

Data Structure and Algorithm

Graph

❖ กราฟ

❖ Adjacency matrix/ list/edge list

❖ Depth First Search

❖ Breadth First Search

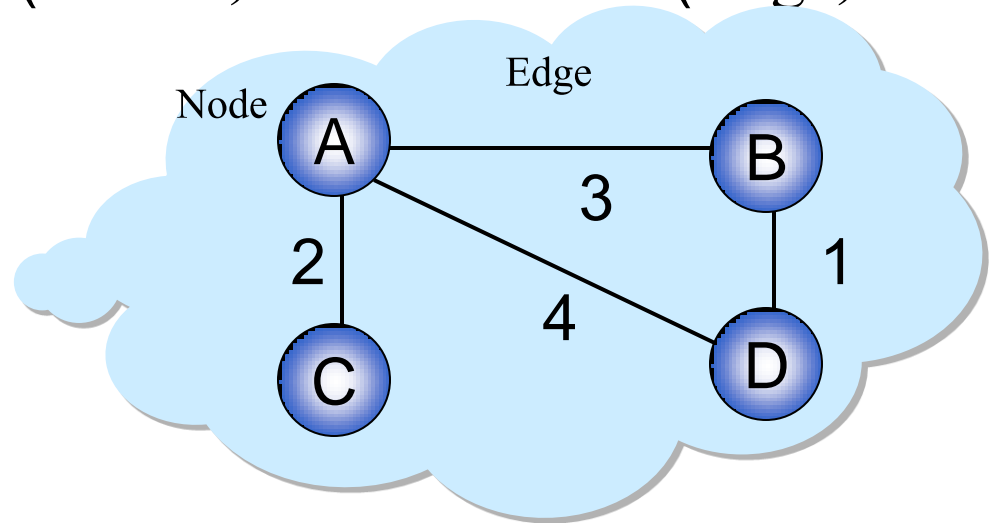
❖ Topological Sort



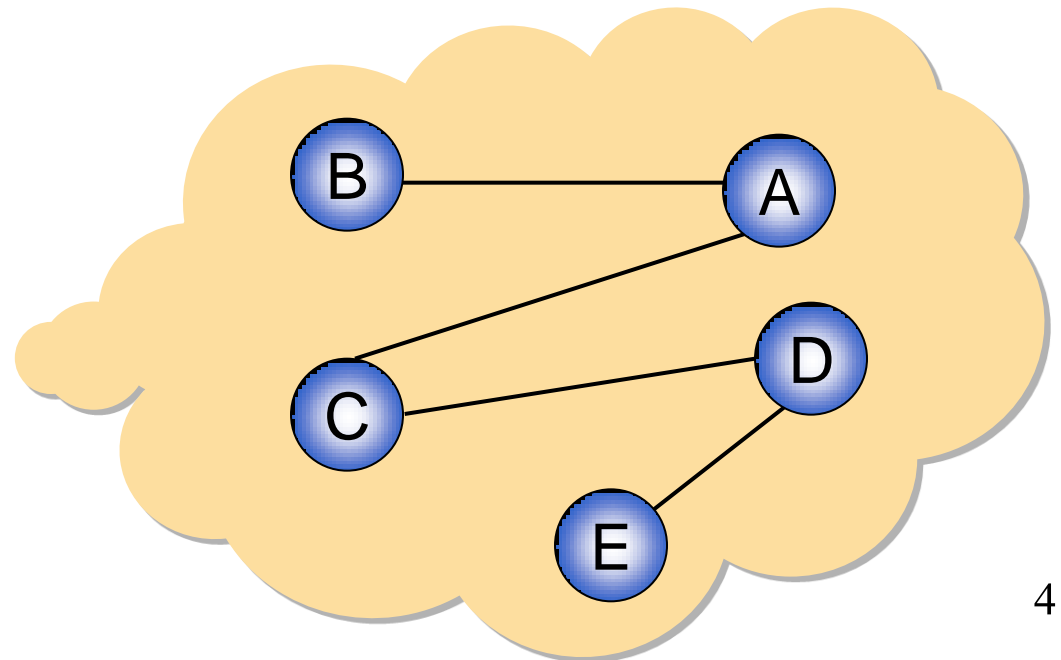
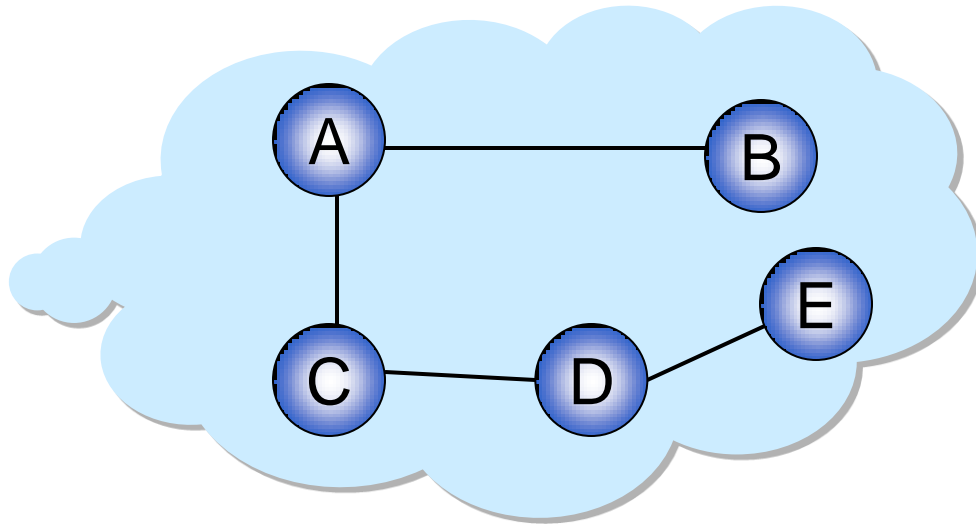
นิยามกราฟ

- ❖ กราฟ คือเซตของโหนด(Vertex) และเส้นเชื่อม (Edge)

$$G = (V, E)$$



- ❖ โหนด แสดงถึง Object เช่น ชื่อเมือง, สถานที่ท่องเที่ยว
- ❖ เส้นเชื่อม (Edge) แสดงความสัมพันธ์ของ 2 โหนด
มีความหมายแล้วแต่การนิยาม เช่น ระยะทาง, เวลา

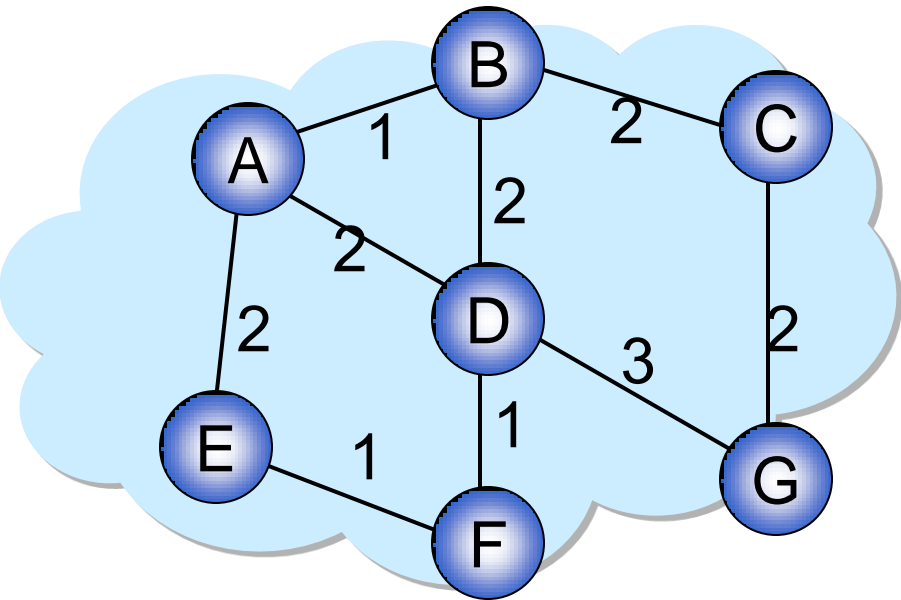
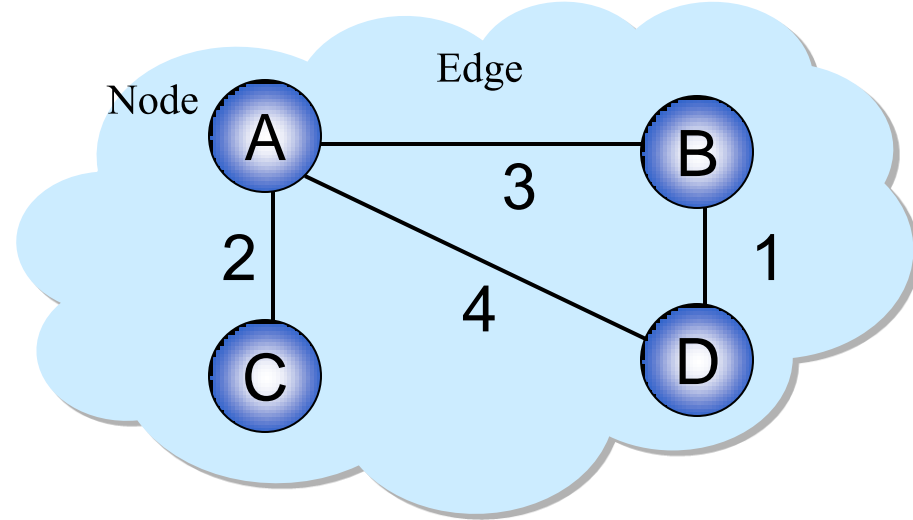


