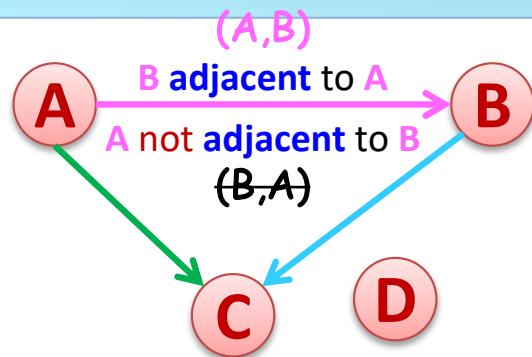




Graph

Kiatnarong Tongprasert

Graph Definitions



Graph $G = (V, E)$ ประกอบด้วย set 2 sets

$$V = \{ A, B, C, D \}$$

1. V = set of vertices (nodes)

$$E = \{ (A,B), (A,C), (B,C) \}$$

B adjacent to A

2. E = set of edges (arcs)

- **Directed graph (Digraph)**

(มีทิศทาง แทนด้วยลูกศรของ edge)

has directions associate with edges.

$$(A,B) \neq (B,A)$$

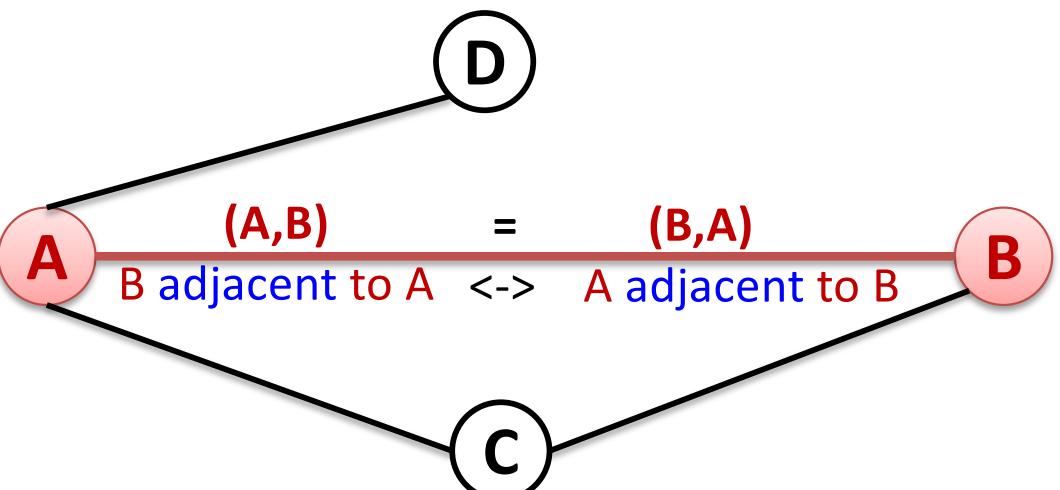
- **Undirected graph** (ไม่มีทิศทาง)

has no direction associate with edges.

$$(A,B) = (B,A)$$

- **B adjacent to A** (ต่อจาก)

ถ้ามี edge $(A,B) \in E$



ดังนั้นสำหรับ undirected graph

$B \text{ adjacent to } A \leftrightarrow A \text{ adjacent to } B$

Graph Definitions

มี 2 paths จาก A ไป D

ABD

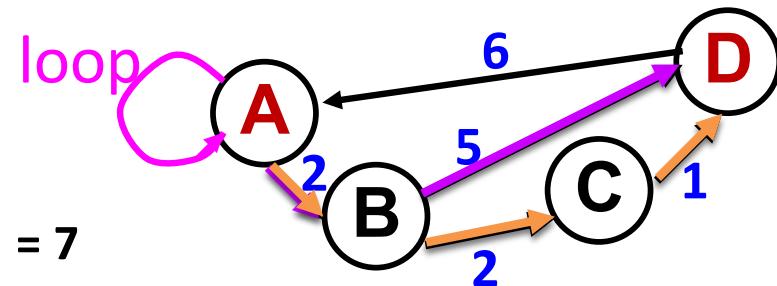
unweighted : path length = 2

weighted : path length = $2 + 5 = 7$

ABCD

unweighted : path length = 3

weighted : path length = $2 + 2 + 1 = 5$

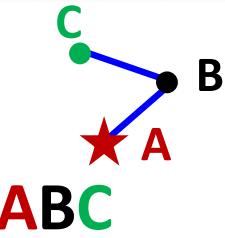
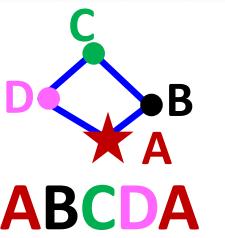
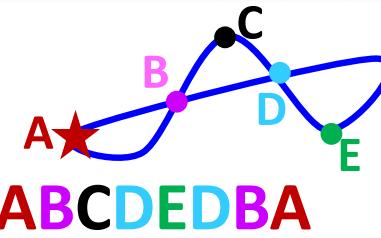
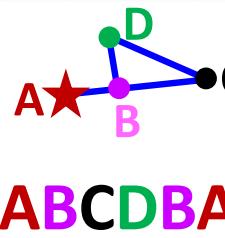


Weighted graph has weight assigned to each edge. (graph ที่มีน้ำหนักกำกับ edge)

Such weights might represent **costs, lengths or capacities, etc.** depending on the problem at hand. (น้ำหนัก อาจแสดงถึงสิ่งที่สนใจ เช่น ราคา ระยะทาง ความจุ เป็นต้น)

- **Path** (เส้นทางจาก node หนึ่งไป node หนึ่ง เช่นจาก W_1 ไป W_n)
 - : sequence of nodes $W_1, W_2, W_3, \dots, W_n$
when $(W_1, W_2), (W_2, W_3), \dots, (W_{n-1}, W_n) \in E$
 - **Path length** = # of edges in a path (unweighted graph) (= จำนวน edges ใน path)
= sum of weights of all edges in a path (weighted graph)
 - **Loop** : path of length 0 from v to v ie. think that there is edge(v,v).

Cycle, Simple Path

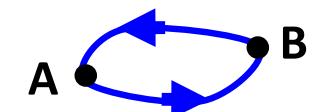
 ABC	 ABCD	 ABCDEDBA	 ABCDBA
simple path	simple path	non simple path	non simple path
acyclic	simple cycle	cycle	cycle

Path : เส้นทางเดิน

Simple path: path ซึ่งผ่าน vertex นั้นอย่างมาก 1 ครั้ง เว้น vertex แรกกับ vertex สุดท้าย ข้ามได้ (เส้นทางที่ไม่ผ่าน vertex ซ้ำ)

Cycle graph (circular graph) :

มี cycle อย่างน้อย 1 cycle (มี vertex ซึ่งวนกลับมาที่เดิม เป็น closed chain)

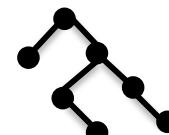


Simple Cycle : Simple path + Cycle

Cycle in undirected graph: edges ต้องไม่ใช่ edge เดียวกัน

ie. path UVU ไม่ควรเป็น cycle เพราะ (U,V) และ (V,U) เป็น edge เดียวกัน

Acyclic Graph: no cycle



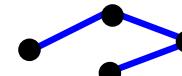
Directed Acyclic Graph = DAG ==> Tree



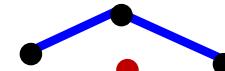
Connected VS Disconnected

- **Undirected graph**

- Connected มี path จากทุก vertex ไปยังทุก vertex

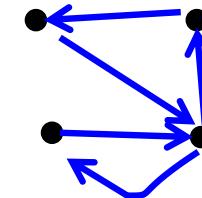


- Disconnected



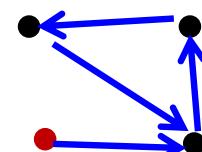
- **Directed graph**

- Strongly Connected มี path จากทุก vertex ไปยังทุก vertex

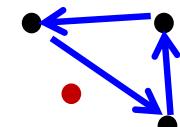


- Weakly Connected หากเปลี่ยนเป็น Undirected graph แล้วมี path จากทุก vertex ไปยังทุก vertex

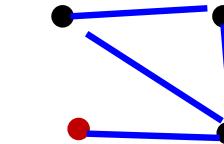
path จากทุก vertex ไปยังทุก vertex



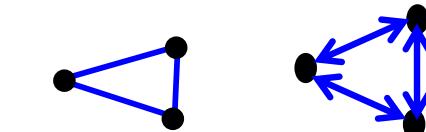
- Disconnected



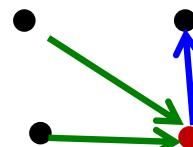
- **Complete graph** มี edge เชื่อมทุกคู่ของ nodes



