Lecture Notes, CSE 232, Fall 2014 Semester

Dr. Brett Olsen

Week 6 - Introduction to Graph Theory

Graphs OK, this week we're going to start on graph theory, and we're actually going to continue this topic next week as well, the only topic that I'm going to explicitly cover for multiple weeks. The reason for this is two-fold: First, graphs are important. A ton of different interesting problems can be usefully modeled as graphs and solved using known graph algorithms. Secondly, graphs are complicated. There's a lot of different kinds of graphs, there's several different ways to store graphs internally, and there's an enormous number of useful and interesting graph algorithms, many of which we still won't have time to cover during this class.

So, what's a graph? A graph $G = (V, E)$ is a collection of *vertices* V and *edges* E composed of pairs of vertices from V. We can use graphs as a way of representing relationships between elements: element A and element B are connected in some way. Maybe the elements are cities, and the edges denote roads going between those cities, with the graph then representing a road network. Or maybe the elements are people, and the edges show relationships between people, so the graph represents a social network. But in general, we're talking about elements and the relationships between those elements. This is a very broad definition, and precisely because it is so broad, it's a very useful framework for fitting lots of different kinds of problems into.

In [3]:
```
%matplotlib inline
```

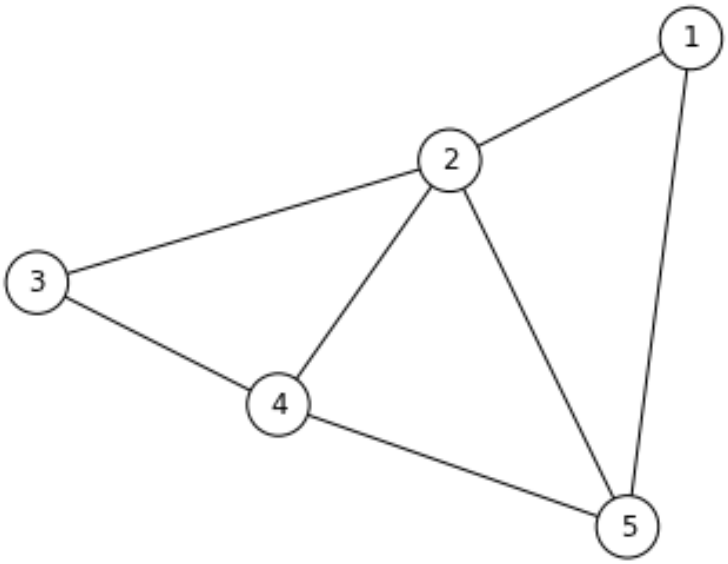
In [5]:
```
import networkx as nx
import matplotlib.pyplot as plt
```

In [9]:
```
G = nx.Graph()

#Add vertices
for i in range(1, 6):
    G.add_node(i)

#Add edges
G.add_edge(1, 2)
G.add_edge(1, 5)
G.add_edge(2, 3)
G.add_edge(2, 4)
G.add_edge(2, 5)
G.add_edge(3, 4)
G.add_edge(4, 5)

#Plot the graph
nx.draw_graphviz(G, prog='dot', node_color='white', node_size=700, arrows=False)
```



Types of Graphs The first step in trying to solve any problem by treating it as a graph is to correctly define the kind of graph we want to model the problem with. What are the vertices going to represent? How are the edges defined? What's the general shape of the graph? There are a number of different variations of graphs which are going to be treated differently when solving them.

Directed vs. Undirected Edges are pairs of vertices that are connected in the graph. Suppose we have some edge, say, $E(1, 2)$, which says that element 1 is connected to element 2 in our graph. If the presence of this edge implies that the reverse edge $E(2, 1)$ is also in the graph, then we call that graph undirected, meaning that edges are symmetric - there's no direction to the edge. If element 1 is connected to element 2, then 2 must be connected to 1. If that's not the case, then some of the edges must be one-way only.

Suppose we're drawing a social graph between employees at some company, where edges are drawn between employees who work together on a project. That relationship is symmetric, so we would end up with an undirected graph. If instead we wanted to draw a graph showing the employee hierarchy, we could draw edges where the first element is the boss of the second element, and we would have the head of the company at the top, with edges from them to their direct underlings, and so on. These edges are not transitive: X being the boss of Y doesn't mean that Y is the boss of X, and so this would be a directed graph.

