EE3980 Algorithms

Homework 3. Network Connectivity problem

By 105061212 王家駿

2019/03/23

1. Introduction

In this homework, we construct an undirected graph with input vertices and edges, and $\overset{\text{unite}}{\underline{\text{union}}}$ the vertices(i,j) in a set if there exists a path from i to j. In the end, we calculate the number of the distinct sets in the graph, and record the CPU time used for arrange the vertices in the sets. The above steps would be executed with different methods, including **Weighted set union** and **Collapsing set find**, so then we could find out the difference of time consumed between these algorithms.

2. Implementation

In the program, we first read all inputs and store them in some arrays. Then we use the three methods, **Connect1**, **Connect2**, **Connect3**, which would be explained later, to implement the network connectivity, and also record the CPU time consumed. Finally, we show the number of sets and the CPU time on the screen.

2.1. Network connectivity

In this problem, we have V vertices, numbered from 1 to V, and E edges input. Since our final goal is to $\overset{\text{unite}}{\underline{\text{union}}}$ all the $\overset{\text{connected}}{\underline{\text{connective}}}$ vertices together, we could first assume that there are V sets initially with only one element (vertex) in each set. Then for every edge, we could connect the vertices of each edge. That is, union the set of the vertices together. After all the edges are checked, we could easily find out the sets remained, which indicates that the vertices in the same set are connected together.

In order to implement our method, we use an array S with size V. In the array, S[i] means i and S[i] are in the same set. Besides, if S[i] is equal to -1, that means i is the root of the set.

The array S could also be represented like a forest, which is made up by several trees. The i with S[i] = -1 acts like the tree root, and j pointing to the root i acts like the child of i, and so does other nodes. So, we might use the tree representation to explain our methods later to make them clearer.

2.2. SetFind

```
Algorithm SetFind(i)
{
    while(S[i] >= 0) do i:=S[i];     // keep finding the root
    return i;
}
```

If we want to find the set which a given element i is belong to, we could use the SetFind function. In this function, we starting finding from the element i, and keep iterates to the S[i] until S[i] is no less than zero. Since i and S[i] are belong to the same set for all i, we can make sure that all the iterations go in the same set, and at the moment S[i] is no less than zero, we could know that we find the root.

In this function, the iteration at line 3 would execute at most h times, where h represents the tree height of S. At its worst case, the tree is totally skewed and the iteration execute for V times. At its best case, the given element i is the root of a set, so the loop only executes one time.

For the space capacity, there is no need of extra space for this algorithm, so the space complexity is $O(1)$.

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

Worst-case time complexity: $O(V)$

Average-case time complexity: $O(h)$

Space complexity: $O(1)$

2.3. SetUnion

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

By using the set union function, we could link the two different sets together. In this function, we just assign j to S[i], so the i and j would be in the same set. Thus, all the elements which are in the same set with i would be in the same set with j, vice versa. We could easily union the two sets by simply an assignment.

No matter what the input vertices i and j are, all we should do is one assignment, so the time complexity of this function in any cases is $O(1)$, so is the space complexity because of no extra space needed.

Time complexity: $O(1)$

Space complexity: $O(1)$

2.4. Connect1

```
Algorithm Connect1(G, R)
{
    for each vi in V do S:={vi};        // one element for each set
    NS:=number of vi;                   // number of disjoint sets

    for each e = (vi, vj) do{           // for each edge
        Si:=SetFind(vi);
        Sj:=SetFind(vj);

        if(Si != Sj) then{              // if two sets
            NS:=NS-1;
            SetUnion(Si, Sj);           // union them
        }
    }
    for each vi in V do{                // record root to R table
        R[i]:=SetFind(vi);
    }
}
```

This is the function that we implement the network connectivity. At first, we have to initialize the array S to make the V sets containing only one element, and this could be implemented by simply setting -1 to all the value in S. That is, set all the vertices be the root of a set.

Then for each edge, we would decide whether we have to union the two sets of the vertices or not. With the two vertices, we use the **SetFind** function to find out the sets which the vertices are belong to. If they are in the same set, we don't have to do anything and go to the next iteration; otherwise, we link the two sets together by the **SetUnion** function mentioned above. After the iteration, the sets remain are the results of network connectivity.

