Graph Theory: An Introduction to Theory and Simulation in Python

Michael Hughes

January 15, 2009

1 Background

Many people might think that graph theory is the study of visual representations of data, like line charts. But that is not the type of graph we're concerned with in this chapter.

Instead, we consider the *mathematical objects* known as graphs. In mathematics, a *graph* represents pairwise relationships between discrete elements in a collection. We call each element in the collection a *vertex* or a *node*. We call the collection of vertices the vertex set and denote it as V. If there exists a relationship between a pair of nodes, we say that an *edge* exists between them. If there is an edge from node v to node w, we say (v, w) is an edge. The collection of edges is denoted E.

For example, consider a graph that represents highway travel possibilities. Cities like Denver, Salt Lake, and Tokyo could be nodes in the graph, and an edge could exist between cities if a highway connects them directly. In our example, there would likely be an edge between Denver and Salt Lake City, but no edge between Denver and Tokyo.

Graph theory is the branch of mathematics that studies the structure, properties, and behavior of graphs. A thorough introduction to graph theory is a little too much for this chapter to offer. For a quick primer, visit Wikipedia: `wikipedia.org/wiki/Graph_(mathematics)`. We only scratch the surface of some important concepts here.

There's a lot of vocabulary in graph theory. We summarize a few key background concepts here:

We call a graph a *simple graph* is a graph which has no self loops and no multiple edges. A self loop is an edge that connects a vertex to itself. A self-loop for vertex a would be the edge (a, a). This means the edge set is NOT a multi-set (there are no repetitive entries in the set E) and contains no edges such that $(a, a) \in E$

A *regular graph* is a graph in which the number of edges incident to every node is uniform. Another way to say this is that the *degree* of each node in a regular graph is the same. If the uniform degree of the graph is some integer k, we call the graph a k-regular graph.

A *complete graph* is a graph in which each node has an edge connecting it to every other node. Another way to think of this is that every edge that can exist in the graph (excluding self loops and multiple edges) does exist.

We can prove that a complete graph is regular. First, we know that for a single node to be "complete", it must be connected to every other node in the graph. If the graph has totally n nodes, this means each node must connect to all other $n-1$ nodes. If we let k be the degree of each node, then for a graph to be complete $k = n-1$ must be the case for every node. This implies k is uniform for all nodes, so a complete graph must be regular.

A *path* is a sequence of adjacent edges in the graph. If two edges are *adjacent*, there is a node which they both are incident to.

A *cycle* is a path which starts and ends at the same node (e.g. a path that loops or cycles back to its starting point).

A graph is *connected* if there is a path from every node to every other node.

A *tree* is a graph which is connected but contains no cycles. Alternatively, a tree is a graph in which any two nodes have exactly one path between them.

Simply put, a *forest* is a graph which has no cycles and is disconnected. Alternatively, we can think of a forest as a graph made up of several disjoint trees. If a tree (or more generally, a component) is *disjoint* from the rest of a graph, this means the component is connected within itself and contains no edge connecting it to other nodes outside the component.

2 Computing with Graphs

To represent a graph in Python, we must choose an appropriate data structure that allows us to relate vertices and edges in an efficient manner. We choose to implement a graph as a dictionary of dictionaries. A graph maps a vertex v to an inner dictionary which maps a vertex w to an edge if $(v, w) \in E$. Each vertex is considered as an object with a label attribute. Each edge is represented as a tuple which contains two vertex objects.

Here's a summary of numerous graph methods we've created

1. `get_edge(v,w)` takes two vertices `v,w` and returns the edge between them if it exists and `None` otherwise.

   To access an edge between given vertices in the graph object, we use the vertices as keys for the outer and inner dictionaries. So any edge can be accessed with the command `self[v][w]`. If the edge does exist, the result of this statement can easily be returned. However, if such an edge doesn't exist, we must catch the `KeyError` exception and return `None` instead. A simple `try` statement is sufficient for this operation.

2. `remove_edge(e)` takes an edge `e` and removes all references to it from the graph.

   To remove an edge, we simply use the `del` command, as in `del self[v][w]` where `v,w` are easy extracted from the input edge `e`. Remember, since a graph is undirected, we must also perform `del self[w][v]` to completely eradicate all references to the edge.

3. `vertices()` returns a list of all edges in the graph.

   This is easily accomplished by obtaining a list of the keys to the outer dictionary, as in `self.keys()`.

4. `edges()` returns a list of all edges in the graph.

   This can be accomplished by grabbing a list of all vertices (using `vertices()`) and then looping through all pairs of these and attempting to access their edge using `get_edge()`. If the result of this command is not `None` (meaning the edge exists), then we can add the edge to the overall set of edges. Note that it's important to use a `set` in this case because a given instance of an edge is stored multiple times in the dictionary structure, so we don't want to track these multiple copies.

