

Graphs

COMP2611: Data Structures
2019/2020

Adapted heavily from Brown's CS16

Outline

- ▶ What is a Graph?
 - ▶ Terminology
 - ▶ Properties
 - ▶ Types of graphs
- ▶ How do we represent graphs?
 - ▶ Performance of dif
- ▶ BFS/DFS on graphs
- ▶ Applications

What is a Graph?

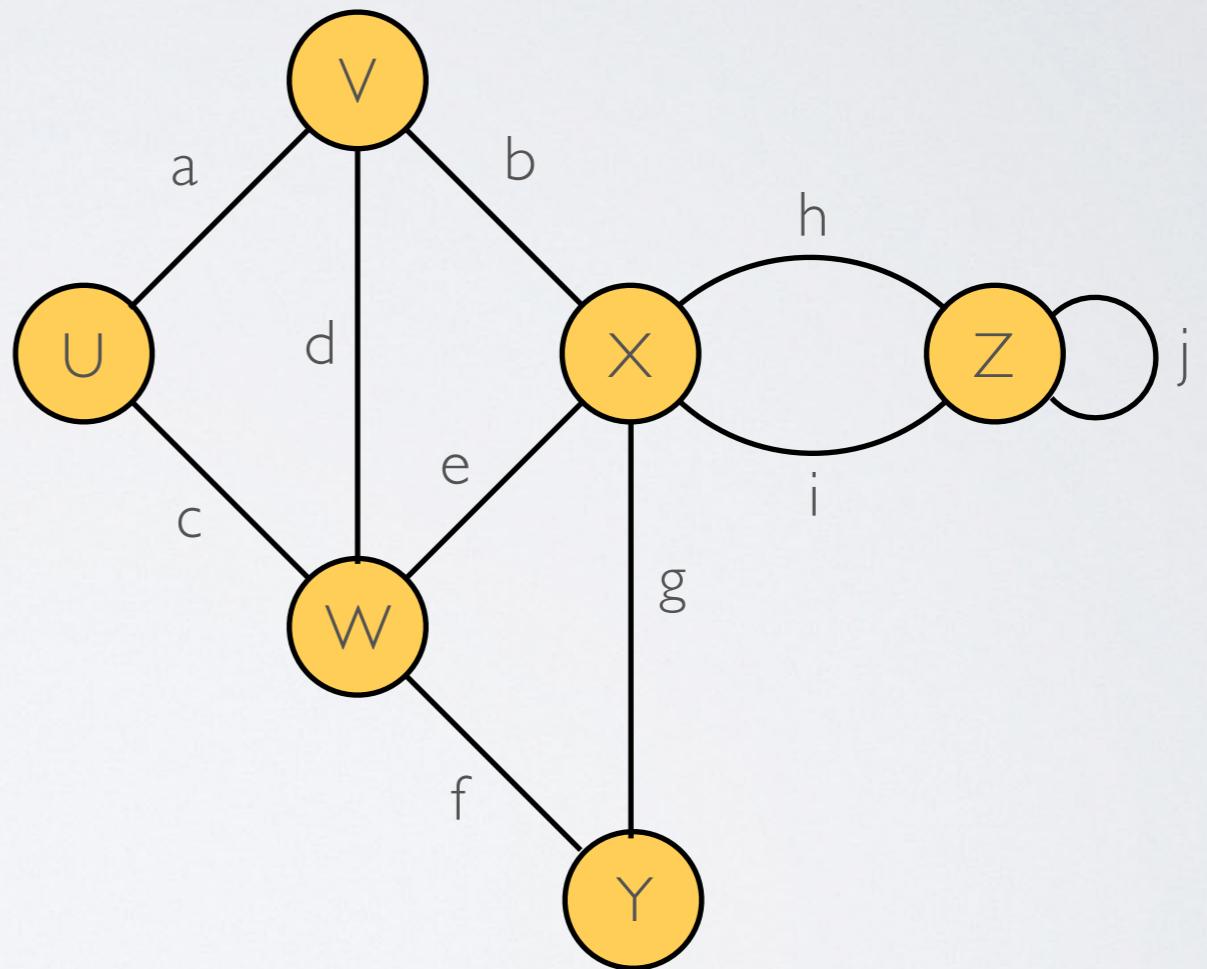
- ▶ A graph is defined by
 - ▶ a set of vertices V
 - ▶ a set of edges E
 - ▶ E is typically a subset of $V \times V$
- ▶ Vertices and edges can both store data
- ▶ Example
 - ▶ vertices can represent scientists and can store their biographical data
 - ▶ edges can represent collaborations

A Graph



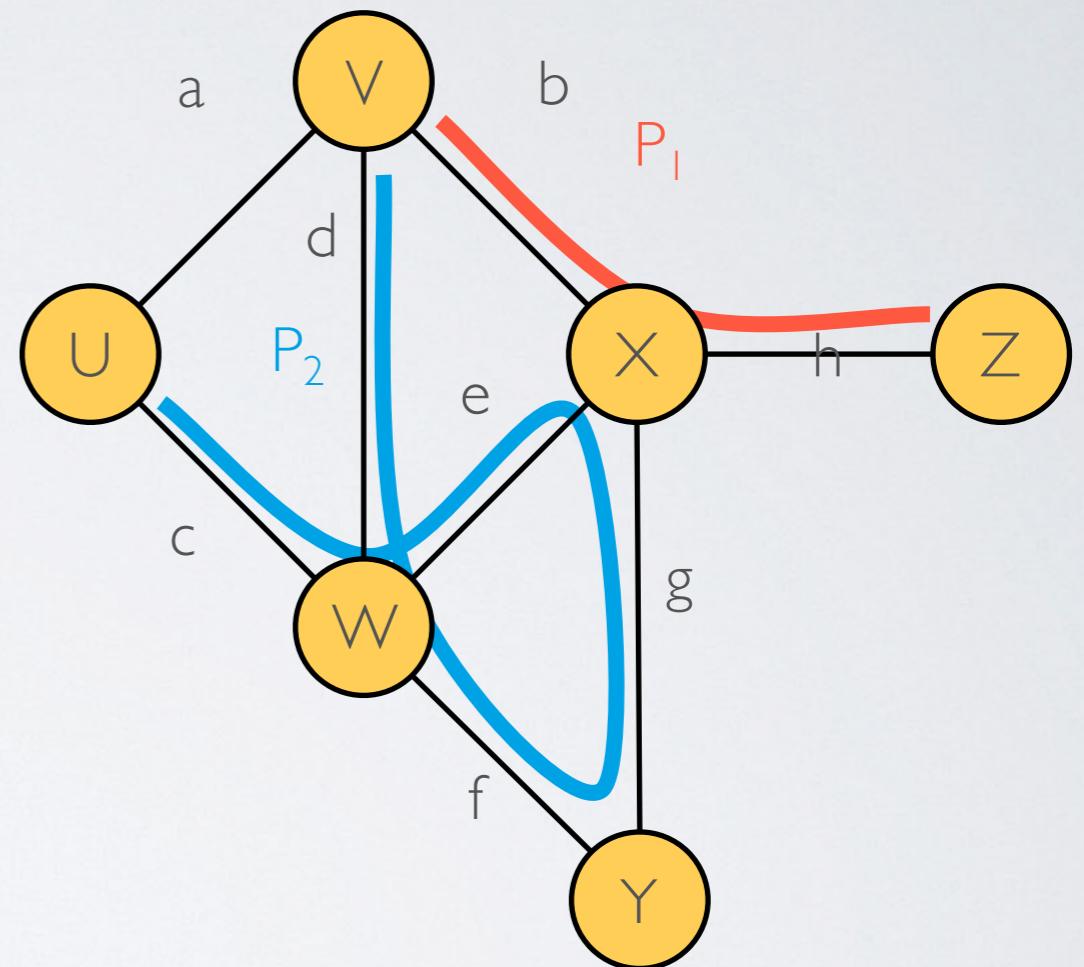
Terminology

- ▶ Endpoints or end vertices of an edge
 - ▶ **U** and **V** are endpoints of edge **a**
- ▶ Incident edges of a vertex
 - ▶ **a, b, d** are incident to **v**
- ▶ Adjacent vertices
 - ▶ **U** and **V** are adjacent
- ▶ Degree of a vertex
 - ▶ **X** has degree of 5
- ▶ Parallel (multiple) edges
 - ▶ **h, i** are parallel edges
- ▶ Self-loops
 - ▶ **j** is a self-looped edge



Terminology

- ▶ A path is a sequence of alternating vertices and edges
 - ▶ begins and ends with a vertex
 - ▶ each edge is preceded and followed by its endpoints
- ▶ Simple path
 - ▶ path such that all its vertices and edges are visited at most once
- ▶ Examples
 - ▶ $P_1 = V \rightarrow_b X \rightarrow_h Z$ is a simple path
 - ▶ $P_2 = U \rightarrow_c W \rightarrow_e X \rightarrow_g Y \rightarrow_f W \rightarrow_d V$ is not a simple path, but is still a path



Applications

- ▶ Flight networks
- ▶ Road networks & GPS
- ▶ The Web
 - ▶ pages are vertices
 - ▶ links are edges
- ▶ The Internet
 - ▶ routers and devices are vertices
 - ▶ network connections are edges
- ▶ Facebook
 - ▶ profiles are vertices
 - ▶ friendships are edges

Graph Properties

- ▶ A graph $G'=(V', E')$ is a **subgraph** of $G=(V, E)$
 - ▶ if $V' \subseteq V$ and $E' \subseteq E$
- ▶ A graph is **connected** if
 - ▶ there exists path from each vertex to every other vertex
- ▶ A path is a **cycle** if
 - ▶ it starts and ends at the same vertex
- ▶ A graph is **acyclic**
 - ▶ if it has no cycles

