ค่ายโอลิมปิกวิชาการวิชาคอมพิวเตอร์ ค่าย 2
ระหว่างวันที่ 18 เมษายน ถึง 30 เมษายน พ.ศ. 2566

โครงสร้างข้อมูลกราฟ 2 และอัลกอริทึมกราฟ

อ.ลือพล พิพานเมฆาภรณ์ luepol.p@sci.kmutnb.ac.th

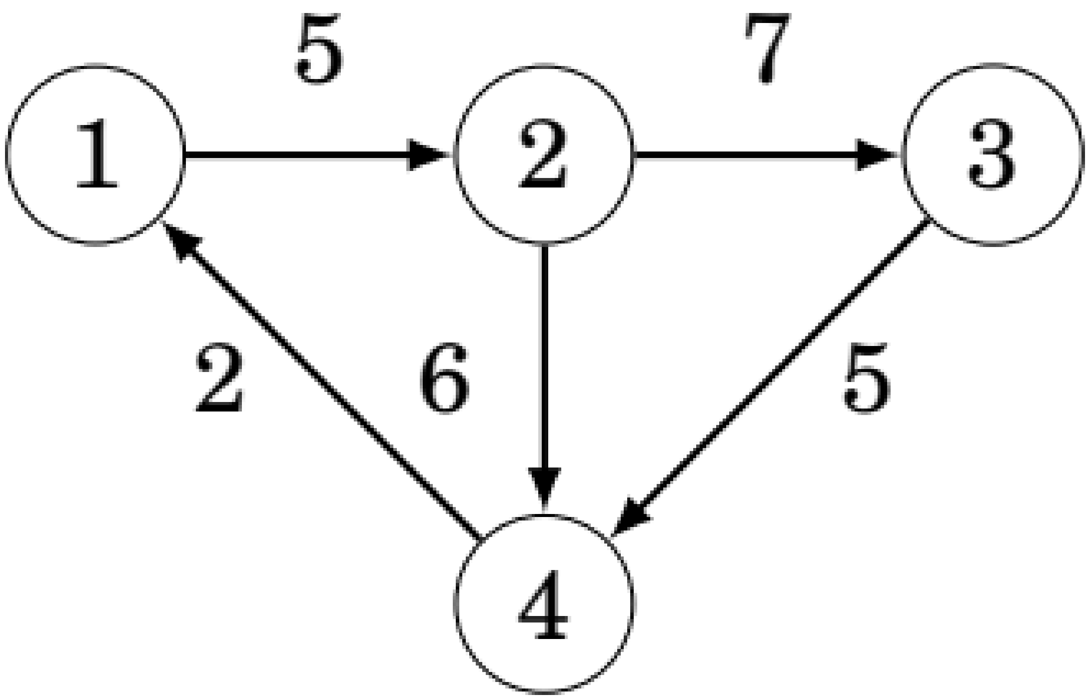
ภาควิชาวิทยาการคอมพิวเตอร์และสารสนเทศ มหาวิทยาลัยเทคโนโลยีพระจอมเกล้าพระนครเหนือ

เนื้อหา

- กราฟชนิดถ่วงน้ำหนัก (Weighted graph)
  - เมตริกซ์ประชิด (Adjacency matrix)
  - ลิสต์ประชิด (Adjacency list)
- ปัญหาการหาเส้นทางสั้นที่สุด (Shortest path)
  - Single-source shortest path problem
  - All-pair shortest path problem
  - การเก็บ path ของคำตอบ
- ปัญหาการหาต้นไม้แผ่ทั่วที่เล็กที่สุด (Minimum spanning tree)
  - Prim's algorithm
  - Kruskal's algorithm

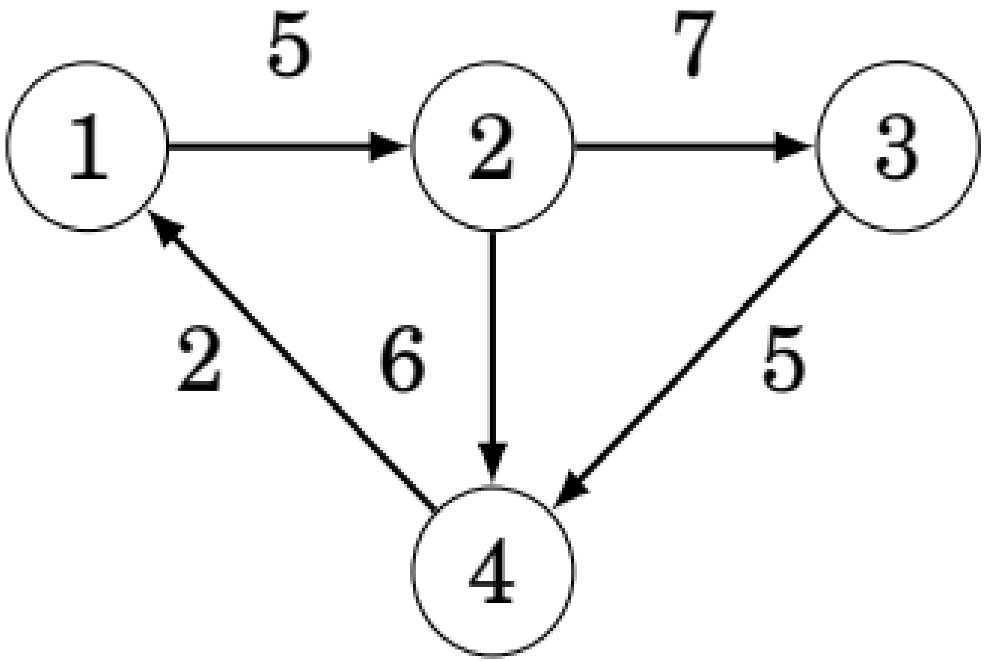
กราฟชนิดถ่วงน้ำหนัก (Weighted graph)

- กราฟชนิดถ่วงน้ำหนัก (weighted graph) เป็นรูปแบบกราฟที่มีการระบุค่าน้ำหนัก (weight) ให้กับเอดจ์ในกราฟ
- ค่าน้ำหนักอาจแทนด้วยระยะทาง คะแนน หรือเงื่อนไขอื่นขึ้นอยู่กับการใช้งานกราฟ
- การแสดงกราฟชนิดถ่วงน้ำหนักทำได้ 2 วิธี ได้แก่
    - ➢ เมตริกซ์ประชิด (adjacency matrix)
    - ➢ ลิสต์ประชิด (adjacency list)



เมตริกซ์ประชิด (Adjacency matrix)

```
int adj[N][N];
```



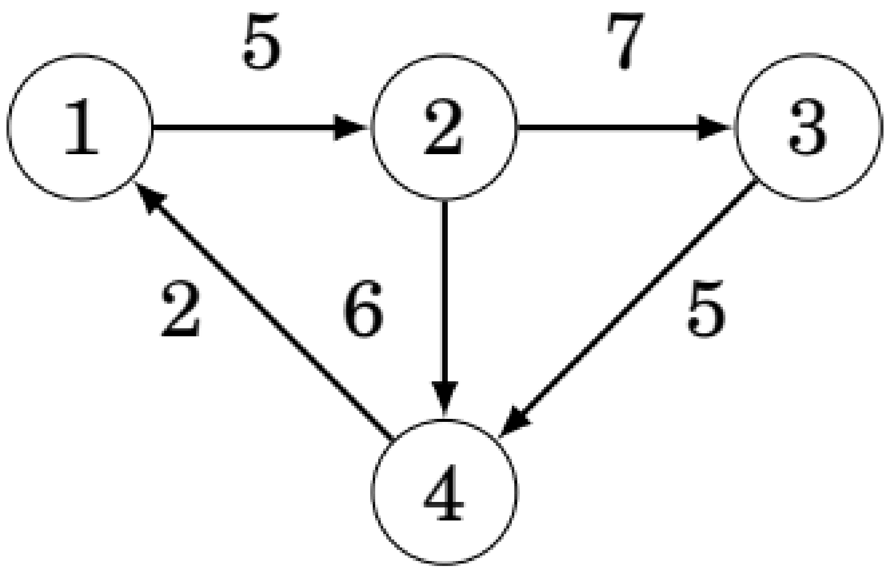
	1	2	3	4
1	0	5	0	0
2	0	0	7	6
3	0	0	0	5
4	2	0	0	0

ลิสต์ประชิด (Adjacency list)

- อาร์เรย์ของเวกเตอร์ (vector) โดยที่สมาชิกแต่ละตัวแสดงในรูปแบบของ pair<int, int>
  - pair.*first* แทนเวอร์เท็กซ์ปลายทาง
  - pair.*second* แทนค่าน้ำหนักระหว่างเวอร์เท็กซ์

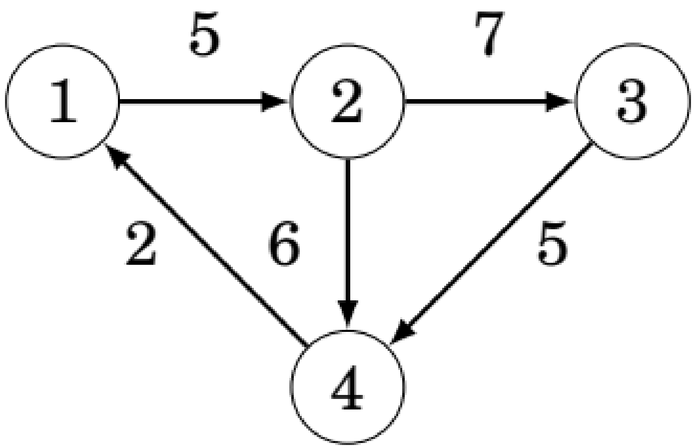
```
vector<pair<int,int>> adj[N];
```

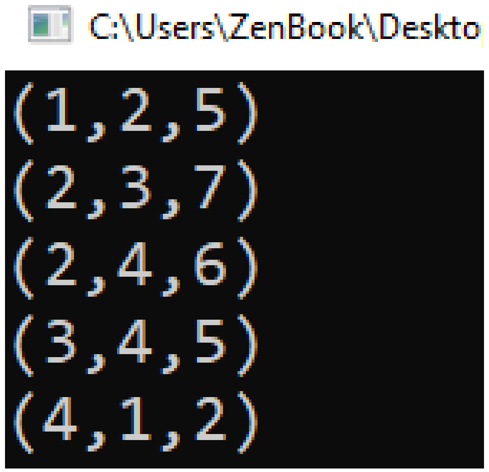
```
int main()
{   vector<pair<int, int> > adj[4];

    adj[1].push_back({2, 5});
    adj[2].push_back({3, 7});
    adj[2].push_back({4, 6});
    adj[3].push_back({4, 5});
    adj[4].push_back({1, 2});

    return 0;
}
```



การเข้าถึงสมาชิกในลิสต์ประชิด

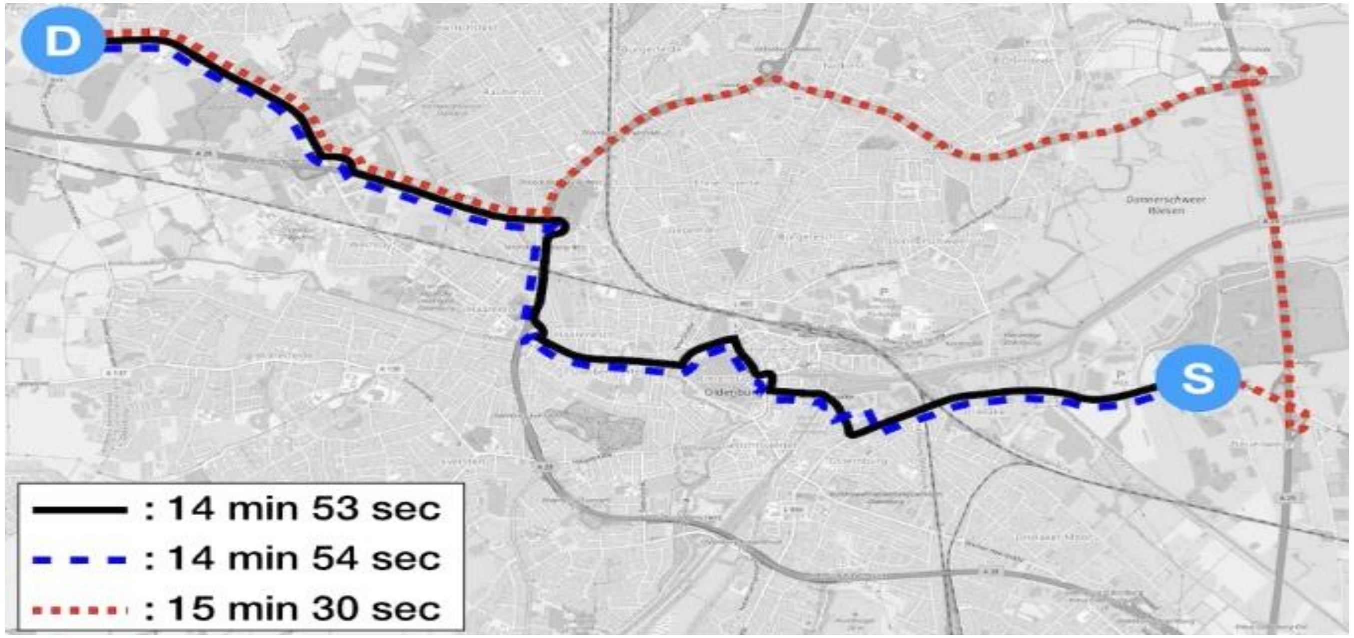
```
void printGraph(vector<pair<int, int> > adj[], int V)
{   int v, w;
     for (int u = 0; u < V; u++) {
        for (auto it = adj[u].begin(); it != adj[u].end(); it++) {
          v = it->first;
          w = it->second;
          cout << "(" << u+1 << "," << v+1 << "," << w << ")" << endl;
          }
     }
}
```





ปัญหาเส้นทางที่สั้นที่สุด (Shortest path) ในกราฟ

- นิยาม ปัญหาเส้นทางสั้นที่สุด (shortest path) หมายถึงเส้นทางระหว่างเวอร์เท็กซ์ในกราฟที่ให้ความยาวรวมของเอจด์ (edge) น้อยที่สุด
- เป็นปัญหาของกราฟมีน้ำหนัก (weighted graph) เนื่องจากค่าน้ำหนักของเอดจ์อาจแทนระยะทางระหว่างเวอร์เท็กซ์ได้ ดังนั้นเป็นการหาผลรวมที่น้อยที่สุดของเอจด์จาก u ไป v



