Strongly Connected Components

Two Passes of DFS

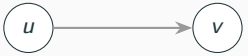
From Undirected to Directed

In an undirected graph, one DFS per unvisited vertex finds the connected components: if u reaches v, then v reaches u.

From Undirected to Directed

In an undirected graph, one DFS per unvisited vertex finds the connected components: if u reaches v, then v reaches u.

In a directed graph that symmetry is gone.

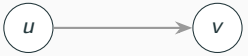


u reaches v, but v does not reach u.

From Undirected to Directed

In an undirected graph, one DFS per unvisited vertex finds the connected components: if u reaches v, then v reaches u.

In a directed graph that symmetry is gone.



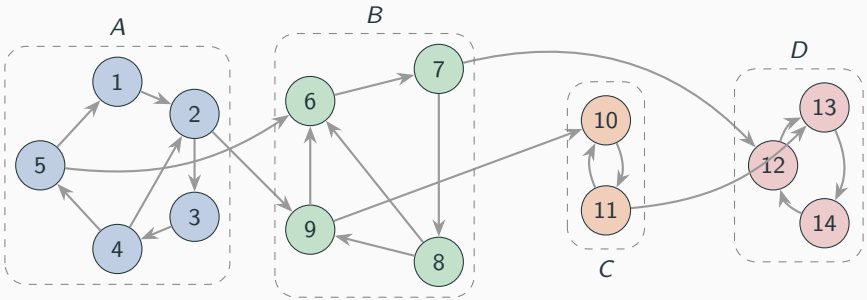
u reaches v, but v does not reach u.

So what is the right notion of a "component" for directed graphs?

Strongly Connected Components

Definition

A strongly connected component (SCC) of a directed graph is a maximal set $S \subseteq V$ such that for every $u, v \in S$ there is a path from u to v and a path from v to u.



Four SCCs: $A = \{1, \ldots, 5\}$, $B = \{6, \ldots, 9\}$, $C = \{10, 11\}$, $D = \{12, 13, 14\}$.

The Contracted Graph

Contract Each SCC

Shrink every SCC to a single vertex, keeping an edge $X \to Y$ whenever some edge of G goes from X to Y.



Contract Each SCC

Shrink every SCC to a single vertex, keeping an edge $X \to Y$ whenever some edge of G goes from X to Y.



Observation 1

The contracted graph is always a DAG.

Why a DAG?

Suppose the contracted graph had a cycle

$$X_1 \to X_2 \to \cdots \to X_k \to X_1.$$

Why a DAG?

Suppose the contracted graph had a cycle

$$X_1 \to X_2 \to \cdots \to X_k \to X_1.$$

- Inside each X_i, every vertex reaches every other.
- Following the cycle, every vertex of X_1 reaches every vertex of X_k, and every vertex of X_k reaches every vertex of X_1.

Why a DAG?

Suppose the contracted graph had a cycle

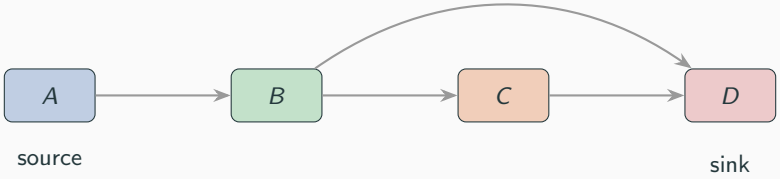
$$X_1 \to X_2 \to \cdots \to X_k \to X_1.$$

- Inside each X_i, every vertex reaches every other.
- Following the cycle, every vertex of X_1 reaches every vertex of X_k, and every vertex of X_k reaches every vertex of X_1.

So $X_1 \cup \cdots \cup X_k$ is strongly connected, contradicting the maximality of each X_i. $\square$

Sources and Sinks

- A source SCC has no edges coming in from other SCCs.
- A sink SCC has no edges going out to other SCCs.

