EE 3980 Algorithms

Homework 4. Network Connectivity Problem

105061110 周柏宇

2020/4/5

1. Introduction

In this homework, we implemented a generic connectivity algorithm to solve the network connectivity problem. In the connectivity algorithm, there are two disjoint set operations: *SetFind* and *SetUnion*. We not only implemented the vanilla *SetFind* and *SetUnion*, but also implemented their alternatives *CollapsingFind* and *WeightedUnion*, respectively. To measure the performance, we execute some combinations of the *SetFind* and *SetUnion* to solve graphs with different number of vertices and edges. In the end, we record the result and observe the growth of CPU time to verify our derivation of their time complexity.

2. Analysis & Implementation

2.1 Generic Connectivity 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)
{
     // one element for each set
    for each v_i in V do S_i := {v_i};
```

```
 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 != S_j then { // unite two sets
      // number of disjoint sets decrease by 1
      NS := NS - 1;
      SetUnion(S_i, S_j);
    }
  }
  for each v_i in V do { // record root to R table
    R[i] := SetFind(v_i);
  }
}
```

In the generic connectivity algorithm, we are given a graph and output a table R that records which vertices are connected. The algorithm has two fundamental operations – *SetFind* and *SetUnion*. We first assume all vertices are disjoint and set them as roots in array S. Afterward, we enumerate all edges in the graph. For the vertices connected by the edge, we first find their roots (done by *SetFind*) and see if they are the same, which suppose to be since they are connected by an edge. If not, we then unite the two sets as one (done by *SetUnion*). Finally, to clean up the representation, we enumerate all vertices and record the root of each vertex in array R.

As we can see, the generic connectivity algorithm is mainly composed of three loops. The worst-case time complexity contributed by each of them are:

First loop = $\mathcal{O}(|V|)$

Second loop = $O\big(|E| * (2 * F + U)\big) = O\big(|E| * (F + U)\big)$

Third loop = $O(|\mathrm{V}| * \mathrm{F})$

where $|\mathrm{V}|$ is the number of vertices, $|E|$ is the number of edges and F, U is the time complexity of *SetFind* and *SetUnion*, respectively. Assuming $|E| \geq |\mathrm{V}|$, we can see that the second loop makes up most of the time complexity, i.e. $O\big(|E| * (F + U)\big)$. Next, we will experiment some implementations of *SetFind* and *SetUnion* and their combinations to see how the connectivity algorithm performs.

In the generic connectivity algorithm, we need to store the roots for every element in array R. Therefore, the space complexity is $O(|\mathrm{V}|)$.

2.2 SetUnion

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

In *SetUnion*, we combine two disjoint sets as one set by setting one element's parent as the other element. In this way, *SetFind* will return the same root, and the two elements will be in the same set. The time complexity is $O(1)$ and so is the space complexity.

Best-case time complexity: $O(1)$

Average time complexity: $\mathcal{O}(1)$

Worst-case time complexity: $\mathcal{O}(1)$

Space complexity: $\mathcal{O}(1)$

2.3 SetFind

```
// Find the set that element i is in.
// Input: element i
// Output: root element of the set
Algorithm SetFind(i)
{
    // if its parent is a nonnegative value,
    // then go to its parent
    while (p[i]>=0) do i := p[i];
    return i;
 }
```

In *SetFind*, we will return element *i*'s root element. As for how many elements we have to go through to find the root, it can be upper-bounded by the number of elements in the disjoint set, denoted as m. However, m depends on which set the element is in, to have a specific number, we can only upper-bounded m by the number of elements of the largest set possible, which is the case that $m = |\mathrm{V}|$.

SetFind does not store the array p; it only traverses it. Therefore, the space complexity is $\mathcal{O}(1)$.

Best-case time complexity: $\mathcal{O}(1)$

Worst-case time complexity: $\mathcal{O}(m)$ or $\mathcal{O}(|\mathrm{V}|)$

Space complexity: $\mathcal{O}(1)$

2.4 WeightedUnion

