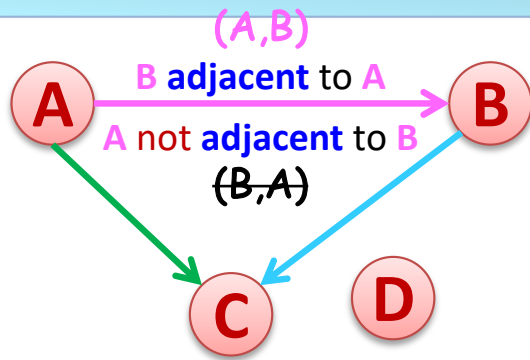




Graph

Kiatnarong Tongprasert

Graph Definitions



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

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

B adjacent to A

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

1. V = set of vertices (nodes)

2. E = set of edges (arcs)

- Directed graph (Digraph)**

(มีทิศทาง แทนด้วยลูกศรของ edge)

has directions associate with edges.

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

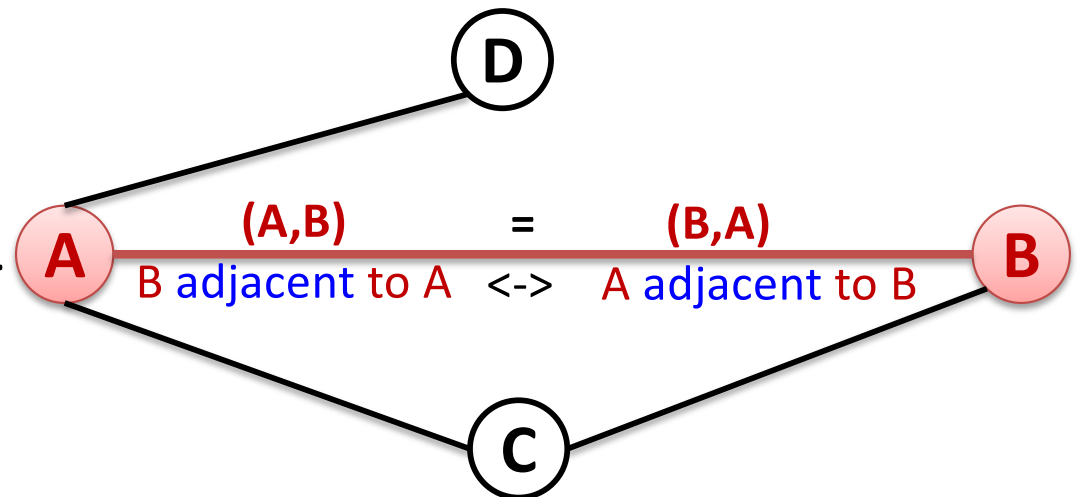
- Undirected graph (ไม่มีทิศทาง)**

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

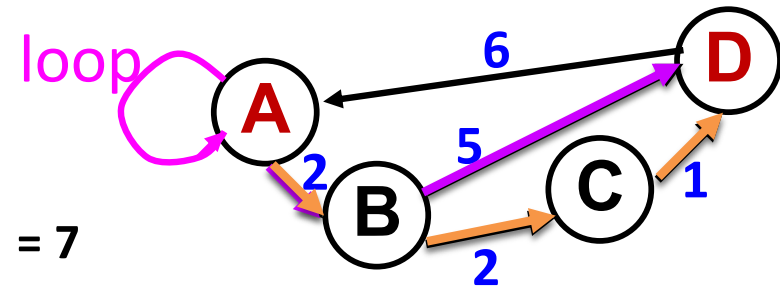
มี 2 paths จาก A ไป D

ABD

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

ABCD

unweighted : path length = 3
weighted : path length = 2 + 2 + 1 = 5

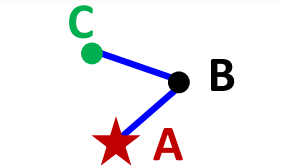
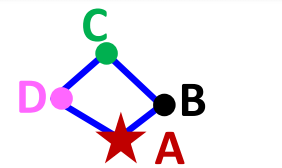
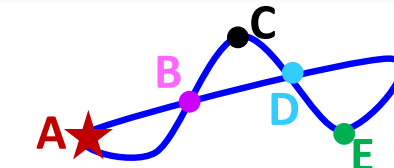
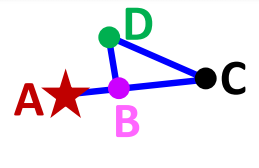


Weighted graph has **weight** assigned to each edge. (graph ที่มีน้ำหนักกำกับ edge)

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 เช่น จาก 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

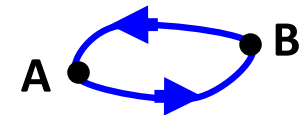
 ABC	 ABCD A	 AB C D E D B A	 AB C D B A
simple path	simple path	non simple path	non simple path
acyclic	simple cycle	cycle	cycle

Path : เส้นมีอนทางเดิน

Simple path: path ซึ่ง vertices ไม่ซ้ำ เว้น vertex แรกกับ vertex สุดท้ายซ้ำได้
ไม่เดินไปที่ๆเดินไปแล้ว (ยกเว้นกลับมาที่เก่า) ผ่าน vertices อย่างน้อย 1 ครั้ง

Cycle graph (circular graph) :

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



Simple Cycle : Simple path + Cycle

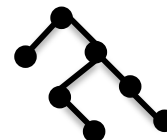
Cycle in undirected graph: edges ต้องไม่ใช่ edge เดียวกัน

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

Acyclic Graph: no cycle



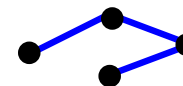
Directed Acyclic Graph = DAG ==> Tree



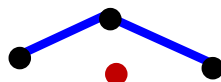
Connected VS Disconnected

- **Undirected graph**

- **Connected** มี path จากทุก vertex ไปยังทุก vertex

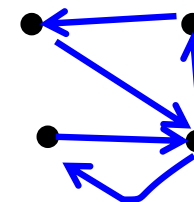


- **Disconnected**



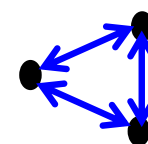
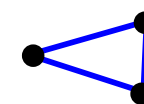
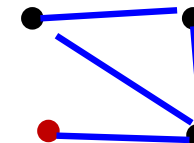
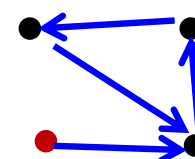
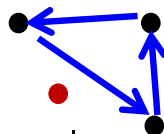
- **Directed graph**

