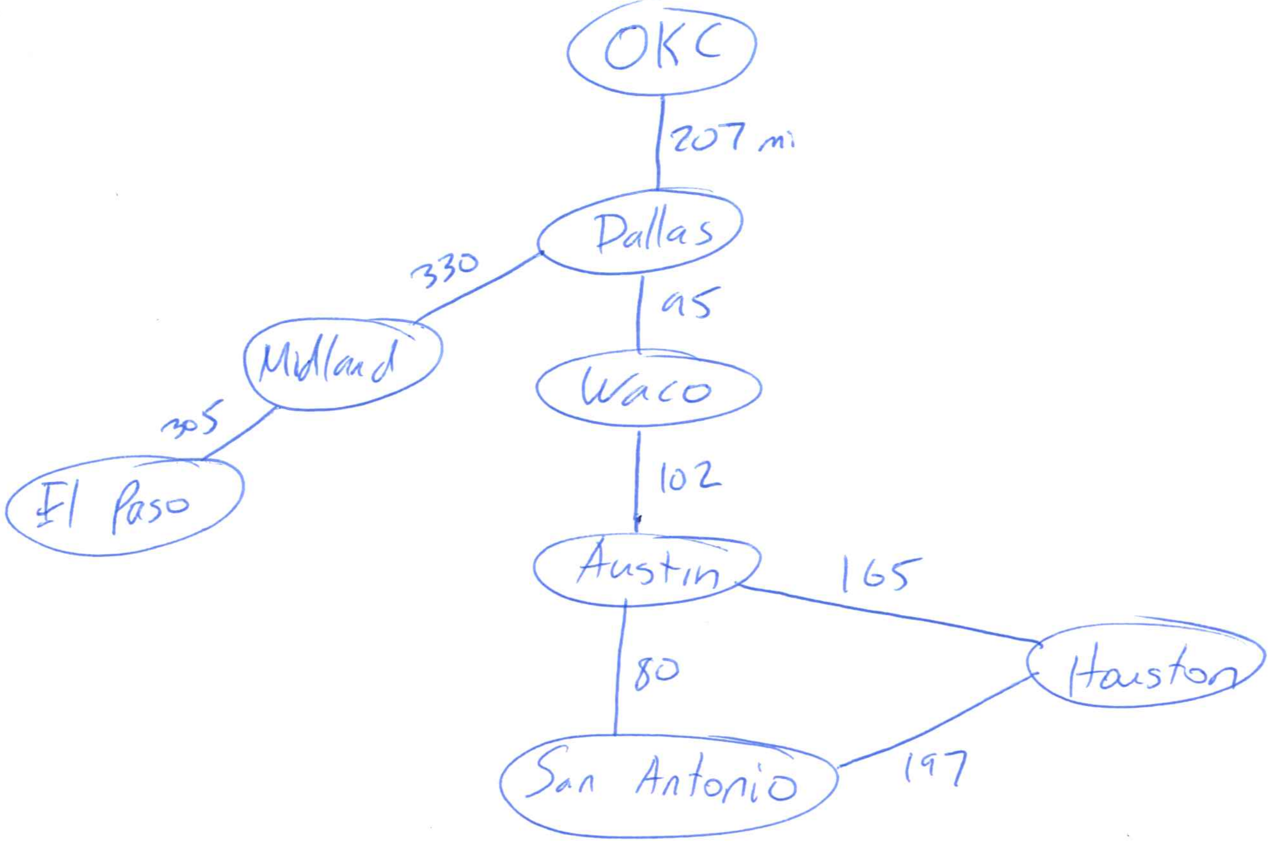
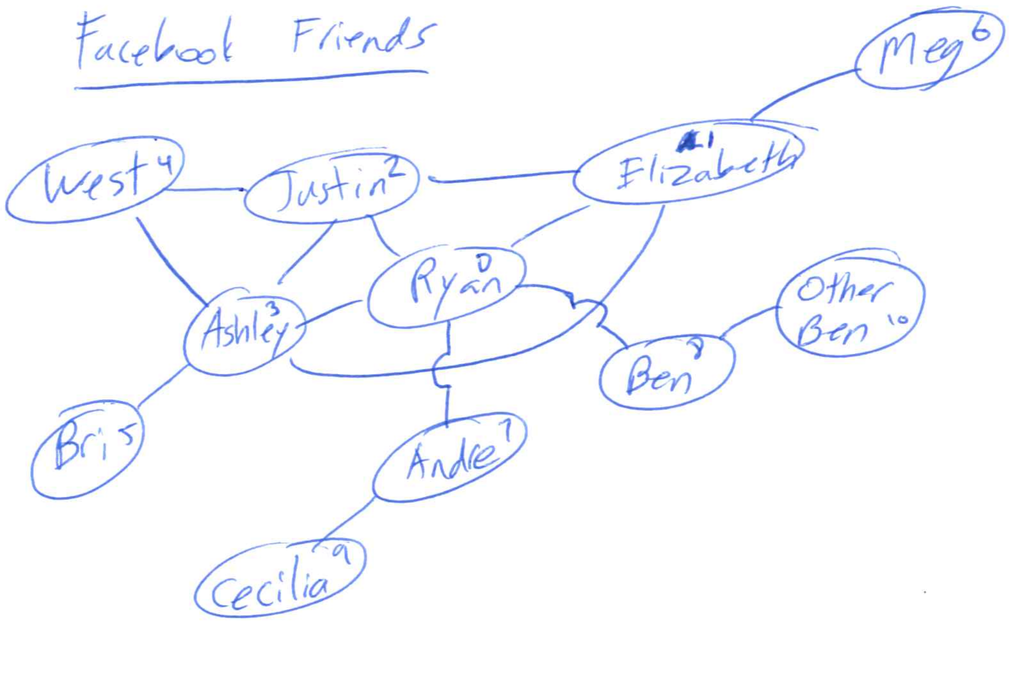
Example Graphs.

