10802EE 398000 Algorithms

Homework 7. Grouping Friends

● Introduction

In this report, I will design an algorithm to divide a group of N people into friend subgroups. The definition of a friend subgroup is the following: Given any two persons belong to a friend subgroup if

1. They have sent and received messages directly between them.

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

The input files c1.dat – c10.dat are given. The first line of each file consists of two integers: the total number of people being studied, and the number of communication records. It then followed by the Chinese names of all the people, and the communication records (in the format of "sender -> receiver").

To solve this problem, I transform input information into the directed graph G(V, E). V represents the total number of people being studied. E represents the number of communication records. Each vertex (or "node") represents the person. Each edge represents the communication record. For example, there are eight people 林一, 吳二, 李三, 張四, 黃五, 戴六, 欽七, 拜八. The communication records are 林一 -> 吳二, 吳二 -> 李三, 吳二 -> 戴六, 吳二 -> 黃五, 李三 -> 張四, 李三 -> 欽七, 張四 -> 李三, 張四 -> 拜八, 黃五 -> 林一, 黃五 -> 戴六, 戴六 -> 欽七, 欽七 -> 戴六, 欽七 -> 拜八. Then the graph is G(8, 13). It will be like **Figure 1.**

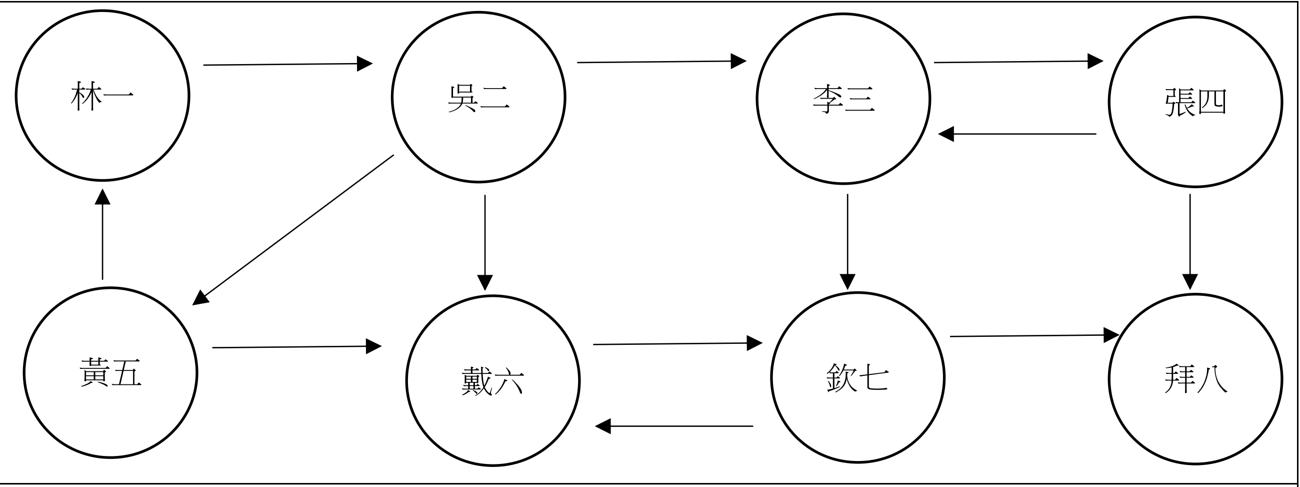

Figure 1 the example graph

We can now transform the definition of a friend subgroup as below: **a maximal set of vertices C ⊆ V such that for every pair of vertices u and v they are mutual reachable; that is, vertex u is reachable from v and vice versa.** This is the definition of **Strongly Connected Component**. Thus, we have to find the strongly connected components of the graph which represents friend subgroups.

There are three parts in the program. **First**, we have to read the input file and make the graph. **Second**, we have to find strongly connected components of the graph. **Last**, we have to print the results and free allocated memory.

In **Approach** section, I will focus on introducing **the second part, the subgroup generation part**. I will show function pseudocodes. Besides, I will explain the correctness of them, analyze their time complexities and space complexities.

In **Results** section, I will measure **the CPU time of the second part, the subgroup generation part**, and compare the results with those I have derived in **Approach** to see whether they are same.

In **Observations** section, I will introduce some methods to improve the running time of the program. Also, I will discuss some problems I have encountered in this homework.

