกแล้วทำให้กราฟเป็น disconnect เรียกว่า articulation points



กราฟในรูปไม่เป็น biconnecte

- Depth-first search ทำให้การค้นหา articulation points ทั้งหมดใน connected graph ใช้เวลาเป็น linear-time
  - ทำ depth-first search เริ่มต้นที่ vertex ใด ๆ และให้หมายเลขโนดเมื่อเข้าถึงตามลำดับ preorder number แต่ละ vertex v และเรียกหมายเลขนี้ว่า Num(v)
  - จากนั้น สำหรับทุก ๆ vertex v ใน depth-first search spanning tree ให้หา



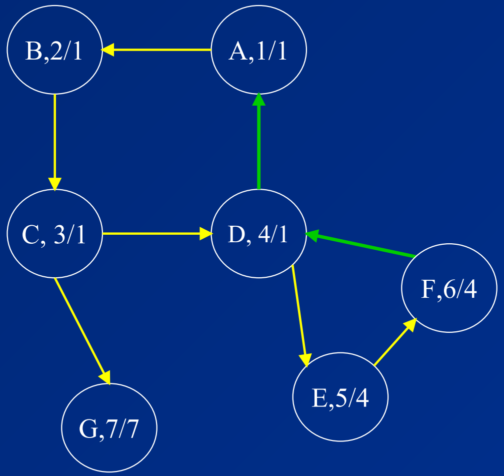


Figure 9.62 graph มี articulation points C และ D

Figure 9.63 Depth-first tree ของกราฟและค่า num และ low

- depth-first search tree ใน Figure 9.63 แสดง preorder number และตามด้วยหมายเลขต่ำสุดของ vertex ที่ไปถึงได้ตามกฎข้างบน
  - หมายเลข vertex ต่ำสุดที่ไปถึงได้โดย A, B, และ C คือ vertex 1 (A) เนื่องจากทั้งหมดนั้นใช้ 3 tree edges ไปยัง D และอีกหนึ่ง back edge เพื่อไปยัง A
- เราสามารถคำนวณค่า low

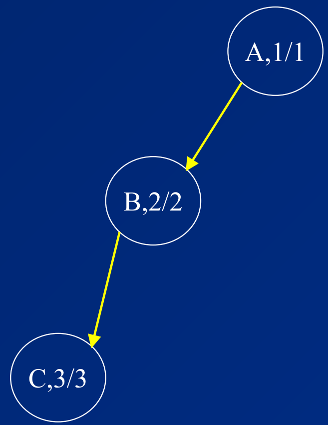
- จากนิยามค่าของ Low(v)

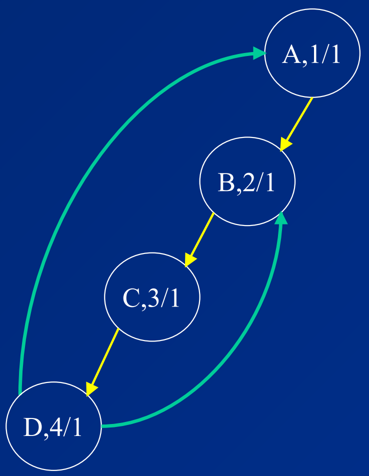
คือค่าที่น้อยที่สุดของ

1. Num(v)
2. ค่า Num(w) ที่น้อยที่สุดในกลุ่ม back edges (v, w)
3. ค่า Low(w) ที่น้อยที่สุดใน tree edges (v, w)

เงื่อนไขแรกเกิดเมื่อไม่มีการใช้ edges

ข้อสองเกิดเมื่อไม่มีการใช้ tree





1. Num(v)

2. ค่า Num(w) ที่น้อยที่สุด

3. ค่า Low(w) ที่น้อยที่สุด

ในกลุ่ม back edges (v, w) ใน tree edges (v, w)