In the end, we store the sets which each vertex belongs to into an array R by using **SetFind** function. And we called it the **set table**.

To estimate the time complexity, first we check the loops at line 3, 6, and 15. The iteration goes V times at line 3 and 15, E times at line 6. In the loop of line 3 and 15, the time complexity is $O(V)$. While in the loop of line 6, each iteration would do at most two **SetFind** and one **SetUnion**. According to the time complexity of these functions we estimate before, the time complexity is $O(h)$ for the average case. Thus, the overall complexity must be $O(h * E + 2V)$. For the ten testing data, the number of vertices and edges do not differ too much, and we could roughly estimate that the time complexity is $O(h * E)$.

For the space complexity, we only need extra spaces in the loop of line 6, so the space complexity for this algorithm is $O(E)$.

Time complexity: $O(h * E)$

Space complexity: $O(E)$

Furthermore, we revise the **SetFind** and **SetUnion** algorithms to have a better performance on time. In **Connect2**, we replace the **SetUnion** by **WeightedUnion**; while in **Connect3**, we not only use the **WeightedUnion** but replace the **SetUnion** by **CollapsingFind**. Then we measure the time used by the revised algorithms.

2.5. WeightedUnion

```
Algorithm WeightedUnion(i, j)
{
    temp:=S[i]+S[j];                    // two sets element sum

    if(S[i] > S[j]) then{               // if i has fewer elements
        S[i]:=j; S[j]:=temp;            // link i to j
    }
    else{                               // if j has fewer elements
        S[j]:=i; S[i]:=temp;            // link j to i
    }
}
```

This function would replace the **SetUnion** function. Unlike the **SetUnion**, in which i always link to j, this function would either link i to j or j to i, depending on the number of elements in the set. If set i has the fewer element, link set i to set j (make the root of set i point to a node of set j); and if j has the fewer element, link set j to set i.

The time complexity would still be $O(1)$ since the algorithm takes only three assignments, but it might work faster than the previous one. In order to get a better performance on time consumed, we should prevent the tree from being skewed, which would spend more time at iterating to find root in **SetFind**. Thus, by using the **WeightedUnion**, we could choose the smaller set to be linked, and prevent the tree from skewing one side.

Since the maximum tree height is $\log_2 V + 1$ by using weighted set union, which is derived from the handout of the class, we could find out that the iteration only goes at most $\log_2 V + 1$ steps in the function **SetFind**, if this algorithm is applied. Thus, the total time complexity of **Connect1** could become $O(E * \log_2 V) = O(V * \log_2 V)$, and it is implemented in **Connect2.**

2.6. CollapsingFind

```
Algorithm CollapsingFind(i)
{
    r:=i;

    while(S[r] > 0) do r:=S[r];         // find the root
    while(i != r) do {                  // collapse the elements on the path
        temp:=p[i]; S[i]:=r; i:=temp;
    }

    return r;
}
```