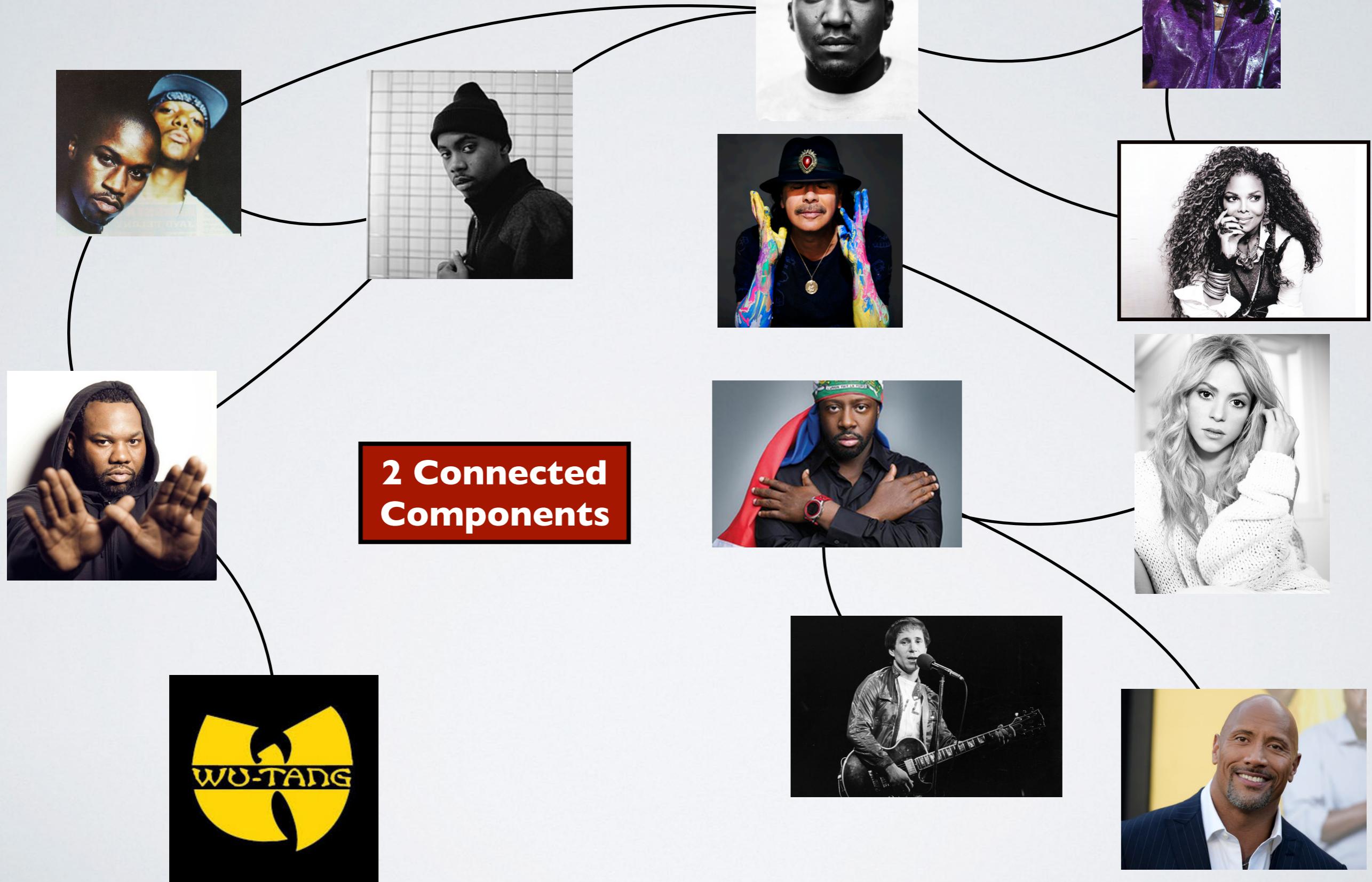
A Subgraph



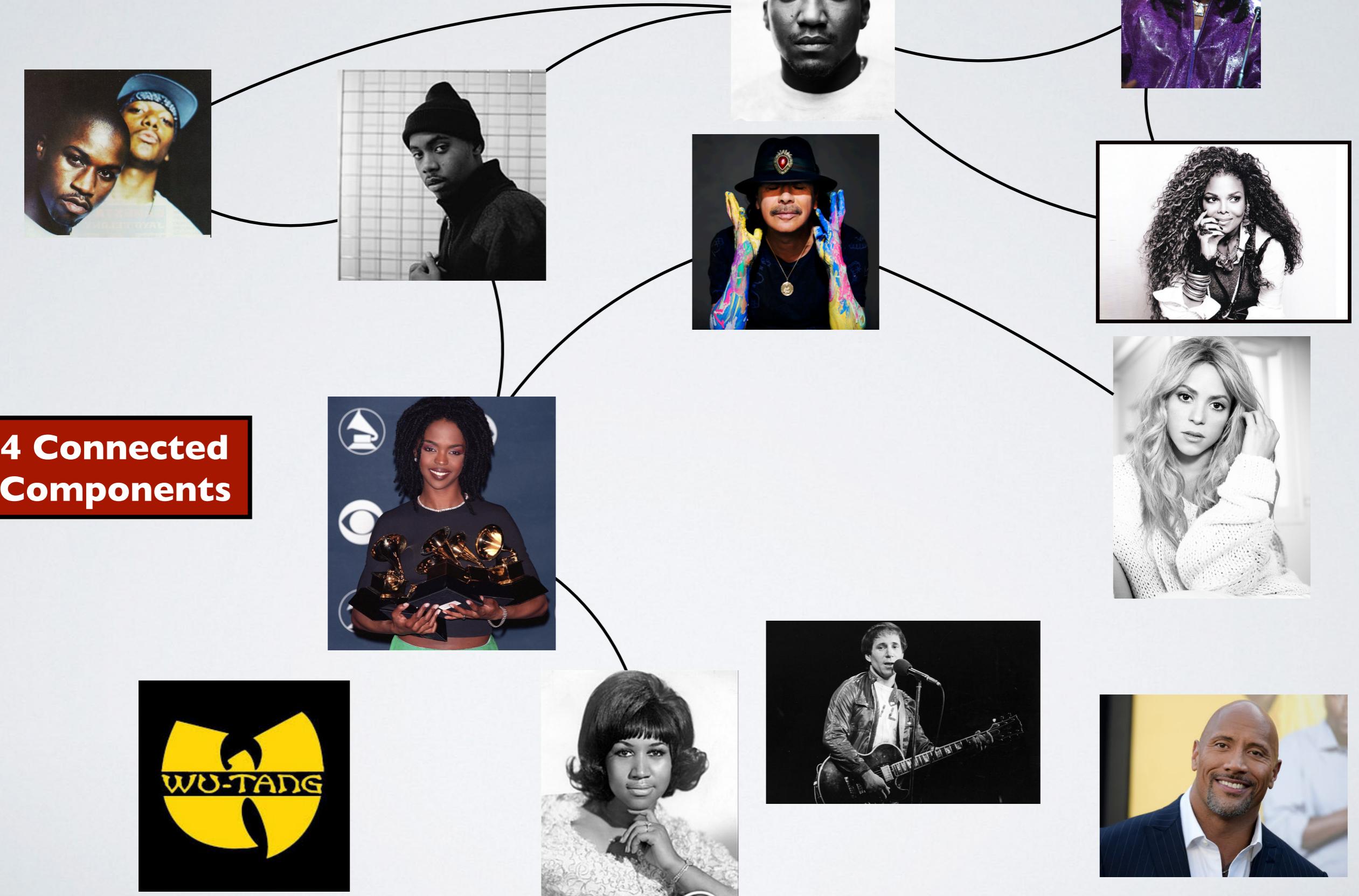
A Connected Graph



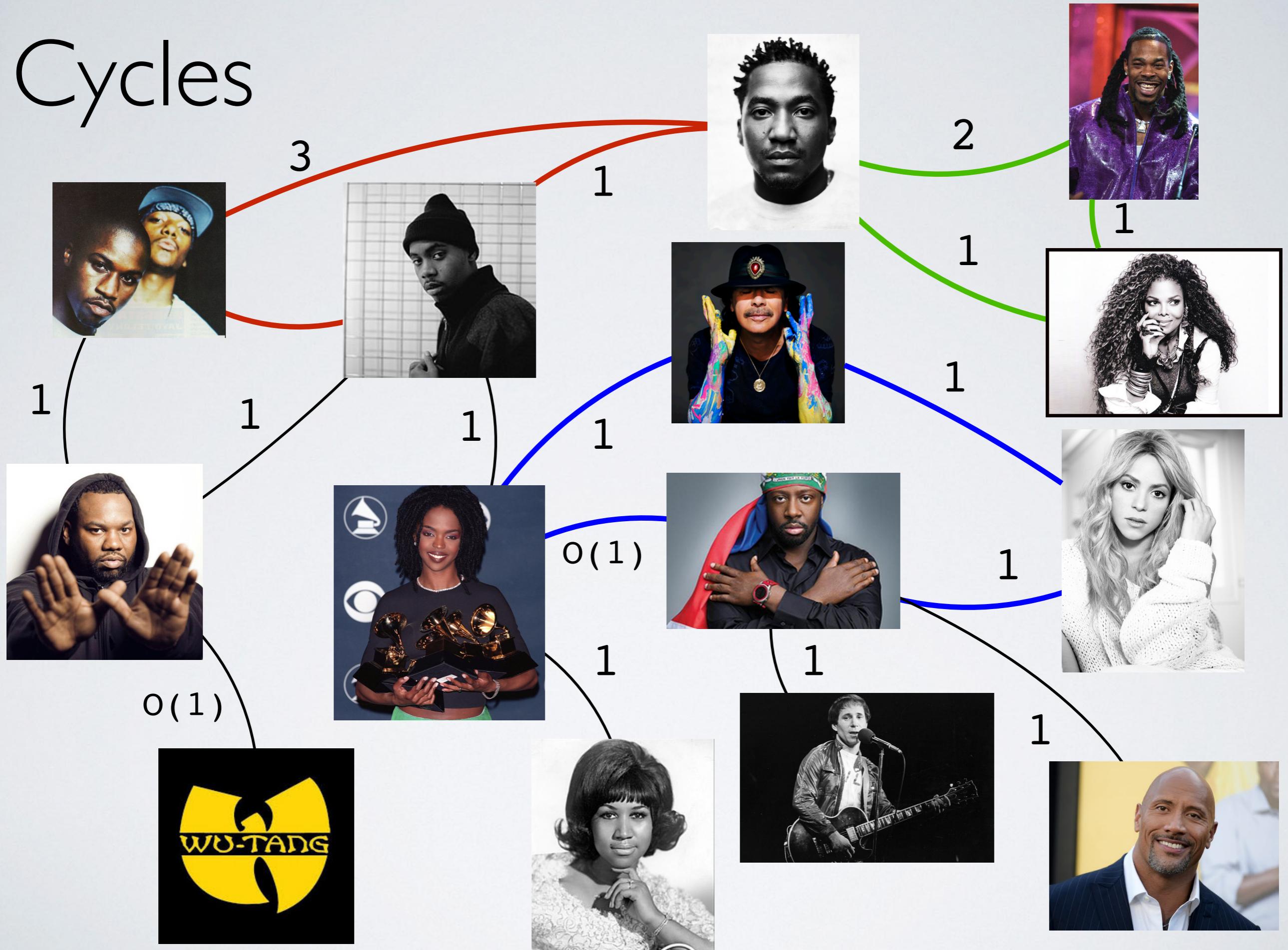
An Unconnected Graph



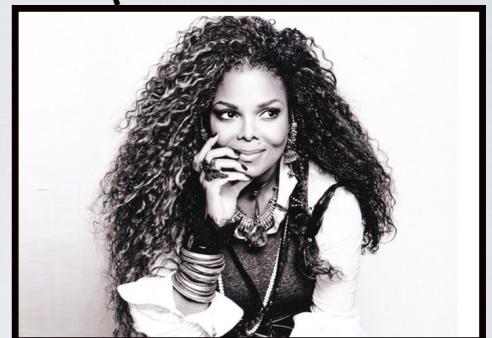
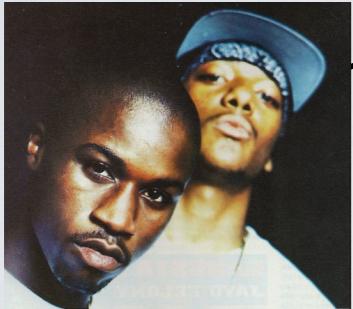
An Unconnected Graph



