Problem Set 6

Name: Thomas Ren

Collaborators: None

multicol multirow graphics subfigure

Problem 6-1.

(a) 1. **DAG Relaxation**:

iteration	$d(a,a)$	$d(a,b)$	$d(a,c)$	$d(a,d)$	$d(a,e)$	$d(a,f)$
0	0	∞	∞	∞	∞	∞
1	0	2	∞	∞	∞	∞
2	0	2	∞	6	∞	∞
3-5	0	2	3(3)	6	5(4)	4(5)
6-8	0	2	3	6	5	4
$\delta(a,v)$	0	2	3	6	5	4

Table 1: DAG Relaxation: For each iteration we change our estimation by one edge. During iteration *3-5*, we add a parenthesis after our estimated distance to show the iteration.

2. **Dijkstra:**

Node deleted from Q	$d(s,v)$					
	a	b	c	d	e	f
None	0	∞	∞	∞	∞	∞
a	0	2	∞	6	∞	∞
b	0	2	3(3)	6	5(4)	4(5)
d	0	2	3	6	5	4
e	0	2	3	6	5	4
f	0	2	3	6	5	4
$\delta(a,v)$	0	2	3	6	5	4

Table 2: Dijkstra

(b) The $\delta(s,v)$ are listed in the table.

Problem 6-2. First, we abstract the subject to determine which kind of problem we need to solve:

1. Graph is weighted directed, without specification, we consider it as a general graph which may contain negative-weight cycles.
2. We solve a single source shortest path problem**(SSSP)**.
3. We return a k-edge shortest distance $\delta_k(s, v)$, where path Π from s to v must be within k edges.

To solve this problem, we need to modify the **Bellman-Ford** algorithm to make it return k-edge distance instead of $|V| - 1$-edge distance, i.e. the graph duplication procedure.

- Refine the graph to eliminate unreachable vertices. Finish in $O(|V| + |E|)$-time
  - Run BFS from source node s to get a level list.
  - Construct a new graph G' with nodes that are at most k-edges away from s.
  - Since the graph G' is a connected, $|V| = O(|E|)$.
- Use graph duplication, only that this time we only duplicate k times designated by G^*. Finish in $O(k(|V| + |E|) = O(k|E|)$-time.
- Run our **modified Bellman-Ford**, here is **DAG Relaxation** on G^*. Finish in $O(k|E|)$-time.
- Running time analysis: The whole procedure runs in $O(|V| + |E| + k|E| + k|E|) = O(|V| + k|E|)$-time.
- Correctness: Our modified Bellman-Ford returns the shortest path for those vertices that are at most k-edges away and ∞ for those that are not reachable.

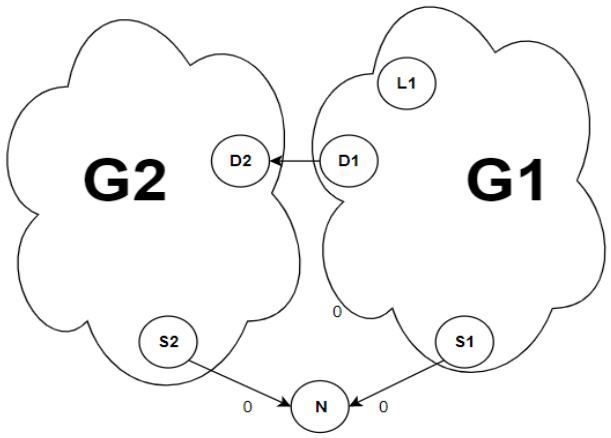

Figure 1: Dynamite Detonation

Problem 6-3. The only algorithm that we know about computing weighted shortest path is the **DAG Relaxation**, so we will use that.

To use DAG Relaxation, we need to build our graph to be a DAG(directed acyclic graph). For convenience, I draw a picture to illustrate the properties of our graph and how we can run DAG Relaxation on this graph. Show as **Figure 1**

Graph Construction:

- For graph *without denotation*, we denote this as **G1**.
  - The starting position L here is denoted by $L1$. Same to the ending position S being designated as $S1$.
  - For each clearing $c_i \in C$, we assign a vertex designated as $c_{i,1}$.
  - For every slope (c_i, c_j, l_{ij}), we add an edge $(c_{i,1}, c_{j,1})$ if $e_i > e_j$ or $(c_{j,1}, c_{i,1})$ if $e_i < e_j$, otherwise we do nothing, i.e. Bames doesn't traverse any flat slope. Besides, we add an edge weight lij to represent the length of that slope.
  - Denote the place with detonator as $D1$.
- For graph after Bames decided to detonate the dynamite, we denote this as **G2**.
  - G2 can only exists after Bames hit the detonator at place D1. So connect $(D1, D2)$ with weight 0 since hit the button will change the elevation *immediately*. Note that the edge is not reciprocal because we can't restore the dynamited elevations.
  - Construct vertices the same way as G1.
  - Construct edges the same way as G1, only this time we do with elevations e_i'.
- Add a node N that have connection edges $(S2, N)$ and $(S1, N)$, both with weights 0.

To prove that this graph cannot have cycles, i.e. is a **DAG**, we can easily prove by *contradiction*.

1. Suppose there exists a cycle $c_0, c_1, \cdots, c_n, c_0$.
2. According to our principles of graph construction, we know that for any two nodes c_i, c_j, we have $e_i < e_j$ iff $i > j$.
3. Thus we have $e_0 > e_1 > \cdots > e_n > e_0$.
4. However, we know that $c_0 > c_0$ is false, so contradiction holds. □

Thereby, we can just run DAG Relaxation on our constructed graph to know the shortest path from L to S by computing $\delta(L1, N)$.

