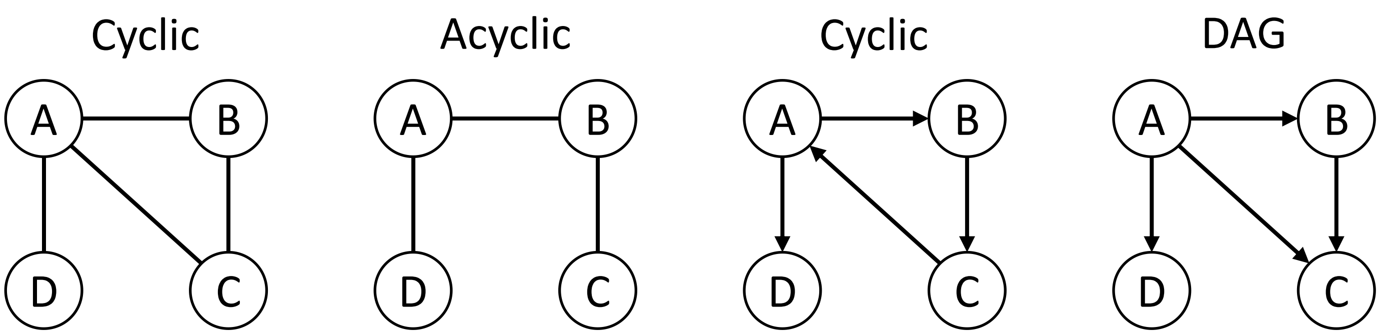
(Some) Applications of DFS

Data Structures and Algorithms

Nanjing University, Fall 2021
郑朝栋

Directed Acyclic Graphs (DAG)

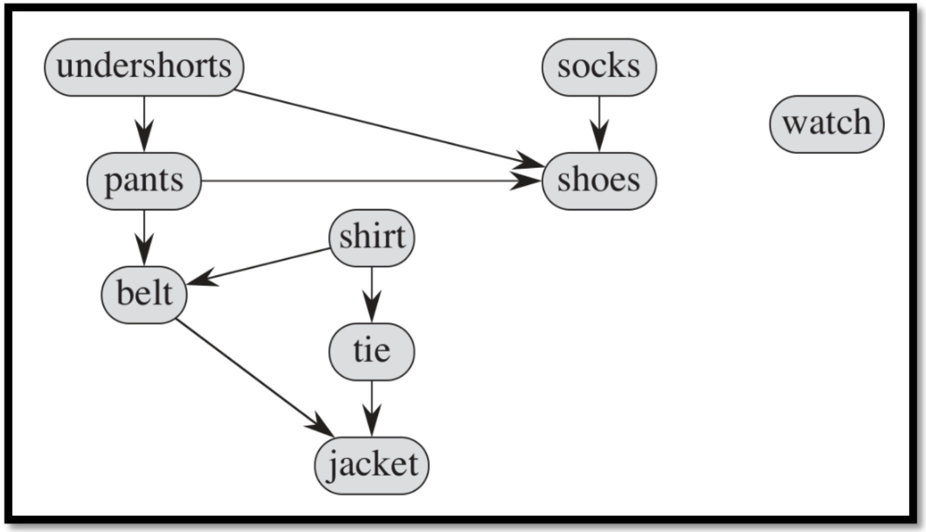
- A graph without cycles is called **acyclic**.
- A directed graph without cycles is a **directed acyclic graph** (**DAG**).
- DAGs are good for modeling relations such as: *causalities*, *hierarchies*, and *temporal dependencies*.



Application of DAG

- DAGs are good for modeling relations such as: *causalities*, *hierarchies*, and *temporal dependencies*.
- **Example:**
  - Consider how you get dressed in the morning.
  - Must don certain garments before others (e.g., socks before shoes).
  - Other items may be put on in any order (e.g., socks and pants).
  - This process can be modeled by a DAG!

What is a valid order to perform all the task?



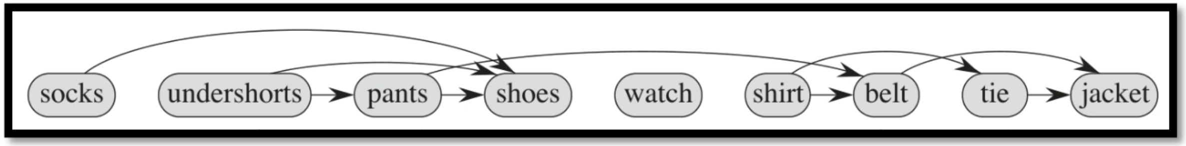
Topological Sort

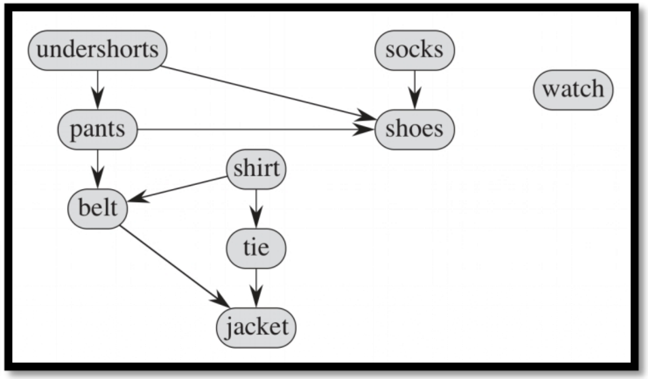
- A **topological sort** of a DAG G is a linear ordering of its vertices such that if G contains an edge (u, v) then u appears before v in the ordering.
- $E(G)$ defines a *partial order* over $V(G)$, a **topological sort** gives a *total order* over $V(G)$ satisfying $E(G)$.

- Topological sort is impossible if the graph contains a cycle.
- A given graph may have multiple different valid topological ordering.

How to generate a topological ordering?