In general, most of the algorithms we'll discuss are meant to work on undirected graphs, but there's a couple of important ones that are specifically designed for directed graphs.

Unweighted vs. Weighted Suppose we want a graph of the road network between cities in Missouri. If we only wanted to calculate which cities were connected to which other cities, then we could do this on the kinds of graphs we've already discussed. More likely, though, we might want to do something like calculate the path between two cities that has, say, the shortest driving time. To do this, we need a weighted graph, where each edge has in addition to its pair of vertices, a weight associated with it, which in this case would

give the driving time along that road. Then the driving time along any particular path between two cities can be calculated by summing the weights of the edges along that path.

So any time there's some information associated with a particular edge that isn't just which vertices are connected, we'll use a weighted graph to hold that data. For road networks, this might be driving time or distance. For social networks, this might be the strength of the relationship between two people. For flow networks like an electrical circuit, this might be the capacity of each edge, and so on.

Simple vs. Non-Simple There are a couple of types of edges that can cause problems in a lot of algorithms if they're present: self-loops, an edge (x, x) where a vertex is connected with itself, and multiedges, where the same edge (x, y) occurs more than once. When these types of edges are present, they take some special care in handling them appropriately so that infinite loops, etc., are avoided. As such, any graph that avoids them is called simple, and we'll usually take care to deal only with simple graphs when at all possible.

Cyclic vs. Acyclic The next distinction we want to look at is whether our graph contains any cycles or not. What's a cycle? A cycle is any path that leads from an element back to itself. Self-loops, where an element connects direct with itself, are obviously cycles. But in general, we're interested in larger cycles. So in our example above, 2 is connected to 4, which is connected to 3, which is connected back to 2, and we've identified a loop or a cycle in the graph.

Trees are a subset of acyclic graphs which are also undirected and connected (if unconnected, the graph is equivalent to a set of trees, usually called a forest). Trees have the useful property that there's only one path between any two nodes. When we talked about suffix tries (a type of tree) a few weeks ago each node represented a particular string which could be represented by the unique path from the root to that node.

If instead we look at directed acyclic graphs, usually abbreviated DAG, these are a natural way to represent priority or scheduling problems: task A must be performed before task B, while task C must be performed before task D. These types of graphs are directed because the relationship between tasks is asymmetric, while they're acyclic because any cycle in the graph would imply a contradiction - that some task must be performed before itself. The classic problem on these graphs is called topological sorting, which asks for a feasible order to perform the tasks while obeying the restraints.

Storing Graphs When working with graphs, there are several different ways to represent them internally:

Object & Pointer At the most basic level, we can represent each vertex as an instance of a Vertex or Node object which contains first a label marking which element that vertex represents and secondly a list of pointers that point to other nodes which this vertex is connected to. Then we can walk around through the graph along edges by following those pointers. This is a feasible way to represent fairly small graphs - it's not very memory efficient and it's not very fast, so for larger graphs we want to use a more abstract method.

Adjacency Matrix The next method, called an adjacency matrix, consists of a 2-dimensional V by V array M, where V is the number of vertices in the graph. If edge (x, y) is in the graph, then $M[x, y] = 1$, and if it is not in the graph, then $M[x, y] = 0$. Here, for example, is the adjacency matrix for the graph shown above:

```
x=  1   2   3   4   5
   /-----------------
y=1|0   1   0   0   1
  2|1   0   1   1   1
  3|0   1   0   1   0
  4|0   1   1   0   1
  5|1   1   0   1   0
```

This representation has a couple of advantages: it's very rapid to check whether a particular edge exists in the graph, and it's very fast to add or remove edges. We can also check whether the graph is undirected by whether the associated adjacency matrix is symmetric. Weighted graphs can be represented by matrices where we place the edge weight instead of 1 in the matrix for edges that are present.

However, the downside is that it takes a very large amount of memory to store the matrix, and often we're working with sparse matrices where there aren't many edges compared to vertices, so most of the matrix is just zeros and a lot of the space isn't needed.