- **Indegree** จำนวน edges ที่เข้า vertex



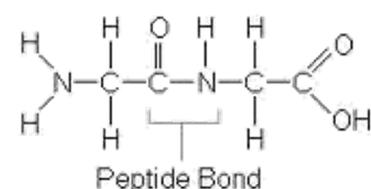
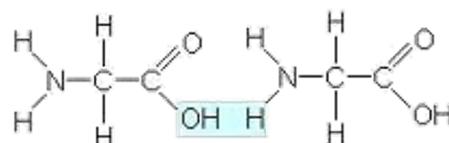
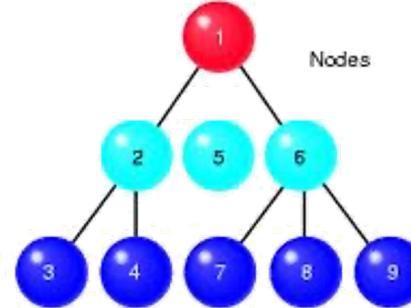
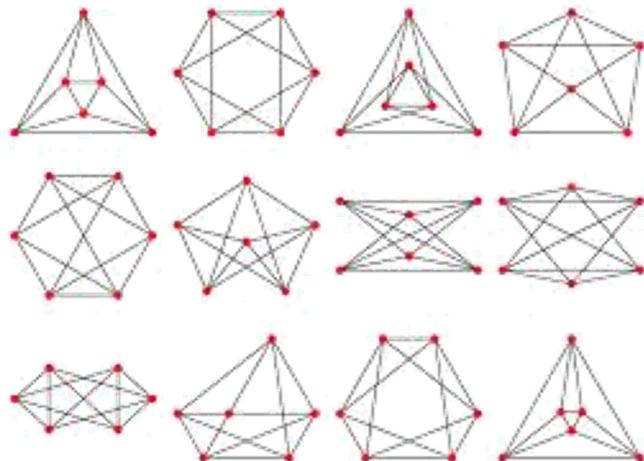
- **Outdegree** จำนวน edges ที่ออกจากร vertex



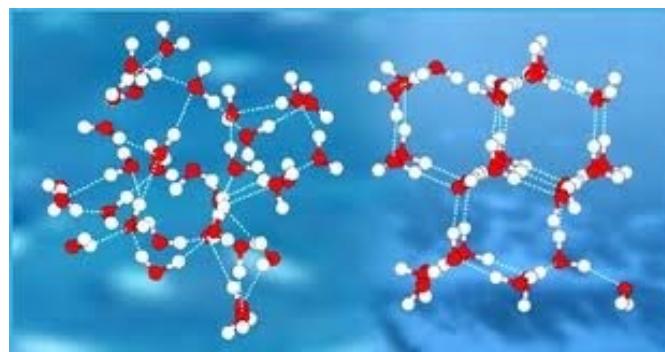
- has **indegree = 2**
- has **outdegree = 1**

Graph Examples

Airport System, Traffic Flow, ...

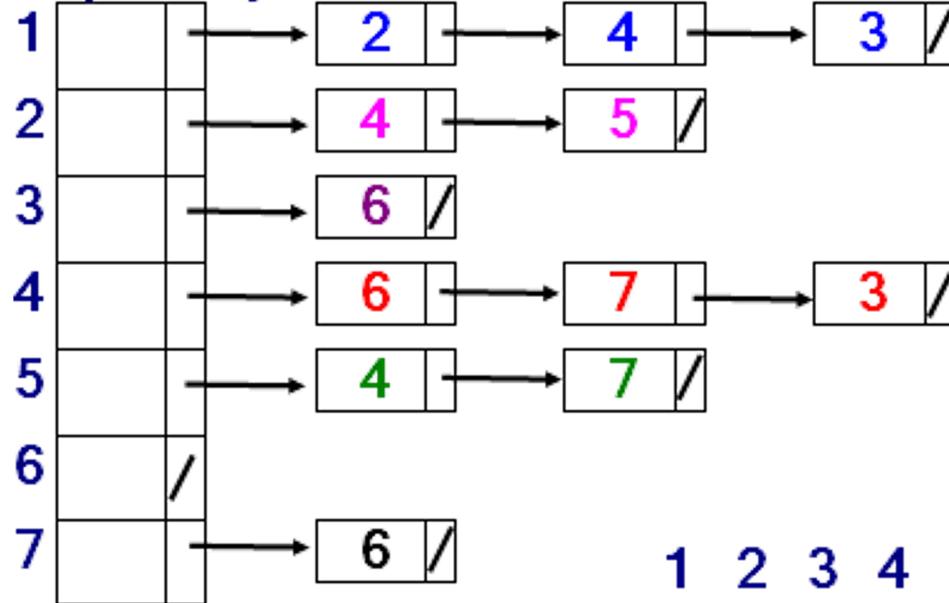


A molecule of water is removed from two glycine amino acids to form a peptide bond.

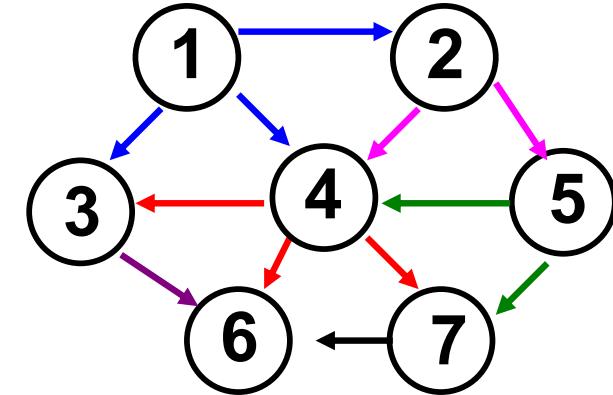


Graph Edge-Representations

Adjacency list



Linked List of vertices
กราฟมีการ insert/delete vertex บ่อย



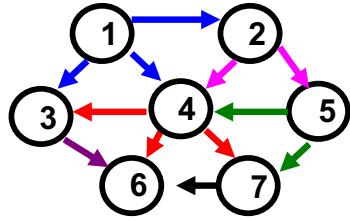
1 2 3 4 5 6 7

Adjacency matrix

1	T	T	T			
2			T	T		
3					T	
4		T		T	T	
5			T		T	
6						
7						T

- * simple
- * good for dense graph
- * bad for sparse matrix
- ex. street map
- * space ($\Theta|v^2|$)
- * undirected: (1,3), (3,1) = T

Graph Node-Representation

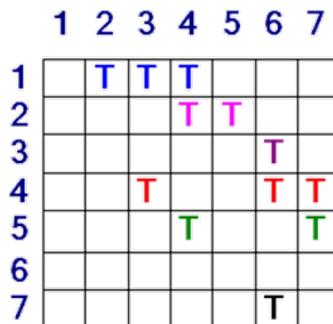
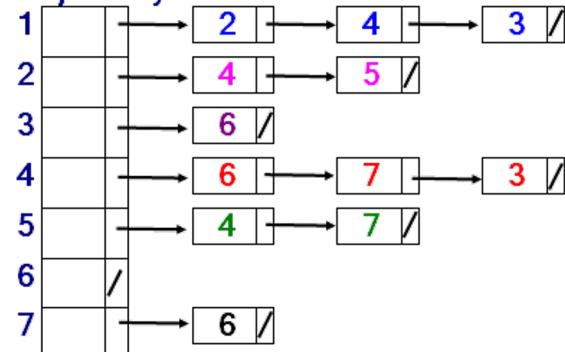


Node-Representation

	name	phone	address	...
0				
1	V1	0891761111		
2	V2			
3	V3			
4	V4			
5	V5			
6	v6			
7	v7			