นิยามกราฟ

$$G = (V, E)$$

$$\text{❖ } V = \{A, B, C, D\}$$

$$\text{❖ } E = \{(A, B, 3), (A, D, 4), (A, C, 2), (B, D, 1)\}$$



จงแสดง **V** และ **E** ของกราฟนี้

V(G) =

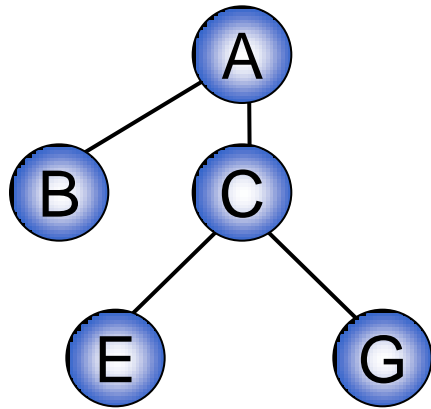
E(G) =

Graph VS. Tree

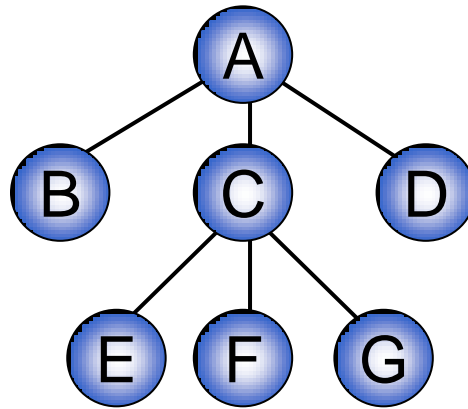
❖ กราฟเป็น Super Set ของต้นไม้

❖ Tree ต้องมี parent Node เพียงโหนดเดียว, แต่ Graph ไม่จำเป็น

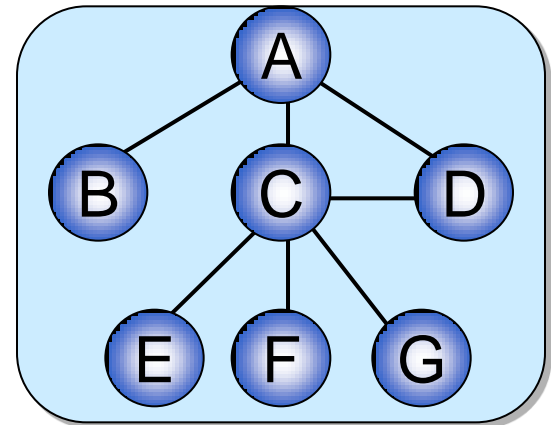
❖ บางโหนดอาจไม่มีเส้นเชื่อมได้ เช่น บางเมืองไม่มีสายการบิน



(1) Binary tree

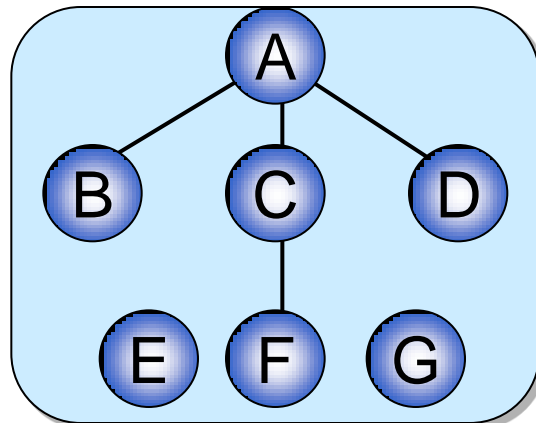


(2) Non binary tree



(3) Graph

โหนด A,C,D มี parent โหนดมากกว่า 1

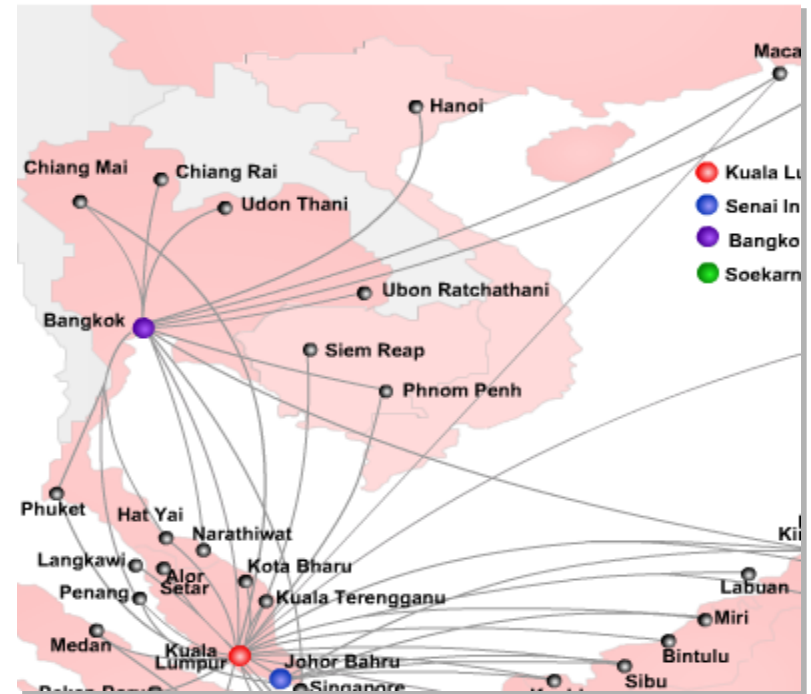


(4) Graph

โหนด E,G ไม่มีเส้นเชื่อม

ประโยชน์ของกราฟ (Routing การหาเส้นทาง)

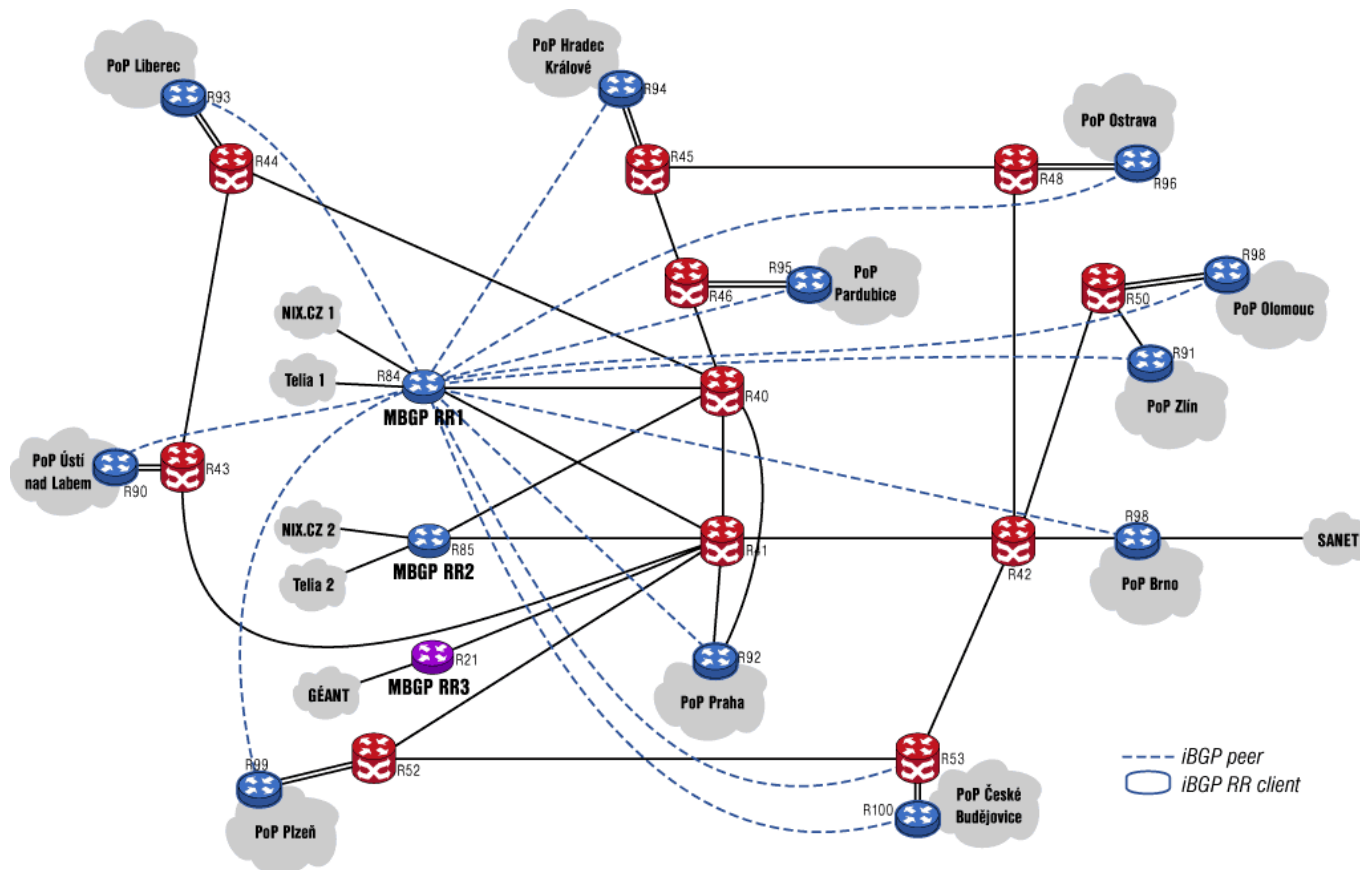
สายการบิน (การเชื่อมต่อของสายการบิน ตารางบิน)



ประโยชน์ของกราฟ (Routing การหาเส้นทาง)

❖ Network (การเชื่อมต่อของอุปกรณ์ Router)

❖ เพื่อใช้ในการรับส่งข้อมูลในเครือข่าย



ประโยชน์ของกราฟ (Algorithm Design)

- ❖ Map Coloring คือวิธีการระบายสีในแผนที่โดยใช้สีน้อยที่สุด
- ❖ พื้นที่ติดกันห้ามใช้สีเดียวกัน



ตัวอย่างอื่น

Graph

transportation

communication

World Wide Web

social

food web

software systems

scheduling

circuits

Nodes

street intersection

computers

web pages

people

species

functions

tasks

gates

Edges

highways

fiber optic cables

hyperlinks

relationships

predator-prey

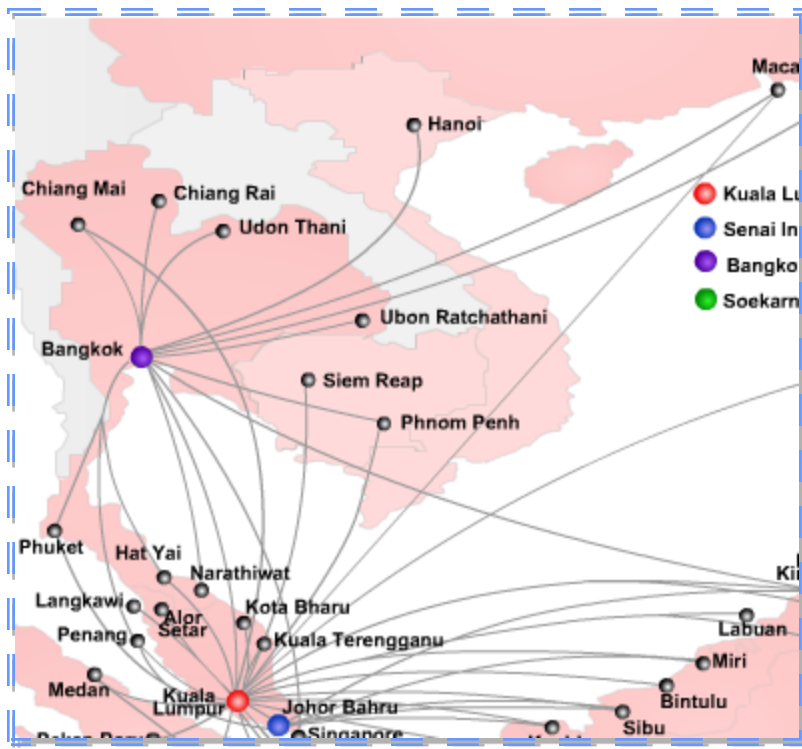
function calls

precedence constraints

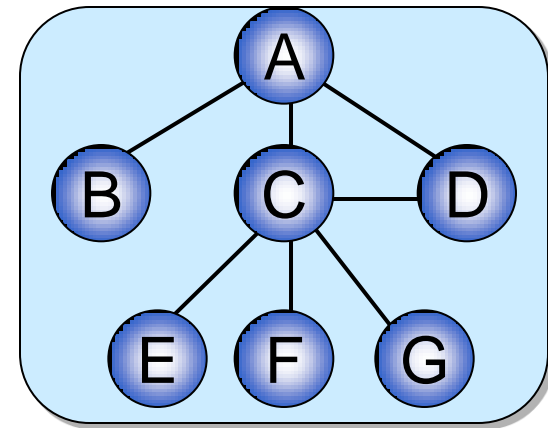
wires

Directed & Undirected Graph

- ❖ Undirected Graph คือกราฟที่เส้นเชื่อม**ไม่มี**ลูกศรกำกับทิศทาง
- ❖ หมายถึงความสัมพันธ์ของ 2 โหนดแบบ**ไปและกลับ**



