Network Connectivity Problem

110191019 YU-TING CHUNG

Introduction

In this homework, I will implement and analyze four algorithms to solve the Network Connectivity Problem, which is a dynamic connectivity problem in graph theory. The goal is to determine whether two given nodes in a network are connected. This problem is commonly solved using the Disjoint Set Union (Union-Find) algorithm, which efficiently manages, and merges connected components.

The four functions to be implemented are:

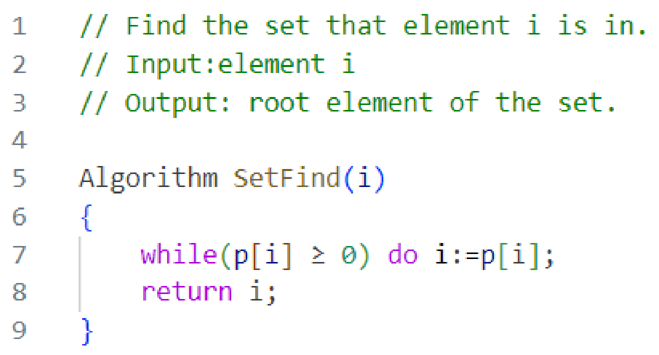
1. Connect 1: Uses standard SetFind and SetUnion.
2. Connect 2: Replaces SetUnion with WeightedUnion.
3. Connect 3: Uses WeightedUnion and CollapsingFind (except for one SetFind in line 13).
4. Connect4: Similar to Connect 3 but replaces CollapsingFind with PathHalvingFind.

The program will read 12 files (h1.dat to h12.dat) and test each function 100 times. The main function will measure CPU time and number of disjoint sets for each approach. Finally, I will compare the results with theoretical complexity analysis to evaluate efficiency.

Approach

➢ SetFind

1. 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;
}
```

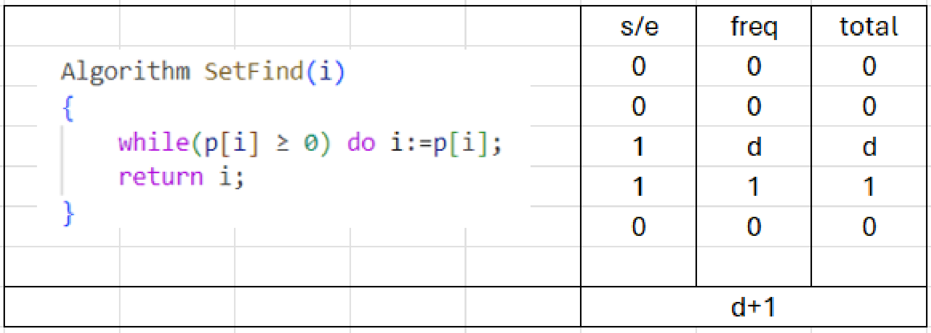
SetFind finds the root of the given element in its disjoint set.

2. Proof of correctness:

   With each iteration of the while loop, SetFind progressively traverses the parent node chain toward the root of the set. If p[i] is negative (e.g., -1), it signifies that the node is the root of the set and prompts the function to return it. This approach guarantees that every search operation correctly locates the root of the set containing i. It can ensure the algorithm functions accurately.

3. Time and space complexities:

Time complexity:



	s/e	freq	total
`Algorithm SetFind(i)`	0	0	0
`{`	0	0	0
`    while(p[i] ≥ 0) do i:=p[i];`	1	d	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.

Best case:

The best case for this is when all nodes in the graph are not connected.

It finds root in one step, where d = 1, so it is O(1) for this case.

Worst case:

The worst case happens when the graph is all connected. It

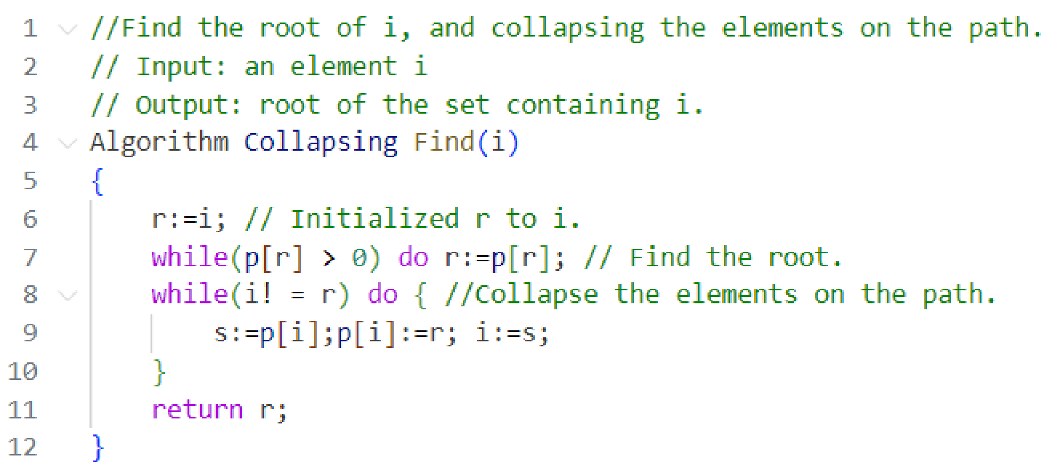
goes through all the nodes to find the root. It will be O(V), where d = V.

Space complexity:

1 integer: i, and d elements in array p. The space complexity is O(d).

- CollapsingFind

1. 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 Collapsing Find(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;
}
```

The CollapsingFind algorithm finds the root of a given element i in a disjoint set and applies path compression to optimize future queries. It first traces i to its root (r) by following parent pointers (p[r]). Then, it collapses the path by directly linking all visited nodes to the root (r), and reduces the depth of the tree.

