Report needs double space.

EE3980 Algorithms

hw04 Network Connectivity Problem

106061146 陳兆廷

Introduction:

In this homework, I will be analyzing, implementing, and observing 3 algorithms. The goal of the algorithms is to find connectivity in a graph. The input of them will be an array representing edges of a graph, and the output of them will be number of disjoint sets, and root table, which contains roots of each vertices' sets.

During the analysis analyzing process, I will be using table-counting method to calculate the time complexities of 3 algorithms. Furthermore, I will find the best-case, worst-case, and average-case conditions in 3 algorithms separately accordingly. Before implementing on C code, I will try to predict the result based on my analysis. Finally, I will calculate their space complexity for the **total spaces used by the algorithm**.

The implementation of 3 algorithms on C code mainly focus on the average time, best-case and worst-case conditions. 10 testing data are given by Professor Chang.

Lastly, the observation of them will be focusing on the time complexity of the results from implementation. Moreover, I will compare 3 algorithms with themselves ? and make some rankings. Finally, I will check the experimented results with my analysis.

Analysis:

1. Graph Representation:

There are several ways of representing graphs and sets in programming. In this homework, we use 2 methods to do this.

a. Edge representation:

This representation of a graph is using a 2d array to store all the edges in a graph. First row of the array indicates number of vertices(V) and edges(E) of this graph. And the rest E rows represent edges that connecting 2 nodes. For an example, in graph *Fig. 1*, the representation of this graph will be *Table 1*. There are 4 vertices and 3 edges connecting 1-3, 1-4, 2-4. There are no directions for edges.

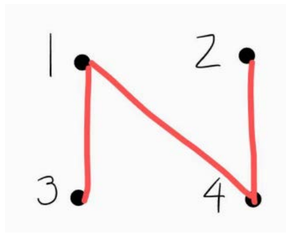


Fig 1

4↵	3↵
1↵	3↵
1↵	4↵
2↵	4↵

Table 1

b. Disjoint sets representation:

This way of representing disjoint sets is using a 1d array with size of vertices. Index of each position indicates an edge ? to that specific node, and if the index is -1, it means that it is the root. Take *Fig. 1* for an example again. The representation of disjoint sets in this graph is *Table 2*. They are all in the same set, therefore they are all connected to someone. Except 1, which is the root of this set.

1	2	3	4
-1	4	1	1

Table 2

c. Tree representation:

?

This way of representing a graph is seeing it as a tree. There are multiple combination of this tree. In *Fig. 2*, they are all tree representation of graph *Fig. 1*.

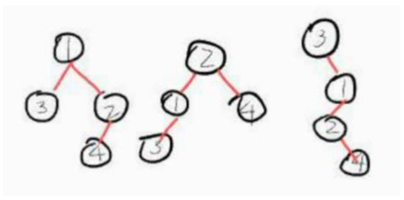


Fig 2 Difficult to read.

2. SetFind

a. Abstract:

SetFind finds the root of the given element in its disjoint set.

b. Algorithm:

```
// Find the set that element i is in.
// Input: element i
// Output: root element of the set.
Algorithm SetFind(i)
{
    while (p[i] >= 0) do i := p[i];
    return i ;
}
```

c. Proof of correctness:

During each iteration in the while loop, SetFind goes through the set closer and closer to the root. Eventually, it terminates when it finds -1, which indicate that this node is the root of the set containing this element *i*.

d. Time complexity:

	s/e	freq	total
`Algorithm SetFind(i)`	0	0	0
`{`	0	0	0
`    while (p[i] >= 0) do i := p[i];`	d	1	d
`    return i ;`	1	1	1
`}`	0	0	0
	d + 1		

The time complexity of SetFind is O(d), where d is the depth of this element i in its disjoint set.

<u>Best case:</u>

The best case for this is when all nodes in the graph is not connected. **O(1)** for this case since it finds root in one step, where d == 1.

