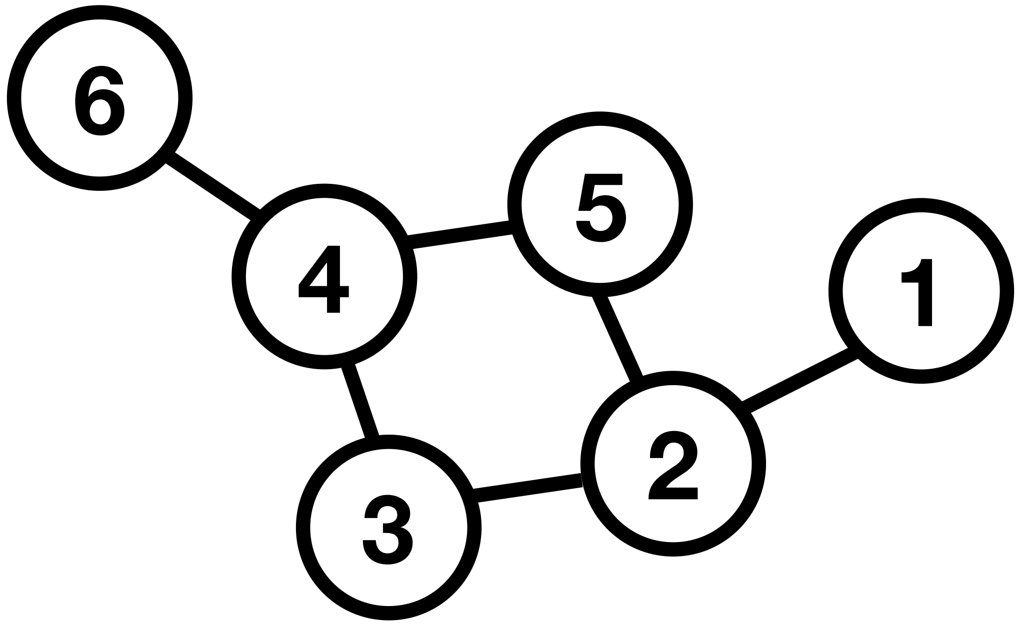
CS 106X

Lecture 22: Graphs

Friday, March 3, 2017



Programming Abstractions (Accelerated)
Winter 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapter 18

Today's Topics

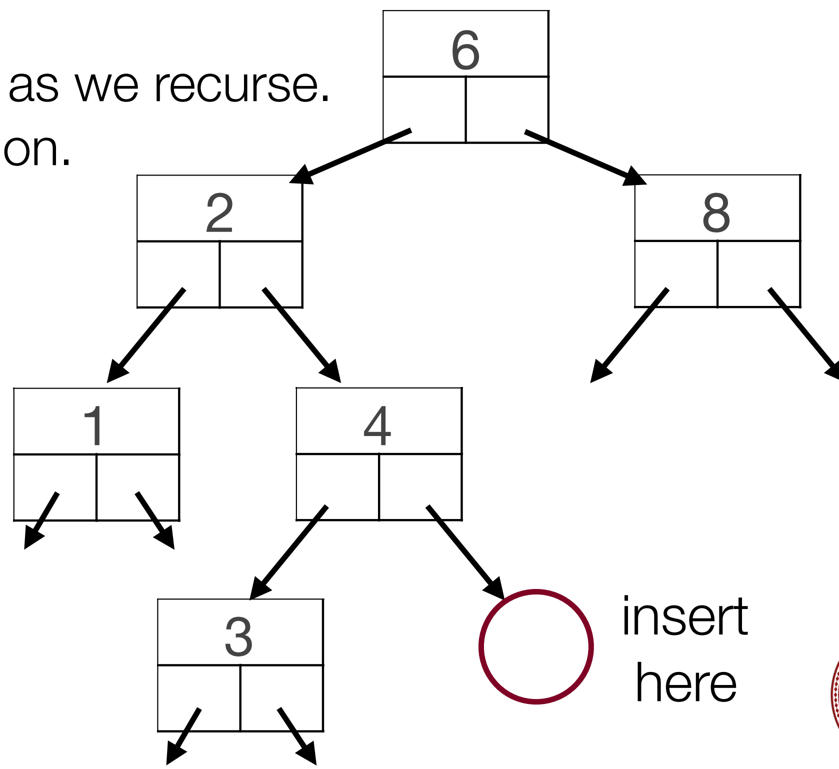
- Logistics
  - Regrade requests due Today
  - Meeting sign-up with Chris:
    - http://stanford.edu/~cgregg/cgi-bin/inperson/index.cgi

- Binary Search Trees: using references to pointers
- Assignment 6: Huffman Encoding and 21 Questions Redux
  - YEAH hours video from last quarter on Huffman: https://youtu.be/BZarC2LkjeI
- Introduction to Graphs

Using References to Pointers

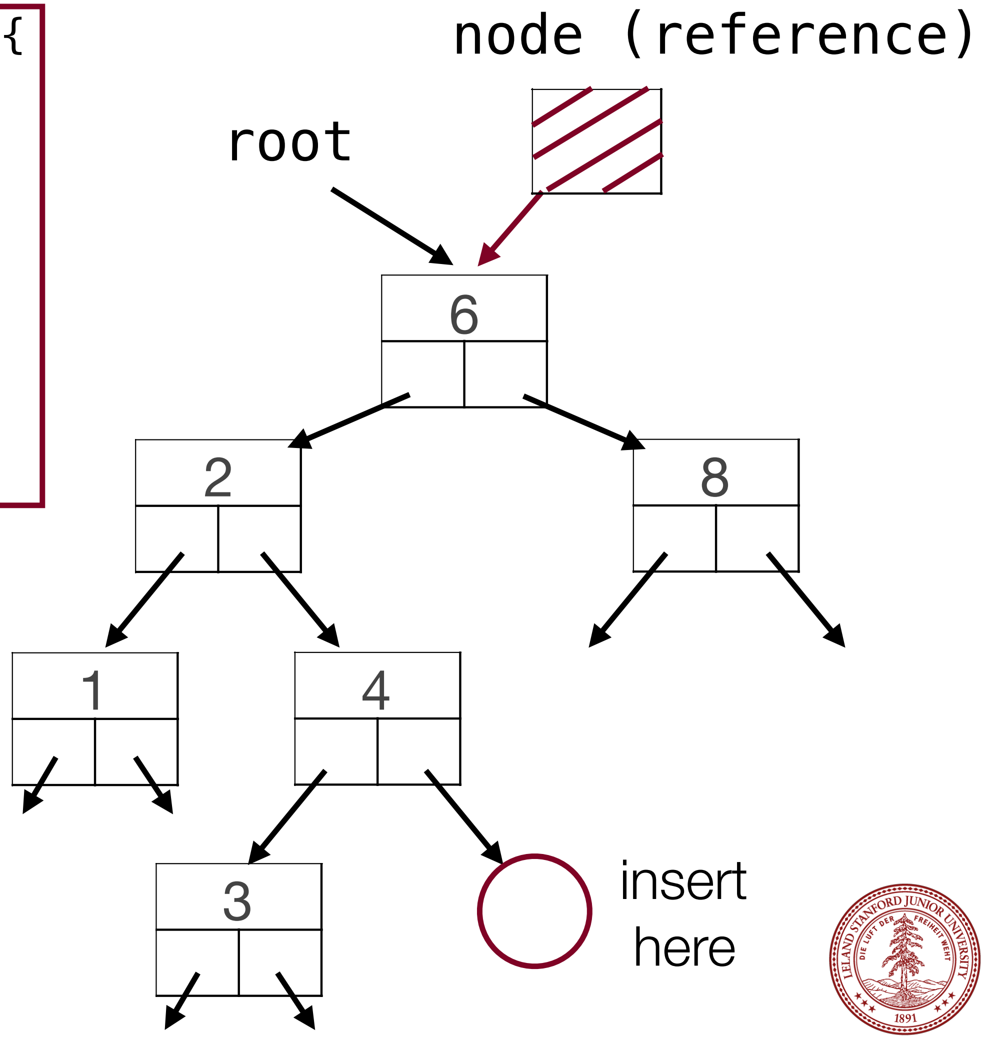
- To insert into a binary search tree, we must update the left or right pointer of a node when we find the position where the new node must go.
- In principle, this means that we could either
  1. Perform arms-length recursion to determine if the child in the direction we will insert is NULL, or
  2. Pass a *reference to a* pointer to the parent as we recurse.
- The second choice above is the cleaner solution.

```
set.insert(5)
```



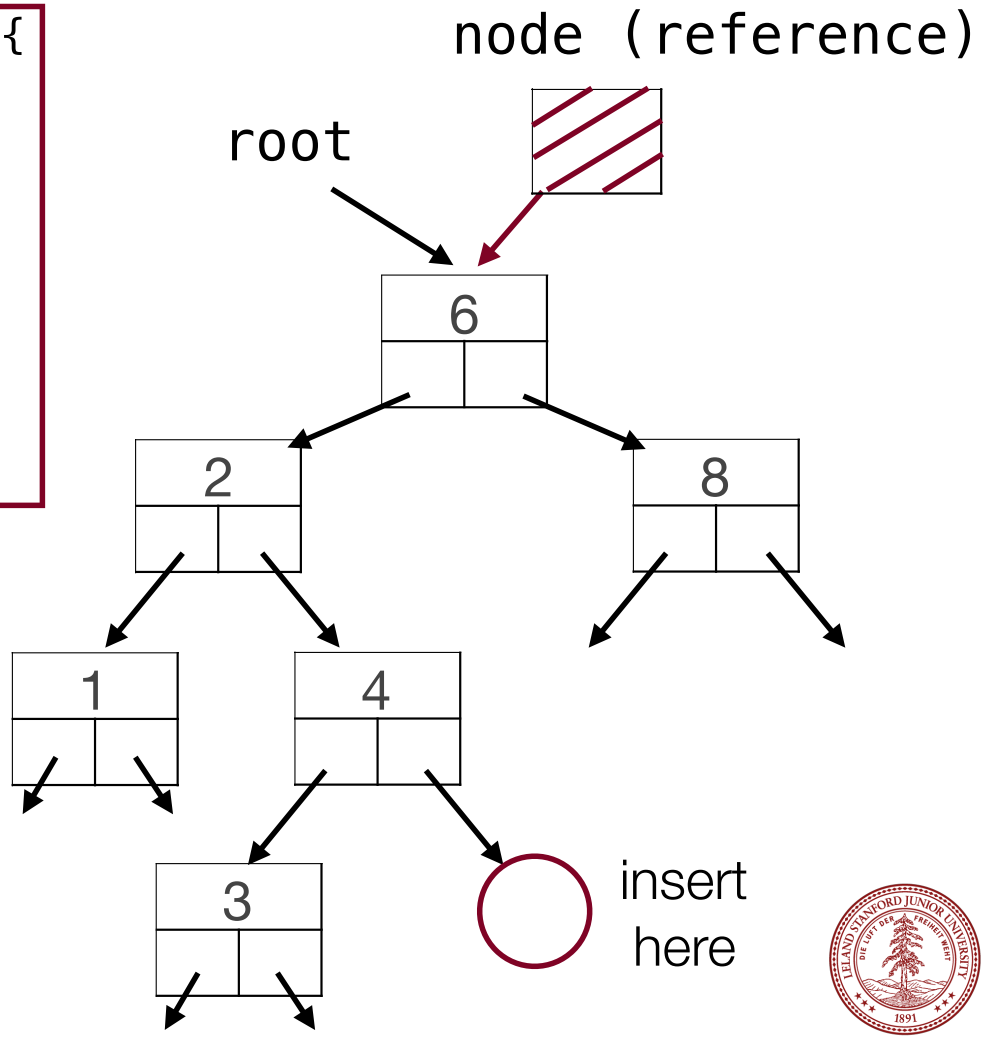
Using References to Pointers

```
void StringSet::add(string s, Node* &node) {
    if (node == NULL) {
        node = new Node(s);
        count++;
    } else if (node->str > s) {
        add(s, node->left);
    } else if (node->str < s) {
        add(s, node->right);
    }
}
```



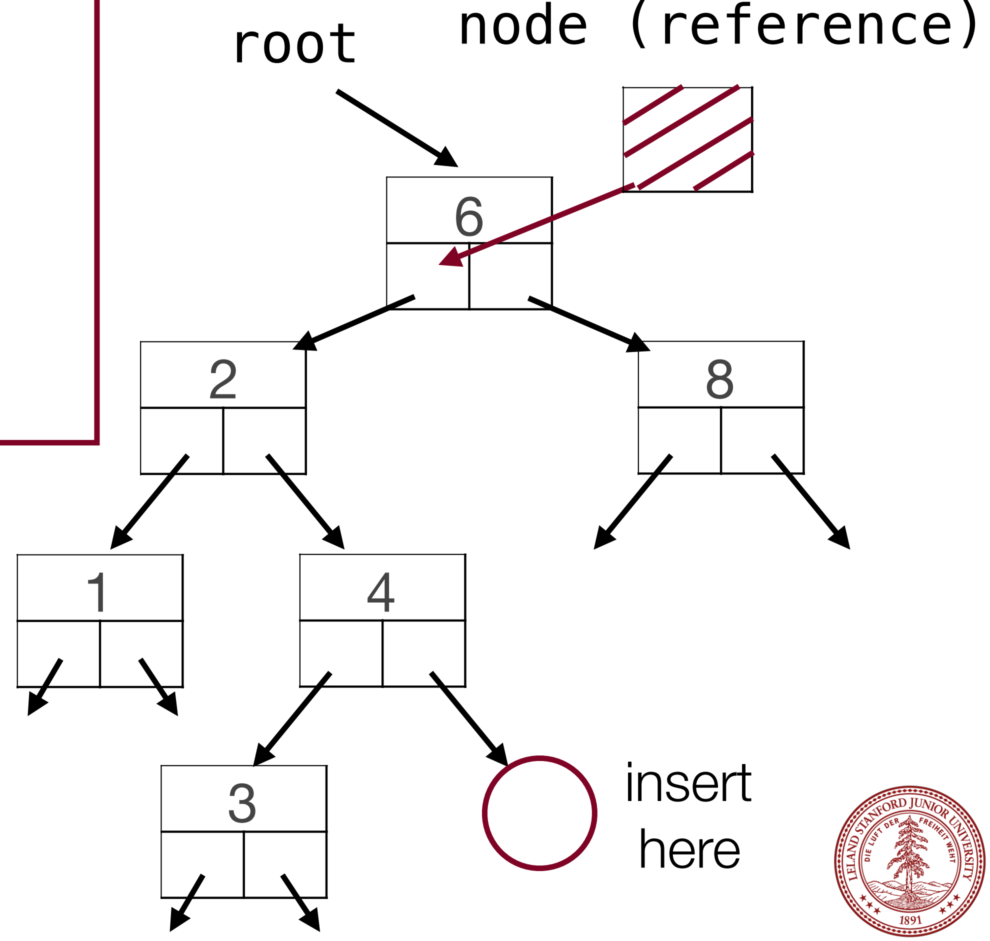
Using References to Pointers

```
void StringSet::add(string s, Node* &node) {
    if (node == NULL) {
        node = new Node(s);
        count++;
    } else if (node->str > s) {
        add(s, node->left);
    } else if (node->str < s) {
        add(s, node->right);
    }
}
```



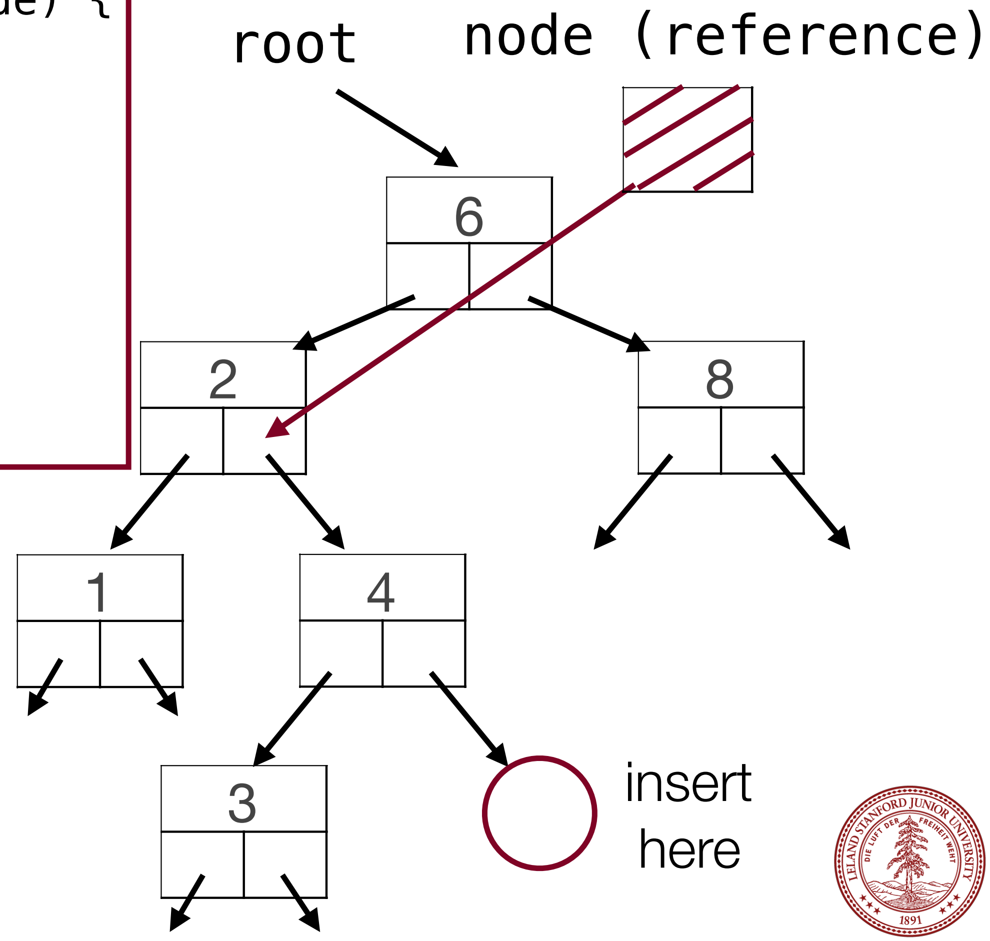
Using References to Pointers

```
void StringSet::add(string s, Node* &node) {
    if (node == NULL) {
        node = new Node(s);
        count++;
    } else if (node->str > s) {
        add(s, node->left);
    } else if (node->str < s) {
        add(s, node->right);
    }
}
```



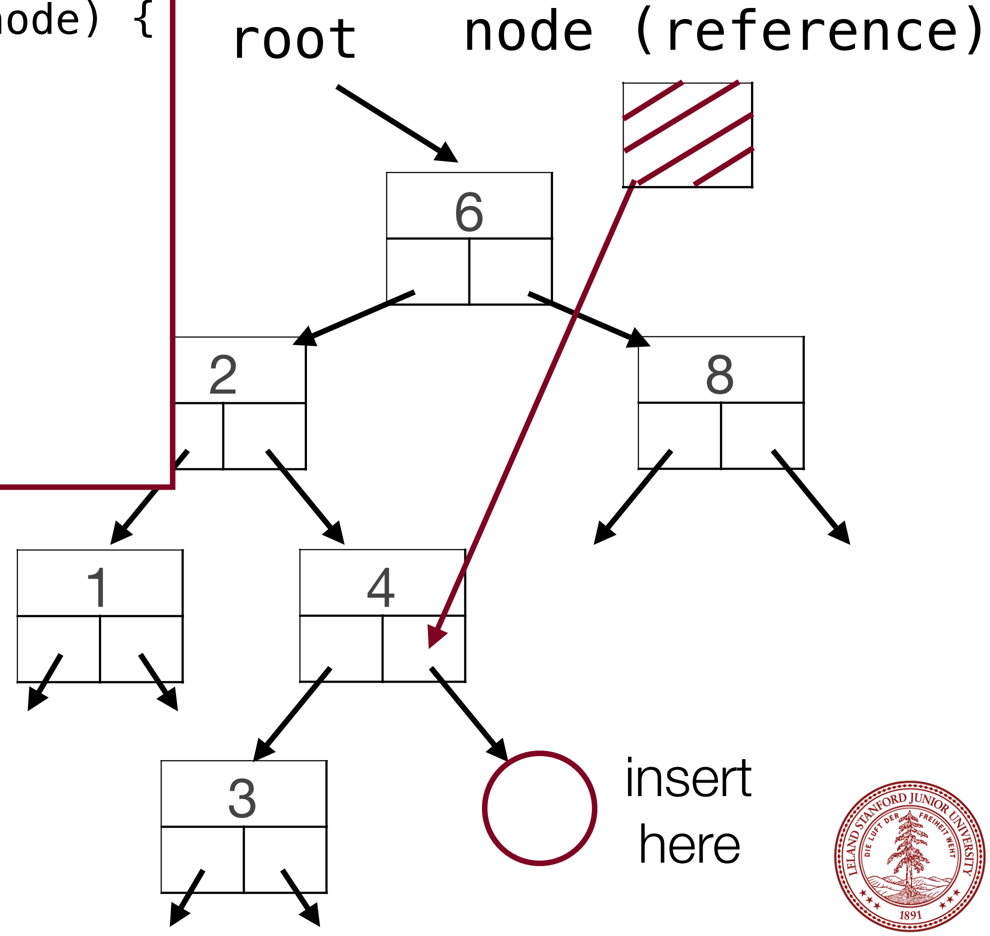
Using References to Pointers

```
void StringSet::add(string s, Node* &node) {
    if (node == NULL) {
        node = new Node(s);
        count++;
    } else if (node->str > s) {
        add(s, node->left);
    } else if (node->str < s) {
        add(s, node->right);
    }
}
```



