Hanoi University of Science and Technology
School of Information and Communications Technology

Data structures and Algorithms

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication Technology
E-mail: phuongnk@soict.hust.edu.vn

Course outline

Hanoi University of Science and Technology
School of Information and Communications Technology

Chapter 7. Graph

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication Technology
E-mail: phuongnk@soict.hust.edu.vn

Contents

Contents

Shortest path representation

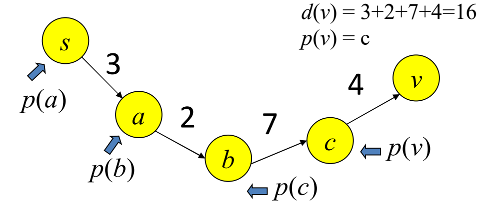
Shortest path algorithms works on 2 arrays:

- $d(v)$ = the length of shortest path from s to v that algorithm found so far (upper bound for the length of the shortest path from s to v).
- $p(v)$ = a predecessor of v in this shortest path (used to back trace the path from s to v) .

$\delta(s,v) \le d(v)$

Initialization

```
for v ∈ V(G)
    do d[v] ← ∞
       p[v] ← NULL
d[s] ← 0
```



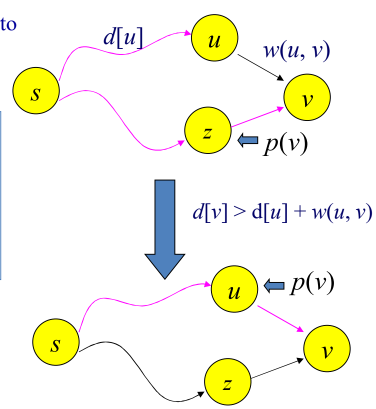
Relaxation

Building the shortest path from s to v:

Assume the current known shortest path from s to v: **s … $z \rightarrow v$**

Relaxing an edge (u,v) means testing whether we can improve the shortest path to v found so far by going through u

```
Relax(u, v)
   if (d[v] > d[u] + w(u, v))
   {      d[v] ← d[u] + w(u, v)
          p[v] ← u
   }
```



Properties of Relaxation

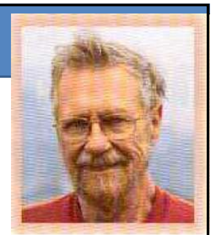
```
Relax(u, v)
   if (d[v] > d[u] + w(u, v))
   {
        d[v] ← d[u] + w(u, v)
        p[v] ← u
   }
```

Shortest path algorithms differ in

- *how many times* they relax each edge, and
- *the order* in which they relax edges

Dijkstra Algorithm

- In case the weights on the edges are non-negative, the algorithm proposed by Dijkstra is more efficient than the Ford-Bellman algorithm.
- Algorithms are built by labeling vertices. The label of the vertices is initially temporary. At each iteration there is a temporary label that becomes a permanent label. If the label of a vertex u becomes fixed, $d[u]$ gives us the length of the shortest path from the source s to u. Algorithm ends when the labels of all vertices become fixed.



Edsger W.Dijkstra

(1930-2002)

Dijkstra algorithm

- **Input:** A directed graph $G=(V,E)$ and weight matrix $w[u,v] \geq 0$ where $u,v \in V$, source vertex $s \in V$;
  - *G does not have* negative-weight cycle
- **Output:** Each $v \in V$

  $d[v] = \boxed{\delta(s, v);}$ Length of the shortest path from s to v

  $p[v]$ - the predecessor of v in this shortest path from s to v.

Use greedy algorithm:

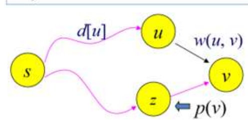
Maintain a set S of vertices for which we know the shortest path

At each iteration:

- grow S by one vertex , choosing shortest path through S to any other vertex not in S
- If the cost from S to any other vertex has decreased, update it

Dijkstra algorithm

```
Relax(u, v)
  if (d[v] > d[u] + w(u, v)) {
        d[v] ← d[u] + w(u, v)
        p[v] ← u
  }
```




```
Dijkstra ( )
{
    for  v ∈ V\s  { // Initialize
        d[v] = w[s,v] ;
        p[v] = s;
    }
    d[s] = 0;  p[s] = s;
    S = {s};   // S: the set of vertices with fixed label (shortest path from s to it has been found)
    T = V \ {s};          // T: the set of vertices with temporary label
    while  (T ≠ ∅)               //Loop
    {
        Find vertex  u ∈ T satisfying  d[u] = min{ d[z] :  z ∈ T};
        T =  T \ {u};  S= S ∪ {u};     //Fixed label of vertex u
        for v ∈ adj[u] and v ∈ T //Assign new label to each vertex v of T if necessary (if value d[v] is decreased)
            Relax(u, v)
    }
}
```