<u>Worst case:</u>

The worst case for this happens then the graph is all connected. It travels through all the nodes to find the root. It will be **O(V)**, where d == V.

e. Space Complexity:

The algorithm uses 1 integer: i, and d elements in array p. The space complexity would be **O(d)**.

3. SetUnion

a. Abstract:

SetUnion sets 2 roots in the same set.

b. Algorithm:

```
// Form union of two sets with roots, i and j.
// Input: roots, i and j
// Output: none.
Algorithm SetUnion(i, j)
{
    p[i] := j ;
}
```

c. Proof of correctness:

It connects 2 elements, i and j, by assigning j directly to i

d. Time complexity:

	s/e	freq	total
`Algorithm SetUnion(i, j)`	0	0	0
`{`	0	0	0
`    p[i] := j ;`	1	1	1
`}`	0	0	0
	1		

The time complexity of SetFind is **O(1)**.

e. Space Complexity:

The algorithm uses 2 integers, i and j, and 1 element in array p. The space complexity would be **O(1)**.

4. CollapsingFind

a. Abstract:

CollapsingFind finds the root in the array and makes index of all elements in the set to be the root of the set.

b. Algorithm:

```
// Find the root of i, and collapsing the elements on the path.
// Input: an element i
// Output: root of the set containing i.
Algorithm CollapsingFind(i )
{
    r := i ; // Initialized r to i.
    while (p[r ] > 0) do r := p[r ] ; // Find the root.
    while (i ≠ r) do { // Collapse the elements on the path.
        s := p[i ] ; p[i ] := r ; i := s ;
    }
    return r ;
}
```

c. Proof of correctness:

The finding parts is same as **2. FindSet**. During the collapsing process, it goes through each element in this set and set them to the root of the set using while root, and terminates when all of them is connected to the root of the set.

d. Time complexity:

	s/e	freq	total
`Algorithm CollapsingFind(i )`	0	0	0
`{`	0	0	0

`r := i ;`	1	1	1
`while (p[r ] > 0) do r := p[r ] ;`	d	1	d
`while (i ≠ r) do {`	d	1	d
`    s := p[i ] ; p[i ] := r ; i := s ;`	3	d	3d
`}`	0	0	0
`return r ;`	1	1	1
`}`	0	0	0
	4d + 1		

The time complexity of CollapsingFind is O(d), where d is the depth of this element i in its disjoint set.

Best case:

The best case for this is when all nodes in the graph is not connected. **O(1)** for this case since it finds root in one step, where d == 1.

Worst case:

The worst case for this happens then the graph is all connected. It travels through all the nodes to find the root. It will be **O(V)**, where d == V.

Space Complexity:

The algorithm uses 3 integers, i, j and s, and d elements in array p. The space complexity would be **O(d)**.

5. WeightedUnion

a. Abstract:

WeightedUnion sets 2 roots in the same set and follows a certain weighted rule. In this case, it connects roots from the one with the more elements to the one with lesser elements.

b. Algorithm:

```
// Form union of two sets with roots, i and j, using the weighting rule.
// Input: roots of two sets i, j
// Output: none.
Algorithm WeightedUnion(i, j )
{
    temp := p[i ] + p[j ] ; // Note that temp < 0.
    if (p[i ] > p[j ]) then { // i has fewer elements.
        p[i ] := j ;
        p[j ] := temp ;
    }
    else { // j has fewer elements.
        p[j ] := i ;
        p[i ] := temp ;
    }
}
```

c. Proof of correctness:

When adding p[i] and p[j], it is computing the number of elements in the set. In the if statements, it connects the set that has fewer elements to another.

d. Time complexity:

	s/e	freq	total
`Algorithm WeightedUnion(i, j )`	0	0	0
`{`	0	0	0
`    temp := p[i ] + p[j ] ;`	1	1	1
`    if (p[i ] > p[j ]) then {`	1	1	1
`        p[i ] := j ;`	1	1	1
`        p[j ] := temp ;`	1	1	1
`    }`	0	0	0
`    else {`	0	0	0
`        p[j ] := i ;`	1	1	1
`        p[i ] := temp ;`	1	1	1
`    }`	0	0	0
`}`	0	0	0
	4 (either *if* or *else*)		

The time complexity of Weighted Union is **O(1)**.

e. Space Complexity:

The algorithm uses 3 integers, i, j and temp, and 2 elements in array p. The space complexity would be **O(1)**.

