# Lecture Notes, CSE 232, Fall 2014 Semester

Dr. Brett Olsen

## Week 7 - More Graph Theory

OK, last week we gave an introduction to graphs, talked a little bit about translating problems into graphs, and then discussed graph traversal algorithms - breadth-first and depth-first search. Today we're going to cover as many interesting graph algorithms as we can!

```
In [1]: %matplotlib inline
```

```
In [2]: import networkx as nx
        import matplotlib.pyplot as plt
```

**Graph Algorithms** Let's start off with some ways we can use the graph traversal algorithms we discussed last week. First, some more obvious applications:

**Connected Components (Undirected)** Consider how to identify the number of connected components in an undirected graph, where "connected components" means that within each connected component there is a path from any vertex to any other vertex. So in an unconnected graph, there is not always a path between any two vertices. Think about how a graph traversal will work on an unconnected graph. If we start from a particular vertex, we are guaranteed to visit every vertex that can be reached from our starting point - which defines a connected component. So if there are still unvisited vertices, there must be at least one more connected component. We can count how many there are by running a graph traversal starting from unvisited vertices until all are visited.

**Flood Fill (Implicit)** Flood fill is a common problem on 2-d grids: we have some cells on a grid (say pixels of an image) and we want to replace a connected region of color or type with another color. We can solve this by representing the grid as an undirected graph where vertices are gridpoints and edges are present between vertices where those cells are neighbors of each other and the same color. Then a connected region of color on the grid is equivalent to a connected component on the graph.

**Shortest Path (Unweighted)** Suppose we're given an unweighted graph and we're asked to find a shortest path from some source vertex to some target vertex. Because the graph is unweighted, the shortest path is equivalent to the path with the shortest number of edges. And we learned last week that a breadth-first search will visit vertices in order of their distance from the starting vertex. So we can start a breadth-first search from the source vertex and keep going until we find the target vertex. We'll have to modify the search a little bit so we can reconstruct the distance or path, depending on what's desired. We can do this by modifying the boolean array keeping track of whether we've visited each vertex or not to either: 1) keep track of the parent vertex in the traversal tree or 2) keep track of the distance of each vertex from the source. With (1) we can quickly reconstruct the path identified by our BFS, and with (2) we can just pull the distance of our target out of the array.

**Topological Sort (DAG)** A topological sort of a directed acyclic graph (DAG) orders the vertices of the graph so that every directed edge goes from an earlier vertex to a later vertex. Every DAG has at least one but possibly more valid topological sorts. As I mentioned last week, for graphs where edges represent precedence constraints (job x must be done before job y), a topological sort will give a valid processing order.

More interestingly, suppose that we want to find some particular path between x and y. If we perform a topological sort, we know that the only vertices that can be a part of the path are those between x and y in the sort. Any vertices before x can't be reached starting from x, while any vertices after y can't end up at y, so we can restrict our search to a smaller subgraph.

How do we construct this order? Well, first we can note that the first element in the sort must have no incoming edges. So we'll start by iterating through all vertices in our adjacency list and counting the number of incoming edges. Then we'll push all vertices with no incoming edges onto a stack. Then we repeat the following process:

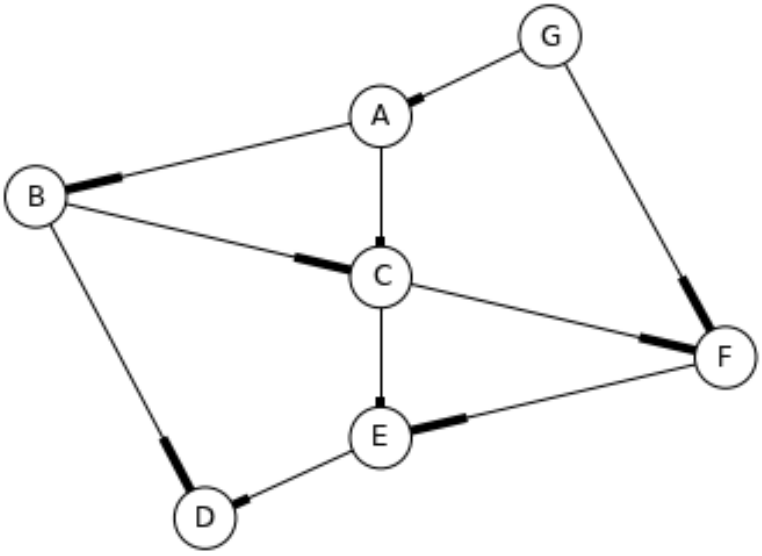
1) Pop off a free vertex $X$ from the stack. It's the next element in the topological sort.
2) Go through all edges outgoing from $X$ and remove them, decrementing the incoming edge count for each vertex pointed to by $X$.
3) Push any vertices that now have no incoming edges onto the stack.

