EXTENDING DFS

Several algorithms are built on top of DFS, i.e., using DFS traversal order. They can also be implemented using DFS as a skeleton.

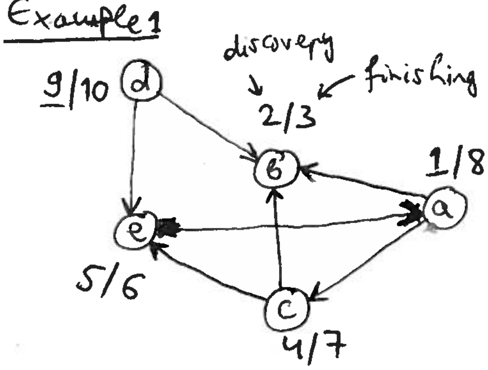
DISCOVERY AND FINISHING TIMES

DISCOVERY refers to an event of "entering" a node for the first time. FINISHING refers to "leaving" a node for the last time. We introduce "timer" and increment it on each event and record discovery and finishing times of each ~~node~~ node in DFS traversal.

```
DFS-visit(V, Adj, s):
    time = time + 1
    dt[s] = time   // discovery time
    for v in Adj[s]:
        if v not in parent:
            parent[v] = s
            DFS-visit(V, Adj, v)
    time = time + 1
    ft[s] = time   // finishing time
```

```
DFS(V, Adj):
    parent = {}
    dt = {}
    ft = {}
    time = 0    // init time to 0
    for s in V:
        if s not in parent:
            parent[s] = None
            DFS-visit(V, Adj, s)
```

Example 1



Order of events:
start DFS from 'a'
① discover 'a'
call DFS on 'b'
② discover 'b' | stuck
③ finish 'b'
back to 'a'
call DFS on 'c'
④ discover 'c'
check edge (c, b): 'b' already visited
call DFS on 'e'
⑤ discover 'e'
check edge (e, a): 'a' already visited
⑥ finish 'e'
Back to 'c' : stuck
⑦ finish 'c'
back to 'a' ; stuck
⑧ finish 'a'

start DFS from 'd'
⑨ discover 'd'
check edges (d, b), (d, e)
b, e already visited
⑩ finish 'd'

PARENTHESIS THEOREM

Observe time intervals between discovery and finishing time for each node. Then, for any two ~~vertices~~ nodes u and v one of the following is true:

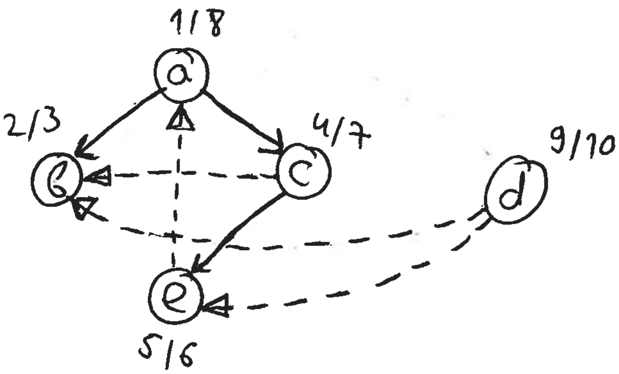
1) intervals [dt[u], ft[u]] and [dt[v], ft[v]] are disjoint if u and v are <u>not</u> in an ancestral relationship in DFS tree.
(u and v are in ancestral relationship if either u is ancestor of v, or v is ancestor of u)

2) intervals $[dt[u], ft[u]]$ and $[dt[v], ft[v]]$ are nested if either u or v is ancestor of the other node; for example, if u is ancestor of v, then $dt[u] < dt[v] < ft[v] < ft[u]$. (again, we are looking at DFS tree)

Informal proof: Note that any node u is "discovered" right before and "finished" right after the subtree of DFS tree rooted at u is visited. Therefore, if node v is in that subtree, u is ancestor of v, and v is visited entirely between discovery and finishing of u. Case when v is ancestor of u is analogous. On the other hand, if u and v are unrelated, let u be visited before v (without loss of generality), then v cannot be discovered before u is finished, because v would be in a subtree of u otherwise. Therefore, the two intervals must be disjoint in this case.

