Advanced Algorithms/Analysis

Graphs and DFS (Read Ch. 3 from reference book (Ch 22 from textbook))
CS 721
Fal 2018

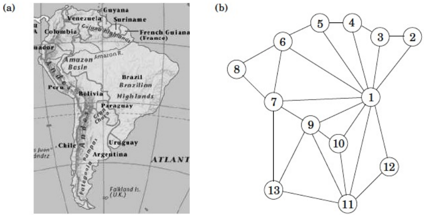
September 4, 2018

Graph Algorithms: Why Graph?

- Wide range of problems are expressed in graphs. Consider coloring of political map
  - What is the minimum number of colors needed to map the world so that no two neighboring countries have same color?

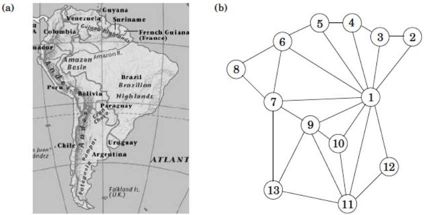
Graph Algorithms: Why Graph?

- Wide range of problems are expressed in graphs. Consider coloring of political map
  - What is the minimum number of colors needed to map the world so that no two neighboring countries have same color?
    - Express the problem as a graph



Graph Algorithms: Why Graph?

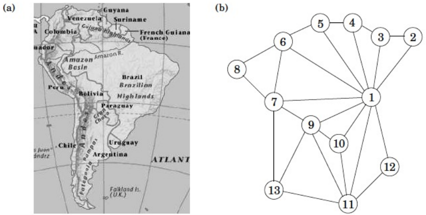
- Wide range of problems are expressed in graphs. Consider coloring of political map
  - What is the minimum number of colors needed to map the world so that no two neighboring countries have same color?
    - Express the problem as a graph



  - Suppose university needs to schedule final exam for all of its classes so that and wants to use fewest timeslots in doing so. What is the number of fewest time slots?

Graph Algorithms: Why Graph?

- Wide range of problems are expressed in graphs. Consider coloring of political map
  - What is the minimum number of colors needed to map the world so that no two neighboring countries have same color?
    - Express the problem as a graph



  - Suppose university needs to schedule final exam for all of its classes so that and wants to use fewest timeslots in doing so. What is the number of fewest time slots?
    - Again, express the problem as a graph

Graph

- Formally a graph is specified by a set of vertices V and a set of edges E between the selected pair of points
  - Between two nodes x and y, if the edge is undirected we represent it as $\{x, y\}$
  - If the edge is directed from x to y, we represent it as (x, y)

Graph

- Formally a graph is specified by a set of vertices V and a set of edges E between the selected pair of points
  - Between two nodes x and y, if the edge is undirected we represent it as $\{x, y\}$
  - If the edge is directed from x to y, we represent it as (x, y)
- Representation of graph

Graph

- Formally a graph is specified by a set of vertices V and a set of edges E between the selected pair of points
  - Between two nodes x and y, if the edge is undirected we represent it as $\{x, y\}$
  - If the edge is directed from x to y, we represent it as (x, y)
- Representation of graph
  - Adjacency matrix
    - Suppose theere are $n = |V|$ vertices $v_1, \ldots, v_n$, the graph is represented by an $n \times n$ adjacaency matrix whose (i, j)-th entry is $a_{ij} = 1$ if there is an edge between v_i to v_j and 0 otherwise.

Graph

- Formally a graph is specified by a set of vertices V and a set of edges E between the selected pair of points
  - Between two nodes x and y, if the edge is undirected we represent it as $\{x, y\}$
  - If the edge is directed from x to y, we represent it as (x, y)
- Representation of graph
  - Adjacency matrix
    - Suppose theere are $n = |V|$ vertices $v_1, \dots, v_n$, the graph is represented by an $n \times n$ adjacaency matrix whose (i, j)-th entry is $a_{ij} = 1$ if there is an edge between v_i to v_j and 0 otherwise.
    - Requires $O(n^2)$ storage, and it can be cheked whether an edge is present in graph or noot in $\Theta(1)$ time

Graph

- Formally a graph is specified by a set of vertices V and a set of edges E between the selected pair of points
  - Between two nodes x and y, if the edge is undirected we represent it as $\{x, y\}$
  - If the edge is directed from x to y, we represent it as (x, y)
- Representation of graph
  - Adjacency matrix
    - Suppose theere are $n = |V|$ vertices $v_1, \dots, v_n$, the graph is represented by an $n \times n$ adjacaency matrix whose (i, j)-th entry is $a_{ij} = 1$ if there is an edge between v_i to v_j and 0 otherwise.
    - Requires $O(n^2)$ storage, and it can be cheked whether an edge is present in graph or noot in $\Theta(1)$ time
  - Adjavcency list
    - It contains $|V$ linked list, one per vertex. The linked list of vertex u holds the list of vertices to which u has an outgoing edge.

