Lecture 6: Graphs

CS 5006/7: Algorithms, C and Systems

Adrienne Slaughter, Joe Buck

Northeastern University

February 12, 2019

Topics

- Intro/definitions
- Paths and Djikstra (REVIEW)
- Acyclic graphs and topological ordering
- Connectivity in Directed Graphs

What is a Graph?

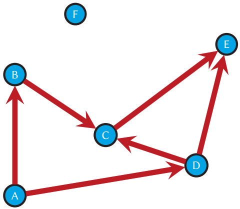
A graph is simply a collection of nodes plus edges

- Linked lists, trees, and heaps are all special cases of graphs
- The nodes are known as vertices (node = vertex)
- Formal Definition:
  - A graph G is a pair (V, E) where
  - V is a set of vertices or nodes
  - E is a set of edges that connect vertices

An Example

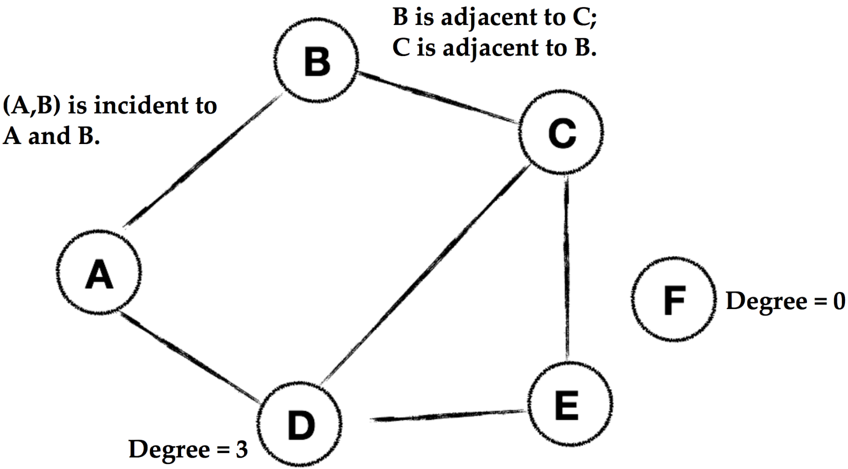
Here is a directed graph $G = (V, E)$

- Each edge is a pair (v_1, v_2), where v_1, v_2 are vertices in V
  - $V = \{A, B, C, D, E, F\}$
  - $E = \{(A, B), (A, D), (B, C), (C, D), (C, E), (D, E)\}$



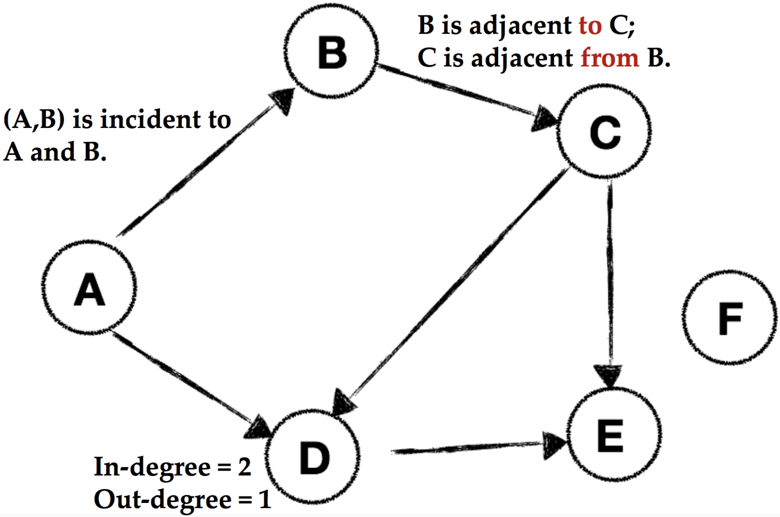
Terminology: Undirected Graph

- Two vertices u and v are ***adjacent*** in an undirected graph G if $\{u, v\}$ is an edge in G
  - edge $e = \{u, v\}$ is ***incident*** with vertex u and vertex v
- The ***degree*** of a vertex in an undirected graph is the number of edges incident with it
  - a self-loop counts twice (both ends count)
  - denoted with $deg(v)$


B is adjacent to C;
C is adjacent to B.
(A,B) is incident to
A and B.
B
C
A
F
Degree = 0
E
D
Degree = 3

Terminology: Directed Graph

- Vertex u is ***adjacent to*** vertex v in a directed graph G if (u, v) is an edge in G
  - vertex u is the initial vertex of (u, v)
- Vertex v is ***adjacent from*** vertex u
  - vertex v is the ***terminal*** (or end) vertex of (u, v)
- Degree
  - ***in-degree*** is the number of edges with the vertex as the terminal vertex
  - ***out-degree*** is the number of edges with the vertex as the initial vertex

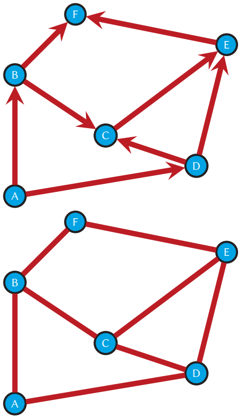

B is adjacent to C;
C is adjacent from B.
(A,B) is incident to
A and B.
In-degree = 2
Out-degree = 1
A
B
C
D
E
F

Kinds of Graphs

- directed vs undirected
- weighted vs unweighted
- simple vs non-simple
- sparse vs dense
- cyclic vs acyclic
- labeled vs unlabeled

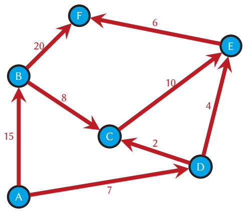
Directed vs Undirected

- Undirected if edge (x, y) implies edge (y, x).
  - otherwise directed
- Roads between cities are usually undirected (go both ways)
- Streets in cities tend to be directed (one-way)



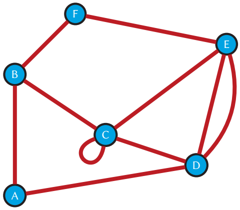
Weighted vs Unweighted

- Each edge or vertex is assigned a numerical value (weight).
- A road network might be labeled with:
    - length
    - drive-time
    - speed-limit
- In an unweighted graph, there is no distinction between edges.



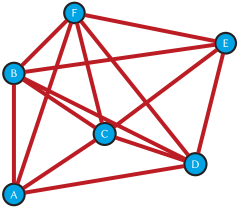
Simple vs Not simple

- Some kinds of edges make working with graphs complicated
- A ***self-loop*** is an edge (x, x) (one vertex).
- An edge (x, y) is a ***multiedge*** if it occurs more than once in a graph.



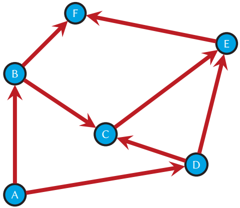
Sparse vs Dense

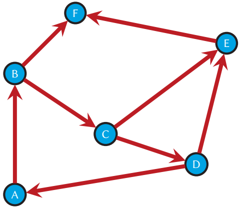
- Graphs are sparse when a small fraction of vertex pairs have edges between them
- Graphs are dense when a large fraction of vertex pairs have edges
- There's no formal distinction between sparse and dense



Cyclic vs Acyclic

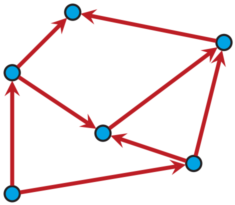
- An ***acyclic*** graph contains no cycles
- A ***cyclic*** graph contains a cycle
- Trees are *connected, acyclic, undirected* graphs
- Directed acyclic graphs are called ***DAGs***





Labeled vs Unlabeled

- Each vertex is assigned a unique name or identifier in a ***labeled*** graph
  - In an unlabeled graph, there are no named nodes
- Graphs usually have names—e.g., city names in a transportation network
- We might ignore names in graphs to determine if they are isomorphic (similar in structure)