Example 2

DFS tree as a result of DFS traversal in example 1 will be:



full lines - tree edges
dashed lines - forward, back or cross edges
e.g., (c,b), (d,b) and (d,e) are cross edges
(e,a) is a back edge

~~Note that~~ Observe that $[dt, ft]$ intervals of pairs of nodes that are in ancestral relationship are nested, such as: a and b, a and c, a and e, c and e. On the other hand, intervals of other node pairs are disjoint, e.g., b and c, b and e, a and d, b and d, etc.

For example, you can use discovery and finishing times to find a relationship between any two nodes in a DFS tree (or any tree) in $O(1)$, after the traversal is performed.

TOPOLOGICAL SORT REVISITED

A possible topological order of nodes in a DAG is given by a reverse order of finishing times.

```
TopSort(V, Adj):
    DFS(V, Adj)
    ordered_nodes = "sort nodes in reverse order of finishing times"
    return ordered_nodes
```

Note that the sorting operation can be performed in $O(|V|)$ by going through all nodes once and putting each node at a proper place in ordered_nodes array. (Assume that we compute only finishing times, and NOT compute discovery time, and, therefore, finishing times will take values $1, \ldots, |V|$.)

```
TopSort(V, Adj):
    DFS(V, Adj)
    ordered_nodes = "list of |V| elements"   //initialization
    for each v in V:
        ordered_nodes[ft[v]] = v
    return ordered_nodes
```

Alternatively, if we use DFS only to get topological order, we can integrate construction of ordered_nodes array within DFS-visit. This is another example of implementation using DFS skeleton.

```
DFS-visit(V, Adj, s):
    for v in Adj[s]:
        if v not in parent:
            parent[v] = s
            DFS-visit(V, Adj, v)
    ordered_nodes.append(s)

//add node to the list when "leaving"
```

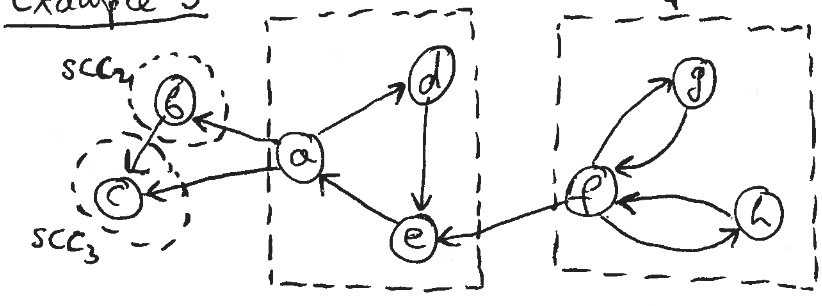
```
DFS-TopSort(V, Adj):
    parent = {}
    ordered_nodes = []
    for s in V:
        if s not in parent:
            parent[s] = None
            DFS-visit(V, Adj, s)

    return ordered_nodes.reverse()
```

STRONGLY-CONNECTED COMPONENTS (SCC)

Strongly-connected component of a directed graph is a maximal (that cannot be extended) subset of nodes such that there is a path from any node to any other node in the subset.

Example 3



four SCCs:

SCC_1: a, d, e

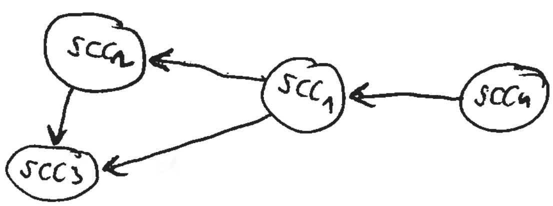
SCC_2: b

SCC_3: c

SCC_4: f, g, h

~~If we create a graph~~

If we substitute each component with a node and put an edge from SCC_i to SCC_j if there is an edge from any node in SCC_i to any node in SCC_j in the original graph, we get "inter-component" graph. For a graph in example 3, "inter-component graph would be:



Note that such a graph is always DAG, because, if there was a cycle, all components on a cycle could be merged into a single component, which is contradiction to the assumption that components are maximal.

