**EE 3980 Algorithms**

Homework 4. Network Connectivity Problem

105061110 周柏宇

2020/4/5

1. **Introduction**

*In this homework, we implemented heap sort and compared its performance with selection sort, insertion sort, bubble sort and shaker sort by sorting lists containing a different number of randomly-ordered English words. We not only measured their average performance but also rearranged the input data according to our analysis to test their best-case and worst-case execution time. In the end, we plot out the result and observe the growth of CPU time to verify our derivation of their time complexity.*

1. **Analysis & Implementation**
   1. **Generic Connectivity Algorithm**
2. // Given G(V,E) find connected vertex sets,
3. // generic version.
4. // Input: G(V,E)
5. // Output: Disjoint connected sets R[1:n]
6. Algorithm Connectivity(G, R)
7. {
8. // one element for each set
9. **for** each v\_i in V **do** S\_i := {v\_i};
10. NS := |V|; // number of disjoint sets
11. **for** each e = (v\_i, v\_j) **do** { // connected vertices
12. S\_i := SetFind(v\_i); S\_j := SetFind(v\_j);
13. **if** S\_i != S\_j then { // unite two sets
14. // number of disjoint sets decrease by 1
15. NS := NS - 1;
16. SetUnion(S\_i, S\_j);
17. }
18. }
19. **for** each v\_i in V **do** { // record root to R table
20. R[i] := SetFind(v\_i);
21. }
22. }

In the generic connectivity algorithm, we are given a graph and output a table R that records which vertices are connected with edges. 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. 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 =

Second loop =

Third loop =

where is the number of vertices, is the number of edges and is the time complexity of *SetFind* and *SetUnion*, respectively. Assuming , we can see that the second loop makes up most of the time complexity, i.e. . Next, we will experiment some implementations of *SetFind* and *SetUnion* and their combination to see how the connectivity algorithm performance.

In the generic connectivity algorithm, we need to store the roots for every element in array R. Therefore, the space complexity is .

* 1. **SetUnion**

1. // Form union of two sets with roots, i and j.
2. // Input: roots, i and j
3. // Output: none
4. Algorithm SetUnion(i, j)
5. {   // set i's parent as j
6. p[i] := j;
7. }

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 and so is the space complexity.

Best-case time complexity:

Average time complexity:

Worst-case time complexity:

Space complexity:

* 1. **SetFind**

1. // Find the set that element i is in.
2. // Input: element i
3. // Output: root element of the set
4. Algorithm SetFind(i)
5. {
6. // if its parent is a nonnegative value,
7. // then go to its parent
8. **while** (p[i]>=0) **do** i := p[i];
9. **return** i;
10. }

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 largest set possible, which is the case that all vertices are connected.

*SetFind* does not store the array p; it only traverses it. Therefore, the space complexity is .

Best-case time complexity:

Worst-case time complexity:

Space complexity:

* 1. **WeightedUnion**

1. // Form union of two sets with roots, i and j,
2. // using the weighting rule.
3. // Input: roots of two sets i and j
4. // Output: none
5. Algorithm WeightedUnion(i, j)
6. {   // temp = (total number of elements of i and j) x -1
7. temp := p[i] + p[j];
8. **if** (p[i] > p[j]) then { // i has fewer elements
9. p[i] := j; // connect root i to j
10. p[j] := temp; // update number of elements
11. }
12. **else** { // j has fewer elements
13. p[j] := i; // connect root j to i
14. p[i] := temp; // update number of elements
15. }
16. }

*WeightedUnion* is an alternative to *SetUnion*, which aims to improve the average time complexity by lowering the height of the tree i.e. 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 that a tree with *m* nodes to have height no greater than ⌊lg *m*⌋+1” and we will prove it by mathematical induction.

Proof

For m = 1, height = 1 and ⌊lg 1⌋+1=1. The claim holds.

Assuming after 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 , 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.