- เนื่องจากเราจะหาค่า low ของโนดลูกทั้งหมดของ v ก่อนที่เราจะได้ค่า Low(v) นั่นคือเป็น postorder traversal
- กล่าวสำหรับแต่ละ edge (v, w) แล้ว เราสามารถบอกได้ว่ามันเป็น tree edge หรือ back edge ได้ด้วยการตรวจดูค่า Num(v) และ Num(w) ดังนั้นไม่ยากที่จะคำนวณค่า Low

- สิ่งที่เหลืออยู่คือ การใช้สารสนเทศที่ได้มาในการหา articulation points
  - รากของ tree เป็น articulation point ถ้าหากว่ามันมีโนดลูกมากกว่าหนึ่งโนด ทั้งนี้เพราะว่าการย้ายโนดรากจะทำให้ subtrees ของมันเป็น disconnects
  - ส่วน vertex v อื่นๆ จะเป็น articulation point ถ้าหากว่า v มีโนดลูก w ใดๆ ที่มี Low(w) $\boxed{?}$

Figure 9.64

แสดงผลการใช้อัลกอริทึมกับก
ราฟตัวอย่างโดยเริ่มที่โนด C

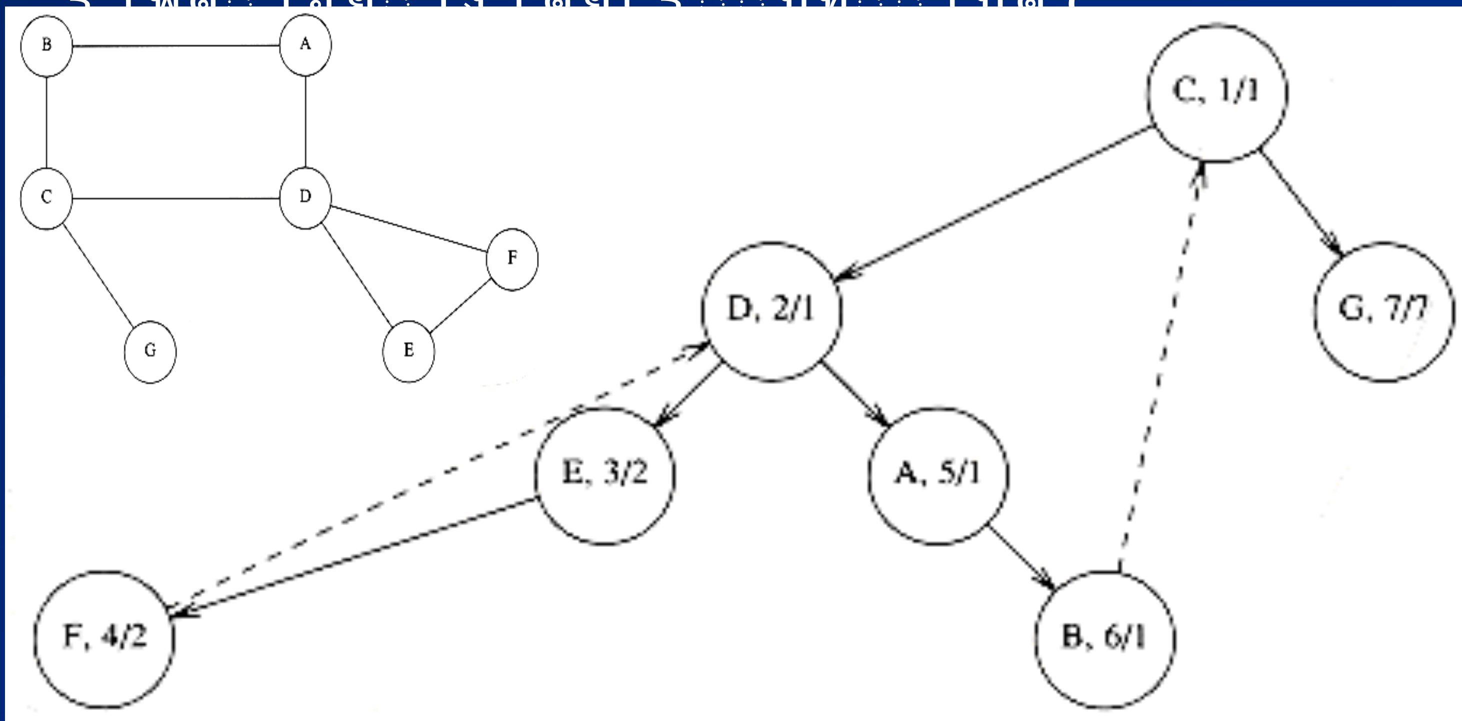


Figure 9.64 Depth-first tree โดยเริ่มที่ C

- pseudocode

  สำหรับอัลกอริทึมแสดงไว้ใน Figure 9.65 ถึง Figure 9.67

  - ให้ Vertex ประกอบด้วย คือ visited (initialized to false), num, low, และ parent
  - ทำการบรรทึกตัวแปรของคลาส Graph ชื่อ counter, โดยกำหนดค่าเริ่มต้นเป็น 1 เพื่อใช้เป็นตัวกำหนดค่า preorder traversal numbers คือ num

- ดังที่กล่าว เราสามารถคำนวณ

- การ traverse สามครั้ง ดูจะสิ้นเปลือง
  - คำสั่งใน Figure 9.65 เป็นการทำงานรอบแรก
  - การทำงานรอบที่สองและสาม ซึ่งเป็น postorder traversals ทำได้ด้วยโปรแกรมในรูป Figure 9.66 โดยบรรทัดที่ 8 ใช้เพื่อจัดการกรณีพิเศษ คือ ถ้า w เป็น adjacent ของ v แล้วการทำ recursive call กับ w

```
/* assign num and compute parents */
void  assignNum( Vertex v )
{
            vertex w;
/*1*/    v.num = counter++;
/*2*/    v.visited = true;
/*3*/    for each w adjacent to v
/*4*/        if ( !w.visited )
                {
/*5*/            w.parent = v;
/*6*/            assignNum( w );
                }
}
```

Figure 9.65 Routine Num เพื่อกำหนดค่าให้แก่ vertices (pseudocode)

Pseudocode

เพื่อคำนวณค่า low และทดสอบการเป็น articulation points (โดยไม่มีการทดสอบราก)

