Recursive DFS

A recursive depth-first search from u

procedure $\text{dfs}_{(V,E)}(\ u : V)$
  precond: u is white
  colour u grey
  previsit(u)
  for $v \mid u \to v$ do
    if v is white
      $\text{dfs}_{(V,E)}(\ v\ )$
    end if
  end for
  postvisit(u)
  colour u black

Postvisit and previsit are “hook methods” that we can fill in for various applications.

To DFS an entire graph

DFS (V, E)
turn all nodes in V white
while there is a white node
  let u be any white node
  $\text{dfs}_{(V,E)}(u)$

If $u \xrightarrow{*} v$ but not $v \xrightarrow{*} u$, then, when u is visited, either

- v is already black when u is previsited, or
- v will be found between the previsit and postvisit of u, and thus turned black before u.

In either case, u will be postvisited after v.

Notation: If $u \stackrel{*}{\to} v$ but not $v \stackrel{*}{\to} u$, we'll write $u \prec v$ and say u *preceeds* v.

Theorem: If we run DFS (V, E), then for any nodes u and v such that $u \prec v$, u will be postvisited after v.

Topological Sort

Suppose we have a number of tasks to do but only one worker (or machine) to do them.

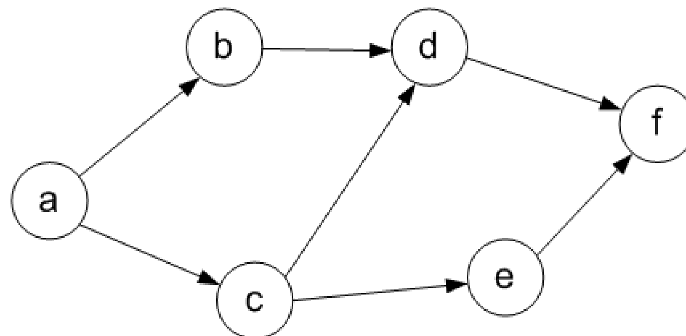
Represent each instance with a directed acyclic graph (DAG)

Nodes Each task is represented by a node

Edges If task u must be done before task v, we have an edge $u \to v$.

We will presume that the graph is acyclic

Problem: Find an order that the tasks can be done in.



Can be done in several orders. E.g.

$$[a, b, c, d, e, f]$$
$$[a, c, d, f, e, b]$$

A topological sort algorithm

On previsit do nothing.

On postvisit push u onto stack S

topologicalSort$_{(V,E)}$(var S : Stack$[V]$)
  $S :=$ new Stack$[V]$
  colour all nodes in V white
  while there is a white node do
    let u be any white node
    dfs$_{(V,E)}(u)$
  end while

If $u \prec v$, then, when u is searched, either

- v is already on the stack, or
- v will be found between the previsit and postvisit of u, and thus placed on the stack before u .

Postcondition: For any nodes u and v such that $u \prec v$, u will be above v on S.

And so, for an acyclic graph, if u precedes v in the graph then u will be closer to the top of the stack.