Edge-Representations

Adjacency list



อาจใช้ array เก็บ records ของ vertices ดังรูปข้างสุด

และ link กับ ข้อมูล adjacency list ของแต่ละ vertex โดยใช้เลข index ที่เหมือนกัน

Graph representation python

```
# implement adjacency list
graph = {
    1: [2, 3, 4],
    2: [1, 3, 4],
    3: [1, 2, 4],
    4: [1, 2, 3]
}

# show adjacency list

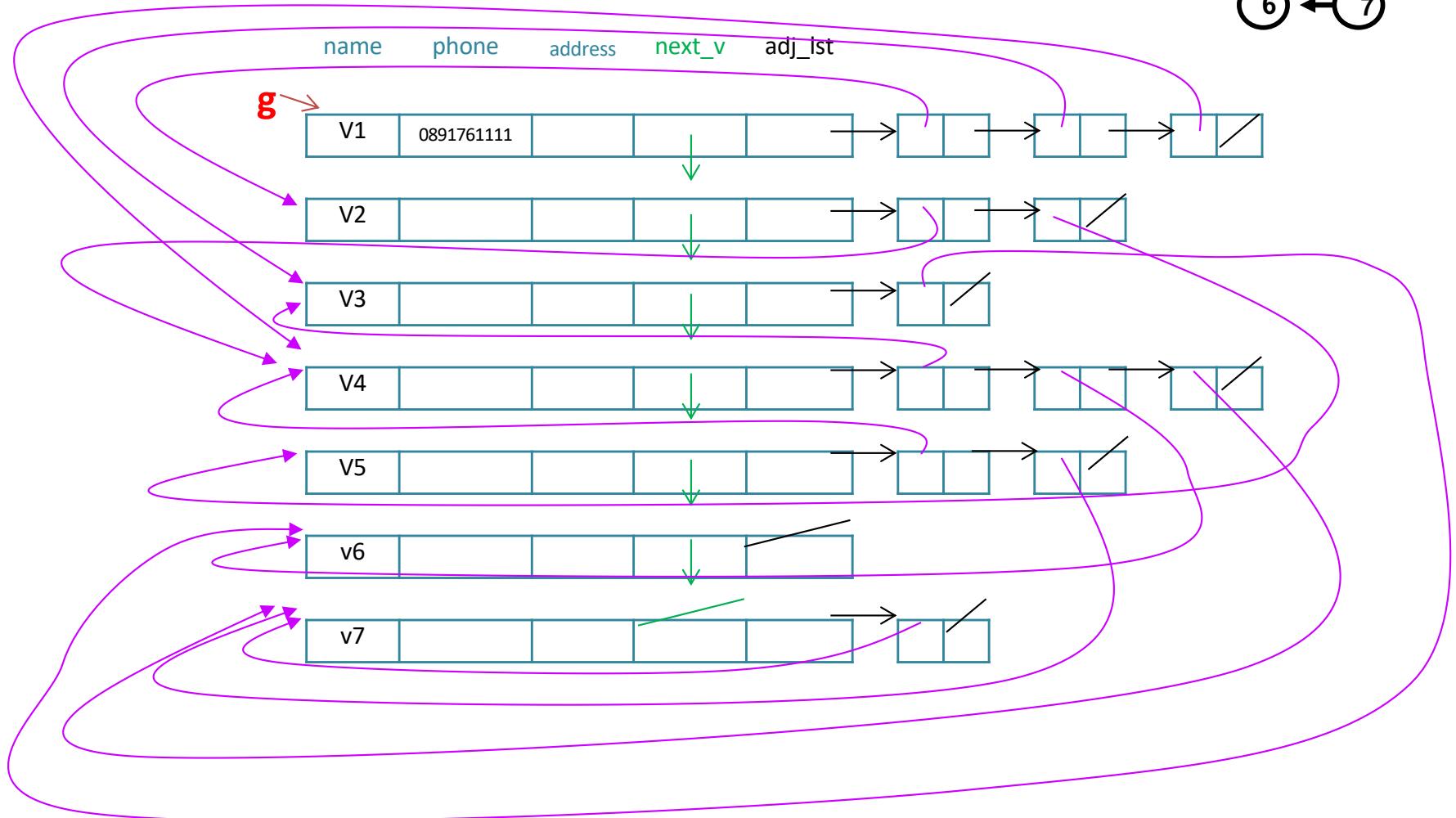
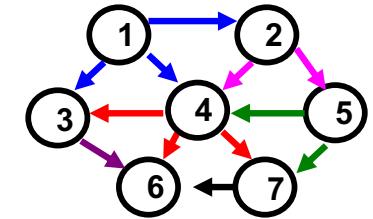
for vertex, neighbors in graph.items():
    print(f 'vertex {vertex}: {neighbors}')

# implement adjacency matrix
num_nodes = 4
adj_matrix = [[1] * num_nodes for _ in range(num_nodes)]

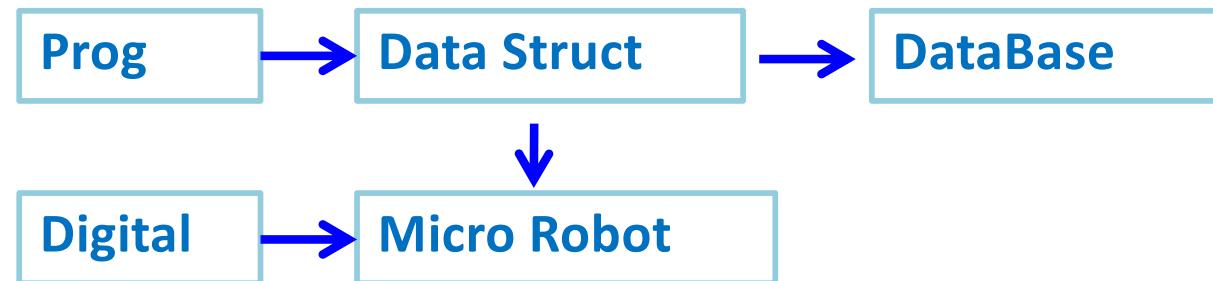
# show adjacency matrix
for row in adj_matrix:
    print(row)
```

Graph Representation

กรณี มีการเปลี่ยนแปลง เพิ่ม/ลด vertex บ่อยๆ ใช้ Adjacency list อาจเก็บข้อมูล vertex ใน linked list



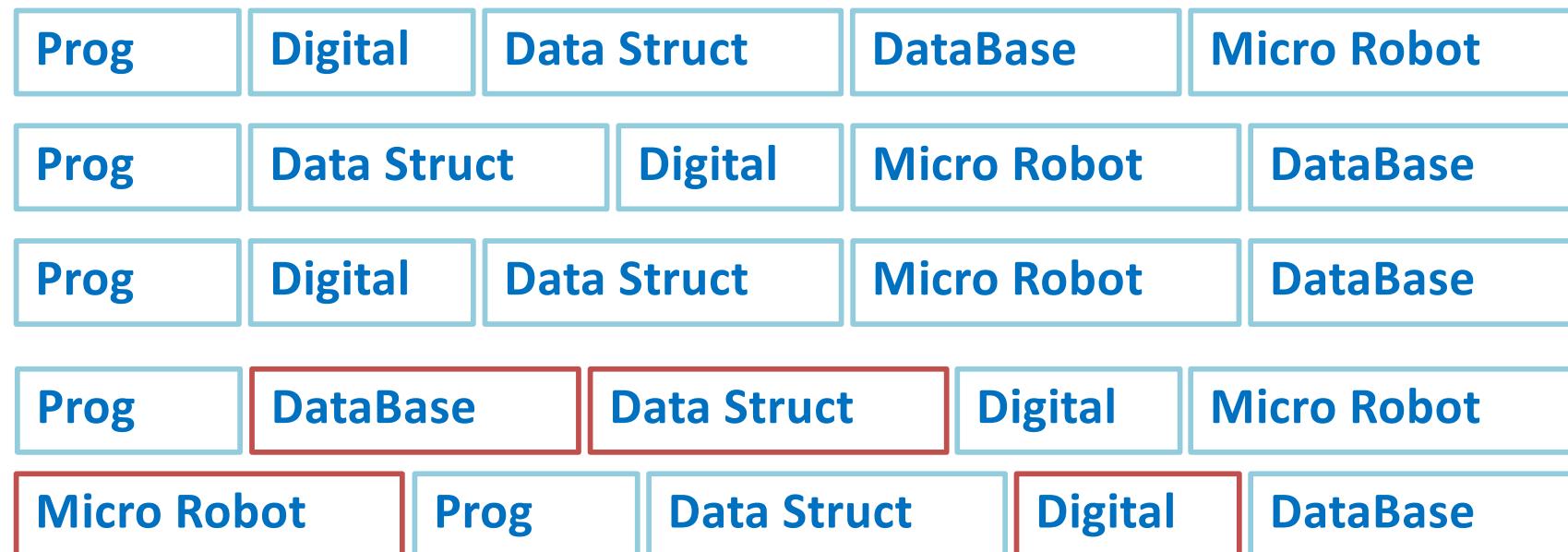
Topological sort



Topological sort : การเรียงใน acyclic graph ที่เรียงจากซ้ายไปขวา

ชึ้น ถ้ามี path จาก v_i ถึง v_j และ v_j จะต้องอยู่หลัง v_i

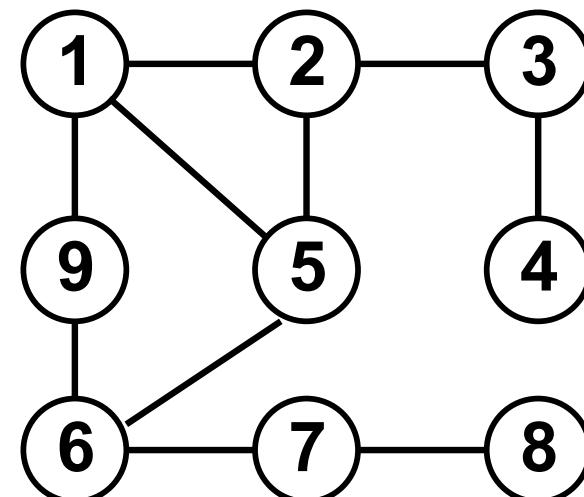
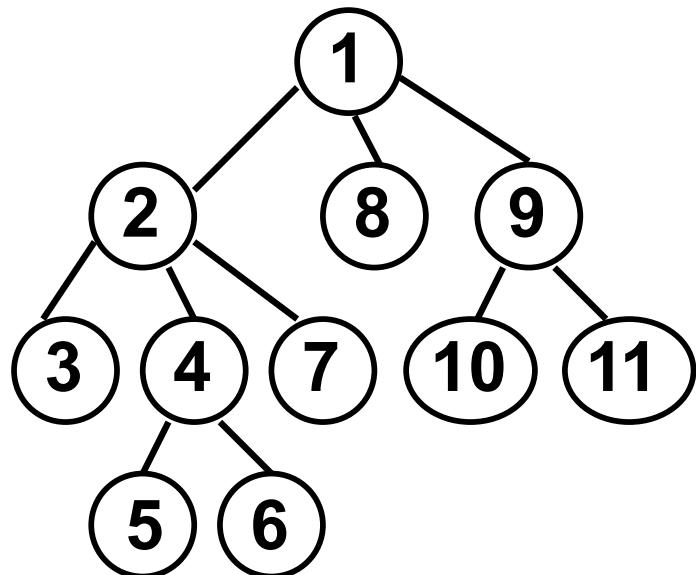
เช่น



