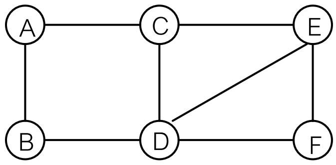
บทที่ 8

กราฟ (Graph)

โครงสร้างข้อมูลแบบกราฟ จะถูกนำไปประยุกต์ใช้ในงานต่าง ๆ มากมาย เช่น ถนน, ทางรถไฟ, เส้นทางการบิน, การเชื่อมโยงเครือข่ายคอมพิวเตอร์ และอื่นๆ

8.1 นิยาม

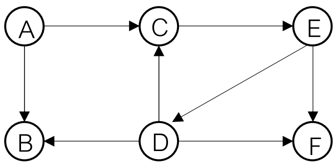
กราฟแทนด้วย G = (V, E) ซึ่ง V คือ เซ็ตของโหนด (vertex) และ E คือ เซ็ตของกิ่ง (edge) ซึ่งเชื่อมต่อระหว่างโหนดแต่ละคู่ จากรูป 8.1 กราฟ A ประกอบด้วย โหนด V={A, B, C, D, E, F} และ E={AB, AC, BD, CD, CE, DE, DF, EF}

A =


<u>รูป 8.1</u> กราฟ A

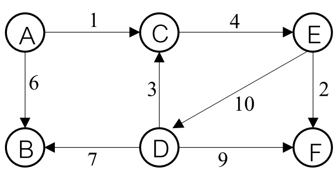
คำศัพท์ที่เกี่ยวกับกราฟ

1. **กราฟที่มีทิศทาง (directed graph)** คือ กราฟที่แต่ละกิ่งมีทิศทางกำหนดไว้ ตามรูป 8.2

A =


<u>รูป 8.2</u> กราฟที่มีทิศทาง A

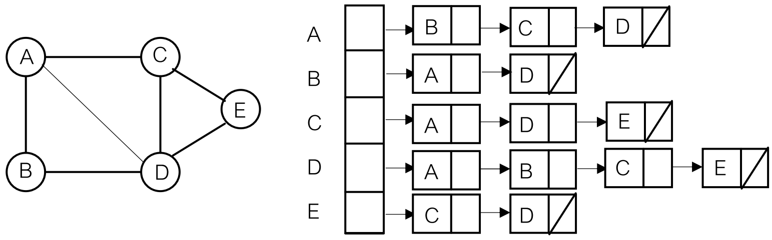
2. **กราฟที่ไม่กำหนดทิศทาง (undirected graph)** คือ กราฟที่แต่ละกิ่งไม่มีทิศทางกำหนดไว้ ตามรูป 8.1
3. **เส้นทาง (path)** คือ ลำดับของกิ่งที่เชื่อมโยงระหว่างโหนดสองโหนดใด ๆ
4. **ความยาวของเส้นทาง (path length)** คือ จำนวนกิ่งที่ประกอบขึ้นเป็นเส้นทาง
5. **วง (cycle)** คือ เส้นทางที่มีจุดเริ่มต้นและจุดสิ้นสุดเป็นจุดเดียวกัน เช่น {CE, ED, DC}
6. **กราฟที่เชื่อมถึงกัน (connected graph)** คือ กราฟที่โหนดแต่ละคู่มีเส้นทางเชื่อมโยงกัน
7. **กราฟที่มีน้ำหนัก (weighted graph)** คือ กราฟที่มีการกำหนดค่าให้กับแต่ละกิ่ง

A =


<u>รูปที่ 8.3</u> แสดงกราฟที่มีน้ำหนัก

8.2 โครงสร้างข้อมูลสำหรับกราฟ

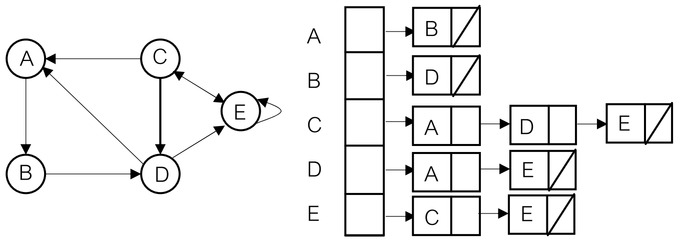
การแทนกราฟในเชิงคณิตศาสตร์ หรือในทางโปรแกรมที่นิยมใช้มี 2 แบบ คือ adjacency list และ adjacency matrix ตามรูป 8.4 และ 8.5



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	0
C	1	0	0	1	1
D	1	1	1	0	1
E	0	0	1	0	1

(ก) กราฟ G มี 5 โหนด 7 กิ่ง (ข) adjacency list แทนกราฟ G (ค) adjacency matrix

รูปที่ 8.4 การแทนกราฟไม่มีทิศทาง



	A	B	C	D	E
A	0	1	0	0	0
B	0	0	0	1	0
C	1	0	0	1	1
D	1	0	0	0	1
E	0	0	1	0	1

(ก) กราฟ G มี 5 โหนด 7 กิ่ง (ข) adjacency list แทนกราฟ G (ค) adjacency matrix

รูปที่ 8.5 การแทนกราฟมีทิศทาง

รูปที่ 8.4(ข) เป็นการแทนกราฟด้วย adjacency-list ของกราฟ G = (V, E) อาเรย์ของ list จะเท่ากับจำนวนโหนด = V และ list จำนวน = 2xE ในกรณีที่แทนกราฟไม่มีทิศทาง สำหรับกราฟมีทิศทาง ตามรูป 8.5 (ข) จะมีจำนวน list เท่ากับ E

รูปที่ 8.4, 8.5 (ค) เป็นการแทนกราฟ G = (V, E) ด้วย adjacency-matrix A ขนาด = [V][V] โดยที่

$$a[i][j] = \begin{cases} 1 & if\,(i, j) \in E \\ 0 & otherwise \end{cases}$$

a[i][j] มีค่า = 1 เมื่อมีกิ่งเชื่อมระหว่างโหนด i และ j มิฉะนั้น a[i][j] มีค่า = 0

