Finding Strongly Connected Components

Yufei Tao

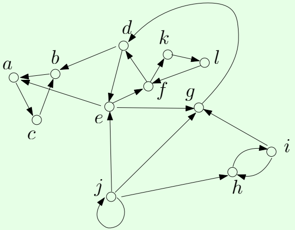
Department of Computer Science and Engineering
Chinese University of Hong Kong

Recall that we have applied DFS to solve two non-trivial problems: cycle detection and topological sort. Today we will see yet another interesting problem that can be elegantly solved by this remarkable algorithm: finding strongly connected components.

"Strongly Connected"

Let $G = (V, E)$ be a directed graph.
Two distinct vertices $u, v \in V$ are **strongly connected** if there is a path from u to v, and also a path from v to u.

Example:



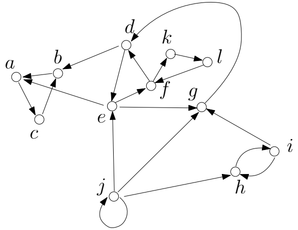
d and l are strongly connected; so are a and c.
But a and d are not.

Strongly Connected Equivalent Classes

A **strongly connected equivalent class** (SCEC) of G is a subset S of V such that

- Any two distinct vertices $u, v \in S$ are strongly connected.
- S is maximal in the sense that we cannot put any more vertex into S without violating the above property.

Example



- $\{a, b, c\}$ is an SCEC.
- $\{a, b, c, d\}$ is not an SCEC.
- $\{d, e, f, k, l\}$ is not an SCEC (because we can still add vertex g).
- $\{e, d, f, k, l, g\}$ is an SCEC.

SCECs are Disjoint

Theorem: Suppose that S_1 and S_2 are two different SCECs of G. Then, $S_1 \cap S_2 = \emptyset$.

Proof: Assume that there is a vertex v in both S_1 and S_2. Then, for any vertex $u_1 \in S_1$ and any vertex $u_2 \in S_2$:

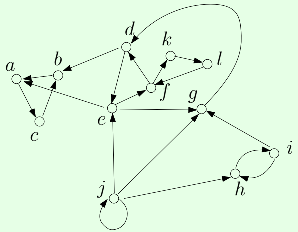
- There is a path from u_1 to u_2: we can first go from u_1 to v within S_1, and then from v to u_2 within S_2.
- Likewise, there is also a path from u_2 to u_1.

Hence, neither S_1 nor S_2 is maximal, contradicting the fact that they are SCECs. $\square$

The Strongly Connected Problem Problem

Problem: Given a directed graph $G = (V, E)$, we want to divide V into disjoint subsets, each of which is an SCEC.

Example:

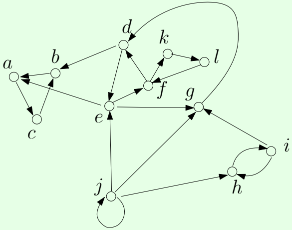


We should output: $\{a, b, c\}$, $\{d, e, f, g, k, l\}$, $\{h, i\}$, and $\{j\}$.

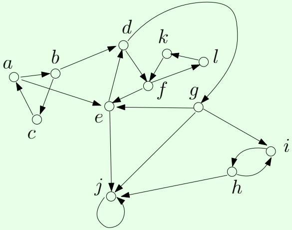
Algorithm

Step 1: Obtain the reversed graph G^R by reversing the directions of all the edges in G.

Example:



Input graph



Reversed graph

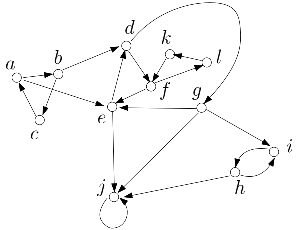
Algorithm

Step 2: Perform DFS on G^R, and obtain the sequence L^R that the vertices in G^R turn black (i.e., whenever a vertex is popped out of the stack, append it to L^R).

Obtain L as the reverse order of L^R.

Example

Reverse graph G^R:



We may perform DFS starting from any vertex. When a restart is needed, we may do so from any vertex that is still white. The following is a possible order that the vertices are discovered: $f, l, k, e, j, d, g, i, h, a, b, c$.

