SCC Algorithms

September 29, 2020

1 Strongly Connected Components

We say that a directed graph $G = (V, E)$ is strongly connected if for every pair of vertices u and v, there is a path from u to v and there is a path from v to u.

Let $G = (V, E)$ be a directed graph. We say that a set of vertices $V' \subset V$ is a strongly connected component of G if the following holds.

1. For every pair of vertices u and v in V', there is a path from u to v that goes through only vertices in V', and here is a path from v to u that goes through only vertices in V'.

2. There is no super set V'' of V' with the following property: For every pair of vertices u and v in V'', there is a path from u to v that goes through only vertices in V'', and here is a path from v to u that goes through only vertices in V''.

Given a directed graph $G = (V, E)$ our goal is to identify all strongly connected components of G. Consider the following graph with 6 vertices A, B, C, D, E, F. With following edges: A to B, B to C, C to A, D to E, E to F, F to D, B to D. This graph has two SCC's: A, B, C and D, E, F.

Before we derive at our algorithm, we start with the following observation:

Let V' be a SCC of given graph: For every vertex $u \in V'$, we if make the call $DFS(G, u)$ then this procedure will discover every vertex in V'.

For example, consider the above graph: If we make $DFS(G, D)$ then it will mark vertices D, E, and F. This suggests the following initial algorithm to compute SCC: Pick a vertex v and call $DFS(G, v)$. Collect all the vertices marked. This is one SCC of the graph. Now pick another (unmarked) vertex u and call $DFS(G, u)$, and collected all vertices marked. This is another SCC of the graph. Repeat this process.

Let us see how this algorithm works on our example graph: Suppose we first pick D and call $DFS(G, D)$. This will mark D, E and F. Thus $\{D, E, F\}$ is a component of G. Now pick another unmarked vertex, say B, and call $DFS(G, B)$. This will mark A, B, C which is another component.

Even though the above algorithm seems to be correct, there is a caveat. Suppose that the first DFS call we make is $DFS(G, B)$. What happens? This procedure will mark all vertices of the graph. Clearly $\{A, B, C, D, E, F\}$ is not a SCC.

Our algorithm works when the first DFS call we make is $DFS(G, D)$ and does not work if the first call we make if $DFS(G, B)$. How can we know which DFS call should be made first? For the algorithm to work correctly, we must first start DFS from a vertex in the *bottom most* component, next from a vertex in the next *bottom most component* and so on. Thus our first goal is to determine an order in which DFS call should be made so that our algorithm works. Thus our algorithm to compute SCC consists of 2 steps

Step 1. Compute an ordering of vertices

Step 2. Perform DFS based on this ordering to compute SCC.

We will again use DFS to compute an ordering! Below is an algorithm. This algorithm assigns a number to each vertex which is exactly the finish time of DFS on that vertex. Given a graph G let G^r be a graph in which every edge direction is reversed.

Compute Order Algo

```
1. Input $G = (V, E)$.
2. Compute $G^r = (V, E^r)$.
3. Set $Discover[u]$ to $F$ for every $u \in V$.
4. counter = 0;
5. For every $x \in V$
   • if $Disvover[x]$ is $F$, Call $FinishDFS(G^r, x)$.
```

Procedure $FinishDFS(G, v)$

```
1. $Discover[x] = T$;
2. For every $y$ such that $\langle x, y \rangle \in E$
   (a) if $Discover[y]$ is $F$, $DFS(G, y)$.
3. counter++;
4. FinishTime[x] = counter;
```

The above algorithm assigns a finish time to each vertex. Note that vertex x for which $DFS(G^r, x)$ finishes first will have a finish time of 1. In general, if $DFS(G^r, x)$ finishes before $DFS(G^r, y)$, then $FinishTime(x) < FinishTime(y)$. Now we are ready to give the algorithm to compute SCCs of a graph.