- **Strongly Connected** มี path จากทุก vertex ไปยังทุก vertex



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

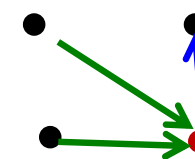
- **Disconnected**



- **Complete graph** มี edge เชื่อมทุกคู่ของ nodes

- **Indegree** จำนวน edges ที่เข้า vertex

- **Outdegree** จำนวน edges ที่ออกจาก vertex

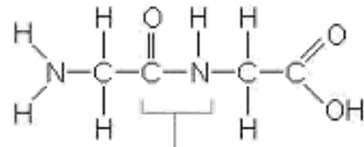
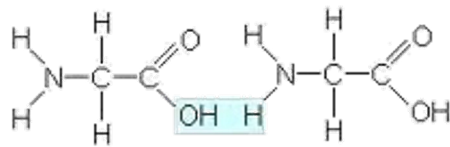
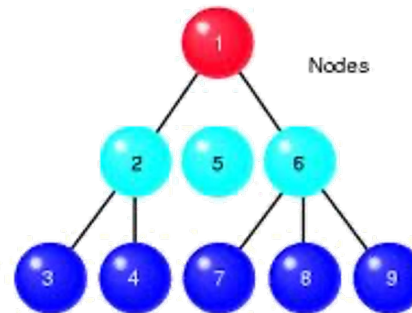
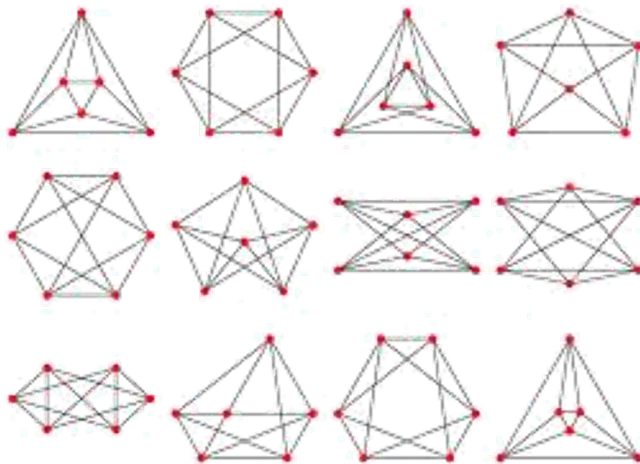


- has **indegree** = 2

- has **outdegree** = 1

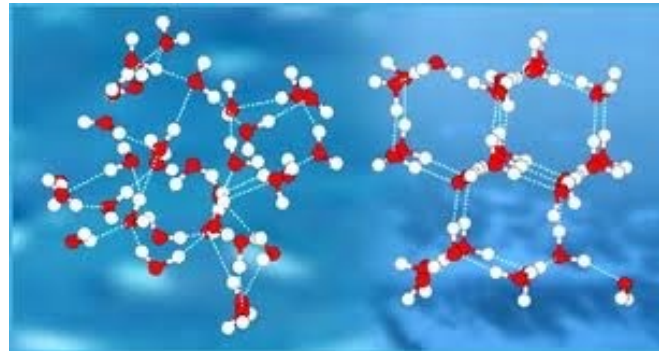
Graph Examples

Airport System, Traffic Flow, ...