```
void assignLow( Vertex v )
{
                vertex w;
/*1*/       v.low = v.num;              /* Rule 1 */
/*2*/       for each w adjacent to v
                {
/*3*/           if  ( w.num > v.num )   /* forward edge */
                   {
/*4*/                assignLow( w );
/*5*/                if  ( w.low >= v.num )
/*6*/                    System.out.println ( v + " is an articulation point" );
/*7*/                v.low = min( v.low, w.low );            /* Rule 3 */
                   }
                   else
/*8*/           if  ( v.parent != w )       /* back edge */
/*9*/               v.low = min( v.low, w.num );            /* Rule 2 */
                }
}
```

Figure 9.67 ทดสอบ articulation points ใน depth-first search (ไม่รวมราก) (pseudocode)

```
void findArt ( Vertex v )
{
                vertex w;
/*1*/       v.visited = true;
/*2*/      v.low = v.num = counter++;    /* Rule 1 */
/*3*/      for each w adjacent to v
                {
/*4*/          if ( !w.visited )     /* forward edge */
                   {
/*5*/             w.parent = v;
/*6*/             findArt( w );
/*7*/             if  ( w.low >= v.num )
/*8*/                 System.out.println ( v + " is an articulation point" );
/*9*/             v.low = min( v.low, w.low );   /* Rule $ */
                   }
                   else
/*10*/         if  ( v.parent != w )                /* back edge */
/*11*/             v.low = min( v.low, w.num );       /* Rule 2 */
                }
```

9.6.3 Euler Circuits

- พิจารณารูป Figure 9.68
  - ปัญหาที่ต้องการคำตอบคือ สร้างรูปนี้ใหม่โดยลากเส้นแต่ละเส้นเพียงครั้งเดียวและไม่ยกดินสอขึ้นขณะเขียนรูป
  - ปัญหาอาจจะเพิ่มเติมขึ้นโดยให้เริ่มและจบที่จุดเริ่มต้น
- เราสามารถเปลี่ยนปัญหานี้เป็นปัญหาของทฤษฎีกราฟได้โดยกำหนดให้จุดตัดของเส้นเป็น vertex ส่วน edges

