CS2110 Lecture 32 Apr. 7, 2021

- HW 7 due tonight
- DS 8 available later today, due Sunday
- HW 8 available tomorrow, due next Thursday

Last time

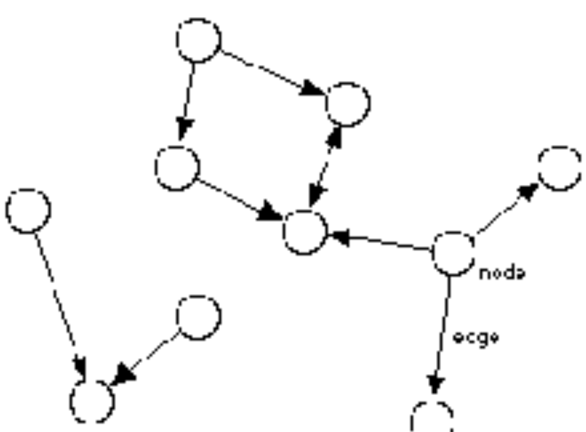
- Intro to optimization algorithms, greedy algorithms
- Introduced graphs (the computer science/mathematical kind), not the charts you've been plotting with pylab

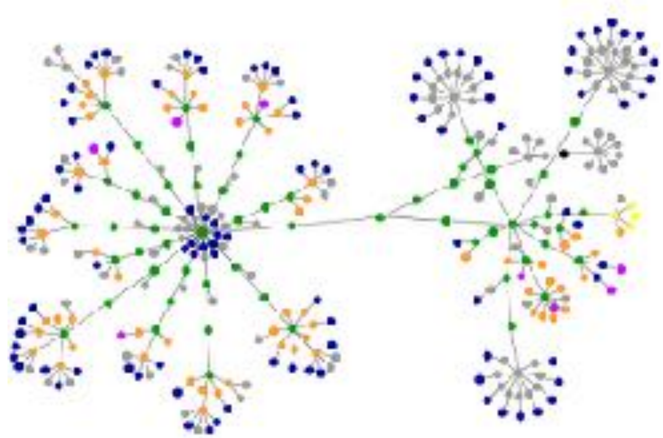
Today

- Graph representations
- Basic graph algorithms

Graphs and optimization problems based on graphs

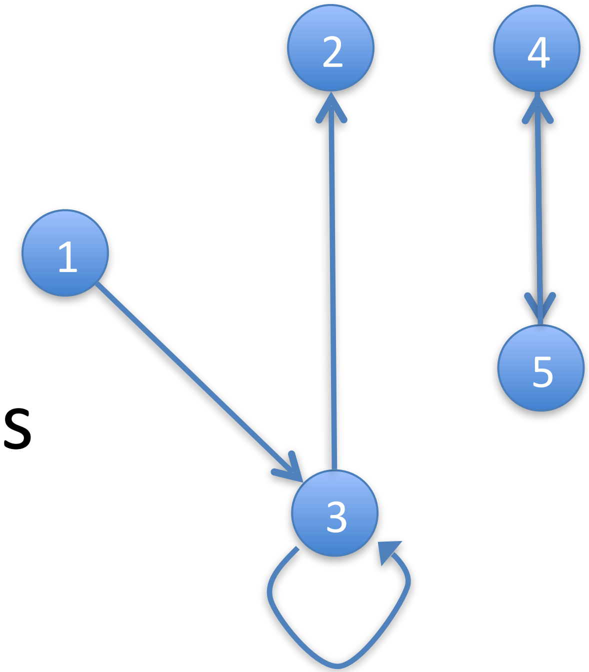
- *Many* important real-world problems can be modeled as optimization problems on **graphs**
- A graph is:
  - A set of nodes (vertices)
  - A set of edges (arcs) representing connections between pairs of nodes
- There are several types of graphs:
  - **Directed**. Edges are “one way” from source to destination)
  - **Undirected**. Edges have no particular direction – can travel either way, “see” each node from other, etc.
  - **Weighted**. Edges have associated numbers called weights that can be used to represent cost, time, flow capacity, etc.
- See, e.g., Ch. 2 of free online book – Think Complexity - http://greenteapress.com/complexity/html/thinkcomplexity003.html

