EE 3980 Algorithms

Homework 7. Grouping Friends

105061110 周柏宇

2020/4/26

1. Introduction

In this homework, we try to find a group of people in which they can communicate directly or indirectly through other members in the same group. We first argue that this problem is essentially the same as finding the strongly connected components in a directed graph. Next, we analyze the algorithms used to solve the problem. To validate our derivation, we run the program with a different number of people and communication records to see the trend of CPU time growth.

2. Analysis & Implementation

2.1 Grouping Friends and Strongly Connected Components

In this homework, we are given the data which contains a list of names and communication records (name1 -> name2). The objective is to form the groups such that the member of the group satisfies the following rules:

1. They have sent and received messages directly between them.
2. They have sent and received messages through one or more friends

between them.

If we think of the people as the vertices and the communication records as the directed edges. Then the rules are equivalent to that the members of the group are mutually reachable. Such set of vertices are called strongly connected components.

2.2 Strongly Connected Components

Here is a high-level description of the algorithm which can be used to find the strongly connected components

```
// to find the strongly connected components of the
// graph G = (V, E)
// input: G
// output: strongly connected components
Algorithm SCC(G)
{
     Construct the transpose graph G^T;
     DFS_Call(G); // perform DFS to get array f[1 : |V|]
     Sort V of G^T in order of
              decreasing value of f[v], v ∈ V
     DFS_Call(G^T);
     Each tree of the resulting DFS forest is
                  a strongly connected components
 }
```

As we can see, the algorithm consists of 4 parts. First, we need to construct the transposed version of the graph G^T. Second, we traverse the graph G to get the finish order. Next, we use the finish order to sort the vertices of G^T. Using the

property that members in the strongly connected components will be the same in G^T, we traverse G^T to get a forest of DFS trees. Each resulting tree is a strongly connected component.

2.3 Depth First Search

```
1. // initialization and recursive DFS function call
2. // input: graph G
3. // output: f[|V|]: finish order
4. //           SCCs: resulting DFS forest
5. Algorithm DFS_Call(G)
6. {
7.     for v := 1 to |V| do {
8.         visited[v] := 0; // initialize to not visited
9.         f[v] := 0; // reset finish order
10.      }
11.      time := 0; // global variable to track time
12.      for v := 1 to |V| do { // to handle forest case
13.          if (visited[v] = 0) then {
14.              SCCs[ptr] := -1; // separate trees
15.              ptr := ptr + 1;
16.              DFS_d(v);
17.          }
18.      }
19.  }
```

```
1. // depth first search starting from vertex v of graph G
2. // input: starting node v
3. // output: f[|V|]: finish order
4. //           SCCs: resulting DFS forest
5. Algorithm DFS_d(v)
6. {
7.     visited[v] := 1;
8.     SCCs[ptr] := v; // store v in the same tree
9.     ptr := ptr + 1;
10.    for each vertex w adjacent to v do {
```

```
11.         if (visited[w] = 0) then DFS_d(w);
12.     }
13.     time := time + 1;
14.     f[v] := time; // record finish order
15.  }
```

In depth first search, we traverse a graph in a manner that the if a vertex is unvisited, then we move to that vertex immediately, which makes the spanning tree deep. If we use adjacent matrix to store a graph, then the time complexity will be $\mathcal{O}(n^2)$ because in `DFS_Call` the loop run over 1 to *n*, and in `DFS_d` the loop must check *n* vertices to find all of its adjacent vertices. On the otherhand, if we use the adjacent list to store the graph, in `DFS_Call` we still need to run over 1 to *n*, but in `DFS_d` we already store the adjacent vertices together, therefore we don't need to run over all 1 to *n* to find its adjacent vertices. To be precise, we will spend a total of *e* times traversing the edges. Therefore, the time complexity is $\mathcal{O}(n + e)$

Besides storing the graph, both data structures need to store array f and visited, which takes $\mathcal{O}(n)$ additional space.

Time complexity: $\mathcal{O}(n + e)$ for adjacency list, $\mathcal{O}(n^2)$ for adjacency matrix

Overall space complexity: $\mathcal{O}(n) + \mathcal{O}(e)$ for adjacency list, $\mathcal{O}(n) + \mathcal{O}(n^2)$ for adjacency matrix

2.4 Sort Vertices

Because we are sorting elements based on their finish order, which are all integers, I think it would be a good opportunity to implement the counting sort. Here is the generic counting sort:

```
1. // sort A[1 : n] in nondecreasing order
2. // and puts results into B[1 : n].
3. // assume 1 <= A[i] <= k, for all i
4. // input: A, int n, k
5. // output: B contains sorted results
6. Algorithm CountingSort_geric(A, B, n, k)
7. {
8.     // initialize C to all 0
9.     for i := 1 to k do C[i] := 0;
10.    // count # elements in C[A[i]]
11.    for i := 1 to n do {
12.        C[A[i]] := C[A[i]] + 1;
13.    }
14.    // C[i] is the accumulate # of elements
15.    for i := 1 to k do {
16.        C[i] := C[i] + C[i - 1];
17.    }
18.    // store sorted order in array B
19.    for i := n to 1 step -1 do {
20.        B[C[A[i]]] := A[i];
21.        C[A[i]] := C[A[i]] - 1;
22.    }
23. }
```

Counting sort uses the value to be sorted, elements of array `A`, as index to store the position information in array `C`. When we need to store it to array `B`, we only need to look up for `A[i]` in array `C`.

