Lecture 7

Graphs: A *graph* is a pair (V, E), where V is called the set of *vertices* and E is called the set of *edges*. Edges are pairs of vertices and represent some form of connection between two vertices. Many pieces of data can be thought of in a graph representation. For example, we can think of the Facebook graph: there would be one vertex for each person on Facebook, and an edge exists between two people if they are friends. Such graphs are called *undirected*, meaning that the the connection an edge represents between two vertices is mutual: if you and I form an edge on Facebook then I am your friend, and you are also my friend. The Twitter graph on the other hand is not undirected. Suppose an edge (x,y) means that x follows y on Twitter. Then John may follow Bob, but Bob may not follow John, which means the graph would have the edge (John, Bob) but not (Bob, John). Graphs in which an edge (x,y) are treated as being different from (y,x), such as the Twitter graph, are called *directed*.

Graph examples:

- Social networks (Facebook, MySpace, Twitter, Google+, ...). People are vertices, and edges mean friendship/following/etc.
- The web graph. Web pages are vertices, and an edge (x,y) means page x has a link to page y.
- Road networks. Street intersections are vertices, and an edge (x,y) means a person at intersection x can reach intersection y by a segment of road.
- Airport networks. Airports are vertices, and an edge (x,y) means there's a direct flight from x to y.
- Family trees. People are vertices, and an edge (x,y) means x is one of the parents of y.

Today we discuss algorithms to answer a few basic questions about graphs. Before doing that though, we have to discuss how the graph is given as input to the computer. The two most popular ways of specifying a graph are via an *adjacency matrix* or an *adjacency list*.

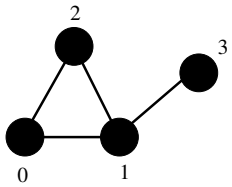

Figure 1: Example undirected graph.

Suppose the graph has n vertices, which we label from 0 to $n-1$. Then, an adjacency matrix is simply a `list` A of n `list`s, where each `list` is of size n. The entry $A[i][j]$ is `True` if the edge (i, j) exists in the graph, and otherwise it is `False`. For example, we could represent the graph in Figure 1 as follows:

$$\text{A} = [[\texttt{False}, \texttt{True}, \texttt{True}, \texttt{False}], [\texttt{True}, \texttt{False}, \texttt{True}, \texttt{True}],$$
$$[\texttt{True}, \texttt{True}, \texttt{False}, \texttt{False}], [\texttt{False}, \texttt{True}, \texttt{False}, \texttt{False}]].$$

In adjacency list representation, we have a variable A which is a `list` of n `lists`. The ith list contains a list of all vertices j such that (i, j) is an edge in the graph. So, we could represent Figure 1 in adjacency list representation as follows:

$$A = [[1, 2], [0, 2, 3], [0, 1], [1]].$$

Each representation has its own advantages, and the situation should dictate which one is best to use. For example, the adjacency matrix representation always requires $\Theta(n^2)$ memory, whereas the adjacency list representation can take much less memory if the graph has few edges. Meanwhile, the adjacency matrix representation can tell you whether i has an edge to j in $O(1)$ time (just look at $A[i][j]$), whereas it could take $\Omega(n)$ time in the adjacency list representation if i has many edges, since you might have to scan the entire ith list.

Graph exploration: Some basic questions concerning graphs are the following:

- **Connectedness:** Given vertices i, j, are they *connected*? That is, is there a *path* from i to j? A path is just a sequence of edges that share endpoints. In other words, can I start at i, follow some sequence of edges, then end up at j?
- **Connected components:** A connected a component of an undirected graph is a set of vertices that are all connected to each other via paths, and are not connected to anything else.
- **Shortest path:** Given vertices i, j, what's the shortest number of edges you have to cross to get from i to j?

Today we will look at shortest paths (a problem software like Google Maps needs to solve).

We will consider graphs where edges have *lengths*. For example, in a graph representing an airport network, edges have associated lengths corresponding to the amount of time it takes to fly from one airport to the next. Then, we might be interested in getting from one airport to another while minimizing total flight time.

A *weighted* graph is a triple (V, E, w), where w is a *weight* function. Each edge $e \in E$ has a weight $w(e)$, which may be positive, zero, or negative. Now how can we find the shortest path from one vertex to the others in such a graph? This can be done using recursion and memoization. The non-recursive, iterative implementation of this approach is called the *Bellman-Ford* algorithm.

The basic idea is to create a recursive function `shortestPathHelper`(x, y, t) which finds the shortest path from x to y which takes at most t steps. One option is that it is the same as the shortest path taking at most $t - 1$ steps, and the other is that we should travel to some vertex z first in $t-1$ steps then take the edge (z, y) in the tth step. We recurse on both options and take the better of the two, and we use memoization to make the function faster. Note that if it's possible to get from x to y at all, then it is possible to do so in $n - 1$ steps, where the graph has n vertices, so the length of the shortest path from x to y is `shortestPathHelper`(x, y, n-1).