Depth First Traversals

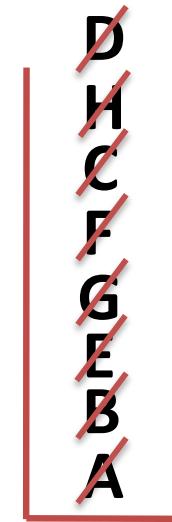
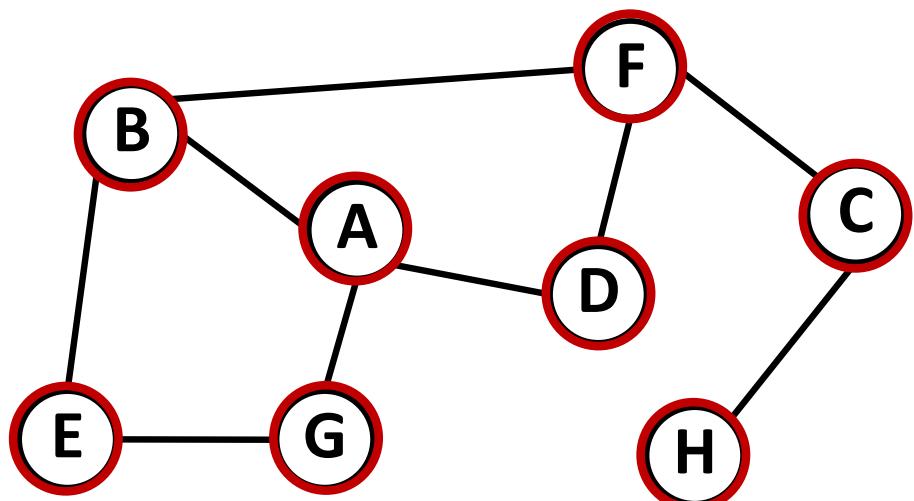
Depth First Traversal

visit V หาก V มี adjacent node ที่ยังไม่ได้ visit ให้ visit ตัวใดตัวหนึ่ง แล้ว ทำอย่างนี้ กับ node ที่พึ่ง visit ไปเรื่อยๆ เมื่อ node ที่พึ่ง visit ไม่มี adjacent node ที่ยังไม่ได้ visit เหลือแล้ว จึงค่อยกลับมา visit adjacent node ของ node ก่อนหน้าที่ยังเหลืออยู่ depth first traverse จึงใช้ stack ช่วย



Depth First Traversals

Depth First Traversals ไปด้านลึกก่อน : ใช้ stack ช่วย

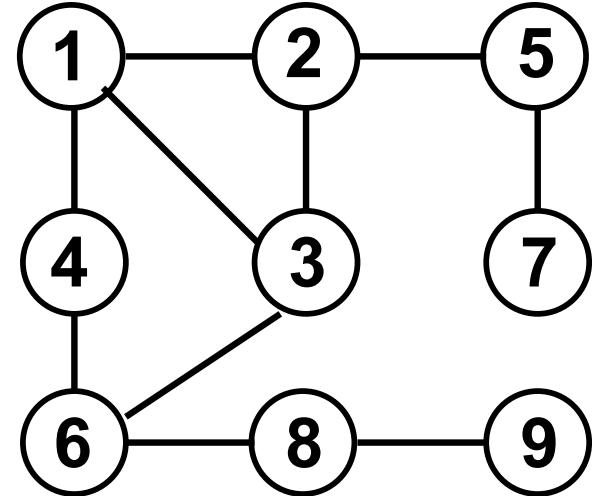
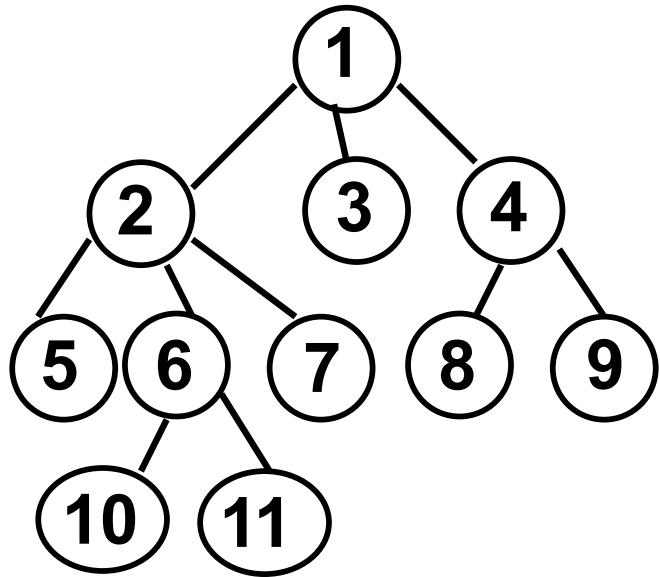


A, B, ... , H ?

Result: **A B E G F C H D**

Depth First Traversal จะได้หลาย solutions เพื่อให้ได้ solution ที่เหมือนกัน จะกำหนดว่า ถ้า traverse ไปได้หลาย node ให้ไป node ที่มีค่าน้อยที่สุดเสมอ เช่น ถ้าไปได้ทั้ง B E F ต้องเลือกไป B เพราะ B มีค่าน้อยที่สุด

Breadth First Traversals



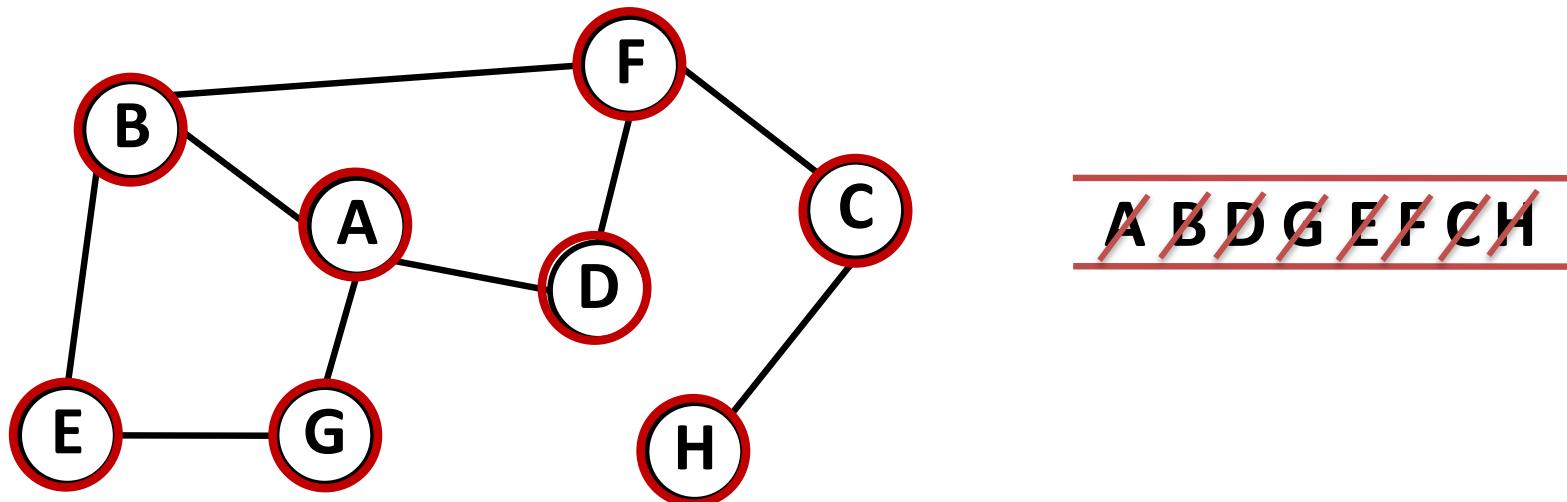
Breadth First Traversal (Level Order)

visit V ถ้า V มี adjacent node ที่ยังไม่ได้ visit ให้ visit ทุกตัวที่ adjacent กับมัน และ^{ทำขบวนการนี้ไปเรื่อยๆ กับ node ที่ถูก visit ไปตามลำดับการถูก visit ก่อนหลัง}

breadth first traverse จึงใช้ queue ช่วย

Breadth First Traversals

Breadth First Traversals : visit ทุกตัวที่ adjacent กับ node ที่เพิ่ง visit
ใช้ queue ช่วย



Result: **A B D G E F C H**

Depth First Traversal จะได้หลาย solutions เพื่อให้ได้ solution ที่เหมือนกัน
จะกำหนดว่า หากกำหนดว่าถ้า traverse ไปได้หลาย node ให้ไป node ที่มีค่าน้อย
ที่สุดเสมอ เช่น ถ้าไปได้ทั้ง B E F ต้องเลือกไป B เพราะ B มีค่าน้อยที่สุด

Depth First Traversals

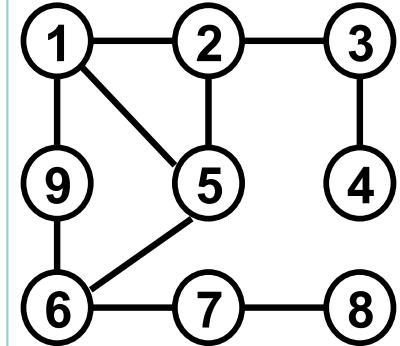
depth_first (vertex)

1. init bool **visited** [MAX] = false for all vertices.

visited	F	F	F	F	F	F	F	F	F	F
	0	1	2	3	4	5	6	7	8	9

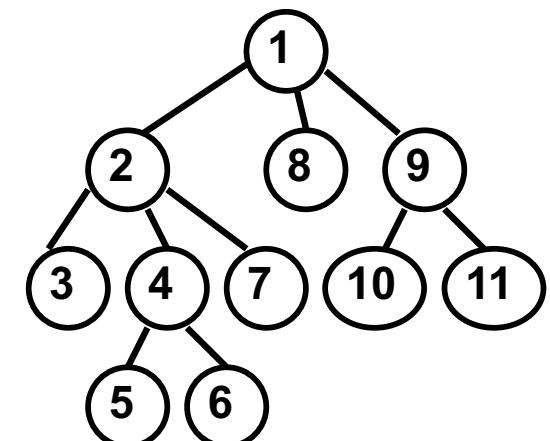
... ...

