EE3980 Algorithm

HW7. Grouping Friends

106061225 楊宇羲

1. Problem Description:

In this assignment, we are going to analyze about a special component that exists in a graph, which is called the "strongly connected component", which is considered to be able to applied in the field of communication. In a directed graph, we can define a strongly component as a component in a graph that each vertices can mutually reach every other vertices. In other words, is a vertex A is able to reach B with a given path but B can't get through A with any possible route, then the SCC that contains vertex A definitely will not contain vertex B. In this report, besides discussing how to determine all the SCCs in a graph, I will also focus on the time complexity and space complexity toward best case, average case and worst case of the implementation of this algorithm, then plot the result about relation between process time and graph size. We are provided with 10 lists of names which are consist of different size number of name and connection of each name in it. When scan in all the data in it, we have to be able to print out the number of total subgroup number and each element in a subgroup with no order requirement.

2. Approach:

The further explanation of how to obtain SCC will be discuss in the next part analysis. To approach our expected result, a definition of structure used for storing adjacent list is necessary, which is defined as below:

```
typedef struct CONN CONN;
struct CONN {
        int index;
        struct CONN *next;
};
```

Can be merged.

I am going to declare several dynamically allocated global lists and array used for storing the core name list, transpose list and the separation (root of each tree in the forest). The main function pseudo code and used global variable will be as below:

```
CONN **adlist, **adlistT, *roots;        // adjacent list, transposed list, subgroup roots
Int N, M, SUBSUM;                        // size of name, size of connection, total subgroup
Int *f, *s, *visited;                    // store first sort, second sort, check visit
```

```
Algorithm main(void)
{
        ReadData();                     // read the date and the price of the corresponding date
        t := GetTime();                 // start counting time
        SCC();
        t := GetTime() - t;             // stop counting time
        curr := roots;                  // curr is a CONN used to travel the list
        while (curr->next != NULL) {
                Write (curr, f[i]);     // f[I] store the name in topological order
                i++;
        }
}
```

There are also some function needed to be written to carry out the DFS algorithm and store the name list in order to print out the SCC.

```
Algorithm DFS_Call (graph)
{
        for I := 1 to N do {
                visited[I] := 0;        // not visited
                f[I] := 0;              // store list initialize
        }
        for I := 1 to N do {
                if (visited[I] == 0) then {
                        top_sort(I, f);
                        store root of tree in roots;
                        SUBSUM++;
                }
        }
}

Algorithm top_sort (v, graph)
{
        visited[v] := 1;
        curr = graph->next;
        while (curr) do {
                if (visited[curr->index] == 0) then {
                        top_sort(curr->index, graph);
                }
                curr = curr->next;
        }
        visited[I] := 2;
        add v to head of f list;
}
```

After the the DFS functions are set up, we can eventually call the SCC function and perform DFS on the adjacent list and transposed adjacent list to get the SCCs.

```
Algorithm SCC (void)
{
        for I from 1 to N do s[I] = I;
        DFS_Call (adlist);
        for I from 1 to N do f[I] = s[I];
        DFS_Call(adlistT);
}
```

In this homework, we are going to use adjacent list to represent every edges exist in the graph, which is represented as below:

A -> B -> C -> D -> NULL
B -> C ->A -> NULL
C -> A -> NULL
D -> B ->C ->NULL

In comparison with adjacent array, lists only takes little space since there aren't many edges to store if it is directional graph. When reading data, we store the edges in the *adlist* and *adlistT* simultaneously in the ReadData function, so that there is no need to build the transpose list in the SCC function.

After all functions used to determine the SCCs are done, we can run the SCC function for **one** time then use the *GetTime* function to calculate the process time and finally being able to analyze the relation about the edge number, name size and the process speed.

3. Analysis:

In this part we are going to analyze why performing DFS in adjacent list and transposed list can obtain the SCC. If we start from one random vertex and perform DFS on this directed graph, we might obtain a spanning tree or maybe several spanning trees, which indicates which vertices are reachable. However, SCC need to be "mutually" reachable, which infers that every point the original point can reach must also be able to reach the original point. Applying the property that a graph has the same SCC as the transposed graph (meaning every edges' direction is being opposite). So we can also perform second time DFS toward the transposed graph and double check to obtain the correct number of subgroup of strongly connected components.