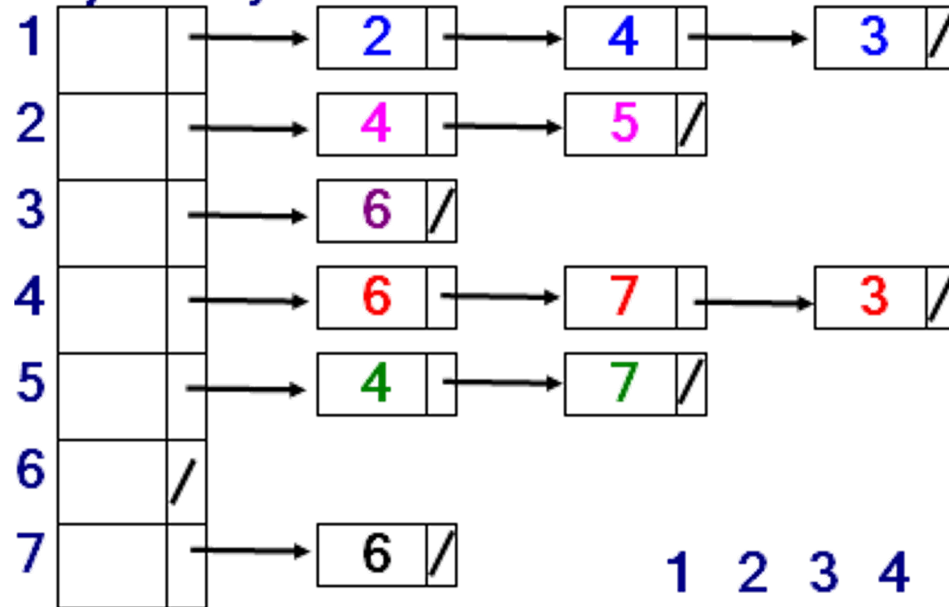
Peptide Bond

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



Graph Edge-Representations

Adjacency list



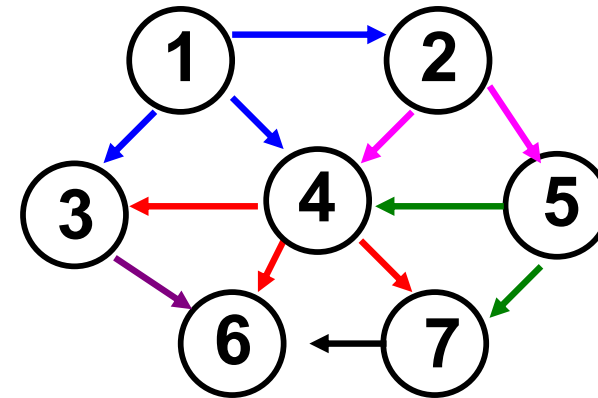
Or

Linked List of
vertices

กรณีที่มีการ

insert/delete

vertex บ่อย



1 2 3 4 5 6 7

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

Adjacency matrix

* simple

* good for dense graph

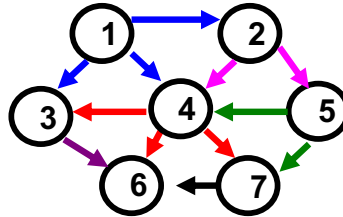
* bad for sparse matrix

ex. street map

* space $\theta(|V|^2)$

* undirected: $(1,3), (3,1) = T$

Graph Node-Representation

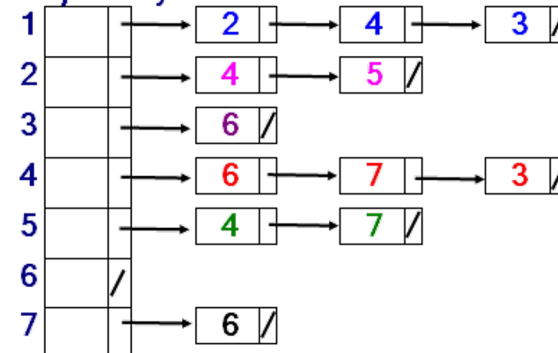


Node-Representation

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

Edge-Representations

Adjacency list



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

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 {node}: {neighbors}')
```

```
# implement adjacency matrix
```

```
num_nodes = 4
```

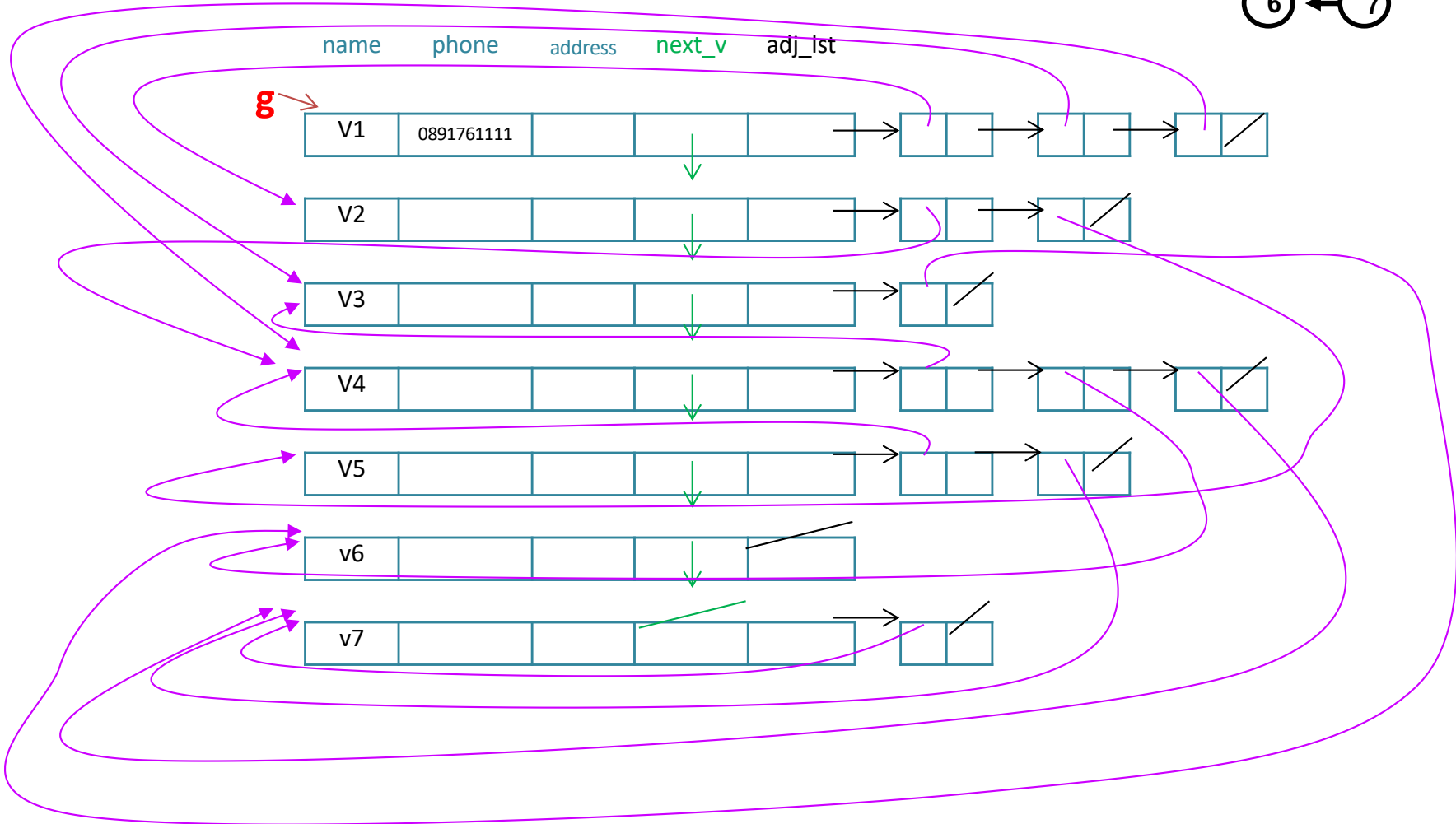
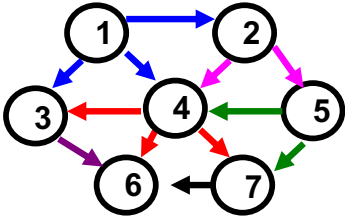
```
adj_matrix = [[1] * num_nodes for _ in range(num_nodes)]
```