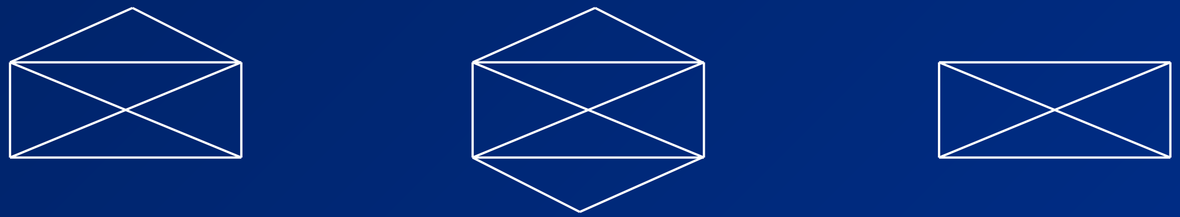


Figure 9.68 Three drawings

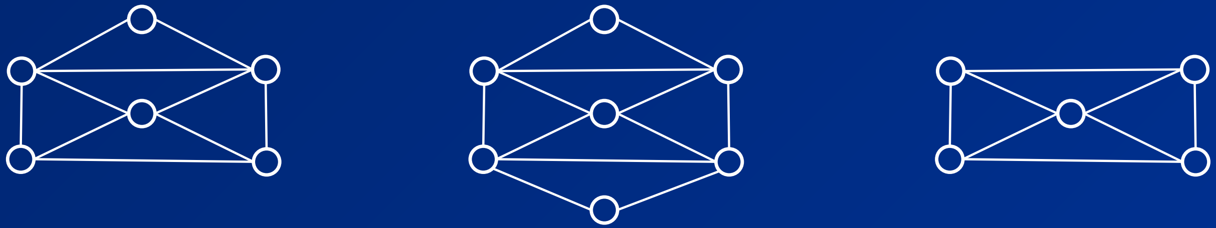


Figure 9.69 Conversion of puzzle to graph

- จากนั้นเราต้องหา path ในกราฟที่เข้าถึง edge แต่ละ edge เพียงครั้งเดียว
  - หรือต้องการแก้ปัญหาที่ต้องการหา cycle ที่เข้าถึงทุก edge เพียงครั้งเดียว (เริ่มและจบที่ vertex เดียวกัน)
- Euler แก้ปัญหาดังกล่าวนั้นได้ในปี 1736 ปัญหานี้เรียกว่า Euler path (หรือ Euler tour) หรือ Euler circuit problem แล้วแต่ตัวปัญหา

- กราฟที่เป็น Euler circuit ได้นั้นกราฟจะต้องเป็นแบบ connected และแต่ละ vertex จะต้องมีจำนวน edge เป็นเลขคู่เท่านั้น (มีเส้นทางเข้า และเส้นทางออก จึงจะใช้ edge ครั้งเดียว ในการเดินทางผ่าน vertex ได้)
- กราฟที่เป็น Euler path มี vertices ที่มีจำนวน edge เป็นเลขคี่ได้ 2 vertise โดยเริ่มที่ vertex ที่มี edge

Hierholzer's algorithm

Hierholzer's 1873 paper provides a different method for finding Euler cycles that is more efficient than Fleury's algorithm:

Choose any starting vertex v, and follow a trail of edges from that vertex until returning to v. It is not possible to get stuck at any vertex other than v, because the even degree of all vertices ensures that, when the trail enters another vertex w there must be an unused edge leaving w. The tour formed in this way is a closed tour, but may not cover all the vertices and edges of the initial graph.

As long as there exists a vertex u that belongs to the current tour but that has adjacent edges not part of the tour, start another trail from u, following unused edges until returning to u, and join the tour formed in this way to the previous tour.