● Approach

To find strongly connected components, we have to implement DFS_Call, DFS_d, topsort_Call, and topsort first. Please note that I use the **adjacency list** to represent the graph in the program.

```
// Depth first search starting from vertex v of graph G to get the finish time array.
// t is the global variable with initial value 0
// Input: starting node v
// Output: f[|V|]: finish time.
Algorithm DFS_d(v)
{
    visited[v] := 1;
    for each vertex w adjacent to v do {
        if (visited[w] = 0) then {
            DFS_d(w);
        }
    }
    f[t] := v;
    t := t + 1;
}
```

Algorithm 1 The pseudocode of DFS_d

DFS_d is the simple depth-first search from the input vertex v. First, mark the vertex v as visited. Then visit each unvisited vertex w adjacent to v. Last, record the finish time of the vertex v. Please note that **I use f[t] = v instead of f[v] = t and let t be consecutive from 0 to v - 1**. This will make the following process smoother. I will explain later.

The space complexity is O(V + E) because it is characterized by G(V, E), the graph with the number of vertices equal V and the number of edges equal E.

The time complexity is O(V + E) because DFS will travel every node through each edge.

To handle the forest case, we have to use DFS_Call to call the DFS_d function.

```
// Initialization and recursive DFS function call.
// Input: graph G
// Output: f[|V|]: finish time.
1 Algorithm DFS_Call(G)
2 {
3       for v := 1 to n do { // Initialize to not visited and no predecessor.
4               visited[v] := 0;
5               f[v] := 0;
6       }
7       t := 0; // Global variable to track finish time.
8       for v := 1 to n do { // To handle forest case.
9           if (visited[v] = 0) then DFS_d(v);
10      }
11 }
```

Algorithm 2 The pseudocode of DFS_Call

The most important part is line 8 to line 10, the loop is used to find whether there is any unvisited vertices after doing the DFS_d. If there is any, then do the DFS_d from that vertex.

The space complexity is O(V + E) because it is characterized by G(V, E), the graph with the number of vertices equal V and the number of edges equal E.

The time complexity is also O(V + E) because DFS will travel every node through each edge.

Next, the top_sort is really similar to DFS_d. It is used to find the name list slist[i] of the i-th subgroup. The only difference between DFS_d and top_sort is that I replace the finish time recording part (line 9 to 10 in DFS_d) with “add v to the head of slist[groupNumber]”. Since the DFS will only record the finish time of each node once, it will also only add the name to the name list once.

```
// Topological sort using depth-first search algorithm.
// Input: v starting vertex
// Output: the member of groupNumber-th subgroup
Algorithm top_sort(v, slist[groupNumber])
{
    visited[v] := 1;
    for each vertex w adjacent to v do {
        if (visited[w] = 0) then top_sort(w);
    }
    add v to the head of slist[groupNumber];
}
```

Algorithm 3 The pseudocode of top_sort

To call top_sort, we have to implement topsort_Call which is shown below. It is really similar to DFS_Call. However there is the important difference: we have to **do the DFS from the nodes in order of decreasing value of its finish time respectively**. I will explain the purpose later.

```
// Initialization and recursive top_sort function call.
// Input: graph G, the index of vertices in order of decreasing finish time (finishTimeDe array)
// Output: slist array (the member of each subgroup), the number of groups (groupNumber)
Algorithm topsort_Call(G, slist)
{
    for v := 1 to n do {
        visited[v] := 0;
        slist := NULL ;
    }
    groupNumber = 0;
    for i := 1 to n do {
        if (visited[finishTimeDe[i]] = 0) then {
            top_sort(finishTimeDe[i], slist[groupNumber]);
            groupNumber = groupNumber + 1;
        }
    }
}
```

Algorithm 4 The pseudocode of topsort_Call

The finishTimeDe array is the index of vertices in order of decreasing finish time. For example, if the finish time of three vertices node 0, node1, node2 are 2, 0, 1. Then the finishTimeDe array will be like {0, 2, 1}. Thus, we have to start the topsort from node 0, then from node 2, last from node 1.

The space complexity is O(V + E) because it is characterized by G(V, E), the graph with the number of vertices equal V and the number of edges equal E.

The time complexity is also O(V + E) because DFS will travel every node through each edge.

