EE3980 Algorithms

hw07 Grouping Friends

106061146 陳兆廷

Introduction:

In this homework, I will be analyzing, implementing, and observing 1 algorithm. The goal of the algorithms is to find all strongly connected components in a graph. The input of them will be a list of names and all connections between each of them, and the output of them will be number of subgroups and all the subgroups.

During the analysis process, I will be using counting method to calculate the time complexities of the algorithm. Furthermore, I will try to find the best-case, worst-case, and average-case conditions for the algorithm, respectively. Before implementing on C code, I will try to predict the result based on my analysis. Finally, I will calculate their space complexity for the total spaces used by the algorithm.

The implementation of the algorithm on C code mainly focus on the average time, best-case and worst-case conditions. 10 testing data are given by Professor Chang, and the working environment is on my ubuntu with Linux kernel 5.3.0, and gcc version 7.5.0.

Lastly, the observation of them will be focusing on the time complexity of the results from implementation. I will check the experimented results with my analysis.

Analysis:

1. Graph representation:

There are several ways of representing graphs and sets in programming. In this homework, we use 2 methods to do this.

a. Edge representation:

This representation of a graph is using a 2d array to store all the edges in a graph. First row of the array indicates number of vertices(V) and edges(E) of this graph. And the rest E rows represent edges that connecting 2 nodes. For an example, in graph Fig. 1, the representation of this graph will be Table 1. There are 4 vertices and 3 edges connecting 1-3, 1-4, 2-4. There are no directions for edges.

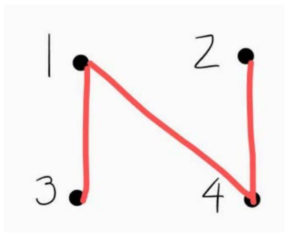


Fig 1

4	3
1	3
1	4
2	4

Table 1

b. Adjacent list representation:

This representation of a graph is using a 2d linked list to store all the edges in a graph. The number of rows is equal to the number of vertices, and nodes that are linked to each row represent the nodes that are connected to those vertices. For an example, in graph Fig. 2, the representation of this graph will be Table 2.

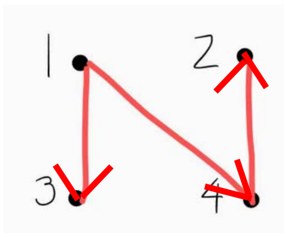


Fig 2

```
1 -> 3 -> 4 -> NULL
2 -> NULL
3 -> NULL
4 -> 2 -> NULL
```

Table 2

2. Strongly connected components (SCC):

Given a directed graph $G = (V, E)$, a strongly connected component is a maximal set of vertices $C \subseteq V$ 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. For example, in Fig 3, there are 2 sets of strongly connected components: {1, 3, 4} and {2}. Strongly connected components in G and G' are the same.

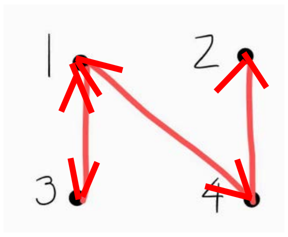


Fig 3

3. Depth-First Search (DFS):

Depth-first search is a way of traversing graphs or searching nodes in graphs. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

4. DFS_call (function that calls DFS) and DFS_d:

a. Abstract:

DFS_call initialize and call DFS function (*DFS_d*) recursively. It saves the number of strongly connected components and the number of nodes in each component.

DFS_d implements depth-first search on the given graph, *G*, and starts from the given node, *v*. It returns a list that represents the travel order of each node in *G*.

G in the below algorithm indicates the adjacent list representation of a graph, *G*. Array *l[i]* indicates the traversing order of the DFS algorithm.

b. Algorithm:

```
// More sophisticated Depth first search starting from vertex v of graph G.
// Input: starting node v, graph G
// Output: f[|V|]: travel order.
Algorithm DFS_d(G, v)
{
    visited [v] := 1 ;
    time := time + 1 ;
    d [v] := time ;
    for each vertex w adjacent to v in G do {
        if (visited [w] = 0) then {
            DFS_d(w) ;
        }
    }
    visited [v] := 2 ;
    f [N - 1 - time] := v;
    t := time + 1 ;
}
```

```
// Initialization and recursive DFS function call.
// Input: graph G, array l[V]: travel order
// Output: f[|V|]: finish time, steps: SCC index number SCCs, sub: SCCs num
Algorithm DFS_Call(G, l)
{
    sub := 0;
    steps[0] := 0;
    for v := 1 to n do { // Initialize to not visited and no predecessor.
        visited [v] := 0 ;
        f [v] := 0 ;
    }
    time := 0 ; // Global variable to track time.
    for v := 1 to n do { // To handle forest case.
        if (visited [l[v]] = 0) then {
            DFS_d(G, l[v]) ;
            steps.push(time);    // push time in front of steps.
            sub++;
        }
    }
}
```

c. Proof of correctness:

Claim 1: DFS is called exactly once by *DFS_d*.