Let $\rightleftarrows$ denote a relationship between nodes, such that $u \rightleftarrows v$ if and only if there is a path from u to v ($u \overset{p}{\rightarrow} v$) and a path from v to u ($v \overset{p}{\rightarrow} u$). Then, $\rightleftarrows$ is transitive. In other words: $u \rightleftarrows v$ and $v \rightleftarrows w \Rightarrow u \rightleftarrows w$. This is obviously true because there is a path from u to w through v ($u \overset{p}{\rightarrow} v \overset{p}{\rightarrow} w$) and from w to u through v ($w \overset{p}{\rightarrow} v \overset{p}{\rightarrow} u$). Also $\rightleftarrows$ is obviously symmetric.

As a consequence, each node belongs only to one strongly-connected component, because, if a node did belong to two or more components, they could be merged into a single component (by transitivity, there would be a path from any node to any other node through the common node). Therefore, SCCs are vertex-disjoint.

Example 4

How many SCC's are in a DAG?

– There are |V| SCCs in a DAG, because no two nodes can belong to the same SCC since that would mean that there is a cycle in a graph.

SCC ALGORITHM

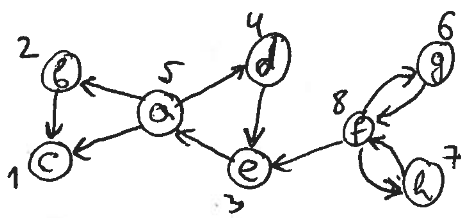
One way to find SCCs in a graph is given by the following high-level algorithm:

```
SCC(V, Adj):
    call DFS(V, Adj) that computes finishing times
    V' = nodes sorted in decreasing order of finishing time
    Adj^T = Adj transposed
    call DFS(V', Adj^T) that assigns each DFS-tree to a component
                        (each DFS tree would correspond to a single SCC)
```

Note that V' is not necessarily in a topological order, because a topological order does not exist if a graph contains cycles.

Example 5: Illustration of SCC algorithm on a graph from example 3

~~After~~ In the first step, we find finishing times:



start DFS from 'a'
call DFS on 'b'
call DFS on 'c'
① finish 'c'
back to 'b'
② finish 'b'

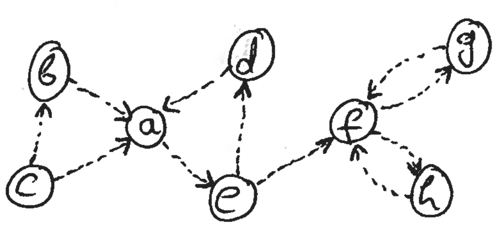
back to 'a'
call DFS on 'd'
call DFS on 'e'
③ finish 'e'
back to 'd'
④ finish 'd'
back to 'a'
⑤ finish 'a'

start DFS from 'f'
call DFS on 'g'
⑥ finish 'g'
back to 'f'
call DFS on 'h'
⑦ finish 'h'
back to 'f'
⑧ finish 'f'

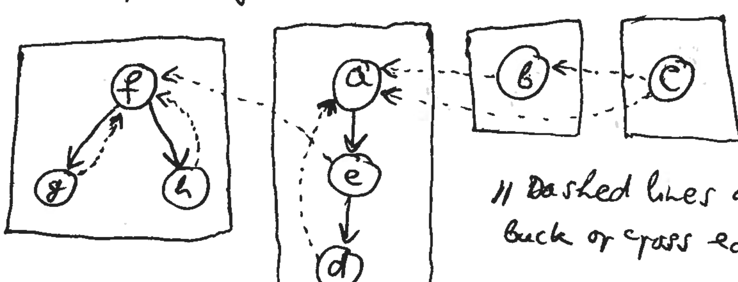
Nodes in reverse finishing time order:

$V' = f, h, g, a, d, e, b, c$

Transposed graph is:



DFS on a transposed graph in order given by V' yield the following trees:



// Dashed lines are forward, back or cross edges.

Note that each DFS tree corresponds to a strongly-connected component in the original graph (look at example 3).

