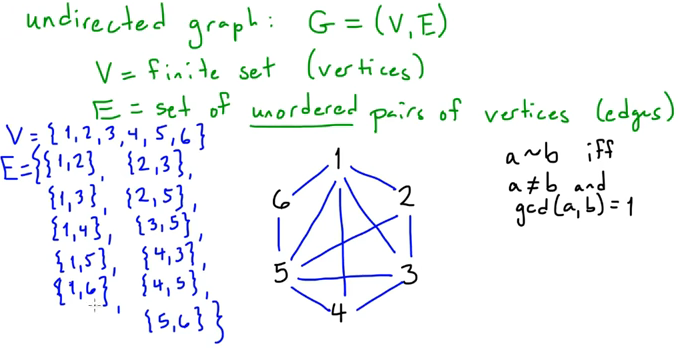
GR1: Strongly Connected Components

Notes for CS-8803-GA: Introduction to Graduate Algorithms

Georgia Tech (Dr. Eric Vigoda), Fall 2017

as recorded by Brent Wagenseller

**Chris Pyrby’s Graph Refresher**

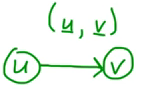
Recall that a graph G is typically a collection of vertices and edges – in other words, G = (V, E). In this sense, V is actually a set of vertices, and E is a set of unordered pairs of vertices (also known as Edges). Note that V and E are two separate sets (although E is comprised of pairs of vertices from V). Example:  


Note that in an undirected graph, the {} are used as this denotes unordered.

A singleton set would be a vertex that loops back to itself and is represented, as an example, {1}, {2}, etc.

**Directed Graphs**

Usually, a ‘graph’ is undirected unless specifically called a ‘**directed graph**’ (or a **digraph**). A directed graph lists ordered pairs (so their order matters). Instead of listing the pair with {}, it uses () instead which suggests which vertex is the origin; its represented by an arrow, for example,



There can be many edges that connect the same two points; therefore, when this happens, E technically is no longer a set, but is now a multiset (because ((1,2), (1, 2)) can exist in a **multiset**, whereas in a set it could not). Graphs that use multisets are called **multigraphs**. A multigraph can be directed or undirected; note that (a, b) and (b, a) can still be a set in a directed graph as they are NOT the same!

If its not specified,

* Graphs are assumed to be undirected
* Edges are assumed to be a set, not a multiset.

**Pryby – Other Graph Terms**

(as found on <https://www.youtube.com/watch?v=cJ8UIDAMccE> )

Verticies are **adjacent** (also: **neighbors**) if there is an edge between them. This is represented as: a ~ b

b ∈ N(a) = set of vertices with an edge from a to b; neighborhood of a

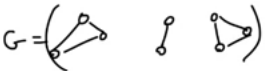
* + - ‘Graph Theory’ is basically just nodes (V, verticies, which is a set) connected via edges (E, also a set) in a larger graph (G)
      * E is unordered
      * Typically, {} indicates unordered
      * Also if the item should be a set and there is only one item in the brackets – a la {1} – it actually means {1,1}
    - the cardinality of G (|G|) is the number of nodes
    - the ‘**degree**’ of any particular node (Vn, for example) just means the number of edges (or connections) to other nodes
    - vertex A is **adjacent** to vertex B IF it has a connecting edge
      * In an undirected graph…
        + This is typically represented as a ~ b
        + These are known as neighbors

to specify neighbors, we can say

b ∈ N(a) and a ∈ N(b)