Finally, I can start to introduce SCC algorithm. There are four steps in finding strongly connected components: **construct the transpose graph, perform DFS on the original graph to get finish time array, sort the finish time array in order of decrease value, and perform DFS on the transpose graph to get the name list of subgroup.** The algorithm is shown below:

```
// To find the strongly connected components of the graph G = (V,E).
// Input: graph G
// Output: strongly connected components.
1 Algorithm SCC(G)
2 {
3       Construct the transpose graph GT ;
4       DFS_Call(G); // Perform DFS to get array f[1 : n].
5       Sort V of GT in order of decreasing value of f[t], t ∈ V.
6       topsort_Call(GT ); // Perform DFS on GT.
7       Each tree of the resulting DFS forest is a strongly connected component.
8 }
```

Algorithm 5 The pseudocode of SCC

The whole procedure can be illustrated below:

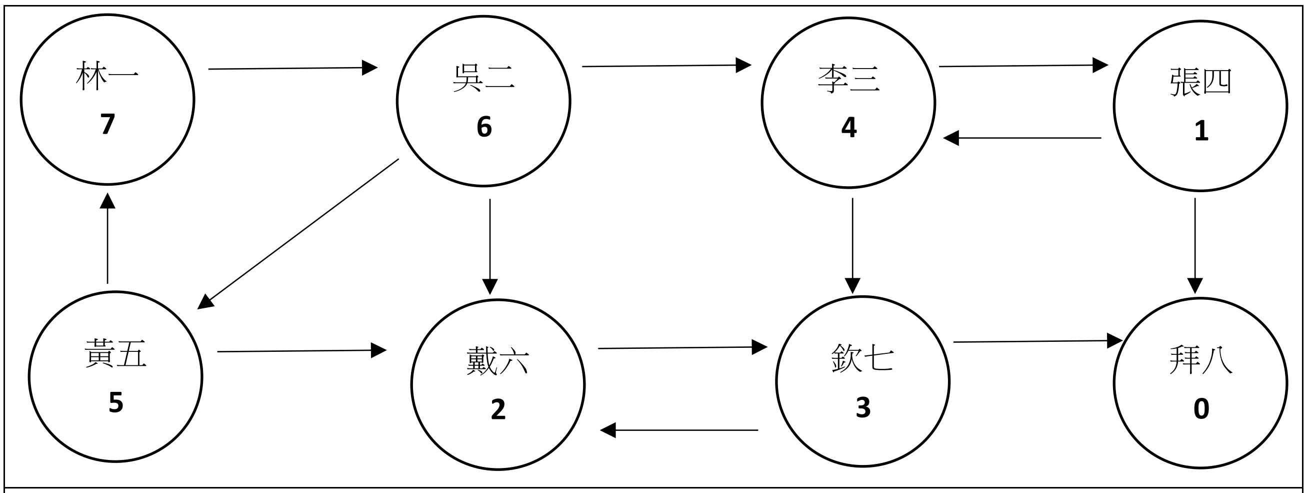

Figure 2 find the finish time of each node.

After finding all finish time, we can get the finishTime array {7, 3, 5, 6, 2, 4, 1, 0} where 林一 is node 0, 吳二 is node 1, 李三 is node 2, 張四 is node 3, 黃五 is node 4, 戴六 is node 5, 欽七 is node 6, 拜八 is node 7. Please note that **finishTime[i] represents the node index and i represents the finish time.** Then we have to sort the index of vertices in order of decreasing finish time to get finishTimeDe array. This can be done by single loop. All we have to do is to reverse the finishTime array and get finishTimeDe array {0, 1, 4, 2, 6, 5, 3, 7}.

The transpose graph of the original graph is shown below:

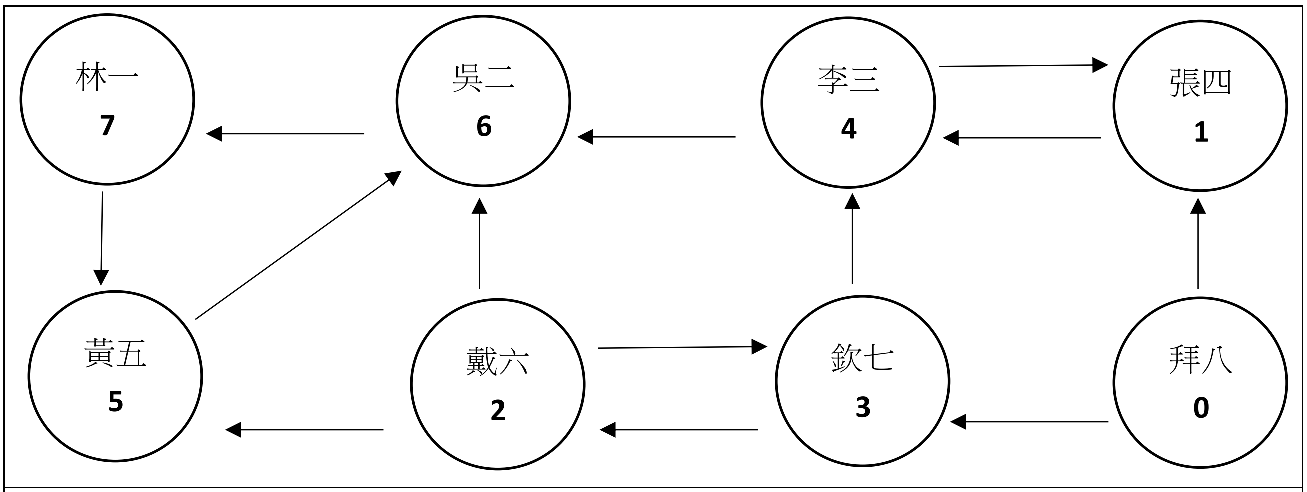

Figure 3 find the finish time of each node.