Adjacency List In general, the best representation (barring a few specific applications) for a graph is as an adjacency list. This consists of a list of lists. We start off with a list L of length V, element j of L then contains a list of all vertex indices k where (j, k) is an edge in the graph. Here is the adjacency list for the graph shown above:

```
1 -> [2, 5]
2 -> [1, 3, 4, 5]
3 -> [2, 4]
4 -> [2, 3, 5]
5 -> [1, 2, 4]
```

This structure is slightly less efficient for quickly identifying if a particular edge (i, j) is in the graph, but is much more space efficient, faster at traversing the graph, faster to find all of the edges of a vertex, and so is generally preferred. If you don't have a good reason not to use an adjacency list, you should use one.

Traversing a Graph The most fundamental graph theory problem is graph traversal - take a graph and a starting vertex and systematically explore the graph until we've reached every possible vertex from our starting position. There are two concerns when traversing a graph. First, *efficiency*. We don't want to revisit vertices that we've already visited. We'll be wasting our time when we could be exploring new parts of the graph (and if you're not careful, you could even get stuck in an infinite loop). Second, *correctness*. We need to do our exploration systematically so that we haven't missed any vertices that we could have visited.

That suggests that we should start with two data structures. First, we'll have a boolean array of size V that marks whether we've visited each vertex or not, that will start off filled with False elements. Secondly, we'll have some data structure (there are a couple of options here) that marks nodes that are available to be visited - we know we can reach those from the area of the graph that we've already traversed - and this starts off empty.

How can we use this to comprehensively traverse the graph? First we pick a starting vertex and set its visited flag as True. Then we look in our adjacency list for all vertices that are connected to our starting vertex and put those into the structure of available vertices. Then we repeat the following process:

1) Pick a vertex from the available vertices to be traversed.
2) Set that vertex as visited.
3) Look in the adjacency matrix for all elements connected to this vertex.
4) Add all of the connected vertices that haven't already been visited and aren't already available to the available vertices.

When there are no more available vertices, then we've traversed the graph and we're done. There are two important ways to do this, which depend on how we pick the next vertex to visit.

Breadth First Search Let's suppose we use a queue to store the available vertices. So any time we remove an available vertex from the queue, it will be the earliest one that was added. Let's explore our sample graph using this method, starting from vertex 1, and drawing the traversal tree as we go.

Vertex 1 is connected to 2 and 5, so we add those to our queue. Then we remove 2 from the queue as our next available vertex. Connected to 2 are 3 and 4, which get added to the queue. We then remove 5 as the earliest added vertex and it has no new connections, so we remove similarly 3 and 4, and we've traversed the whole graph. Vertices 2 and 5 were reached from vertex 1, while 3 and 4 were reached from vertex 2.

This is what's called a breadth-first search. We start off by visiting all neighbors of the starting vertex. Then we visit all neighbors of neighbors of the starting vertex, at a distance 2 from the start, and then all that are 3 edges from the start, and so on. This algorithm has the particular benefit that it tells us the minimum number of edges between the starting vertex and all other vertices in the graph.

Depth First Search Another alternative would be to use a stack to store the available vertices, so that when we choose an available vertex, it will be the most recently added one. This means that when we move to a new vertex, we'll continue exploring neighbors of that vertex before we look at any other neighbors of the parent vertex. Following this method gets us much deeper traversal trees - for the sample graph, we simply go from 1 to 2 to 3 to 4 to 5 and then have explored all vertices. This is called a depth-first search.

One benefit of a depth-first search is that it's trivially implementable with recursion. Instead of keeping an explicit stack of available vertices, we simply pick the first unvisited vertex connected to our current one, and call a new depth-first search on that vertex. This is still using a stack, but now it's the recursive stack of function calls, and so it's abstracted away for a simpler implementation. Be a little careful with this recursive method because you're limited by the maximum depth of the function call stack.

That's as far as we're going to go today. Even just being able to construct and traverse a graph is enough to be able to solve a number of very interesting problems, including the one I've assigned for homework this week that you can get started on during the lab section. Next week we'll go into some more complicated graph algorithms, and many of them will be dependent on good implementation of these BFS and DFS algorithms.

Lab Section Again, we're going to work on this week's homework problem during lab section: GCJ 1C 2012 - Diamond Inheritance