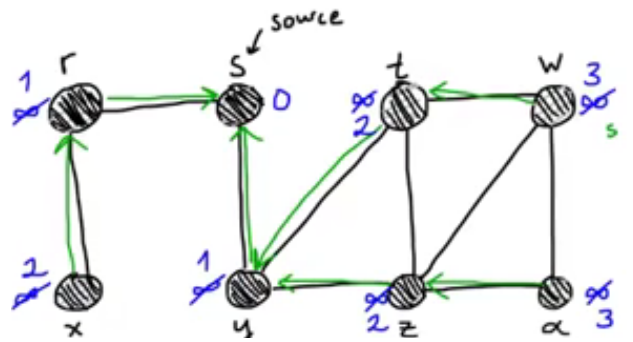
‘N’ can be used to specify neighbors

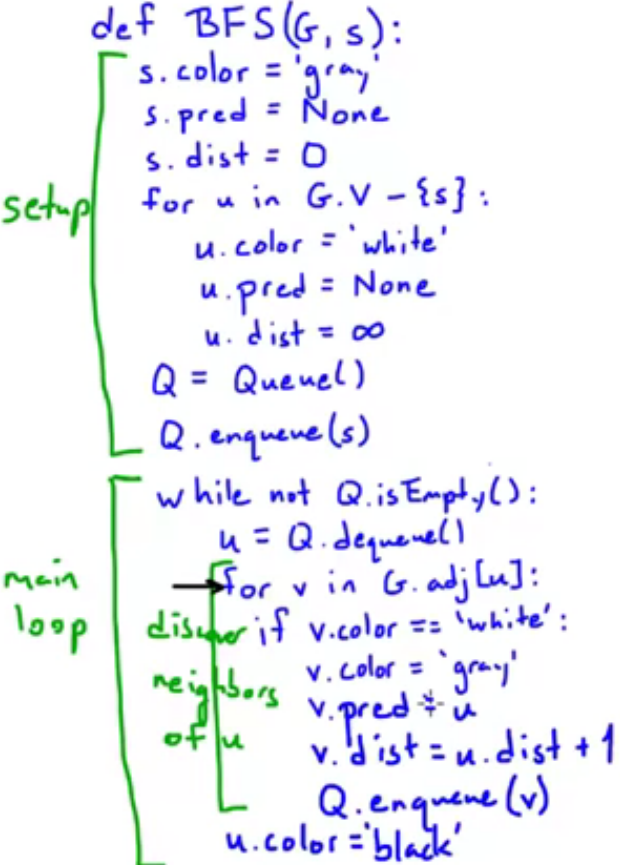
* + - * In a directed graph…
        + ~ is not used, and is usually more wordy in English (‘an edge from u to v’)
        + The neighbor ‘N’ notation can be used for directed graphs
    - There can be multiple edges that pair the same two verticies, even in the same direction
      * If this happens, E ceases to be a set and is now a **multiset**
        + There can be the same edges listed multiple times
      * This is usually not the case; if its not specified, the graph is assumed to not contain multiple edges
    - Directed vs undirected graphs
      * A directed graph has edges with directions (and is noted with parenthesis
        + Can be shortened to ‘digraph’
      * An undirected graph has no direction and is represented as a set
        + Undirected is the default, unless specified otherwise
    - A **path** is a walk that does NOT repeat ANY single vertex
      * This differs from a trail in that trails do not repeat edges; paths do not repeat a vertex
      * Path lengths are measured in edges
    - A **closed path** is a **cycle**
      * Note that the first and last vertex can, in fact, be the same; this is the only caveat
      * Cycle lengths are measured in edges
      * A cycle of length 3 is called a triangle
      * A cycle of length ‘n’ is called a n-triangle
      * A path or cycle that visits ALL vertices is a **Hamiltonian Cycle**
    - An **adjacency matrix** simply lists the vertexes in rows AND columns; if there is a connection the cell value is 1, otherwise 0
    - A **connected component** of G is a maximal set of mutually reachable vertices
* ‘Maximal’ means ‘if any more vertices are added, its no longer true that they all vertices are mutually reachable from each other
* Example:



* + - A **connected graph** is a graph where every vertex in G is mutually reachable from every other vertex, OR if there is just one connected component in G
      * For directed graphs:
        + Each are mutually reachable iif there is a path from u to v and then v to u
    - A **strongly connected component** (SCC) of G is a maximal set of mutually reachable vertices in G
    - G is strongly connected if and only if G has just one SCC
    - A **tree** is a graph that does NOT loop
      * That is to say, if you went from vertex A to B to C, you cannot get back to A without backtracking
      * In other words, any two verticies are connected by exactly one path
    - And undirected graph with no cycles is a **forest**
      * also, must be more than 1 graph; so there will be multiple unconnected graphs (which are individually trees)
    - A directed graph without any cycles is called a **DAG** (directed acyclic graph)

