CS 10: Problem solving via Object Oriented Programming

Winter 2017

Tim Pierson
260 (255) Sudikoff

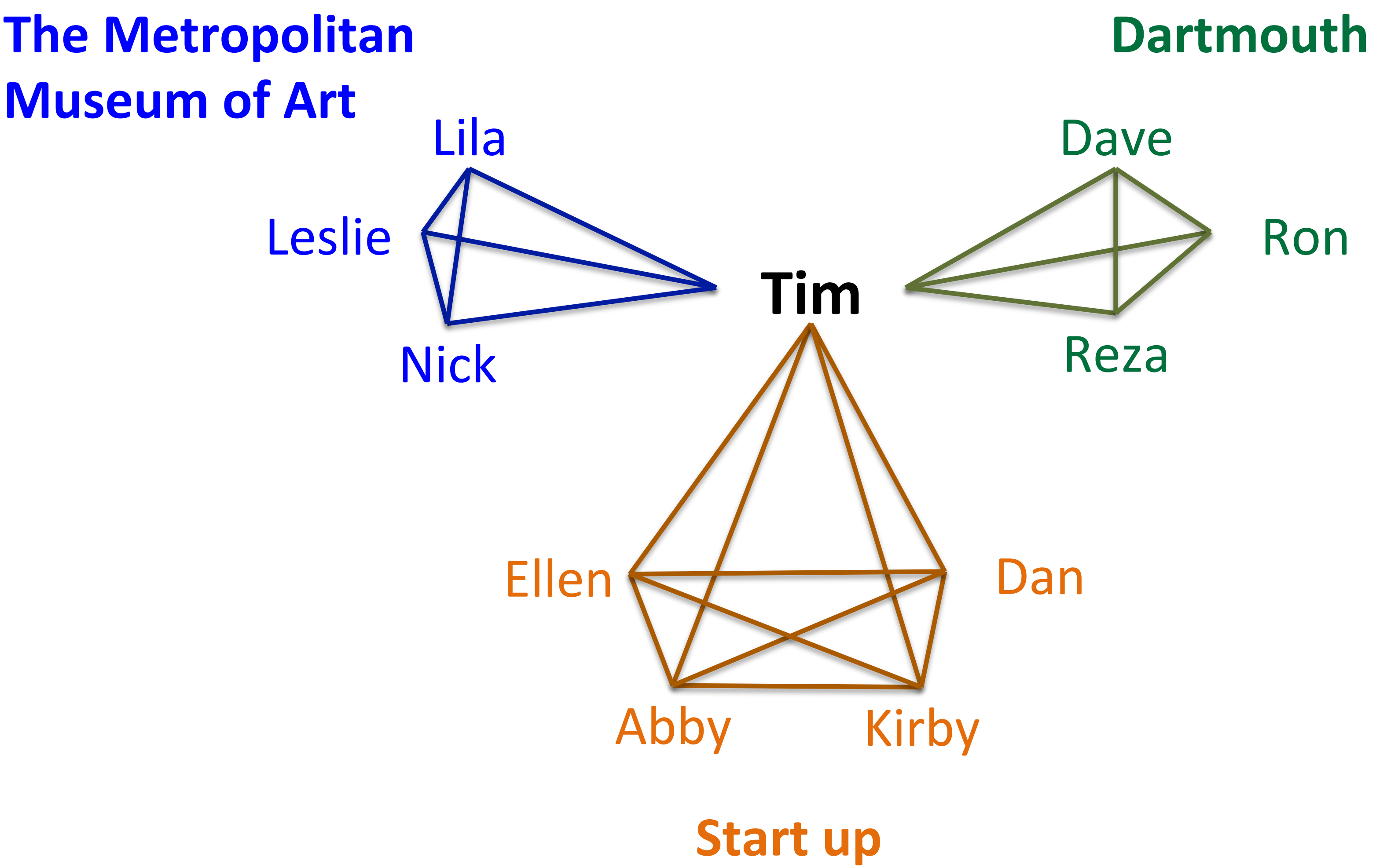
Day 15 – Relationships

Agenda

1. Graph interface
2. Four common representations
3. Implementation

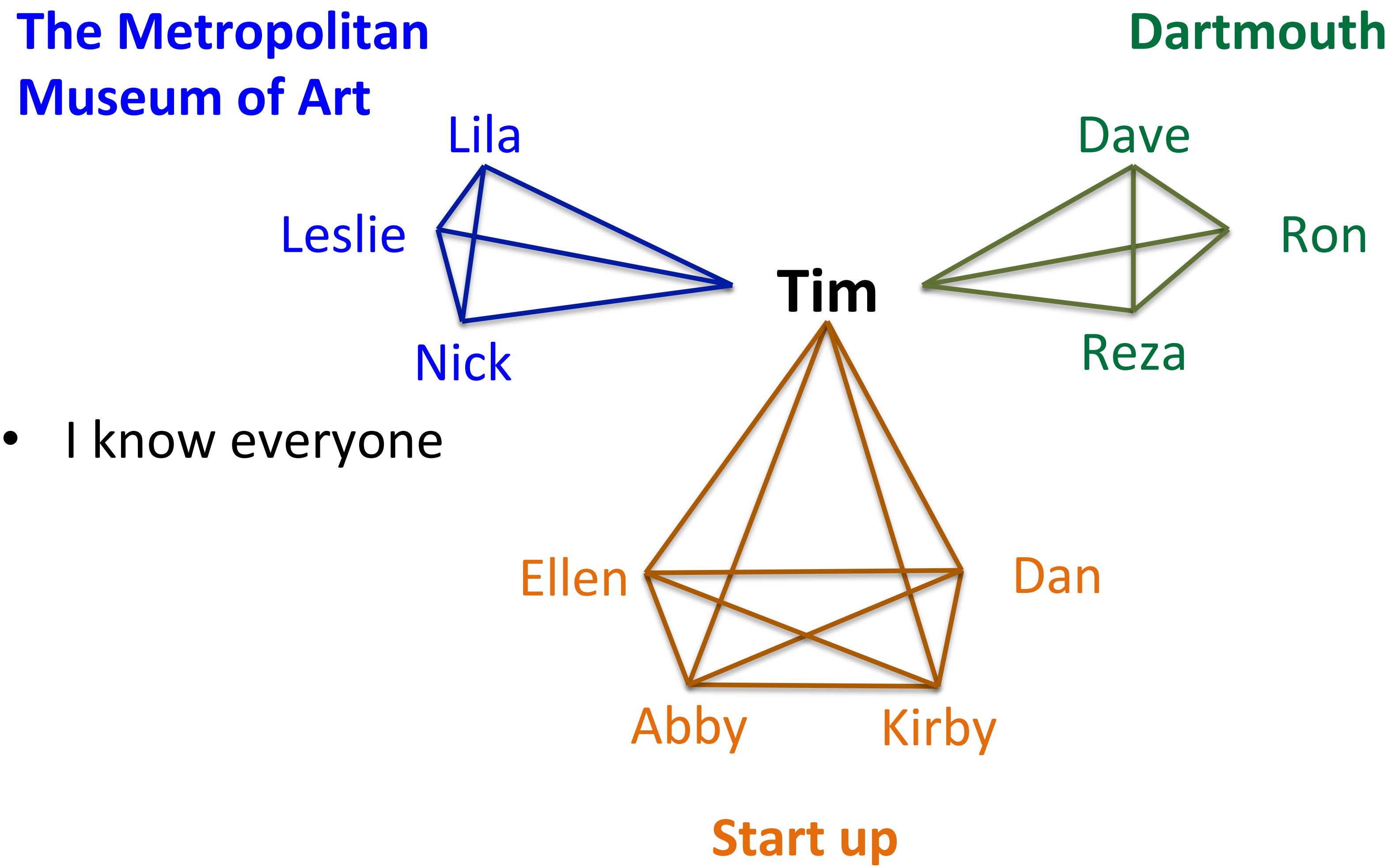
Graphs are a useful way to represent different types of relationships