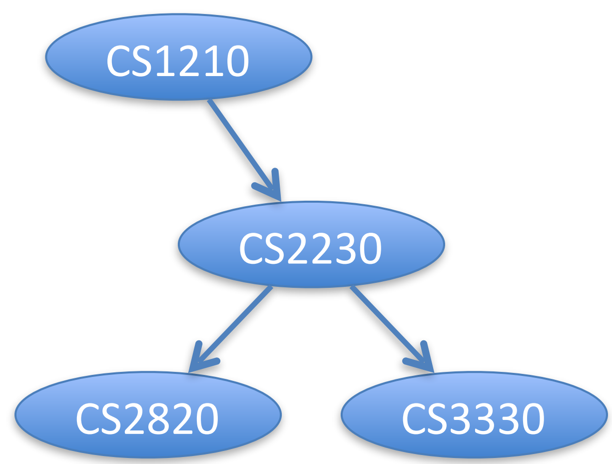




Directed graph

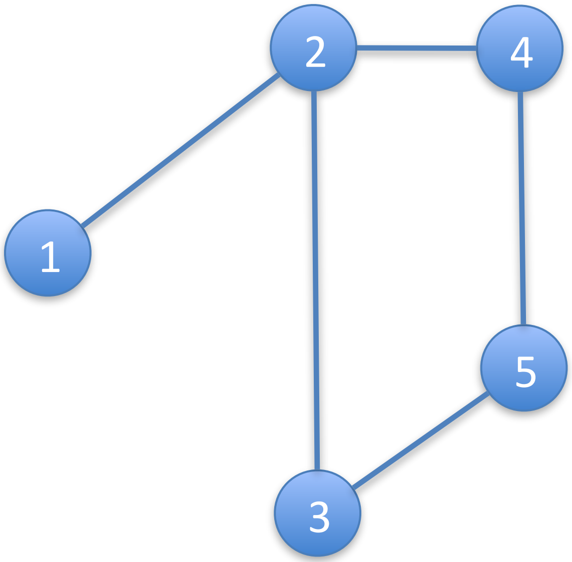
- edges are "one way" from source to destination
- Can have two (one each way) between a pair of nodes
- Node can have edge to self
- Example relationships:
  - course prerequisite
  - hyperlink between web pages
  - street between intersections
  - Twitter follower
  - Infection spread from-to

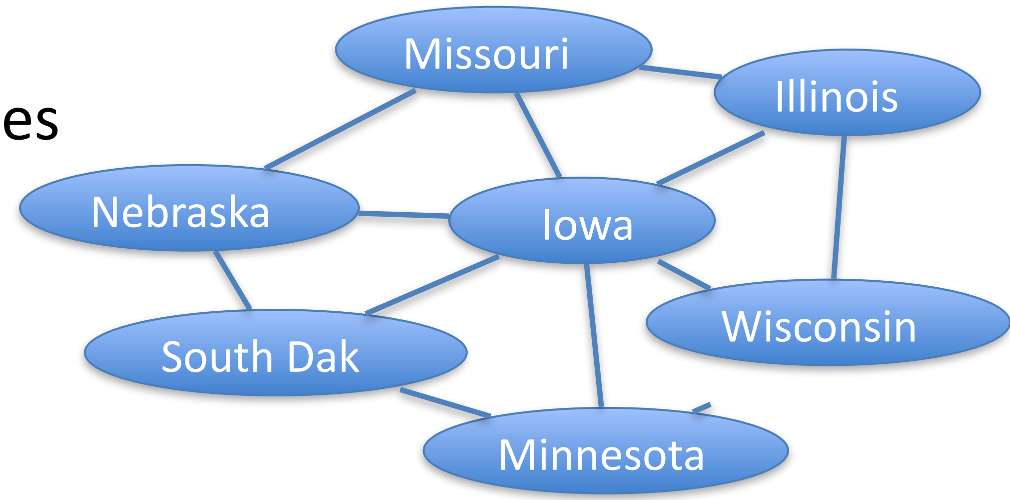




Undirected graph

- Edges have no direction. Can “travel” either direction
- Can have only one edge between a pair of nodes*
- Node cannot have edge to self
- Example relationships:
  - Facebook friend
  - Bordering countries/states