Algorithm

Perform DFS from some vertex v until return to v along path p

If some part of graph not include,

perform DFS from first vertex v' on p has an un-traversed edge (path p')

splice p' into p

continue until all edges traversed

- ดังนั้นกราฟแบบ connected ใดๆ และ vertices ทั้งหมดมี edge เป็นจำนวนคู่ แสดงว่ากราฟนั้นมี Euler circuit และสามารถหา circuit ได้ด้วยเวลาเป็น linear time โดยใช้ depth-first search
- พิจารณากราฟรูป Figure 9.70 ซึ่งมี Euler circuit
  - ถ้าเริ่มต้นที่ vertex 5 และ traverse ไปตามวงจร 5, 4, 10, 5 ซึ่งมีสภาวะดังรูป Figure 9.71
  - จากนั้นเริ่มใหม่ที่ vertex 4 และมี depth-first search เป็น 4, 1, 3, 7, 4, 11,

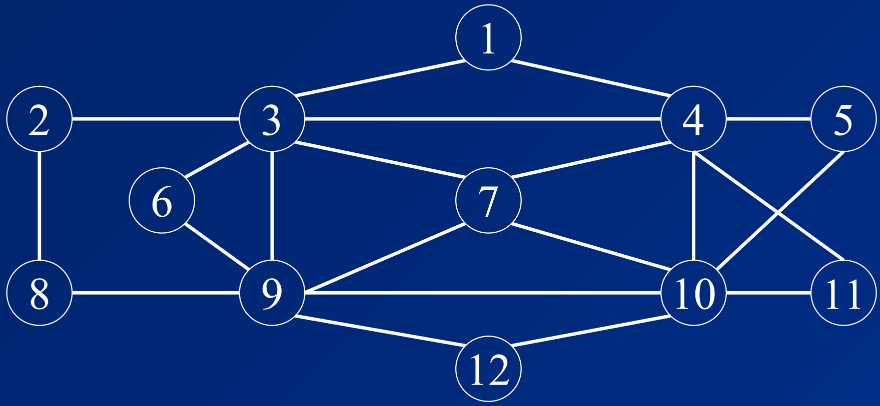


Figure 9.70 Graph for Euler circuit problem

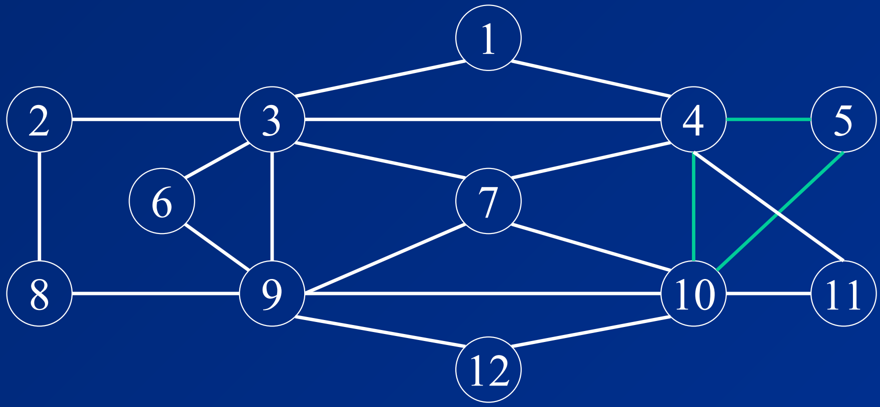


Figure 9.71 Graph remaining after 5, 4, 10, 5

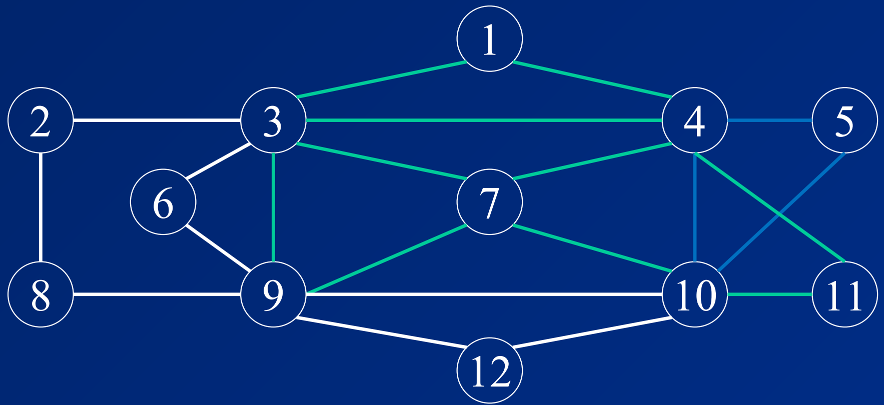


Figure 9.72 Graph after the path 5, 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5

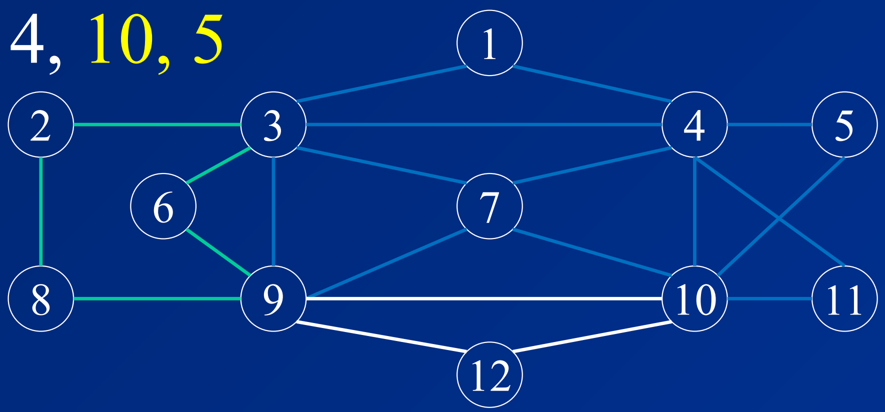


Figure 9.73 Graph remaining after the path 5, 4, 1, 3, 2, 8, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5