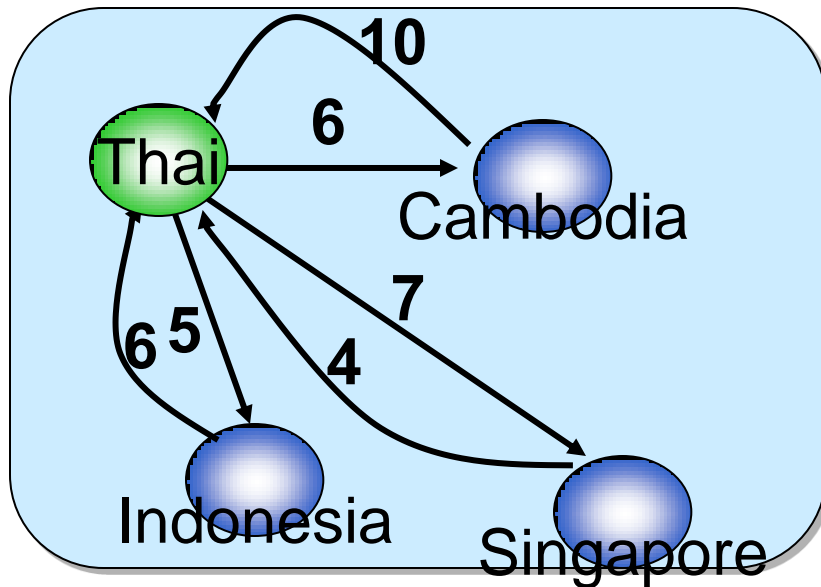
Undirected Graph แสดงสายการบินของ Air Asia



Undirected Graph แสดงโหนด
และเส้นเชื่อมของกราฟรูปหนึ่ง

Directed & Undirected Graph

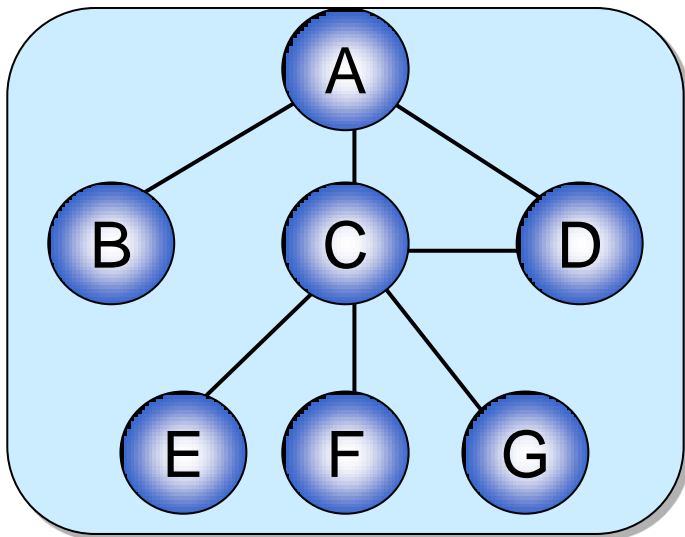
- ❖ Directed Graph คือกราฟที่เส้นเชื่อมมีลูกศรกำกับทิศทาง
- ❖ เช่น Edge แสดงค่าโดยสารที่มีราคา ไป-กลับไม่เท่ากัน
- ❖ หรือ ค่าโทรศัพท์ที่ไทยไปสิงคโปร์ แพงกว่าสิงคโปร์โทรหาไทย



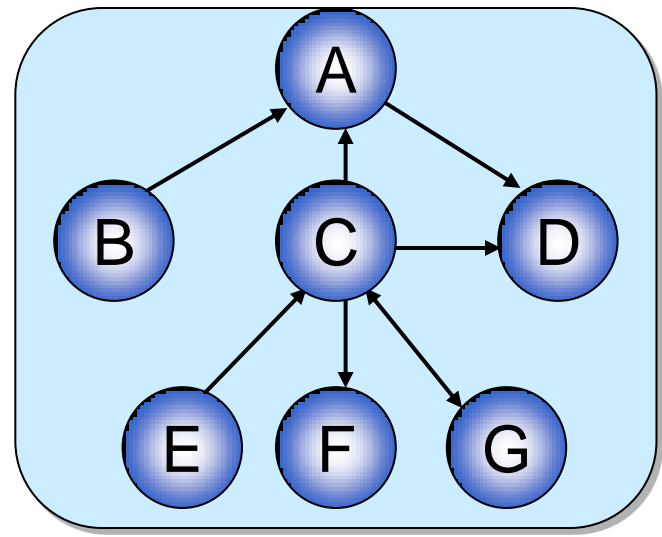
Undirected Graph แสดงค่า
อัตราค่าโทรศัพท์ระหว่างประเทศ
(เป็นราคาสมมติเท่านั้น)

Unweighted Graph (กราฟไม่มีน้ำหนัก)

- ❖ ไม่ระบุข้อมูลหรือค่าบางอย่าง (แตกต่างจากสิ่งอื่นๆ)
 - เช่น ถนนที่เชื่อมเมือง **2** เมืองแต่ไม่ระบุระยะทาง
 - ผังรถไฟฟ้าใต้ดิน แต่ไม่ระบุราคาค่าโดยสารระหว่างสถานี
 - หรือมองว่าค่าข้อมูลเหล่านั้นมี**ค่าเท่ากันหมด**
- ❖ อาจเป็น Directed หรือ Undirected Graph ก็ได้



Unweighted & Undirected Graph

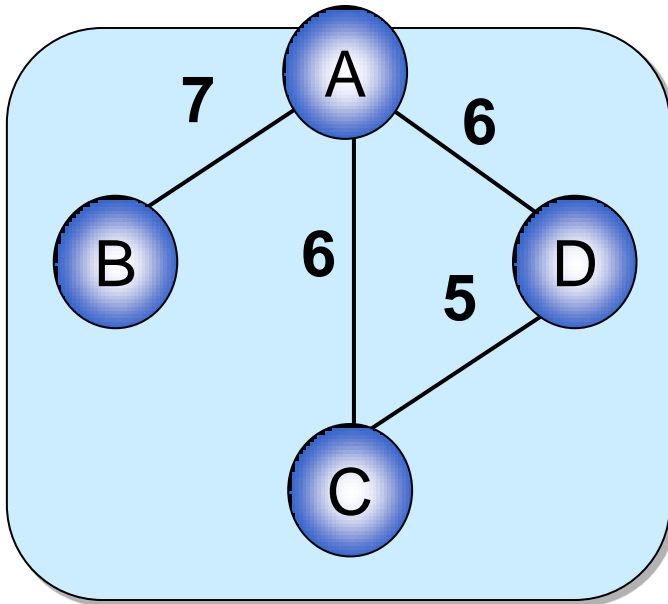


Unweighted & directed Graph

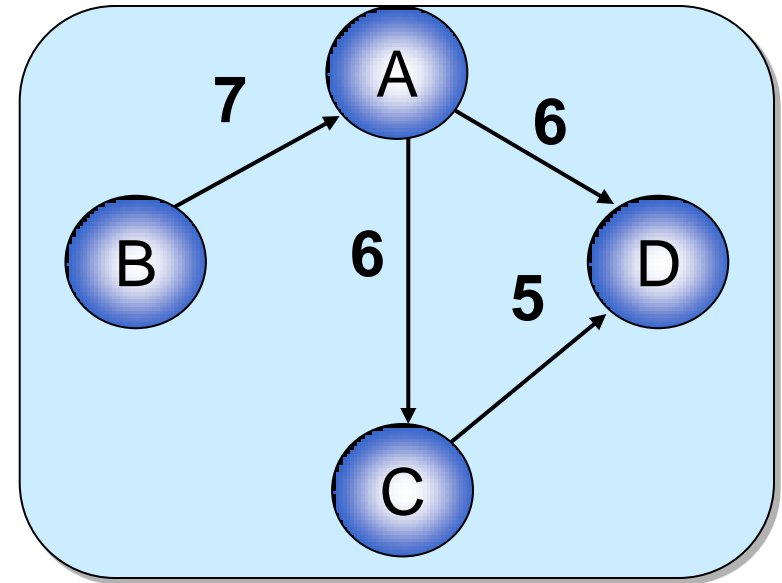
Weighted Graph (กราฟมีน้ำหนัก)

❖ เส้นเชื่อมระบุข้อมูลหรือค่าบางอย่างที่ต้องการบ่งชี้

- เช่น ถนนที่เชื่อมเมือง **2** เมืองพร้อมระยะระยะทางระหว่างเมือง
- อาจเป็น Directed หรือ Undirected Graph ก็ได้