**Breadth-First Search**

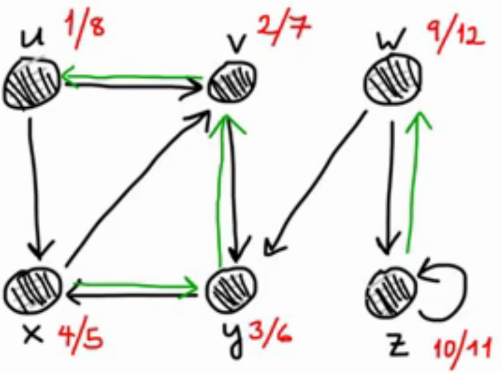


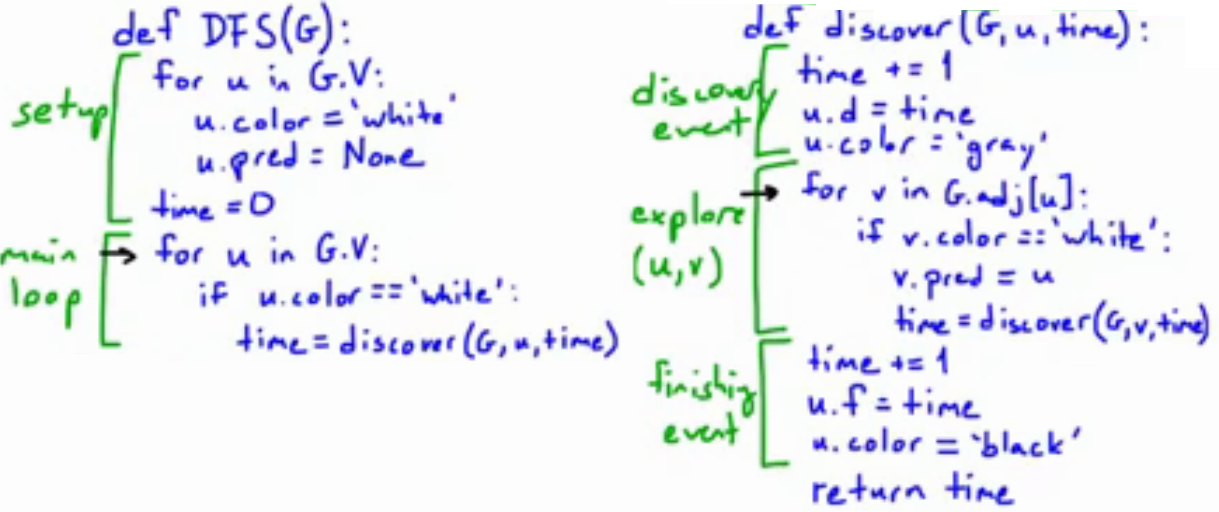


* Basic points of BFS
  + ‘s’ is the source vertex
  + Dist is the distance from the source / root vertex ‘s’
  + ‘pred’ is the predecessor
  + The color schemes indicate the level of visitation
    - White = never visited
    - Gray = visited, but not all neighbors explored
    - Black = visited, all neighbors explored
* The tree resulting from BFS may depend on order of traversing the adjacent list of u but the distances will be the same

**Depth-First Search (DFS)**

* The depth-first search (DFS) finds ALL vertices in G (unlike BFS, which uses a source).
* The predecessor subgraph is defined as Gp = (V, Ep), where Ep is a set of edges in the form {(v.predecessor, v): v.pred != None}
  + The predecessor subgraph is a forest
* Uses colors like BFS, but also uses timestamps
  + Times range from (1, …, 2\*|v|)
    - No two events happen at the same time
  + u.d = discovery time
  + u.f = finishing time



****

* Basic steps
  + Cycle through all vertices
    - Mark them as visited
    - For each vertex, find all adjacencies and immediately search them
      * Mark them as visited and note their predicessor
      * Keep track of ‘time’ which increments through ALL recursive calls (meaning, there is no ‘find’ or ‘complete’ event that shares a time with anything else)
    - Once all neighbors are discovered, color it black, note its end time, move on
* This algorithm drills down to the lowest point it can (not guaranteed to be the absolute lowest point, just the lowest point picked), THEN works backwards and upwards
  + This is why it’s a DFS
  + Before a node is marked ‘black’ and before its siblings are checked, ALL future generations of the on-deck vertex MUST be checked

**Beginning Notes From Dr. Vigoda / On Dr. Vigoda’s Lecture**

**DFS on Directed Graphs and Strongly Connected Components**

In this lecture we’ll review the classic DFS (**depth first search**) algorithm, look at its application to directed graphs and then use it find strongly connected components of a general, directed graph.

We will also look at **DAGs**, which are **directed acyclic graphs** (acyclic means it has no cycles). We will also see how to topologically sort DAGs; what this means is we can order the vertices so that all edges go from left to right.

For general connected graphs, we are going to be looking to find the **SCCs** (**strongly connected components**); this is the analog of connected components for directed graphs. SCCs are found with 2 DFS (depth-first searches).

We begin by reviewing the basic DFS algorithm for undirected graphs.  The below pseudocode finds the connected components for an undirected graph that is given in adjacency list representation.

**Undirected Connected Components**

How do we get connected components in an undirected graph? Easy – run Depth-first search and keep track of component number!

Below is the pseudocode.  There is a global counter *ccnum* which is the number of the current connected component that is being explored, and each vertex is assigned a number *cc[v]* which is the connected component for that vertex.