Use greedy algorithm:

Maintain a set S of vertices for which we know the shortest path

At each iteration:

- grow S by one vertex , choosing shortest path through S to any other vertex not in S
- If the cost from S to any other vertex has decreased, update it

Dijkstra algorithm

```
void Dijkstra ( )
{
    for  v ∈ V\s    // Initialize
    {
        d[v] = w[s,v] ;
        p[v]=s;
    }
    d[s] = 0;   p[s] = s; S = {s};
    T = V \ {s};
    while  (T ≠ ∅)               //Loop
    {
        Find vertex  u ∈ T satisfying  d[u] = min{ d[z] :  z ∈ T};
        T =  T \ {u};  S= S ∪ {u};
        for   v ∈ adj[u] and v ∈ T
            if   (d[v] > d[u] + w[u,v])
            {
                d[v] = d[u] + w[u,v] ;
                p[v] = u ;
            }
    }
}
```

$O(|V|^2)$

$O(|E|)$ for whole loop, as the adjacent list each vertex of the graph will be traversed exactly once

The computation time: $O(|V|^2 + |E|)$

Dijkstra algorithm

- **Comment:** If only need to find the shortest path from *s* to *t* then the algorithm could stop when *t* has fixed label ($t \in S$).

(1) Direct implement

Dijkstra_Table(G, s)

```
for v ∈ V\s do {
    d[v] ← infinity;
    found[v] ← FALSE;
}
d[s] ← 0; found[s] = TRUE;
  // s is source vertex
for i = 1 to |V|-1 do {
    u ← the vertex with min d among all
      vertices k having found[k] = false;
    found[u]=TRUE;
    for (v ∈ Adj(u) && !found[v]) do
      if (d[v] > d[u] + c[u, v]) {
            d[v] = d[u] + c[u, v];
            p[v] = u;
      }
}
```

Computation time **Dijkstra_Table(G, s):** $O(|V|^2+|E|)$.


```
INF = int(1e9)
def input_data():
   global V, E, s, t, Adj
   V, E = map(int, input().split())
   Adj =[[] for _ in range(V+1)]
   for _ in range(E):
     u, v, weight = map(int, input().split())
     Adj[u].append((v, weight))
   s, t = map(int, input().split())
def dijkstra_table():
```

Dijkstra_Table(G, s)

```
for u ∈ V do {
    d[u] ← infinity;
    found[u] ← FALSE;
}
d[s] ← 0;
  // s is source vertex
for i = 1 to |V|-1 do {
    u ← the vertex with min d among all
      vertices k having found[k] = false;
    found[u]=TRUE;
    for (v ∈ Adj(u) && !found[v]) do
      if (d[v] > d[u] + c[u, v]) {
            d[v] = d[u] + c[u, v];
            p[v] = u;
      }
}
```


(2) Implement Dijkstra by using priority queue

For each iteration of the algorithm we need to find a vertex with minimum d, to implement it more efficiently, we use min priority queue

Dijkstra_Table(G, s)

```
for u ∈ V do {
    d[u] ← infinity;
    found[u] ← FALSE;
}
d[s] ← 0;
  // s is source vertex
for i = 1 to |V|-1 do {
    u ← the vertex with min d among all
      vertices k having found[k] = false;
    found[u]=TRUE;
    for (v ∈ Adj(u) && !found[v]) do
      if (d[v] > d[u] + c[u, v]) {
            d[v] = d[u] + c[u, v];
            p[v] = u;
      }
}
```

(2) Implement Dijkstra by using priority queue

```
Dijkstra_Heap(G, s)
PQ = []
for u ∈ V do {
    d[u] = infinity;
    found[u] = FALSE;
}
d[s] ← 0; //s is source vertex
PQ.enqueue(s, d[s]);
for i = 1 to |V|-1 do {
    u ← PQ.dequeue(); // get vertex with min d
    found[u] = TRUE;
    for (v ∈ Adj(u)  && !found[v]) do
        if (d[v] > d[u] + c[u, v]) {
            d[v] = d[u] + c[u, v];
            p[v] = u;
            PQ.Decrease_Key(v, d[v]);
        }
}
```

```
Dijkstra_Table(G, s)
for u ∈ V do {
        d[u] ← infinity;
        found[u] ← FALSE;
}
d[s] ← 0;
  // s is source vertex
for i = 1 to |V|-1 do {
    u ← the vertex with min d among all
      vertices k having found[k] = false;
    found[u]=TRUE;
    for (v ∈ Adj(u) && !found[v]) do
      if  (d[v] > d[u] + c[u, v]) {
            d[v] = d[u] + c[u, v];
            p[v] = u;
      }
}
```