However, by just implement the above pseudo code will not get the job done.

In this case, we are sorting a list of indices with their finish time, rather than sorting the finish time itself. Therefore, we need some modification.

```
1. // sort idx[1 : n] in nonincreasing order using key
2. // assume 1 <= A[i] <= k, for all i
3. // input: idx, int n, k
4. // output: sorted idx
5. Algorithm CountingSort(idx, key, n, k)
6. {
7.     // initialize C to all 0
8.     for i := 1 to k do C[i] := 0;
9.     // count # elements in C[key[idx[i]]]
10.      for i := 1 to n do {
11.          C[key[idx[i]]] := C[key[idx[i]]] + 1;
12.      }
13.      // C[i] is the accumulate # of elements
14.      for i := 1 to k do {
15.          C[i] := C[i] + C[i - 1];
16.      }
17.      for i := n to 1 step -1 do {
18.          // store sorted order back to array idx
19.          swap(idx[n - C[key[idx[i]]]], idx[i]);
20.          C[key[idx[i]]] := C[key[idx[i]]] - 1;
21.      }
22.  }
```

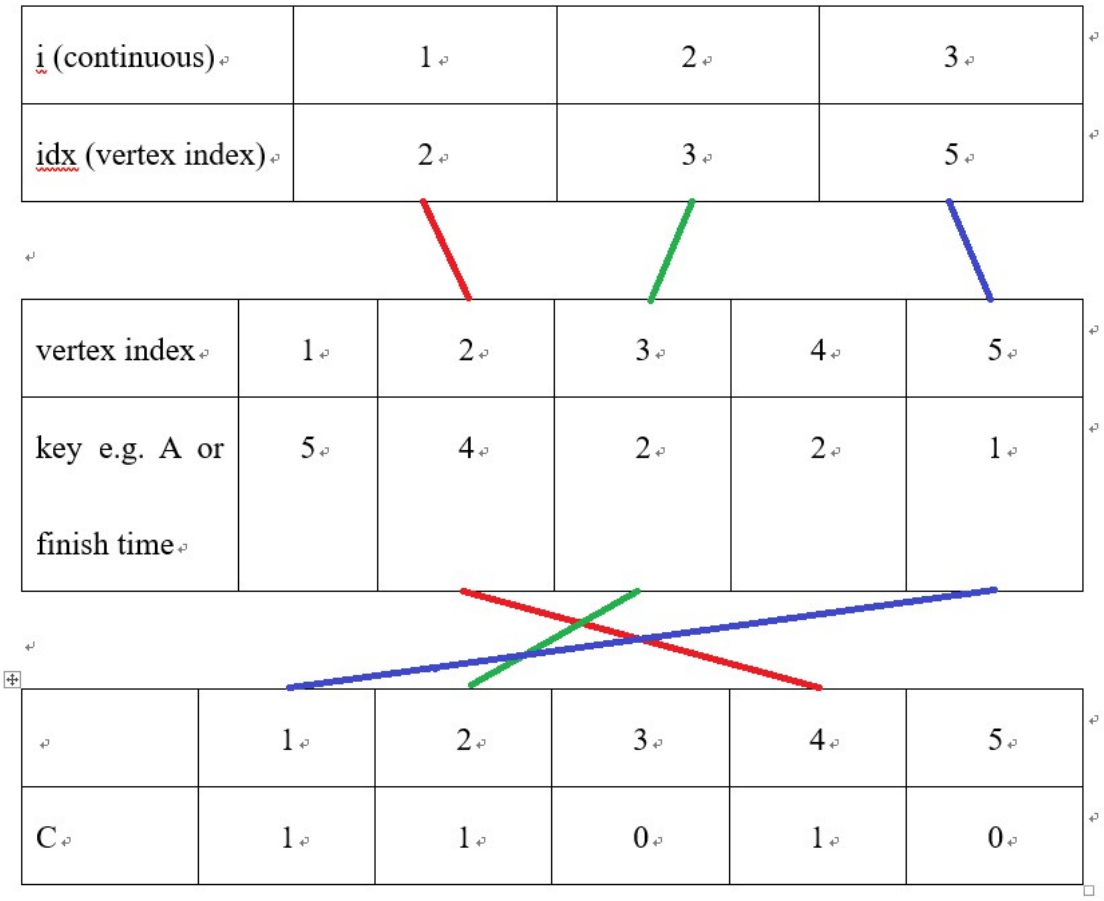
In `CountingSort_geric()`, we put the key `A[i]` in array `C` using `A[i]` as index, but in the modified version, we put `key[idx[i]]` because `key` right now has the same roll as `A` and we no longer assume the index to be 1 to n continuously. `idx` is the mapping from 1 to n to the real index of the vertices. In the last loop, we do not need extra array `B` like we do in `CountingSort_geric()`, instead we

permute the elements of array `idx` to the correct place. Also, in line 19, we store the element backward to achieve the nonincreasing property.

