Explanation of Algorithm

- DFS numbers of all vertices initialized to 0.
  - It serves as vertices marked "unvisited"
  - Assigning DFS number to a vertes implies it is marked "visited"
- It then selects an arbitrary vertex $v = x$, builds a DFST, and computes LOW[v] in one single pass.
- An edge leading to a new vertex is a tree edge.
- The tree edge is pushed on to the stack before making the recursive call.

Explanation of Algorithm

- On return from call LOW[v] is computed.
- If v is an articulation point then its DFS number must be greater than or equal to LOW point of the child.
- At this point, all the edges up to and including v, w are output as a biconnected component.
- If w is a old vertex (DFS number is nonzero) then it is a back edge.
- Only back edge to a proper ancestor of the parent $(w \neq u)$ would lower the LOW point.

Correctness of Algorithm

Lemma (*Correctness of LOW point computation*)

When Search(w) procedure completes, the edges in the stack above (v, w) are the edges in the same biconnected components as (v, w).

- To prove it, we use induction on the number of biconnected components, b.
- If there is just one biconnected components, i.e., $b = 1$, it is trivial.
  - In this case, there is no articulation point.
  - So all the edges of G will be on the stack.

Correctness of Algorithm

- Induction hypothesis: assume this to be true for all graphs having b biconnected components.
- Let G have $b + 1$ biconnected components.
- Consider the first Bicon(w, v) call that ends with LOW[w] $\geq$ dfn[v] for a tree edge (v, w).
- No edge has been removed yet from the STACK.
- Since LOW[w] $\geq$ dfn[v], all the set of edges above (v, w) are incident on the descendants of w and first block B_1 has been detected.
- So the edges on the STACK above (v, w) are exactly the edges in the same biconnected component as (v, w).

Correctness of Algorithm

- Now after removing edges of B_1, the algorithm works on induced graph $G' = G - B_1$, in exactly the same way as it had worked on graph G.
- But G' has b biconnected components.
- By induction, algorithm should correctly obtain all b biconnected components of G'.

Notion of Connectedness in Directed Graphs

- For every pair of vertices u, v there exists a path.
- But path here means directed path.
- Connectivity in directed graph implies both $u \leadsto v$ and $v \leadsto u$.
- In general, it is possible that path $u \leadsto v$ may exist but $v \not\leadsto u$.
- Or even u and v are not reachable from each other.
- There is also a notion of weak connectivity: it possible to reach any vertex from any other vertex by traversing edges in some direction (ignoring direction).
- It essentially means every vertex has either indegree or outdegree of at least 1.

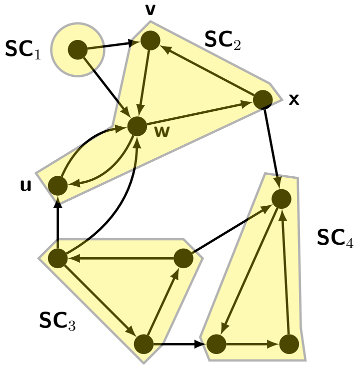
Strong Connected Components (SCC)

Definition (*SCC*)

Let $G = (V, E)$ be directed graph. G is strongly connected iff for every pair of vertices v, w, there is a directed path from v to w and also a directed path from w to v.

- SCC is an equivalence relation.
- If u, v are in same SCC then uRv and vRu where R: there exists a directed path.
- If uRv and vRw then obviously, uRw.
- Collapsing each SCC to a vertex we get a condensation graph which is a DAG.

Strong Connected Components



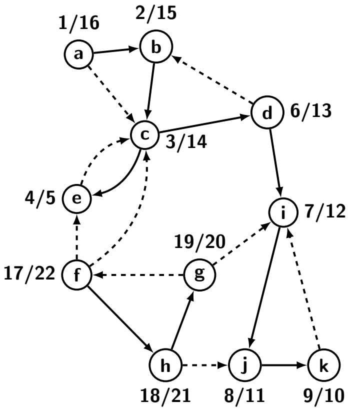
- There are four SCCs.
- One SCC has just one vertex.
- Consider $u, v \in SC_2$.
- There exists pair of paths: $u \to w \to x \to v$ and $v \to w \to u$.

Strong Connected Components

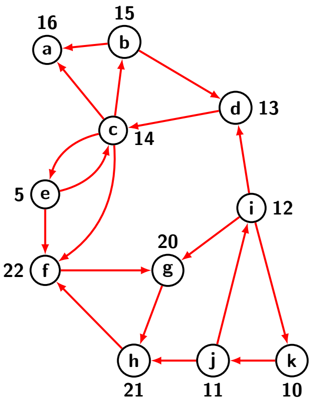
- Compute Finish time f[u] for $u \in V$.
- Reverse each edge of G to obtain G^T.
- Call DFS on G^T but apply it decreasing order of finish time f[u] as computed in first DFS.
- Output the vertices of each tree in DFS forest of G^T as a separate SCC.

Strong Connected Components

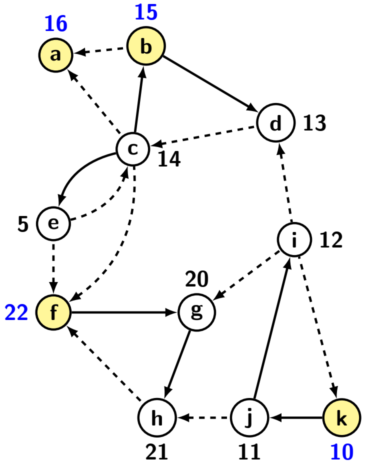
Original Graph G



Transformed Graph G^T (by reversing edge directions).



Strong Connected Components



- Start vertices in decreasing order of finish time: f, a, b, and k.
- DFS from from these vertices discover SCCs:
  - f: $\{f, g, h\}$
  - a: $\{a\}$
  - b: $\{b, c, d, e\}$
  - k: $\{i, j, k\}$

Correctness of SCC Algorithm

- Let $d[v]$: DFS discovery time, and
- Let $f[v]$: DFS finish time.
- Let S be subset of V.

$$d[S] = \min_{u \in S} d[u], \text{ and } f[S] = \max_{u \in S} f[u]$$

Lemma

Let S_1 and S_2 be two distinct SCCs. If $\exists$ an edge $(u, v) \in E$, where $u \in S_1$ and $v \in S_2$ then $f[S_1] > f[S_2]$

Correctness of SCC Algorithm

Proof.

- Case 1 ($d[S_1] < d[S_2]$): Let x be first vertex in S_1 to be discovered.
- At this time none of the vertices in S_1 and S_2 have been marked ”visited”
- For any vertex $w \in S_2$, $\exists$ a path from x to w.
- So, all vertices in S_2 are descendants of x
- Therefore, $f[x] = f[S_2] < f[S_1]$

$\square$

Correctness of SCC Algorithm

Proof continues.

- Case 2 ($d[S_1] > d[S_2]$): Let y be the first vertex discovered in S_2.
- All vertices in S_2 are descendant of y.
- Therefore, by definition $f[y] = f[S_2]$.
- Since, S_1 and S_2 are distinct SCC there cannot be path from any vertex of S_2 to a vertex of S_1.
- So all vertices in S_1 are "unvisited" when DFS of S_2 is complete.
- Therefore, $f[S_2] < f[S_1]$.

□