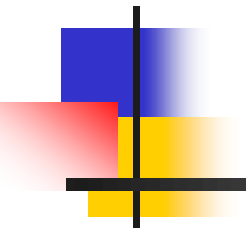
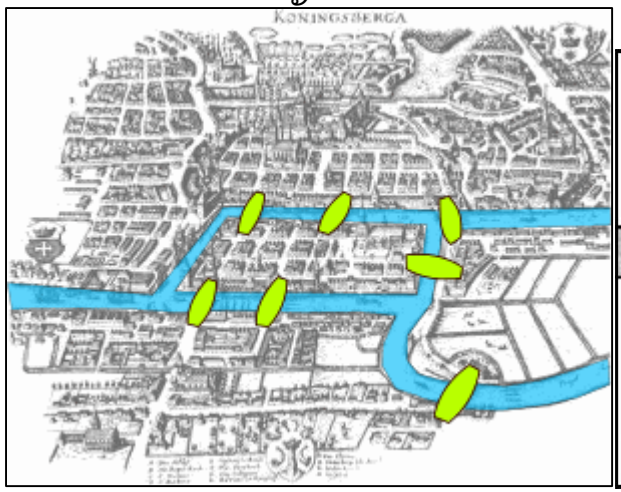


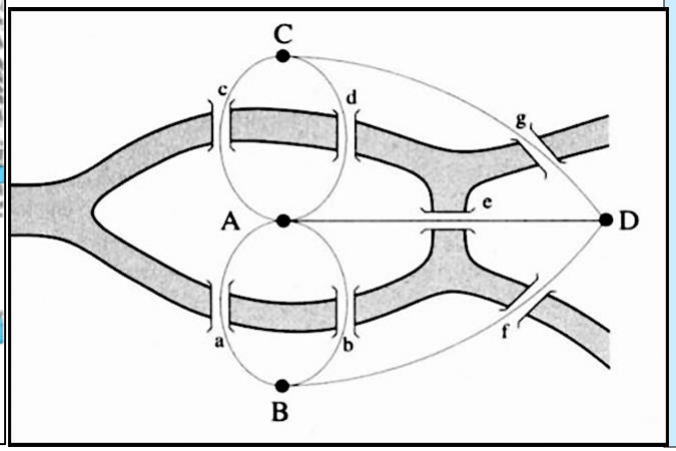
Chapter 9

Graph

Graph

- เป็นโครงสร้างความสัมพันธ์ของข้อมูลที่มีการเชื่อมโยงแบบหลายทิศทาง(n:n) ซึ่งต่างจากลิงค์ลิสต์(1:1) หรือ ต้นไม้ (1:n)
- กราฟประกอบด้วยเซ็ตของโหนด(Nodes หรือ Vertices) ที่เชื่อมต่อกันด้วยเซ็ตของเอดจ์(Edges หรือ Links) ซึ่งแสดงความสัมพันธ์ระหว่างคู่ของโหนด
- เอดจ์ หรือเส้นเชื่อม ที่แสดงความสัมพันธ์ระหว่างคู่ของโหนด อาจกำหนดให้มีทิศทางหรือไม่ก็ได้ ถ้ากำหนดให้มีทิศทางเรียกว่าDirected Graphs ถ้าไม่มีทิศทางเรียกว่า Undirected Graphs หรือถ้ามีทั้ง 2 อย่าง เรียกว่า Mixed Graphs
- กราฟที่กำหนดค่าน้ำหนักของเอดจ์(ความสัมพันธ์) เรียกว่า Weighted Graphs
- การทราเวิร์ส(Traverse) ในกราฟ คือการเข้าถึงแต่ละโหนดผ่านทางเอดจ์ ที่เชื่อมอยู่
- ปัญหาแรกของกราฟ





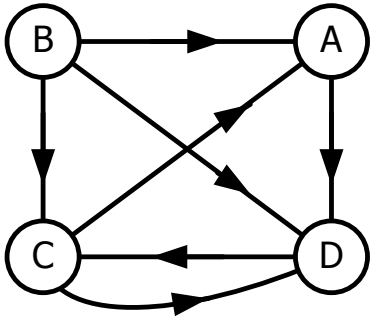
สะพานทั้งเจ็ดแห่งเมืองเคอนิกสแบร์ก (Seven Bridges of Königsberg) เมืองเคอนิกสแบร์ก มีเกาะอยู่ 2 เกาะ กลางแม่น้ำเพรเกิล เชื่อมต่อถึงกันด้วย สะพาน 7 สะพาน คำถามคือ เป็นไปได้หรือไม่ที่จะเดินผ่านให้ครบทุกสะพาน โดยผ่านแต่ละสะพานเพียงครั้งเดียว และกลับมาที่จุดเริ่มต้นได้ เลออนฮาร์ด ออยเลอร์ ได้ตีพิมพ์บทความพิสูจน์ให้เห็นว่าเป็นไปไม่ได้

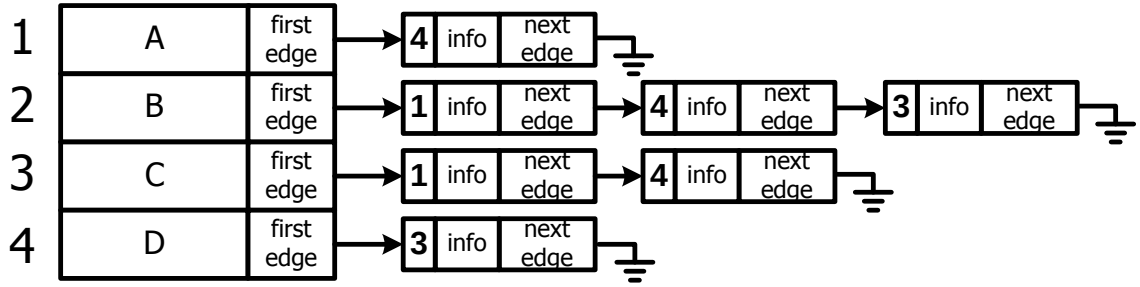
- The paper written by Leonhard Euler on the *Seven Bridges of Königsberg* and published in 1736 is regarded as the first paper in the history of graph theory.