Cycles



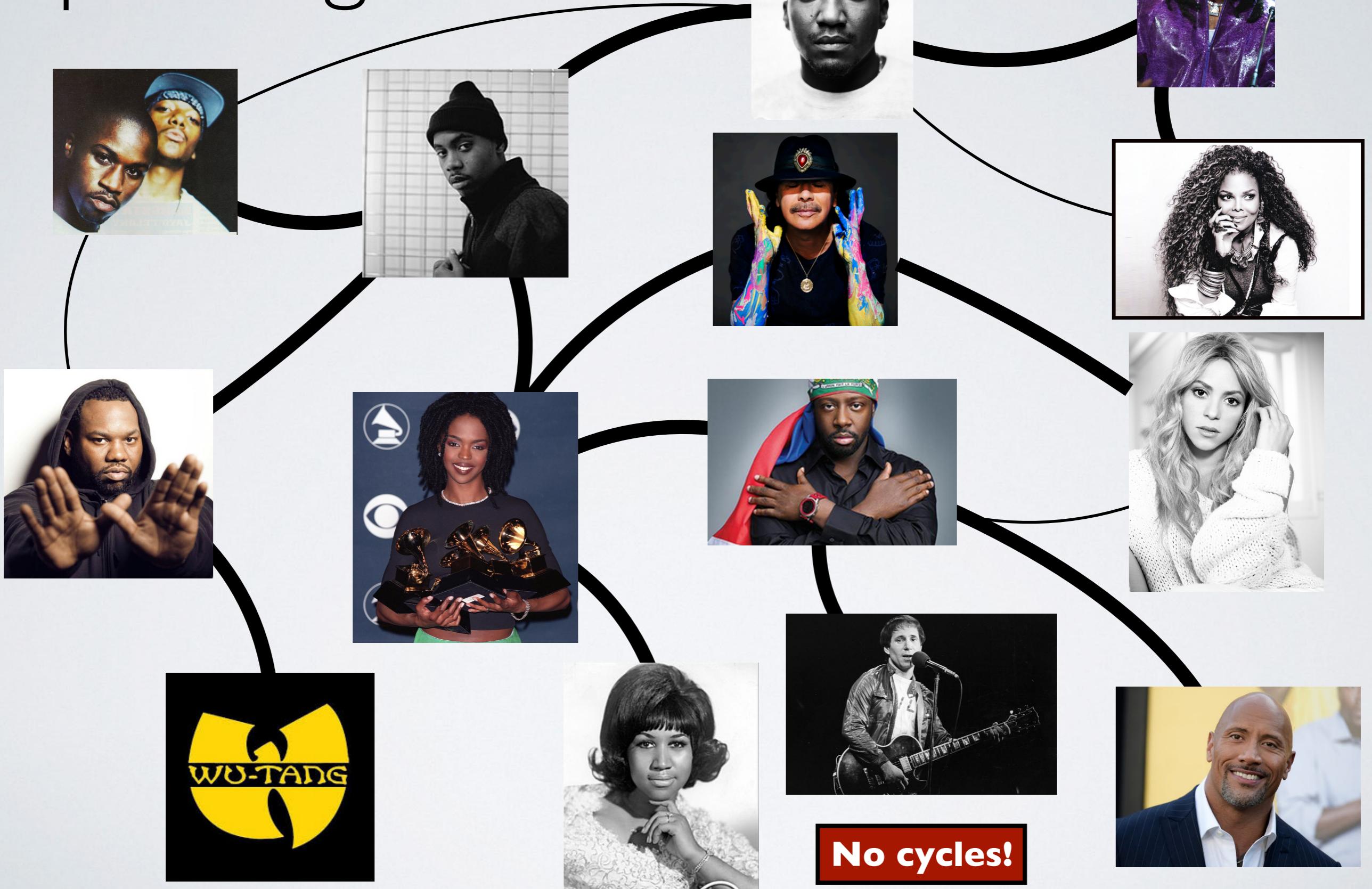
An Acyclic Graph



Graph Properties

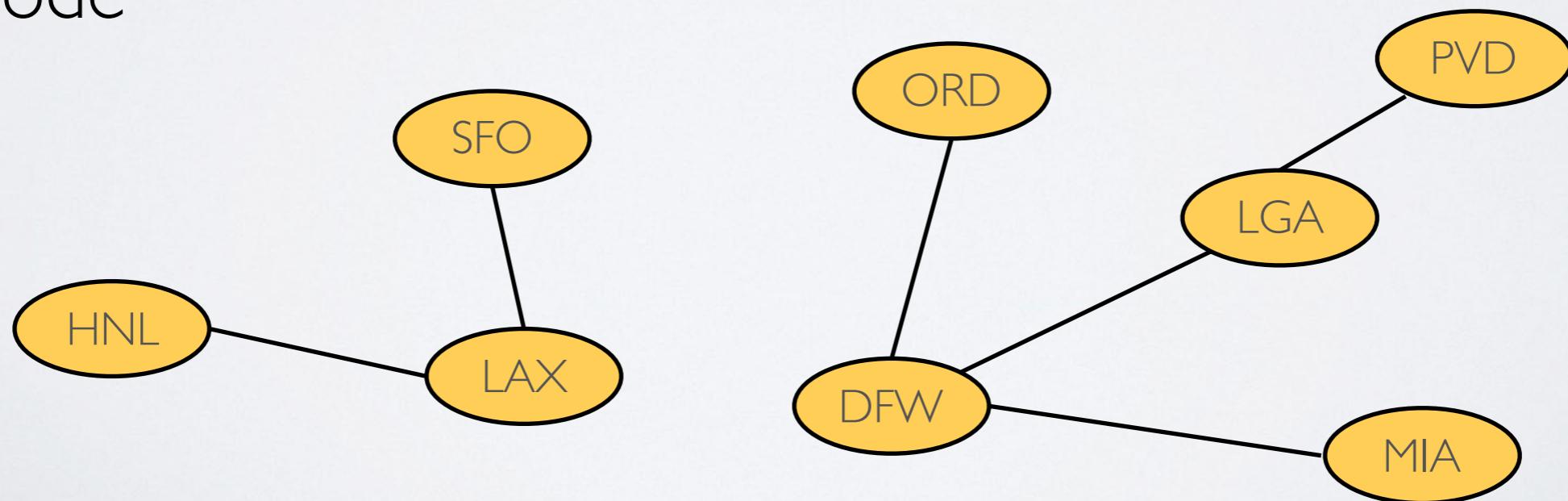
- ▶ A **spanning tree** of G is a subgraph with
 - ▶ all of G 's vertices in a single tree
 - ▶ and enough edges to connect each vertex w/o cycles

Spanning Tree

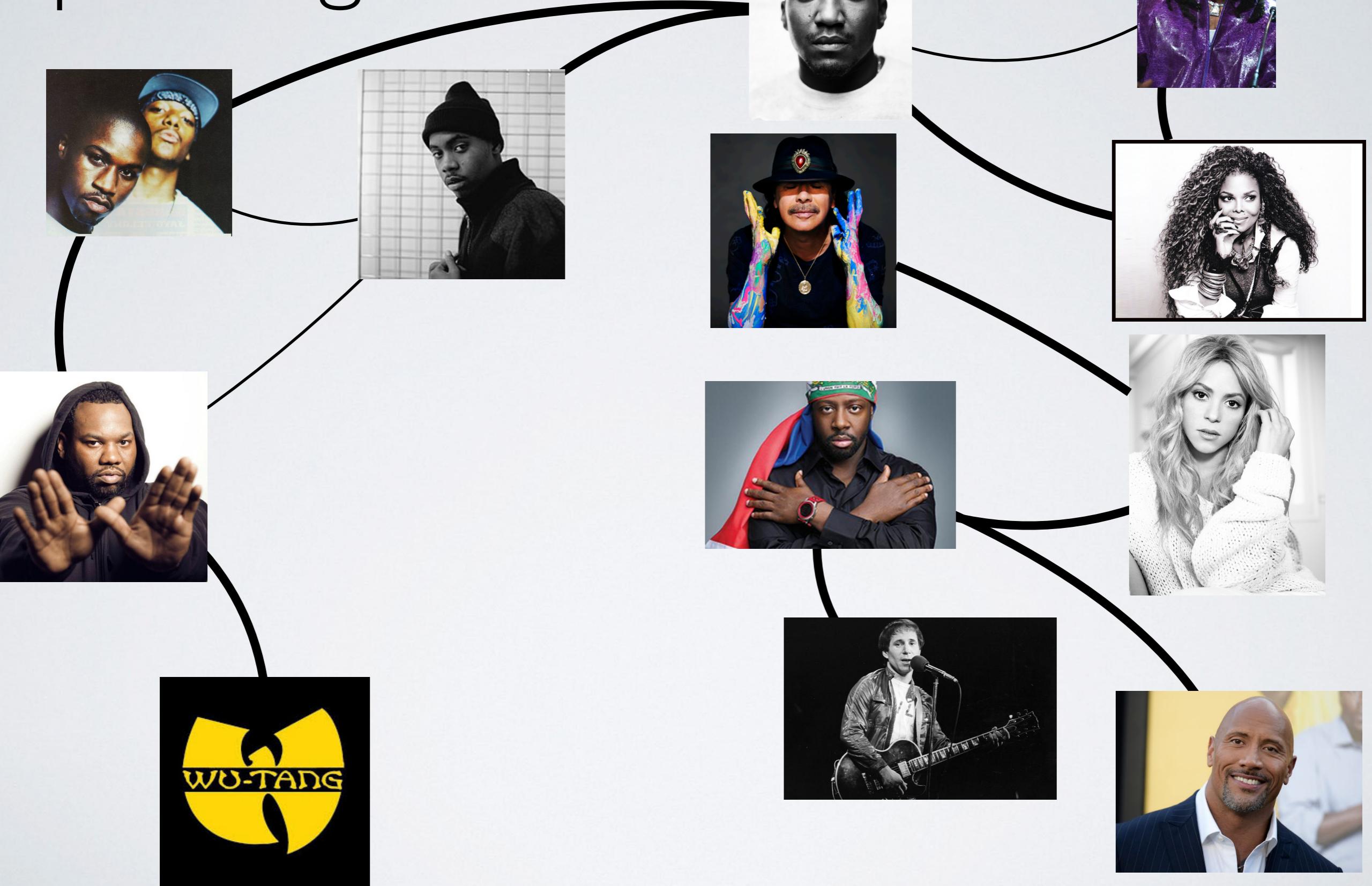


Graph Properties

- ▶ A **spanning forest** is
 - ▶ a subgraph that consists of a spanning tree in each connected component of graph
- ▶ Spanning forests never contain cycles
 - ▶ this might not be the “best” or shortest path to each node