5. `out_vertices(v)` takes a Vertex and returns a list of adjacent vertices.

   We can access a dictionary of v's edges by calling the outer dictionary with `v` as a key. Next, we can simply use the `keys()` method on this inner dictionary to obtain all nodes that v is adjacent to.

6. `out_edges(v)` takes a Vertex and returns a list of the edges incident to v

   This can be done similarly to `out_vertices(v)`, except instead of calling the `keys()` method for v's inner dictionary we call `values()`. This will return a list of all edges contained in the inner dictionary, which are by definition connected to v.

7. `add_all_edges()` will make a complete graph from the vertices in the graph object

   To build a complete graph, we obtain a vertex list using `vertices()` and then iterate over all pairs of these vertices, creating an edge for every pair.

3 Constructing a Regular Graph

Constructing a regular graph of given size n and uniform degree k is not a trivial task.

First, we realize that constructing such a graph is not always possible. So we define and prove the necessary preconditions. Of course, we can see that for any (simple) graph, there cannot be more edges than nodes. Quantitatively, it must be the case that $k < n$. However, there are other cases when constructing a *regular* graph is not possible. We define and prove this condition below.

An (n, k)-regular graph can only exist if $nk/2$ is an integer

Proof. Any graph must have a whole number of edges in order to be a graph. A graph defined by parameters (n, k) will have $\frac{nk}{2}$ edges. This is the case because each of the n nodes will each have k incident edges, so there are nk instances of incidence. Each edge is incidence to exactly 2 nodes by definition. Therefore, there are $nk/2$ edges in a k-regular graph, and this must be an integer for the graph to be constructible. □

With this requirement in our grasp, we can develop an algorithm to reliably construct a regular graph.

In plain English, the basic idea is to always try to connect nodes that have the least number of edges. The procedure starts with a node v with the most number of edges remaining and then proceeds to add all of its edges. This edge creation process always attempts to pair v first with the remaining node with the most number of edges remaining, then with the next most desperate node, and so on. This ensures no one gets left out. We repeat this procedure until all nodes have k edges.

In the actual implementation, we keep track of each node's edge count by placing a node in a list with other nodes of the same edge count. The big picture data structure then is a list of lists, where the outer list L has a list $l_i$at each index i holding all nodes that have exactly i incident edges. We refer to each l_i list as an "edge bin", since we can imagine it as a bin holding all nodes with exactly i incident edges.

To construct the graph, we use the following algorithm:

```
Input: Uniform degree $k$; List of edge bins $L$
Output: Set of edges
foreach Edge Bin $b$ in $L$, lowest first do
    foreach Vertex $v$ in $b$ do
        remove $v$ from $b$;
        while $v$ has less than $k$ edges do
            foreach Edge Bin $b'$ in $L$, lowest first do
                foreach Vertex $w$ in $b'$ do
                    if no_edge_exists($v$,$w$) then
                        add_edge($v$,$w$);
                        move $w$ from $b'$ to $b' + 1$;
                    end
                end
            end
        end
    end
end
```

Algorithm 1: Regular graph construction algorithm

A rigorous proof of correctness is beyond the scope of this work, but we have tested the algorithm for many k and n values and are confident that it works well.

4 String Representation of Graphs

For our `Graph` object to be versatile and robust, we need a proper implementation of `__repr__`, so that we can print out a representation of a graph as a string to the console and then later reconstruct that same graph using only the string given earlier.

We realize that the constructor for `Graph` uses the syntax `Graph(V, E)`, where `V` is the vertex set of the graph and `E` is the edge set. We can easily get the vertex set of a graph object `g` by printing the list `g.vertices()`. Similarly, we can get the edge set with `g.edges()`. Thus, our solution is a single line of code if we harness the power of Python string formatting.

```
def __repr__(self):
    """return a string representation of this object that can
    be evaluated as a Python expression"""
    return 'Graph(%s, %s)' % ( repr(self.vertices()), repr(self.edges()) )
```

4.1 Preventing Redundant Vertex Creation

A problem related to graph re-creation is that we'd like the clone to actually be a reference to the original underlying data structures rather than a copy. Concretely, if we have a graph `g` and we use `repr` and then `eval` to create a graph `h` exactly like `g`, we'd like to make sure that node `v` of `g` is the same as node `v` in `g`. To do this, we need to only allow one instance of a node with a given label to exist at a single time. If the user attempts to create another instance of an existing vertex, we must return a reference to the one that already exists.

