CS1 Lecture 34 Apr. 12, 2017

- HW 6 scores will be posted today
- HW 8 due next Wed.
  - 6 point question available now, 4 point question will be added Friday
- Discussion sections this week. Important for HW8. A graph problem requiring slight modification of BFS algorithm.
- Exam 2: April 20, 6:30-8:00pm

Last time

- Graph representation
- A graph traversal algorithm
- Introduction to HW8 graph problem

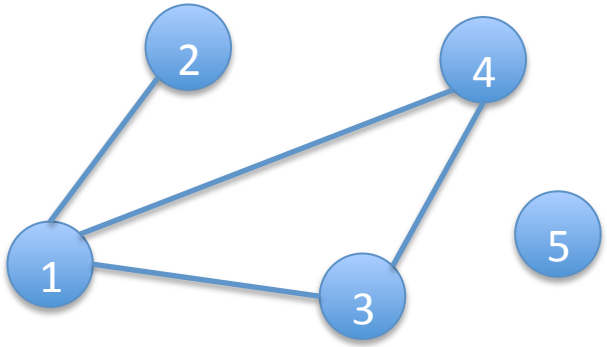
Today

- Review graph representation and bfs traversal
- Discuss HW8 word ladder problem
- Short depth-first search overview

Review - Adjacency list

Use a dictionary with

- Nodes as keys
- Values are lists of neighbor nodes



KEY	VALUE
1	[2, 4, 3]
2	[1]
3	[1, 4]
4	[3, 1]
5	[]

Review - **adjacency list** representation for undirected graphs in Python

Two classes: Node and Graph

basicgraph.py

Node

- properties:
  - name : string
  - status: string

- methods
  - getName
  - getStatus, setStatus
  - __repr__ : we'll print nodes as <*name*>

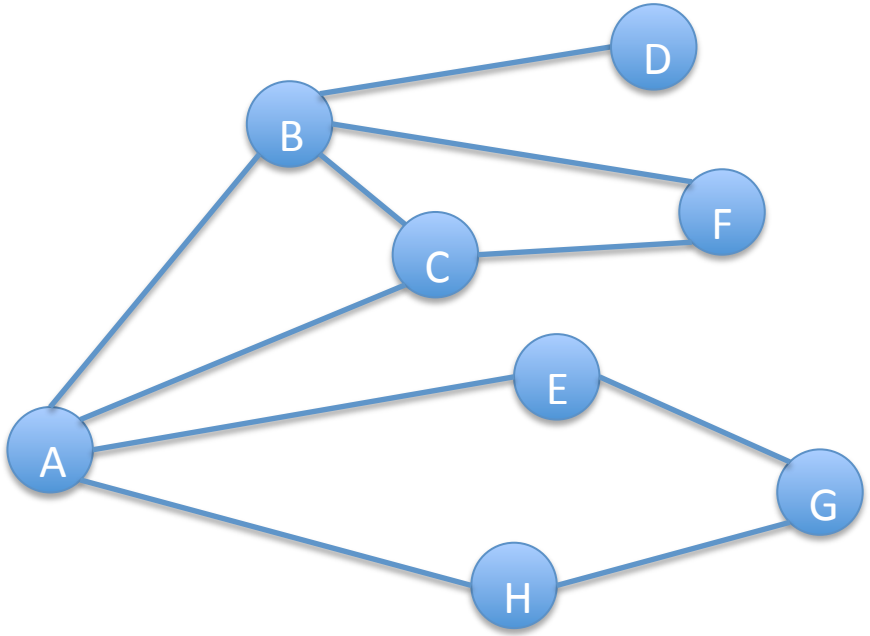
Note: we will add additional properties when we write functions that traverse/walk through graphs

Review – adjacency list representation in Python

Graph

- properties
  - nodes: a list of Node objects
  - edges: a list of 2-tuples of nodes
  - adjacencyLists: a dictionary with all nodes as keys. The value associated with a key n1 (where n1 is a node) is a list of all the nodes, n2, for which (n1,n2) is an edge.
- methods
  - addNode(node) : nodes must be added to graphs before edges
  - addEdge(node1, node2) : presumes both nodes in graph already
  - neighborsOf(node) : returns list of neighboring nodes
  - getNode(name)
  - hasNode(node)
  - hasEdge(node1, node2) : return T if edge node1-node2 in graph
  - __repr__

basicgraph.py



KEY	VALUE
A	[B,C,E,H]
B	[A,C,D,F]
C	[A,B,F]
D	[B]
E	[A,G]
F	[B,C]
G	[E,H]
H	[A,G]

This graph is generated by genDemoGraph() in basicGraph.py

For exams, you need to be able to 1) draw graph given adjacency list dictionary, and/or 2) show adjacency list dictionary given graph drawing

Many real-world problems can be represented as problems involving graphs. The algorithms to solve those problems often involve **graph traversals**, organized exploration or "walkthroughs" of the graph. Two famous ones: depth-first search and breadth-first search. We will cover only breadth-first search. You will not be responsible for knowing the details of breadth-first search (for exam purposes) but you need to understand it well enough to *use* it in HW8.