In some graphs though, such as those in Figure 1, there is no shortest path from some vertex to another. For example, to go from vertex 0 to 3, we could take the route $0 \to 1 \to 3$ for a total length of $-1 + 1 = 0$. However note that the cycle $0 \to 1 \to 2 \to 0$ has a total length of -1. Thus, by repeatedly going on this cycle over and over again, we can make our total length arbitrarily small before finally heading over to vertex 3. Thus, in essence, the length of the shortest path from

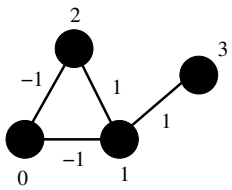

Figure 2: The numbers on edges are weights. This graph has a negative weight cycle $0 \to 1 \to 2 \to 0$.

0 to 3 is $-\infty$. We modify our `shortestPath` code to detect such negative-weight cycles. If any such cycle is found, starting from our starting vertex x, then we return -1. Otherwise we return a list with all the shortest path distances.

How can we detect a negative-weight cycle? Let such a reachable cycle be $v_0, v_2, \ldots, v_{k-1}$. Let $d[u]$ be the shortest path distance from x to u taking at most $n-1$ steps. For the sake of contradiction, assume that we cannot improve the distance to any of the v_i by looking at paths of length n. That means that $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$ for all i (with the understanding that v_{-1} is just v_{k-1}). Summing up all these inequalities gives

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Since each $d[v_i]$ appears exactly once in the summations on both sides, we can cancel to then get

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

This is a contradiction, since we assumed that this cycle had negative total weight.

Our implementation now follows.

```
# returns length of shortest path from x to y using at most t steps
def shortestPathHelper(B, x, y, t, mem, seen):
    if t == 0:
        if x == y:
            return 0
        return float('infinity')
    elif seen[y][t]:
        return mem[y][t]

    seen[y][t] = True

    # first option: do it in t-1 steps
    ans = shortestPathHelper(B, x, y, t-1, mem, seen)

    # second option: go to a vertex z that has an edge to y first, in
    # at most t-1 steps, then take the edge (z, y)
```

```
    for p in B[y]:
        z = p[0]
        weight = p[1]
        val = shortestPathHelper(B, x, z, t-1, mem, seen)
        ans = min(ans, weight + val)

    mem[y][t] = ans
    return ans

# A is the adjacency list of the graph
# A[u][i][0] is the ith neighbor of vertex u, and A[u][i][1] is the
# weight of the edge (u, A[u][i][0])
#
# returns a list L so that L[j] is the length of the shortest path
# from x to j, assuming no negative-weight cycle is reachable from
# i. returns -1 if a negative-weight cycle is reachable from i.
def shortestPath(A, x):
    mem = []
    # mem[i][j] is float('infinity') if we can't get from x to i in at
    # most j steps. Otherwise, it's the length of the shortest path
    # from x to i taking at most j steps.
    for i in xrange(len(A)):
        mem += [[float('infinity')]*(len(A)+1)]

    seen = []
    # seen[i][j] is True if we've already filled in mem[i][j] and is
    # False otherwise
    for i in xrange(len(A)):
        seen += [[False]*(len(A)+1)]

    # B is an inverse adjacency list.  B[i] is a list of all vertices
    # j such that (j, i) is an edge, plus the weight of the edge
    B = []
    for i in xrange(len(A)):
        B += [[]]
    for i in xrange(len(A)):
        for p in A[i]:
            # p is the pair [j, weight(i, j)]
            B[p[0]] += [[i, p[1]]]

    # check if a negative weight cycle is reachable from x
    for z in xrange(len(A)):
        val1 = shortestPathHelper(B, x, z, len(A) - 1, mem, seen)
        val2 = shortestPathHelper(B, x, z, len(A), mem, seen)
        if val2 < val1:
```

```
        return -1

L = []
for y in xrange(len(A)):
    L += [shortestPathHelper(B, x, y, len(A) - 1, mem, seen)]
return L
```

Naïvely one could say that the running time of the algorithm is $O(n^3)$: in the memoized helper function there are n choices for y, n choices for t, and the loop over B[y] might run for $n-1$ steps. For some graphs this could happen, but in fact the algorithm's running time is $\Theta(n(m+n))$, where the graph has m edges. nm is always at most n^3 since m is at most n^2, but it can be a lot faster if the graph doesn't have too many edges. The reason the running time is $\Theta(n(m+n))$ is the following. Look at the `for` loop "`for p in B[y]`" in `shortestPathHelper`. The total number of (p, y) values this loop executes with is exactly m: each (p, y) pair corresponds to an edge in the graph. Then, there are n possible values of t, giving nm.

The n^2 term comes from there being n possible values of both y and t, though this term can be removed with a better implementation, which we give below. The below implementation is an iterative implementation of the approach above, but written iteratively instead of recursively. It is known as the Bellman-Ford algorithm. The code is also quite a bit shorter than the recursive implementation given above.

```
def bellmanFord(A, x):
    # E is a list of edges with weights
    E = []
    for i in xrange(len(A)):
        for p in A[i]:
            E += [[i] + p]
    # dist[i] is the length of the shortest path to i
    dist = [float('infinity')]*len(A)
    dist[x] = 0
    for i in xrange(len(A) - 1):
        for e in E:
            u = e[0]
            v = e[1]
            weight = e[2]
            dist[v] = min(dist[v], dist[u] + weight)

    # look for negative weight cycles
    for e in E:
        u = e[0]
        v = e[1]
        weight = e[2]
        if dist[u] + weight < dist[v]:
            return -1

    L = []
    for i in xrange(len(A)):
        L += [dist[i]]
    return L
```