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.

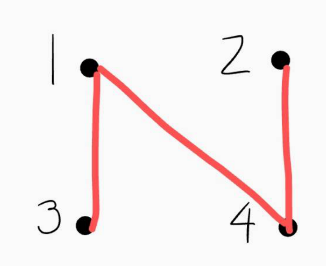
**Analysis:**

1. **Graph representation:**

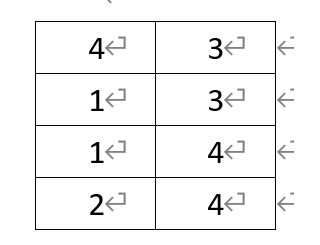
There are several ways of representing graphs and sets in programming. In this homework, we use 2 methods to do this.

* 1. **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



Table

* 1. 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.

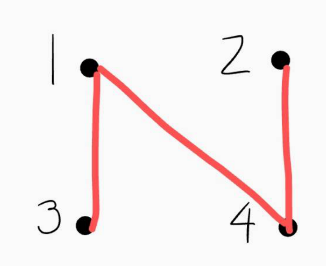
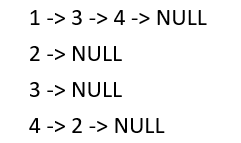


Fig 2

Table 2



1. **Strongly connected components (SCC):**

Given a directed graph , a strongly connected component is a maximal set of vertices such that for every pair of vertices and they are mutual reachable; that is, vertex is reachable from 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 and are the same.

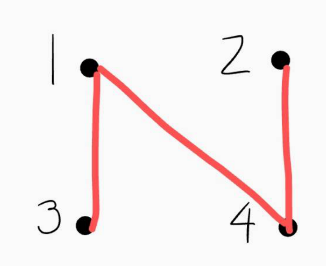


Fig 3

1. **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.

1. **DFS\_call (function that calls DFS) and DFS\_d:**
2. 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.

1. Algorithm:
2. // More sophisticated Depth first search starting from vertex v of graph G.
3. // Input: starting node v, graph G
4. // Output: f[|V|]: travel order.
5. Algorithm DFS\_d(G, v)
6. {
7. visited [v] := 1 ;
8. time := time + 1 ;
9. d [v] := time ;
10. **for** each vertex w adjacent to v in G **do** {
11. **if** (visited [w] = 0) then {
12. DFS\_d(w) ;
13. }
14. }
15. visited [v] := 2 ;
16. f [N - 1 - time] := v;
17. t := time + 1 ;
18. }
19. // Initialization and recursive DFS function call.
20. // Input: graph G, array l[V]: travel order
21. // Output: f[|V|]: finish time, steps: SCC index number SCCs, sub: SCCs num
22. Algorithm DFS\_Call(G, l)
23. {
24. sub := 0;
25. steps[0] := 0;
26. **for** v := 1 to n **do** { // Initialize to not visited and no predecessor.
27. visited [v] := 0 ;
28. f [v] := 0 ;
29. }
30. time := 0 ; // Global variable to track time.
31. **for** v := 1 to n **do** { // To handle forest case.
32. **if** (visited [l[v]] = 0) then {
33. DFS\_d(G, l[v]) ;
34. steps.push(time);    // push time in front of steps.
35. sub++;
36. }
37. }
38. }
39. 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.

1. 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.

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.

1. Space Complexity:

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

1. **SCC:**
2. Abstract:

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

1. Algorithm:
2. // To find the strongly connected components of the graph G = (V, E ).
3. // Input: graph G
4. // Output: strongly connected components.
5. Algorithm SCC(G )
6. {
7. **for** i in V **do** f[i] = i
8. DFS\_Call(G, f) ; // Perform DFS to get array f [1 : n].
9. DFS\_Call(GT, f) ; // Perform DFS on GT.
10. }
11. Proof of correctness: (infer from lec42.pdf)

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

Lemma 1

Let and be two distinct strongly connected components in a directed graph . If , and there is a path from to then cannot contain a path from to .

Proof 1:

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

Lemma 2:

Let and be two distinct strongly connected components in a directed graph . Suppose there is an edge , where and . Then, the largest finish time for each node in is larger than the largest finish time for each node in .

Proof 2:

There must be one DFS\_d that goes through edge , where the time for that DFS\_d is larger than the former one by 1, hence proved.

Lemma 3:

Let and be two distinct strongly connected components in a directed graph . Suppose there is an edge , where and . Then, the largest finish time for each node in is smaller than the largest finish time for each node in .

Proof 3:

Since , , 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() 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 , and . Because of lemma 2, for any strongly connected component other than , the largest finish time for each node in is larger than the largest finish time for each node in . Furthermore, when DFS\_call visits , all other descendants in are not visited. Now, using lemma 3, we can say that any edges in that leave must be to strongly connected components that have already been visited. Thus, no vertex in any strongly connected component other than will be a descendant of during DFS\_call(). Therefore, the vertices of DFS\_call() form exactly one strongly connected component. Finally, by induction, every tree of SCC in DFS\_call() are strongly connected components.

1. 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.

1. Space Complexity:

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

1. **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 algorithm1 times and record the CPU runtime for it.

Workflow :

1. t = GetTime();               // initialize time counter
2. SCC();
3. t = GetTime() - t;  // calculate CPU time

Results:

|  |  |  |  |
| --- | --- | --- | --- |
| V | E | SubGroup | *SCC* |
| 10 | 25 | 3 | 9.53674E-07 |
| 20 | 79 | 5 | 5.00679E-06 |
| 40 | 274 | 6 | 1.00136E-05 |
| 80 | 996 | 9 | 2.78950E-05 |
| 160 | 3926 | 14 | 7.20024E-05 |
| 320 | 15547 | 20 | 7.35998E-04 |
| 640 | 61786 | 30 | 7.35998E-04 |
| 1280 | 246515 | 43 | 3.81079E-02 |
| 2560 | 984077 | 63 | 2.04743E-01 |
| 5120 | 3934372 | 92 | 9.09799E-01 |

**Observation:**

1. Speed, Time complexity:

**Actual Speed (> means faster):**

***MaxSubArrayN > MaxSubArray> MaxSubArrayBF2 > MaxSubArrayBF*.**

The result matches my analysis precisely. The time complexity of *MaxSubArrayBF2* and *MaxSubArrayN* are **O(N^2)** and **O(N)**. Modify and deviate from maximum subarray approach made the algorithm did made the iteration depth for *MaxSubArrayBF2* lesser by N. By seeking peak and valley in an array did made the time complexity goes by O(N) only.

Overall, the implemented results meet my analysis.

**Conclusions:**

1. Time and space complexities of the 5 algorithms:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *MaxSubArrayBF* | *MaxSubArray* | *MaxSubArrayBF2* | *MaxSubArrayN* |
| Time | O(N^3) | O(N lg N) | O(N^2) | O(N) |
| Space | O(N) | O(N) | O(N) | O(N) |

1. Actual runtime comparison:

**Speed (> means faster):**

***MaxSubArrayN > MaxSubArray> MaxSubArrayBF2 > MaxSubArrayBF*.**

1. It goes faster when not using the maximum subarray property.