The input to a DFS is G (that is to say, G=(V, E)), and the output is a collection of verticies labeled by a connected component number. It appears that the entire point of a DFS is you are just given an unordered collection of verticies and edges and the point of a DFS is to put them in some order.

DFS uses a counter, which is the current connected component number (initialized to zero). This version is for an undirected graph, and the connected component number just keeps track of the ‘tree’ in the depth-first forest. The array ‘visited’ keeps track of if we already visited that vertex or not. We then cycle through the vertices (in an arbitrary order). If we find a vertex we have not seen, it means we found a new connected component; when this happens, we increment cc and then explore that vertex. NOTE: In Dr. Pryby’s lessons this counter was called ‘time’; also, Dr. Pryby’s DFS was for a directed graph, where the DFS below is for an undirected graph.

DFS-cc(G):

* For all v \in V
  + visited(v) = False
* ccnum = 0
* For all v \in V:
  + If not visited(v) then
    - ccnum++
    - Explore(v)

Explore(z):

* visited(z) = True
* cc(z) = ccnum
* for (z,w) \in E
  + if not visited(w) then
    - Explore(w)

This simply explores all children / connected vertices FIRST, drilling down to as low as it can go and then bubbling back up (much like Pryby’s interpretation).

The above algorithm runs in *O(n+m)*time where *n=|V|*and *m=|E|.*

**Directed Graph**

We want to extend the above algorithm to directed graphs.  This will require more work and we’ll need to extend the basic DFS algorithm to keep track of further information.  Recall that in a rooted tree we can label the vertices with numbering based on preorder, inorder, and postorder traversal of the tree.  We will use preorder and postorder numbers.  Thus we will have a global counter *clock* and when we first visit a node we assign a preorder number and increment the clock.  When we finish exploring the neighbors of a node we assign it a postorder number and increment the clock.

Here is the pseudocode:

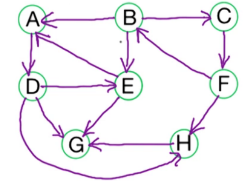
DFS(G):

* for all v \in V
  + visited(v) = False
* clock = 0
* ccnum = 0
* for all v \in V
  + if not visited(v) then
    - ccnum++
    - Explore(v)

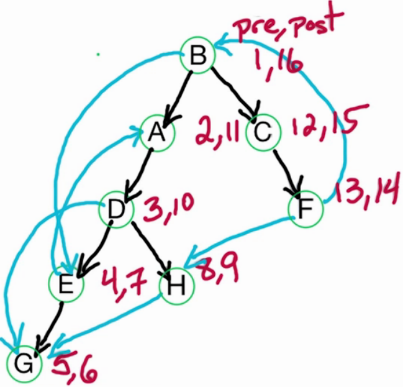
Explore(z):

* visited(z) = True
* pre(z) = clock; clock++;
* cc(z) = ccnum
* for (z,w) \in E
  + if not visited(w) then
    - Explore(w)
* post(z) = clock
* clock++

A visual example:



* Start at B
  + Assume linked list of neighbors are in alphabetical order
* Here is the resultant (pre, post) numbers as well as a visual representation:



* + The black lines are the official lines used to traverse; the blue lines also need to be recorded but its noted that its visiting a vertex that was already visited

There are 4 types of edges v \rightarrow win the DFS tree (see Eric’s handwritten notes for an example):

1. **Tree edges**: explored edges meaning that when considering this edge *w* is visited for the first time and thus we then run *Explore(w).*
   1. These are the black edges in the above example
   2. THESE DO NOT HAVE TO BE TREES; they can be forests, although this example happens to be a tree; this may be more rightly called a ‘forest edge’
2. The blue edges
   1. **Back edges**: if *v* is a descendant of *w.*
      1. In other words, these are the edges going back to an ancestor (where the ancestor was obviously already explored)
      2. EXAMPLES: E → A, F → B
   2. **Forward edges**: if *v*is an ancestor of *w*(and it is not a tree edge).
      1. In other words, these are the edges forward to a child (but the child was already explored via a separate child)
      2. EXAMPLE FROM ABOVE: D → G, B → E (forgotten on the diagram above but its there)
   3. **Cross edges**: all remaining edges (in these cases *v* and *w*have no ancestor-descendant relationship).
      1. These are the ‘odd man out’ edges
      2. Technically they have a relationship in the real graph, but that relationship is not captured because one is not an ancestor of the other (although they will share a common ancestor)

It will be useful to consider the properties of the postorder numbers for the 4 types of edges.  For edge v \rightarrow w:

* Tree edge: post(v) > post(w)
  + The post-order number of the head of the edge is greater than the post-order number of the child/tail
    - The ‘older’ ancestor has the highest post-order number
      * This is because they ‘finish last’, after all of their children are finished – so ancestors will have a higher number for tree edges
* Back edge: *post(w) > post(v)*
  + The post-order number of the head/beginning of the edge is smaller than the post-order number of the child/tail
    - Again, the ‘older’ ancestor has the highest post-order number
* Forward edge: post(v) > post(w)
  + Same as a tree edge
  + The post-order number of the head of the edge is greater than the post-order number of the child/tail
  + The ‘older’ ancestor has the highest post-order number
* Cross edge: post(v) > post(w)
  + Same as tree edge
  + This is because, by definition, the child was explored first so it has a lower post number (otherwise, the child would have been the endpoint from a tree edge of v and would not be considered a cross-edge)
* In short, for the back edge the post-number increases, but for all other edges the post-order number will be smaller

Notice that for back edges *the postorder number of the tail w > postorder number of the head v*, but it is the opposite for all other types of edges.  Back edges are important for detecting cycles.

*Theorem:* A directed graph G has a cycle if and only if its DFS tree has a back edge.

*Proof.* (\Longrightarrow):Consider a cycle C=v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_\ell \rightarrow v_0.  Some vertex of *C* is visited first by the DFS algorithm.  Let v_idenote the first vertex of *C*that’s visited.  From v_iwe can reach every other vertex in *C,*thus the rest of the vertices of *C* are in the subtree rooted at v_i and therefore they are all descendants of v_i in the DFS tree.  Therefore, the edge v_{i-1}\rightarrow v_{i}will be a back edge.  
(\Longleftarrow): Say the back edge is e=v\rightarrow w.  Since vis a descendant of win the DFS tree there is a path \cal Pfrom vto w.  Therefore, \cal P \cup \{ e \} is a cycle.

Cycles

* A graph G has a **cycle** iff its DFS tree has a back edge.
  + The starting vortex doesn’t matter
  + The ordering of the vertices in the adjacency list doesn’t matter
* <Proof explained in lecture>

**Topological Sorting**

A **directed acyclic graph**, or **DAG**, is a graph with no cycles.  Our goal is topologically sort the graph.  This is an ordering of the vertices so that all edges go from earlier to later in the ordering.

For a DAG we know that every DFS tree has no back edges.  Therefore all edges v\rightarrow wsatisfy: post(v)>post(w) (that is to say, for every edge v → w, the postorder of v will always be greater).  Hence we can topologically sort a DAG by decreasing postorder number.

Note that in the first vertex in the topological sorting is a ***source*** vertex meaning that it has no incoming edges, whereas the last vertex is a ***sink*** vertex, which is a vertex with no outgoing edges.  There may be other source or sink vertices.  Finally since the vertex with the highest postorder number is first in our topologically sorting algorithm we know that in a DAG the vertex with highest postorder number is guaranteed to be a source.  Similarly, in a DAG the vertex with lowest postorder number is guaranteed to be a sink.

In effect, we take the vertex with the highest postorder first, then order the rest of the vertices until we get to the sink (postorder = 1 will always be a sink, but sinks do not have to always equal 1).

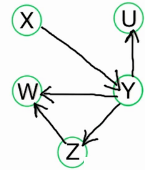
To do a topological sort of a DAG, simply run DFS and then sort by decreasing postorder number.

It takes an array size of 2n to store the results of the DFS (as it takes n time ti drill all the way down and n time to return to the ‘surface’)

An Alternative topological sorting algorithm is to find a sink, store its reference and delete it (then follow its ancestors upwards). Keep doing this until all vertices are found and eliminated, and you now have a list of vertices in topological order for the graph.

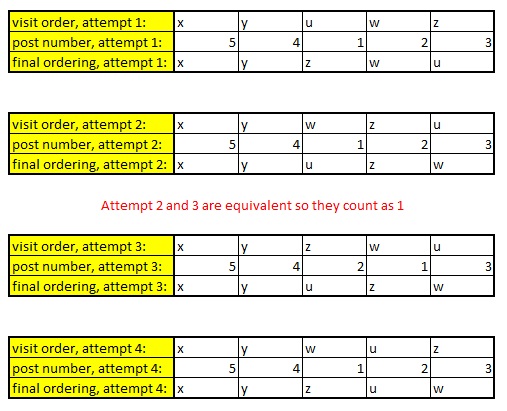
**QUIZ: Topological Ordering**

What are all of the valid topological orderings of the following graph?



Answer: XYUZW, XYZWU, XYZUW

Note the following chart that determines it by hand:



It has to be noted that a vertex is only assigned a post-number after all of its children have been assigned post-numbers.  
  