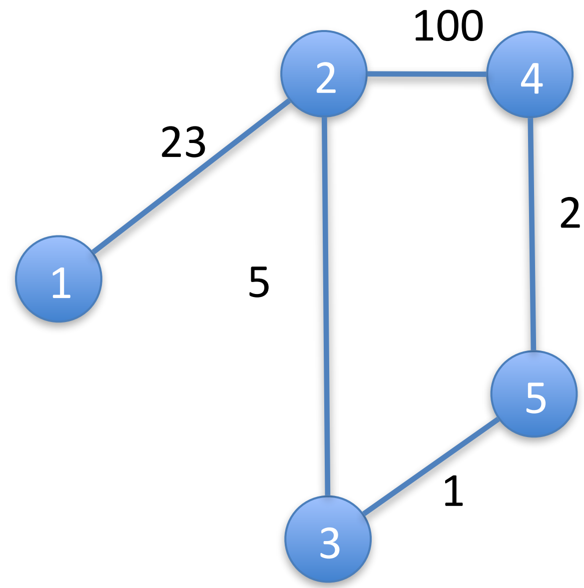


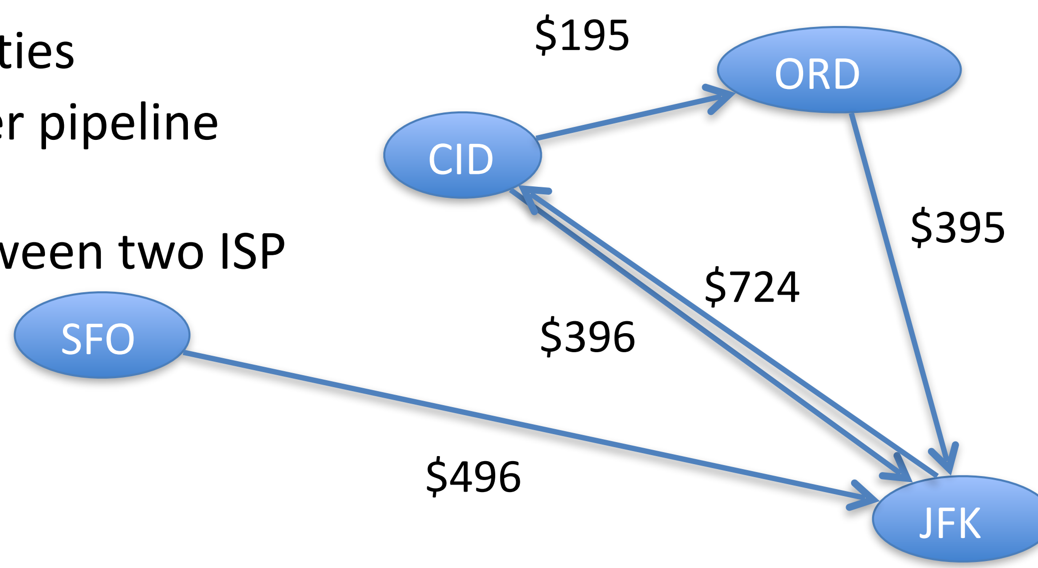


*another kind of graph – multigraph – relaxes this rule

Weighted graphs

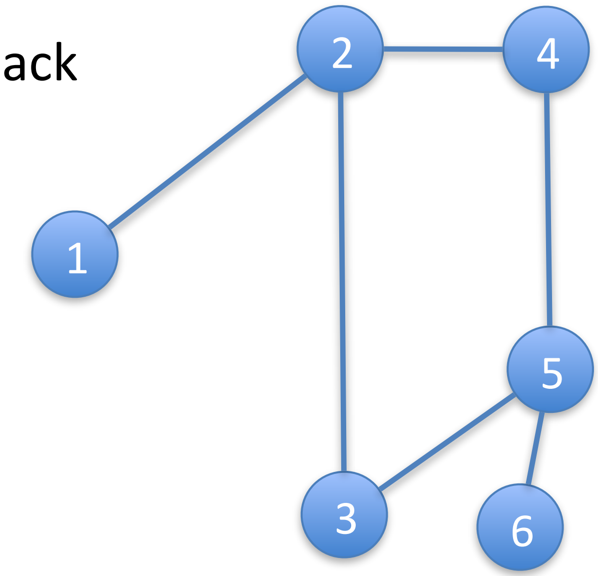
- Variant of both directed and undirected graphs in which each edge has an associate number called a **weight** or cost
- Edge weight provides additional information about the relationship between the nodes.
- Example relationships:
  - Airfare between two cities
  - Distance between two cities
  - Flow capacity of oil/water pipeline between two points
  - Network bandwidth between two ISP nodes

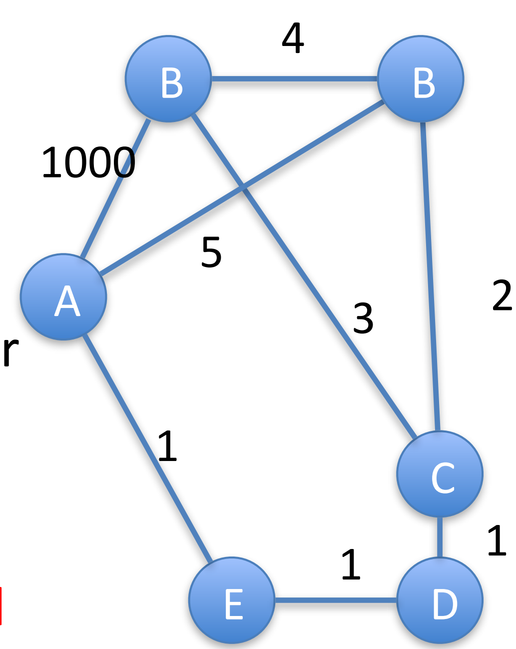




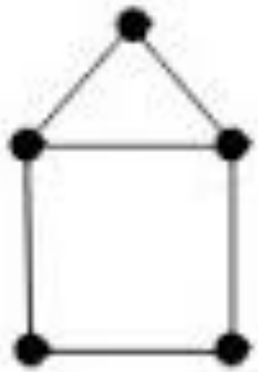
Classic graph problems

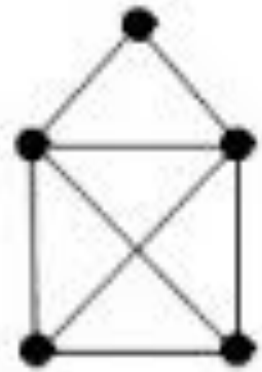
- Determine if a graph has a cycle, a path that loops back to start points (e.g. 2-4-5-3-2)
- Find a path (non-branching) that traverses each (undirected) edge exactly once
  - Leonhard Euler and the Bridges of Königsberg
  - Not possible in graph on top right
- Find the shortest path between source s and destination
  - Different algorithms for weighted/unweighted graphs
- Find longest path between source and destination
- Find a path that visits each vertex exactly once
  - A, E, D, C, F, B, A in example on bottom right
- Path of minimum cost that visits each vertex once
  - A, E, D, C, B, F, A (cost 15) in example
- Assign no more than n different colors to vertices under constraint that no pair of connected vertices has the same color