ในกรณีกราฟไม่มีทิศทางจะได้เมทริกซ์ A เป็น เมทริกซ์สมมาตร (symmetry matrix) ตามรูป 8.4 (ค)

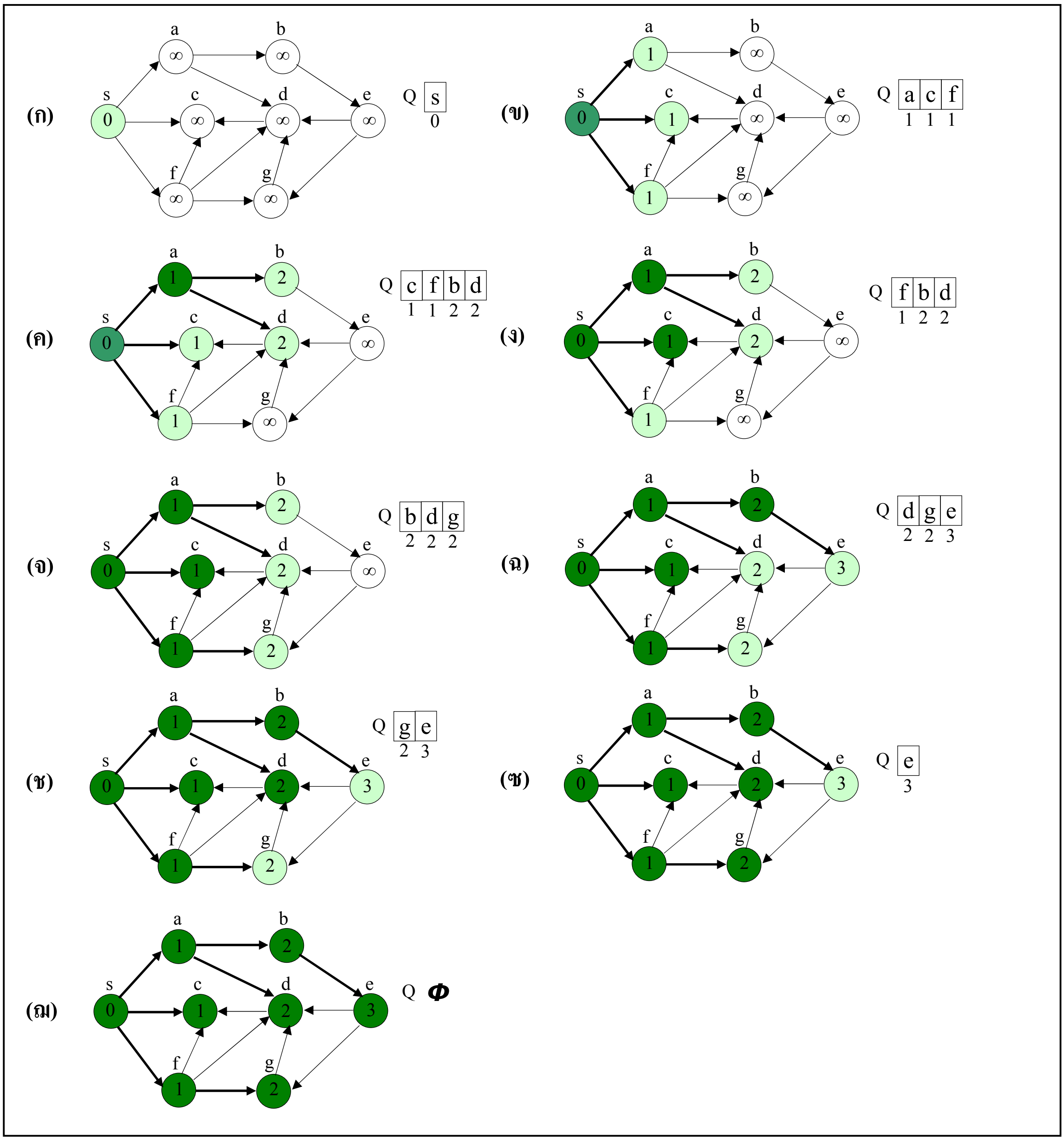
การแทนกราฟที่มีน้ำหนัก G = (V, E) และ w(u,v) เป็นน้ำหนักของกิ่งที่เชื่อมระหว่างโหนด u และ v จะกำหนดค่าให้กับสมาชิกของเมทริกซ์ A เท่ากับ

$$a[i][j] = \begin{cases} w[i][j] & if\,(i, j) \in E \\ \infty & otherwise \end{cases}$$

8.3 **การท่องกราฟในแนวนอน (Breadth-first search)**

การท่องกราฟในแนวนอนเป็นขั้นตอนวิธีที่จะสืบค้นกราฟจากโหนดเริ่มต้น s ไปที่โหนดอื่นๆที่อยู่ใกล้กับโหนดเริ่มต้น(s) ก่อนแล้วจึงสืบค้นกราฟไปยังโหนดที่อยู่ห่างออกไป(จำนวนกิ่งเชื่อมมากขึ้น)

ตัวอย่าง 8.1 แสดงการสืบค้นกราฟในแนวนอน



รูปที่ 8.5 การท่องกราฟแนวนอน(Breadth-first search)

<u>วิธีทำ</u> กำหนดให้

d[u] = distance (ระยะทาง) : คือจำนวนกิ่งที่ต้องผ่านจากโหนดเริ่มต้นมาที่โหนด u

¶[u] = path (ทางผ่าน) : คือโหนดที่ต้องผ่านก่อนที่จะถึงโหนด u

color : white = โหนดที่ยังไม่มีการแวะ

gray = โหนดที่กำลังอยู่ในคิว

black = โหนดที่สืบค้นแล้ว

1. ให้โหนดเริ่มต้น d[s] = 0 และ โหนดอื่นๆ มี d[u] = ∞, ¶[u]=nil จากนั้นใส่โหนด s ลงในคิว ตามรูป 8.5 (ก)

