EE3980 Algorithm

HW3. Network Connectivity Problem

106061225 楊宇羲

1. Problem Description:

???

Given a network with its amount of vertexes <u>and each connected edge,</u> we are required to determine the number of disjoint sets. <u>the total amount of disjoint sets</u>. The general steps will be given at below as pseudo code. We will first find one edge's two end points' root element, if their root element are the same, then the two vertexes are already in the same subset, meaning that there is no need to set union between the two end points. However, if the two connected vertexes belong to different sets set ,then it means that they are now connected and we ??? <u>should set union</u>, the number <u>amount</u> of disjoint sets should also decrease. Instead of seeking the root element and set union directly, we can make some optimizations towards these two methods to improve the overall performance, which are replacing SetUnion() to WeightedUnion() and SetFind() to CollapseFind(). This report is going to analyze the original methods and compare it with the two optimized functions.

2. Approach:

We are provided with 9 graphs with different size of vertexes and edges, we can analyze these function by 3 connect functions with: original SetFind and SetUnion, substitute SetUnion with WeightedUnion and substitute SetFind with CollapseFind. Then execute the three connect functions for each N times (set N = 100 as default) and calculate the total execute time. The main function pseudo code will be like the below:

```
Algorithm main()
{
    ReadGraph();
    t0 := GetTime();
    for ( i:=1 to N ) do Connect1();      // N is the repetition time
    t1 := GetTime();
    write((t1 - t0) / N, NS);             // connect1 's outcome
    for ( i:=1 to N) do Connect2();
    t2 := GetTime();
    write((t2 - t1) / N, NS);             // connect2's outcome
    for ( i:=1 to N) do Connect3();
    t3 := GetTime();
    write((t3 - t2) / N, NS);             // connect3's outcome
}
```

3. Analysis:

In the general connect function, we will do following things with some defined global variable:

Not clear!
Need more detailed explanations.

```
graph[v][2];                        // for storing the graph
v, e;                               // for storing the amount of edges and vertexes
p[v];                               // for storing each vertexes' parent element

AlgorithmConnect(graph, e, v)
{
    NS := |V|;                      // initially set the amount of disjoint sets to number of vertexes
    for every e = (vi, vj) belongs to graph do
    {
        Si := SetFind(vi);          // Find vi's root
        Sj := SetFind(vj);          // Find vj's root
        if (Si != Sj) then
        {
            NS := NS - 1;           // disjoint set decreases
            SetUnion(Si, Sj);       // they are now in the same subset
        }
    }
    for every vi belongs to V do
    {
        R[i] := SetFind(vi);        // check every vertexes' root element
    }
}

AlgorithmSetFind()
{
    while (p[i] >= 0) do i := p[i];     // seek until the element's parent is negative
    return i;                           // return root element
}

AlgorithmSetUnion()
{
    p[i] := j;              // make an element's parent to the other element (make link)
}
```

We will obtain a chart to record every elements' 'parent' element. While uses the SetFind() function to seek every elements' parent we can find the root of them, which has its parent denoted as -1. From this concept we can know whether 2 randomly picked element are in the same subset or not by checking if their root are the same. Since the 2 elements we pick are an edge, we can know that they are going to be connected, so if their root are not the same we can apply SetUnion()

function to link them up, making their root to be the same(note that the SetUnion has no order issue since the graph has no direction).

For further approach, we can improve the performance by replacing the *SetUnion* function to the *WeightedUnion* function, shown as below:

```
// If p[i] < 0, p[i] := element followed behind i;
Algorithm WeightedUnion()
{
        temp := p[i] + p[j];     // total elements after union
        if (p[i] > p[j]) then    // root i has fewer element
        {
                p[i] := j;       // i follows j
                p[j] := temp;
        }
        else                     // root j has fewer element
        {
                p[i] := temp;    // j follows i
                p[j] := i;
        }
}
```

What it had done to improve the overall speed is that rather than not determine who will be the parent as in SetUnion(), it decides who to be the parent by checking the amount of elements followed by each root. If we let the one with fewer weight to be the parent, then the total graph will be mow skewed since each union operation will increase the height by 1, making every time's SetFind() taking longer time to access. With this method applied, we can ensure that each time we

How?

call SetFind() we take no longer than $\lg|v| + 1$ times to find the root element. As we replace the SetUnion() function to WeightedUnion() function, this will be the whole Connect2() function.

Is this true?