Shortest path problem

ปัญหาเส้นทางที่สั้นที่สุด (Shortest path)

ปัญหาการหาเส้นทางที่สั้นที่สุด มักสามารถแบ่งออกเป็นสองประเภท ได้แก่

1. การหาเส้นทางสั้นที่สุดโดยกำหนดจุดเริ่มต้น (single source) เรียกปัญหานี้ว่า

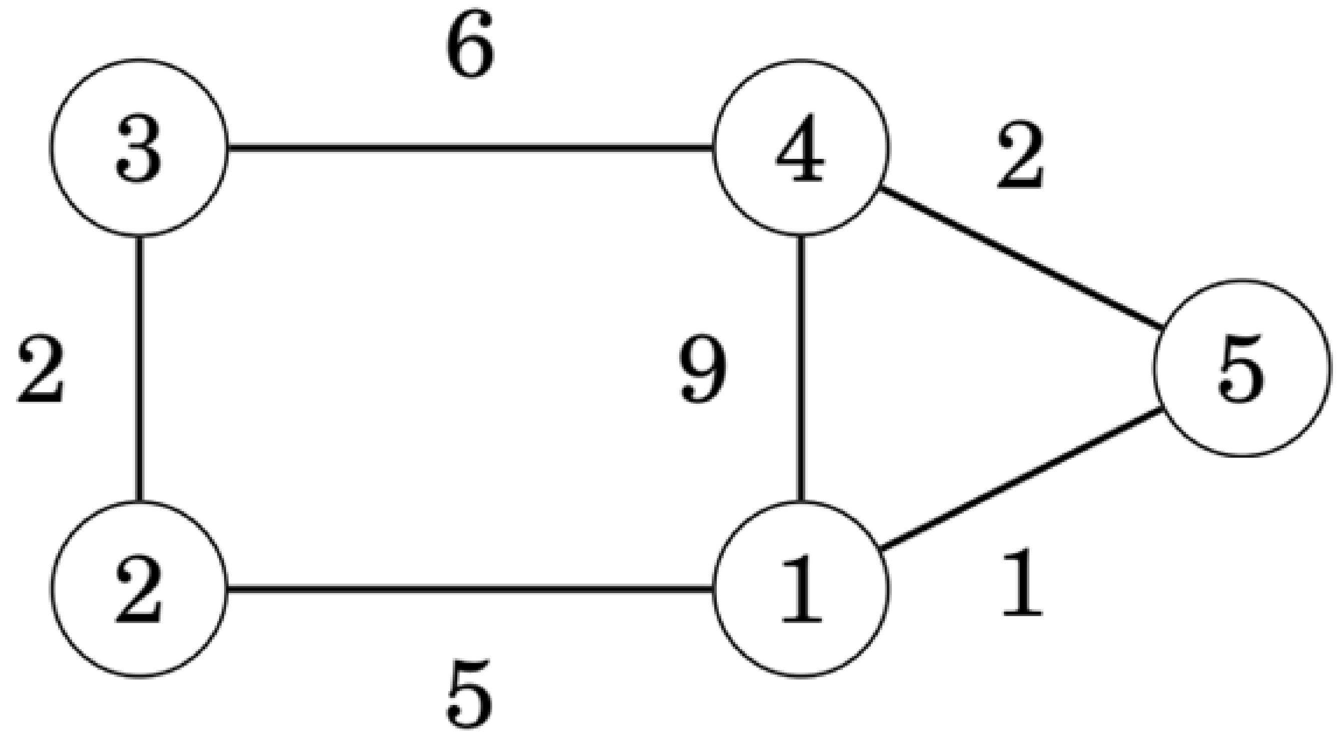
   Single Source Shortest Path (SSSP)

2. การหาเส้นทางที่สั้นที่สุดที่จุดใดก็ได้ เรียกปัญหานี้ว่า

   All-Pair Shortest Path (APSP)

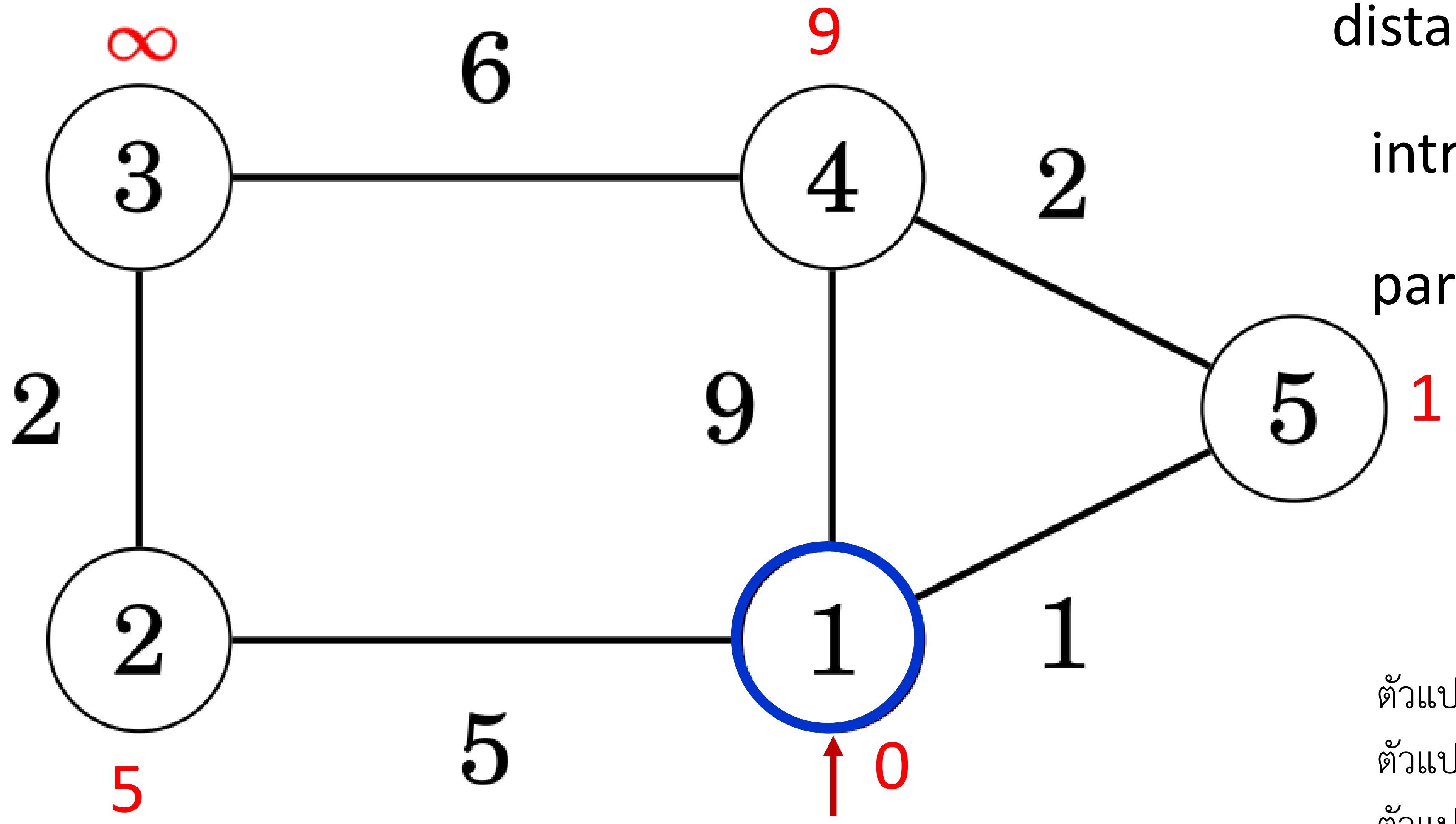
อัลกอริทึม Dijkstra

กำหนดจุดเริ่มต้น (single source) ในกราฟ จากนั้นจะเริ่มแผ่ไปยังเอดจ์ (edge) รอบข้างที่มีความยาวสั้นที่สุด และแผ่ต่อไปโดยมีเงื่อนไขว่าเอจด์จะถูกเลือกก็ต่อเมื่อระยะทางจาก source รวมกับน้ำหนักของเอจด์น้อยที่สุด



ต้องการหาเส้นทางสั้นสุดจากเวอร์เท็กซ์หมายเลข 1 ไปยังทุกเวอร์เท็กซ์ในกราฟ

Dijkstra algorithm



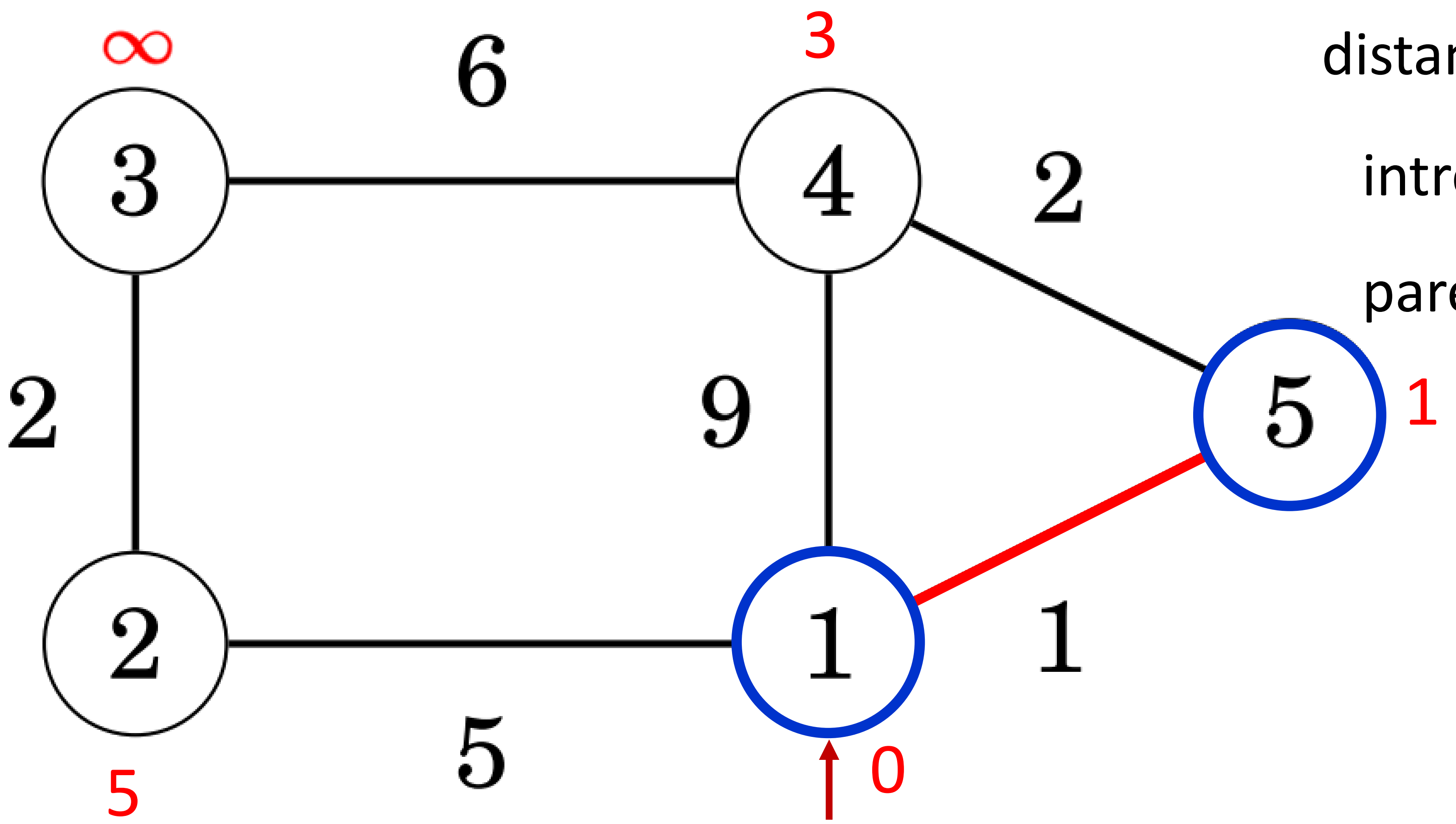
	1	2	3	4	5
distance	0	5	∞	9	1
intree	1	0	0	0	0
parent	-1	-1	-1	-1	-1