Graph & Mixed Lists

- ตัวอย่างการออกแบบโครงสร้างของกราฟโดยใช้ array of linked lists

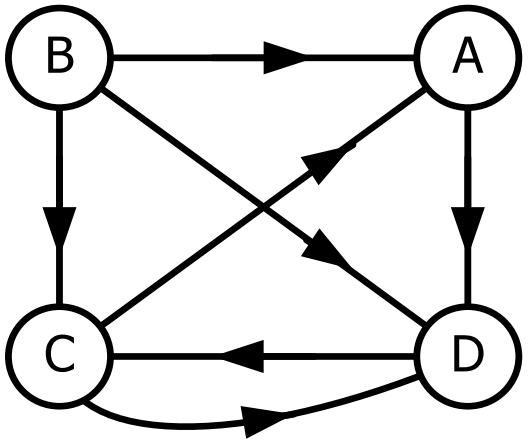
```
class   Edge { int edgeInfo;
               int adjNode; //index
               Edge  next_edge;
}
class  Vertex { String label;
                 int status ;
                Edge  first_edge;
}
class GraphArray {
      Vertex [] node ;
      public GraphArray(int max) {
            node = new Vertex[max];
      }
}
```





Graph & Matrix (2D-Array)

- ตัวอย่างการออกแบบโครงสร้างกราฟโดยใช้ Matrices




```
class Vertex { String label ;
                int status ;  // status of traverse
        public Vertex(String label ) {
                .............
        }
}
class Graph { Vertex [] node; // set of nodes
                int[][] adjMat; // adjacency of nodes
                int vCount ;
        public Graph(int max) {
                ........................
        }
 }
```

vCount = 4

node[0]	A
node[1]	B
node[2]	C
node[3]	D

AdjMat =

	[0]	[1]	[2]	[3]
[0]	0	0	0	1
[1]	1	0	1	1
[2]	1	0	0	1
[3]	0	0	1	0

Graph & Matrix Array

```
class Vertex { String label ;
               int status ;  //status of traverse
      public Vertex(String name)  {
               label = name;
               status = 0 ;
      }
}
class Graph { Vertex []  node ;
      static   int  [][] adjMat;
      static   int vCount = 0 ;
      public Graph(int max) {
            node = new Vertex[max];
            adjMat = new int[max][max];
            vCount = 0;
            for (int i=0; i<max; i++)
                for (int j=0; j<max; j++)
                   adjMat[i][j] = 0; // no link
      }
      void addNode(String name) {
            node[vCount++]  = new Vertex (name) ;
      }
      void addEdge(int fromNode, int toNode) {
            adjMat[fromNode][toNode] = 1;     // directed & undirected graph
            adjMat[toNode][fromNode] = 1;     // add for undirected graph
      }
}
```

Breadth-first Algorithm

กำหนดสถานะของโหนด 1 = ยังไม่เคยเห็น, 2 = อยู่ในคิว 3 = ทำเสร็จแล้ว

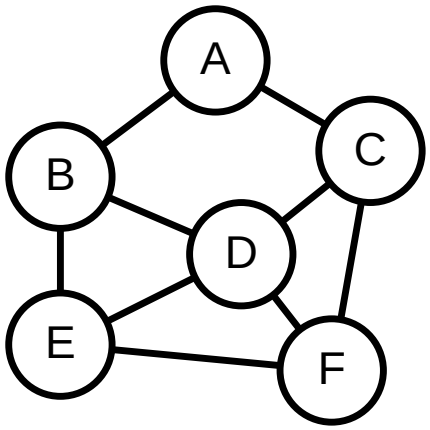
1. ทำให้ทุกโหนดมีสถานะเป็น 1 (โหนดใหม่ยังไม่เคยเห็น) กำหนด start node เริ่มต้น

2. ใส่ start node ไว้ในคิว แล้วเปลี่ยนสถานะเป็น 2 (อยู่ในคิว) // addLast

3. ดึงโหนดออกจากคิวไปใช้ และเปลี่ยนสถานะเป็น 3 (ผ่านการ Access) //removeFirst

4. ใส่ทุกโหนดที่ต่ออยู่(จาก 3) ไว้ในคิว และเปลี่ยนสถานะเป็น 2 (อยู่ในคิว)

5. ทำซ้ำข้อ 3 - 4 จนกว่าคิวจะว่าง



Operation	Queue		
start Add A	A		
remove A, add B, C	B	C	
remove B, add E, D	C	E	D
remove C, add F	E	D	F
remove E	D	F	
remove D	F		
remove F	null		

The order of process is as follow: A, B, C, E, D and F

Depth-first Algorithm

กำหนดสถานะของโหนด 1 = ยังไม่เคยเห็น, 2 = อยู่ในคิว 3 = ทำเสร็จแล้ว

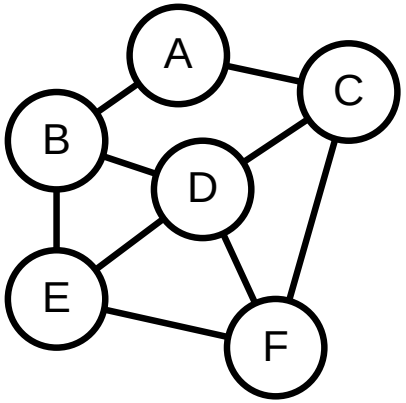
1. ทำให้ทุกโหนดมีสถานะเป็น 1 (โหนดใหม่ยังไม่เคยเห็น) กำหนด start node เริ่มต้น

2. PUSH start node ไว้ในแสต็ก แล้วเปลี่ยนสถานะเป็น 2 (อยู่ในแสต็ก) // addFirst

3. POP โหนดออกจากแสต็กไปใช้ และเปลี่ยนสถานะเป็น 3 (ผ่านการ Access) //removeFirst

4. PUSH ทุกโหนดที่ต่ออยู่(จาก 3) ไว้ในแสต็กและเปลี่ยนสถานะเป็น 2 (อยู่ในแสต็ก)

5. ทำซ้ำข้อ 3 - 4 จนกว่าแสต็กจะว่าง



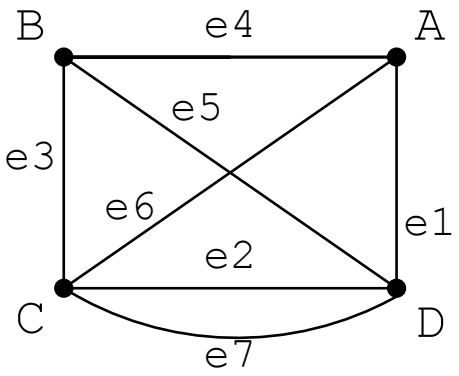
Operation	Stack		
start push A	A		
pop A, push B, C	C	B	
pop C, push D, F	F	D	B
pop F, push E	E	D	B
pop E	D	B	
pop D	B		
pop B	null		

The order of process is as follow: A, C, F, E, D and B

4. Representing Graph with Matrix

- ใช้ incidence matrix หรือใช้ Adjacency matrix เพื่อแสดงความสัมพันธ์ระหว่าง โหนดกับเอดจ์
- Incidence matrix คือ zero-one matrix ที่แสดงความสัมพันธ์ระหว่างโหนดกับเอดจ์ ซึ่งค่าของแต่ละอีลีเมนต์ aij มีค่าเป็น
  - 1 ถ้ามีเส้นทาง(Edge) ej ต่ออยู่กับโหนด vi
  - 0 ถ้าไม่มีเส้นทาง ej ต่อกับโหนด vi

$$A = \begin{array}{c|cccc} & e1 & e2 & \dots & ej \\ v1 & a11 & a12 & & a1j \\ v2 & a21 & & & \\ v3 & a31 & & & \\ \dots & & & & \\ vi & ai1 & ai2 & & aij \end{array}$$