(ก) Q = {s}								
vertex	**s**	**a**	**b**	**c**	**d**	**e**	**f**	**g**
d	0	∞	∞	∞	∞	∞	∞	∞
¶	nil	nil	nil	nil	nil	nil	nil	nil
color	gray	white	white	white	white	white	white	white

2. ขณะที่ Q ยังไม่เป็นเซ็ตว่างให้ทำดังนี้

a. ให้ u = head(Q) //head(Q) คือหัวแถวของ Q

b. สำหรับแต่ละ v ∈ Adj(u) // Adj(u) คือ โหนดที่เชื่อมโดยตรงกับโหนด u

ถ้าเป็นโหนดที่ระบายสีขาว ให้ระบายสีเทา และ

d(v) = d(u)+1 และ $\pi(v) = u$ และใส่ค่า v ลงในคิว Q

c. ดึงหัวแถวของ Q ออกมา และระบายสีเป็นสีดำ เพื่อแสดงว่าได้สืบค้นแล้ว

จะได้ว่าดึง s ออกมาจากคิว พิจารณาโหนดที่ต่ออยู่กับ s ได้แก่โหนด a, c, f จะได้ d[a], d[c], d[f] = 1 และ¶[a], ¶[c], ¶[f] = u แล้วใส่ a, c, f ไปในคิวตามรูป 8.5 (ข)

(ข) Q = {a,c,f}								
vertex	**s**	**a**	**b**	**c**	**d**	**e**	**f**	**g**
d	0	1	∞	1	∞	∞	1	∞
¶	nil	s	nil	s	nil	nil	s	nil
color	black	gray	white	gray	white	white	gray	white

3. ขณะที่ Q ยังไม่เป็นเซ็ตว่าง จะดึง a ออกมาจากคิว จะได้ตามรูป 8.5 (ค)

(ค) Q = {c,f,b,d}								
vertex	**s**	**a**	**b**	**c**	**d**	**e**	**f**	**g**
d	0	1	2	1	2	∞	1	∞
¶	nil	s	a	s	a	nil	s	nil
color	black	black	gray	gray	gray	white	gray	white

4. ขณะที่ Q ยังไม่เป็นเซ็ตว่าง จะดึง c ออกมาจากคิว จะได้ตามรูป 8.5 (ง)

(ง) Q = {f,b,d}								
vertex	**s**	**a**	**b**	**c**	**d**	**e**	**f**	**g**
d	0	1	2	1	2	∞	1	∞
¶	nil	s	a	s	a	nil	s	nil
color	black	black	gray	black	gray	white	gray	white

5. ขณะที่ Q ยังไม่เป็นเซ็ตว่าง จะดึง f ออกมาจากคิว จะได้ตามรูป 8.5 (จ)

(จ) Q = {b,d,g}								
vertex	**s**	**a**	**b**	**c**	**d**	**e**	**f**	**g**
d	0	1	2	1	2	∞	1	2
¶	nil	s	a	s	a	nil	s	f
color	black	black	gray	black	gray	white	black	gray

6. ขณะที่ Q ยังไม่เป็นเซ็ตว่าง จะดึง b ออกมาจากคิว จะได้ตามรูป 8.5 (ฉ)

(ฉ) Q = {d,g,e}								
vertex	**s**	**a**	**b**	**c**	**d**	**e**	**f**	**g**
d	0	1	2	1	2	3	1	2
¶	nil	s	a	s	a	b	s	f
color	black	black	black	black	gray	gray	black	gray

7. ขณะที่ Q ยังไม่เป็นเซ็ตว่าง จะดึง d ออกมาจากคิว จะได้ตามรูป 8.5 (ช)

(ช) Q = {g,e}								
vertex	**s**	**a**	**b**	**c**	**d**	**e**	**f**	**g**
d	0	1	2	1	2	3	1	2
¶	nil	s	a	s	a	b	s	f
color	black	black	black	black	black	gray	black	gray

8. ขณะที่ Q ยังไม่เป็นเซ็ตว่าง จะดึง g ออกมาจากคิว จะได้ตามรูป 8.5 (ซ)

(ซ) Q = {e}								
vertex	**s**	**a**	**b**	**c**	**d**	**e**	**f**	**g**
d	0	1	2	1	2	3	1	2
¶	nil	s	a	s	a	b	s	f
color	black	black	black	black	black	gray	black	black

9. ขณะที่ Q ยังไม่เป็นเซ็ตว่าง จะดึง e ออกมาจากคิว จะได้ตามรูป 8.5 (ฌ)

(ฌ) Q = {}								
vertex	**s**	**a**	**b**	**c**	**d**	**e**	**f**	**g**
d	0	1	2	1	2	3	1	2
¶	nil	s	a	s	a	b	s	f
color	black	black	black	black	black	black	black	black