```
# show adjacency matrix
```

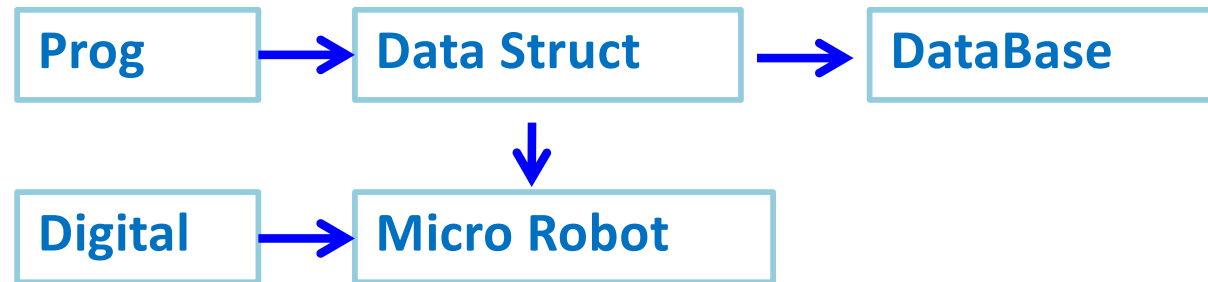
```
for row in adj_matrix:  
    print(row)
```

Graph Representation

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

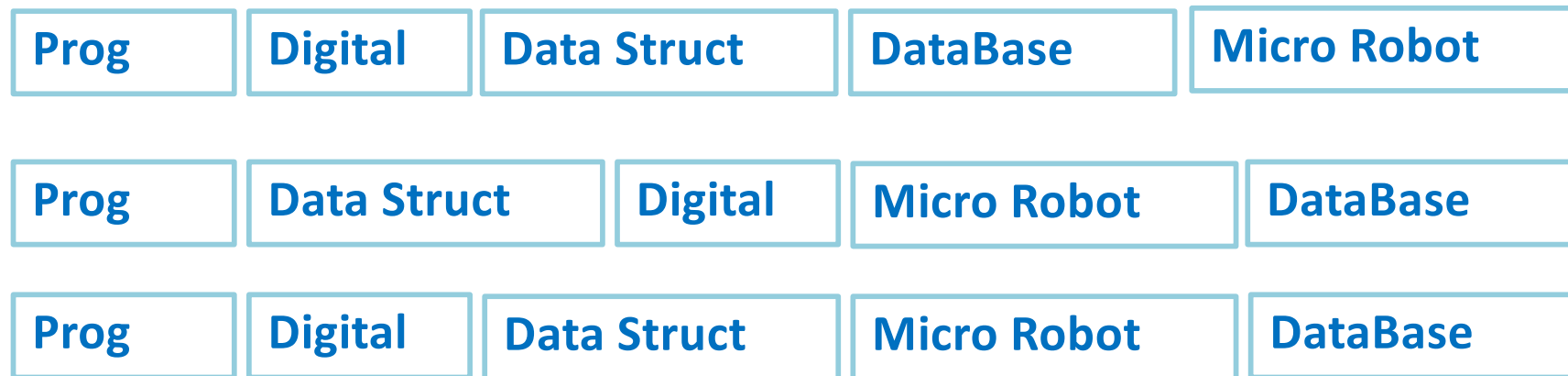


Topological Order



Topological Order : order ใน acyclic graph

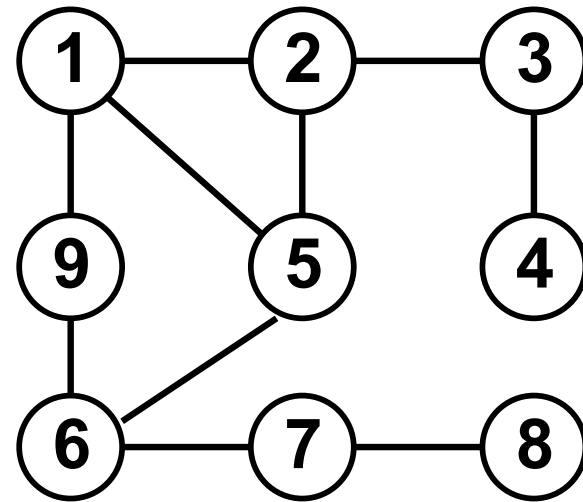
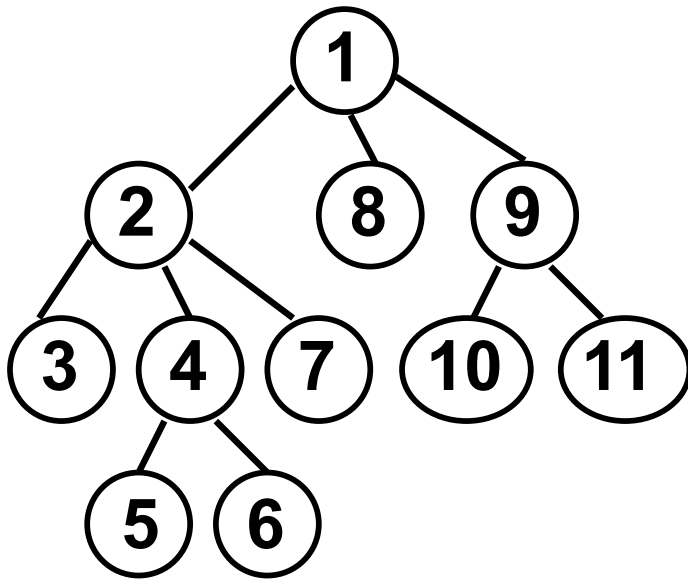
ซึ่ง ถ้ามี path จาก v_i ถึง v_j แล้ว v_j จะต้องอยู่หลัง v_i ใน order
เช่น



Depth First Traversals

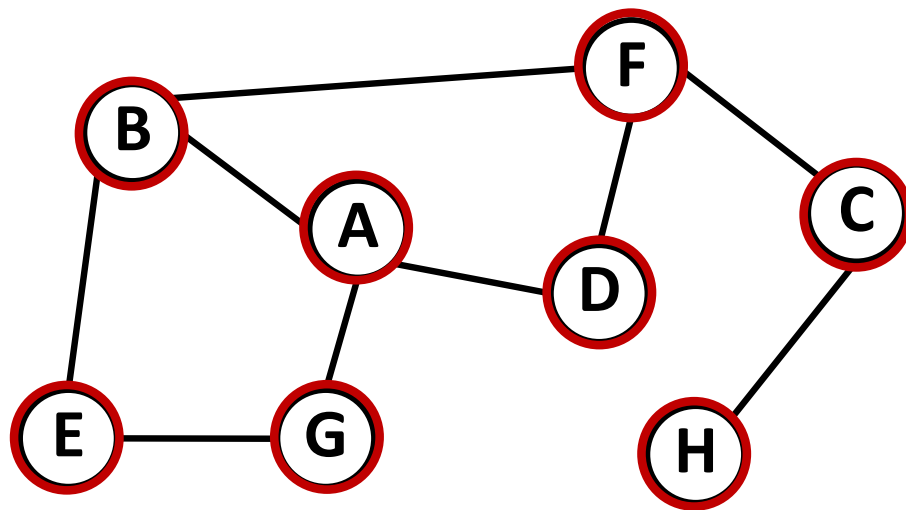
Depth First Traversal

ถ้าเพิ่ง visit V หาก V มี adjacent node ที่ยังไม่ได้ visit ให้ visit ตัวใดตัวหนึ่ง แล้ว ทำอย่างนี้กับ node ที่เพิ่ง visit ไปเรื่อยๆ เมื่อ node ที่เพิ่ง visit ไม่มี adjacent node ที่ยังไม่ได้ visit เหลือแล้ว จึงค่อยกลับมา visit adjacent node ของ node ก่อนหน้าที่ยังเหลืออยู่ depth first traverse จึง**ใช้ stack ช่วย**



Depth First Traversals

Depth First Traversals **ไปด้านลึกก่อน** : ใช้ stack ช่วย



~~D~~
~~H~~
~~C~~
~~F~~
~~G~~
~~E~~