6. Connect1

a. Abstract:

Connect1 finds the connection of nodes in a graph, and output a table that stores all node's root.

b. Algorithm:

```
// Given G(V;E ) find connected vertex sets, generic version.
// Input: G(V;E )
// Output: Disjoint connected sets R[1 : n].
Algorithm Connectivity(G;R)
{
    for each $v_i \in V$ do $S_i$ := {$v_i$} ; // One element for each set.
    NS := |V| ; // Number of disjoint sets.
    for each e = ($v_i$; $v_j$) do { // Connected vertices
        $S_i$ := SetFind($v_i$) ; $S_j$ := SetFind($v_j$) ;
        if $S_i \neq S_j$ then { // Unite two sets.
            NS := NS - 1 ; // Number of disjoint sets decreases by 1.
            SetUnion($S_i$; $S_j$) ;
        }
    }
    for each $v_i \in V$ do { // Record root to R table.
        R[i ] := SetFind($v_i$) ;
    }
}
```

c. Proof of correctness:

The algorithm can be divided into 3 parts. Initializing, setting union, and recording root.

Initializing S_i to v_i makes all the vertices, which are nodes, roots of a disjoint set, as if there are no edges in a graph with V vertices.

The second part gradually add all the edges, which are connection between vertices, to the disjoint set array. We can say that before the i-th iteration, 1 ~ i – 1 edges are stored in the disjoint set S_i. During the i-th iteration, the i-th edge is checked if it is already connected through *SetFind*, otherwise it will be connected through *SetUnion*. At the start of i + 1 iteration, the cycle goes on. It terminates when all the edges are added to the disjoint set array S_i.

The recording roots part goes through all vertices and stores roots into root table.

d. Time complexity:

	s/e	freq	total
Algorithm Connectivity(G;R)	0	0	0
{	0	0	0
for each $v_i \in V$ do S_i := $\{v_i\}$;	V	2	2V
NS := |V| ;	1	1	1
for each e = (v_i; v_j) do {	E	1	E
S_i := SetFind(v_i) ;	d	E	E*d
S_j := SetFind(v_j) ;	d	E	E*d
if $S_i \neq S_j$ then {	1	E	E
NS := NS - 1 ;	1	E	E
SetUnion(S_i; S_j) ;	1	E	E
}	0	0	0
}	0	0	0
for each $v_i \in V$ do {	V	1	V
R[i] := SetFind(v_i) ;	1	V	V
}	0	0	0
}	0	0	0
d is the depth of each disjoint set	2Ed + 4(E + V) + 1		

Since *SetUnion* add connections directly, *Connect1* travels through all the nodes to find the root. It will be O(V) in *SetFind*, where d == V. And the overall **time complexity would be O(E*V).**

e. Space Complexity:

The algorithm uses 5 integers, S_i, S_j, NS and 2 i for looping, and V elements in array R, S, and E*2 elements in G. The space complexity would be **O(E + V)**.