เป็นการท่องกราฟในแนวนอนครบทุกโหนดแล้ว จะได้

s → a : {s, a}

s → b : {s, a, b}

s → c : {s, a}

s → d : {s, a, d}

s → e : {s, a, b, e}

s → f : {s, f}

s → g : {s, f, g}

<u>โปรแกรม 8.1</u> bfs.java

```
// bfs.java
// demonstrates breadth-first search
// to run this program: C>java BFSApp
import java.awt.*;
////////////////////////////////////////////////////////////////
class Queue
   {
   private final int SIZE = 20;
   public int[] queArray;
   public int front;
   public int rear;

   public Queue()            // constructor
      {
         queArray = new int[SIZE];
         front = 0;
         rear = -1;
      }
   public void enQueue(int j) // put item at rear of queue
      {
        if(rear == SIZE-1)
           rear = -1;
        queArray[++rear] = j;
      }
   public int deQueue()       // take item from front of queue
      {
      int temp = queArray[front++];
      if(front == SIZE)
        front = 0;
      return temp;
      }
   public boolean isEmpty()  // true if queue is empty
      {
      return ( rear+1==front || (front+SIZE-1==rear) );
      }
   }  // end class Queue
////////////////////////////////////////////////////////////////
class Vertex
   {
   public char label;        // label (e.g. 'A')
   public boolean wasVisited;
   public int distance;
   public char path;
   public int pathNo;
```

```
// -----------------------------------------------------------
   public Vertex(char lab)   // constructor
      {
      label = lab;
      wasVisited = false;
      distance = 1000;
      path = ' ';
      }
// -----------------------------------------------------------
   }  // end class Vertex
////////////////////////////////////////////////////////////////
class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // list of vertices
   private int adjMat[][];      // adjacency matrix
   private int nVerts;          // current number of vertices
   private Queue theQueue;
// ------------------
   public Graph()               // constructor
      {
         vertexList = new Vertex[MAX_VERTS];
                                // adjacency matrix
         adjMat = new int[MAX_VERTS][MAX_VERTS];
         nVerts = 0;
        for(int j=0; j<MAX_VERTS; j++)      // set adjacency
           for(int k=0; k<MAX_VERTS; k++)   //    matrix to 0
                adjMat[j][k] = 0;
         theQueue = new Queue();
      }  // end constructor
// -----------------------------------------------------------
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
      }
// -----------------------------------------------------------
   public void addEdge(int start, int end)
      {
         adjMat[start][end] = 1;
         adjMat[end][start] = 1;
      }
// -----------------------------------------------------------
   public void displayVertex(int v)
      {
        System.out.println("visits :  " + vertexList[v].label);
        System.out.println("distance =  " + vertexList[v].distance);
```

```
      System.out.println("path =  " + vertexList[v].path);
    }
// -------------------------------------------------------------
  public void bfs()                  // breadth-first search
    {                                // begin at vertex 0
    vertexList[0].wasVisited = true; // mark it
    vertexList[0].distance = 0; // mark it
    displayVertex(0);                // display it
    theQueue.enQueue(0);             // insert at tail
    int v2;
    while( !theQueue.isEmpty() )     // until queue empty,
      {     int v1 = theQueue.deQueue();   // remove vertex at head
            // until it has no unvisited neighbors
            while( (v2=getAdjUnvisitedVertex(v1)) != -1 )
              {                              // get one,
                  vertexList[v2].wasVisited = true;  // mark it
                vertexList[v2].distance =vertexList[v1].distance+1;  // mark it
                  vertexList[v2].path=vertexList[v1].label;
                  vertexList[v2].pathNo=v1;
                displayVertex(v2);               // display it
                theQueue.enQueue(v2);              // insert it
         }   // end while
      }  // end while(queue not empty)
    // queue is empty, so we're done
    for(int j=0; j<nVerts; j++)           // reset flags
      vertexList[j].wasVisited = false;
    }  // end bfs()
// -------------------------------------------------------------
  // returns an unvisited vertex adj to v
  public int getAdjUnvisitedVertex(int v)
    {
    for(int j=0; j<nVerts; j++)
      if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
        return j;
    return -1;
    }  // end getAdjUnvisitedVert()
// -------------------------------------------------------------
  public void printPath(int from, int to)
  { if  ( from == to)
         {System.out.print(vertexList[from].label+"\t"); }
    else if (vertexList[to].path ==' ')
         {System.out.println("no path from s to " +vertexList[to].label);}
    else
         {printPath(from, vertexList[to].pathNo);
         System.out.print(vertexList[to].label+"\t");}
    }//end printPath
```

```
  }  // end class Graph
////////////////////////////////////////////////////////////////
class BFSApp
  {
  public static void main(String[] args)
    {
    Graph theGraph = new Graph();
    theGraph.addVertex('s');    // 0  (start for bfs)
    theGraph.addVertex('a');    // 1
    theGraph.addVertex('b');    // 2
    theGraph.addVertex('c');    // 3
    theGraph.addVertex('d');    // 4
    theGraph.addVertex('e');    //5
    theGraph.addVertex('f');    // 6
    theGraph.addVertex('g');    // 7

    theGraph.addEdge(0, 1);
    theGraph.addEdge(0, 3);
    theGraph.addEdge(0, 6);
    theGraph.addEdge(1, 2);
    theGraph.addEdge(1, 4);
    theGraph.addEdge(2, 5);
    theGraph.addEdge(4, 3);
    theGraph.addEdge(5, 4);
    theGraph.addEdge(5, 7);
    theGraph.addEdge(6, 3);
    theGraph.addEdge(6, 4);
    theGraph.addEdge(6, 7);
    theGraph.addEdge(7, 4);

    theGraph.bfs();             // breadth-first search
    System.out.println("***** print path *****");
           for(int i = 1 ;i<=7;++i)
                    {theGraph.printPath(0,i);
                      System.out.println();
                    }
    }  // end main()
  }  // end class BFSApp
```

Outputs

```
visits :  s
distance =  0
path =
visits :  a
distance =  1
path =  s
visits :  c
distance =  1
path =  s
visits :  f
distance =  1
path =  s
visits :  b
distance =  2
path =  a
visits :  d
distance =  2
path =  a
visits :  g
distance =  2
path =  f
visits :  e
distance =  3
path =  b
*****print path*****
s	a
s	a	b
s	c
s	a	d
s	a	b	e
s	f
s	f	g
```

ประสิทธิภาพ (complexity)

complexity ของ graph G = (V, E) จะมีการ enqueue, dequeue จำนวน V ครั้งตามจำนวนโหนด = O(V) จากนั้นจะมีการไล่ไปตามกิ่งที่เชื่อมระหว่างโหนด อีกจำนวน E ครั้งตามจำนวนกิ่ง ดังนั้นเวลาทั้งหมดในการสืบค้นกราฟตามแนวนอน (Breadth-first search) คือ O(V+E)