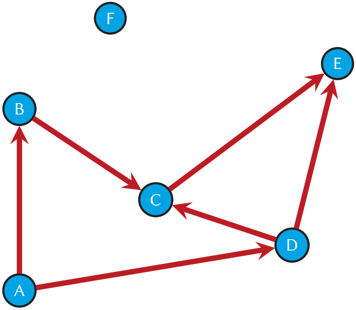


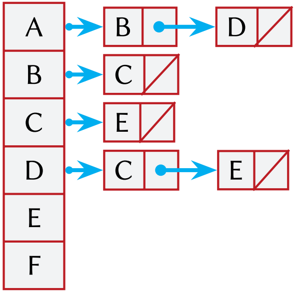
Graph Representation

Two ways to represent a graph in code:

- Adjacency List
  - A list of nodes
  - Every node has a list of adjacent nodes
- Adjacency Matrix
  - A matrix has a column and a row to represent every node
  - All entries are 0 by default
  - An entry $G[u, v]$ is 1 if there is an edge from node u to v

For each v in V, $L(v)$ = list of w such that (v, w) is in E:





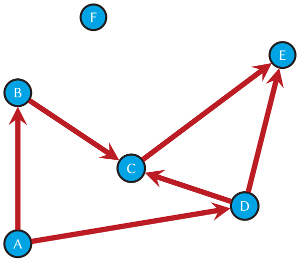
Storage space:

$$a|V| + b|E|$$

$$a = sizeof(node)$$

$$b = sizeof(\text{ linked list element})$$

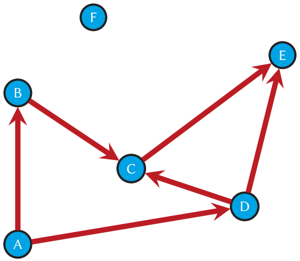
Adjacency Matrix



$$
\begin{array}{c|cccccc}
 & A & B & C & D & E & F \\
\hline
A & 0 & 1 & 0 & 1 & 0 & 0 \\
B & 1 & 0 & 1 & 0 & 0 & 0 \\
C & 0 & 1 & 0 & 1 & 1 & 0 \\
D & 1 & 0 & 1 & 0 & 1 & 0 \\
E & 0 & 0 & 1 & 1 & 0 & 0 \\
F & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

Storage space: $|V|^2$

Adjacency Matrix



$$\begin{array}{c c} & \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \\ E \\ F \end{array} & \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Storage space: $|V|^2$

Does this matrix represent a directed or undirected graph?

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?

1. adjacency matrix

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?

1. adjacency matrix

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?

1. adjacency matrix
2. adjacency list

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?
3. Less memory on small graphs?

1. adjacency matrix
2. adjacency list

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?
3. Less memory on small graphs?

1. adjacency matrix
2. adjacency list
3. adjacency list (m+n) vs (n2)

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?
3. Less memory on small graphs?
4. Less memory on big graphs?

1. adjacency matrix
2. adjacency list
3. adjacency list (m+n) vs (n2)

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?
3. Less memory on small graphs?
4. Less memory on big graphs?

1. adjacency matrix
2. adjacency list
3. adjacency list (m+n) vs (n2)
4. adjacency matrices (a little)

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?
3. Less memory on small graphs?
4. Less memory on big graphs?
5. Edge insertion or deletion?

1. adjacency matrix
2. adjacency list
3. adjacency list (m+n) vs (n2)
4. adjacency matrices (a little)

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?
3. Less memory on small graphs?
4. Less memory on big graphs?
5. Edge insertion or deletion?

1. adjacency matrix
2. adjacency list
3. adjacency list (m+n) vs (n2)
4. adjacency matrices (a little)
5. adjacency matrices O(1) vs O(d)

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?
3. Less memory on small graphs?
4. Less memory on big graphs?
5. Edge insertion or deletion?
6. Faster to traverse the graph?

1. adjacency matrix
2. adjacency list
3. adjacency list (m+n) vs (n2)
4. adjacency matrices (a little)
5. adjacency matrices O(1) vs O(d)

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?
3. Less memory on small graphs?
4. Less memory on big graphs?
5. Edge insertion or deletion?
6. Faster to traverse the graph?

1. adjacency matrix
2. adjacency list
3. adjacency list (m+n) vs (n2)
4. adjacency matrices (a little)
5. adjacency matrices O(1) vs O(d)
6. adjacency list

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?
3. Less memory on small graphs?
4. Less memory on big graphs?
5. Edge insertion or deletion?
6. Faster to traverse the graph?
7. Better for most problems?

1. adjacency matrix
2. adjacency list
3. adjacency list (m+n) vs (n2)
4. adjacency matrices (a little)
5. adjacency matrices O(1) vs O(d)
6. adjacency list

Comparing Matrix vs List

1. Faster to test if (x, y) is in a graph?
2. Faster to find the degree of a vertex?
3. Less memory on small graphs?
4. Less memory on big graphs?
5. Edge insertion or deletion?
6. Faster to traverse the graph?
7. Better for most problems?

1. adjacency matrix
2. adjacency list
3. adjacency list (m+n) vs (n2)
4. adjacency matrices (a little)
5. adjacency matrices O(1) vs O(d)
6. adjacency list
7. adjacency list

Analyzing Graph Algorithms

- Space and time are analyzed in terms of:
  - Number of vertices $m = |V|$
  - Number of edges $n = |E|$
- Aim for polynomial running times.
- But: is $O(m^2)$ or $O(n^3)$ a better running time?
  - depends on what the relation is between n and m
  - the number of edges m can be at most $n^2 \leq n^2$.
  - connected graphs have at least $m \geq n - 1$ edges
- Stil do not know which of two running times (such as m^2 and n^3) are better,
- Goal: implement the basic graph search algorithms in time $O(m + n)$.
  - This is linear time, since it takes $O(m + n)$ time simply to read the input.
- Note that when we work with connected graphs, a running time of $O(m + n)$ is the same as $O(m)$, since $m \geq n - 1$.

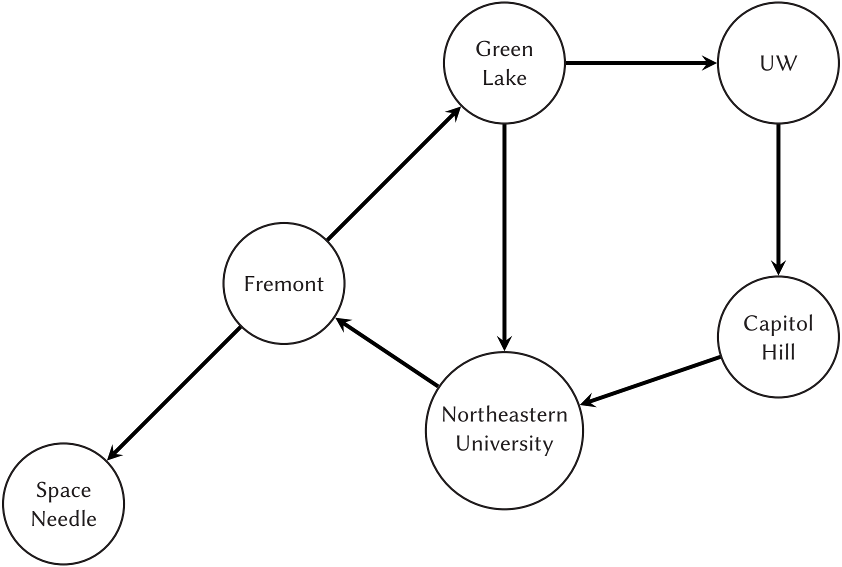
Graph Traversals

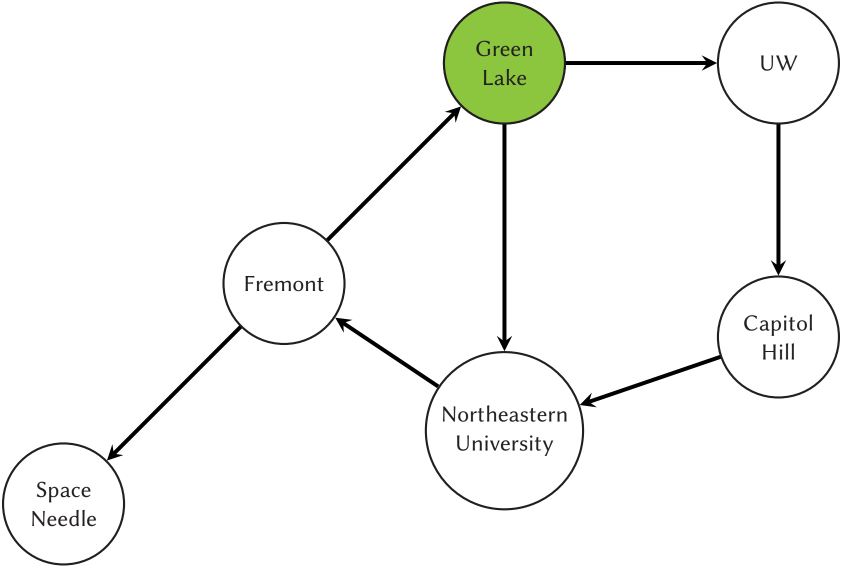
Two basic traversals:

- Breadth First Search (BFS)
- Depth First Search (DFS)

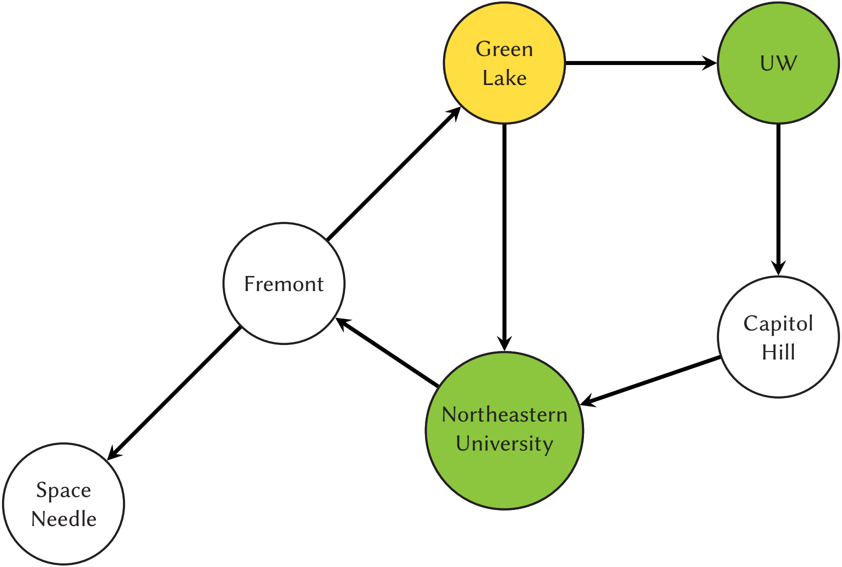
BFS

Example...

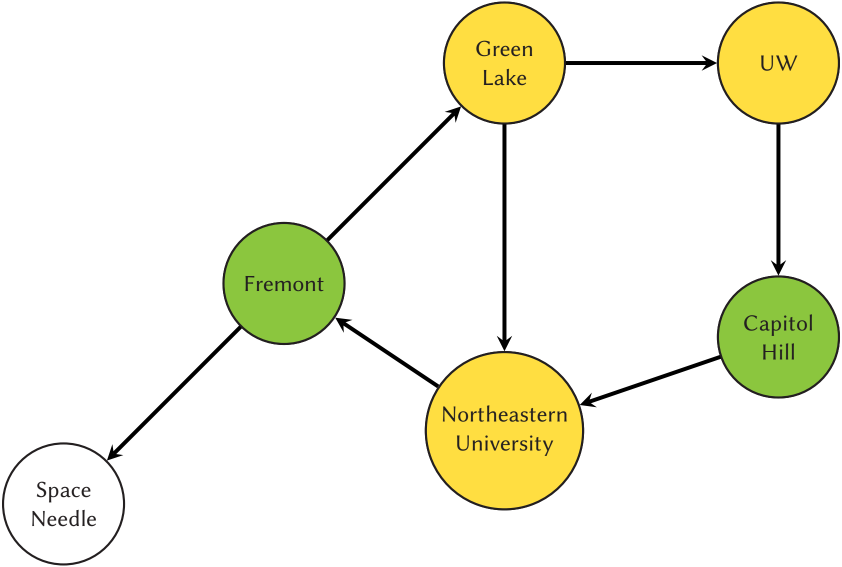

Green Lake
UW
Fremont
Capitol Hill
Northeastern University
Space Needle



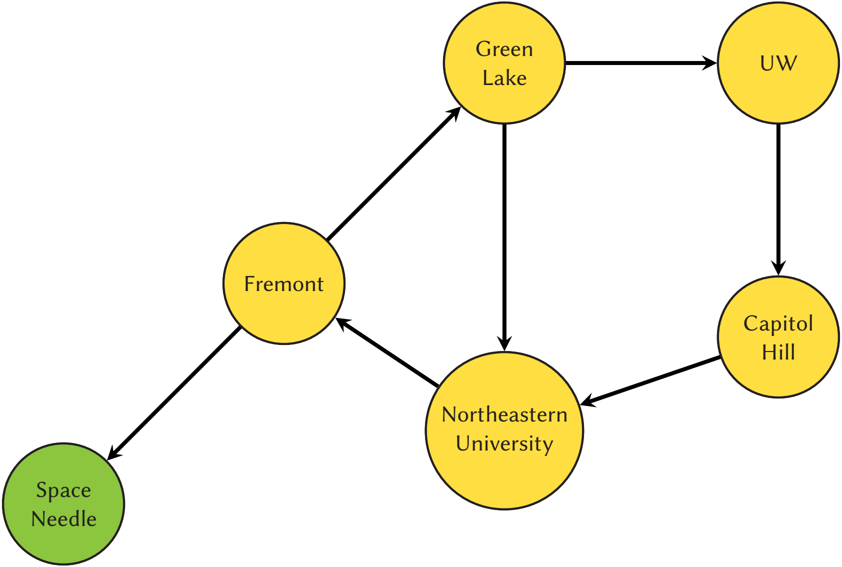
What's the best way for me to get from Green Lake to Space Needle?



What's the best way for me to get from Green Lake to Space Needle?



What's the best way for me to get from Green Lake to Space Needle?



What's the best way for me to get from Green Lake to Space Needle?

BFS: The Algorithm

- Start at the start.
- Look at all the neighbors. Are any of them the destination?
- If no:
  - Look at all the neighbors of the neighbors. Are any of them the destination?
  - Look at all the neighbors of the neighbors of the neighbors. Are any of them the destination?

BFS: Runtime

- If you search the entire network, you traverse each edge at least once: $O(|E|)$
  - That is, O(number of edges)
- Keeping a queue of who to visit in order.
  - Add single node to queue: $O(1)$
  - For all nodes: O(number of nodes)
  - $O(|V|)$
- Together, it's $O(V + E)$

- Depth first search needs to check which nodes have been output or else it can get stuck in loops.
- In a connected graph, a BFS will print all nodes, but it will repeat if there are cycles and may not terminate
- As an aside, in-order, pre-order and postorder traversals only make sense in binary trees, so they aren't important for graphs. However, we do need some way to order our out-vertices (left and right in BST).

Traverse, psuedocode

```
void traverseGraph(Node start) {
  Set pending = emptySet()
  pending.add(start)
  mark start as visited
  while(pending is not empty) {
    next = pending.remove()
    for each node u adjacent to next
      if (u is not marked visited) {
        mark u
        pending.add(u)
      }
    }
  }
```

Using

- Assuming we can add and remove from our "pending" DS in $O(1)$ time, the entire traversal is $O(|E|)$
- Traversal order depends on what we use for our pending DS.
  - Stack : DFS
  - Queue: BFS
- These are the main traversal techniques in CS, but there are others!

- Breadth-first always finds shortest length paths, i.e., "optimal solutions"
- Better for "what is the shortest path from x to y"
  - But depth-first can use less space in finding a path
- If longest path in the graph is p and highest out- degree is d then DFS stack never has more than $d * p$ elements
- But a queue for BFS may hold $O(|V|)$ nodes

DFS

BFS vs DFS: Problems

BFS Applications

- Connected components
- Two-coloring graphs

DFS Applications

- Finding cycles
- Topological Sorting
- Strongly Connected Components

Single-Source Shortest Path

Input Directed graph with non-negative weighted edges, a starting node s and a destination node d

Problem Starting at the given node s, find the path with the lowest total edge weight to node d

Example A map with cities as nodes and the edges are distances between the cities. Find the shortest distance between city 1 and city 2.