Spanning Forest



Graph Properties

- ▶ G is a tree if and only if it satisfies any of these conditions
 - ▶ G has $|V| - 1$ edges and no cycles
 - ▶ G has $|V| - 1$ edges and is connected
 - ▶ G is connected, but removing any edge disconnects it
 - ▶ G is acyclic, but adding any edges creates a cycle
 - ▶ Exactly one simple path connects each pair of vertices in G

Graph Proof I

- ▶ Prove that
 - ▶ the sum of the degrees of all vertices of some graph G ...
 - ▶ ... is twice the number of edges of G
- ▶ Let $V = \{v_1, v_2, \dots, v_p\}$, where p is number of vertices
- ▶ The total sum of degrees D is such that
 - ▶ $D = \deg(v_1) + \deg(v_2) + \dots + \deg(v_p)$
- ▶ But each edge is counted twice in D
 - ▶ one for each of the two vertices incident to the edge
- ▶ So $D = 2 |E|$, where $|E|$ is the number of edges.

Graph Proof 2

- ▶ Prove using induction that if G is connected then
 - ▶ $|E| \geq |v|-1$, for all $|v| \geq 1$
- ▶ Base case $|v|=1$
 - ▶ If graph has one vertex then it will have 0 edges
 - ▶ so since $|E|=0$ and $|v|-1=1-1=0$, we have $|E| \geq |v|-1$
- ▶ Inductive hypothesis
 - ▶ If graph has $|v|=k$ vertices then $|E| \geq k-1$
- ▶ Inductive step
 - ▶ Let G be any connected graph with $|v|=k+1$ vertices
 - ▶ We must show that $|E| \geq k$

Graph Proof 2

- ▶ Inductive step
 - ▶ Let \mathbf{G} be any connected graph with $|V| = k+1$ vertices
 - ▶ We must show that $|E| \geq k$
- ▶ Let \mathbf{u} be the vertex of minimum degree in \mathbf{G}
 - ▶ $\deg(u) \geq 1$ since \mathbf{G} is connected
- ▶ If $\deg(u) = 1$
 - ▶ Let \mathbf{G}' be \mathbf{G} without \mathbf{u} and its **1** incident edge
 - ▶ \mathbf{G}' has **k** vertices because we removed **1** vertex from \mathbf{G}
 - ▶ \mathbf{G}' is still connected because we only removed a leaf
 - ▶ So by inductive hypothesis, \mathbf{G}' has at least **k-1** edges
 - ▶ which means that \mathbf{G} has at least **k** edges

Graph Proof 2

- ▶ If $\deg(u) \geq 2$
 - ▶ Every vertex has at least two incident edges
 - ▶ So the total degree D of the graph is $D \geq 2(k+1)$
 - ▶ But we know from the last proof that $D=2|E|$
 - ▶ so $2|E| \geq 2(k+1) \Rightarrow |E| \geq k+1 \Rightarrow |E| \geq k$
- ▶ We showed it is true for $|V|=1$ (base case)...
 - ▶ ...and for $|V|=k+1$ assuming it is true for $|V|=k$...
 - ▶ ...so it is true for all $|V| \geq 1$

Edge Types

- ▶ Undirected edge

- ▶ unordered pair of vertices (LH,CS)

- ▶ for example a collaboration



1



- ▶ Directed edge

- ▶ ordered pair of vertices (LH,CS)

- ▶ first vertex LH is the origin



1



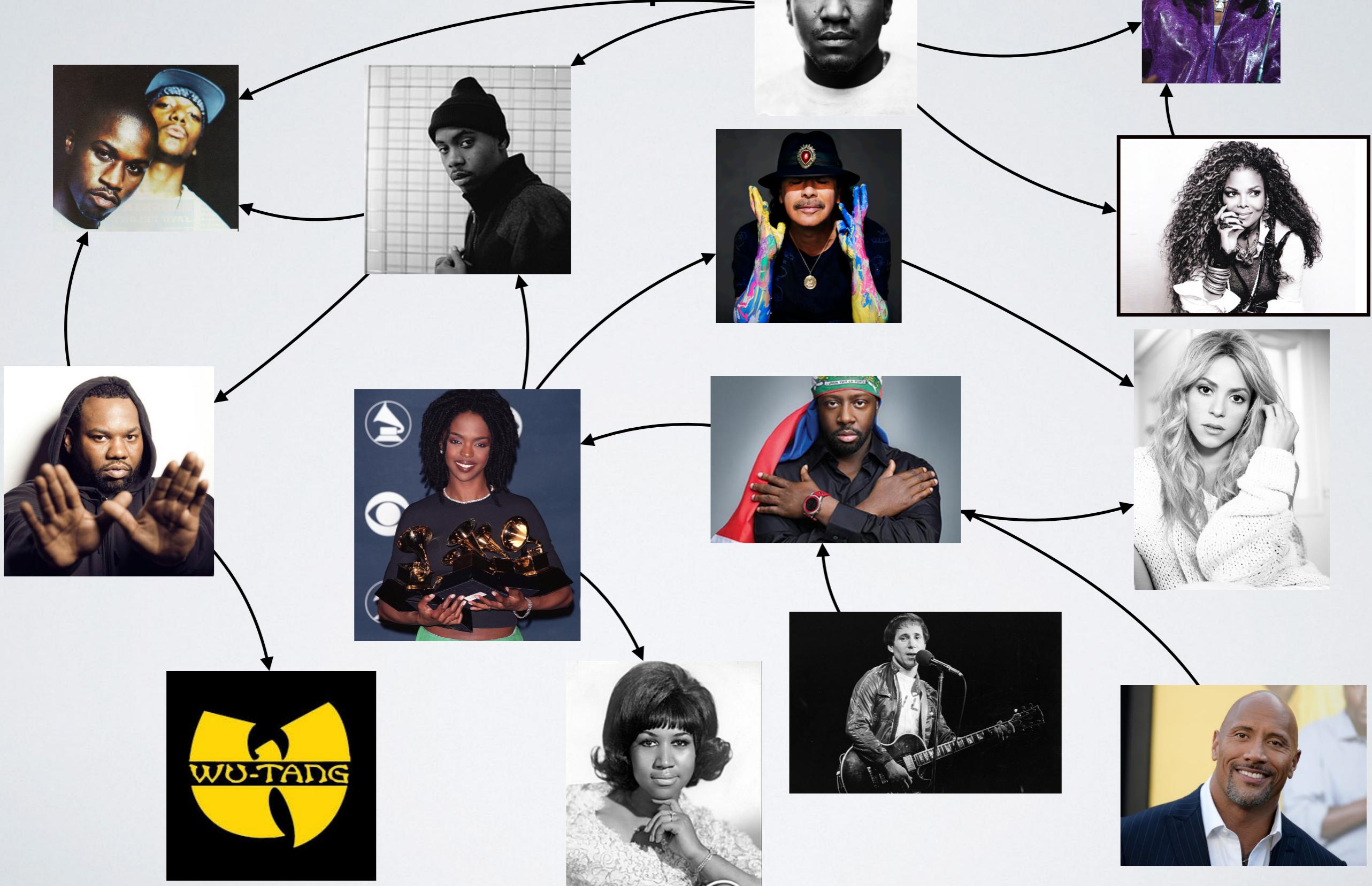
- ▶ second vertex CS is the destination

- ▶ LH collaborated with CS on CS's album

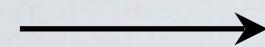
An Undirected Graph



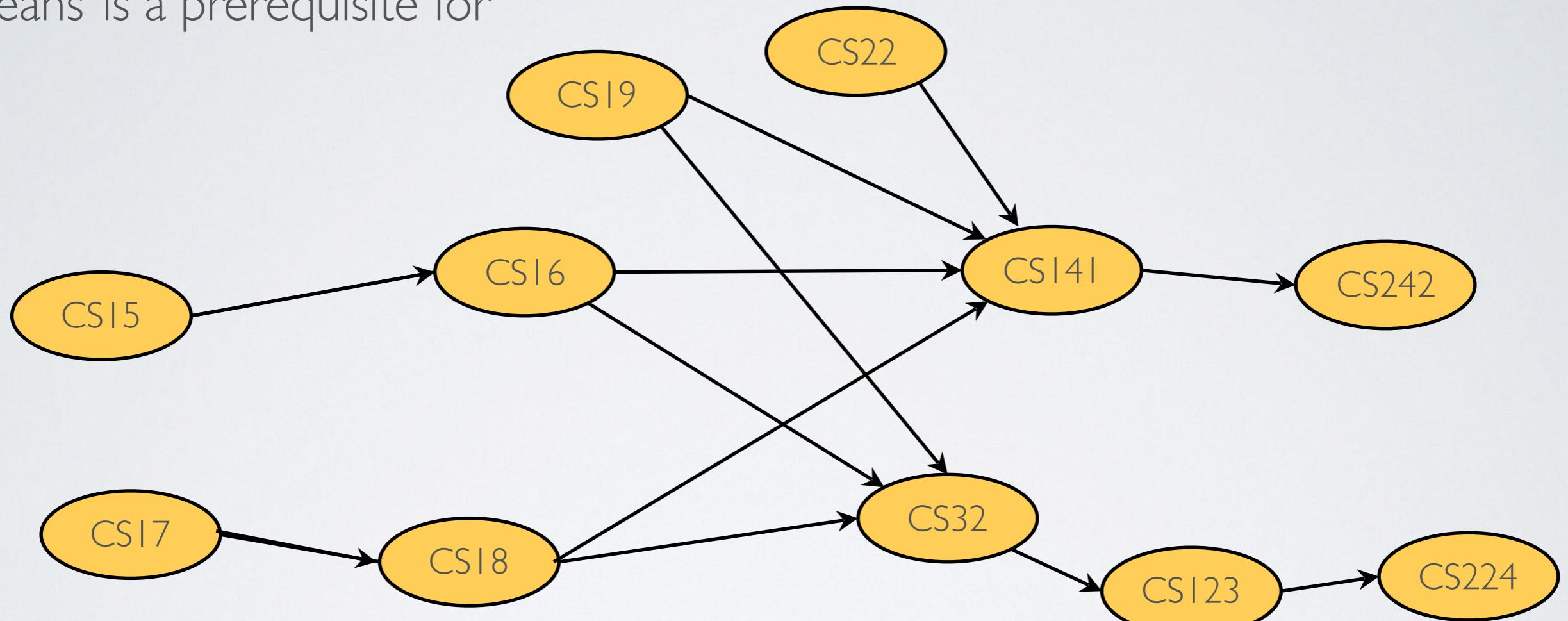
A Directed Graph



Directed Acyclic Graph (DAG)

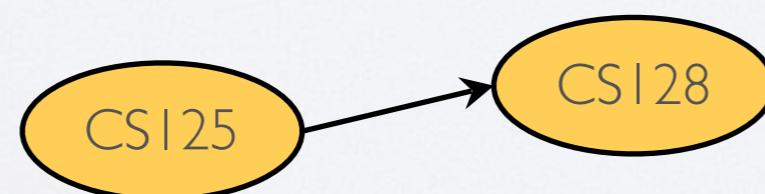


means 'is a prerequisite for'



We'll talk much
more about DAGs
in future lectures...