8.4 **การท่องกราฟในแนวดิ่ง (Depth-first search)**

การท่องกราฟในแนวดิ่งเป็นขั้นตอนวิธีที่สืบค้นกราฟจากโหนดเริ่มต้น s แล้วทำการสืบค้นค่าในกราฟลงในแนวดิ่งก่อน ด้วยการสืบค้นไปตามกิ่งที่เชื่อมต่อจากโหนดดังกล่าว และทำการเก็บโหนดที่เชื่อมต่อลงในสแตกไปเรื่อยๆ จนเมื่อไม่มีโหนดเชื่อมต่อแล้ว ก็จะทำการย้อนรอย (backtrack) โดยการดึงโหนดออกจากสแตกเพื่อย้อนกลับไปยังโหนดที่มีกิ่งที่ยังไม่ได้สืบค้นเหลืออยู่ วิธีการนี้จะดำเนินต่อไปจนกว่าจะสืบค้นครบทุกโหนดในกราฟ ในระหว่างการสืบค้นจะใส่สีให้แต่ละโหนดเริ่มต้นเป็นสีขาว และเปลี่ยนเป็นสีเทาเมื่อแวะเข้าไปที่โหนดนั้น และจะเปลี่ยนเป็นสีดำเมื่อย้อนรอยกลับมาที่โหนดอีกครั้ง พร้อมกับมีเวลากำกับแต่ละโหนดเมื่อแวะ (discover time) และเมื่อย้อนรอย(finish time)

ตัวอย่าง 8.2 แสดงการสืบค้นในแนวดิ่ง

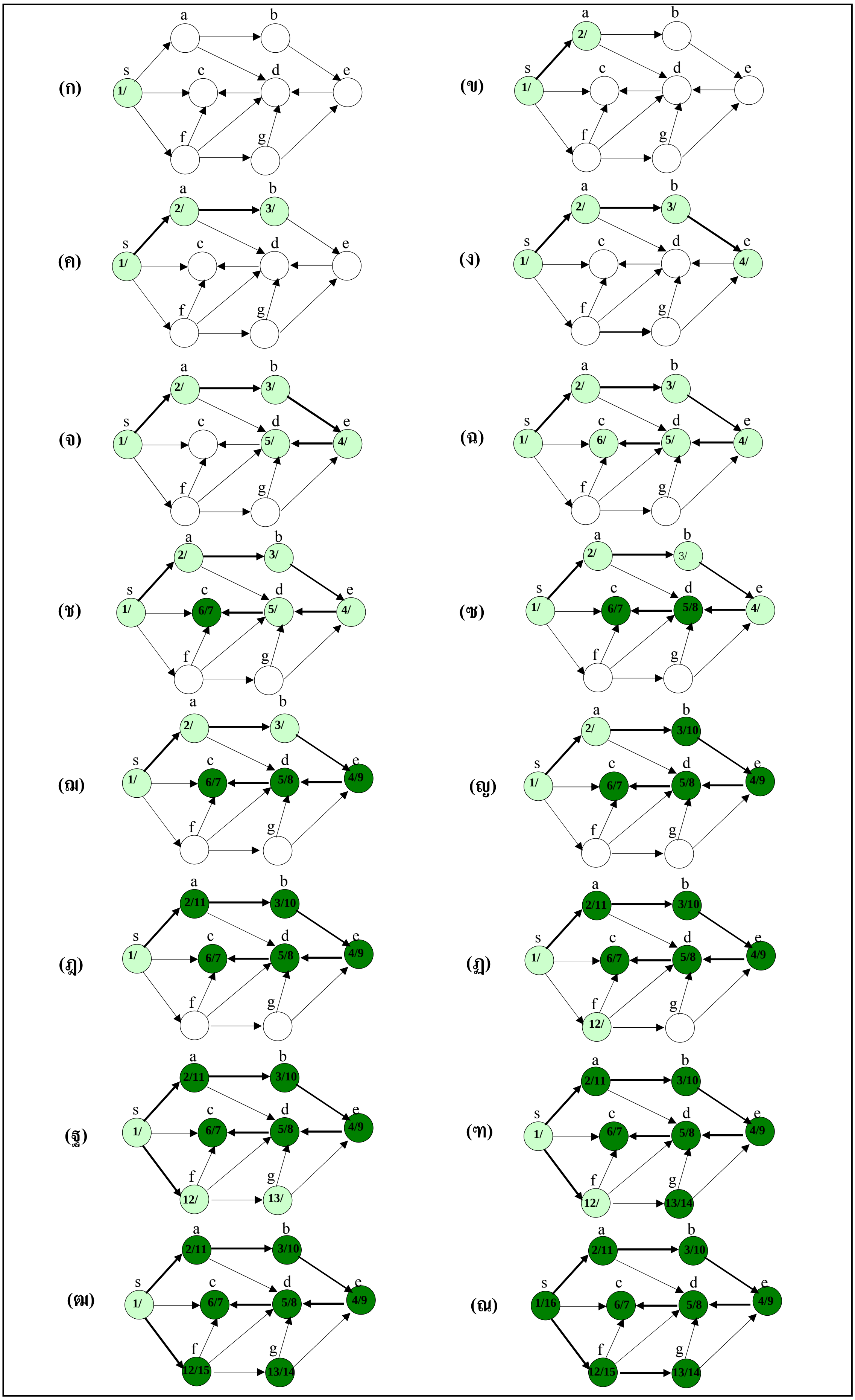
วิธีทำ กำหนดค่าเริ่มต้นแต่ละโหนด u ∈G ให้ระบายสีเป็นสีขาว และกำหนดให้ $\pi(u) = NIL$

ให้ time = 0 //เวลาเริ่มต้นในการสืบค้นเป็น 0

สำหรับแต่ละโหนด u ∈G ถ้าเป็นสีขาว ให้เริ่มสืบค้นโหนดต่างๆ โดยเรียก method dfsVisit(u) ตามลำดับในรูป ตามรูป 8.6 และ ตาราง 8.1