(2) Implement Dijkstra by using priority queue

The computation time:
O((|E|+|V|)log|V|)

```
Dijkstra_Heap(G, s)
PQ = []
for u ∈ V do {
    d[u] = infinity;
    found[u] = FALSE;
}
d[s] ← 0;
PQ.enqueue(s, d[s]); // s is source vertex   → O(log|V|)
for i = 1 to |V|-1 do {
    u ← PQ.dequeue(); // get vertex with min d   → O(log|V|)
    found[u] = TRUE;
    for (v ∈ Adj(u)  && !found[v]) do
        if (d[v] > d[u] + c[u, v]) {
            d[v] = d[u] + c[u, v];
            p[v] = u;
            PQ.Decrease_Key(v, d[v]);   → O(log|V|)
        }
}
```

O(|V|+log|V|)

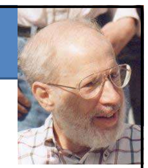
O(|V|log|V|)

O(|E|log|V|)

Contents

1. Dijkstra algorithm
2. **Kruskal algorithm**

Kruskal algorithm



Joseph Kruskal
(1928 - ~)

```
Generic-MST(G, c)
  T = ∅
  //T  is the subset edges of a minimum spanning tree
  while T is not the spanning tree do
       Finding edge (u, v) is "safe" edge for T
       T = T ∪{(u, v)}
  return T
```

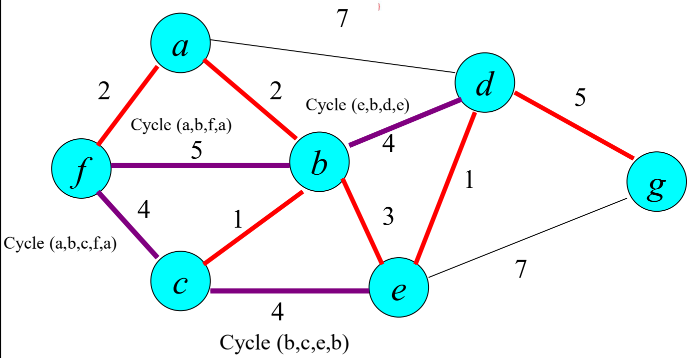
Kruskal algorithm:

- *T* is **forest (empty).**
- The "safe" edge included in *T* at each iteration is the edge with smallest weight **among edges connecting its connected components**.

```
void Kruskal ( )
{
    Sort m edges of the graph e_1,e_2, . . . , e_m in ascending order of weight;
    T = ∅;      // T: set of edges of the minimum spanning tree
    for (i = 1; i <=m; i++)
        if (T ∪{e_i} does not contain cycle)
                      T =  T ∪ {e_i};
}
```

Kruskal algorithm: an example

```
void Kruskal ( )
{
    Sort m edges of the graph e1,e2, . . . , em in ascending order of weight;
    T = ∅;      // T: set of edges of the minimum spanning tree
    for (i = 1; i <=m; i++)
        if (T ∪{ei} does not contain cycle)
                    T =  T ∪ {ei};
}
```



Weight of the minimum spanning tree:
2+2+1+3+1+5=14

Computation time


```
void Kruskal ( )
{
    Sort m edges of the graph e1,e2, . . . , em in ascending order of weight;
    T = ∅;      // T: set of edges of the minimum spanning tree
    for (i = 1; i <=m; i++)
        if (T ∪{ei} does not contain cycle)
                    T =  T ∪ {ei};
}
```


- Step 1. Sort the edges in ascending order of weight
  - Could use heap sort/merge sort : $O(m \log m)$
- Iterations: Each iteration we need to check whether $T \cup \{ e_i \}$ contains the cycle?
  - Could use DFS to check with time $O(m+n)$.
  - Total time: $O(m(m+n))$

Computation time: $O(m \log m + m(m+n))$ where n, m is the number of vertices and edges of the graph, respectively.