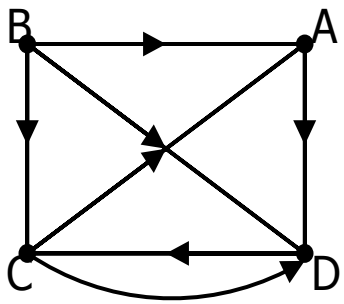


$$M = \begin{array}{c|ccccccc} & e1 & e2 & e3 & e4 & e5 & e6 & e7 \\ A & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ B & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ C & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ D & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{array}$$

Adjacency matrix

- **Adjacency matrix** คือ **zero-one matrix** ที่แสดงความสัมพันธ์ระหว่าง**โหนดกับโหนด** ซึ่งค่าของแต่ละอีลีเมนต์ aij มีค่าเป็น
  - **1** ถ้ามีเส้นทางเดินจาก vi ไปยัง vj
  - **0** ถ้าไม่มีเส้นทางเดิมจาก vi ไปยัง vj

M =		v1	v2	...	vj
	v1	a11	a12		a1j
	v2	a21	a22		
	v3				
	...				
	vi	ai1			aij

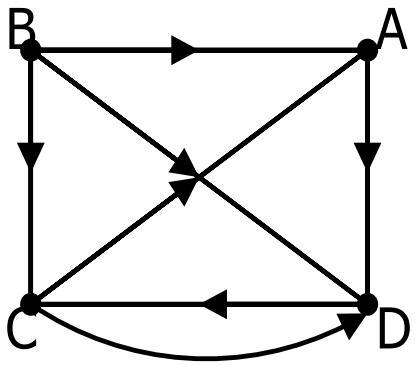


M =

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

Adjacency matrix of directed graph

- find Adjacency matrix of directed graph



M =

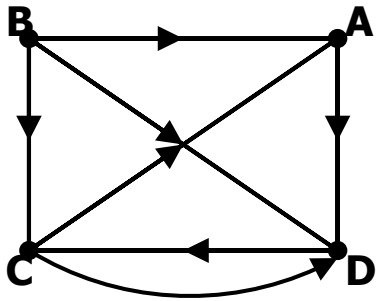
	A	B	C	D
A	0	0	0	1
B	1	0	1	1
C	1	0	0	1
D	0	0	1	0

AA = 0
AB = 0
AC = 0
AD = 1
CA = 1
CB = 0
CC = 0
CD = 1

BA = 1
BB = 0
BC = 1
BD = 1
DA = 0
DB = 0
DC = 1
DD = 0

Adj with Path matrix of length 2 or M^2

- Adjacency matrix ที่มี Path length = 2



M^2 =

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

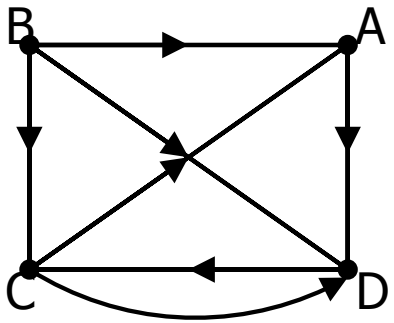
AA = 0
AB = 0
AC = A→D→C = 1
AD = 0
CA = 0
CB = 0
CC = CDC = 1
CD = CAD = 1

BA = B→C→A = 1
BB = 0
BC = B→D →C = 1
BD = B→C→D = 1
DA = D→C→A = 1
DB = 0
DC = 0
DD = DCD = 1

Adjacency Matrix with path length=n-1

M^k is Adjacency Matrix of graph with path length = k
กราฟที่มีจำนวน n nodes จะมี path length ได้ไม่เกิน k = n – 1

เส้นทางที่เดินไปยังโหนดอื่นๆ ได้โดยไม่เดินกลับมาหาตัวเอง ถ้ามีค่าเกินแสดงว่าเดินวนรอบมาที่เดิม



$$M = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 0 & 0 & 1 \\ B & 1 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 \\ D & 0 & 0 & 1 & 0 \end{array}$$

$$M^2 = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 0 & 1 & 0 \\ B & 1 & 0 & 1 & 1 \\ C & 0 & 0 & 1 & 1 \\ D & 1 & 0 & 0 & 1 \end{array}$$

$$M^3 = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & 1 & 0 & 0 & 1 \\ B & 1 & 0 & 1 & 1 \\ C & 0 & 0 & 1 & 0 \\ D & 0 & 0 & 0 & 1 \end{array}$$

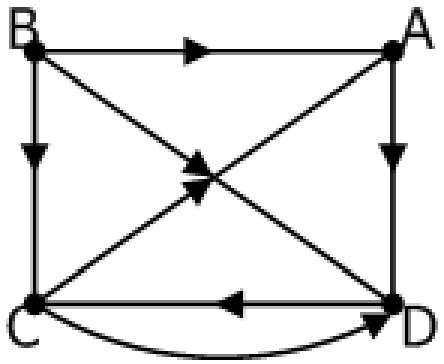
Transitive Closure (Path Matrix)

- **Transitive Closure คือ ความสัมพันธ์ที่ใช้บอกว่ามีเส้นทางเชื่อมระหว่างโหนดในกราฟหรือไม่**
- **สมมุติให้ MP คือเมทริกซ์ที่แสดงว่ามีเส้นทางที่เชื่อมระหว่างโหนดในกราฟ**

$MP = M^1 \,||\, M^2 \,||\, M^3 \,....\, ||\, M^{n-1}$

Example Graph with 3 node

$MP = M^1 \,||\, M^2 \,||\, M^3$



$$MP = \begin{vmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{vmatrix} \begin{vmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

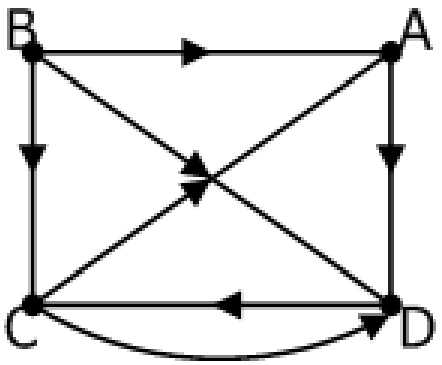
$$MP = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{vmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{vmatrix} \end{array}$$

Transitive Closure

- สมมุติให้ ML คือเมทริกซ์ที่แสดงจำนวนเส้นทางที่เป็นไปได้ทั้งหมดในกราฟ

  ML = $M^1 + M^2 + M^3 .... + M^{n-1}$
- **Example**

  ML = $M^1 + M^2 + M^3$



$$ML = \begin{vmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{vmatrix} + \begin{vmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{vmatrix} + \begin{vmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

$$ML = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{vmatrix} 1 & 0 & 1 & 2 \\ 3 & 0 & 3 & 3 \\ 1 & 0 & 2 & 2 \\ 1 & 0 & 1 & 1 \end{vmatrix} \end{array}$$