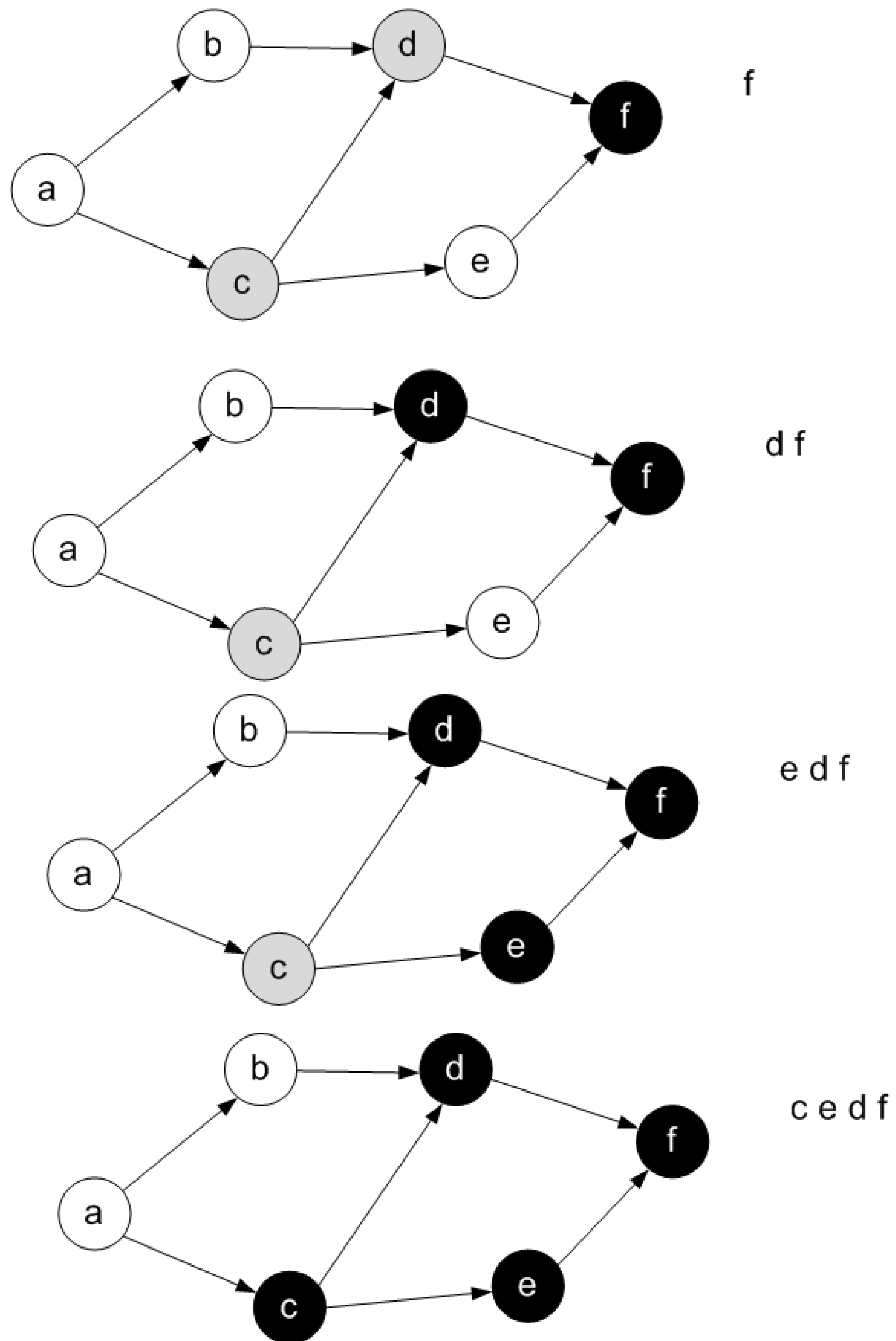
At the end of the algorithm, the nodes on the stack are in order (from top to bottom) so that the top task/node can be done first, then the next from the top etc.

Why it works.

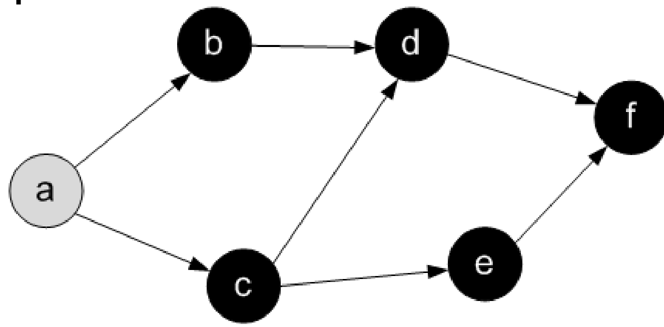
- When a node is turned black, all nodes that must be done later are already on the stack.

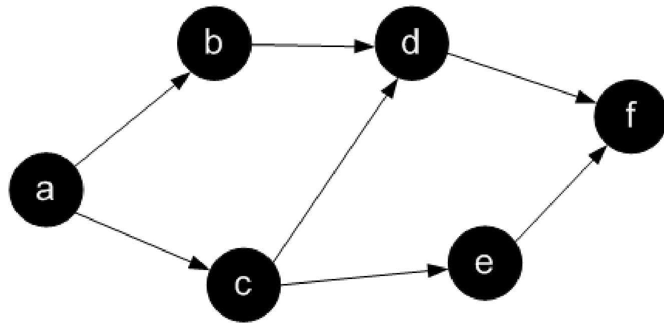
Time $\Theta(|V| + |E|)$

Example. Suppose we start with node c.