To construct the transpose graph, I simply travel each edge and swap the sender and receiver vertices to make the direction of edges be opposite.

Finally, we can start doing the DFS on transpose graph. Please note that **the DFS has to do from the node in order of decreasing value of its finish time respectively.** That is, for finishTimeDe = {0, 1, 4, 2, 6, 5, 3, 7}, we will first do the DFS(topsort) from node 0, then from node 1, then from node 4, etc. After doing this, we can get the slist. slist[0] = {林一, 黃五, 吳二} which indicates {0, 4, 1}. slist[1] = {李三, 張四} which indicates {2, 3}. slist[2] = {欽七, 戴六} which indicates {6, 5}. slist[3] = {拜八} which indicates {7}. This is the whole procedure of finding strongly connected components.

The space complexity of SCC is O(V + E) because it is characterized by G(V, E), the graph with the number of vertices equal V and the number of edges equal E.

The time complexity of SCC is O(V + E) because the time complexity of "Construct the transpose graph GT" is **O(V + E)**, the time complexities of DFS_Call and topsort_Call are both **O(V + E)**, and the time complexity of "Sort V of GT in order of decreasing value of f[t], t ∈ V" is **O(V)**.

The main function for this homework and the time and space complexities table are shown in the next page.

```
// main function
Algorithm main()
{
    read the input data;    // part 1
    make the graph; // part 1
    t = GetTime();
    SCC(); // part2: find strongly connected components
    t = GetTime() – t;
    output results;
    free allocated memory;
}
```

Algorithm 6 The pseudocode of main function

	Space complexity	Time complexity
SCC	**O(V + E)**	**O(V + E)**
DFS_Call	O(V + E)	O(V + E)
DFS_d	O(V + E)	O(V + E)
topsort_Call	O(V + E)	O(V + E)
topsort	O(V + E)	O(V + E)

Table 1 The time and space complexities.

● Results

Input			CPU Time of **SCC()** function (seconds)	Number of subgroups
file	Number of Vertices	Number of Edges		
c1.dat	10	25	7.86781E-06	3
c2.dat	20	79	1.88351E-05	5
c3.dat	40	279	4.88758E-05	6
c4.dat	80	996	1.68085E-04	9
c5.dat	160	3926	6.34193E-04	14
c6.dat	320	15547	2.71797E-03	20
c7.dat	640	61786	1.21491E-02	30
c8.dat	1280	246515	6.61249E-02	43
c9.dat	2560	984077	3.34829E-01	63
c10.dat	5120	3934372	1.52E+00	92

