10802EE 398000 Algorithms

Homework 4. Network Connectivity Problem report

● Introduction

The connectivity problem is to determine if two given nodes in the network are connected or not. In this report, I will implement a general algorithm to solve this problem, including Connect1(), Connect2(), and Connect3(). The input of them is a Graph, and output is an array R such that R[v] is the node number of the root of the set of node v. Once this is done, one can test the connectivity of two nodes, i, j, by checking if R[i] = R[j].

Connect1(), Connect2(), and Connect3() are different in SetFind(i) (find the set that node i belongs to) and SetUnion(i, j) (unite two sets with roots i and j). The simple SetFind(i) and the simple SetUnion(i, j) are applied in Connect1(). The simple SetFind(i) and the WeightedUnion(i, j) are applied in Connect2(). The CollapsingFind(i) and the WeightedUnion(i, j) are applied in Connect3(). I will illustrate more details in **Approach** part.

I will also explain the correctness of SetUnion(i, j), WeightedUnion(i, j), SetFind(i), CollapsingFind(i), Connect1(), Connect2(), Connect3(),. After that, I will analyze the time complexity and space complexity of them. Last, I will implement Connect1(), Connect2(), and Connect3(), measure their CPU time, and compare the results with the theoretical results to see whether they are the same or not.

Approach

The psedocodes of SetUnion(i, j), WeightedUnion(i, j), SetFind(i), CollapsingFind(i), and a general algorithm to solve the network connectivity problem are shown below. I would like to explain the p array first, which appears in nearly every algorithm. P means "parent". The elements p[i] in p array means the parent node of the node i. For the root node r, p[r] will be less than zero.

```
// Form union of two sets with roots, i and j.
// Input: roots, i and j
// Output: none.
1 Algorithm SetUnion(i, j)
2 {
3         p[i] := j;
4 }
```

Algorithm 1 The pseudocode of SetUnion.

The SetUnion unites two sets with roots i and j. We let the parent node of the node i equal to j. This indicates the root i of one set is now the leaf of the other set. Thus, we make the set with root j include the set with root i. That is, the set with root i is now a subset of the set with root j.

The space complexity of SetUnion is O(1) because i and j are independent of the instance characteristics.

The time complexity is obviously O(1) because there is only one statement.

The problem of the simple SetUnion function is that it is possible to construct a degenerate tree (skewed tree), which makes the tree like a list. It will decrease the performance of SetFind(i) function which will be explained later. This issue can be alleviated by using the weighting rule.

```
// Form union of two sets with roots, i and j, using the weighting rule.
// Input: roots of two sets i, j
// Output: none.
1 Algorithm WeightedUnion(i, j)
2 {
3       temp := p[i] +p[j]; // Note that temp < 0.
4       if (p[i] > p[j]) then { // i has fewer elements.
5             p[i] := j;
6             p[j] := temp;
7       }
8       else { // j has fewer elements.
9             p[j] := i;
10            p[i] := temp;
11      }
12 }
```

Algorithm 2 The pseudocode of WeightedUnion.

First, I would like to introduce the **weighting rule: If the number of nodes in the tree with root i is less than the number of nodes in the tree with root j, then make j the parent of i; otherwise make i the parent of j**. Intuitively, this method can make the tree more balanced. That is, the height of the new tree can be less than the tree which SetUnion function constructs.

Algorithm 2 can implement the weighting rule. Because node i and node j are roots, p[i] and p[j] will store the number of nodes in each set (with minus sign). First, let temp = p[i] + p[j] which is the weight of the new tree (with minus sign). Second, judging which tree has the less node. If p[i] > p[j], we can get | p[i] | < | p[j] |. Thus, the tree with root i has the less number of nodes than the tree with root j has, then make j the parent of i (p[i] = j). Finally, let p[j] = temp, which is the number of nodes of the new tree (with minus sign). If p[i] <= p[j], it does the nearly same thing, but make i the parent of j.

