Lecture 13: DAGs and Shortest Paths

Tim LaRock

larock.t@northeastern.edu

bit.ly/cs3000syllabus

Business

Homework 3: Due next Monday at midnight Boston time via Gradescope!

Midterm 2: Will be June 10^{th} through June 12^{th} (same deal as last time)

- Topics will be graph algorithms and network flow

Final exam: Will be June 18^{th} (8PM) through June 22^{nd} (8PM)

- Whole course will be fair game, but focus will be on last 2 weeks

There will be 1-2 more (short) homeworks

Today

Review typology of edges in a DFS tree

Post-ordering of nodes in a traversal

Directed acyclic graphs and topological node orderings

Introduction to node betweenness centrality

Shortest path algorithms to compute betweenness centrality

Typology of Edges in DFS

For every node discovered during a DFS execution, we can keep track of its parent.

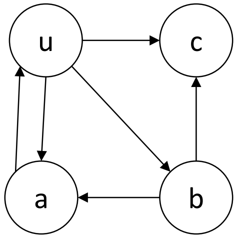
The graph of the parent-child relationships is a tree where each edge can be assigned to one of four types:

Tree edge:
- Explore new nodes

Forward edge:
- Ancestor to descendant

Backward edge:
- Descendant to ancestor

Cross edges:
- No ancestral relationship



Typology of Edges in DFS

For every node discovered during a DFS execution, we can keep track of its parent.

The graph of the parent-child relationships is a tree where each edge can be assigned to one of four types:

Tree edge: $(u, a), (u, b), (b, c)$
- Explore new nodes

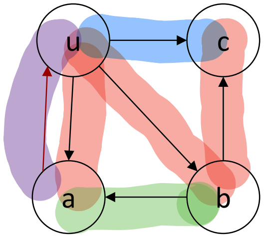
Forward edge: (u, c)
- Ancestor to descendant

Backward edge: (a, u)
- Descendant to ancestor

Cross edges: (b, a)
- No ancestral relationship

$u \to a$

$u \to b \to a$



Backwards edges identify *cycles* in the graph!

A cycle is a closed walk (starts and ends at the same vertex) that visits each vertex in the walk at most once.

Post-Order

A *post-ordering* of a graph $G = (V, E)$ is an ordering of the nodes based on "when" DFS from each node finished.

To get a post-order, we maintain a global clock variable that is initialized to 1.

Every time we finish calling DFS on all of a node's neighbors, we set its post-order value to the current value of clock, then increment clock.

Recursive DFS with post-ordering

```
G = (V,E) is a graph
visited[u] = 0 for all u ∈ V
clock = 1

DFS(u):
  visited[u] = 1
  For v ∈ Neighbors(u):
    If visited[v] = 0:
      parent[v] = u
      DFS(v)

  post-visit(u)
```

```
post-visit(u):
  set postorder[u] = clock
  clock ← clock + 1
```

Post-Order

DFS(node)

node	visited	clock
u	1	~~1~~ ~~2~~ ~~3~~ 4
a	1	1
b	1	~~2~~ 3
c		2

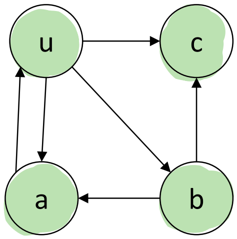
Recursive DFS with post-ordering

$G = (V, E)$ is a graph
visited$[u] = 0$ for all $u \in V$
clock = 1

$DFS(u)$:
  visited[u] = 1
  For $v \in Neighbors(u)$:
    If visited[v] = 0:
      parent[v] = u
      DFS(v)

  post-visit(u)

post-visit(u):
  set postorder[u] = clock
  clock ← clock + 1



Vertex	u	a	b	c
Postorder	4	1	3	2

Post-Order

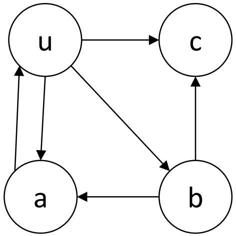
Recursive DFS with post-ordering

```
G = (V,E) is a graph
visited[u] = 0 for all u ∈ V
clock = 1

DFS(u):
  visited[u] = 1
  For v ∈ Neighbors(u):
    If visited[v] = 0:
      parent[v] = u
      DFS(v)

  post-visit(u)
```