ตัวแปร distance เก็บระยะทางจาก source

ตัวแปร intree เก็บสถานะการเลือก vertex

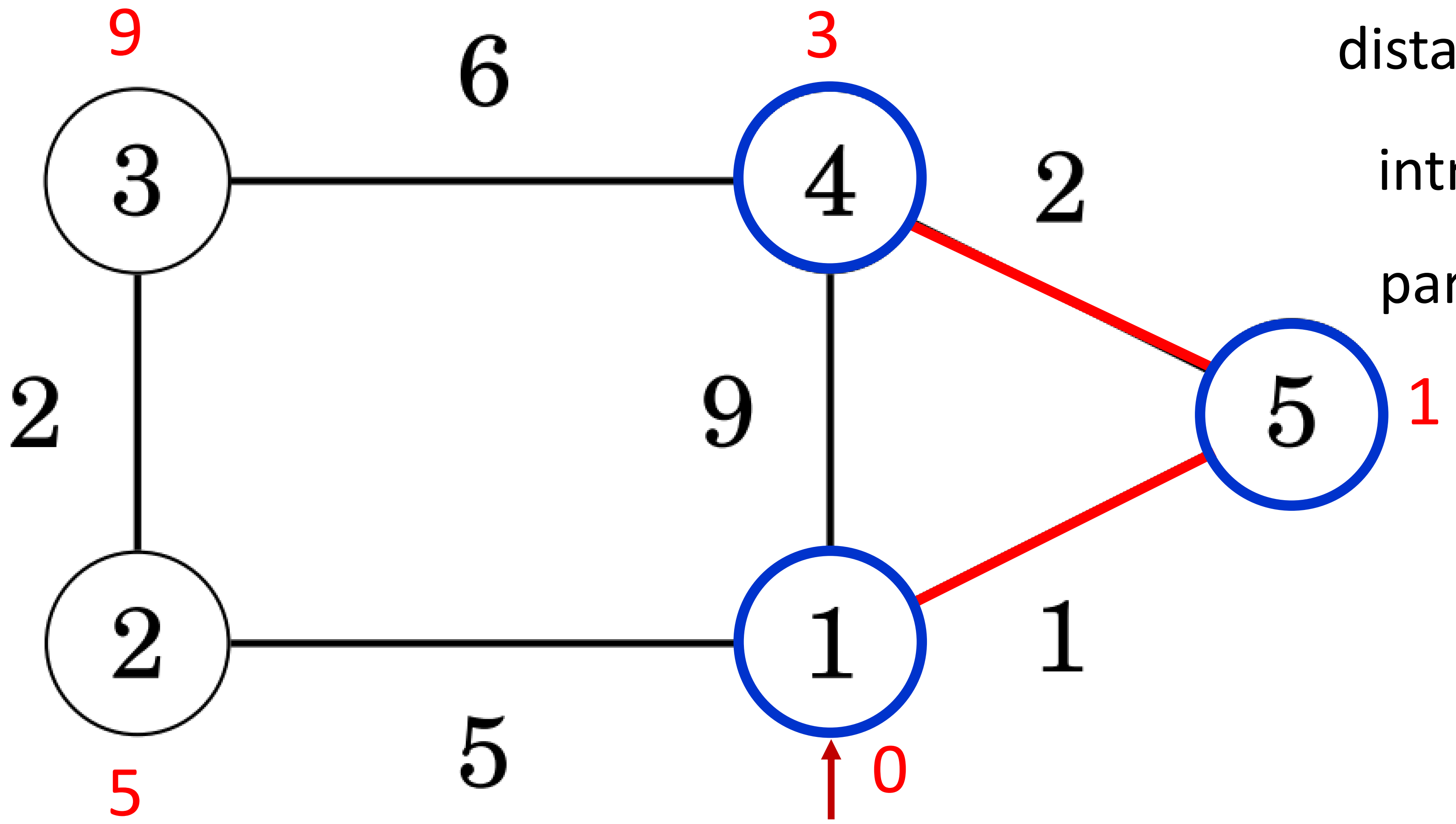
ตัวแปร parent เก็บ path ของต้นไม้

Dijkstra algorithm



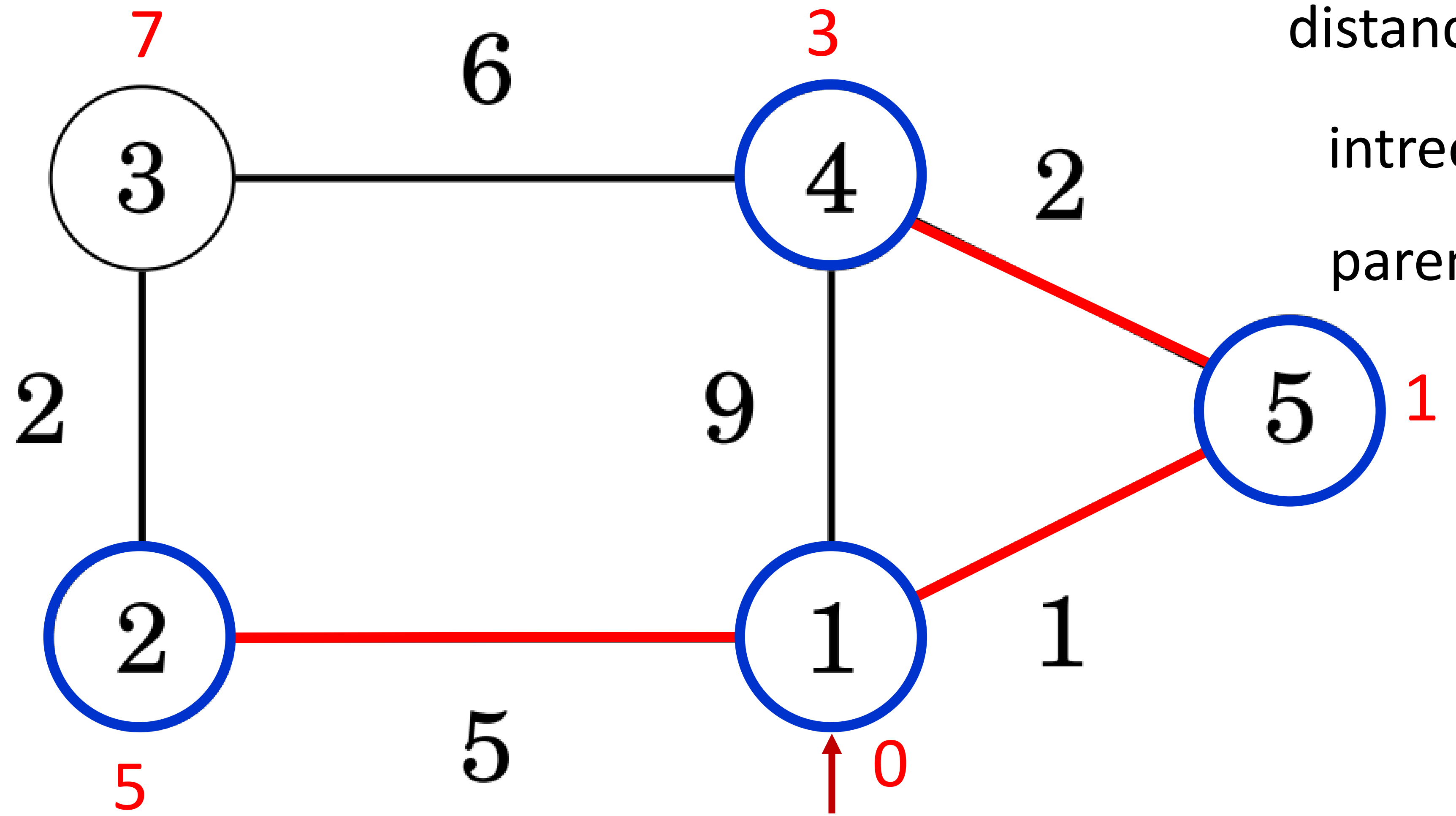
	1	2	3	4	5
distance	0	5	∞	3	1
intree	1	0	0	0	1
parent	-1	-1	-1	-1	1

Dijkstra algorithm



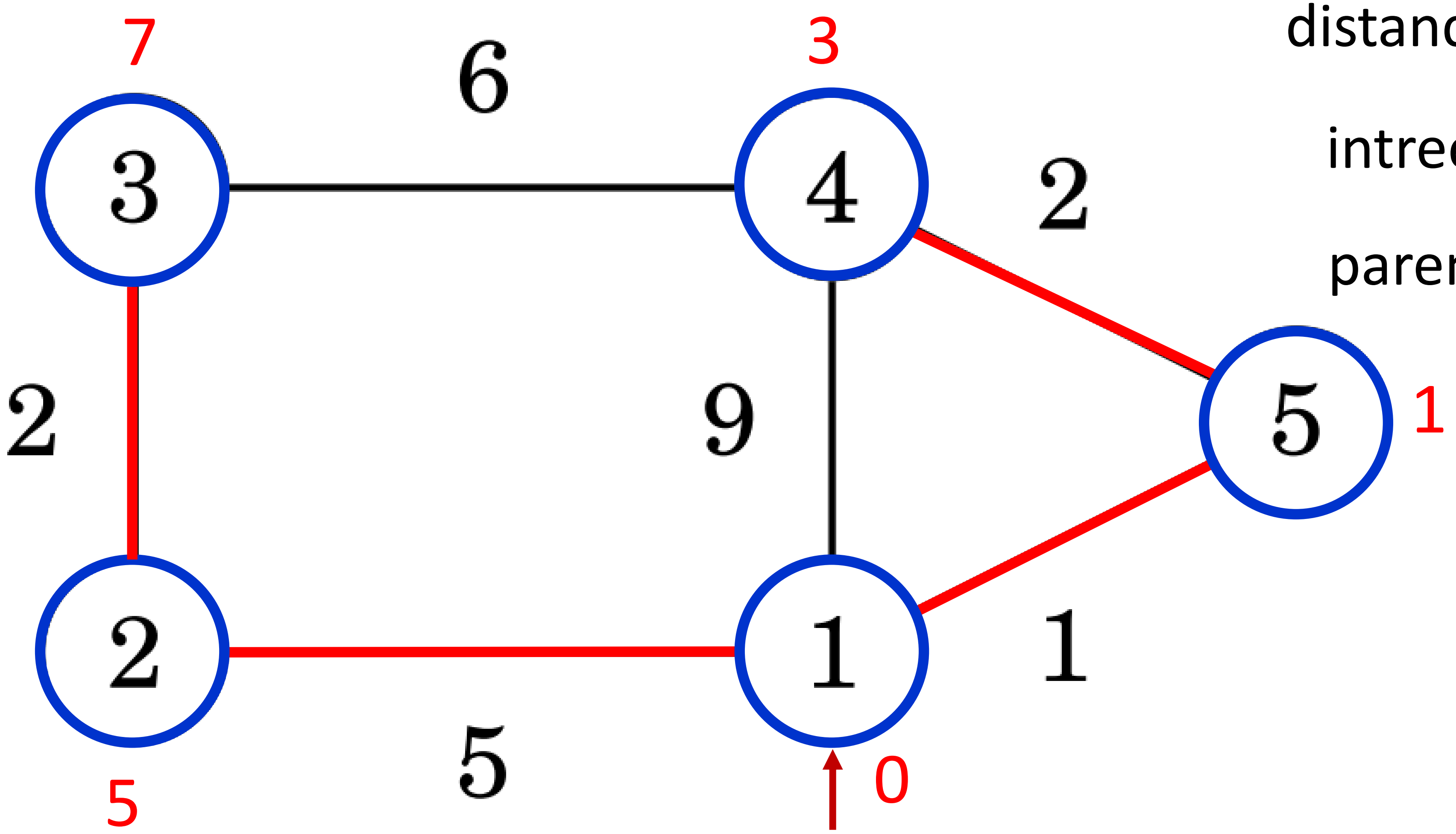
	1	2	3	4	5
distance	0	5	9	3	1
intree	1	0	0	1	1
parent	-1	-1	-1	5	1

Dijkstra algorithm



	1	2	3	4	5
distance	0	5	7	3	1
intree	1	1	0	1	1
parent	-1	1	-1	5	1

Dijkstra algorithm



	1	2	3	4	5
distance	0	5	7	3	1
intree	1	1	1	1	1
parent	-1	1	2	5	1

Dijkstra algorithm implemented in C

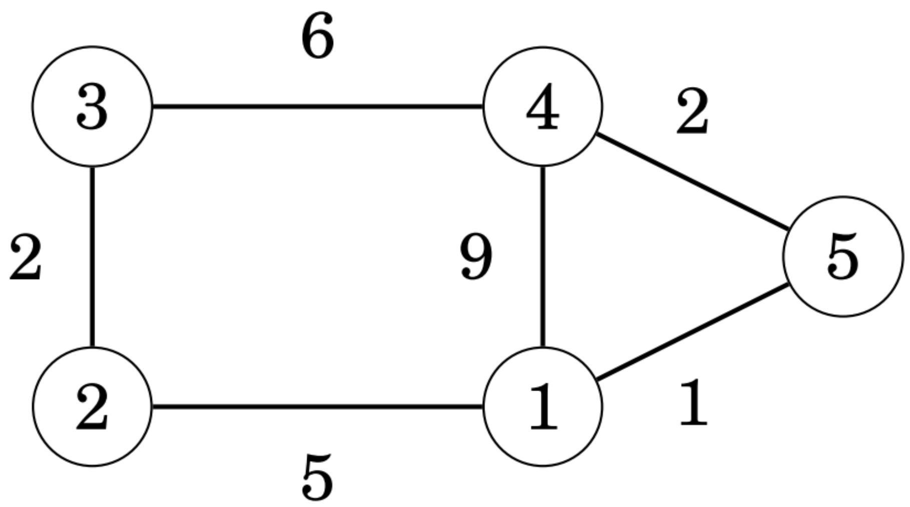
```c
#define inf 50000
#define false 0
#define true 1

void dijkstra(int graph[][V], int src)
{  int dist[V], inTree[V], parent[V];
   for (int i = 0; i < V; i++)    // initial values
        dist[i] = inf, parent[i] = -1, inTree[i]=-1;
    dist[src] = 0;                             // start vectex
    for(int i = 0; i < V-1; i++)
     { int u = minDistance(dist, inTree);                    // find node u with minimum distance
       inTree[u] = true;                                     // add u into the tree
       for (int v = 0; v < V; v++)                           // update all minimum weights
          if (inTree[v] == false && graph[u][v] > 0 && dist[u] + graph[u][v] < dist[v])
             { dist[v] = dist[u] + graph[u][v];
               parent[v] = u;                                // update parent
             }
     }
}
```

```c
int minDistance(int dist[], int inTree[])
{ int min = inf, min_index;

 for(int v = 0; v < V; v++)
  if (inTree[v] == false && dist[v] <= min)
      { min = dist[v];
        min_index = v;    }
 return min_index;
}
```

Dijkstra algorithm in C++ STL

```
#include<bits/stdc++.h>
using namespace std;

typedef pair<int, int> iPair;

int main()
{   int V = 5;
    vector<iPair > adj[V];

    addEdge(adj, 1, 2, 5);
    addEdge(adj, 1, 4, 9);
    addEdge(adj, 1, 5, 1);
    addEdge(adj, 4, 5, 2);
    addEdge(adj, 3, 4, 6);
    addEdge(adj, 2, 3, 2);

    Dijkstra(adj, V, 1);
}
```