Improved implementation:

- Each connected component C of forest F is setup as a set.
- Denote First(C) be the first vertex in connected component C.
- Each vertex j in C, set First(j) = First(C) = first vertex in C.
- Note: Adding edge (i,j) to forest F creates cycle iff i and j belongs to the same connected component, it means First(i) = First(j).
- When connecting connected components C and D together, we connect the smaller one (less number of vertices) to the larger one (more number of vertices):

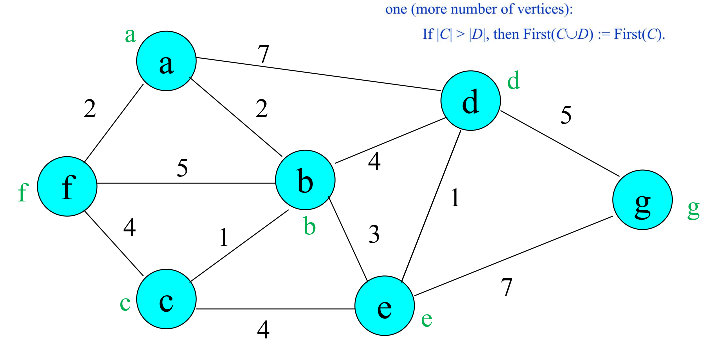
  If $|C| > |D|$, then First($C \cup D$) := First(C).

❖ *T* is **forest (empty).**
❖ The "safe" edge included in *T* at each iteration is the edge with smallest weight **among edges connecting its connected components**.

Kruskal – Example

- Denote First(C) be the first vertex in connected component C.
- Each vertex j in C, set First(j) = First(C) = first vertex in C.
- Note: Adding edge (i,j) to forest F creates cycle iff i and j belongs to the same connected component, it means First(i) = First(j).
- When connecting connected components C and D together, we connect the smaller one (less number of vertices) to the larger one (more number of vertices):

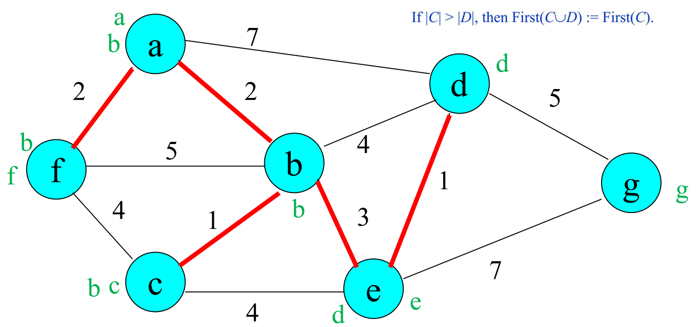
  If $|C| > |D|$, then First($C \cup D$) := First(C).



Kruskal – Example

- Denote First(C) be the first vertex in connected component C.
- Each vertex j in C, set First(j) = First(C) = first vertex in C.
- Note: Adding edge (i,j) to forest F creates cycle iff i and j belongs to the same connected component, it means First(i) = First(j).
- When connecting connected components C and D together, we connect the smaller one (less number of vertices) to the larger one (more number of vertices):

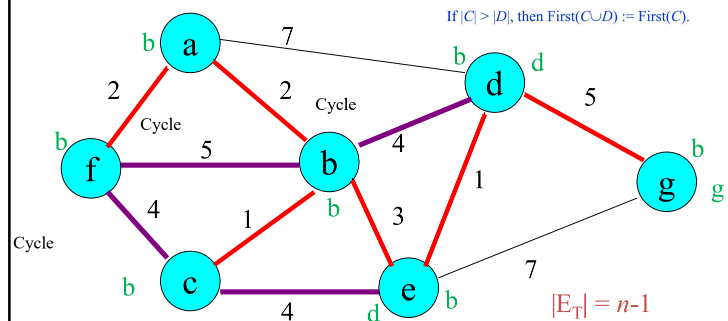
  If $|C| > |D|$, then First($C \cup D$) := First(C).



Kruskal – Example

- Denote First(C) be the first vertex in connected component C.
- Each vertex j in C, set First(j) = First(C) = first vertex in C.
- Note: Adding edge (i,j) to forest F creates cycle iff i and j belongs to the same connected component, it means First(i) = First(j).
- When connecting connected components C and D together, we connect the smaller one (less number of vertices) to the larger one (more number of vertices):

  If $|C| > |D|$, then First($C \cup D$) := First(C).



Cycle as vertex c and e belongs to the same connected component

$|E_T| = n$-1

Length of MST: 14

Improved implementation: Computation time

- Time to determine whether 2 vertices i, j belong to the same connected component: First(i) = First(j) : $O(1)$ for each i, j. There are m edges → Total: $O(m)$.
- Time to connect 2 connected components S and Q, where $|S| \geq |Q|$:
  - $O(1)$ for each vertex of Q (the one with smaller number of vertices)
  - Each vertex i in smaller connected component: connect $\log n$ times as maximum. (As the number vertices of connected component containing i is doubled after each connection.)

Total time to connect over all algorithm: $O(n \log n)$.

- Computation time:

  $O(m\log m + m + n \log n)$.