7. Connect2

a. Abstract:

Connect2 finds the connection of nodes in a graph, and output a table that stores all node's root. The difference between *Connect1* and *Connect2* is that the *SetUnion* function changes to *WeightedUnion*.

b. Algorithm:

```
// Given G(V;E ) find connected vertex sets, generic version.
// Input: G(V;E )
// Output: Disjoint connected sets R[1 : n].
Algorithm Connectivity(G;R)
{
    for each $v_i \in V$ do $S_i$ := $\{v_i\}$ ; // One element for each set.
    NS := |V| ; // Number of disjoint sets.
    for each e = ($v_i$; $v_j$) do { // Connected vertices
        $S_i$ := SetFind($v_i$) ; $S_j$ := SetFind($v_j$) ;
        if $S_i \neq S_j$ then { // Unite two sets.
            NS := NS - 1 ; // Number of disjoint sets decreases by 1.
            WeightedUnion($S_i$; $S_j$) ;
        }
    }
    for each $v_i \in V$ do { // Record root to R table.
        R[i ] := SetFind($v_i$) ;
    }
}
```

c. Proof of correctness:

The proof of Connect1 can be used here.

d. Time complexity:

	s/e	freq	total
Algorithm Connectivity(G;R)	0	0	0
{	0	0	0
for each $v_i \in V$ **do** S_i := $\{v_i\}$;	V	2	2V
NS := |V| ;	1	1	1
for each e = (v_i; v_j) **do** {	E	1	E
S_i := SetFind(v_i) ;	d	E	E*d
S_j := SetFind(v_j) ;	d	E	E*d
if $S_i \neq S_j$ then {	1	E	E
NS := NS - 1 ;	1	E	E
WeightedUnion(S_i; S_j) ;	4	E	4E
}	0	0	0
}	0	0	0
for each $v_i \in V$ **do** {	V	1	V
R[i] := SetFind(v_i) ;	1	V	V
}	0	0	0
}	0	0	0
d is the depth of each disjoint set	2Ed + 7E + 4V + 1		

When *WeightedUnion* sets an union, first, we can presume a fact that when a set with m nodes is connected using *WeightedUnion*, the depth of it is no larger than $\lfloor \log m \rfloor + 1$.

Let us say it now unite node i and node j, which has a and m - a elements in them respectively, and m / 2 >= a >= 1. After they are connected, the depth of it is no larger than $\lfloor \log m - a \rfloor + 1$, which is also no larger than $\lfloor \log m \rfloor + 1$ and applies to the fact we presumed above. Furthermore, if we look at them separately, $\lfloor \log a \rfloor + 1 \leq \left\lfloor \log \frac{m}{2} \right\rfloor + 1 \leq$ $\lfloor \log m \rfloor + 1$ also applies to the fact.

In this case, m would be lesser than V. Therefore the time complexity would be **O(E lg V)**.

e. Space Complexity:

The algorithm uses 5 integers, S_i, S_j, NS and 2 i for looping, and V elements in array R, S, and E*2 elements in G. The space complexity would be **O(E + V)**.

8. Connect3

a. Abstract:

Connect3 finds the connection of nodes in a graph and output a table that stores all node's root. The difference between *Connect2* and *Connect3* is that the *SetFind* functions when adding connections change to *CollapsingFind*.

b. Algorithm:

```
// Given G(V;E ) find connected vertex sets, generic version.
// Input: G(V;E )
// Output: Disjoint connected sets R[1 : n].
Algorithm Connectivity(G;R)
{
    for each $v_i \in V$ do $S_i$ := $\{v_i\}$ ; // One element for each set.
    NS := |V| ; // Number of disjoint sets.
    for each e = ($v_i$; $v_j$) do { // Connected vertices
        $S_i$ := CollapsingFind($v_i$) ; $S_j$ := CollapsingFind($v_j$) ;
        if $S_i \neq S_j$ then { // Unite two sets.
            NS := NS - 1 ; // Number of disjoint sets decreases by 1.
            WeightedUnion($S_i$; $S_j$) ;
        }
    }
    for each $v_i \in V$ do { // Record root to R table.
        R[i ] := SetFind($v_i$) ;
    }
}
```

c. Proof of correctness:

The proof of Connect1 can be used here.