Djikstra's Algorithm: Overview

- Find the "cheapest" node— the node you can get to in the shortest amount of time.
- Update the costs of the neighbors of this node.
- Repeat until you've done this for each node.
- Calculate the final path.

Djikstra's Algorithm: Formally

DJIKSTRA(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
    $u =$ EXTRACT-MIN(Q)
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        RELAX (u, v, w)

DJIKSTRA(G, w, s)

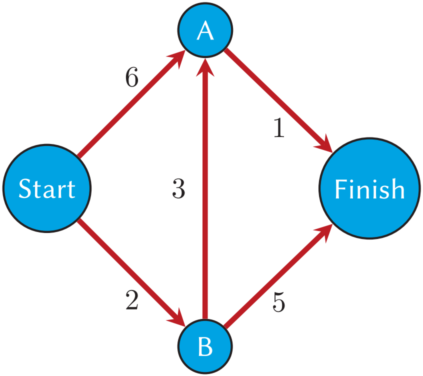
$\triangleright$ G is a graph
$\triangleright$ w is the weighting function such that $w(u, v)$ returns the weight of the
$\triangleright$ s is the starting node
for each vertex $u \in G$
    $u.d = w(s, u)$ $\triangleright$ where $w(s, u) = \infty$ if there is no edge (s, u).
$S = \emptyset$ $\triangleright$ Nodes we know the distance to
$Q = G.V$ $\triangleright$ min-PriorityQueue starting with all our nodes, ordered by dis
while $Q \neq \emptyset$
    $u =$ EXTRACT-MIN(Q) $\triangleright$ Greedy step: get the closest node
    $S = S \cup \{u\}$ $\triangleright$ Set of nodes that have shortest-path-distance found
    for each vertex $v \in G.Adj[u]$
        RELAX (u, v, w)

RELAX(u, v, w)

$\triangleright$ u is the start node
$\triangleright$ v is the destination node
$\triangleright$ w is the weight function

Djikstra's: A walkthrough

- Find the "cheapest" node— the node you can get to in the shortest amount of time.
- Update the costs of the neighbors of this node.
- Repeat until you've done this for each node.
- Calculate the final path.

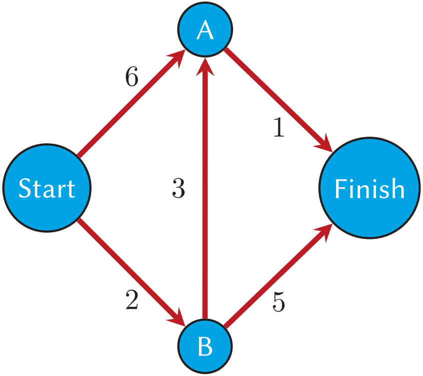


Breadth First Search: distance = 7

Step 1: Find the cheapest node

1. Should we go to A or B?
   - Make a table of how long it takes to get to each node from this node.
   - We don't know how long it takes to get to Finish, so we just say infinity for now.

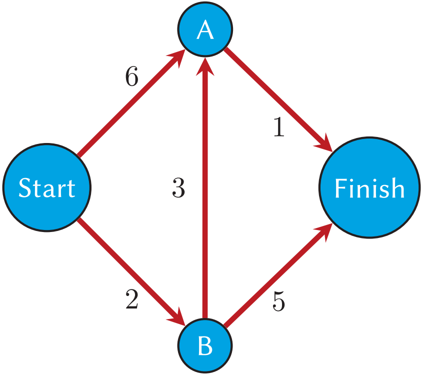
Node	Time to Node
A	6
B	2
Finish	∞



Step 2: Take the next step

1. Calculate how long it takes to get (from Start) to B's neighbors by following an edge from B
   - We chose B because it's the fastest to get to.
   - Assume we started at Start, went to B, and then now we're updating Time to Nodes.

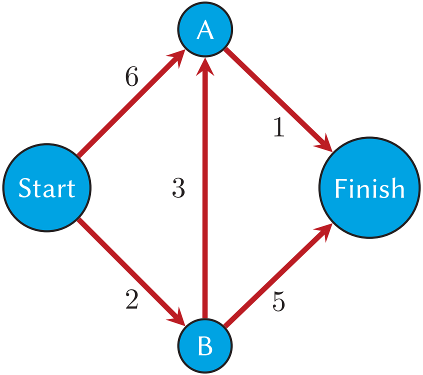
Node	Time to Node
A	~~6~~5
B	2
Finish	~~∞~~ 7



Step 3: Repeat!

1. Find the node that takes the least amount of time to get to.
   - We already did B, so let's do A.
   - Update the costs of A's neighbors
     - Takes 5 to get to A; 1 more to get to Finish

Node	Time to Node
A	~~6~~5
B	2
Finish	~~7~~6



Section 3

Topological Ordering

Topological Ordering

Input: Directed acyclic graph $G = (V, E)$

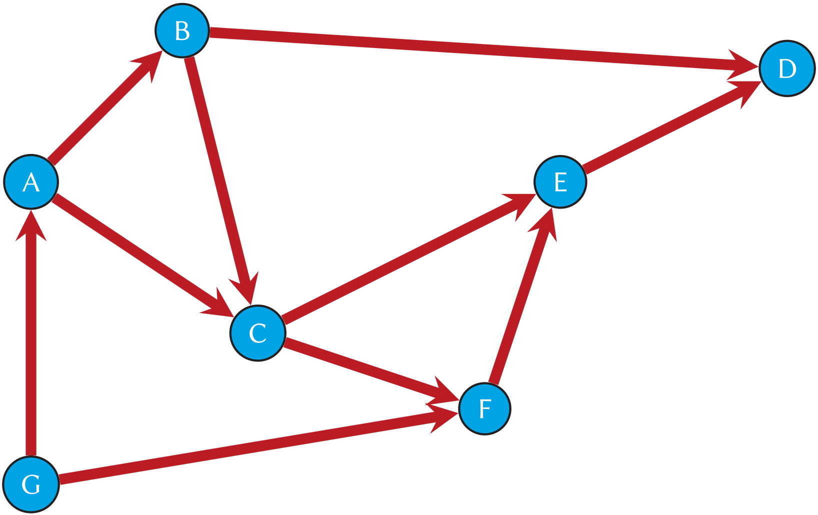
Problem: Find a linear ordering of the vertices V such that for each edge (i, j) in E, vertex i is to the left of j.

Example: Scheduling of tasks that have precedence constraints

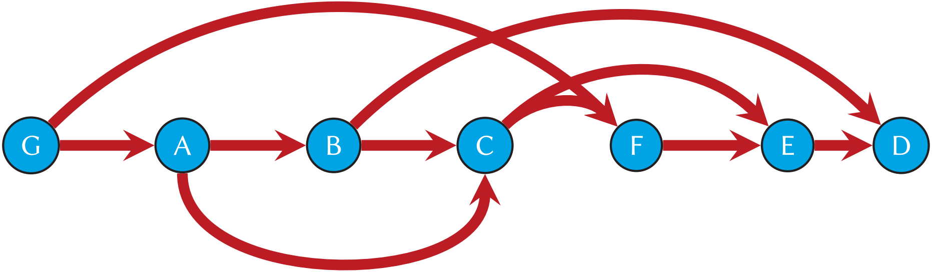
Example: Class Ordering

- Vertices are classes, and edges represent pre-reqs
- A ***topological ordering*** is any ordering that is a valid sequence of courses

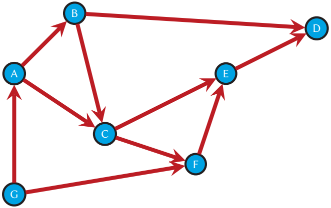
Toposort: Example Input

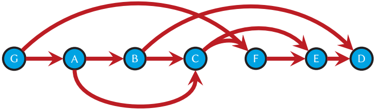


Toposort: Example Output



Toposort: Convince yourself





- Nodes are ordered in a linear fashion
- All the (directed) edges point to "future" nodes
- No edges are "pointing back"

Things to Know

- Why only DAGs?