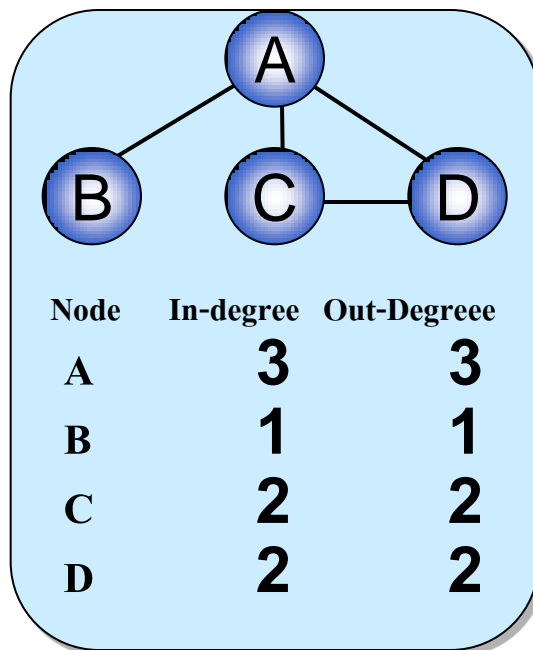
Weighted & Undirected Graph



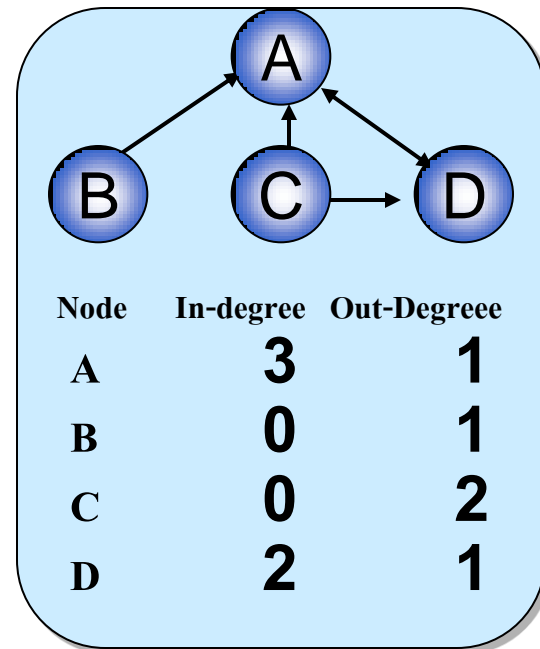
Weighted & directed Graph

อินดีกรีและเอาท์ดีกรี

- ❖ แต่ละโหนดจะมี**จำนวนเส้น**เชื่อมระหว่างโหนดไม่เท่ากัน
- ❖ In-degree แสดงจำนวนเส้นเชื่อมที่**เข้า**มายังโหนดนั้นๆ
- ❖ Out-degree แสดงจำนวนเส้นเชื่อมที่**ออก**จากโหนดนั้นๆ
- ❖ ใน Undirected Graph จำนวน In-degree และ Out-degree จะ**เท่ากัน**



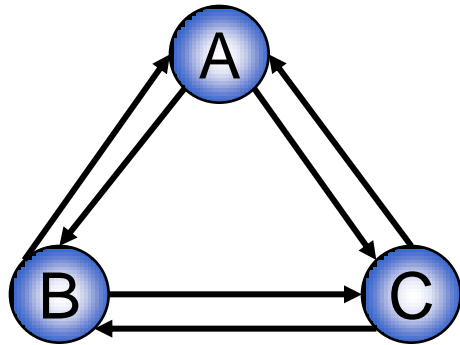
Undirected Graph



Directed Graph

Complete Graph (กราฟสมบูรณ์)

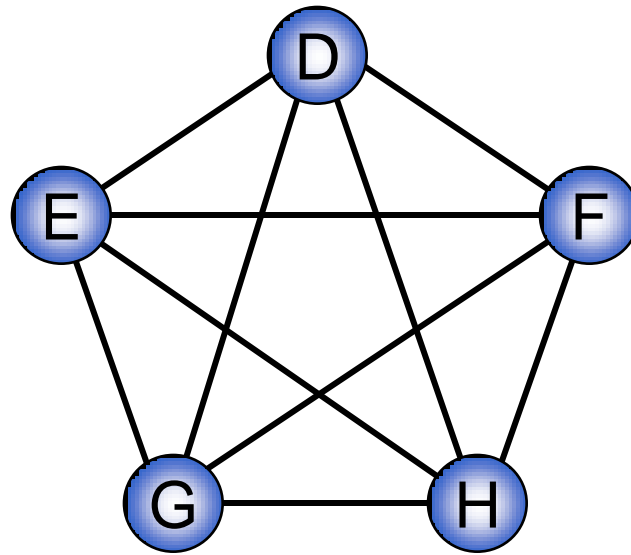
❖ กราฟที่ทุกโหนดมีเส้นเชื่อมถึงโหนดอื่นๆ ทั้งหมด



กราฟมีทิศทาง

$$\text{จำนวน Edge} = N * (N-1)$$

$$\text{เช่น } 3 * (3-1) = 6$$



กราฟไม่มีทิศทาง

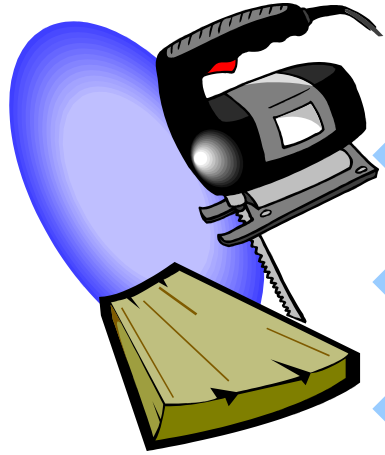
$$\text{จำนวน Edge} = \frac{N * (N-1)}{2}$$

$$\text{เช่น } 5 * (5-1) / 2 = 10$$

List of important graph terminology

Vertices/Nodes	Edges	Set V ; size $ V $
Un/Weighted	Un/Directed	Sparse
Path	Cycle	Isolated
Self-Loop	Multiple Edges	Multigraph
DAG	Tree/Forest	Eulerian
Set E ; size $ E $	Graph $G(V, E)$	
Dense	In/Out Degree	
Reachable	Connected	
Simple Graph	Sub-Graph	
Bipartite	Complete	

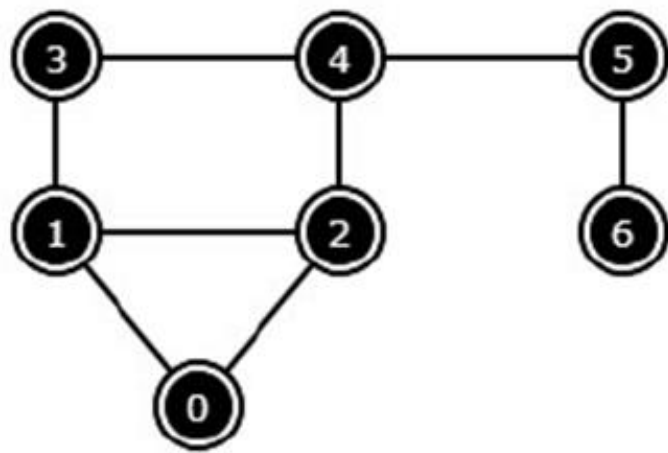
Graph Representation



❖ *Adjacency Matrix*

❖ *Adjacency List*

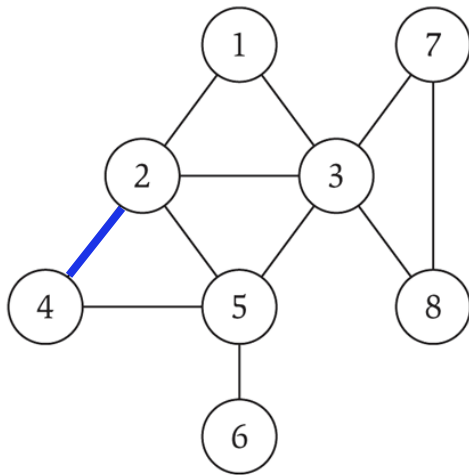
❖ *Edge List*



Adjacency Matrix	Adjacency List	Edge List
0 1 2 3 4 5 6 0 0 2 5 0 0 0 0 1 2 0 7 1 0 0 0 2 5 7 0 0 4 0 0 3 0 1 0 0 3 0 0 4 0 0 4 3 0 9 0 5 0 0 0 0 9 0 8 6 0 0 0 0 0 8 0	0: 1 2 1: 0 2 3 2: 0 1 4 3: 1 4 4: 2 3 5 5: 4 6 6: 5	0: 0 1 1: 0 2 2: 1 0 3: 1 2 4: 1 3 5: 2 0 6: 2 1 7: 2 4 8: 3 1 9: 3 4 10: 4 2 11: 4 3

Adjacency Matrix (Undirected Graphs)

- ❖ Adjacency matrix. n -by- n matrix with $A_{uv} = 1$ if (u, v) is an edge.
 - Two representations of each edge.
 - Space proportional to n^2 .
 - Checking if (u, v) is an edge takes $\Theta(1)$ time.
 - Identifying all edges takes $\Theta(n^2)$ time.

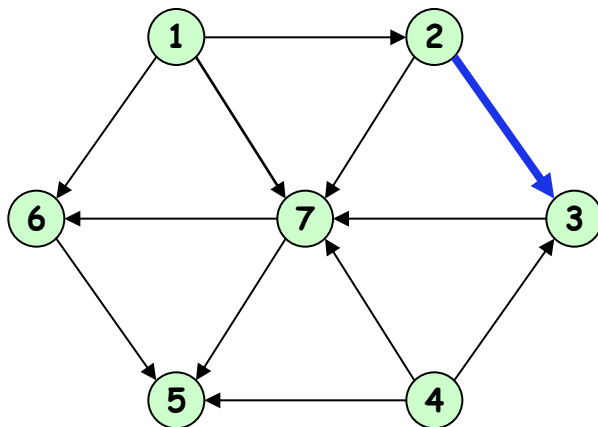


	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Graph Representation: Adjacency Matrix (Directed Graphs)

Adjacency matrix. n -by- n matrix with $A_{uv} = 1$ if (u, v) is an edge.

- One representations of each edge.
- Space proportional to n^2 .
- Checking if (u, v) is an edge takes $\Theta(1)$ time.
- Identifying all edges takes $\Theta(n^2)$ time.



	1	2	3	4	5	6	7
1	0	1	0	0	0	1	1
2	0	0	1	0	0	0	1
3	0	0	0	0	0	0	1
4	0	0	1	0	1	0	1
5	0	0	0	0	0	0	0
6	0	0	0	0	1	0	0
7	0	0	0	0	1	1	0

Adjacency Matrix

```
#define MaxNodes 50
struct node {
    int info;
};
struct edge {
    int adj;
};
struct graph {
    struct node  nodes[MaxNodes];
    struct edge  edges[MaxNodes][MaxNodes];
};
struct graph g;
```

Graph Representation: Adjacency List

Adjacency list. Node indexed array of lists.