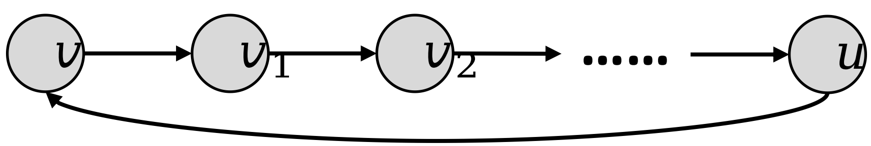
A topological ordering arranges the vertices along a horizontal line so that all edges go “from left to right”.





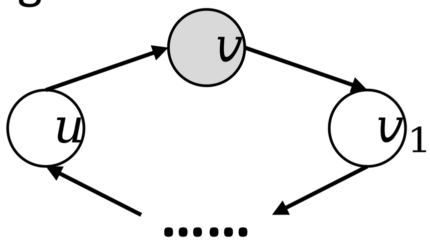
Topological Sort

- A **topological sort** of a DAG G is a linear ordering of its vertices such that: if G contains an edge (u, v) then u appears before v in the ordering.
- Directed graphs containing cycles have no topological ordering.
  Q: Does every DAG has a topological ordering?
  Q: How to tell if a directed graph is acyclic? If acyclic, how to do topo-sort?
- **Lemma 1**: Directed graph G is acyclic iff a DFS of G yields no back edges.
- **Proof of [==>]:**
- For the sake of contradiction, assume DFS yields back edge (u, v).
- So v is ancestor of u in DFS forest, meaning a path from v to u in G.
- But together with edge (u, v) this creates a cycle. Contradiction!



Topological Sort

- A **topological sort** of a DAG G is a linear ordering of its vertices such that: if G contains an edge (u, v) then u appears before v in the ordering.
- Directed graphs containing cycles have no topological ordering.
  Q: Does every DAG has a topological ordering?
  Q: How to tell if a directed graph is acyclic? If acyclic, how to do topo-sort?
- **Lemma 1**: Directed graph G is acyclic iff a DFS of G yields no back edges.
- **Proof of [<==]:**
- For the sake of contradiction, assume G contains a cycle C.
- Let v be the first node to be discovered in C.
- By the White-path theorem, u is a descendant of v in DFS forest.
- But then when processing u, (u, v) becomes a back edge!



Topological Sort

- A **topological sort** of a DAG G is a linear ordering of its vertices such that: if G contains an edge (u, v) then u appears before v in the ordering.
- Directed graphs containing cycles have no topological ordering.
  Q: Does every DAG has a topological ordering?
  Q: How to tell if a directed graph is acyclic? If acyclic, how to do topo-sort?
- **Lemma 1**: Directed graph G is acyclic iff a DFS of G yields no back edges.
- **Lemma 2**: If we do a DFS in DAG G, then $u.f > v.f$ for every edge (u, v) in G.
- **Proof:**
- When exploring (u, v), v cannot be GRAY. (Otherwise we have a back edge.)
- If v is WHITE, then v becomes a descendant of u, and $u.f > v.f$.
- If v is BLACK, then trivially $u.f > v.f$.

Topological Sort

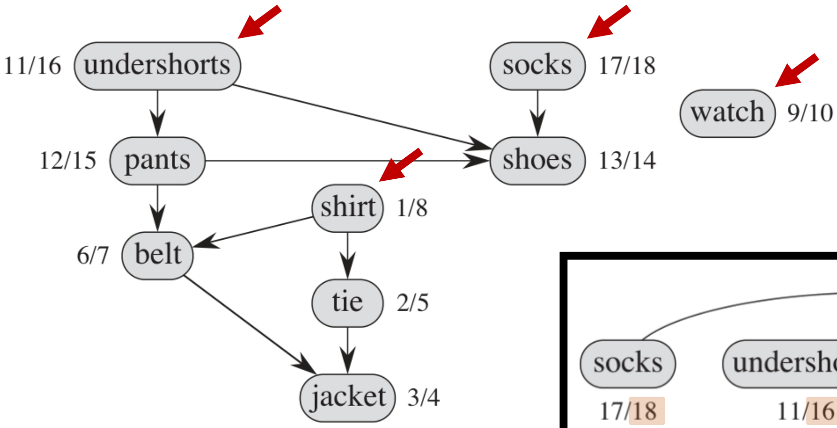
- A **topological sort** of a DAG G is a linear ordering of its vertices such that: if G contains an edge (u, v) then u appears before v in the ordering.
- Directed graphs containing cycles have no topological ordering.
  Q: Does every DAG has a topological ordering?
  Q: How to tell if a directed graph is acyclic? If acyclic, how to do topo-sort?
- **Lemma 1**: Directed graph G is acyclic iff a DFS of G yields no back edges.
- **Lemma 2**: If we do a DFS in DAG G, then $u.f > v.f$ for every edge (u, v) in G.

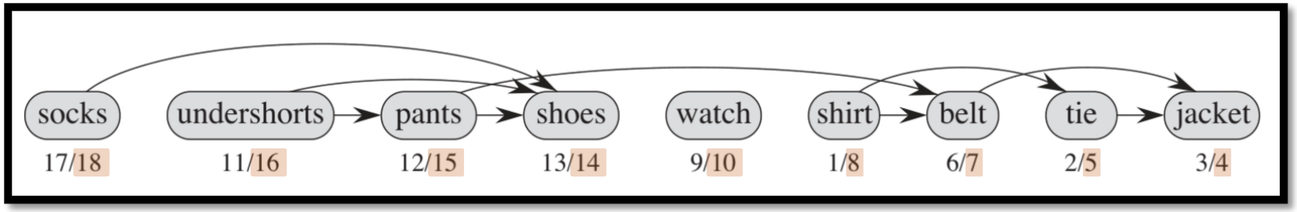
Time complexity is $O(n + m)$.

- **Topo-Sort of G:**
  (**a**) Do DFS on G, compute finish times for each node along the way.
  (**b**) When a node finishes, insert it to the *head* of a list.
  (**c**) If no back edge is found, then the list eventually gives a topo-ordering.
- **Thm**: Every DAG has a topological ordering.
- **Thm**: Descreasing order of finish times of DFS on DAG gives a topo-ordering.