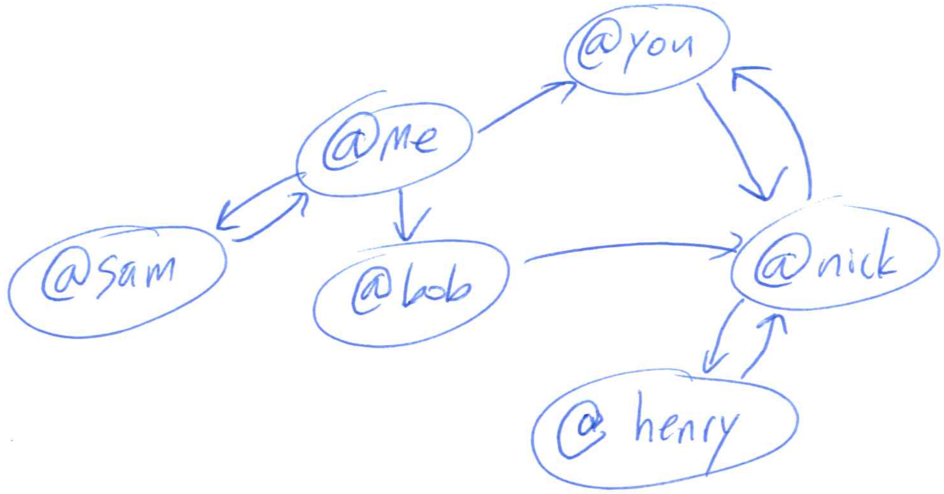
Roads & Cities (major highways only) → perhaps we only do busses...



