Calculation of Strongly Connected Components in Graphs

Special Mathematics Lecture: Graph Theory (Spring 2024)

Yijiang Liu

July 19, 2024

As mentioned in the class, strongly connected components (SCCs) are used to describe the maximal strongly connected subgraphs of a directed graph. They are commonly used in social network analysis, optimizing web crawlers, and optimizing recursive and cyclic structures in compilers. This report mainly focuses on the algorithms for solving SCCs and their proofs.

1 Tarjan's algorithm

Tarjan's algorithm is an efficient method for finding SCCs in a directed graph. Specifically, Tarjan's algorithm assigns discovery times and low link values to each node, where the low link value represents the smallest discovery time reachable from the node, including itself and its descendants. By traversing adjacent nodes and recursively performing DFS, the algorithm updates the low link values. When the DFS for a node is complete, if its low link value equals its discovery time, it indicates the root of an SCC, with all nodes in the stack up to this node forming a strongly connected component. Tarjan's algorithm runs in linear time, $O(V + E)$, where V is the number of vertices and E is the number of edges, making it highly efficient for identifying all SCCs in a graph. The algorithm was proposed by Robert Tarjan in 1972, it is logically highly consistent with the SCC-finding algorithm introduced in the lecture.

2 Kosaraju's Algorithm

Kosaraju's algorithm is another commonly used method for computing strongly connected components (SCCs) in a directed graph. It is generally believed that S. Rao Kosaraju first proposed this algorithm in an unpublished paper in 1987. A similar algorithm was documented by Micha Sharir in his book published in 1981. Both implementations rely heavily on the same property of graphs: the strongly connected components remain unchanged when the graph is transposed. Kosaraju's algorithm employs two depth-first search (DFS) operations and can find SCCs in linear time complexity.

2.1 Algorithm Steps:

1. **First DFS on Original Graph:** Perform a DFS traversal on the original graph G. The order of traversal for the first DFS does not really matter and can

follow the vertex indices. We record the finish time of each vertex. This finish time sequence constitutes a topological order, where vertices finished first are at the front and those finished last are at the back.

2. **Graph Transposition:** Transpose the original graph G to obtain G^T, where the vertices remain the same but the direction of all edges in G are reversed. Specifically, for each edge (u,v) in G, add an edge (v,u) in G^T.
3. **Second DFS on Transposed Graph:** Perform a DFS on G^T in the reverse order of the finish times recorded in the first DFS (i.e., start with the vertex with the highest finish time). Each DFS tree obtained during this traversal represents an SCC in the original graph G.

2.2 Algorithm Implementation:

The algorithm typically utilizes a stack to implement DFS. During the first DFS traversal, we visit the graph G and push each vertex onto a stack after all its descendant vertices have been fully explored. This ensures that vertices with no outgoing edges are at the bottom of the stack, while vertices with many outgoing edges (potentially connecting to other SCCs) are at the top. Since SCCs remain unchanged in the transposed graph, the second DFS traversal involves popping vertices from the stack and performing DFS on G^T. The order of vertices in the stack ensures that we start the DFS from vertices that can reach many others. Each DFS tree formed in this traversal corresponds to an SCC.

2.3 Example:

This report will use the following graph G as an example to illustrate Kosaraju's algorithm. The graph consists of ten vertices, labeled A to J. Kosaraju's algorithm will be applied to determine all the SCCs in this graph.