d. Time complexity:

	s/e	freq	total
`Algorithm Connectivity(G;R)`	0	0	0
`{`	0	0	0
for each $v_i \in V$ **do** S_i := $\{v_i\}$;	V	2	2V
NS := |V| ;	1	1	1
for each e = $(v_i;\ v_j)$ **do** {	E	1	E
S_i := CollapsingFind(v_i) ;	4d	E	4E*d
S_j := CollapsingFind(v_j) ;	4d	E	4E*d
if $S_i \neq S_j$ then {	1	E	E
NS := NS - 1 ;	1	E	E
WeightedUnion(S_i; S_j) ;	4	E	4E
}	0	0	0
}	0	0	0
for each $v_i \in V$ **do** {	V	1	V
R[i] := SetFind(v_i) ;	1	V	V
}	0	0	0
}	0	0	0
d is the depth of each disjoint set	4Ed + 7E + 4V + 1		

Since CollapsingFind has the same time complexity with SetFind, therefore, same as Connect2, the time complexity would be **O(E lg V).**

Not correct.

Although *CollapsingFind* is slower than *SetFind*, the benefit of *CollapsingFind* is that we could find faster in the last part of the algorithm, where it is finding the root table. Which would be faster, *Connect2* or *Connect3?* I must do some experiments to find out.

e. Space Complexity:

The algorithm uses 7 integers, S_i, S_j, NS, 2 in CollapsingFind and 2 i for looping, and V elements in array R, S, and E*2 elements in G. The space complexity would be **O(E + V)**.

9. Comparison:

	Connect1	Connect2	Connect3

Time Complexity	O(V*E)	O(E lg V)	O(E lg V)
Space Complexity	O(V+E)	O(V+E)	O(V+E)

Speed (fast>slow): Connect2 ? Connect3 >> Connect1.

10. Notes

It is too hard to find the Best-case and Worst-case scenarios if we do not change the V and E of each test input.

Implementation:

1. Speed Test:

Speed Test is to find the actual speed and time complexities of the three algorithms, Connect1, Connect2 and Connect3. We use 10 test inputs given by Professor and get the CPU runtimes before and after the algorithms perform their tasks. The implementation is done by my laptop.

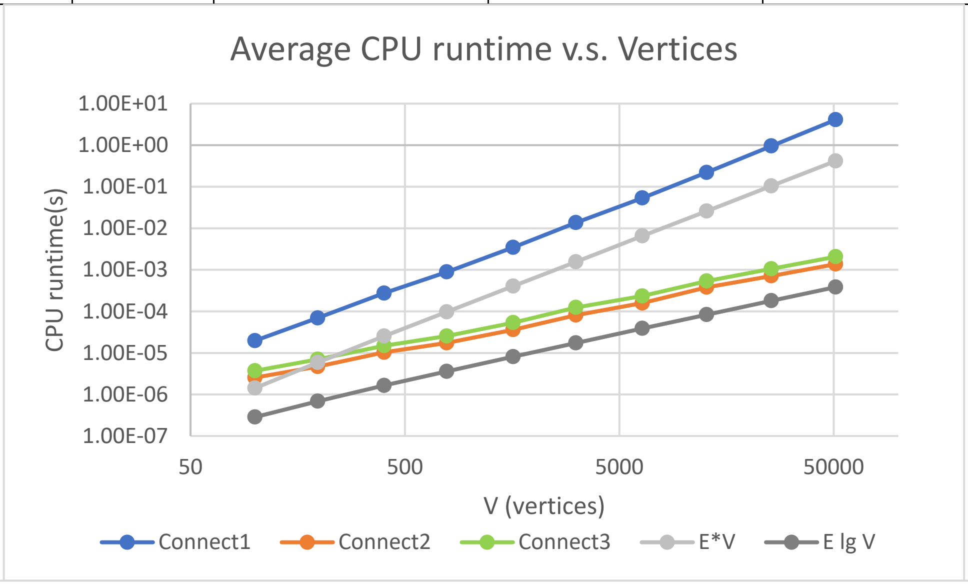
Workflow:

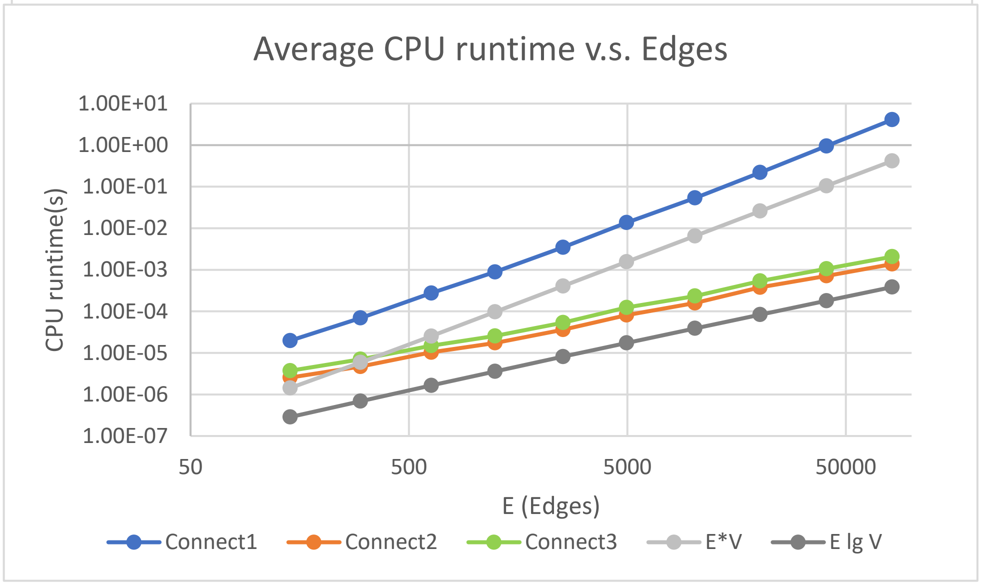
```
 t = GetTime();                              // initialize time counter
 for i := 0 to 100 do {
     Connect(G, R);                         // get connections
}
 t = (GetTime() - t) / 100;                   // calculate CPU time / iteration
```

Results:

V	E	Connect1	Connect2	Connect3
100	143	1.97E-05	2.55E-06	3.72E-06
196	300	6.93E-05	4.69E-06	7.01E-06
400	633	2.74E-04	1.03E-05	1.49E-05
784	1239	8.87E-04	1.74E-05	2.55E-05
1600	2538	3.47E-03	3.62E-05	5.35E-05
3136	4958	1.37E-02	8.11E-05	1.24E-04
6400	10201	5.35E-02	1.58E-04	2.34E-04
12769	20265	2.20E-01	3.79E-04	5.37E-04
25600	40795	9.53E-01	7.12E-04	1.06E-03