Graph

- Formally a graph is specified by a set of vertices V and a set of edges E between the selected pair of points
  - Between two nodes x and y, if the edge is undirected we represent it as $\{x, y\}$
  - If the edge is directed from x to y, we represent it as (x, y)
- Representation of graph
  - Adjacency matrix
    - Suppose theere are $n = |V|$ vertices $v_1, \ldots, v_n$, the graph is represented by an $n \times n$ adjacaency matrix whose (i, j)-th entry is $a_{ij} = 1$ if there is an edge between v_i to v_j and 0 otherwise.
    - Requires $O(n^2)$ storage, and it can be cheked whether an edge is present in graph or noot in $\Theta(1)$ time
  - Adjavcency list
    - It contains $|V$ linked list, one per vertex. The linked list of vertex u holds the list of vertices to which u has an outgoing edge.
    - $O(|E|)$ storage while the worst case time to check whether an edge is present in a graph or not in $O(n)$ time.

Which representation is better?

- Number of edges $|E|$ can be colse to $O(n)$ or close to $O(n^2)$.
  - Dense graph: When $|E|$ is close to $O(n^2)$.
  - Dense graph: When $|E|$ is colse to $O(n)$
- Exactly where $|E|$ lies in this range is a cruicial factor in selecting teh right graph algorithm.

Depth first search (DFS) in a undirected graph

- DFS is a versatile linear time (in graph representation) procedure that reveals a wealth of information about a graph.

Depth first search (DFS) in a undirected graph

- DFS is a versatile linear time (in graph representation) procedure that reveals a wealth of information about a graph.
- Very basic question: What parts of the graph are reachable from a given vertex?

Depth first search (DFS) in a undirected graph

- DFS is a versatile linear time (in graph representation) procedure that reveals a wealth of information about a graph.
- Very basic question: What parts of the graph are reachable from a given vertex?

Figure 3.3 Finding all nodes reachable from a particular node.

```
procedure explore(G, v)
Input:    G = (V, E) is a graph; v ∈ V
Output:   visited(u) is set to true for all nodes u reachable from v

visited(v) = true
previsit(v)
for each edge (v, u) ∈ E:
   if not visited(u):  explore(u)
postvisit(v)
```

Depth first search (DFS) in a undirected graph

- DFS is a versatile linear time (in graph representation) procedure that reveals a wealth of information about a graph.
- Very basic question: What parts of the graph are reachable from a given vertex?

Figure 3.3 Finding all nodes reachable from a particular node.

```
procedure explore(G, v)
Input:    G = (V, E) is a graph; v ∈ V
Output:   visited(u) is set to true for all nodes u reachable from v

visited(v) = true
previsit(v)
for each edge (v, u) ∈ E:
   if not visited(u):  explore(u)
postvisit(v)
```

- The previsit and postvisit procedures are optional, meant for performing operations on a vertex when it is first discovered and also when it is being left for the last time. We will soon see some creative uses for them.

Explore function

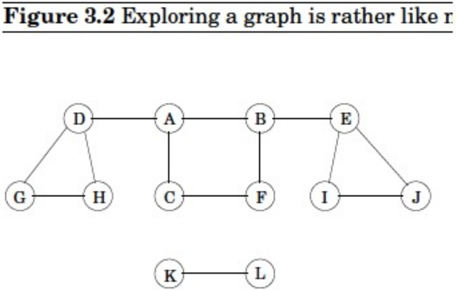
Figure 3.3 Finding all nodes reachable from a particular node.

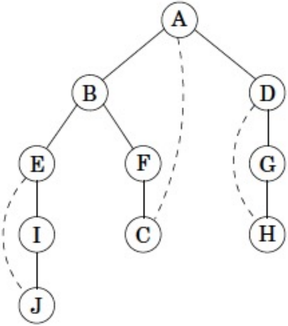
procedure explore(G, v)
Input: $G = (V, E)$ is a graph; $v \in V$
Output: visited(u) is set to true for all nodes u reachable from v

visited(v) = true
previsit(v)
for each edge $(v, u) \in E$:
    if not visited(u): explore(u)
postvisit(v)

- How do we confirm explore works correctly?
  - Proof in class

Example of explore function





Depth First Search

- The explore procedure visits only the portion of the graph reachable from its starting point.

