

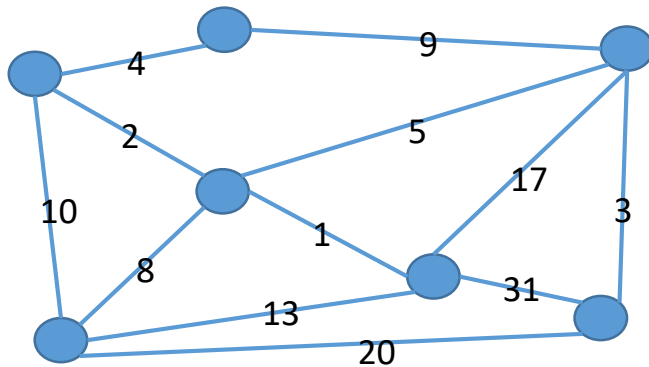
Minimum Spanning Tree

Minimum Spanning Tree

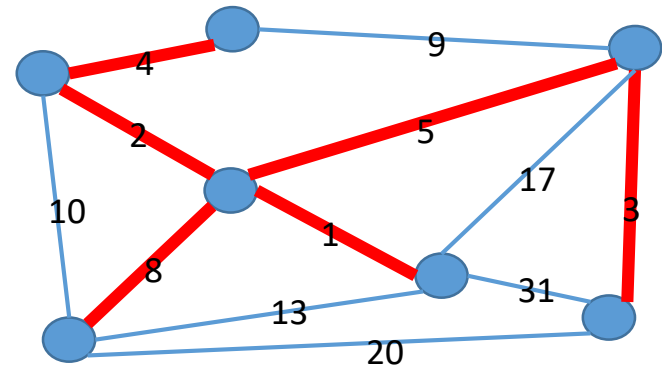
- Minimum Spanning Tree Problem (MST)
- Cut property and Cycle property
- MST algorithms
 - Prim
 - Kruskal and Union-Find

Minimum Spanning Tree

- กำหนด graph $G=(V, E)$ ที่แต่ละเส้นเชื่อม $e \in E$ มีน้ำหนัก c_e
- MST เป็น subset ของเส้นเชื่อม $T \subseteq E$ ที่
 - T เป็นต้นไม้ (ไม่มีวงจร)
 - T เชื่อมกับทุก ๆ โหนด
 - ผลรวมของน้ำหนักบนเส้นเชื่อมต่ำที่สุด



$G=(V,E)$



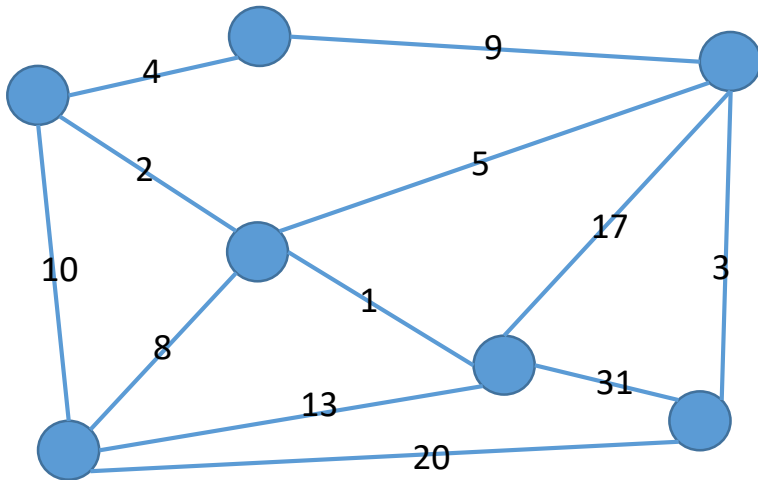
$T, \sum_{e \in T} c_e = 23$

Applications

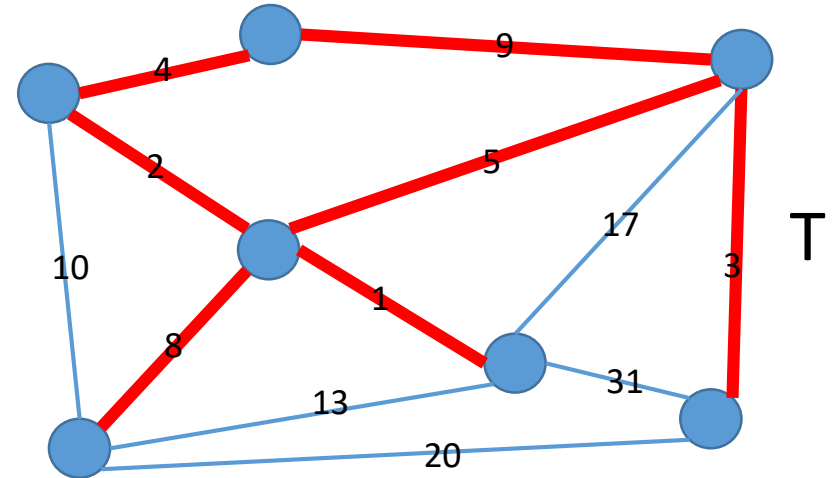
- Network design
 - Telephone, electrical, cable TV
- Approximation algorithms for NP-hard problem
 - Traveling salesman problem
- Indirect applications
 - peer to peer streaming
 - data mining
 - clustering
 - ...

Minimum Spanning Tree

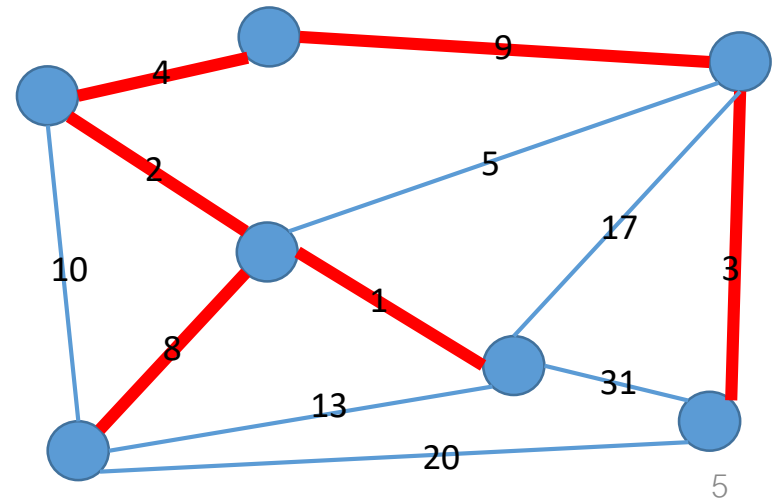
- ในรูปเป็น minimum spanning tree หรือไม่