- Two representations of each edge.

degree = number of neighbors of v

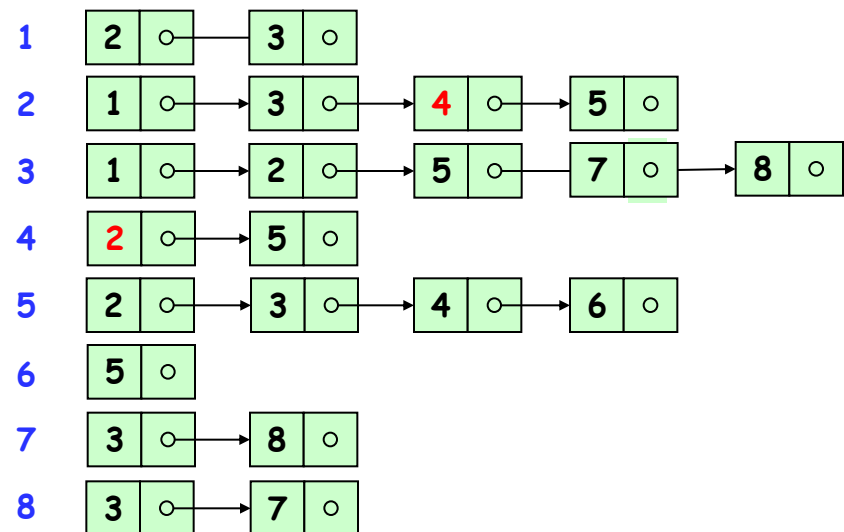
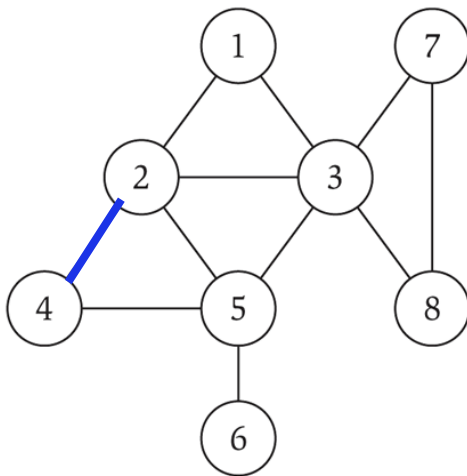
- Let degree n_v be the numbers of incident edges of v

- Then $O(\sum_{v \in V} n_v) = 2m$

- Space takes $\Theta(m + n)$.

- Checking if (u, v) is an edge takes $O(\deg(u))$ time.

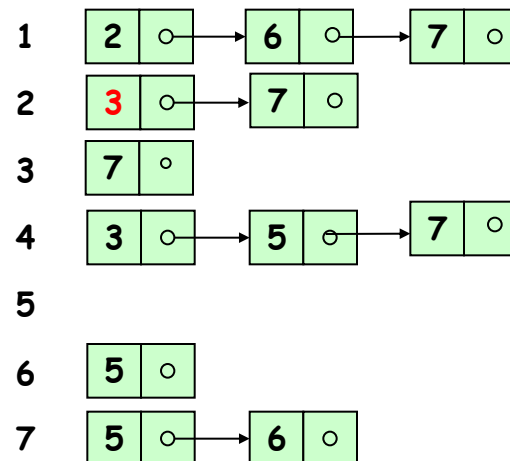
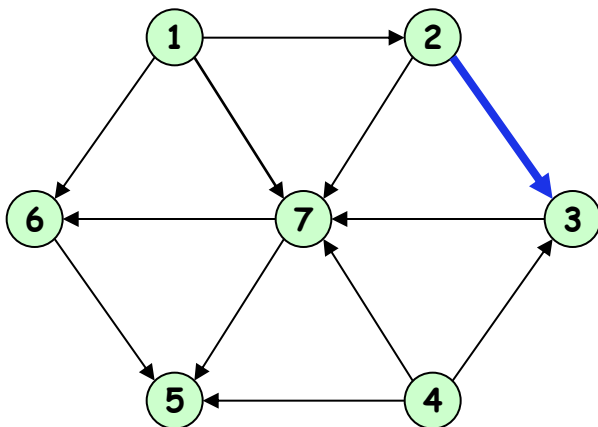
- Identifying all edges takes $\Theta(m + n)$ time.



Graph Representation: Adjacency List

Adjacency list. Node indexed array of lists.

- One representations of each edge.
 - Let degree n_v be the numbers of incident edges of v
 - degree = number of neighbors of v
 - Then $O(\sum_{v \in V} n_v) = m$
- Space takes $\Theta(m + n)$.
- Checking if (u, v) is an edge takes $O(\deg(u))$ time.
- Identifying all edges takes $\Theta(m + n)$ time.



Exercise

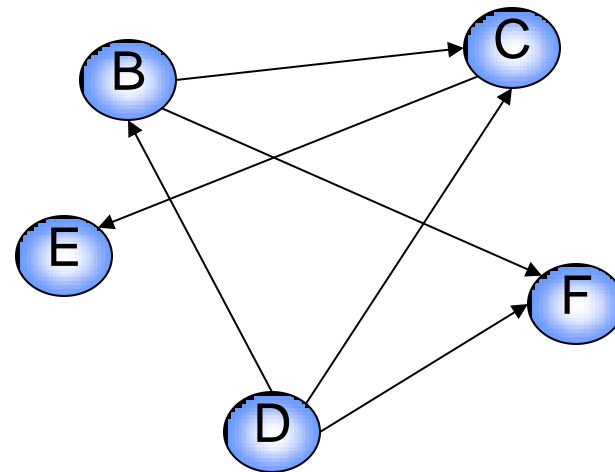
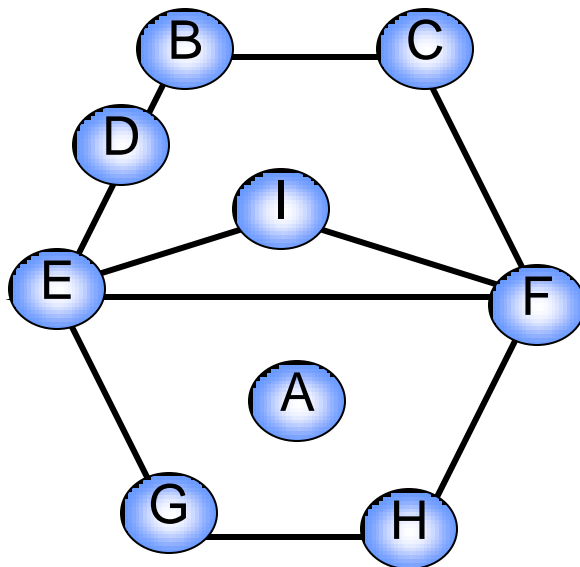
- ❖ ให้เขียนโปรแกรมเพื่อรับ input กราฟแบบต่าง ๆ โดยเป็นกราฟที่มี weight ดังนี้
 - ลักษณะข้อมูล adjacency matrix
ใช้ array
 - ลักษณะข้อมูล adjacency list
ใช้ vector
 - ลักษณะข้อมูล edge list
ใช้ queue
- ❖ ให้พิมพ์ข้อมูลทั้งหมด อยู่ในรูป matrix
- ❖ ให้ download input จาก facebook group

ข้อมูล file

- ❖ บรรทัดแรก เป็นจำนวน node
- ❖ บรรทัดถัดมาเป็นข้อมูล node ที่ 1 คือ แถวแรกของ adjacency matrix (แบบมี weight)
- ❖ เมื่อวนรับข้อมูล matrix ทั้งหมด แล้วก็รับข้อมูล adj. list
- ❖ บรรทัดแรกของส่วนนี้ คือ จำนวน node
- ❖ บรรทัดถัดไป ขึ้นต้นด้วย จำนวน neighbor ของ node ที่ 1 ตามด้วย node neighbor และ weight
- ❖ เมื่อวนรับข้อมูล adj. list แล้วให้รับข้อมูล edge list ต่อ
- ❖ บรรทัดแรกของส่วนนี้ คือ จำนวน edge
- ❖ บรรทัดถัดมา ขึ้นต้นด้วย node ต้นทาง node ปลายทาง และ weight
- ❖ วนรับข้อมูล edge list จนหมด

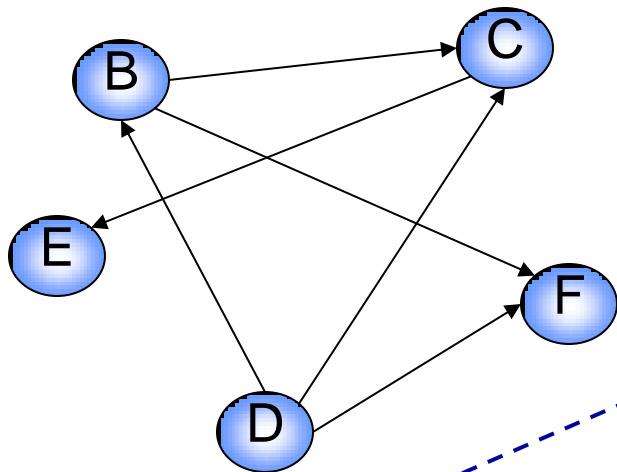
การท่องเที่ยวในกราฟ

- ❖ คือการ**ค้นหาเส้นทาง**จากโหนดหนึ่ง ไปยังโหนดที่ต้องการในกราฟ
- ❖ หากหาเส้นทางได้ไปยังโหนดได้ แสดงว่าโหนดเริ่มต้นสามารถเชื่อมต่อกับโหนดนั้นได้ เช่นหาเส้นทางการบินจากกรุงเทพ ไปยัง Dallas อเมริกา
- ❖ ในกราฟ อาจมีบางโหนดที่ไม่สามารถเชื่อมกันก็เป็นได้
- ❖ 2 วิธี **Breadth-first Search, Depth-first Search**



โหนด F สามารถเดินทางไปยัง B ได้หรือไม่?

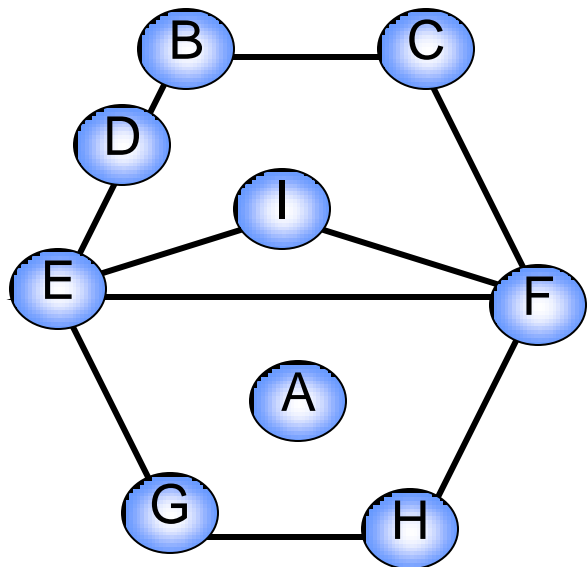
อาจมีบางโหนดที่ไม่สามารถเชื่อมกันก็เป็นได้



❖ ใน Directed Graph

❖ หากเริ่มจาก B : $B \rightarrow C$, $B \rightarrow F$, $B \rightarrow C \rightarrow E$ แต่ไม่สามารถไปถึง D ได้