51076	81510	4.09E+00	1.37E-03	2.07E-03





2. CollapsingFind Test:

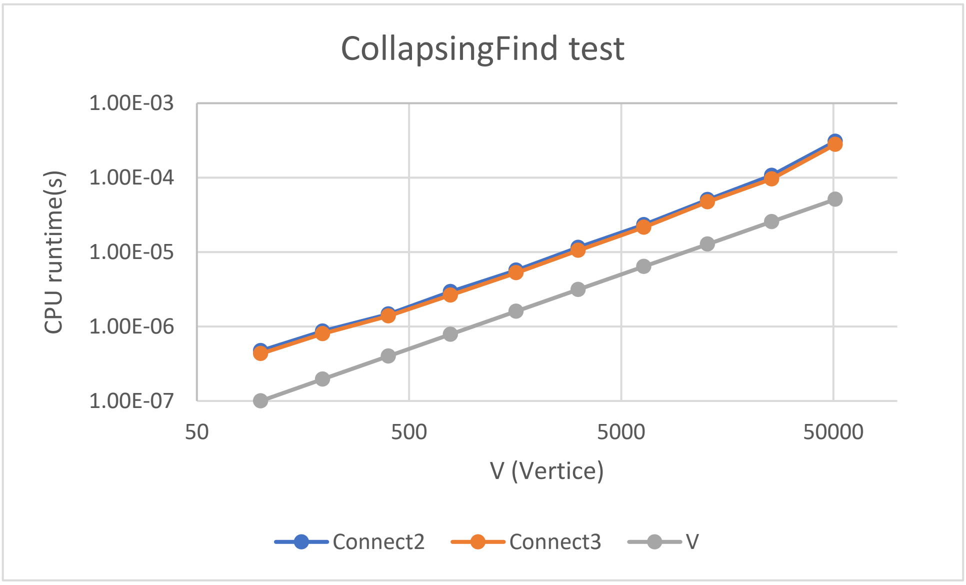
CollapsingFind Test is to find whether **CollapsingFind did make establishing root table faster or not**. We use 10 test inputs given by Professor and get the CPU runtimes **before and after the algorithms goes through SetFind in the root table finding part**.

Workflow:

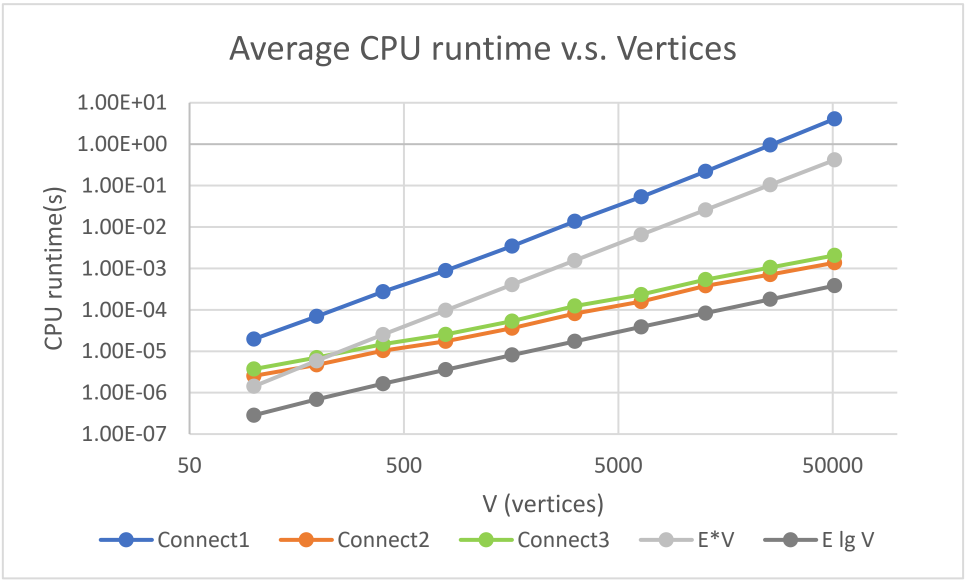
```
for i := 0 to 100 do {
    t += Connect(G, R, t);        // get time in root finding part
}
t = t / 100;                    // calculate CPU time / iteration
```

Results:

V	E	Connect2	Connect3
100	143	4.72E-07	4.32E-07
196	300	8.65E-07	8.01E-07
400	633	1.47E-06	1.39E-06
784	1239	2.93E-06	2.64E-06
1600	2538	5.70E-06	5.26E-06
3136	4958	1.15E-05	1.05E-05
6400	10201	2.33E-05	2.15E-05
12769	20265	5.06E-05	4.73E-05
25600	40795	1.07E-04	9.60E-05
51076	81510	3.07E-04	2.80E-04



Observation:



1. Speed, Time complexity:

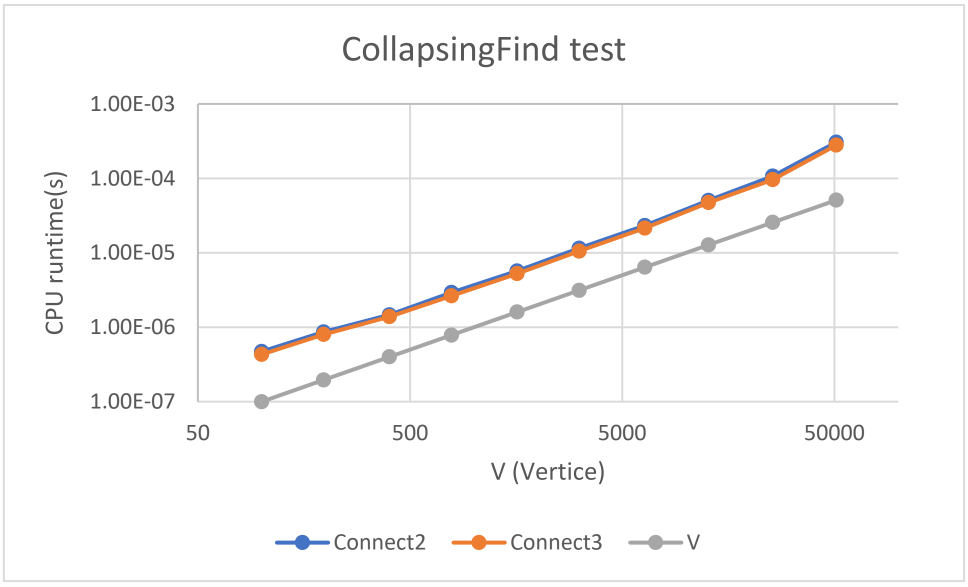
Actual Speed (> means faster): Connect2 > Connect3 > Connect1

The result matches my analysis precisely. The time complexity of *Connect1* is **O(E*V)** and those of *Connect2* and *Connect3* are **O(E lg V)**. The reason behind this is because *WeightedUnion* makes the depth of each set no more than log(V), so that is why *Connect1* is much slower than *Connect2* and *Connect3*.

Furthermore, *Connect2* is slightly faster than *Connect3*. The difference between them is that *Connect3* uses *CollapsingFind.* This should use some extra steps in during the make connection process, since *CollapsingFind* has 4 times more steps than *SetFind*. And we can know that if *CollapsingFind* actually did save some time when finding each nodes' root, it did not compensate the extra time spent during *CollapsingFind*.

Overall, the implemented results meet my analysis.

2. Benefit of CollapsingFind:



According to my experiment, ***CollapsingFind* did improve the root finding process**. Root finding part in *Connect3* is still slightly faster than that in *Connect2*. It shows that although *CollapsingFind* dragged the speed down in setting connection part, it helps user access root table faster. This fact meets our assumption. The reason behind the fact that overall Connect3 is still slower than Connect2 , I presume, is because *WeightedUnion* has done most of the improvements. Adding *CollapsingFind* did not improve much, but only dragged the speed down.

Conclusions:

1. Time and space complexities of the 3 algorithms:

	Connect1	**Connect2**	**Connect3**
Time Complexity	**O(V*E)**	**O(E lg V)**	**O(E lg V)**
Space Complexity	**O(V+E)**	**O(V+E)**	**O(V+E)**

2. Actual runtime comparison:

   Speed (> means faster): Connect2 > Connect3 > Connect1
3. *WeightedUnion* did most of the improvement among last 2 algorithms.
4. *CollapsingFind* **did** make root Sfinding process faster.

[Writing] hw04a.pdf spelling errors: Vertice(2)
[Coding] hw04.c spelling errors: initiallized(1)
[Report] should use double line space.
[Connect3] time complexity over-estimated.
[English] writing needs more practice.
[Good] effort in doing this homework!

Score: 88