$G=(V,E)$



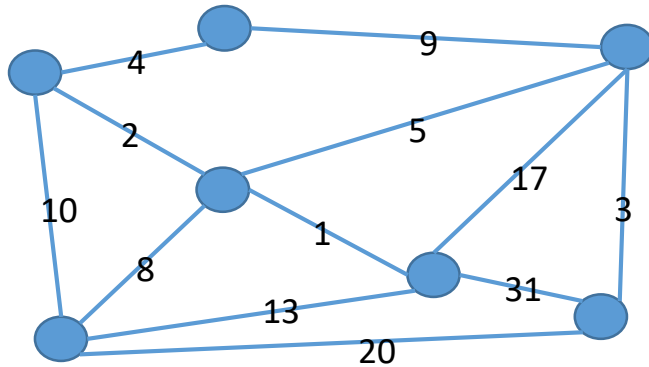
T



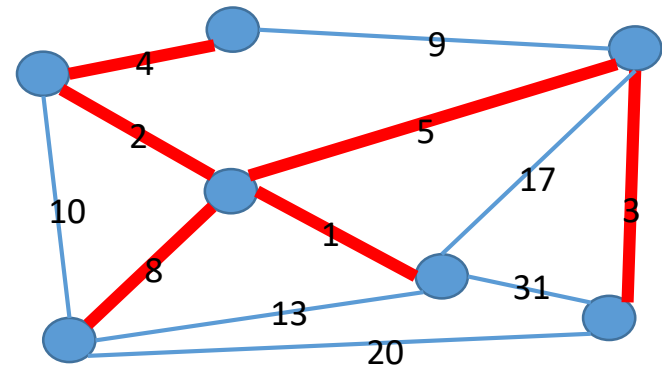
T'

Minimum Spanning Tree

- กำหนด graph $G=(V, E)$ ที่แต่ละเส้นเชื่อม $e \in E$ มีน้ำหนัก c_e
- MST เป็น subset ของเส้นเชื่อม $T \subseteq E$ ที่
 - T เป็นต้นไม้
 - T เชื่อมกับทุกๆ โหนด
 - ผลรวมของน้ำหนักบนเส้นเชื่อมต่ำที่สุด



$G=(V,E)$



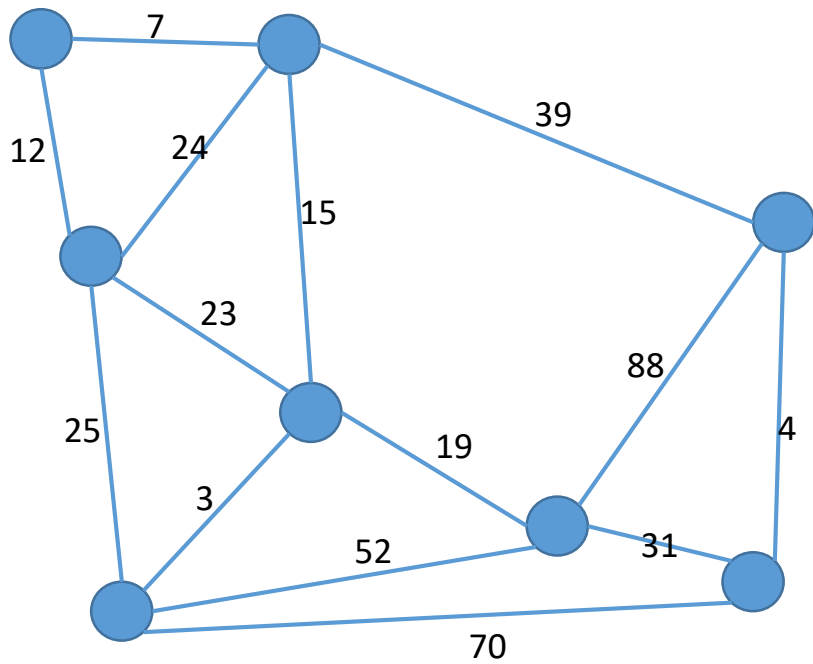
$$T, \sum_{e \in T} c_e = 23$$

- เราจะหา minimum spanning tree แบบ greedy ได้อย่างไร

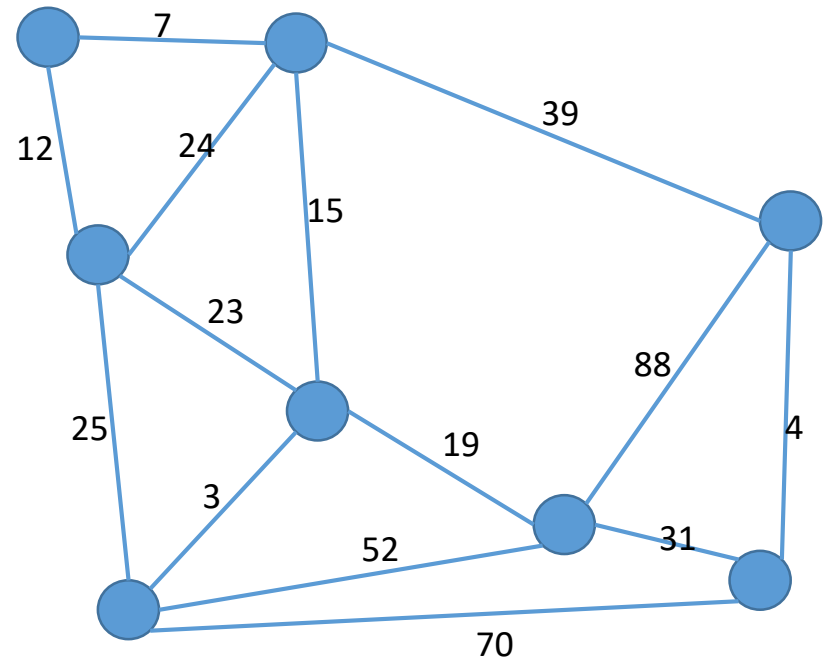
อัลกอริทึมเชิงละโมบ

- Kruskal's algorithm
 - เริ่มต้นด้วย $T = \emptyset$ จากนั้นพิจารณาเส้นเชื่อมทีละเส้นตามลำดับน้ำหนักน้อยไปมาก เพิ่มเส้นเชื่อม e ไปใน T หากไม่ทำให้เกิดวงจร (cycle)
- Prim's algorithm
 - เริ่มต้นด้วยโหนด s หลังจากนั้นจะสร้าง T โดยขยายตัวออกจาก s ในแต่ละขั้นจะเพิ่มเส้นเชื่อม e ที่มีน้ำหนักน้อยที่สุดออกจาก T ที่เส้นเชื่อมนั้นมีจุดปลายด้านหนึ่งติดกับโหนดใน T

ตัวอย่าง



Kruskal's algorithm



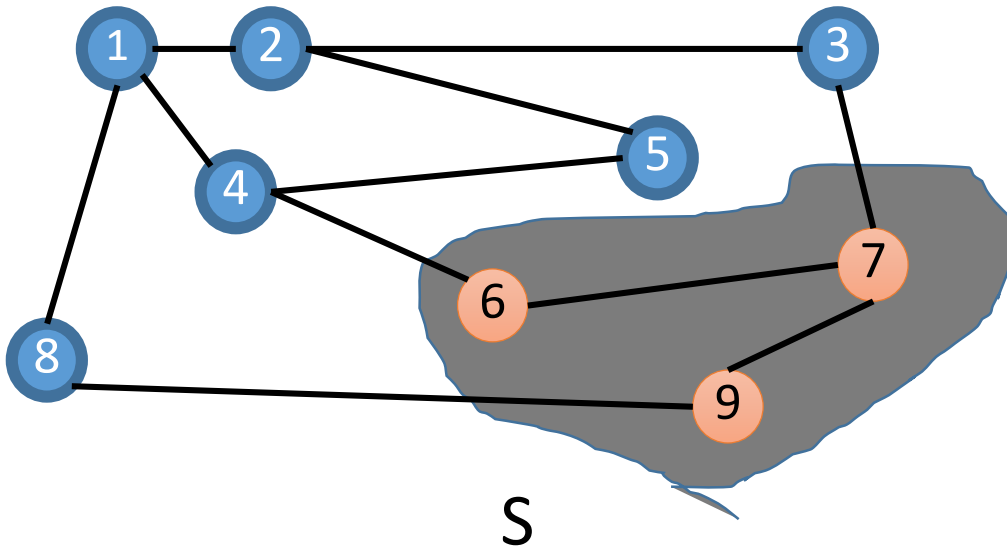
Prim's algorithm

พิสูจน์อัลกอริทึม

- เราจะพิสูจน์อัลกอริทึมเหล่านี้ด้วยคุณสมบัติ 2 อย่าง
 - Cut property
 - Cycle property