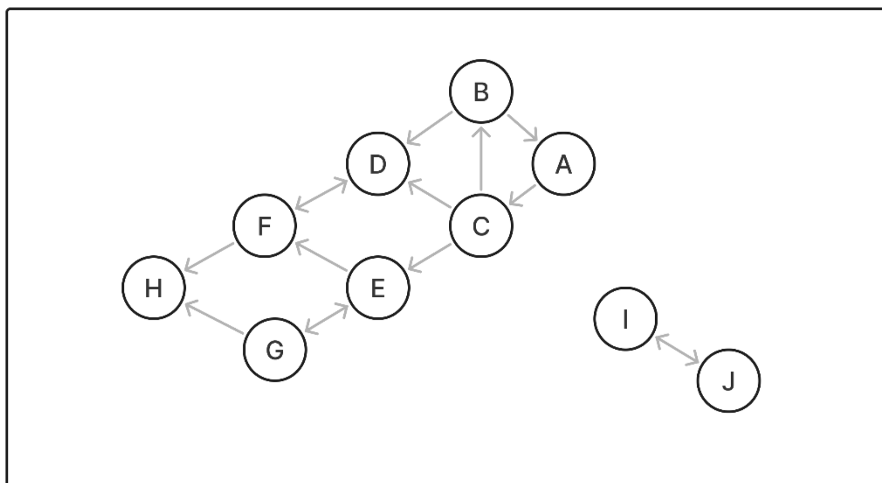

Figure 1 Graph G

	A	B	C	D	E	F	G	H	I	J
A	0	0	1	0	0	0	0	0	0	0
B	1	0	0	1	0	0	0	0	0	0
C	0	1	0	1	0	0	0	0	0	0
D	0	0	0	0	0	1	0	0	0	0
E	0	0	0	0	0	1	1	0	0	0
F	0	0	0	1	0	0	0	1	0	0
G	0	0	0	0	1	0	0	1	0	0
H	0	0	0	0	0	0	0	0	0	0
I	0	0	0	0	0	0	0	0	0	1
J	0	0	0	0	0	0	0	0	1	0

Figure 2 Using the Adjacency Matrix Method to Represent Graph G

First, perform a depth-first search on the graph G. Without loss of generality, I will start the traversal from vertex A. The resulting time sequence is shown in the diagram below, where each vertex records its initial visit time and its finish time.

A	B	C	D	E	F	G	H	I	J
1, 16	3, 10	2, 15	4, 9	11, 14	5, 8	12, 13	6, 7	17, 20	18, 19

Figure 3 Vertex with it's initial time and finish time

This allows us to sort the vertices by their finish times, yielding the sequence s={H,F,D,B,G,E,C,A,J,I}.

Now, transpose the original graph, and represent the result using an adjacency matrix as follows:

	A	B	C	D	E	F	G	H	I	J
A	0	1	0	0	0	0	0	0	0	0
B	0	0	1	0	0	0	0	0	0	0
C	1	0	0	0	0	0	0	0	0	0
D	0	1	1	0	0	1	0	0	0	0
E	0	0	0	0	0	0	1	0	0	0
F	0	0	0	1	1	0	0	0	0	0
G	0	0	0	0	1	0	0	0	0	0
H	0	0	0	0	0	1	1	0	0	0
I	0	0	0	0	0	0	0	0	0	1
J	0	0	0	0	0	0	0	0	1	0

Figure 4 Using the Adjacency Matrix Method to Represent Graph G^T

According to the reverse order of s, which is the order of finish times from largest to smallest (i.e., I, J, A, C, E, G, B, D, F, H), we perform the second DFS as follows:

Starting from vertex I: mark I as visited, then visit its neighbor J.
Starting from vertex J: mark J as visited, J's neighbor I has already been fully visited.
Obtain SCC: {I, J}

Starting from vertex A: mark A as visited, then visit its neighbor B.
Starting from vertex B: mark B as visited, then visit its neighbor C.
Starting from vertex C: mark C as visited, C's neighbor A has already been fully visited.
Obtain SCC: {A, B, C}

Starting from vertex E: mark E as visited, then visit its neighbor G.
Starting from vertex G: mark G as visited, G's neighbor E has already been fully visited.
Obtain SCC: {E, G}

Starting from vertex D: mark D as visited, then visit its neighbor F.
Starting from vertex F: mark F as visited, F's neighbor D has already been fully visited.
Obtain SCC: {D, F}

Starting from vertex H: mark H as visited.
Obtain SCC: {H}

Through the second DFS, we obtained the following strongly connected components S := {s_1, s_2, s_3, s_4, s_5}, which are {{I, J}, {A, B, C}, {E, G}, {D, F}, {H}}.