My coworkers



Graphs are a useful way to represent different types of relationships

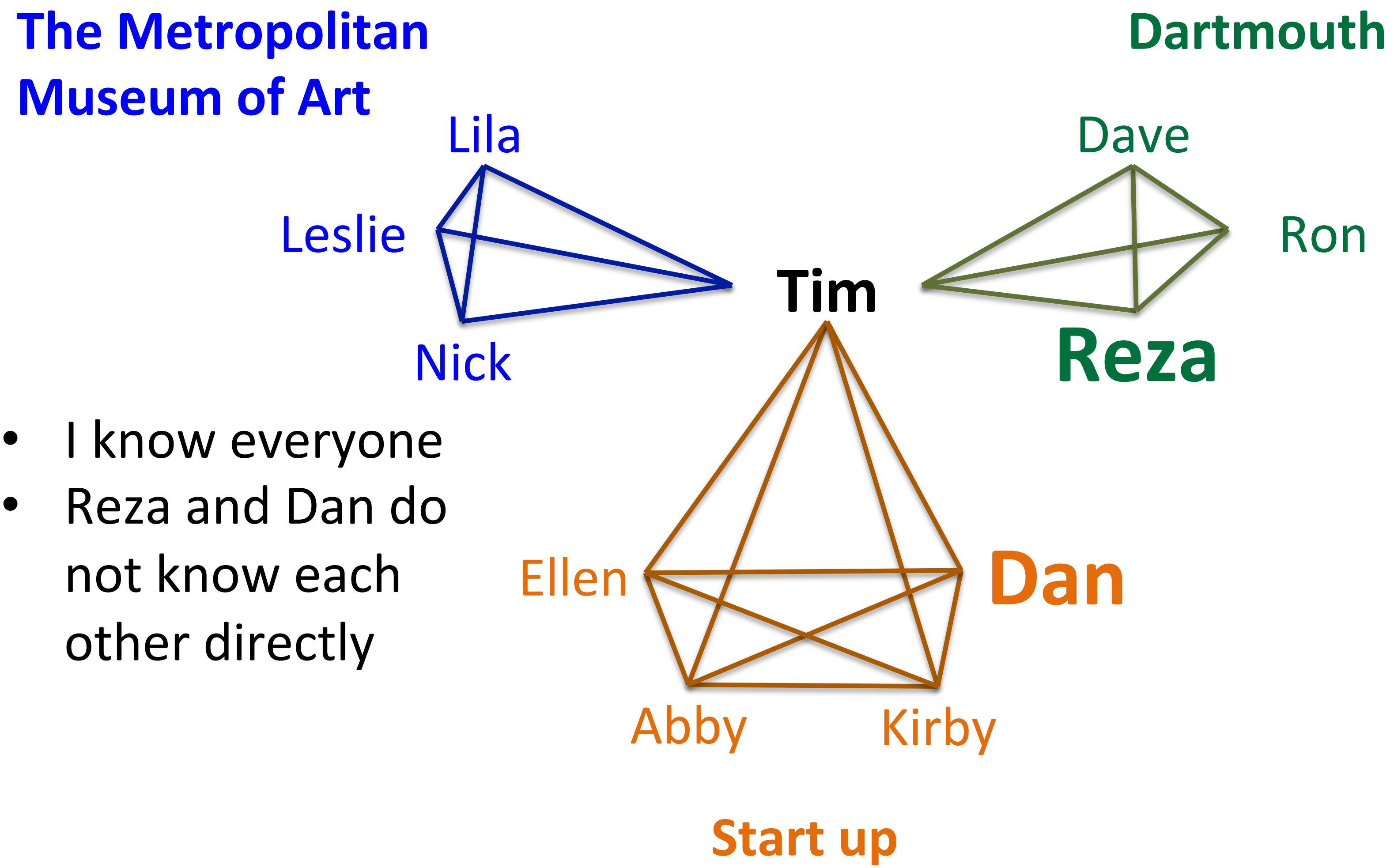
My coworkers



- I know everyone

Graphs are a useful way to represent different types of relationships

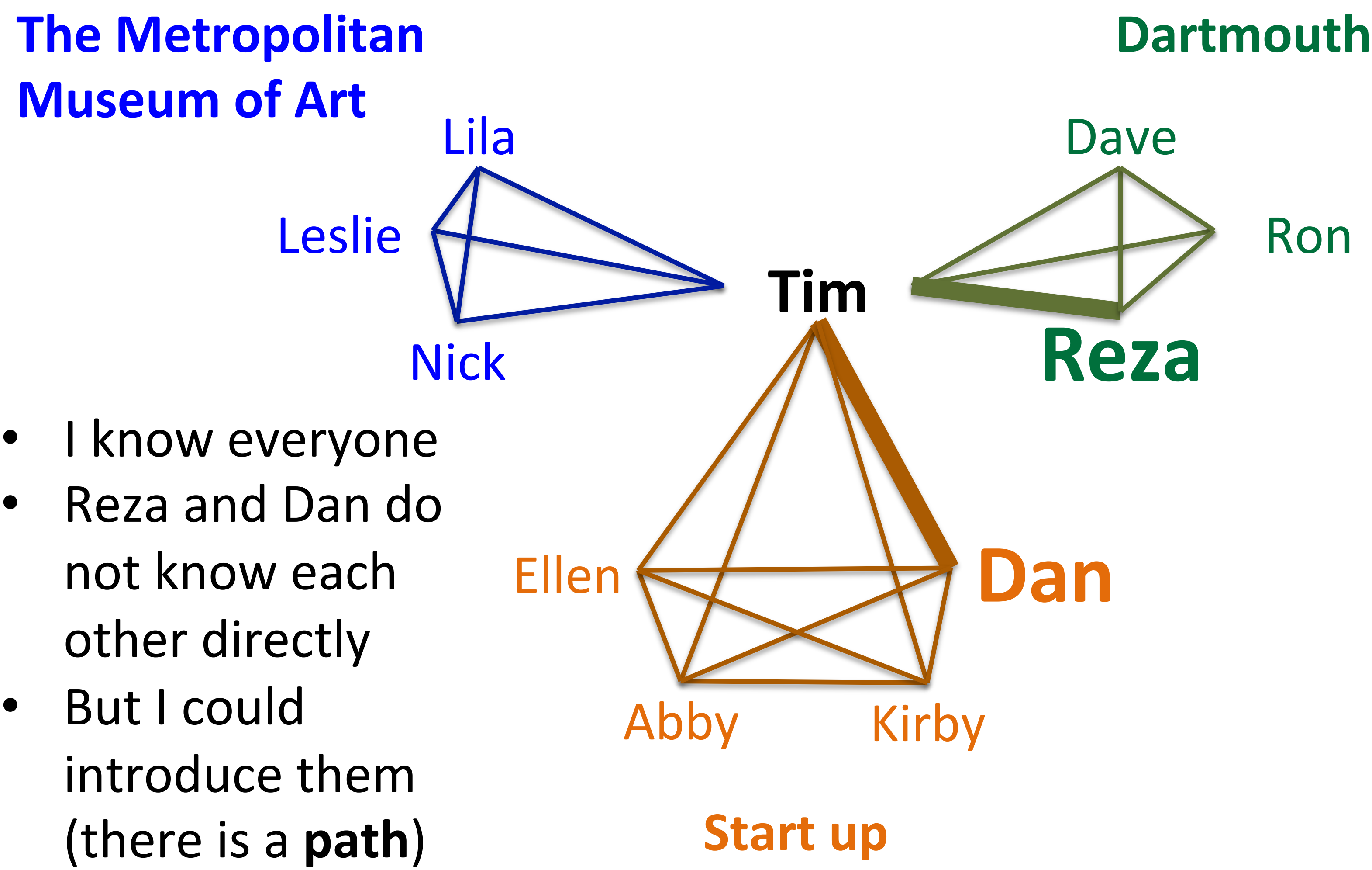
My coworkers



- I know everyone
- Reza and Dan do not know each other directly

Graphs are a useful way to represent different types of relationships

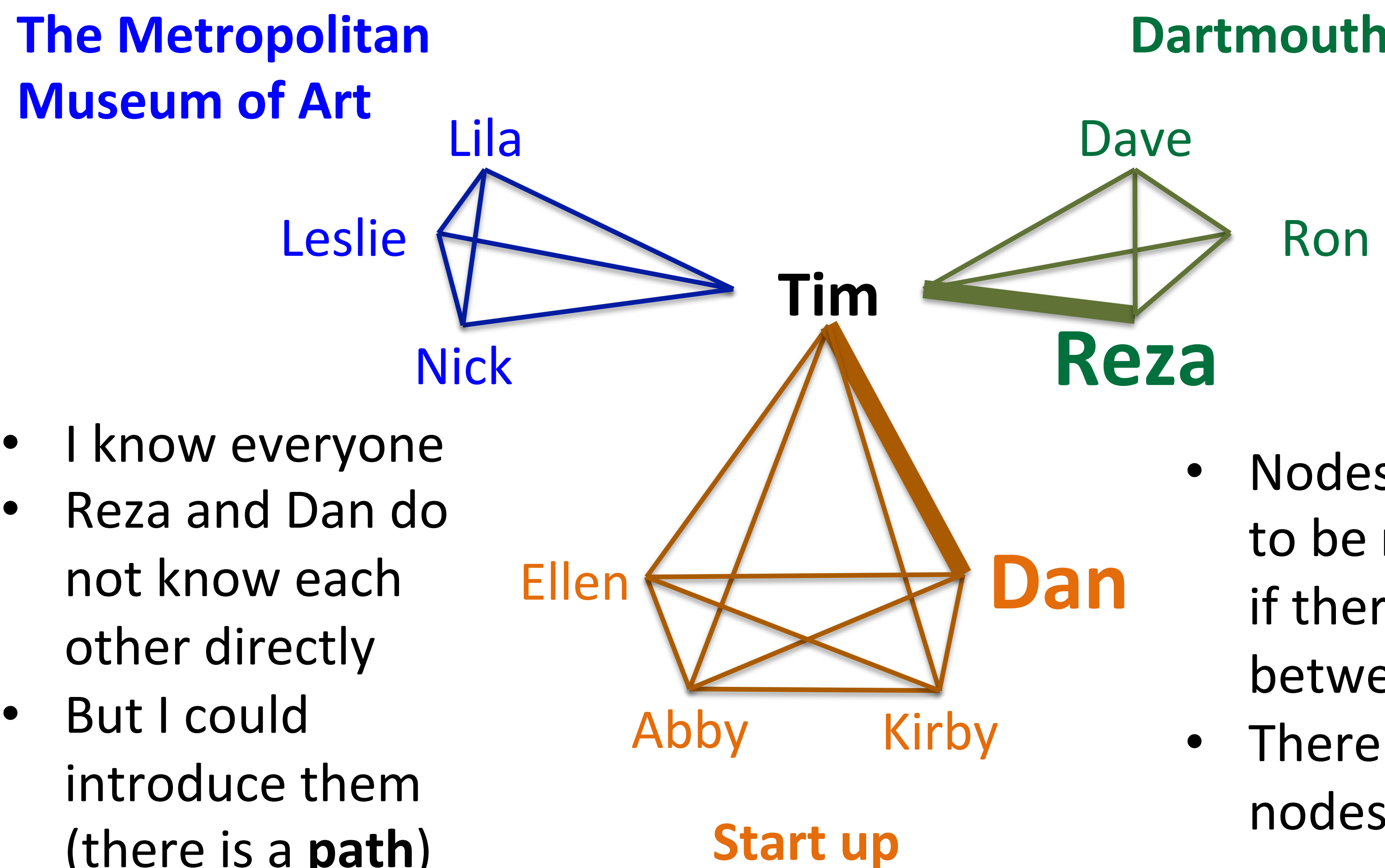
My coworkers



- I know everyone
- Reza and Dan do not know each other directly
- But I could introduce them (there is a **path**)

Graphs are a useful way to represent different types of relationships

My coworkers



- I know everyone
- Reza and Dan do not know each other directly
- But I could introduce them (there is a **path**)

- Nodes are said to be **reachable** if there is a path between then
- There may be nodes that are unreachable

