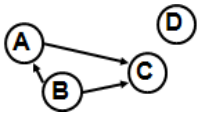
เอกสารนี้มีไว้สำหรับช่วย ใช้ให้ถูกเจตนารม ไม่ใช่ทำให้ไม่ต้องอ่านหนังสือ เพราะนั่นไม่ใช่ช่วย แต่เป็นการทำลาย

Graph : เซ็ทของ vertices (nodes) และ edges (arcs, lines, arrows)

G = (V,E) : **V** = set of vertices, **E** = set of edges

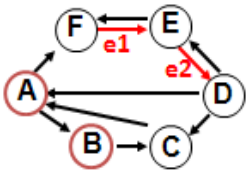


V = { A, B, C, D }

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

Undirected graph : กราฟที่ edges ไม่มีทิศ

Order pairs (A,B) = (B,A)

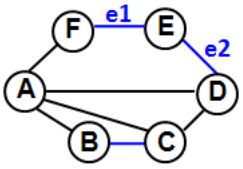


B **adjacent** to A ถ้ามี edge (A,B) **Є** E

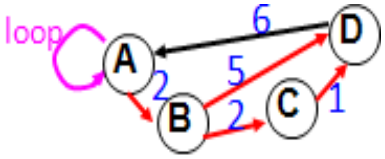
- ดังนั้นสำหรับ undirected graph : B adjacent to A ↔ A adjacent to B

Directed graph (Digraph) : กราฟที่ edges มีทิศ

Order pairs (F,E) ≠ (E,F)



Weighted graph : กราฟที่มี weight กำกับ edge, weight อาจเป็นอะไรก็ได้ที่เราสนใจ เช่น ระยะทาง เวลา ความยาว ปริมาณ ความจุ ...



Path เส้นทางจาก node A ไป B คือ sequence ของ nodes : A,n1,n2,...,ni,B เมื่อ มี edges (A,n), (n1,n2), ..., (ni,B)

Path length: จำนวน edge ใน path

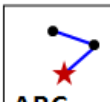
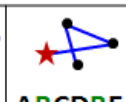
Loop : path of length 0 จาก V to V (คิดว่ามี edge (v,v))

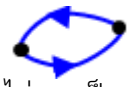
- 2 paths from **A to D** : ABCD and ABD
- ABD length = 2 unweigthed
- ABCD length = 3 unweigthed
- ABCD length = 2 + 2 + 1 = 5 weigthed

Simple path : path ซึ่ง vertices ไม่ซ้ำ ยกเว้น vertex แรก กับ vertex สุดท้ายซ้ำได้

Cycle in digraph : path ซึ่งวนกลับมาที่เดิม start vertex = end vertex และมี path length อย่างน้อย 1

Simple cycle : cycle of simple path

 ABC	 ABCA	 ABCDEDBA	 ABCDBE
simple path	simple path	non simple path	non simple path
acyclic	simple cycle	cycle	



Cycle in undirected graph : edges ต้องไม่ใช่ edge เดียวกัน ie. path UVU ไม่ควรเป็น cycle เพราะ (U,V) = (V,U)

Acyclic Graph : no cycle

Directed Acyclic Graph = **DAG** ==> Tree

Undirected graph : **connected** มี path จากทุก vertex ไปยังทุก vertex

Undirected graph : **Disconnected**

Directed graph : **Strongly Connected** มี path จากทุก vertex ไปยังทุก vertex

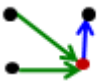
Directed graph : **Weakly Connected** ถ้าไม่นับลูกหัวลูกศรแล้วเชื่อมกัน

Directed graph : **Disconnected**

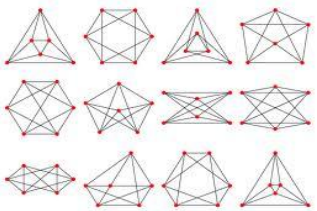
Complete graph มี edge เชื่อมทุกคู่ของ vertices