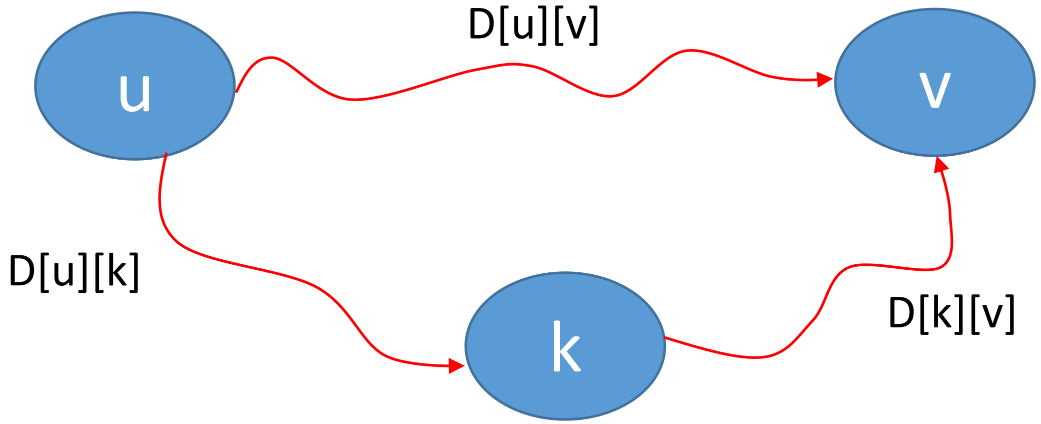
```
void addEdge(vector <pair<int, int> > adj[],
int u, int v, int wt)
{
    adj[u].push_back(make_pair(v, wt));
    adj[v].push_back(make_pair(u, wt));
}
```



```
void Dijkstra(vector<pair<int,int> > adj[], int V, int src)
{
  priority_queue< iPair, vector <iPair> , greater<iPair> > pq;
  vector<int> dist(V, INF);

    pq.push({0, src});               // push start vertex src
    dist[src] = 0;                   // initial distance of src

    while (!pq.empty())
    { int u = pq.top().second;       // get u from pq.
      pq.pop();

      for (auto x : adj[u])
       {  int v = x.first;             // get v and weight from adj list.
          int weight = x.second;

          if (dist[v] > dist[u] + weight)
            {   dist[v] = dist[u] + weight;   // Update weight of v
                pq.push( {dist[v], v} );     // push v into pq.
            }
        }
    }
}
```

Floyd-Warshall algorithm

- อัลกอริทึมสำหรับการหาเส้นทางที่สั้นที่สุดในกราฟชนิดถ่วงน้ำหนัก
- แตกต่างจากอัลกอริทึม Dijkstra อัลกอริทึม Floyd-warshall จะสำรวจทุกเส้นทางที่สั้นที่สุดระหว่างทุกคู่ของเวอร์เท็กซ์ในกราฟ โดยการรันอัลกอริทึมเพียงครั้งเดียว
- อัลกอริทึม Floyd-warshall จะบันทึกและมีการปรับปรุงคำตอบ (ระยะทางสั้นที่สุดจาก u ไป v) โดยสร้างเมตริกซ์ระยะทาง (distance matrix) D



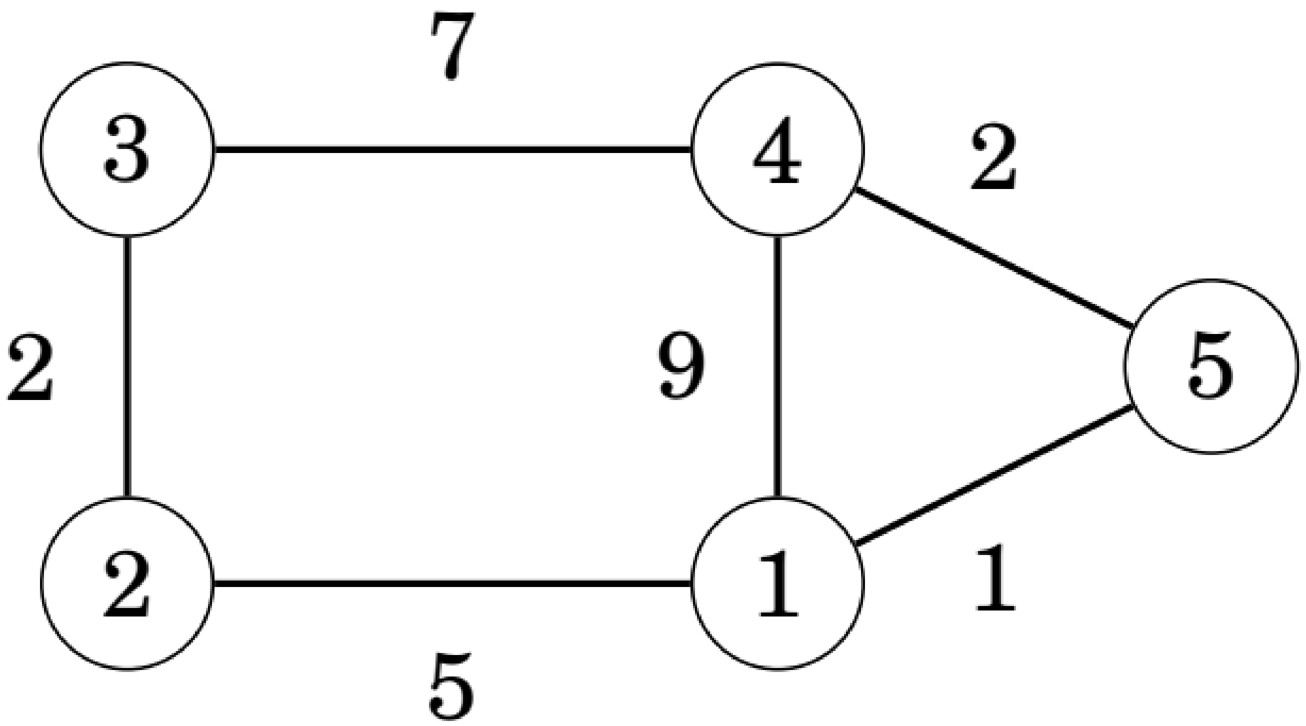
$$D[u][v] = \min \{ D[u][v] ,\ D[u][k] + D[k][u] \}$$

โดยที่ **k** เป็น intermediate vertex

Floyd-Warshall algorithm

กำหนดค่าเริ่มต้นให้กับเมตริกซ์ D ด้วยค่าน้ำหนักในแต่ละเอจด์ โดยที่

- ค่า inf. หากไม่มีเอจด์เชื่อมระหว่าง u และ v โดยตรง
- ระยะทางระหว่างเวกเท็กซ์ u ไป u จะมีค่าเท่ากับ 0

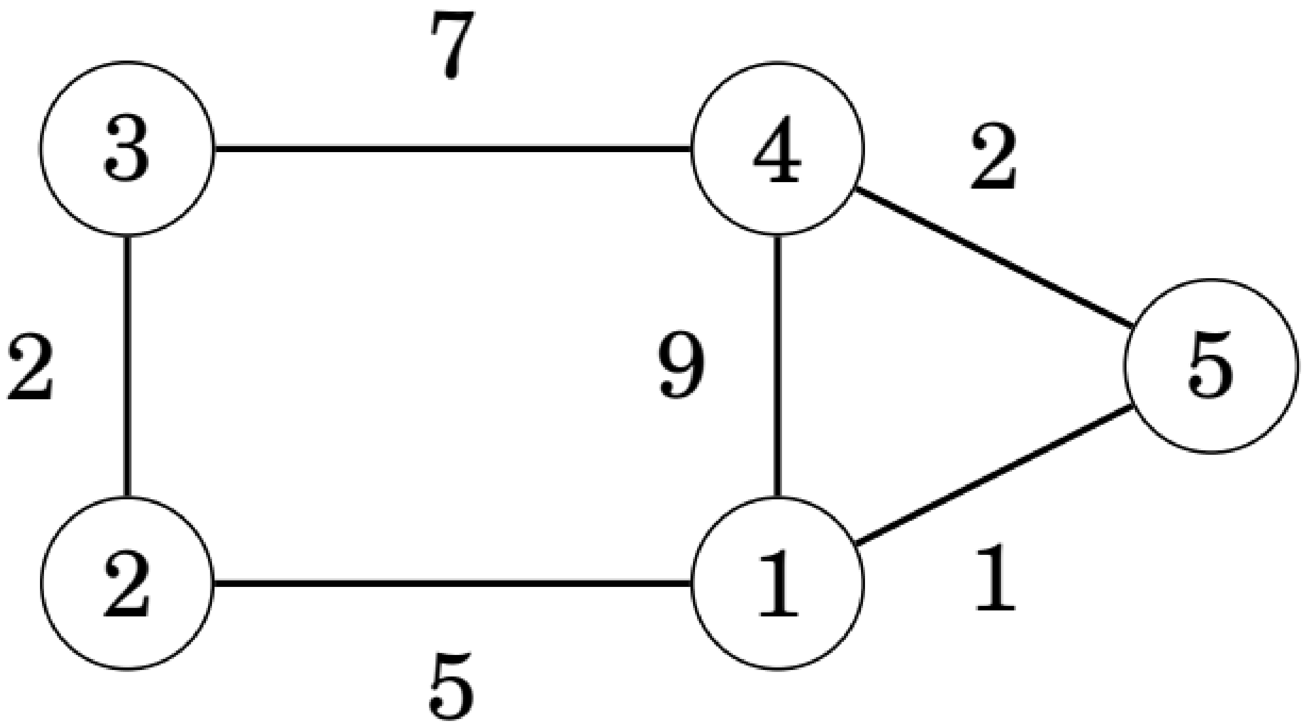


	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	∞	∞
3	∞	2	0	7	∞
4	9	∞	7	0	2
5	1	∞	∞	2	0

Floyd-Warshall algorithm

ปรับปรุงเมตริกซ์ D โดยการเลือกเวอร์เท็กซ์ 1 เป็น intermediate vertex

D[u][v] = min { D[u][v] , D[u][1] + D[1][v] }



$D^{(0)}$

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	∞	∞
3	∞	2	0	7	∞
4	9	∞	7	0	2
5	1	∞	∞	2	0

$D^{(1)}$

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	**14**	**6**
3	∞	2	0	7	∞
4	9	**14**	7	0	2
5	1	**6**	∞	2	0

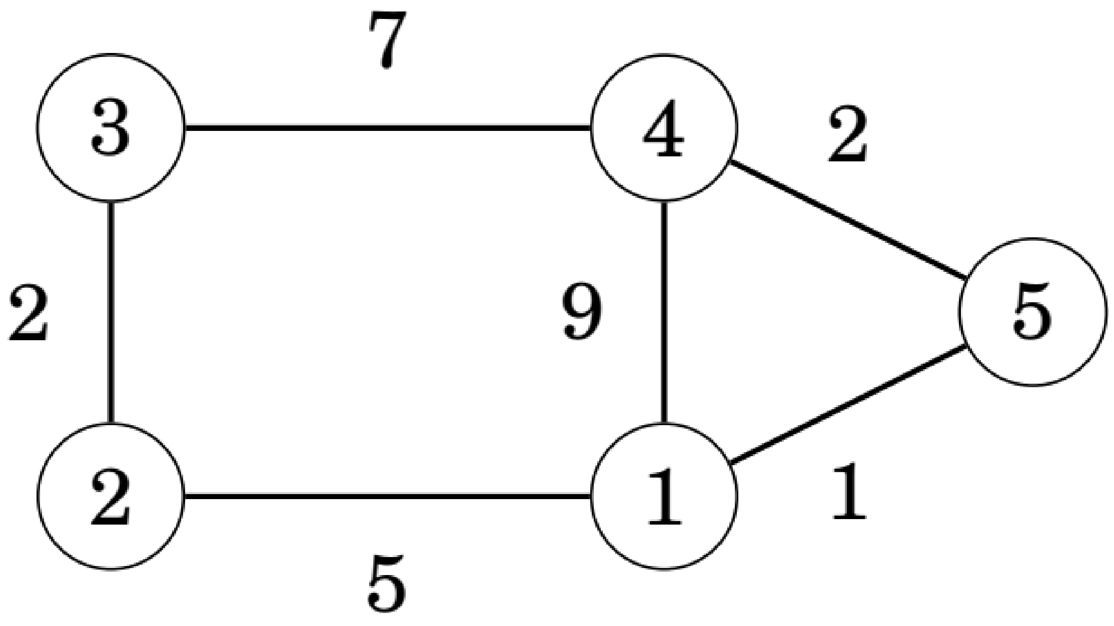
Floyd-Warshall algorithm

ปรับปรุงเมตริกซ์ D โดยการเลือกเวอร์เท็กซ์ 2 เป็น intermediate vertex

D[u][v] = min { D[u][v] , D[u][2] + D[2][v] }

$D^{(1)}$

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	**14**	**6**
3	∞	2	0	7	∞
4	9	**14**	7	0	2
5	1	**6**	∞	2	0



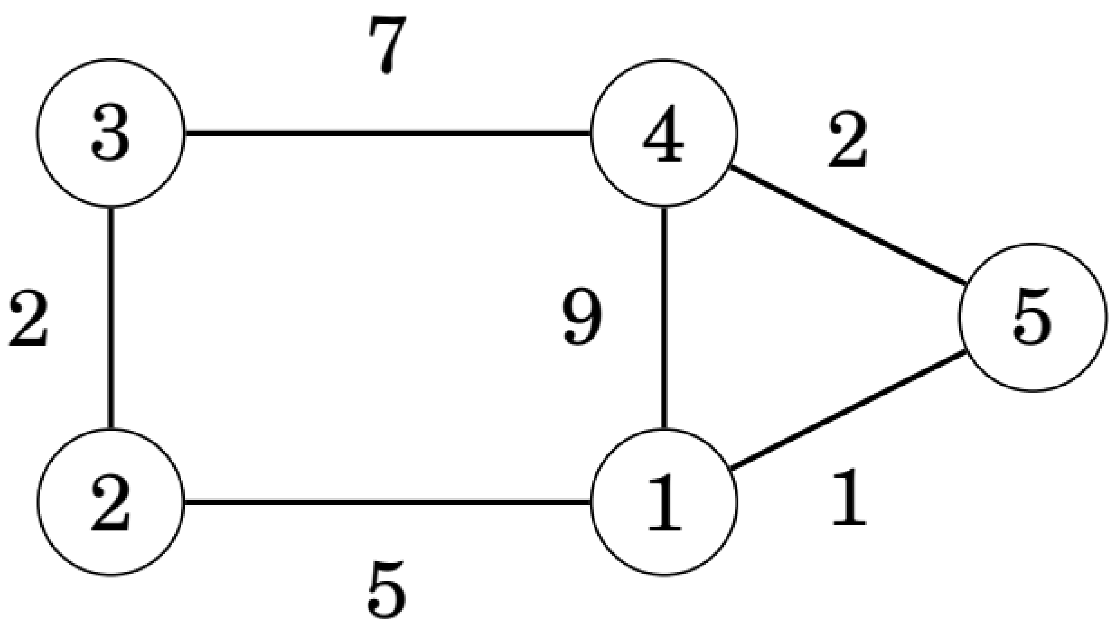
$D^{(2)}$

	1	2	3	4	5
1	0	5	**7**	9	1
2	5	0	2	14	6
3	**7**	2	0	7	**8**
4	9	14	7	0	2
5	1	6	**8**	2	0