```
// Form union of two sets with roots, i and j,
// using the weighting rule.
// Input: roots of two sets i and j
// Output: none
Algorithm WeightedUnion(i, j)
{   // temp = (total number of elements of i and j) x -1
    temp := p[i] + p[j];
    if (p[i] > p[j]) then { // i has fewer elements
           p[i] := j; // connect root i to j
           p[j] := temp; // update number of elements
     }
     else { // j has fewer elements
           p[j] := i; // connect root j to i
           p[i] := temp; // update number of elements
     }
 }
```

WeightedUnion is an alternative to *SetUnion*, which aims to improve the time complexity by lowering the height of the tree, i.e. the number of elements to go through before reaching the root of the set. In *WeightedUnion*, we first compare the number of elements of two roots, and connect the root with fewer elements to the root with more elements. Finally, we update the number of elements.

Although the time complexity of *WeightedUnion* is the same as *SetUnion*, it can reduce the time complexity of *SetFind*. Therefore, the overall time complexity of connectivity algorithm improves. We claim that "using *WeightedUnion* to perform set union can result in a tree with *m* nodes to have height no greater than $\lfloor \lg m \rfloor + 1$" and we will prove it by mathematical induction.

<u>Proof</u>

For $m = 1$, height = 1 and $\lfloor \lg 1 \rfloor + 1 = 1$. The claim holds.

Assuming for the first $m - 1$ operations, our claim still holds.

Considering the last step *WeightedUnion*(k, j), without the loss of generality, we assume that set j has a elements and set k has $m - a$ elements where $1 \leq a \leq \frac{m}{2}$, i.e. sets j has fewer elements. We can see that for a tree T with m nodes, the height of the tree must be the same as that of k, i.e.

$$\text{the height of T} \leq \lfloor \lg (m - a) \rfloor + 1 \leq \lfloor \lg m \rfloor + 1$$

or one more than that of j, i.e.

$$\text{the height of T} \leq \lfloor \lg \mathrm{a} \rfloor + 2 \leq \lfloor \lg \frac{\mathrm{m}}{2} \rfloor + 2 \leq \lfloor \lg m \rfloor + 1$$

For either case, our claim holds. Therefore, we can bound the height of the tree with m nodes with $\mathcal{O}(\lg m)$, which becomes the worst-case time complexity of *SetFind*.

Best-case time complexity: $\mathcal{O}(1)$

Average time complexity: $\mathcal{O}(1)$

Worst-case time complexity: $\mathcal{O}(1)$

Space complexity: $\mathcal{O}(1)$

2.5 CollapsingFind