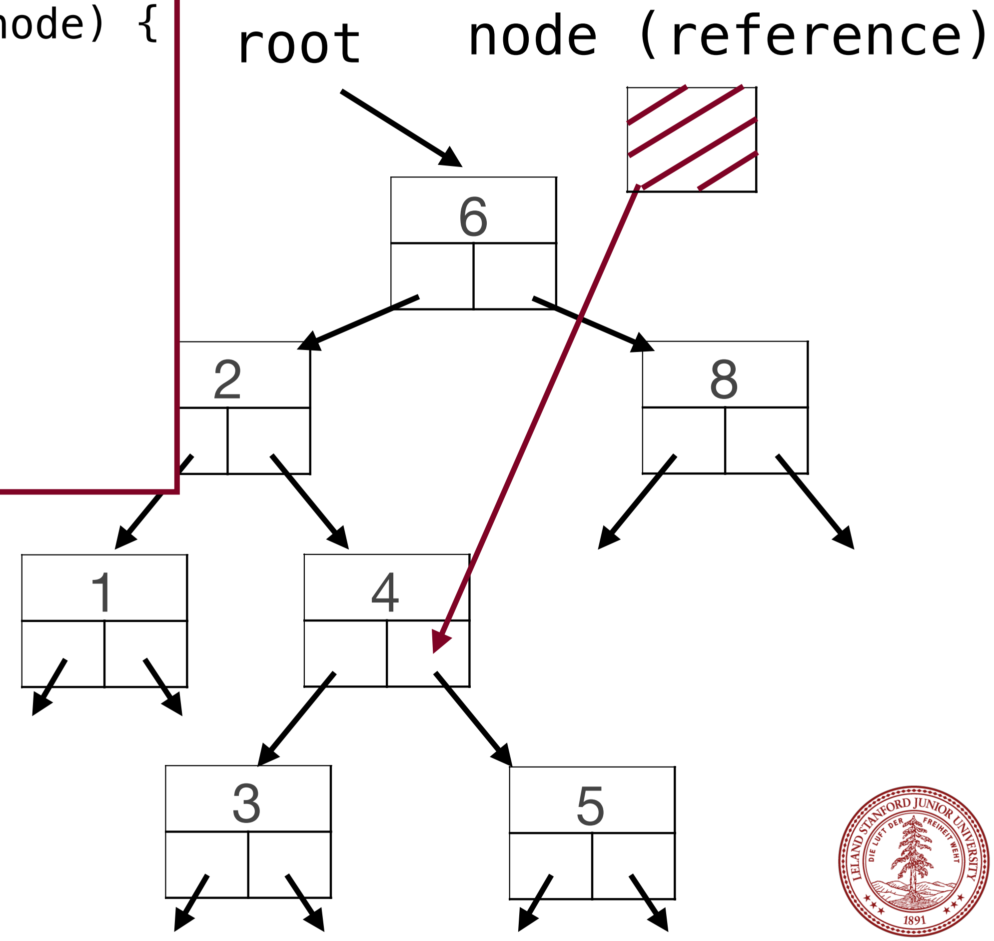
Using References to Pointers

```
void StringSet::add(string s, Node* &node) {
    if (node == NULL) {
        node = new Node(s);
        count++;
    } else if (node->str > s) {
        add(s, node->left);
    } else if (node->str < s) {
        add(s, node->right);
    }
}
```



Using References to Pointers

```
void StringSet::add(string s, Node* &node) {
    if (node == NULL) {
        node = new Node(s);
        count++;
    } else if (node->str > s) {
        add(s, node->left);
    } else if (node->str < s) {
        add(s, node->right);
    }
}
```



Assignment 6a: 21 Questions Redux

Remember this?

```
Is it an animal? y
Can it fly? n
Does it have a tail? y
Does it squeak? n
Are you thinking of: lion? y
Hooray, I win!
```

Assignment 6a: 21 Questions Redux

Now we do this!

```
Is it an animal? y
Can it fly? n
Does it have a tail? y
Does it squeak? y
Are you thinking of: lion? n
Drat, I lost. What was your object? elephant
Type a Y/N question to distinguish elephant from lion: Does it have a trunk?
And what is the answer for elephant? yes
```

Assignment 6b: Huffman Encoding

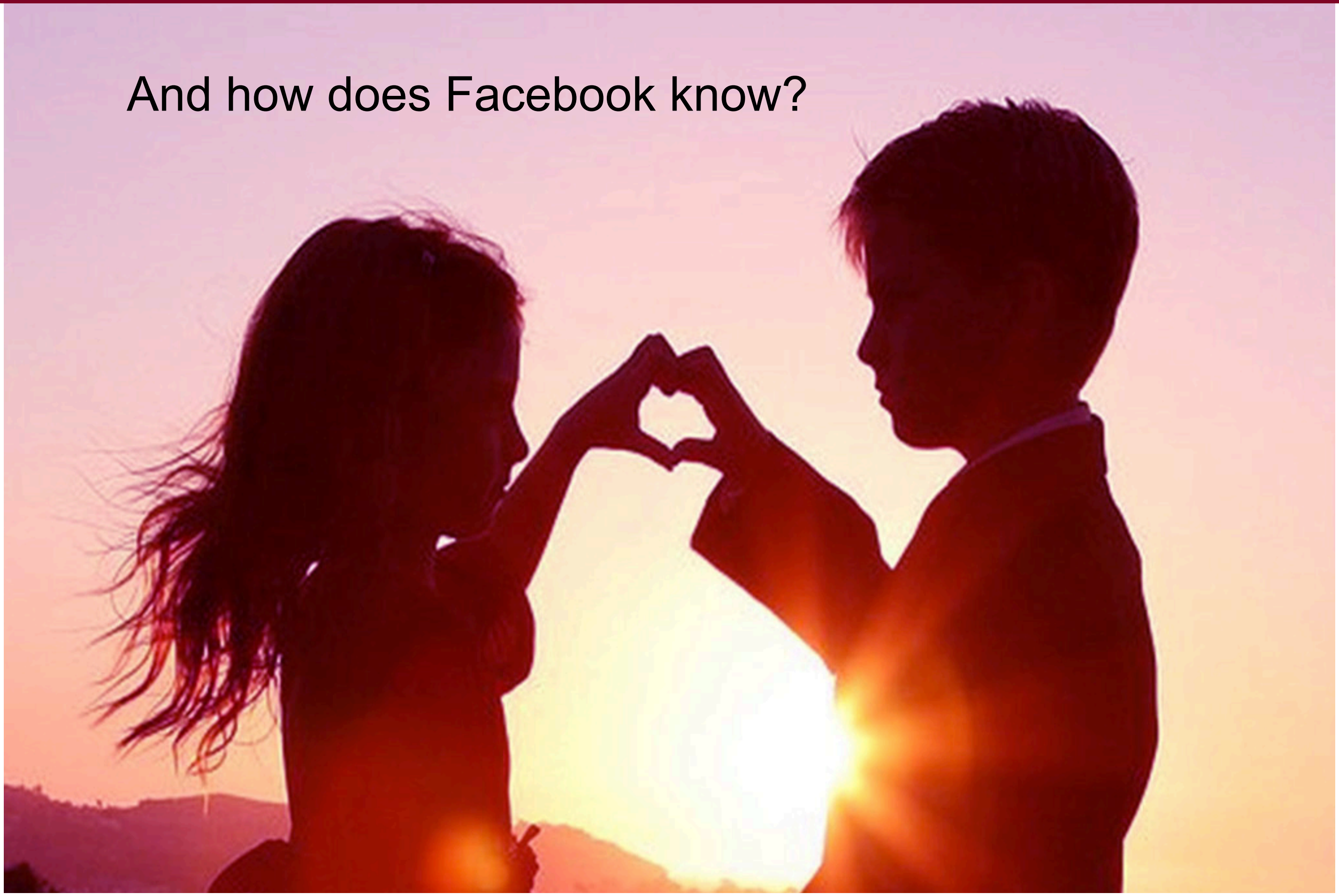
Beautiful mathematically

Used in everyday life (both JPEG and MP3)

Sweet history

Great practice with trees

Intro to Graphs: Who do You Love?



Tree Definition

Only One Parent

No Cycles


WHAT IF I TOLD YOU
THERE ARE NO "RULES"
MEMECREATOR.EU

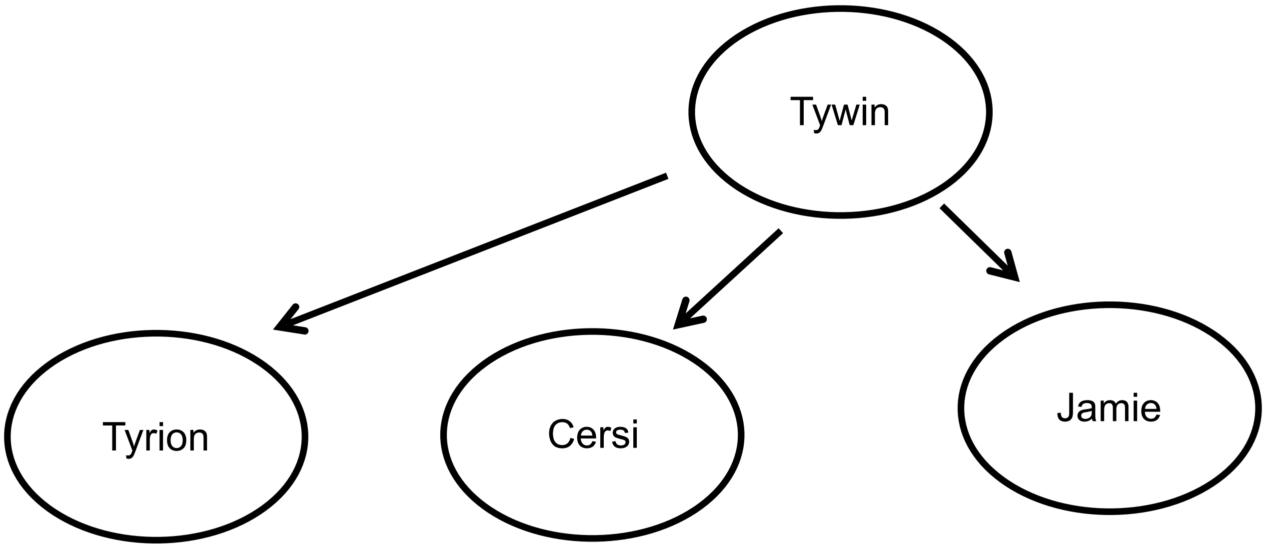
Graph Definition

A **graph** is a mathematical structure for representing relationships using nodes and edges.

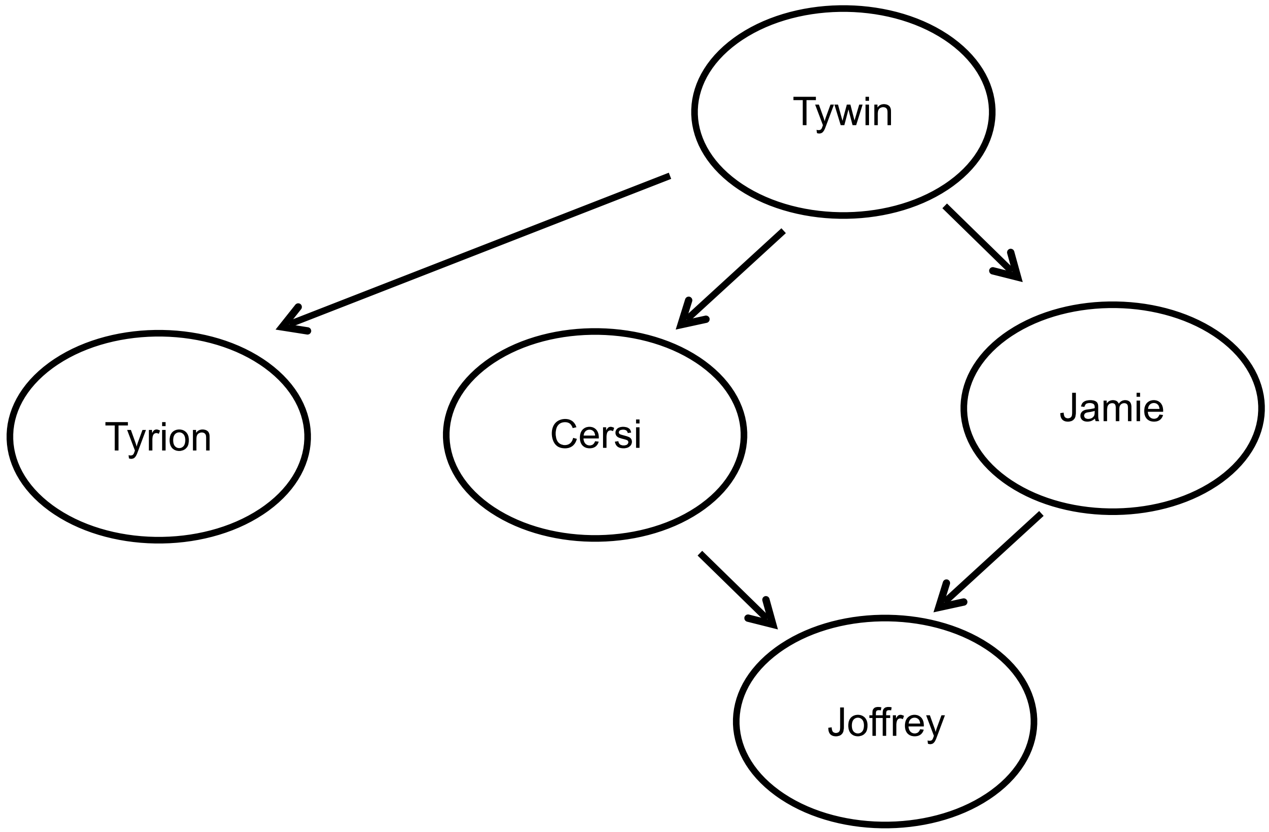
*Just like a tree without the rules


We can have a family tree?

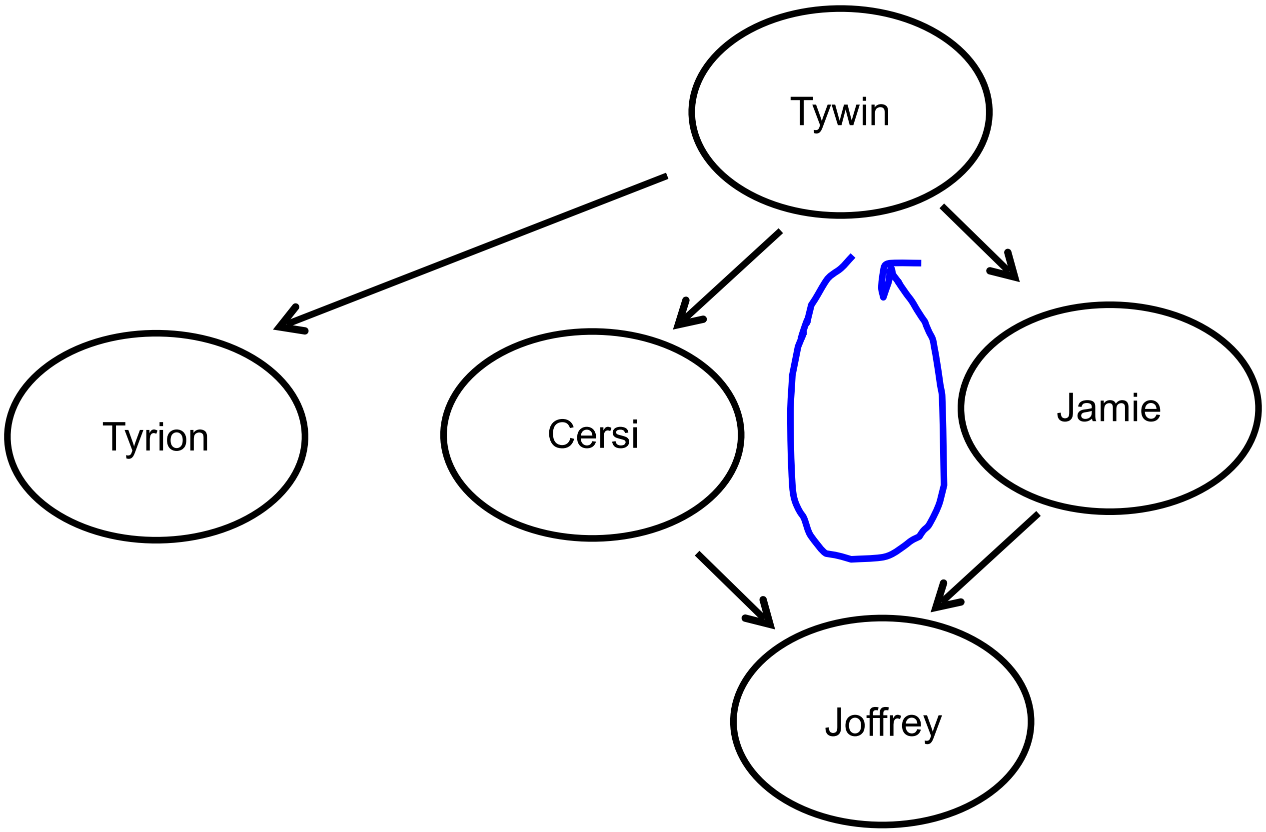
Family Tree



Not a Tree

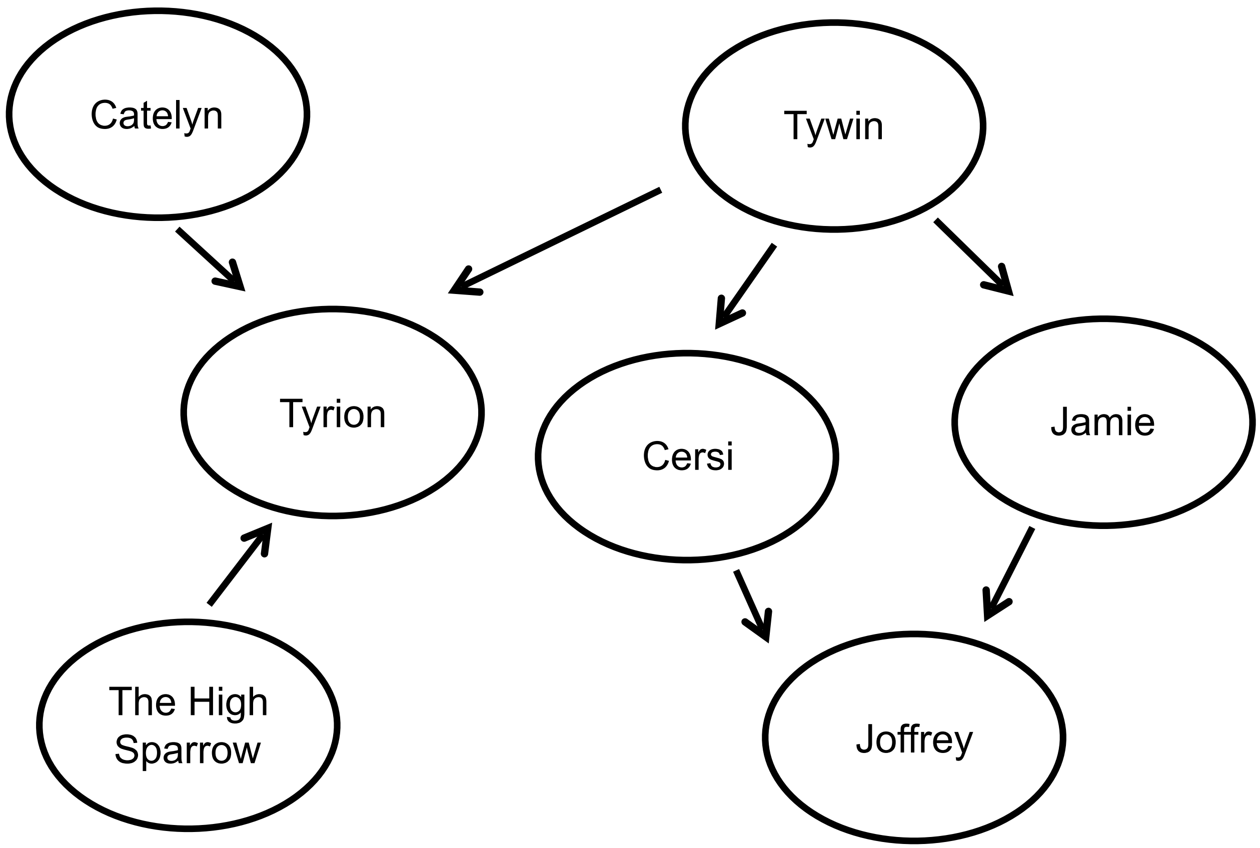


Not a Tree




graph
We can have a family ~~tree~~?