Topological Sort

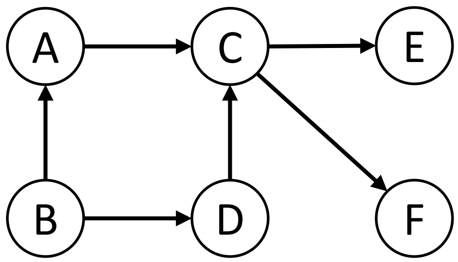
- A **topological sort** of a DAG G is a linear ordering of its vertices such that: if G contains an edge (u, v) then u appears before v in the ordering.
- **Thm**: Every DAG has a topological ordering.
- **Thm**: Descreasing order of finish times of DFS on DAG gives a topo-ordering.
- **Topo-Sort of G:**
  (**a**) Do DFS on G, compute finish times for each node along the way.
  (**b**) When a node finishes, insert it to the *head* of a list.
  (**c**) If no back edge is found, then the list eventually gives a topo-ordering.





Source and Sink in DAG

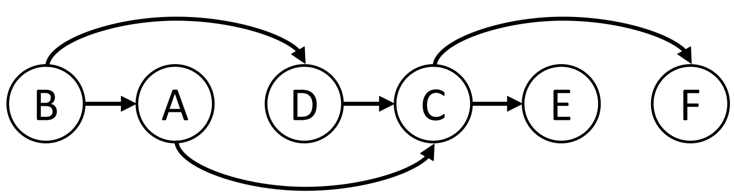
- A **source node** is a node with no incoming edges;
  A **sink node** is a node with no outgoing edges.
  - **Example**: B is source; both E and F are sink.
- **Claim:** Each DAG has at least one source and one sink. (Why?)
- **Observations:** In DFS of a DAG, node with max finish time must be a source.
  (Node with max finish time appears first in topo-sort, it cannot have incoming edges.)
- **Observations:** In DFS of a DAG, node with min finish time must be a sink.
  (Node with min finish time appears last in topo-sort, it cannot have outgoing edges.)

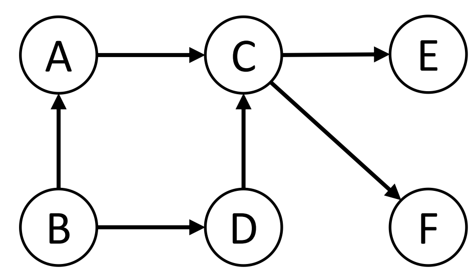


Alt Algorithm for Topo-Sort

- **Claim:** Each DAG has at least one source and one sink.
- **Observations:** In DFS of a DAG, node with max finish time must be a source.
  (Node with max finish time appears first in topo-sort, it cannot have incoming edges.)
- **Observations:** In DFS of a DAG, node with min finish time must be a sink.
  (Node with min finish time appears last in topo-sort, it cannot have outgoing edges.)
- **An alternative algorithm for topo-sort in a DAG:**
  (1) Find a source node s in the (remaining) graph, output it.
  (2) Delete s and all its outgoing edges from the graph.
  (3) Repeat until the graph is empty.

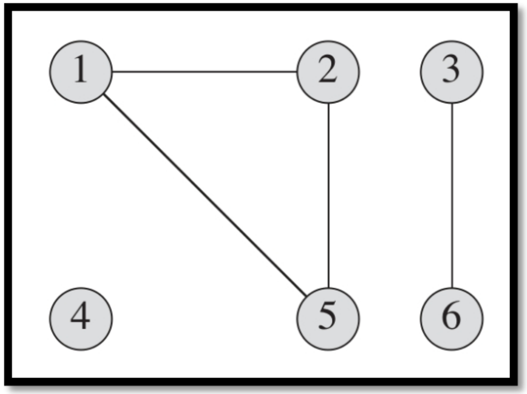
Formal proof of correctness? How efficient can you implement it?



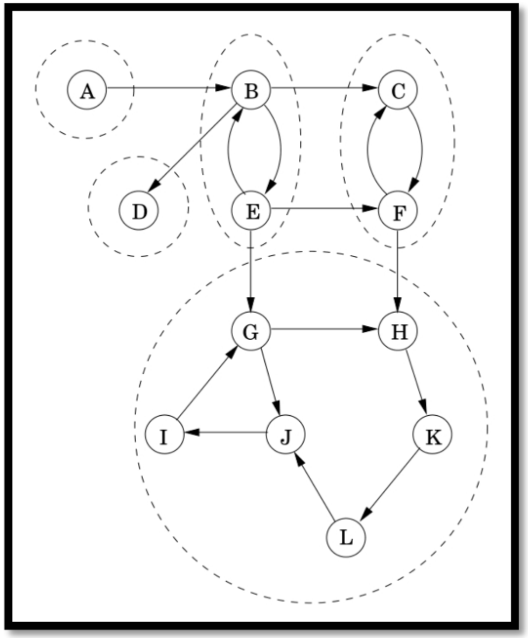


(Strongly) Connected Components

- For an undirected graph G, a **connected component** is a maximal set $C \subseteq V(G)$, such that for any pair of nodes u and v in C, there is a path from u to v.
- **E.g.**: $\{4\}, \{1,2,5\}, \{3,6\}$



- For a directed graph G, a **strongly connected component** is a maximal set $C \subseteq V(G)$, such that for any pair of nodes u and v in C, there is a directed path from u to v, and vice versa.
- **E.g.**: $\{A\}, \{D\}, \{B, E\}, \{C, F\}, \{G, H, I, J, K, L\}$