Next a is picked.

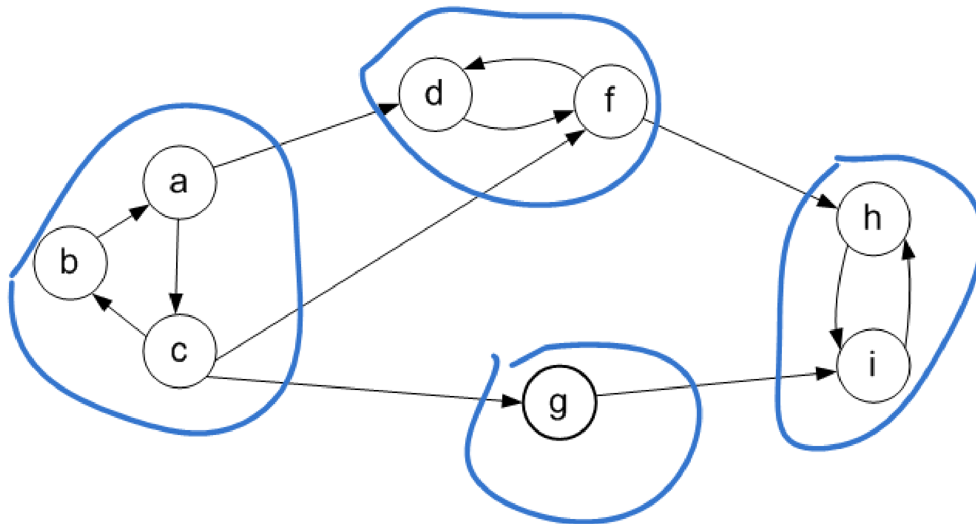




Strongly connected components

Suppose we have a directed graph with cycles.

If $u \xrightarrow{*} v$ and $v \xrightarrow{*} u$ then u and v are said to be in the same **strongly connected component.**



The strongly connected components are $\{a, b, c\}$, $\{d, f\}$, $\{g\}$ and $\{h, i\}$.

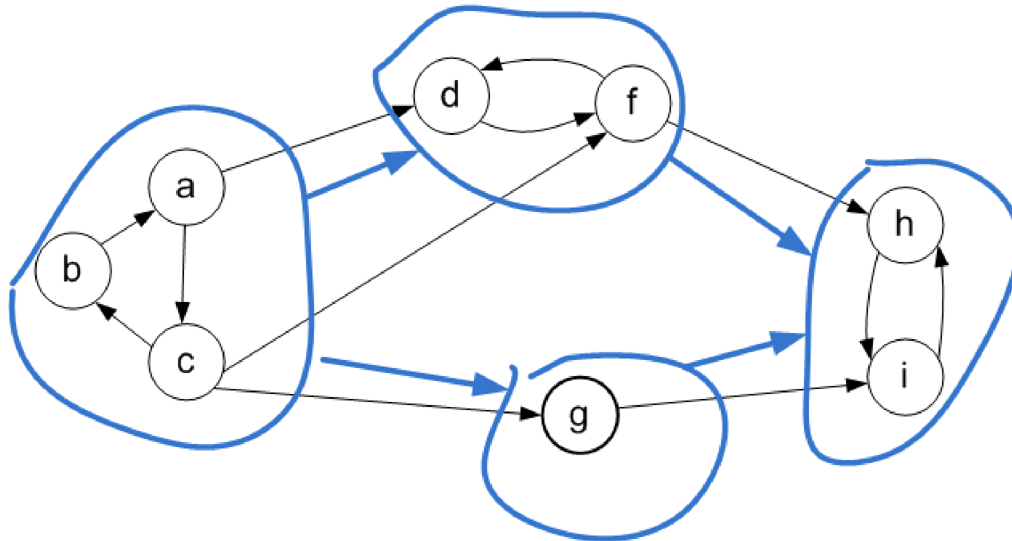
Finding strongly connected components has a number of applications.