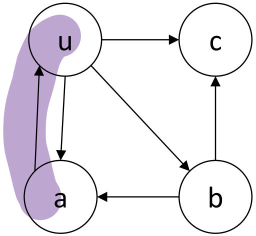
```
post-visit(u):
  set postorder[u] = clock
  clock ← clock + 1
```



Vertex	u	a	b	c
Postorder	4	1	3	2

Post-Order

Observation: If postorder[u] < postorder[v], then (u, v) is a backwards edge! Why? Consider backwards edge (a, u)

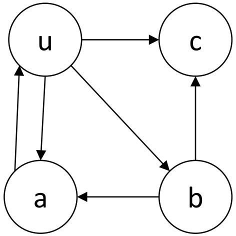


Vertex	u	a	b	c
Postorder	4	1	3	2

Post-Order

Observation: If postorder[u] < postorder[v], then (u, v) is a backwards edge! Why? Consider backwards edge (a, u)

- DFS(a) can't finish until its children are finished

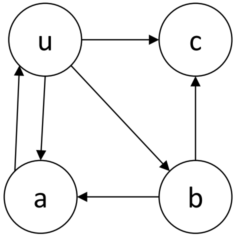


Vertex	u	a	b	c
Postorder	4	1	3	2

Post-Order

Observation: If postorder$[u]$ < postorder$[v]$, then (u, v) is a backwards edge! Why? Consider backwards edge (a, u)

- DFS(a) can't finish until its children are finished
- Since postorder$[a]$ < postorder$[u]$, then DFS(a) finished before DFS(u), meaning DFS(u) was not called by DFS(a)

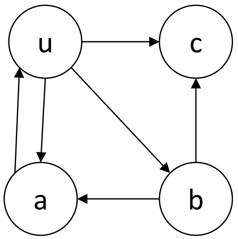


Vertex	u	a	b	c
Postorder	4	1	3	2

Post-Order

Observation: If postorder[u] < postorder[v], then (u, v) is a backwards edge! Why? Consider backwards edge (a, u)

- DFS(a) can't finish until its children are finished
- Since postorder[a] < postorder[u], then DFS(a) finished before DFS(u), meaning DFS(u) was not called by DFS(a)

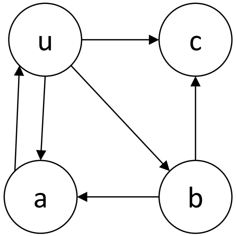


Vertex	u	a	b	c
Postorder	4	1	3	2

Post-Order

Observation: If postorder[u] < postorder[v], then (u, v) is a backwards edge! Why? Consider backwards edge (a, u)

- DFS(a) can't finish until its children are finished
- Since postorder[a] < postorder[u], then DFS(a) finished before DFS(u), meaning DFS(u) was not called by DFS(a)
- For this situation to arise, when we ran DFS(a), we must have had visited[u] = 1, implying DFS(u) ran first



Vertex	u	a	b	c
Postorder	4	1	3	2

Post-Order

$u \to a$

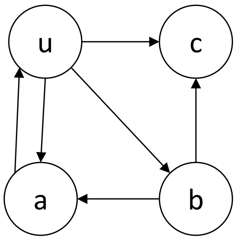
$a \to u$

(a, u)

$1 < 4$

Observation: If postorder[u] < postorder[v], then (u, v) is a backwards edge! Why? Consider backwards edge (a, u)

- DFS(a) can't finish until its children are finished
- Since postorder[a] < postorder[u], then DFS(a) finished before DFS(u), meaning DFS(u) was not called by DFS(a)
- For this situation to arise, when we ran DFS(a), we must have had visited[u] = 1, implying DFS(u) ran first
- Which means DFS(u) started first but finished after DFS(a), which can only happen for a backwards edge!