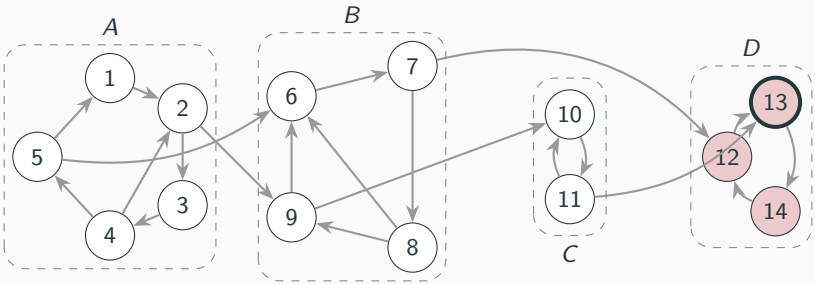


Every DAG has at least one source and one sink, and may have several.

Finding a Sink

The Idea

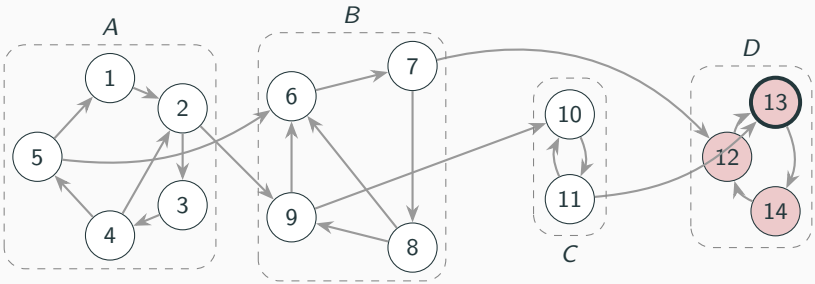
Run DFS from a vertex in a sink SCC. No edge leaves the sink, so DFS visits exactly that SCC.



DFS from 13 visits $13, 14, 12$: exactly D.

The Idea

Run DFS from a vertex in a sink SCC. No edge leaves the sink, so DFS visits exactly that SCC.



DFS from 13 visits $13, 14, 12$: exactly D.

Then delete D and repeat on what remains. But how do we find a vertex in a sink without knowing the SCCs?

Post Numbers Point the Way

Observation 2

If there is an edge from SCC X to SCC Y (so X comes before Y in the DAG), then

$$\max_{v \in X} post[v] > \max_{w \in Y} post[w].$$

By transitivity the same holds whenever X reaches Y in the contracted DAG.

Post Numbers Point the Way

Observation 2

If there is an edge from SCC X to SCC Y (so X comes before Y in the DAG), then

$$\max_{v \in X} post[v] > \max_{w \in Y} post[w].$$

By transitivity the same holds whenever X reaches Y in the contracted DAG.

So the vertex with the largest post number lies in a source SCC.

Proof of Observation 2

Let z be the first vertex of $X \cup Y$ that DFS visits.

Proof of Observation 2

Let z be the first vertex of $X \cup Y$ that DFS visits.

Case 1: $z \in X$**.** When z is visited, all of $X \cup Y$ is unvisited, and z reaches every vertex of $X \cup Y$. So DFS visits all of them before z finishes:

$$post[z] > post[w] \quad \text{for every } w \in Y.$$

Proof of Observation 2

Let z be the first vertex of $X \cup Y$ that DFS visits.

Case 1: $z \in X$**.** When z is visited, all of $X \cup Y$ is unvisited, and z reaches every vertex of $X \cup Y$. So DFS visits all of them before z finishes:

$$post[z] > post[w] \quad \text{for every } w \in Y.$$

Case 2: $z \in Y$**.** Since X reaches Y and the contracted graph is a DAG, Y cannot reach X. So `visit(z)` finishes all of Y without touching X, and every vertex of X is visited, and finishes, later.

Proof of Observation 2

Let z be the first vertex of $X \cup Y$ that DFS visits.