2. for all un-visited vertex **v** // run for disconnected graph node
traverse (v, visited);



traverse(v, visited)

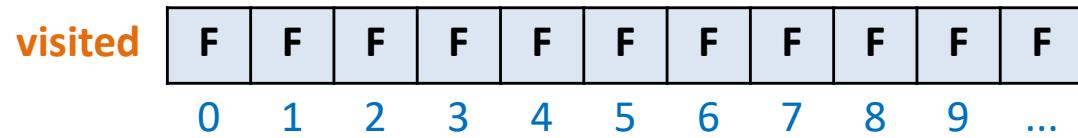
1. **print(v) ;**
2. **visited[v] = true;** // set v to be already visited
3. for all un-visited **w** that adjacent to **v**
traverse (w, visited);



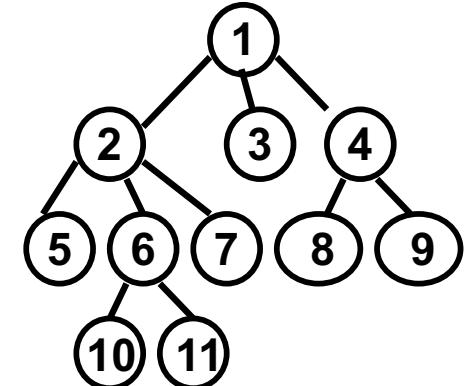
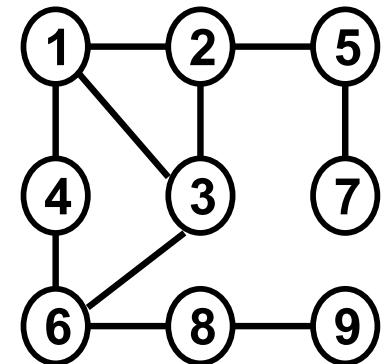
Breadth First Traversals

breadth_first (vertex)

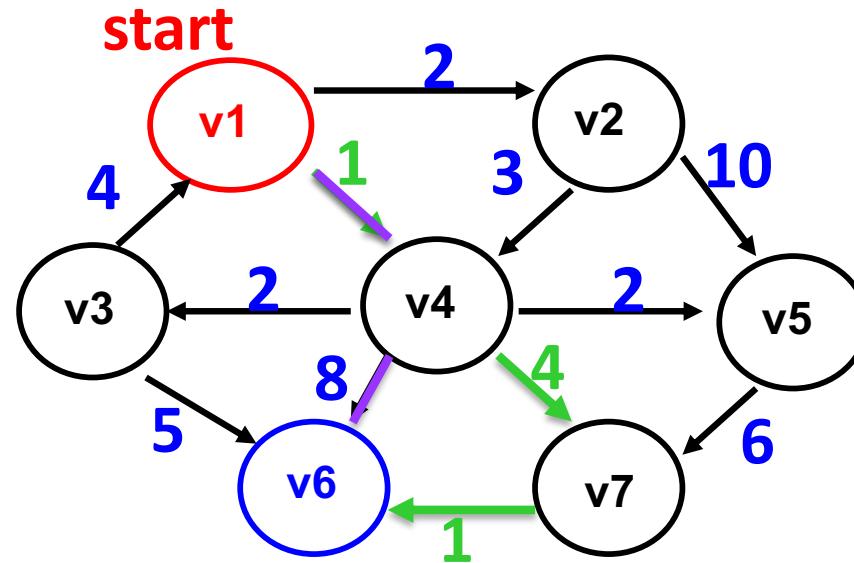
1. init bool **visited** [MAX]; = false for all vertices.



2. init empty queue q;
3. for all un-visited vertex **v** // run for disconnected graph node
 enqueue(q, **v**)
 while (not empty q)
 w = q.dequeue()
 if (!**visited**[**w**])
 visited[**w**] = true; // set **w** to be already visited
 print(**w**);
 for all un-visited **x** that adjacent **w** and **x** is not in q
 q.enqueue(**x**)



Shortest Path



- shortest **weighted** path **v1** to **v6** :
= **v1,v4,v7,v6** cost = **1+4+1 = 6**
- shortest **unweighted** path **v1** to **v6** :
= **v1,v4,v6** cost = **2**

Greedy Algorithm

- Greedy Algorithm : เลือกอันที่ดีที่สุดสำหรับ stage ปัจจุบัน
(อาจไม่ได้ optimum solution)
- ตย. แลกเหรียญให้ได้จำนวนเหรียญน้อยที่สุด

	quarter	25	cents
suppose we have 12_cent_coin == >		12	cents
	dime	10	cents
	nikle	5	cents
	penny	1	cents

15 cents : Greedy \rightarrow 12, 1, 1, 1
 : (optimum \rightarrow 10, 5)

Weighted Shortest Paths (Dijkstra's algorithm)

Greedy : for each current stage, choose the best.

Data Structures : สำหรับ vertex v ได้ เก็บข้อมูล 3 ตัว :

distance = ระยะจากจุด start ไปยัง vertex นั้นๆ

known เป็นจริง เมื่อทราบระยะ distance ที่สั้นที่สุดแล้ว

path = vertex ก่อนหน้ามันใน shortest path

vertices ทั้งหมด : **known** = false;

start_vertex : distance = 0;

vertices อื่นๆ : distance = ∞ ;

for(; ;)

v = vertex ที่มี dist. น้อยที่สุด ที่ known ยังเป็น false

if (ไม่มี **v**)

break;

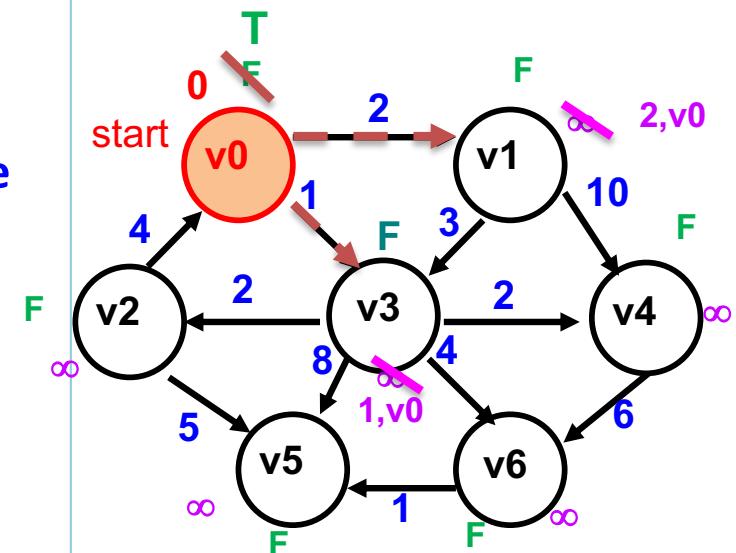
v.known = true;