Cut และ Cut set

- Cut: Cut คือ subset ของโหนด
- Cutset: Cutset D ของ Cut S คือ subset ของเส้นเชื่อมที่มีจุดปลายด้านหนึ่งอยู่ใน S

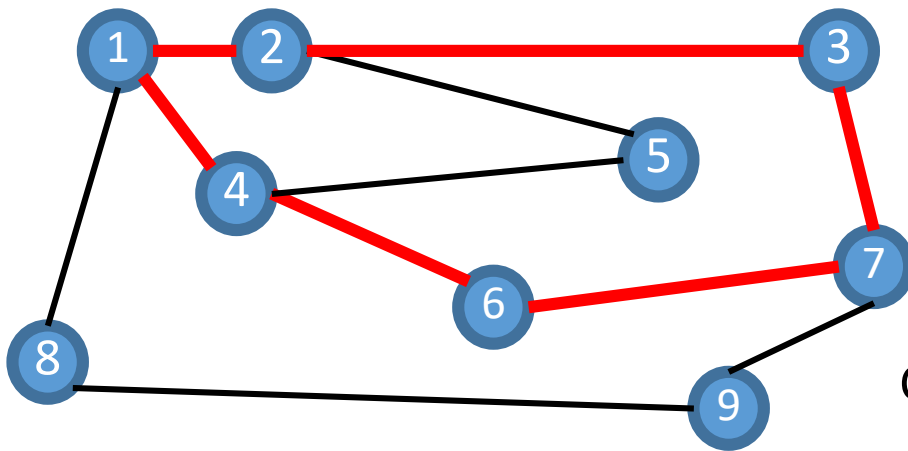


Cut C = {6, 7, 9}

Cutset D = 4-6, 3-7, 8-9

Cycle

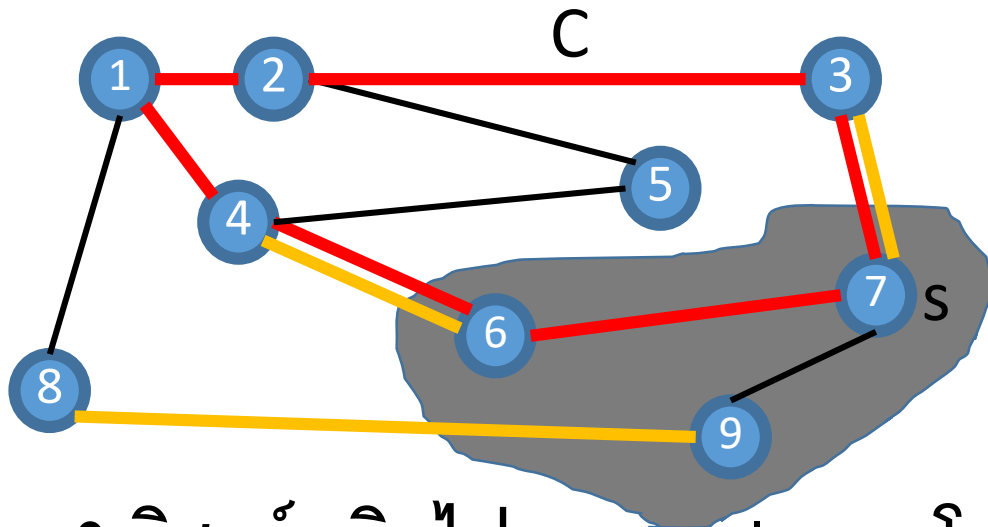
- Cycle: Cycle คือเซตของเส้นเชื่อมที่เชื่อมต่อกันเป็นวงจรและจุดปลายเป็นจุดเดียวกัน



Cycle C = 1-2, 2-3, 3-7, 7-6, 6-4, 4-1

Cycle-Cut Intersection

- พิจารณาการทับกันระหว่าง cycle กับ cutset จำนวนของเส้นเชื่อมที่ทับกันเป็นเท่าไร (1, 2, คี่, คู่)



Cutset $D = 4-6, 3-7, 8-9$

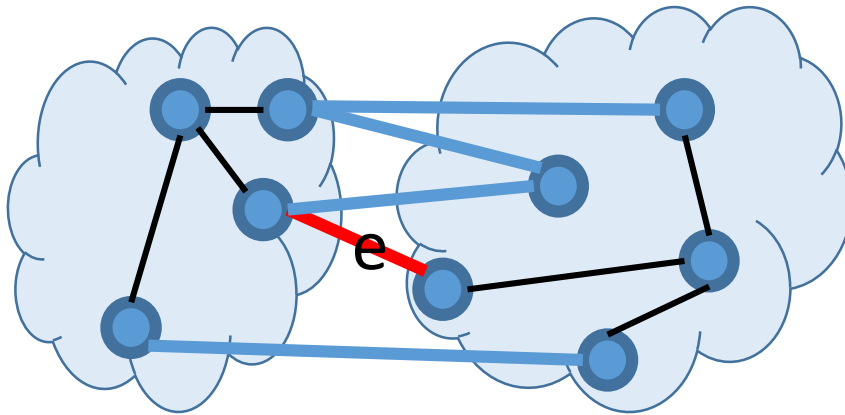
Cycle $C = 1-2, 2-3, 3-7, 7-6, 6-4, 4-1$

Intersection = $3-7, 6-4$

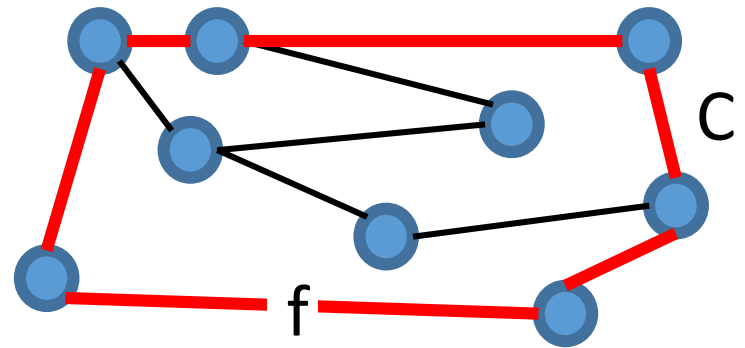
- พิสูจน์ เดินไปตาม cycle จากโหนด $s \in S$ พบว่าทุก ๆ เส้นเชื่อมที่ออกจาก S จะมีเส้นเชื่อมที่เข้า S ก่อนจะไปถึง s

Cut and Cycle property

- Cut Property ให้ S แทน cut ใดๆ และให้ e แทนเส้นเชื่อมที่มีน้ำหนักน้อยที่สุดที่มีจุดปลายด้านหนึ่งอยู่ใน S แล้ว MST จะมีเส้นเชื่อม e อยู่
- Cycle property ให้ C แทน cycle ใดๆ และให้ f แทนเส้นเชื่อมที่มีน้ำหนักมากที่สุดใน C แล้ว MST จะไม่มีเส้นเชื่อม f