In Connect3(), we substitute the SetFind() function with the further CollapseFind() function, which will directly make every parent element in the parent table to root, so that we don't have to make redundant loop iteration every time we call the SetFind() function.

functions

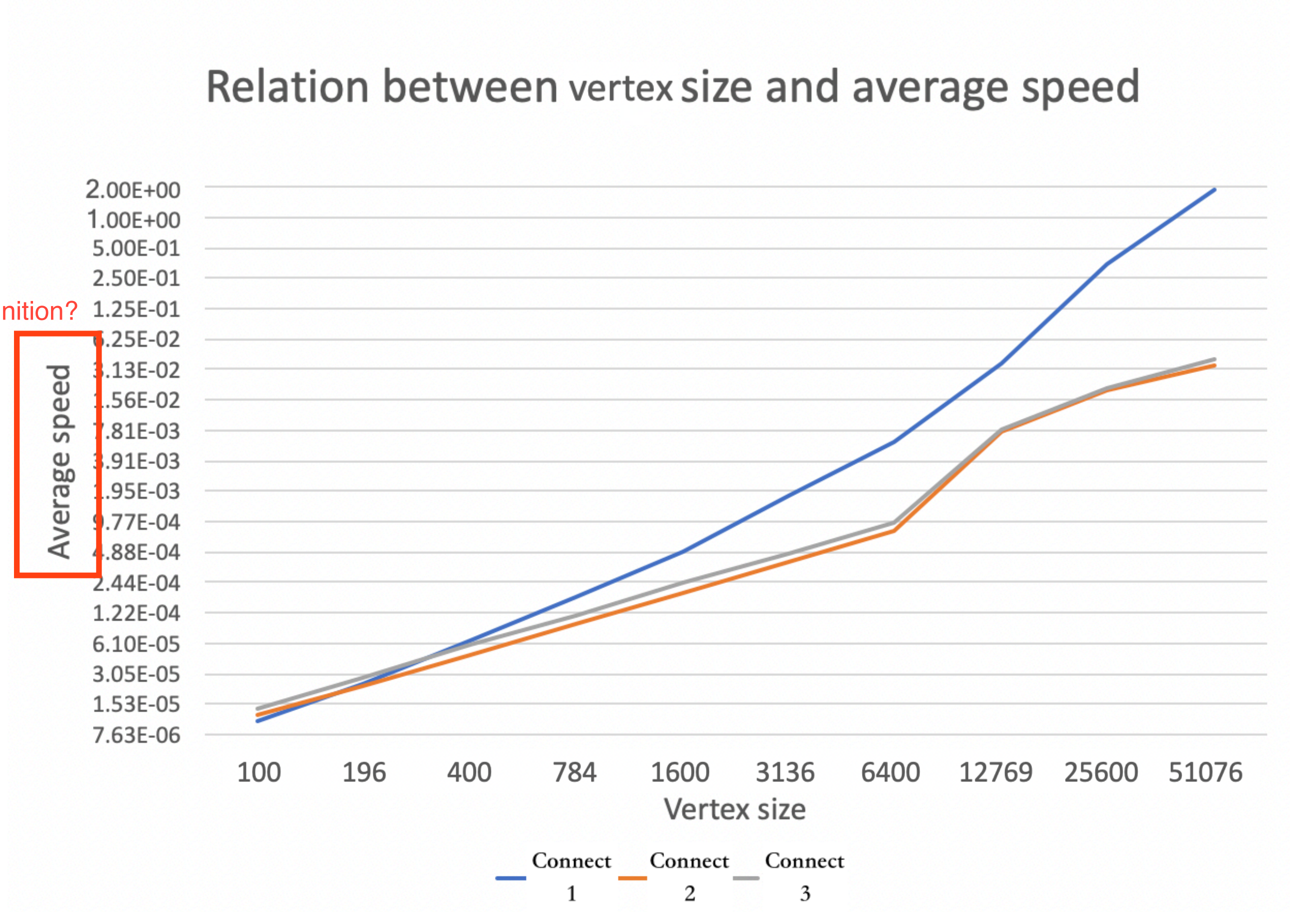
Among these three function, we will found out that changing from connect1 to connect2 will bring significant improvement. Since the domination of time complexity is located in the SetFind() and SetUnion() in the loop, we can know the difference by analyzing the for loop. In Connect1 there are two SetFind() and one SetUnion() in the for loop with $O(|E|)$ iterations. In worst case, SetUnion() will always be $O(1)$ since it just changes merely one element of the array.

However, if the tree representation of the graph is skewed, then the worst case occurs, the average access time per one SetFind() will be $|V| * (|V| + 1) / 2$ How?, making the total access time of connect1 become $|E| * (|V|^2 + |V|)$ times, making the complexity of $O(n^2)$ while connect2 using WeightedUnion will take $3|E| * (|V| * \lg|V|)$ times since the height of the tree representation of connect2 will not excess $\lg|V|$ How?, and the constant 3 comes from the comparison in weighted union function, so the complexity will be $O(n\lg n)$. In connect3, the collapse finding function sets all elements' parent directly to the root, however the total execution steps works similarly as connect2 since it also needs to change the value via the tree, complexity also being $O(n\lg n)$.