Graphs Don't Have Roots



Simple Graph

```
struct Node{
 string value;
 Vector<Edge *> edges;
};

 struct Edge{
  Node * start;
  Node * end;
 };

 struct Graph{
  Set<Node *> nodes;
  Set<Edge*> edges;
 };
```

Simple Graph

```
struct Node{
 string value;
 Vector<Edge *> edges;
};

struct Edge{
 Node * start;
 Node * end;
};
struct Graph{
 Set<Node *> nodes;
 Set<Edge*> edges;
};
```

We allow for more interesting edges

Simple Graph

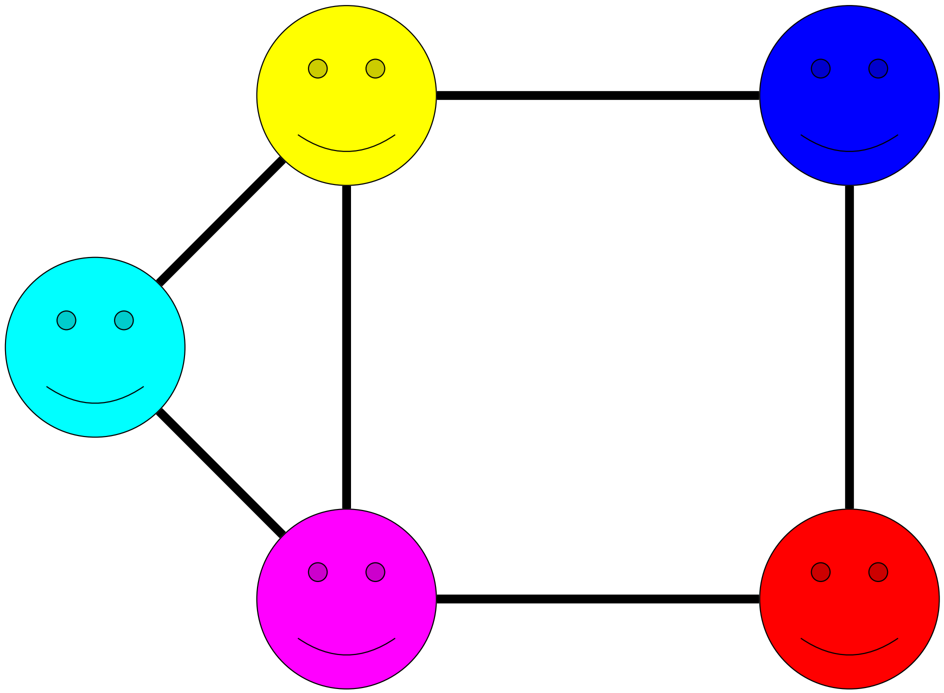
```
struct Node{
 string value;
 Vector<Edge *> edges;
};
```

```
struct Edge{
 Node * start;
 Node * end;
 double weight;
};
```

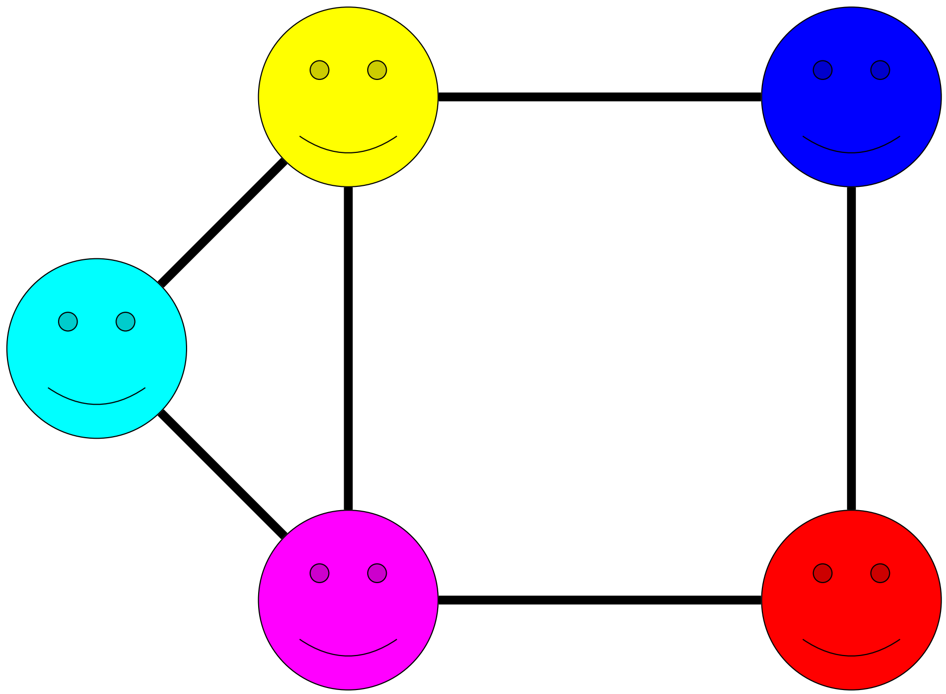
We allow for more interesting edges

```
struct Graph{
 Set<Node *> nodes;
 Set<Edge*> edges;
```

Simple Graph

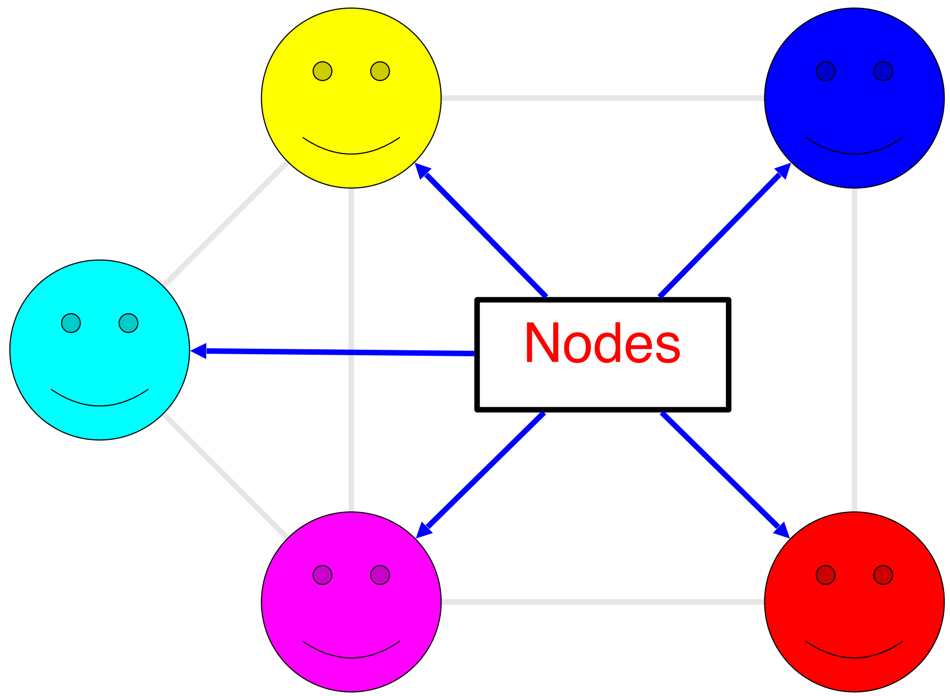


Simple Graph



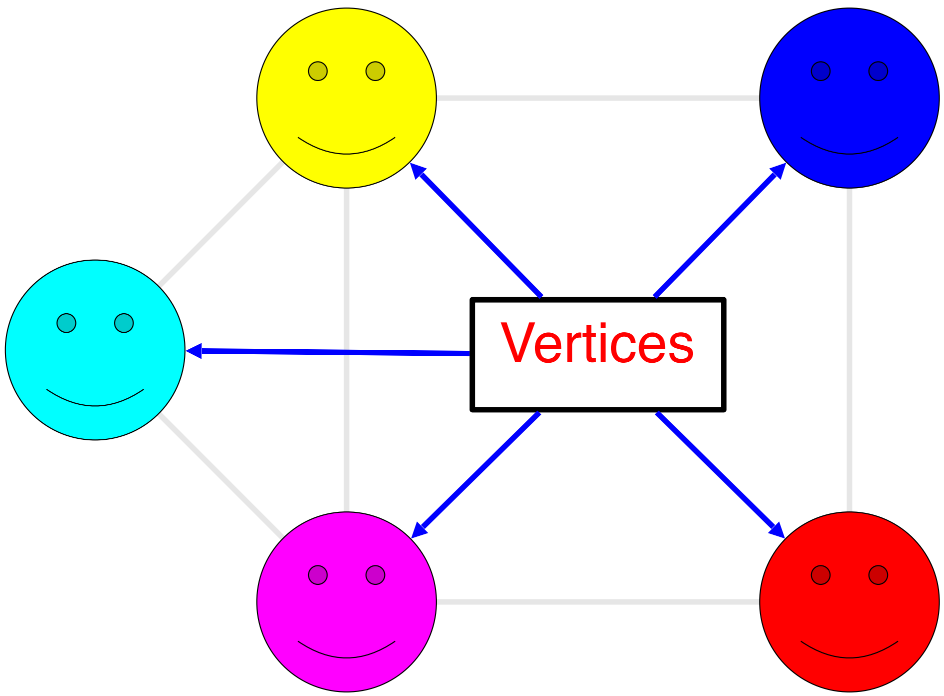
A graph consists of a set of **nodes** connected by **edges**.

Graph Nodes



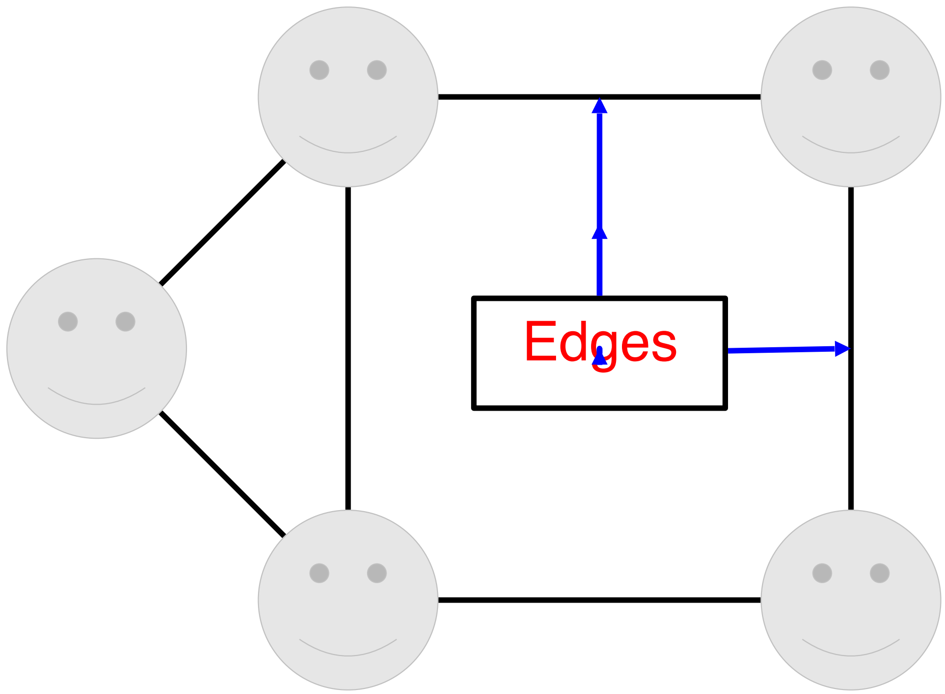
A graph consists of a set of **nodes** connected by **edges**.

Nodes are Also Called Vertices



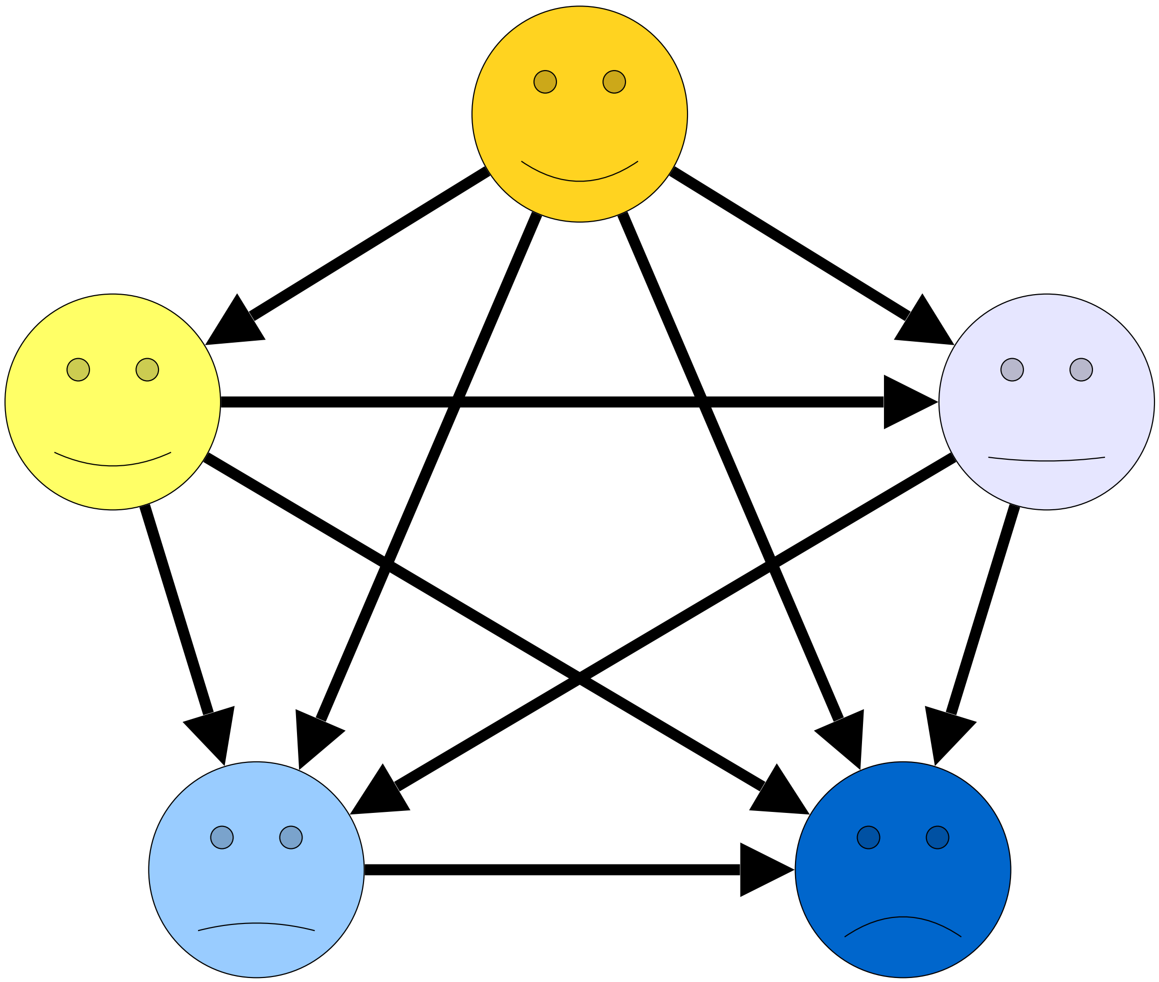
A graph consists of a set of **nodes** connected by **edges**.

Graph Edges

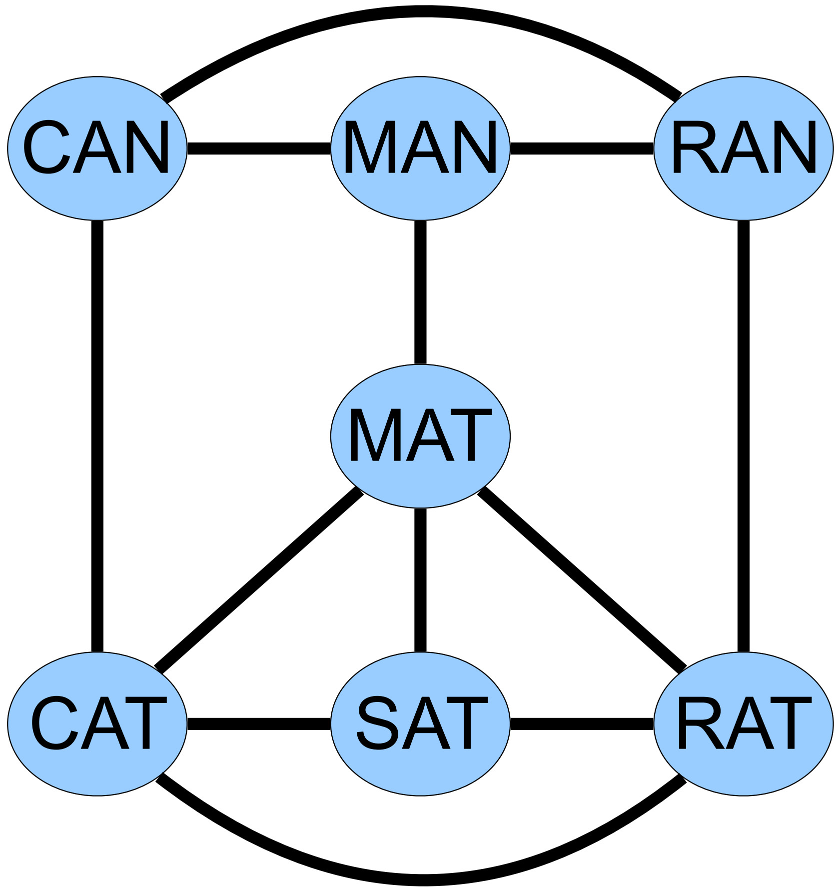


A graph consists of a set of **nodes** connected by **edges**.

