# 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

```
// initialization and recursive DFS function call
// input: graph G
// output: f[|V|]: finish order
//           SCCs: resulting DFS forest
Algorithm DFS_Call(G)
{
  for v := 1 to |V| do {
    visited[v] := 0; // initialize to not visited
    f[v] := 0; // reset finish order
   }
   time := 0; // global variable to track time
   for v := 1 to |V| do { // to handle forest case
     if (visited[v] = 0) then {
       SCCs[ptr] := -1; // separate trees
       ptr := ptr + 1;
       DFS_d(v);
     }
   }
 }

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

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

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 $O(n^2)$ 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 $O(n + e)$

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

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

Overall space complexity: $O(n) + O(e)$ for adjacency list, $O(n) + O(n^2)$ 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:

```
// sort A[1 : n] in nondecreasing order
// and puts results into B[1 : n].
// assume 1 <= A[i] <= k, for all i
// input: A, int n, k
// output: B contains sorted results
Algorithm CountingSort_geric(A, B, n, k)
{
    // initialize C to all 0
    for i := 1 to k do C[i] := 0;
    // count # elements in C[A[i]]
    for i := 1 to n do {
        C[A[i]] := C[A[i]] + 1;
    }
    // C[i] is the accumulate # of elements
    for i := 1 to k do {
        C[i] := C[i] + C[i - 1];
    }
    // store sorted order in array B
    for i := n to 1 step -1 do {
        B[C[A[i]]] := A[i];
        C[A[i]] := C[A[i]] - 1;
    }
 }
```

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.

```
// sort idx[1 : n] in nonincreasing order using key
// assume 1 <= A[i] <= k, for all i
// input: idx, int n, k
// output: sorted idx
Algorithm CountingSort(idx, key, n, k)
{
    // initialize C to all 0
    for i := 1 to k do C[i] := 0;
    // count # elements in C[key[idx[i]]]
      for i := 1 to n do {
          C[key[idx[i]]] := C[key[idx[i]]] + 1;
      }
      // C[i] is the accumulate # of elements
      for i := 1 to k do {
          C[i] := C[i] + C[i - 1];
      }
      for i := n to 1 step -1 do {
          // store sorted order back to array idx
          swap(idx[n - C[key[idx[i]]]], idx[i]);
          C[key[idx[i]]] := C[key[idx[i]]] - 1;
      }
  }
```

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

| | | | |
|---|---|---|---|
| i (continuous) | 1 | 2 | 3 |
| idx (vertex index) | 2 | 3 | 5 |

| | | | | | |
|---|---|---|---|---|---|
| vertex index | 1 | 2 | 3 | 4 | 5 |
| key e.g. A or finish time | 5 | 4 | 2 | 2 | 1 |

| | | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| C | 1 | 1 | 0 | 1 | 0 |

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 $O(e)$ space. If we analyze the total complexity:

1. construct the transposed graph $G^T$ How?

   Time complexity: $O(e)$

   Additional space complexity: $O(e)$

2. DFS on G:

   Time complexity: $O(n + e)$

   Additional space complexity: $O(n)$

3. Sort vertices:

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

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

Space complexity: $O(n)$

4. DFS on $G^T$

Time complexity: $O(n+e)$

Additional space complexity: $O(n)$

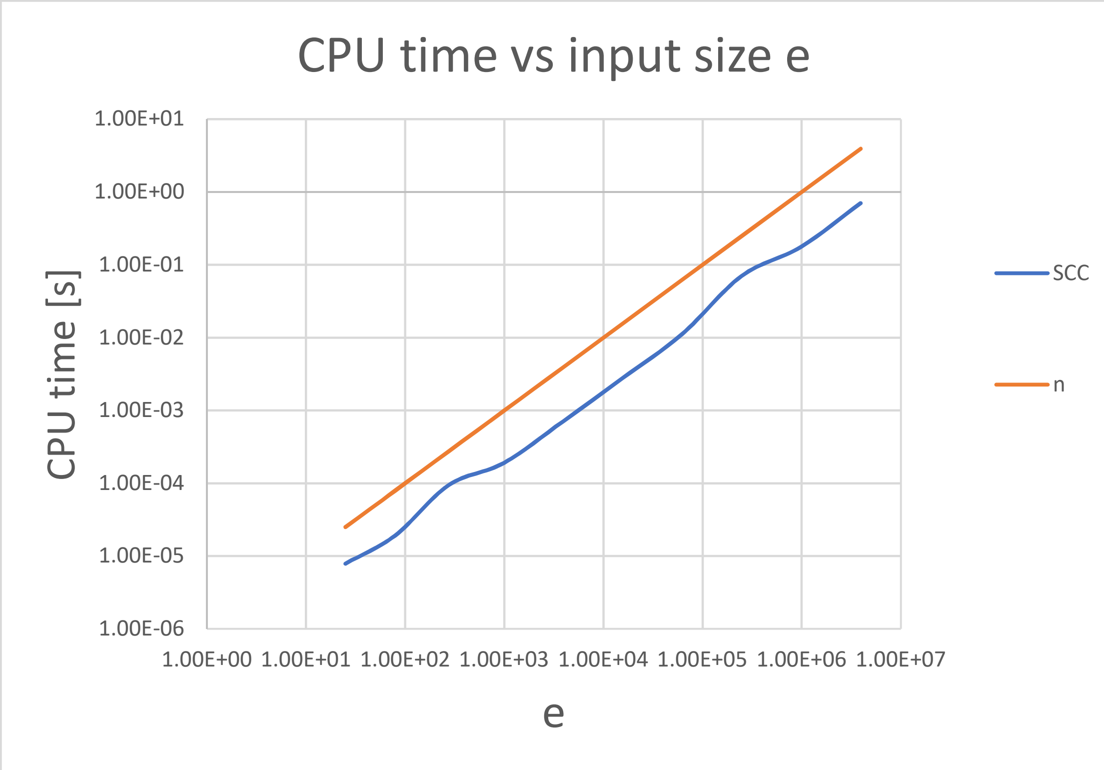
Therefore, the time total complexity is around $O(e)$ and so does the space complexity.

# 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.

# hw07.c