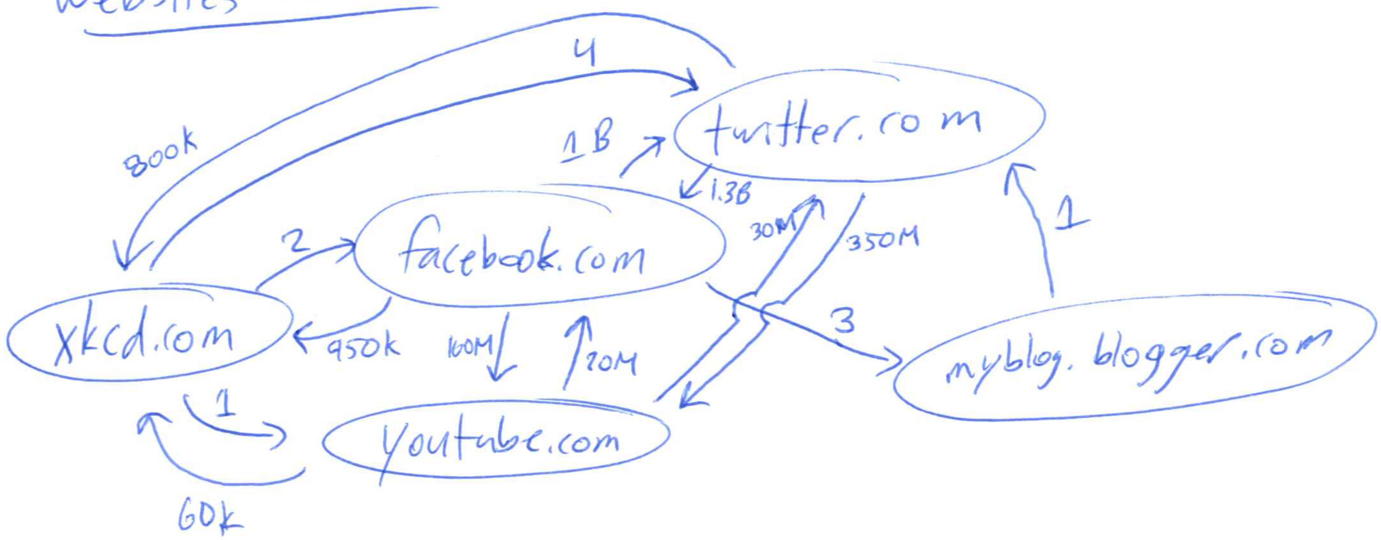
Facebook Friends



Twitter



Websites & Links



Graph Theory: Graphs – The ultimate data structure (+ misc. other topics)

- Example graphs (draw each on board)
  - Cities & Roads (undirected, weighted) → create what we'll later call an "adj. mat."
  - Facebook friends (undirected, unweighted) → create what we'll later call an "adj. list" ↳ the list of friends, for now.
  - Twitter followers (directed, unweighted)
  - Websites & Links (directed, weighted by # of links?) → create...
- Define a graph:

  A pair (V, E), where

  V is a set of vertices (aka, nodes)

  E is a set of edges (aka, connections)

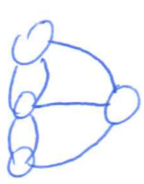
  $$E = \{\ldots, (v_i, v_j, w_k), \ldots \mid (v_i, v_j) \in V^2, w_k \in \mathbb{R}\}$$

- Terminology:

  Vertex (Node), Edge (connection), Directed/Undirected, Weighted/Unweighted, Neighbor, Degree, Complete (edges between all pairs), Connected, Path, Cycle, Subgraph, Connected component (subgraph that is connected)

- Why Graph "Theory"? Let's give the first "theorem":

  By Euler in 1736 (what didn't that guy do?):



  Eulerian path: iff zero or two odd degrees

  Eulerian ~~[illegible]~~ circuit: iff exacty two odd degrees

- TSP, NP-Hard
- More question we could ask:

precise
- Shortest path between two nodes? → (roads...)
- Shortest path between all pairs of nodes?

ambiguous
- How to find "missing" edges? → (friends...)
- How to identify "important" nodes? → (social influencers)

- How to represent in a program:

  - Adjacency Matrix

	a	b	c	d	e
a	∞	5	∞	∞	∞
b	5	∞	∞	10	4
c	∞	∞	∞	–	–
d	∞	10	–	∞	–
e	∞	4	–	–	∞

→ not nec. ∞

→ directed vs undirected

→ weighted vs unweighted

Storage: $O(n^2)$ $\quad n = |V|$

Lookup: $O(1)$

  - Adjacency List (Map):

$$\left\{ \begin{array}{l} a: \{b:5\}, \\ b: \{\ldots\}, \\ c: \{\ldots\}, \\ d: \{\ldots\}, \\ e: \{\ldots\} \end{array} \right\}$$