Acyclic = without cycles

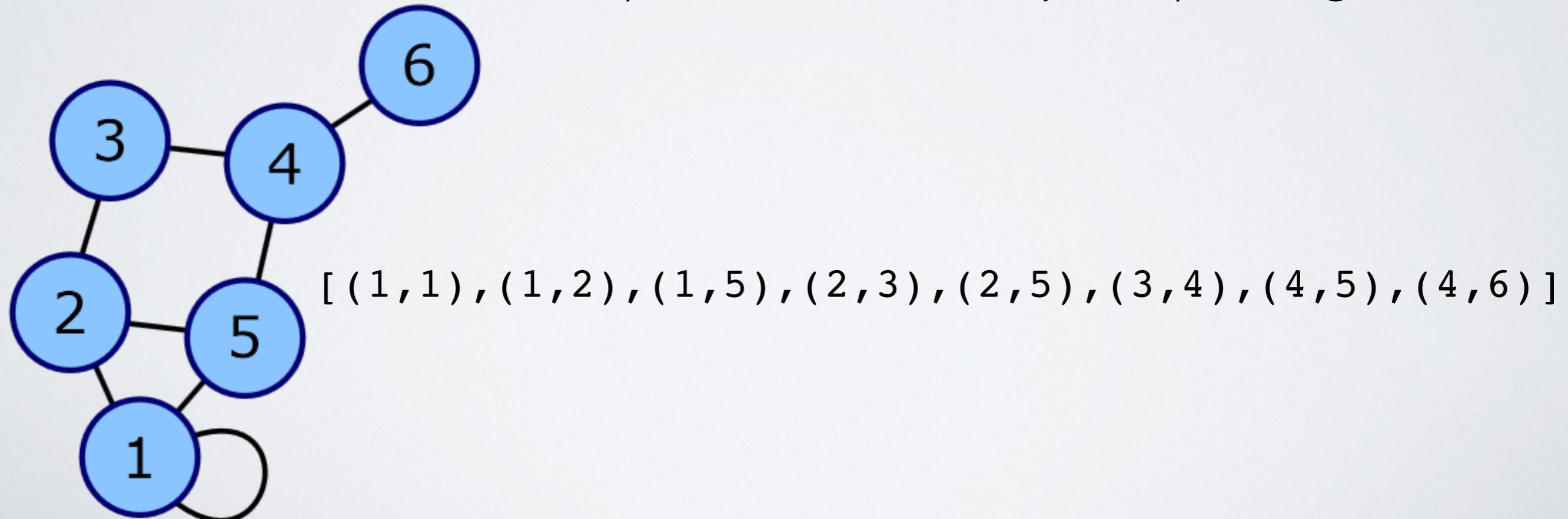


Graph Representations

- ▶ Vertices usually stored in a List or Set
- ▶ 3 common ways of representing which vertices are adjacent
 - ▶ Edge list (or set)
 - ▶ Adjacency lists (or sets)
 - ▶ Adjacency matrix

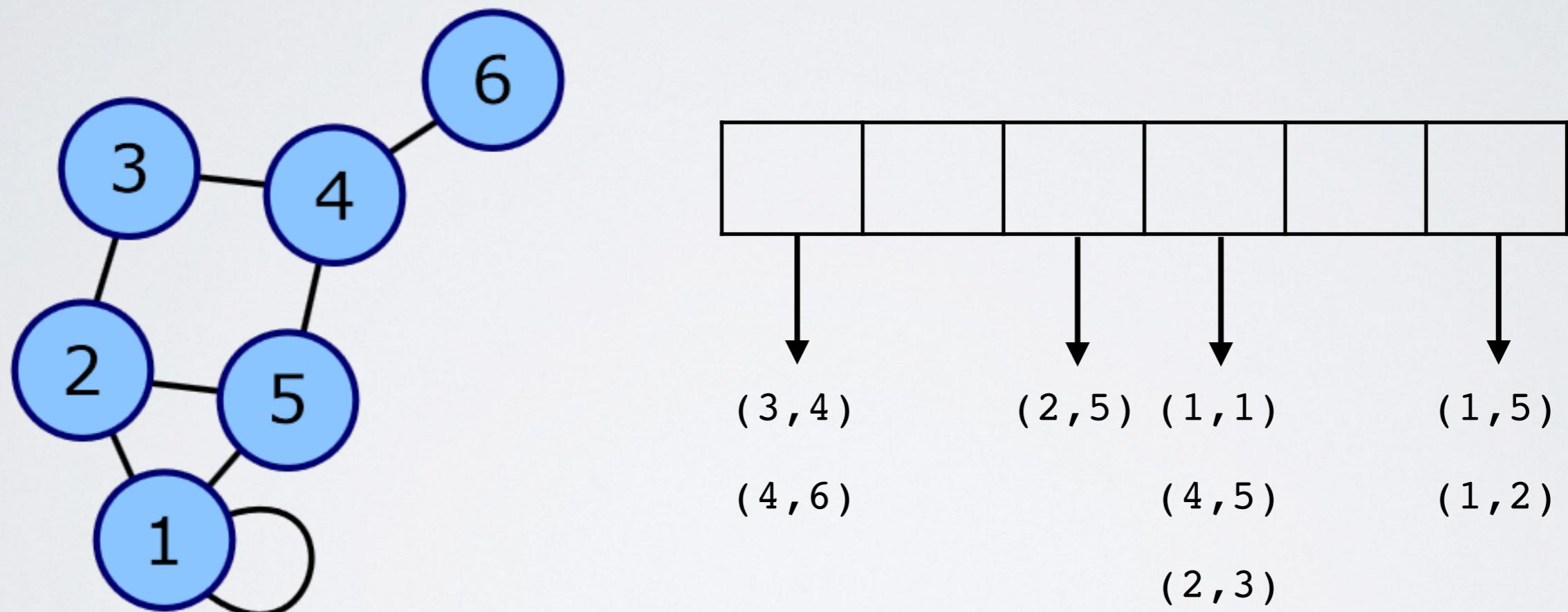
Edge List

- ▶ Represents adjacencies as a list of pairs
- ▶ Each element of list is a single edge **(a , b)**
- ▶ Since the order of list doesn't matter
 - ▶ can use hashset to improve runtime of adjacency testing



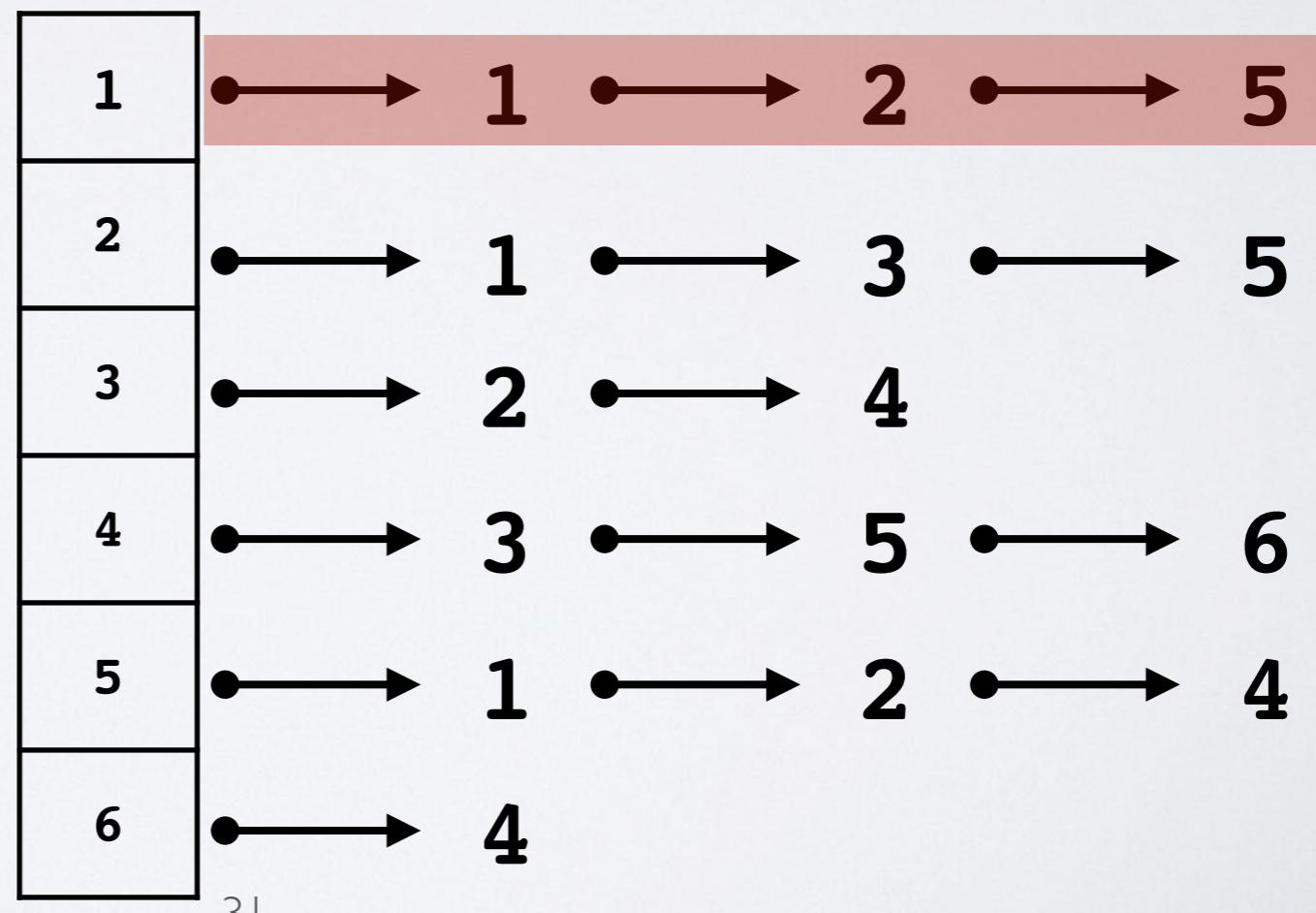
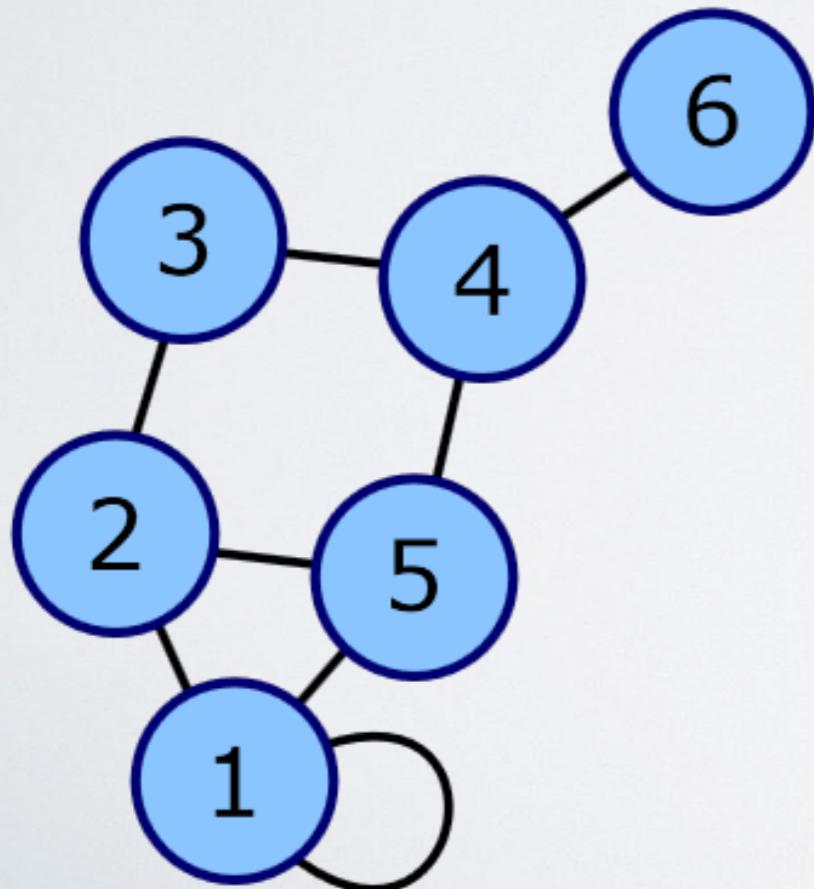
Edge Set