Review - classic breadth-first search (bfs)

The goal of classic bfs is simply to travel to/explore, in an efficient and organized fashion, all nodes reachable from some given startNode. (The general goal of the other classic traversal, depth-first-search, is the same,. It just explores in a different order.)

First, we'll add a property, **status**, to nodes. Legal values will be 'unseen', 'seen', and 'processed'. The traversal process will use the values as it encounters nodes to keep from revisiting/re-processing already-processed nodes.

The algorithm in words:

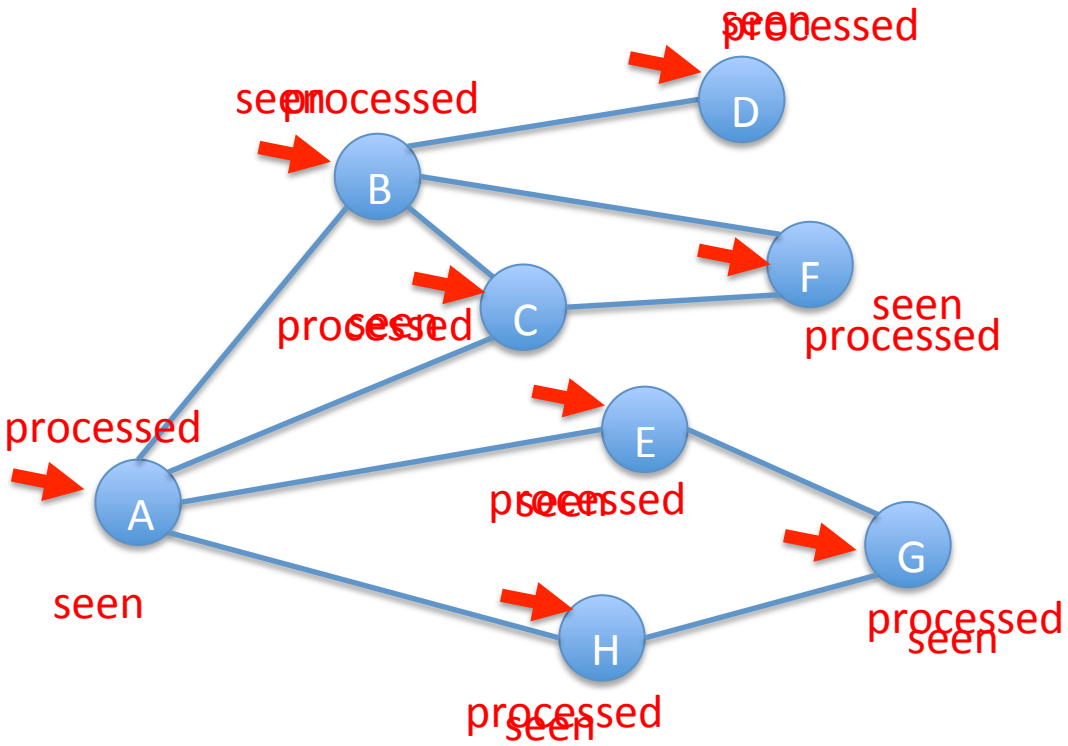
1. Mark all nodes as 'unseen'
2. Initialize an empty queue (first-in, first-out data structure from Disc. Sec. 6)
3. Mark the starting node as 'seen' and place it in the queue.

4. Remove the front node of the queue and call it the current node
5. Consider each neighbor of the current node. If its status is 'unseen', mark it as seen and put it on the queue.
6. Mark the current node 'processed' (*note: this step can be left out*).
7. If queue not empty go back to step 4.

This will explore all node reachable from the startNode in a breadth-first manner.