Also, the starting vertex doesnt matter.  If, for example, the first vertex you find is w, and the second is u, the third is z, and then either:  
A) you find x first -OR-

B) you find y, then x last  
  
that situation - regardless if you pick A) or B) - will work out to attempt #4. In case B), we never actually explore ANY neighbors (as they have all already been visited) but regardless, clock++ still happens (twice) so the pre/post are still set correctly.

**DAG Structure**

Properties of a DAG

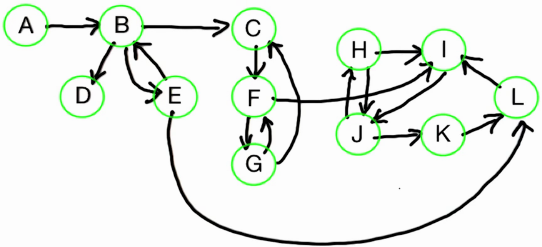
* DAGs are guaranteed to have at least once source and at least one sink
  + If a source is missing, it means its not a DAG as there would be a cycle with a back edge
  + There can be multiple sources and sinks
* The vertex with the highest post order is guaranteed to be a source (as long as the graph is a DAG)
* The last vertex with the lowest post order is guaranteed to be a sink (as long as the graph is a DAG)

**Strongly Connected Components**

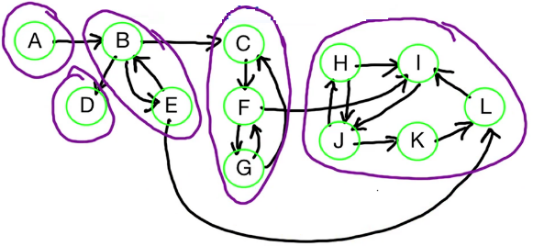
How do we solve connectivity in general directed graphs?  First off we need to define the appropriate notion of connectivity.  We say that vertices v and w are strongly connected if there exists a path from v to w and a path from w to v.  Then a **strongly connected component**, denoted as a SCC, is a maximal set of strongly connected vertices. Visual Example:



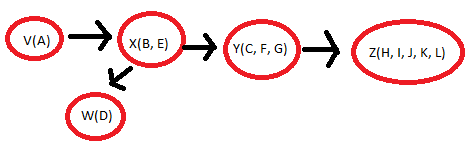
Note that there can be any number of nodes between v and w, so long as there is a path and return path.

It seems that a strongly connected component divides the entire graph into sections that are reachable by other members in the same section (with lone vertices counting as one unit of strongly connected components, if it does not fit in with other strongly connected components). For example, in this graph:  
  


Has 5 connected components (circled in purple):



It seems (although I am not certain) that a **meta-graph** ‘lumps’ the different SCCs together; so if we do this we get a graph with 5 (modified) vertices:



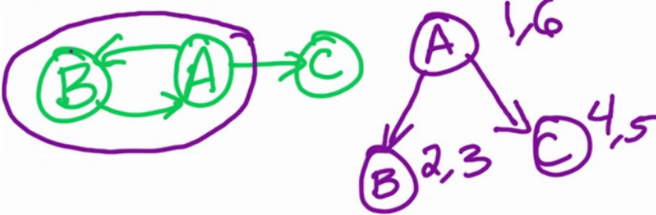
Notice that this new graph happens to be a DAG; matter of fact, once you find all the unique SCCs, you are left with a DAG!

Take a directed graph, partition the vertices into their SCC’s.  Now consider the meta-graph where there is a meta-vertex for each SCC and an edge from SCC S to S’ if some v∈ S has an edge to some w∈ S’.   This meta-graph is a DAG because if there is a cycle then the SCC’s on that cycle should be merged into one SCC. Thus the SCC’s can be topologically sorted.  We will find the SCC’s and we will also sort these SCC’s in topological order.  Moreover we will do this with just 2 runs of DFS.  This algorithm is due to Rao Kosaraju (see the [Wikipedia page for Kosaraju’s algorithm](https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm)).

Our basic approach is to cleverly choose a start vertex *v.* We then run *Explore(v)* to discover the SCC containing *v*, but we only want to visit this SCC containing *v* and no other vertices.  How should we choose this *v?*We want *v* to be in a **sink SCC** (meaning that the SCC containing *v*is a sink vertex in the meta-graph of SCC’s).

A key property of a sink component is that, once we enter the collection of vertices that comprise the sink component, we can continuously loop around the vertices. This is a major way we can tell if we are in a sink component. Now, our key task is to find a vertex that is guaranteed to be in a sink component; once we do this, we can remove the entire sink component and do it again, and keep doing it until there is no graph left.