2. Proof of correctness:

   CollapsingFind initially identifies the root of the set by a while loop. During the backtracking phase, it links all nodes along the path directly to the root, which effectively compresses the structure and optimizes future searches. Since all nodes ultimately point to the same root, the integrity of the set remains intact. It ensures the correctness of the algorithm.

3. Time and space complexities:

Time complexity:

	s/e	freq	total
`Algorithm Collapsing Find(i)`	0	0	0
`{`	0	0	0
`r:=i; // Initialized r to i.`	1	1	1
`while(p[r] > 0) do r:=p[r]; // Find the root.`	1	d	d
`while(i! = r) do { //Collapse the elements on the path`	1	d	d
`s:=p[i];p[i]:=r; i:=s;`	d	3	3d
`}`	0	0	0
`return r;`	1	1	1
`}`	0	0	0
		5d+2	

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

Best case:

When all nodes in the graph are not connected. It finds root in one step, where d =1, so it is O(1) for this case.

Worst case:

The worst case happens when the graph is all connected. It goes through all the nodes to find the root. It is O(V), where d = V.

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

➢ PathHalvingFind

1. Algorithm:

```
//Find the root of i,and reduce the tree height on the way
//Input: an element i
//Output: root of the set containing i.
Algorithm PathHalvingFind(i)
{
    if p[i] <0 the n return i; // i is a root
    r:=i; s:=p[r]; // r is the current node; s is its parent
    while p[s] >0 do{ // s is not a root
        p[r] :=p[s]; // reducting path length of r to the root
        r:=s; s:=p[s]; //move toward root
    }
    return s;
}
```

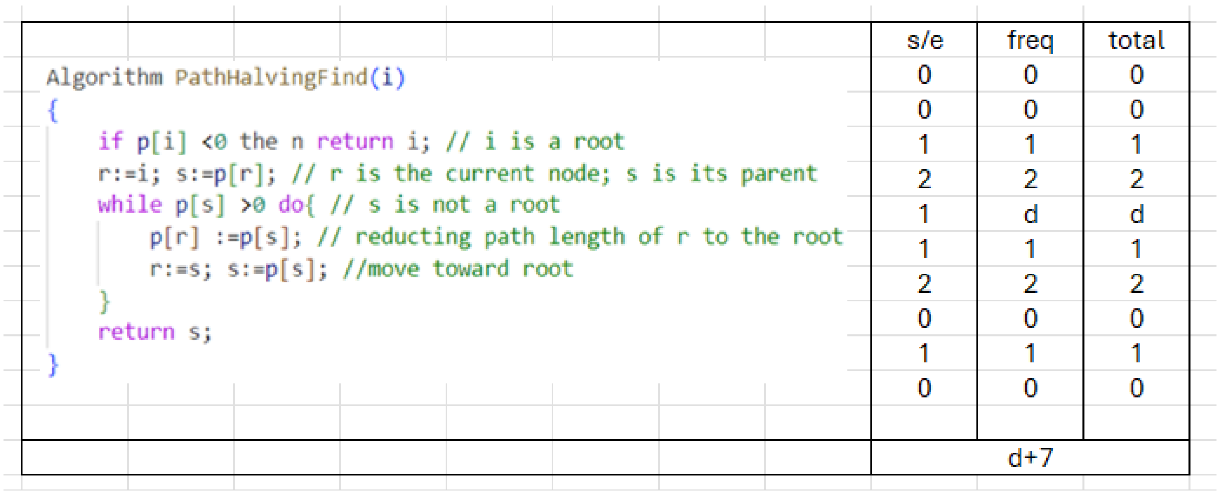
The PathHalvingFind algorithm finds the root of a given element i while it reduces the tree height. Instead of fully collapsing the path, it skips every other node by linking each node directly to its grandparent (p[r] = p[s]).

2. Proof of correctness:

PathHalvingFind reduces the tree height by progressively updating each node’s parent to its grandparent. In the while loop, every iteration effectively shortens the search path by half. Since all nodes remain connected to the correct root, and the tree structure is preserved, the correctness of the algorithm is guaranteed.

3. Time and space complexities:

Time complexity:



```
Algorithm PathHalvingFind(i)
{
    if p[i] <0 the n return i; // i is a root
    r:=i; s:=p[r]; // r is the current node; s is its parent
    while p[s] >0 do{ // s is not a root
        p[r] :=p[s]; // reducting path length of r to the root
        r:=s; s:=p[s]; //move toward root
    }
    return s;
}
```

s/e	freq	total
0	0	0
0	0	0
1	1	1
2	2	2
1	d	d
1	1	1
2	2	2
0	0	0
1	1	1
0	0	0
	d+7	

Best case:

When all nodes in the graph are not connected. It finds root in one step, where d =1, so it is O(1) for this case.

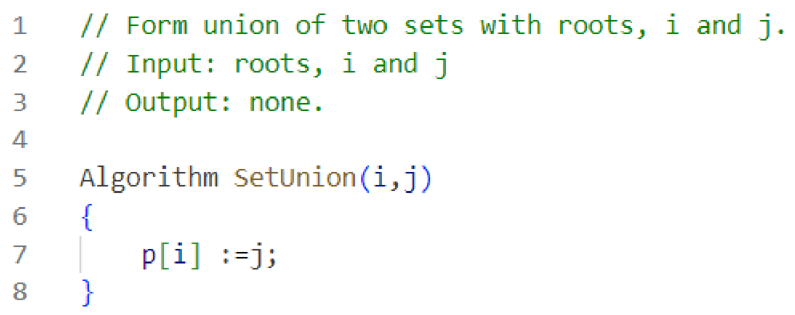
Worst case:

The worst case happens when the graph is all connected. It goes through all the nodes to find the root. It is O(V), where d = V.