For example in compiling, if each node represents a procedure and each arrow represents a potential call, we can discover which subroutines are recursive by finding SCCs.

Kosaraju's algorithm

If the graph is a DAG, each node is in its own SCC.

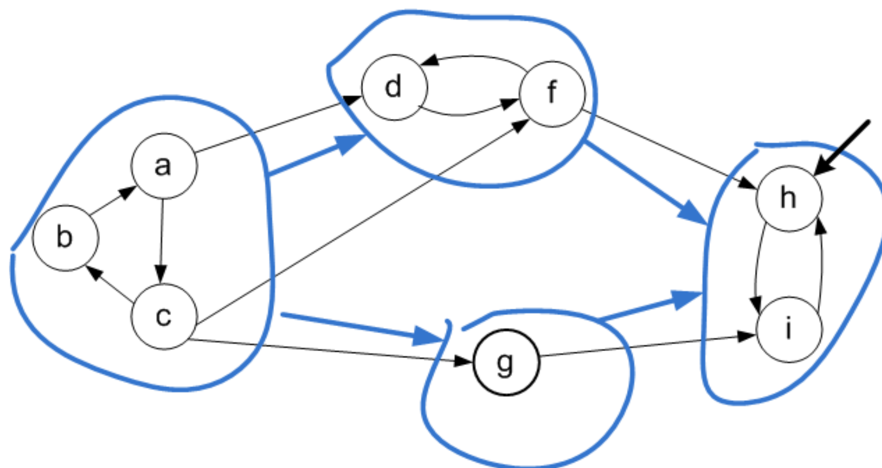
For any graph, the strongly connected components themselves form a DAG.

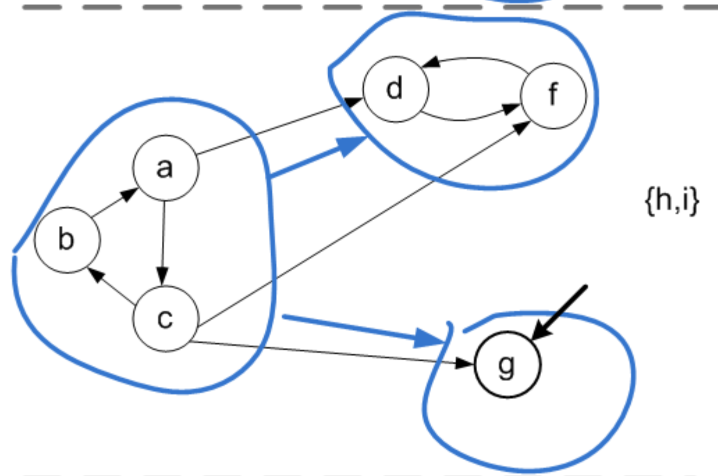


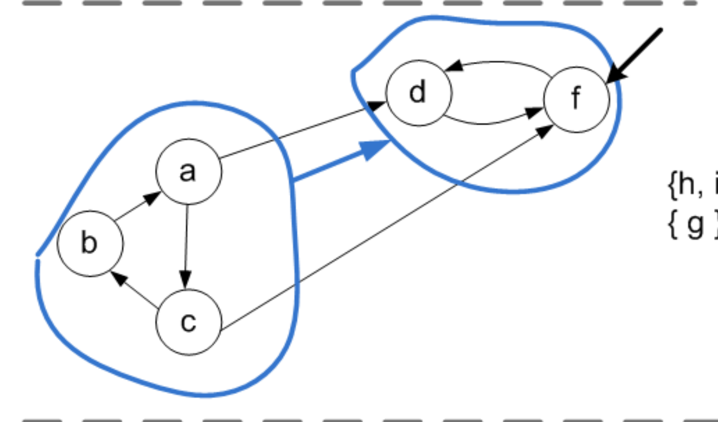
Call an SCC that has no out edges in this DAG a "terminal" SCC. There must be at least one.