❖ หากเริ่มจาก F : ไม่สามารถเชื่อมกับโหนดอื่นๆ ได้เลย



❖ ใน Undirected Graph โหนดที่เชื่อมกันสามารถเข้าถึงกันได้หมด

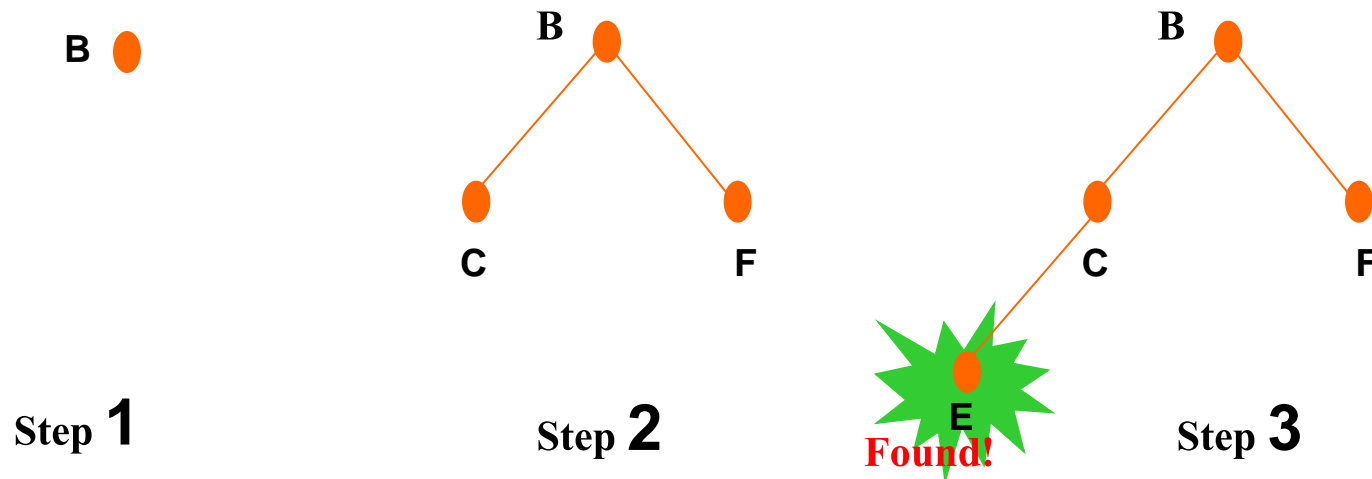
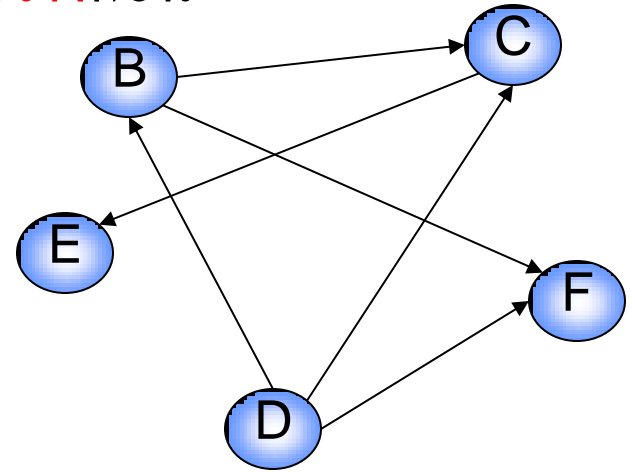
❖ หากเริ่มจาก B : สามารถเข้าถึงได้ทุกโหนด ยกเว้น A

❖ หากเริ่มจาก A : ไม่สามารถเชื่อมกับโหนดอื่นๆ ได้เลย เพราะ A ไม่เชื่อมต่อกับใครเลย

โหนด F สามารถเดินทางไปยัง B ได้หรือไม่?

Breadth-first Search

- ❖ คือการค้นหาโหนดใดในกราฟ โดยดูใน**แนวกว้าง**ก่อน
- ❖ ใช้ **Queue** เป็นเครื่องมือในการช่วยค้นหา
- ❖ ตัวอย่างการหาโหนด E เริ่มจากโหนด B



Implementing BFS using Queue

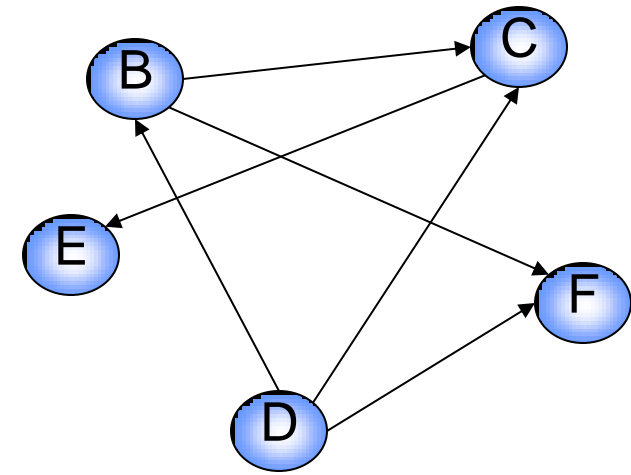


BFS(s)

```
set Discovered[s] = true and set Discovered[v] = false for all other v
init a queue L to consist of s only
set the current BFS tree T =  $\emptyset$ 
set d[s] = 0
while L is not empty
    u = L.dequeue()
    for each edge (u, v) incident to u
        if Discovered[v] is false then
            set Discovered[v] = true
            add edge (u, v) to the tree T
            L.enqueue(v)
            set d[v] = d[u] + 1
        end if
    end for
end while
```

Breadth-first Search (ตัวอย่าง)

เริ่มจาก B ต้องการค้นหา E



Step

Node

Queue

Step 1: Queue = {B}

Queue = {B}

Step 2: หยิบโหนด B ออกมาสร้างโหนด

B ●

Queue = {ว่าง}

Step 3: สำหรับโหนด C ที่เชื่อมอยู่กับโหนด B

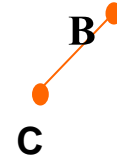
B ●
|
C ●

Queue = {ว่าง}

Breadth-first Search (ตัวอย่าง)

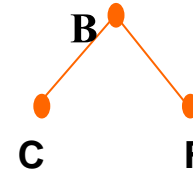
<i><u>Step</u></i>	<i><u>Node</u></i>	<i><u>Queue</u></i>
--------------------	--------------------	---------------------

Step 5: C ไม่ใช่ endVertex จับใส่ Queue



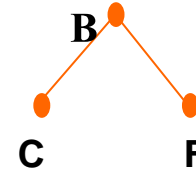
Queue={C}

Step 6: สำหรับโหนด F ที่เชื่อมอยู่กับโหนด B



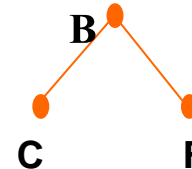
Queue={C}

Step 7: F ไม่ใช่ endVertex จับใส่ Queue



Queue={C,F}

Step 8: หยิบโหนด C ออกมาสร้างโหนด



Queue={F}

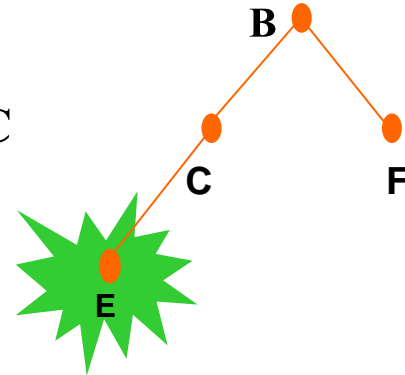
Breadth-first Search (ตัวอย่าง)

<u>Step</u>	<u>Node</u>	<u>Queue</u>
-------------	-------------	--------------

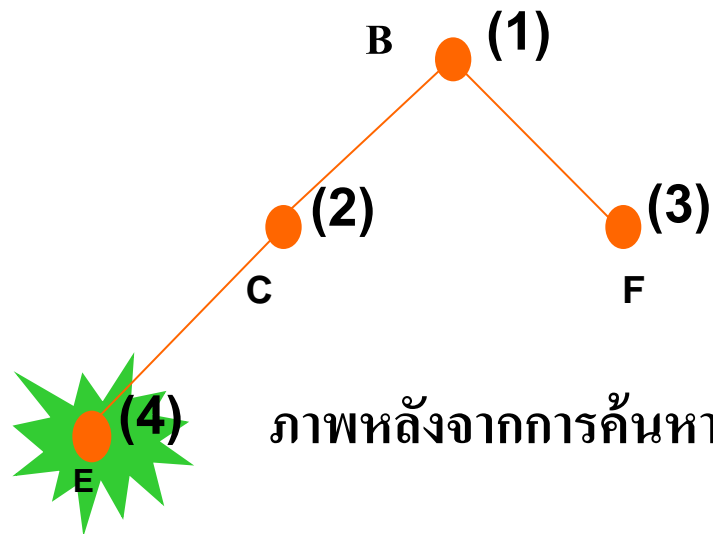
Step 9: สำหรับโหนด E ที่เชื่อมอยู่กับโหนด C

Node E คือ endVertex

ดังนั้นให้คืนค่าและจบการค้นหา



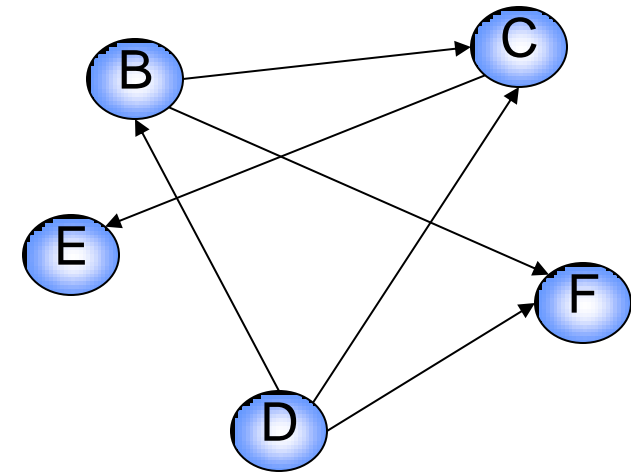
Queue = {F}



ภาพหลังจากการค้นหา

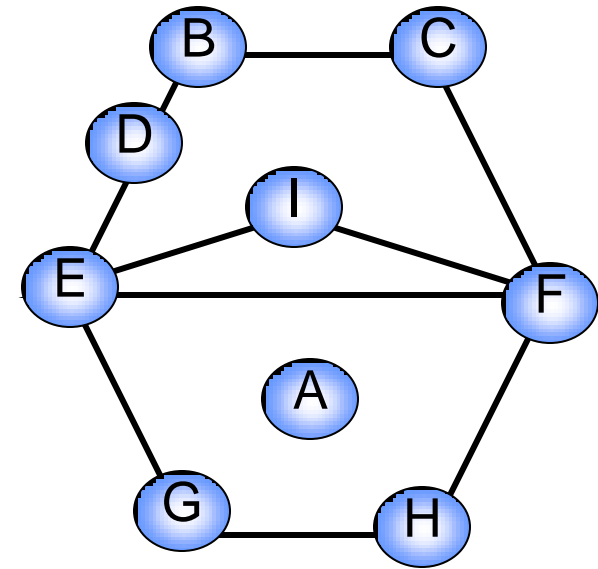
Breadth-first Search (แบบฝึกหัด)