Space complexity: 3 integers, i, r and s, and d elements in array p. The space complexity would be O(d).

➢ SetUnion

1. Algorithm:



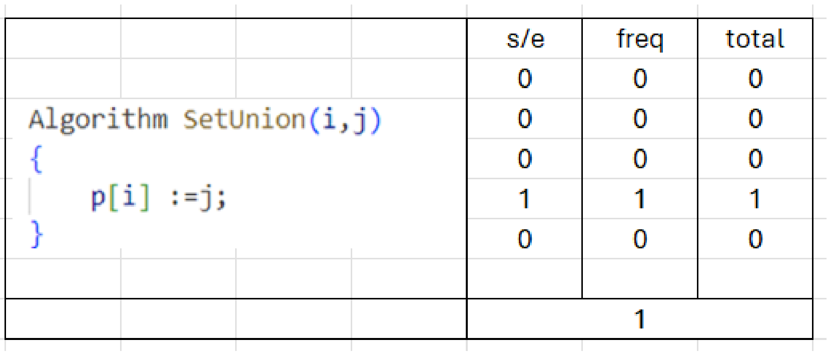
```
// Form union of two sets with roots, i and j.
// Input: roots, i and j
// Output: none.

Algorithm SetUnion(i,j)
{
    p[i] :=j;
}
```

SetUnion sets 2 roots in the same set.

2. Proof of correctness:

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

3. Time and space complexities:



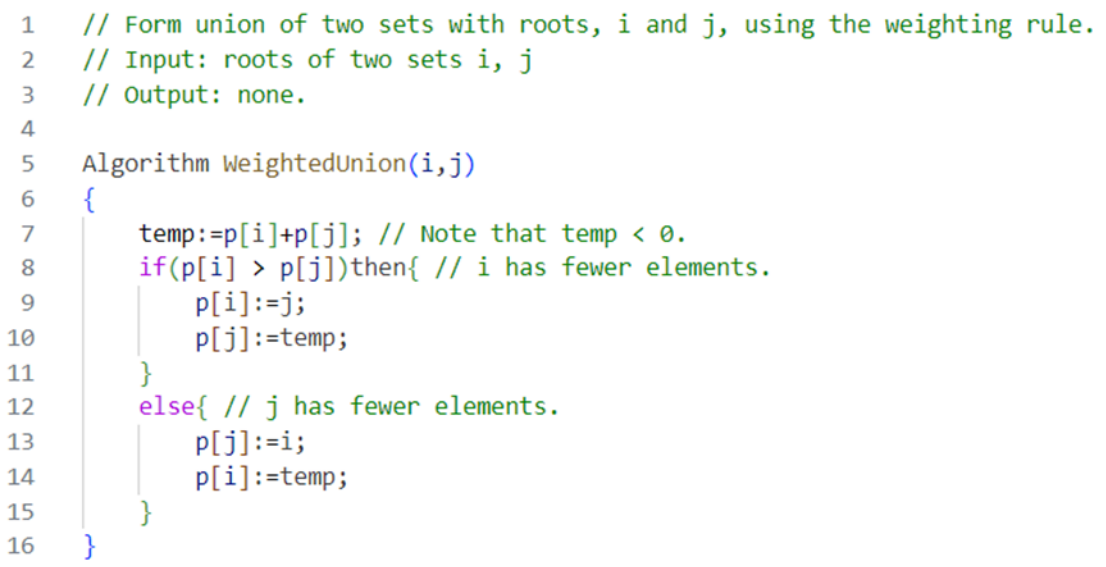
	s/e	freq	total
	0	0	0
`Algorithm SetUnion(i,j)`	0	0	0
`{`	0	0	0
`p[i] :=j;`	1	1	1
`}`	0	0	0
		1	

   Time complexity: O(1)

   Space complexity: 2 integers, i and j, and 1 element in array p. The space complexity is O(1).

➢ **WeightedUnion**

1. 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;
    }
}
```

The WeightedUnion algorithm merges two disjoint sets by linking the smaller tree to the larger tree. It updates the parent array (p[ ]), and stores the total size of the merged set in the root node.

2. Proof of correctness:

The WeightedUnion operation merges two sets by attaching the smaller set's root to the larger set's root, and this keeps the tree balanced. The union is correct because it properly combines the sets and maintains a low tree depth.

4. Time and space complexities:

Time complexity:

	s/e	freq	total
`Algorithm WeightedUnion(i,j)`	0	0	0
`{`	0	0	0
`    temp:=p[i]+p[j]; // Note that temp < 0.`	1	1	1
`    if(p[i] > p[j])then{ // i has fewer elements.`	1	1	1
`        p[i]:=j;`	1	1	1
`        p[j]:=temp;`	1	1	1
`    }`	0	0	0
`    else{ // j has fewer elements.`	0	0	0
`        p[j]:=i;`	1	1	1
`        p[i]:=temp;`	1	1	1
`    }`	0	0	0
`}`	0	0	0
		4	

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

Space complexity: 3 integers, i, j and temp, and 2 elements in array p. The space complexity is O(1).

➢ Connectivity

1. 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 vi∈V do Si := {vi} ; // One element for each set.
    NS :=|V|; // Number of disjoint sets.
    for each e = (vi,vj) do { // Connected vertices
        Si := SetFind(vi); Sj := SetFind(vj);
        if Si != Sj then { // Unite two sets.
            NS :=NS-1; // Number of disjoint sets decreases by 1.
            SetUnion(Si,Sj);
        }
    }
    for each vi ∈ V do { // Record root to R table.
        R[i] := SetFind(vi);
    }
}
```

