Graphs : Strongly Connected Components via DFS[1]

1 Strongly Connected Components (SCCs) using DFS

We now see that the *strongly connected components* of a graph G can be found in *linear* $O(n+m)$ time. This is a truly surprising algorithm which really illustrates the power of DFS. Let's try to first sketch the main ideas behind the algorithm, and then subsequently give the final description and analysis. To do so, consider the graph in Figure 1.

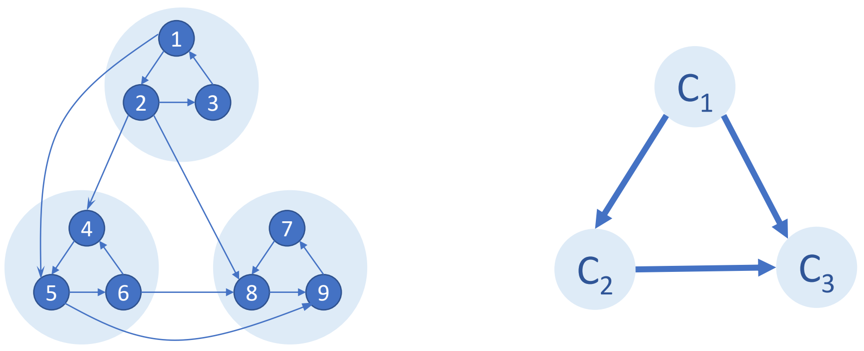


Figure 1: Illustrative example for SCC.

In the graph on the left, there are three strongly connected components marked in light blue circles. The graph on the right is one whose vertices are these three components, and we have an edge between two components (for instance from C_1 to C_2) if and only if there is an edge (u, v) in the original graph with $u \in C_1$ and $v \in C_2$. Note there could be, and in this example there are, multiple such edges. We require that there be at least one. In general, given a graph G as on the left, then the graph on the right is called G^{scc}; note we don't have this graph up front but is useful for analysis and designing the algorithm.

Claim 1. G^{scc} is a DAG.

Proof. Suppose not, and suppose there was a cycle $(C_1, C_2, \ldots, C_k, C_1)$ in G^{scc}. This means there are vertices x_i, y_i in C_i (possibly x_i and y_i are the same) such that (x_i, y_{i+1}) is an edge in G, and (x_k, y_1) is also an edge in G. But then, we argue next, $C_1 \cup \cdots \cup C_k$ should have been one strongly connected component, which would be a contradiction. Take any two vertices u and v in the union. Say $u \in C_i$ and $v \in C_j$ where $i \leq j$ without loss of generality (otherwise, we swap names). We now show a path from u to v. First, we go from u to x_i which is possible since C_i is strongly connected, then we take the (x_i, y_{i+1}) edge to y_{i+1}, and from there to x_{i+1} (since C_{i+1} is strongly connected), and so on and so forth till we reach y_j, upon reaching which, we take the path from y_j to v (which again exists since C_j is strongly connected). □

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 19th Mar, 2022*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

Before we move on to discovering the SCCs, let us see why the algorithm for undirected graphs is not enough. Recall what we did for undirected graphs; we ran DFS on G in ***any arbitrary*** order and returned the connected components of the forest. Why doesn't it work? Well, in the graph in Figure 1 consider what happens when we run DFS from the vertex 1. You see that *all* the vertices are reachable from 1 and thus end up in the tree rooted at 1. The resulting vertices are not strongly connected. To stress why this is not an issue in undirected graphs note that in undirected graphs if there is a path from a vertex 1 to a set of vertices S, then there is a path from any vertex in S to 1 as well. This is patently false in directed graphs.

An Encouraging Idea. Suppose that in the graph in Figure 1, we ran DFS from vertex number 9. Then, we would definitely discover all the vertices that 9 can reach. But these are precisely the ones in C_3, the strongly connected component connecting 9. Why is this? This is because, there is no edge which starts from inside C_3 and goes outside. That is, because C_3 is a *sink* component of G^{scc}. But this is wonderful; there is some vertex from which if we start DFS we get at least one strongly connected component. Let us make this (finding one strongly connected component) our goal for now.