```c
// EE3980 HW07 Grouping Friends
// 105061110, 周柏宇
// 2020/04/24

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#define N 100 // number of class in the hash table

// structure to store the index whose name is in the same class
typedef struct node {
    int idx;
    struct node *next;
} Node;

int n_names; // number of people
int n_links; // number of communication records
int n_SCCs; // number of friend groups
char **names; // array of names
int **adj_l; // adjacency list
int *adj_l_size; // size of list in adj_l
int *adj_l_ptr; // current position of list in adj_l
int **adj_lT; // adjacency list (transposed)
int *adj_lT_size; // size of list in adj_lT
int *adj_lT_ptr; // current position of list in adj_lT
int *f; // node's finish order
int *idx; // index mapping
int *visited; // record visited nodes
int time_DFS; // DFS clock
int *SCCs; // record nodes traversed by DFS
int SCCs_ptr; // current position of SCCs
int *C; // temporary array for CountingSort()
Node **bucket; // hash table

void readData(); // read data, construct the graph and its transposed version
void readData(void); // read data, construct the graph and its transposed version
int nameToIndex(char *name); // convert name to array index
unsigned long hash(char *str); // hash function
double GetTime(void); // get local time in seconds
void SCC(); // find strongly connected components of a graph
void SCC(void); // find strongly connected components of a graph
void DFS_Call(int **G, int *len, int *idx);
                              // initialization and recursive DFS function call
void DFS_d(int **G, int *len, int v); // DFS from vertex v of the graph G
void CountingSort(int *idx, int *key, int n, int k);
                                 // sort the idx by its key using counting sort
```

```
void freeAll(); // free all allocated memory
void freeAll(void); // free all allocated memory

int main(void)
{
    int i; // loop index
    int component_idx = 0; // index of strongly connected component
    double start, end; // timestamp

    readData(); // read data, construct the graph and its transposed version
    start = GetTime(); // start time
    SCC(); // find strongly connected components of a graph
    end = GetTime(); // end time
    // print out input information, number of subgroups and execution time
    printf("N = %d M = %d Subgroup = %d ", n_names, n_links, n_SCCs);
    printf("CPU time = %.5e\n", end - start);
    printf("Number of subgroups: %d\n", n_SCCs); // print out number of groups
    for (i = 0; SCCs[i] != -2; i++) { // print out member of groups
        if (i == 0) {
            printf("  Subgroup %d:", ++component_idx);
        }
        else if (SCCs[i] == -1) {
            printf("\n  Subgroup %d:", ++component_idx);
        }
        else {
            printf(" %s", names[SCCs[i]]);
        }
    }
    printf("\n");
    freeAll(); // free all allocated memory

    return 0;
}

void readData()
void readData(void)
{
    int i, j, k, l; // indices
    int class; // class of a name determined by the hash function
    char tmp[20], tmp2[20], tmp3[20]; // temporary variable for input
    int *newptr; // temporary pointer
    Node *newNode; // temporary Node