Let's look at an example:

```
In [6]: G = nx.DiGraph()

        G.add_nodes_from(['A', 'B', 'C', 'D', 'E', 'F', 'G'])
        G.add_edges_from([['G', 'A'],
                          ['G', 'F'],
                          ['A', 'B'],
                          ['A', 'C'],
                          ['C', 'F'],
                          ['C', 'E'],
                          ['F', 'E'],
                          ['B', 'C'],
                          ['B', 'D'],
                          ['E', 'D']])

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




The initial incoming edge count for vertices from A to G is `[A:1, B:1, C:2, D:2, E:2, F:2, G:0]`. Vertex G has no incoming edges, giving us a stack of `[G]`. We pop G off the stack and add it to our sort order, which is now `[G]`. Then we take the edges from G to A and F and remove them, decrementing the counts in our array for each of those vertices, to `[A:0, B:1, C:2, D:2, E:2, F:1, G:0]`. This reduces the incoming edges to A to zero, so it gets pushed onto the stack. Repeating, we go through vertices:

A, giving new vertex counts `[A:0, B:0, C:1, D:2, E:2, F:1, G:0]`

B, giving `[A:0, B:0, C:0, D:1, E:2, F:1, G:0]`

C, giving `[A:0, B:0, C:0, D:1, E:1, F:0, G:0]`

F, giving `[A:0, B:0, C:0, D:1, E:0, F:0, G:0]`

E, giving `[A:0, B:0, C:0, D:0, E:0, F:0, G:0]`

and finally D, which finishes us and we have the final sort of `G, A, B, C, F, E, D`. For this graph, this is the only valid topological sort. For graphs where there are multiple possible sorts, which one you get is going to depend on how you pick which vertex with no incoming edges to use at each point.

**Minimum Spanning Tree (Weighted)** A spanning tree of a graph G is a subset of the edges E that form a tree connecting all the vertices V. We saw, when looking at breadth-first and depth-first searches that we obtained traversal trees describing the way in which we reached each vertex. These traversal trees are spanning trees of the graph, containing a subset of all edges in the graph. For an unweighted graph, each traversal tree has the same number of edges, and so is the same size. But for a weighted graph where different edges have different costs, different spanning trees will have different costs based on which edges they contain. How could we find the minimum spanning tree, which is the cheapest way to connect all vertices together?

Well, suppose we have a list of all edges in the graph, sorted from smallest weight to largest. Our minimum spanning tree is a subset of all those edges. We start off with none of them, so every vertex is disconnected from any other vertex. Then adding any edge to our tree would connect two vertices, increasing the connectivity by the same amount. So which edge should we pick? The one with the smallest weight. So we take the edge with the smallest weight, and add it to our list of edges in the minimum spanning tree. Then we move on to the next smallest. Should we add this one too? Well, we only want to add it if

it increases the connectivity of our graph. If it does, then we'll add it; otherwise we discard it. Then we keep going through all the edges from smallest to largest, adding them to our spanning tree if they increase connectivity and discarding them if we don't.

The key to this algorithm (called Kruskal's algorithm) is being able to identify whether an edge increases the connectivity of the graph. We can do this using the union-find structure that we discussed in the second week of class. If you don't remember, this gives us access to two methods: find(x), which returns a value that represents the group to which x belongs, so if find(x) and find(y) return the same value, then x and y are in the same group, and union(x, y), which merges the groups to which x and y belong so that they are now in the same group.

So we'll use this union-find structure to represent the connectivity of the graph. Start off with every vertex in a separate group. Then we can check if a particular edge between x and y increases the connectivity by checking whether find(x) and find(y) return the same value. If they do, we add that edge to our spanning tree, and then merge x and y because that connection has been made. With this data structure, minimum spanning trees can be found in $O(eloge)$, relative only to the number of edges in the graph, and so is quite fast for sparse graphs.

**Shortest Paths (Weighted)** So for unweighted graphs we can use a breadth-first search to find a shortest path between any two nodes, because the depth in the BST traversal tree is equivalent to the minimum distance. How do we solve this problem in a weighted graph? A simple BST won't work - we might get a path with two edges of weight 10 and 15 rather than a shorter path with three edges of weight 5, 6, and 8. The algorithm we'll discuss is called Dijkstra's algorithm, and given a starting vertex $s$, it will give us the shortest distance to every vertex in the graph.

We'll start with two data structures: an array $M$ that will store the minimum distance to each vertex we've found so far, and a priority queue $P$ that will contain pairs of (distance to $s$, vertex), keyed on distance, so that the closest vertex to $s$ will always be at the top of the priority queue. We initialize the distance array to be MAXINT for all vertices except $s$, which is set to 0. Similarly, the priority queue initially contains only the base case (0, $s$). Then we repeat the following steps until the priority queue is empty:

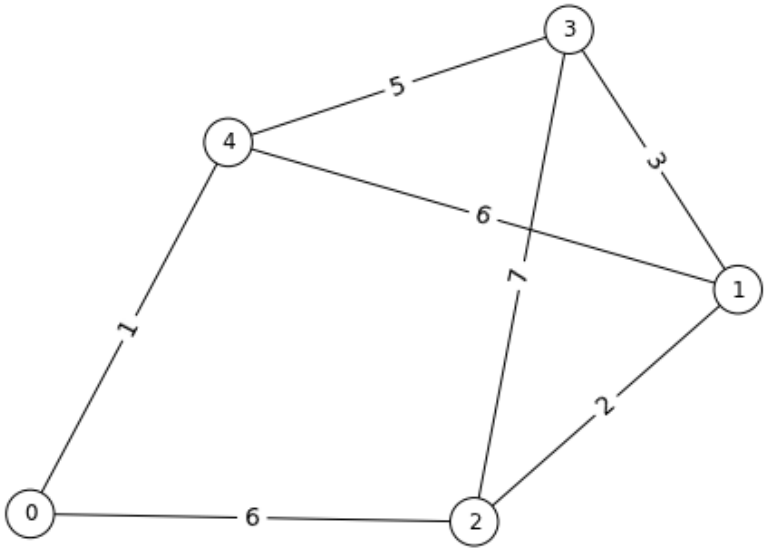
1) Remove the *minimum distance* pair ($d$, $v$) from $P$. If $d$ is larger than or equal to $M[v]$, then we already know at least as good of a path to get to $v$ from $s$, and we skip it and move on to the next pair. Otherwise, continue.

2) Go through each edge of $v$ to target vertex $t$ of weight $w$. The distance to those target vertices is $d+w$ - the cost of getting to $v$ plus the cost of going through that edge to $t$. If $d + w < M[t]$, then this is a shorter path to $t$ than we had before, and we update the distance array. When we update, we also add the pair ($d + w$, $t$) to the priority queue.

So at each point, we're finding the closest possible vertex to $s$, adding that distance to our graph, and then using its edges to update the cost of paths to any vertices it's adjacent to. Let's look at an example.