From our understanding of topological ordering in DAGs, we know that the vertex with the smallest $\mathsf{last}[v]$ is a sink vertex in a DAG. Perhaps, we could conjecture something similar for a general graph: in any graph G and any DFS run, the vertex with the smallest $\mathsf{last}[v]$ must lie in a sink component of G^{scc}. If that is the case, then we could get what we want. *Unfortunately, this is not true.* Consider the graph in Figure 1 again, with DFS being run in the $\{1, 2, \ldots, 9\}$ order, and the adjacency lists also being visited in this order. We see that vertex 3 has the smallest $\mathsf{last}[]$, and indeed 3 lies in a source component of G^{scc}.

Remark: ***A philosophical interlude.*** *In research, we often think we have a good understanding of objects, and this leads us to make some conjectures. Just like we did above. And often they are wrong. I'll not lie – disappointment is usually the first response. But what really defines a researcher is resilience. Counterexamples are the world's ways of telling us, "Your understanding was incomplete. Refine them. Think harder." And when we do get back to the drawing board, or square one, the world often rewards us with* epiphanies.

Epiphany 1. Although the vertex with the *smallest* $\mathsf{last}[v]$ may not be in a SINK component of G^{scc}, it is in fact true that the vertex with the *largest* $\mathsf{last}[v]$ *does indeed* lie in the SOURCE component of G^{scc}. Again going back to the example in G^{scc}, we see that the vertex 1 has the *largest* last, and it is in the source component C_1 of G^{scc}.
In fact, more is true. For any component $C \in G^{\mathsf{scc}}$, define

$$f(C) = \max_{v \in C} \mathsf{last}[v]$$

That is, $f(C)$ is the largest last in that component.

Lemma 1. If (C_i, C_j) is an edge in G^{scc}, then $f(C_i) > f(C_j)$.

Before we prove the above lemma, let us see why it implies that the vertex with the largest last must lie in a source component of G^{scc}. Suppose x is the vertex with the largest last, and suppose it lies in component C_j. Clearly, $f(C_j) = \mathsf{last}[x]$, for x is the largest last vertex. Now if C_j were not a source component, there would be some component C_i with (C_i, C_j) an edge in G^{scc}. The above lemma would imply $f(C_i) > f(C_j) = \mathsf{last}[x]$. That is, there is a vertex $y \in C_i$ with $\mathsf{last}[y] > \mathsf{last}[x]$. That contradicts the choice of x. Thus, x must lie in a source component. Let us now prove the lemma.

Proof. First we make a claim about strongly connected components.

Claim 2. For any strongly connected component C if $x \in C$ has the largest last, then it also has the smallest first. In particular, x's interval contains all the intervals of every other vertex in C.

Proof. Suppose not. Suppose $y \in C$, $y \neq x$ has the smallest first. Since C is strongly connected, there is a path from y to x. y has the smallest first among all vertices in this path, so by the *path property*, it must have the the largest last among all vertices in this path. In particular, $\mathsf{last}[y] > \mathsf{last}[x]$. Contradiction. See figure below for illustration ☐

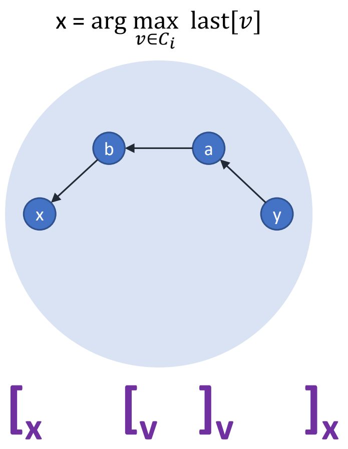


Figure 2: *If $y \neq x$ is the vertex with the smallest* first *in C, then (a) there is some path y, a, b, x from y to x, and (b) since* $\mathsf{first}[y] < \mathsf{first}[a], \mathsf{first}[b], \mathsf{first}[x]$, *by the path property says* $\mathsf{last}[y] > \mathsf{last}[x]$. *Therefore,* $\mathsf{first}[x] < \mathsf{first}[v] < \mathsf{last}[v] < \mathsf{last}[x]$ *for all $v \in C$.*

The proof of the lemma is as follows. Let x be the vertex in C_i with the largest last and y be the vertex in C_j with the largest last. Thus, $f(C_i) = \mathsf{last}[x]$ and $f(C_j) = \mathsf{last}[y]$. For the sake of contradiction, suppose $\mathsf{last}[x] < \mathsf{last}[y]$. By the Nested Interval Property, either (a) $\mathsf{first}[y] < \mathsf{first}[x]$, that is, x's interval is completely contained in y's interval, or (b) $\mathsf{last}[x] < \mathsf{first}[y]$, that is x's interval is disjoint and lies before y's interval.