Terminology: A DAG represents a ***partial order*** and a topological sort produces a ***total order*** that is consistent with it

Things to Know

- Why only DAGs?
  - A cycle means there's no correct answer

Terminology: A DAG represents a ***partial order*** and a topological sort produces a ***total order*** that is consistent with it

Things to Know

- Why only DAGs?
  - A cycle means there's no correct answer
- Can every DAG be topo sorted?

Terminology: A DAG represents a ***partial order*** and a topological sort produces a ***total order*** that is consistent with it

Things to Know

- Why only DAGs?
  - A cycle means there's no correct answer
- Can every DAG be topo sorted?
  - Yes.

Terminology: A DAG represents a ***partial order*** and a topological sort produces a ***total order*** that is consistent with it

Things to Know

- Why only DAGs?
  - A cycle means there's no correct answer
- Can every DAG be topo sorted?
  - Yes.
- Is there always a unique answer?

Terminology: A DAG represents a ***partial order*** and a topological sort produces a ***total order*** that is consistent with it

Things to Know

- Why only DAGs?
  - A cycle means there’s no correct answer
- Can every DAG be topo sorted?
  - Yes.
- Is there always a unique answer?
  - No, DAGs can be sorted in different ways

Terminology: A DAG represents a ***partial order*** and a topological sort produces a ***total order*** that is consistent with it

Things to Know

- Why only DAGs?
  - A cycle means there's no correct answer
- Can every DAG be topo sorted?
  - Yes.
- Is there always a unique answer?
  - No, DAGs can be sorted in different ways
  - Especially when there are fewer constraints

Terminology: A DAG represents a ***partial order*** and a topological sort produces a ***total order*** that is consistent with it

Using topological sort

- Figuring out how to graduate

Using topological sort

- Figuring out how to graduate
- Computing an order in which to recompute cells in a spreadsheet

Using topological sort

- Figuring out how to graduate
- Computing an order in which to recompute cells in a spreadsheet
- Determining an order to compile files using a Makefile

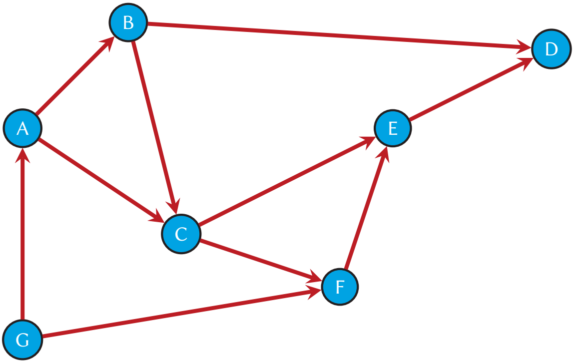
Using topological sort

- Figuring out how to graduate
- Computing an order in which to recompute cells in a spreadsheet
- Determining an order to compile files using a Makefile
- In general, taking a dependency graph and finding an order of execution

Toposort: The Algorithm

- Mark each vertex with its in-degree
- While there are vertices not yet in the final output:
  - Choose a vertex v with labeled in-degree of 0
  - Output v and conceptually remove it from the graph
  - For each vertex u adjacent to v:
    - Decrement the in-degree of u

An Example...



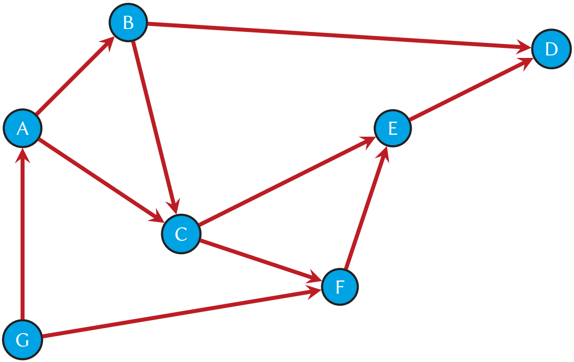
Node	In-Degree
A	1
B	1
C	2
D	2
E	2
F	2
G	0

Output order:

< >

First, calculate the in-degree of each node.

An Example...



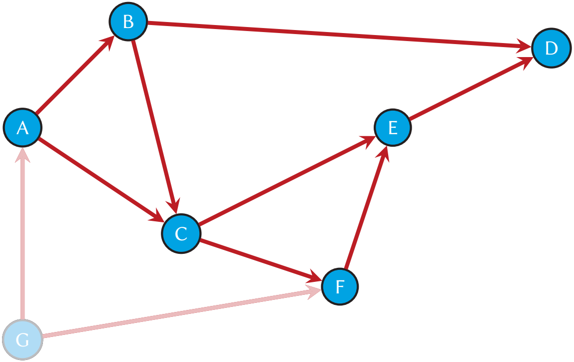
Node	In-Degree
A	1
B	1
C	2
D	2
E	2
F	2
G	0

Output order:

< >

Start with the node with in-degree of 0

An Example...



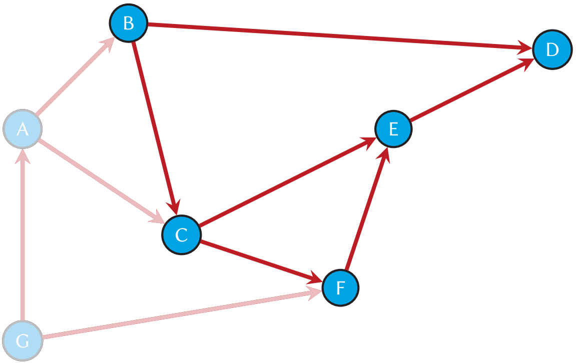
Node	In-Degree
A	~~1~~ 0
B	1
C	2
D	2
E	2
F	~~2~~ 1
G	0

Output order:

<G >

Output that node, “remove it from the graph”, decrementing the in-degree for each node it points to.

An Example...



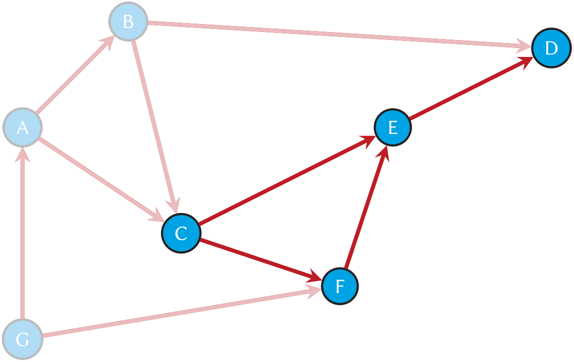
Node	In-Degree
A	~~1~~ 0
B	~~1~~ 0
C	~~2~~ 1
D	2
E	2
F	~~2~~ 1
G	0

Output order:

<G , A >

Repeat.

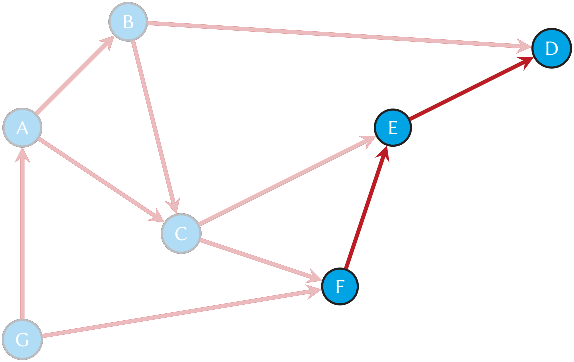
An Example...



Node	In-Degree
A	$\cancel{1}$ 0
B	$\cancel{1}$ 0
C	$\cancel{2}$ $\cancel{1}$ 0
D	$\cancel{2}$ 1
E	2
F	$\cancel{2}$ 1
G	0

Output order:

<G , A , B >

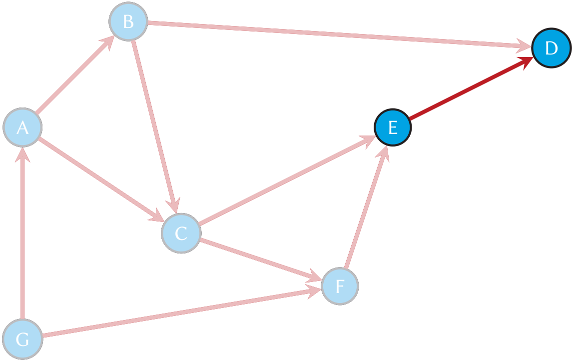


Node	In-Degree
A	$\cancel{1}$ 0
B	$\cancel{1}$ 0
C	$\cancel{2}$ $\cancel{1}$ 0
D	$\cancel{2}$ 1
E	$\cancel{2}$ 1
F	$\cancel{2}$ $\cancel{1}$ 0
G	0

Output order:

<G , A , B , C >

An Example...