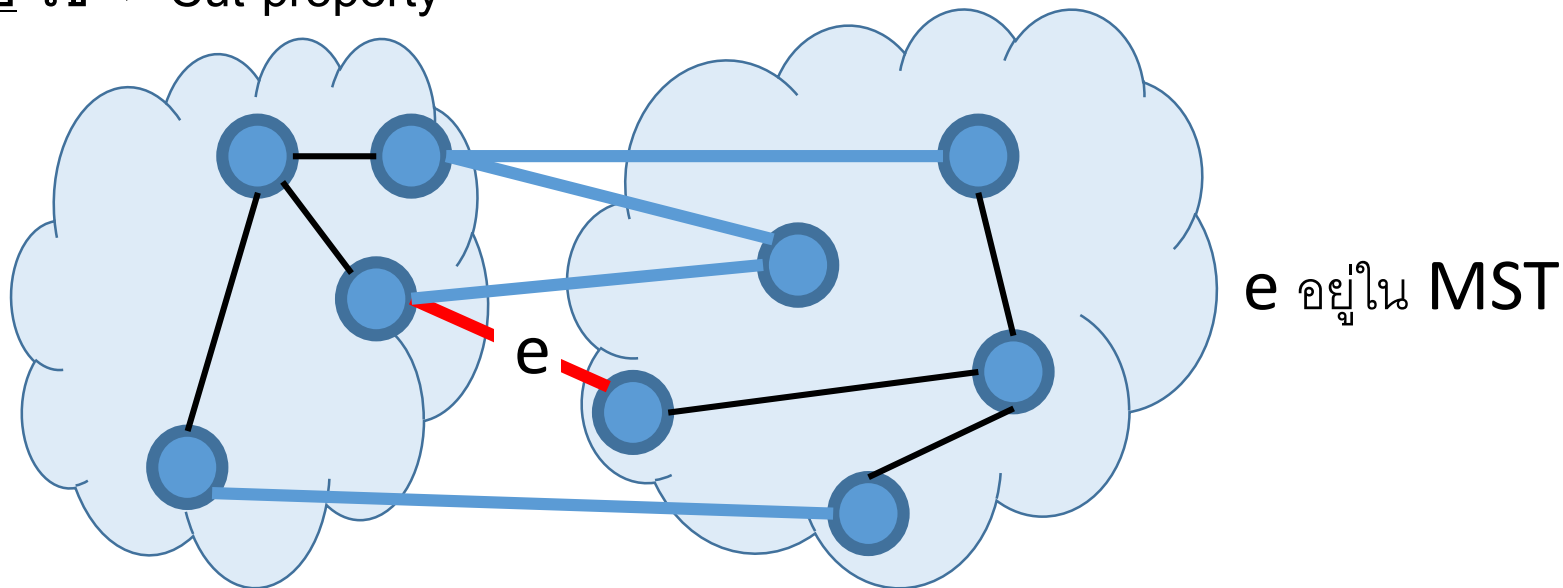
e อยู่ใน MST



f ไม่อยู่ใน MST

Cut property

- เพื่อให้ง่ายต่อการพิสูจน์ สมมติให้น้ำหนักของเส้นเชื่อมทุกเส้นไม่เท่ากัน
- คำถาม ให้ S แทน subset ใดๆ ของโหนดและให้ e แทนเส้นเชื่อมที่มีน้ำหนักน้อยที่สุดที่มีจุดปลายด้านหนึ่งอยู่ใน S เส้นเชื่อม e นั้นควรอยู่ในทุกๆ MST หรือไม่
- คำตอบ ใช่ -> Cut property



Cut property

- เพื่อให้ง่ายต่อการพิสูจน์ สมมติให้น้ำหนักของเส้นเชื่อมทุกเส้นไม่เท่ากัน
- cut property ให้ S แทน subset ใดๆ ของโหนดและให้ e แทนเส้นเชื่อมที่มีน้ำหนักน้อยที่สุดที่มีจุดปลายด้านหนึ่งอยู่ใน S แล้ว MST T^* จะมีเส้นเชื่อม e
- พิสูจน์

จะใช้เทคนิคอะไร

Cut property

- เพื่อให้ง่ายต่อการพิสูจน์ สมมติให้น้ำหนักของเส้นเชื่อมทุกเส้นไม่เท่ากัน
- cut property ให้ S แทน subset ใดๆ ของโหนดและให้ e แทนเส้นเชื่อมที่มีน้ำหนักน้อยที่สุดที่มีจุดปลายด้านหนึ่งอยู่ใน S แล้ว MST T^* จะมีเส้นเชื่อม e
- พิสูจน์ พิสูจน์โดยข้อขัดแย้ง

สมมติว่า e ไม่อยู่ใน T^* จะเกิดอะไรขึ้น

พบข้อขัดแย้งแสดงว่าสิ่งที่สมมติมาไม่จริง

Cut property

- เพื่อให้ง่ายต่อการพิสูจน์ สมมติให้นำหนักของเส้นเชื่อมทุกเส้นไม่เท่ากัน
- cut property ให้ S แทน subset ใดๆ ของโหนดและให้ e แทนเส้นเชื่อมที่มีน้ำหนักน้อยที่สุดที่มีจุดปลายด้านหนึ่งอยู่ใน S แล้ว MST T^* จะมีเส้นเชื่อม e

- พิสูจน์ พิสูจน์โดยข้อขัดแย้ง

สมมติว่า e ไม่อยู่ใน T^* จะเกิดอะไรขึ้น

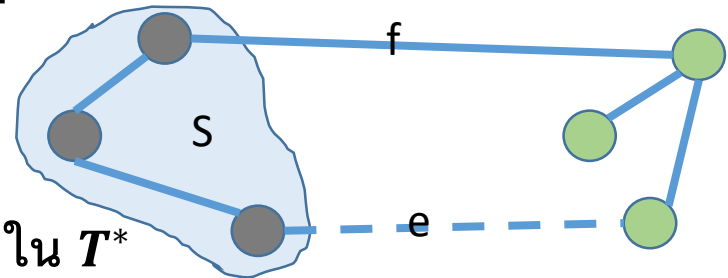
เมื่อเราเพิ่ม e เข้าไปใน T^* จะเกิด cycle C ใน T^*

แสดงว่ามีเส้นเชื่อมอื่นใน C สมมติว่าเป็นเส้น f ที่มีปลายข้างหนึ่งอยู่ใน S

$T' = T^* \cup \{e\} - \{f\}$ ยังเป็น spanning tree อยู่

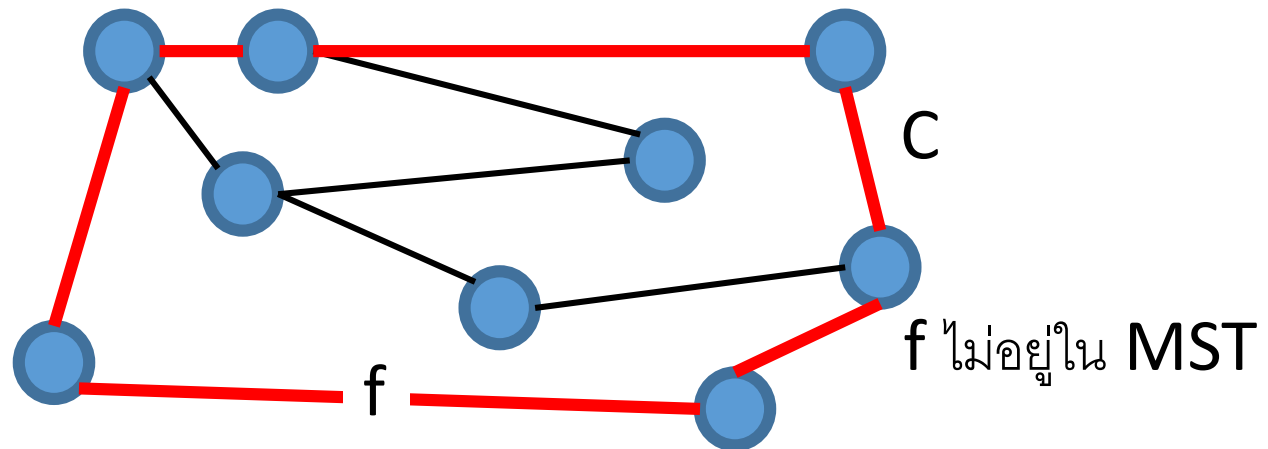
ดังนั้น $c_e < c_f$ ทำให้ได้ว่า $cost(T') < cost(T^*)$