The corresponding turn-black sequence is
$L^R = (k, l, j, h, i, g, d, e, f, c, b, a)$.
Hence, $L = (a, b, c, f, e, d, g, i, h, j, k, l)$.

Algorithm

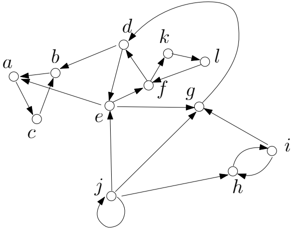
Step 3: Perform DFS on the original graph G by obeying the following rules:

- **Rule 1:** Start the DFS at the first vertex of L.
- **Rule 2:** Whenever a restart is needed, start from the first vertex of L that is still white.

Output the vertices in each DFS-tree as an SCEC.

Example

From the last step, we have $L = (a, b, c, f, e, d, g, i, h, j, k, l)$.
The original graph G:



Start DFS from a, which finishes after discovering $\{a, c, b\}$.
Restart from f, which finishes after discovering $\{f, k, l, d, e, g\}$
Restart from i, which finishes after discovering $\{i, h\}$
Restart from j, which finishes after discovering $\{j\}$

The DFS returns 4 DFS-trees, whose vertex sets are shown as above.
Each vertex set constitutes an SCEC.

Time Analysis

The overall execution time is $O(|V| + |E|)$.

Next, we will prove that the algorithm is correct.

The proof is based on the **white path theorem** on DFS. This is an important theorem which should have been taught in the "data structure" course.

If you need to review this theorem and/or its proof, you can refer to the course homepage of Prof. Yufei Tao's offering of the course CSCI2100: *www.cse.cuhk.edu.hk/∼taoyf/course/2100/18-fall*.

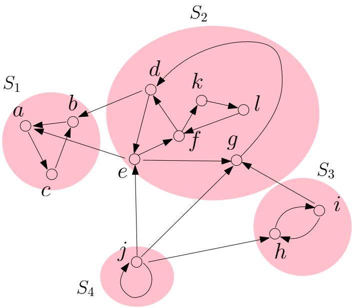
SCEC Graph

Let G be the input directed graph, with SCECs $S_1, S_2, ..., S_t$ for some $t \geq 1$.

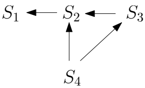
Let us define a SCEC graph G^{EC} as follows:

- Each vertex in G^{EC} is a distinct SCEC in G.
- Consider two vertices (a.k.a. SCECs) S_i and S_j ($1 \leq i, j \leq t$). G^{EC} has an edge from S_i to S_j if and only if
  - $i \neq j$, and
  - There is a path in G from a vertex in S_i to a vertex in S_j.

Example



SCC Graph



SCEC Graph

Lemma: G^{EC} is a DAG.

Proof: Suppose that there is a cycle in G^{EC}, which must involve at least 2 SCECs—say S_i, S_j—as no vertex in G^{EC} has an edge to itself. Then, any vertex in S_i is reachable from any vertex in S_j, and vice versa. This violates the fact that S_i, S_j are SCECs (violating maximality). $\square$

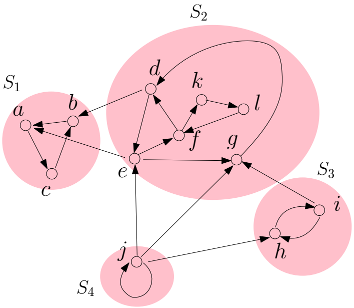
SCEC Graph

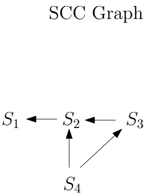
Define an SCEC as a sink SCEC if it has no outgoing edge in G^{EC}.

Lemma: There must be at least one sink SCEC in G^{EC}.

Proof: Since G^{EC} is a DAG, it admits a topological order. The last vertex of the topological order cannot have any outgoing edges. $\square$

Example





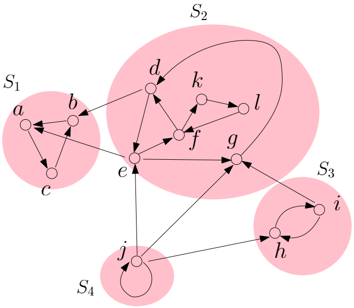
S_1 is a sink vertex.