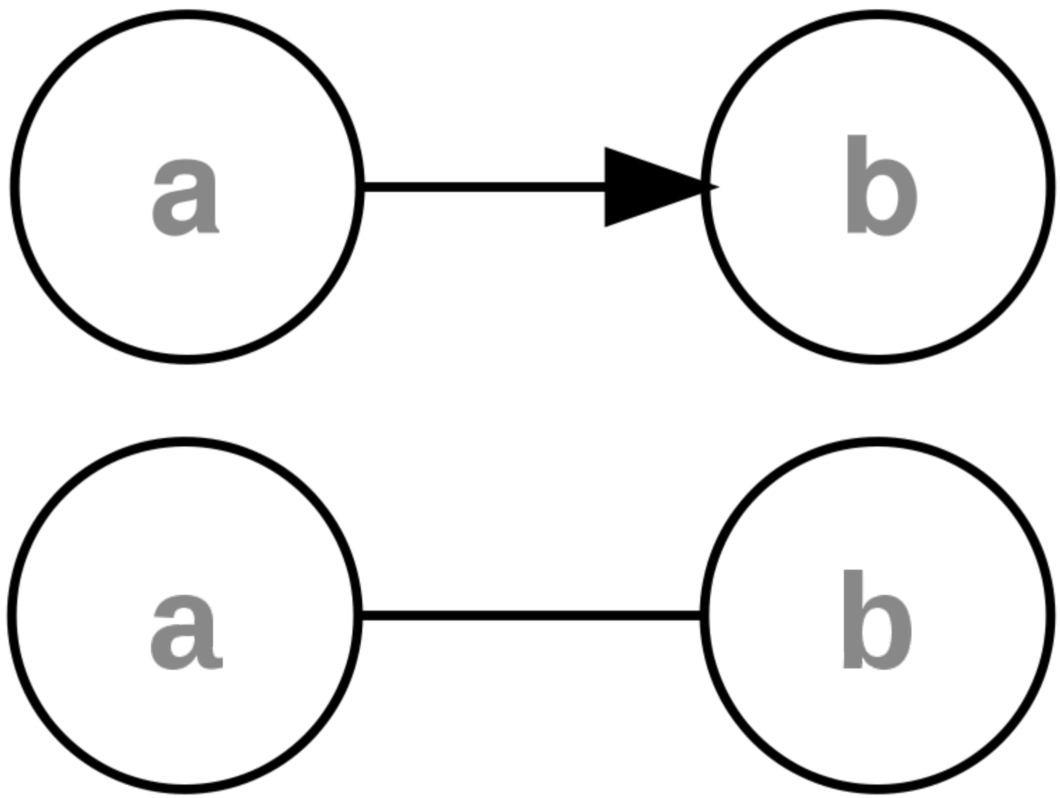
Directed Graph



Undirected Graph



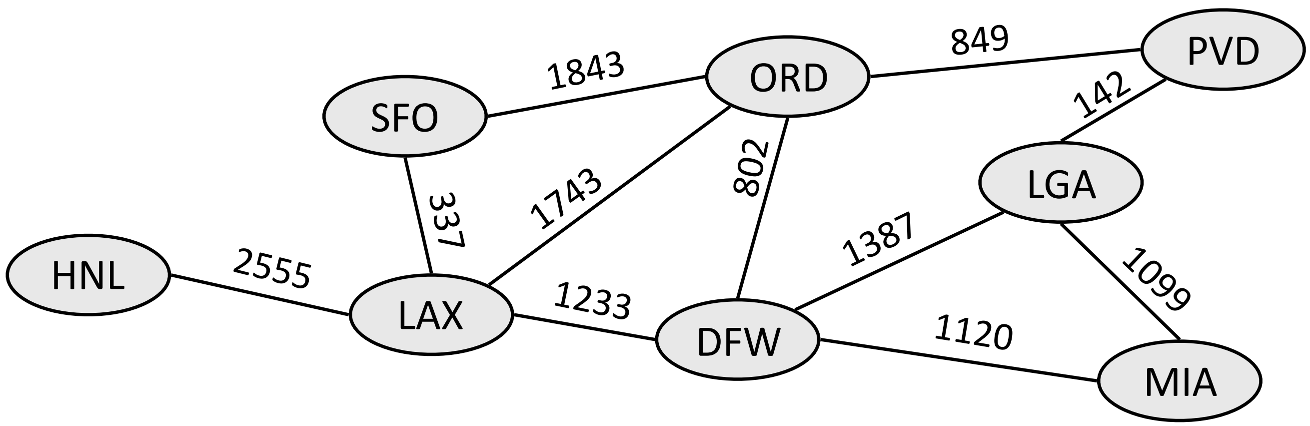
Directed vs Undirected



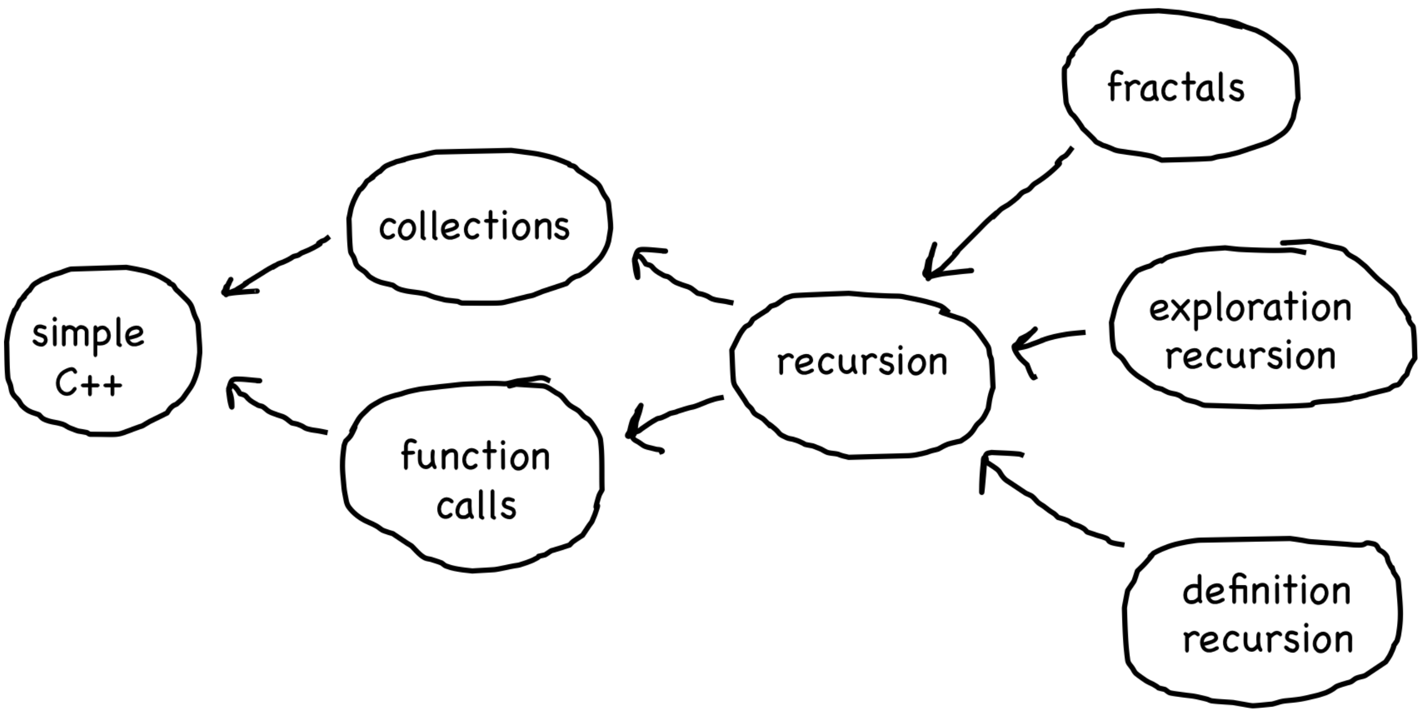
Weighted graphs

weight: Cost associated with a given edge.

example: graph of airline flights, weighted by miles between cities:

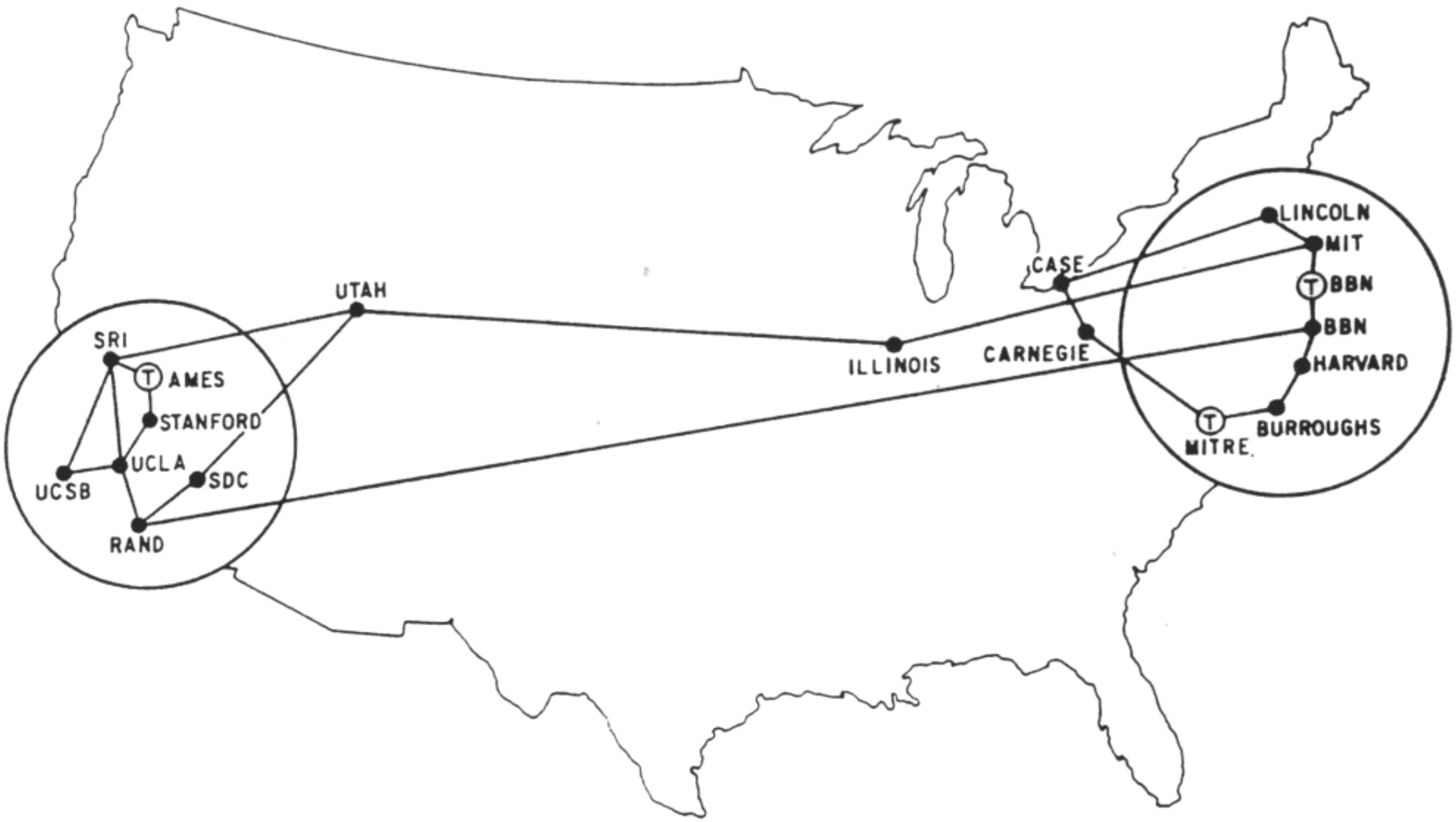


Prerequisite Graph



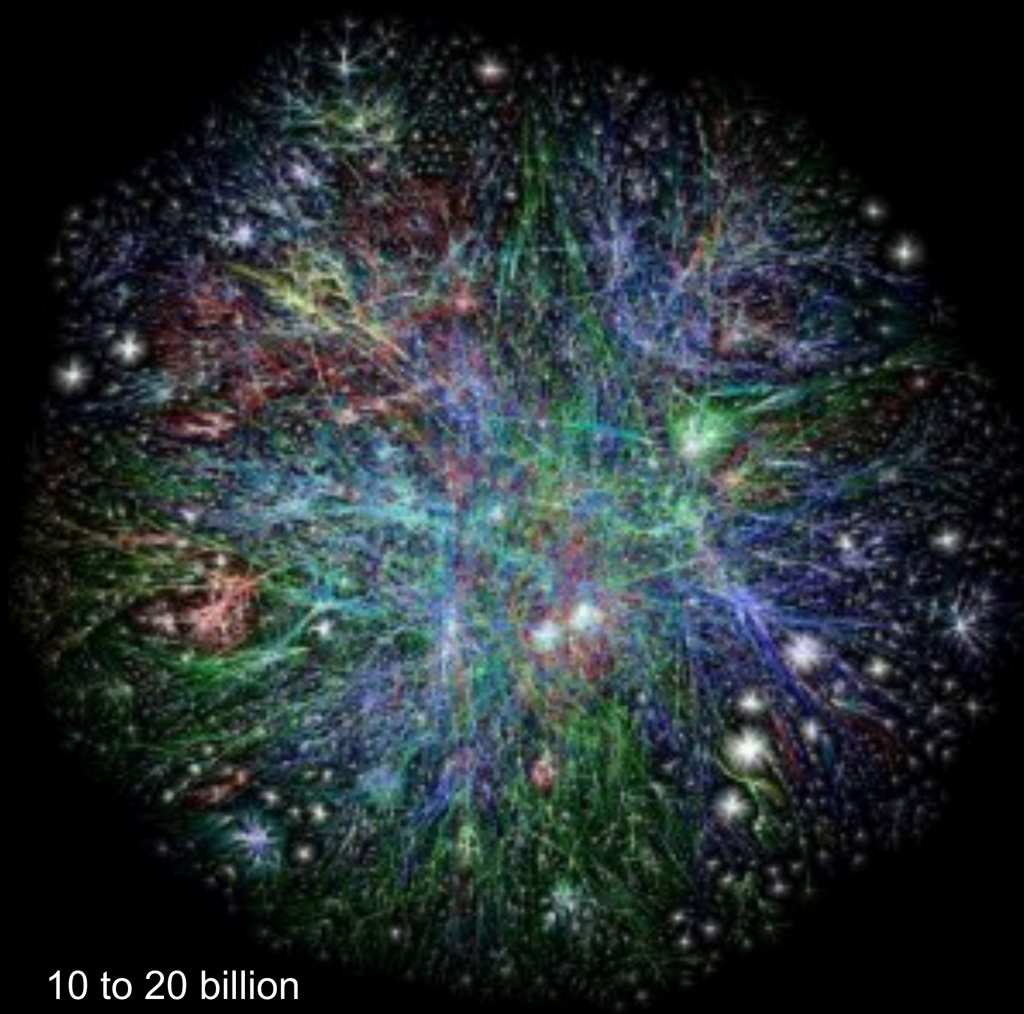
Social Network

The Internet



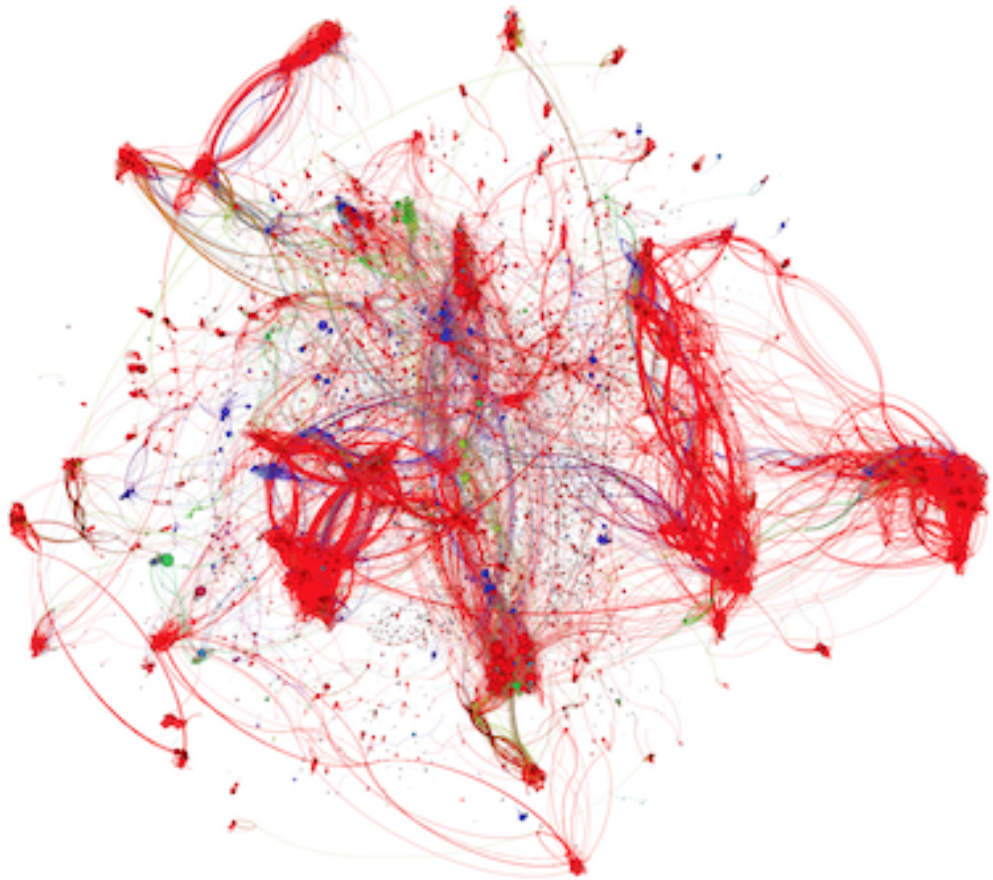
MAP 4 September 1971

The Internet



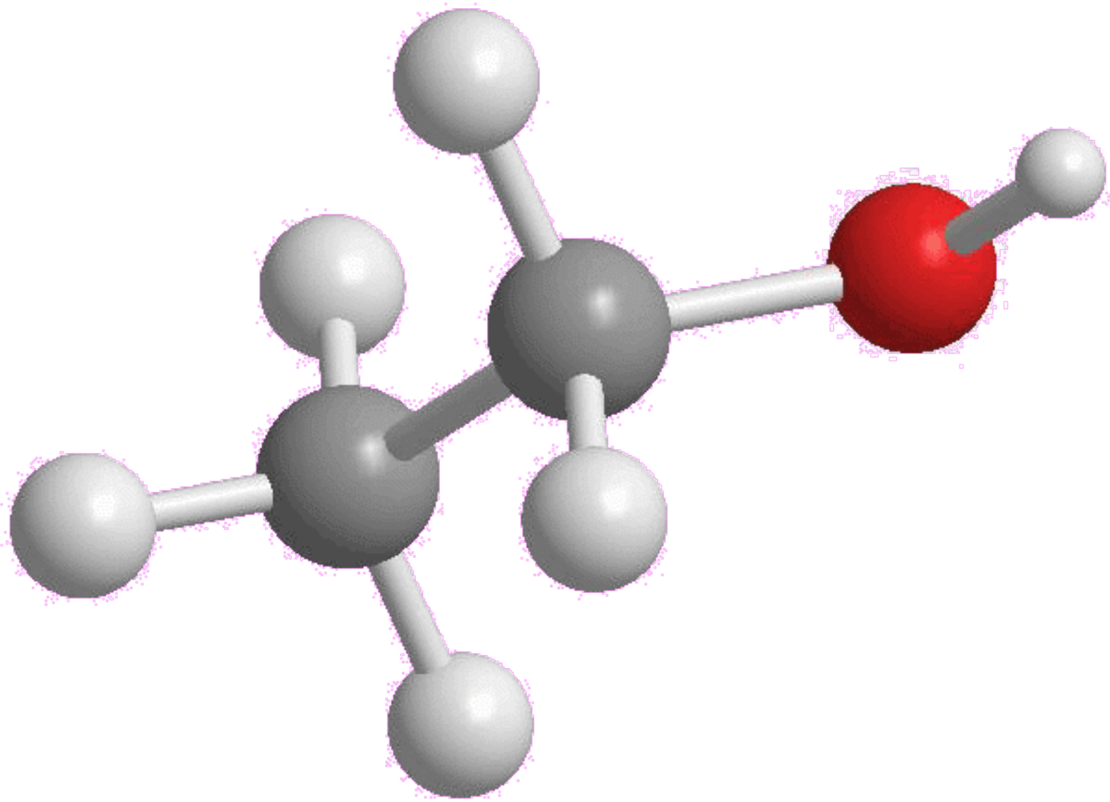
10 to 20 billion

CS Assignments



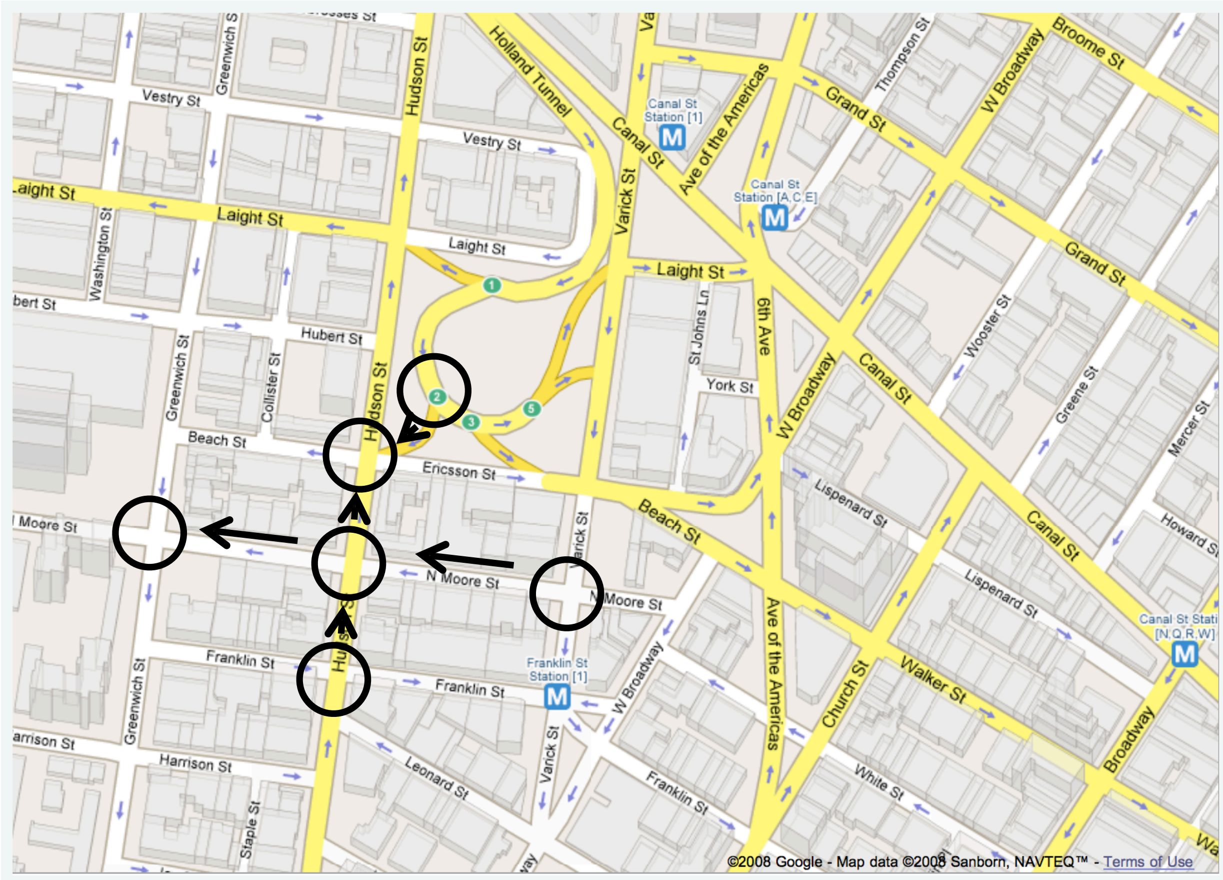
50,000 unique implementations of logistic regression in CS229