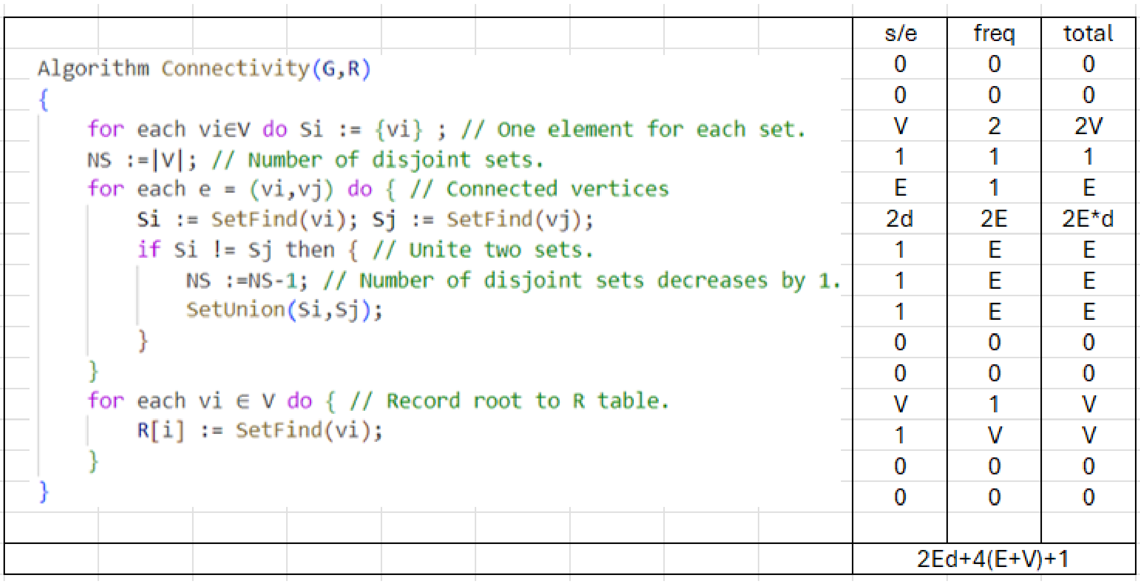
The Connectivity algorithm determines the connected components in a given graph G(V, E) by the Union-Find data structure. It initializes each node as its own set, then processes each edge to merge connected components by SetFind and SetUnion. It can reduce the number of disjoint sets. Finally, it records the root of each node in R and provides the final connected sets.

2. Proof of correctness:

The Connectivity algorithm correctly finds and outputs the connected components of the graph. It efficiently groups vertices into disjoint sets using the SetFind and SetUnion operations, so it ensures that each connected component is represented as a separate set in the output array R. Therefore, the algorithm is correct.