พบข้อขัดแย้งว่าเริ่มต้นเรามี T^* ที่มีค่าน้อยที่สุด แต่เราพบ T' ที่มีค่าน้อยกว่า แสดงว่าสิ่งที่สมมติมาไม่จริง



Cycle property

- เพื่อให้ง่ายต่อการพิสูจน์ สมมติให้น้ำหนักของเส้นเชื่อมทุกเส้นไม่เท่ากัน
- คำถาม ให้ C แทน cycle ใดๆ MST มีทุกเส้นเชื่อมใน C หรือไม่
- คำตอบ ไม่ใช่
- คำถาม แล้วเส้นไหนไม่ควรอยู่ใน C
- คำตอบ เส้นที่น้ำหนักมากที่สุด \rightarrow Cycle property



Cycle property

- เพื่อให้ง่ายต่อการพิสูจน์ สมมติให้น้ำหนักของเส้นเชื่อมทุกเส้นไม่เท่ากัน
- cycle property ให้ C แทน cycle ใดๆ ใน G และให้ f แทนเส้นเชื่อมที่มีน้ำหนักมากที่สุดใน C แล้ว MST T^* จะต้องไม่มี f
- พิสูจน์

จะใช้เทคนิคอะไร

Cycle property

- เพื่อให้ง่ายต่อการพิสูจน์ สมมติให้น้ำหนักของเส้นเชื่อมทุกเส้นไม่เท่ากัน
 - cycle property ให้ C แทน cycle ใดๆ ใน G และให้ f แทนเส้นเชื่อมที่มีน้ำหนักมากที่สุดใน C แล้ว MST T^* จะต้องไม่มี f
 - พิสูจน์ พิสูจน์โดยข้อขัดแย้ง
- สมมติว่า f อยู่ใน T^* จะเกิดอะไรขึ้น

พบข้อขัดแย้งแสดงว่าสิ่งที่สมมติมาไม่จริง

Cycle property

- เพื่อให้ง่ายต่อการพิสูจน์ สมมติให้น้ำหนักของเส้นเชื่อมทุกเส้นไม่เท่ากัน
- cycle property ให้ C แทน cycle ใดๆ ใน G และให้ f แทนเส้นเชื่อมที่มีน้ำหนักมากที่สุดใน C แล้ว MST T^* จะต้องไม่มี f

- พิสูจน์ พิสูจน์โดยข้อขัดแย้ง

สมมติว่า f อยู่ใน T^* จะเกิดอะไรขึ้น

การลบเส้นเชื่อม f ออกจาก T^* จะทำให้ T^* ขาด

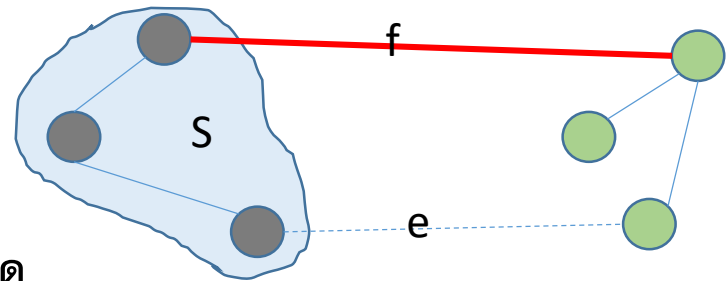
ให้ S เป็นด้านหนึ่งของ cut ใน T^*

จะมีเส้นเชื่อมอื่นใน C สมมติว่าเป็น e ที่มีจุดปลายด้านหนึ่งอยู่ใน S

$T' = T^* \cup \{e\} - \{f\}$ ยังเป็น spanning tree อยู่

ดังนั้น $c_e < c_f$ ทำให้ได้ว่า $cost(T') < cost(T^*)$

พบข้อขัดแย้งแสดงว่าสิ่งที่สมมติมาไม่จริง



Kruskal's algorithm

ใช้ priority queue ในการเก็บ
เส้นเชื่อมที่อยู่นอกกลุ่ม cloud
โดยที่มี

Key: น้ำหนัก

Element: เส้นเชื่อม

ตอนท้ายสุดเราจะเหลือกลุ่ม
cloud เพียงกลุ่มเดียวที่รวมเป็น
MST

Algorithm KruskalMST(G)

for each vertex V in G do

define a Cloud(v) of $= \{v\}$

let Q be a priority queue.

Insert all edges into Q using their
weights as the key

$T = \emptyset$

while T has fewer than $n-1$ edges do

edge $e = T.\text{removeMin}()$

Let u, v be the endpoints of e

if $\text{Cloud}(v) \neq \text{Cloud}(u)$ then

Add edge e to T

Merge $\text{Cloud}(v)$ and $\text{Cloud}(u)$

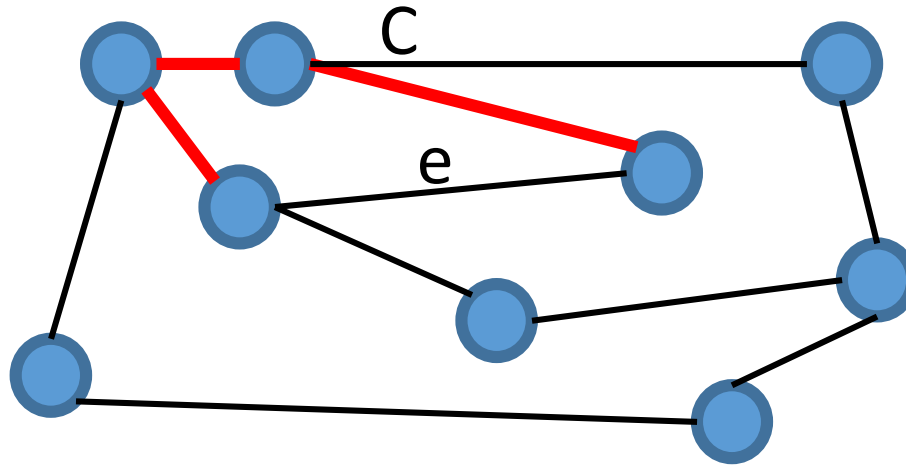
return T

Kruskal's algorithm

ทฤษฎีบท Kruskal's algorithm ให้ผลลัพธ์เป็น MST

พิสูจน์ เราจะเพิ่มเส้นเชื่อมเข้าไปใน T ทีละเส้นจากน้อยไปมาก ซึ่งแบ่งเป็น 2 กรณี

กรณี 1 ถ้าเพิ่มเส้นเชื่อม e ไปใน T แล้วสร้าง cycle C แล้ว e คือเส้นเชื่อมที่มีน้ำหนักมากที่สุดใน C จาก cycle property จะยืนยันว่า e จะไม่อยู่ใน MST