The space complexity of WeightedUnion is O(1) because i and j are independent of the instance characteristics.

The time complexity is O(1) since there are only constant statements. However, the execution time will be longer than SetUnion because there will be four statements in each call rather than one. Although the execution time is a little bit longer than SetUnion, it can construct a more balanced tree which is better for SetFind. Furthermore, **let T be a tree with m nodes created as a result of a sequence of unions each performed using WeightedUnion algorithm. The height of T is no greater than $\lfloor \log_2 m \rfloor + 1$.** The proof is shown below:

Using mathematical induction, the first step is true when two sets of one element are united. Assume it is true for m – 1 operations, consider the last step of the union operations, WeightedUnion(k, j). If set j has a elements, then set k has m – a elements. And, 1 <= a <= m / 2. The height of T must be the same as that of k or one more than that of j. In the former case, the height of T is <= $\lfloor \log_2 (m-a) \rfloor + 1$ <= $\lfloor \log_2 m \rfloor + 1$. In the latter case, the height of T is <= $\lfloor \log_2 a \rfloor + 2$ <= $\lfloor \log_2 m/2 \rfloor + 2$ <= $\lfloor \log_2 m \rfloor + 1$.□

```
// Find the set that element i is in.
// Input: element i
// Output: root element of the set.
1 Algorithm SetFind(i)
2 {
3       while (p[i] ≥ 0) do i := p[i];
4             return i;
5 }
```

Algorithm 3 The pseudocode of SetFind.

The SetFind(i) will find the root node of the set that i belongs to. The while loop in line3 keeps traveling the tree from leaf to root. The statement i = p[i] keeps assigning i to its parent node. When p[i] < 0, we find the root node then return.

The space complexity of SetFind is O(1) because i is independent of the instance characteristics.

The time complexity is O(h) where h is the height of the set which the node i belongs to. The reason is that the while loop keeps traveling the tree from leaf to root, we needs to travel h times for the worst case.

```
// Find the root of i, and collapsing the elements on the path.
// Input: an element i
// Output: root of the set containing i.
1 Algorithm CollapsingFind(i)
2 {
3       r := i; // Initialized r to i.
4       while (p[r] > 0) do r := p[r]; // Find the root.
5       while (i ≠ r) do { // Collapse the elements on the path.
6             s := p[i]; p[i] := r; i := s;
7       }
8       return r;
9 }
```

Algorithm 4 The pseudocode of CollapsingFind.

From the analysis of **Algorithm 3**, it seems that the height of the tree is important. We can reduce the time complexity by reducing the height. Collapsing Rule is a method to improve the height of a set: **If j is an element on the path from i to its root and p[i] ≠ root(i), then set p[j] to root(i)**.

Algorithm 4 implements the collapsing rule. The while loop in line 4 is same as SetFind(). We

can get the root node r of the set which node i belongs to. Line 5 to line 7 are collapsing rule. We collapse the elements on the path, making their parents equal to the root of the set. Thus, we can make non-root nodes point directly to the root.

The space complexity of CollapsingFind is O(1) because i is independent of the instance characteristics.

The time complexity of CollapsingFind is also O(h). However, it roughly doubles the time for an individual find. The advantage of it is to reduce the worst-case time over a sequence of finds. Thus, using CollapsingFind may not necessarily improve the CPU time. It depends on the structure of sets and the elements we want to find.

```
// Given G(V, E) find connected vertex sets, generic version.
// Input: G(V, E)
// Output: Disjoint connected sets R[1 : n].
1 Algorithm Connectivity(G,R)
2 {
3       for each vi∈V do Si := {vi} ; // One element for each set.
4       NS := |V|; // Number of disjoint sets.
5       for each e = (vi,vj) do { // Connected vertices
6             Si := SetFind(vi); Sj := SetFind(vj);
7             if Si ≠ Sj then { // Unite two sets.
8                   NS := NS −1; // Number of disjoint sets decreases by 1.
9                   SetUnion(Si,Sj);
10            }
11      }
12      for each vi ∈ V do { // Record root to R table.
13            R[i] := SetFind(vi);
14      }
15 }
```

Algorithm 5 The algorithm to check whether two nodes in the network are connected or not.