~~B~~
~~A~~

B, G, D ?

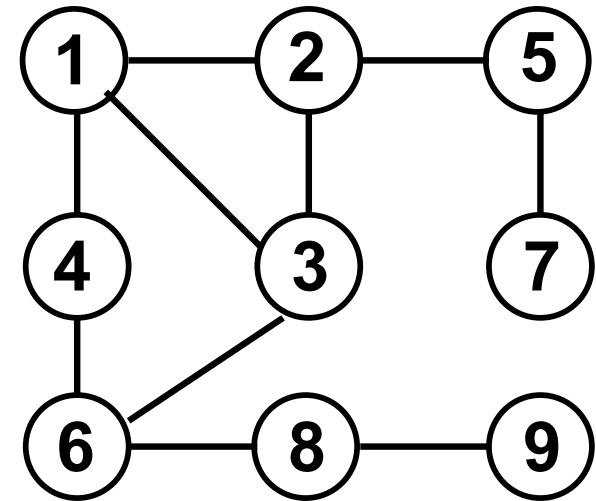
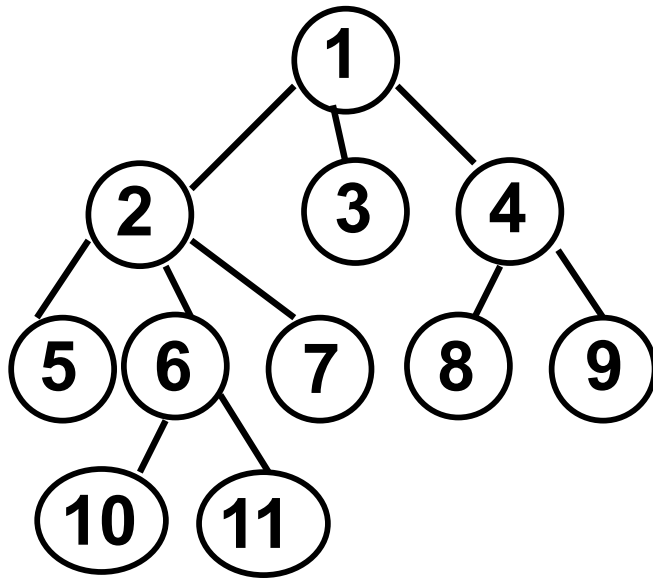
A, B, ... , H ?

ABEGFCHD

Result: **ABEGFCHD**

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

Bredth First Traversals

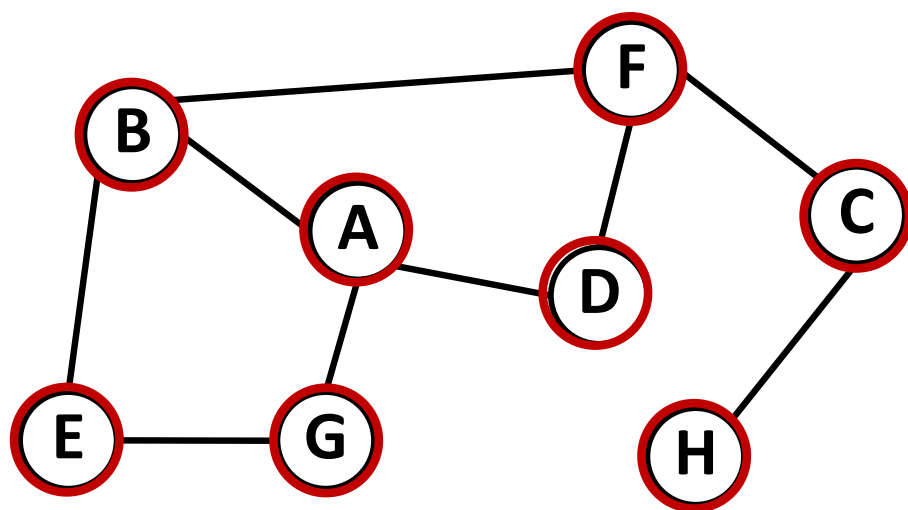


Bredth First Traversal (Level Order)

ถ้าเพิ่ง visit V ถ้า V มี adjacent node ที่ยังไม่ได้ visit ให้ visit ทุกตัวที่ adjacent กับมัน แล้วทำขบวนการนี้ไปเรื่อยๆ กับ node ที่ถูก visit ไปตามลำดับการถูก visit ก่อนหลัง breadth first traverse จึงใช้ queue ช่วย

Bredth First Traversals

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



~~A~~ ~~B~~ ~~D~~ ~~G~~ ~~E~~ ~~F~~ ~~C~~ ~~H~~

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

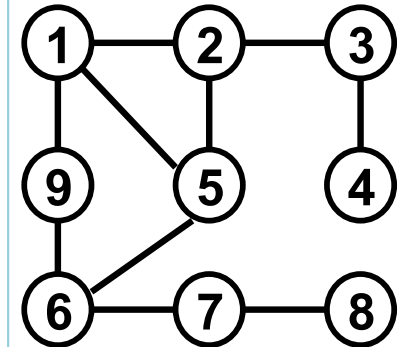
Depth First Traversals

depth_first (vertex)

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

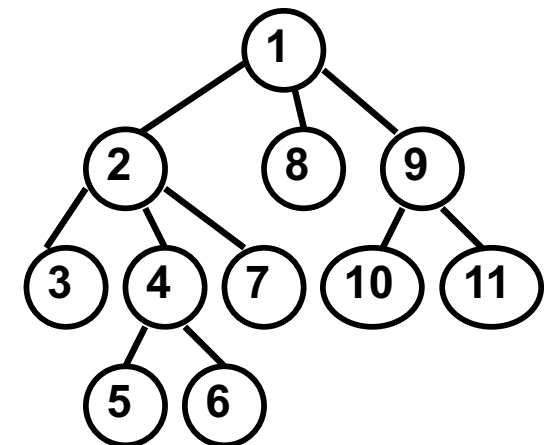
visited	F	F	F	F	F	F	F	F	F	F	...
	0	1	2	3	4	5	6	7	8	9	

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