Case 1: $z \in X$**.** When z is visited, all of $X \cup Y$ is unvisited, and z reaches every vertex of $X \cup Y$. So DFS visits all of them before z finishes:

$$post[z] > post[w] \quad \text{for every } w \in Y.$$

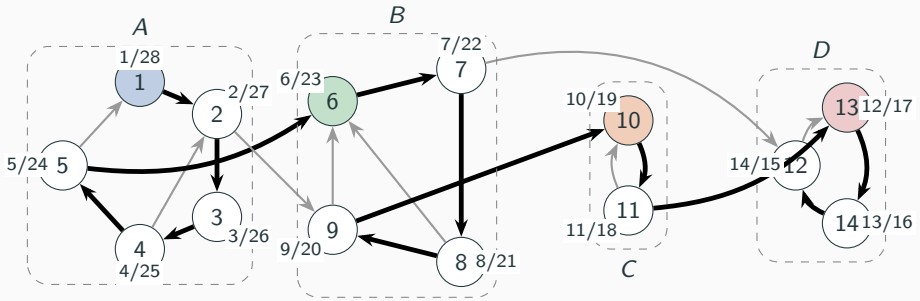
Case 2: $z \in Y$**.** Since X reaches Y and the contracted graph is a DAG, Y cannot reach X. So `visit(z)` finishes all of Y without touching X, and every vertex of X is visited, and finishes, later.

In both cases $\max_X post > \max_Y post$. $\square$

Careful: in case 1 only z is guaranteed to beat all of Y. Other vertices of X may finish early.

Example: pre/post Numbers

DFS on G from vertex 1, neighbours in increasing order. Bold edges form the DFS tree.



Highest post number per SCC: A: 28, B: 23, C: 19, D: 17. They decrease along the DAG $A \to B \to C \to D$.

Flip the Edges

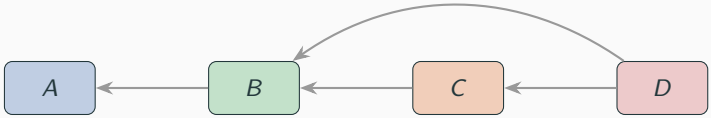
The largest post number finds a source, but we want a sink.

Flip the Edges

The largest post number finds a source, but we want a sink.

Observation 3

The transpose G^T (every edge reversed) has the same SCCs as G, and its contracted DAG is G's with every edge reversed. So the sources of G are exactly the sinks of G^T.



sink of G^T

u and v reach each other in $G \iff$ they do in G^T: just reverse both paths.

The Algorithm

Two Passes of DFS

1. Run DFS on G and record post numbers.

2. Run DFS on G^T, starting new searches in decreasing order of post number. Each DFS tree of this pass is one SCC.

Two Passes of DFS

1. Run DFS on G and record post numbers.

2. Run DFS on G^T, starting new searches in decreasing order of post number. Each DFS tree of this pass is one SCC.

Why it works. The first start is the vertex with the largest post number, which lies in a source of G, so in a sink of G^T. Its DFS collects exactly that SCC.

Two Passes of DFS

1. Run DFS on G and record post numbers.

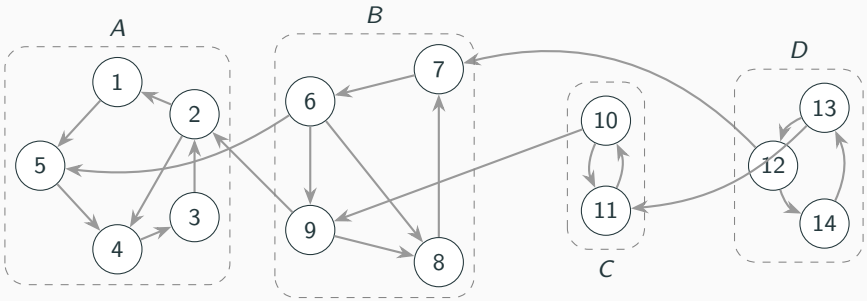
2. Run DFS on G^T, starting new searches in decreasing order of post number. Each DFS tree of this pass is one SCC.