3. Time and space complexities:

Connect 1

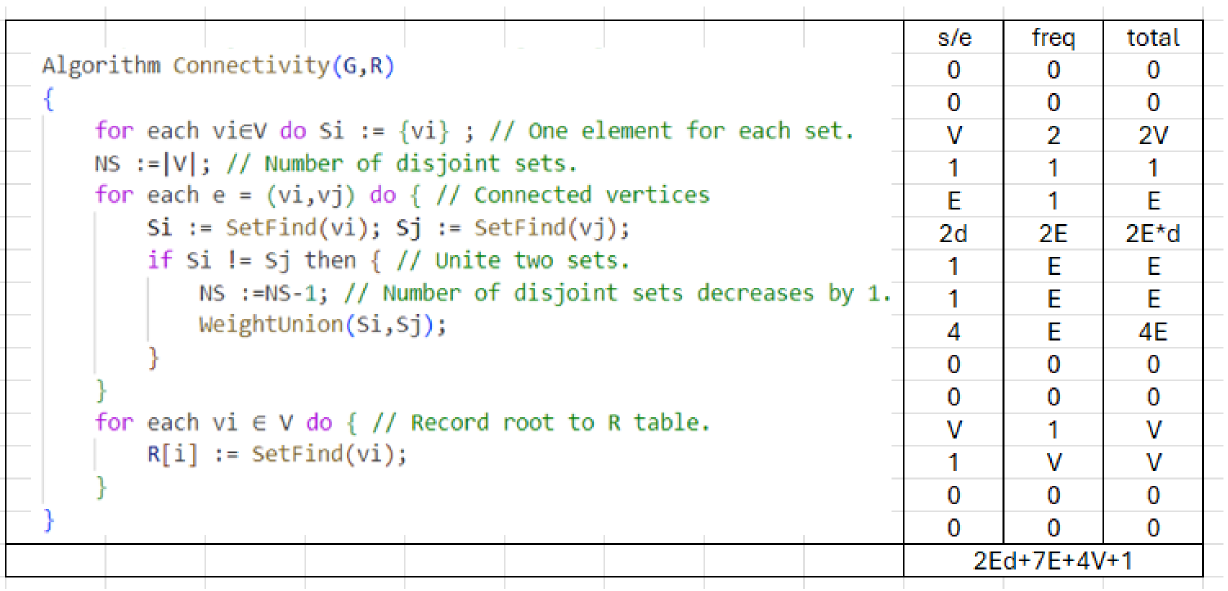


	s/e	freq	total
`Algorithm Connectivity(G,R)`	0	0	0
`{`	0	0	0
`    for each vi∈V do Si := {vi} ; // One element for each set.`	V	2	2V
`    NS :=|V|; // Number of disjoint sets.`	1	1	1
`    for each e = (vi,vj) do { // Connected vertices`	E	1	E
`        Si := SetFind(vi); Sj := SetFind(vj);`	2d	2E	2E*d
`        if Si != Sj then { // Unite two sets.`	1	E	E
`            NS :=NS-1; // Number of disjoint sets decreases by 1.`	1	E	E
`            SetUnion(Si,Sj);`	1	E	E
`        }`	0	0	0
`    }`	0	0	0
`    for each vi ∈ V do { // Record root to R table.`	V	1	V
`        R[i] := SetFind(vi);`	1	V	V
`    }`	0	0	0
`}`	0	0	0
	2Ed+4(E+V)+1		

SetUnion adds connections directly, and Connect1 goes through all the nodes to find the root, so the time complexity of SetFind, where d = V. And the overall time complexity is O(E*V).

Space complexity: 5 integers, NS and 2 i for looping, and V elements in array R, S, and E*2 elements in G. The space complexity is O(E + V).

Connect 2

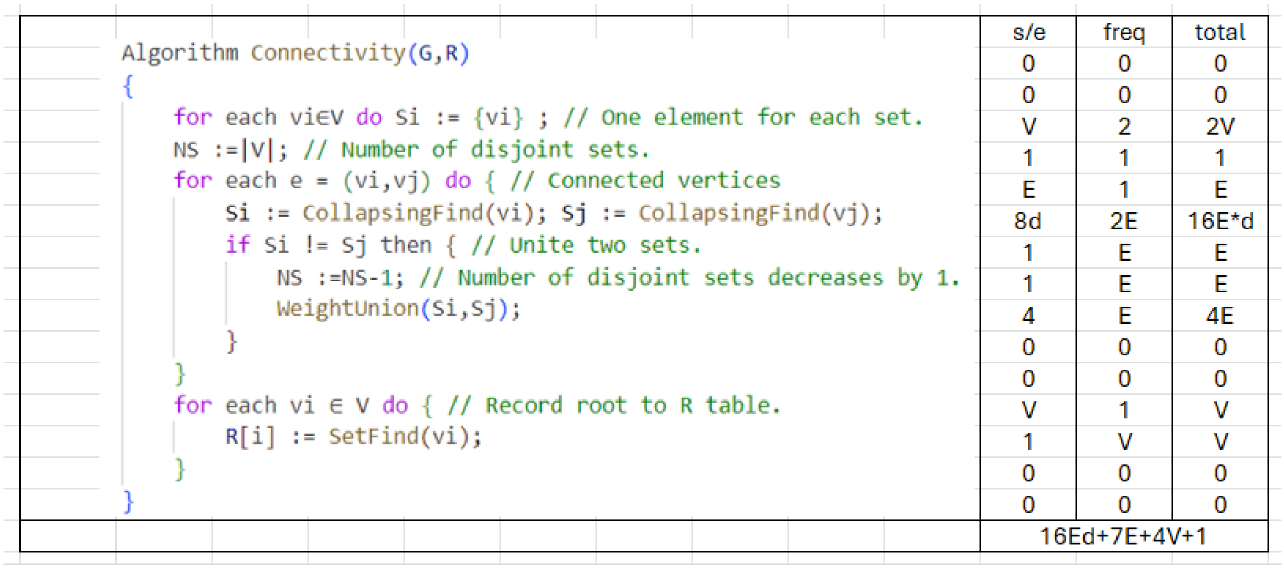


	s/e	freq	total
`Algorithm Connectivity(G,R)`	0	0	0
`{`	0	0	0
`    for each vi∈V do Si := {vi} ; // One element for each set.`	V	2	2V
`    NS :=|V|; // Number of disjoint sets.`	1	1	1
`    for each e = (vi,vj) do { // Connected vertices`	E	1	E
`        Si := SetFind(vi); Sj := SetFind(vj);`	2d	2E	2E*d
`        if Si != Sj then { // Unite two sets.`	1	E	E
`            NS :=NS-1; // Number of disjoint sets decreases by 1.`	1	E	E
`            WeightUnion(Si,Sj);`	4	E	4E
`        }`	0	0	0
`    }`	0	0	0
`    for each vi ∈ V do { // Record root to R table.`	V	1	V
`        R[i] := SetFind(vi);`	1	V	V
`    }`	0	0	0
`}`	0	0	0
	2Ed+7E+4V+1		

Time complexity: O(E lg V).

Space complexity : 5 integers, NS and 2 i for looping, and V elements in array R, S, and E*2 elements in G. The space complexity is O(E + V).