Improved implementation: Computation time

```
void Kruskal ( )
{
    Sort m edges of the graph e1,e2, . . . , em in ascending order of weight;
    T = ∅;      // T: set of edges of the minimum spanning tree
    for (i = 1; i <=m; i++)
        if (T ∪{ei} does not contain cycle)
            T =  T ∪ {ei};
}
```

Time to connect 2 connected components S and Q, where $|S| \geq |Q|$:
- $O(1)$ for each vertex of Q (the one with smaller number of vertices)
- Each vertex i in smaller connected component: connect $\log n$ times as maximum. (As the number vertices of connected component containing i is doubled after each connection.)

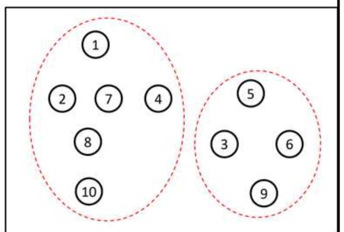
➔ Total time to connect over all algorithm: $O(n \log n)$.

- Step 1. Sort the edges in ascending order of weight
  - Could use heap sort/merge sort : $O(m \log m)$
- Iterations: Each iteration we need to check whether $T \cup \{ e_i = (x, y)\}$ contains the cycle?
  - Could use DFS to check with time $O(m+n)$. → O(1): check First(x) = First(y)
  - Total time: $O(m(m+n))$ → O(m)

Computation time: $O(m \log m + m(m+n))$ where n, m is the number of vertices and edges of the graph, respectively. → $O(m\log m + m + n \log n)$.

Disjoint set

- Disjoint set data structure is used to represent sets that don't have any elements in common. It supports 3 main operations:
  - **makeSet(*x*):** create a set containing only an element *x*
  - **find (*x*):** find the id of a set containing element *x*
  - **union(*a*, *b*):** merge the set having id=*a* and the set having id=*b* to a single set

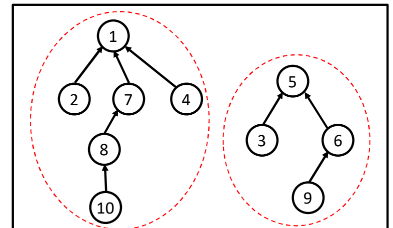


Two disjoint sets

Disjoint set

- Each set is represented by a tree:
  - Each node in the tree corresponds to an element of the set.
  - The root of the tree is the id of the set.
  - Each node has exactly one parent (parent of the root is itself).

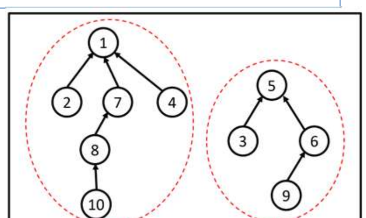
x	1	2	3	4	5	6	7	8	9	10
parent[x]	1	1	5	1	5	5	1	7	6	8



Each tree corresponds to a tree

Disjoint set

```
makeSet(x){ //Create a set containing only element x
     parent[x] = x;
}
find(x){ //return id of the set containing x
    while (x != parent[x]) do x = parent[x];
    return x;
}
```



Each tree corresponds to a tree

Disjoint set

```
makeSet(x){ //Create a set containing only element x
     parent[x] = x;
}
find(x){ //return id of the set containing x
    while (x != parent[x]) do x = parent[x];
    return x;
}
union(a, b) {//merge 2 sets: the set with id = a and the set with id = b
    if (a != b) then
        parent[a] = b; //make id of a'set be the id of b's set
```

Kruskal: NOT use disjoint set

```
1.  Kruskal(G=(V,E),w){
2.      E' = sort edges in ascending order of weight;
3.      E_T = ∅
4.      for (u, v) in E' do {
5.          if E_T ∪ (u, v) does not contain cycle) then
6.              E_T = E_T ∪ (u,v);
7.          if (|E_T| == |V| - 1) then break;
8.      }
9.  }
10. if (|E_T| < |V| - 1) then {
11.     print("ERROR: Graph G is not connected");
12.     return NULL;
13. }
14. else return (V, E_T);
15. }
```

Not use Disjoint set

$O(|E|\ \log|E| + |E|(|E|+|V|))$

```
1.  Kruskal(G=(V,E),w){
2.      for (x = 1; x <= |V|; x++) do makeSet(x);
3.      E' = sort edges in ascending order of weight;
4.      E_T = ∅
5.      for (u, v) in E' do {
6.          idu = find(u);
7.          idv = find(v);
8.          if (idu != idv) then
9.              union (idu, idv);
10.             E_T = E_T ∪ (u,v);
11.             if (|E_T| == |V| - 1) then break;
12.         }
13.     }
14.     if (|E_T| < |V| - 1) then {
15.         print("ERROR: Graph G is not connected");
16.         return NULL;
17.     }
18.     else return (V, E_T);
19. }
```

Use Disjoint set

What is the computation time if disjoint set is used ?