    // input number of people and communication records
    scanf("%d %d", &n_names, &n_links);
    // allocate memory
    names = (char **)malloc(sizeof(char *) * n_names);
    adj_l = (int **)malloc(sizeof(int *) * n_names);
    adj_l_size = (int *)malloc(sizeof(int) * n_names);
    adj_l_ptr = (int *)calloc(n_names, sizeof(int));
```

```
    adj_lT = (int **)malloc(sizeof(int *) * n_names);
    adj_lT_size = (int *)malloc(sizeof(int) * n_names);
    adj_lT_ptr = (int *)calloc(n_names, sizeof(int));
    bucket = (Node **)calloc(N, sizeof(Node *));
    for (i = 0; i < n_names; i++) {
        scanf("%s", tmp); // input names
        names[i] = (char *)malloc(sizeof(char) * (3 * strlen(tmp) + 1));
        strcpy(names[i], tmp);
        class = hash(tmp) % N; // decide the name's class
        if (!bucket[class]) { // start of the linked list
            bucket[class] = (Node *)malloc(sizeof(Node));
            bucket[class]->idx = i; // store the index at this class
            bucket[class]->next = NULL;
        }
        else { // this class already has data
            newNode = (Node *)malloc(sizeof(Node));
            newNode->idx = i; // store the index at this class
            newNode->next = bucket[class];
            bucket[class] = newNode;
        }
    }
    for (i = 0; i < n_names; i++) {
        // initialize the adjacency list for the graph
        adj_l_size[i] = n_links / n_names;
        adj_l[i] = (int *)malloc(sizeof(int) * adj_l_size[i]);
        // initialize the adjacency list for the transposed graph
        adj_lT_size[i] = n_links / n_names;
        adj_lT[i] = (int *)malloc(sizeof(int) * adj_lT_size[i]);
    }
    for (i = 0; i < n_links; i++) {
        // input communication records
        scanf("%s %s %s", tmp, tmp2, tmp3);
        // convert names to array indices
        j = nameToIndex(tmp);
        k = nameToIndex(tmp3);

        // dynamically allocate memory for the adjacency list
        if (adj_l_ptr[j] >= adj_l_size[j]) {
            adj_l_size[j] *= 2; // double the size
            newptr = (int *)malloc(sizeof(int) * adj_l_size[j]);
                                               // allocate memory with new size
            for (l = 0; l < adj_l_ptr[j]; l++) newptr[l] = adj_l[j][l];
                                                 // copy the data to new memory
            free(adj_l[j]); // free old memory block
            adj_l[j] = newptr; // assign the new pointer
        }
        adj_l[j][adj_l_ptr[j]] = k; // store the edge <j, k>
        adj_l_ptr[j]++; // update current position in the list

        // dynamically allocate memory for the adjacency list
```

```
        if (adj_lT_ptr[k] >= adj_lT_size[k]) {
            adj_lT_size[k] *= 2; // double the size
            newptr = (int *)malloc(sizeof(int) * adj_lT_size[k]);
                                                   // allocate memory with new size
            for (l = 0; l < adj_lT_ptr[k]; l++) newptr[l] = adj_lT[k][l];
                                                     // copy the data to new memory
            free(adj_lT[k]); // free old memory block
            adj_lT[k] = newptr; // assign the new pointer
        }
        adj_lT[k][adj_lT_ptr[k]] = j; // store the edge <k, j>
        adj_lT_ptr[k]++; // update current position in the list
    }
}

int nameToIndex(char *name) // convert name to array index
{
    Node *tmp; // temporary Node

    tmp = bucket[hash(name) % N];
    while (tmp) { // return the index if the names are matched
        if (!strcmp(name, names[tmp->idx])) return tmp->idx;
        tmp = tmp->next;
    }

    return -1;
}

unsigned long hash(char *str) // hash function
{
    unsigned long hash = 5381;
    int c;

    while ((c = *str++)) {
        hash = ((hash << 5) + hash) + c;
    }

    return hash;
}

double GetTime(void)                        // get local time in seconds
{
    struct timeval tv;                      // variable to store time

    gettimeofday(&tv, NULL);                // get local time

    return tv.tv_sec + 1e-6 * tv.tv_usec;   // return local time in seconds
}

void SCC() // find strongly connected components of a graph
void SCC(void) // find strongly connected components of a graph
```