- ▶ Store all the edges in a HashSet



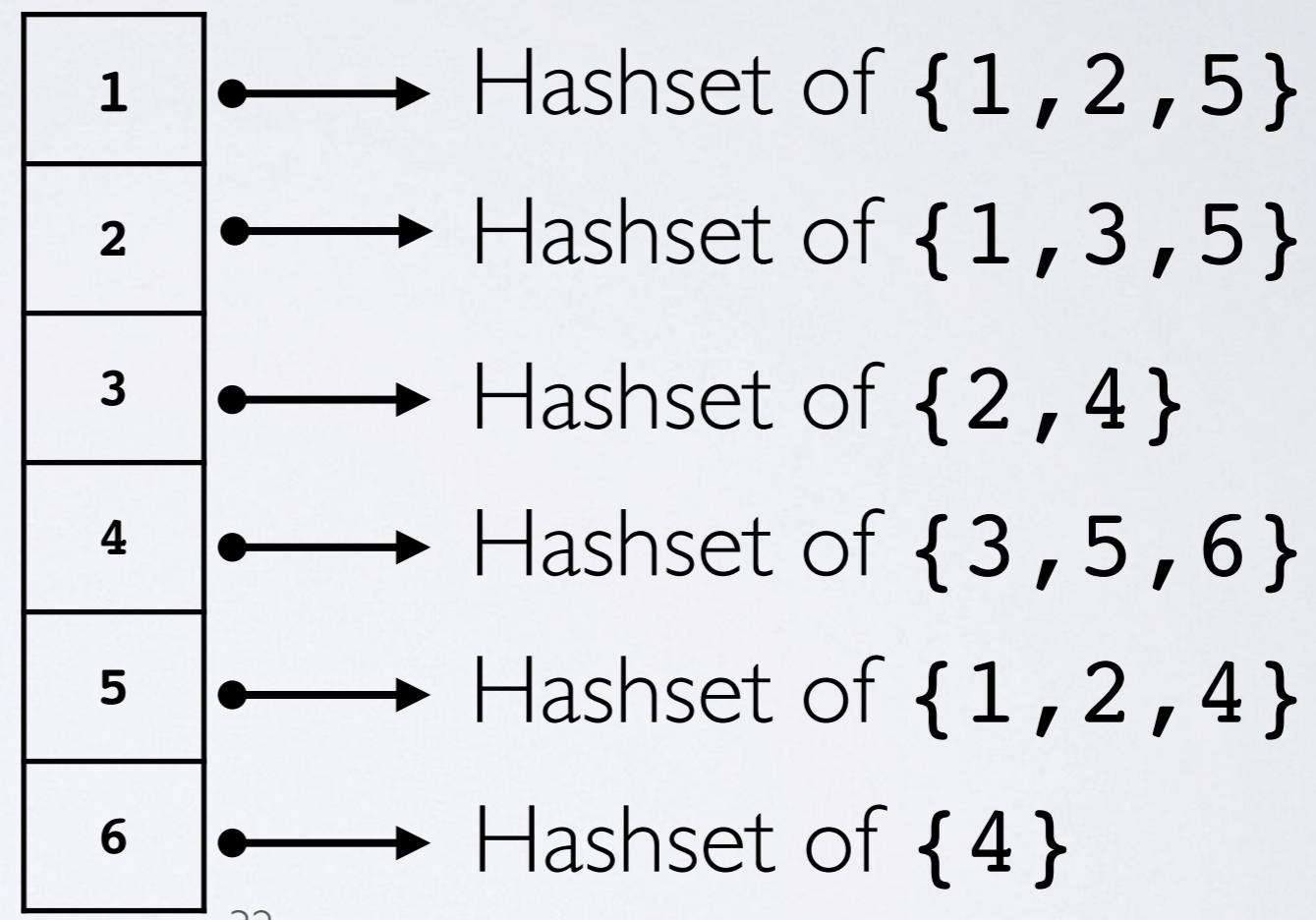
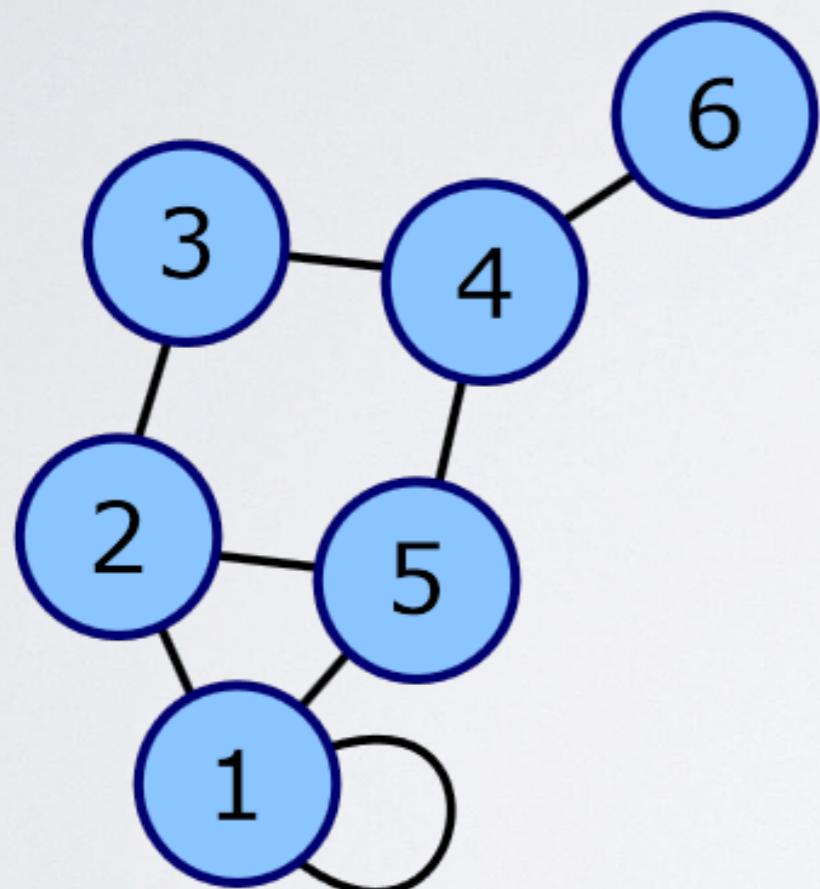
Adjacency Lists

- ▶ Each vertex has an associated list with its neighbors
- ▶ Since the order of elements in lists doesn't matter
 - ▶ lists can be hashsets instead