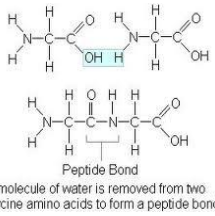
Indegree จำนวน edges ที่เข้า vertex

Outdegree จำนวน edges ที่ออกจาก vertex



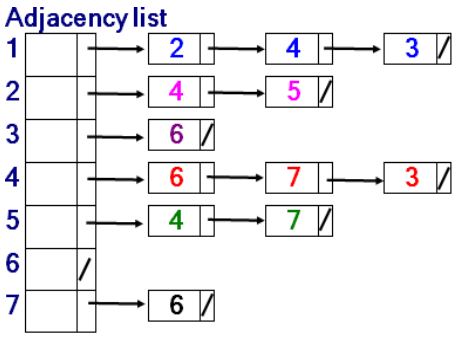
- has indegree = 2
- has outdegree = 1

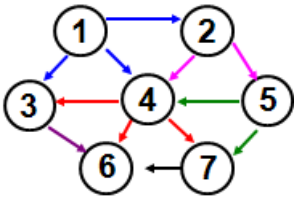


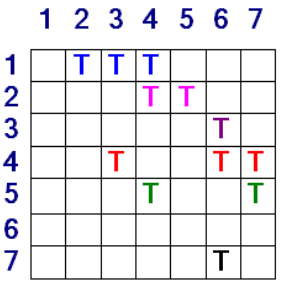


อ.กฤตวัน ศิริบูรณ์

Graph Representations



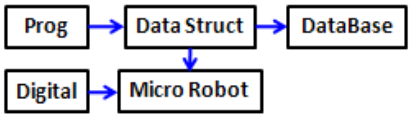




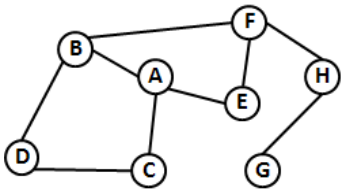
* simple
* good for dense graph
* bad for sparse matrix
ex. street map
* space ($\theta|v^2|$)
* undirected: (1,3) , (3,1) = T
* vertex info : hash table of info and ptr to vertex obj

Topological Order : order ใน acyclic graph ซึ่ง ถ้ามี path จาก v_i ถึง v_j แล้ว v_j จะต้องอยู่หลัง v_i ใน order เช่น

1. Prog | Data Struct | Digital | DataBase | Micro Robot
2. Prog | Digital | Data Struct | DataBase | Micro Robot
3. Prog | Digital | Data Struct | Micro Robot | DataBase



Depth First Traversal : ถ้าพึ่ง visit V หาก V มี adjacent node ที่ยังไม่ได้ visit ให้ visit ตัวใดตัวหนึ่ง แล้วทำอย่างนี้กับ 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

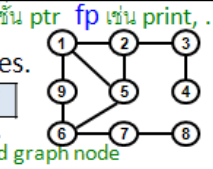


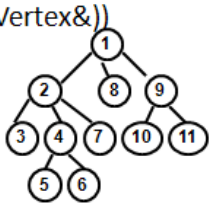
Result :

```
depth_first ( void (*fp)( Vertex& ) ) const // ฟังก์ชัน ptr fp เช่น print, .
1. init bool visited [MAX]; = false for all vertices.
2. for all un-visited vertex v // run for disconnected graph node
       traverse (v, visited, fp); // เช่น fp = &print(Vertex&);

traverse(Vertex& v, bool visited[ ],void(*fp)(Vertex&))
   const
1. (*fp)(v) ;    // run ฟังก์ชั่น fp on v
2. visited[v] = true; // set v to be already visited
3. for all un-visited w that adjacent to v
       traverse (w, visited, fp);
```

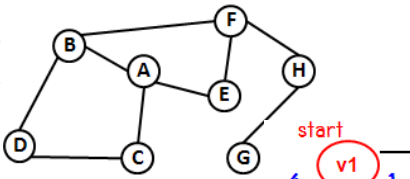
visited: F F F F F F F F F F F (0 1 2 3 4 5 6 7 8 9 ...)