Warshall's Algorithm

- ใช้หา MP จาก path เริ่มต้นได้ โดยไม่ต้องสร้างทีละ path ของ M^k
- แนวคิด ทางเดินใหม่(k)จากโหนด i ไปยังโหนด j (path[i][j]) จะมีค่าเท่ากับ 1(true) ก็ต่อเมื่อ
  1) มีทางเดินเก่าจากโหนด i ไปยังโหนด j ($path_{k-1}$ [i][j] = 1) หรือ
  2) มีทางเดินเก่าจากโหนด i อ้อมไปยังโหนด m ($path_{k-1}$ [i][m] = 1) และ มีทางเดินจากโหนด m กลับไปยังโหนด j ($path_{k-1}$[m][j] = 1)

  $path_k$ [i,j] = $path_{k-1}$[i,j]||($path_{k-1}$[i,m] && $path_{k-1}$ [m,j]) ;

path2 [A,A] = path1[A][A]||(path1[A][B]&&path1[B][A])||(path1[A][C]&&path1[C][A])||(path1[A][D]&&path1[D][A])
= 0 ||(0&&0) ||(0&&1) ||(0&&1) ||(1&&0) = 0

path2 [A,B] = path1[A][B]||(path1[A][A]&&path1[A][B])||(path1[A][C]&&path1[C][B])||(path1[A][D]&&path1[D][B])
= 0 ||(0&&0) ||(0&&0) ||(0&&0) ||(1&&0) = 0

path2 [A,C] = path1[A][C]||(path1[A][A]&&path1[A][C])||(path1[A][B]&&path1[B][C])||(path1[A][D]&&path1[D][C])
= 0 ||(0&&0) ||(0&&1) ||(0&&0) ||(1&&1) = 1

path2 [A,D] = path1[A][D]||(path1[A][A]&&path1[A][D])||(path1[A][B]&&path1[B][D])||(path1[A][C]&&path1[C][D])
= 1 ||(0&&1) ||(0&&1) ||(0&&1) ||(1&&0) = 1

ต้องวนรอบทำที่ path length [k]ตั้งแต่ 2 จนถึง n-1 จึงจะได้ครบทุกเส้นทาง

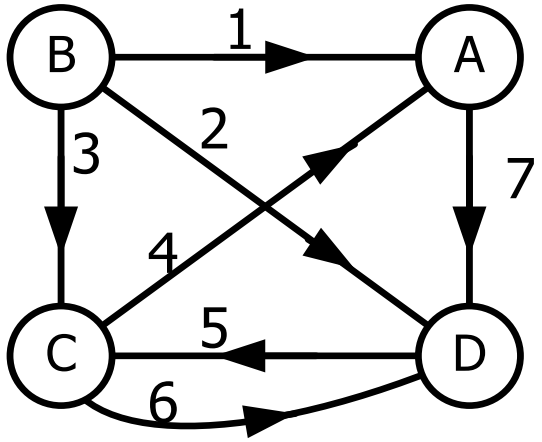
```
for (k = 2; k<n; k++)
```

```
for (i = 0; i <n; i++)
    for (j = 0; j < n; j++)
        for (m = 0; m < n; m++)
            MP[k][i][j] = MP[k-1][i][j]||(MP[k-1][i][m]&&MP[k-1][m][j]);
```

i = ต้นทาง, j = ปลายทาง , m = จุดอ้อม

Weighted Graph

- กราฟที่มี information ของ edge
- Implementations of a weighted graph with Matrices



node[0]	A
node[1]	B
node[2]	C
node[3]	D

Adj =

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

Floyd's Algorithm

- ใช้คำนวณหา Shortest Path ทุกคู่ ของโหนดทั้งหมดที่มีอยู่ในกราฟ
- ใช้หา weighted กราฟของ MP จาก M โดยพิจารณาเลือกเส้นทางที่มี weight น้อยที่สุด (Shortest Path) เพื่อให้ได้เส้นทางที่ดีที่สุด
  - แนวคิด เส้นเดินใหม่(k)จากโหนด vi ไปยังโหนด vj (path[i][j]) สามารถหาได้จากทางเดินเก่า(k-1) โดยเลือกค่าดีที่สุดระหว่าง
    1) weight ของทางเดินเก่าจาก i ไป j หรือ
    2) ผลรวมของ weight ที่ได้จากโหนด i อ้อมไปยังโหนด m และ จากโหนด m กลับไปยังโหนด j นั่นคือ $path_k[i,j] = Min\ (path_{k-1}[i,j]\ ,\ path_{k-1}[i,m] + path_{k-1}[m,j])$

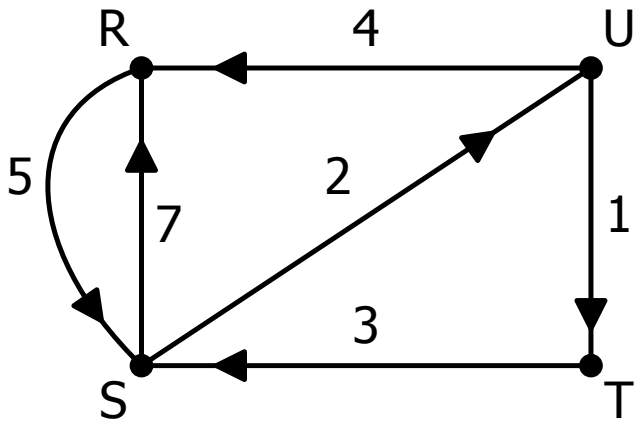
```
for (k = 2; k<n; k++)
{ Copy_Matrix(MPL, MPK);
  for (i = 0; i < n; i++)
     for (j = 0; j < n; j++)
        if (i != j)
          for (m = 0; m < n; m++)
            if (MPK[i][j].length > MPL[i][m].length+MPL[m][j].lenght )
              {MPK[i][j].length = MPL[i][m].length+MPL[m][j].length;    }
               // MPK[i][j].path  =   MPL[i][m].path+"->"+MPL[m][j].path;
}
```

ข้อมูลจริงเก็บที่ Matrix MPK ใช้ MPL จำค่าของMPK รอบก่อนหน้า

```
for (i = 1; i<=n; i++)
  for(j=1; j<=n; j++)
     MPL[i][j] = MPK[i][j];
```

ไม่จำเป็นต้องคำนวณ ที่ i= j เพื่อกลับมาที่เดิม

Weighted Graphs n=4 (find w3)



	R	S	T	U
R	0	5	0	0
S	7	0	0	2
T	0	3	0	0
U	4	0	1	0

ไม่มีเส้นทางต้อง 0 คือ ∞

W1 =

	R	S	T	U
R	∞	5	∞	∞
S	7	∞	∞	2
T	∞	3	∞	∞
U	4	∞	1	∞

	R	S	T	U
R	-	RS	-	-
S	SR	-	-	SU
T	-	TS	-	-
U	UR	-	UT	-

Floyd with path length = 2

	R	S	T	U
R	∞	5	∞	∞
S	7	∞	∞	2
T	∞	3	∞	∞
U	4	∞	1	∞

RU = Min(RU,RS+SU) = Min(∞,5+2) = 7
SR = Min(SR,SU+UR) = Min(7,2+4) = 6
SS = Min(SS,SR+RS) = Min(∞,7+5) = 12
ST = Min(ST,SU+UT) = Min(∞,2+1) = 3
TR = Min(TR,TS+SR) = Min(∞,3+7) = 10
TU = Min(TU,TS+SU) = Min(∞,3+2) = 5
US = Min(US,UT+TS) = Min(∞,1+3) = 4