$O(|V| + |E|\log|E| + |E||V|)$

Kruskal: use disjoint set

```
1. makeSet(x){ //Create a set containing only element x
2.      parent[x] = x;
3. }
4. find(x){ //not compression
5       while (x != parent[x]) do x = parent[x];
6.      return x;
7. }
8. union(a, b) {//not union by rank
11.     if (a != b) then
12.         parent[a] = b;
```

not optimize

$O(|V| + |E|\log|E| + |E||V|)$

```
1.  Kruskal(G=(V,E),w){
2.      for (x = 1; x <= |V|; x++) do makeSet(x);
3.      E' = sort edges in ascending order of weight;
4.      E_T = ∅
5.      for (u, v) in E' do {
6.          idu = find(u);
7.          idv = find(v);
8.          if (idu != idv) then
9.              union (idu, idv);
10.             E_T = E_T ∪ (u,v);
11.             if (|E_T| == |V| - 1) then break;
12.         }
13.     }
14.     if (|E_T| < |V| - 1) then {
15.         print("ERROR: Graph G is not connected");
16.         return NULL;
17.     }
18.     else return (V, E_T);
19. }
```

If use union(a, b) not by rank and find(x) not compression:

- **find(x)**: could take **O(|V|)** in a worst-case scenario (a long, skinny tree).
- **union(a, b):** Each union(a, b) operation takes **O(1)** because we just update the parent.

➔ Kruskal: We call find(x) multiple times → the loop lines 5-13: the overall cost can be **O(|E||V|)**.

Disjoint set

- Path compression: to improve computation time of function `find(x)`
  - When `find(x)` is called, root of the tree containing x is returned.
  - Current implementation: The `find(x)` operation traverses up from x to find root.

```
4. find(x){ //return id of the set containing x
5       while (x != parent[x]) do x = parent[x];
6.      return x;
7. }
```

  - The idea of path compression is to make the found root as parent of x so that we don't have to traverse all intermediate nodes again. If x is root of a subtree, then path (to root) from all nodes under x also compresses. ➔ It speeds up the data structure by **compressing the height** of the trees.

Disjoint set

- Path compression: to improve computation time of function `find(x)`

```
1. find(x){ //return id of the set containing x
2       while (x != parent[x]) do x = parent[x];
3.      return x;
4. }
```

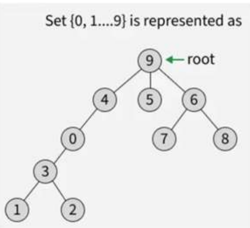
Path compression

```
find(x) {
   if (x != parent[x])
      parent[x] = find(parent[x])   #path compression

   return parent[x];
}
```

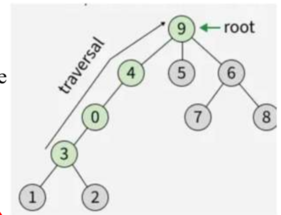
Disjoint set: Path compression

```
find(x) {
   if (x != parent[x])
      parent[x] = find(parent[x])    #path compression

   return parent[x];
}
```

Set {0, 1....9} is represented as

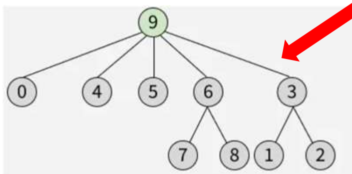


Call find(3): traverse up and find 9 is id of the tree



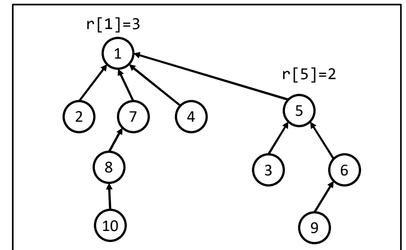
- The idea of path compression is to make the found root as parent of x so that we don't have to traverse all intermediate nodes again. If x is root of a subtree, then path (to root) from all nodes under x also compresses. → It speeds up the data structure by **compressing the height** of the trees.