ตาราง 8.1 ลำดับการสืบค้นกราฟในแนวดิ่ง

	รูป	vertex	time	color	discover	finish	¶[u]	v ∈ adj[u]			stack
1	ก	s	1	g	1	-	-	a, c, f	push		s
2	ข	a	2	g	2	-	s	b,d	push		s,a
3	ค	b	3	g	3	-	a	e	push		s,a,b
4	ง	e	4	g	4	-	b	d	push		s,a,b,e
5	จ	d	5	g	5	-	e	c	push		s,a,b,e,d
6	ฉ	c	6	g	6	-	d	null			
7	ช	c	7	b	6	7	d		pop	backtrack	s,a,b,e
8	ซ	d	8	b	5	8	e		pop	backtrack	s,a,b
9	ฌ	e	9	b	4	9	b		pop	backtrack	s,a
10	ญ	b	10	b	3	10	a		pop	backtrack	s,a
11	ฎ	a	11	b	2	11	s	d color ‘b’	pop	backtrack	s
12	ฏ	f	12	g	12	-	s	g, c color’b’	pop,push		s,f
13	ฐ	g	13	g	13	-	f	e color ‘b’			
14	ฑ	g	14	b	13	14	f		pop	backtrack	s
15	ฒ	f	15	b	12	15	s		pop	backtrack	null
16	ณ	s	16	b	1	16					



<u>รูปที่ 8.6</u> การท่องกราฟในแนวดิ่ง (Depth-first search)

1. ตามรูป (ก) เริ่มจาก โหนด s
   ให้ time = time+1, d[s] = time = 1, color[s] = 'g'
   หาว่ามีโหนดใดที่เชื่อมต่อกับโหนด s ได้แก่ โหนด a, c, f สืบค้นโหนด a
   Push(s) ลงสแตก
2. สืบค้นโหนด a ตามรูป(ข)
   ให้ time = time+1, d[a] = time = 2, color[a] = 'g', ¶[a] = s
   หาว่ามีโหนดใดที่เชื่อมต่อกับโหนด a ได้แก่ โหนด b, Push(a) ลงสแตก
3. สืบค้นโหนด b ตามรูป(ค)
   ให้ time = time+1, d[b] = time = 3, color[b] = 'g', ¶[b] = a
   หาว่ามีโหนดใดที่เชื่อมต่อกับโหนด b ได้แก่ โหนด e, Push(b) ลงสแตก
4. สืบค้นโหนด e ตามรูป(ง)
   ให้ time = time+1, d[e] = time =4, color[e] = 'g', ¶[e] = b
   หาว่ามีโหนดใดที่เชื่อมต่อกับโหนด eได้แก่ โหนด d, Push(e) ลงสแตก
5. สืบค้นโหนด d ตามรูป(จ)
   ให้ time = time+1, d[d] = time =5, color[d] = 'g', ¶[d] = e
   หาว่ามีโหนดใดที่เชื่อมต่อกับโหนด d ได้แก่ โหนด c, Push(d) ลงสแตก
6. สืบค้นโหนด c ตามรูป(ฉ)
   ให้ time = time+1, d[c] = time =6, color[c] = 'g', ¶[c] = d
   ไม่มีโหนดใดที่เชื่อมต่อกับโหนด c
7. ที่โหนด c ตามรูป(ช)
   ให้ time = time+1, f[c] = time =7, color[c] = 'b'
   ย้อนรอยกลับ โดยการ Pop() จากสแตกจะได้โหนด d
8. ที่โหนด d ตามรูป(ซ)
   ให้ time = time+1, f[d] = time =8, color[d] = 'b'
   ย้อนรอยกลับ โดยการ Pop() จากสแตกจะได้โหนด e
9. ที่โหนด e ตามรูป(ฌ)
   ให้ time = time+1, f[e] = time =9, color[d] = 'b'
   ย้อนรอยกลับ โดยการ Pop() จากสแตกจะได้โหนด b
10. ทำการสืบค้นไปเรื่อยๆ เฉพาะโหนดสีขาว ถ้าโหนดสีดำก็ให้ข้ามโหนดนั้นไป จนครบทุกโหนด