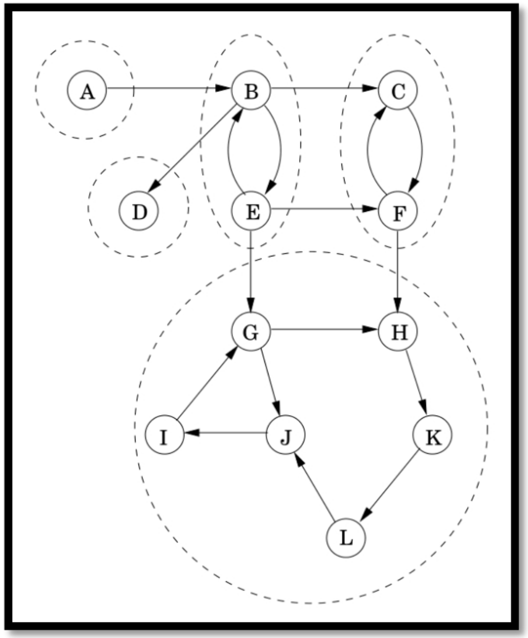


Computing CC and SCC

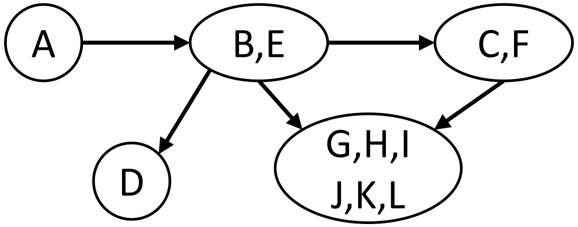
- **Q**: Given an undirected graph, how to compute its connected components (CC) ?
- **A**: Easy, just do DFS (or BFS) on the entire graph.
  (DFS(u), or BFS(u), reaches exactly nodes in the CC containing u.)
- **Q**: Given a directed graph, how to compute its strongly connected components (SCC) ?
- Err, can be done efficiently, but not so obvious…

Component Graph

- Given a directed graph $G = (V, E)$, assume it has k SCC $\{C_1, C_2, \ldots, C_k\}$, then the **component graph** is $G^C = (V^C, E^C)$.
- The vertex set V^C is $\{v_1, v_2, \ldots, v_k\}$, each representing one SCC.
- There is an edge $(v_i, v_j) \in E^C$ if there exists $(u, v) \in E$, where $u \in C_i$ and $v \in C_j$.

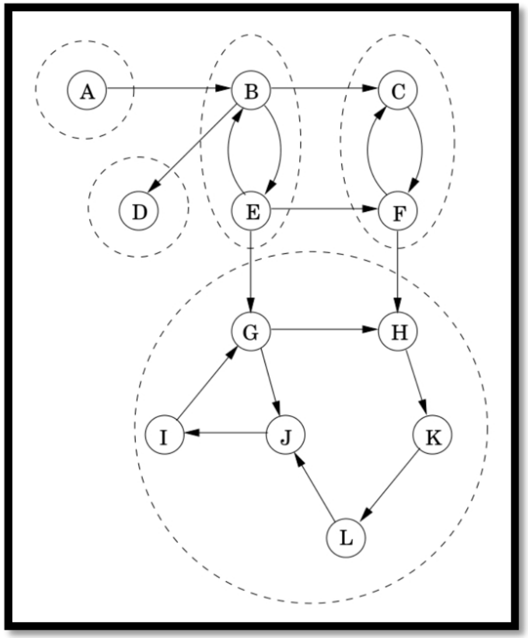


- **Claim:** A component graph is a DAG.
- **Proof:** Otherwise, the components in the circle becomes a bigger SCC, contradiction!

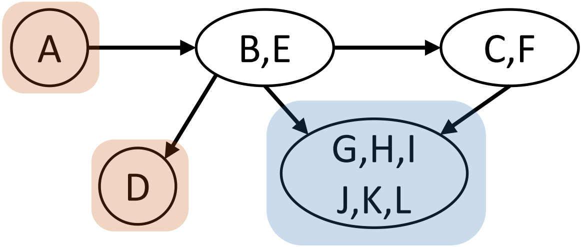


Computing SCC

- A component graph is a DAG.
- Each DAG has at least one **source** and one **sink**.
- If we start from a node in a sink SCC, then we explore exactly nodes in that SCC and stop!
  (Due to the white-path theorem.)

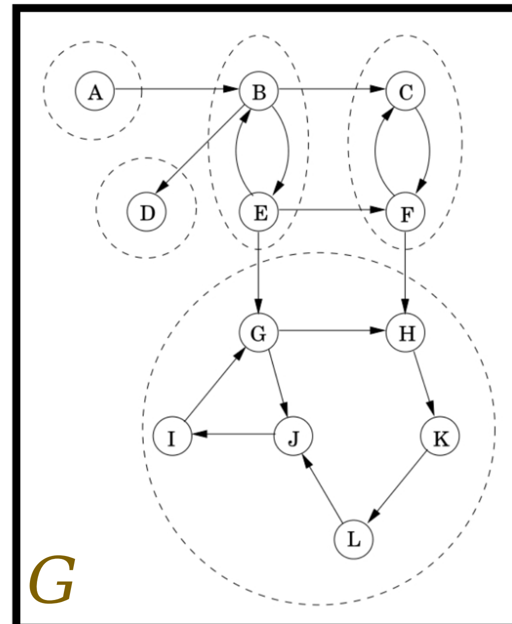


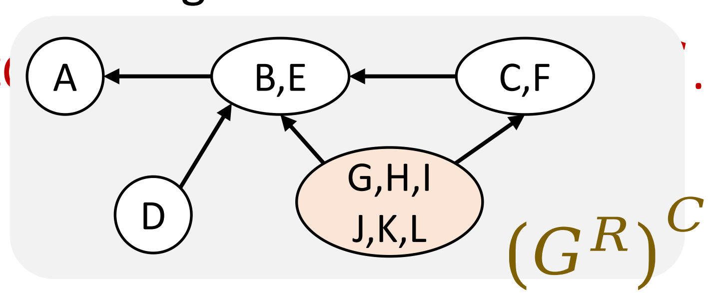
- **A good start, but two problems exist:**
- (1) How to identify a node that is in a sink SCC?
- (2) What to do when the first SCC is done?