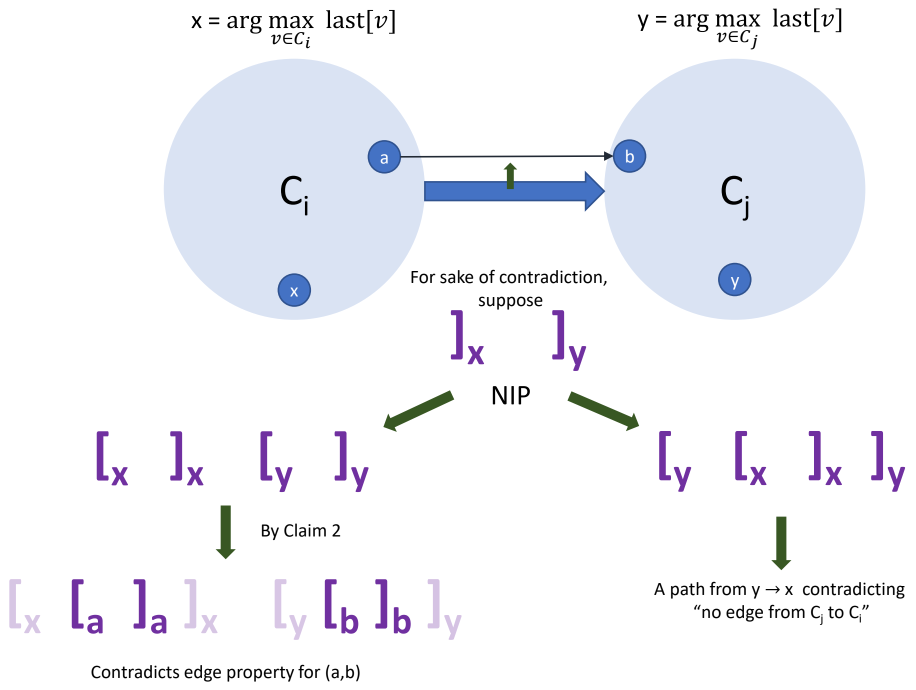
We will reach a contradiction in both cases. Case (a) is easy: if x's interval is completely contained in y's interval, then there is a path from y to x in the DFS forest. In particular, that would imply an edge from C_j to C_i in the G^{scc} contradicting the DAG nature of G^{scc}.

In Case (b), x's interval finishes before y's interval. By the claim above, this means that the interval of *every* vertex in C_i finishes before the interval of *any* vertex in C_j starts. Now since (C_i, C_j) is an edge, there is some $a \in C_i$ and $b \in C_j$ such that (a, b) is an edge in G. From the claim, we see $\mathsf{first}[x] < \mathsf{first}[a] < \mathsf{last}[a] < \mathsf{last}[x]$ and $\mathsf{first}[y] < \mathsf{first}[b] < \mathsf{last}[b] < \mathsf{last}[y]$. Which in turn means $\mathsf{first}[a] < \mathsf{last}[a] < \mathsf{first}[b] < \mathsf{last}[b]$. This contradicts the *edge property* for (a, b).

See Figure 3 for an illustration of the whole proof in a picture.

☐

So, the above lemma gives us a way to recognize a vertex in the *source* component of G^{scc}. But we needed a vertex in the *sink* component of G^{scc}. How are we going to get that? A second epiphany answers this.



x = arg max$_{v \in C_i}$ last[v]
y = arg max$_{v \in C_j}$ last[v]
C_i
C_j
a
b
x
y
For sake of contradiction, suppose
$]_x \quad]_y$
NIP
$[_x \quad]_x \quad [_y \quad]_y$
$[_y \quad [_x \quad]_x \quad]_y$
By Claim 2
$[_x \quad [_a \quad]_a \quad]_x \quad [_y \quad [_b \quad]_b \quad]_y$
Contradicts edge property for (a,b)
A path from y → x contradicting "no edge from C_j to C_i"

Figure 3: *Proof of Lemma 1 in a picture.*

Epiphany 2. Let G_{rev} be the graph where all edges of G have been reversed. Observe that the strongly connected components of G_{rev} are the precisely the same as those in G, and that $(G_{\mathsf{rev}})^{\mathsf{scc}} = (G^{\mathsf{scc}})_{\mathsf{rev}}$. In other words, the source components of $(G)^{\mathsf{scc}}$ are precisely the sink components of $(G_{\mathsf{rev}})^{\mathsf{scc}}$. Therefore, if we run DFS on G and look at the vertex with the largest $\mathsf{last}[v]$ that is guaranteed to be in the *sink* component of $(G_{\mathsf{rev}})^{\mathsf{scc}}$. Which will allow us to find the strongly connected components of (G_{rev}). Which is the same as the strongly connected components of G. Done!