- เพื่อให้อัลกอริทึมมีประสิทธิภาพ เราจะต้องใช้โครงสร้างข้อมูลที่เหมาะสม โดยอาจจะใช้:
  - การเก็บ path ในลักษณะ linked list
  - เพื่อหลีกเลี่ยงการที่จะต้อง scan ใน adjacency lists ซ้ำๆ หลายครั้ง ดังนั้นสำหรับแต่ละ adjacency list เราจึงต้องเก็บ pointer ที่ชี้ไปยัง edge ตัวสุดท้ายที่ scan แล้ว (edge ต้องถูก access ครั้งเดียว)
  - เมื่อทำการแทรกเส้นทาง เราจะทำการหา vertex ใหม่ด้วยการเริ่มต้นที่จุดแทรก
- เวลาที่ใช้ทั้งหมดในขั้นตอนการหา vertex เป็น O(|E|)

9.6.4. Directed Graphs

- การท่องไปใน directed graphs ก็สามารถทำได้ด้วยเวลาเป็น linear time โดยการใช้ depth-first search เช่นเดียวกับใน undirected graphs ดังนี้
  - ถ้ากราฟไม่เป็น strongly connected และการเริ่มการค้นหาที่โนดใด ๆ ใน depth-first อาจจะไม่สามารถท่องไปในทุก ๆ โนดได้ ในกรณีเช่นนี้ เราจะทำ depth-first searches ซ้ำ

- พิจารณา digraph ในรูป Figure 9.74.
  - เราเริ่มการทำ depth-first search ที่ vertex B (ที่ vertex อื่นก็ได้) ซึ่งทำให้ท่องไปใน vertices B, C, A, D, E, และ F
  - จากนั้นก็เริ่มใหม่ที่ vertex ที่ยังไม่ได้ท่องไป ในที่นี้เลือกที่จะเริ่มใหม่ที่ H ซึ่งจะท่องไปใน I และ J
  - สุดท้ายเริ่มใหม่ที่ G ซึ่งเป็น vertex สุดท้ายที่จะท่องไป
  - Figure 9.75 แสดง depth-first search tree