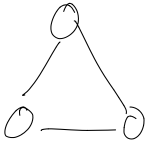


Vertex	u	a	b	c
Postorder	4	1	3	2

Putting the pieces together

$a \to d$

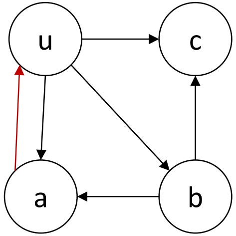
$d \to a$



We determined that backward edges in a DFS tree identify cycles in a graph.

We then showed that we can use DFS with post-ordering to identify backwards edges.

So we can use DFS with post-ordering to determine whether a graph has cycles!



Backwards edges identify *cycles* in the graph!

A cycle is a closed walk (starts and ends at the same vertex) that visits each vertex in the walk at most once.

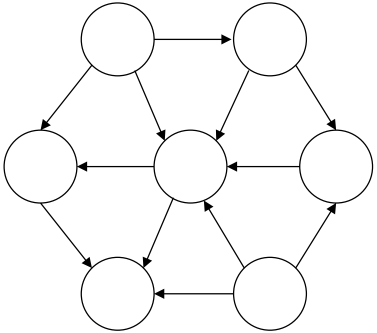
If postorder[u] < postorder[v], then (u, v) is a backwards edge!

Vertex	u	a	b	c
Postorder	4	1	3	2

Directed Acyclic Graphs and Topological Ordering

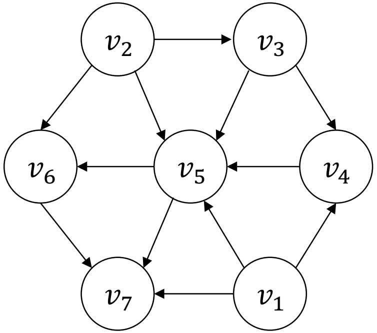
Directed Acyclic Graph (DAG)

- A directed graph with no cycles
- Represent precedence relationships
  - “this” comes before “that”
  - “this” is prior to “that”

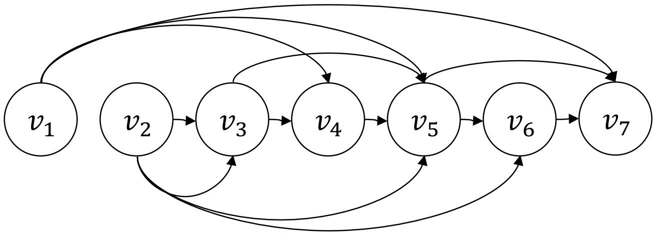


Directed Acyclic Graph (DAG)

- A directed graph with no cycles
- Represent precedence relationships
  - "this" comes before "that"
  - "this" is prior to "that"



A *topological ordering* of a directed graph is a labeling of the nodes so that all edges point "forward", meaning for all directed edges $(v_i, v_j), j > i$



Two problems in one

Problem 1: Is G a DAG?

- We know how to get an answer using DFS!

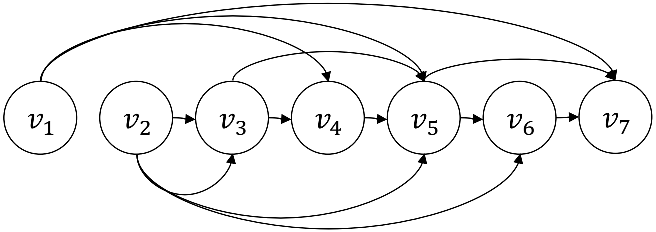
Problem 2: Given a directed graph, can it be topologically ordered?