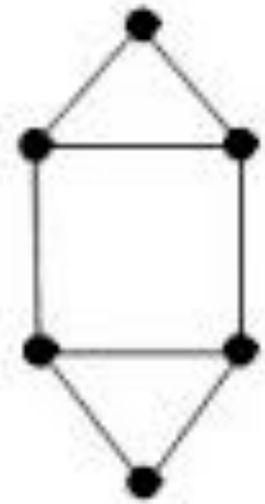


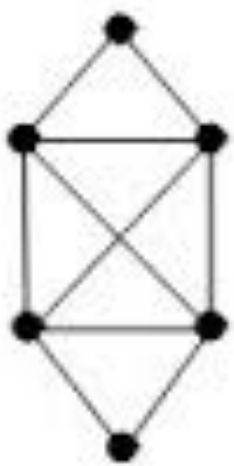


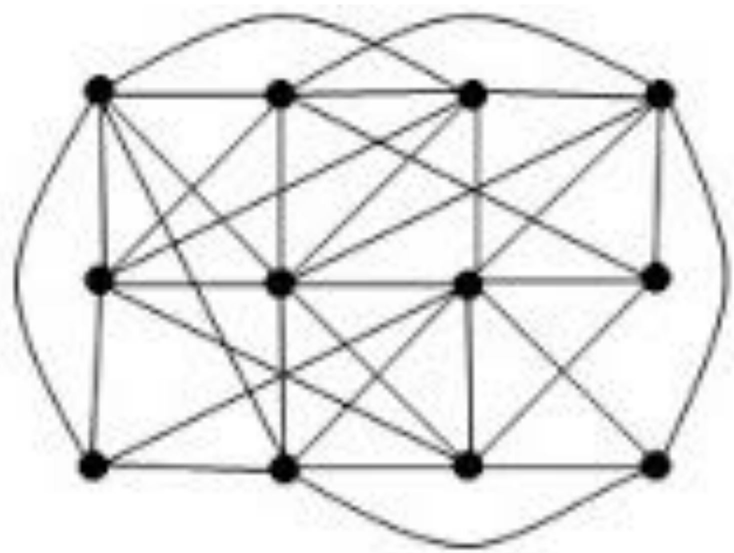
Some of these are easy (have fast algorithms), others hard (no known efficient solution)







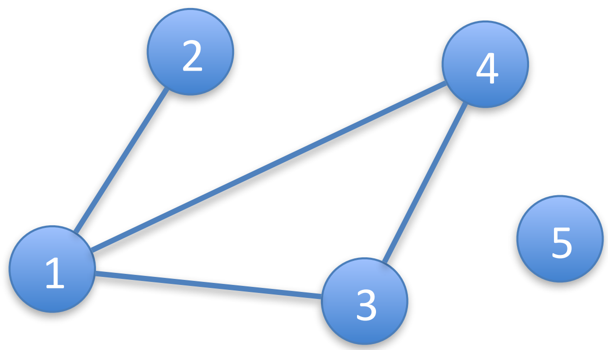




Representing graphs

- How can we represent general graph in Python?
  - Need to keep track of nodes
  - Need to keep track of edges
- Several ways to represent graphs have been developed
    - List of nodes and list of edges
    - Adjacency matrix
    - Adjacency lists
    - Dictionary of dictionaries
  - Efficiency of algorithms that solve graph problems can vary greatly depending on how graph are representated
  - a strong influence on choice is the fact that one of the most common things needed in graph algorithms is access to immediate neighbors of a node (nodes that are destinations of edges for which "current" node is source)

Adjacency matrix



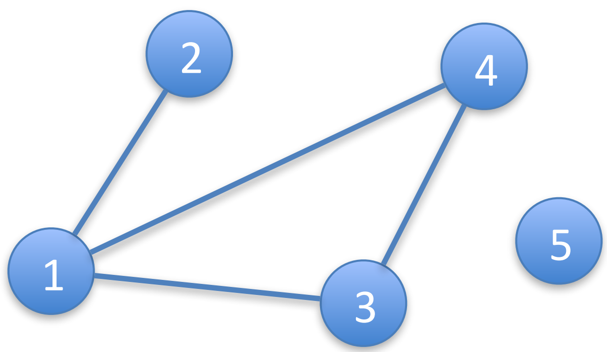
	1	2	3	4	5
1	False	True	True	True	False
2	True	False	False	False	False
3	True	False	False	True	False
4	True	False	True	False	False
5	False	False	False	False	False