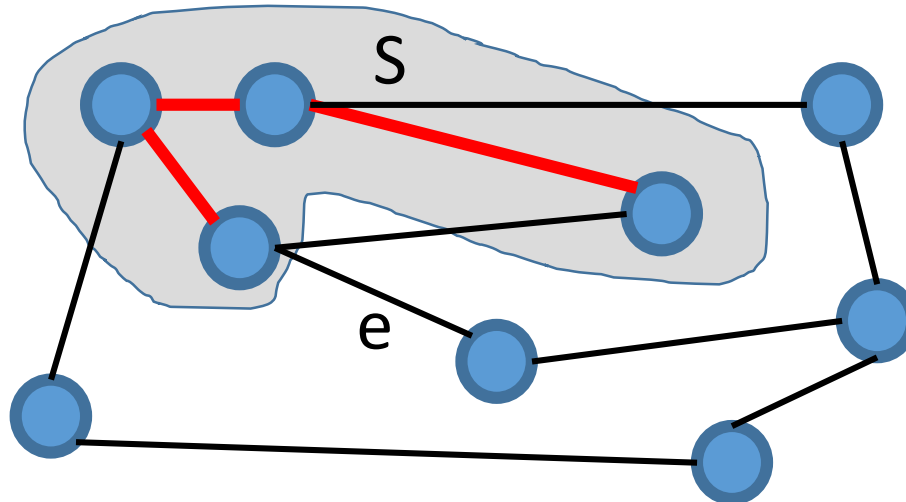


Kruskal's algorithm

ทฤษฎีบท Kruskal's algorithm ให้ผลลัพธ์เป็น MST

พิสูจน์ เราจะเพิ่มเส้นเชื่อมเข้าไปใน T ทีละเส้นจากน้อยไปมาก ซึ่งแบ่งเป็น 2 กรณี

กรณี 2 ถ้าเพิ่มเส้นเชื่อม $e=(v,w)$ ไปใน T แล้วไม่เกิด cycle แล้ว e คือเส้นเชื่อมที่มีน้ำหนักน้อยที่สุดที่มีจุดปลายจุดหนึ่งอยู่ใน S จาก cut property จะยืนยันว่า e จะอยู่ใน MST



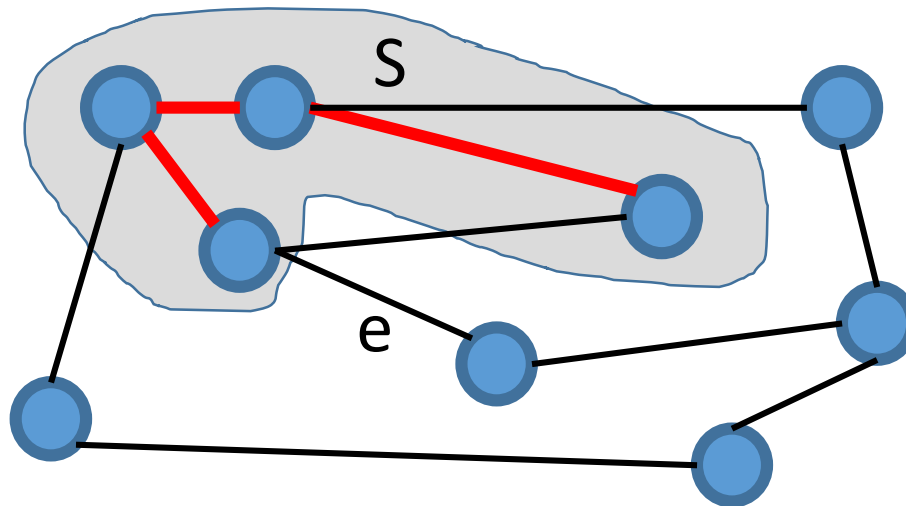
Prim's algorithm

ทฤษฎีบท Prim's algorithm ให้ผลลัพธ์เป็น MST

พิสูจน์ กำหนดให้ S เป็น subset ของโหนดในต้นไม้ปัจจุบัน T

Prim's algorithm จะเพิ่มเส้นเชื่อมที่น้ำหนักน้อยที่สุด e ที่มีสุดปลายจุดหนึ่งใน S

จาก cut property จะยืนยันว่าเส้นเชื่อม e ต้องอยู่ใน MST



Kruskal's algorithm

เรากล่าวถึง cloud ซึ่งแสดงกลุ่มของเส้นเชื่อม ว่าเส้นเชื่อมนี้อยู่กลุ่มไหน อีกทั้งมีการดำเนินการ 2 อย่างได้แก่

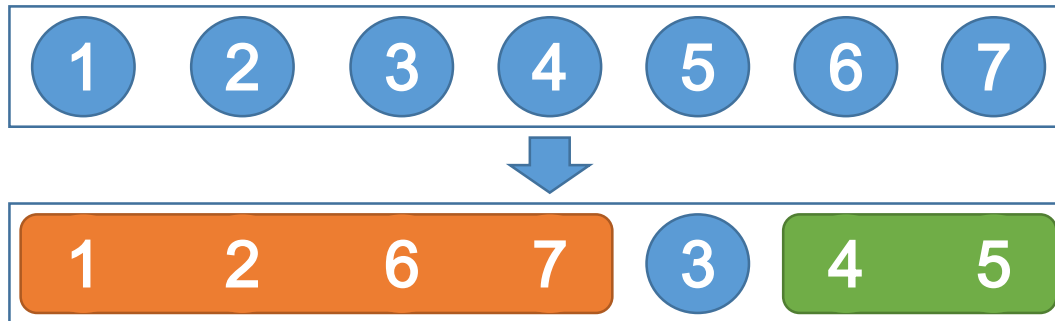
มีการตรวจสอบเส้นเชื่อม $e=(u, v)$ ว่า จุดปลาย u และ v อยู่ cloud เดียวกันหรือไม่

มีการรวม กลุ่ม u และ v เป็นกลุ่มเดียวกัน

ทำอย่างไรดี

Disjoint-set data structure

- Disjoint-set data structure เป็นโครงสร้างข้อมูลที่ติดตามเซตของสมาชิกที่ถูกแบ่งออกเป็นเซตที่ไม่มีสมาชิกร่วมกัน