- Suppose startNode has neighbors n1 and n2. "Breadth first manner" means it travels from start to n1 and then from start to n2 before exploring "beyond n1" – the process moves "broadly" out from the start a single step at a time.
- This enables a very simple modification of bfs to compute shortest (unweighted) distances from start to all other nodes in the graph

BFS starting at node A



Mark all nodes ‘unseen’
Mark A ‘seen’ and put A on queue Q
Until queue empty do:

- Remove the front node of the queue and call it the current node
- Consider each neighbor of the current node. If its status is ‘unseen’, mark as ‘seen’ and put it on the queue.
- Mark the current node ‘processed’

	Q: A
curr: A	Q:
	Q: B, C, E, H
curr: B	Q: C, E, H
	Q: C, E, H, D, F
curr: C	Q: E, H, D, F
	Q: E, H, D, F
curr: E	Q: H, D, F
	Q: H, D, F, G
curr: H	Q: D, F, G
	Q: D, F, G
curr: D	Q: F, G
	Q: F, G
curr: F	Q: G
	Q: G
curr: G	Q:
	Q: DONE

Bread first search

- For unweighted graphs, bfs efficiently finds shortest path from start node to every other node!
  - "Three and a half degrees of separation" https://research.fb.com/three-and-a-half-degrees-of-separation/
  - Wikipedia game: given two topics (with Wikipedia pages), race to get from one to the other clicking only on Wikipedia page links. https://en.wikipedia.org/wiki/Wikipedia:Six_degrees_of_Wikipedia https://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia wikiGame.py
  - HW8 word ladders!
- Links
  - http://en.wikipedia.org/wiki/Breadth-first_search
  - http://interactivepython.org/courselib/static/pythonds/Graphs/ImplementingBreadthFirstSearch.html
  - animations:
    - https://www.cs.usfca.edu/~galles/visualization/BFS.html
    - http://visualgo.net/dfsbfs.html
    - http://www3.cs.stonybrook.edu/~skiena/combinatorica/animations/search.html

Word ladder puzzles

CAT	CAT	CAT *or* CAT
???	COT	COT COT
???	???	DOT COG
DOG	DOG	DOG DOG

Find 3-letter English words for ??? Positions. Each must differ from previous and next word in only one location

This problem is easily representable and solvable using graphs! HW8 Q1

HW8 Q1 Word Ladder problem

Several parts, none complicated. (AGAIN: Implement and test parts – don't just write all the code at once and hope it works!)

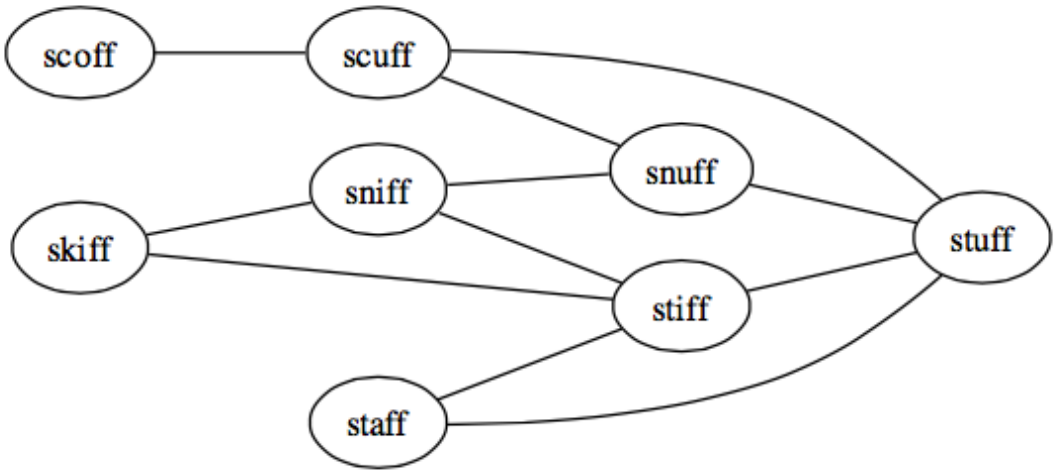
1. Read word list and create graph
   - One node per word
   - Edge between each pair of words differing by one letter
2. Execute breadth-first search starting from start word's node.
3. "extract" ladder from graph (after bfs) by following a parent path (you will add 'parent' property to Node class and set during bfs) from end word's node back to start node
4. Interactive loop requesting input from user: request two words, call findLadder (which should return a representation of the ladder), print result nicely

wordladderStart.py (see HW8 assignment) has stubs for all the functions you need.