Claim: G has a topological ordering if and only if G is a DAG

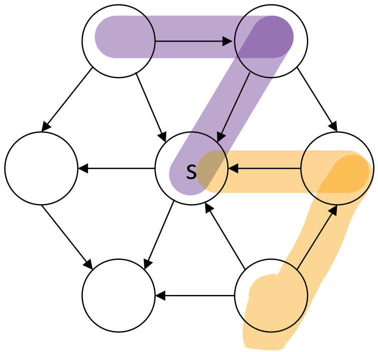
- We will design an algorithm that either outputs a topological ordering or that the graph is not a DAG

Two problems in one

Observation:
In a topological ordering, there is a node with no incoming edges!



Observation:
In a DAG, there is a node with no incoming edges!



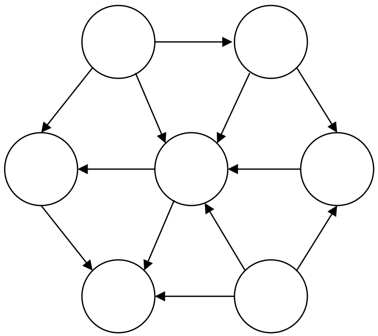
Check by following incoming links backwards until you find a node that has none.

Does every DAG have a node with no incoming edges?

Claim: For every DAG on $n \in \mathbb{N}$ nodes, there is a topological ordering.

Lemma (from previous slide): Every DAG has a node with no incoming edges.

We can prove this by induction on n.



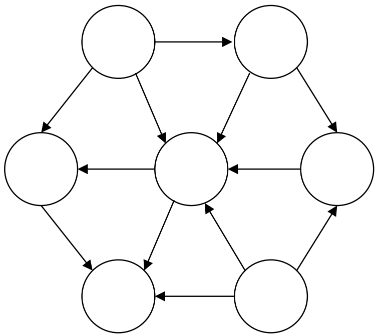
Does every DAG have a node with no incoming edges?

Claim: For every DAG on $n \in \mathbb{N}$ nodes, there is a topological ordering.

Lemma (from previous slide): Every DAG has a node with no incoming edges.

We can prove this by induction on n.

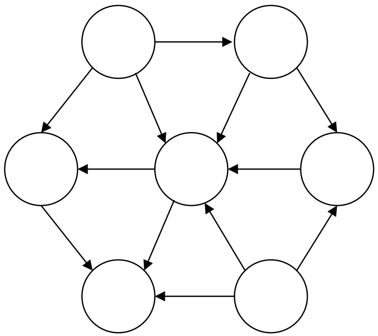
Base case: $n = 1$; trivially true



Does every DAG have a node with no incoming edges?

Claim: For every DAG on $n \in \mathbb{N}$ nodes, there is a topological ordering.

Lemma (from previous slide): Every DAG has a node with no incoming edges.



We can prove this by induction on n.

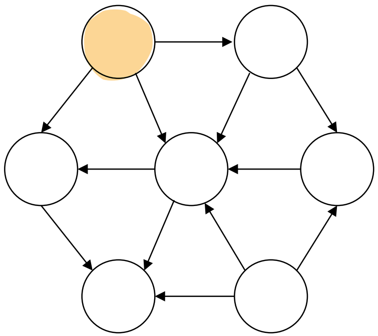
Base case: $n = 1$; trivially true

Inductive step:
- Assume topological ordering exists for DAGs up to n nodes.

Does every DAG have a node with no incoming edges?

Claim: For every DAG on $n \in \mathbb{N}$ nodes, there is a topological ordering.

Lemma (from previous slide): Every DAG has a node with no incoming edges.



We can prove this by induction on n.