Chemical Bonds



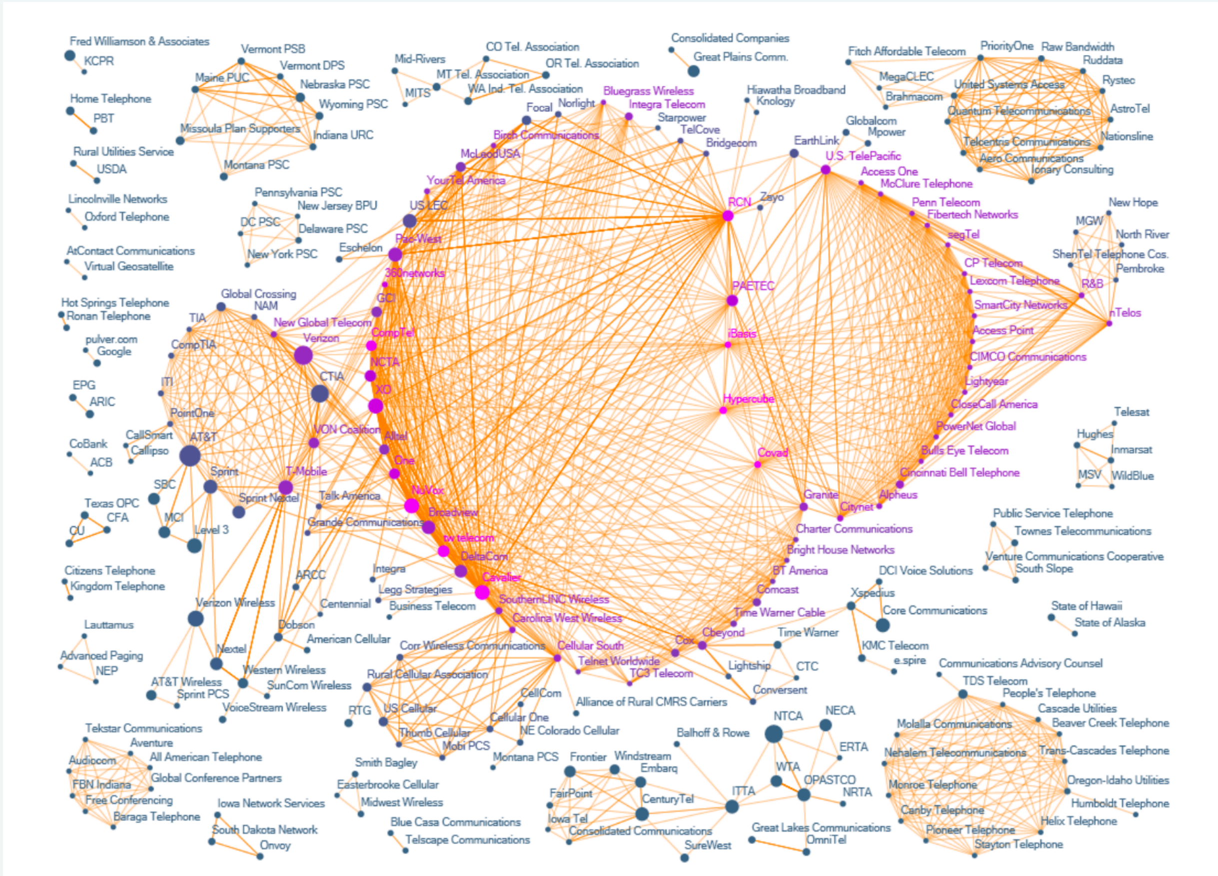
http://4.bp.blogspot.com/-xCtBJ8lKHqA/Tjm0BONWBRI/AAAAAAAAAK4/-mHrbAUOHHg/s1600/Ethanol2.gif

Road Map



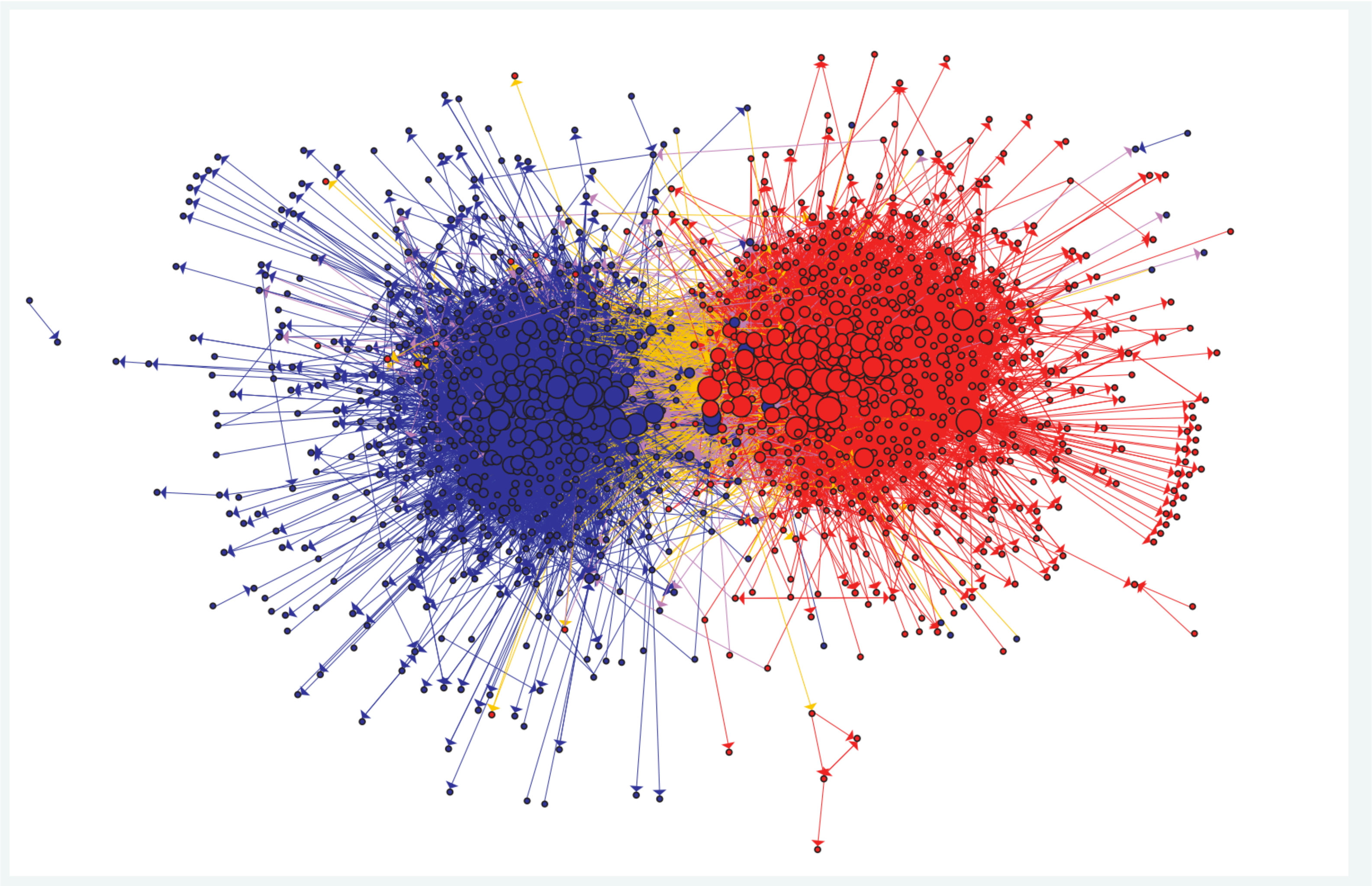
Vestry St
Hudson St
Holland Tunnel
Canal St
Ave of the Americas
Grand St
W Broadway
Broome St
Thompson St
Canal St Station [1]
Canal St Station [A,C,E]
Laight St
Varick St
Washington St
Greenwich St
Hubert St
Collister St
St Johns Ln
6th Ave
York St
Wooster St
Greene St
Mercer St
Beach St
Ericsson St
Lispenard St
Howard St
N Moore St
Franklin St
Franklin St Station [1]
Canal St Station [N,Q,R,W]
Walker St
Church St
Harrison St
Leonard St
White St
Staple St
Broadway
©2008 Google - Map data ©2008 Sanborn, NAVTEQ™ - Terms of Use

Corruption



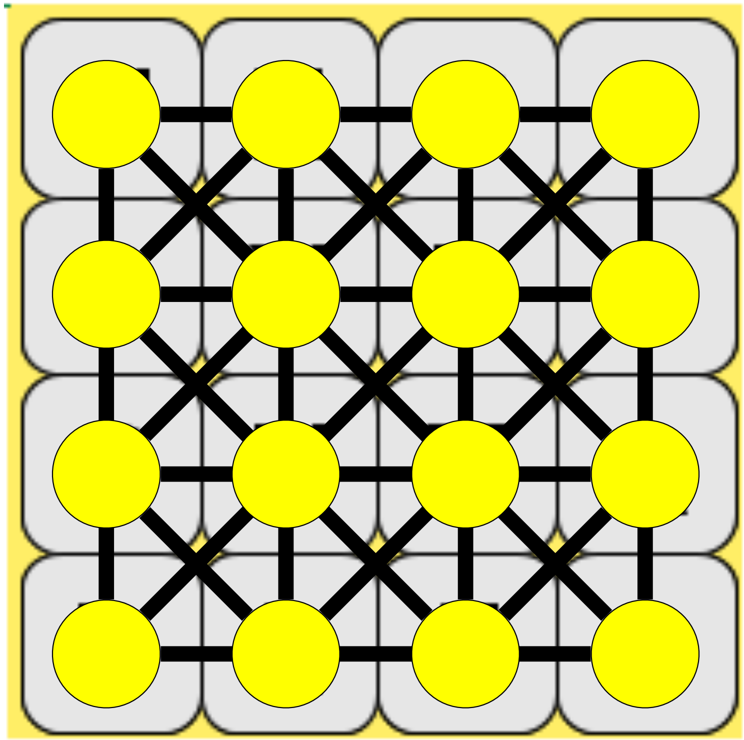
"The Evolution of FCC Lobbying Coalitions" by Pierre de Vries in JoSS Visualization Symposium 2010

Partisanship

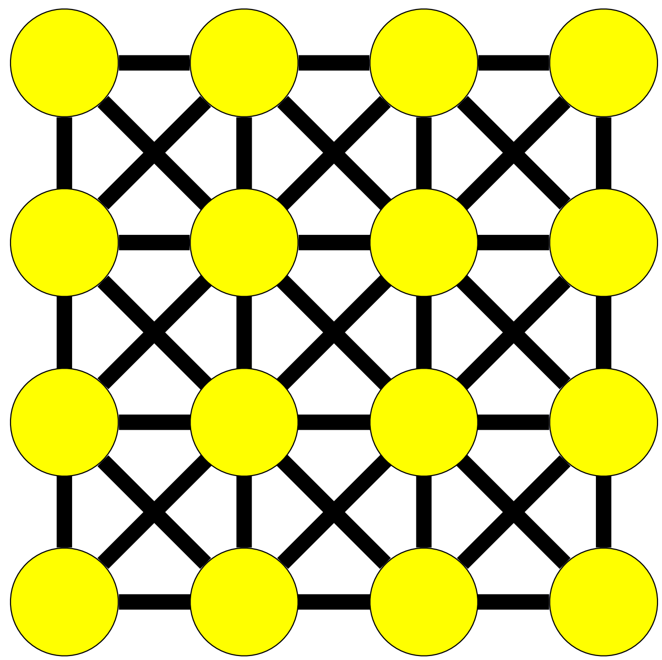


Boggle

Boggle



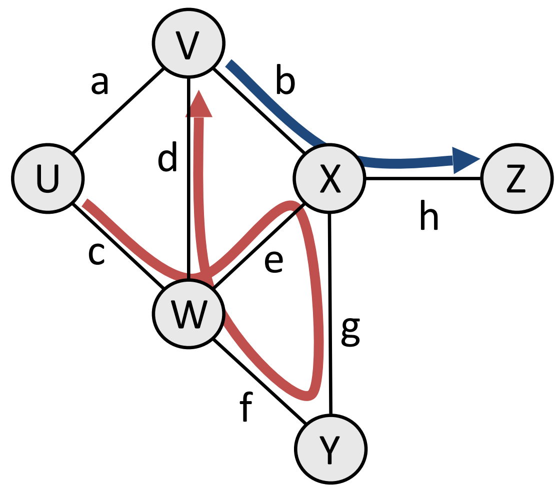
Boggle



Some terms:

Paths

- **path**: A path from vertex *a* to *b* is a sequence of edges that can be followed starting from *a* to reach *b*.
  - can be represented as vertices visited, or edges taken
  - example, one path from *V* to *Z*: {b, h} or {V, X, Z}
  - What are two paths from U to Y?



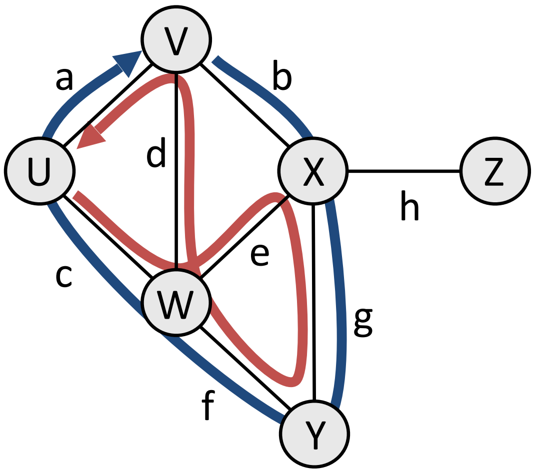
- **path length**: Number of vertices or edges contained in the path.

- **neighbor** or **adjacent***:* Two vertices connected directly by an edge.
  - example: V and X

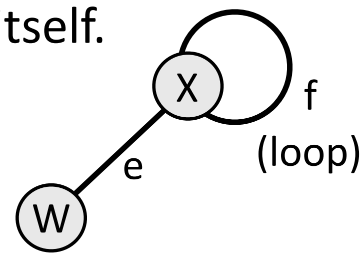
Loops and cycles

- **cycle**: A path that begins and ends at the same node.
  - example: {b, g, f, c, a} or {V, X, Y, W, U, V}.
  - example: {c, d, a} or {U, W, V, U}.

  - **acyclic graph**: One that does not contain any cycles.



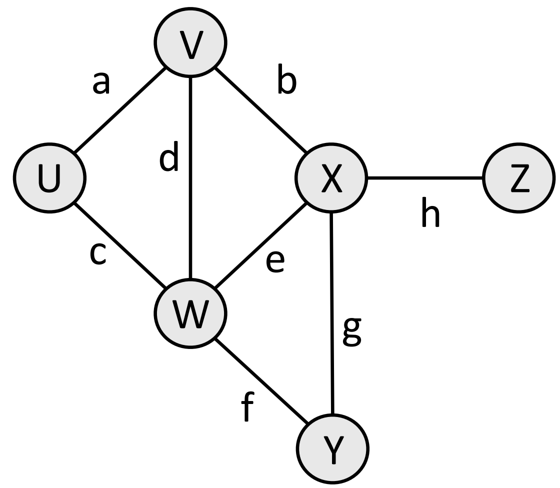
- **loop**: An edge directly from a node to itself.



  - Many graphs don't allow loops.

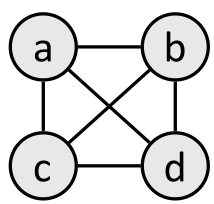
Reachability, connectedness

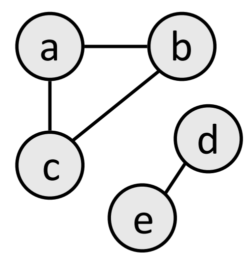
- **reachable**: Vertex *a* is *reachable* from *b* if a path exists from *a* to *b*.



- **connected**: A graph is *connected* if every vertex is reachable from every other.

- **complete**: If every vertex has a direct edge to every other.





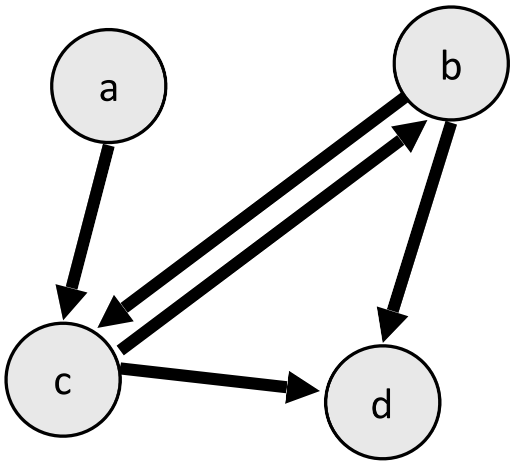
Stanford BasicGraph

The Stanford C++ library includes a `BasicGraph` class.