→ directed vs. undirected
weighted vs un-

$n = |V| \quad e = |E|$

Storage: $O(n+e)$

Lookup: $O(\log n + \log e)$

- Code now... Queue, Stack, Graph Algs...

```
import pandas as pd
from pprint import pprint
from collections import deque


def build_friends_adj_list():
    '''
    Builds and returns an adjacency list that representing the friends graph
    on the whiteboard.
    '''
    n = ['Ryan', 'Elizabeth', 'Justin', 'Ashley', 'West', 'Bri',
         'Meg', 'Andre', 'Ben', 'Cecilia', 'Other Ben']
    adj_list = {n[0] : {n[1], n[2], n[3], n[7], n[8]},
                n[1] : {n[0], n[2], n[3], n[6]},
                n[2] : {n[0], n[1], n[3], n[4]},
                n[3] : {n[0], n[2], n[1], n[5], n[4]},
                n[4] : {n[2], n[3]},
                n[5] : {n[3]},
                n[6] : {n[1]},
                n[7] : {n[0], n[9]},
                n[8] : {n[0], n[10]},
                n[9] : {n[7]},
                n[10]: {n[8]}}
    return adj_list


def bfs_visit_all(graph, start_node):
    '''
    INPUT:
        - graph: an adjacency list representation of an unweighted graph
        - start_node: the starting point in the graph, from which will will
                      begin our exploration

    This function will traverse the graph in a breadth-first way.
    This function implements the breadth-first-search (BFS) algorithm.
    '''
    next_nodes = deque([start_node])
    found_nodes = set([start_node])

    while len(next_nodes) > 0:
        node = next_nodes.popleft()   # <-- using as a queue!

        print "I'm now visiting node:", node

        for neighbor in graph[node]:
            if neighbor not in found_nodes:
                next_nodes.append(neighbor)
                found_nodes.add(neighbor)


def dfs_visit_all(graph, start_node):
    '''
    INPUT:
        - graph: an adjacency list representation of an unweighted graph
        - start_node: the starting point in the graph, from which will will
                      begin our exploration
```