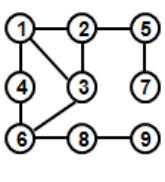
Bredth First Traversal : ถ้าพึ่ง visit V ถ้า V มี adjacent node ที่ยังไม่ได้ visit ให้ visit ทุกตัวที่ adjacent กับมัน แล้วทำขบวนการนี้ไปเรื่อยๆ กับ node ที่ถูก visit ไปตามลำดับการถูก visit ก่อนหลัง bredth first traverse จึงใช้ queue ช่วย

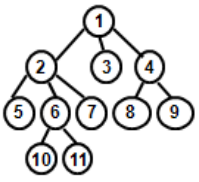
Result :

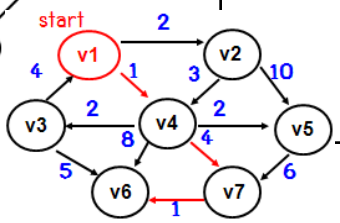



```
bredth_first ( void (*fp)( Vertex& ) ) const // ฟังก์ชัน ptr fp
                                              เช่น print, ...
1. init bool visited [MAX]; = false for all vertices.
2. empty queue q;
3. for all un-visited vertex v // run for disconnected graph node
     q = enqueue(v)
     while (not empty q)
       w = q.dequeue()
       if (!visited[w])
         visited[v] = true; // set v to be already visited
         (*fp) (w);
         for all un-visited x that adjacent w and x is not in q
             q.enqueue(x)
```

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





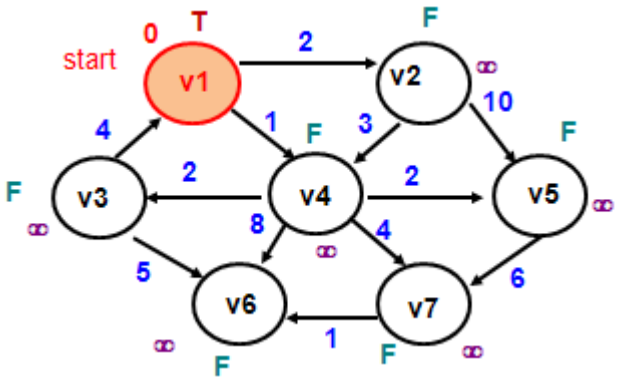


Shortest Path :

- 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 : เลือกอันที่ดีที่สุดสำหรับ stage ปัจจุบัน แย่ตรงที่อาจไม่ optimum