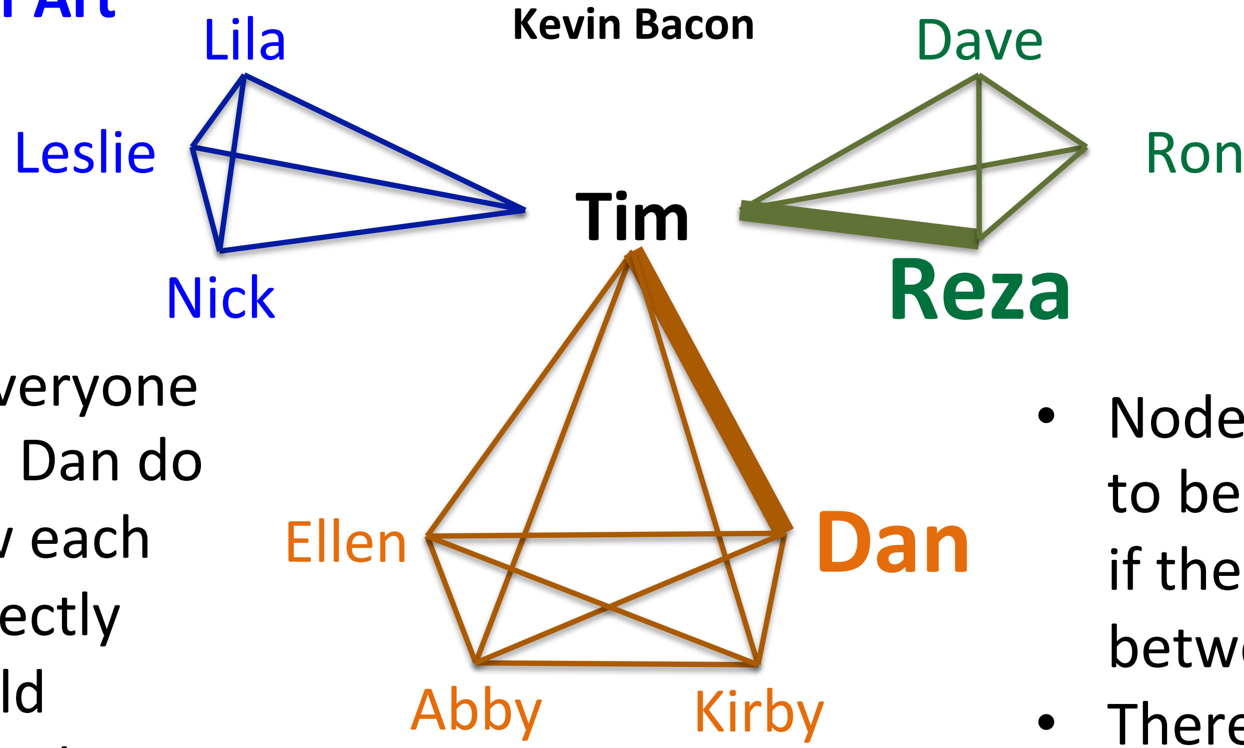
Graphs are a useful way to represent different types of relationships

My coworkers

Kevin Bacon

The Metropolitan Museum of Art

Dartmouth



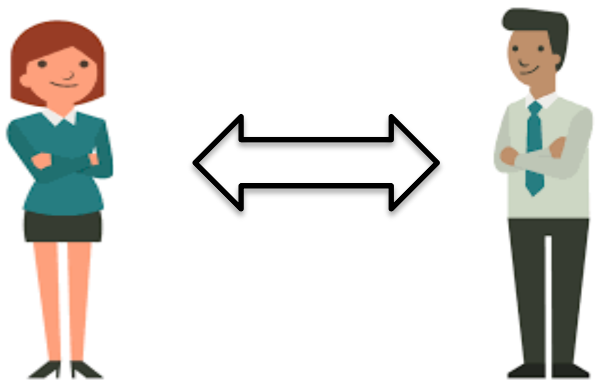
Start up

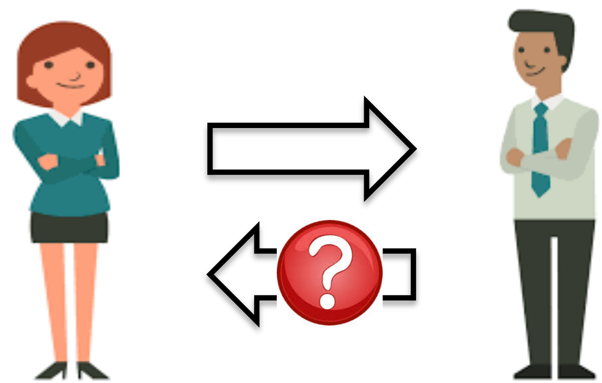
- I know everyone
- Reza and Dan do not know each other directly
- But I could introduce them (there is a **path**)

- Nodes are said to be **reachable** if there is a path between then
- There may be nodes that are unreachable

Image: nbc.com

Two types of relationships: Undirected and directed





Undirected (Symmetrical)
If Alice is friends with Bob, then Bob is friends with Alice

Directed (Asymmetrical)
If Alice follows Bob, then Bob does not necessarily follow Alice

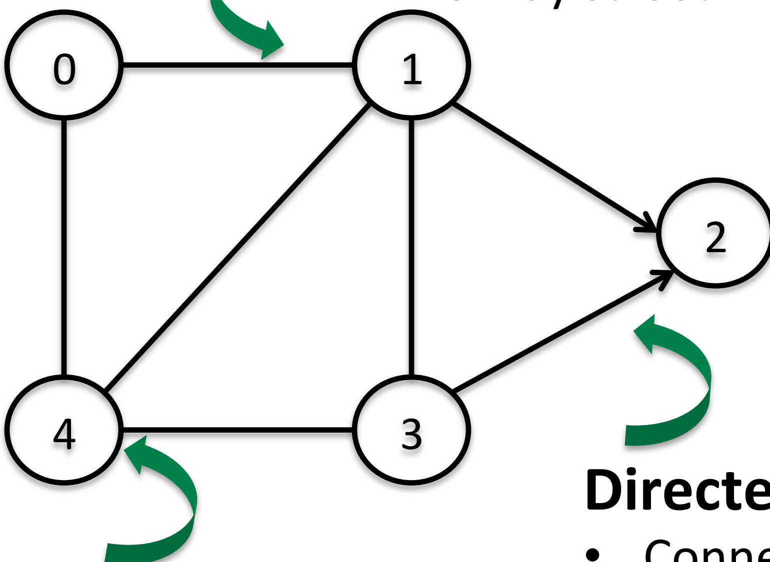
Images: Facebook, Twitter, 1designshop.com

Graphs represent directed or undirected relationships with nodes and edges

Graphs

Undirected edges

- Connect objects in both directions
- “Two way street”



Directed edges

- Connect objects in a single directions
- “One way street”

Nodes (vertices)

- Represent objects
- Could be a person or city or computer or intersection of roads…