จากรูปขวามือให้สร้างภาพขั้นตอนการค้นหาจากโหนด D ไปยัง
โหนด E โดยใช้ Breadth-first Search



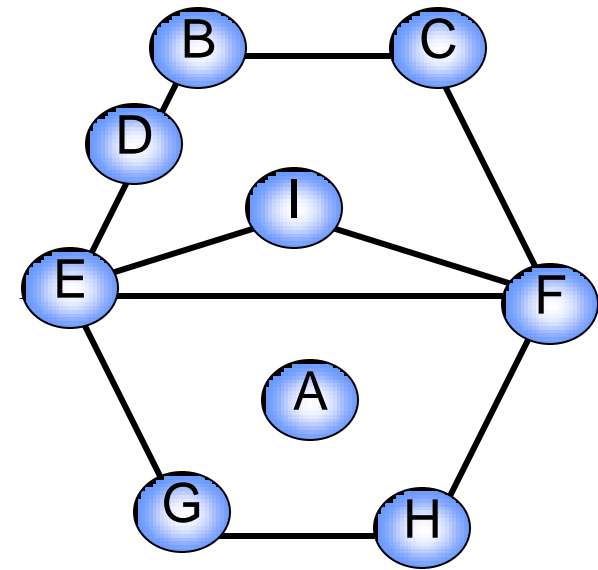
Breadth-first Search (แบบฝึกหัด)

จากรูปขวามือให้สร้างภาพขั้นตอนการค้นหาจากโหนด C ไปยัง
โหนด G โดยใช้ Breadth-first Search



Breadth-first Search (แบบฝึกหัด)

จากรูปขวามือให้สร้างภาพขั้นตอนการค้นหาจากโหนด I ไปยัง
โหนด B โดยใช้ Breadth-first Search

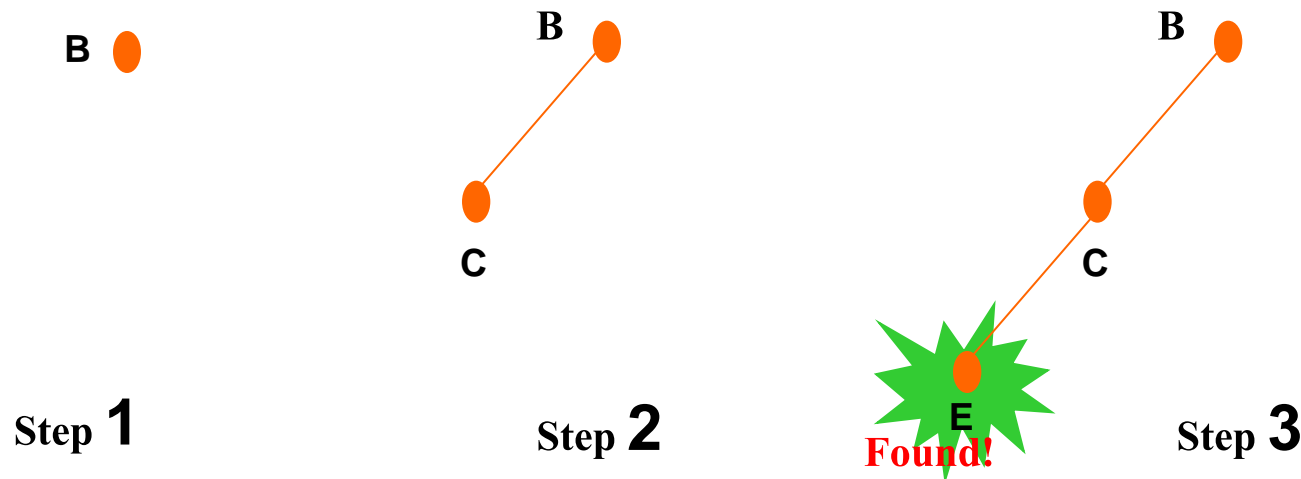
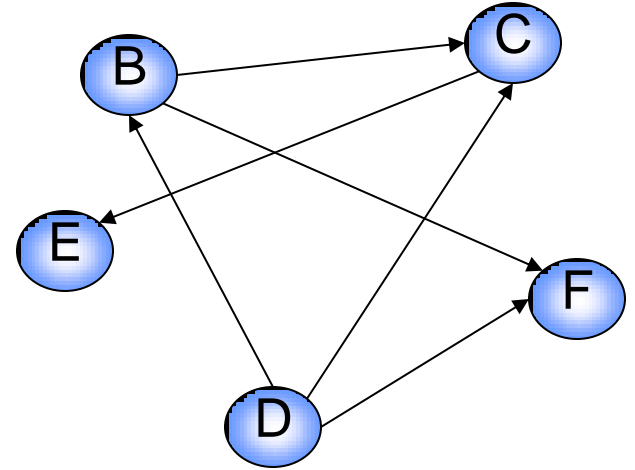


Exercise

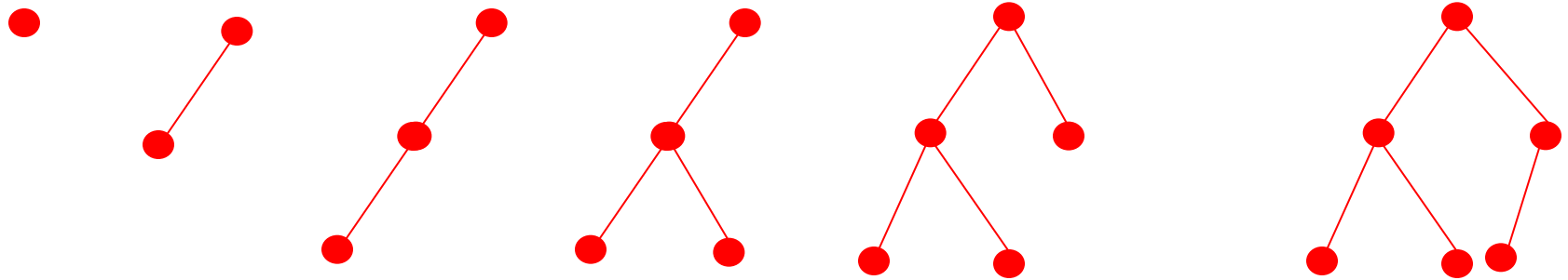
- ❖ อ่าน input file ที่รับข้อมูลกราฟไปเพื่อหา BFS จาก node 5 ไปยัง node ต่าง ๆ
- ❖ Input file
บรรทัดแรก คือ จำนวน โหนด จำนวน edge
บรรทัดถัดไปเป็น node เริ่มต้น node ปลายทางของ edge
ทั้งหมด
- ❖ กำหนดให้ใช้ adjacency list

Depth-First Search

- ❖ คือการค้นหาโหนดใดในกราฟ โดยดูใน**แนวลึก**ก่อน
- ❖ ใช้ **Stack** เป็นเครื่องมือในการช่วยค้นหา
- ❖ ตัวอย่างการหาโหนด E เริ่มจากโหนด B



ภาพ Step การค้นหาของ Depth-First Search



Algorithm: Depth-first Search

1. Push(Stack, startVertex) /*กำหนดค่าใน Stack*/
2. ให้ทำจนพบโหนด endVertex หรือ Stack มีค่าว่าง
 - 2.1 $X = \text{Pop}(\text{Stack})$ //หยิบค่าใน Stack ออกมา
 - 2.2 ถ้า $X = \text{endVertex}$ ให้คืนค่าและจบการค้นหา
 - 2.3 ถ้าไม่
 - 2.3.1 เซ็ตสถานะว่า โหนด X ถูกค้นหามาแล้ว
 - 2.3.2 หาทุกโหนดที่เชื่อมต่อกับ X ไว้ใน List
 - 2.3.3 Push(Stack, ทุกโหนด List แบบกลับลำดับ)

Implementing DFS

DFS(s)

init all Explored[i] = false

init a stack S

S.push(s) // add stack S with one element s

while stack S is not empty

u = S.pop() // take node u from top of stack S

if Explored[u] = false

set Explored[u] = true

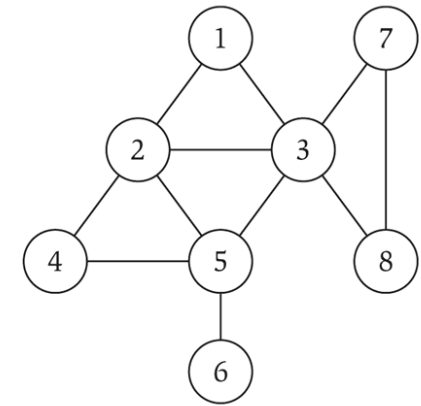
for each edge (u, v) incident to u

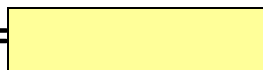
S.push(v) // add v to top of stack S

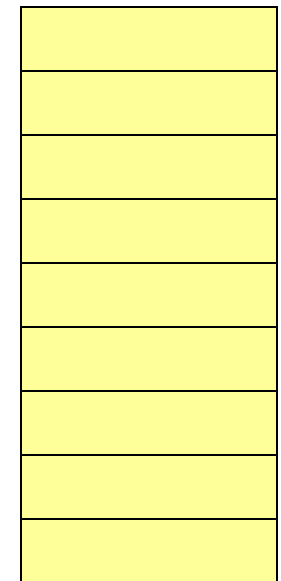
end for

end if

end while



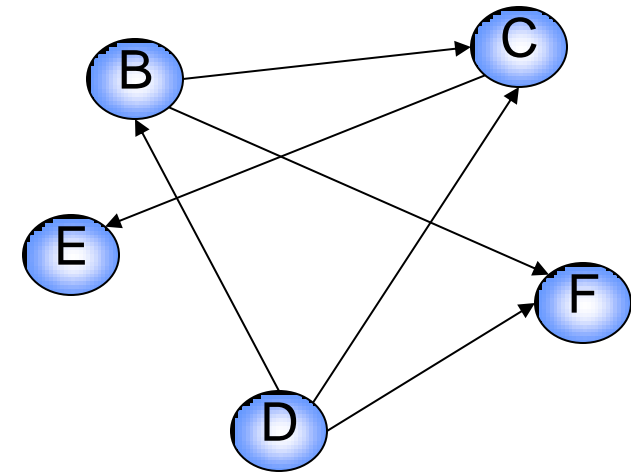
u = 



stack S

Depth-first Search(ตัวอย่าง)