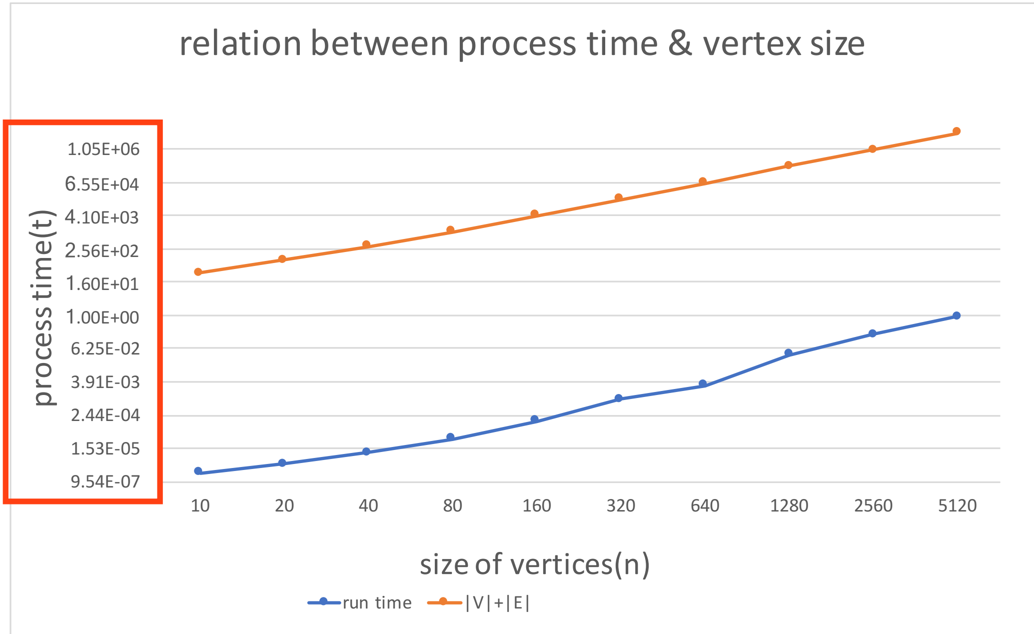
Here is an example of what performing only one time of DFS will causes:

Suppose there is a graph consists of 4 vertices A, B, C, D. A is able to reach every vertex but other vertex is not able to reach any vertex. If we perform DFS on A, then we will get only one SCC which is the whole graph. However, the fact Is that there are four SCCs which each vertices make up their own SCC. Once we apply DFS one more time on A, but transposed graph, we can then get the correct answer which is 4 SCC.

How did you get this?

The total time complexity is O(E+V), where E is the number of edges and V is the number of vertices. Since we call the SCC function one time to get the result, and the SCC function calls two time of DFS_Call, which goes through the whole vertices to get the forest consists of spanning trees.

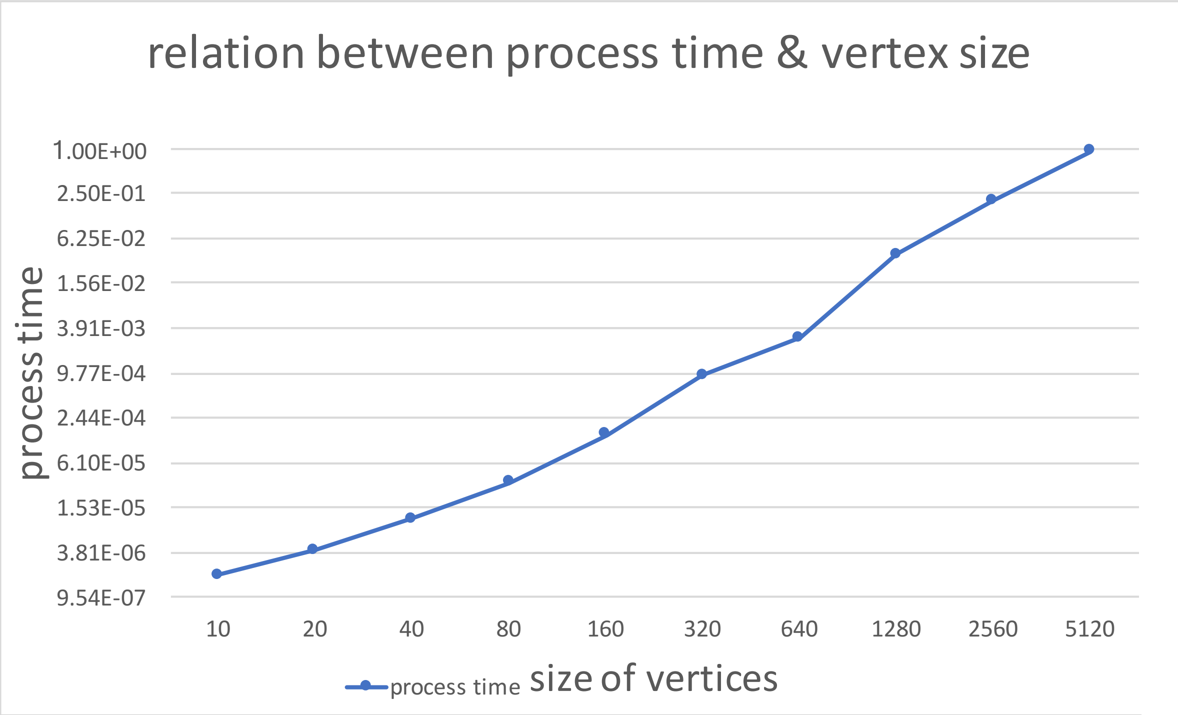
4. Results:

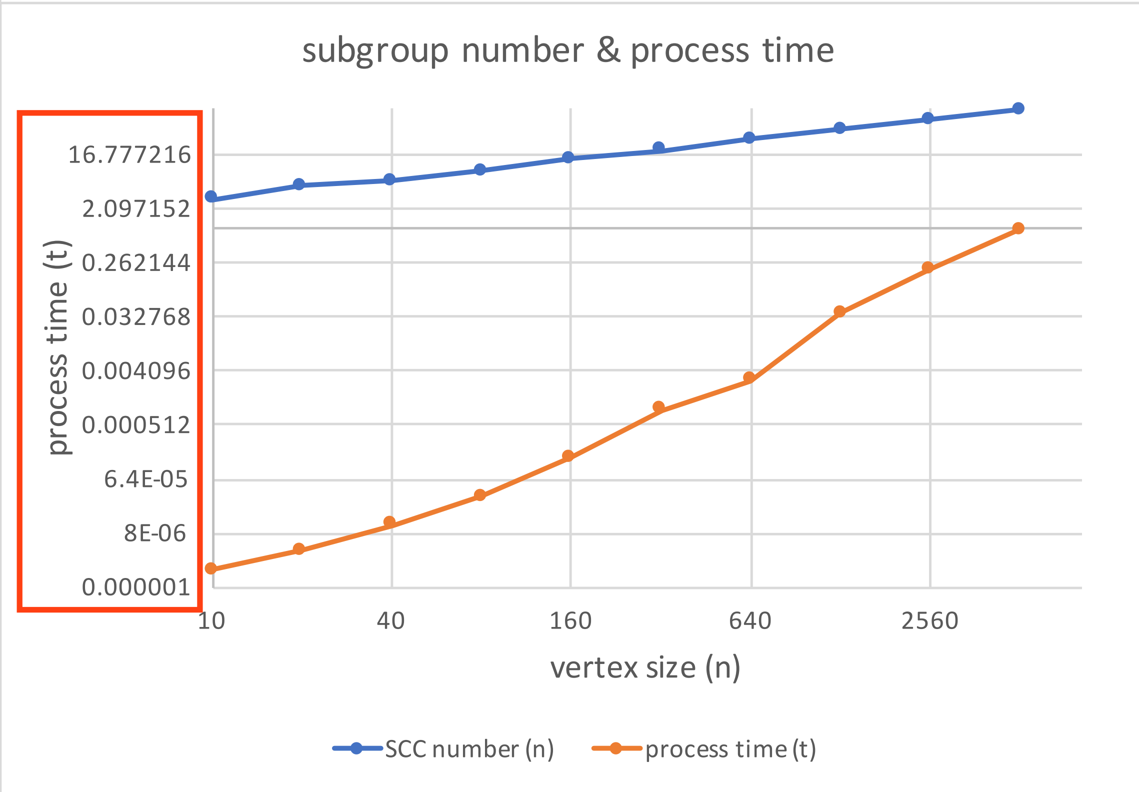


Unit?