4. Results:

Plot of result:

Definition?



Unit : second

Each datasets's result:

dataset	amount of vertexes	Amount of edges	connect1(s)	connect2(s)	connect3(s)	Amount of disjoint sets
1	100	145	1.04E-05	1.17E-05	1.38E-05	1
2	196	303	2.42E-05	2.30E-05	2.79E-05	2
3	400	643	6.38E-05	4.78E-05	5.83E-05	1
4	784	1262	1.75E-04	9.37E-05	1.15E-04	2
5	1600	2597	4.99E-04	1.90E-04	2.40E-04	4
6	3136	5078	1.72E-03	3.88E-04	4.70E-04	3
7	6400	10452	5.93E-03	7.90E-04	9.70E-04	8
8	12769	20812	3.55E-02	7.50E-03	7.90E-03	8
9	25600	41864	3.50E-01	1.90E-02	2.00E-02	33
10	51076	83601	1.89E+00	3.50E-02	4.00E-02	50

Unit: as labeled

5. Observation and conclusion:

In previous assignments we are required to carry out several methods of searching and sorting with most of them almost the same time complexity, so it is important to compare the execution time between them and not just the overall worst case and average case complexity. However, in this assignment there are significant difference between the time complexity, if the data size become larger, there must be a more obvious tendency represented in the plot. After realizing what the weighted union are doing I found out that the order of being the parents can be so decisive on the performance. Furthermore, in my own opinion I think the number of disjoint sets might have the possibility to affect the overall performance since the more the amount of disjoint sets are, the fewer the amount of elements existing in one disjoint sets will be, making the SetFind being faster.

```
$ gcc hw02.c
$ a.out < g4.dat
|V| = 784, |E| =  1262
Connect1 CPU time: 4.24004e-05, Disjoint Sets: 2
Connect2 CPU time: 1.97506e-05, Disjoint Sets: 2
Connect3 CPU time: 2.62284e-05, Disjoint Sets: 2
```

Array 'r' should be declared with a constant dimension or allocated using dynamic memory method. Array dimension is $|V| + 1$, why?

---

score: 57.0

- Overall report writing
  - English writing needs more practice.
  - Writing can be more logical – terms should be defined before referred to.
- Report format
  - Need page numbers.
- Introduction
  - Introduction can still be strengthened
- Approach
  - Is your implementation matching the pseudo codes?
- Time/Space complexity
  - Computation complexity analyses need to be logical with sufficient details.
  - Space complexities?
- Results
  - Figures can be improved.
- Conclusion/observation
  - Can correlate your CPU times to the algorithm complexities
  - Can discuss the impact of the coefficient of Connect3 time complexity.
- Program format can be improved
  - Function definitions need comments.