Proof 1:

Since every time *DFS_d* called a node, v, it sets its visited[v] to 1, it is impossible for *DFS_d* or *DFS_call* to call *DFS_d* for vertices twice.

Claim 2: Each edge is traveled exactly once by *DFS_d*.

Proof 2:

Since claim 1 is proved, edges of each vertices are traveled exactly once in *DFS_d*.

Therefore, *DFS_call* travels through the graph in DFS order and returns information we want during the process.

d. Time complexity:

Since Claim 1 and Claim 2 from previous part is proved, we can see that *DFS_call* goes through exactly V vertices and E edges. Therefore, **the time complexity of *DFS_call* is O(V + E)**, where V is the number of vertices and E is the number of edges.

<u>Best case, Worst case and Average case:</u>

The difference between average case and worst case for this is not obvious. The reason for this is that either way, it goes through all vertices and edges anyway. The best case for this is when a graph has no edges, in another way, it is still O(V + E) where E is 0.

e. Space Complexity:

The algorithm uses several integers and f[V], visited[v], G(V, E). **The space complexity would be O(V + E).**

5. SCC:

a. Abstract:

SCC finds the strongly connected components of the graph G = (V, E).

b. Algorithm:

```
// To find the strongly connected components of the graph G = (V, E ).
// Input: graph G
// Output: strongly connected components.
Algorithm SCC(G )
{
    for i in V do f[i] = i
    DFS_Call(G, f) ; // Perform DFS to get array f [1 : n].
    DFS_Call(GT, f) ; // Perform DFS on GT.
}
```

c. Proof of correctness: (infer from lec42.pdf)

In order to prove this, we must prove 3 lemmas.

<u>Lemma 1</u>

Let C and C' be two distinct strongly connected components in a directed graph $G = (V, E)$. If $u, v \in C$, $u', v' \in C'$ and there is a path from u to u' then G cannot contain a path from v' to v.

<u>Proof 1:</u>

If G contains a path from v' to v, C and C' cannot be two distinct strongly connected components in G, hence proved.

<u>Lemma 2:</u>

Let C and C' be two distinct strongly connected components in a directed graph $G = (V, E)$. Suppose there is an edge $(u, v) \in E$, where $u \in C$ and $v \in C'$. Then, the largest finish time for each node in C is larger than the largest finish time for each node in C'.

Proof 2:

There must be one DFS_d that goes through edge (u, v), where the time for that DFS_d is larger than the former one by 1, hence proved.

Lemma 3:

Let C and C' be two distinct strongly connected components in a directed graph $G = (V, E)$. Suppose there is an edge $(u, v) \in E^T$, where $u \in C$ and $v \in C'$. Then, the largest finish time for each node in C is smaller than the largest finish time for each node in C'.

Proof 3:

Since $(u, v) \in E^T$, $(v, u) \in E$, by lemma 2, lemma 3 is proved.

Proof for SCC:.

Using induction, we can say that in the first k trees produced by DFS_call(G^T) are strongly connected components. For k = 0, the statement still fits since NULL is a strongly connected component.

For the k + 1 step, let the root of this tree be u, and $u \in C$. Because of lemma 2, for any strongly connected component C' other than C, the largest finish time for each node in C is larger than the largest finish time for each node in C'. Furthermore, when DFS_call visits u, all other descendants in C are not visited. Now, using lemma 3, we can say that any edges in G^T that leave C must be to strongly connected components that have already been visited. Thus, no vertex in any strongly connected component other than C will be a descendant of u during DFS_call(G^T). Therefore, the vertices of DFS_call(G^T) form exactly one strongly connected component. Finally, by induction, every tree of SCC in DFS_call(G^T) are strongly connected components.

d. Time complexity:

It basically do *DFS_call* twice. Therefore, **the time complexity of *SCC* is O(V + E)**, where V is the number of vertices and E is the number of edges.

Best case, Worst case and Average case:

The difference between average case and worst case for this is not obvious. The reason for this is that either way, it goes through all vertices and edges anyway. The best case for this is when a graph has no edges, in another way, it is still O(V + E) where E is 0.

e. Space Complexity:

The algorithm uses several integers and f[V], visited[V], G(V, E). **The**

space complexity would be O(V + E).

6. Time & Space:

	SCC
Time complexity	**O(V + E)**
Space complexity	**O(V + E)**

Implementation:

1. Speed Test:

Speed Test is to find the actual speed and time complexities of the algorithm, *SCC*. We use 10 test inputs given by Professor and get the CPU runtimes before and after the algorithms perform their tasks. The implementation is done on my laptop. I will run the algorithm 1 times and record the CPU runtime for it.