traverse(**v**, **visited**, vertex)

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**, **vertex**);



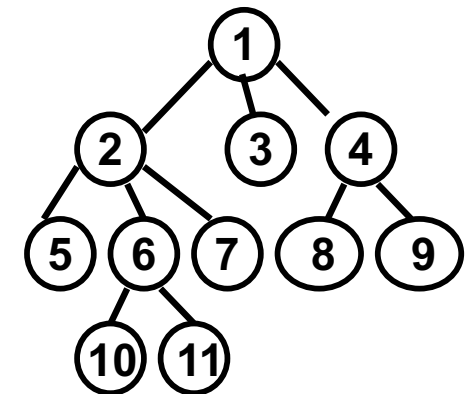
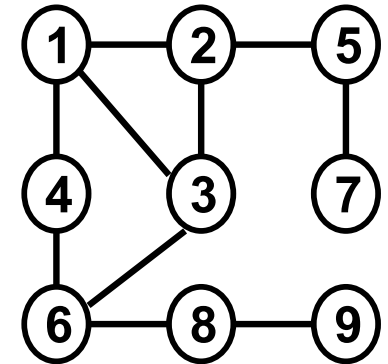
Bredth First Traversals

breath_first (vertex)

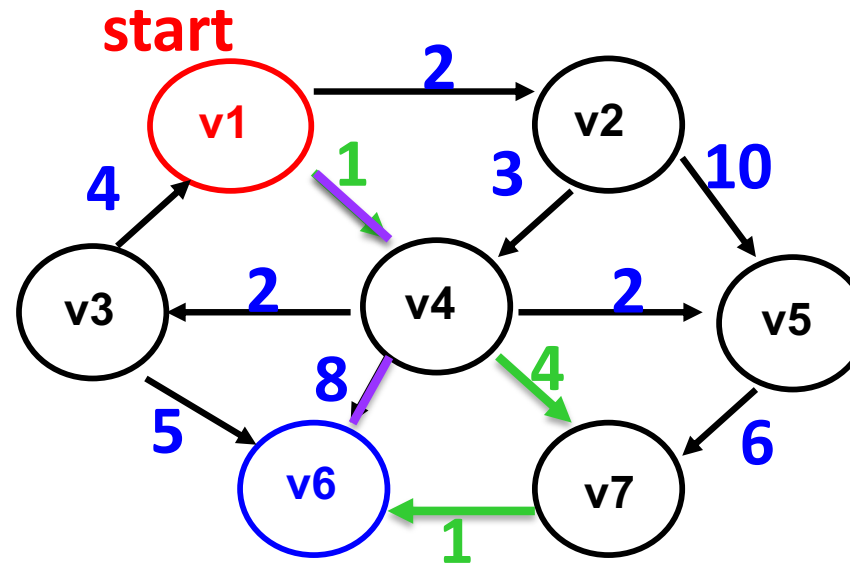
1. init bool **visited** [MAX]; = false for all vertices.

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

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** → 12, 1, 1, 1
 : (optimum → 10, 5)

Weighted Shortest Paths (Dijkstra's algorithm)

Greedy : for each current stage, choose the best.

Data Structures : สำหรับ vertex v ใดๆ เก็บข้อมูล 3 ตัว :

distance = ระยะจากจุด start ไปยัง vertex นั้นๆ

known เป็นจริง เมื่อทราบระยะ distance ที่สั้นที่สุดแล้ว

path = vertex ก่อนหน้านั้นใน 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;

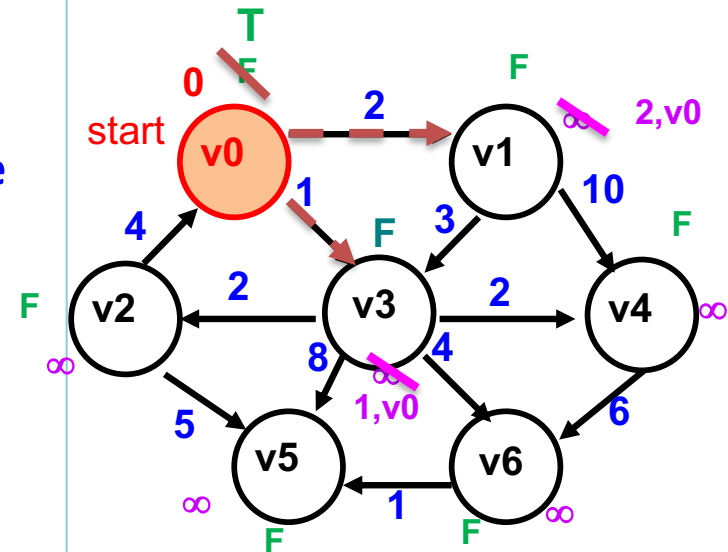
$v.known = true$;

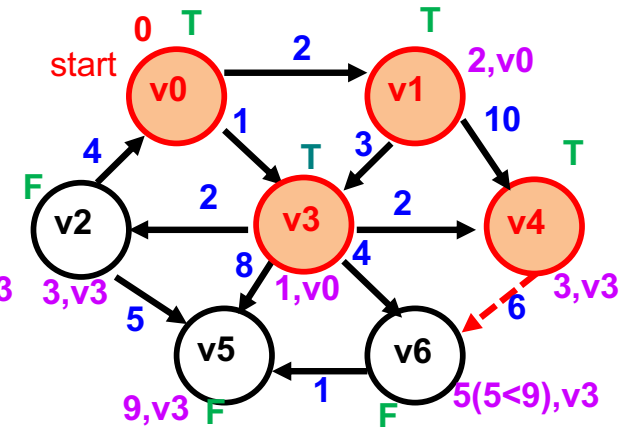
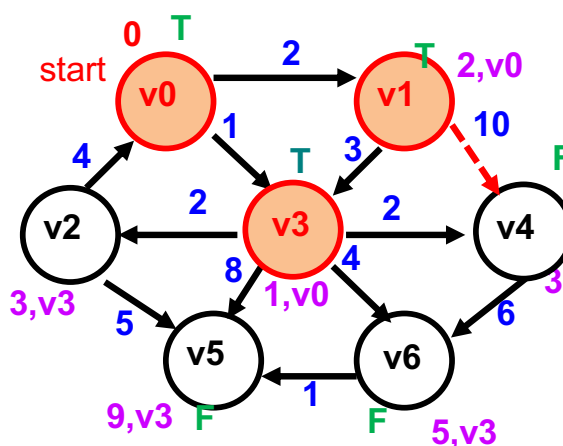