Floyd-Warshall algorithm

ปรับปรุงเมตริกซ์ D โดยการเลือกเวอร์เท็กซ์ 3 เป็น intermediate vertex

D[u][v] = min { D[u][v] , D[u][3] + D[3][v] }



$D^{(2)}$

	1	2	3	4	5
1	0	5	**7**	9	1
2	5	0	2	14	6
3	**7**	2	0	7	**8**
4	9	14	7	0	2
5	1	6	**8**	2	0

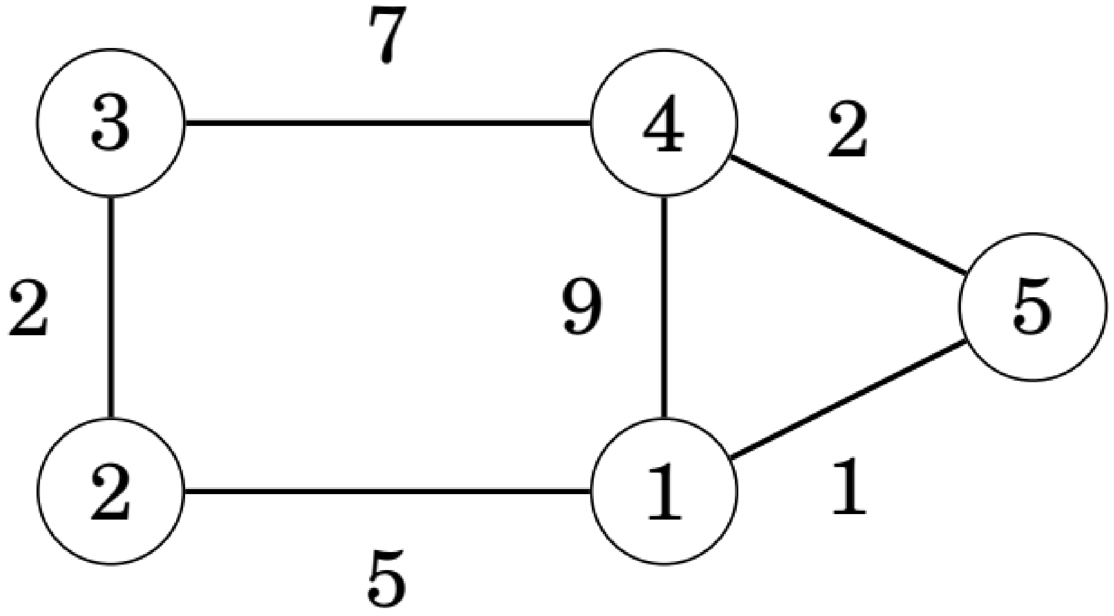
$D^{(3)}$

	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	**9**	6
3	7	2	0	7	8
4	9	**9**	7	0	2
5	1	6	8	2	0

Floyd-Warshall algorithm

ปรับปรุงเมตริกซ์ D โดยการเลือกเวอร์เท็กซ์ 4 เป็น intermediate vertex

D[u][v] = min { D[u][v] , D[u][4] + D[4][v] }



$D^{(3)}$

	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	**9**	6
3	7	2	0	7	8
4	9	**9**	7	0	2
5	1	6	8	2	0

$D^{(4)}$

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

Floyd-Warshall algorithm

```
void Floyd(int dis[][V])
{       for(int k=0; k < V; k++)
         { for(int u=0; u < V; u++)
             { for(int v=0; v < V; v++)
                 if (dis[u][v] > dis[u][k] + dis[k][v]) {
                     dis[u][v] = dis[u][k] + dis[k][v];
                  }
              }
          }
}
```

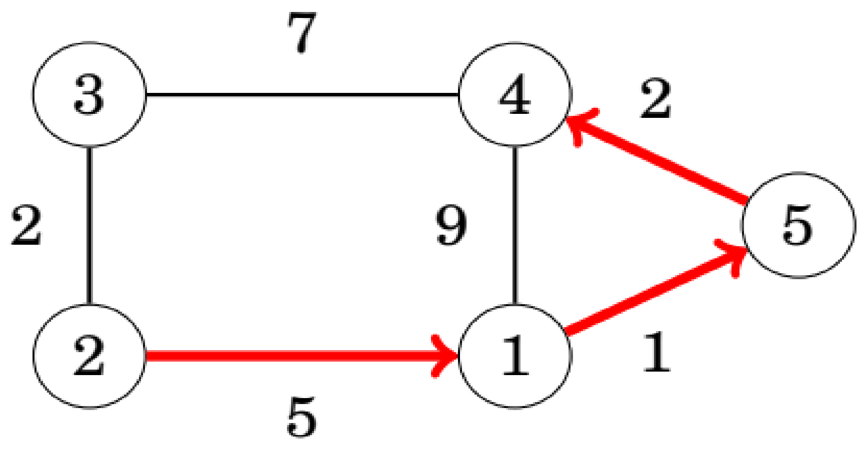
```
void main()
{ int graph[][V] = { {0, 3, inf, 7},
                     {8, 0, 2, inf},
                     {5, inf, 0, 1},
                     {2, inf, inf, 0}};
   Floyd(graph);
}
```

การบันทึกเส้นทางคำตอบ

- ผลลัพธ์ของอัลกอริทึม Floyd-Warshall คือระยะทางสั้นที่สุดในทุกคู่เวอร์เท็กซ์ เช่น เส้นทางสั้นที่สุดจาก 2 ไป 4 เท่ากับ 8

- บันทึก shortest path จาก 2 ไป 4 อย่างไร

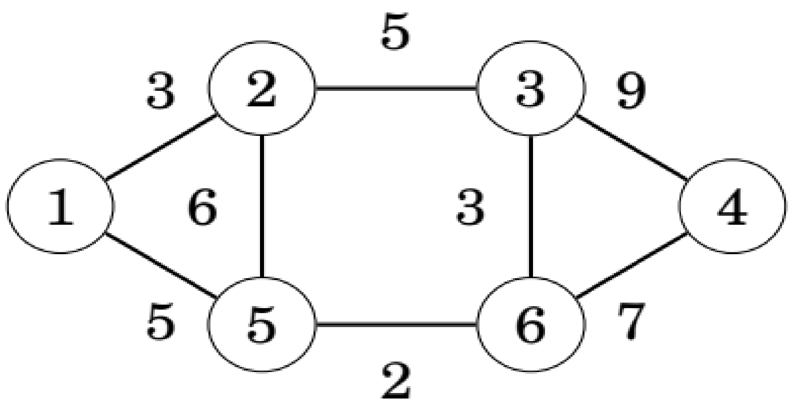
2 -> 1 -> 5 -> 4

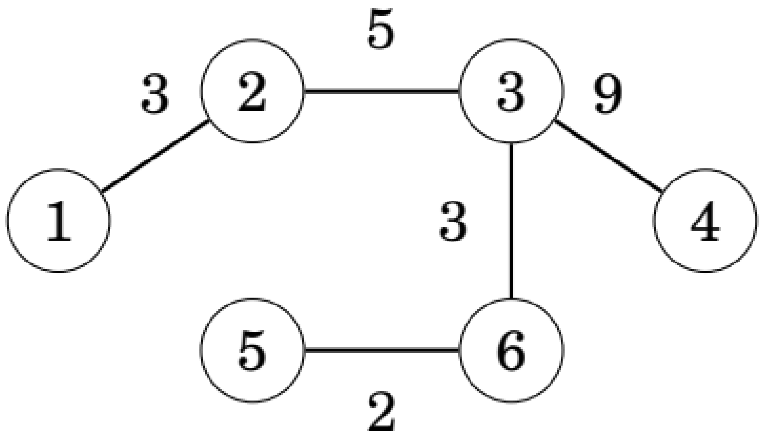


	1	2	3	4	5
1	0	5	7	3	1
2	5	0	2	8	6
3	7	2	0	7	8
4	3	8	7	0	2
5	1	6	8	2	0

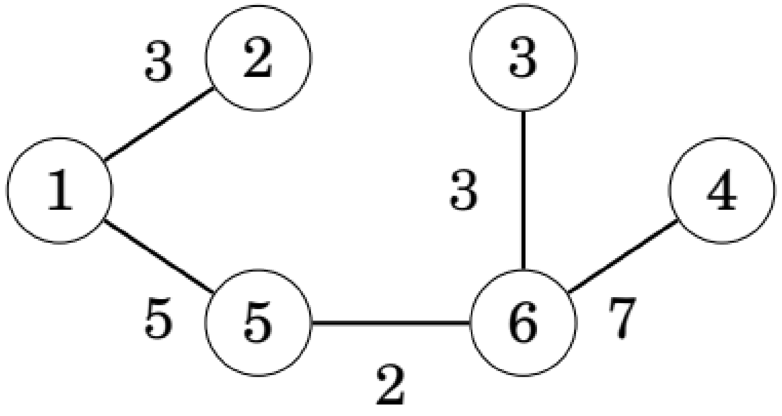
ต้นไม้แผ่ทั่วเล็กที่สุด (Minimum spanning tree)

- **ต้นไม้แผ่ทั่ว (Spanning tree)** ของกราฟ หมายถึง acyclic graph ซึ่งเชื่อมโยงทุกเวอร์เท็กซ์ด้วยเอจด์บางเส้น (|v| - 1)
- **ต้นไม้แผ่ทั่วเล็กที่สุด (Minimum spanning tree)** คือต้นไม้แผ่ทั่วที่ผลรวมของเอจด์มีค่าน้อยที่สุด





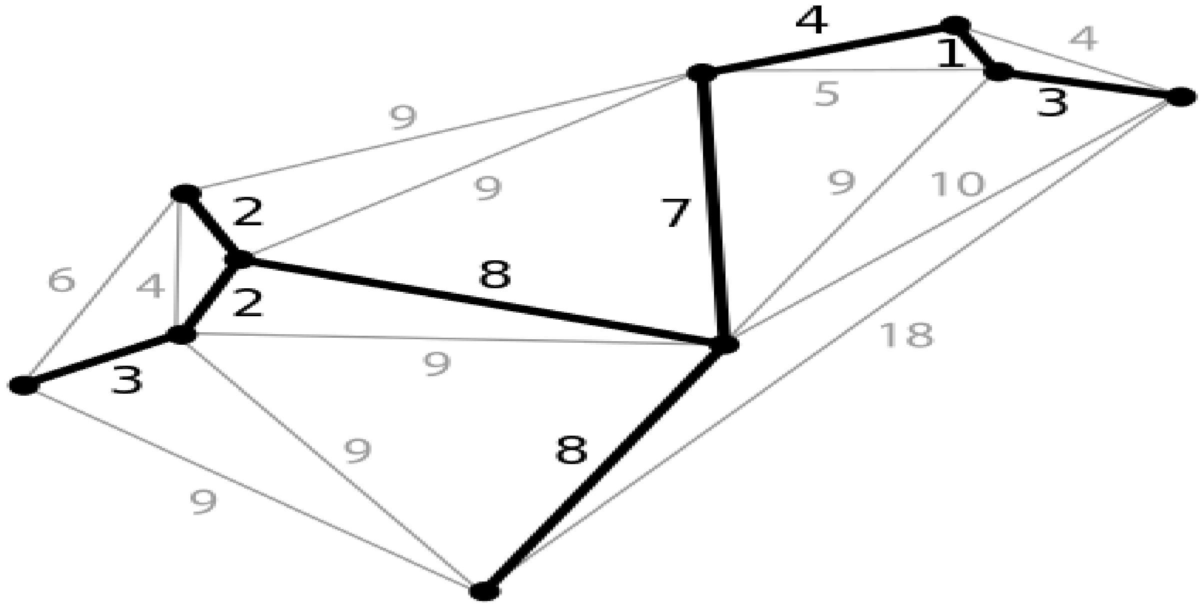
3+5+9+3+2 = 22



3+5+2+3+7 = 20

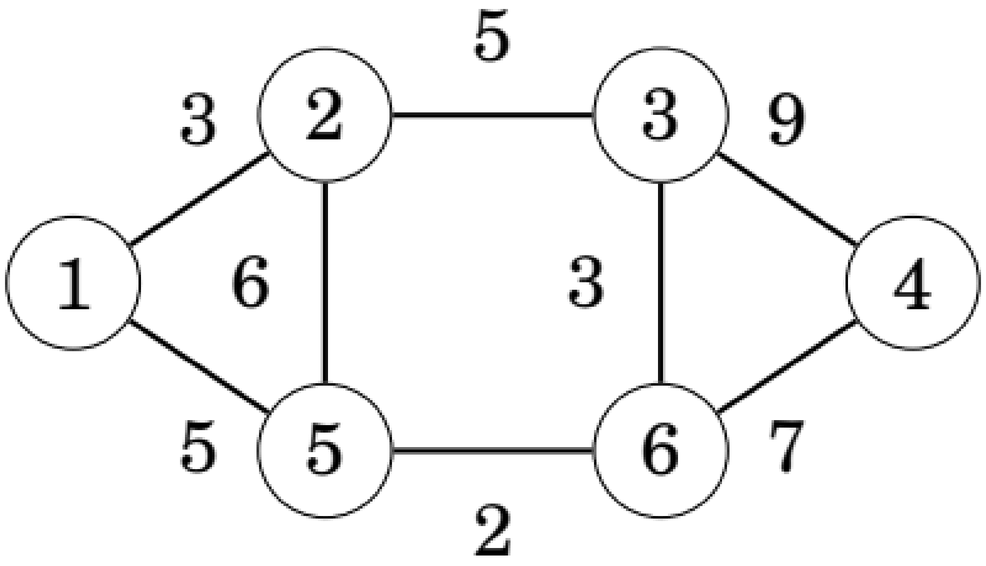
ต้นไม้แผ่ทั่วเล็กที่สุด (Minimum Spanning Tree)

- MST ในกราฟอาจมีได้มากกว่า 1 คำตอบ
- การสำรวจกราฟด้วย BFS /DFS เพื่อหา MST โดยตรงอาจไม่เหมาะ เนื่องจากจำนวนต้นไม้แผ่ทั่วในกราฟมีมากเกินกว่าที่จะสร้างออกมาได้ทั้งหมด (exponential time)
- อัลกอริทึมสำหรับการค้นหาต้นไม้แผ่ทั่วเล็กที่สุด
  - ➢ อัลกอริทึม Prim
  - ➢ อัลกอริทึม Kruskal