```
In [25]: G = nx.Graph()

         G.add_nodes_from([0, 1, 2, 3, 4])
         G.add_weighted_edges_from([(0, 2, 6),
                                    (0, 4, 1),
                                    (1, 2, 2),
                                    (1, 3, 3),
                                    (1, 4, 6),
                                    (2, 3, 7),
                                    (3, 4, 5)])
         #Plot the graph
         plt.figure(figsize=(8, 5))
         pos = nx.graphviz_layout(G)
         nx.draw(G, pos, node_color='white', node_size=700)
         edge_labels = dict([((u,v,),d['weight']) for u,v,d in G.edges(data=True)])
         data = nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=14)
```




Let's calculate the shortest distance from vertex 2 to each other vertex. We'll initialize our distance array with $M[2] = 0$ and our priority queue will contain $\{(0, 2)\}$.

Then we remove the first element in our priority queue and look at the edges from vertex 2, which go to 0, 1, and 3. We update our distance array with new distances $M[0] = 6, M[1] = 2, M[3] = 7$, while adding them to our priority queue, which now contains (in order) $\{(2, 1), (6, 0), (7, 3)\}$.

Again, we remove the first element of the priority queue, which is (2, 1). So we go to vertex 1 at a distance of 2. Vertex 1 has three edges to 2, 3, and 4. The distance from 2 to each of those going through 1 is 4, 5, and 8. For vertex 2, this is larger than the minimum distance we already have (0), so we skip it. For vertex 3, we have a current distance 7, but if we go through 1, it is distance 5, so we update $D[3] = 5$ and add the pair to our priority queue. For vertex 4, we have no current distance, so we set $M[4] = 8$ and update $P$, giving us a queue of $\{(5, 3), (6, 0), (7, 3), (8, 4)\}$.

We remove (5, 3) and go through the edges of vertex 3, but none of them give us an improvement in minimum distance, so we're done.

We remove (6, 0) and go through the edges of vertex 0. The edge to vertex 4 gives us a new shortest path, so we update $M[4] = 7$ and update our priority queue to $\{(7, 3), (7, 4), (8, 4)\}$.

We go through the last three elements of our queue and find that all three of them are no better than the distances we've already calculated, so we do nothing and the algorithm ends, giving us a final distance array of $M = [6, 2, 0, 5, 7]$.

A couple of caveats with shortest path algorithms - be careful with negative weights. If there is a negative weight *cycle* - a path from a vertex back to itself that has a total weight less than zero, then the shortest paths problem is undefined - you could arbitrarily loop through this cycle as many times as you want to get an arbitrarily low value.

**Others** I don't have time to go through everything interesting today, but here's some graph problems and algorithms that are useful and worth looking up if you have time:

- Floyd Warshall's - a very fast to implement DP algorithm that provides the shortest distance between all pairs of vertices
- Prim's algorithm - an alternative way to find a minimum spanning tree

- Articulation points and bridges - Find vertices or edges that when removed, disconnect the graph
- Maximum flow - given a weighted graph with edges given weights corresponding to their capacity, detemine the maximum capacity that can flow from a source vertex to a target vertex

**Lab Section** Again, we're going to work on this week's homework problem during lab section:
Timus Online Judge #1210 - Kind Spirits