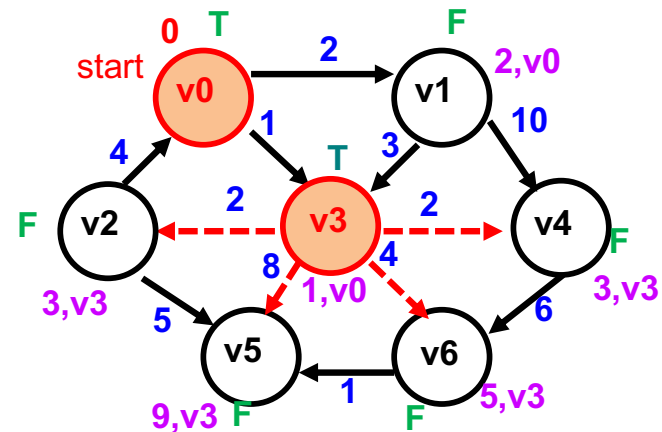
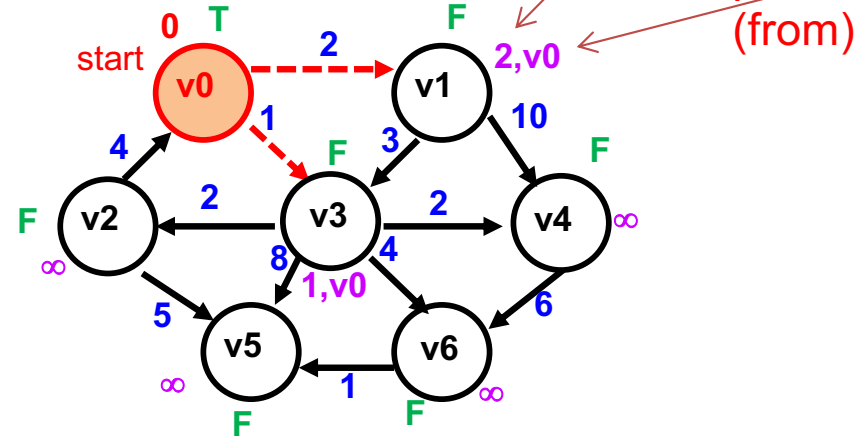
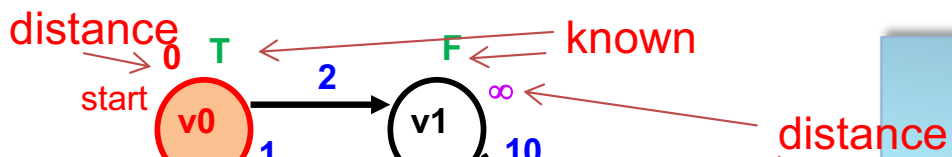
for each w adjacent to v ซึ่งยังไม่ถูก process

if ($w.dist > v.dist + weight(vw)$)

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;

$v.\text{known} = \text{true}$;

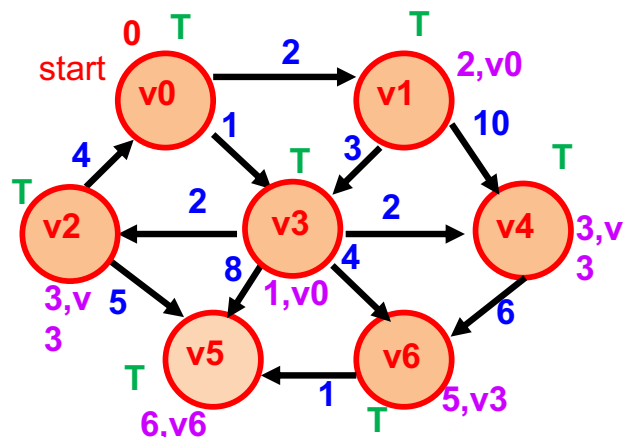
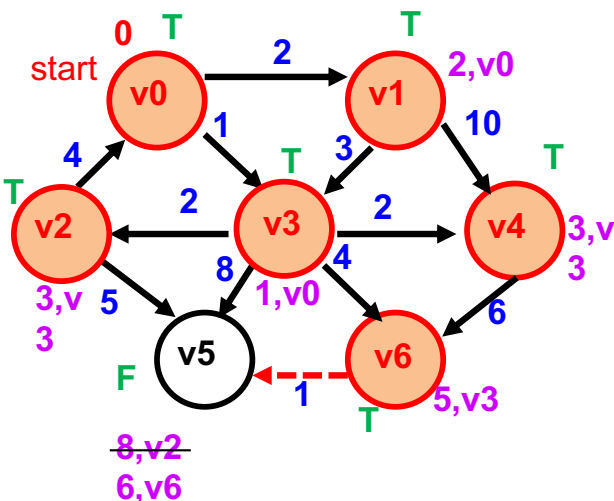
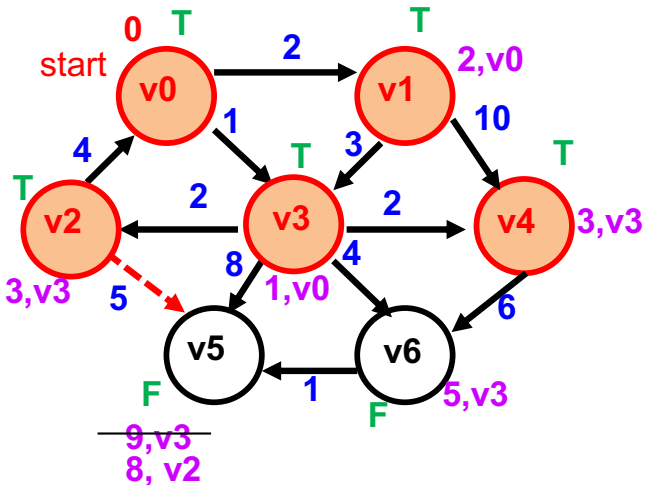
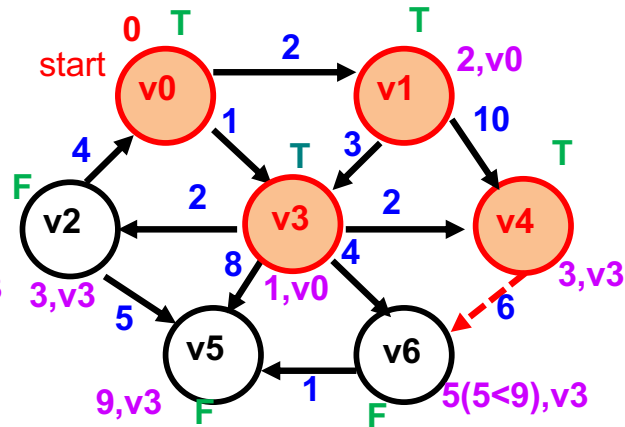
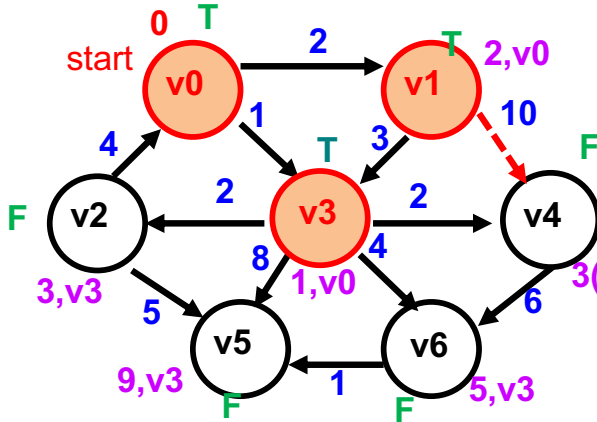
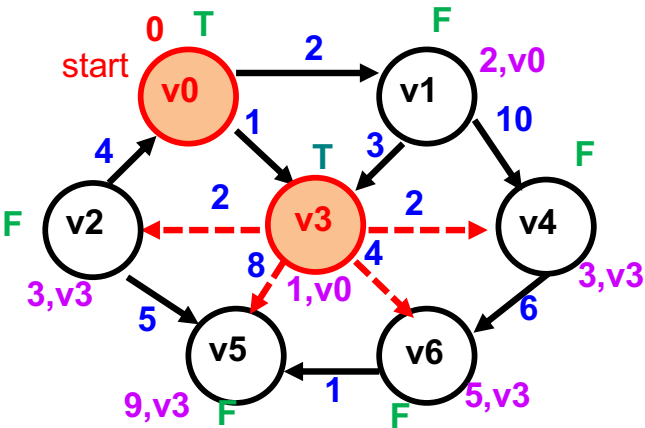
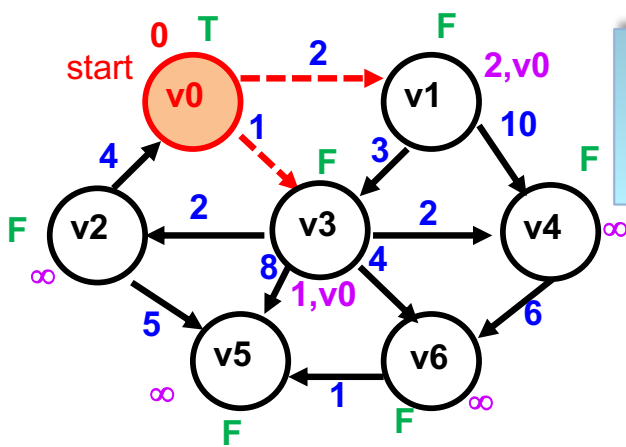
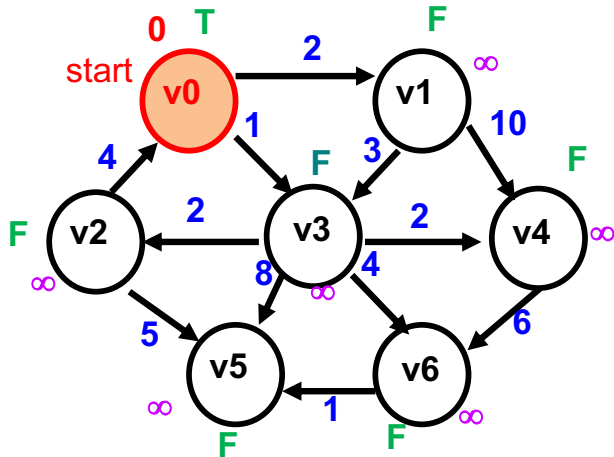
for each w adjacent to v ซึ่งยังไม่ถูก process

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

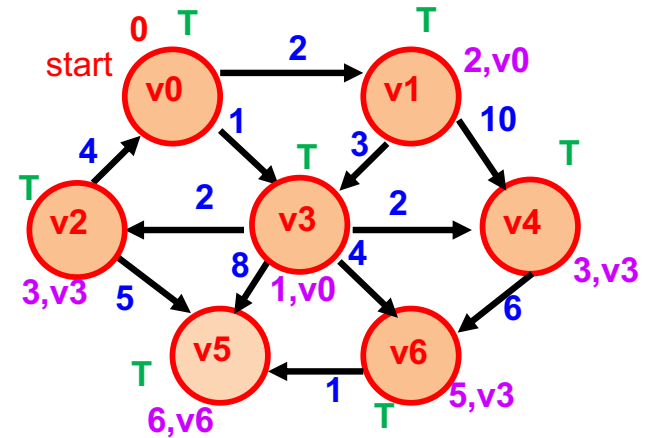
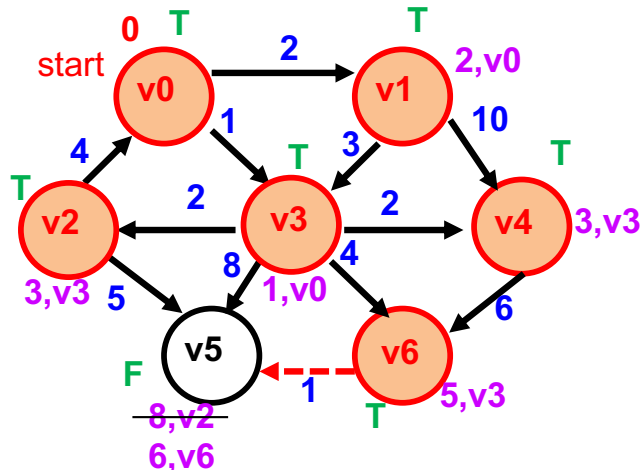
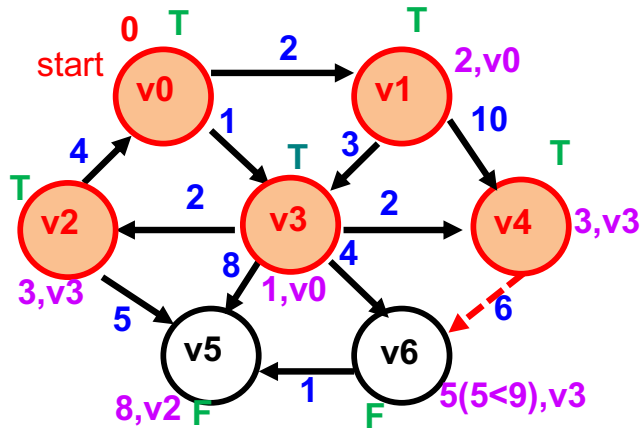
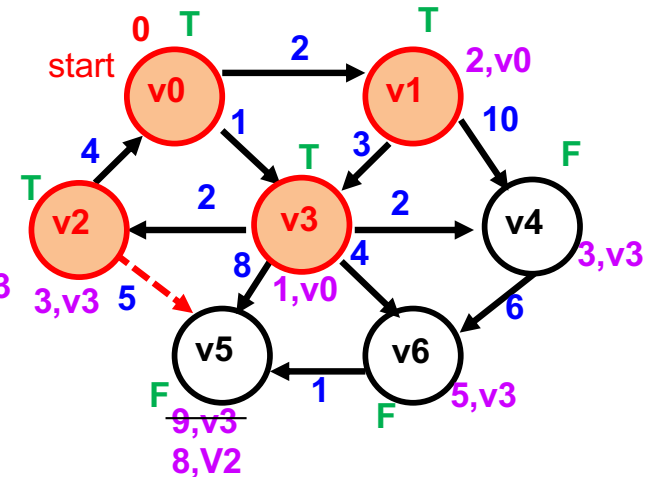
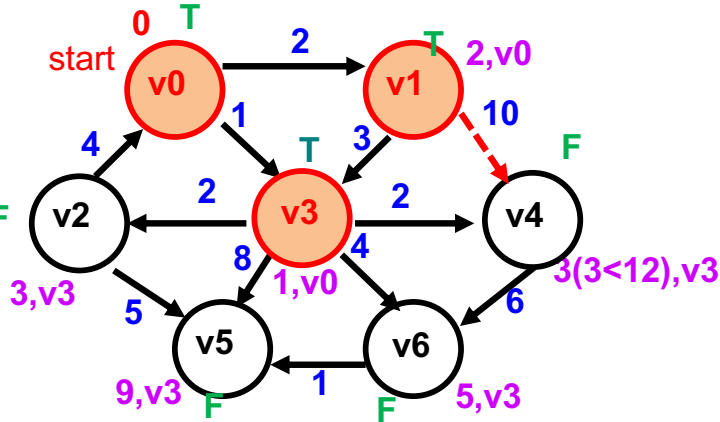
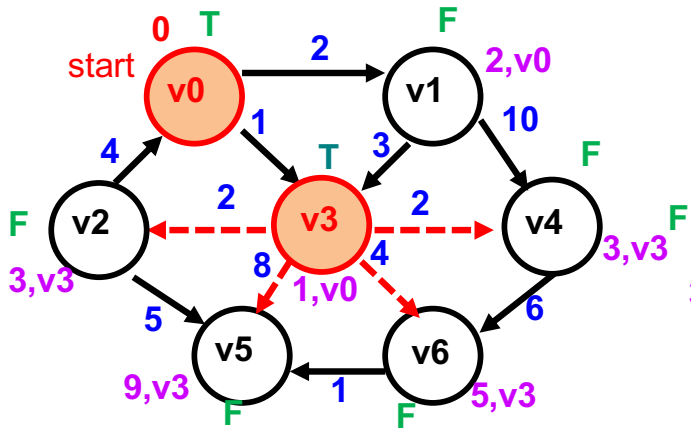
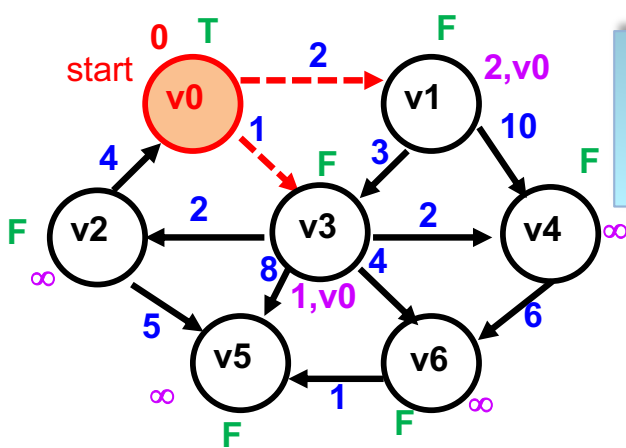