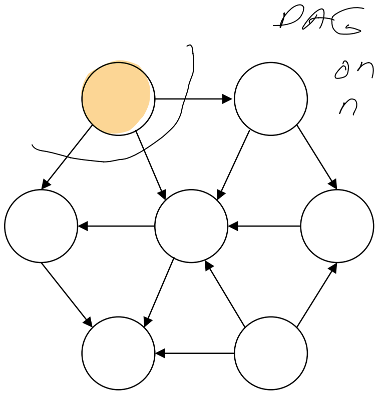
Base case: $n = 1$; trivially true

Inductive step:
- Assume topological ordering exists for DAGs up to n nodes.
- Given a dag on $n + 1$ nodes, identify a node with no incoming edges. We know at least one exists.

Does every DAG have a node with no incoming edges?

Claim: For every DAG on $n \in \mathbb{N}$ nodes, there is a topological ordering.

Lemma (from previous slide): Every DAG has a node with no incoming edges.



We can prove this by induction on n.

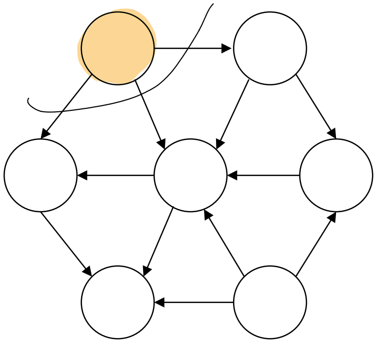
Base case: $n = 1$; trivially true

Inductive step:
- Assume topological ordering exists for DAGs up to n nodes.
- Given a dag on $n + 1$ nodes, identify a node with no incoming edges. We know at least one exists.
- Remove this node, and the remaining DAG on n nodes has a topological ordering by the inductive hypothesis.

~~Does every DAG have a node with no incoming edges?~~

Claim: For every DAG on $n \in \mathbb{N}$ nodes, there is a topological ordering.

Lemma (from previous slide): Every DAG has a node with no incoming edges.



We can prove this by induction on n.

Base case: $n = 1$; trivially true

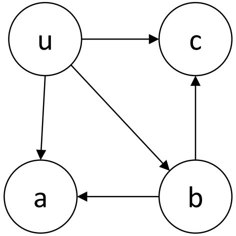
Inductive step:

- Assume topological ordering exists for DAGs up to n nodes.
- Given a dag on $n + 1$ nodes, identify a node with no incoming edges. We know at least one exists.
- Remove this node, and the remaining DAG on n nodes has a topological ordering by the inductive hypothesis.
- Since the node we removed has no incoming edges, it can be trivially added to the beginning of the ordering. Hence the claim.

Topological Orderings

Claim: Ordering nodes by decreasing post-order gives a topological ordering.

Example:

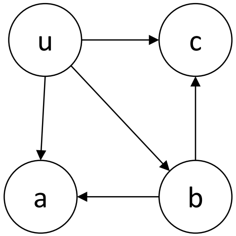


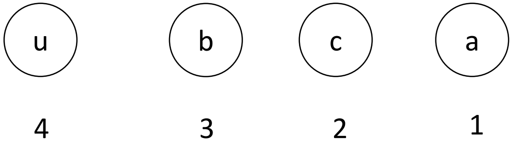
Vertex	u	a	b	c
Postorder	4	1	3	2

Topological Orderings

Claim: Ordering nodes by decreasing post-order gives a topological ordering.

Example:



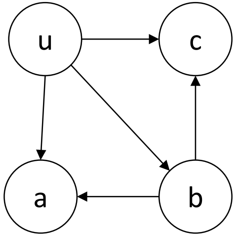


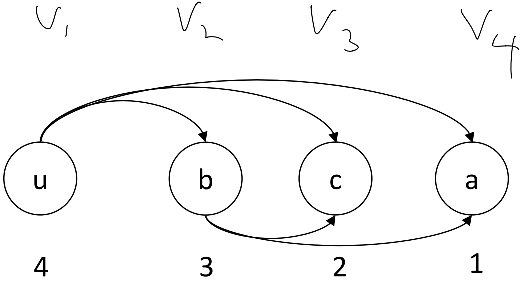
Vertex	u	a	b	c
Postorder	4	1	3	2

Topological Orderings

Claim: Ordering nodes by decreasing post-order gives a topological ordering.