How do we get a vertex that is guaranteed to be in a sink SCC?  Recall that for a DAG the vertex with lowest postorder number is guaranteed to be a sink vertex.  Thus we might expect that for a general directed graph that the vertex with lowest postorder number is guaranteed to be in a sink SCC.  This is not always the case.  In this example, we start out with vertex A, and the lowest postorder number is 3 (which is B), but that is NOT a sink:



But it turns out that the vertex with highest postorder number is guaranteed to be in a **source SCC**.

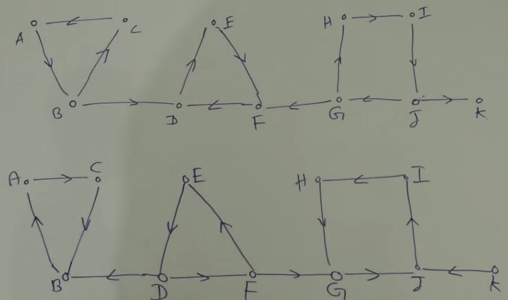
**Key Lemma:** For a directed graph *G*, for any DFS on *G*, the vertex with highest postorder number lies in a source SCC.

The basic idea is that its possible to find a SCC in a DAG because we can absolutely identify a source (although we would prefer a sink as that is the optimal choice). There is something we CAN do, though: we can identify the source, THEN flip the direction on all edges so that source becomes a sink! The graph becomes GR = (V, ER) (this is also true with the associated meta-graph as well), where the R signifies a reverse (the vertices stay the same, but the edges reverse); in other words,



We will now have an identified SCC that is a sink in the graph.

Here is a before and after example of taking the reverse edges in a graph:



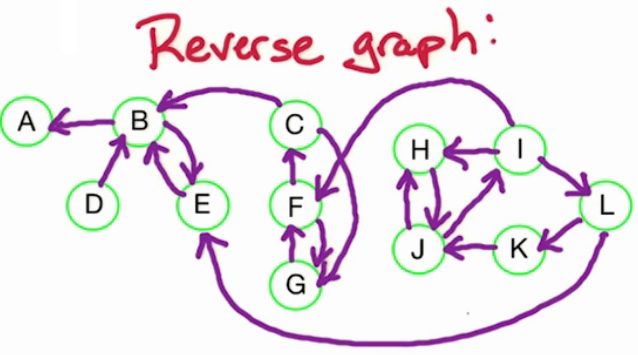
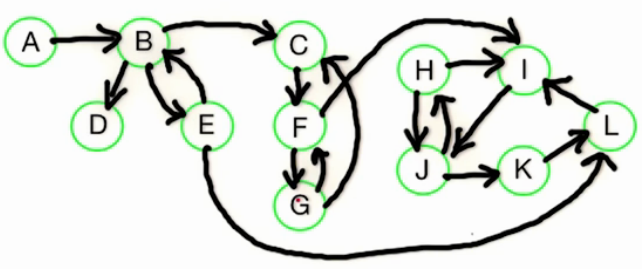
**SCC Algorithm**Take as input a directed graph G=(V,E)in adjacency list representation.

1. Build G^R.
2. Run DFS on G^R(this is the DFS algorithm for directed graphs).
3. Order Vby decreasing postorder numbers from step (2).
4. Run DFS-cc(G) with vertices ordered from step (3).

Complexity: O(V + E).

Basic Idea Behind Kosaraju’s Algorithm

Use this example:



Recall that this graph had two sinks: (D) and (H/I/J/K/L)

The VERY basic idea: We pick a vertex at random and start traversing, marking visited vertices each time we discover a new one; once we run out of vertices, we know we are in a sink and we know all of the elements, so mark them all in the same sink. Once we identify all vertices in that sink, we ‘rip’ it out of the graph and move on to the next sink we find.

NOTE: It seems that a ‘source’ of a reverse graph doesn’t actually have to contain a single source vertex; however, when ‘source’ is mentioned in this context it is referencing the meta-vertex (that is to say, for example, the ENTIRETY of (H, I, J, K, L) is a source meta-vertex while not a single vertex contained within is a source; looking at it as a meta-vertex, it’s clear that there are only two exits to the collection and no entries).