W2 =

	R	S	T	U
R	∞	5	∞	7
S	6	12	3	2
T	10	3	∞	5
U	4	4	1	∞

	R	S	T	U
R	-	RS	-	RSU
S	SUR	SRS	SUT	SU
T	TSR	TS	-	TSU
U	UR	UTS	UT	-

path length = 3

RU = RS + SU = RSU
SR = SU + UR = SUR
SS = SR + RS = SRS
ST = SU + UT = SUT
TR = TS + SR = TSR
TU = TS + SU = TSU
US = UT + TS = UTS

	R	S	T	U
R	∞	5	∞	7
S	6	12	3	2
T	10	3	∞	5
U	4	4	1	∞

RT = Min(RT,RU+UT) = Min(∞,7+1) = 8
SS = Min(SS,SU+US) = Min(12,2+4) = 6
TR = Min(TR,TS+SR) = Min(10,3+6) = 9
TT = Min(TT,TU+UT) = Min(∞,5+1) = 6
UU = Min(UU,US+SU) = Min(∞,4+2) = 6

W3 =

	R	S	T	U
R	12	5	8	7
S	6	6	3	2
T	9	3	6	5
U	4	4	1	6

	R	S	T	U
R	RSR	RS	RSUT	RSU
S	SR	SUTS	SUT	SU
T	TSUR	TS	TSUT	TSU
U	UR	UTS	UT	UTSU

RT = RSU + UT = RSUT
SS = SU + UTS = SUTS
TR = TS + SUR = TSUR
TT = TSU + UT = TSUT
UU = UTS + SU = UTSU

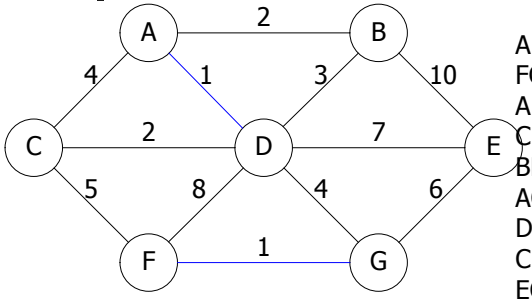
Greedy 's Algorithm

- **Greedy 's Algorithm**
  - หลักการหาคำตอบ(การแก้ปัญหา) โดยการเลือกสิ่งที่ดีที่สุดที่เกิดขึ้นในปัจจุบัน แล้วทำต่อเนื่องไปเรื่อยๆ จนกระทั่งเจอคำตอบที่ดีที่สุด (หรือคำตอบที่ต้องการ) แต่บางปัญหาที่ไม่สามารถสำรวจได้ครบทุกทางเลือกที่เป็นไปได้ อาจหาคำตอบได้ไม่ถูกต้อง
- **Spanning Tree**
  - Tree ที่สร้างจากกราฟแบบไม่มีทิศทาง โดยประกอบด้วยโหนดทุกโหนดของกราฟนั้น และจะต้องไม่เกิดการวนรอบ Tree ที่สร้างขึ้นแล้วมีผลรวมของระยะทาง ทั้งหมดสั้นที่สุด เรียกว่า minimum spanning tree
  - **Kruskal 's Algorithm**
    - สร้าง minimum spanning tree โดยเลือกจาก edge ที่สั้นที่สุดที่เชื่อมระหว่างโหนด และไม่ทำให้เกิดวนรอบไปเรื่อยๆตามลำดับ จนกระทั่งครบทุกโหนด
  - **Prim 's Algorithm**
    - สร้าง minimum spanning tree โดยเลือกโหนดเริ่มต้น(2 โหนด) จากที่ edge ที่สั้นที่สุดก่อน แล้วจึงเลือกโหนดถัดๆไปที่ต่ออยู่และมี edge สั้นที่สุด เพิ่มทีละโหนด โดยไม่ให้เกิดการวนรอบ จนกระทั่งครบทุกโหนด
- **Shotest path** หาเส้นทางที่สั้นที่สุดจากจุดเริ่มต้นไปยังจุดหมาย
  - **Dijkstra 's Algorithm**
    - ใช้สำหรับหาเส้นทางที่สั้นที่สุดระหว่างโหนด 2 โหนด โดยกำหนดจุดเริ่มต้นจากโหนดที่ต้องการ แล้วหาเส้นทางที่สั้นที่สุดที่ออกจากโหนดที่เลือกไว้ เพื่อให้ได้โหนดใหม่ ไปเรื่อยๆ จนกระทั่งครบทุกโหนด (หรือเจอคำตอบของโหนดที่ต้องการ)

Kruskal 's Algorithm

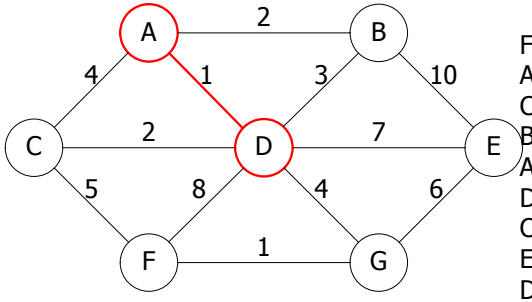
- **Kruskal 's Algorithm ใช้สำหรับสร้าง minimum spanning tree โดยเลือก edge ที่สั้นที่สุดตามลำดับไปเรื่อยๆที่ไม่ทำให้เกิดวนรอบ จนกระทั่งเส้นทางเดินครบทุกโหนด**
  - **สร้างตารางสำหรับเก็บค่าระยะทางระหว่างจุด 2 จุดที่เชื่อมต่อในกราฟทั้งหมด**
    - **ดึงข้อมูลจากกราฟลงในตาราง แล้วเรียงลำดับข้อมูลในตาราง (หรือสร้างตารางเรียงลำดับด้วย Priority Queue)**
  - **สร้างตารางสำหรับเก็บคำตอบ MST**
  - **วนรอบทำซ้ำ**
    - **เลือกเส้นทางที่ดีที่สุด(สั้นที่สุด) นำออกจากตารางลำดับ**
    - **ตรวจสอบเส้นทางที่เลือกกับตารางคำตอบ ถ้าทำให้เกิดวนรอบให้ตัดทิ้ง**
    - **ถ้าไม่เกิดการวนรอบให้เก็บ Edge นั้นไว้เป็นคำตอบ MST ซึ่งจะได้โหนดใหม่และเส้นทางที่เป็นคำตอบ**
  - **วนรอบทำซ้ำจนกระทั่งเลือกได้ครบทุกจุด(เลือกจนหมดตารางลำดับ)**