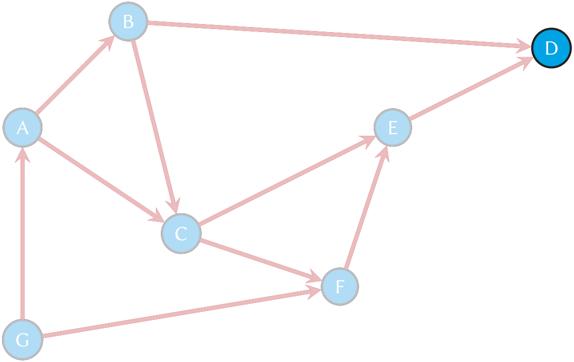


Node	In-Degree
A	$\cancel{1}$ 0
B	$\cancel{1}$ 0
C	$\cancel{2}$ $\cancel{1}$ 0
D	$\cancel{2}$ 1
E	$\cancel{2}$ 1 $\cancel{2}$ $\cancel{1}$ 0
F	$\cancel{2}$ $\cancel{1}$ 0
G	0

Output order:

<G , A , B , C , F >

An Example...

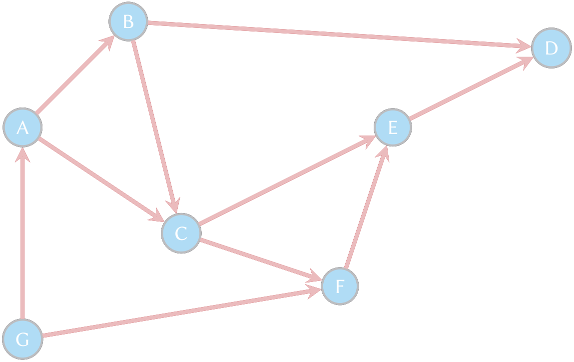


Node	In-Degree
A	$\cancel{1}$ 0
B	$\cancel{1}$ 0
C	$\cancel{2}$ $\cancel{1}$ 0
D	$\cancel{2}$ 1 $\cancel{2}$ $\cancel{1}$ 0
E	$\cancel{2}$ 1 $\cancel{2}$ $\cancel{1}$ 0
F	$\cancel{2}$ $\cancel{1}$ 0
G	0

Output order:

<G , A , B , C , F , E >

An Example...



Node	In-Degree
A	$\cancel{1}$ 0
B	$\cancel{1}$ 0
C	$\cancel{2}$ $\cancel{1}$ 0
D	$\cancel{2}$ 1 $\cancel{2}$ $\cancel{1}$0
E	$\cancel{2}$ 1 $\cancel{2}$ $\cancel{1}$ 0
F	$\cancel{2}$ $\cancel{1}$ 0
G	0

Output order:

<G , A , B , C , F , E , D >

Toposort Notes

- Always need a vertex with in-deg 0 to start
- And we will always have one because there are no cycles!
    - When we have more than one vertex with in-deg 0, it doesn?t matter which we choose.
- This is how we get more than one correct answer

Toposort: Implementation Details

- Don't want to have to search for a zero-degree node every time
- Keep the "pending" 0-deg nodes in a list/stack/queue/etc
- Note, your choice of data structure impacts order or output, but not correctness or efficiency
  - ...as long as push/pop $=O(1)$

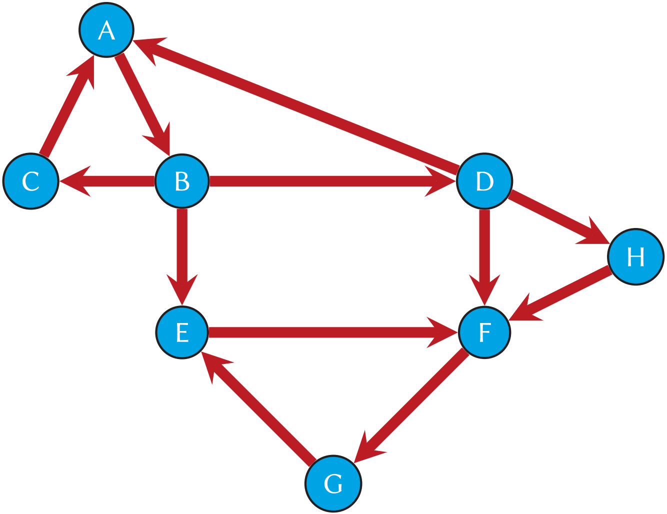
Section 4

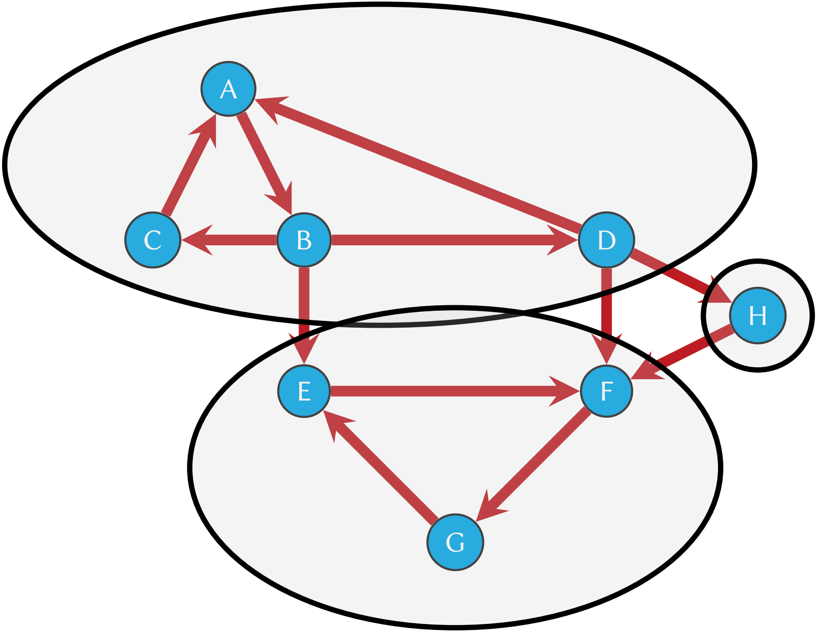
Strongly Connected Components

A directed graph is ***strongly connected*** if there is a directed path between any two vertices.

A directed graph is ***strongly connected*** if there is a directed path between any two vertices.

The ***strongly connected components*** of a graph is a partition of the vertices into subsets (maximal) such that each subset is strongly connected.


A
B
C
D
E
F
G
H


A
B
C
D
E
F
G
H

Strongly Connected Components

The Input: Directed or Undirected graph G

The Problem: Identify the components of G where vertices x and y are members of different components if no path exists from x to y in G.

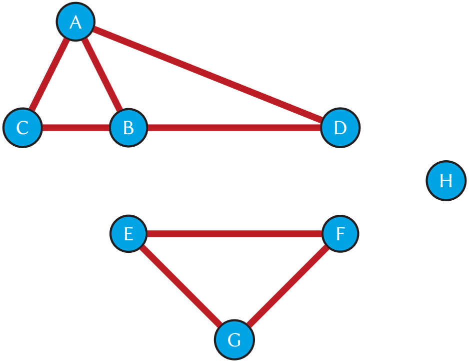
In an undirected graph G, components are ***connected*** or not.

