**Graphs**

**Intro**

**Goals**

* Learn what a graph is
* See lots of real-world examples
* Code a graph implementation
* Check if two nodes are connected

**What is a Graph?**

Graphs are like trees, except they can contain loops (“cycles”).

Also, the relationships can be directed or un-directed.

**Terminology**

**Node (or Vertex)**

basic unit

**Edge (or Arc)**

connects two nodes

**Adjacent**

two nodes are *adjacent* if they’re connected by an edge

**Adjacency List**

for a given node, a list of every node to which it is directly connected

**Examples**

**Food Chain**

This graph is **directed**, showing “who eats whom”

Penguins’ adjacency list: [Squid, Krill]

**Facebook Friends/LinkedIn Connections**

This graph is **undirected** and is not **connected**

Harry’s adjacency list: [Ron, Hermione, Neville]

Hermione’s adjacency list: [Harry, Ron, Neville]

**Processes**

Making Pancakes:

Don’t want to do a step until the necessary prerequisites are done!

Similar idea for manufacturing processes, supply chains, etc.

**Markov Chains**

Other Markov chains: states of health and disease, finance

**More Terminology**

**Vertex or Node**

basic unit

**Edge**

connects two nodes

**Adjacent**

two nodes are *adjacent* if they’re connected by an edge

**Adjacency List**

for a given node, a list of every node to which it is directly connected

**Weight (optional)**

each edge can have a weight (ex: price, or distance)

**Airline Route Map**

Each node is an airport. Each edge is a flight.

The weight of each edge is the price.

What is the cheapest way to go from Atlanta to Oakland?

**Carpooling**

Each node is a rider, and edges represent possible carpooling matches. Only two people can carpool together at a time. How can we match the maximum number of pairs of riders?

(A possible, non-optimal solution.)

There exists a solution where everyone gets a pair. Can you find it?

**Graphs**

* Graphs are often used to model relationships between things
* Trees are **connected**, **acyclic** graphs, and most often directed
* All trees are graphs, but not all graphs are trees
* Trees can have hierarchy, graphs do not

**Linked Lists, Trees, and Graphs**

Linked lists, trees, and graphs are all structures that have a relationship, much like squares, rectangles, and parallelograms do. A linked list is a special, more-restricted form of a tree, and a tree is a special, more-restricted form of a graph.

**Linked List**

Nodes have 0 or 1 child; connected, acyclic, and directed

**Tree**

Nodes have 0+ children; connected, acyclic, and often directed; if directed, only one designated root node

**Graph**

Nodes have 0+ connections; cyclic or acyclic; directed or undirected; disconnected or connected; optional weights

There are other possibilities, including:

* there are “circular linked lists,” where the linked list can contain a cycle (A points to B points to C which points to B). These do not have tails, as there’s no single end-point.
* there are “forests,” which are collections of acyclic graphs but without a single root node. This essentially is a set of trees, hence a “forest.”

**Graphs in Python**

**Friend Graph**

**Node and Graph Class**

*graph-demo/friends.py*

**class** **PersonNode**(object): *"""Node in a graph representing a person."""* **def** \_\_init\_\_(self, name, adjacent=**None**): *"""Create a person node with friends adjacent"""* self.name = name **if** adjacent: **assert** isinstance(adjacent, set), \ "adjacent must be a set!" self.adjacent = adjacent **else**: self.adjacent = set() **def** \_\_repr\_\_(self): *"""Debugging-friendly representation"""* **return** "<PersonNode: *%s*>" % self.name

*graph-demo/friends.py*

**class** **FriendGraph**(object): *"""Graph holding people and their friendships."""* **def** \_\_init\_\_(self): *"""Create an empty graph"""* self.nodes = set() **def** \_\_repr\_\_(self): **return** "<FriendGraph: *%s*>" % [n.name **for** n **in** self.nodes] **def** add\_person(self, person): *"""Add a person to our graph"""* self.nodes.add(person) **def** set\_friends(self, person1, person2): *"""Set two people as friends"""* person1.adjacent.add(person2) person2.adjacent.add(person1)

**Demo: *friends.py***

*graph-demo/friends.py*

harry = PersonNode("Harry") hermione = PersonNode("Hermione") ron = PersonNode("Ron") neville = PersonNode("Neville") trevor = PersonNode("Trevor") fred = PersonNode("Fred") draco = PersonNode("Draco") crabbe = PersonNode("Crabbe") goyle = PersonNode("Goyle") hogwarts = FriendGraph() hogwarts.add\_people([harry, hermione, ron, neville, fred, trevor, draco, crabbe, goyle]) hogwarts.set\_friends(harry, hermione) hogwarts.set\_friends(harry, ron) hogwarts.set\_friends(harry, neville) hogwarts.set\_friends(hermione, ron) hogwarts.set\_friends(neville, hermione) hogwarts.set\_friends(neville, trevor) hogwarts.set\_friends(ron, fred) hogwarts.set\_friends(draco, crabbe) hogwarts.set\_friends(draco, goyle)

**Is Ron connected to Trevor (the toad)?**

Task: Write a function that checks if two people are connected.