SCC ALGORITHM IN DETAIL (example realization)

Here is a possible more detailed algorithm for finding strongly-connected components. Instead of building components directly (as sets of vertices that belong to same components), we will label each vertex with an index of a component to which it belongs. Components can easily be recovered afterwards from this labels.

```
DFS-visit(V,Adj,s):
    for v in Adj[s]:
        if v not in parent:
            parent[v]=s
            DFS-visit(V,Adj,v)
    ordered_nodes.append(s)
```

```
DFS-visit-component(V,Adj,s):
    comp_index[s] = current_index
    for v in Adj[s]:
        if v not in parent:
            parent[v]=s
            DFS-visit-component(V,Adj,v)
```

```
transpose(Adj,V):
    Adj_t = {}
    for v in V:
        Adj_t[v]=[]
    for v in V:
        for w in Adj[v]:
            Adj_t[w].append(v)
    return Adj_t
```

```
SCC(V,Adj):          <- # stage 1
    parent={}
    ordered_nodes=[]
    for s in V:
        if s not in parent:
            parent[s] = None
            DFS-visit(V,Adj,s)
    # stage 2
    parent={}
    comp_index={}
    current_index=0
    Adj_t= transpose(Adj,V)
    for s in ordered_nodes.reverse()
        if s not in parent:
            current_index += 1
            parent[s]=None
            DFS-visit-component(V,Adj_t,s
```

Note that during traversal of one DFS-tree, current_index does not change, so all vertices of that tree will be labeled with a same number (same component index). On the other hand, current_index will be incremented for each DFS-tree.

SCC ALGORITHM – PROOF OF CORRECTNESS (informal proof)

Lemma: Let ft[u] and ft[v] be finishing times of nodes u and v after stage 1 (first DFS traversal), and let ft[u] > ft[v] without loss of generality. Then, u and v are in the same SCC if and only if there is a path from u to v in a transposed graph.

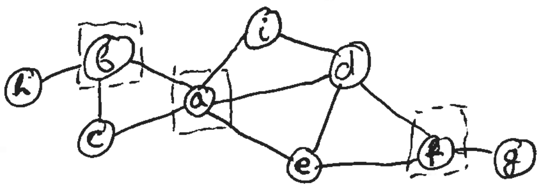
informal proof: If there is no path from u to v in the transposed graph, then there is no path from v to u in the original graph, so u and v are not in the same SCC.

Now suppose there is a path from u to v in the transposed graph, which means there is a path from v to u in the original graph. If v was traversed before u, then it would be ft[v] > ft[u], because u would be in a subtree of v (there is path v ⇝ u). So this is not possible. Therefore, u is visited before v. If there is no path from u to v, then u would be finished before v ft[u] < ft[v], which contradicts with ft[u] > ft[v]. Therefore, there is both u ⇝ v and v ⇝ u, so they must be in the same component.

ARTICULATION POINTS

Let $G=(V,E)$ be a connected undirected graph. An articulation point of G is a vertex whose removal disconnects G.

Example 6



Articulation points are: a, b, f

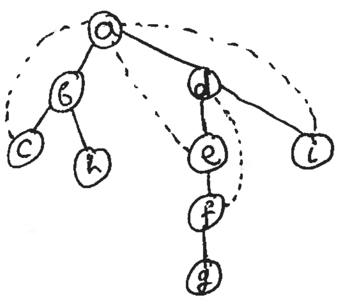
Naive Algorithm

```
ART-POINTS-NAIVE (V, Adj):
    for v in V:
        remove v from the graph (and all edges adjacent to it)
        run DFS alg. to compute number of components
        if num_components > 1:
            v is articulation point
```

Runs in $O(V(V+E))$ time.

We can make $O(V+E)$ algorithm. Let's first show some properties.

Example 7 - DFS tree of the graph from example 6.



dashed edges are back edges (recall that DFS tree of undirected graph doesn't have cross edges)

Lemma 1: ~~A vertex s is an articulation point if~~ If vertex s is not the root of the DFS tree, then s is an articulation point if and only if s has a child v such that there is no back edge from v or any descendant of v (i.e., any node in a subtree rooted in v) to a proper ancestor of v. (proper ancestor is any ancestor other than v itself)

Proof (informal): If such a child v exists, then, by removing s there is no path from v to any ancestor of v (only subtree rooted in v is reachable from v), because there are no edges going from the subtree rooted in v to the rest of the graph (there are no back edges that leave this subtree). Therefore, s is an articulation point. On the other hand, if no such v exists, then, from any subtree rooted in a child v, there is a path through a back edge to an ancestor of s, and therefore to the root. So, after removing s, graph is still connected $\Rightarrow$ s is not an articulation point.