Kruskal 's Algorithm



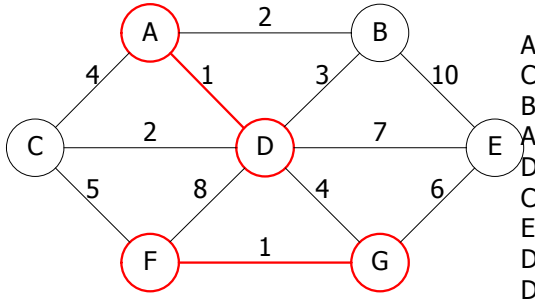
AD = 1
FG = 1
AB = 2
CD = 2
BD = 3
AC = 4
DG = 4
CF = 5
EG = 6
DE = 7
DF = 8
BE = 10

Start



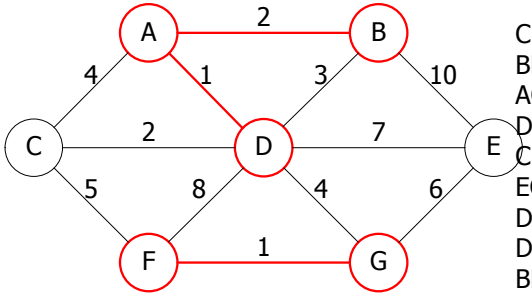
FG = 1
AB = 2
CD = 2
BD = 3
AC = 4
DG = 4
CF = 5
EG = 6
DE = 7
DF = 8
BE = 10

Select AD = 1



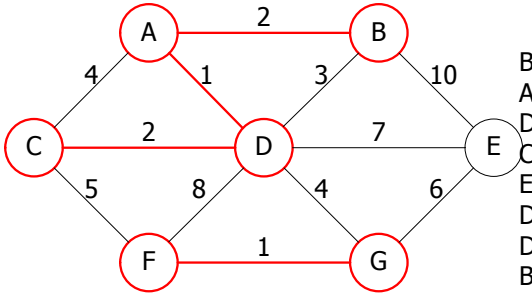
AB = 2
CD = 2
BD = 3
AC = 4
DG = 4
CF = 5
EG = 6
DE = 7
DF = 8
BE = 10

Select FG = 1



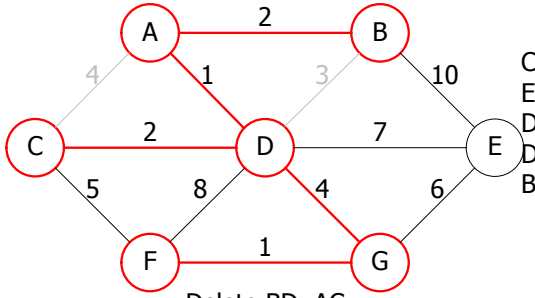
CD = 2
BD = 3
AC = 4
DG = 4
CF = 5
EG = 6
DE = 7
DF = 8
BE = 10

Select AB = 2



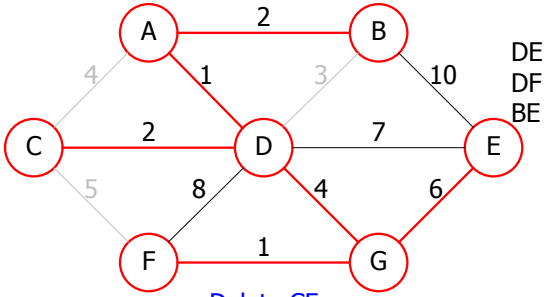
BD = 3
AC = 4
DG = 4
CF = 5
EG = 6
DE = 7
DF = 8
BE = 10

Select CD = 2



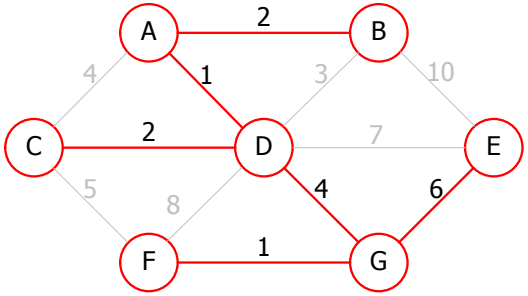
CF = 5
EG = 6
DE = 7
DF = 8
BE = 10

Delete BD, AC
Select DG = 4

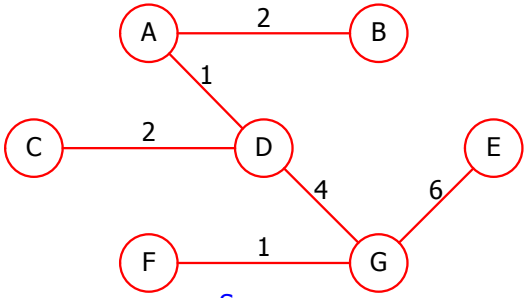


DE = 7
DF = 8
BE = 10

Delete CF
Select EG = 6



Delete DE, DF, BE

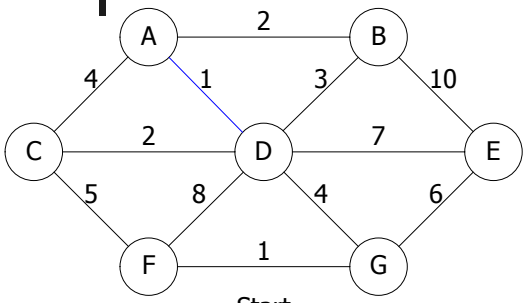


Success
Total weight =1+1+2+2+4+6 =16

Prim 's Algorithm

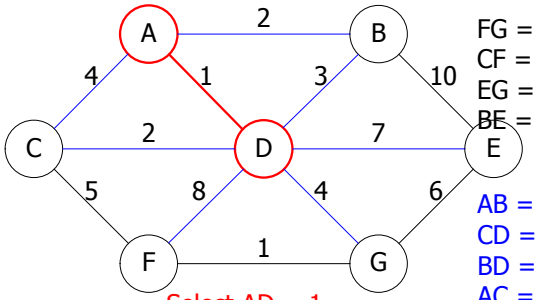
- **Prim 's Algorithm ใช้สำหรับสร้าง minimum spanning tree โดยเลือก edge ที่สั้นที่สุดก่อน เพื่อให้ได้โหนดเริ่มต้น(2 โหนด) แล้วจึงเลือก edge ถัดๆไป ที่ต่ออยู่กับโหนดที่เลือกไว้แล้วทำซ้ำไปเรื่อยๆ โดยไม่ให้เกิดการวนรอบจนกระทั่งครบทุกโหนด**
  - **สร้างตารางสำหรับเก็บค่าระยะทางระหว่างจุด 2 จุดที่เชื่อมต่อในกราฟทั้งหมด**
    - **แปลงข้อมูลจากกราฟลงตาราง**
  - **สร้างตารางผังลำดับรอการเลือก ที่สามารถดึงข้อมูลตัวที่ดีที่สุดออกมาได้ (Priority Queue)**
  - **สร้างตารางสำหรับเก็บคำตอบ MST**
  - **เลือก Edge ที่สั้นที่สุดจากตาราง(นำออกจากตาราง) นำมาใส่ในผังลำดับรอการเลือก ซึ่งจะได้คำตอบเริ่มต้น 2 โหนด**
  - **วนรอบทำซ้ำ (พิจารณาจากผังลำดับรอการเลือก)**
    - **เลือก Edge ที่สั้นที่สุดจากผังลำดับรอการเลือก เพื่อพิจารณาโหนดที่ต่ออยู่**
    - **ถ้าทั้ง 2 โหนดของ Edge นั้นเป็นโหนดที่ถูกเลือกไว้แล้วใน MST แสดงว่าทำให้เกิดลูป ให้ตัดทิ้ง**
    - **ถ้ามีโหนดใหม่ที่ต่ออยู่กับ Edge นั้นยังไม่ถูกเลือก ให้เลือกโหนดและ Edge นั้นเป็นคำตอบเก็บใน MST**
      - **เลือก Edge ทั้งหมดที่เชื่อมอยู่กับโหนดใหม่ที่เลือก(ดึง Edge จากตาราง) มาใส่ในผังลำดับรอการเลือก**
  - **ทำซ้ำจนกระทั่งเลือกได้ครบทุกโหนด**