vertex size(n)	edge size(n)	SCC number (n)	process time (t)
10	25	3	1.90E-06
20	79	5	4.05E-06
40	274	6	1.09E-05
80	996	9	3.29E-05
160	3926	14	1.44E-04
320	15547	20	9.21E-04
640	61786	30	2.88E-03

1280	246515	43	3.73E-02
2560	984077	63	2.03E-01
5120	3934372	92	9.13E-01





5. Observation and conclusion:

We can tell that the result mostly matches with the analysis, the table below is the time and space complexity of this algorithm:

	SCC
Time complexity	$\boxed{O(E+V)}$
Space complexity	$\boxed{O(E+V)}$

O(|E|+|V|)

```
$ gcc hw07.c
$ a.out < c5.dat
N = 160 M = 3926 Subgroup = 14 CPU time = 5.91278e-05
Number of subgroups: 14
  Subgroup 1: 谷慎林 鄭談佶 左優一 張殊媗 李政女 王遐科 廖絜筱
  Subgroup 2: 韓竟浩 周慶讚 陳因萱 嚴乃拓 沈臨濰 遲平榛 郭澄樂
  Subgroup 3: 馬食寧 車奉才 黎學淩 田深衍 譚陶美 魏業繁 高景竑 陳信孟
  Subgroup 4: 劉璧適 范量永 何睦珩 柴此鳴 藍文羚 金字謙 鍾興弋 彭取愷 賀草溥
  ...
  ...
  Subgroup 14: 黃珠同 王水嫻 吳五鍾
```

Good, solution is correct.

---

score: 65.0

- Report format
  - Need double line spacing
- Introduction
  - Introduction can be strengthened – to link the problem to SCC
- Time/Space complexity
  - Both space and time complexity analyses should be more complete.
- Results
  - Tables and figures can still be improved.
- Conclusion/observation
  - Conclusions can be strengthened.
- Program format can be improved
  - Please follow the coding guidelines.