Depth First Search

- The explore procedure visits only the portion of the graph reachable from its starting point.
  - To examine the rest of the graph, we need to restart the procedure elsewhere, at some vertex that has not yet been visited.

Depth First Search

- The explore procedure visits only the portion of the graph reachable from its starting point.
  - To examine the rest of the graph, we need to restart the procedure elsewhere, at some vertex that has not yet been visited

Figure 3.5 Depth-first search.

```
procedure dfs(G)

for all v ∈ V:
   visited(v) = false

for all v ∈ V:
   if not visited(v):  explore(v)
```

Depth First Search

- The explore procedure visits only the portion of the graph reachable from its starting point.
  - To examine the rest of the graph, we need to restart the procedure elsewhere, at some vertex that has not yet been visited

Figure 3.5 Depth-first search.

```
procedure dfs(G)

for all v ∈ V:
   visited(v) = false

for all v ∈ V:
   if not visited(v):  explore(v)
```

- Running time of DFS

Depth First Search

- The explore procedure visits only the portion of the graph reachable from its starting point.
  - To examine the rest of the graph, we need to restart the procedure elsewhere, at some vertex that has not yet been visited

Figure 3.5 Depth-first search.

```
procedure dfs(G)

for all v ∈ V:
   visited(v) = false

for all v ∈ V:
   if not visited(v):  explore(v)
```

- Running time of DFS
  - Each vertex is explored just once

Depth First Search

- The explore procedure visits only the portion of the graph reachable from its starting point.
  - To examine the rest of the graph, we need to restart the procedure elsewhere, at some vertex that has not yet been visited

Figure 3.5 Depth-first search.

```
procedure dfs(G)

for all v ∈ V:
   visited(v) = false

for all v ∈ V:
   if not visited(v):  explore(v)
```

- Running time of DFS
  - Each vertex is explored just once
  - During exploration of a vertex there are the folowing steps
    1. Fixed amount of work to mark this vertex is visited and pre/postvisit
    2. A loop in which adjacent edges are scanned, if they lead somewhere new

Analyzing running time of Depth First Search

- Step 1 takes $O(|V|)$ time

Analyzing running time of Depth First Search

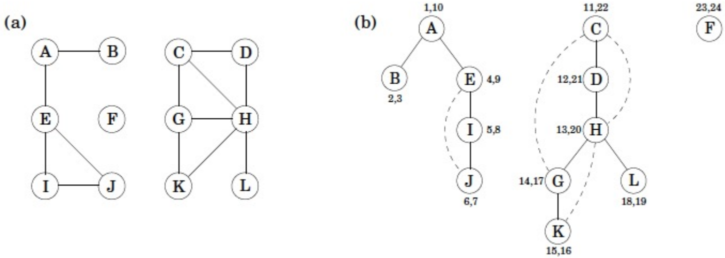
- Step 1 takes $O(|V|)$ time
- In step 2, over the course of the entire DFS, each edge $\{x, y\} \in E$ is examined exactly twice, once during $explore(x)$ and once during $explore(y)$. The overall time for step 2 is therefore $O(|E|)$.

Analyzing running time of Depth First Search

- Step 1 takes $O(|V|)$ time
- In step 2, over the course of the entire DFS, each edge $\{x, y\} \in E$ is examined exactly twice, once during $explore(x)$ and once during $explore(y)$. The overall time for step 2 is therefore $O(|E|)$.
- Thus total running time of DFS is $O(|V| + |E|)$
  - This is efficient because this running time is of the same order that is required to to read the entire graph in adjacency list representation.

Example of DFS

Figure 3.6 (a) A 12-node graph. (b) DFS search forest.