<u>โปรแกรม 8.2</u> dfs.java

```
import java.awt.*;
////////////////////////////////////////////////////////////////
class StackX
  {
  private final int SIZE = 20;
  private int[] st;
  private int top;
  public StackX()           // constructor
    {
    st = new int[SIZE];    // make array
    top = -1;
    }
  public void push(int j)   // put item on stack
    { st[++top] = j; }
  public int pop()          // take item off stack
    { return st[top--]; }
  public int peek()         // peek at top of stack
    { return st[top]; }
  public boolean isEmpty()  // true if nothing on stack
    { return (top == -1); }
  }  // end class StackX
////////////////////////////////////////////////////////////////
class Vertex
  {
  public char label;        // label (e.g. 'A')
  public boolean wasVisited;
  public char color;        // label (e.g. 'A')
  public int pathNo;        // label (e.g. 'A')
  public int discover;        // label (e.g. 'A')
  public int finish;        // label (e.g. 'A')
// ------------------
  public Vertex(char lab)   // constructor
    {
    label = lab;
    wasVisited = false;
            color = 'w';
    }
// ------------------
  }  // end class Vertex
////////////////////////////////////////////////////////////////
class Graph
  {
  private final int MAX_VERTS = 20;
  private Vertex vertexList[]; // list of vertices
  private int adjMat[][];      // adjacency matrix
  private int nVerts;          // current number of vertices
  private StackX theStack;
```

```
  public int time;
// ------------------
  public Graph()              // constructor
    {
    vertexList = new Vertex[MAX_VERTS];
                                  // adjacency matrix
    adjMat = new int[MAX_VERTS][MAX_VERTS];
    nVerts = 0;
    for(int j=0; j<MAX_VERTS; j++)      // set adjacency
      for(int k=0; k<MAX_VERTS; k++)   //    matrix to 0
        adjMat[j][k] = 0;
    theStack = new StackX();
    }  // end constructor
// ------------------
  public void addVertex(char lab)
    {
    vertexList[nVerts++] = new Vertex(lab);
    }
// ------------------
  public void addEdge(int start, int end)
    {
    adjMat[start][end] = 1;
 //    adjMat[end][start] = 1;
    }
// ------------------
  public void displayVertex(int v)
    {int u;
    System.out.println("**************vertex =  "+vertexList[v].label);
    System.out.println("color =  "+vertexList[v].color);
           u=vertexList[v].pathNo;
    System.out.println("path =  "+vertexList[u].label);
    System.out.println("discover =  "+vertexList[v].discover);
    System.out.println("finish =  "+vertexList[v].finish);
    }
// ------------------
  public void dfs()  // depth-first search
    { time = 0;
                        for(int i=0; i<=7;++i)
                        {if (vertexList[i].color =='w')
                        {dfsVisit(i);}
                        }// end for
           }//
          //-------------------------------
  public void dfsVisit(int u)  // depth-first search
                          // begin at vertex 0
            {
    vertexList[u].wasVisited = true;  // mark it
    vertexList[u].color = 'g';  // mark it
           time = time+1;
```

```
    vertexList[u].discover = time;  // mark it
          displayVertex(u);                // display it
    theStack.push(u);                 // push it
         System.out.println("push \t" + vertexList[u].label);
         while( !theStack.isEmpty() )      // until stack empty,
      {
      // get an unvisited vertex adjacent to stack top
                    u = theStack.peek();
      int v = getAdjUnvisitedVertex( theStack.peek() );
      if(v == -1)                    // if no such vertex,
                              {theStack.pop();time=time+1;vertexList[u].finish = time;
                                      System.out.println("pop \t"+vertexList[u].label);
                                                       displayVertex(u);                // display it
         }
      else                          // if it exists,
        {
                              time = time+1;
       vertexList[v].wasVisited = true;  // mark it
        vertexList[v].color = 'g';
        vertexList[v].discover = time;
        vertexList[v].pathNo = u;
                              displayVertex(v);                // display it
              System.out.println("push \t" + vertexList[v].label);
        theStack.push(v);                // push it
        }
      }  // end while
      // stack is empty, so we're done
      for(int j=0; j<nVerts; j++)          // reset flags
        vertexList[j].wasVisited = false;
    }  // end dfs
// ------------------
  // returns an unvisited vertex adj to v
  public int getAdjUnvisitedVertex(int v)
    {
    for(int j=0; j<nVerts; j++)
      if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
        return j;
    return -1;
    }  // end getAdjUnvisitedVert()
// ------------------
  public void printPath(int from, int to)
           { if ( from == to)
           {System.out.print(vertexList[from].label+"\t"); }
                  else
           {
              printPath(from, vertexList[to].pathNo);
                          System.out.print(vertexList[to].label+"\t");
                  }
           }//end printPath
```

```
  }  // end class Graph
////////////////////////////////////////////////////////////////
class DFSApp
  {
  public static void main(String[] args)
    {
    Graph theGraph = new Graph();
    theGraph.addVertex('s');    // 0  (start for dfs)
    theGraph.addVertex('a');    // 1  (start for dfs)
    theGraph.addVertex('b');    // 2
    theGraph.addVertex('c');    // 3
    theGraph.addVertex('d');    // 4
    theGraph.addVertex('e');    //5
    theGraph.addVertex('f');    // 6
    theGraph.addVertex('g');    // 7

    theGraph.addEdge(0, 1);     // SA
    theGraph.addEdge(0, 3);     // AB
    theGraph.addEdge(0, 6);     // AD
    theGraph.addEdge(1, 2);     // ab
    theGraph.addEdge(1, 4);     // BC
    theGraph.addEdge(2, 5);     // DE
    theGraph.addEdge(4, 3);     // AB
    theGraph.addEdge(5, 4);     // BC
    theGraph.addEdge(7, 5);     // AD
    theGraph.addEdge(6, 3);     // DE
    theGraph.addEdge(6, 4);     // DE
    theGraph.addEdge(6, 7);     // DE
    theGraph.addEdge(7, 4);     // DE
    System.out.println("Visits: ");
    theGraph.dfs();             // depth-first search
    System.out.println("*********************\nReady: \n*********************\n");
                    for(int i = 0; i<=7;++i)
                              theGraph.displayVertex(i);
            System.out.println("*****print path*****");
           for(int i = 1 ;i<=7;++i)
                     {theGraph.printPath(0,i);
                        System.out.println();
                    }

    }  // end main()
  }  // end class DFSApp
```

```
Visits:
**************vertex =  s
color =  g
path =  s
discover =  1
```

```
finish =  0
**************vertex =  a
color =  g
path =  s
discover =  2
finish =  0
**************vertex =  b
color =  g
path =  a
discover =  3
finish =  0
**************vertex =  e
color =  g
path =  b
discover =  4
finish =  0
**************vertex =  d
color =  g
path =  e
discover =  5
finish =  0
**************vertex =  c
color =  g
path =  d
discover =  6
finish =  0
pop
**************vertex =  c
color =  g
path =  d
discover =  6
finish =  7
pop
**************vertex =  d
color =  g
path =  e
discover =  5
finish =  8
pop
**************vertex =  e
color =  g
path =  b
discover =  4
finish =  9
pop
**************vertex =  b
color =  g
path =  a
discover =  3
```

```
finish =  10
pop
**************vertex =  a
color =  g
path =  s
discover =  2
finish =  11
**************vertex =  f
color =  g
path =  s
discover =  12
finish =  0
**************vertex =  g
color =  g
path =  f
discover =  13
finish =  0
pop
**************vertex =  g
color =  g
path =  f
discover =  13
finish =  14
pop
**************vertex =  f
color =  g
path =  s
discover =  12
finish =  15
pop
**************vertex =  s
color =  g
path =  s
discover =  1
finish =  16
Ready: **************vertex =  s
color =  g
path =  s
discover =  1
finish =  16
**************vertex =  a
color =  g
path =  s
discover =  2
finish =  11
**************vertex =  b
color =  g
path =  a
discover =  3
```

```
finish =  10
**************vertex =  c
color =  g
path =  d
discover =  6
finish =  7
**************vertex =  d
color =  g
path =  e
discover =  5
finish =  8
**************vertex =  e
color =  g
path =  b
discover =  4
finish =  9
**************vertex =  f
color =  g
path =  s
discover =  12
finish =  15
**************vertex =  g
color =  g
path =  f
discover =  13
finish =  14
*****print path*****
s    a
s    a    b
s    a    b    e    d    c
s    a    b    e    d
s    a    b    e
s    f
s    f    g
```