For the time complexity, we are looping over 1 to n and k, therefore it is $\mathcal{O}(n+k)$. For the space complexity, besides the array to be sorted, we need additional array C, which takes up $\mathcal{O}(k)$ additional space.

Time complexity: $\mathcal{O}(n+k)$

Overall space complexity: $\mathcal{O}(n) + \mathcal{O}(k)$



Schematic diagram of `CountingSort` up to line 11

2.5 Summary of Implementation

As we can see, the analysis above depends heavily on how we store the graph. Because I want to strike a balance between space and time complexity. I use dynamically allocated array to store the adjacent vertices. In the start, I initialize each vertex's the space to be e/n, and double the size if we run out of space. This way, the space complexity is better than adjacency matrix and a little worse than using linked list. We here assume each vertex has exactly e/n adjacent vertices, then the graph takes up $\mathcal{O}(e)$ space. If we analyze the total complexity:

1. construct the transposed graph G^T

   Time complexity: $\mathcal{O}(e)$

   Additional space complexity: $\mathcal{O}(e)$

2. DFS on G:

   Time complexity: $\mathcal{O}(n+e)$

   Additional space complexity: $\mathcal{O}(n)$

3. Sort vertices:

   Calling CountingSort(e/n, n) *n* times

   Time complexity: $n*\mathcal{O}(e/n+n) = \mathcal{O}(e+n^2) \approx \mathcal{O}(e)$

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

4. DFS on G^T

   Time complexity: $\mathcal{O}(n+e)$

   Additional space complexity: $\mathcal{O}(n)$

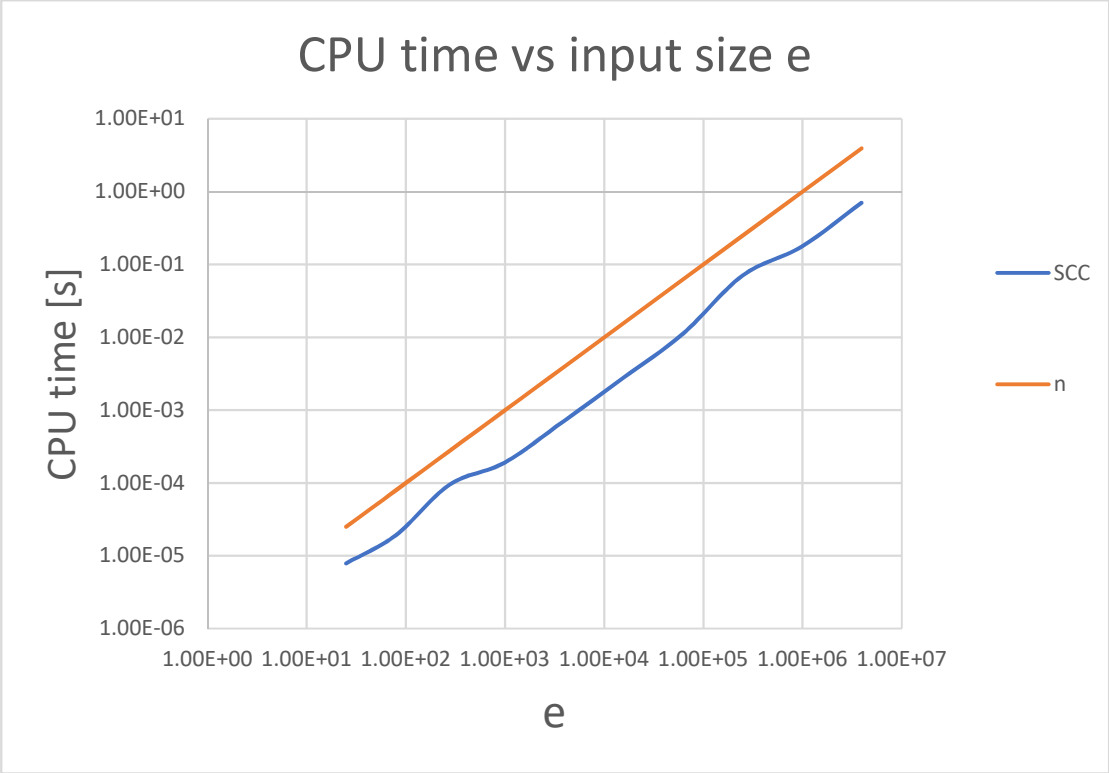
Therefore, the time total complexity is around $\mathcal{O}(e)$ and so does the space complexity.

3. Result and Observation

To measure the performance, we execute the program to solve the friend grouping problem with a different people and communication records.

e	CPU time [s]
25	7.87E-06
79	1.91E-05
274	9.39E-05
996	1.91E-04
3926	7.09E-04
15547	2.76E-03
61786	1.11E-02
246515	6.99E-02

984077	1.76E-01
3934372	7.02E-01



As we can see, the trend indeed follows the linear trend with input size *e*, thus validate our analysis.