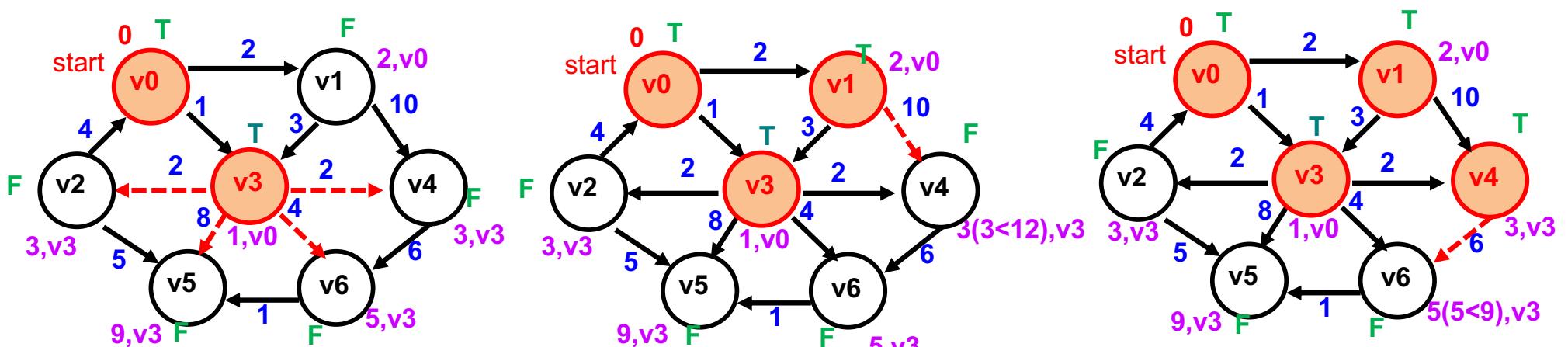
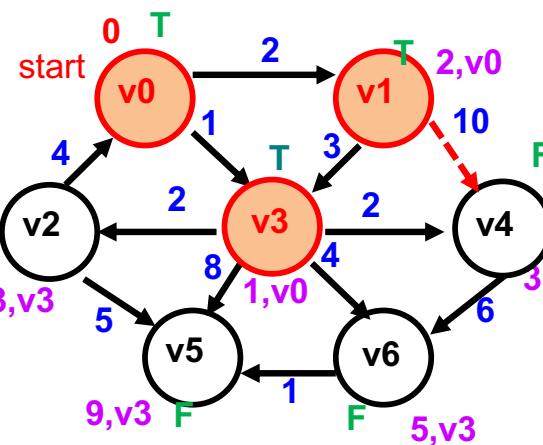
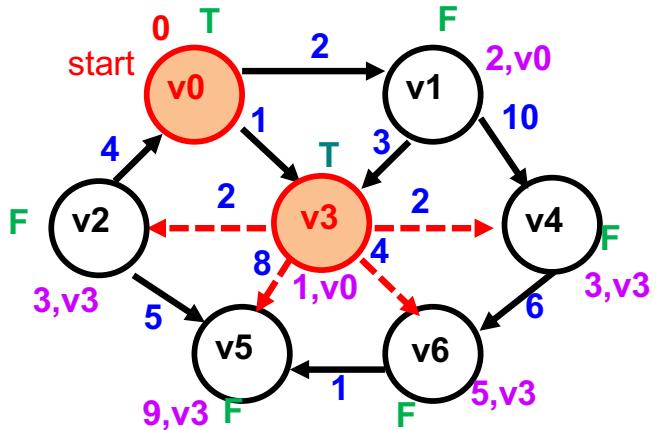
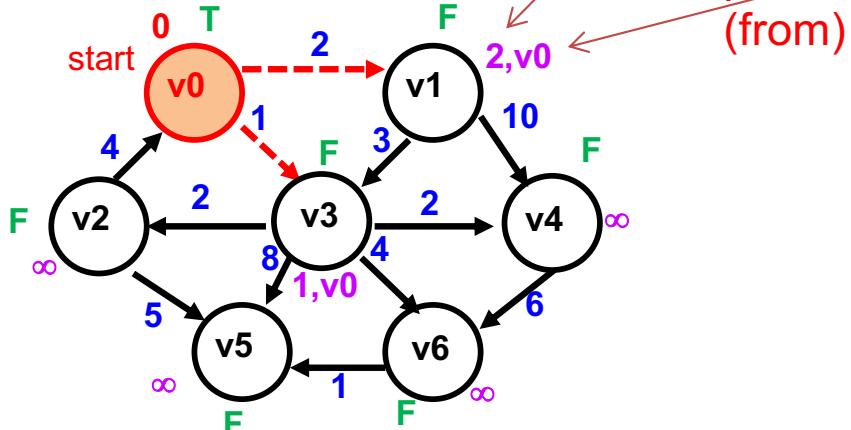
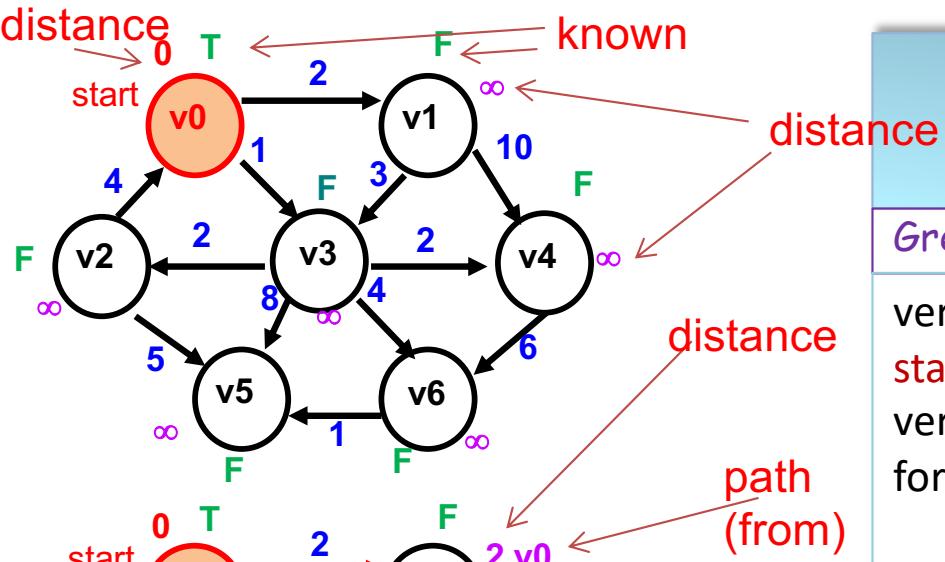
for each w adjacent to v ซึ่งยังไม่ถูก process

if ($w.dist > v.dist + \text{weight}(v,w)$)

ปรับ $w.dist$ เป็นค่าใหม่ซึ่งน้อยกว่า

w.path = **v**;





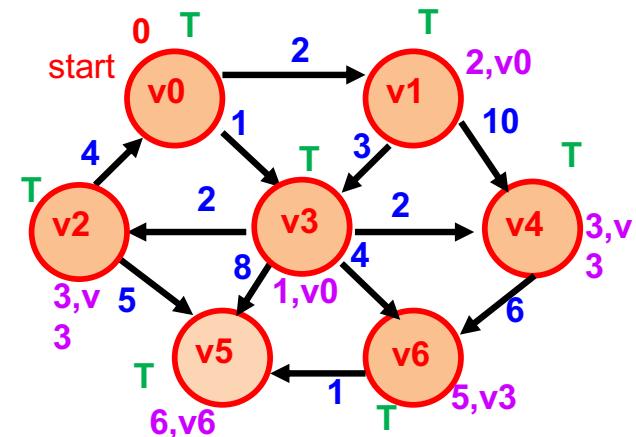
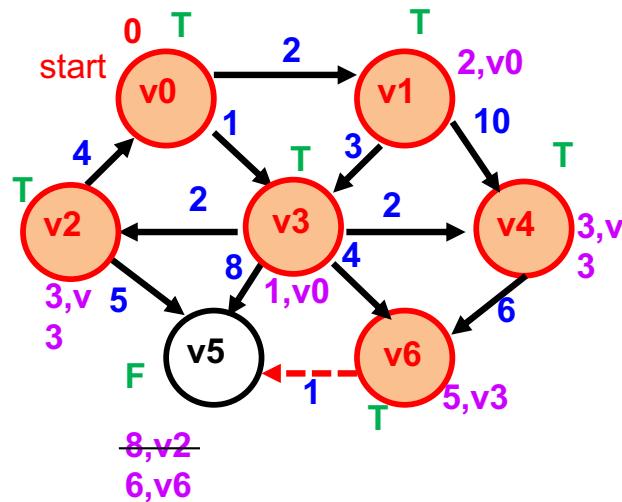
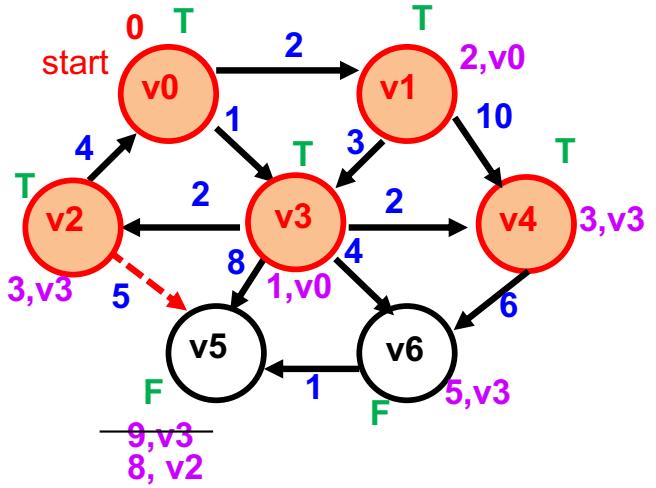
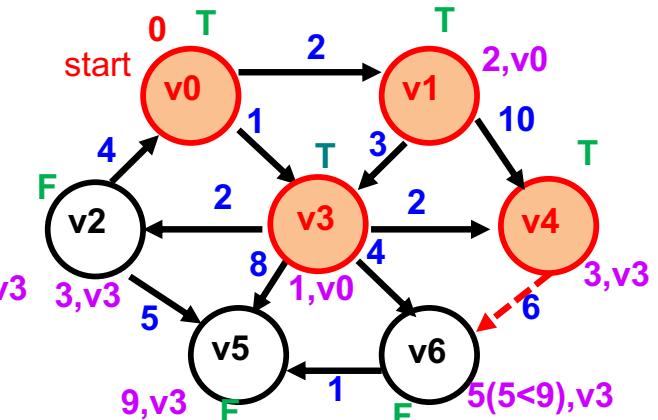
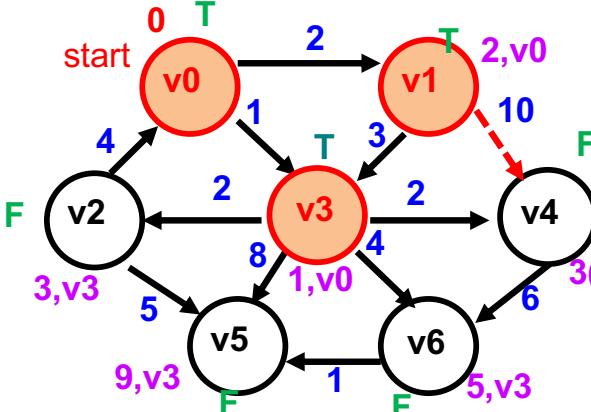
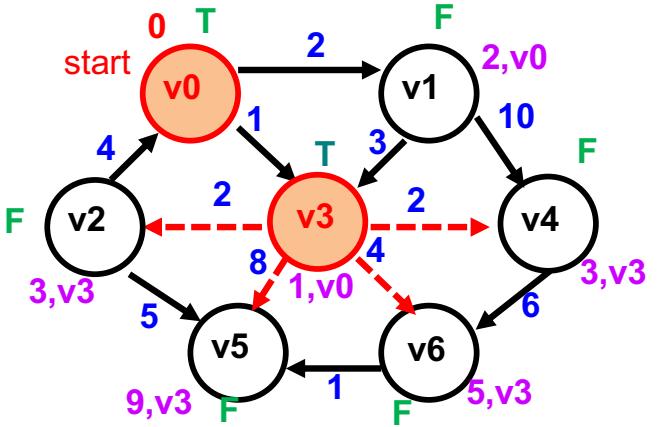
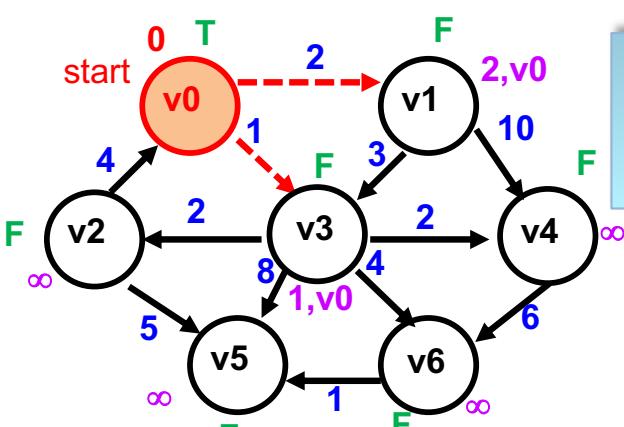
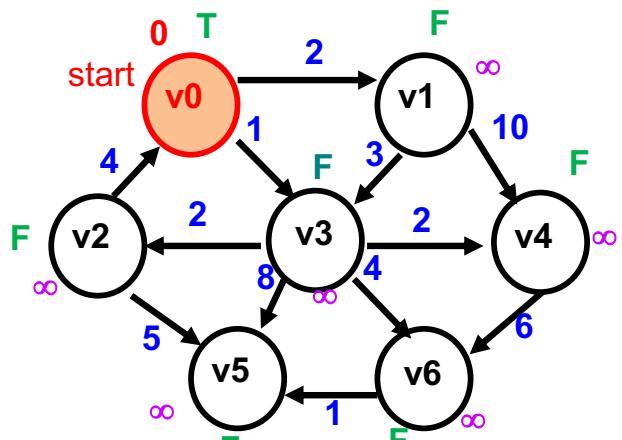
Weighted Shortest Paths (Dijkstra's algorithm)