Connectivity in undirected graphs

- An undirected graph is connected if there is a path between any par of vertices

Connectivity in undirected graphs

- An undirected graph is connected if there is a path between any par of vertices
- Fig 3.6 is not connected but there are three disjoint connected connected regions: $\{A, B, E, I, J\}, \{C, D, G, H, K, L\}, \{F\}$
  - these reregions are called connected components

Connectivity in undirected graphs

- An undirected graph is connected if there is a path between any par of vertices
- Fig 3.6 is not connected but there are three disjoint connected connected regions: $\{A, B, E, I, J\}, \{C, D, G, H, K, L\}, \{F\}$
  - these reregions are called connected components
- DFS is trivially adapted to check if a graph is connected and, more generally, to assign each node v an integer $ccnum[v]$ identifying the connected component to which it belongs.

```
procedure previsit(v)
ccnum[v] = cc
```

  - where cc needs to be initialized to zero and to be incremented each time the DFS procedure calls explore.

Previsit and postvisit ordering

- We have seen how DFS is able to uncover the connectivity structure of an undirected graph in linear time
- Now we will collect a little bit more information about the exploration process

Previsit and postvisit ordering

- We have seen how DFS is able to uncover the connectivity structure of an undirected graph in linear time
- Now we will collect a little bit more information about the exploration process
  - we will note down the times of two important events, the moment of first discovery (corresponding to previsit) and that of final departure (postvisit).
  - How do we generate arrays of pre and postvisit?

Previsit and postvisit ordering

- We have seen how DFS is able to uncover the connectivity structure of an undirected graph in linear time
- Now we will collect a little bit more information about the exploration process
  - we will note down the times of two important events, the moment of first discovery (corresponding to previsit) and that of final departure (postvisit).
  - How do we generate arrays of pre and postvisit?
  - define a simple counter *clock* (global variable), initially set it to 1 and update it as follows:

```
procedure previsit(v)          procedure postvisit(v)
pre[v] = clock                 post[v] = clock
clock = clock + 1              clock = clock + 1
```

Previsit and postvisit ordering

- We have seen how DFS is able to uncover the connectivity structure of an undirected graph in linear time
- Now we will collect a little bit more information about the exploration process
  - we will note down the times of two important events, the moment of first discovery (corresponding to previsit) and that of final departure (postvisit).
  - How do we generate arrays of pre and postvisit?
  - define a simple counter *clock* (global variable), initially set it to 1 and update it as follows:

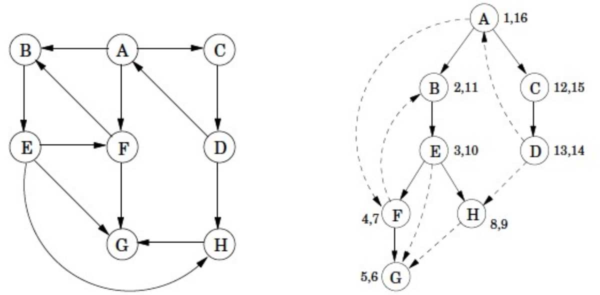
```
procedure previsit(v)        procedure postvisit(v)
pre[v] = clock               post[v] = clock
clock = clock + 1            clock = clock + 1
```

- **Property:** For any nodes u and v, the two intervals $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are either disjoint or one is contained within the other.

DFS in directed graph

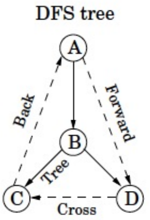
- Our depth-first search algorithm can be run verbatim on directed graphs, taking care to traverse edges only in their prescribed directions.
- Fig 3.7 shows an example

Figure 3.7 DFS on a directed graph.



Types of edges in DFS forest

- There are three types of edges
  - **Tree edges** are actually part of the DFS forest (solid lines). Edge (u, v) is a tree edge if v was discovered by exploring the edge (u, v).
  - **Forward edges** lead from a node to a nonchild descendant in the DFS tree.
  - **Back edges** lead to an ancestor in the DFS tree.
  - **Cross edges** lead to neither descendant nor ancestor; they therefore lead to a node that has already been completely explored (that is, already postvisited).



Types of edges and pre/postorder values

- Ancestor and descendant relationships, as well as edge types, can be read off directly from pre and post numbers

Types of edges and pre/postorder values

- Ancestor and descendant relationships, as well as edge types, can be read off directly from pre and post numbers
- Because of the depth-first exploration strategy, vertex u is an ancestor of vertex v exactly in those cases where u is discovered first and v is discovered during $explore(u)$. This is to say $pre(u) < pre(v) < post(v) < post(u)$

