Strongly Connected Components

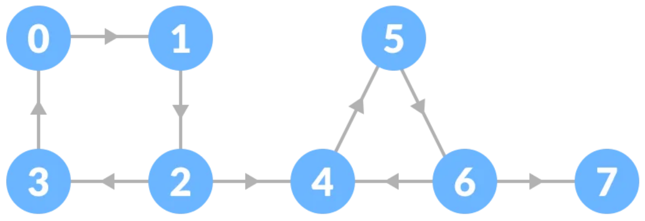
Defintion:

In a directed graph, a **strongly connected component (SCC)** is a set of nodes in which **every pair of nodes can reach eachother**. And you can't add any more nodes without ruining it it's strong connection.

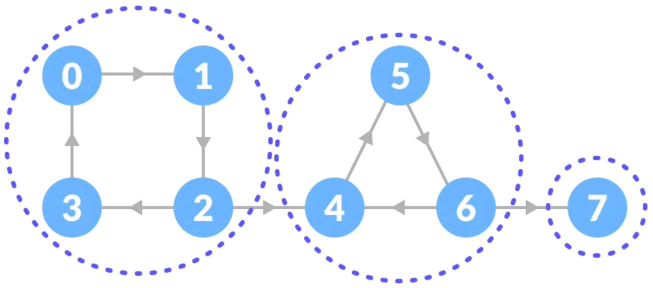
- Any pair of nodes in a component can reach eachother.
- A strongly connected component has to have a cycle.
  In other words, a **strongly connected component** is a portion of a directed graph in whiched there is a path from each vertex to another.

Example from Programiz

Take the **initial graph** depicted below.



Its **strongly connected components** are shown below.



Notice what make each of these components strongly connected. Each node can reach the others through a directed path.

Finding Strong Components

Kosaraju's Algorithm

Kosaraju's Algorithm is a DFS based algorithm used to find Strongly Connected Components in a graph.

- Do a **DFS on the original graph**, recording the **finish times** for each node.
- **Transpose** the graph (reverse the edges), done with an adjacency list.
- Do a **DFS** one by one in the now reversed order of the finish times. Until all nodes are visited.

You can think of the algorithm as this, we use DFS to find the correct order of nodes, and then we do another DFS going in that order.

- The reversed order we get from the transposed graph is the correct order we need.
- 💡: When we do a **post-order DFS on the transposed graph** we know that the last node will be inside of a *sink* SCC *(the sink component is the component without outgoing edges)*.

To represent our graph more clearly we can use a **component graph** to breakdown the problem before we analyze it for Strongly Connected Components. As illustrated in the image below.