Strongly Connected Component Algorithm. We now state the full algorithm. The algorithm takes input G and returns the components of G^{scc}. Furthermore, they are returned in the *topological order* in G^{scc}. This fact is also useful (see a UGP problem regarding this).

```
1: procedure StronConnComp($G$):
2:     ▷ Returns the strongly connected components of G
3:     ▷ The order in which it is returned is the Top. Ord. of $G^{\mathsf{scc}}$.
4:     Run DFS($G, \{1, 2, \ldots, n\}$) to get $\mathsf{last}[v]$ for every vertex.
5:     $\pi$ be the decreasing order of $\mathsf{last}[v]$'s. ▷ Can be found in $O(n)$ time a la Top. Ord.
6:     Obtain $G_{\mathsf{rev}}$. ▷ This takes $O(n + m)$ time.
7:     Run DFS($G_{\mathsf{rev}}, \pi$) and return the connected components of the forest $F$.
```

Theorem 1. The StrongCommComp algorithm returns the strongly connected components of G in a topological order of G^{scc}, in $O(n+m)$ time.

Proof. The running time is easiest to figure out. There are two DFSs which take $O(n+m)$ time. One needs to sort the last in decreasing order. As in Topological Ordering, this can be done in $O(n)$ time. What is interesting is the correctness of the algorithm.

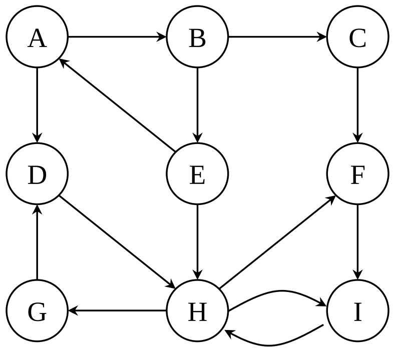
Let $C_1, \ldots, C_k$ be the components of G^{scc}. We claim that the components of the final forest F returned in Line 7 are these components returned in topological order of G^{scc}. More precisely, at the end of Line 7, the variable fcomp contains the number of strongly connected components, that is, k, and for every $1 \leq t \leq \mathsf{fcomp}$, the vertices $C_t := \{v : \mathsf{Fcomp}[v] = t\}$ forms the tth strongly connected component, and the ordering $C_1, \ldots, C_k$ is the topological order of G^{scc}.

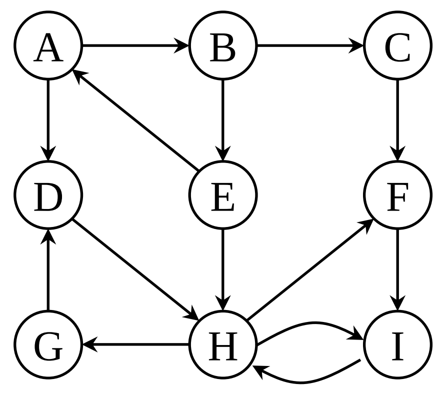
The discussion before the description of the algorithm indicates that C_1 is a sink component of $(G_{\mathsf{rev}})^{\mathsf{scc}}$ (to remind, the first vertex in π must lie in the *source* component of G^{scc} which is the *sink* component of $(G_{\mathsf{rev}})^{\mathsf{scc}}$). That is, C_1 is a true strongly connected component, and being the sink component of $(G_{\mathsf{rev}})^{\mathsf{scc}}$ it is indeed a source component of G^{scc}. What about the remaining C_i's?

To argue about them, we assume the statement is true for the first i components returned in Line 7. That is, they are indeed the first i components in some topological order of G^{scc}. Next, we argue about the $(i+1)$th component.