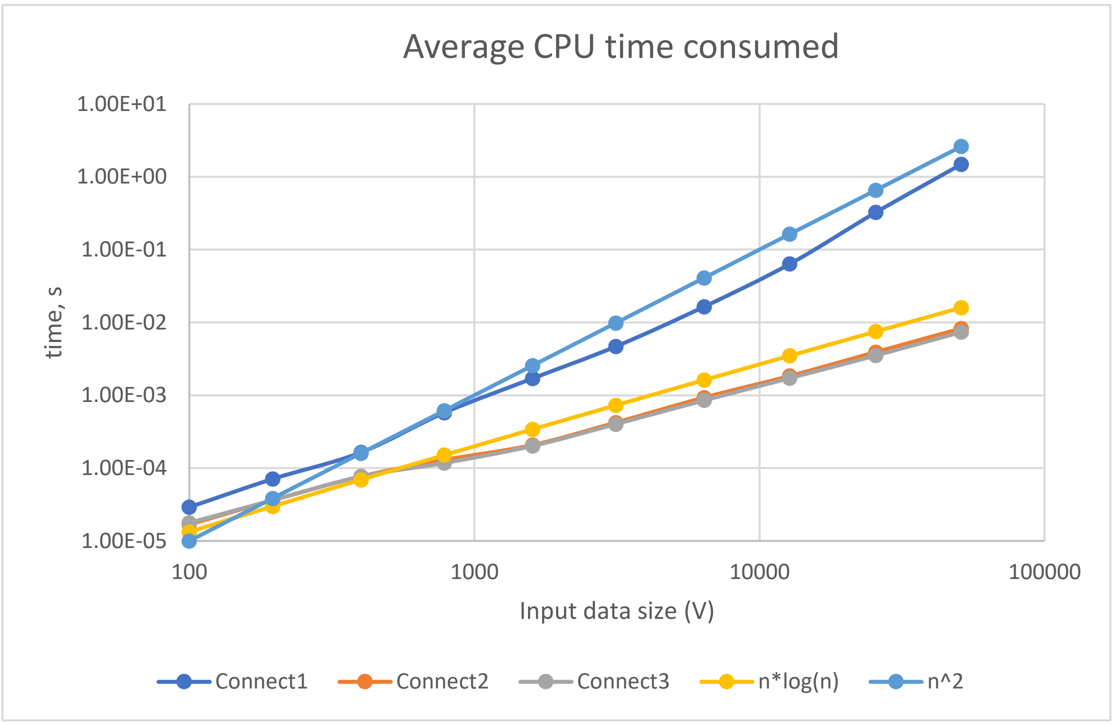
This revised algorithm makes more improvement on time consumed by collapsing the elements while finding. At first, the algorithm finds the root of the set as the same way in **SetFind**. And for each element i on the finding path, we replace S[i] by the root value (collapsing). Although the time consumed for each execution, we could find the root immediately later if the given node has been collapsed before since it points to the root. Thus, the time consumed would decrease if we would do set finding many times.

The time complexity after applying this function is still $O(V * \log_2 V)$, since it doesn't decrease any iteration step. However, the program may run faster because the next time we find for the root of an element, the time complexity of finding becomes $O(1)$. Thus, the time complexity of some iterations for each edge might become $O(1)$. And the input data size seems to have less influence on the complexity, what matters is how the vertices connect and the input edges arrangement order.

3. Executing results

For repeating 100 times, run the testing data from g1.dat to g10.dat with different input data size, and record the average CPU time used.

Data size (V/E)	**Connect1**	**Connect2**	**Connect3**	**Number of sets**
100/147	29.03 μs	16.79 μs	17.68 μs	1
196/287	70.84 μs	36.13 μs	36.21 μs	1
400/608	163.7 μs	76.76 μs	76.59 μs	1
784/1213	575.6 μs	129.3 μs	117.3 μs	3
1600/2489	1.701ms	207.4 μs	201.4 μs	5
3136/4883	4.678ms	422.4 μs	401.9 μs	9
6400/10130	16.45ms	930.3 μs	851.5 μs	14
12769/20251	63.38ms	1.838ms	1.719ms	18
25600/40727	324.2ms	3.916ms	3.517ms	45
51076/81499	1.486s	8.270ms	7.381ms	80



4. Result analysis and conclusion

From the graph, we could observe that **Connect2** and **Connect3** have a trend of $n * \log(n)$, which is same as our estimation. Yet, it seems that **Connect1** doesn't have a trend of neither $n * \log(n)$ or n^2, the line is like staying between them. It makes sense since the worst-case time complexity is $O(n^2)$ with the tree height h=V. However, it's still hard to estimate what the average time complexity because it's also difficult to find out how the tree skews.

How the tree skews might depend on the input edges and its arrangement. We can find it at **SetUnion** , where we assign j to S[i], and it means that we put the tree of set i under the node of set j. If the edges of input data have been arranged well, that is, vi < vj for the edge(vi,vj), then we always let the set with smaller index be the subtree of the set with larger index. This would cause the tree skew and increase the tree height, and lead to a time complexity closed to $O(n^2)$. If the arrangement of (vi,vj) is placed randomly, the tree structure would be like a complete tree, so the tree height would decrease and the time complexity would be closed to $O(n * \log n)$.