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

Therefore,…

Best-case time complexity:

Average time complexity:

Worst-case time complexity:

Space complexity:

* 1. **CollapsingFind**

1. **Result and Observation**

To measure the best-case, average and worst-case performance, we execute each sorting algorithm with their corresponding rearranged input according to our previous analysis. Also, each data point in the table is the average result of 500 executions.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | Average | | | | |
| *Selection sort* | *Insertion sort* | *Bubble sort* | *Shaker sort* | *Heap sort* |
| 10 | 2.28357 | 1.21212 | 2.92778 | 2.85769 | 2.83003 |
| 20 | 6.68621 | 4.98009 | 10.54 | 10.8159 | 8.19206 |
| 40 | 22.172 | 14.8222 | 41.3857 | 42.2621 | 17.6578 |
| 80 | 82.8619 | 25.6281 | 169.11 | 166.694 | 39.4602 |
| 160 | 305.348 | 78.0401 | 684.662 | 673.282 | 91.7859 |
| 320 | 1154.24 | 357.7 | 1798.58 | 2205.04 | 213.186 |
| 640 | 2660.69 | 1275.13 | 5642.05 | 5257.94 | 481.65 |
| 1280 | 8227.41 | 5581.26 | 20420.3 | 19429.4 | 1072.71 |
| 2560 | 34145.1 | 22112.4 | 87123.1 | 78771.4 | 1709.87 |

Table 2. Average CPU time [μs] vs input data size *N*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | Worst Case | | | | Lower Bound of Worst Case |
| *Selection sort* | *Insertion sort* | *Bubble sort* | *Shaker sort* | *Heap sort* |
| 10 | 2.32601 | 0.87595 | 1.28031 | 1.39809 | 2.85387 |
| 20 | 7.26175 | 2.97165 | 4.68588 | 4.76599 | 6.84404 |
| 40 | 11.2519 | 12.1999 | 18.024 | 18.5838 | 16.4618 |
| 80 | 42.1681 | 39.0759 | 70.5919 | 74.8219 | 39.1397 |
| 160 | 159.4 | 150.846 | 279.596 | 281.164 | 88.2602 |
| 320 | 636.564 | 591.236 | 1131.56 | 1123.68 | 200.886 |
| 640 | 2579.73 | 2338.45 | 4399.62 | 4487.31 | 452.89 |
| 1280 | 10645.9 | 9373.83 | 17959.5 | 17540 | 990.148 |
| 2560 | 49081.6 | 38519.5 | 72413.9 | 70900.9 | 1615.98 |

Table 3. Worst-case CPU time [μs] vs input data size *N*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | Best Case | | | | Upper Bound of Best Case |
| *Selection sort* | *Insertion sort* | *Bubble sort* | *Shaker sort* | *Heap sort* |
| 10 | 2.03371 | 0.874043 | 0.748158 | 0.92411 | 0.976086 |
| 20 | 2.52008 | 0.674248 | 3.93009 | 2.93636 | 2.68412 |
| 40 | 8.38614 | 1.44815 | 11.8718 | 12.6138 | 8.3518 |
| 80 | 32.9399 | 3.17574 | 50.5142 | 53.412 | 36.1724 |
| 160 | 113.562 | 5.99384 | 221.974 | 248.068 | 85.0878 |
| 320 | 435.574 | 11.7278 | 988.59 | 1768.83 | 187.258 |
| 640 | 1720.92 | 24.6601 | 3762.97 | 3764.88 | 419.666 |
| 1280 | 7004.51 | 49.8462 | 16679.9 | 16191 | 950.098 |
| 2560 | 28182.4 | 108.496 | 72331.7 | 69787 | 947.914 |

Table 4. Best-case CPU time [μs] vs input data size *N*

For the average case performance, heap sort exhibits excellent performance due to its at most time complexity. When it comes to worst-case performance, heap sort’s time complexity still dominates other quadratic sorting algorithms because all of them have . However, in the best-case scenario, according to our analysis, insertion sort has time complexity and it is the only case that beats heap sort. In our best-case analysis for heap sort, we only bound the time complexity for which seems reasonable. However, whether the best-case time complexity can be bounded tighter by is hard to answer by solely observing the plot because and have a quite similar trend.

For selection sort, although it has the same time complexity for best-case and worst-case. The actual execution time is quite different, which is caused by whether it executes the one operation in the for block.

For bubble sort and shaker sort, they behave very similarly across best-case, average and worst-case, which validates our argument that shaker sort is bubble sort executing in different directions. With proper rearrangement for the input, we expect shaker sort to have same the performance as bubble sort on average or for extreme cases.