利用連結所提供 disjoint set 的函示，設計出最小展開樹程式
http://www.geeksforgeeks.org/union-find-algorithm-set-2-union-by-rank/
程式中需要利用 path compression，find，union，union by rank 等函式解決判斷加入一邊是否形成 cycle 的 case，以 prime algorithm 或以陣列實作 set 者不計分。
圖形的資料結構也需與連結函式相同。
由檔案 test2.txt 輸入圖形，第一列代表節點數，其後每一行代表邊(格視為 頂點 頂點 權重)如 1 3 23，直到 end of file。過程中請輸出集合內容，最後輸出 MST 的邊與權重。

Test2.txt 範例
6
1 2 1.5
2 3 4
4 6 34
3 1 32
5 6 5
2 4 5
3 4 2

Union-Find Algorithm | Set 2 (Union By Rank and Path Compression)

In the previous post, we introduced *union find algorithm* and used it to detect cycle in a graph. We used following *union()* and *find()* operations for subsets.

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

// Naive implementation of union()
void Union(int parent[], int x, int y)
{
    int xset = find(parent, x);
    int yset = find(parent, y);
    parent[xset] = yset;
}
```

Run on IDE
The above *union()* and *find()* are naive and the worst case time complexity is linear. The trees created to represent subsets can be skewed and can become like a linked list. Following is an example worst case scenario.

```
Let there be 4 elements 0, 1, 2, 3


Initially all elements are single element subsets.
```

```
0 1 2 3

Do Union(0, 1)
   1   2   3
  /
 0

Do Union(1, 2)
     2   3
    /
   1
 /
0

Do Union(2, 3)
         3
        /
      2
     /
   1
 /
0
```

The above operations can be optimized to *O(Log n)* in worst case. The idea is to always attach smaller depth tree under the root of the deeper tree. This technique is called ***union by rank***. The term *rank* is preferred instead of height because if path compression technique (we have discussed it below) is used, then *rank* is not always equal to height. Also, size (in place of height) of trees can also be used as *rank*. Using size as *rank* also yields worst case time complexity as O(Logn) (See this for prrof)

```
Let us see the above example with union by rank
Initially all elements are single element subsets.
0 1 2 3

Do Union(0, 1)
   1   2   3
  /
```

```
 0

Do Union(1, 2)
   1    3
 /  \
0    2

Do Union(2, 3)
    1
 /  |  \
0   2   3
```

The second optimization to naive method is ***Path Compression***. The idea is to flatten the tree when *find()* is called. When *find()* is called for an element x, root of the tree is returned. The *find()* operation traverses up from x to find root. 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.

```
Let the subset {0, 1, .. 9} be represented as below and find() is called
for element 3.
             9
         /    |    \
        4     5      6
     /     \        /  \
    0       3     7    8
           /  \
          1    2

When find() is called for 3, we traverse up and find 9 as representative
of this subset. With path compression, we also make 3 as child of 9 so
that when find() is called next time for 1, 2 or 3, the path to root is
reduced.
```

```
              9
         /    /  \    \
        4    5    6     3
      /          /  \   /  \
     0          7    8  1   2
```

The two techniques complement each other. The time complexity of each operations becomes even smaller than O(Logn). In fact, amortized time complexity effectively becomes small constant.

Following is union by rank and path compression based implementation to find cycle in a graph.

```
// A union by rank and path compression based program to detect
cycle in a graph
#include <stdio.h>
#include <stdlib.h>

// a structure to represent an edge in graph
struct Edge
{
    int src, dest;
};

// a structure to represent a graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges
    struct Edge* edge;
};

struct subset
{
    int parent;
    int rank;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = (struct Graph*) malloc( sizeof(struct
Graph) );
    graph->V = V;
    graph->E = E;

    graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct
Edge ) );

    return graph;
```

```
}

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high rank tree
    // (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and increment
    // its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// The main function to check whether a given graph contains cycle
or not
int isCycle( struct Graph* graph )
{
    int V = graph->V;
    int E = graph->E;

    // Allocate memory for creating V sets
    struct subset *subsets =
        (struct subset*) malloc( V * sizeof(struct subset) );

    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Iterate through all edges of graph, find sets of both
    // vertices of every edge, if sets are same, then there is
    // cycle in graph.
    for(int e = 0; e < E; ++e)
```

```
    {
        int x = find(subsets, graph->edge[e].src);
        int y = find(subsets, graph->edge[e].dest);

        if (x == y)
            return 1;

        Union(subsets, x, y);
    }
    return 0;
}

// Driver program to test above functions
int main()
{
    /* Let us create following graph
         0
        |  \
        |    \
        1-----2 */

    int V = 3, E = 3;
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;

    // add edge 1-2
    graph->edge[1].src = 1;
    graph->edge[1].dest = 2;

    // add edge 0-2
    graph->edge[2].src = 0;
    graph->edge[2].dest = 2;

    if (isCycle(graph))
        printf( "Graph contains cycle" );
    else
        printf( "Graph doesn't contain cycle" );

    return 0;
}
```