In our previous estimation, we predict that the time consumed by **Connect3** would be less than **Connect2**, but it doesn't hold for the input vertices fewer than 196. It might occur when the number of edges input is small, because the **CollapsingFind** spends more time at the collapsing, in order to make the next fetching of elements be quicker. And if the element isn't fetched again, that would be just a waste of time. Thus, if the vertices aren't fetched for enough times, the **Connect3** would spend more time than **Connect2**.

Score: 81

o. See return.

[Writing] need more practice in English writing.

[Approach] can show more pseudo codes (for Connect2, Connect3 and main).

[Results] can draw conclusions.

[Coding] need proper indentation.

[Coding] pay attention to spelling.

hw03.c

```c
/* EE3980 HW03 Heap Sort
 * 105061212, Chia-Chun Wang
 * 2019/03/23
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

void readGraph(void);                   // read and store all input
double GetTime(void);                   // get current CPU time
void Connect1(void);                    // network connectivity
void Connect2(void);                    // network connectivity with
                        // weighted set union
void Connect3(void);                    // network connectivity with
                        // collapsing set find
int SetFind(int in);                    // find the set of the element
void SetUnion(int i, int j);                // union two sets
void WeightedUnion(int i, int j);           // union two sets by weight
int CollapsingFind(int in);             // find the set of the element
                        // and collaspse the tree
                               collaspse
void freeMemory(void);                  // free dynamic memories

int V;                      // number of vertices
int E;                      // number of edges
int* S;                     // array of sets
int** edge;                 // 2D array of edges
int* R;                     // the set table
int NS;                     // number of sets
int Nrepeat = 100;              // number of repetitions

int main(void)
{
    int i;                  // loop index
    double t0, t1, t2, t3;          // current CPU time
    double t1_avg, t2_avg, t3_avg;          // average time per execution
    int Ns1, Ns2, Ns3;              // number of sets

    readGraph();                // read and store all input

    t0 = GetTime();             // get current CPU time

    for(i = 1; i <= Nrepeat; i++){      // repeat Nrepeat times
        Connect1();             // do connect1
    }

    t1 = GetTime();             // get current CPU time
    Ns1 = NS;                   // record the number of sets

    for(i = 1; i <= Nrepeat; i++){      // repeat Nrepeat times
    Connect2();             // do connect2
    Need indentation.
    }

    t2 = GetTime();             // get current CPU time
    Ns2 = NS;                   // record the number of sets

    for(i = 1; i <= Nrepeat; i++){      // repeat Nrepeat times
        Connect3();             // do connect3
    }

    t3 = GetTime();             // get current CPU time
    Ns3 = NS;                   // record the number of sets

    // calculate the average CPU time per execution
    t1_avg = (t1-t0) / Nrepeat;
    t2_avg = (t2-t1) / Nrepeat;
    t3_avg = (t3-t2) / Nrepeat;

    // print out the results
    printf("Connect1:\n  Time: %e s\n  Number of Set: %d\n", t1_avg, Ns1);
    printf("Connect2:\n  Time: %e s\n  Number of Set: %d\n", t2_avg, Ns2);
    printf("Connect3:\n  Time: %e s\n  Number of Set: %d\n", t3_avg, Ns3);

    freeMemory();               // free dynamic memories

    return 0;
}

void readGraph(void)                // read and store all input
{
    int i;                  // loop index

    scanf("%d", &V);                // number of vertices
    scanf("%d", &E);                // number of edges

    // allocate dynamic memories
    S = (int*)malloc(sizeof(int) * (V+1));
    R = (int*)malloc(sizeof(int) * (V+1));

    edge = (int**)malloc(sizeof(int*) * (E+1));
    for(i = 1; i <= E; i++){
    for (i = 1; i <= E; i++) {
    edge[i] = (int*)malloc(sizeof(int) * 2);
    Need indentation.
    }

    // store all edges with a 2D array
    for(i = 1; i <= E; i++){
    for (i = 1; i <= E; i++) {
    scanf("%d %d", &edge[i][0], &edge[i][1]);
    }
}

double GetTime(void)                // get local time in seconds
{
    struct timeval tv;              // time interval structure