– Based on an older library class named Graph

You can construct a graph and add vertices/edges:

```
#include "basicgraph.h"
...
BasicGraph graph;
graph.addVertex("a");
graph.addVertex("b");
graph.addVertex("c");
graph.addVertex("d");
graph.addEdge("a", "c");
graph.addEdge("b", "c");
graph.addEdge("c", "b");
graph.addEdge("b", "d");
graph.addEdge("c", "d");
```



BasicGraph members

```
#include "basicgraph.h"    // a directed, weighted graph
```

`g.addEdge(v1, v2);`	adds an edge between two vertexes
`g.addVertex(name);`	adds a vertex to the graph
`g.clear();`	removes all vertexes/edges from the graph
`g.getEdgeSet()` `g.getEdgeSet(v)`	returns all edges, or all edges that start at ***v***, as a `Set` of pointers
`g.getNeighbors(v)`	returns a set of all vertices that ***v*** has an edge to
`g.getVertex(name)`	returns pointer to vertex with the given name
`g.getVertexSet()`	returns a set of all vertexes
`g.isNeighbor(v1, v2)`	returns `true` if there is an edge from vertex ***v1*** to ***v2***
`g.isEmpty()`	returns `true` if queue contains no vertexes/edges
`g.removeEdge(v1, v2);`	removes an edge from the graph
`g.removeVertex(name);`	removes a vertex from the graph
`g.size()`	returns the number of vertexes in the graph
`g.toString()`	returns a string such as `"{a, b, c, a -> b}"`

BasicGraph members

```
#include "basicgraph.h"    // a directed, weighted graph
```

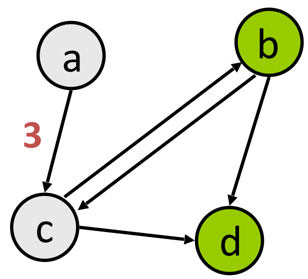
g`.addEdge(`***v1***`, `***v2***`);`	adds an edge between two vertexes
g`.addVertex(`***name***`);`	adds a vertex to the graph
g`.clear();`	removes all vertexes/edges from the graph
g`.getEdgeSet()` ***g***`.getEdgeSet(`***v***`)`	returns all edges, or all edges that start at ***v***, as a `Set` of pointers
g`.getNeighbors(`***v***`)`	returns a set of all vertices that ***v*** has an edge to
g`.getVertex(`***name***`)`	returns pointer to vertex with the given name
g`.getVertexSet()`	returns a set of all vertexes
g`.isNeighbor(`***v1***`, `***v2***`)`	returns `true` if there is an edge from vertex ***v1*** to ***v2***
g`.isEmpty()`	returns `true` if queue contains no vertexes/edges
g`.removeEdge(`***v1***`, `***v2***`);`	removes an edge from the graph
g`.removeVertex(`***name***`);`	removes a vertex from the graph
g`.size()`	returns the number of vertexes in the graph
g`.toString()`	returns a string such as `"{a, b, c, a -> b}"`

Using BasicGraph

The graph stores a struct of information about each vertex/edge:

```
struct Vertex {
    string name;
    Set<Edge*> edges;
    double cost;
    // other stuff
};
```

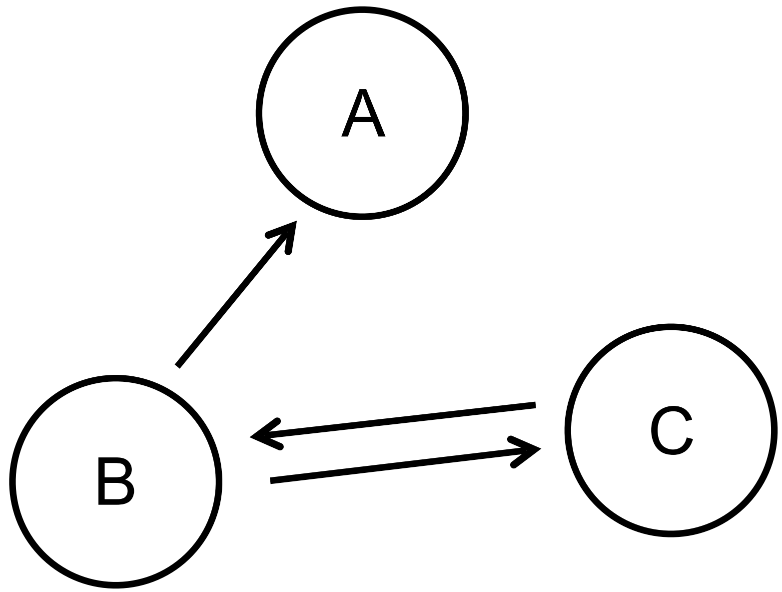
```
struct Edge {
    Vertex* start;
    Vertex* finish;
    double weight;
// other stuff
};
```



You can use these to help implement graph algorithms:

```
Vertex * vertC = graph.getVertex("c");
Edge * edgeAC = graph.getEdge("a", "c");
```

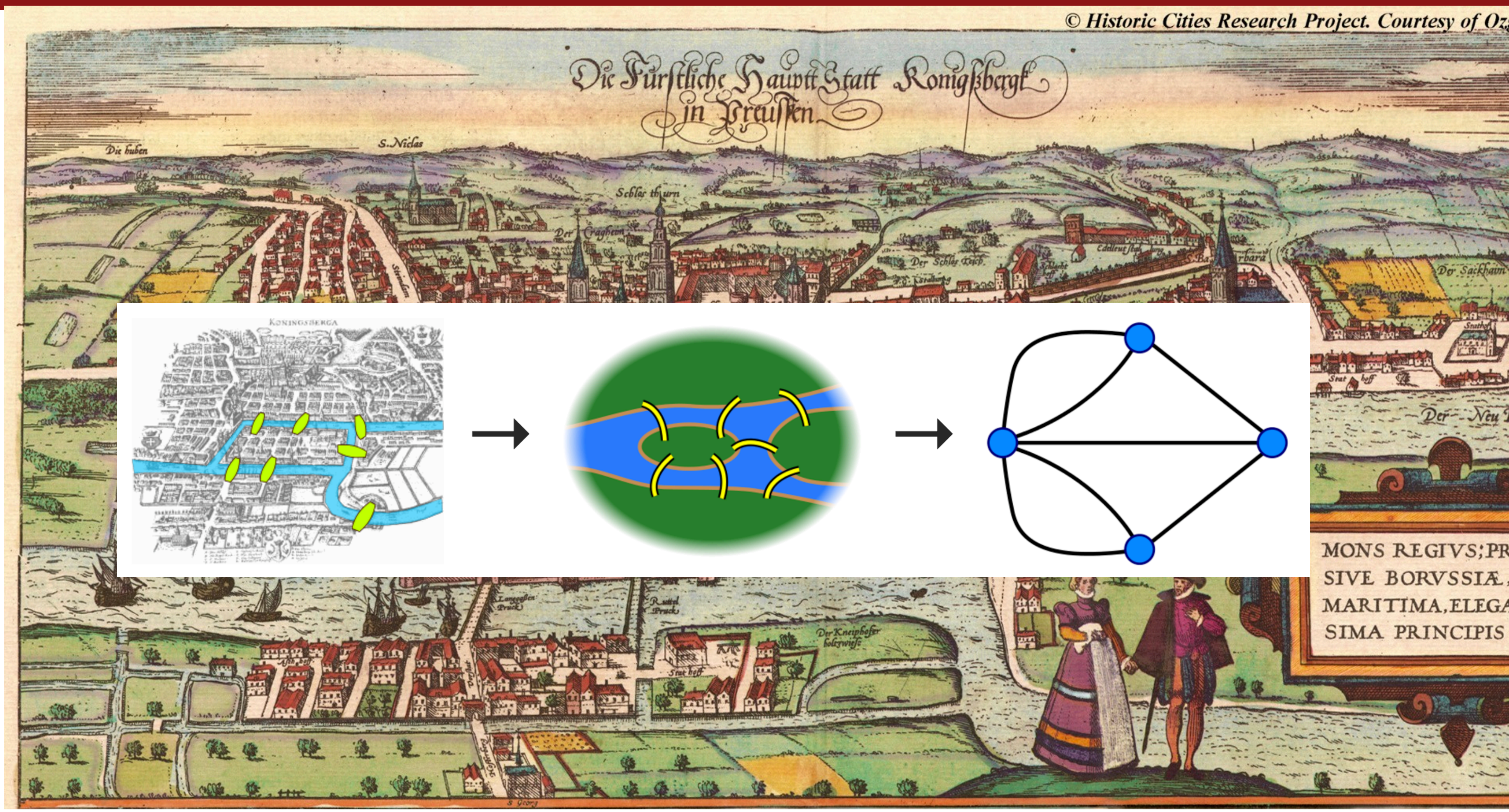
Our First Graph



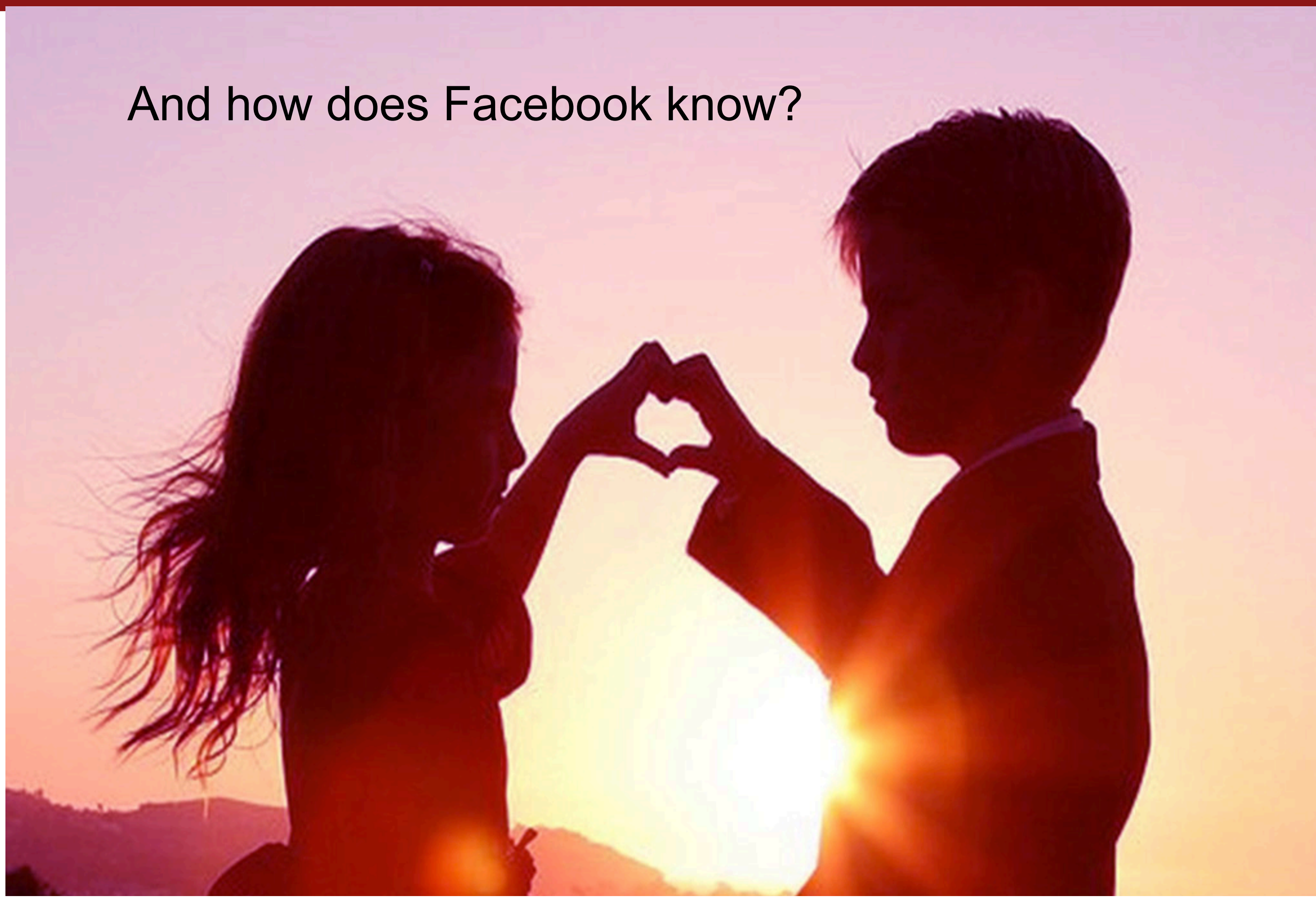
There are other representations…

… this is the one we are going to use.

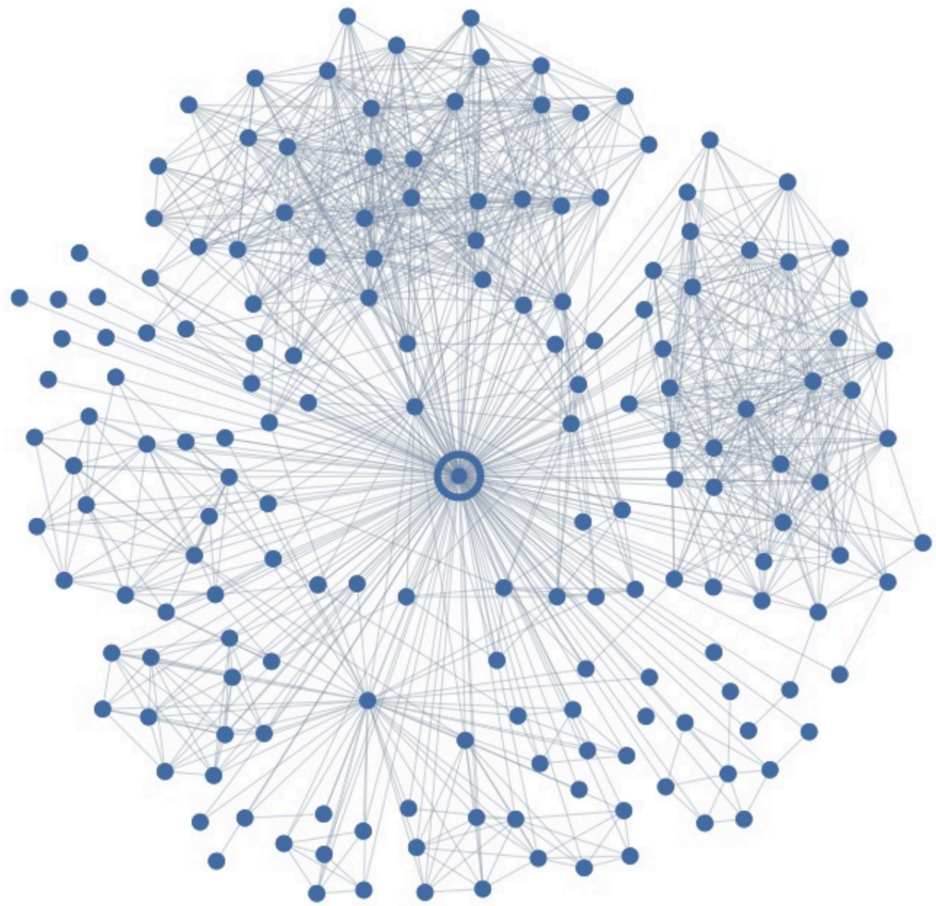
Algorithms


© Historic Cities Research Project. Courtesy of Oz
S.Niclas
MONS REGIVS;
SIVE BORVSSIÆ
SIMA PRINCIPIS

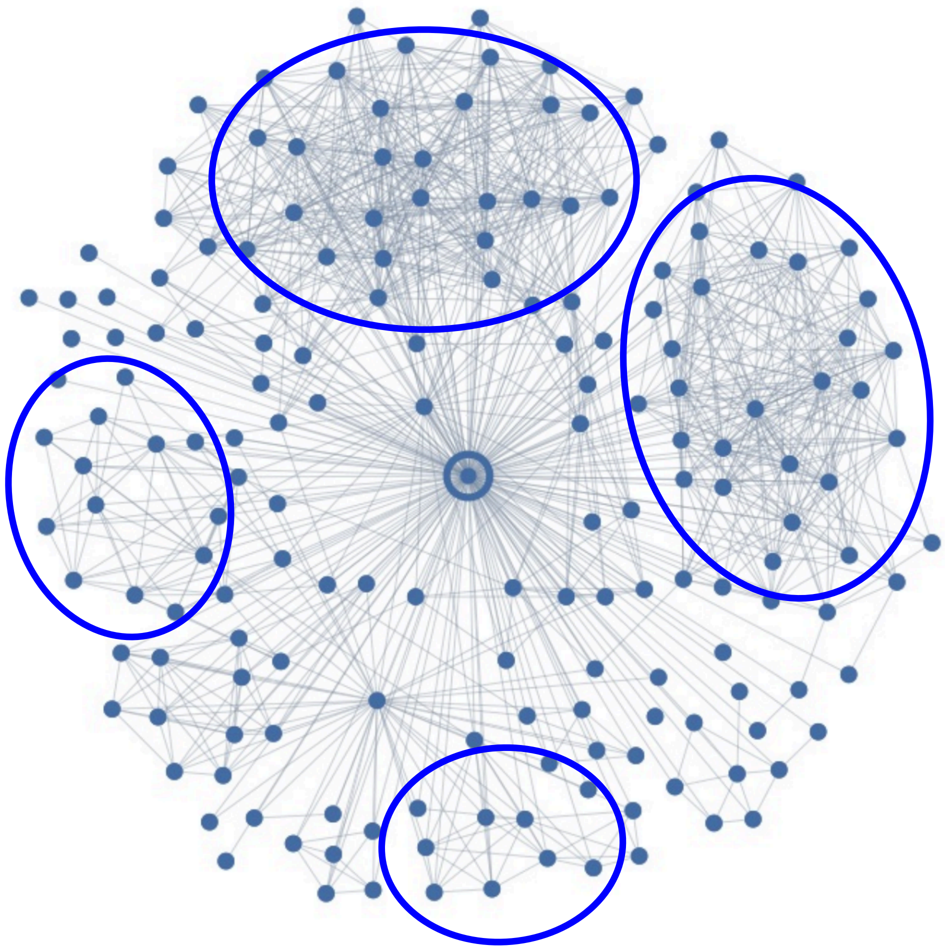
Who Do You Love



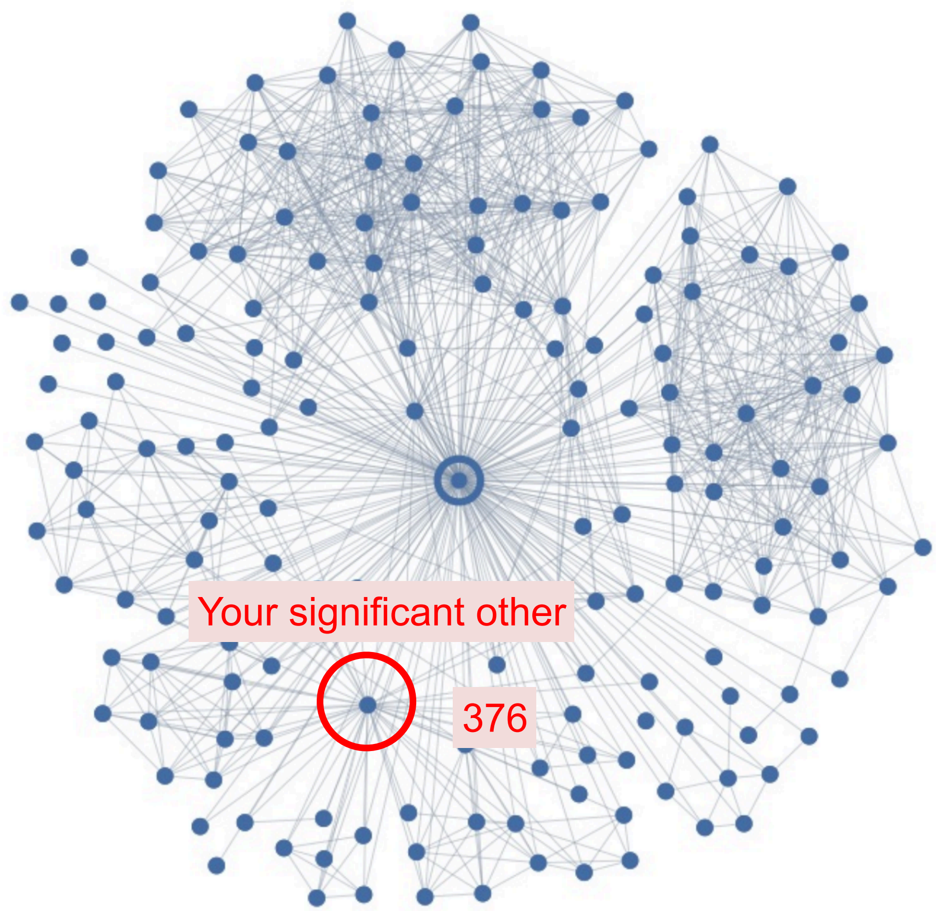
Ego Graph



Maybe I Love These People?



But I Actually Love This Person



Romance and Dispersion

Romantic Partnerships and the Dispersion of Social Ties: A Network Analysis of Relationship Status on Facebook

Lars Backstrom
Facebook Inc.

Jon Kleinberg
Cornell University

ABSTRACT

A crucial task in the analysis of on-line social-networking systems is to identify important people — those linked by strong social ties — within an individual's network neighborhood. Here we investigate this question for a particular category of strong ties, those involving spouses or romantic partners. We organize our analysis around a basic question: given all the connections among a person's friends, can you recognize his or her romantic partner from the network structure alone? Using data from a large sample of Facebook users, we find that this task can be accomplished with high accuracy, but doing so requires the development of a new measure of tie strength that we term 'dispersion' — the extent to which two people's mutual friends are not themselves well-connected. The results offer methods for identifying types of structurally significant people in on-line applications, and suggest a potential expansion of existing theories of tie strength.

Categories and Subject Descriptors: H.2.8 [**Database Management**]: Database applications—*Data mining*
Keywords: Social Networks; Romantic Relationships.

they see from friends [1], and organizing their neighborhood into conceptually coherent groups [23, 25].

