CSCI 4041, Fall 2018, Written Assignment 9

1. (*Modification of Bellman-Ford Algorithm*)

```
def find_cycle(G, w, s)
    Initialize-Single-Source(G, s)
    cycle_map = {}
    cycle_list = []
    for i = 1 to (|V| -1)
        for each edge (u, v) in G.E
            Relax(u, v, w)
    for each edge (u, v) in G.E.
        if v.d > u.d + w(u, v)
            cycle_map[v] = 0
            while cycle_map[v] != 2
                v = v.prev
                if v in cycle_map:
                    cycle_map[v] += 1
                    if v not in cycle_list
                        cycle_list.append(v)
                else:
                    cycle_map[v] = 0
            return cycle_list
    return Null
```

Once a cycle has been detected in this algorithm (i.e. when v.d > u.d + w(u, v)), we trace the path back from v, adding each vertex to a hash table with a value of one. Once a vertex has already been reached, we add one to its value in the hash table. At this point, if we have already reached that vertex before, then we know that this vertex is a part of some cycle, then the only thing to do is to add this vertex to our cycle_list and keep back tracking and adding one to every vertex in the path while also adding those vertices to the cycle_list (since at this point every previous vertex is in the cycle) until we once again reach the initial vertex for a third time, ending the while loop. At this point we return the list.
This algorithm runs in O(VE) because the “negative-weight-cycle-checker” loop goes through at most E times, and the inner while loop goes through at most 2V times, assuming that very vertex is a part of the cycle.

2. (*Modification of Dijkstra's Algorithm*)

```
def most_reliable_path(G, r, s, f)     #where r is a matrix of probabilities, G is a list of vertices,
    for each vertex v in G.V.          #s is the starting vertex IN G, and f is the goal vertex IN G
        v.p = -∞
        v.prev = None
    s.p = 1
    Q = G.V.
    build_max_heap(Q)
    while len(Q) != 0
        u = heap_extract_max(Q)
        for i in range(len(u.adj))
            v = u.adj[i]
            if v.p < u.p*r(u, v)
                key = u.p*r(u, v)
                v.d = key
                v.prev = u
                i = Q.index(v)
                heap_decrease_key(Q, i, key)
    path = []
    u = f
    while u != None
        path.append(u)
        u = u.prev
    path.reverse()
    return path
```

3.

	A	**B**	**C**	**D**	**E**
A	0	6	3	6	5
B	4	0	7	10	9
C	1	3	0	5	4

D	-4	2	-3	0	-1
E	∞	∞	∞	∞	0