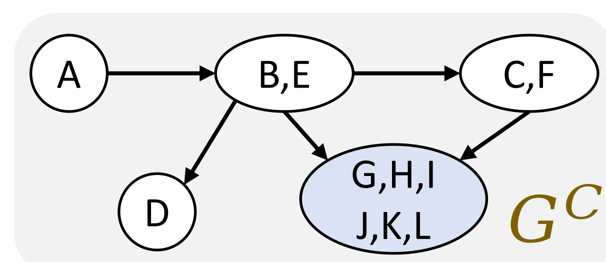


Computing SCC

- If we start from a node in a sink SCC, then we explore exactly nodes in that SCC and stop!
- **Good idea but two problems exist:**
  (**1**) How to identify a node that is in a sink SCC?
  (**2**) What to do when the first SCC is done?
- Don't do it directly: find a node in a *source* SCC!
- Reverse the direction of each edge in G gets G^R.
- G and G^R have same set of SCCs.
- G^C and $(G^R)^C$ have same vertex set, but the direction of each edge is reversed.
- A source SC .

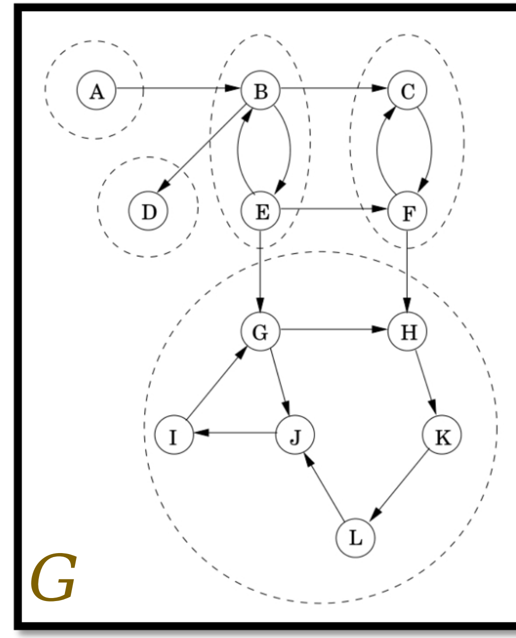


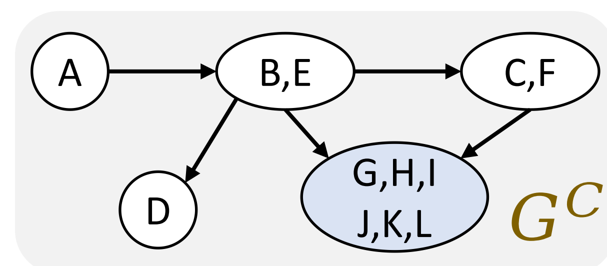


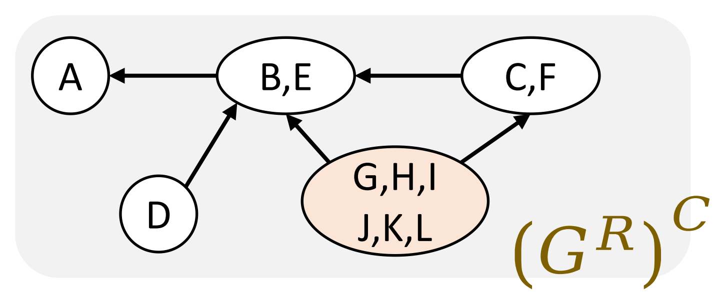


Computing SCC

- If we start from a node in a sink SCC, then we explore exactly nodes in that SCC and stop!
- **Good idea but two problems exist:**
(**1**) How to identify a node that is in a sink SCC?
(**2**) What to do when the first SCC is done?
- Compute G^R in $O(n+m)$ time,
then find a node in a *source* SCC in G^R!
- But how to find such a node?
- Do DFS in G^R, then the node with maximum finish time is guaranteed to be in source SCC.
- **Lemma:** for any edge $(u,v) \in E(G^R)$,
if $u \in C_i$ and $v \in C_j$,
then $\max_{u \in C_i}\{u.f\} > \max_{v \in C_j}\{v.f\}$.





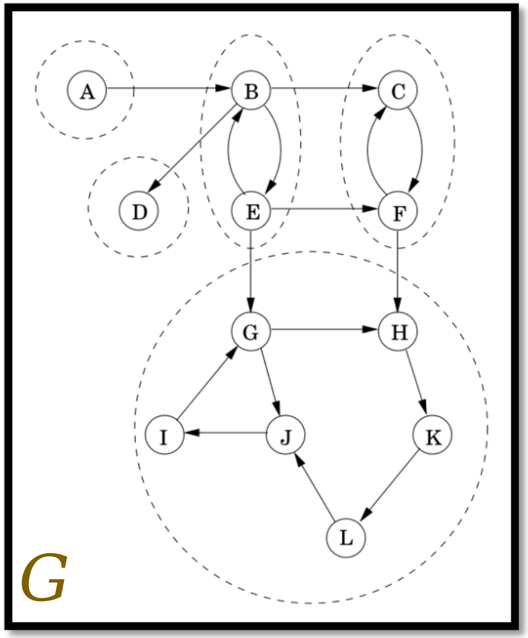


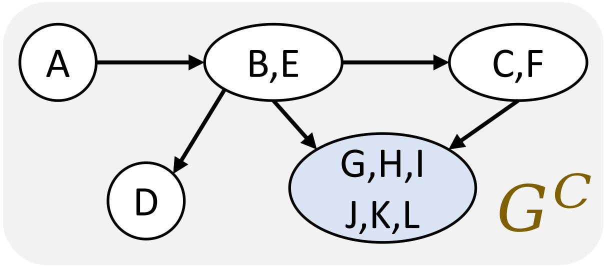
Computing SCC

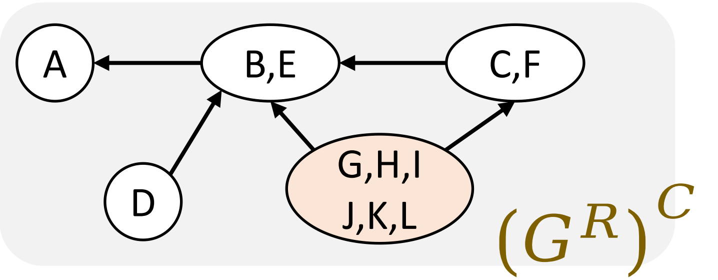
- **Lemma**: for any edge $(u, v) \in E(G^R)$, if $u \in C_i$ and $v \in C_j$,
  then $\max_{u \in C_i}\{u.f\} > \max_{v \in C_j}\{v.f\}$.
- **Proof**:
- Consider nodes in C_i and C_j, let w be the first node visited by DFS.
- If $w \in C_j$, then all nodes in C_j will be visited before any node in C_i is visited.
- In this case, the lemma clearly is true.
- If $w \in C_i$, at the time that DFS visits w, for any node in C_i and C_j, there is a white-path from w to that node.
- In this case, due to the white-path theorem, the lemma again holds.