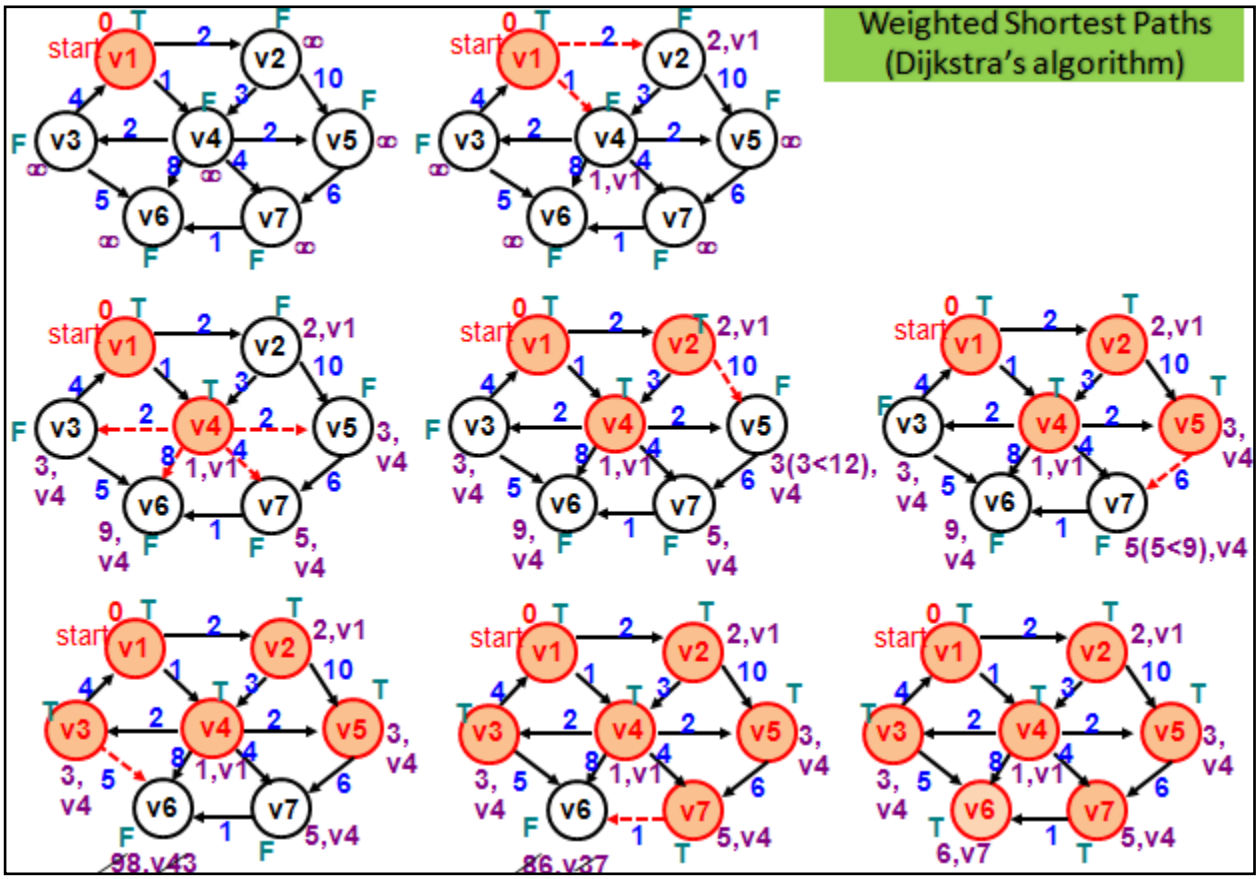
ตย. แลกเหรียญให้ได้จำนวนเหรียญน้อยที่สุด
quarter 25 cents
dime 10 cents
12 cents (สมมุติว่ามี)
nikle 5 cents
penny 1 cents
15 cents : greddy → 12, 1, 1, 1
(optimum : 10, 5)



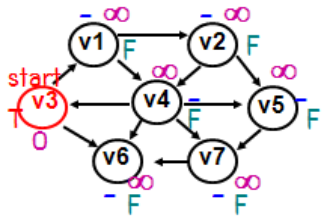
Weighted Shortest Paths (Dijkstra's algorithm)

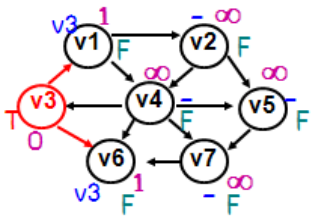
```
s.dist = 0;
for( ; ;)
    v = smallest unknown dist. vertex
    if (v ไม่มี)
        break;
    v.known = true;
    for each w adjacent to v ซึ่งยังไม่ถูก process
        if (w.dist > v.dist + weight(vw))
            ปรับ w.dist เป็นค่าใหม่ซึ่งน้อยกว่า
            w.path = v;
```

Greedy: for each current stage, choose the best.