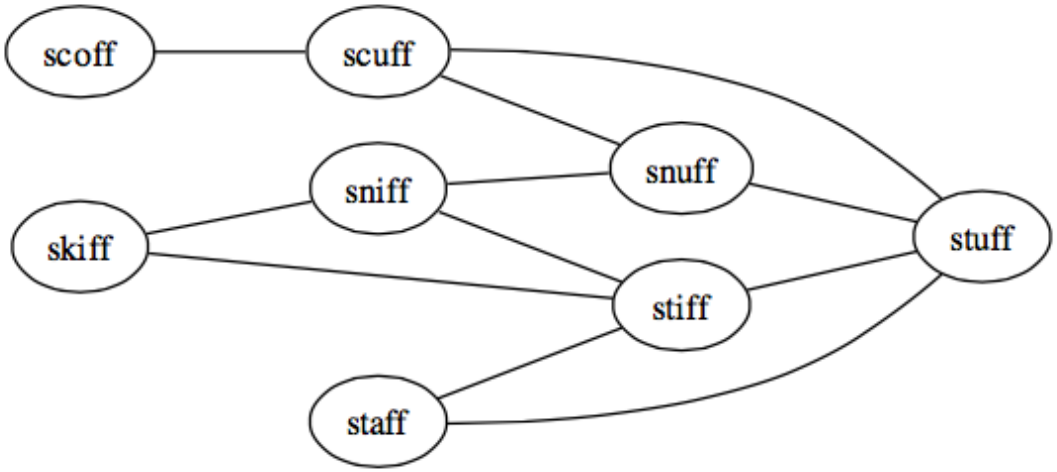
NOTES:

- it's okay to use simple O(n^2) process to create the graph
  - If you are interested, there is a faster way: http://interactivepython.org/courselib/static/pythonds/Graphs/BuildingtheWordLadderGraph.html
- Create a tiny word file to test on. Choose the words, draw the graph by hand, and check program's results. See example on next slide.
- findWordLadder should do both:
  1. bfs – modified to keep track of 'parent'. When a neighbor node is marked 'seen', set parent to current node. This is similar to what you do in disc. section this week!
  2. ladder can be constructed/printed by following parent path from end word to start word

  (NOTE: with interactive loop repeatedly asking for two new words, are there cases where step1, the bfs, of findWordLadder is not necessary? Don't worry about this – it's no problem to always do the bfs. It's fast!)



- Breadth first search from ‘scoff’:
  - order in which nodes are finished/processed:
    - dist 0 scoff
    - dist 1 scuff
    - dist 2 snuff, stuff
    - dist 3 sniff, stiff, staff
- wordLadder correctly gives:
  - scoff -> scuff -> stuff for scoff-to-stuff ladder
  - scoff -> scuff -> stuff -> stiff for scoff-to-stiff ladder



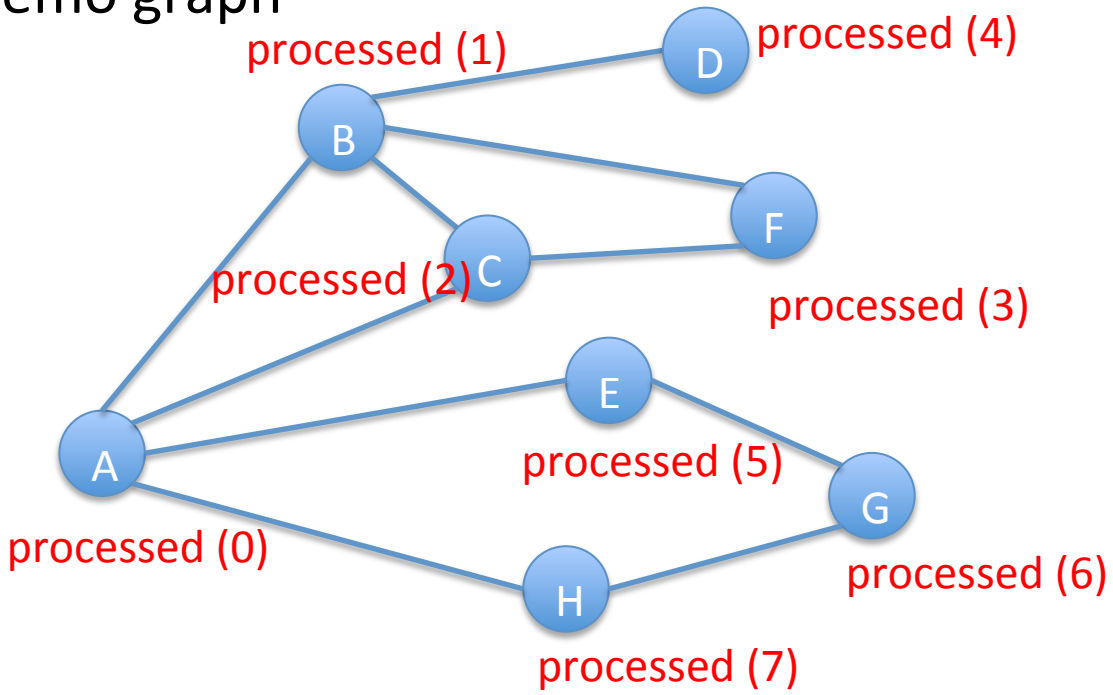
What if we wanted to find a longer path?