    gettimeofday(&tv, NULL);            // write local time into tv

    return tv.tv_sec + tv.tv_usec * 0.000001;   // return time with microsecond
}

// network connectivity
void Connect1(void)
{
    int i;                  // loop index
    int Si, Sj;                 // set root

    for(i = 1; i <= V; i++){            // initailze the set
                                           initailze
    S[i] = -1;              // only one element in each set
    Need indentation.
    }

    NS = V;                 // number of sets = vertices

    for(i = 1; i <= E; i++){            // for every edges
    // find the set root of vertices of each edge
        Si = SetFind(edge[i][0]);
    Sj = SetFind(edge[i][1]);

    if(Si != Sj){               // if not in the same set
        NS--;               // number of sets - 1
        SetUnion(Si, Sj);           // union two sets
    }
    }

    for(i = 1; i <= V; i++){            // for each vertice
                                                    vertice
        R[i] = SetFind(i);          // record the set belong
    }
}

// network connectivity with weighted set union
void Connect2(void)
{
    int i;                  // loop index
    int Si, Sj;                 // set root

    for(i = 1; i <= V; i++){            // initailze the set
        S[i] = -1;              // only one element in each set
    }

    NS = V;                 // number of sets = vertices

    for(i = 1; i <= E; i++){            // for every edges
        // find the set root of vertices of each edge
        Si = SetFind(edge[i][0]);
        Sj = SetFind(edge[i][1]);

        if(Si != Sj){               // if not in the same set
            NS--;               // number of sets - 1
            WeightedUnion(Si, Sj);      // union two sets
        }
    }

    for(i = 1; i <= V; i++){            // for each vertice
        R[i] = SetFind(i);          // record the set belong
    }
}

// network connectivity with weighted set union and collapsing set find
void Connect3(void)
{
    int i;                  // loop index
    int Si, Sj;                 // set root

    for(i = 1; i <= V; i++){            // initailze the set
        S[i] = -1;              // only one element in each set
    }

    NS = V;                 // number of sets = vertices

    for(i = 1; i <= E; i++){            // for every edges
        // find the set root of vertices of each edge
        Si = CollapsingFind(edge[i][0]);
        Sj = CollapsingFind(edge[i][1]);

        if(Si != Sj){               // if not in the same set
            NS--;               // number of sets - 1
            WeightedUnion(Si, Sj);      // union two sets
        }
    }

    for(i = 1; i <= V; i++){            // for each vertice
        R[i] = SetFind(i);          // record the set belong
    }
}

int SetFind(int in)             // find the set of the element
{
    int i;                  // input element

    i = in;

    while(S[i] >= 0){               // if the element is not root
        i = S[i];               // keep finding
    }

    return i;
}

void SetUnion(int i, int j)         // union two sets
{
    S[i] = j;                   // assign one set to another
}

void WeightedUnion(int i, int j)        // union two sets by weight
{
    int temp;                   // temporary integer

    temp = S[i] + S[j];             // sum the number of elements

    if(S[i] > S[j]){                // if i has fewer elements
                            // assign set i to j
    S[i] = j;
    S[j] = temp;
    }
    else{                   // if i has more elements
                        // assign set j to i
    S[j] = i;
    S[i] = temp;
    }
}

// find the set of an element and collapse the tree
int CollapsingFind(int in)
{
    int i;                  // input element
    int r;                  // root to be found
    int temp;                   // temporary integer

    i = in;                 // find start from input
    r = in;

    while(S[r] > 0){                // if not the root
    r = S[r];               // keep finding
    }

    while(i != r){              // if not the root
        // make the element point to root
    temp = S[i];
    S[i] = r;
    i = temp;
    }

    return r;
}

void freeMemory(void)               // release dynamic memories
{
    int i;                  // loop index

    // free all dynamic memories allocated before
    free(S);
    free(R);

    for(i = 0; i <= E; i++){
        free(edge[i]);
    }
    free(edge);
}
```