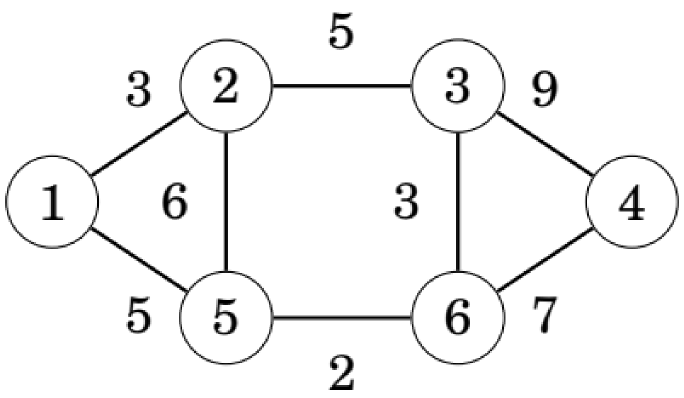


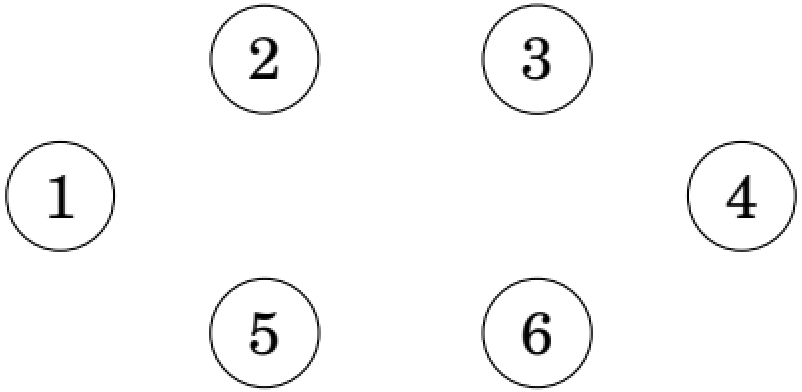
อัลกอริทึม Prim

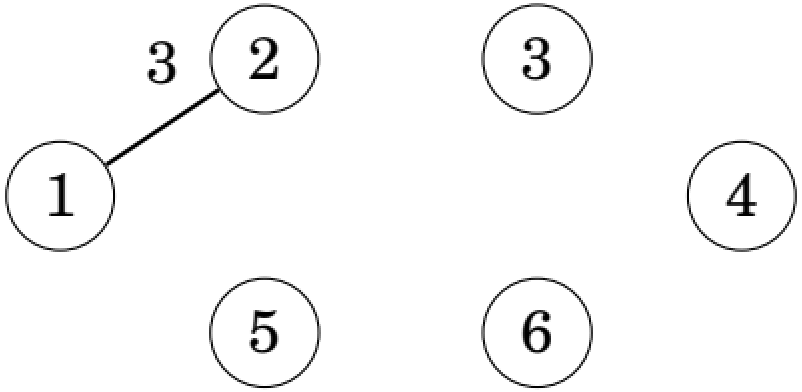
- เป็นอัลกอริทึมค้นหาต้นไม้แผ่ทั่วเล็กที่สุดที่คล้ายคลึงกับอัลกอริทึม Dijkstra
- โดยเริ่มต้นจะต้องกำหนดเวอร์เท็กซ์เริ่มต้นในกราฟ จากนั้นอัลกอริทึมจะเลือกเอจด์ที่มีค่าน้อยสุด (minimum-weight edge) จากเวอร์เท็กซ์เริ่มต้นและแผ่ทั่วจนครอบคลุมทุกเวอร์เท็กซ์
- อัลกอริทึม Prim จะเลือกเอจด์จนครบทุกเวอร์เท็กซ์ (|V| - 1) โดยที่ V หมายถึงจำนวนเวอร์เท็กซ์

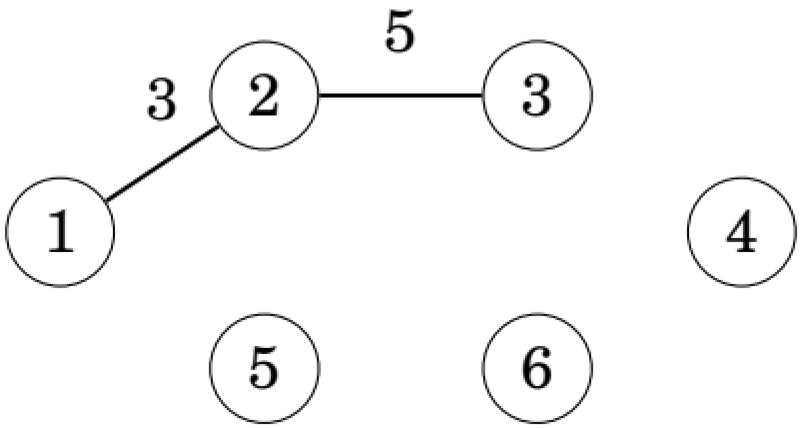


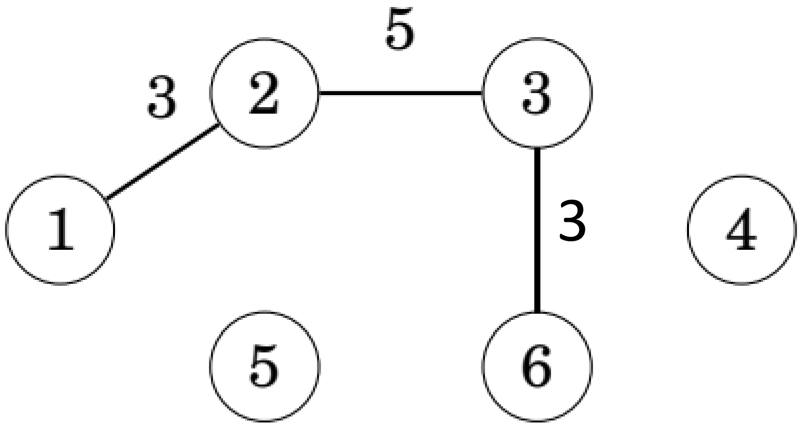
Prim's algorithm



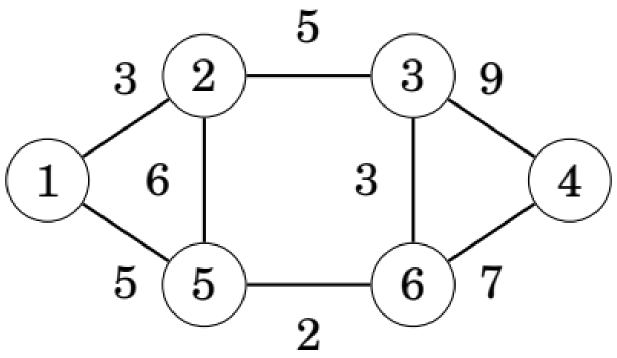


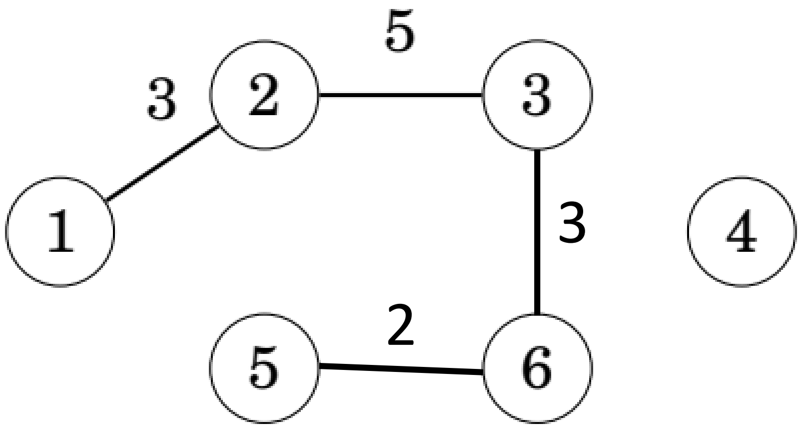


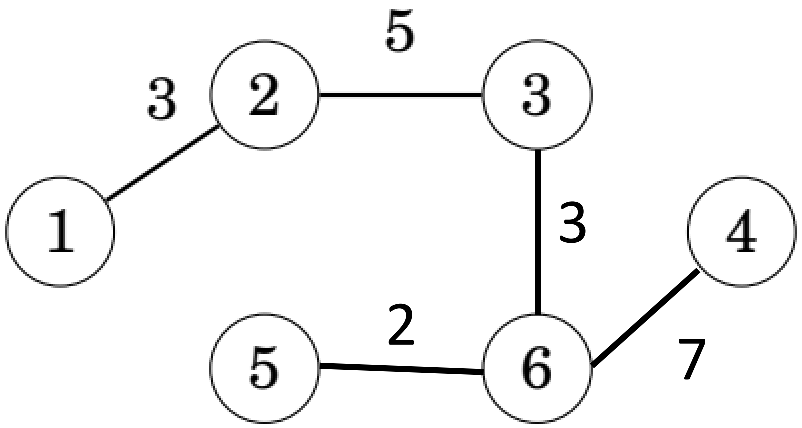




Prim's algorithm





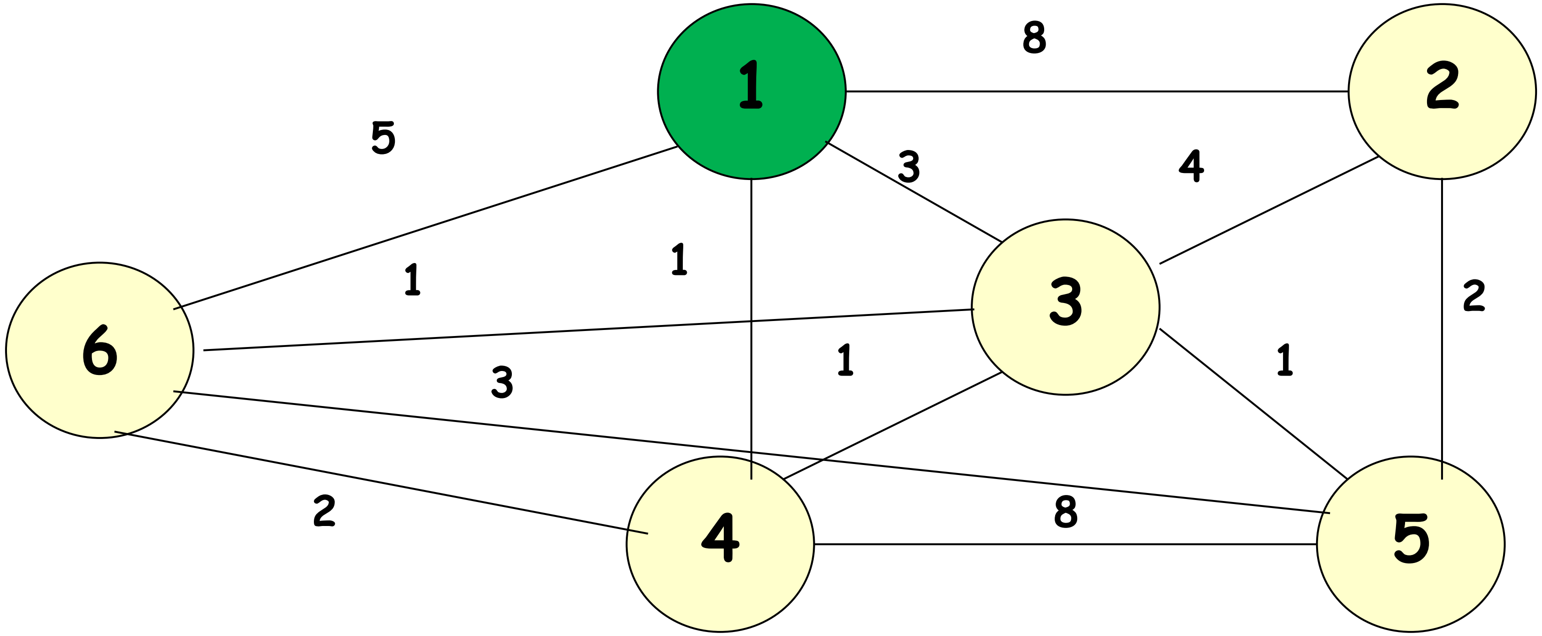


MST

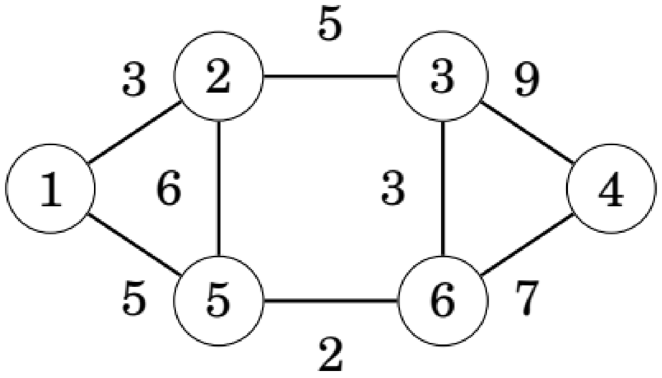
3+5+2+3+7 = 20

ทดสอบ

จงหา minimum spanning tree ของกราฟด้านล่างด้วยวิธี Prim



```
#include<bits/stdc++.h>
using namespace std;
```