Computing SCC

- If we DFS in G from a node in a sink SCC, then we explore exactly nodes in that SCC and stop!
- **Problem 1 in the strategy:**
  How to identify a node in a sink SCC of G?
- **Lemma:** for any edge $(u, v) \in E(G^R)$, if $u \in C_i$ and $v \in C_j$, then $\max_{u \in C_i}\{u.f\} > \max_{v \in C_j}\{v.f\}$.
- Compute G^R in $O(n+m)$ time, do DFS in G^R and find the node with max finish time.
  (This node is in a source SCC of G^R.)
- **Problem 2 in the strategy:**
  What to do when the first SCC is found?
- For remaining nodes in G, the node with max finish time (in DFS of G^R) is again in a sink SCC, for whatever remains of G.





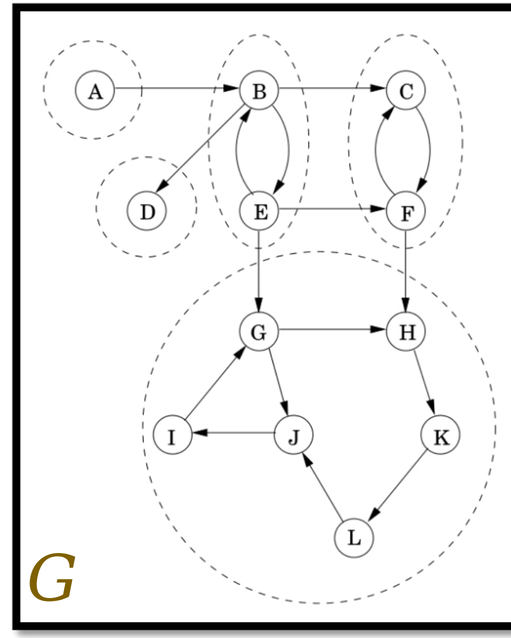


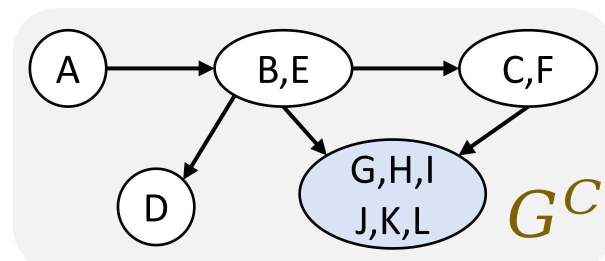
Computing SCC

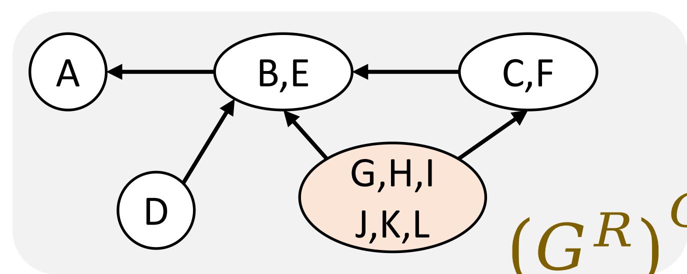
- **Algorithm Description:**
- Compute G^R.
- Run DFS on G^R and record finish times f.
- Run DFS on G, but in `DFSAall`, process nodes in decreasing order of f.
- Each DFS tree is a SCC of G.

- **Time Complexity:**
- $O(n+m)$ for computing G^R.
- Two passes of DFS, each costing $O(n+m)$.
- Thus total runtime is $O(n+m)$.

- **There are faster algorithms!**
- Tarjan's algorithm uses DFS only once.
- Still takes $O(n+m)$, but smaller constant.







*Tarjan's SCC Algorithm

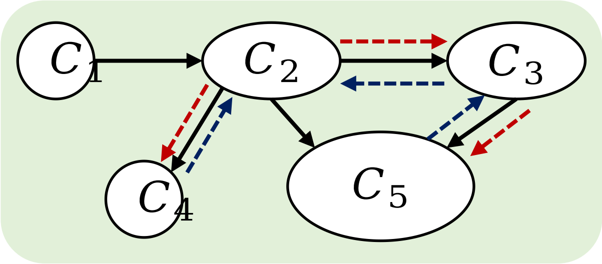
- If we start from a node in a sink SCC, then we explore exactly nodes in that SCC and stop!
- But how to find such a node?
- Previous algorithm's approach:
  A node in a source SCC in G^R must be in a sink SCC in G.
- Tarjan comes up with a method
  to identify a node in some sink SCC directly!

Robert Tarjan

American computer scientist and mathematician
Recipient of the 1986 Turing Award for "fundamental achievements in the design and analysis of algorithms and data structures"
(Such as linear time selection using median of medians, Fibonacci heap, first optimal analysis of the UnionFind data structure.)