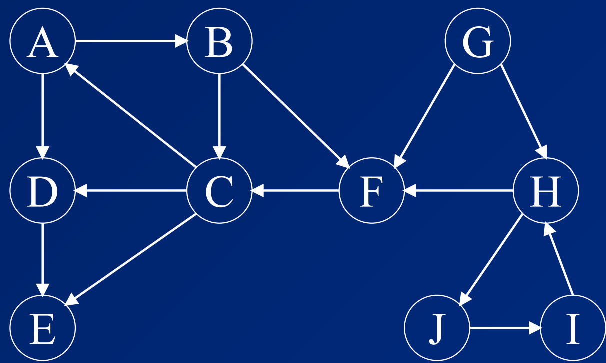


Figure 9.74 directed graph

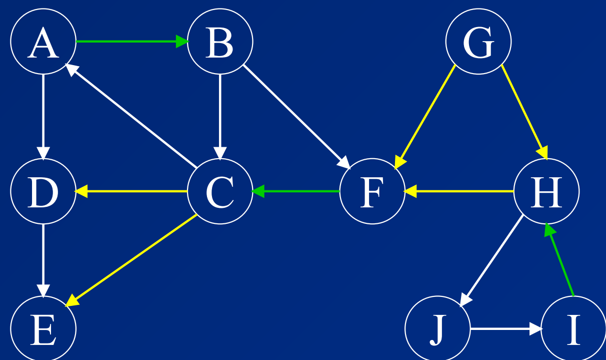


Figure 9.75 Depth-first search ของกราฟรูป 9.74

- จะเห็นว่ามี edges อยู่สามชนิดที่ไม่ได้เป็นเส้นที่นำไปสู่ vertices ใหม่
  - back edges ดังเช่น (A, B) และ (I, H)
  - forward edges ดังเช่น (C, D) และ (C, E) จาก node ของ tree ไปยังโนดที่ตามมา
  - cross edges ดังเช่น (F, C) และ (G, F) ซึ่งเชื่อมต่อโนดของ tree 2 tree ที่ไม่เกี่ยวข้องกันโดยตรง

- ประโยชน์อย่างหนึ่งจาก depth-first search คือใช้เพื่อทดสอบว่า directed graph เป็น acyclic หรือไม่

  กฎก็คือ directed graph เป็น acyclic ถ้าหากว่ามันไม่มี back edges

- คงจำได้ว่าเราใช้ topological sort เพื่อการนี้ได้เช่นกัน
  - การทำ topological sorting อีกทางหนึ่งคือ กำหนดค่า topological numbers ให้แก่ vertices เป็น n, n - 1, . . . ,1 ด้วยการทำ postorder traversal ใน depth-first

PROOF OF CORRECTNESS

Check Point: Edge classification by DFS

Edge (u,v) of G is classified as a:

(1) Tree edge iff u discovers v during the DFS: P[v] = u

If (u,v) is NOT a tree edge then it is a:

(2) Forward edge iff u is an <u>ancestor</u> of v in the DFS tree

(3) Back edge iff u is a <u>descendant</u> of v in the DFS tree

(4) Cross edge iff u is <u>neither</u> an ancestor nor a descendant of v

Check Point: Edge classification by DFS

Edge (u,v) of G is classified as a:

(1) **Tree edge** iff u discovers v during the DFS: P[v] = u

If (u,v) is NOT a tree edge then it is a:

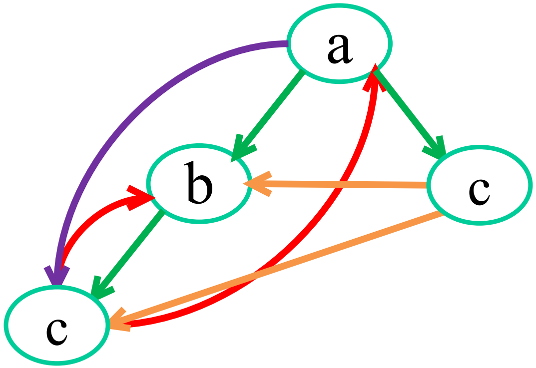
(2) **Forward edge** iff u is an <u>ancestor</u> of v in the DFS tree