Prim 's Algorithm



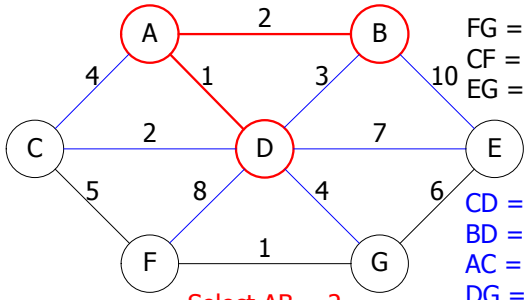
Start
Add AD

FG = 1
AB = 2
CD = 2
BD = 3
AC = 4
DG = 4
CF = 5
EG = 6
DE = 7
DF = 8
BE = 10
AD = 1



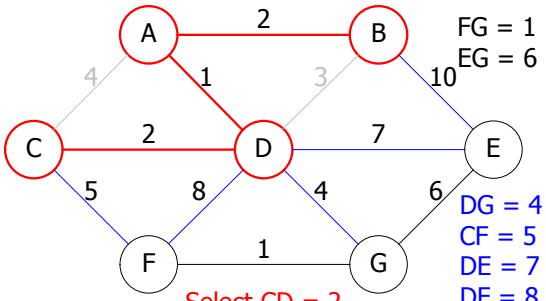
Select AD = 1
Add AB,AC,BD,CD,DE,DF,DG

FG = 1
CF = 5
EG = 6
BE = 10
AB = 2
CD = 2
BD = 3
AC = 4
DG = 4
DE = 7
DF = 8



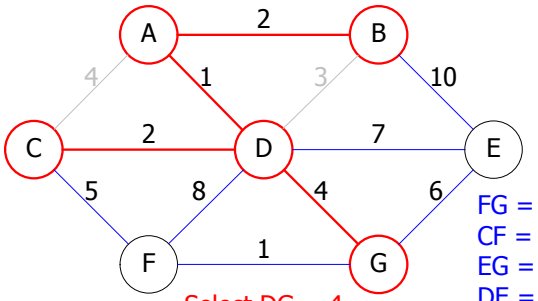
Select AB = 2
Add BE

FG = 1
CF = 5
EG = 6
CD = 2
BD = 3
AC = 4
DG = 4
DE = 7
DF = 8
BE = 10



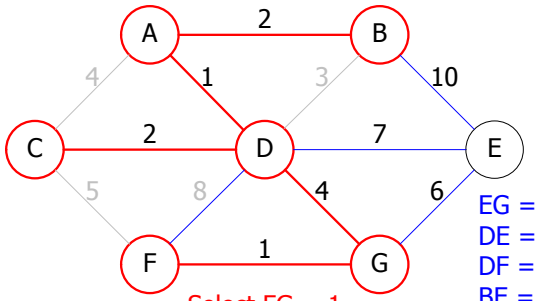
Select CD = 2
Add CF
Delete BD,AC

FG = 1
EG = 6
DG = 4
CF = 5
DE = 7
DF = 8
BE = 10



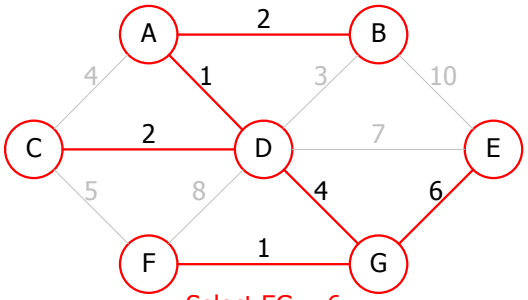
Select DG = 4
Add EG, FG

FG = 1
CF = 5
EG = 6
DE = 7
DF = 8
BE = 10

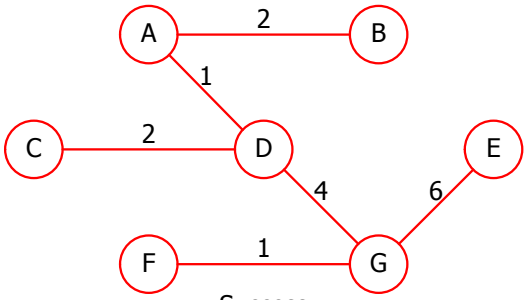


Select FG = 1
Delete CF

EG = 6
DE = 7
DF = 8
BE = 10



Select EG = 6
Delete DE,DF,BE

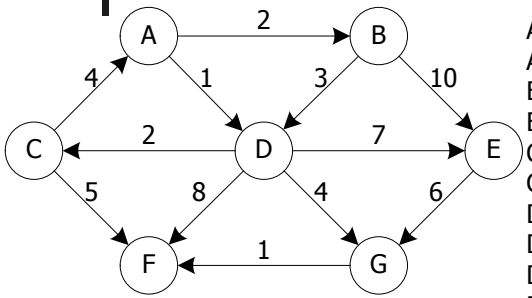


Success
Total weight =1+2+2+4+1+6 =16

Dijkstra 's Algorithm

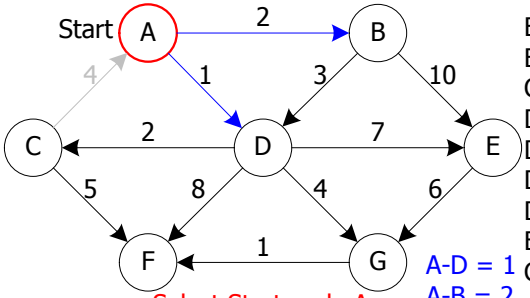
- **Dijkstra 's Algorithm ใช้หาเส้นทางที่มี weight น้อยที่สุดระหว่างจุด 2 จุด ที่ต้องการโดยใช้หลักการของ greedy algorithm**
  - **สร้างตารางสำหรับเก็บค่าระยะทางระหว่างจุด 2 จุดที่เชื่อมต่อในกราฟทั้งหมด**
    - **แปลงข้อมูลจากกราฟลงตาราง**
  - **สร้างตารางลำดับสำหรับเก็บเส้นทางที่รอการพิจารณา (Priority Queue)**
  - **สร้างตารางสำหรับเก็บเส้นทางที่ดีที่สุด(Shortest Path) ที่เลือกไว้เป็นคำตอบ**
  - **เลือกจุดเริ่มต้นที่ต้องการเป็นต้นทาง (ค่าเริ่มต้นระยะทาง = 0) สร้างผังเส้นทาง**
  - **วนรอบทำซ้ำ**
    - **ดึง Edge ที่มีจุดเริ่มต้นออกจากโหนดที่เลือกทั้งหมด(จากตารางเริ่มต้น) นำมาใส่ในตารางที่รอการพิจารณา (ต้องคำนวณระยะทางใหม่ที่เริ่มจากจุดต้นทาง) มายังโหนดนั้น**