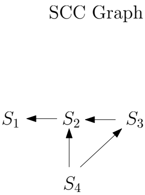
DFS in a Sink SCEC

Lemma: Let S be a sink SCEC of G^{EC}. Suppose that we perform a DFS starting from any vertex in S. Then the first DFS-tree output must include all and only the vertices in S.

Proof: Let $v \in S$ be the starting vertex of DFS. By the white path theorem of DFS, the DFS-tree must include all the vertices that v can reach. These are exactly the vertices in S. $\square$

Example





Performing DFS from any vertex in S_1 will discover S_1 as the first SCEC.

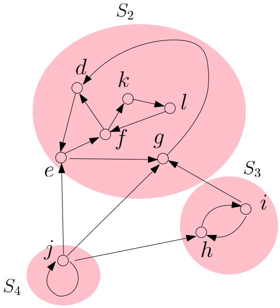
Finding SCECs—The Strategy

The previous lemma suggests the following strategy for finding all the SCECs:

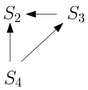
1. Performing DFS from any vertex in a sink SCEC S.
2. Delete all the vertices of S from G, as well as their edges.
3. Accordingly, delete S from G^{EC}, as well as its edges.
4. Repeat from Step 1, until G is empty.

Example

After deleting S_1, we have:



SCC Graph

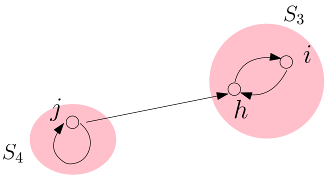


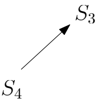
Now, S_2 becomes the sink SCEC. Performing DFS from any vertex in S_2 discovers S_2 as the second SCEC.

Example

After deleting S_2, we have:

SCC Graph



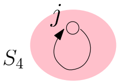


Now, S_3 becomes the sink SCEC. Performing DFS from any vertex in S_3 discovers S_3 as the third SCEC.

Example

After deleting S_3, we have:

SCC Graph



S_4

Now, S_4 becomes the sink SCEC. Performing DFS from any vertex in S_4 discovers S_4 as the last SCEC.

A Property of the Ordering L

Next, we will show that this is exactly the strategy taken by our algorithm. In particular, we resort to the ordering L to correctly identify the sequence of sink SCECs!

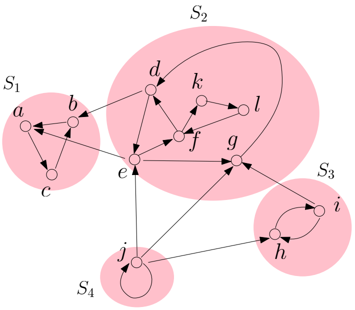
Lemma: Let S_1, S_2 be SCECs such that there is a path from S_1 to S_2 in G^{EC}. In the ordering of L, the earliest vertex in S_2 must come before the earliest vertex in S_1.

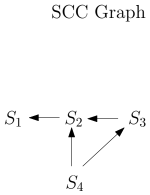
Proof: Let $X_1, X_2, ..., X_t$ be a path on G^{EC} such that $X_1 = S_1$ and $X_t = S_2$. Consider the DFS performed on the reversed graph G^R, Let v be the first vertex discovered among all the vertices of $X_1 \cup X_2 \cup ... \cup X_t$ in this DFS.

By the white path theorem, at the moment when v is discovered by DFS, there is a white path in G^R from v to all the vertices in X_1. In other words, all the vertices in X_1 must turn black no later than v in the DFS.

Let u be the vertex in S_2 that turns black the last. It follows from the previous paragraph that all the vertices in X_1 must turn black before u. Therefore, u is behind all the vertices of S_1 in L^R, which indicates that u is before all the vertices of S_1 in L. $\square$

Example





Recall that we obtained earlier $L = (a, b, c, f, e, d, g, i, h, j, k, l)$. The red vertices a, f, i, j are, respectively, the earliest vertex in L of S_1, S_2, S_3, and S_4.

This essentially completes the proof of the correctness of our SCEC algorithm.

Did we delete any vertices from G? In fact, we did, as far as DFS is concerned. To see this, recall that, after a vertex is popped out of the stack, DFS colors it black. These vertices are never touched again, and hence, effectively deleted.