The input of the Connectivity is a graph G(V, R). The output is R array which stores the root node of every vertex. In line 2, we assign one vertex for each set. Thus, we have NS = |V| sets initially. Then we begin estimating each edge to unite two sets that two vertices of an edge belong to. Finally, we can get the array which stores the parent node of every vertex and has already been united. Then we can use line 12 to 14 to find the root of each vertex and get the R array.

Connect1(), Connect2(), and Connect3() are nearly same as Algorithm Connectivity. Specifically, three functions are different in SetFind and SetUnion implemented. Connect1() uses the SetFind function listed in **Algorithm 3** and the SetUnion function as **Algorithm 1**. Connect2() replaces the SetUnion function by **Algorithm 2**, WeightedUnion but keep the same SetFind function. Connect3() not only uses WeightedUnion but also replaces the SetFind on line 6 by **Algorithm 4,** CollapsingFind. But, the SetFind function on line 13 remains as it is.

The space complexity of this algorithm will be O(n + G), where n is the size of array R and G is the size of the graph. The size of the graph depends on how we store it.

The time complexity analysis of three kinds of Connect function is shown below:

I. Connect1() : We focus on the for loop for asymptotic analysis. For line 3 in **Algorithm 5,** Connectivity, we get O(V). For line 5 to 11, we get O(E * h). For line 12 to 14 we get O(V * h), where h is the height of the set which node i is in. Thus, the time complexity is generally max{O(V), O(E * h), O(V * h)}.

II. Connect2() : The only difference is that h must be no greater than $\lfloor \log_2 V \rfloor + 1$. We get

$O(E * \log_2 V)$ and $O(V * \log_2 V)$ for line 5 to 11 and line 12 to 14 respectively. Thus, the time complexity is generally $\max\{O(V), O(E * \log_2 V), O(V * \log_2 V)\}$.

III. Connect3() : The time complexity is same as Connect2() because the time complexity of CollapsingFind is same as SetFind() which is O(h). Thus, the time complexity is generally $\max\{O(V), O(E * \log_2 V), O(V * \log_2 V)\}$.

It is possible to make the representation of time complexities easier by observing the input data. I will show it in the next page.

```
// Driver function to measure 3 Connect functions.
// Input: network file contains G(V,E )
// Output: Disjoint connected sets R[1 : n].
1 Algorithm main()
2 {
3       readGraph(); // Read a network from stdin.
4       t0 := GetTime(); // Record time.
5       for i := 1 to Nrepeat do Connect1();
6       t1 := GetTime(); Ns1 := NS ; // Record time and number of sets found.
7       for i := 1 to Nrepeat do Connect2();
8       t2 := GetTime(); Ns2 := NS ; // Record time and number of sets found.
9       for i := 1 to Nrepeat do Connect3();
10      t3 := GetTime(); Ns3 := NS ; // Record time and number of sets found.
11      write((t1−t0)/Nrepeat, (t2−t1)/Nrepeat, (t3−t2)/Nrepeat, Ns1, Ns2, Ns3);
12 }
```

Algorithm 6 The pseudocode of main function.

	Space complexity	Time complexity
SetUnion(i, j)	O(1)	O(1)
WeightedUnion(i, j)	O(1)	O(1)
SetFind(i)	O(1)	O(h)
CollapsingFind(i)	O(1)	O(h)
Connect1(G, R)	O(n + G)	max{O(V), O(E * h), O(V * h)}
Connect2(G, R)	O(n + G)	max{O(V), O(E * $\log_2$ V), O(V * $\log_2$ V)}
Connect3(G, R)	O(n + G)	max{O(V), O(E * $\log_2$ V), O(V * $\log_2$ V)}

Table 1 the general space and time complexities of **Algorithm1** to **Algorithm5.**
Note that h is the height of the set which the node i belongs to. n is the size of array R. G is the size of the graph. E is the number of edges. V is the number of vertices.