Example:





Vertex	u	a	b	c
Postorder	4	1	3	2

Topological Orderings

Claim: Ordering nodes by decreasing post-order gives a topological ordering. as long as G is a DAG (i,j) $\hat{i} > \hat{j}$

Proof:

- We know that a DAG has no backward edges, since backward edges imply the presence of cycles.
- Suppose the decreasing post-ordering is not a topological ordering
  - There must be an edge (u, v) such that postorder[u] < postorder[v]
  - But such an edge would be a backward edge, implying a cycle
  - We showed such an edge can't exist in a DAG. Contradiction!

Topological Ordering Wrap

DFS(node)
increment clock
: DFS(child1)
: DFS(child2)
Return
postorder(node) = clock

A DAG is a directed graph with no cycles.

For any DAG, we can find a topological ordering in $O(n + m)$ time using DFS, since a reverse post-ordering is a topological ordering.

If we are not sure our input graph is a DAG, we can still use DFS to identify backwards edges in the DFS tree, which imply cycles.

```
TopologicalOrdering(G = (V,E)):
  Run DFS(G) with post-order
  If ∃_{u,v} s.t. postorder[u] < postorder[v]:
    Return False
  Else:
    Return reverse(postorder)
```

Shortest Paths

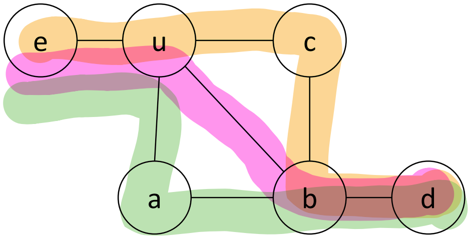
Shortest Paths: Definition

The shortest path between nodes s and t is the path between the two nodes with the fewest edges.

We can define the *distance* $d(u, v)$ between two nodes as the length of the shortest path between them.

The length of the longest shortest path in a graph is called the *diameter*.

if $(s,t) \in E$:
$d(s,t) = 1$

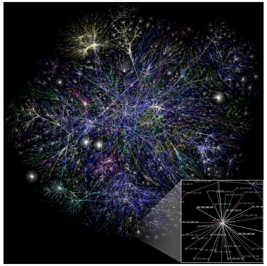


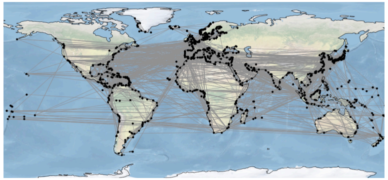
$d(e,d) = 3$

Shortest paths: Who cares?

Shortest paths: Who cares?

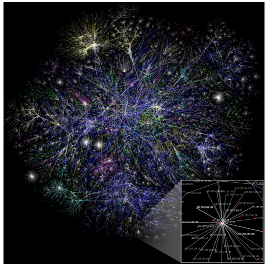
We may be interested in the shortest path between two nodes for many applications that involve things moving through networked systems, such as navigation software (e.g. Google Maps), routing internet traffic, or shipping cargo around the world

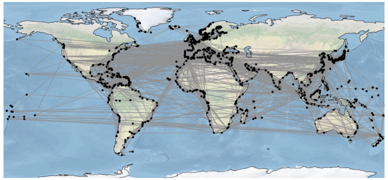




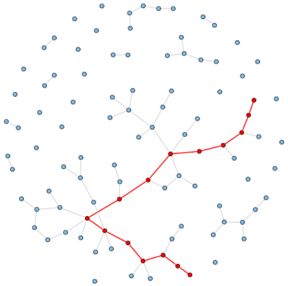
Shortest paths: Who cares?