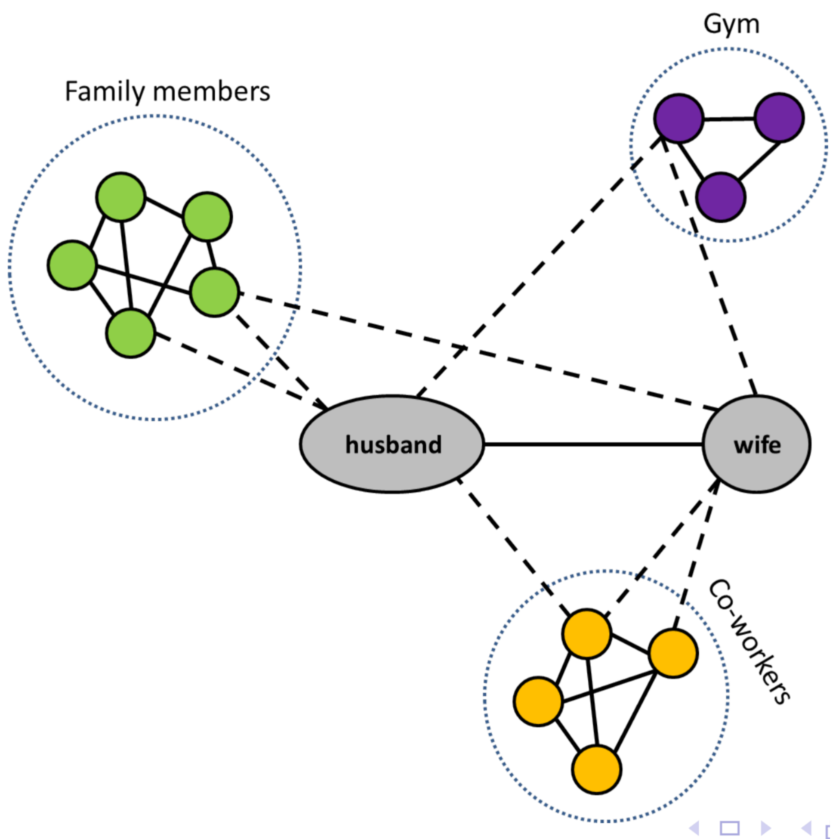
Tie Strength.
Tie strength forms an important dimension along which to characterize a person's links to their network neighbors. Tie strength informally refers to the 'closeness' of a friendship; it captures a spectrum that ranges from strong ties with close friends to weak ties with more distant acquaintances. An active line of research reaching back to foundational work in sociology has studied the relationship between the strengths of ties and their structural role in the underlying social network [15]. Strong ties are typically 'embedded' in the network, surrounded by a large number of mutual friends [6, 16], and often involving large amounts of shared time together [22] and extensive interaction [17]. Weak ties, in contrast, often involve few mutual friends and can serve as 'bridges' to diverse parts of the network, providing access to novel information [5, 15].

A fundamental question connected to our und...
strong ties is to identify the m...
person's social network ...

October 2013

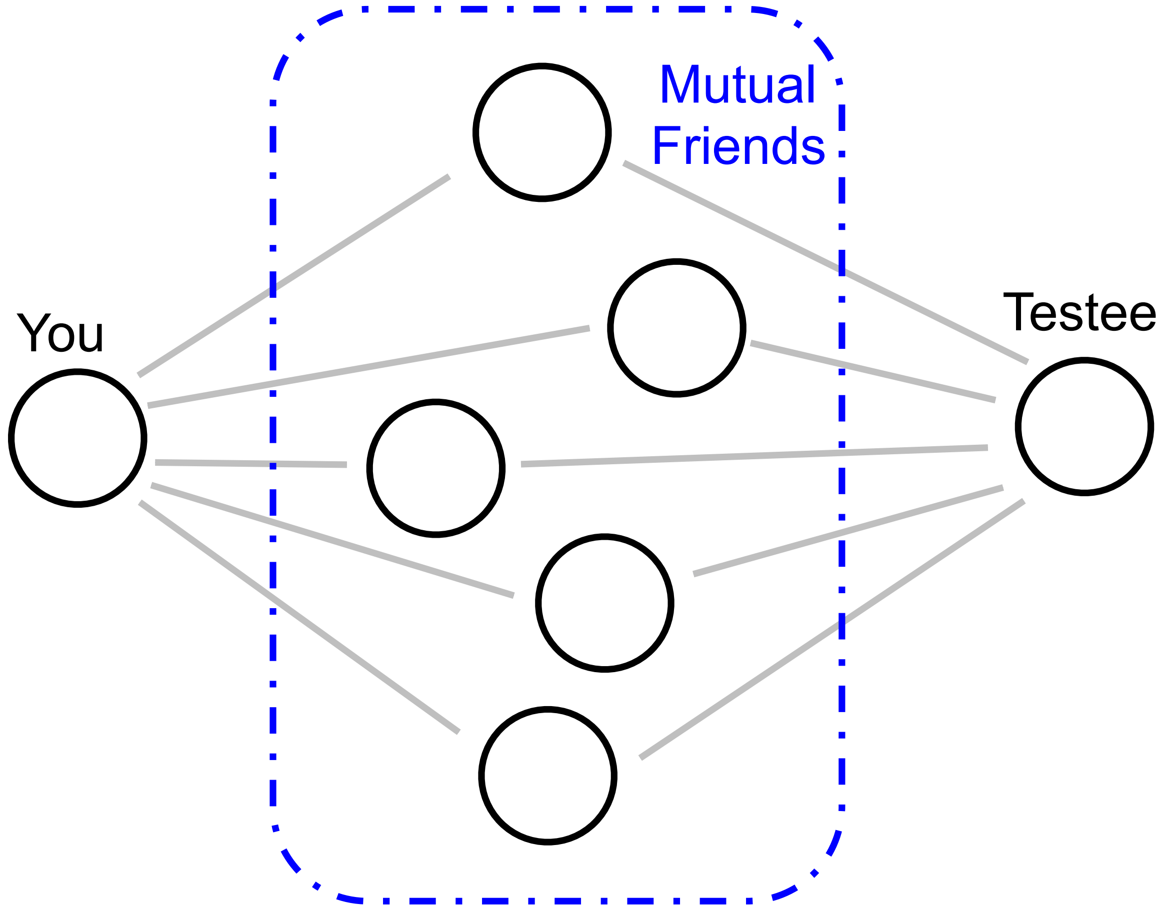
http://arxiv.org/pdf/1310.6753v1.pdf

Dispersion Insight



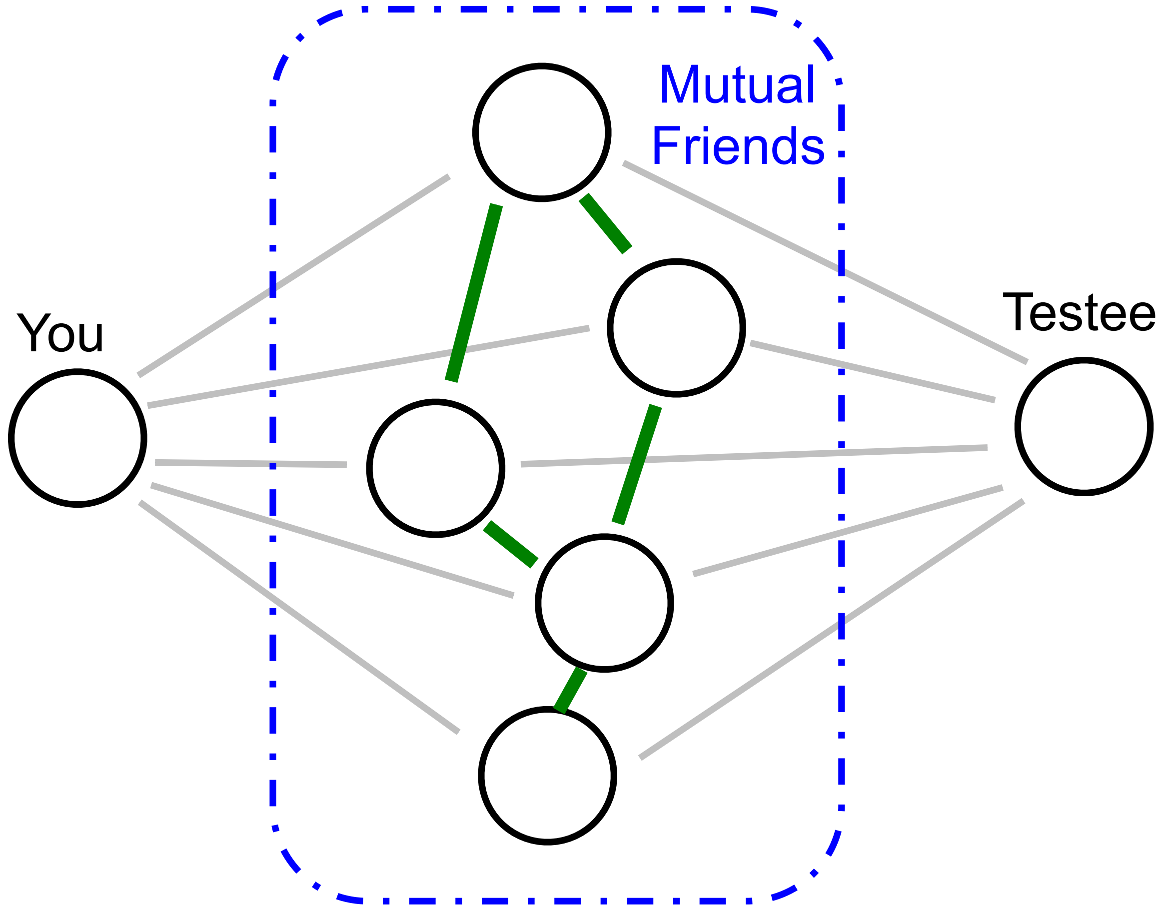
Dispersion: The extent to which two people's mutual friends are not directly connected

Dispersion



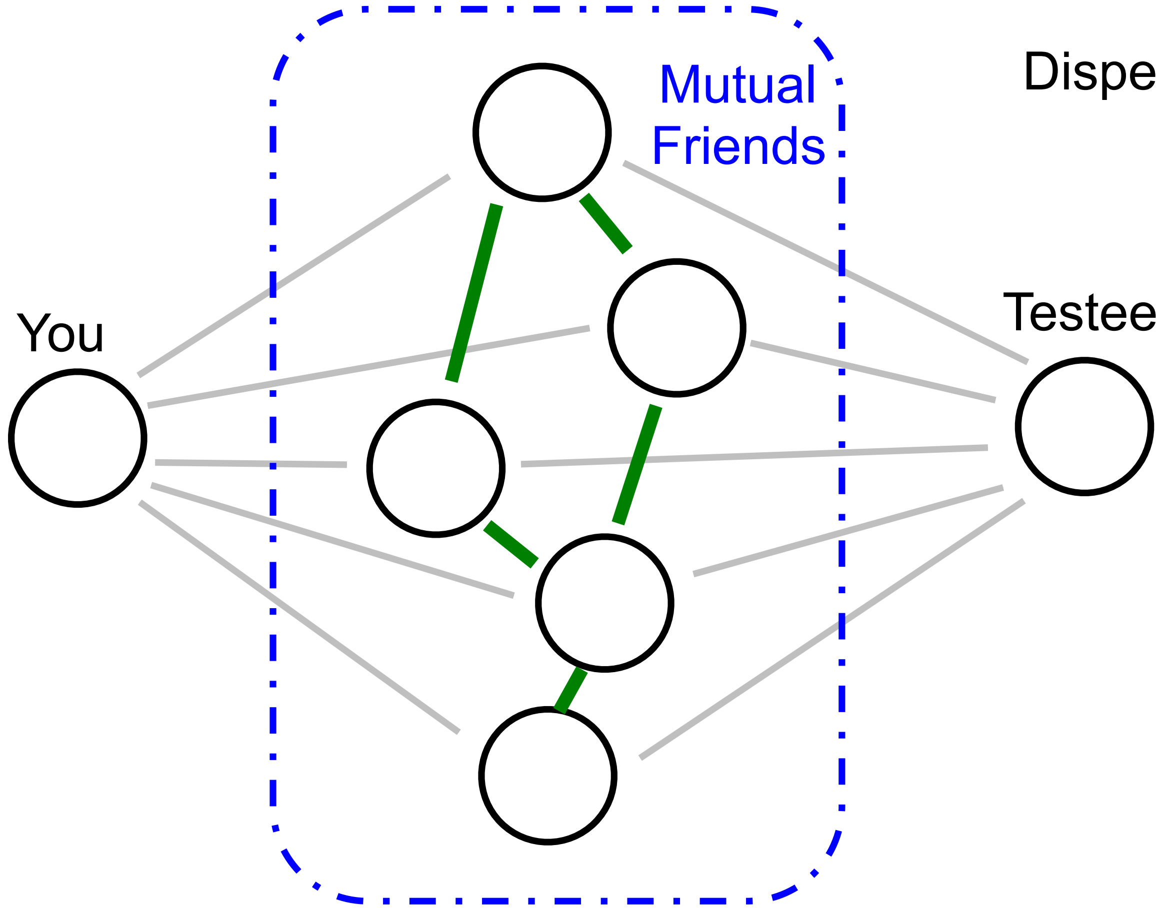
Dispersion: The extent to which two people's mutual friends are not directly connected

Dispersion



Dispersion: The extent to which two people's mutual friends are not directly connected

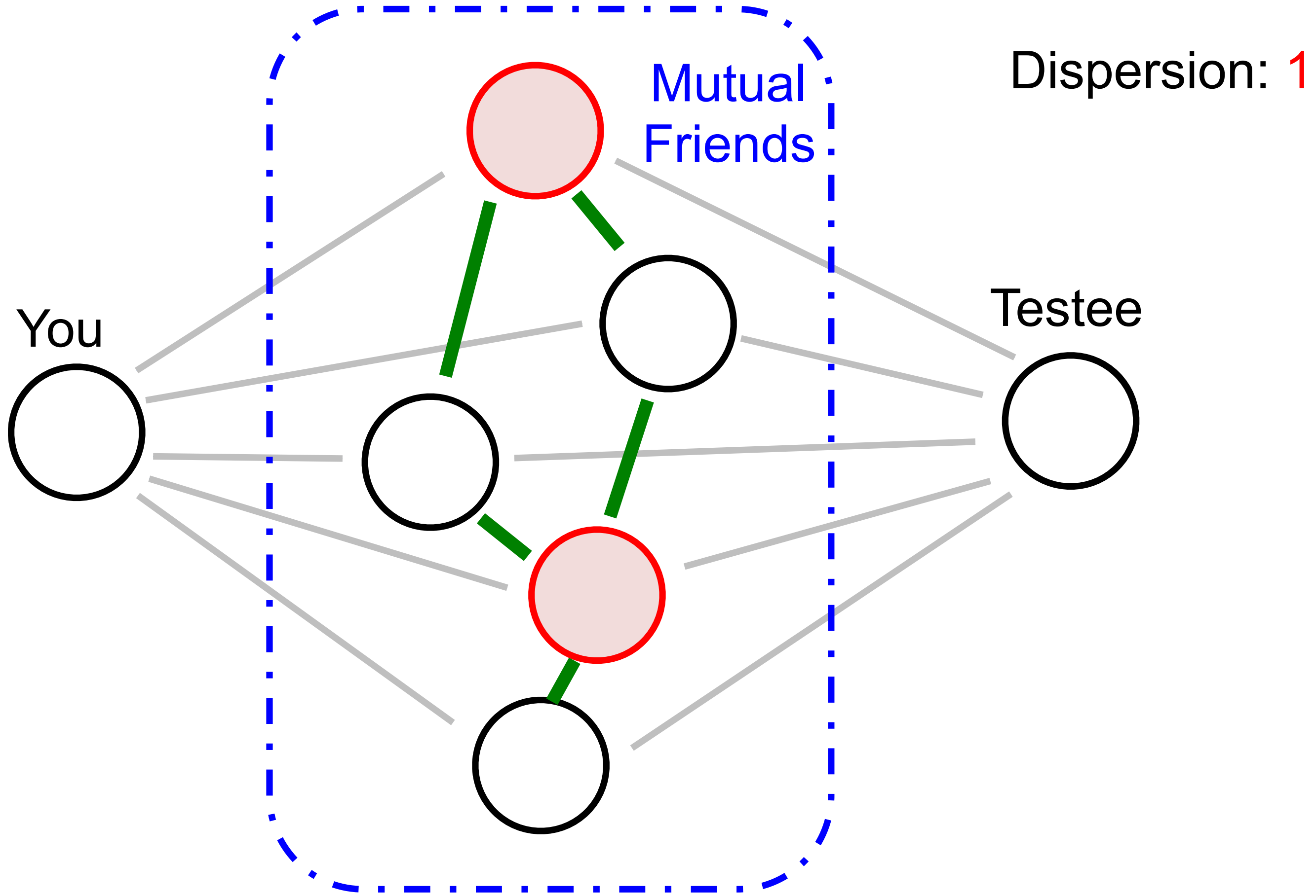
Dispersion



Dispersion: 0

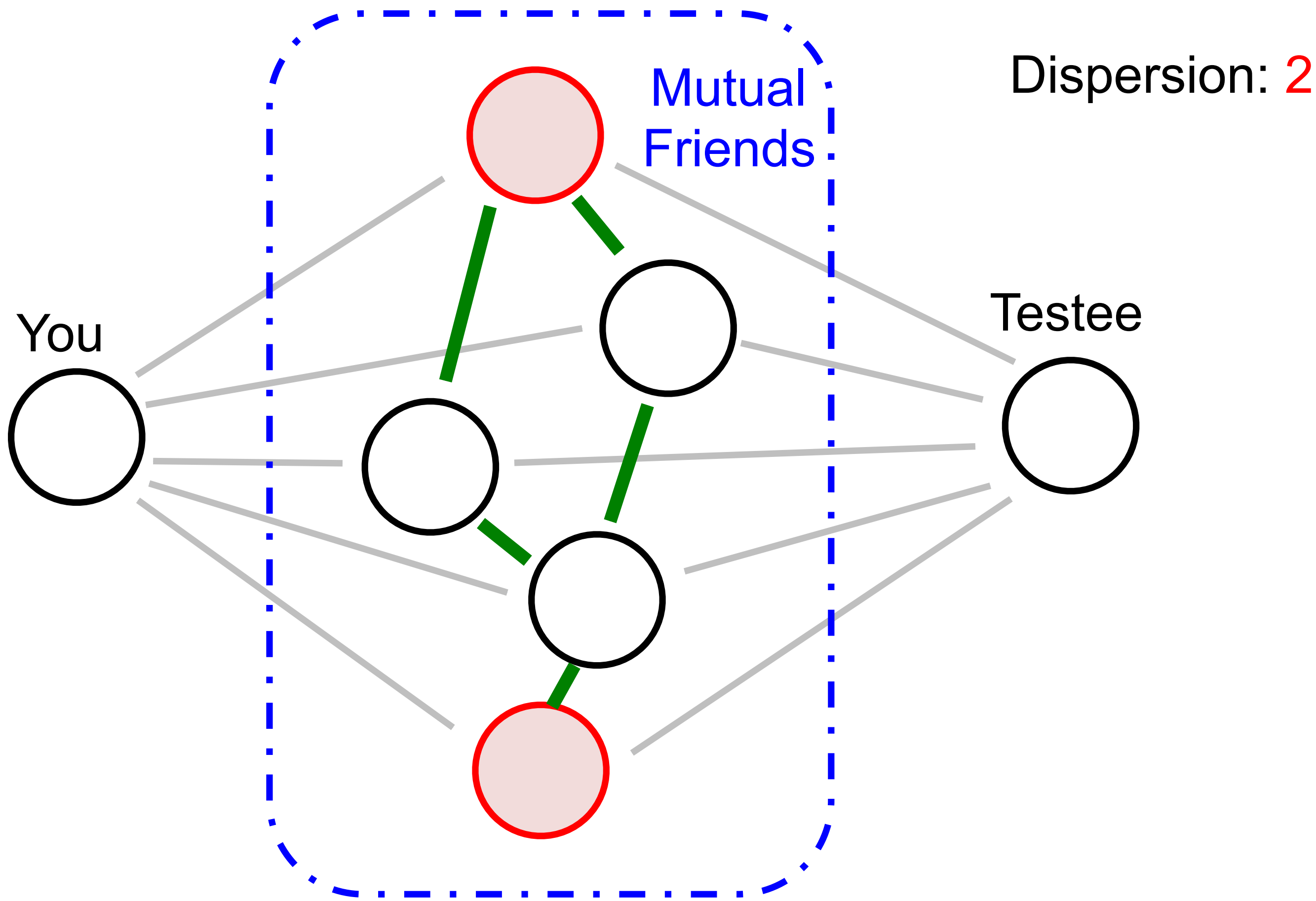
Dispersion: The extent to which two people's mutual friends are not directly connected