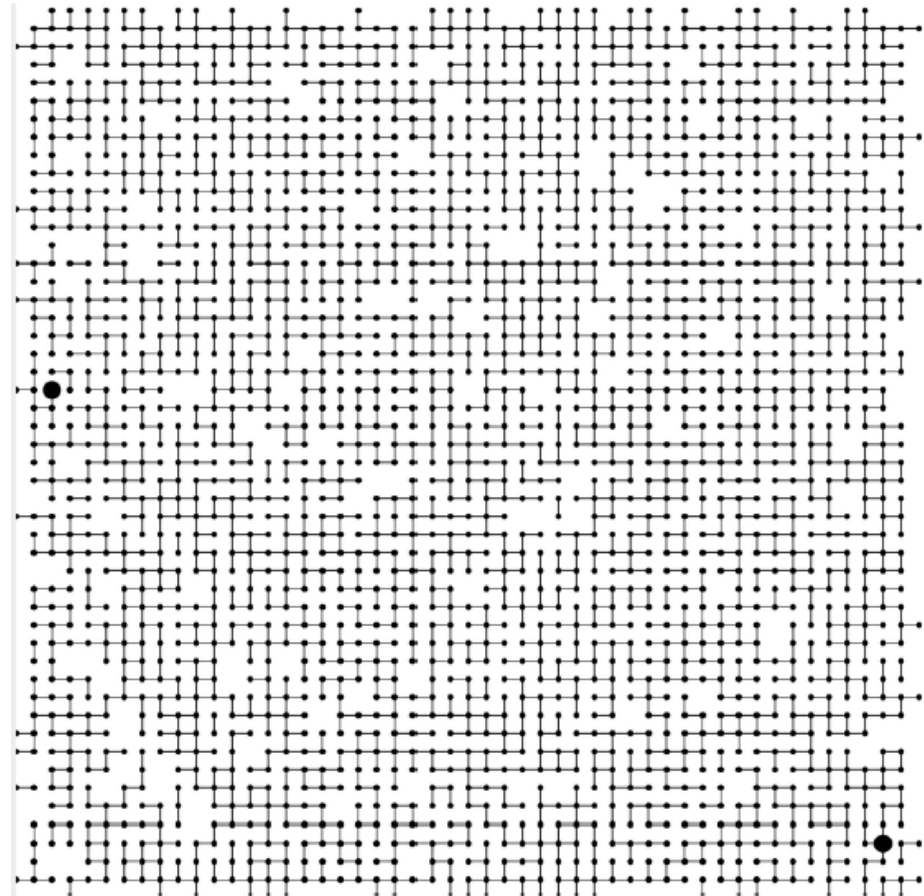
7 singletons

หลังจาก **union**,
บางเซตจะถูกรวมเข้าด้วยกัน

ตัวอย่าง: Network Connectivity

Basic abstractions

- set of objects
- union operation: connect two objects
- find operation: is there is a path connecting one object to another?



Objects

Union-find applications involve manipulating objects of all types.

- Computer in a network
- Web pages on the Internet
- Variable name aliases
- Connected components
- ...

When programming, convenient to name them to 0 to N-1

- Hide details not relevant to union-find
- Integers allow quick access to object-related info
- could use symbol table to translate from object names

Union-find abstractions

Simple model captures the essential nature of connectivity

- Objects

0 1 2 3 4 5 6 7 8 9

- Disjoint sets of objects

0 1 {2 3 7} 4 {5 6} {8 9}

- Find query: are objects 2 and 9 in the same set?

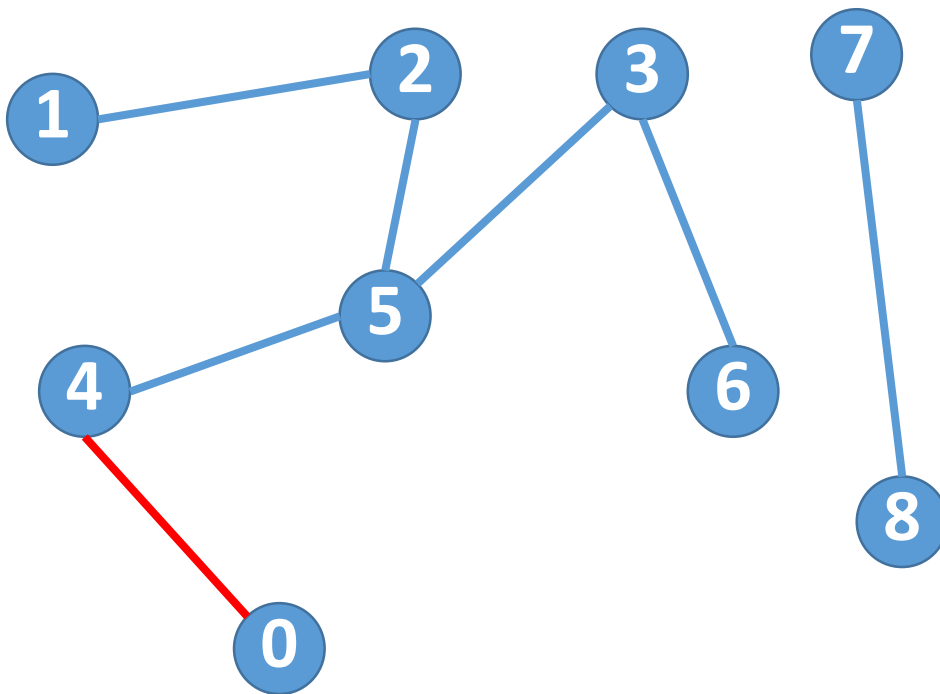
0 1 {2 3 7} 4 {5 6} {8 9}

- Union command: merge sets containing 3 and 8

0 1 {2 3 7 8 9} 4 {5 6}

Connected Components

- Connected component: set of mutually connected vertices



3 connected components

After union(0,4),
2 connected components

Quick-find

Data structure

- Integer array `id[]` of size `N`
- Interpretation: `p` and `q` are connected if they have the same `id`.

<code>i</code>	0	1	2	3	4	5	6	7	8	9
<code>id[i]</code>	0	1	9	9	9	6	6	7	8	9

5 and 6 are connected

2,3,4 and 9 are connected

Quick-find

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	9	9	6	6	7	8	9

5 and 6 are connected
2,3,4 and 9 are connected

- Find: check if p and q have the same id.
- Union. To merge components containing p and q , change all entries with $\text{id}[p]$ to $\text{id}[q]$.

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	6	6	6	6	6	7	8	6

union of 3 and 6
2,3,4,5,6 and 9 are connected

problem: many value can change

Quick-find: Implementation

```
int id[N];  
void QuickFind(int N){  
    for(int i=0; i<id.lenght(); i++)  
        id[i] = i;  
}
```

```
int find(int p, int q){  
    return id[p]==id[q];  
}
```

Quick-find algorithm may take $\sim MN$ steps to process M union commands on N objects

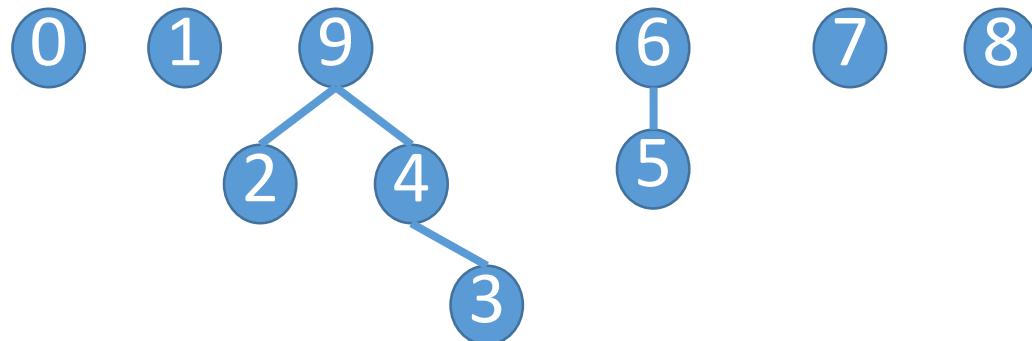
```
void unite(int p, int q){  
    int temp = id[p];  
    for(int i=0; i<id.lenght(); i++){  
        if(id[i]==temp)  
            id[i] = id[q];  
    }  
}
```

Quick-union

Data structure

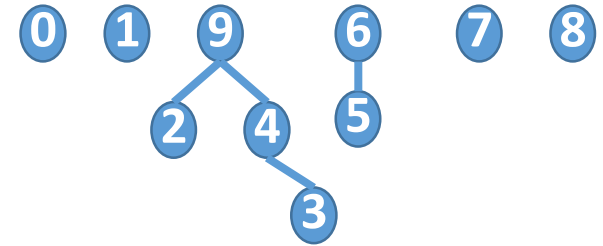
- Integer array `id[]` of size `N`
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` `id[id[...id[i]...]]`