SCC Algo

```
1. Input directed graph $G = (V, E)$.
2. Call $ComputeOrder(G)$.
3. Let $v_1, v_2, \cdots v_n$ be the ordering vertices of $v$ such that $FinishTime(v_i) > FinishTime(v_{i+1})$.
4. Set $Discover[u]$ to $F$ for every $u \in V$.
5. for $i$ in range $[1, \cdots, n]$, if $Discover[v_i]$ is $F$
   • Set $S = \emptyset$.
   • $SccDFS(G, v_i)$.
```

```
    • Output $S$.
```

where $SccDFS$ is the following algorithm;

```
Procedure $SccDFS(G, x)$
1. Mark v;
2. $S = S \cup \{x\}$.
3. For every $y$ such that $\langle x, y \rangle \in E$
    • if $u$ is unmarked, $DFS(G, u)$.
```

The run time of this algorithm is $O(m + n)$.

We will now prove the correctness of the above algorithm. For this we will consider *component graph* of G. Let $C_1, C_2, \cdots C_\ell$ be SCC's of G. We say that *there is an edge from C_i to C_j*, if there exists a vertex $x \in C_i$ and a vertex $y \in C_j$ such that $\langle x, y \rangle$ is an edge of G. A component that has no edges to any other component is called *bottom most/sink component.*

We make two observations.

Observation 6. If we start $DFS(G, x)$ from a vertex $x \in C_i$, then it will discover every vertex of C_i.

Observation 7. If there is an edge from C_i to C_j, then there is no edge from C_j to C_i.

Observation 8. If C_i is a SCC in G, then C_i is a SCC in G^r.

Claim If there is an edge from C_i to C_j, then there is a vertex in C_j whose finish time is higher than the finish time of every vertex from C_i (when DFS was performed on the reverse graph).

Proof. We consider two cases.

Case 1: First vertex (among the vertices of C_i and C_j) on which DFS is called belongs to C_i. Let that vertex be u. By observation 8, C_i is a SCC in G^r. By observation 6, $DFS(G^r, u)$ will discover every vertex of C_i. By observation 7, since there is no edge from C_i to C_j (in the reverse graph), $DFS(G^r, u)$ will not call DFS on any vertex from C_j. Thus every DFS is finished on every vertex from C_i before we start DFS on any vertex from C_j. Thus the claim holds.

Case 2. First vertex (among the vertices of C_i and C_j) on which DFS is called belongs to C_j. Let that vertex be u. What happens when we do $DFS(G^r, u)$. At some time this will make a call call to $DFS(G^r, x)$ such that there is an edge from x to a vertex y in C_j (note that such a vertex x must exist). When this happens, $DFS(G^r, x)$ will attempt to call $DFS(G^r, y)$. If y were already discovered, then it must be the case that at some time $DFS(G^r, z)$ is called for some $z \in C_i$. This will discover every vertex from C_i and DFS is finished on every vertex from C_i. Thus finish time of x is more than the finish time of every vertex from C_i. If y were not discovered, then $DFS(G^r, y)$ will be made. Again, every vertex of C_i will be discovered and will be finished before control comes back to x. Thus again, finish time of x is more than the finish time of every vertex from C_i.

The correctness of the algorithm follows from the above claim. Think about it.

2 Tarjan's SCC Detection Algorithm

Basic Definitions & Background. You have already read and learned about the depth-first exploration (we will refer DFS explore). The basic idea is to recursively "explore" from vertices that are yet to be marked visited/discovered. We will associated with each vertex two properties: starttime and finishtime–they represent the order in which the recursive call to the vertex initiates and terminates, respectively. Initially, both starttime and finishtime of all vertices are undefined. Given a directed graph $G = (V, E)$, its DFS exploration is encoded as follows:

```
1. cnt = 0;
2. for all v in V
3.   if v does not have starttime then
4.     DFS(v);

5. DFS(v)
6.   starttime(v) = cnt; cnt++;
7.   for all v -> v'
8.     if v' does not have starttime then
9.       DFS(v');
10.  endtime(v) = cnt; cnt++;
```

Note that, not all edges of G results in a recursive call (Lines 7–9). The edges that results in recursive calls are referred to as *tree edges*; these edges constitute the DFS-tree resulting from the DFS exploration of the graph. There are three other types of edges:

1. forward edge: an edge from a vertex u to its descendent v in the DFS-tree. (starttime(u) < starttime(v) and finishtime(v) < finishtime(u));

2. backward edge (backedge): an edge from a vertex u to its ancestor v in the DFS-tree (dual of forward edge condition);

3. cross edge: any edge that is not a tree, forward or backward edge. This type of edge does not connect ancestor-descendants in the DFS-tree.