Table 2 Results table

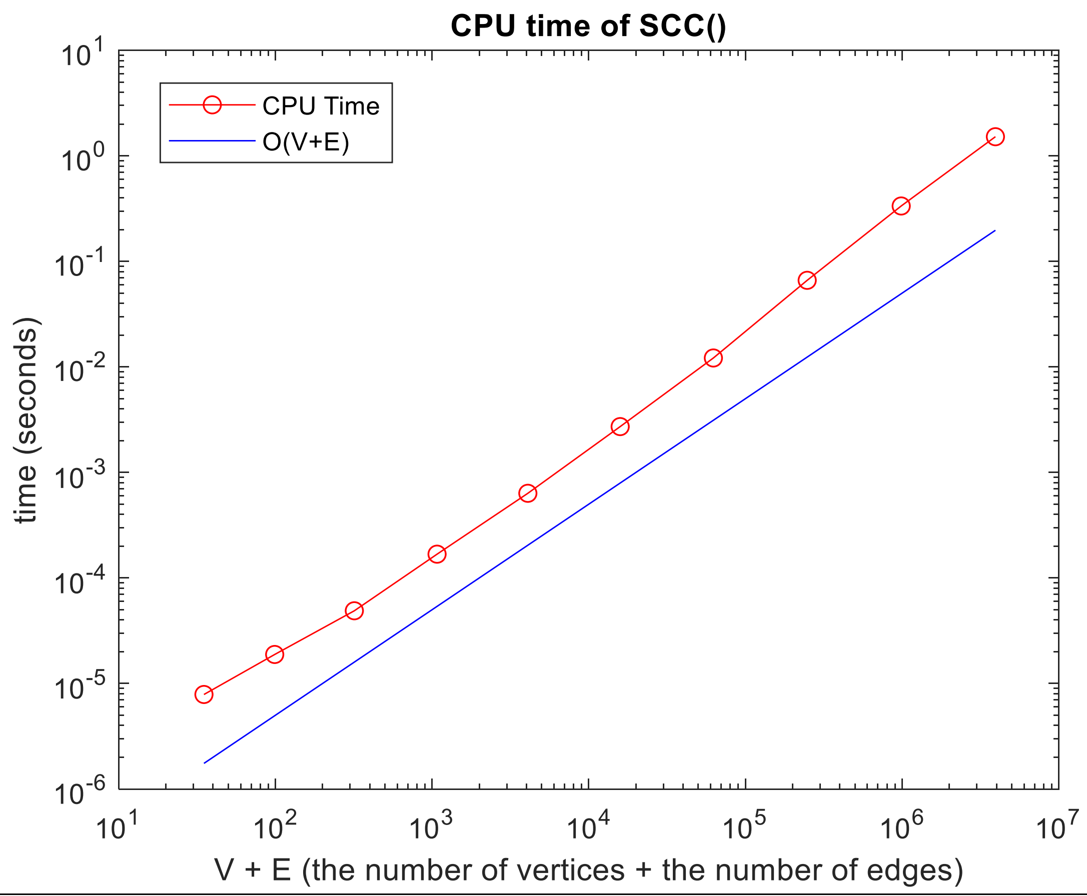

Figure 4 CPU time of SCC() function

The results in **Figure 4** is consistent with **Table 1**. The time complexity of SCC function is O(V + E).

● Observations

The first problem is to make reading the input graph and constructing the graph faster. Please note that I create the structure Node to store each node in the graph. The structure contains the information of the node including its index, its name, visited or not, and the next pointer. Because the names of input vertices are in Chinese, we have to make each name map to the node index. Most importantly, once we are given the name, we have to find the corresponding node and its index as fast as possible.

To do this, I use the hash table. The **hashMap** is shown below:

```
// To find the index of the node in the hash table
// Input: name
// Output: sum (the index of the node in the hash table)
1 Algorithm hashMap(name)
2 {
3       length := the length of the name;
4       for i := 1 to length do
5               sum = sum + (name[i] * (i + 1) * (i + 1) * (-1));
5       make sure that sum is positive value;
6       sum = sum mod HASH_TABLE_SIZE; // HASH_TABLE_SIZE is the size of hash table
7 }
```

Algorithm 7 The pseudocode of hashMap

hashMap can map the name to the index of it in the hash table. However, this is not enough. We want to get the index of node not the index in the hash table. Thus, **findIndex** is implemented:

```
// To find the index of the node in the graph
// Input: name
// Output: index
1 Algorithm findIndex(name)
2 {
3       idx := hashMap(name);
4       travel the idx-th bucket of the hash table to see whether there is a node named name;
5       return the index value of the node. If finding is failed, return -1;
6 }
```

Algorithm 8 The pseudocode of findIndex

After using the hash table, we can make great progress in reducing the overall runtime of the program. Before using the hash table (I use the simple linear search in the beginning), the runtime is about four minutes for c10.dat. After using the hash table, the runtime can shrink to only six seconds.

There are some interesting parts in designing the hash function. The purpose of the hash function is to map the name to the index of it in the hash table. Ideally, it must be an one - to - one function. However, there might be some cases such that it becomes a many - to - one function. In this case, we called collision. Thus, I use the overflow pages to deal with this problem, but it will increase the CPU time. To avoid the collision happen, a good hash function is really important. A good hash function has to make the output of it be uniformly distributed.

The second problem is how to speed up the process of inserting new node to the adjacency list. That is, to make the graph with the time complexity O(V + E). I use the last pointer array to point to the last node. last[i] will point to the last node of graph[i]. Thus, we can simply insert the node at last[i] -> next. This makes each insertion become O(1).

After dealing with these problems, the program can be much faster.

The third problem is that I have heard someone uses the adjacency matrix and can reach the faster CPU time for SCC function. I think there are two reasons: first, the overhead of malloc is too large (100 ~ 10000 times slower than +-*/). Second, for c10.dat, V^2 is smaller than V + E.

The reason way I still use the adjacency list is that it is more flexible in storage. That is, if the edges of the graph is much fewer than the vertices of it. Then it will waste a lot of space if using adjacency matrix. Thus, under different conditions, they have their pros and cons.