Lab 12

Exercise 1: Modify the code for the Bellman-Ford single-source shortest path algorithm so that it takes as input the origin x and destination y and returns the actual shortest *path* from x to y and not just the length of the shortest path. You can assume that y is reachable from x and that there are no negative weight cycles.

Example solution: The basic idea is to add a list `from` of length n, where n is the number of vertices. `from`[u] is the vertex right before u on the shortest path from x to u (if u is x, then we set from[u] to -1).

```
def bellmanFord(A, x, y):
    # 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)
    from = [-1]*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]
            if dist[u] + weight < dist[v]:
                from[v] = u
                dist[v] = 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

    # build the shortest path, from the end to the beginning
    L = []
    at = y
    while at != -1:
        L += [at]
        at = from[at]
    L.reverse()
    return L
```

Exercise 2: Modify the code for Floyd-Warshall (either the recursive or iterative approach) to keep track of a matrix `next`[][] so that `next`[u][v] is some intermediate vertex on the shortest path from u to v (or the Python value `None` if there is no intermediate vertex). Now write a recursive procedure `findPath` which takes u, v, and the `dist` and `next` matrices, and returns the shortest path from u to v as a list. For example, if the shortest path from 0 to 3 is $0 \to 1 \to 3$, then `findPath` should return [0,1,3].

Example solution: In fact, `findPath` does not need the `dist` matrix.

```
# findPath should be called with the next list that's generated in
# floyd Warshall
def findPath(u, v, next):
    # if there's no intermediate vertex, we just directly take the
    # edge (u,v)
    if next[u][v] == -1:
        return [u, v]
    # otherwise we recurse. in the second recursion we don't take the
    # 0th element so that next[u][v] doesn't show up twice in the
    # final output
    return findPath(u, next[u][v], next) + findPath(next[u][v], v, next)[1:]

def floydWarshall(w):
    # now dist is a copy of the weight function
    dist = w[:]
    next = []
    for i in xrange(len(w)):
        next += [[-1]*len(w)]

    for k in xrange(len(w)):
        for u in xrange(len(w)):
            for v in xrange(len(w)):
                if dist[u][k] + dist[k][v] < dist[u][v]:
                    dist[u][v] = dist[u][k] + dist[k][v]
                    next[u][v] = k
    return dist
```

Exercise 3: Modify the code for Floyd-Warshall to return -1 if there exists a negative weight cycle somewhere in the graph. Otherwise, it should return a matrix of distances as before.

Example solution: There's a negative weight cycle if and only if some vertex has a negative distance to itself.

```
def floydWarshall(w):
    # now dist is a copy of the weight function
    dist = w[:]
    for k in xrange(len(w)):
        for u in xrange(len(w)):
            for v in xrange(len(w)):
                dist[u][v] = min(dist[u][v], dist[u][k] + dist[k][v])
    for i in xrange(len(w)):
        if dist[i][i] < 0:
            return -1
    return dist
```