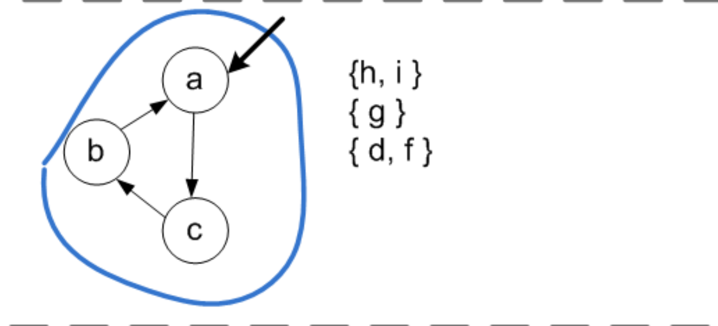
If we could somehow find a node in a "terminal" SCC and do a graph search from that node, we would find all (and only) nodes in that SCC.

Then, if we remove all those nodes and repeat the process with another (now) terminal SCC, we get another SCC and so on.









{h, i }
{ g }
{ d, f }
{ a, b, c }

A ‘first stab’ algorithm

Find all SCCs: first stab
    while there are nodes left do
        Somehow find a node u in a terminal SCC
        Find all nodes reachable from u
        Let U be all nodes found in that search
        Output U and delete all nodes in U from G

How can we find a node in a terminal SCC?

Topological sorting to the rescue

If we do a TopologicalSort [from a few slides back] to produce a stack S

- For any two nodes u and v in different components,
- if u is in a component that precedes v's component,
- then $u \prec v$
- and so u is closer to the top of the stack than v.

Thus after topological sort, the node on the top of the stack is sure to be in an *initial* component. (I.e. a component with no incoming edges.)

Transposing the graph

Note that, if we turn all edges around (transpose the graph) the SCCs remain the same.

Components that are initial in the original graph are terminal in the transposed graph.

This suggests a 'second stab' algorithm

while there are nodes left do
    Do a topological sort of G
    Let u be the node at the top of the stack.
    $\{u$ is in an initial SCC of $G\}$
    Let G' be the transpose of G.
    $\{u$ is in a terminal SCC of $G'\}$
    Let U be all nodes reachable from u in G'
    Output U and delete all nodes in U from G

We can optimize this algorithm by only doing the topological sort and the transpose once.

Kosaraju's algorithm

This gives us the following algorithm

Kosaraju's strongly connected components algorithm
var $S : \mathrm{Stack}[V]$
topologicalSort$_{(V,E)}(S)$
{ all nodes are on S }
var $(V', E') := \mathrm{transpose}(V, E)$
{ inv I (see below)}
while S is not empty do
    val $u := S.\mathrm{top}()$
    { u is in a terminal component of (V', E') }
    val $U :=$ all nodes reachable from u in (V', E')
    { U is a terminal component of (V', E') }
    output U
    remove each node in U from S and also from (V', E')

The invariant I is

- The nodes in V' are the same as the nodes in S.
- For any two nodes v and w in (V', E'),
  - if, in (V', E'), $w \prec v$,
  - then, in the original graph, $v \prec w$
  - and (so) v is closer to the top of stack S than w.

From the invariant: if S is nonempty, the top node of S is in a terminal component of (V', E')

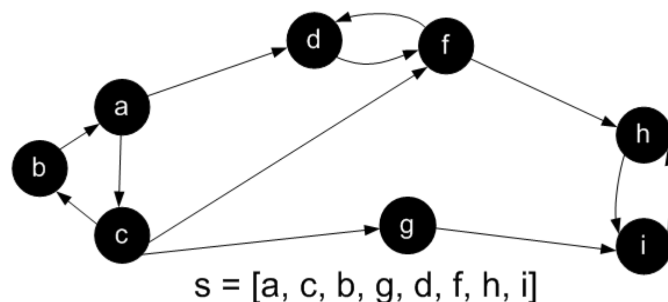
We've seen that the topological sort and transpose establish this invariant.

It remains to prove that removing a terminal component from both S and (V', E') preserves the invariant.

- Essentially: removing U from (V', E') does not affect the reachability relations between the remaining nodes since U is terminal. Hence the information contained in S is still valid.

Example

First a topological sort



Now the main loop of the algorithm.

Each snapshot (except the last) shows the state of (V', E') after U has been computed.

The nodes in U are thus black.