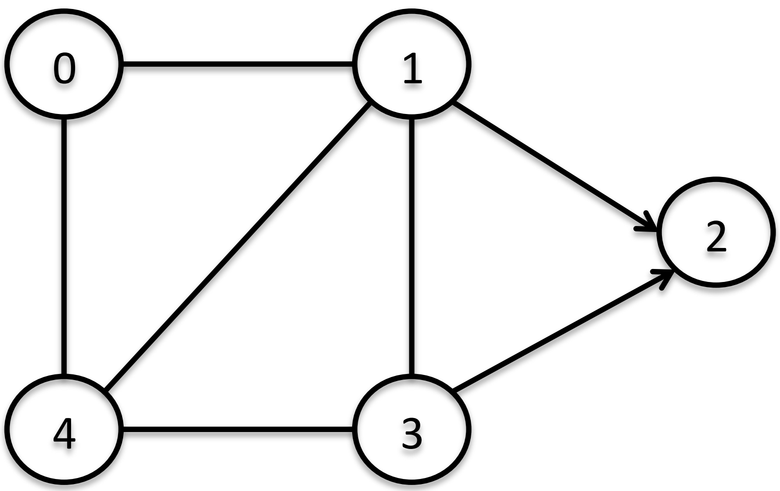
Undirected graph
Only undirected edges

Directed graph
Only directed edges

Mixed graph
Has both directed and undirected edges

Both nodes and edges can hold information about the relationship

Graphs



Nodes

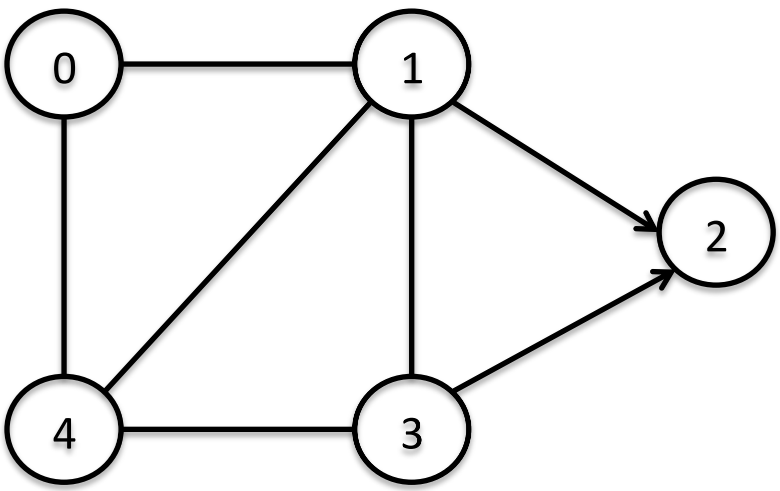
- Represent an object
- Can be as simple as a String
- Could be more complex like a Person object

Edges

- Can hold information about relationship
  - Distance between cities
  - Capacity of a pipe
  - Label of relationship type ("follower", "friend", "co-worker")

Graph ADT defines several useful methods

Graph.java



Graph methods

`outDegree/inDegree`

Count of edges out of or into a node

`outNeighbors/inNeighbors`

Other nodes connected from/to a node

`hasEdge`

True if one node connected to another

`getLabel`

Return label on edge from one node to another

`insertVertex`

Add node to graph

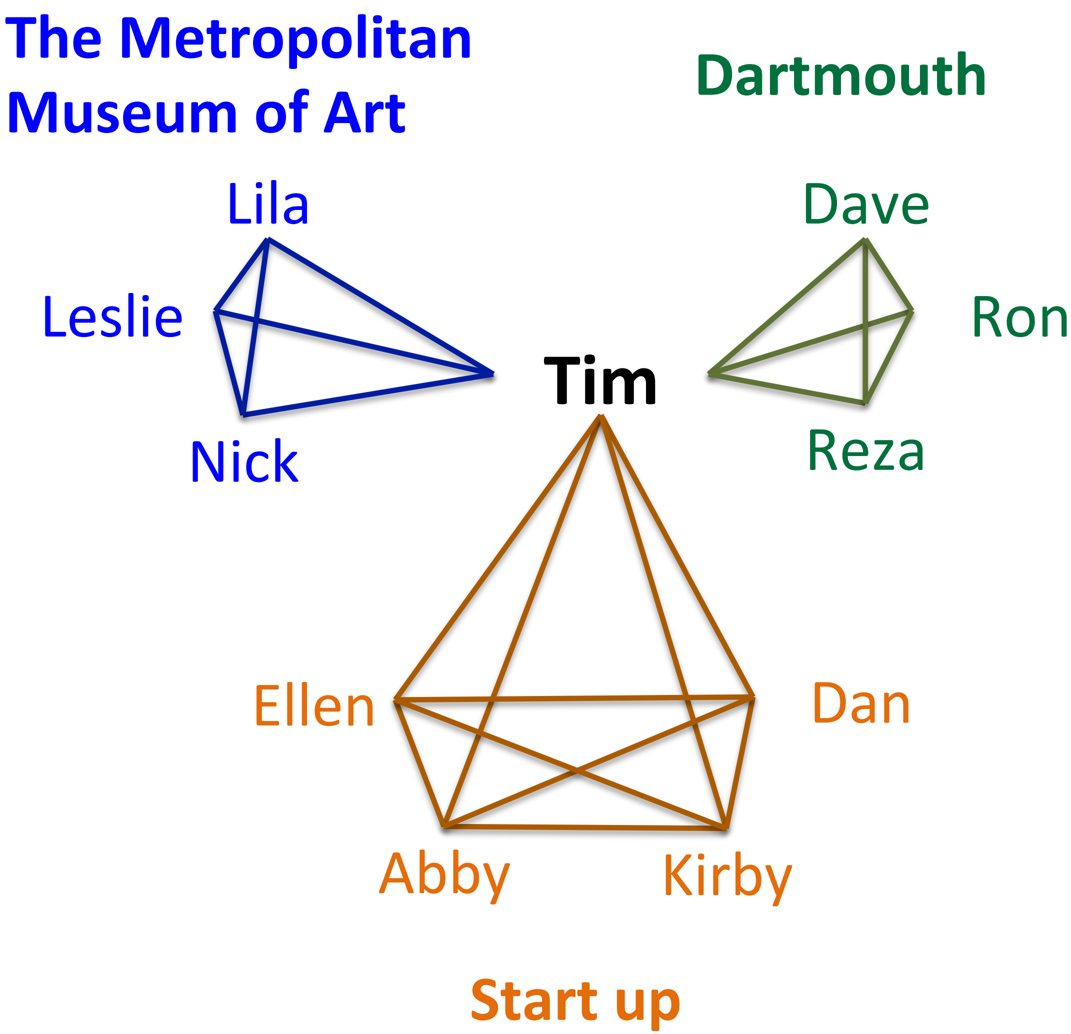
`insertDirected/Undirected`

Add edge to graph between two nodes

`removeVertex/Directed/Undirected`

Remove node or edge

We can use Graph ADT methods to answer interesting questions



Questions we can answer

- Who is the most connected? (most in edges)
- Who are mutual acquaintances ("cliques" where all nodes have edges to each other)
- Who is a friend-of-a-friend but is not yet a friend? (breadth-first search, next class)

We can use Graph ADT methods to answer interesting questions

RelationshipTest.java

- Undirected edges implemented as a directional edge in both directions
- Run to breakpoint after building graph