- Appealingly simple to understand and implement
- Use, e.g. a list of lists containing True/False, 0/1, or similar
- NOT the most common graph representation for most problems. Can you think of a reason why?
  - Consider representing Facebook friends graph where each node is a FB user and an edge exists between two nodes whenever the two are FB friends.
  - One billion nodes. Adjacency matrix 1B x 1B in size! Your computer doesn't have that much storage. But FB graph ***can*** be represented in computer! How?
  - The 1B x 1B would be mostly False/0 – most people don't have huge number of friends. Should be representable in closer to 1B * median number of friends. Other representations enable this huge memory savings.

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	[]

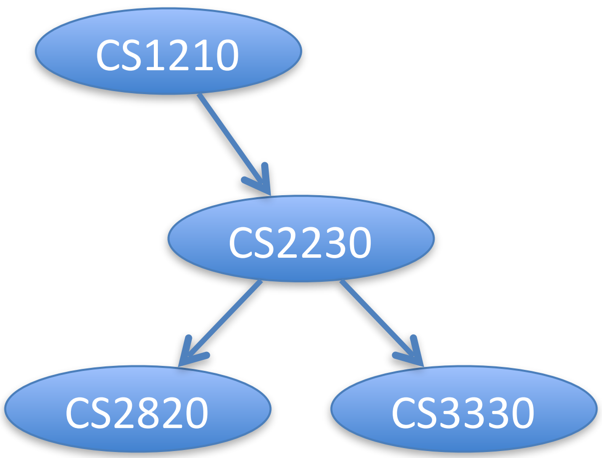
Compared to adjacency matrix:

+ Much less space (when, as is common, most nodes have only a small relatively small number of neighbors). Facebook graph. People have hundreds of friends, not many milliions

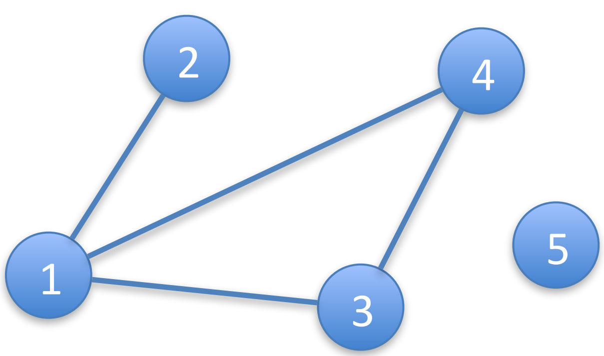
- Query of “does edge (i,j) exist?” not O(1). Must search list associated with node i to see if j is there. Turns out this is not crucial in many graph algorithms. (could address this using dictionary of dictionaries but often not necessary)

Adjacency list graph representation

Suitable for both undirected and directed graphs
(and can be use for weighted graphs as well)



KEY	VALUE
CS2230	[CS2820, CS3330]
CS2820	[]
CS1210	[CS2230]
CS3330	[]



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

An **adjacency list** representation for undirected graphs in Python

Two classes: Node and Graph

basicgraph.py

Node

- properties:
  - name : string
  - status: string (we'll use this to "mark" nodes during traversals)
- methods
  - getName
  - __repr__ : we'll print nodes as <*name*>