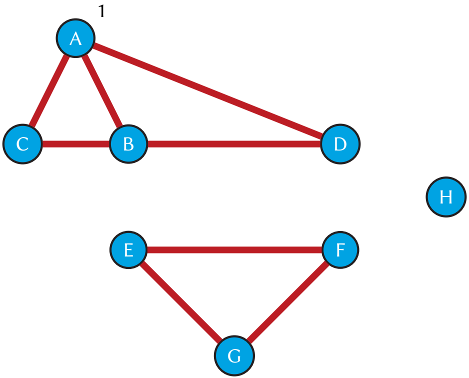
In a directed graph G, components can be ***strongly connected*** or not.

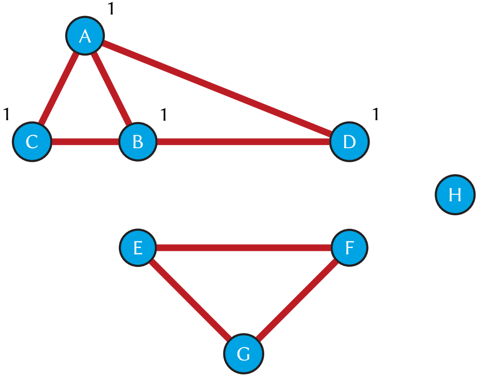
Connected Components: Algorithm Overview

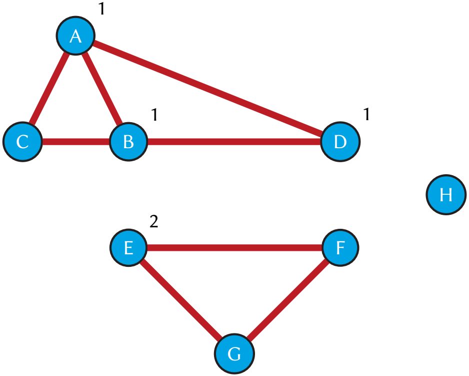
Connected Components in an undirected graph

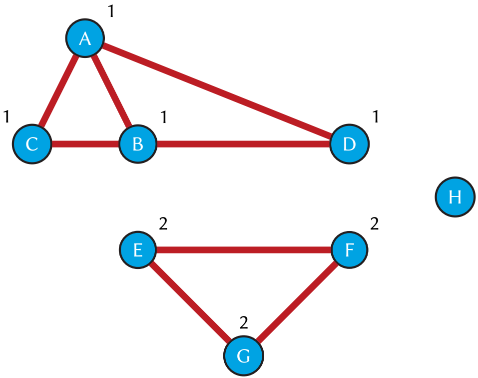
- Use either DFS or BFS
- Set the component_id for each node to 0.
- Set cur_component = 1.
- Start traversing:
  - For every node, set the component_id = cur_component.
  - When there are no more nodes to traverse, increment cur_component.

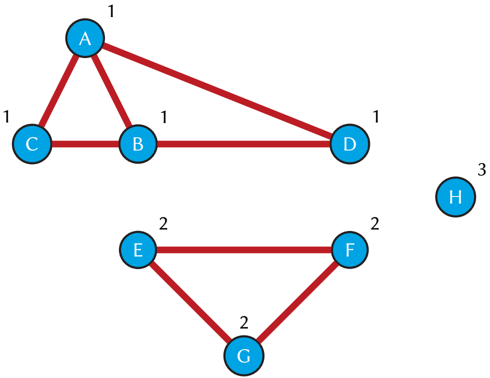

A
C
B
D
H
E
F
G


1
A
C
B
D
H
E
F
G


1
A
1
C
1
B
1
D
H
E
F
G


1
A
1
C
1
B
1
D
H
2
E
F
G


1
A
1
C
B
1
D
1
H
2
E
F
2
2
G

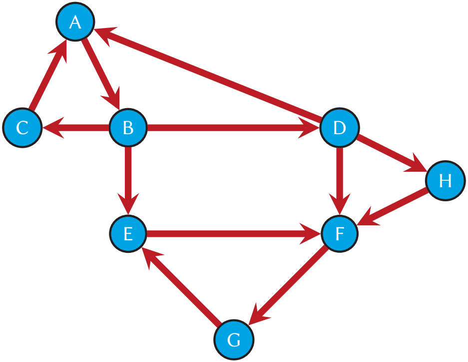

1
A
1
C
1
B
1
D
3
H
2
E
2
F
2
G

Simple for undirected... but what about directed??

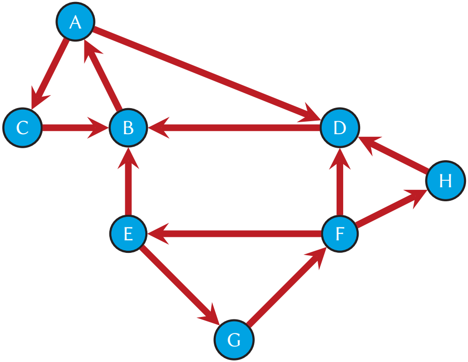
Transpose Graph

- $G^T = (V, E^T)$:
  - $E^T = \{(u, v) : (v, u) \text{ in } E\}$
  - E^T consists of the edges in G with their directions reversed
- G and G^T have the same strongly connected components

G and G^T



G



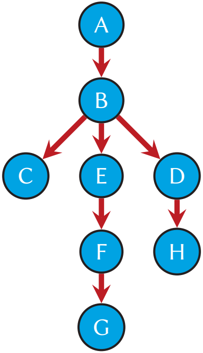
G^T

Strongly Connected Components: The Algorithm

1. Call $DFS(G)$ to compute finishing times $f[u]$ for each vertex u
2. Compute G^T
3. Call $DFS(G^T)$, but follow the vertices in order of decreasing $f[u]$
4. Output the vertices of each tree in the depth- first forest of step 3 as a separate component

Calculating Finish time

- When doing a graph traversal, there's a step that you start evaluating a node v and it's adjacency list.
- The finish time marks when the search finish's exploring v's adjacency list.

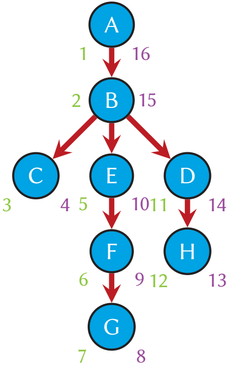


A DFS tree representing a traversal of the graph starting at Node A.

Node ID	$f(u)$
A	
B	
C	
D	
E	
F	
G	
H	

Calculating Finish time

- When doing a graph traversal, there's a step that you start evaluating a node v and it's adjacency list.
- The finish time marks when the search finish's exploring v's adjacency list.
- In the following graph, the start times are marked in green; the finish times are marked in purple.

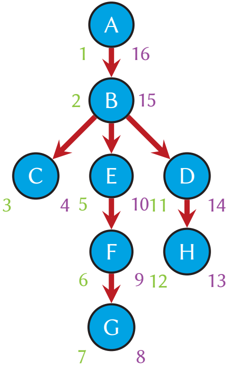


A DFS tree representing a traversal of the graph starting at Node A.

Node ID	$f(u)$
A	
B	
C	
D	
E	
F	
G	
H	

Calculating Finish time

- When doing a graph traversal, there's a step that you start evaluating a node v and it's adjacency list.
- The finish time marks when the search finish's exploring v's adjacency list.
- In the following graph, the start times are marked in green; the finish times are marked in purple.



A DFS tree representing a traversal of the graph starting at Node A.

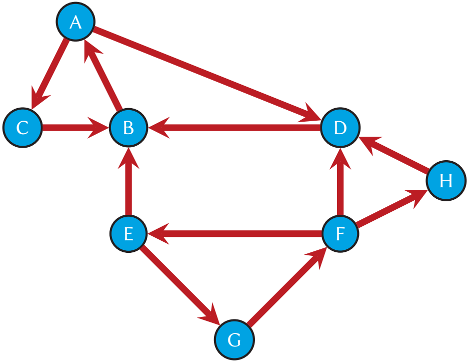
Node ID	$f(u)$
A	16
B	15
C	4
D	14
E	10
F	9
G	8
H	13

Step 2: Computer G^T

- A number of ways to do this, but whatever way, make sure the edges are reversed.