We may make the complexity representation much easier. In this homework, I use a structure to store two vertices of each edge and a p array to represent each vertex. Thus, the space complexity of G is equal to V + 2E, where E is the number of edges. Besides, the size of array R is equal to V, the number of vertices. Consequently, the space complexity is equal to O(2V + 2E).

For time complexity, it seems that the number of vertices of the input data g1.dat to g10.dat are all less than the number of edges. So the time complexities of Connect1, Connect2, and Connect3 can be written as max{ O(E * h), O(V * h)}, max{ O(E * $\log_2$ V), O(V * $\log_2$ V)}, and max{ O(E * $\log_2$ V), O(V * $\log_2$ V)}.

Further, it seems that number of sets (shown in **table 3**) is small relative to the number of vertices. Thus, it is probably a skew tree. The height of a skew tree is equal to V, so the time complexity of Connect1 can be written as max{O(E * V), O(V * V)}.

Finally, from **table 3,** it seems that when the number of vertices double, the number of edges is also double between each testcase. Thus, we can replace E by V for asymptotic time analysis in this homework. We get $O(V^2)$, O(V * $\log_2$V), and O(V * $\log_2$V).

	Space complexity	Time complexity
SetUnion(i, j)	O(1)	O(1)
WeightedUnion(i, j)	O(1)	O(1)
SetFind(i)	O(1)	O(h)
CollapsingFind(i)	O(1)	O(h)
Connect1(G, R)	O(2V + 2E)	$O(V^2)$
Connect2(G, R)	O(2V + 2E)	$O(V * \log_2 V)$
Connect3(G, R)	O(2V + 2E)	$O(V * \log_2 V)$

Table 2 the space and time complexities of **Algorithm1** to **Algorithm5** in this homework.
Note that h is the height of the set which the node i belongs to. G is the size of the graph. E is the number of edges. V is the number of vertices.

● Results and Observation

Graph (Input)	G(V, E)	CPU Time (seconds)			Number of Sets		
		Connect1()	Connect2()	Connect3()	Ns1	Ns2	Ns3
g1.dat	G(100, 143)	5.38206E-05	9.17912E-06	1.19901E-05	1	1	1
g2.dat	G(196, 300)	1.84791E-04	1.77193E-05	2.32506E-05	3	3	3
g3.dat	G(400, 633)	6.43389E-04	3.11112E-05	4.00686E-05	1	1	1
g4.dat	G(784, 1239)	2.01368E-03	4.83298E-05	6.47807E-05	3	3	3
g5.dat	G(1600, 2538)	7.32569E-03	9.77492E-05	1.30632E-04	5	5	5
g6.dat	G(3136, 4958)	2.64130E-02	1.91371E-04	2.55060E-04	6	6	6
g7.dat	G(6400, 10201)	1.09307E-01	3.85060E-04	5.17731E-04	9	9	9
g8.dat	G(12769, 20265)	4.28036E-01	7.72879E-04	1.04099E-03	10	10	10
g9.dat	G(25600, 40795)	1.73264E+00	1.55191E-03	2.08137E-03	54	54	54
g10.dat	G(51076, 81510)	6.77601E+00	3.08869E-03	4.17918E-03	92	92	92

Table 3 CPU Time and Number of Sets (Ns) results table

Table 3 consists with the theoretical result shown in **Table 2**. When number of vertices double, the CPU time of Connect1() is about four times. The CPU time of Connect2() and Connect3() is nearly double because $2 * \log_2 2 = 2$.