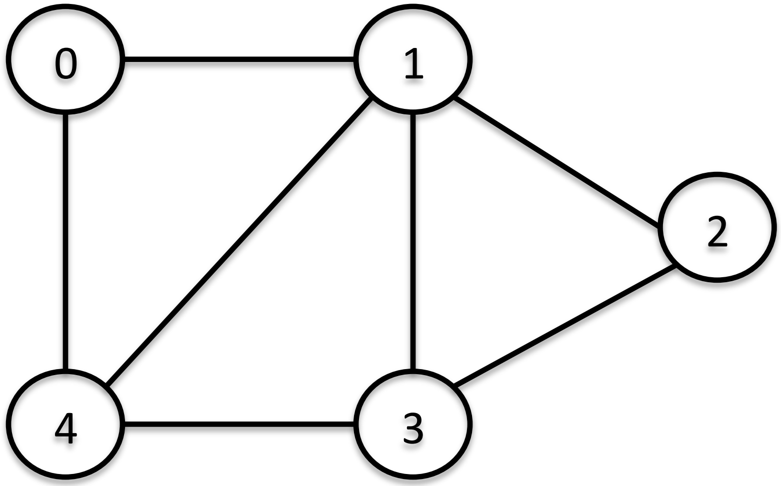
Agenda

1. Graph interface
2. Four common representations
3. Implementation

Edge Lists create an unordered list of vertex pairs where each entry is an edge

1. Edge List

```
{ {0,1}, {0,4}, {1,2}, {1,3},
{1,4}, {2,3}, {3,4} }
```



Assume:

n nodes (here 5)

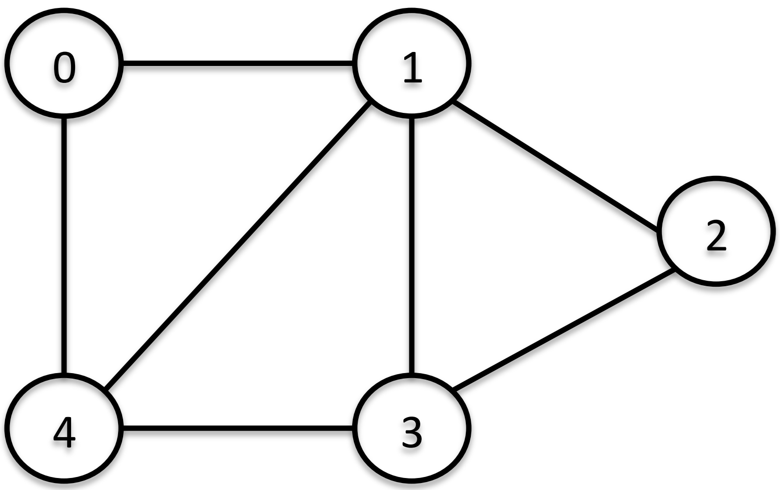
m edges (here 7)

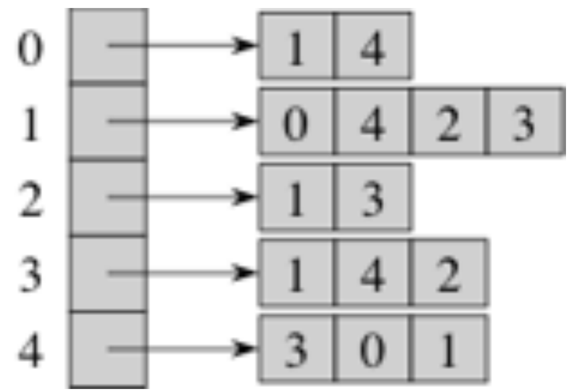
Notes:

- Number nodes 0..n-1
- Store node `i` in array at index `i`
- Edge List stores pairs of indexes that reference nodes in array
- Each Edge List entry represents an edge between two nodes
- Insert fast, just add to list
- Everything else slow
- Example: `removeVertex` is O(m), have to remove all edges to/from node, so search all edges leading to or from node

Adjacency Lists store adjacent nodes in a list; gives improved performance

2. Adjacency List





Assume:

n nodes (here 5)

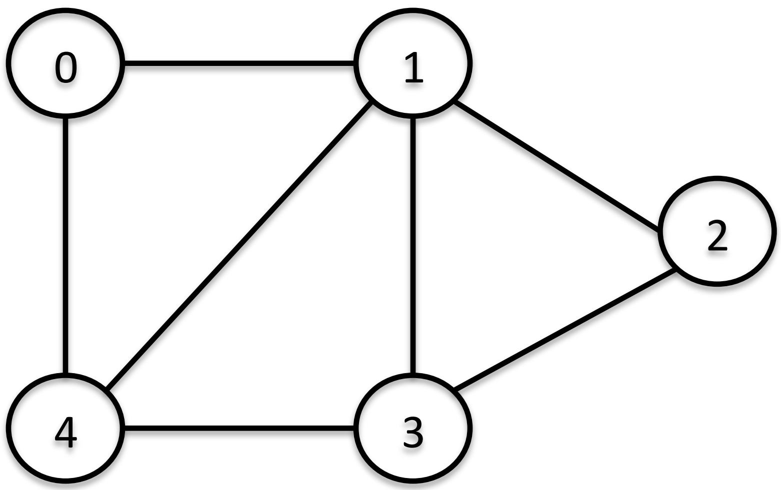
m edges (here 7)

Notes:

- Store linked list of nodes in or out of each vertex (same if undirected)
- Might keep two lists, one for in neighbors and one for out neighbors
- Faster to get neighbors than Edge List, just iterate in O(degree(v)) vs. O(m)

Adjacency Matrices create an n x n array to indicate existence of edges

3. Adjacency Matrix



To

From	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Assume:

n nodes (here 5)

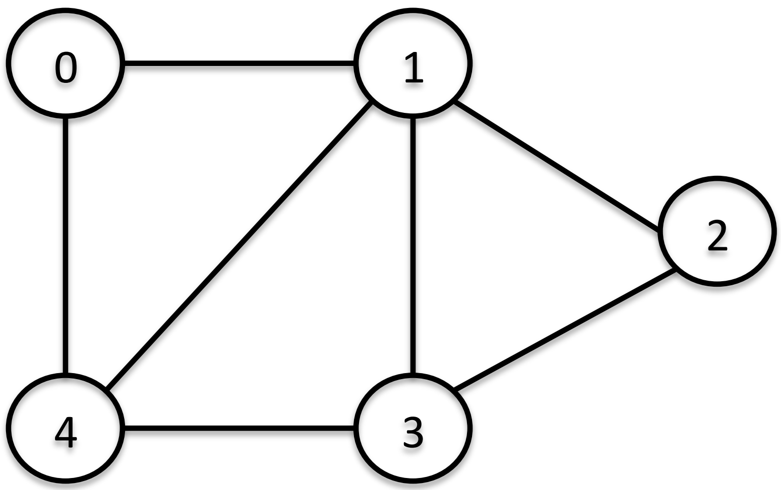
m edges (here 7)