Connect 3

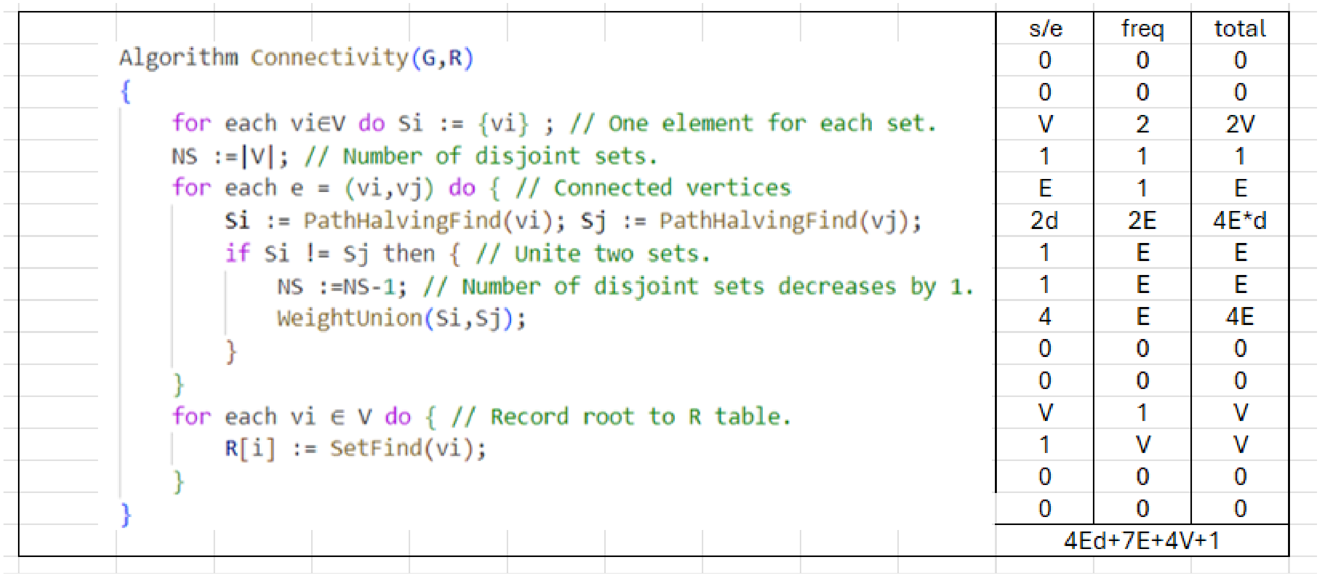


	s/e	freq	total
`Algorithm Connectivity(G,R)`	0	0	0
`{`	0	0	0
`for each vi∈V do Si := {vi} ; // One element for each set.`	V	2	2V
`NS :=|V|; // Number of disjoint sets.`	1	1	1
`for each e = (vi,vj) do { // Connected vertices`	E	1	E
`Si := CollapsingFind(vi); Sj := CollapsingFind(vj);`	8d	2E	16E*d
`if Si != Sj then { // Unite two sets.`	1	E	E
`NS :=NS-1; // Number of disjoint sets decreases by 1.`	1	E	E
`WeightUnion(Si,Sj);`	4	E	4E
`}`	0	0	0
`}`	0	0	0
`for each vi ∈ V do { // Record root to R table.`	V	1	V
`R[i] := SetFind(vi);`	1	V	V
`}`	0	0	0
`}`	0	0	0
	16Ed+7E+4V+1		

Time complexity: O(E lg V).

Space complexity : 7 integers, 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 is O(E + V).