We may be interested in the shortest path between two nodes for many applications that involve things moving through networked systems, such as navigation software (e.g. Google Maps), routing internet traffic, or shipping cargo around the world



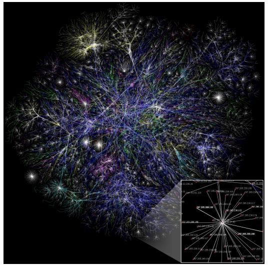


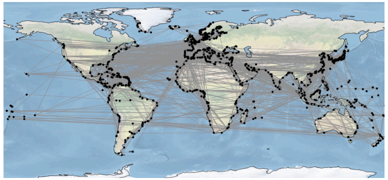
We can also use shortest paths to measure the “size” of a network, since the *longest shortest path* between any two nodes in a network is called its “diameter”



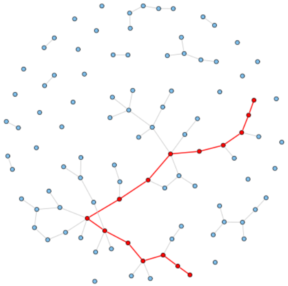
Shortest paths: Who cares?

We may be interested in the shortest path between two nodes for many applications that involve things moving through networked systems, such as navigation software (e.g. Google Maps), routing internet traffic, or shipping cargo around the world



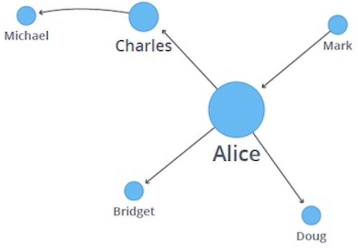


We can also use shortest paths to measure the "size" of a network, since the *longest shortest path* between any two nodes in a network is called its "diameter"



Tendency of a node to appear in shortest paths is a measure of *node centrality* called *betweenness*.

Centrality measures the "importance" of a node to various phenomena relevant to a graph. Highly central nodes often play a role in how things spread through networks (like, say, an infectious disease).

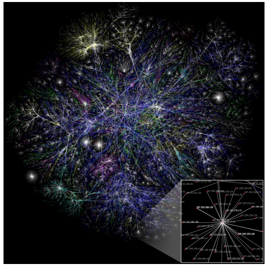


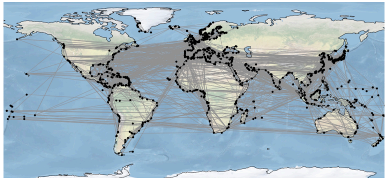
Visualization of Betweenness Centrality

Shortest paths: Who cares?

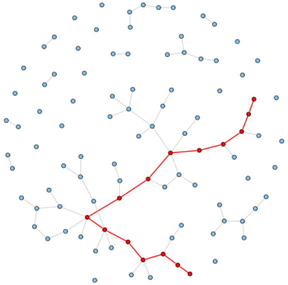
We are going to use shortest paths to compute betweenness centrality!

We may be interested in the shortest path between two nodes for many applications that involve things moving through networked systems, such as navigation software (e.g. Google Maps), routing internet traffic, or shipping cargo around the world



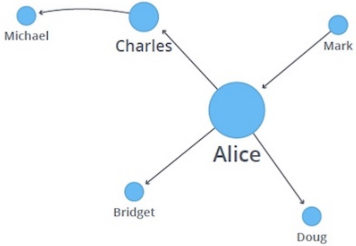


We can also use shortest paths to measure the "size" of a network, since the *longest shortest path* between any two nodes in a network is called its "diameter"



Tendency of a node to appear in shortest paths is a measure of *node centrality* called *betweenness*.

Centrality measures the "importance" of a node to various phenomena relevant to a graph. Highly central nodes often play a role in how things spread through networks (like, say, an infectious disease).