Why it works. The first start is the vertex with the largest post number, which lies in a source of G, so in a sink of G^T. Its DFS collects exactly that SCC.

After those vertices are marked, the next unvisited vertex with the largest post number lies in a source of what remains of G (Observation 2 still applies to the remaining SCCs), hence in a sink of what remains of G^T. Repeat.

Second Pass on G^T

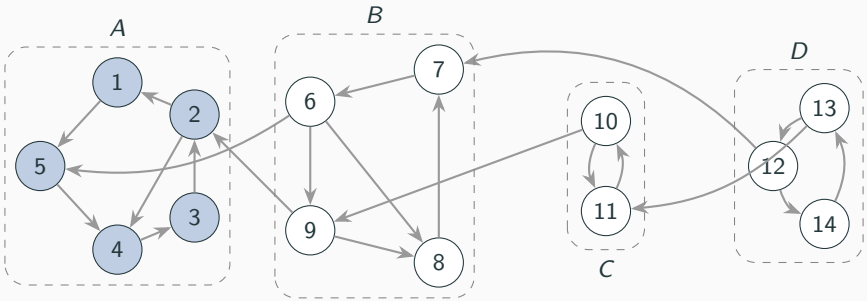
Starts in decreasing post order: 1 (28), then 6 (23), 10 (19), 13 (17).



The graph G^T: every edge of G reversed.

Second Pass on G^T

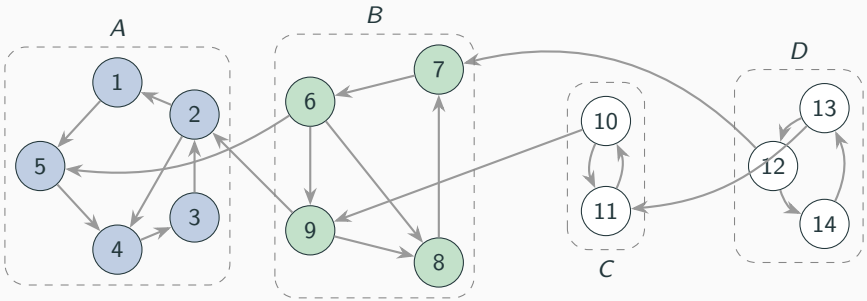
Starts in decreasing post order: 1 (28), then 6 (23), 10 (19), 13 (17).



DFS from 1 on G^T reaches only $\{1, 2, 3, 4, 5\}$: SCC A.

Second Pass on G^T

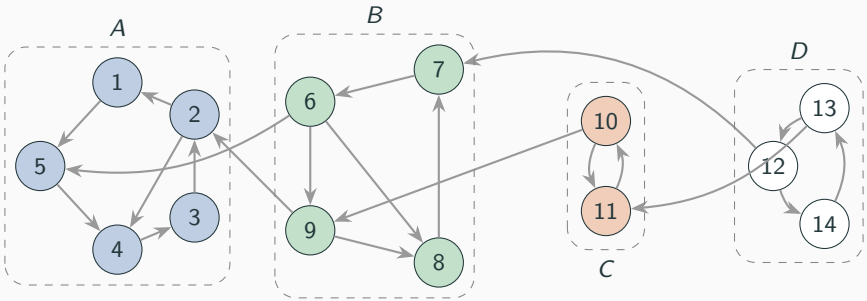
Starts in decreasing post order: 1 (28), then 6 (23), 10 (19), 13 (17).



DFS from 6 reaches $\{6, 7, 8, 9\}$; edge $6 \to 5$ hits a visited vertex. SCC B.