Tarjan's SCC Algorithm

- Let's have a closer look at the order that DFS examines nodes:
  - First node in C_2 (root of C_2)
  - Some nodes in C_2
  - First node in C_3 (root of C_3)
  - Some nodes in C_3
  - First nodes in C_5 (root of C_5)
  - All other nodes in C_5 (C_5 is a sink SCC)
  - All other nodes in C_3 (C_3 becomes a sink SCC by then)
  - Some nodes in C_2
  - First nodes in C_4 (root of C_4)
  - All other nodes in C_4 (C_4 is a sink SCC)
  - All other nodes in C_2 (C_2 becomes a sink SCC by then)
  - First node in C_1 (root of C_1)
  - All other nodes in C_1 (C_1 becomes a sink SCC by then)



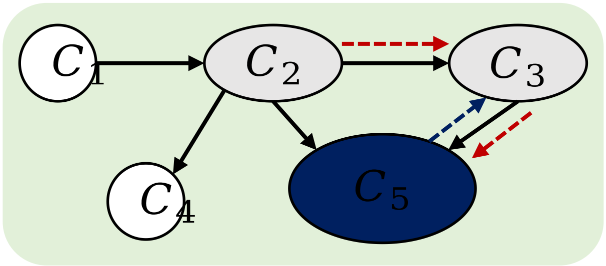
Tarjan's SCC Algorithm

- Let's have a closer look at the order that DFS examines nodes:

stack bottom

- First node in C_2 (root of C_2)
- Some nodes in C_2
- First node in C_3 (root of C_3)
- Some nodes in C_3
- First nodes in C_5 (root of C_5)
- All other nodes in C_5 (C_5 is a sink SCC)
- All other nodes in C_3 (C_3 bec
- Some nodes in C_2
- First nodes in C_4 (root of C_4)
- All other nodes in C_4 (C_4 is a sink SCC)
- All other nodes in C_2
- First node in C_1 (roo
- All other nodes in C_1

stack top



If we can identify root of C_5, call it r_5, then all nodes visited during DFS starting from r_5 are the nodes in C_5.

If we push a node to a stack when it is discovered, when DFS returns from r_5, all nodes above r_5 in the stack are in C_5 and can be popped!

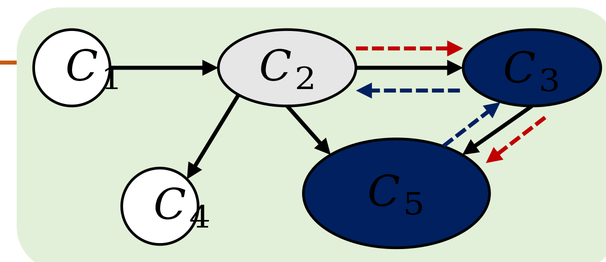
Tarjan's SCC Algorithm

- Let's have a closer look at the order that DFS examines nodes:

stack bottom

- First node in C_2 (root of C_2)
- Some nodes in C_2
- First node in C_3 (root of C_3)
- Some nodes in C_3
- First nodes in C_5 (root of C_5)
- All other nodes in C_5 (C_5 is a sink SCC)
- All other nodes in C_3 (C_3 becomes a sink SCC by then)
- Some nodes in C_2
- First nodes in C_4 (
- All other nodes in (
- All other nodes in C_2 (becomes a sink SCC by then)
- First node in C_1 (roo
- All other nodes in C_1

stack top



Given that we know nodes in C_5, if we can identify root of C_3, call it r_3, then all nodes not in C_5 visited during DFS starting from r_3 are the nodes in C_3!

If we push a node to a stack when it is discovered, when DFS returns from r_3, all nodes above r_3 in the stack are in C_3 and can be popped!

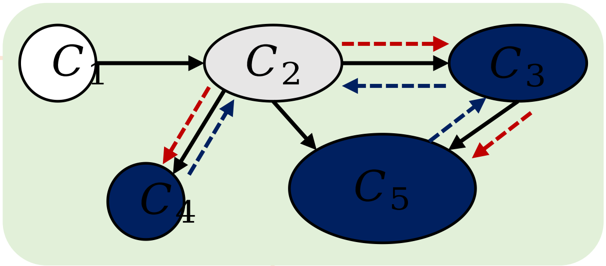
Tarjan's SCC Algorithm

- Let's have a closer look at the order that DFS examines nodes:

stack bottom

- First node in C_2 (root of C_2)
- Some nodes in C_2
- First node in C_3 (root of C_3)
- Some nodes in C_3
- First nodes in C_5 (root of C_5)
- All other nodes in
- All other nodes in
- Some nodes in C_2
- First nodes in C_4 (root of C_4)
- All other nodes in C_4 (C_4 is a sink SCC)
- All other nodes in C_2
- First node in C_1 (roo
- All other nodes in C_1

stack top



If we can identify root of C_4, call it r_4, then all nodes visited during DFS starting from r_4 are the nodes in C_4.

If we push a node to a stack when it is discovered, when DFS returns from r_4, all nodes above r_4 in the stack are in C_4 and can be popped!

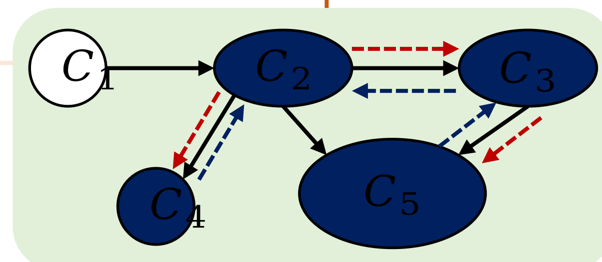
Tarjan's SCC Algorithm

- Let's have a closer look at the order that DFS examines nodes:

stack bottom

- First node in C_2 (root of C_2)
- Some nodes in C_2
    - First node in C_3 (root of C_3)
    - Some nodes in C_3
        - First nodes in C_5 (root of C_5)
        - All other nodes in C_5 (C_5 i
    - All other nodes in C_3 (C_3
- Some nodes in C_2
    - First nodes in C_4 (root of (
    - All other nodes in C_4 (C_4 is a sink SCC)
- All other nodes in C_2 (C_2 becomes a sink SCC by then)
- First node in C_1 (root
- All other nodes in C_1

stack top



Given that we know nodes in $C_5 \& C_3 \& C_4$, if we can identify root of C_2, call it r_2, then all nodes not in $C_5 \& C_3 \& C_4$ visited during DFS starting from r_2 are the nodes in C_2.

If we push a node to a stack when it is discovered, when DFS returns from r_2, all nodes above r_2 in the stack are in C_2 and can be popped!