Complexity สำหรับ method dfs() แต่ละโหนดจะเรียก dfsVisit ดังนั้นใช้เวลา = O(V) สำหรับ method dfsVisit() แต่ละโหนด u จะเรียกหากิ่งที่เชื่อม $v \in$ adj[u] ใช้เวลา = O(E) คำนวณเวลารวมในการสืบค้นกราฟในแนวดิ่ง คือ O(V+E)

8.3 การเรียงตามลำดับเหตุการณ์ (Topological Sort)

ในกรณีที่เป็นกราฟที่มีทิศทางและไม่มีวง (directed acyclic graph) เราสามารถนำโหนดของกราฟดังกล่าวมาเรียงลำดับในแนวนอน ซึ่งเรียงตามลำดับเหตุการณ์ที่ต้องทำก่อนหลัง

การเรียงตามลำดับเหตุการณ์มักนำไปประยุกต์ใช้ในงานต่าง ๆ ที่สามารถเขียนความสัมพันธ์ออกมาในรูปของกราฟ และการเรียงลำดับโหนดจะแสดงถึงลำดับก่อนหลังของ

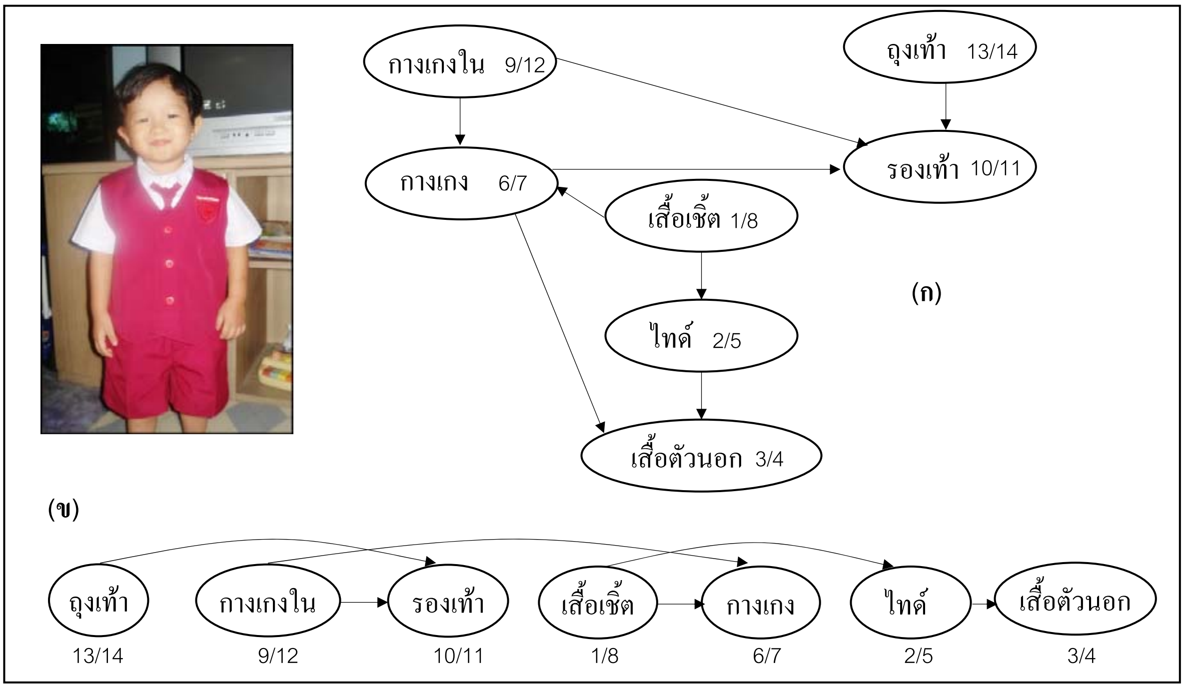
เหตุการณ์ ซึ่งช่วยให้ผู้ใช้ทราบว่าควรลงมือทำงานใดก่อน และมีงานใดจะต้องเสร็จเรียบร้อยก่อนจึงจะเริ่มงานบางงานได้ หรืองานใดสามารถทำพร้อมกับงานใดได้บ้าง การเรียงตามลำดับเหตุการณ์สามารถทำได้โดยใช้การสืบค้นในแนวดิ่ง

ขั้นตอนวิธีของการเรียงตามลำดับเหตุการณ์

Topological-Sort (G)

1. ทำการสืบค้นในแนวดิ่ง เพื่อหาค่า ลำดับเวลาที่ออกจากโหนด(finish)
2. เรียงลำดับโหนด ตามลำดับเวลาที่ออกจากโหนด โดยวางต่อข้างหน้า link list

<u>ตัวอย่างที่ 8.3</u> แต่งตัวให้เด็กชายหมูหยองไปโรงเรียน โดยต้องใส่ กางเกงในก่อนใส่กางเกงนักเรียน สวมถุงเท้าก่อนใส่รองเท้า ส่วนจะใส่ถุงเท้าก่อนหรือหลังใส่กางเกงก็ได้



<u>รูปที่ 8.7</u> (ก) กราฟที่มีทิศทางและไม่มีวงที่ได้จากการสืบค้นในแนวดิ่ง
(ข) ลำดับเหตุการณ์ โดยเรียงตามเวลาที่ออกจากโหนด (f) จากมากไปหาน้อย

Complexity : O(V + E) ตามที่ใช้ในการทำสืบค้นในแนวดิ่ง