Second Pass on G^T

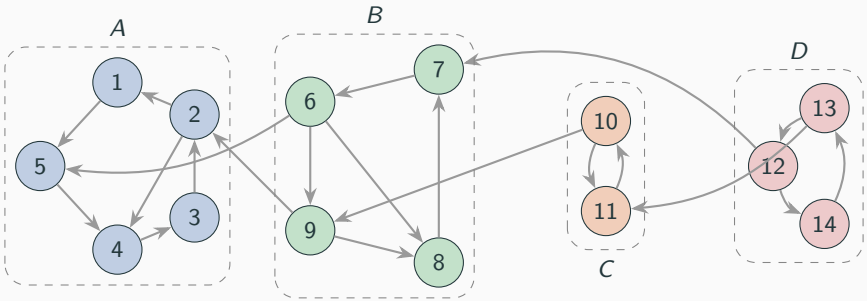
Starts in decreasing post order: 1 (28), then 6 (23), 10 (19), 13 (17).



DFS from 10 reaches $\{10, 11\}$: SCC C.

Second Pass on G^T

Starts in decreasing post order: 1 (28), then 6 (23), 10 (19), 13 (17).



DFS from 13 reaches $\{13, 12, 14\}$: SCC D. Done.

Running Time

- First DFS on G: $O(n + m)$.
- Build G^T: one pass over the adjacency lists, $O(n + m)$.
- Second DFS on G^T: $O(n + m)$. Sorting by post number is free: just record vertices in the order they finish and read the list backwards.

All SCCs in $O(n + m)$ time.

In Julia: First Pass

```
function strongly_connected_components(n, adj)
    # Step 1: DFS on G, recording vertices as they finish
    visited = falses(n)
    post_order = Int[]
    function dfs1(v)
        visited[v] = true
        for w in adj[v]
            if !visited[w]
                dfs1(w)
            end
        end
        push!(post_order, v) # v finishes
    end
    for v in 1:n
        if !visited[v]
            dfs1(v)
        end
    end
```

In Julia: Transpose

Reverse every edge: $u \to v$ in G becomes $v \to u$ in G^T.

```
# Step 2: build the transpose G^T
adj_t = [Int[] for _ in 1:n]
for u in 1:n
    for v in adj[u]
        push!(adj_t[v], u)
    end
end
```

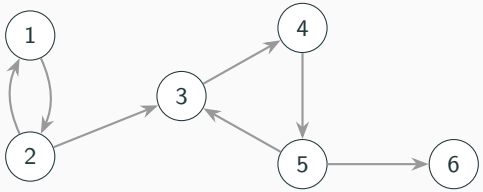
One pass over all adjacency lists: $O(n + m)$.

In Julia: Second Pass

```
    # Step 3: DFS on G^T in decreasing post order
    fill!(visited, false)
    scc_id = zeros(Int, n)
    num_scc = 0
    function dfs2(v, id)
        visited[v] = true
        scc_id[v] = id
        for w in adj_t[v]
            if !visited[w]
                dfs2(w, id)
            end
        end
    end
    for v in reverse(post_order) # largest post number first
        if !visited[v]
            num_scc += 1
            dfs2(v, num_scc)
        end
    end
    return num_scc, scc_id
end
```

Exercise

Run the algorithm on this graph (neighbours in increasing order, first pass starting from 1).
Give the post numbers and the SCCs in the order they are found.



Solution

First pass on G**.** $1 \to 2 \to 3 \to 4 \to 5 \to 6$, then everything unwinds:

vertex	1	2	3	4	5	6
pre/post	1/12	2/11	3/10	4/9	5/8	6/7

Solution

First pass on G. $1 \to 2 \to 3 \to 4 \to 5 \to 6$, then everything unwinds:

vertex	1	2	3	4	5	6
pre/post	1/12	2/11	3/10	4/9	5/8	6/7

Second pass on G^T, in order $1, 2, 3, 4, 5, 6$:

- from 1: $\{1, 2\}$ (in G^T, $2 \to 3$ is reversed, so 3 is not reached);
- from 3: $\{3, 5, 4\}$;
- from 6: $\{6\}$, since its only edge in G^T goes to 5, already visited.

Three SCCs: $\{1, 2\}$, $\{3, 4, 5\}$, $\{6\}$.