hw07.c

```
// EE3980 HW07 Grouping Friends
// 106061225, 楊宇羲
// 2021/4/28

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

typedef struct CONN CONN;
struct CONN{
    int index;
    struct CONN *next;
};
Comments?

char **name;                            // list of people
int N;                                  // number of people
int M;                                  // number of connection
int SUBSUM;                             // number of SCC
int order;                              // position to store list
int *s, *f, *visited;                   // store sorted array, check visit
CONN **adlist;                          // adjacent list
CONN **adlistT;                         // adjacent transposed list
CONN *roots;                            // store root of SCC

double GetTime(void);                   // get local time in seconds
void ReadData(void);                    // store data
void PrintData(double t);               // print result
void SCC(void);                         // find SCC
void DFS_Call(CONN **graph);            // call DFS, store array
void top_sort(int i, CONN **graph);     // Depth First search recursion

int main(void)
{
    double t;                           // count time
    ReadData();                         // read in data
    t = GetTime();                      // Get CPU time
    SCC();                              // find SCC groups
```

```
    t = GetTime() - t;                       // Calculate CPU runtime
    PrintData(t);                            // print outcome

    return 0;
}

void ReadData(void)                          // read in data
{
    int i, j;                                // for loop
    int flag = 0;                            // name found
    char *source, *destination = NULL;       // an edge's head and tail

    scanf("%d %d", &N, &M);                  // size of vertex and edge

    /* some dynamic allocated array*/
    /* some dynamic allocated array */
    source = (char*)malloc(15 * sizeof(char));
    destination = (char*)malloc(15 * sizeof(char));
    name = (char**)malloc(N * sizeof(char*));
    adlist = (CONN**)malloc(N * sizeof(CONN*));
    adlistT = (CONN**)malloc(N * sizeof(CONN*));

    s = (int*)malloc(N * sizeof(int));
    visited = (int*)malloc(N * sizeof(int));
    f = (int*)malloc(N * sizeof(int));


    for (i = 0; i < N; i++) {                // store name to name list
        name[i] = (char*)malloc(15 * sizeof(char));
        adlist[i] = (CONN*)malloc(sizeof(CONN));
        adlist[i]->index = i;
        adlist[i]->next = NULL;
        adlistT[i] = (CONN*)malloc(sizeof(CONN));
        adlistT[i]->index = i;
        adlistT[i]->next = NULL;
        scanf("%s", name[i]);
    }

    for (i = 0; i < M; i++) {                // build adlist & adlistT

        CONN *curr;                          // travel through list
```

```
        CONN *build1;                       // build through one direction
        CONN *build2;                       // build through opposite direction

        build1 = (CONN*)malloc(sizeof(CONN));
        build1->next = NULL;
        build2 = (CONN*)malloc(sizeof(CONN));
        build2->next = NULL;

        scanf("%s -> %s",source, destination);
        scanf("%s -> %s", source, destination);

        for (j = 0; flag == 0; j++) {       // found destination name
            if (strcmp(destination, name[j]) == 0) {
                build1->index = j;
                curr = adlist[j];
                build2->next = curr->next;
                curr->next = build2;
                flag = 1;
            }
        }
        flag = 0;
        for (j = 0; flag == 0; j++) {       // found source name
            if (strcmp(source, name[j]) == 0) {
                build2->index = j;
                curr = adlistT[j];
                build1->next = curr->next;
                curr->next = build1;
                flag = 1;
            }
        }
        flag = 0;
    }

}

void PrintData(double t)                    // print outcome
{
    CONN *curr;                             // travel through list
    int i, j, k;                            // for determine index

    curr = roots;                           // find root position
```

```
    i = 0;
    k = 1;
    printf("N = %d M = %d Subgroup = %d CPU time = %g\n", N, M, SUBSUM, t);
    printf("Number of subgroups: %d\n", SUBSUM);

    while (curr->next != NULL) {
        printf("  Subgroup %d: ", k);       // each n is a subgroup
        for (j = 0; j < (curr->index - curr->next->index - 1); j++) {
            printf("%s " , name[f[i]]);
            printf("%s ", name[f[i]]);
            i++;
        }
        printf("%s" , name[f[i]]);
        printf("%s", name[f[i]]);
        printf("\n");

        curr = curr->next;
        i++;
        k++;
    }
}
Need a blank line here.
void SCC(void)                              // start determine SCC
{
    int i;
Need a blank line here.
    for (i = 0; i < N; i++) {               // initialize graph's travel order
        s[i] = i;
    }
    DFS_Call(adlist);                       // do DFS on adlist

    for (i = 0; i < N; i++) {               // initialize travel order
        s[i] = f[i];
    }
    DFS_Call(adlistT);                      // do DFS on transposed adlist
}

void DFS_Call(CONN **graph)
{
    int i;                                  // for loop
    CONN *curr;                             // travel through lists
```

```

    SUBSUM = 0;                             // initialize subgroup num
    roots = NULL;                           // store root position
    curr = (CONN*)malloc(sizeof(CONN));
    curr->index = 0;
    curr->next = roots;
    roots = curr;

    for (i = 0; i < N; i++) {               // initialize visit and order
        visited[i] = 0;
        f[i] = 0;
    }
    order = 0;
    for (i = 0; i < N; i++) {
        if (visited[s[i]] == 0) {           // if not visited
            top_sort(s[i], graph);          // DFS s[i] in graph
            CONN *curr;                     // store travel times
Do not mix declarations with statements
            curr = (CONN*)malloc(sizeof(CONN));
            curr->index = order;
            curr->next = roots;
            roots = curr;
            SUBSUM++;                       // subgroup num + 1
        }
    }
}

void top_sort(int v, CONN **graph)          // Depth First search
{
    CONN *curr;                             // travel through list

    visited[v] = 1;                         // v[i] is traveling
    for (curr = graph[v]->next; curr != NULL; curr = curr->next) {
        if (visited[curr->index] == 0) {    // adjacent index is not visited
            top_sort(curr->index, graph);   // DFS that vertice
        }
    }

    visited[v] = 2;                         // v visited
    f[N - order - 1] = v;                   // store travel orders
    order++;                                // search next index
```

```
}

double GetTime()                              // calculate time
double GetTime(void)                              // calculate time
{
    struct timeval tv;

    gettimeofday(&tv, NULL);
    return tv.tv_sec + 1e-6 * tv.tv_usec;   // sec + micro sec
}

```