Problem Set 6

Name: Your Name

Collaborators: Name1, Name2

Problem 6-1. —

(a) **DAG Relaxation**: [(a, b), (a, d), (d, b), (d, e), (b, e), (b, c), (b, f), (e, f), (f, c)]

Dijkstra: [(a,b), (a, d), (d, b), (b, c), (b, e), (b, f), (f, c), (e, f), (c, e)]

(b)

v	a	b	c	d	e	f
$\delta(s, v)$	0	2	3	6	5	4

Problem 6-2. General shortest path problem in which we can use Bellman-ford, knowing that there is always some path between every two vertices with at most k edges, implies that $|V|$ is upper bounded by $|E| \rightarrow |v| = O(|E|)$, and we can perform **DAG Relaxation** for k levels.

- **If G has no negative cycles:** Let $v \in V$ be any vertex. Consider path $p = <v_0, v_1, ..., v_k>$ from $v_0 = s$ to $v_k = v$ that is a shortest path with minimum number of edges. No negative weight cycles $\rightarrow$ p is simple $\rightarrow k \leq |V| - 1$.
  After 1 pass through E, we have $d[v_1] = \delta(s, v_1)$, because we will relax the edge (v_0, v_1) in this pass and we can't find a shorter path than this shortest path. After 2 passes through E, we have $d[v_2] = \delta(s, v_2)$, because in the second pass we will relax the edge (v_1, v_2). After i passes through E, we have $d[v_i] = \delta(s, v_i)$. After $k \leq |V| - 1$ passes through E, we have $d[v_k] = d[v] = \delta(s, v)$. this takes $O(k|E|)$.
- **If G has negative cycles:** we can perform the previous algorithm and then do **Bellman-ford** check After k passes, if we find an edge that can be relaxed, it means that the current shortest path from s to some vertex is not simple and vertices are repeated. Since this cyclic path has less weight than any simple path the cycle has to be a negative-weight cycle.

Bellman-ford takes $O(|V|(|V| + |E|))$ as in this case we just relax on k levels and $|v| = O(|E|)$ so $O(K|E|)$.

Problem 6-3. construct graph $G_1 = (V, E)$, V is the clearings c_i with the oroginal elevations e_i and E the slopes, same way construct graph $G_2 = (V, E)$, V is the clearings c_i with the elevations e_i after all dynamite detonations and E the slopes, compine G_1 and G_2 in graph G with directed edges with zero-weight length from D_1 towards D_2.
Claim: G is a **DAG**.
proof: as Bames will only traverse slopes so as to decrease her elevation, implies that all edges go from a higher c_i to lower $c_j \Rightarrow$ directed, this also guarantees that Bames will not need to revisit a vertex more than once as he goes downhill only.

Run **DAG Relaxation** from L_1 to some vertex T connected to s_1 and s_2 with edge with zero-weight length in $O(|E| + |v|) = O(n)$.

Problem 6-4. construct graph $G = (V, E)$, V is the locations in which practile can exist, E is the transitions (l_i, l_j, w_ij) with directed edges from location i to location j.

•Add a supernode s connected to all vertex $v \in V \Rightarrow O(n) vertices and O(2n = O(n) edges)$.

•Run **Bellman-ford** from s to detect negative-weight cycles.

•if there is no negative-weight cycles negate all weights and run **Bellman-ford** again.

•if there is no positive-weight cylces then the force is conservative.

Problem 6-5.

Problem 6-6.

```python
def johnson(n, S):
    '''
    Input:  n | Number of vertices in the graph
            S | Tuple of triples (u, v, w) representing edge (u, v) of weight w
    Output: D | Tuple of tuples where D[u][v] is the distance from u to v
              |   or INF if v is not reachable from u
              |   or None if the input graph contains a negative-weight cycle
    '''
    D = [[INF for _ in range(n)] for _ in range(n)]

    Adj = [[] for _ in range(n)]
    w = {}

    for (u, v, weight) in S:
        Adj[u].append(v);
        w[(u, v)] = weight

    Adj.append([i for i in range(n)])
    for i in range(n):
        w[(n, i)] = 0

    def wbf(u, v): return w[(u, v)]

    bellf = bellman_ford(Adj, wbf, n)
    if bellf is None: return None

    h,_ = bellf
    def wdy(u, v): return w[(u, v)] + h[u] - h[v]

    for u in range(n):
        dy,_ = dijkstra(Adj, wdy, u)
        for v in range(n):
            if dy[v] < INF:
                D[u][v] = dy[v] - h[u] + h[v]

    D = tuple(tuple(row) for row in D)
    return D
```