    - **ลบ Edge ที่มีจุดปลายทางชี้มายังโหนดที่เลือกไว้ทั้งหมด ทั้งจากตารางเริ่มต้นและตารางรอพิจารณา (เพื่อทำให้ไม่เกิดการวนรอบ)**
    - **ดึงโหนดต่อไปที่มีระยะทางรวมสั้นที่สุด(ต้องนับจากจุดต้นเริ่มต้น) จากตารางรอการพิจารณา เพื่อนำมาใช้เป็นคำตอบของเส้นทางที่ดีที่สุด(Shortest path) ของโหนดใหม่**
  - **ทำซ้ำจนกระทั่งครบทุกโหนด หรือเจอโหนดที่ต้องการ**

Dijkstra from node A



A-B = 2
A-D = 1
B-D = 3
B-E = 10
C-A = 1
C-F = 5
D-C = 2
D-E = 7
D-F = 8
D-G = 4
E-G = 6
G-F = 1

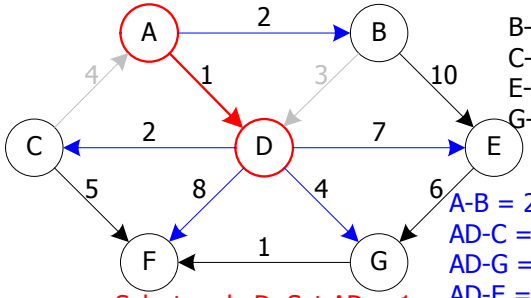
Start



B-D = 3
B-E = 10
C-F = 5
D-C = 2
D-E = 7
D-F = 8
D-G = 4
E-G = 6
G-F = 1

A-D = 1
A-B = 2

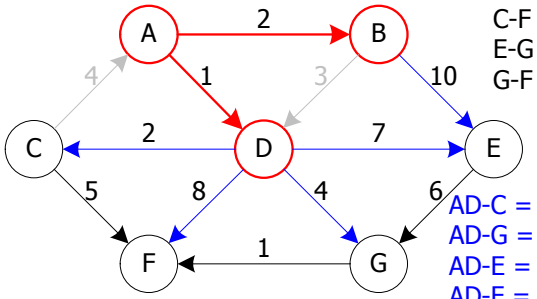
Select Start node A
Add A-D,A-B
Delete C-A



B-E =10
C-F = 5
E-G = 6
G-F = 1

A-B = 2
AD-C = 1+2
AD-G = 1+4
AD-E = 1+7
AD-F = 1+8

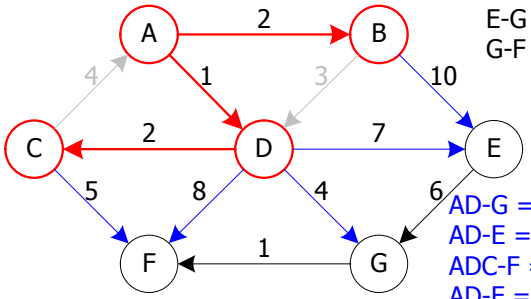
Select node D, Set AD = 1
Add AD-C,AD-G,AD-E,AD-F
Delete B-D



C-F = 5
E-G = 6
G-F = 1

AD-C = 1+2
AD-G = 1+4
AD-E = 1+7
AD-F = 1+8
AB-E = 2+10

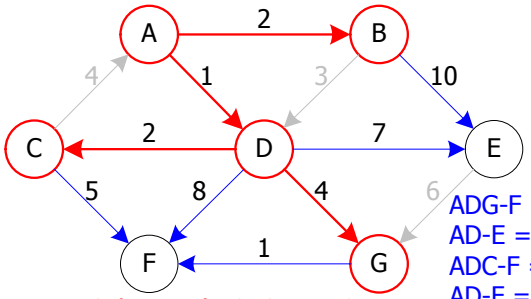
Select node B, Set AB = 2
Add AB-E



E-G = 6
G-F = 1

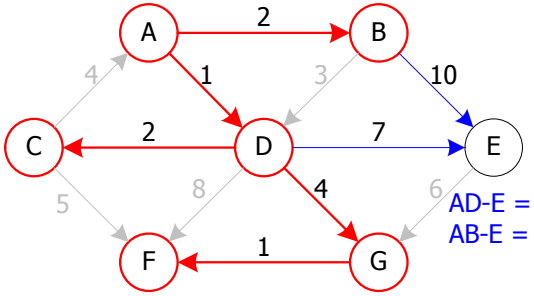
AD-G = 1+4
AD-E = 1+7
ADC-F = 3+5
AD-F = 1+8
AB-E = 2+10

Select node C, Set ADC = 3
Add ADC-F



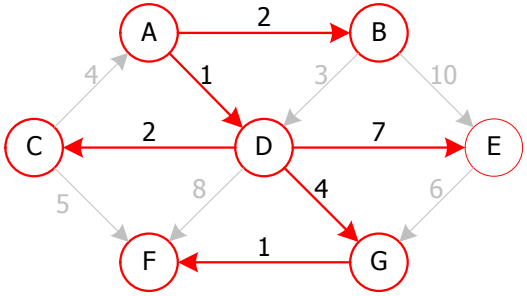
ADG-F = 5+1
AD-E = 1+7
ADC-F = 3+5
AD-F = 1+8
AB-E = 2+10

Select node G, Set ADG = 5
Add ADG-F
Delete E-G

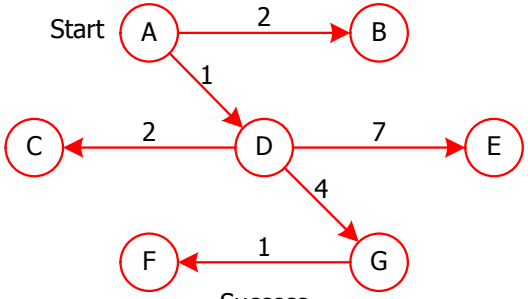


AD-E = 1+7
AB-E = 2+10

Select node F, Set ADGF = 6
Delete ADC-F,AD-F



Select node E, Set ADE = 8
Delete AB-E



Success
(Start at node A)