Adjacency Set

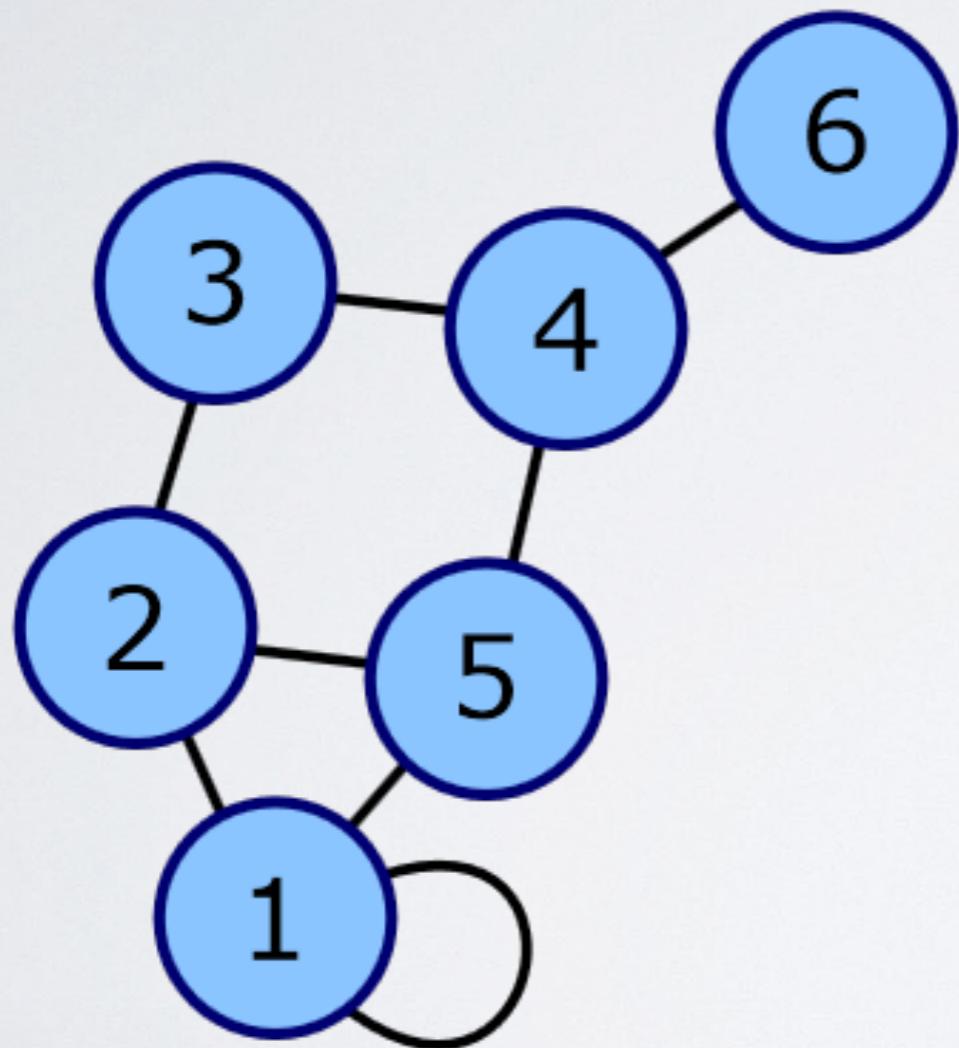
- ▶ Each vertex associated Hashset of its neighbors



Adjacency Matrix

- ▶ Matrix with n rows and n columns
 - ▶ n is number of vertices
 - ▶ If u is adjacent to v then $M[u, v] = T$
 - ▶ If u is not adjacent to v then $M[u, v] = F$
- ▶ If graph is undirected then $M[u, v] = M[v, u]$

Adjacency Matrix



	1	2	3	4	5	6
1	T	T	F	F	T	F
2	T	F	T	F	T	F
3	F	T	F	T	F	F
4	F	F	T	F	T	T
5	T	T	F	T	F	F
6	F	F	F	T	F	F

Adjacency Matrix

- ▶ Initialize matrix to predicted size of graph
 - ▶ we can always expand later
- ▶ When vertex is added to graph
 - ▶ reserve a row and column of matrix for that vertex
- ▶ When vertex is removed
 - ▶ set its entire row and column to false
- ▶ Since we can't remove rows/columns from arrays
 - ▶ keep separate collection of vertices that are actually present in graph

Big-O Performance

	Edge Set	Adjacency List	Adjacency Matrix
Overall Space ¹	$O(V + E)$	$O(V + E)$	$O(V ^2)$
<code>vertices()</code> ¹	$O(1)^*$	$O(1)^*$	$O(1)^*$
<code>edges()</code>	$O(1)^*$	$O(E)$	$O(V ^2)$
<code>incidentEdges(v)</code>	$O(E)$	$O(1)^*$	$O(V)$
<code>areAdjacent(v₁, v₂)</code>	$O(1)$	$O(1)$	$O(1)$
<code>insertVertex(v)</code>	$O(1)$	$O(1)$	$O(V)$
<code>insertEdge(v₁, v₂)</code>	$O(1)$	$O(1)$	$O(1)$
<code>removeVertex(v)</code>	$O(E)$	$O(V)$	$O(V)$
<code>removeEdge(v₁, v₂)</code>	$O(1)$	$O(1)$	$O(1)$

¹ In all approaches, we maintain an additional list or set of vertices

* in place
(return pointer)

Big-O Performance (Edge Set)

Operation	Runtime	Explanation
<code>vertices()</code>	$O(1)$	Return set of vertices
<code>edges()</code>	$O(1)$	Return set of edges
<code>incidentEdges(v)</code>	$O(E)$	Iterate through each edge and check if it contains vertex v
<code>areAdjacent(v₁,v₂)</code>	$O(1)$	Check if (v_1, v_2) exists in the set
<code>insertVertex(v)</code>	$O(1)$	Add vertex v to the vertex list
<code>insertEdge(v₁,v₂)</code>	$O(1)$	Add element (v_1, v_2) to the set
<code>removeVertex(v)</code>	$O(E)$	Iterate through each edge and remove it if it has vertex v
<code>removeEdge(v₁,v₂)</code>	$O(1)$	Remove edge (v_1, v_2)

Big-O Performance (Adjacency Set)

Operation	Runtime	Explanation
<code>vertices()</code>	$O(1)$	Return the set of vertices
<code>edges()</code>	$O(E)$	Concatenate each vertex with its subsequent vertices
<code>incidentEdges(v)</code>	$O(1)$	Return v 's edge set
<code>areAdjacent(v₁,v₂)</code>	$O(1)$	Check if v_2 is in v_1 's set
<code>insertVertex(v)</code>	$O(1)$	Add vertex v to the vertex set
<code>insertEdge(v₁,v₂)</code>	$O(1)$	Add v_1 to v_2 's edge set and vice versa
<code>removeVertex(v)</code>	$O(V)$	Remove v from each of its adjacent vertices' sets and remove v 's set
<code>removeEdge(v₁,v₂)</code>	$O(1)$	Remove v_1 from v_2 's set and vice versa

Big-O Performance (Adjacency Matrix)

Operation	Runtime	Explanation
<code>vertices()</code>	$O(1)$	Return the set of vertices
<code>edges()</code>	$O(V ^2)$	Iterate through the entire matrix
<code>incidentEdges(v)</code>	$O(V)$	Iterate through v's row or column to check for trues Note: row/col are the same in an undirected graph.
<code>areAdjacent(v₁,v₂)</code>	$O(1)$	Check index (v ₁ ,v ₂) for a true
<code>insertVertex(v)</code>	$O(V) *$	Add vertex v to the matrix (* $O(1)$ amortized)
<code>insertEdge(v₁,v₂)</code>	$O(1)$	Set index (v ₁ ,v ₂) to true
<code>removeVertex(v)</code>	$O(V)$	Set v's row and column to false and remove v from the vertex list
<code>removeEdge(v₁,v₂)</code>	$O(1)$	Set index (v ₁ ,v ₂) to false

BFT and DFT

- ▶ Remember BFT and DFT on trees?
- ▶ We can also do them on graphs
 - ▶ a tree is just a special kind of graph
 - ▶ often used to find certain values in graphs

BFT/DFT on Graphs

1 min

Activity #2

BFT/DFT on Graphs

1 min

Activity #2

BFT/DFT on Graphs

Omin

Activity #2

Breadth First Traversal: Tree vs. Graph

```
function treeBFT(root):  
    //Input: Root node of tree  
    //Output: Nothing  
    Q = new Queue()  
    Q.enqueue(root)  
    while Q is not empty:  
        node = Q.dequeue()  
        doSomething(node)  
        enqueue node's children
```

doSomething() could
print, add to list, decorate
node etc...

```
function graphBFT(start):  
    //Input: start vertex  
    //Output: Nothing  
    Q = new Queue()  
    start.visited = true  
    Q.enqueue(start)  
    while Q is not empty:  
        node = Q.dequeue()  
        doSomething(node)  
        for neighbor in adj nodes:  
            if not neighbor.visited:  
                neighbor.visited = true  
                Q.enqueue(neighbor)
```

Mark nodes as visited otherwise you will loop
forever!

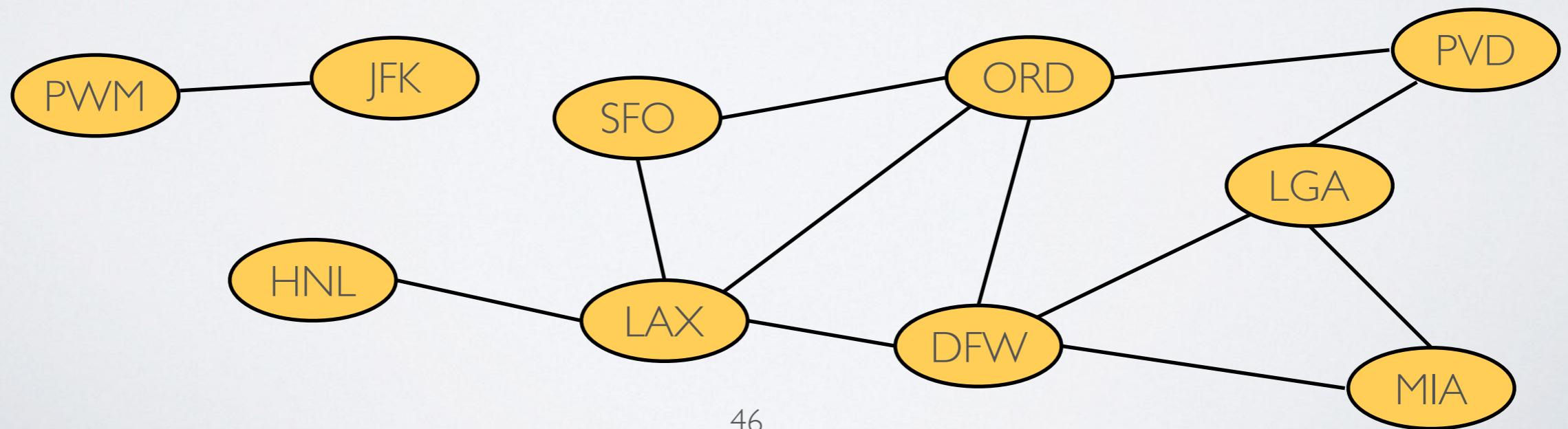
Depth First Traversal

- ▶ To do DFT on graph, replace queue with stack
- ▶ Can also be done recursively

```
function recursiveDFT(node):  
    // Input: start node  
    // Output: Nothing  
    node.visited = true  
    for neighbor in node's adjacent vertices:  
        if not neighbor.visited:  
            recursiveDFT(neighbor)
```