*graph-demo/friends.py*

**class** **FriendGraph**(object): **def** are\_connected(self, person1, person2): *"""Are two people friends? Breadth-first search."""* to\_visit = Queue() to\_visit.enqueue(person1) seen = set() seen.add(person1) **while** **not** to\_visit.is\_empty(): current = to\_visit.dequeue() print "**\n**checking", current **if** current **is** person2: **return** **True** **else**: **for** friend **in** current.adjacent - seen: to\_visit.enqueue(friend) seen.add(friend) print "added to queue:", friend **return** **False**

This is a *breadth-first* search (would be *depth-first* if we used a stack)

[**+Ron**] [**+Harry, +Hermione, +Fred**]

[**+Ron**] [**+Harry, +Hermione, +Fred**] [Hermione, Fred, **+Neville**]

[**+Ron**] [**+Harry, +Hermione, +Fred**] [Hermione, Fred, **+Neville**] [Fred, Neville]

[**+Ron**] [**+Harry, +Hermione, +Fred**] [Hermione, Fred, **+Neville**] [Fred, Neville] [Neville]

[**+Ron**] [**+Harry, +Hermione, +Fred**] [Hermione, Fred, **+Neville**] [Fred, Neville] [Neville] [**+Trevor**]

[**+Ron**] [**+Harry, +Hermione, +Fred**] [Hermione, Fred, **+Neville**] [Fred, Neville] [Neville] [**+Trevor**] []

**Is Ron connected to Crabbe?**

Follow same steps as before, but at the end:

[**+Ron**] [**+Harry, +Hermione, +Fred**] [Hermione, Fred, **+Neville**] [Fred, Neville] [Neville] [**+Trevor**] []

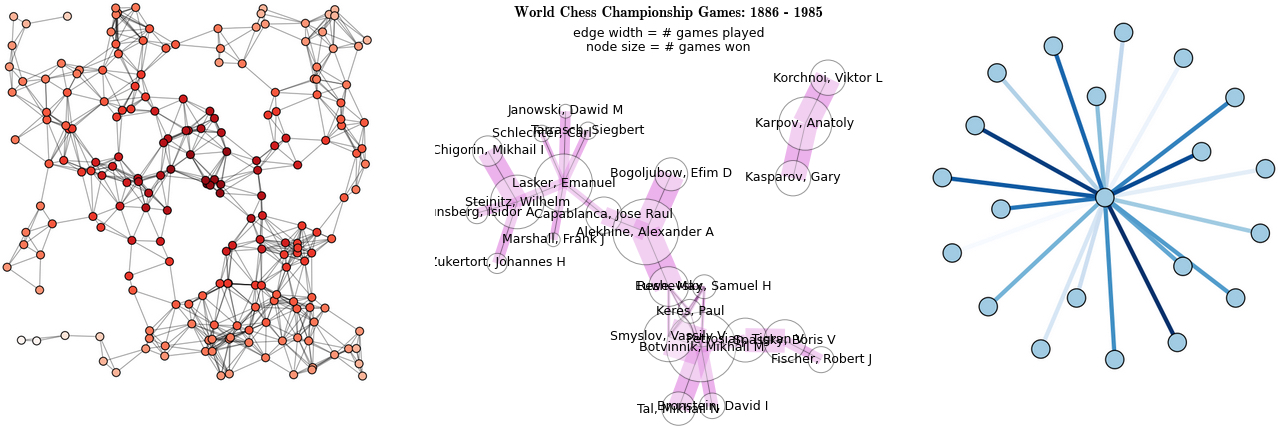
**Recursive Solution**

*graph-demo/friends.py*

**class** **FriendGraph**(object): **def** are\_connected\_recursive(self, person1, person2, seen=**None**): *"""Are two people friends? Recursive depth-first search."""* **if** **not** seen: seen = set() **if** person1 **is** person2: **return** **True** seen.add(person1) *# Keep track that we've visited here* **for** person **in** person1.adjacent - seen: **if** self.are\_connected\_recursive(person, person2, seen): **return** **True** **return** **False**

This is a *depth-first* search (using the call stack)

**Further Study**



* Python Library: [NetworkX](http://networkx.github.io/documentation/latest/index.html)
* [Problem Solving with Algorithms and Data Structures](http://interactivepython.org/courselib/static/pythonds/index.html) (awesome FREE book!)
* Graph Database: Neo4j
* Joe Celko, *SQL for Smarties* (graphs and trees in SQL)