Types of edges and pre/postorder values

- Ancestor and descendant relationships, as well as edge types, can be read off directly from pre and post numbers
- Because of the depth-first exploration strategy, vertex u is an ancestor of vertex v exactly in those cases where u is discovered first and v is discovered during *explore*(u). This is to say $pre(u) < pre(v) < post(v) < post(u)$
- Summary of the various possibilities for an edge (u, v) is as follows

pre/post *ordering for* (u, v)	*Edge type*
$[_u \; [_v \;]_v \;]_u$	Tree/forward
$[_v \; [_u \;]_u \;]_v$	Back
$[_v \;]_v \; [_u \;]_u$	Cross

Directed Acyclic Graph (DAG)

- A cycle in directed graph is a circular path starting and ending at the same vertex

Directed Acyclic Graph (DAG)

- A cycle in directed graph is a circular path starting and ending at the same vertex
- A graph without a cycle is acyclic

Directed Acyclic Graph (DAG)

- A cycle in directed graph is a circular path starting and ending at the same vertex
- A graph without a cycle is acyclic
- Test for acyclicity can be tested in linear time, using a single DFS

Directed Acyclic Graph (DAG)

- A cycle in directed graph is a circular path starting and ending at the same vertex
- A graph without a cycle is acyclic
- Test for acyclicity can be tested in linear time, using a single DFS
- **Property:** A directed graph has a cycle if and only if its depth-first search reveals a back edge.
  - Proof in class

Directed Acyclic Graph (DAG)

- A cycle in directed graph is a circular path starting and ending at the same vertex
- A graph without a cycle is acyclic
- Test for acyclicity can be tested in linear time, using a single DFS
- **Property:** A directed graph has a cycle if and only if its depth-first search reveals a back edge.
  - Proof in class
- Directed acyclic graphs, or DAGs for short, come up all the time. They are good for modeling relations like causalities, hierarchies, and temporal dependencies.

DAG and topological sort

- If a graph is DAG it is possible to linearize or topologically sort its vertices so that before performing a task (represented by a vertex) all other tasks (represented by other vertices) that are connected to it must be completed

DAG and topological sort

- If a graph is DAG it is possible to linearize or topologically sort its vertices so that before performing a task (represented by a vertex) all other tasks (represented by other vertices) that are connected to it must be completed
- In other words, we can order the vertices of a DAG based on directed edges such that for any edge (u, v), u should appear before v in the ordering

DAG and topological sort

- If a graph is DAG it is possible to linearize or topologically sort its vertices so that before performing a task (represented by a vertex) all other tasks (represented by other vertices) that are connected to it must be completed
- In other words, we can order the vertices of a DAG based on directed edges such that for any edge (u, v), u should appear before v in the ordering
- Once again DFS tells us how to do it
  - simply perform tasks in decreasing order of their post numbers.

DAG and topological sort

- If a graph is DAG it is possible to linearize or topologically sort its vertices so that before performing a task (represented by a vertex) all other tasks (represented by other vertices) that are connected to it must be completed
- In other words, we can order the vertices of a DAG based on directed edges such that for any edge (u, v), u should appear before v in the ordering
- Once again DFS tells us how to do it
  - simply perform tasks in decreasing order of their post numbers.
- Topological-Sort(G)
  1. Call $dfs(G)$ to compute $post[v]$ for each vertex v
  2. As each vertex is done visiting (post is updated), insert onto the front of a linked list
  3. Return linked list of vertices

Running time of topological sort

- Time for DFS is $O(|V| + E|)$ time to insert onto linked list is $O(|V|)$ (constant time for each insert)

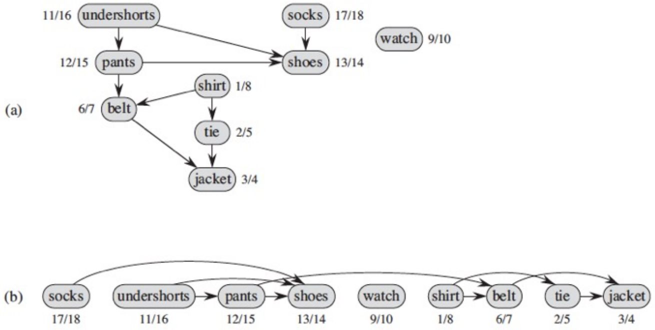


Figure 22.7 (a) Professor Bumstead topologically sorts his clothing when getting dressed. Each directed edge (u, v) means that garment u must be put on before garment v. The discovery and finishing times from a depth-first search are shown next to each vertex. **(b)** The same graph shown topologically sorted, with its vertices arranged from left to right in order of decreasing finishing time. All directed edges go from left to right.

Why does topological sort works correctly?

