Finding Strongly Connected Components

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong
We just can’t get enough of the beautiful algorithm of DFS!

In this lecture, we will use it to solve a problem—finding strongly connected components—that seems to be rather difficult at first glance. As you probably have guessed, the algorithm is once again very simple, and runs DFS only twice.
Strongly Connected Component

Let $G = (V, E)$ be a directed graph.

A strongly connected component (SCC) of $G$ is a subset $S$ of $V$ such that

- For any two vertices $u, v \in S$, it must hold that:
  - There is a path from $u$ to $v$.
  - There is a path from $v$ to $u$.
- $S$ is maximal in the sense that we cannot put any more vertex into $S$ without violating the above property.
Example

Consider the following graph:

- \{a, b, c\} is an SCC.
- \{a, b, c, d\} is not an SCC.
- \{d, e, f, k, l\} is not an SCC (because we can still add vertex g).
- \{e, d, f, k, l, g\} is an SCC.
**Theorem:** Suppose that $S_1$ and $S_2$ are both SCCs 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 SCCs. □
The Problem of Finding SCCs

Given a directed graph $G = (V, E)$, the goal of the finding strongly connected components problem is to divide $V$ into disjoint subsets, each of which is an SCC.
The goal is to output the following 4 SCCs: \{a, b, c\}, \{d, e, f, g, k, l\}, \{h, i\}, and \{j\}. 
Algorithm

**Step 1:** Obtain the reverse graph $G^R$ by reversing the directions of all the edges in $G$. 
Example

Input graph

Reverse graph

Finding Strongly Connected Components
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, $L = (a, b, c, f, e, d, g, i, h, j, k, l)$. 
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 SCC.
Example

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

![Graph Diagram]

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 SCC.
Time Analysis

Steps 1 and 2 obviously require only $O(|V| + |E|)$ time.

Regarding Step 3, the DFS itself takes $O(|V| + |E|)$ time, but we still need to discuss the time to implement Rule 2. Namely, whenever DFS needs a restart, how do we find the first white vertex in $L$ efficiently? This will be left as an exercise—where you will be asked to do so in $O(|V|)$ total time.

Hence, the overall execution time is $O(|V| + |E|)$. 
Next, we will prove that the algorithm is correct. Once again, the correctness is due to the remarkable properties of DFS.
Let $G$ be the input directed graph, with SCCs $S_1, S_2, \ldots, S_t$ for some $t \geq 1$.

Let us define a SCC graph $G^{SCC}$ as follows:

- Each vertex in $G^{SCC}$ is a distinct SCC in $G$.
- Consider two vertices (a.k.a. SCCs) $S_i$ and $S_j$ ($1 \leq i, j \leq t$). $G^{SCC}$ 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

Finding Strongly Connected Components
Lemma: \( G^{SCC} \) is a DAG.

Proof: Suppose that there is a cycle in \( G^{SCC} \), which must involve at least 2 SCCs—say \( S_i, S_j \)—as no vertex in \( G^{SCC} \) 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 SCCs (violating maximality). \( \square \)
Define an SCC as a sink SCC if it has no outgoing edge in $G^{SCC}$.

**Lemma:** There must be at least one sink SCC in $G^{SCC}$.

**Proof:** Since $G^{SCC}$ is a DAG, it admits a topological order. The last vertex of the topological order cannot have any outgoing edges.
$S_1$ is a sink vertex.
Lemma: Let $S$ be a sink SCC of $G^{SCC}$. 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 $S$. \qed
Performing DFS from any vertex in $S_1$ will discover $S_1$ as the first SCC.
Finding SCCs—The Strategy

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

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

After deleting $S_1$, we have:

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

After deleting $S_2$, we have:

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

After deleting $S_3$, we have:

Now, $S_4$ becomes the sink SCC. Performing DFS from any vertex in $S_4$ discovers $S_4$ as the last SCC.
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 SCCs!
**A Property of the Ordering $L$**

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

**Proof:** Left to you as an exercise.
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 SCC algorithm.

You may want to ask: but we never delete any vertices from $G$! In fact, we did, as far as DFS is concerned. To see this, recall that DFS colors all the “done” vertices black. These vertices are never touched again, and hence, effectively deleted.