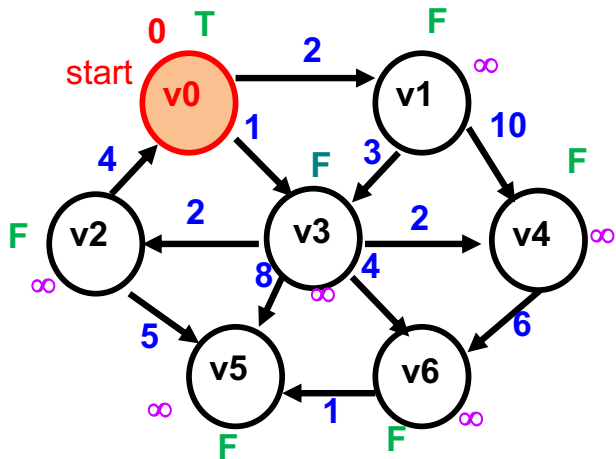
ปรับ $w.\text{dist}$ เป็นค่าใหม่ซึ่งน้อยกว่า

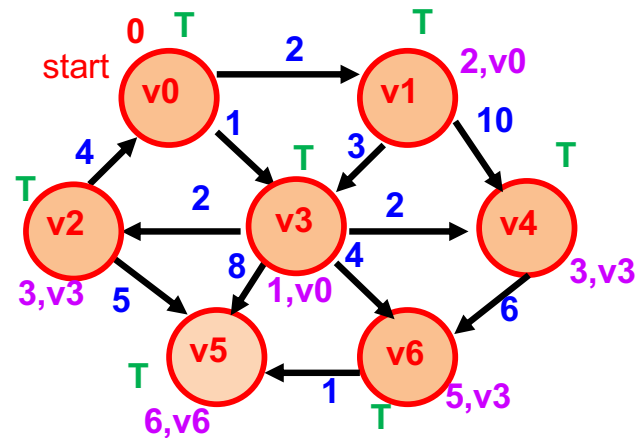
$w.\text{path} = v$;

Weighted Shortest Paths (Dijkstra's algorithm)



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 v5 //เขียน vertex เริ่มต้น v0 และ vertex ปลายทาง v5
 - v0 v6 v5 //vertex ปลายทาง v5 มาจาก path v6
 - v0 v3 v6 v5 //vertex v6 มาจาก path v3
 - v0 v3 v6 v5 //vertex v3 มาจาก path v0
 - v0 $\xrightarrow{1}$ v3 $\xrightarrow{4}$ v6 $\xrightarrow{1}$ v5 path length = 6 //vertex v0 มาถึง v5 มี shortest path ดังนี้
 - ต้องการหา shortest path จาก start (ในที่นี้คือ v0) ไปยัง v4
 - v0 v4 //เขียน vertex เริ่มต้น v0 และ vertex ปลายทาง v4
 - v0 v3 v4 //vertex ปลายทาง v4 มาจาก path v3
 - v0 $\xrightarrow{1}$ v3 $\xrightarrow{2}$ v4 path length = 1+2 = 3 //v3 มาจาก v0: shortest path จึงเป็นดังนี้
 - shortest path จาก v0 ไปยัง v6
 - v0 v6
 - v0 $\xrightarrow{1}$ v3 $\xrightarrow{4}$ v6 //จะเห็นว่าเป็นเส้นทางผ่านของ v0 ไป v5 ในตัวอย่างแรก