A simple way to accomplish this is to first create a global, static data structure (probably a list) with in the `Vertex` class definition that tracks all vertices that exist for a given Python execution. Next, we must write the code that checks this list and modifies it at the appropriate times. This involves overriding the `__new__` constructor of `Vertex`. Each time the user attempts to create a vertex, `__new__` is invoked, so in our modified version of `new` we search the existing node list to see if a node with the given label already exists. If so, we return the existing node. If not, we create a new vertex, add it to the list of existing vertices, and return it.

The argument could be made that if we're concerned about avoiding reference detachment, we should implement a similar construct for the `Edge` class as well. We leave this implementation as an exercise to the interested reader.

5 Directed Graphs

Up to this point, we've considered all graphs as *undirected*. This means we have considered edges as having no starting point and no ending point. In other words, the relationship between the vertices is reciprocal (as in v is a friend of w and w is a friend of v). However, we can imagine a situation in which we'd like to model pairwise relationships which aren't reciprocal. For example, if we use a graph to model a prehistoric food web, we'd like to show that Tyrannosaurs can eat Triceratops, but not vice versa. This type of graph is known as a *directed* graph.

To implement a directed graph in Python, we can create a class called `Digraph` which inherits all the basic structure and features of the `Graph` class. However, we must override some functionality to make the distinction between directed and undirected edges. Notably, when we add or remove edges, we only need to deal with (v, w) rather than both (v, w) and (w, v).

We also need to develop some new methods for `Digraph`. Significantly, we can now provide a methods which return lists of the incoming edges and incoming nodes for a given vertex.

For the `in_edges` method, which takes a vertex v and returns the list of edges which end at v, we first obtain a list of all the edges by calling `edges()`. Next, we iterate through all these edges and extract the second vertex w in their underlying tuple, which is the node the edge points in to. If this vertex w is equal to v, the edge belongs to v's incoming edge set and should be added to the list.

The `in_vertices` method takes a vertex v and returns a list of the immediate neighbors of v that have edges which end at v. To operate, this method internally calls `in_edges` to retrieve a list of v's incoming edges. We then cycle through all the edges in this list, extract the starting node from each one, and add this start node to a list which is eventually returned when the method call finishes.

Directed graphs are useful in many applications. We'll set this concept aside for now, however, and return to the world of undirected graphs for the remainder of this chapter.

6 Random Graphs

A popular area of study within graph theory considers the properties of *random* graphs. There are many different ways one could think of to construct a graph probabilistically. One of the most simple and most fascinated was first documented by mathematicians Paul Erdos and Alfred Renyi in the 1950s. The Erdos-Renyi random graph model assigns each possible edge in a n-vertex undirected graph a fixed probability p of existence. Each edge's existence is determined independently of its neighbors, though every edge uses the same value of p.

Implementing this model in Python is very straightforward, especially because we already have a procedure for adding all edges to the graph. We need only slightly modify this `add_all_edges` code so that for each possible pairing of distinct vertices, we generate a random value r between 0 and 1 and allow the edge to exist if r is less than the threshold probability p given as a parameter. One caveat: we must be careful not to consider a pair of vertices more than once (that is, we should test only (v, w) and not both (v, w) and (w, v)). We're using undirected graphs for this model, so the edge is the same regardless of the ordering of vertex names.

A simple way around this is to make sure the inner loop of the pairwise matching algorithm always stays above (or below) the outer loop. A working example is given below:

```
nodecount = len(allnodes)
for i in range(nodecount):
    for j in range(i+1, nodecount):
        randfloat = random.random()
        if ( randfloat < p ):
            e = Edge( allnodes[i], allnodes[j] )
            self.add_edge( e )
```

The value of j is always greater than i, so we are never at risk of encountering the same vertex pairs twice.

7 Connected Graphs

One of the most fascinating properties of Erdos=Renyi random graphs has to do with connectivity. Recall that in a connected graph, a path exists between all possible distinct node pairs. Before we can explore the application to random graphs further, we must develop an algorithm for determining if a graph is connected. We'll focus on undirected graphs, because Erdos-Renyi graphs are undirected.

The most straightforward method would be simply to start at a particular vertex s and make sure we can find a path from s to all other vertices in the graph. If we can find a way from s to any point t and from s to any other point u, it follows that a path from t to u must exist (through s). This is a simple proof that our algorithm will be correct (provided we code it correctly!).

Let's explore the vertices which s can reach using a technique known as Breadth-First Search. The basic idea here is that we first find all vertices which are 1 step away from s, then all vertices that are 2 steps away, and so on.

Functionally, we implement this procedure using a queue. To start, we add some arbitrary node to the queue. Next, we enter the loop routine. Here, we pop the next element from the queue, obtain a list of all its neighbors, add the onese we haven't yet visited to the queue, and repeat until the queue is empty. Finally, to determine if the graph is connected, we check to see if the size of the visited list is the same as the size of the graph.