Greedy : for each current stage, choose the best.

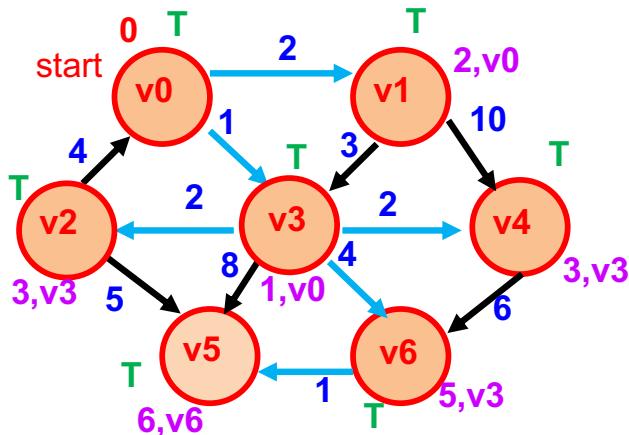
```

vertices ทั้งหมด : known = false;
start_vertex : distance = 0;
vertices อื่นๆ : distance = ∞;
for( ; );
    v = vertex ที่มี dist. น้อยที่สุด ที่ known ยังเป็น false
    if (ไม่มี v )
        break;
    v.known = true;
    for each w adjacent to v ซึ่งยังไม่ถูก process
        if (w.dist > v.dist + weight(vw))
            ปรับ w.dist เป็นค่าใหม่ซึ่งน้อยกว่า
            w.path = v;
```

Weighted Shortest Paths (Dijkstra's algorithm)



Weighted Shortest Paths (Dijkstra's algorithm)



วิธีอ่านค่า shortest path

- จากรูปผลลัพธ์ของ Dijkstra's algorithm เราสามารถหา shortest path จาก start vertex ไปยังทุก vertices ที่เหลือได้ ดังนี้ เช่น
 - ต้องการหา shortest path จาก start (ในที่นี้คือ v0) ไปยัง v5
 - v0 $\xrightarrow{1}$ v3 $\xrightarrow{4}$ v6 $\xrightarrow{1}$ v5 //path length = 6 //vertex v0 มายัง v5 มี shortest path ดังนี้
 - v0 $\xrightarrow{1}$ v3 $\xrightarrow{4}$ v6 $\xrightarrow{1}$ v5 //เขียน vertex ตั้งต้น v0 และ vertex ปลายทาง v5
 - v0 $\xrightarrow{1}$ v3 $\xrightarrow{4}$ v6 $\xrightarrow{1}$ v5 //vertex ปลายทาง v5 มาจาก path v6
 - v0 $\xrightarrow{1}$ v3 $\xrightarrow{4}$ v6 $\xrightarrow{1}$ v5 //vertex v6 มาจาก path v3
 - v0 $\xrightarrow{1}$ v3 $\xrightarrow{4}$ v6 $\xrightarrow{1}$ v5 //vertex v3 มาจาก path v0
 - ต้องการหา shortest path จาก start (ในที่นี้คือ v0) ไปยัง v4
 - v0 $\xrightarrow{1}$ v3 $\xrightarrow{2}$ v4 //path length = 3 //เขียน vertex ตั้งต้น v0 และ vertex ปลายทาง v4
 - v0 $\xrightarrow{1}$ v3 $\xrightarrow{2}$ v4 //vertex ปลายทาง v4 มาจาก path v3
 - v0 $\xrightarrow{1}$ v3 $\xrightarrow{2}$ v4 //v3 มาจาก v0: shortest path จึงเป็นดังนี้
 - shortest path จาก v0 ไปยัง v6
 - v0 $\xrightarrow{1}$ v3 $\xrightarrow{4}$ v6 //จะเห็นว่าเป็นเส้นทางผ่านของ v0 ไป v5 ในตัวอย่างแรก

Social Network Analysis: SNA

การวิเคราะห์เครือข่ายสังคม (Social Network Analysis: SNA) คือกระบวนการศึกษาความสัมพันธ์ และโครงสร้างระหว่างบุคคล กลุ่ม หรือองค์กรในเครือข่ายทางสังคม โดยใช้ граф ใน การแสดงความ เชื่อมโยงระหว่างผู้คนหรือหน่วยต่างๆ ในระบบเครือข่าย

องค์ประกอบหลักของ SNA:

- 1.**โหนด (Nodes):** แทนบุคคล องค์กร หรือวัตถุที่เชื่อมโยงกัน เช่น คนในเครือข่ายสังคมออนไลน์ ผู้ใช้ใน Facebook, Twitter, หรือ LinkedIn
- 2.**เส้นเชื่อม (Edges):** แทนความสัมพันธ์หรือการปฏิสัมพันธ์ เช่น ความเป็นเพื่อน การสนทนากา ทำงานร่วมกัน หรือการส่งต่อข้อมูล
- 3.**Degree Centrality:** จำนวนของเส้นเชื่อมที่โหนดหนึ่งๆ มี ซึ่งสะท้อนถึงความสำคัญหรืออิทธิพลใน เครือข่าย
- 4.**Closeness Centrality:** การวัดว่าหนึ่งโหนดใกล้ชิดกับโหนดอื่นๆ ในเครือข่ายเพียงใด
- 5.**Betweenness Centrality:** การวัดว่าโหนดหนึ่งมีความสำคัญเพียงใดในการเชื่อมต่อส่วนต่างๆ ของ เครือข่าย
- 6.**Cluster หรือ Community Detection:** การแบ่งเครือข่ายออกเป็นกลุ่มย่อยที่มีความเชื่อมโยงกัน มากกว่ากับกลุ่มอื่น

ประโยชน์ของการวิเคราะห์เครือข่ายสังคม:

1. การทำความเข้าใจโครงสร้างสังคม ช่วยให้เห็นภาพรวมของการปฏิสัมพันธ์ในกลุ่มสังคม เช่น ใครคือผู้ที่มีอิทธิพล หรือใครที่เป็นศูนย์กลางของเครือข่าย
2. การกระจายข้อมูลหรืออิทธิพล SNA ช่วยวิเคราะห์ว่าใครเป็นผู้กระจายข้อมูลหรือเป็นสื่อกลางในเครือข่าย
3. การวิเคราะห์ความสัมพันธ์ทางธุรกิจ ใช้ในการวิเคราะห์ความร่วมมือระหว่างบริษัท องค์กร หรือกลุ่มลูกค้า
4. การประยุกต์ในโลกออนไลน์ ช่วยวิเคราะห์แพลตฟอร์มโซเชียลมีเดีย เช่น การวิเคราะห์เครือข่ายของผู้ใช้ Facebook หรือ Twitter เพื่อเข้าใจถึงการแพร่กระจายของข้อมูลและความสนใจของกลุ่มผู้ใช้

Social Network Analysis: SNA

1. โนนด (Nodes) และเส้นเชื่อม (Edges)

- โนนด (Nodes):** ตัวแทนของหน่วยในเครือข่าย เช่น บุคคล องค์กร หรือแม้แต่ตัวๆ ซึ่งใน SNA เรามักจะศึกษาโนนดในบริบทของการเชื่อมโยงทางสังคม
- เส้นเชื่อม (Edges):** เป็นตัวแทนของความสัมพันธ์หรือการปฏิสัมพันธ์ระหว่างโนนด ซึ่งสามารถเป็นได้ทั้งแบบมีทิศทาง (Directed) และไม่มีทิศทาง (Undirected)
 - ความสัมพันธ์แบบมีทิศทาง เช่น ในเครือข่าย Twitter หาก A ติดตาม B ความสัมพันธ์นี้มีทิศทางจาก A ไป B แต่ไม่ใช่กลับกัน
 - ความสัมพันธ์แบบไม่มีทิศทาง เช่น ในเครือข่ายเพื่อนบน Facebook ความเป็นเพื่อนเป็นความสัมพันธ์ที่เท่าเทียมกันระหว่างทั้งสองโนนด

2. โครงสร้างของเครือข่าย (Network Structure)