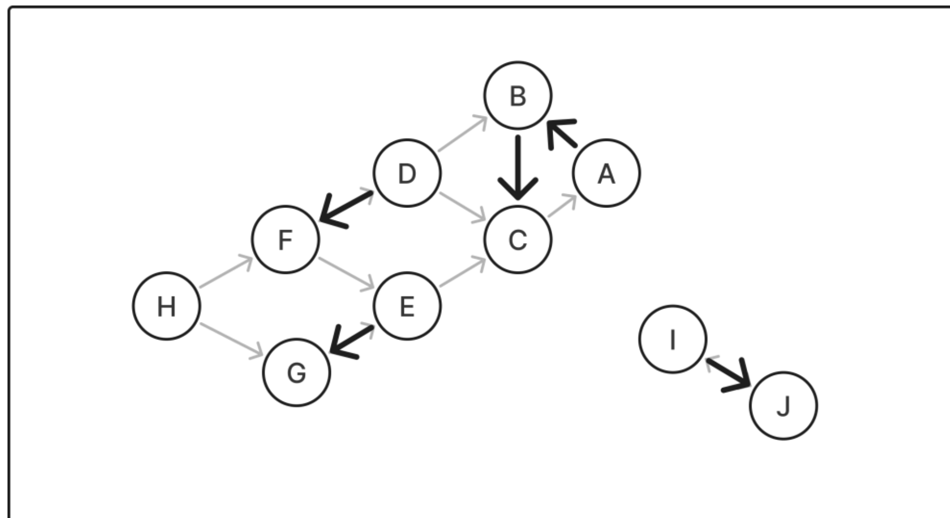

Figure 5 DFS on the transposed graph G^T

2.4 Correctness Proof:

① **<u>Proving the subgraph obtained by the algorithm is strongly connected:</u>**
For any two vertices X and Y in the graph, assume there is a path from X to Y in the transposed

graph. This implies there is a path from Y to X in the original graph.

According to the order specified by the algorithm for the second DFS, X is guaranteed to be popped from the stack later than Y. This means that in the DFS tree of the original graph G (denoted as T), X must be an ancestor of Y, and there must be a path from X to Y in the original graph.

Contradiction proof:

If X is not an ancestor of Y in the DFS tree T of the original graph, and given that X is popped from the stack later than Y, Y cannot be an ancestor of X. This contradicts the existence of a path from Y to X in the original graph. Therefore, the assumption is incorrect.

Hence, there exist paths from X to Y and from Y to X in the original graph, proving its strong connectivity.

② **Proving the algorithm finds the Maximum strongly connected subgraph (SCCs):**

This means proving that the strongly connected subgraphs obtained by the algorithm are maximal, i.e., there are no strongly connected vertices u and v that are not included in the same SCC by the algorithm.

Contradiction proof:

Suppose there exist such vertices M and N.

When performing DFS on G^T , assume u is visited first.

For the last vertex P on the path from M to N that is visited by DFS, if P has already been visited, then during that DFS traversal, vertex N would also be visited through P. This implies that N is visited earlier than M, contradicting our assumption.

Otherwise if P has not been visited before, then it will surely be visited in this round of DFS, thus continuing to visit vertex N.

Therefore, the strongly connected subgraphs obtained by the algorithm are maximal strongly connected subgraph, which also means SCCs.

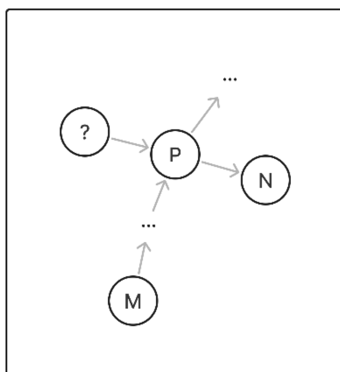

Figure 6 Possible situation

Reference

[1]ScienceBlogs. Computing Strongly Connected Components. https://scienceblogs.com/goodmath/2007/10/30/computing-strongly-connected-c

[2]S. Richard. Graph theory. http://www.math.nagoya-u.ac.jp/~richard/teaching/s2024/Graph.pdf, 2024.

[3]Wikipedia. Kosaraju's algorithm — Wikipedia, the free encyclopedia.

[4]Wikipedia. Tarjan's strongly connected components algorithm — Wikipedia, the free encyclopedia.

[5] https://blog.csdn.net/u011815404/article/details/86703264