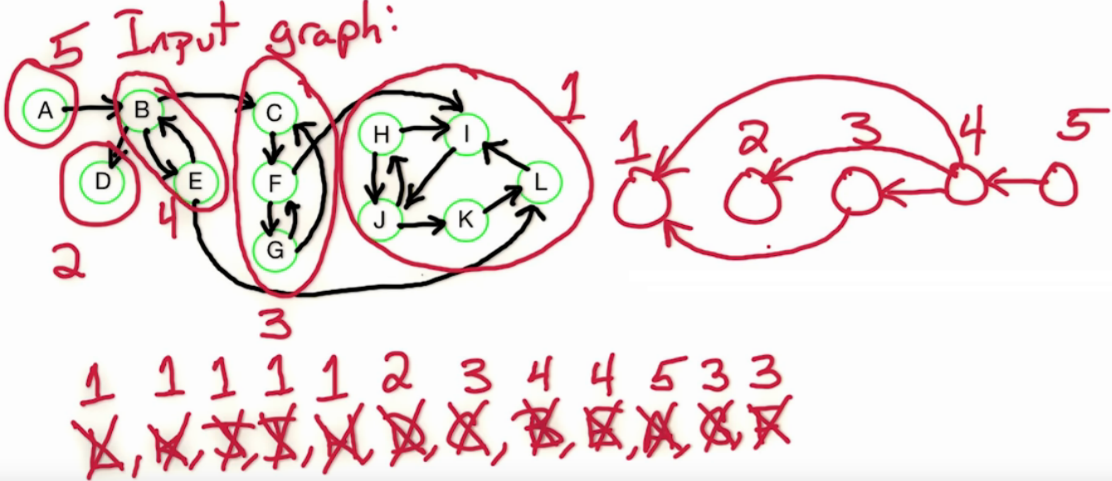
A bit more detailed: Initially, all vertices in the graph are marked unvisited. We reverse the graph and run the directed DFS algorithm on it; once this is done, its guaranteed that a source (recall that before the reversal, it was a sink) will have the highest DFS post-order number; and, matter of fact, once this source is stripped, the next highest post-order number will be of a vertex associated with the source (now, after the previous source being cut from the graph). For this reason, we keep a list of vertices, arranged in descending order of GR’s post-order number.

NOW that we have a listing of GR’s post-order numbers in descending order, we do a DFS-cc on the original graph, but instead of picking random vertices to start, we start with the vertex with the highest post-order number in GR (hence why we kept this list) (NOTE this is the undirected version of DFS, unlike the directed DFS we used to get a listing of the postorder numbers above). The ‘visited’ array is re-set, as well as a groupID that we will assign uniquely to meta-vertices (that is to say, the collection of vertices in a meta-vertex). We start traversing this list.

We pick the vertex with the highest GR post-order number. We then traverse all of its neighbors and children (in the original graph), marking them all visited; we keep traversing until there are no more vertices to traverse in this chain. Since we are using the edges in the original graph BUT we started out with the highest post-order in the reverse graph, we are guaranteed to have fully explored exactly one complete sink in the original graph. At this point, we mark all of these vertices as visited, mark them all with the same groupID, increment the group ID, pop all vertices from our GR post-order list that we just visited, and find the next highest GR post-order that needs to be visited. With our new groupID, the process repeats until there are no more elements in the GR post-order list.

The groupIDs actually identify the SCCs (aka the meta-vertices) in the graph!

One other cool point: the edges from meta-vertex to meta-vertex are captured in a DAG in topological order (albeit reversed, if you take SCC #1 as the leftmost component). Observe:



This means we can take any (general) directed graph, and with two runs of DFS, we can find its SCCs and we can structure these SCCs in topological order.

The runtime is O(V + E), as we need to cycle through both the vertices and edges multiple times.

**Proof of Key Lemma.**

It remains to prove the key lemma that the vertex v with maximum postorder number lies in a source SCC.  To prove the lemma we’ll use the following simpler claim.

**Claim:** For SCC’s Sand S', if there is an edge from some v \in Sto some w \in S', then the maximum postorder number in S > maximum postorder number in S’.

Before we prove the claim, let’s use it to prove the lemma.  Notice that by the claim we can topologically sort the SCC’s by the maximum postorder number in each SCC (the claim then implies that all edges go from earlier SCC’s to later SCC’s in this ordering and thus it’s a valid topologically sorting of the SCC’s).  Now look at the first SCC in this ordering, call it S.  SCC S contains the vertex vwith the maximum postorder number, and since S is first in the topological ordering it is a source SCC.  Hence vlies in a source SCC and that proves the lemma.  We simply need to prove the claim now.

**Proof of claim:** Let ube the first vertex in S \cup S'that’s visited.

If u \in S',  then we visit and finish all of S’ before visiting any of S because S’ has no path to S (since S and S’ are distinct SCCs).  Hence, post(w) < post(z) for all w \in S', z \in S.

If u \in S, then the rest of S\cup S'is in u’s subtree in the DFS tree.  Hence, post(u) > post(z) for all z \in S \cup S' - \{v\}.  This completes the proof of the claim.