There is another classic graph traversal method called **depth first search** (https://en.wikipedia.org/wiki/Depth-first_search).

The basic idea of it is to explore deeply before exploring broadly. You will study the algorithm in later courses but it is quite concise:

```
DFS(node):
    mark node 'processed'
    for each 'unseen' neighbor of node:
        mark neighbor seen
        DFS(neighbor)
```

DFS starting at node A (on original demo graph



DFS(A)

DFS(B)

DFS(C)

DFS(F)

DFC(D)

DFS(E)

DFS(G)

DFS(H)

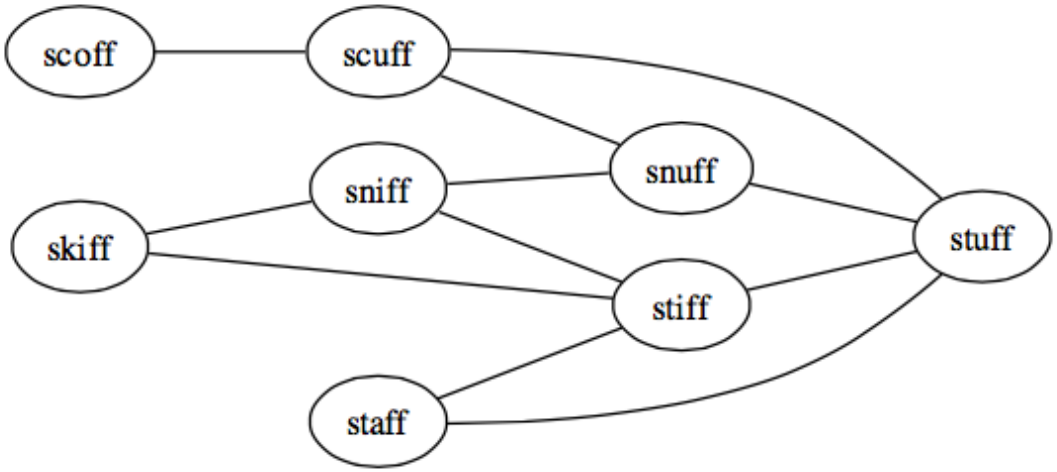
DONE

Mark all nodes 'unseen'
Call DFS on desired start node

DFS(node):

- mark node 'processed'
- for each 'unseen' neighbor of node:
  - mark neighbor seen
  - DFS(neighbor)

Note: number in parentheses corresponds to order in which nodes were marked processed



- Depth first search from ‘scoff’:
  - order in which nodes are finished/processed:
    - scoff, scuff, snuff, sniff, skiff, stiff, staff, stuff
- and wordLadder gives:
  - scoff -> scuff -> snuff -> sniff -> skiff -> stiff -> staff -> stuff for scoff-to-stuff ladder
- for black->white in full words5.text file?
  - DFS quickly finds ladder 192 long
  - Is that the longest possible? NO! There is one at least 648 long (I don’t know if there’s a longer one or not.)
  - Depth first search might find long paths, but not necessarily longest. In fact there is no known efficient general algorithm for finding longest path in a graph!

modifying Node class and bfs for wordLadder

1. **add parent property** and **getParent** and **setParent** methods to Node class

2. modify bfs:

Mark all nodes ‘unseen’
Set all nodes’ parents to None
Mark start node ‘seen’ and put it on queue

Until queue empty do:

- Remove the front node of the queue and call it the current node
- Consider each neighbor of the current node. If its status is ‘unseen’, mark as ‘seen’, **set its parent to current node**, and put it on the queue.
- Mark the current node ‘processed’

Next time

- Introduction to randomization and simulation/Monte Carlo techniques
  - Related, for fun: Monty Hall problem http://en.wikipedia.org/wiki/Monty_Hall_problem
  - M. vos Savant (whose column generated huge public awareness) quotes psychologist M. Piattelli-Palmarini: "... no other statistical puzzle comes so close to fooling all the people all the time." Herbranson and Schroeder: Pigeons repeatedly exposed to the problem show that they rapidly learn always to switch, unlike humans. ☺
  - we will use simulation, randomization to help "see" solution