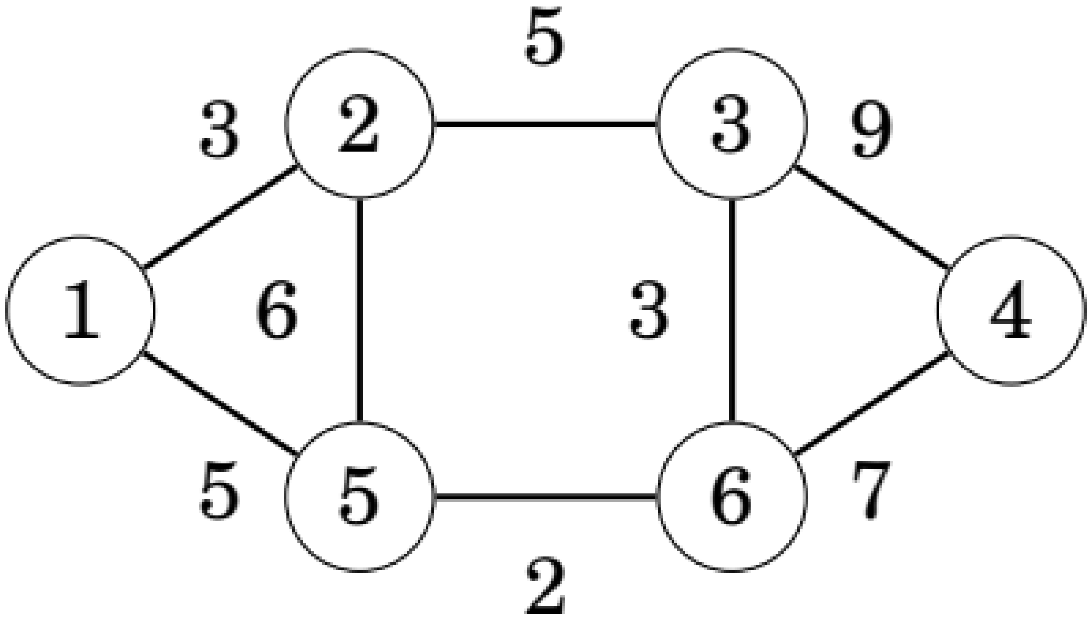

```
typedef pair<int, int> iPair;

int main()
{   int V = 6;
    vector<iPair > adj[V];

    addEdge(adj, 1, 2, 3);
    addEdge(adj, 1, 5, 5);
    addEdge(adj, 2, 5, 6);
    addEdge(adj, 2, 3, 5);
    addEdge(adj, 5, 6, 2);
    addEdge(adj, 3, 6, 3);
    addEdge(adj, 3, 4, 9);
    addEdge(adj, 4, 6, 7);


    PrimMST(adj, V, 1);
}
```

```
void addEdge(vector <pair<int, int> > adj[], int u,
int v, int wt)
{
    adj[u].push_back(make_pair(v, wt));
    adj[v].push_back(make_pair(u, wt));
}
```

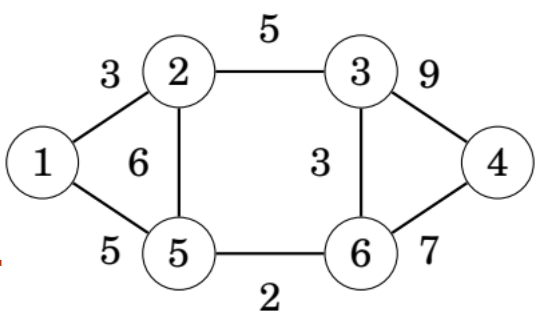
```
primMST(vector<pair<int,int> > adj[], int V, int src)
{ priority_queue< iPair, vector <iPair> , greater<iPair> > pq; // Priority Queue

   vector<int> key(V, INF);                           // keep distance of vertex
   vector<int> parent(V, -1);                         // keep path of MST
   vector<bool> inMST(V, false);                      // keep status of vertex

    pq.push(make_pair(0, src));
    key[src] = 0;
    while (!pq.empty())
    { int u = pq.top().second;                              // get u from pq.
      pq.pop();

      if(inMST[u] == true)  continue;
      inMST[u] = true;                                     //  add u into MST

      list< pair<int, int> >::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
          { int v = (*i).first;                          //get v and weight from adj list of u
            int weight = (*i).second;

            if (inMST[v] == false && key[v] > weight)
              {  key[v] = weight;
                 pq.push(make_pair(key[v], v));                  // push v into pq.
                 parent[v] = u;
              }
          }
    }
}
```

Kruskal’s algorithm

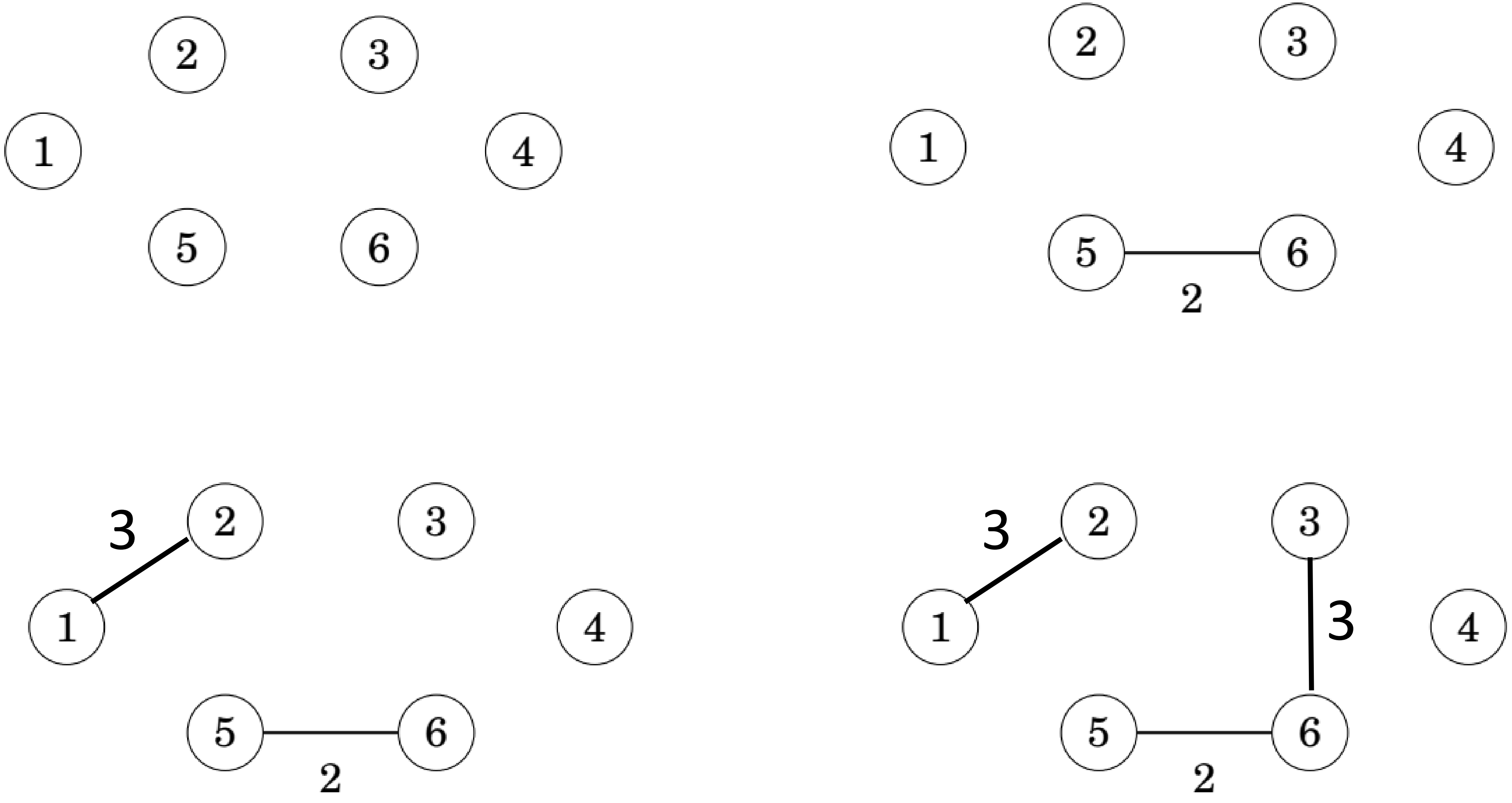
- เริ่มต้นจากเรียงลำดับเอดจ์ (edge) ตามค่าน้ำหนักจากน้อยไปมาก
- เพิ่มเอดจ์เข้าไปในคำตอบ หากเอดจ์นั้นไม่ทำให้เกิด **วงวน** (cycle) จนกระทั่งครอบคลุมทุกเวอร์เท็กซ์



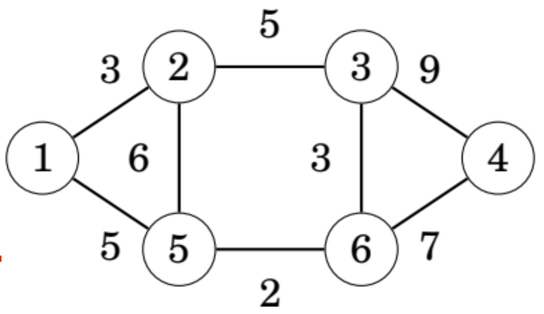
Kruskal’s algorithm

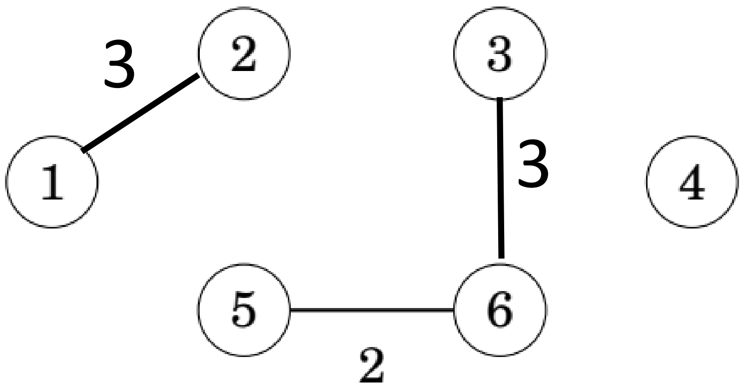


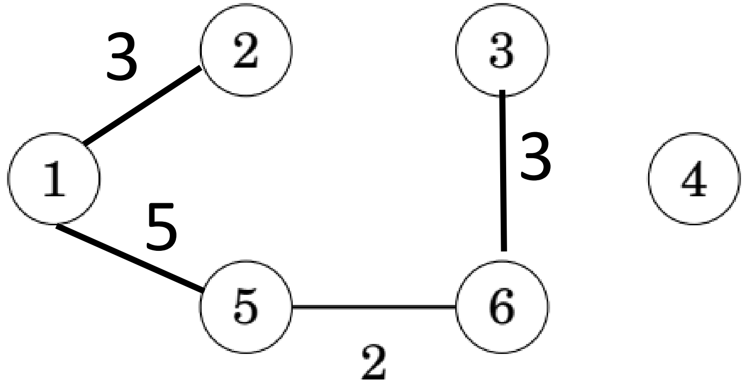
edge	weight
5–6	2
1–2	3
3–6	3
1–5	5
2–3	5
2–5	6
4–6	7
3–4	9