Overview of the algorithm Tarjan's SCC algorithm will use backedges to identify candidate vertices that belong the same SCC. We will proceed by introducing couple of concepts/properties that are central to understanding the algorithm. These concepts are defined in the context of DFS exploration. Given a graph $G = (V, E)$ containing $C_1, C_2, \ldots, C_k$ strongly connected components (SCCs), for a DFS exploration of the graph

1. the *entry-vertex* of an SCC is the first vertex in that SCC that is visited in the DFS exploration. That is, the starttime of the entry-vertex in an SCC is smaller than that of any other vertices in the same SCC.

2. the *index*(v) is the starttime of the vertex v and the *lowlink*(v) is the smallest index of vertex u, an ancestor[1] of v in the DFS tree, that v may reach.

[1]ancestor relationship of a vertex includes itself.

Based on the above concepts, the following claims hold true (WLOG self-loops are not considered).

Claim 1 *For a DFS exploration, there is exactly one entry vertex per SCC.*

Why?

Claim 2 *In a DFS exploration, for any entry vertex v, if v belongs to a non-trivial SCC (containing more than one vertex), there is a backedge to v.*

For a non-trivial SCC C, in a DFS exploration if v is the entry vertex, then it must be reachable from at least one other vertex u in C. As u and v belongs to the same SCC, and v is the entry vertex in the DFS exploration, u is reachable from v in the DFS tree, which, in turn, implies that the edge from u to v is a backedge.

Claim 3 *In a DFS exploration, for any entry vertex v of SCC C, index(v) = lowlink(v) and for all other vertices u (if one exists) in C, index(u) > lowlink(u).*

Assume that v is an entry vertex of an SCC C and its index(v) $\neq$ lowlink(v); in particular, the index(v) > lowlink(v).[2] Therefore, v can reach one of its ancestor w (with lowlink(v) = index(w)) in the DFS tree, which further implies that v and w can reach other and w must be in the SCC C. In other words, the w is another entry vertex of C, which contradicts claim 1.

The second part of the claim follows from the first part.

Claim 4 *If a vertex v in an SCC has a tree edge (from the DFS tree) to another vertex u in another SCC, then lowlink(v) < lowlink(u).*

Use component graph to prove this.

The above claims are used in the following **steps for Tarjan's algorithm**.

1. For all vertices v that are yet to get an index value, start the recursive DFS exploration.

   (a) Initialize the index and lowlink values to be same (start time of the recursion for the vertex).

   (b) Push v to a Stack.

2. For all children of v

   (a) Recursively explore all the children of v that are yet to get an index value. These may be children belonging to the same SCC as v, in which case, their lowlink can be less than equal to the lowlink of v. Update lowlink(v) to a child's lowlink, if the latter is less than lowlink(v).

   (b) If a child u is already in the stack, then it must have been visited before v. If it is the case that the edge from v to u is cross edge, then v and u do not belong to the same SCC. This is avoided (see Step 3 below) by ensuring that at the time v is pushed to the stack only its ancestors from DFS tree are present in the stack. That is, the edge from v to u is a backedge; the lowlink(v) is updated to index(v) if the latter is less than the former.

[2]Note that, index(v) cannot be less than lowlink(v) as the node itself is considered one its ancestor.

3. After all the children have been examined, if the lowlink(v) = index(v), then one can infer that v is the entry vertex of some SCC. All vertices in the stack from the top till v belong to the same SCC. Pop the vertices from the stack until v is popped as well and add all the popped vertices to a new SCC.

The claims 1, 2, 3 and 4 are used in the above algorithm—lowlink of each vertex is computed as DFS exploration proceeds, and when the exploration of a vertex is complete, if it is identified as the entry vertex in the DFS exploration, then that vertex along with its descendents (in DFS tree) that are not already part of some other SCC are placed in a new SCC.

We need to show that the lowlink is computed in a way (step 2) that satisfies claim 3 and popping vertices from the stack till the identified entry vertex is popped indeed identifies a new SCC (step 3). We will first prove an invariant property for the stack.

Claim 5 *In the above algorithm, when any vertex v is pushed to the stack, the contents of the stack are ancestors of v from the DFS tree.*

WLOG, assume that v is the first vertex (starting from the bottom of the stack) such that the stack contents below v contains at least one vertex u that is not v's ancestor in the DFS tree.