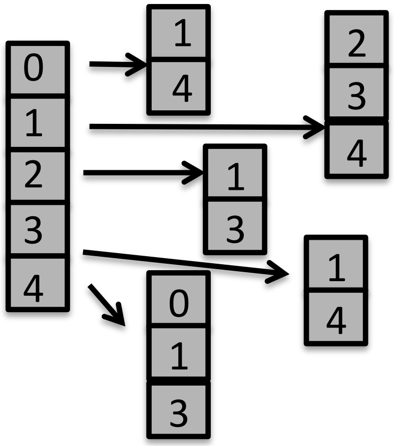
Notes:

- Create n x n matrix A, set A[i,j] = 1 if edge from node i to node j, else 0
- Works if no parallel edges
- Undirected graph A[i,j] == A[j,i]
- `hasEdge(u,v)` is now O(1), whereas in Adjacency List it was O(degree(u))
- Finding neighbors now O(n) because have to check entire row or column
- Adding/removing vertices O(n^2), have to rebuild entire matrix

Adjacency Maps create a Map for each node and a second Map to adjacent nodes

4. Adjacency Map





Assume:

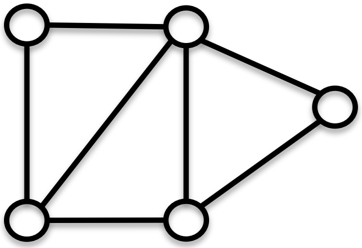
n nodes (here 5)

m edges (here 7)

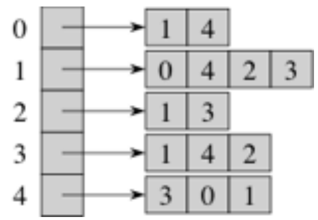
Notes:

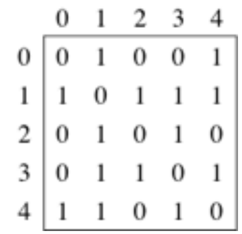
- Create Map of nodes
- Each entry in Map holds a second Map of adjacent nodes
- No need to number nodes in order
- `hasEdge(u,v)` now O(1)
  - Look up u in Map O(1)
  - Look up v in second Map O(1)

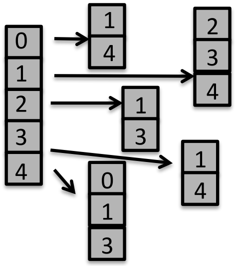
How a Graph is implemented has a big impact on run-time performance



```
{{0,1},
{0,4}, {1,2},
{1,3}, {1,4},
{2,3}, {3,4}}
```







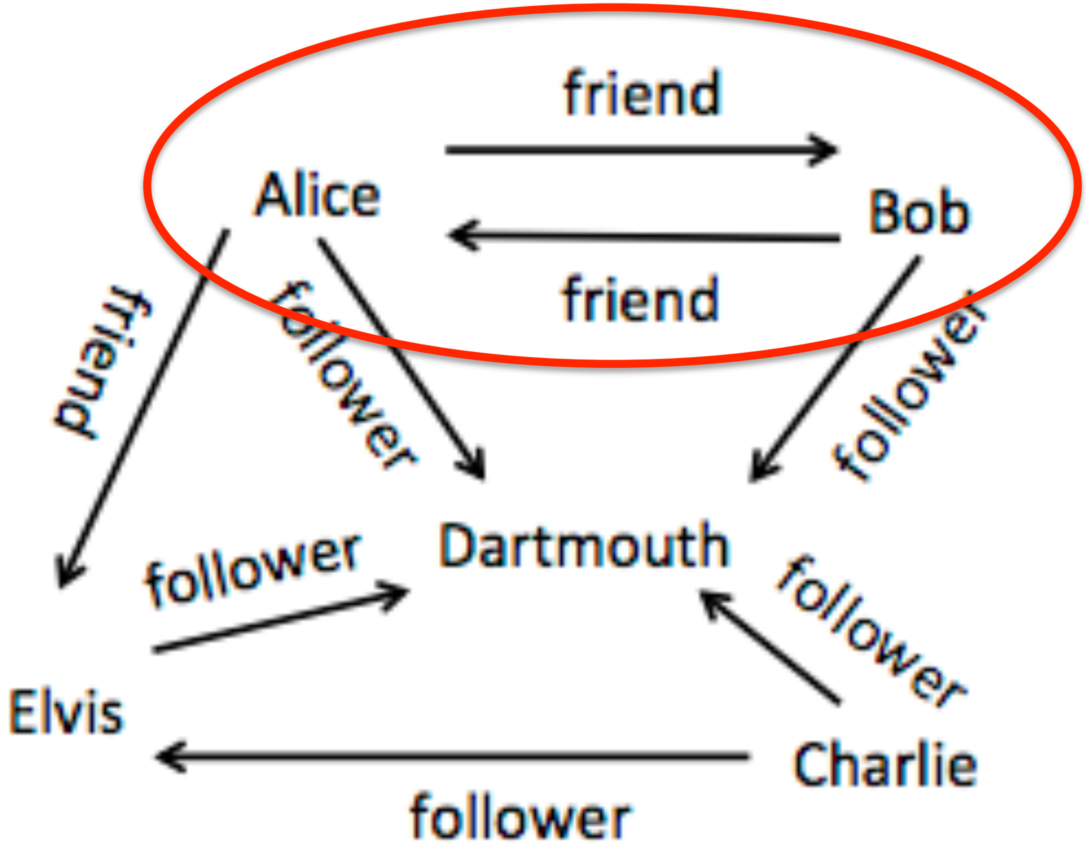
Method	**Edge List**	**Adjacency List**	**Adjacency Matrix**	**Adjacency Map**
`in/outDegree(v)`	O(m)	O(1)	O(n)	O(1)
`in/outNeighbors(v)`	O(m)	$O(d_v)$	O(n)	$O(d_v)$
`hasEdge(u,v)`	O(m)	$O(\min(d_u,d_v))$	O(1)	O(1)
`insertVertex(v)`	O(1)	O(1)	$O(n^2)$	O(1)
`removeVertex(v)`	O(m)	$O(d_v)$	$O(n^2)$	$O(d_v)$
`insertEdge(u,v)`	O(1)	O(1)	O(1)	O(1)
`removeEdge(u,v)`	O(m)	O(1)	O(1)	O(1)

n = number of nodes (5), m = number of edges (7), d_v = degree of node v

Agenda

1. Graph interface
2. Four common representations
3. Implementation

Our implementation will allow a mixed graph (directed and undirected edges)



Undirected edges are two directed edges, one in each direction

Graph.java specifies the Graph Interface

Graph.java

- Interface specifying graph methods

AdjancyMapGraph.java stores in an out edges in two different Maps

AdjacenyMapGraph.java

- Maintain two Maps for each vertex
  - One for contains `in` edges, one contains `out` edges
  - `out` and `in` are `Map<V, Map<V,E>>`
    - `Key` is vertex (string)
    - `Value` is a Map with vertex as key and edge as value (both strings)
- `insertVertex (V v)`
  - Add vertex `v` to `in` and `out` Maps with new HashMap for edges (no edges set yet)
- `insertDirected (V u, V v, E e)`
  - Update `out` on `u` by adding to Map to `v` and label `e`
  - Update `in` on `v` by adding to Map from `u` and label `e`
- `insertUndirected (V u, V v, E e)`
  - Add two directed edges between `u` and `v`, one going each direction