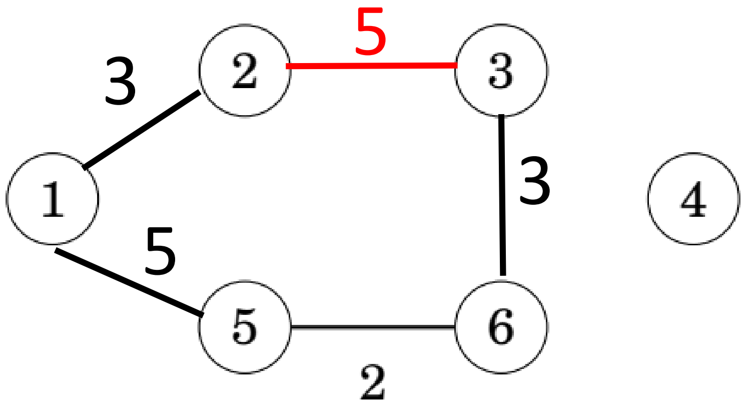
Kruskal's algorithm

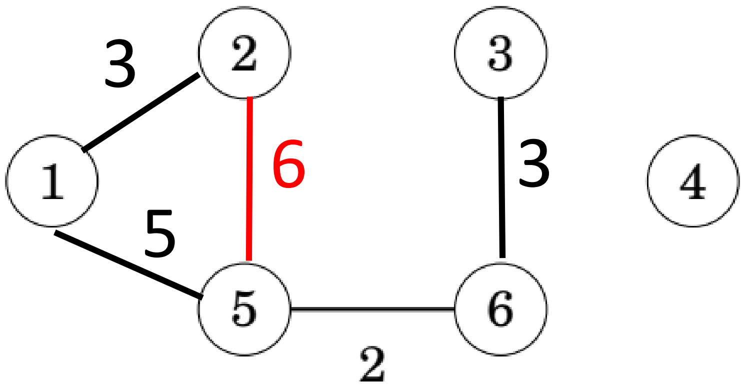




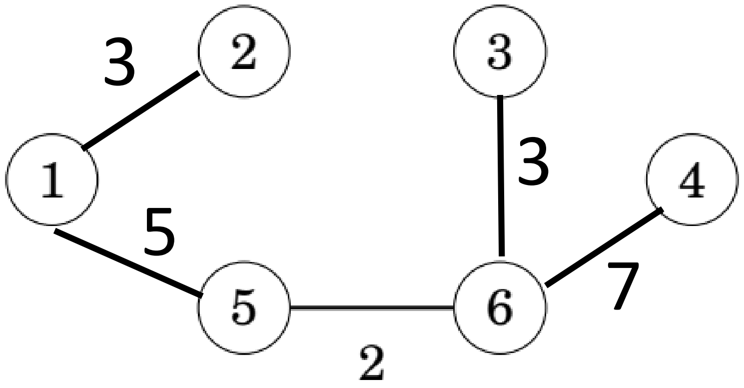


edge	weight
5–6	2
1–2	3
3–6	3
1–5	5
2–3	5
2–5	6
4–6	7
3–4	9



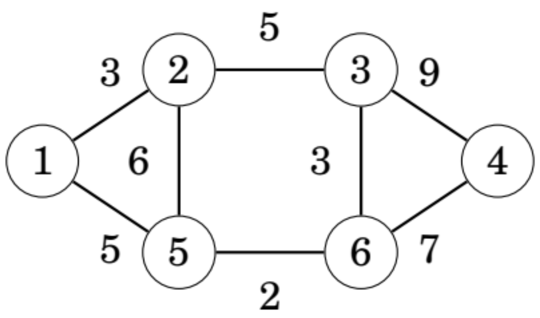


Kruskal’s algorithm



MST

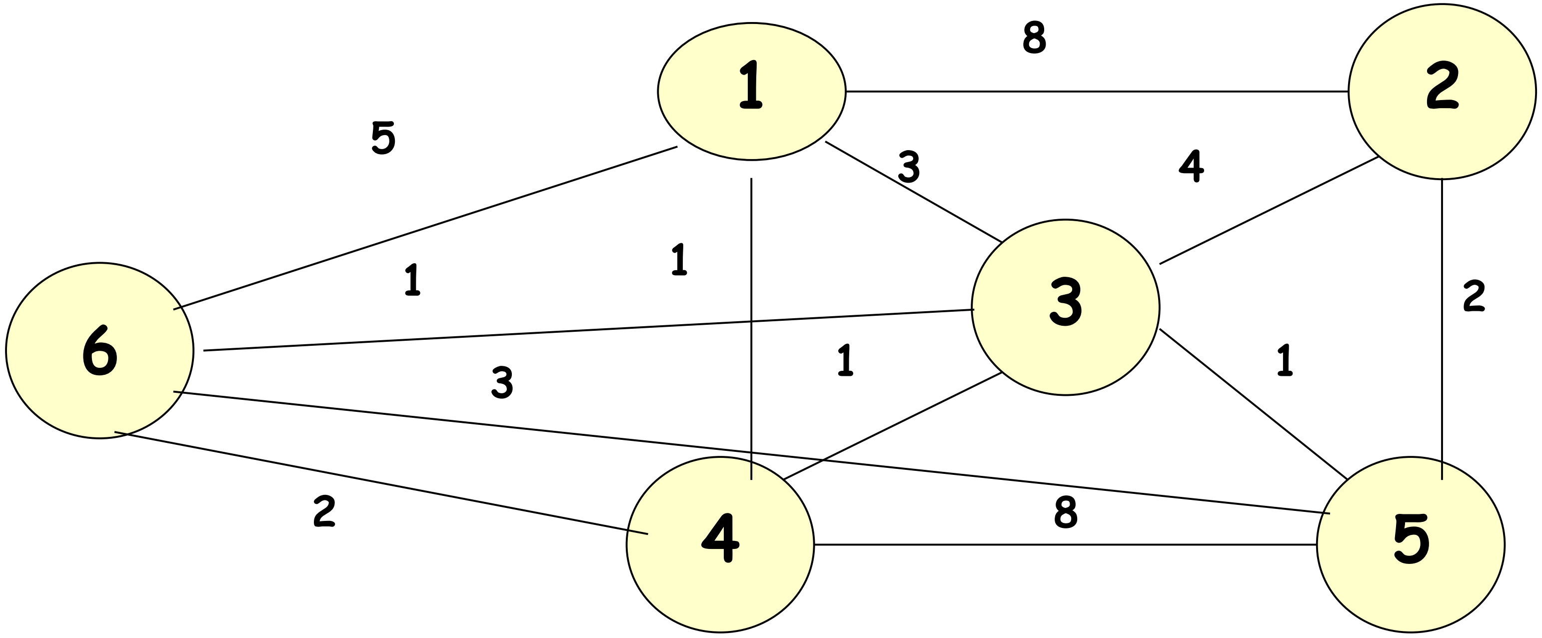
3+5+2+3+7 = 20



edge	weight
5–6	2
1–2	3
3–6	3
1–5	5
2–3	5
2–5	6
4–6	7
3–4	9

ทดสอบ

จงหา minimum spanning tree ของกราฟด้านล่างด้วยวิธี Kruskal

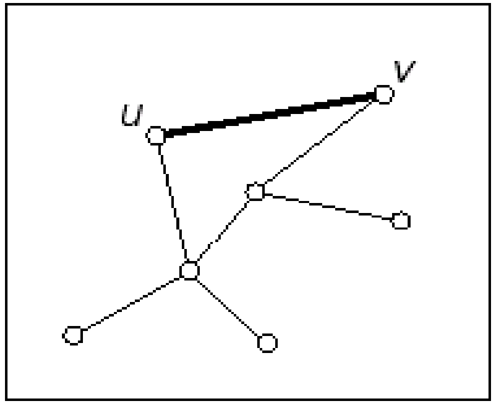


Kruskal's algorithm

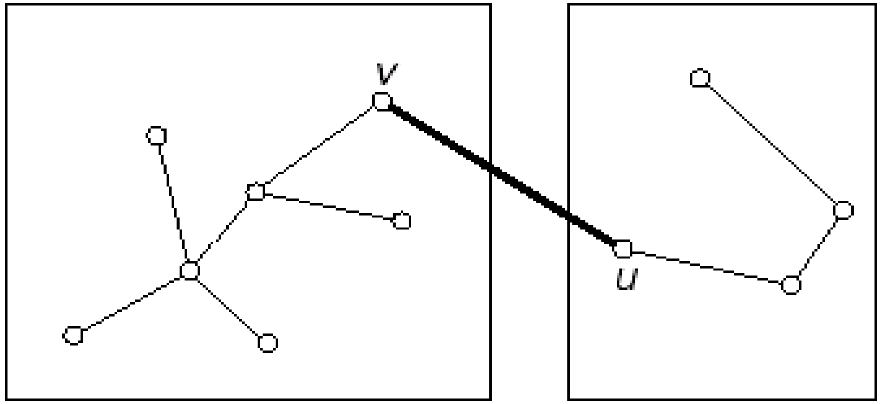
```
//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G = ⟨V, E⟩
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights w(e_{i_1}) ≤ ··· ≤ w(e_{i_{|E|}})
E_T ← ∅;   ecounter ← 0      //initialize the set of tree edges and its size
k ← 0                        //initialize the number of processed edges
while ecounter < |V| − 1 do
    k ← k + 1
    if E_T ∪ {e_{i_k}} is acyclic
        E_T ← E_T ∪ {e_{i_k}};   ecounter ← ecounter + 1
return E_T
```

การตรวจสอบวงวน

- พิจารณาว่าแต่ละเวอร์เท็กซ์ในกราฟเป็นต้นไม้ย่อย (sub-tree)
- ทุกครั้งที่เพิ่มเอจด์ e_{uv} ต้องเช็คว่า u และ v อยู่ในต้นไม้ย่อยเดียวกันหรือไม่



(a)

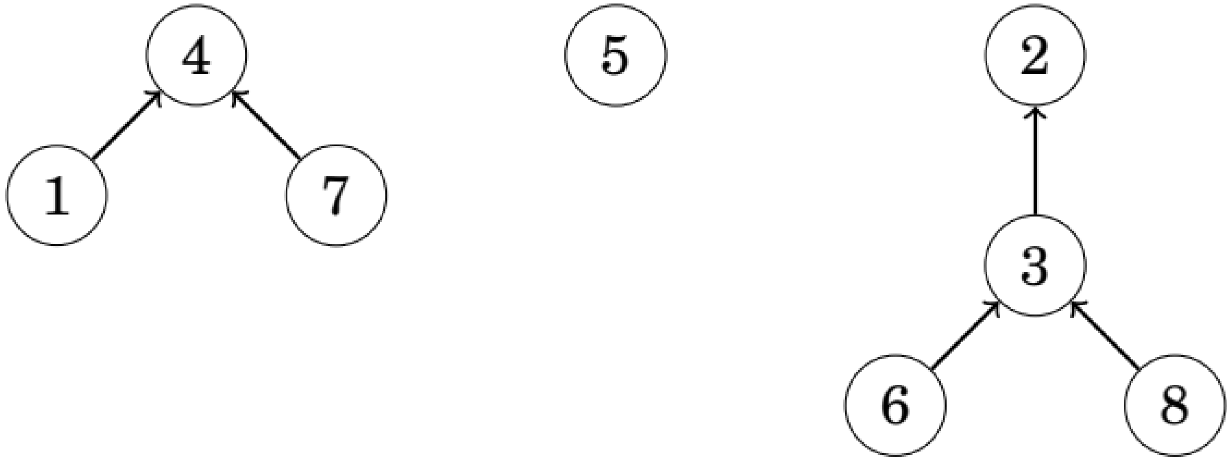


(b)

- ต้องการอัลกอริทึมที่มีประสิทธิภาพที่ใช้ในการตรวจสอบว่าเวอร์เท็กซ์ใดบ้างที่อยู่ในต้นไม้ย่อยเดียวกันบ้าง ซึ่งก็คือ Union-Find structure

อัลกอริทึม Union-Find

- Union-find เป็นอัลกอริทึมที่ถูกออกแบบมาเพื่อจัดการกับปัญหา disjoint sets เช่น {1,4,7}, {5}, {2,3,6,8}
- โดยแสดง disjoint sets ให้อยู่ในรูปของต้นไม้ย่อยและกำหนดให้สมาชิกตัวหนึ่งในแต่ละเซตเป็นตัวแทนของเซตนั้น (root)



	1	2	3	4	5	6	7	8
parent	4	-1	2	-1	-1	3	4	3

โอเปอเรชั่น Find

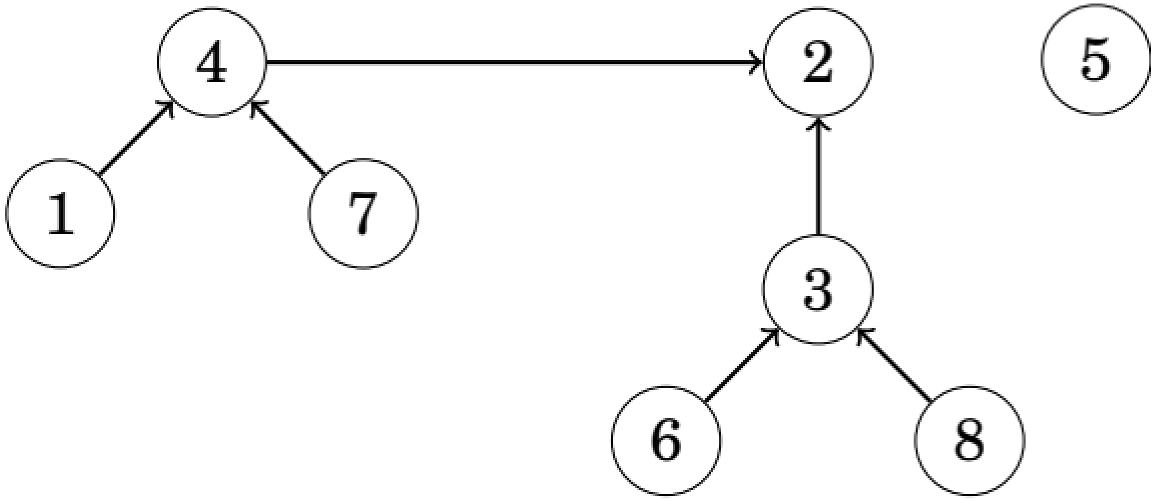
- ทำหน้าที่ค้นหาตัวแทนของสมาชิกในเซต เช่น ต้องการทราบว่า 6 มีตัวแทนเป็นใคร หรือ 7 มีตัวแทนเป็นใคร


```
int find(int parent[], int i)
{ if(parent[i] == -1)
      return i;
   return find(parent, parent[i]);
}
```

	1	2	3	4	5	6	7	8
parent	4	-1	2	-1	-1	3	4	3

โอเปอเรชั่น Union

- ทำหน้าที่รวมสองเซตย่อยเข้าด้วยกัน เช่น ต้องการรวม {1, 4, 7} และ {2, 3, 6, 8} เป็นเซตเดียวกัน
- วิธีการคือเปลี่ยนสถานะตัวแทนตัวใดตัวหนึ่งในอาร์เรย์ parent ให้เป็นลูกของตัวแทนอีกตัวหนึ่ง เช่น 4 จะกลายเป็นลูกของ 2 ซึ่งจะได้ผลลัพธ์เป็น {1, 4, 7, 2, 3, 6, 8}




```
void union(int parent[],int i, int j)
{    parent[j] = i;
}
```

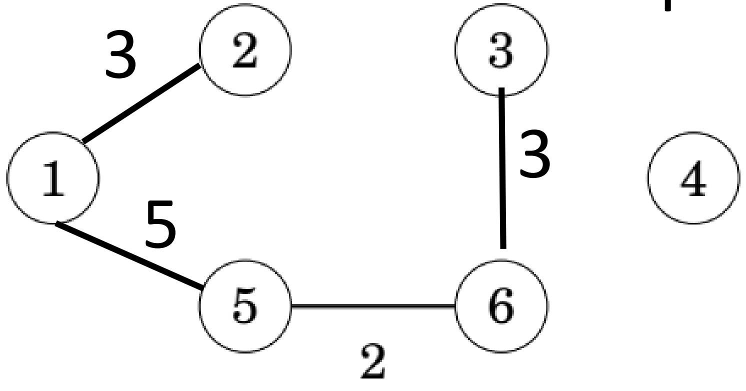
	1	2	3	4	5	6	7	8
parent	4	-1	2	2	-1	3	4	3

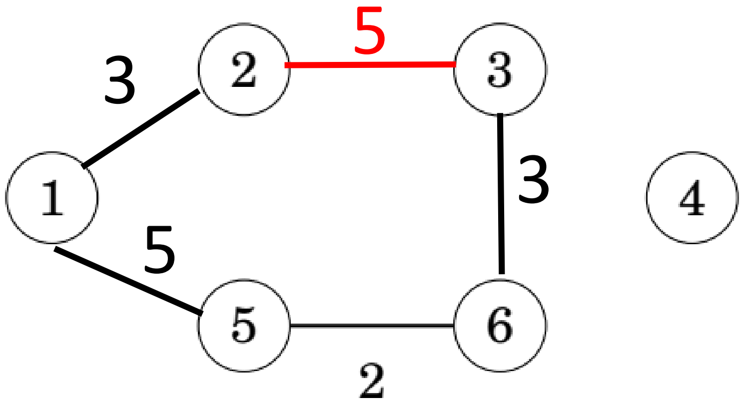
การตรวจสอบวงวนด้วย Union-find

- สร้างฟังก์ชั่น is_cycle() เพื่อตรวจสอบเอดจ์ (u, v) ว่าอยู่ในเซตเดียวกันหรือไม่
- สมมุติให้ เอจด์ (2, 3) ถูกเลือกเป็นลำดับถัดไป
- is_cycle(2, 3) = ???

```
int is_cycle(int u, int v)
{   x = find(parent, u);
    y = find(parent, v);
  return (x==y)? true:false;
}
```

	1	2	3	4	5	6
parent	2	-1	6	-1	1	5





Kruskal's algorithm

```
void kruskal(int g[][V])
{ int parent[V], ne=0, a, b, u, v;
  for(i=0; i<V; i++)
     parent[V] = -1;

 while(ne < V)
     { for (int i = 0, min = inf; i < V; i++)
           for (int j = 1; j<V; j++)
              if (g[i][j] < min)
                  {  min = g[i][j];
                       a = u = i;
                       b = v = j;
                  }

              if ( !is_cycle(u, v) )
              { ne++;
                union(parent, u, v);
              }
          g[a][b] = g[b][a] = inf;
     }
}
```

```
int find(int parent[], int i)
  {  if(parent[i] == -1)
           return i;
     return find(parent, parent[i]);
  }
```

```
void union(int parent[],int i, int j)
 {  parent[j] = i;
 }
```

```
int is_cycle(int u, int v)
{   x = find(parent, u);
    y = find(parent, v);
  return (x==y)? true:false;
}
```