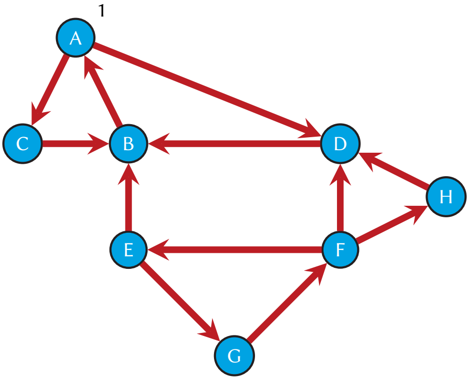
Step 3: Do DFS on G^T

- Find the node with the LAST finish time
- DFS to all the other nodes you can, without visiting a node twice
- Those are all Component 1.
- While there are more nodes that haven't been visited, repeat.



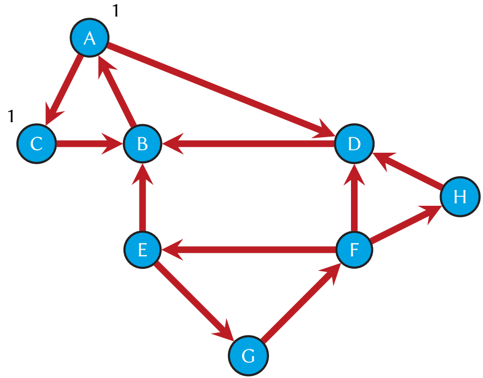
G^T

Node ID	$f(u)$
A	16
B	15
C	4
D	14
E	10
F	9
G	8
H	13



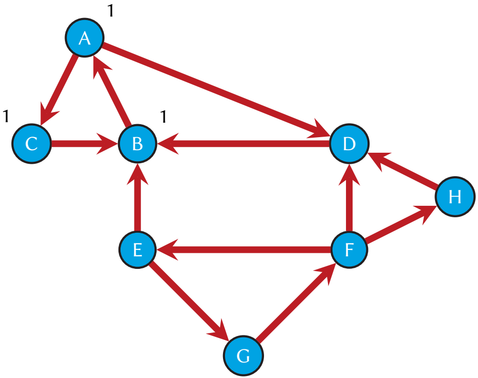
G^T

Node ID	$f(u)$
A	16
B	15
C	4
D	14
E	10
F	9
G	8
H	13



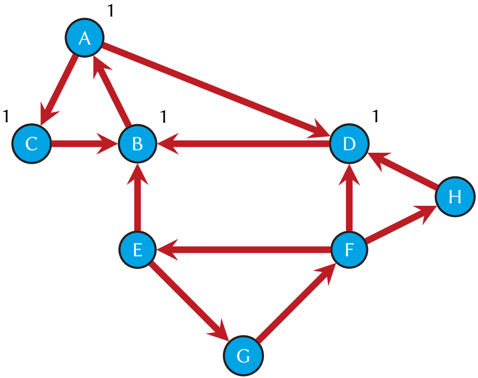
G^T

Node ID	$f(u)$
A	16
B	15
C	4
D	14
E	10
F	9
G	8
H	13



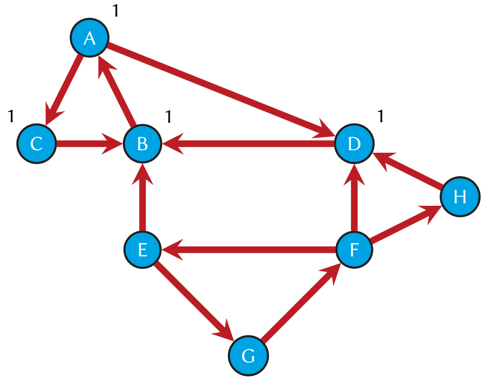
G^T

Node ID	$f(u)$
A	16
B	15
C	4
D	14
E	10
F	9
G	8
H	13



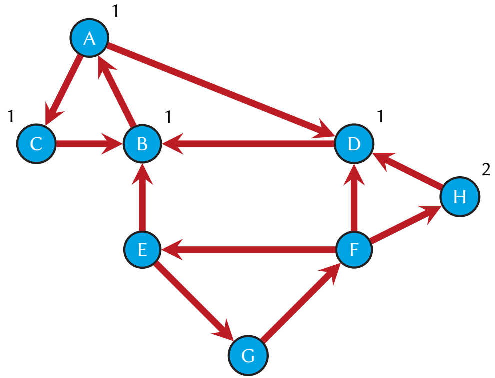
G^T

Node ID	$f(u)$
A	16
B	15
C	4
D	14
E	10
F	9
G	8
H	13



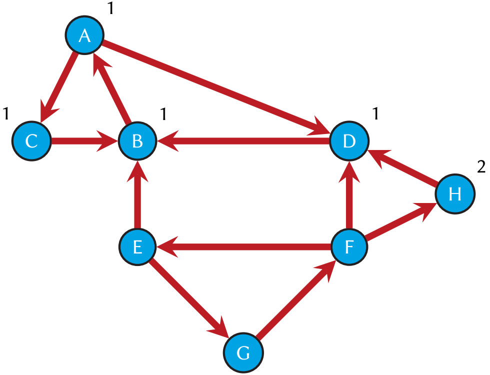
G^T

Node ID	$f(u)$
A	~~16~~-1
B	~~15~~-1
C	~~4~~-1
D	~~14~~-1
E	10
F	9
G	8
H	13



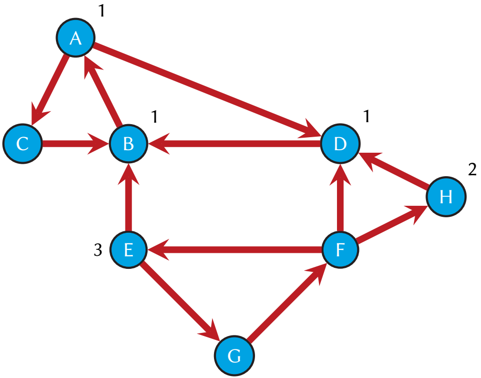
G^T

Node ID	$f(u)$
A	~~16~~-1
B	~~15~~-1
C	~~4~~-1
D	~~14~~-1
E	10
F	9
G	8
H	13



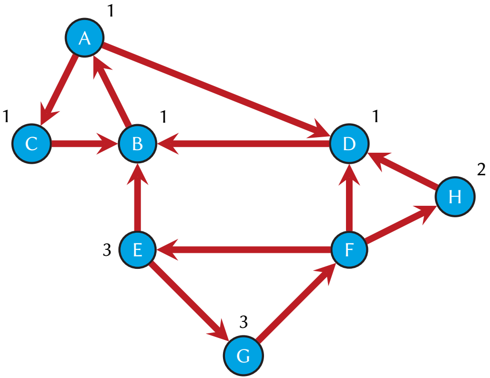
G^T

Node ID	$f(u)$
A	~~16~~-1
B	~~15~~-1
C	~~4~~-1
D	~~14~~-1
E	10
F	9
G	8
H	~~13~~-1



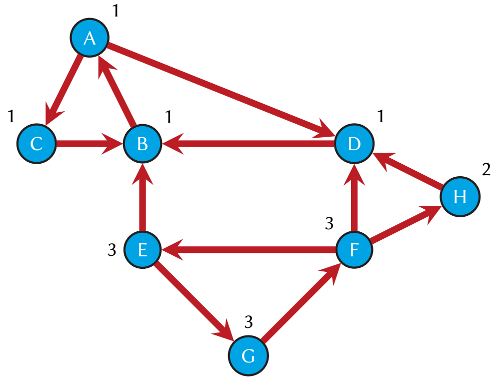
G^T

Node ID	$f(u)$
A	~~16~~-1
B	~~15~~-1
C	~~4~~-1
D	~~14~~-1
E	10
F	9
G	8
H	~~13~~-1



G^T

Node ID	$f(u)$
A	~~16~~-1
B	~~15~~-1
C	~~4~~-1
D	~~14~~-1
E	10
F	9
G	8
H	~~13~~-1



G^T

Node ID	$f(u)$
A	$\cancel{16}$-1
B	$\cancel{15}$-1
C	$\cancel{4}$-1
D	$\cancel{14}$-1
E	10
F	9
G	8
H	$\cancel{13}$-1

Section 5

Summary

Summary

What problems did we work on today?

- Sorting
- Strongly Connected Components