Connect 4

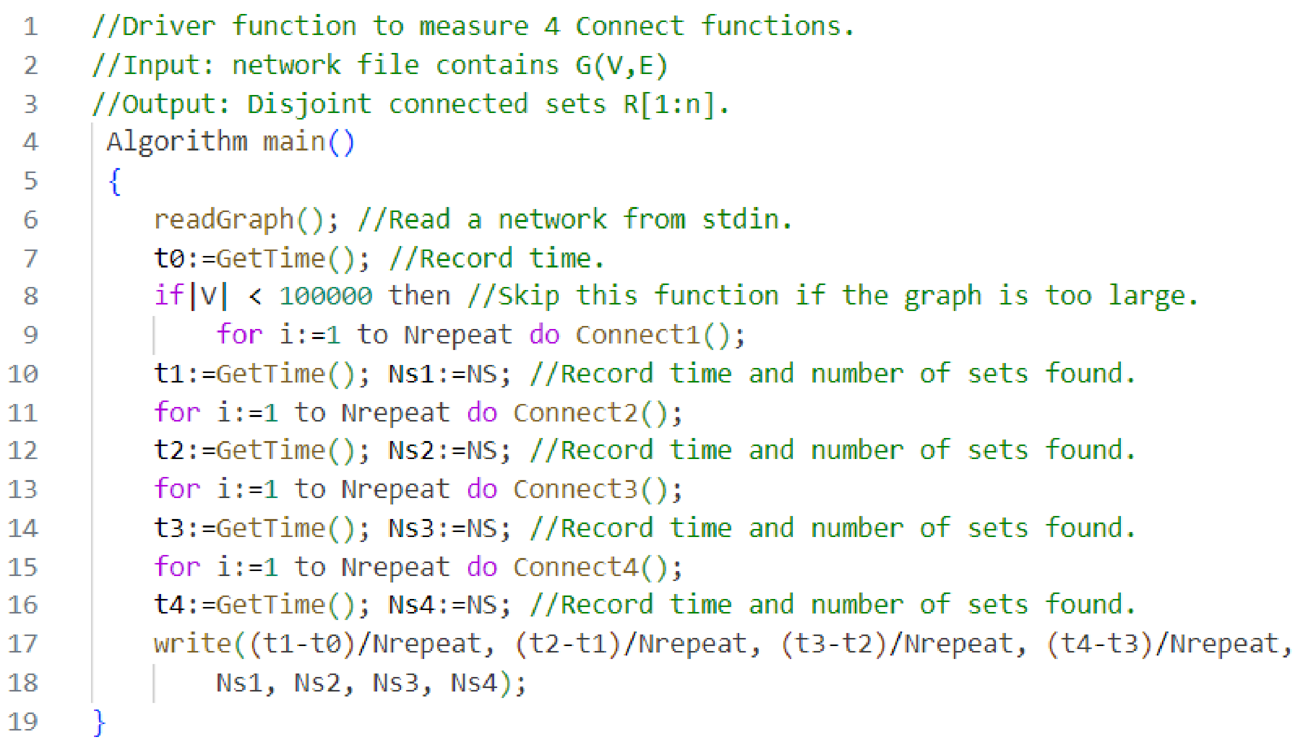


	s/e	freq	total
`Algorithm Connectivity(G,R)`	0	0	0
`{`	0	0	0
`for each vi∈V do Si := {vi} ; // One element for each set.`	V	2	2V
`NS :=|V|; // Number of disjoint sets.`	1	1	1
`for each e = (vi,vj) do { // Connected vertices`	E	1	E
`Si := PathHalvingFind(vi); Sj := PathHalvingFind(vj);`	2d	2E	4E*d
`if Si != Sj then { // Unite two sets.`	1	E	E
`NS :=NS-1; // Number of disjoint sets decreases by 1.`	1	E	E
`WeightUnion(Si,Sj);`	4	E	4E
`}`	0	0	0
`}`	0	0	0
`for each vi ∈ V do { // Record root to R table.`	V	1	V
`R[i] := SetFind(vi);`	1	V	V
`}`	0	0	0
`}`	0	0	0
	4Ed+7E+4V+1		

Time complexity: O(E lg V).

Space complexity : 8 integers, NS, 3 in PathHalvingFind and 2 i for looping, and V elements in array R, S, and E*2 elements in G. The space complexity is O(E + V).

- Main function



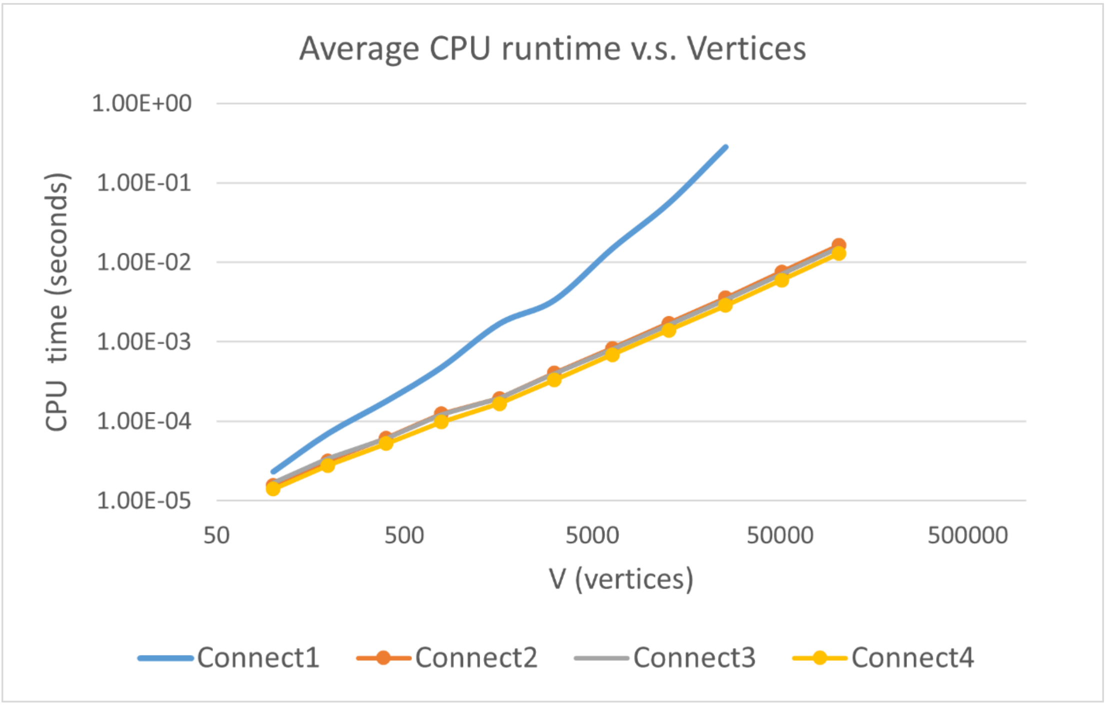
```
//Driver function to measure 4 Connect functions.
//Input: network file contains G(V,E)
//Output: Disjoint connected sets R[1:n].
 Algorithm main()
 {
    readGraph(); //Read a network from stdin.
    t0:=GetTime(); //Record time.
    if|V| < 100000 then //Skip this function if the graph is too large.
        for i:=1 to Nrepeat do Connect1();
    t1:=GetTime(); Ns1:=NS; //Record time and number of sets found.
    for i:=1 to Nrepeat do Connect2();
    t2:=GetTime(); Ns2:=NS; //Record time and number of sets found.
    for i:=1 to Nrepeat do Connect3();
    t3:=GetTime(); Ns3:=NS; //Record time and number of sets found.
    for i:=1 to Nrepeat do Connect4();
    t4:=GetTime(); Ns4:=NS; //Record time and number of sets found.
    write((t1-t0)/Nrepeat, (t2-t1)/Nrepeat, (t3-t2)/Nrepeat, (t4-t3)/Nrepeat,
        Ns1, Ns2, Ns3, Ns4);
}
```