hw03.c

```c
// EE3980 HW03 Network Connectivity Problem
// 106061225, 楊宇羲
// 2021/3/22

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

double GetTime(void);                   // Get local time
void ReadGraph(void);                   // Scan the graph
void Connect1(void);                    // SetFind & SetUnion
void Connect2(void);                    // SetFind & WeightedUnion
void Connect3(void);                    // CollapseFind & WeightedUnion
void SetUnion(int i, int j);            // Original union function
void WeightedUnion(int i, int j);       // Let the smaller subset become children
int SetFind(int i);                     // Original find root function
int CollapseFind(int i);                // Make elements' root be the same

int **graph;                            // 2D array for vertexes & edges
int *p;                                 // 1d array for storing parent element
int ns;                                 // Amount of disjoint sets
int v, e;                               // Amount of vertexes & edges

int main(void)
{

    int N = 100;                        // Repetitions
    int i;                              // For loop
    double  t0, t1, t2, t3, t;          // For time counting

    ReadGraph();                        // Scan in necessary information

    t0 = GetTime();                     // Start counting time
    for (i = 0; i < N; i++)             // Repeat N times
    {
        Connect1();                     // Execute Connect1 function
    }
    t1 = GetTime();                     // End of Connect1, start of Connect2
    t = (t1 - t0) / N;                  // Time calculation
```

```
    printf("Connect1 CPU time: %g, Disjoint Sets: %d\n", t, ns);

    for (i = 0; i < N; i++)          // Repeat N times
    {
        Connect2();                  // execute Connect2 function
    }
    t2 = GetTime();                  // End of Connect2, start of Connect3
    t = (t2 - t1) / N;               // Time calculation
    printf("Connect2 CPU time: %g, Disjoint Sets: %d\n", t, ns);

    for (i = 0; i < N; i++)          // Repeat N times
    {
        Connect3();                  // Execute Connect3 function
    }
    t3 = GetTime();                  // End of Connect3
    t = (t3 - t2) / N;               // Time calculation
    printf("Connect3 CPU time: %g, Disjoint Sets: %d\n", t, ns);

    return 0;
}

void ReadGraph(void)
Comments?
{
    int i;                                          // For loop

    scanf("%d %d", &v, &e);                         // Determine size of graph
    printf("|V| = %d, |E| =  %d\n", v, e);

    graph = (int**)malloc(e * sizeof(int*));    // Dynamic allocation
    p = (int*)malloc((v + 1) * sizeof(int));    // Dynamic allocation

    for (i = 0; i < e; i++)                         // Scan array
    {
        graph[i] = (int*)malloc(2 * sizeof(int));
        scanf("%d %d", &graph[i][0], &graph[i][1]);
    }
}

double GetTime(void)
Comments?
```

```
{
    struct timeval tv;

    gettimeofday(&tv, NULL);
    return tv.tv_sec + 1e-6 * tv.tv_usec;    // sec + micro sec
}

void Connect1(void)                          // SetFind and SetUnion
Comments?
{
    int i;                                   // For loop
    int r[v + 1];                            // For storing root element
Array dimension should be a constant int, not variable.

    ns = v;                                  // Initialize amount of Disjoint sets

    for (i = 0; i <= v; i++) p[i] = -1; // Initialize elements' root
    Why i in [0, v], |V| + 1 elements?
    for (i = 0; i < e; i++)
    {
        int S0 = SetFind(graph[i][0]);  // Start seek for each vertexes' root
Do not mix declarations with statements
        int S1 = SetFind(graph[i][1]);  // Start seek for each vertexes' root
        if (S0 != S1)
        {
            ns--;                            // Every union makes ds decreases
            SetUnion(S1, S0);                // Set connection if there isn't
        }
    }
    for (i = 0; i <= v; i++) r[i] = SetFind(i); // Find each elements' root
}

void Connect2(void)                          // SetFind and WeightedUnion
Comments?
{
    int i;                                   // For loop
    int r[v + 1];                            // For storing root element

    ns = v;                                  // Initialize amount of disjoint sets

    for (i = 0; i <= v; i++) p[i] = -1; // Initialize elements' root
```

```
    for (i = 0; i < e; i++)
    {
        int S0 = SetFind(graph[i][0]);  // Same as Connect1
```

Do not mix declarations with statements

```
        int S1 = SetFind(graph[i][1]);  // Same as Connect1
        if (S0 != S1)
        {
            ns--;                       // Every union makes ds decreases
            WeightedUnion(S0, S1);      // Uses the weighted union function
        }
    }
    for (i = 0; i <= v; i++) r[i] = SetFind(i); // Find each elements' root
}

void Connect3(void)                     // CollapseFind and WeightedUnion
```

Comments?

```
{
    int i;                              // For loop
    int r[v + 1];                       // For storing elements' root

    ns = v;                             // Initialize amount of disjoint sets

    for (i = 0; i <= v; i++) p[i] = -1; // Initialize elements' root
    for (i = 0; i < e; i++)
    {
        int S0 = CollapseFind(graph[i][0]); // Uses Collapse Find function
```

Do not mix declarations with statements

```
        int S1 = CollapseFind(graph[i][1]); // Uses Collapse Find function
        if (S0 != S1)
        {
            ns--;
            WeightedUnion(S0, S1);      // Uses the weighted union function
        }
    }
    for (i = 0; i <= v; i++) r[i] = SetFind(i); // Find each elements' root
}

int SetFind(int i)
```

Comments?

```
{
    while (p[i] >= 0) i = p[i];         // Trace until p[i] < 0 (root)
```

```

    return i;                            // i will be the root element
}

int CollapseFind(int i)
Comments?
{
    int r = i;                           // Initialization
    int temp;

    while (p[r] > 0) r = p[r];           // Trace until p[r] < 0 (root)
    while (i != r){     // This loop sets every elements' parent to root
    while (i != r) {     // This loop sets every elements' parent to root
        temp = p[i];    // Find next element
        p[i] = r;       // Update new root
        i = temp;       // Found next element
    }
    return r;
}

void SetUnion(int i, int j)
Comments?
{
    p[i] = j;           // Set one's parent as another (order matters)
}

void WeightedUnion(int i, int j)
Comments?
{
    int temp = p[i] + p[j]; // Total weight

    if (p[i] > p[j])        // If i has fewer children, j becomes parent
    {
        p[i] = j;
        p[j] = temp;        // New weight
    }
    else                    // If j has fewer children, i becomes parent
    {
        p[i] = temp;        // New weight
        p[j] = i;
    }
```

```
}
```