- Proof:

Why does topological sort works correctly?

- Proof:
  - It suffices to show that for a pair of distinct vertices $u, v \in V$, if $G = (V, E)$ contains an edge from u to v then $post[v] < post[u]$.

Why does topological sort works correctly?

- Proof:
  - It suffices to show that for a pair of distinct vertices $u, v \in V$, if $G = (V, E)$ contains an edge from u to v then $post[v] < post[u]$.
  - A DAG can not have cycle, that means its DFS forest can not have a back edge (proved earlier).

Why does topological sort works correctly?

- Proof:
  - It suffices to show that for a pair of distinct vertices $u, v \in V$, if $G = (V, E)$ contains an edge from u to v then $post[v] < post[u]$.
  - A DAG can not have cycle, that means its DFS forest can not have a back edge (proved earlier).
  - If an edge is (u, v) is a tree/forward edge then $post[v] < post[u]$.

Why does topological sort works correctly?

- Proof:
  - It suffices to show that for a pair of distinct vertices $u, v \in V$, if $G = (V, E)$ contains an edge from u to v then $post[v] < post[u]$.
  - A DAG can not have cycle, that means its DFS forest can not have a back edge (proved earlier).
  - If an edge is (u, v) is a tree/forward edge then $post[v] < post[u]$.
  - If an edge (u, v) is a cross edge (u, v) can not be an edge of G. So order of their post values does not matter.

Source and sink in a DAG

- Since DAG can be linearized by decreasing *post* number, the vertex with smallest post number comes last in linearization
  - It must be a "sink"– no outgoing edges
- Similarly, the node with highest *post* is a "source", a node with no incoming edge.

Strongly connected components

- In directed graphs, two nodes u and v are connected if there is a path from u to v and a path from v to a.

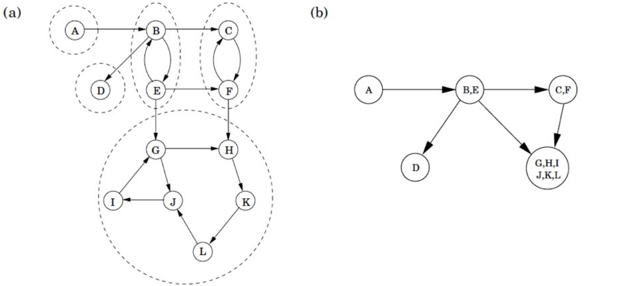
Strongly connected components

- In directed graphs, two nodes u and v are connected if there is a path from u to v and a path from v to a.
- This relation partitions V into disjoint sets that we call "strongly connected components"

Strongly connected components

- In directed graphs, two nodes u and v are connected if there is a path from u to v and a path from v to a.
- This relation partitions V into disjoint sets that we call "strongly connected components"
- In the following graph there are five strongly connected components

Figure 3.9 (a) A directed graph and its strongly connected components. (b) The meta-graph.



Strongly connected components

- Shrink each strongly connected component down to a single meta node and draw an edge from one meta node to another if there is an edge between their respective components

Strongly connected components

- Shrink each strongly connected component down to a single meta node and draw an edge from one meta node to another if there is an edge between their respective components
- The resulting meta graph must be a DAG. Why?

Strongly connected components

- Shrink each strongly connected component down to a single meta node and draw an edge from one meta node to another if there is an edge between their respective components
- The resulting meta graph must be a DAG. Why?
  - because a cycle containing several strongly connected components would merge them all into a single strongly connected components
- **Property:** Every directed graph is a DAG of its strongly connected components

Strongly connected components

- Shrink each strongly connected component down to a single meta node and draw an edge from one meta node to another if there is an edge between their respective components
- The resulting meta graph must be a DAG. Why?
  - because a cycle containing several strongly connected components would merge them all into a single strongly connected components
- **Property:** Every directed graph is a DAG of its strongly connected components
- That means connectivity structure is a directed graph is two tiered:

Strongly connected components

- Shrink each strongly connected component down to a single meta node and draw an edge from one meta node to another if there is an edge between their respective components
- The resulting meta graph must be a DAG. Why?
  - because a cycle containing several strongly connected components would merge them all into a single strongly connected components
- **Property:** Every directed graph is a DAG of its strongly connected components
- That means connectivity structure is a directed graph is two tiered:
  - At the top level we have a DAG (which is rather simple structure and can be linearize)