The following figures are the comparison of three algorithms:

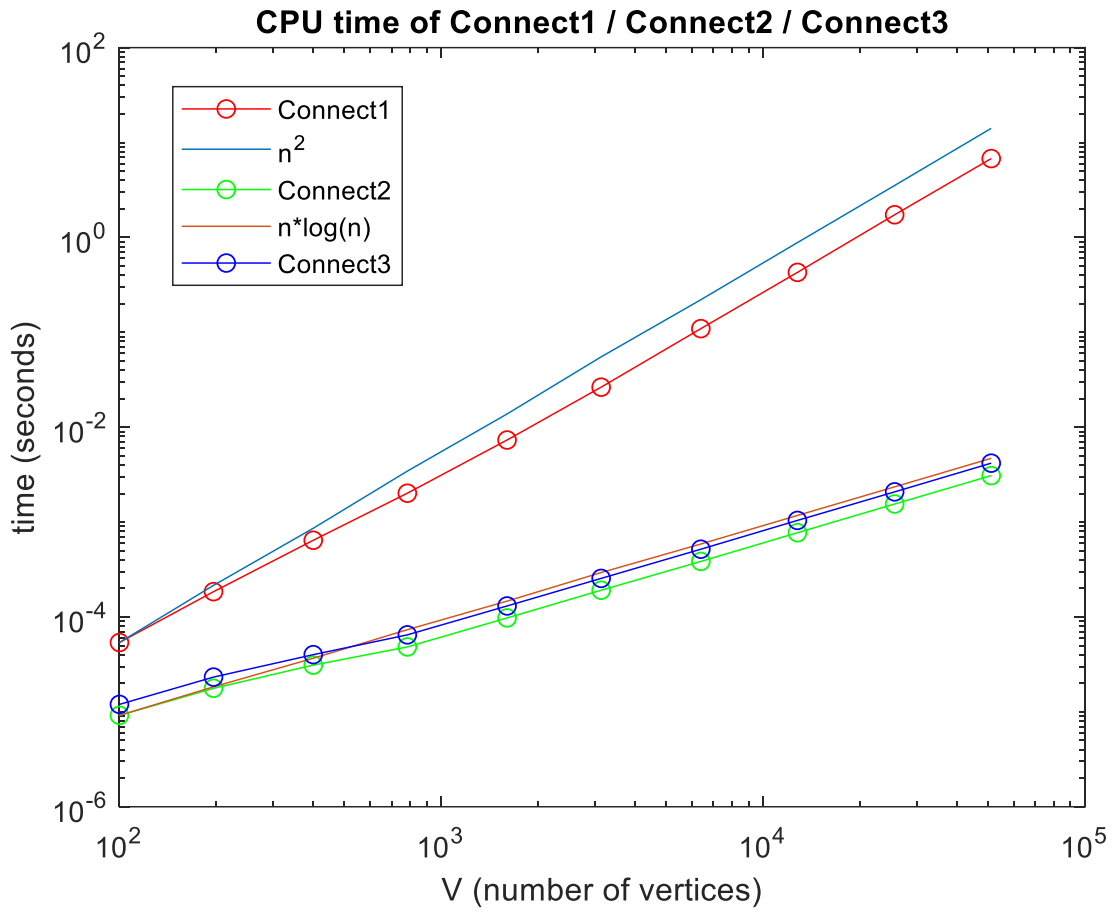

Figure 1 CPU time of Connect1 / Connect2 / Connect3

The result consists with **Table 2** again. The time complexity of Connect1, Connect2, and Connect3 are $O(V^2)$, O(V * log V), O(V * logV) respectively.

The last thing I want to discuss is why Connect3 is a little bit faster then Connect2. Although we use the CollapsingFind in Connect3, we have already come to the conclusion that CollapsingFind roughly doubles the time for an individual find. The advantage of it is to reduce the worst-case time over a sequence of finds. Thus, using CollapsingFind may not necessarily improve the CPU time. It depends on the structure of sets and the elements we want to find.