Problem 6-4. The subject ask us to find a cycle which has zero-weight.
Recall that we can use **Bellman-Ford** to witness a negative-weight cycle, if $\delta_{|V|}(s,v) < \delta_{|V|-1}(s,v)$. The problem is that if we stay at v after the first time we relax v, it would be the same distance after we go through a zero-weight cycle.
To solve the problem we can modify the *graph duplication* procedure to make sure we won't stay at the same place after we relaxed this vertex. Easy to think, we can use BFS since in graph duplication, each level is determined by the number of edges rather than edge weights.
The procedure is as follows:

1. Run BFS to record the shortest edge distances from s to every other vertices. Designated by δ_v for node v.
2. Use graph duplication but do not connect edges between a node to its duplicate after δ_v for node v.
3. Run DAG Relaxation on the duplicated graph.

Our modified Bellman-Ford runs in $O(|V||E|)$-time. Since $|E| = O(n) = O(|V|)$, running time is $O(|V|^2) = O(n^2)$.
Then to check whether a zero-weight cycle exist, we check whether there exists the exact same distance after δ_v relaxations, i.e. if $\delta_{\delta_v}(s,v) = \delta_k(s,v), k = \delta_v, \cdots, |V|$ then there exists a zero-weighted cycle.

Problem 6-5. As usual, we abstract the information in the subject to show how to construct our graph and then apply an algorithm to fit the required running time.
Abstraction:

- •Intersections are vertices that contain certain information:
  - –elevation e_i.
  - –whether contains a gas station to fill up capacity.
- •Roads are edges which contains driving time to weight the edge.
- •The car's capacity is has g states, where g is bounded by $O(n)$.
- •There exists certain rules for the change of state:
  - –For directed edge $r_i = (x_i, x_j)$, capacity changes from g' to $g' - (e_j - e_i)$ if $e_i < e_j$, otherwise state stays at g'.
  - –For intersection that has a gas station, the capacity state at $x_{i,g}$ can be changed to $x_{i,g+1}$ at with time t_G.

So here is how we construct a graph:

- •For every intersection, we build g vertices designated by $x_{i,j}, i = 0, 1, \cdots, n-1, j = 0, 1, \cdots, g-1$. **Number bound**: $|V| = g \times n = O(n^2)$.
- •For every *undirected* road $r_i = (x_i, x_j)$, wlog assume $e_j > e_i$, we add directed edge $(x_{i,m}, x_{j,m-(e_j-e_i)\times t_G})$ for $m = (e_j - e_i) \times t_G + 1, (e_j - e_i) \times t_G + 1, \cdots, g$.
  And another directed edge $(x_{j,m}, x_{i,m})$ for $m = 0, 1, \cdots, g$.
  Since $|E|$ is bounded by $O(|V|)$, the edges so far are bounded by $O(2 \times g \times n) = O(n^2)$.
- •For a intersection x_i that contains a gas station, we add directed edges $(x_{i,m}, x_{i,m+1}$ for $m = 0, 1, \cdots, g-2$, each with weight t_G.
  For these gas station edges, the number of them is bounded by $O(n \times g) = O(n^2)$.

After these steps, we can simply run **Dijkstra** from s to solve a *single source shortest path problem (SSSP)* and return the minimum time cost while keeping a positive gas capacity in $O(|V| \log |V| + |E|)$-time.
Considering that our constructed graph has $|V| = O(n^2), \quad |E| = O(n^2)$, the only thing left is our **Priority Queue's size**.
Since our graph is strictly positive, we never visit an intersection twice, so our priority queue can be the number of vertices, i.e. $size(Q) = O(n^2)$.
Our Dijkstra algorithm runs in $O(n^2 \log n^2 + n^2)$-time, namely $O(n^2 \log n)$-time.

Problem 6-6. Everything describing the algorithm and the *pesudocode* can be found the *lecture note 14 and recitation 14*.
Here, we only present our implementation of **Python code**.

```
INF = 99999     # distance magnitudes will not be larger than this number

def convert_S_to_Adj_and_w(n,S):
    Node_list = [i for i in range(n)]
    # print(Node_list)

    # Initialize Adj and w
    Adj,w = {},{}
    for node in Node_list:
        Adj[node] = []
        w[node] = {}

    # Convert edges to Adj and w
    for (u,v,weight) in S:
        Adj[u].append(v)
        w[u][v] = weight

    return Adj,w

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)]
    ###################
    # YOUR CODE HERE #
    # Build Adjacency and w data structure
    Adj,W = convert_S_to_Adj_and_w(n,S)
    def w(u,v):
        return W[u][v]

    # Add a super node and connect them
    Node_list = [i for i in range(n)]
    Adj[n] = Node_list
    W[n] = {}
    for node in Node_list:
        W[n][node] = 0

    # Compute h[u] for node u using Bellman-Ford
    if bellman_ford(Adj,w,n) is None:
        return None
    else:
```

```
47          h, _ = bellman_ford(Adj,w,n)
48
49      # Reweight the graph
50      for u in W:
51          for v in W[u]:
52              W[u][v] += h[u]-h[v]
53
54      # Delete our super node
55      del W[n]
56      Adj.pop(n)
57      # Use dijkstra to conpute shortest path and reweight back
58      for u in W:
59          d,_ = dijkstra(Adj,w,u)
60          for v in range(len(d)):
61              if d[v] != INF:
62                  d[v] += h[v] - h[u]
63                  D[u][v] = d[v]
64
65
66      ###################
67      D = tuple(tuple(row) for row in D)
68      return D
```