Strongly connected components

- Shrink each strongly connected component down to a single meta node and draw an edge from one meta node to another if there is an edge between their respective components
- The resulting meta graph must be a DAG. Why?
  - because a cycle containing several strongly connected components would merge them all into a single strongly connected components
- **Property:** Every directed graph is a DAG of its strongly connected components
- That means connectivity structure is a directed graph is two tiered:
  - At the top level we have a DAG (which is rather simple structure and can be linearize)
  - If we want further details, we can look inside one of this nodes of this DAG and examine ful-fledged strongly connected component within

Efficient algorithm for strongly connected components

- We can find strongly connected components of a directed graph in linear time making use of depth first search

Efficient algorithm for strongly connected components

- We can find strongly connected components of a directed graph in linear time making use of depth first search
- The algorithm is based on some properties that we have seen before

Efficient algorithm for strongly connected components

- We can find strongly connected components of a directed graph in linear time making use of depth first search
- The algorithm is based on some properties that we have seen before
  - **Property 1:** If the *explore* subroutine is started at a node u, then it will terminate precisely when all nodes reachable from u has been visited.

Efficient algorithm for strongly connected components

- We can find strongly connected components of a directed graph in linear time making use of depth first search
- The algorithm is based on some properties that we have seen before
  - **Property 1:** If the *explore* subroutine is started at a node u, then it will terminate precisely when all nodes reachable from u has been visited.
    - Therefore if we call *explore* on a node that lies somewhere in a sink strongly connected component, then we will retrieve exactly the component

Efficient algorithm for strongly connected components

- We can find strongly connected components of a directed graph in linear time making use of depth first search
- The algorithm is based on some properties that we have seen before
  - **Property 1:** If the *explore* subroutine is started at a node u, then it will terminate precisely when all nodes reachable from u has been visited.
    - Therefore if we call *explore* on a node that lies somewhere in a sink strongly connected component, then we will retrieve exactly the component
  - This suggests a way of finding one strongly connected component but still leavses two major problems

Efficient algorithm for strongly connected components

- We can find strongly connected components of a directed graph in linear time making use of depth first search
- The algorithm is based on some properties that we have seen before
  - **Property 1:** If the *explore* subroutine is started at a node u, then it will terminate precisely when all nodes reachable from u has been visited.
    - Therefore if we call *explore* on a node that lies somewhere in a sink strongly connected component, then we will retrieve exactly the component
  - This suggests a way of finding one strongly connected component but still leavses two major problems
    - (A) How do we find a node that we know for sure lies in a sink strongly connected component?

Efficient algorithm for strongly connected components

- We can find strongly connected components of a directed graph in linear time making use of depth first search
- The algorithm is based on some properties that we have seen before
  - **Property 1:** If the *explore* subroutine is started at a node u, then it will terminate precisely when all nodes reachable from u has been visited.
    - Therefore if we call *explore* on a node that lies somewhere in a sink strongly connected component, then we will retrieve exactly the component
  - This suggests a way of finding one strongly connected component but still leavses two major problems
    - (A) How do we find a node that we know for sure lies in a sink strongly connected component?
    - (B) How we we continue once the first component is discovered?

Efficient algorithm for strongly connected components

- We will start with problem (A)

Efficient algorithm for strongly connected components

- We will start with problem (A)
  - Though there is no easy way pick a node that is guaranteed to line in a sink strongly connected component, there is a way to get a node in a source strongly connected

Efficient algorithm for strongly connected components

- We will start with problem (A)
  - Though there is no easy way pick a node that is guaranteed to line in a sink strongly connected component, there is a way to get a node in a source strongly connected
  - **Property 2:** The node that receives the highest *post* number in a depth first search must lie in a source strongly connected component
    - This follows from a more general property

Efficient algorithm for strongly connected components

- We will start with problem (A)
  - Though there is no easy way pick a node that is guaranteed to line in a sink strongly connected component, there is a way to get a node in a source strongly connected
  - **Property 2:** The node that receives the highest *post* number in a depth first search must lie in a source strongly connected component
    - This follows from a more general property
  - **Property 3:** If C and C' are strongly connected components. and there is an edge from a node in C to a node in C', then the highest *post* number in C is bigger than the highest *post* number in C'.