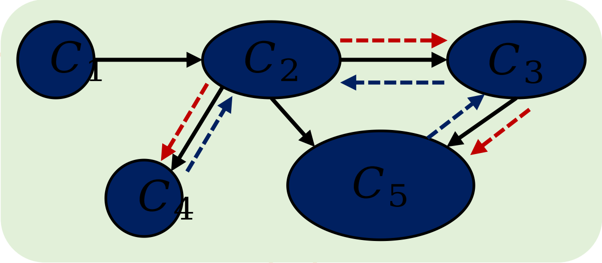
Tarjan's SCC Algorithm

- Let's have a closer look at the order that DFS examines nodes:

stack bottom

- First node in C_2 (root of C_2)
- Some nodes in C_2
- First node in C_3 (root of C_3)
- Some nodes in C_3
- First nodes in C_5 (root of C_5)
- All other nodes in C
- All other nodes in C
- Some nodes in C_2
- First nodes in C_4 (ro
- All other nodes in C_4
- All other nodes in C_2
- First node in C_1 (root of C_1)
- All other nodes in C_1 (C_1 becomes a sink SCC by then)

stack top



Given that we know nodes in C_2, if we can identify root of C_1, call it r_1, then all nodes not in C_1 visited during DFS starting from r_1 are the nodes in C_1

If we push a node to a stack when it is discovered, when DFS returns from r_1, all nodes above r_1 in the stack are in C_1 and can be popped!

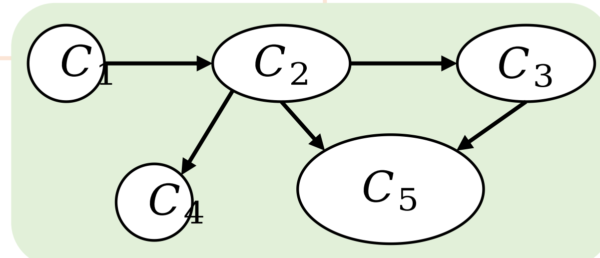
Tarjan's SCC Algorithm

- Let's have a closer look at the order that DFS examines nodes:

stack bottom

- First node in C_2 (root of C_2)
- Some nodes in C_2
- First node in C_3 (root of C_3)
- Some nodes in C_3
- First nodes in C_5 (root of C_5)
- All other nodes in C_5 (C_5 is a sink SCC)
- All other nodes in C
- Some nodes in C_2
- First nodes in C_4 (ro
- All other nodes in C_4
- All other nodes in C_2 (C_2 becomes a sink SCC by then)
- First node in C_1 (root of C
- All other nodes in C_1 (C_1 becomes a sink SCC by then)

stack top



For each SCC C_i, let r_i be its root.
If we push a node to a stack when it is discovered, when DFS returns from r_i, all nodes above r_i in the stack are in C_i and can be popped!

But how to identify each root r_i?

Tarjan's method to identify root of SCC

- Fix some DFS process, for each vertex v, let C_v be the SCC that v is in. Then, $low(v)$ is the smallest discovery time among all nodes in C_v that are reachable from v via a path of tree edges followed by at most one non-tree edge.
- By definition, $low(v) \leq v.d$ as v is reachable from itself.

- **Lemma:** Node v is the root of a SCC iff $low(v) = v.d$.

Tarjan’s method to identify root of SCC

- Fix some DFS process, for each vertex v, let C_v be the SCC that v is in. Then, $low(v)$ is the smallest discovery time among all nodes in C_v that are reachable from v via a path of tree edges followed by at most one non-tree edge.
- **Lemma:** Node v is the root of a SCC iff $low(v) = v.d$.
- **Proof: [==>]** (the easy direction)
- If v is the root of C_v, then it is the first discovered node in C_v.
- Hence v has the smallest discovery time among all nodes in C_v.
- By definition of $low(v)$, clearly $low(v) = v.d$.

Tarjan's method to identify root of SCC

- Fix some DFS process, for each vertex v, let C_v be the SCC that v is in. Then, $low(v)$ is the smallest discovery time among all nodes in C_v that are reachable from v via a path of tree edges followed by at most one non-tree edge.
- **Lemma:** Node v is the root of a SCC iff $low(v) = v.d$.
- **Proof: [<==]** (the hard direction)
- For the sake of contradiction assume $x \neq v$ is the root of C_v.
  (I.e., x is the first discovered node in C_v.)
- Let $x' \neq v$ be v's parent in the DFS tree. Since C_v is a SCC, v can reach all nodes in C_v, including the ones on path $x \to x'$. Thus, when executing DFS from v, it will examine a path containing zero or more tree edges and then a back edge pointing to some node x'' in path $x \to x'$.
- But this means $low(v) < v.d$ since $low(v) \leq x''.d < v.d$. Contradiction!

Tarjan's SCC Algorithm

For each SCC C_i, let r_i be its root. If we push a node to a stack when it is discovered, when DFS returns from r_i, all nodes above r_i in the stack are in C_i.

Let $low(v)$ be the smallest discovery time among all nodes in C_v that are reachable from v via a path of tree edges followed by at most one non-tree edge.
Lemma: Node v is the root of a SCC iff $low(v) = v.d$.

Tarjan(G):

```
time = 0
Let S be a stack
for (each node v)
  v.root = NIL
  v.visited = false
for (each node v)
  if (!v.visited)
    TarjanDFS(v)
```

TarjanDFS(v):

```
v.visited = true, time = time+1
v.d = time, v.low = v.d
S.push(v)
for (each edge (v,w))
  if (!w.visited)  // tree edge
    TarjanDFS(w)
    v.low = min(v.low, w.low)
  else if (w.root == NIL)  // non tree edge in C_v
    v.low = min(v.low, w.d)
                  .d)

                ), w.root = v
```

Time complexity is $O(m+n)$.
(One DFS pass, and push/pop once for each node.)

Reading

- [CLRS] Ch.22 (22.4-22.5)
- *If you want to know more about Tarjan's SCC algorithm:
  - [Erickson v1] Ch.6 (6.6)
  - Tarjan's original paper entitled "Depth-First Search and Linear Graph Algorithms" (https://doi.org/10.1137/0201010)