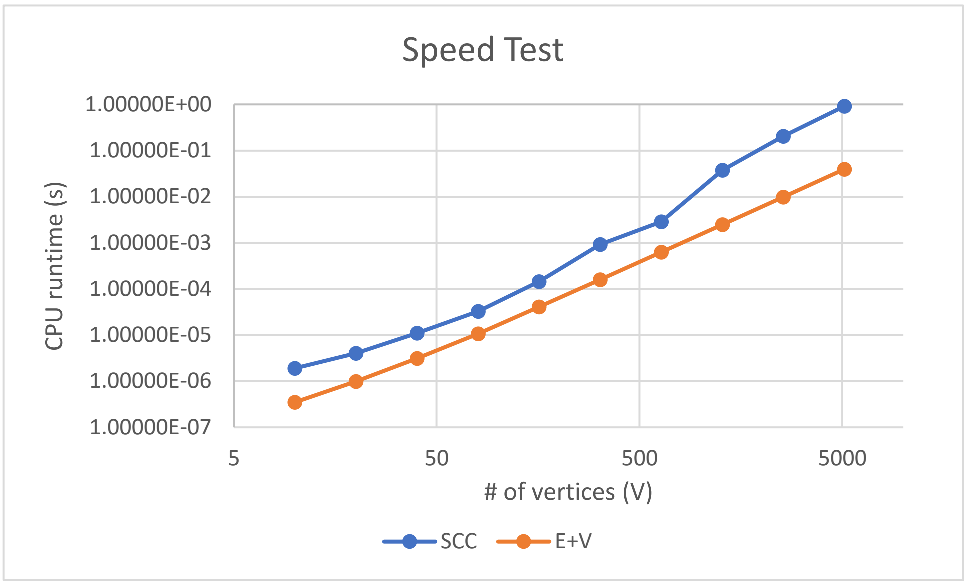
Workflow :

```
t = GetTime();                 // initialize time counter
SCC();
t = GetTime() - t;             // calculate CPU time
```

Results:

V	E	SubGroup	*SCC*
10	25	3	1.90735E-06
20	79	5	4.05312E-06
40	274	6	1.09673E-05
80	996	9	3.29018E-05
160	3926	14	1.44005E-04
320	15547	20	9.21011E-04
640	61786	30	2.88081E-03

1280	246515	43	3.73988E-02
2560	984077	63	2.03121E-01
5120	3934372	92	9.13190E-01

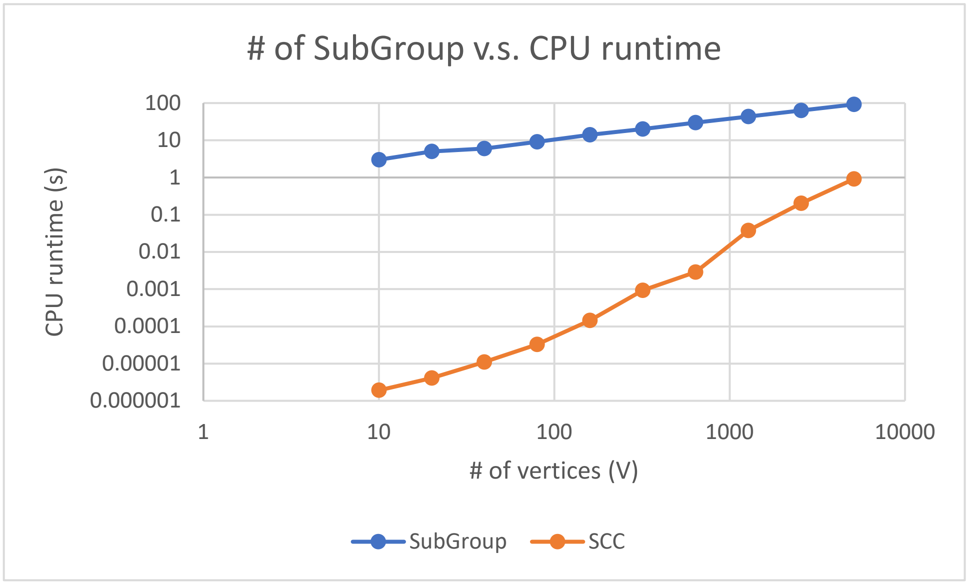


Observation:

1. Speed, Time complexity:

   The result matches my analysis precisely. The time complexity of *SCC* is **O(V+E)**.

2. Number of subgroups v.s. CPU runtime:



I plotted number of subgroups and CPU runtime into a graph out of curiosity. It clearly shows that they are not related at all.

Conclusions:

1. Time and space complexities of SCC:

	SCC
Time complexity	**O(V + E)**
Space complexity	**O(V + E)**

2. Actual runtim: **O(V + E)**