```
1. // Find the root of i,
2. // and collapsing the elements on the path.
3. // Input: an element i
4. // Output: root of the set containing i
5. Algorithm CollapsingFind(i)
6. {
7.     r := i; // initialize r to i
8.     while (p[r] >= 0) do r := p[r]; // find the root
9.     // collapse the elements on the path
10.    while (i != r) do {
11.        s := p[i]; p[i] := r; i := s;
12.    }
13.    return r;
14. }
```

CollapsingFind is the alternative of *SetFind*, as we can see the difference is the extra lines from 10 to 12. What the extra lines do is to make all the nodes on the path from the element *i* to root to be the direct children of the root, which is referred to as “Collapsing”. With collapsing, we effectively reduce the height of the tree to at most 2 (the root and the leave). The worst-case time complexity is $\mathcal{O}(SetFind)$, but on average case, we expect some benefit from shorter trees.

Best-case time complexity: $\mathcal{O}(1)$

Worst-case time complexity: $\mathcal{O}(SetFind)$

Space complexity: $\mathcal{O}(1)$

3. Result and Observation

To measure the performance, we execute the generic connectivity algorithm with some combinations of *SetFind*, *CollapsingFind*, *SetUnion* and *WeightedUnion* to solve graphs with different number of vertices and edges. Also, each data point in the table is the average result of 100 executions.

3.1 Connect1

The first combination of connectivity algorithm is using the vanilla *SetFind* and *SetUnion*. According to our analysis in previous section, the time complexity of the connectivity problem is $\mathcal{O}\big(|E| * (F + U)\big)$. In this case, $F = \mathcal{O}(|\mathrm{V}|)$ and $U = \mathcal{O}(1)$, which makes $\mathcal{O}\big(|E| * (F + U)\big) = \mathcal{O}(|E| * |\mathrm{V}|)$.

3.2 Connect2

The second combination is to replace *SetUnion* with *WeightedUnion*. What changes is the time complexity of *SetFind*. It changes from $\mathcal{O}(|\mathrm{V}|)$ to $\mathcal{O}(\lg |\mathrm{V}|)$. Hence, the total complexity becomes $\mathcal{O}(|E| * \lg |\mathrm{V}|)$.

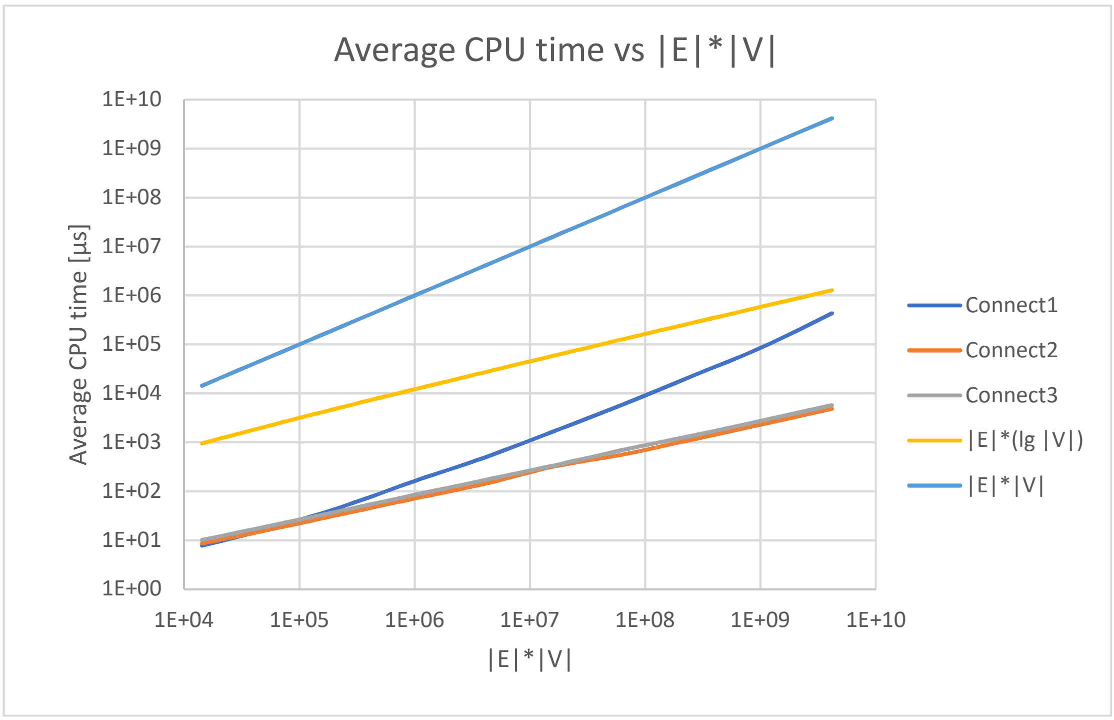
3.3 Connect3

The third combination is to not only replace *SetUnion* with *WeightedUnion* but also replace *SetFind* in line 11 with *CollapsingFind*. Since we still use *WeightedUnion* to unite the sets, the total time complexity is still $\mathcal{O}(|E| * \lg |\mathrm{V}|)$.

filename	**|V|**	**|E|**	**NS**	**Connect1**	**Connect2**	**Connect3**

g1.dat	100	143	1	7.71046	8.64029	10.1399
g2.dat	196	300	3	17.9815	16.9706	20.1488
g3.dat	400	633	1	51.2695	35.5005	42.0284
g4.dat	784	1239	3	158.36	69.8614	83.2987
g5.dat	1600	2538	5	490.15	141.59	169.84
g6.dat	3136	4958	6	1627.57	313.501	331.149
g7.dat	6400	10201	9	6044.03	567.629	705.941
g8.dat	12769	20265	10	22919.9	1143.1	1368.15
g9.dat	25600	40795	54	88628.2	2340.06	2786.22
g10.dat	51076	81510	92	430442	4834.23	5724.59

Table 1. Average CPU time [μ s] vs different graphs



In the figure above, we plot the average CPU time solving different graphs with respect to the number of edges times the number of vertices of the given graph. According to our analysis, we expect the first combination using vanilla *SetFind* and *SetUnion* to have the worst time complexity $O(|E| * |\mathrm{V}|)$. As we can see, the slope of *Connect1* is very similar to the $|E| * |\mathrm{V}|$ line. Furthermore, the slope of *Connect2* and *Connect3* resemble the slope of $|E| * \lg |\mathrm{V}|$, which validates our analysis.

As for why the advantage of using *CollapsingFind* in *Connect3* is not significant, my explanation is that performing "Collpasing" is a costly job. If we count the steps in the while loop in the collapsing part, it is actually three times the number of steps as *SetFind*. In our experiment, the execution time is shortest across all inputs using vanilla *SetFind* with *WeightedUnion*.