(3) **Back edge** iff u is a <u>descendant</u> of v in the DFS tree

(4) **Cross edge** iff u is <u>neither</u> an ancestor nor a descendant of v

Edge classification by DFS

Tree edges
Forward edges
Back edges
Cross edges

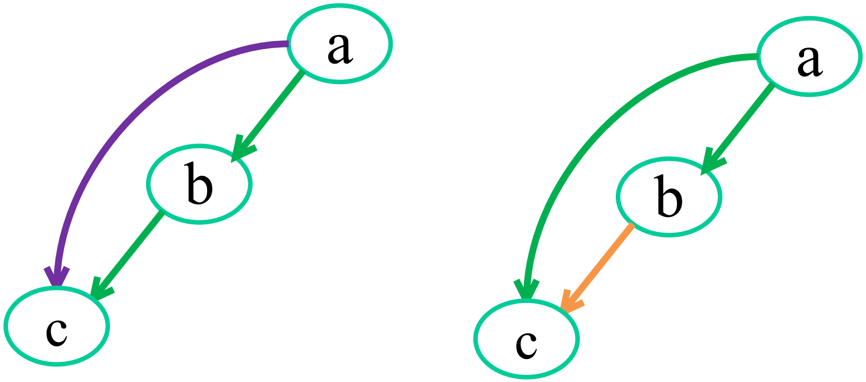


The edge classification depends on the particular DFS tree!

Edge classification by DFS

Tree edges
Forward edges
Back edges
Cross edges

Both are valid



The edge classification depends on the particular DFS tree!

DAGs and back edges

- Can there be a back edge in a DFS on a DAG ?
- NO! Back edges close a cycle!
- A graph G is a DAG <=> there is no back edge classified by DFS(G)

Back to topological sort

- TOPOLOGICAL-SORT(G):
  1) call DFS(G) to compute finishing times f[v] for each vertex v
  2) as each vertex is finished, insert it onto the front of a linked list
  3) return the linked list of vertices

Proof of correctness

- Theorem: TOPOLOGICAL-SORT(G) produces a topological sort of a DAG G
- The TOPOLOGICAL-SORT(G) algorithm does a DFS on the DAG G, and it lists the nodes of G in order of decreasing finish times f[]
- We must show that this list satisfies the topological sort property, namely, that for every edge (u,v) of G, u appears before v in the list
- Claim: For every edge (u,v) of G: $f[v] < f[u]$ in DFS

Proof of correctness

"For every edge (u,v) of G, $f[v] < f[u]$ in this DFS"

- The DFS classifies (u,v) as a tree edge, a forward edge or a cross-edge (it cannot be a back-edge since G has no cycles):

i. If (u,v) is a tree or a forward edge $\Rightarrow$ v is a descendant of u $\Rightarrow$ $f[v] < f[u]$

ii. If (u,v) is a cross-edge

Proof of correctness

"For every edge (u,v) of G: f[v] < f[u] in this DFS"

ii. If (u,v) is a cross-edge:

Q.E.D. of Claim

- as (u,v) is a cross-edge, by definition, neither u is a descendant of v nor v is a descendant of u:

d[u] < f[u] < d[v] < f[v]

or

d[v] < f[v] < d[u] < f[u]

since (u,v) is an edge, v is surely discovered before u's exploration completes

$\boxed{f[v] < f[u]}$

Proof of correctness

- TOPOLOGICAL-SORT(G) lists the nodes of G from highest to lowest finishing times
- By the Claim, for every edge (u,v) of G:

$$f[v] < f[u]$$

$\Rightarrow$ u will be before v in the algorithm's list

- Q.E.D of Theorem