When find(3) is terminated, path compression is already applied → we get directly link element 0 and 3 to 9



Disjoint set

- Reduce the size of each tree: to improve computation time of function `union(x,y)`
  - When merging two trees, we will join the tree with the lower height to the tree with the higher height.
  - We use the additional variable r[x] to record the height of a node x in the tree



Union 2 disjoint sets

Disjoint set

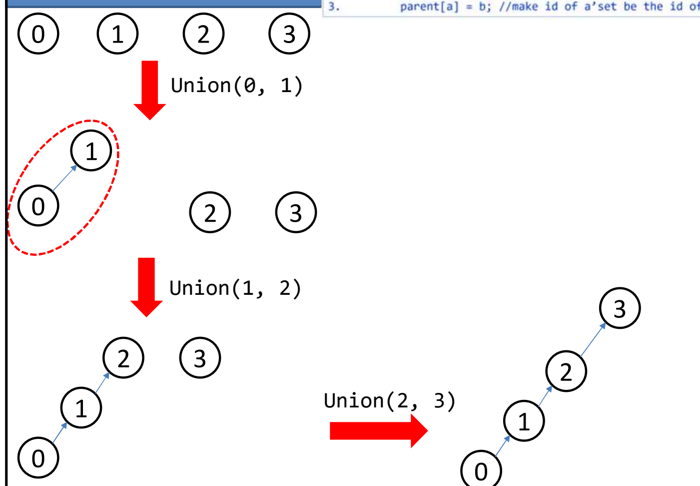
```
union(a, b) {//merging 2 sets: the set with id=a and the set with id=b
  if (a != b) then
        parent[a] = b; //make id of a'set be the id of b's set
```

```
union(a, b) {//merging 2 sets: the set with id=a and the set with id=b
    if (a != b) then
        if (r[a] > r[b]) then parent[b] = a;
        else{
            parent[a] = b;
            if (r[a] == r[b]) then r[b] = r[b] + 1;
        }
}
```

Union by rank to reduce the height of tree

Disjoint set

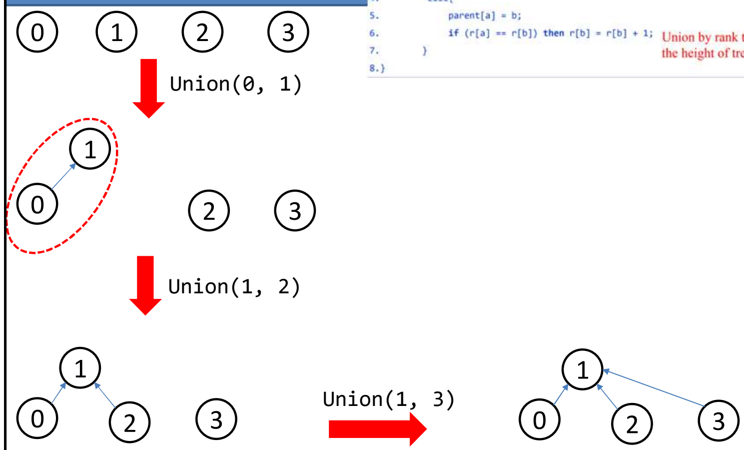
```
union(a, b) {//merging 2 sets: the set with id=a and the set with id=b
  if (a != b) then
        parent[a] = b; //make id of a'set be the id of b's set
```



Disjoint set

```
1. union(a, b) {//merging 2 sets: the set with id=a and the set with id=b
2.     if (a != b) then
3.         if (r[a] > r[b]) then parent[b] = a;
4.         else{
5.             parent[a] = b;
6.             if (r[a] == r[b]) then r[b] = r[b] + 1;
7.         }
8.}
```

Union by rank to reduce the height of tree



Kruskal: disjoint set

```
1.  Kruskal(G=(V,E),w){
2.      E' = sort edges in ascending order of weight;
3.      E_T = ∅
4.      for (u, v) in E' do {
5.          if E_T ∪ (u, v) does not contain cycle) then
6.              E_T = E_T ∪ (u,v);
7.          if (|E_T| == |V| - 1) then break;
8.      }
9.  }
10. if (|E_T| < |V| - 1) then {
11.     print("ERROR: Graph G is not connected");
12.     return NULL;
13. }
14. else return (V, E_T);
15. }
```

Not use Disjoint set

$O(|E|\ \log|E| + |E|(|E|+|V|))$

```
1.  Kruskal(G=(V,E),w){
2.      for (x = 1; x <= |V|; x++) do makeSet(x);
3.      E' = sort edges in ascending order of weight;
4.      E_T = ∅
5.      for (u, v) in E' do {
6.          idu = find(u);
7.          idv = find(v);
8.          if (idu != idv) then {
9.              union (idu, idv);
10.             E_T = E_T ∪ (u,v);
11.             if (|E_T| == |V| - 1) then break;
12.         }
13.     }
14.     if (|E_T| < |V| - 1) then {
15.         print("ERROR: Graph G is not connected");
16.         return NULL;
17.     }
18.     else return (V, E_T);
19. }
```

Use Disjoint set

What is the computation time if disjoint set is used ?