Unweighted Shortest Paths



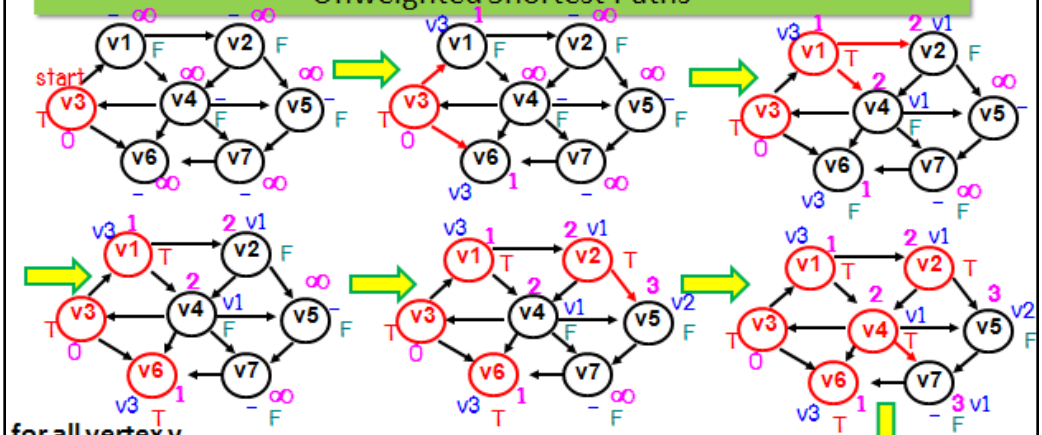


v	known	distance	path
1	F	1	v3
2	F	∞	-
3	T	0	-
4	F	∞	-
5	F	∞	-
6	F	1	v3
7	F	∞	-

v	known	distance	path
1	F	∞	-
2	F	∞	-
3	T	0	-
4	F	∞	-
5	F	∞	-
6	F	∞	-
7	F	∞	-

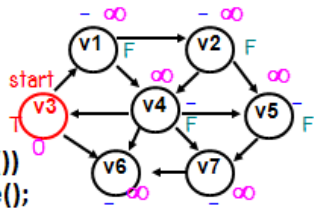
```
for all vertex v
    v.dist = ∞;     v.known = F;     v.path = "-";
s.dist = 0;   //s=start vertex
for (currDist =0; currDist < NUMVERTICES; currDist ++)
    for each vertex v that !v.known && v.dist = currDist
        v.known = T;   //process แล้ว
        for each w ที่ adjacent กับ v และ w.dist = ∞
                w.dist = currentdist +1;
                w.path = v;
```

Unweighted Shortest Paths




```
for all vertex v
    v.dist = ∞;     v.known = F;     v.path = "-";
s.dist = 0;   //s=start vertex
for (currDist =0; currDist < NUMVERTICES; currDist ++)
    for each vertex v that !v.known && v.dist = currDist
        v.known = T;   //process แล้ว
        for each w ที่ adjacent กับ v และ w.dist = ∞
                w.dist = currentdist +1;
                w.path = v;
```

Unweighted Shortest Paths (using queue)




```
q.enqueue(s);
s.dist = 0;
while(!q.empty())
  v = q.dequeue();
  v.known = true;
  for each w adjacent to v & w.dis== infinity
      w.dist = w.dist +1;
      w.path = v;
      q.enqueue(w);
```

	Initial			v3 dequeued			v1 dequeued		
v	known	dis	p	known	dis	p	known	dis	p
1	F	∞	-	F	1	v3	T	1	v3
2	F	∞	-	F	∞	-	F	2	v1
3	F	0	-	T	0	-	T	0	-
4	F	∞	-	F	∞	-	F	2	v1
5	F	∞	-	F	∞	-	F	∞	-
6	F	∞	-	F	1	v3	F	1	v3
7	F	∞	-	F	∞	-	F	∞	-
Q:	v3			v3v1v6			v6v2v4		

V3

	v6 dequeued			v2 dequeued			v4 dequeued			v5 dequeued			v7 dequeued		
v	known	dis	p	known	dis	p	known	dis	p	known	dis	p	known	dis	p
1	T	1	v3	T	1	v3	T	1	v3	T	1	v3	T	1	v3
2	F	2	v1	T	2	v1	T	2	v1	T	2	v1	T	2	v1
3	T	0	-	T	0	-	T	0	-	T	0	-	T	0	-
4	F	2	v1	F	2	v1	T	2	v1	T	2	v1	T	2	v1
5	F	∞	-	F	3	v2	F	3	v2	T	3	v2	T	3	v2
6	T	1	v3	T	1	v3	T	1	v3	T	1	v3	T	1	v3
7	F	∞	-	F	∞	-	F	3	v4	F	3	v4	T	3	v4
Q:	v2v4			v4v5			v5v7			v7			empty		