Dispersion



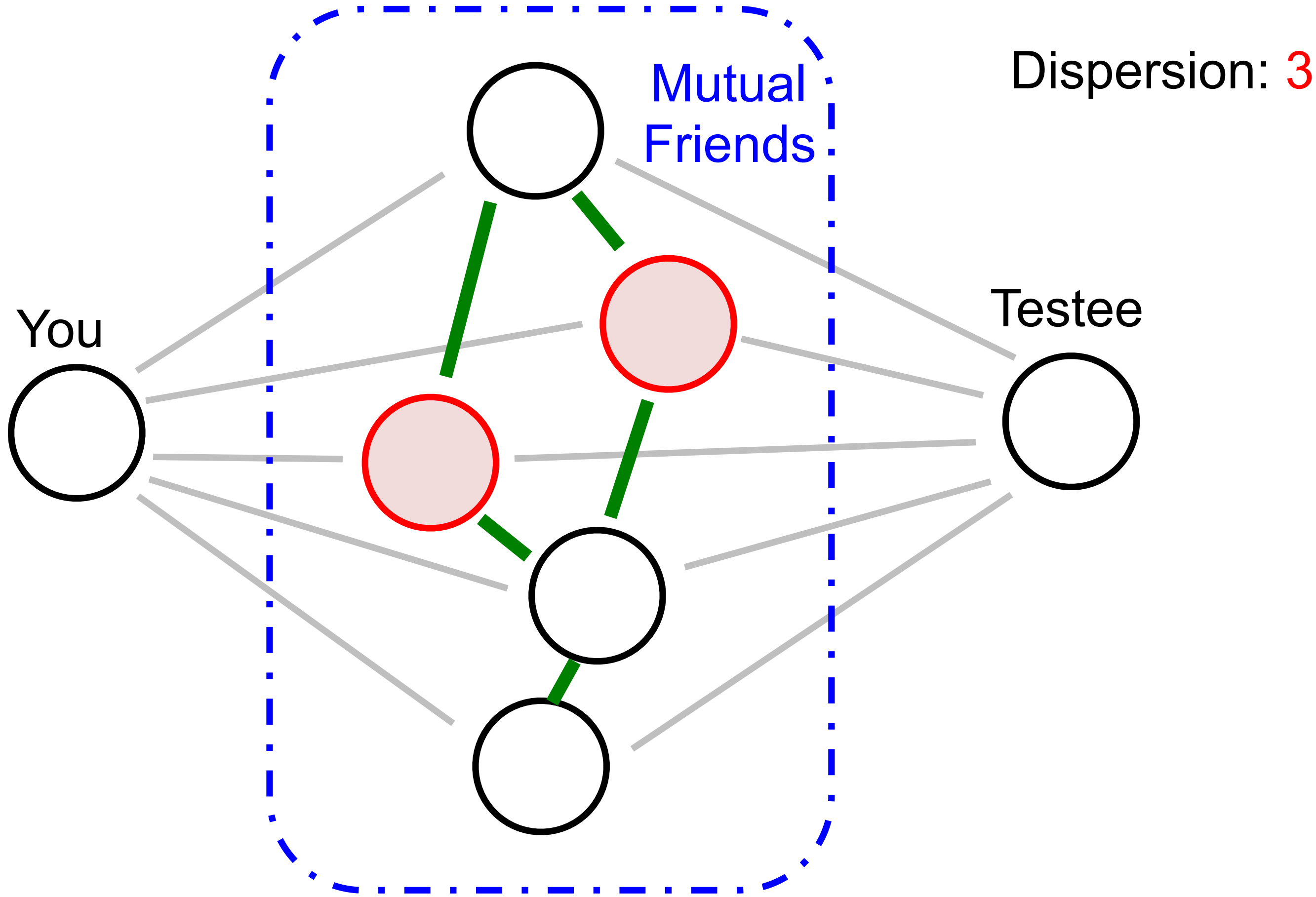
Dispersion: The extent to which two people's mutual friends are not directly connected

Dispersion



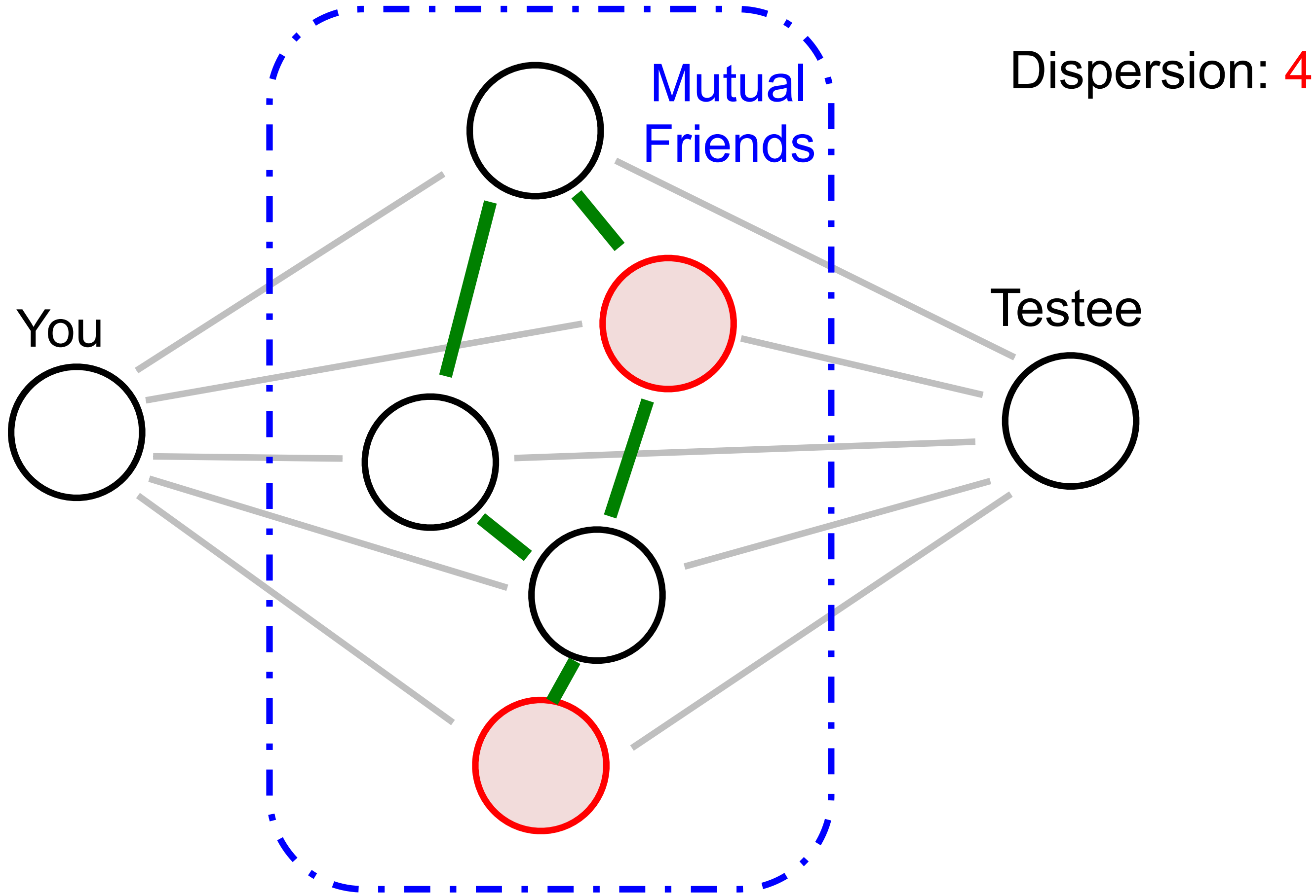
Dispersion: The extent to which two people's mutual friends are not directly connected

Dispersion



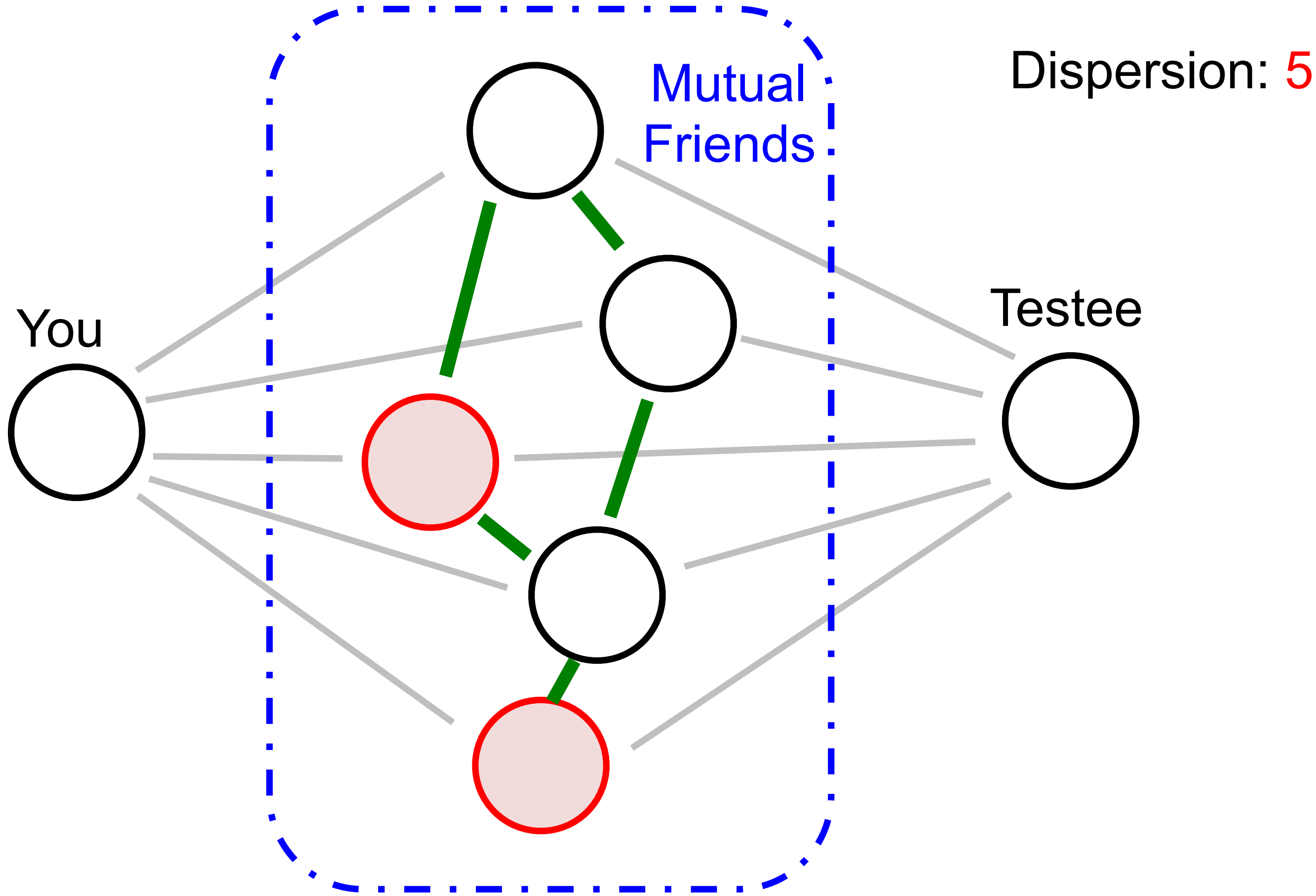
Dispersion: The extent to which two people's mutual friends are not directly connected

Dispersion



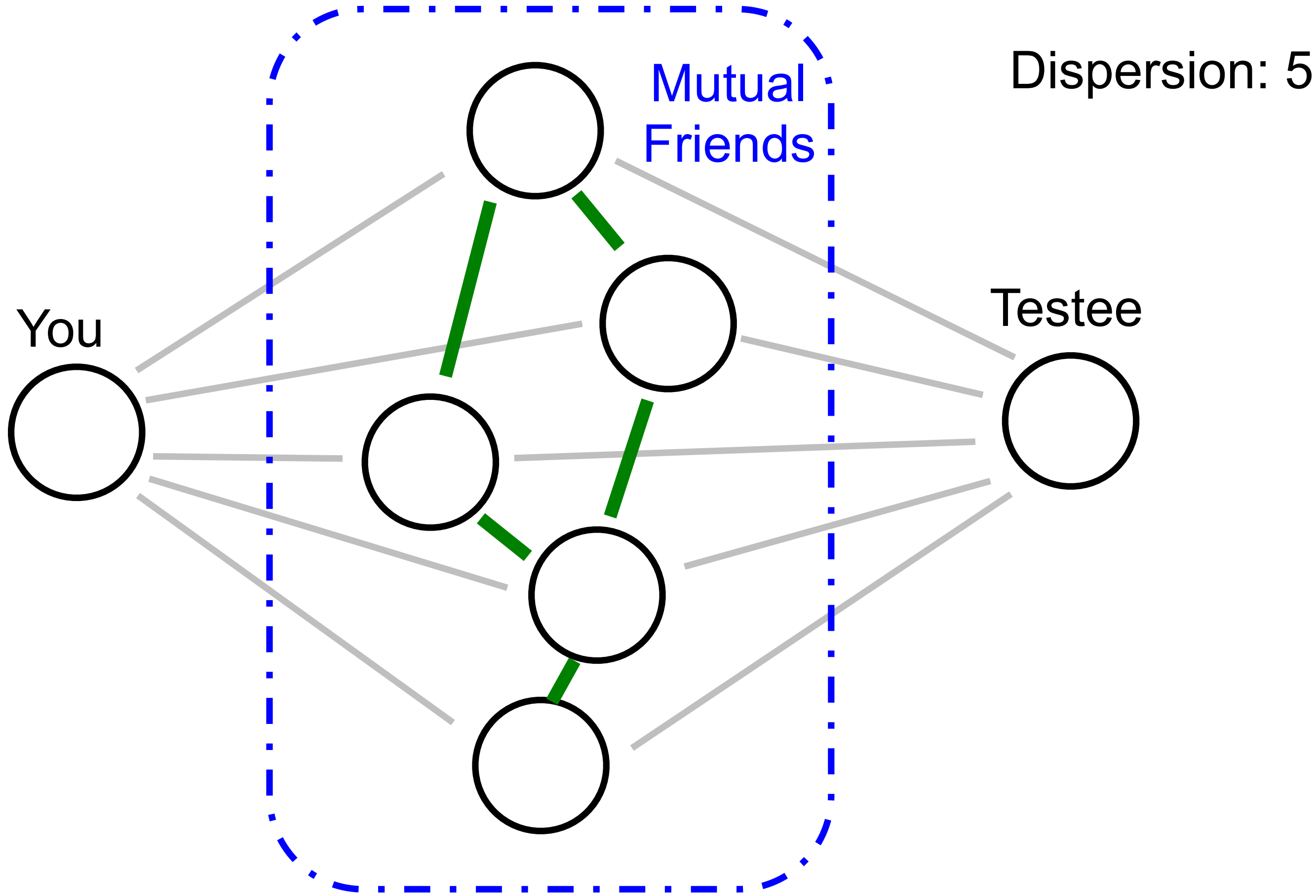
Dispersion: The extent to which two people's mutual friends are not directly connected

Dispersion



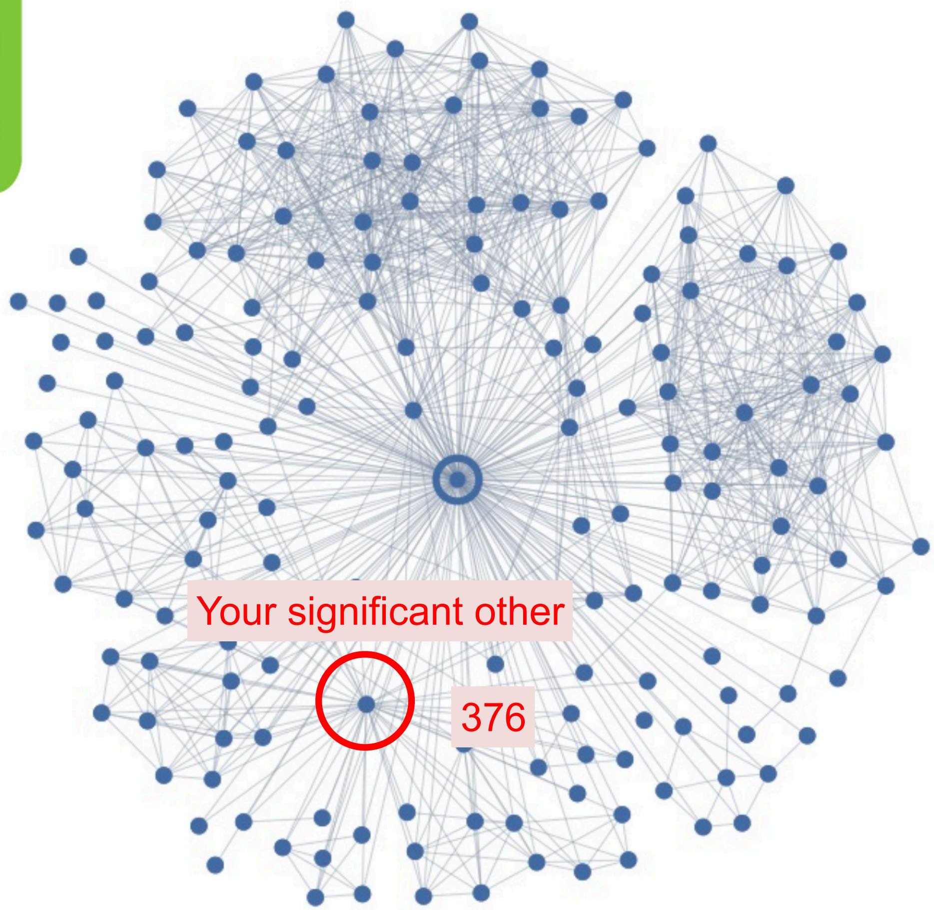
Dispersion: The extent to which two people's
mutual friends are not directly connected

Dispersion



Dispersion: The extent to which two people's mutual friends are not directly connected

Who Do You Love?



References and Advanced Reading

References:

- Wikipedia on graphs: https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)
- Wolfram Graph theory: http://mathworld.wolfram.com/Graph.html

Advanced Reading:

- Facebook graph API: https://developers.facebook.com/docs/graph-api
- Different graph lecture: https://www.youtube.com/watch?v=ylWAB6CMYiY

Extra Slides

Public Key Cryptography

Alice

Last time, we talked about hashing, and we have also discussed cryptographic hashing, which uses a hash function on some text (a file, for instance) to create a single number that represents that text.

One very cool and interesting use of hashing is in "public key cryptography," which enables users to share information *without passing the key between them.*

Bob

In other forms of secret message passing, two parties share a key (or password) that is used to encrypt and decrypt messages. But, this means that both parties need to share the key at some point, and they need to do that securely. This is difficult if you cannot meet directly with the person you want to exchange information with!

Public Key Cryptography

Alice

In Public Key Cryptography, two parties each generate a pair of keys: one is "public" and the other is "private".

Alice:
Public key: **hu76on9FLMRBk**…
Private key: **wlbu+qJ/RSzE**…

Bob

Bob:
Public key: **yhaLESwK+rGT1**…
Private key: **xMoWixEsCvqxk9c**…

There are two awesome properties of public and private keys:

1. If you hash text with your public key, only your private key will decrypt it.
2. If you hash text with your private key, only your public key will decrypt it.

Public Key Cryptography: Example

Alice

Let's say Alice wants to send Bob a secret message.

She asks Bob for his public key, which he gives her (and anyone else who wants it).

Bob's Public key: **yhaLESwK+rGT1**…

Bob

Alice then uses Bob's public key to encrypt her message:

"Meet me at 7pm in Gates" —> "bvbigKXsg0A3QAwtmc1x0LgXfgAoFOIj"

Bob's *private* key is the only key that will decrypt the message (even Alice can't decrypt it!)

Public Key Cryptography: Example

Alice

Bob

Alice never had to meet up with Bob to send a message securely — Bob can safely pass his public key around, because it can only be used to encrypt, not to decrypt.

Public Key Cryptography: Example 2

Alice

Let's say Alice wants to prove to Bob that a message is from her. She is not concerned whether the message itself is secret, but she wants to *sign* the message.

She encrypts her message with her *private key*.

Only her public key can decrypt the message.

Bob

When Bob gets Alice's message, he decrypts it with her public key (which is freely available) and because it decrypts properly, he knows that it must have been from her (because only her private key could have encrypted it).

This is the basis for message signatures!