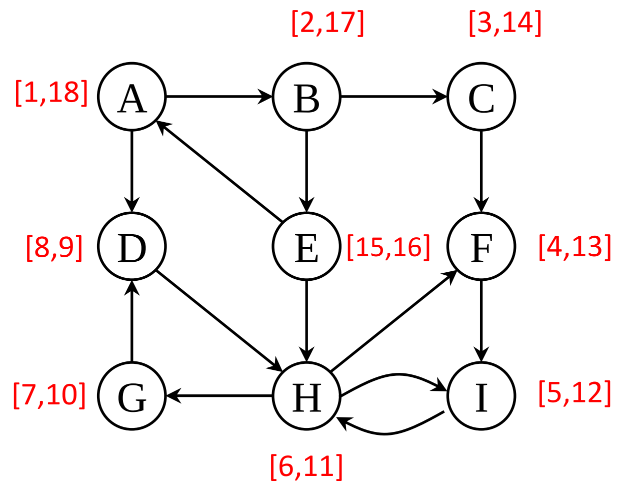
Note, by the way DFS works, that the vertex v picked as root at the $(i+1)$th step of Line 7 is the first vertex in π not in $C_1 \cup \ldots \cup C_i$. Since π is the decreasing order of lasts obtained in Line 4, we get that $\mathsf{last}[v] = \max_{u \notin C_1, \cdots, C_i} \mathsf{last}[u]$. Suppose v lies in component C_j of G^{scc}. We now claim that Lemma 1 implies C_j is a *source* component in $G^{\mathsf{scc}} \setminus (C_1 \cup \cdots \cup C_i)$. Suppose not: then there is some (C_k, C_j) edge in G^{scc} with $k > i$. From the lemma, we get that $f(C_k) > f(C_j)$ (otherwise the Lemma is violated). But this contradicts the choice of v. Therefore C_j is a source component of $G^{\mathsf{scc}} \setminus (C_1 \cup \cdots \cup C_i)$. That is, C_j is a sink component of $(G_{\mathsf{rev}})^{\mathsf{scc}} \setminus (C_1 \cup \cdots \cup C_i)$. Now notice that the DFS run from vertex v on G_{rev} will only discover vertices in C_j as it is a sink component in $(G_{\mathsf{rev}})^{\mathsf{scc}} \setminus (C_1 \cup \cdots \cup C_i)$. That is, the $(i+1)$th step discovers a component C_{i+1} which is a sink component of $(G_{\mathsf{rev}})^{\mathsf{scc}} \setminus (C_1 \cup \cdots \cup C_i)$. That is, it is a source component of $G^{\mathsf{scc}} \setminus (C_1 \cup \cdots \cup C_i)$. But that implies C_{i+1} is the next component in the topological order of G^{scc} after $C_1, \ldots, C_i$. $\square$

It is instructive to run the algorithm StrongConnComp by hand. Indeed, why don't you try it on this. The first time around, run DFS in the alphabetical order of vertices. Turn page and check your answer.

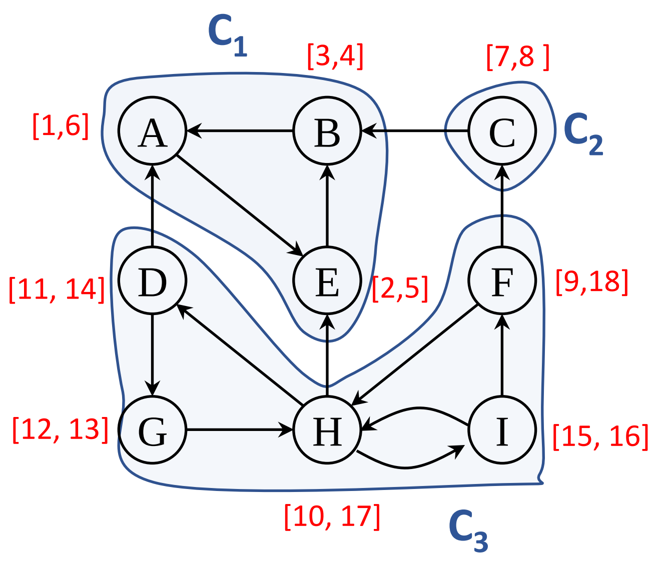




Run DFS in “alphabetical” order



Decreasing order of lasts: (A, B, E, C, F, I, H, G, D)
Run DFS in this order on ***reverse*** of G



The DAG G^{scc}