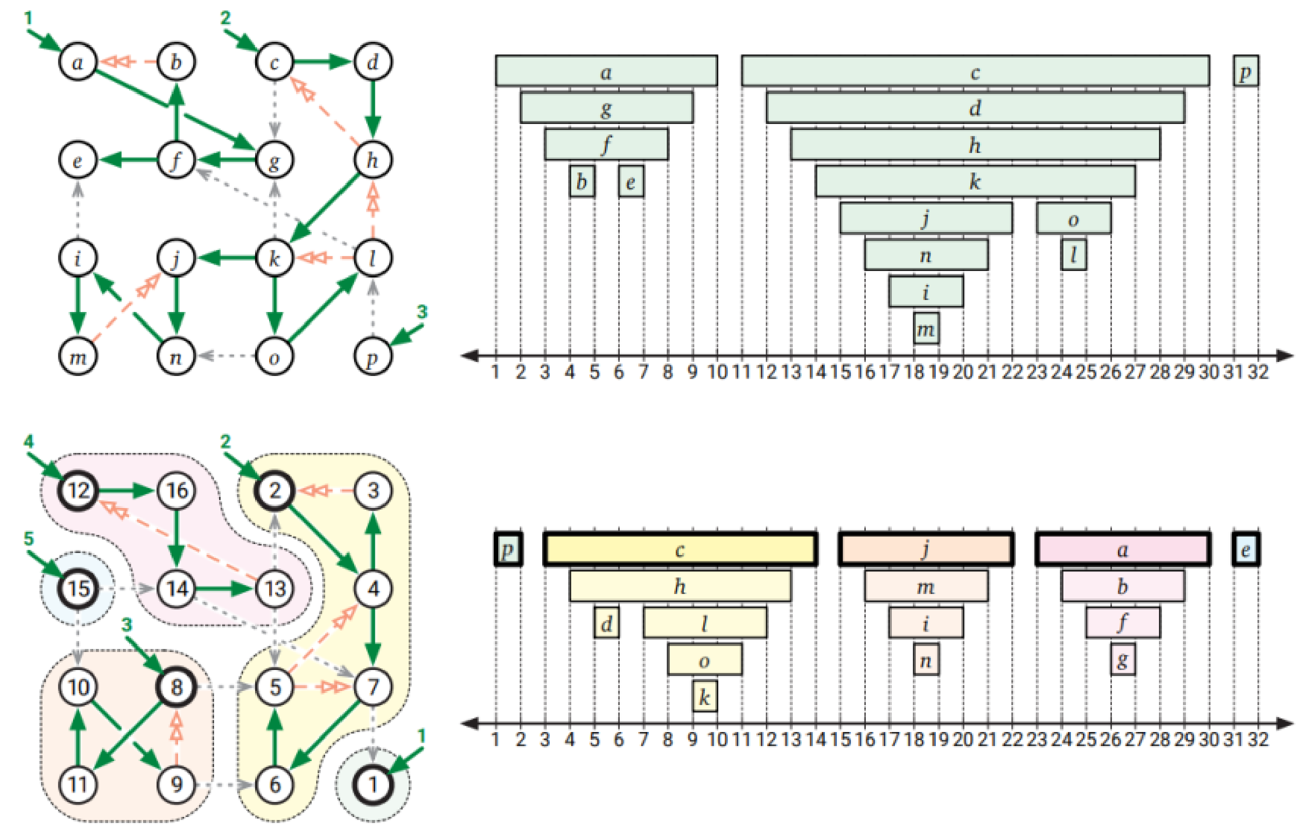


Figure 6.17. The Kosaraju-Sharir algorithm in action. Top: Depth-first traversal of the reversed graph. Bottom: Depth-first traversal of the original graph, visiting root vertices in reversed postorder from the first traversal.

The **Component Graph** is a DAG where each node is a *component* (connected subgraph).

- There aren't any cycles.
- And there are fewer nodes to take care of.
- Each node will be a component that we will then check for a strong connection. When we traverse across this component graph we need to be aware of a few key things:

1. 💡: **All of the nodes on the right** of a Component Graph will be **explored before** the nodes on the **left**. Thus their finishing times will come first.
2. Once a node is fully explored that is its finish time.
3. and that value is what is put into the list of finish times.
4. The node that finished last will be the first component.

Pseudocode

Now with that all clarified, let's take a deeper look at the Pseudocode.

1. First Notice that we have two DFS calls in the `kosaraju` function.
2. We do a postorder DFS on the reversed graph first to get the correct order.
3. Then we do the second DFS pass going in that order.

For the sake of revision, just remember that the `kosaraju` function is **two lines**:

- Reverse the edges.
- Perform a dfs on that reversed graph.

```
s = null
order = []
function dfs_loop(G):
        for node in G:
                if node is unvisited do:
                        s = node
                        dfs(G, node)

function dfs(G, V):
        V.explored = [true, ..., true]
        V.leader = s
        for each edge:(v, W):
                if W.explored is false do:
                        dfs(G, W)

function kosaraju():
        G_reversed = dfs(G)
        dfs(G_reversed)
```

Source
Each node's "leader" is the first node that we encountered within its SCC. We then group by leader and find each SCC.

We reverse the graph to transpose it which allows our dfs to count the SCCs.

Time Complexity

The overall time complexity of Kosaraju's Algorithm is `O(V + E)`:

1. The two DFS operations run in `O(V + E)` time
2. So does the reversal (transposing) of the graph you have to again traverse across all vertices and edges. Which is `O(V + E)`
   Source