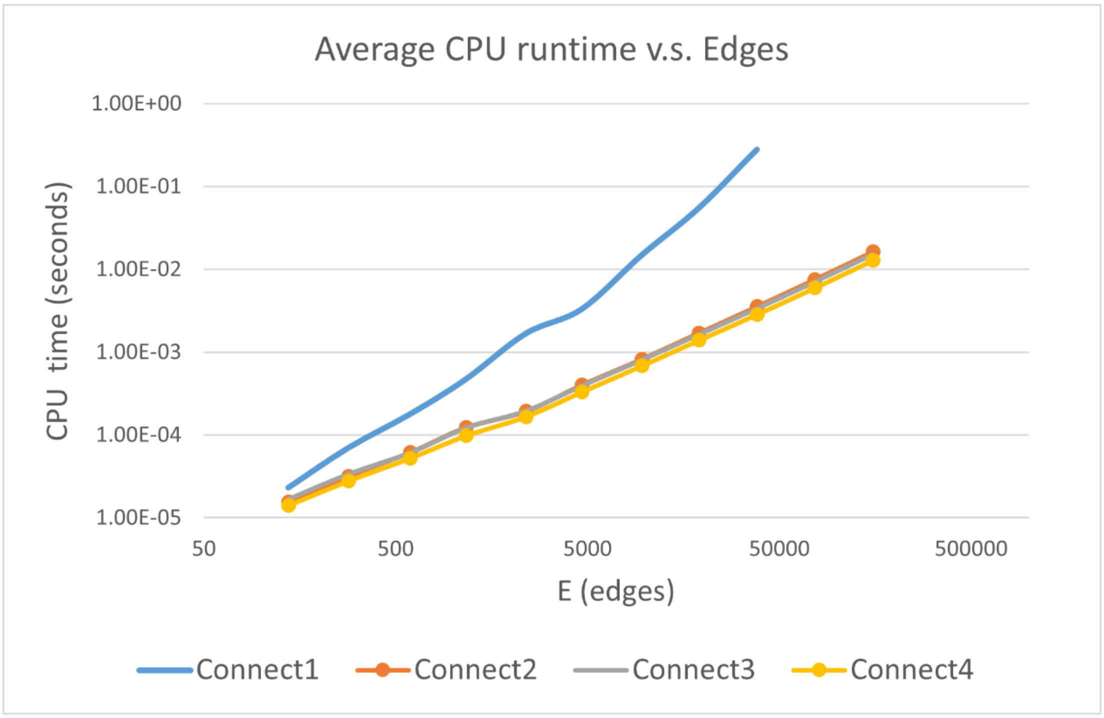
This pseudocode serves as the main driver function to evaluate the performance of four Union-Find algorithms on a given network G(V, E). First, it reads the network by readGraph and records the initial time (t0). If the number of nodes |V| < 100000, Connect1 is executed Nrepeat (100) times to measure its performance. Connect2, Connect3, and Connect4 will run no matter how large data is, and each run Nrepeat times. After each function, the execution time (t1 - t4) and the number of disjoint sets (Ns1 - Ns4) are recorded. Finally, the program outputs the average execution time for each function and the number of disjoint sets found.

Results and observations

1. Results

V	E	Connect1	Connect2	Connect3	Connect4
100	138	2.33316E-05	1.54996E-05	1.68300E-05	1.40500E-05
196	284	7.07197E-05	3.17907E-05	3.38197E-05	2.77901E-05
400	594	1.79999E-04	6.13809E-05	6.10089E-05	5.22614E-05
784	1164	4.77030E-04	1.23620E-04	1.21889E-04	9.78303E-05
1600	2387	1.69147E-03	1.92978E-04	1.95642E-04	1.65958E-04
3136	4676	3.34843E-03	4.03352E-04	3.92458E-04	3.30610E-04
6400	9606	1.50030E-02	8.26690E-04	8.08668E-04	6.83892E-04
12769	19064	5.55445E-02	1.70960E-03	1.63592E-03	1.39276E-03
25600	38273	2.81331E-01	3.57238E-03	3.36522E-03	2.85317E-03
51076	76530	1.35493E+00	7.56797E-03	7.04552E-03	5.96373E-03
102400	153574	0.00000E+00	1.63393E-02	1.51655E-02	1.28226E-02





2. Observation

(fast to slow): Connect4 > Connect3 > Connect2>Connect1

Connect1 is the slowest because it is the basic search method, which simply walks up the parent node to find the root node. If the tree structure is large, this method needs to traverse the entire chain, leading to longer runtime.

Connect2, Connect3, and Connect4 have similar running times and are faster than Connect1 because they all use WeightedUnion, a merging method that chooses to merge smaller sets into larger sets to reduce the height of the tree. Reduces the recursion depth during search, so it becomes faster.

The running time of Connect3 and Connect4 is similar because

PathHalvingFind only halves the time, while CollapsingFind points directly to the root. The optimization effects of the two are very similar, and the difference is almost invisible in actual operation. In addition, WeightedUnion ensures that the height of the tree is maintained at a low level, further reducing the number of search recursions, so the two methods perform almost the same under large data.

Conclusion

1. Time and space complexities comparison:

Algorithm	Optimization	Time Complexity	Space Complexity
Connect 1	SetFind + SetUnion	O(V*E)	O(E + V)
Connect 2	SetFind + WeightedUnion	O(E lgV)	O(E + V)
Connect 3	CollapsingFind + WeightedUnion	O(E lg V)	O(E + V)
Connect 4	PathHalvingFind + WeightedUnion	O(E lg V)	O(E + V)

2. Connect1 is the slowest because it is not optimized.
3. Connect 2 is improved through WeightedUnion to reduce the height when merging.
4. Connect3 and Connect4 use different optimization search methods, but because their time complexities are almost the same, their running times are close.