Lemma 2: The root of the DFS tree is an articulation point if and only if it has more than one child in a DFS tree.

Informal proof: If the root has one child v, then by removing the root, the graph is obviously still connected (v is connected to all other nodes). If the root has more than one child, then after it is removed, its children (and their subtrees) are disconnected from each other, because there cannot exist cross edges between them (DFS property). In this case, the root is an articulation point.

Example 8

Apply lemma 1 and lemma 2 to find articulation point of the graph from example 6 given its DFS tree in example 7.
Node 'a', as a root, is an articulation point because it has 2 children ('b' and 'd').
Node 'b' has a child 'h' that doesn't have back edges, so 'b' is an articulation point.
Similarly, 'f' is an articulation point because its child 'f' has no back edges.
Other nodes are not articulation points. For example, 'd' has children 'e' and 'i' whose both subtrees has a back edge to 'a' which is an ancestor of 'd'.

In order to check whether a subtree rooted in v that is a child of s has a back edge to an proper ancestor of s, we can use discovery time.

Lemma 3: If u is an ancestor of v, then $dt[u] < dt[v]$.
Proof: This is a direct consequence of the parenthesis theorem.
Note that the opposite is not true, i.e. from $dt[u] < dt[v]$ doesn't necessarily follow that u is an ancestor of v, u and v might be in different subtrees. For example, nodes 'b' is not an ancestor of node 'd' in example 7, although 'b' is visited before 'd' and ~~therefore~~ $dt['b'] < dt['d']$.
However, in our algorithm, we will always compare discovery times of nodes that are in ancestral relationship.

We define, for each node s, $low[s]$ to be a discovery time of the "oldest" ancestor of s to which there is back edge either from s or any of its descendant in DFS tree (any node in a subtree rooted in s). Since this ancestor is discovered before all nodes of a subtree rooted in s and before all other ancestors of s reachable via back edge from this subtree, it will have the smallest discovery time. Therefore, we can

Compute low[s] in the following way (if there is no a proper ancestor of s reachable via back edge, we will set low[s]=dt[s]):

$$low[s] = \min \begin{cases} dt[s] \\ dt[w], \text{ for all nodes } w \text{ such that there is a back edge } (u,w) \text{ from} \\ \qquad \text{any node } u \text{ in the subtree of } s. \end{cases}$$

<u>Lemma 4:</u> Node s, not root, is an articulation point if and only if $low[v] \geq dt[s]$ for some node v that is a child of s.

<u>Informal Proof:</u> $low[v] < dt[s]$ is equivalent condition to the one in lemma 1, i.e. that there is no back edge from v's subtree to a proper ancest of s. (use lemma 3 to show that). Therefore, lemma 4 follows.

Computing low[s] according to the formula above is inefficient, because we would need to traverse the whole subtree of s again for each s. However, assuming that low[v] is already computed for each child v of s, we can give the following recursive formula:

$$low[s] = \min \begin{cases} dt[s] \\ dt[w], \text{ for all nodes } w \text{ such that } (v,w) \text{ is a back edge} \\ low[v], \text{ for all children } v \text{ of } s \end{cases}$$

Using this formula, articulation points can be found efficiently in one DFS "pass" ($O(V+E)$ time). Note that because we assume G is connected, $|E| > |V|-1$, and therefore $O(V+E)=O(E)$.

```
parent={}, dt={}, low={}
time=0, art_points=[]
choose some starting node root
      parent[root]=None
DFS-visit(V, Adj, root)
_______
DFS-visit(V, Adj, s):
    time=time+1
    dt[s]=time
    low[s]=dt[s]  //default
    num_children=0
    is_art_point=False
    for v in Adj[s]:
        if v not in parent:
            parent[v]=s
            num_children +=1
            DFS-visit(V, Adj, v)
            if low[v] >= dt[s]:  // Lemma 1
                is_art_point=True
            low[s]=min(low[s],low[v])
        else:
            if parent[s] != v and dt[v]<dt[s]:  //Back edge
                low[s]=min(low[s],dt[v])
    if s == root and num_children>1:  // Lemma 2
        is_art_point=True
    if is_art_point:
        art_points.append(s)
```

set to True if s is artic. point