$O(|V| + |E|\log|E| + |\boldsymbol{E}||\boldsymbol{V}|)$

$O(|V| + |E|\log|E| + |\boldsymbol{E}|\boldsymbol{log}|\boldsymbol{V}|)$

$O(|V| + |E|\log|E| + |\boldsymbol{E}|\ \alpha(|\boldsymbol{V}|))$

Kruskal: disjoint set

```
1. makeSet(x){ //Create a set containing only element x
2.     parent[x] = x;
3. }
4. find(x){ //not compression
5      while (x != parent[x]) do x = parent[x];
6.     return x;
7. }
8. union(a, b) {//not union by rank
11.    if (a != b) then
12.        parent[a] = b;
```

not optimize

$O(|V| + |E|\log|E| + |E||V|)$


```
1.  Kruskal(G=(V,E),w){
2.      for (x = 1; x <= |V|; x++) do makeSet(x);
3.      E' = sort edges in ascending order of weight;
4.      E_T = ∅
5.      for (u, v) in E' do {
6.          idu = find(u);
7.          idv = find(v);
8.          if (idu != idv) then
9.              union (idu, idv);
10.             E_T = E_T ∪ (u,v);
11.             if (|E_T| == |V| - 1) then break;
12.         }
13.     }
14.     if (|E_T| < |V| - 1) then {
15.         print("ERROR: Graph G is not connected");
16.         return NULL;
17.     }
18.     else return (V, E_T);
19. }
```


If use union(a, b) not by rank and find(x) not compression:

- **find(x)**: could take **O(|V|)** in a worst-case scenario (a long, skinny tree).
- **union(a, b):** Each union(a, b) operation takes **O(1)** because we just update the parent.

➔ Kruskal: We call find(x) multiple times → the loop lines 5-13: the overall cost can be **O(|E||V|)**.

Kruskal: disjoint set

```
makeSet(x){ //Create a set containing only element x
    parent[x] = x; r[x] = 0;
}
find(x){ //not compression
    while (x != parent[x]) do x = parent[x];
    return x;
}
union(a, b) {//union by rank
    if (a != b) then
        if (r[a] > r[b]) then parent[b] = a;
        else{
            parent[a] = b;
            if (r[a] == r[b]) then r[b] = r[b] + 1;
        }
}
```


```
1.  Kruskal(G=(V,E),w){
2.      for (x = 1; x <= |V|; x++) do makeSet(x);
3.      E' = sort edges in ascending order of weight;
4.      E_T = ∅
5.      for (u, v) in E' do {
6.          idu = find(u);
7.          idv = find(v);
8.          if (idu != idv) then {
9.              union (idu, idv);
10.             E_T = E_T ∪ (u,v);
11.             if (|E_T| == |V| - 1) then break;
12.         }
13.     }
14.     if (|E_T| < |V| - 1) then {
15.         print("ERROR: Graph G is not connected");
16.         return NULL;
17.     }
18.     else return (V, E_T);
19. }
```


$O(|V| + |E|\log|E| + |\boldsymbol{E}|\boldsymbol{log}|\boldsymbol{V}|)$

If use union(a, b) by rank and find(x) not compression:

- find(x): O(log|V|)
- union(a, b): O(1) to update parent and rank.

Kruskal: disjoint set

```
makeSet(x){ //Create a set containing only element x
    parent[x] = x; r[x] = 0;
 }
find(x) { //path compression
   if (x != parent[x])
     parent[x] = find(parent[x])

   return parent[x];
}
union(a, b) {//union by rank
     if (a != b) then
         if (r[a] > r[b]) then parent[b] = a;
         else{
             parent[a] = b;
             if (r[a] == r[b]) then r[b] = r[b] + 1;
         }
}
```

```
Kruskal(G=(V,E),w){
    for (x = 1; x <= |V|; x++) do makeSet(x);
    E' = sort edges in ascending order of weight;
    E_T = Ø
    for (u, v) in E' do {
        idu = find(u);
        idv = find(v);
        if (idu != idv) then {
            union (idu, idv);
            E_T = E_T ∪ (u,v);
            if (|E_T| == |V| - 1) then break;
        }
    }
    if (|E_T| < |V| - 1) then {
        print("ERROR: Graph G is not connected");
        return NULL;
    }
    else return (V, E_T);
}
```

$O(|V| + |E|\log|E| + |\boldsymbol{E}|\ \alpha(|\boldsymbol{V}|))$

If use union(x, y) by rank and find(x) path compression:

- find(x): **α(|V|) where** α(|V|) is the **inverse Ackermann function (**If $|V|\approx 10^{80}$ then $\alpha(|V|) \leq 5$), so find(x) takes almost constant O(1).
- union(a, b): O(1)