Visualization of Betweenness Centrality

Betweenness Centrality

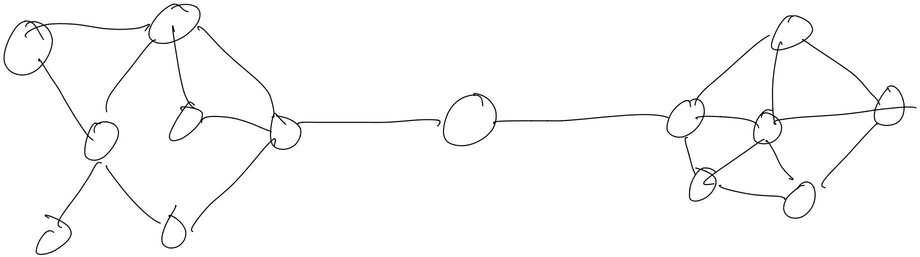
Betweenness centrality is used as a proxy for the importance of a node in facilitating connections between other nodes.

For node u, betweenness is measured as the ratio of shortest paths between all other pairs of nodes (s, t) that u lies on. Formally:

$$B(u) = \sum_{s \neq t \neq u} \frac{\sigma_{st}(u)}{\sigma_{st}}$$

# of shortest paths between s & t that involve u

Where σ_{st} is the number of shortest paths between nodes s and t and $\sigma_{st}(u)$ is the number of those shortest paths that include u.



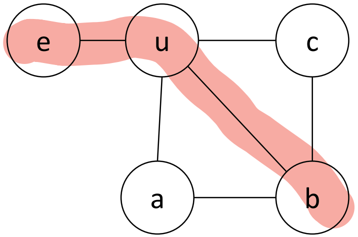
Betweenness Centrality

Betweenness centrality is used as a proxy for the importance of a node in facilitating connections between other nodes.

For node u, betweenness is measured as the ratio of shortest paths between all other pairs of nodes (s, t) that u lies on. Formally:

$$B(u) = \sum_{s \neq t \neq u} \frac{\sigma_{st}(u)}{\sigma_{st}}$$

Where σ_{st} is the number of shortest paths between nodes s and t and $\sigma_{st}(u)$ is the number of those shortest paths that include u.



$$B(e) = 0$$

$$B(u) = \frac{\sigma_{eb}(u)}{\sigma_{eb}} + \dots_{(s,t)} = \frac{1}{1} + \dots_{(s,t)}$$

How do we compute shortest paths?

To compute betweenness centrality, we need to compute shortest paths for all pairs of nodes.

Rather than jumping straight there, let's first solve a simpler problem: finding the length of the shortest path from a single node s to all other nodes in the graph (called the *single source shortest path* problem)

We can use BFS!

Single Source Shortest Paths with BFS

`dist`[v] stores the current estimate of the distance between our source node s and the node v, initialized to infinity

We walk along every edge of the graph and check whether the distance currently stored for v could be made shorter by routing through u

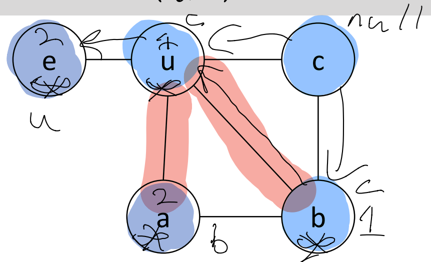
If yes, we update the distance, and store u as the *predecessor* (similar to parent) of v in the shortest path

Once BFS is done, we have the shortest path length from s to every node v stored in `dist[`v`]` and we can recover a shortest path for any node by following pred back to s

```
SSSP-BFS(s):
  dist[u] ← ∞ for all u ∈ V
  pred[u] ← null for all u ∈ V
  dist[s] ← 0
  Q ← s
  While Q is not empty:
    u ← Pull(Q)
    For v ∈ Neighbors(u):
      If dist[v] > dist[u] + 1:
        dist[v] = dist[u] + 1
        pred[v] = u
        Push(Q,v)
```

$e \leftarrow u \leftarrow c$

$d(c,b) = 1$
$d(c,a) = 2$
$d(c,u) = 1$
$d(c,e) = 2$



Next time

More shortest paths and betweenness centrality!

No new suggested reading

Single source

Work on homework 3!

All pairs shortest paths