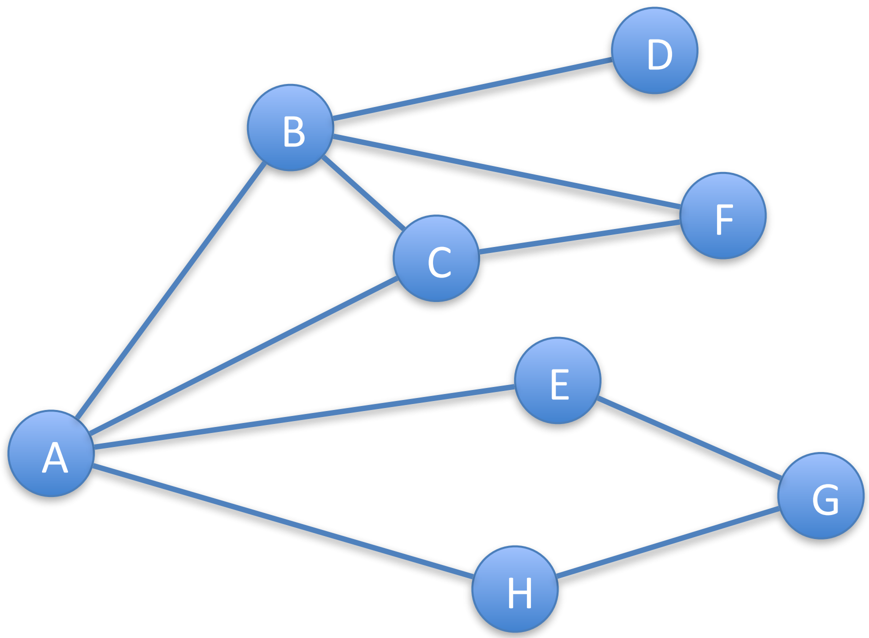
Note: in your HW8 you'll add one or more additional properties that help with traversing/walking through graphs to solve specific problems

Adjacency list representation for undirected graphs

Graph

- properties
  - nodes: a list of Node objects
  - 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__

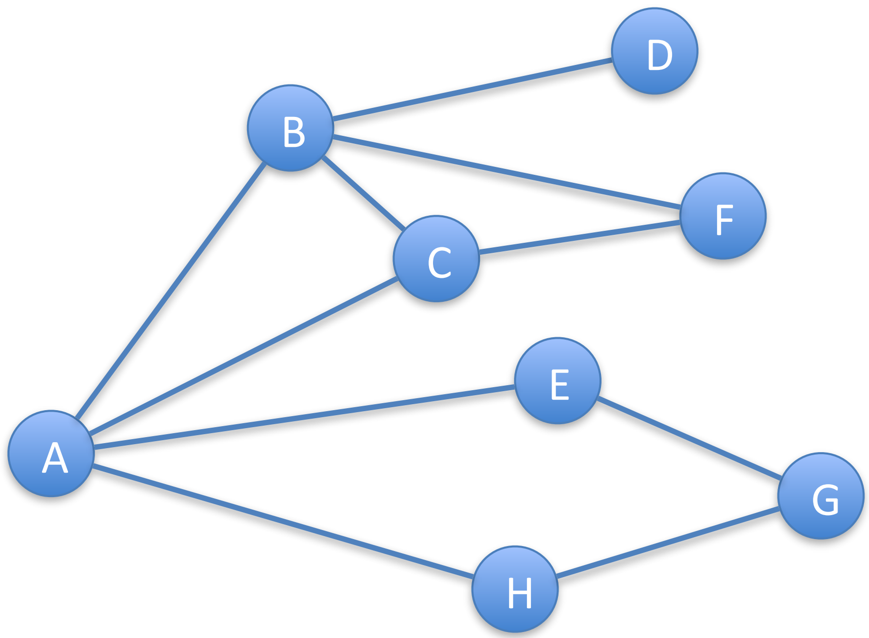
basicgraph.py



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

Note: 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



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]

As I've said, 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 are: depth-first search and breadth-first search. I will present 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 and extend* it in HW8.

Word ladder puzzles

CAT	CAT	CAT
???	COT	COT
???	???	DOT
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!

Next time

- Graph traversals
  - Breadth first search
  - Depth first search
- HW 8