Efficient algorithm for strongly connected components

- We will start with problem (A)
  - Though there is no easy way pick a node that is guaranteed to line in a sink strongly connected component, there is a way to get a node in a source strongly connected
  - **Property 2:** The node that receives the highest *post* number in a depth first search must lie in a source strongly connected component
    - This follows from a more general property
  - **Property 3:** If C and C' are strongly connected components. and there is an edge from a node in C to a node in C', then the highest *post* number in C is bigger than the highest *post* number in C'.
    - Proof in class

Efficient algorithm for strongly connected components

- We will start with problem (A)
  - Though there is no easy way pick a node that is guaranteed to line in a sink strongly connected component, there is a way to get a node in a source strongly connected
  - **Property 2:** The node that receives the highest *post* number in a depth first search must lie in a source strongly connected component
    - This follows from a more general property
  - **Property 3:** If C and C' are strongly connected components. and there is an edge from a node in C to a node in C', then the highest *post* number in C is bigger than the highest *post* number in C'.
    - Proof in class
- Property 3 can be restated as ,- *The strongly connected components can be linearize by arranging them in decreasing order of their highest post numbers.*

Efficient algorithm for strongly connected components

- Property 3 is a generalization of our earlier algorithm for linearizing DAG, where in a DAG each node is a singleton strongly connected component

Efficient algorithm for strongly connected components

- Property 3 is a generalization of our earlier algorithm for linearizing DAG, where in a DAG each node is a singleton strongly connected component
- Property 2 helps us find a node in source strongly connected component of G. But we need the node to be in a sink strongly connected component!

Efficient algorithm for strongly connected components

- Property 3 is a generalization of our earlier algorithm for linearizing DAG, where in a DAG each node is a singleton strongly connected component
- Property 2 helps us find a node in source strongly connected component of G. But we need the node to be in a sink strongly connected component!
  - This can be done by considering the reverse graph G^R, which is same as G except direction of each edge is reversed

Efficient algorithm for strongly connected components

- Property 3 is a generalization of our earlier algorithm for linearizing DAG, where in a DAG each node is a singleton strongly connected component
- Property 2 helps us find a node in source strongly connected component of G. But we need the node to be in a sink strongly connected component!
  - This can be done by considering the reverse graph G^R, which is same as G except direction of each edge is reversed
  - G^R has exactly the same strongly connected components as G

Efficient algorithm for strongly connected components

- Property 3 is a generalization of our earlier algorithm for linearizing DAG, where in a DAG each node is a singleton strongly connected component
- Property 2 helps us find a node in source strongly connected component of G. But we need the node to be in a sink strongly connected component!
  - This can be done by considering the reverse graph G^R, which is same as G except direction of each edge is reversed
  - G^R has exactly the same strongly connected components as G
  - So if we do a depth first search on G^R, the node with highest *post* number comes from a source strongly connected component of G^R which is a sink strongly connected component of G (Problem A solved!)

Efficient algorithm for strongly connected components

- Now we deal with problem (B)

Efficient algorithm for strongly connected components

- Now we deal with problem (B)
- How do we continue after first sink component is identified?

Efficient algorithm for strongly connected components

- Now we deal with problem (B)
- How do we continue after first sink component is identified?
  - This solution is also provided by property 3

Efficient algorithm for strongly connected components

- Now we deal with problem (B)
- How do we continue after first sink component is identified?
  - This solution is also provided by property 3
  - Once we have found the first strongly connected component and deleted it from graph, the node with the highest *post* number among those remaining will belong to sink strongly connected component of whatever is remains of G

Efficient algorithm for strongly connected components

- Now we deal with problem (B)
- How do we continue after first sink component is identified?
  - This solution is also provided by property 3
  - Once we have found the first strongly connected component and deleted it from graph, the node with the highest *post* number among those remaining will belong to sink strongly connected component of whatever is remains of G
  - Therefore we can keep using the *post* numbers from our initial depth first search on G^R to successfully output the second strongly connected component, the third strongly connected component and so on.

Efficient algorithm for strongly connected components

- The resulting algorithm is as follows:
- $\underline{StronglyConnectedComponent(G)}$
    1. Run depth first search on G^R
    2. Run the undirected connected component algorithm that we have seen earlier on G, and during depth first search, process the vertices in decreasing order of their *post* number from step 1

Efficient algorithm for strongly connected components

- The resulting algorithm is as follows:
- $\underline{StronglyConnectedComponent(G)}$
  1. Run depth first search on G^R
  2. Run the undirected connected component algorithm that we have seen earlier on G, and during depth first search, process the vertices in decreasing order of their *post* number from step 1
- This algorithm runs in linear time. Only the constant in linear time is about twice that of straight depth first search.