```
{
    int i; // loop index

    // allocate memory
    f = (int *)malloc(sizeof(int) * n_names);
    idx = (int *)malloc(sizeof(int) * n_names);
    SCCs = (int *)calloc(n_names * 2 + 1, sizeof(int));
    visited = (int *)malloc(sizeof(int) * n_names);
    C = (int *)malloc(sizeof(int) * n_names);

    // initialize the index
    for (i = 0; i < n_names; i++) idx[i] = i;

    // traverse the graph in the given order
    DFS_Call(adj_l, adj_l_ptr, idx);

    // sort the index using the array f as key (decreasing order)
    CountingSort(idx, f, n_names, n_names);

    // sort the adjacency list of the transposed graph in-place
    // using the array f as key (decreasing order)
    for (i = 0; i < n_names; i++) {
        CountingSort(adj_lT[i], f, adj_lT_ptr[i] - 1, n_names);
    }

    // traverse the transposed graph in the given order
    DFS_Call(adj_lT, adj_lT_ptr, idx);
}

// initialization and recursive DFS function call
void DFS_Call(int **G, int *len, int *idx)
{
    int i, j; // loop index

    // initialize
    for (i = 0; i < n_names; i++) visited[i] = 0;
    time_DFS = 0;
    SCCs_ptr = 0;
    n_SCCs = 0;
    for (i = 0; i < n_names; i++) {
        j = idx[i]; // decide the vertex to travel according idx
        if (visited[j] == 0) {
            SCCs[SCCs_ptr++] = -1; // separate different subgroups with -1
            n_SCCs++; // number of subgroups increase by one
            DFS_d(G, len, j); // start DFS from vertex j
        }
    }
    SCCs[SCCs_ptr] = -2; // end of array
}
```

```

void DFS_d(int **G, int *len, int v) // DFS from vertex v of the graph G
{
    int i, j; // loop index

    visited[v] = 1; // vertex v has been visited
    SCCs[SCCs_ptr++] = v; // add vertex v to the current subgroup
    for (i = 0; i < len[v]; i++) {
        j = G[v][i]; // decide next vertex to travel
        if (visited[j] == 0) {
            DFS_d(G, len, j); // DFS from vertex j
        }
    }
    f[v] = time_DFS++; // record the finishing order
}

// sort the idx by its key using counting sort
void CountingSort(int *idx, int *key, int n, int k)
{
    int i; // loop index
    int tmp, tmp_idx; // temporary variable

    for (i = 0; i < k; i++) C[i] = 0; // initialize C to all 0
    for (i = 0; i < n; i++) {
        C[key[idx[i]]]++; // count # elements in C[key[idx[i]]]
    }
    for (i = 1; i < k; i++) {
        C[i] += C[i - 1]; // C[i] is the accumulate # of elements
    }
    for (i = n - 1; i >= 0; i--) {
        // store sorted order back to array idx
        tmp_idx = n - (C[key[idx[i]]] - 1) - 1; // decreasing order
        tmp = idx[tmp_idx];
        idx[tmp_idx] = idx[i];
        idx[i] = tmp;
        C[key[idx[i]]]--;
    }
}

void freeAll() // free all allocated memory
void freeAll(void) // free all allocated memory
{
    int i; // loop index
    Node *tmp, *next;

    for (i = 0; i < n_names; i++) {
        free(names[i]);
        free(adj_l[i]);
        free(adj_lT[i]);
    }
```

```
    for (i = 0; i < N; i++) {
        tmp = bucket[i];
        while (tmp) {
            next = tmp->next;
            free(tmp);
            tmp = next;
        }
    }
    free(names);
    free(adj_l);
    free(adj_l_size);
    free(adj_l_ptr);
    free(adj_lT);
    free(adj_lT_size);
    free(adj_lT_ptr);
    free(visited);
    free(f);
    free(idx);
    free(SCCs);
    free(C);
    free(bucket);
}
```

[Program Format] can be improved.
[Writing] hw07a.pdf spelling errors: vertexs(1)
[CPU] time for c10.dat: 1.47000e-01 sec
[Approach] counting sort is not needed, DFS can be used to get sorted results.
[Approach] what is the transposed graph and how did you create one?
[Report] writing can be more clear.

Score: 81