เริ่มจาก B ต้องการค้นหา E



Step

Node

Stack

Step 1: Stack = {B}

Stack = {B}

Step 2: ยังไม่พบโหนด และ Stack ยังไม่ว่าง

Stack = {B}

Step 3: X = Pop(Stack) //ค่าของ X คือ โหนด B

B ●


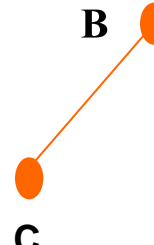
Stack = {ว่าง}

Step 4: X ไม่ใช่ โหนดที่ต้องการค้นหา
ให้กำหนดว่า B เป็นโหนดที่ค้นหาแล้ว

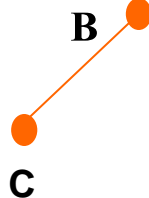
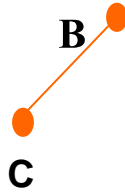
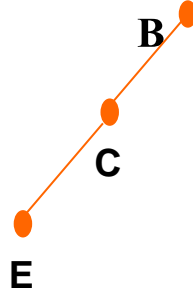
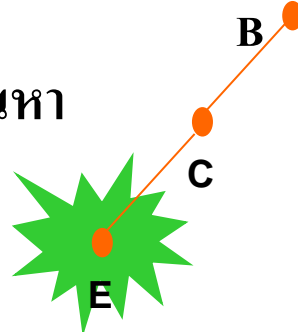
B ●

Stack = {ว่าง}

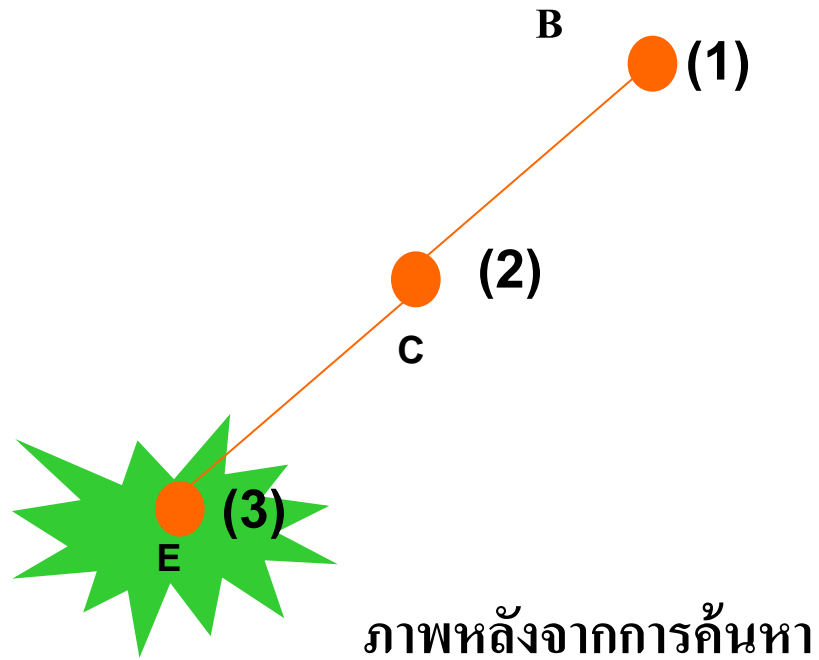
Depth-first Search(ตัวอย่าง)

<u>Step</u>	<u>Node</u>	<u>Stack</u>
Step 5: List = {C, F} //หาทุกโหนดที่ต่อกับ X	B ●	Stack = {ว่าง}
Step 6: Push(Stack, List กลับลำดับ)	B ●	Stack = {F,C}
Step 7: X =Pop(Stack) //ค่าของ X คือ โหนด C	 B ● C ●	Stack = {F}
Step 8: X ไม่ใช่ โหนดที่ต้องการค้นหา ให้กำหนดว่า C เป็นโหนดที่ค้นหามาแล้ว	 B ● C ●	Stack = {F}

Depth-first Search(ตัวอย่าง)

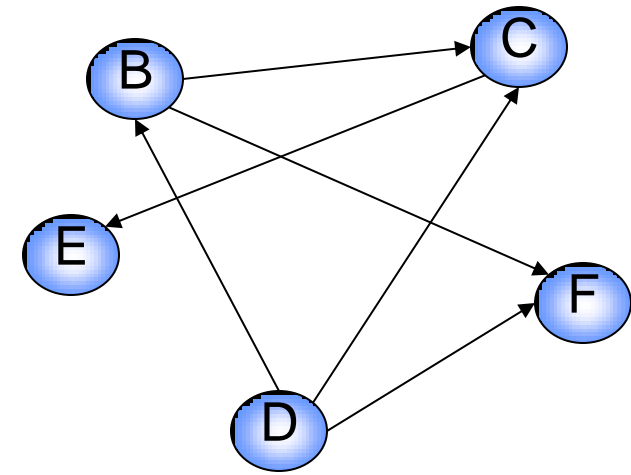
<i>Step</i>	<i>Node</i>	<i>Stack</i>
Step 9: List = {E} //หาทุกโหนดที่ต่อกับ X		Stack = {F}
Step 10: Push(Stack, List กลับลำดับ)		Stack = {F,E}
Step 11: X =Pop(Stack) //ค่าของ X คือโหนด E		Stack = {F}
Step 12: X คือโหนดที่ค้นหา ให้คืนค่าและหยุดการค้นหา		Stack = {F}

Depth-first Search(ตัวอย่าง)



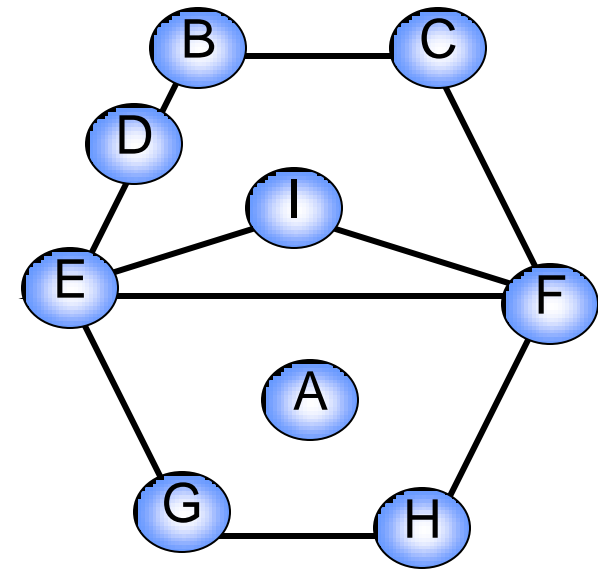
Depth-first Search (แบบฝึกหัด)

จากรูปขวามือให้สร้างภาพขั้นตอนการค้นหาจากโหนด D ไปยัง
โหนด E โดยใช้ Depth-first Search



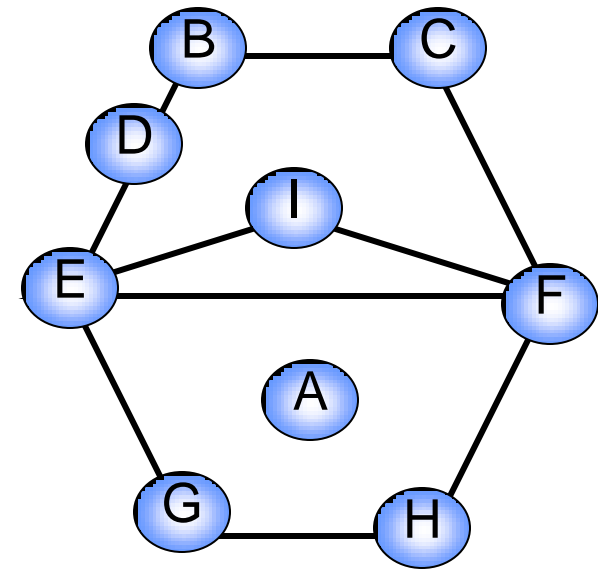
Depth-first Search (แบบฝึกหัด)

จากรูปขวามือให้สร้างภาพขั้นตอนการค้นหาจากโหนด C ไปยัง
โหนด G โดยใช้ Depth-first Search



Depth-first Search (แบบฝึกหัด)

จากรูปขวามือให้สร้างภาพขั้นตอนการค้นหาจากโหนด I ไปยัง
โหนด B โดยใช้ Depth-first Search

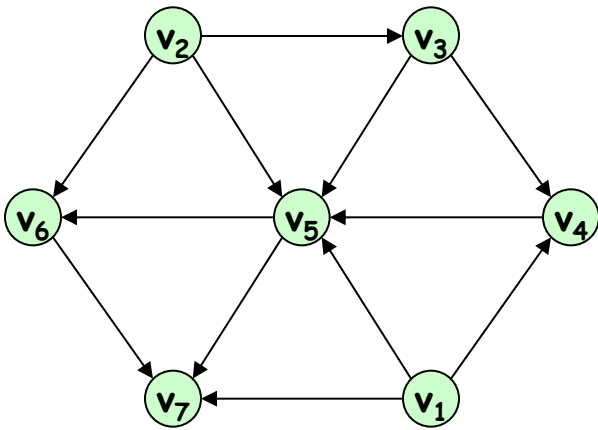


Exercise

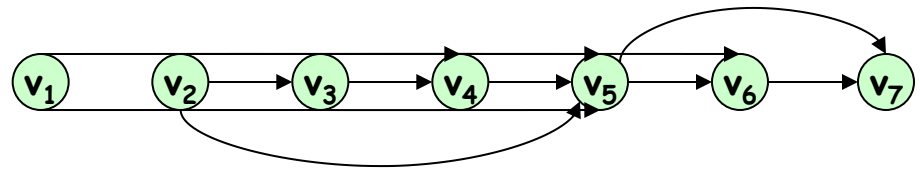
- ❖ อ่าน input file ที่รับข้อมูลกราฟ
โดยใช้ DFS เพื่อพิมพ์ component ต่างๆ ของกราฟ และสรุปจำนวน component ทั้งหมด
- ❖ Input file
บรรทัดแรก คือ จำนวนโหนด
บรรทัดถัดไปเป็นข้อมูล โหนดแรก
เริ่มด้วยจำนวน neighbor ของโหนด ตามด้วย neighbor ของ node
ตามด้วยน้ำหนักร
ข้อมูลโหนดถัดไป ก็จะอยู่บรรทัดถัดไปตามลำดับ
- ❖ กำหนดให้ใช้ adjacency list

Directed Acyclic Graphs (DAG)

- ❖ An **DAG** is a directed graph that contains no directed cycles.
- ❖ Example. Precedence constraints: edge (v_i, v_j) means v_i must precede v_j .
- ❖ Def. A **topological order** of a directed graph $G = (V, E)$ is an ordering of its nodes as v_1, v_2, \dots, v_n so that for every edge (v_i, v_j) we have $i < j$.



a DAG



a topological ordering

Exercise

- ❖ อ่าน input file ที่รับข้อมูล directed acyclic graph โดยใช้ DFS เพื่อพิมพ์ topological sort
- ❖ Input file
บรรทัดแรก คือ จำนวน โหนด
บรรทัดถัดไปเป็นข้อมูล โหนดแรก
เริ่มด้วยจำนวน neighbor ของ โหนด ตามด้วย neighbor ของ node ตามด้วยน้ำหนักร
ข้อมูล โหนดถัดไป ก็จะอยู่บรรทัดถัดไปตามลำดับ
- ❖ กำหนดให้ใช้ adjacency list