- **Degree:** จำนวนเส้นเชื่อมที่โหนดหนึ่งมีอยู่ เช่น หากโหนดหนึ่งมีเส้นเชื่อมกับโหนดอื่นๆ ทั้งหมด 5 เส้น หมายความว่ามีความสัมพันธ์กับโหนดอื่นๆ 5 โหนด ซึ่งสามารถวัดเป็น **In-degree** (จำนวนเส้นเชื่อมที่เข้ามาหาโหนดนั้น) และ **Out-degree** (จำนวนเส้นเชื่อมที่โหนดนั้นเชื่อมออกไป)
- **Density:** ความหนาแน่นของเครือข่าย หมายถึงระดับความเชื่อมโยงของเครือข่าย ถ้าเครือข่ายมีความเชื่อมโยงมาก Density ก็จะสูง
- **Path:** เส้นทางระหว่างโหนดที่แสดงถึงลำดับความสัมพันธ์ เช่น A → B → C
- **Cluster หรือ Community:** กลุ่มของโหนดที่มีความเชื่อมโยงระหว่างกันมากกว่าความเชื่อมโยงกับโหนดนอกกลุ่ม

3. การวัดความสำคัญของโหนด (Centrality Metrics)

การวิเคราะห์เครือข่ายจะมองว่าโหนดไหนสำคัญที่สุดหรือมีอิทธิพลมากที่สุด โดยใช้เกณฑ์ต่างๆ ได้แก่:

- Degree Centrality:** โหนดที่มีจำนวนเส้นเชื่อมมากที่สุด มีอิทธิพลในเครือข่ายมากที่สุด
- Betweenness Centrality:** โหนดที่ทำหน้าที่เป็นทางเชื่อมผ่านระหว่างกลุ่มโหนดต่างๆ หรือ เป็นทางผ่านของข้อมูล โหนดนี้มีบทบาทสำคัญในการควบคุมการแพร่กระจายของข้อมูล
- Closeness Centrality:** โหนดที่อยู่ใกล้ชิดกับโหนดอื่นๆ ในเครือข่ายมากที่สุด สามารถเข้าถึงข้อมูลจากโหนดอื่นได้รวดเร็วกว่าโหนดอื่น
- Eigenvector Centrality:** วัดความสำคัญของโหนดโดยพิจารณาจากความเชื่อมโยงกับโหนดสำคัญอื่นๆ หากโหนดหนึ่งเชื่อมโยงกับโหนดสำคัญหลายๆ โหนด โหนดนั้นก็จะมี Eigenvector Centrality สูง

4. การแบ่งกลุ่มเครือข่าย (Community Detection)

การศึกษาการแบ่งกลุ่มในเครือข่ายช่วยให้เราเห็นความสัมพันธ์ที่ชัดเจนระหว่างกลุ่มต่างๆ:

- **Modularity:** เป็นวิธีการวัดการแบ่งกลุ่มในเครือข่าย โดยเครือข่ายที่มี Modularity สูงมากจะมีการแบ่งกลุ่มที่ชัดเจน ซึ่งแต่ละกลุ่มมีความเชื่อมโยงภายในสูงแต่เชื่อมโยงกับกลุ่มอื่นต่ำ
- **Girvan-Newman Algorithm:** เป็นวิธีการหา Community โดยตัดเส้นเชื่อมที่มี Betweenness Centrality สูงๆ ออก เพื่อแยกกลุ่มออกจากกัน

5. การนำไปประยุกต์ใช้ของ SNA

1. เครือข่ายออนไลน์ (Online Social Networks): การวิเคราะห์แพลตฟอร์มโซเชียลมีเดีย เช่น Facebook, Twitter ช่วยให้เราเข้าใจว่าข้อมูลหรือข่าวสารแพร่กระจายอย่างไร และใครเป็นผู้นำในการเผยแพร่ข้อมูล

2. การตลาดและการโฆษณา: ช่วยวิเคราะห์ว่าใครเป็นผู้ที่มีอิทธิพลในกลุ่มเป้าหมาย (Influencer) และช่วยระบุกลุ่มลูกค้าที่น่าจะตอบสนองต่อผลิตภัณฑ์หรือแคมเปญมากที่สุด

3. การวิเคราะห์เครือข่ายองค์กร: ช่วยให้เข้าใจโครงสร้างการสื่อสารภายในองค์กร รวมถึงการวางแผนยุทธ์ในการทำงานร่วมกันระหว่างทีมต่างๆ

4. การแพร่ระบาดของโรค: ใช้เพื่อศึกษาการแพร่ระบาดของโรคในชุมชน หรือวิเคราะห์ว่าใครเป็นคนแพร่เชื้อหลักในกลุ่มเครือข่าย

1. Degree Centrality

Degree Centrality วัดจำนวนเส้นเชื่อม (Edges) ที่เชื่อมต่อกับโหนดหนึ่งๆ โหนดที่มี Degree สูงกว่ามักมีบทบาทสำคัญในเครือข่าย

สูตรคำนวณ Degree Centrality:

$$C_D(v) = \frac{\text{Degree of node } v}{\text{Total nodes} - 1}$$

```
import networkx as nx
# สร้างกราฟตัวอย่าง
G = nx.Graph()
G.add_edges_from([(1, 2), (1, 3), (2, 3), (2, 4)])
# คำนวณ Degree Centrality
degree_centrality = nx.degree_centrality(G)
print(degree_centrality)
```

2. Closeness Centrality

Closeness Centrality วัดว่าหนึ่งโหนดอยู่ใกล้โหนดอื่นๆ เพียงใด โดยคำนวณจากระยะทางระหว่างโหนด

สูตรคำนวณ Closeness Centrality:

$$C_C(v) = \frac{n - 1}{\sum_u d(u, v)}$$

```
# คำนวณ Closeness Centrality
```

```
closeness_centrality = nx.closeness_centrality(G)  
print(closeness_centrality)
```

3. Betweenness Centrality

Betweenness Centrality วัดว่าโหนดทำหน้าที่เป็นทางผ่านของการเชื่อมต่อระหว่างโหนดอื่นๆ ในเครือข่ายมากน้อยเพียงใด

สูตรคำนวณ Betweenness Centrality:

$$C_B(v) = \sum_{s \neq v \neq t} \frac{\sigma(s, t | v)}{\sigma(s, t)}$$

คำนวณ Betweenness Centrality

```
betweenness_centrality = nx.betweenness_centrality(G)  
print(betweenness_centrality)
```

4. Eigenvector Centrality

Eigenvector Centrality วัดความสำคัญของโหนดโดยพิจารณาจากความสำคัญของโหนดที่เชื่อมต่อด้วย

สูตรคำนวณ Eigenvector Centrality:

$$C_E(v) = \frac{1}{\lambda} \sum_{u \in N(v)} C_E(u)$$

คำนวณ Eigenvector Centrality

```
eigenvector_centrality = nx.eigenvector_centrality(G)  
print(eigenvector_centrality)
```

Social Network Analysis: SNA

```
import networkx as nx
import matplotlib.pyplot as plt

# สร้างกราฟเครือข่ายขนาดใหญ่
G = nx.Graph()

# เพิ่มโหนดและเส้นเชื่อม(Edges) ที่รับข้อมูลมากขึ้น
edges = [('Alice', 'Bob'), ('Alice', 'Carol'), ('Bob', 'Dave'),
          ('Carol', 'Eve'), ('Dave', 'Eve'), ('Alice', 'Eve'),
          ('Bob', 'Frank'), ('Frank', 'Eve'), ('Carol', 'Frank'),
          ('George', 'Alice'), ('George', 'Dave'), ('George', 'Eve'),
          ('Helen', 'Bob'), ('Helen', 'Carol'), ('Helen', 'Frank'),
          ('Ivy', 'Alice'), ('Ivy', 'Carol'), ('Ivy', 'George'),
          ('Jack', 'Dave'), ('Jack', 'Eve'), ('Jack', 'Frank'),
          ('Kathy', 'Bob'), ('Kathy', 'George'), ('Kathy', 'Jack'),
          ('Leo', 'Carol'), ('Leo', 'Eve'), ('Leo', 'Helen')]

G.add_edges_from(edges)

# คำนวณ Degree Centrality เพื่อใช้กำหนดขนาดของโหนด
degree_centrality = nx.degree_centrality(G)
node_size = [v * 1500 for v in degree_centrality.values()] # ขยายขนาดโหนดเพื่อให้แสดงเด่นชัด

# สร้างสีของโหนดเพื่อแสดง Community ต่างๆ
node_colors = ['skyblue' if G.degree(n) > 3 else 'lightgreen' for n in G.nodes]

# วาดรูป
plt.figure(figsize=(10, 10)) # ขยายขนาดรูป
nx.draw_networkx(G, node_size=node_size, node_color=node_colors, with_labels=True,
                 font_size=10, edge_color='gray')

# แสดงภาพ
plt.title("Larger Social Network Analysis (SNA) Visualization", fontsize=15)
plt.show()
```

Social Network Analysis: SNA

สรุป

SNA ช่วยให้เราเข้าใจโครงสร้างและการเชื่อมโยงในเครือข่ายสังคม สามารถใช้วัดผลกระทบหรือการแพร่กระจายของข้อมูลภายในเครือข่ายได้ ทำให้เราเห็นถึงความสำคัญของโนนดและความสัมพันธ์ในเครือข่ายนั้นๆ ซึ่งสามารถประยุกต์ใช้ในหลาย ๆ ด้าน ไม่ว่าจะเป็นธุรกิจ การสื่อสาร หรือการศึกษาการแพร่กระจายของโรค