There must be some vertex w that is one of the common ancestors of both u and v, and u and v are in two different branches of the DFS tree rooted at w.

As u is not v's ancestor in the DFS tree, u cannot reach v. The exploration from u terminates before exploration from v starts (finishtime(u) < starttime(v)). Furthermore, all vertices in the stack below u are u's ancestors (as per our assumption). Therefore, on termination of exploration from u, one of the following holds:

1. u's lowlink is equal to its index; in which case u will be removed from the stack (step 3 of the algorithm); OR

2. u's lowlink is less than its index value, which implies there is some ancestor of u (below u in the stack) that has its lowlink equal to index value; in which case u is removed from the stack as well (step 3 of the algorithm).

u cannot be in stack below v if u is not v's ancestor, which contradicts our assumption.

Lemma 1 *Lowlink computed as per Step 2 of the above algorithm satisfies claim 3.*

The correctness of Step 2a of lowlink computation is immediate. The lowlink value is recursively updated (if needed) from the unexplored children. Either the child belongs to the same SCC as v or the child belongs to a different SCC. In case of former, the lowlink of the child is less than or equal to the lowlink of the v. In case of latter, the child cannot reach any vertex with index value less than v and therefore, its lowlink is not less than the lowlink already assigned to v and is not used to update lowlink of v (Claim 4).

The step 2b, on the other hand, may be the result of crossedge or backedge. The existence of crossedge implies the existence of some vertex in the stack that is not v's ancestor. This is invalidated by claim 5. Therefore, step 2b corresponds to only backedges, in which case the lowlink is correctly updated as well.

Step 2 only considers the tree edges and backedges.

Lemma 2 *Step 3 of the algorithm identifies **only** the vertices of the SCC to which v belongs.*

Assume that there is at least one vertex u in stack above v, when exploration from v finishes, and u does not belong to the SCC C_v of v. Therefore, u belongs to a different SCC C_u that can be reached from v and none of the vertices from that C_u can reach any vertex in C_v.

The entry vertex (say, w, which is not necessarily $\neq u$) of C_u is visited at least before u (below u in stack) and after v (above v in stack) due to DFS exploration; The lowlink and index value of w are equal (Lemma 1) and finishtime(w) < finishtime(v). This implies, on termination of exploration from w, all vertices (including u), which are descendents of w and are present in stack, are removed from the stack and added to the C_u (Step 3 of algorithm).

Therefore, u cannot be present in the stack, when exploration from v finishes.

Lemma 3 *Step 3 of the algorithm identifies **all** the vertices of the SCC to which v belongs.*

Assume all vertices belonging to SCC of v are not present in the stack above v, when the recursive exploration from v finishes. Let u be one such vertex that belongs to the SCC of v but it not present in the stack above v. From Lemma 2, we know that u cannot be removed from the stack and added to a different SCC. This implies, u is not visited before the exploration from v finishes. This contradicts the property of DFS exploration: all descendents of v are visited before exploration from v finishes.

Pseudocode The correctness of the algorithm follows from the above lemmas.

```
1. cnt = 0;
2. for all vertices v in V
3.   if v does not have an index value
4.     TarjanDFS(v);

5. TarjanDFS(v)
    // Step 1
6.    index(v) = lowlink(v) = cnt; cnt++;
7.    push v to stack S; onstack(v) = true;
    // Step 2
8.    for all v -> v'
      // Step 2a
9.      if v' does not have an index value
10.       DFS(v');
11.       lowlink(v) = min(lowlink(v), lowlink(v'));
      // Step 2b
12.     elseif onstack(v') == true
13.        lowlink(v) = min(lowlink(v), index(v'));
    // Step 3: on termination of for all
14.   if index(v) == lowlink(v)
15.     create a new SCC C
16.     pop all vertices from stack till v (including v) and add to C
```

Properties to note The algorithm does not *necessarily* compute the lowlink of a vertex v as the smallest index of its ancestor. It computes lowlink such that the claim 3 is satisfied (Lemma 1).

Investigate the order in which SCCs are identified by Tarjan's and Kosaraju's algorithm. Is there any relationship of the order and the topological order in the component graph.