<code>i</code>	0	1	2	3	4	5	6	7	8	9
<code>id[i]</code>	0	1	9	4	9	6	6	7	8	9



Quick-union

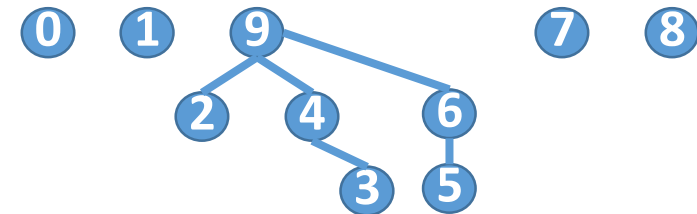
i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	4	9	6	6	7	8	9



- Find(p, q): check if p and q have the same root.
- Union(p, q). Set the id of q's root to the id of p's root

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	4	9	6	9	7	8	9

After union(6,4)



only one value changes
problem: trees can get tall

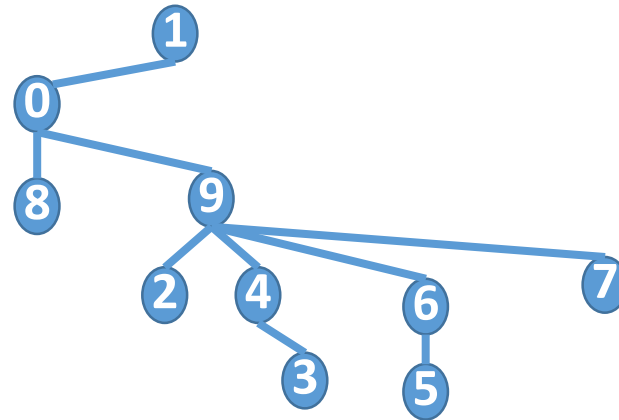
Quick-union: Implementation

```
int id[N];
void QuickUnion(){
    for(int i=0; i<id.lenght(); i++)
        id[i] = i;
}
int root(int i){
    while(i !=id[i])
        i=id[i];
    return i;
}
int find(int p, int q){
    return root(p)==root(q);
}
void union(int p, int q){
    int i = root(p);
    int j = root(q);
    id[i]=j;
}
```

Practice :

Draw the final tree when call union operation with this sequence:

3-4, 4-9, 8-0, 2-3, 5-6, 5-9, 7-3, 4-8, 6-1



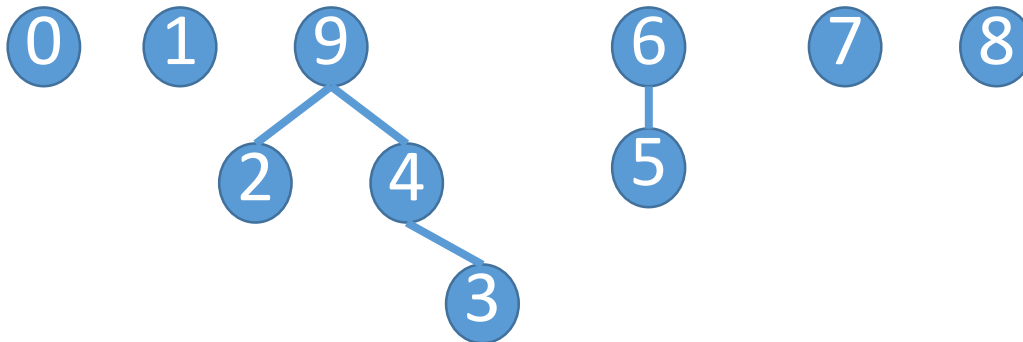
Improvement 1: weighting

Weighted quick-union

- Modify quick-union to avoid tall trees.
- Keep track of size of each component.
- Balance by linking small tree below large one.

Ex. Union of 5 and 3

- Quick union: link 9 to 6
- Weight quick union: link 6 to 9



Weighted quick-union

Almost identical to quick union.

Maintain extra array `sz[]` to count number of elements in the tree rooted at `i`

Find. Identical to quick-union

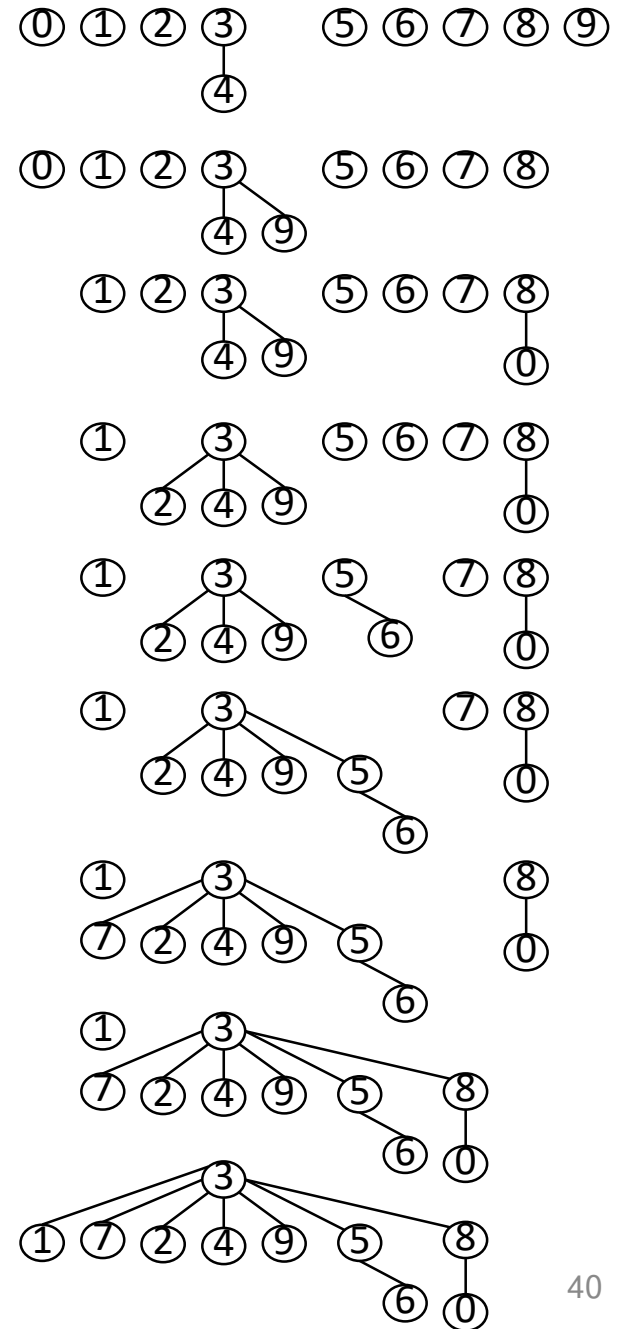
Union. Modify quick-union to

merge smaller tree into larger tree

update the `sz[]` array

```
if(sz[i] < sz[j]) {  
    id[i] = j; sz[j] += sz[i];  
} else {  
    id[j] = i; sz[i] += sz[j];  
}
```

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



trees stay flat

Performance

Data Structure	Union	Find
Quick-find	N	1
Quick-union	N	N
Weighted QU	$\lg N$	$\lg N$

- Homework: ให้ Implement Union-Find Data structure