AdjancyMapGraph.java stores in an out edges in two different Maps

AdjacenyMapGraph.java

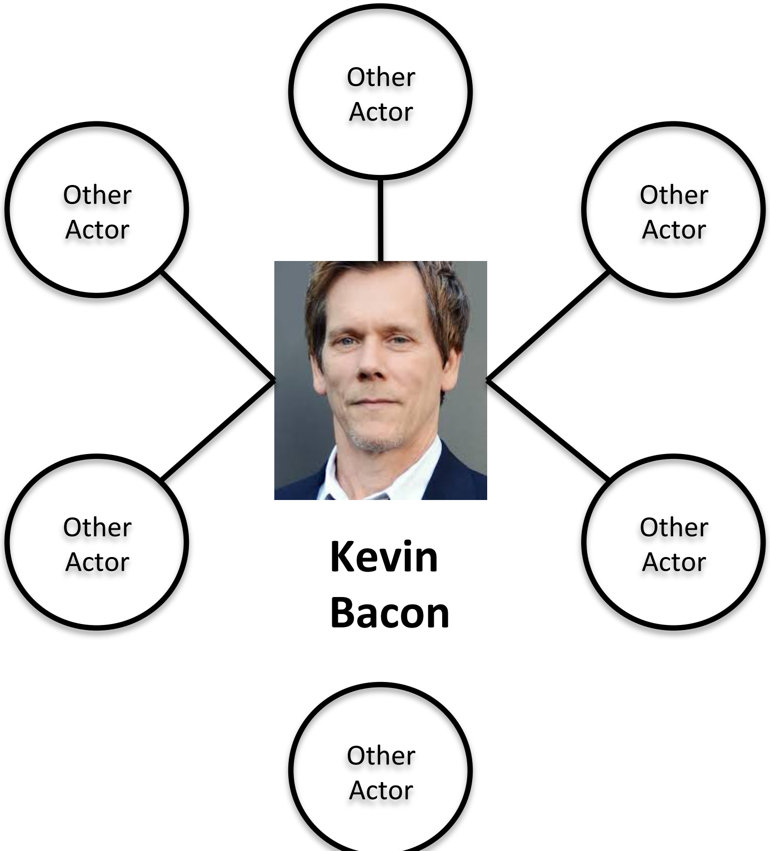
- `removeDirected(V u, V v)`
  - Remove edge from both in and out
- `removeUndirected(V u, V v)`
  - Call `removeDirected` twice, once for each node direction
- Review other methods

RelationshipTest.java shows how it all works

RelationshipTest.java

- Run

We can use Graph ADT methods to answer interesting questions

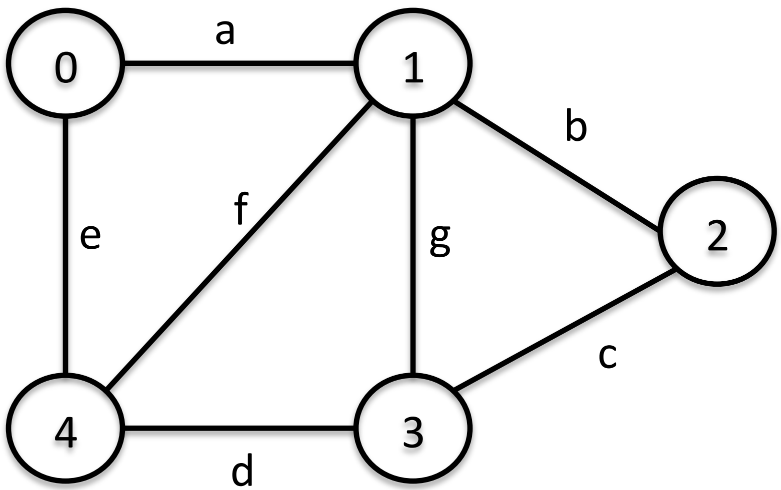


Questions we can answer

- Who is the most popular? (most in edges)
- Who are mutual acquaintances ("cliques" where all nodes have edges to each other)
- Who is a friend-of-a-friend but is not yet a friend? (breadth-first search, next class)

Image: nbc.com

Three common ways to represent graphs: Edge List, Adjacency List, Adjacency Matrix

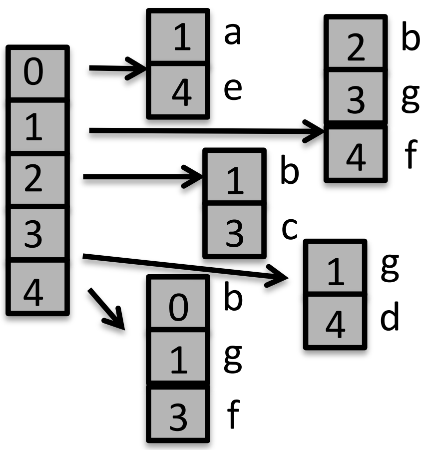


Assume:

n nodes (here 5)

m edges (here 7)

3. Adjacency Map

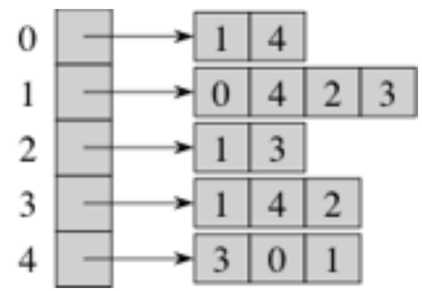


Notes:

- Create n x n matrix A, set A[i,j] = 1 if edge from node i to node j, else 0
- Works if no parallel edges
- Undirected graph A[i,j] == A[j,i]
- `hasEdge(u,v)` is now O(1), whereas in Adjacency List it was O(degree(u))
- Finding neighbors now O(n) because have to check entire row or column
- Adding/removing vertices O(n^2), have to rebuild entire matrix

How a Graph is implemented has a big impact on run-time performance

```
{{0,1}, {0,4},
{1,2}, {1,3},
{1,4}, {2,3},
{3,4}}
```



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Method	**Edge List**	**Adj. List**	**Adj. Matrix**
`in/outDegree(v)`	O(m)	O(1)	O(n)
`in/outNeighbors(v)`	O(m)	$O(d_v)$	O(n)
`hasEdge(u,v)`	O(m)	$O(\min(d_u,d_v))$	O(1)
`insertVertex(v)`	O(1)	O(1)	$O(n^2)$
`removeVertex(v)`	O(m)	$O(d_v)$	$O(n^2)$
`insertEdge(u,v)`	O(1)	O(1)	O(1)
`removeEdge(u,v)`	O(m)	O(1)	O(1)

n = number of nodes, m = number of edges, d_v = degree of node v