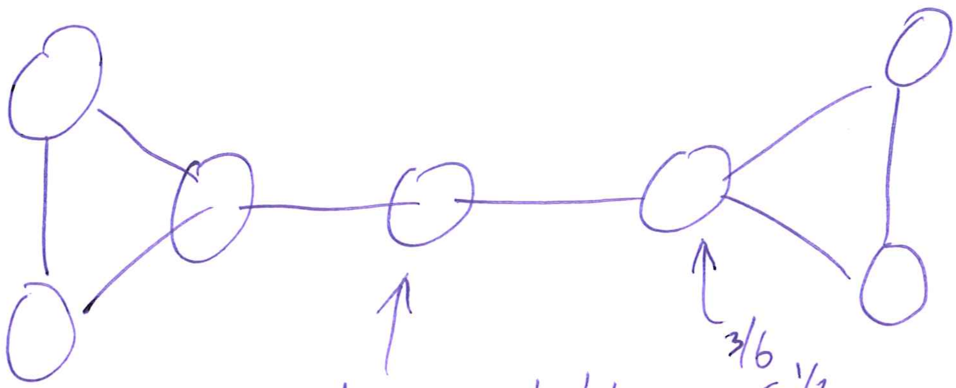
```
    This function will traverse the graph in a depth-first way.
    This function implements the depth-first-search (DFS) algorithm.

    NOTICE THE ONLY CHANGE IS THAT WE'RE USING A STACK NOW!
    '''
    next_nodes = deque([start_node])
    found_nodes = set([start_node])

    while len(next_nodes) > 0:
        node = next_nodes.pop()   # <-- using as a stack!

        print "I'm now visiting node:", node

        for neighbor in graph[node]:
            if neighbor not in found_nodes:
                next_nodes.append(neighbor)
                found_nodes.add(neighbor)


def bfs_visit_all_limit(graph, start_node, max_depth):
    '''
    INPUT:
        - graph: an adjacency list representation of an unweighted graph
        - start_node: the starting point in the graph, from which will will
                      begin our exploration

    This function will traverse the graph in a breadth-first way.
    This function implements the breadth-first-search (BFS) algorithm.

    This function will quit when we reach a certain max_depth.
    '''
    next_nodes = deque([(start_node, 0)])  # <-- NEW: using tuples
    found_nodes = set([start_node])

    while len(next_nodes) > 0:
        node, depth_here = next_nodes.popleft()  # <-- NEW

        print "I'm now visiting node:", node

        if depth_here + 1 <= max_depth:  # <-- NEW
            for neighbor in graph[node]:
                if neighbor not in found_nodes:
                    next_nodes.append((neighbor, depth_here+1))  # <-- NEW
                    found_nodes.add(neighbor)


def dfs_visit_all_recursive(graph, node, found_nodes=None):
    '''
    INPUT:
        - graph: an adjacency list representation of an unweighted graph
        - node: the starting point in the graph, from which will will
                begin our exploration
        - found_nodes: if not None, the set of nodes we've already processed

    This function will traverse the graph in a depth-first way.
```

```
    This function implements the depth-first-search (DFS) algorithm.

    This version uses recursion.

    DEMO STACK OVERFLOW!!!!! (i.e. add a bug and run)
    '''
    if found_nodes is None:
        found_nodes = set([node])
    else:
        found_nodes.add(node)

    print "I'm now visiting node:", node

    for neighbor in graph[node]:
        if neighbor not in found_nodes:
            dfs_visit_all_recursive(graph, neighbor, found_nodes)


def connected_component(graph, start_node):
    '''
    INPUT:
        - graph: an adjacency list representation of an unweighted graph
        - start_node: the starting point in the graph, from which will will
                      begin our exploration
    RETURN:
        - set_of_nodes: the set of nodes in this connected component

    This function will return the nodes in the connected component that
    contains start_node.
    '''
    found_nodes = set()
    dfs_visit_all_recursive(graph, start_node, found_nodes)
    return found_nodes


def build_cities_adj_matrix():
    '''
    Builds and returns an adjacency matrix representing the cities graph
    on the whiteboard.
    '''
    inf = float('inf')
    cities = ['OKC', 'Dallas', 'Waco', 'Austin', 'San Antonio',
              'Houston', 'Midland', 'El Paso']
    adj_matrix = [[  0, 207, inf, inf, inf, inf, inf, inf],
                  [207,   0,  95, inf, inf, inf, 330, inf],
                  [inf,  95,   0, 102, inf, inf, inf, inf],
                  [inf, inf, 102,   0,  80, 165, inf, inf],
                  [inf, inf, inf,  80,   0, 197, inf, inf],
                  [inf, inf, inf, 165, 197,   0, inf, inf],
                  [inf, 330, inf, inf, inf, inf,   0, 305],
                  [inf, inf, inf, inf, inf, inf, 305,   0]]
    graph = pd.DataFrame(adj_matrix, index=cities, columns=cities)
    return graph


def floyd_warshall(graph):
```

```
    '''
    INPUT:
        - graph: an adjacency matrix representation of a graph
    RETURN:
        - a pandas dataframe of the length of the shortest path between
          all pairs of nodes

    DISCUSS BIG-O OF THIS!!!!!
    '''
    d = graph.copy()
    n = d.shape[0]
    for i in xrange(n):
        for j in xrange(n):
            for k in xrange(n):
                if  d.iloc[i,k] > d.iloc[i,j] + d.iloc[j,k]:
                    d.iloc[i,k] = d.iloc[i,j] + d.iloc[j,k]
    return d


if __name__ == '__main__':

    friend_graph = build_friends_adj_list()
    pprint(friend_graph)

    bfs_visit_all(friend_graph, 'Ryan')
    print

    dfs_visit_all(friend_graph, 'Ryan')
    print

    bfs_visit_all_limit(friend_graph, 'Ryan', 1)
    print

    dfs_visit_all_recursive(friend_graph, 'Ryan')
    print

    print connected_component(friend_graph, 'Ryan')

    city_graph = build_cities_adj_matrix()
    print city_graph

    print floyd_warshall(city_graph)
```

Centrality

degree centrality(v): $\dfrac{\text{degree of } \cancel{\text{node}}\ v}{|V|-1}$



$\frac{3}{6} = \frac{1}{2}$

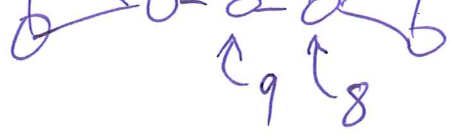
degree centrality $= \frac{2}{6} = \frac{1}{3}$

betweenness centrality:

$$\text{betweenness}(v) = \sum_{s \neq v \neq t} \text{\% of shortest paths from } s \text{ to } t \text{ through } v$$

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

$$\text{betweenness centrality}(v) = \frac{\text{betweenness}(v)}{(n-1)(n-2)}$$



9 8

☆ Mention btness / edge centrality