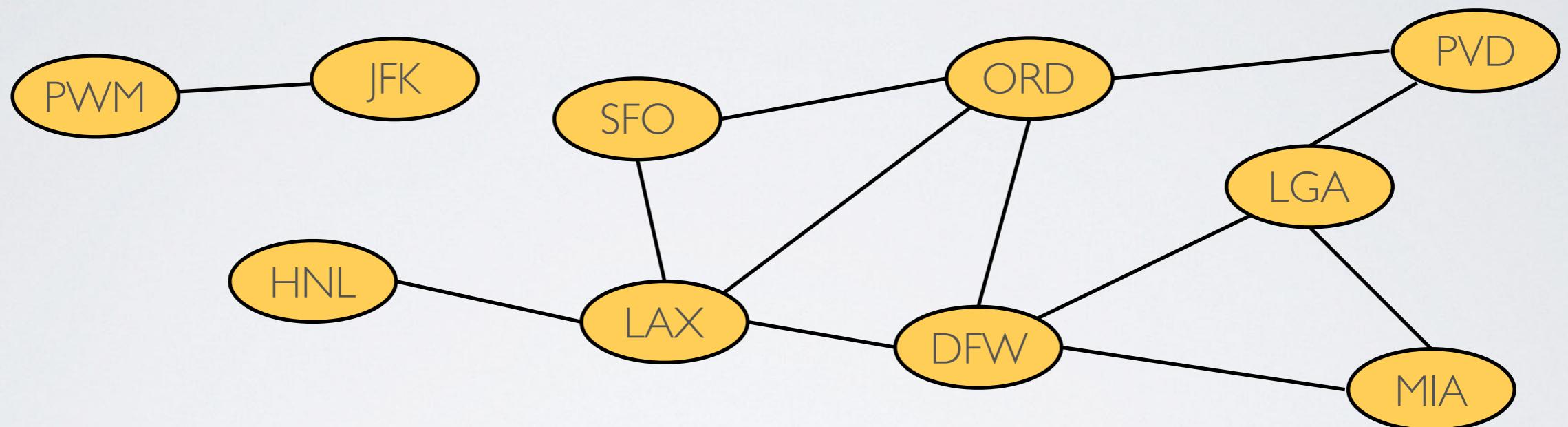
Applications: Flight Paths Exist

- ▶ Given undirected graph with airports & flights
 - ▶ is it possible to fly from one airport to another?
- ▶ Strategy
 - ▶ use breadth first search starting at first node
 - ▶ and determine if ending airport is ever visited



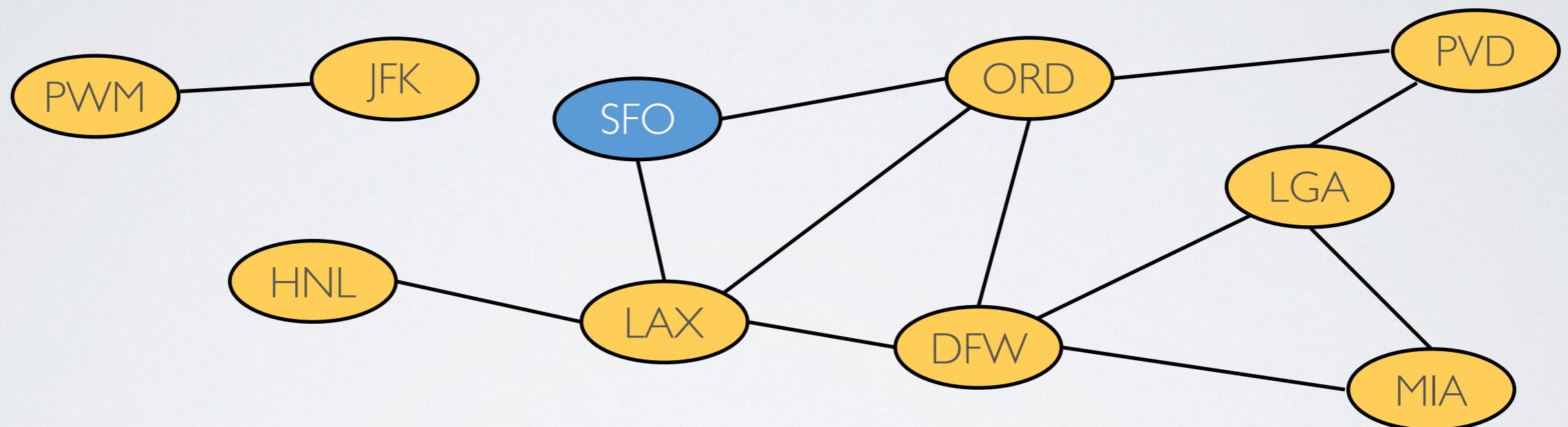
Applications: Flight Paths Exist

- ▶ Is there flight from SFO to PVD?



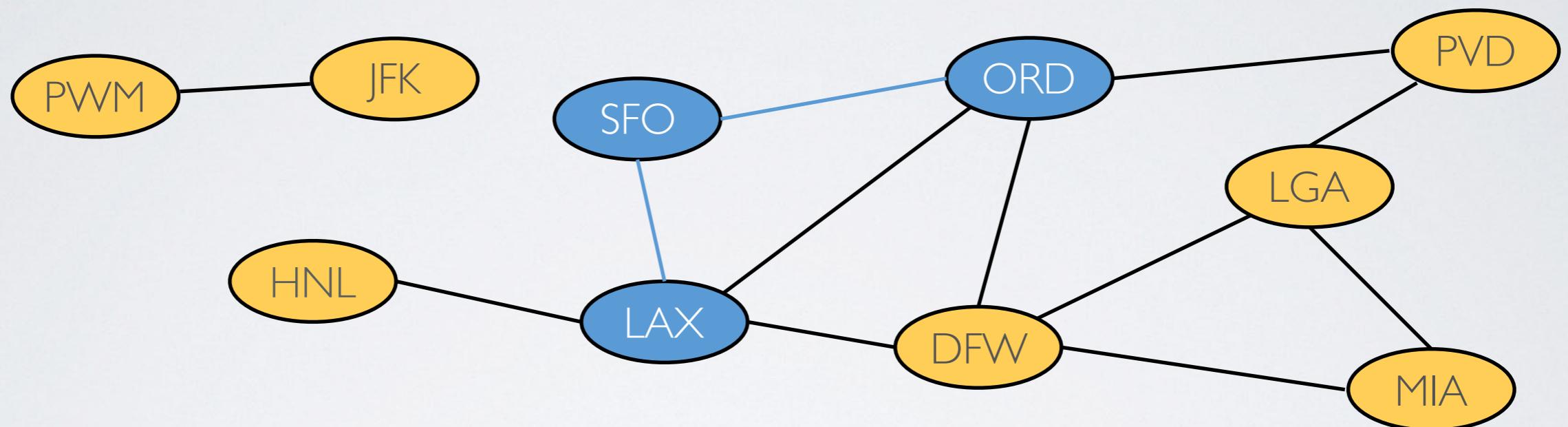
Applications: Flight Paths Exist

- ▶ Is there flight from SFO to PVD?



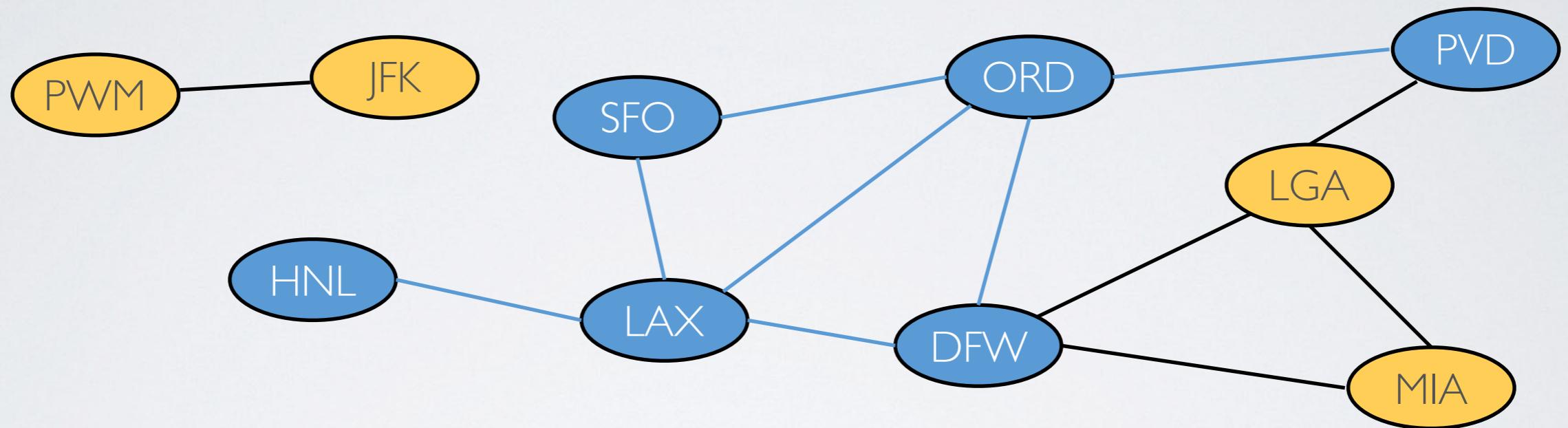
Applications: Flight Paths Exist

- ▶ Is there flight from SFO to PVD?



Applications: Flight Paths Exist

- ▶ Is there flight from SFO to PVD?



- ▶ Yes! but how do we do it with code?

Flight Paths Exist Pseudo-Code

```
function pathExists(from, to):
    //Input: from: vertex, to: vertex
    //Output: true if path exists, false otherwise
    Q = new Queue()
    from.visited = true
    Q.enqueue(from)
    while Q is not empty:
        airport = Q.dequeue()
        if airport == to:
            return true
        for neighbor in airport's adjacent nodes:
            if not neighbor.visited:
                neighbor.visited = true
                Q.enqueue(neighbor)
    return false
```

Applications: Flight Layovers

- ▶ Given undirected graph with airports & flights
 - ▶ decorate vertices w/ least number of stops from a given source
 - ▶ if no way to get to a an airport decorate w/ ∞
- ▶ Strategy
 - ▶ decorate each node w/ initial ‘stop value’ of ∞
 - ▶ use breadth first search to decorate each node...
 - ▶ ...w/ ‘stop value’ of one greater than its previous value

Flight Layovers Pseudo-Code

```
function numStops(G, source):
    //Input: G: graph, source: vertex
    //Output: Nothing
    //Purpose: decorate each vertex with the lowest number of
    //          layovers from source.

    for every node in G:
        node.stops = infinity

    Q = new Queue()
    source.stops = 0
    source.visited = true
    Q.enqueue(source)
    while Q is not empty:
        airport = Q.dequeue()
        for neighbor in airport's adjacent nodes:
            if not neighbor.visited:
                neighbor.visited = true
                neighbor.stops = airport.stops + 1
                Q.enqueue(neighbor)
```