Algorithms: Strongly Connected Components

2017-18570 Sungchan Yi

Dept. of Computer Science and Engineering

May 15th, 2019

Contents

1 Introduction

In this homework, we implement a program that finds the strongly connected components (SCC) of a given directed graph using Kosaraju's Algorithm with C++ language. We must be able to find the SCCs when the directed graph is given in a form of adjacency matrix, or adjacency list. The following are the restrictions on given inputs.

- First line contains number of vertices V and number of edges E.
- The next E lines will contain information about edges.
- It will be given as u v, denoting that there exists a directed edge from u to v.

The program should output each SCC in a line such that the vertices in each SCC are sorted, and the lines appear in the output are in lexicographic order.

2 Implementation

The algorithm was implemented similarly to the algorithm in the textbook.

STRONGLY-CONNECTED-COMPONENTS(G) (Kosaraju)

1. Call DFS(G) and compute finishing times $u.f$ for each vertex u.
2. Compute G^T.
3. Call DFS(G^T), but consider the vertices in order of decreasing $u.f$.
4. Sort and output the vertices of each tree in the depth-first forest formed in 3 as a separate SCC

DFS(G) and computing G^T takes $O(V+E)$ time. Thus the overall complexity is $O(V+E)$. (Without considering the sorting time for each SCC)

3 Experiments

3.1 Enviornment

The following is the test environment.

- OS: Ubuntu 18.04.2 LTS
- CPU: Intel Core i5-6200U CPU @ 2.30GHz × 4
- g++: 7.4.0
- RAM: 8GB

Here is how the test was done. Run the algorithm, measure the running time without the sorting of vertices and SCCs.

3.2 Results

3.2.1 Adjacency List Implementation

Test File	V	E	Running Time (μs)
`in1.txt`	250	600	343
`in2.txt`	500	1200	660
`in3.txt`	750	1800	1021
`in4.txt`	1000	2400	1350

3.2.2 Adjacency Matrix Implementation

Test File	V	E	Running Time (μs)
in1.txt	250	600	3545
in2.txt	500	1200	12780
in3.txt	750	1800	30825
in4.txt	1000	2400	63320

4 Analysis

Consider running time y as $\alpha V + \beta E + \gamma$, and use least squares approximation. Here is the python code for the approximation.

```python
import numpy as np

# A is fixed
a = np.array([
    [250, 600, 1], [500, 1200, 1], [750, 1800, 1], [1000, 2400, 1]
])

# Running Time
b = np.array([
    [343], [660], [1021], [1350]
])

# Moore-Penrose Inverse
aplus = np.linalg.pinv(a)

# Calculate least-square minimum length solution
xplus = np.matmul(aplus, b)

print(xplus)

# Calculate R^2 value
bbar = np.mean(b)
sstot = np.linalg.norm(b - bbar) ** 2
ssres = np.linalg.norm(np.matmul(a, xplus) - b) ** 2
r2 = 1 - ssres/sstot
print(r2)
```

4.1 Adjacency List Implementation

For the adjacency list implementation, we got

$$y = 0.2001V + 0.4803E - 2$$

with $R^2 = 0.9994$

4.2 Adjacency Matrix Implementation

For the matrix implementation, we got

$$y = 11.6787V + 28.0289E - 21725$$

with $R^2 = 0.9344$. But this seemed incorrect, since it takes $O(V^2)$ time when transposing the graph. So we modified the prediction to be $O(V^2 + E)$, and got

$$y = 0.0930V^2 - 15.5633E + 7350$$

with $R^2 = 0.9992$. Another thing to check is the negative coefficient in $-15.5633E$. This implies that the running time is faster when there are more edges. But this problem is negligible, since the test cases had $E \leq V^2$.[1] Running more tests would give a correct result.

5 Conclusion

For this assignment, we implemented an algorithm to find the strongly connected components using Kosaraju's Algorithm. The adjacency list implementation had time complexity $O(V + E)$ as expected, and matrix implementation had $O(V^2 + E)$ as time complexity, also as expected.

[1] And thus V^2 is the leading term, and we should have run the least squares approximation for $y = \alpha V^2$. The negative coefficients for lower order terms do not change the overall time complexity.