CMPT 435L ALGORITHMS
Assignment 3

November 15th 2024

Jonathan "Grant" Bever

Link to github repo

PART 1: Using the Bellman-Ford dynamic programming algorithm for Single Source Shortest Path (SSSP)

The Bellman-Ford algorithm is an approach to finding the shortest paths from a single source vertex to all other vertices in a weighted directed graph.
The algorithm begins by initializing the distance to the source vertex as 1 and all other vertices as infinity. It then systematically refines these distance estimates. The algorithm has a time complexity of O(VE), where V is the number of vertices and E is the number of edges, and is particularly valuable because it can handle negative edge weights as long as there are no negative cycles.

After bringing our data into linked object representation we can apply the Bellman Ford algorithm to get this.

```
export function bellmanFord(graph: Graph, source: number):
   BellmanFordResult {
    const distances = new Map<number, number>();
    const predecessors = new Map<number, number | null>();
    const vertices = graph.vertices;
    const edges = graph.edges;
    // Initialize distances and predecessors
    vertices.forEach(vertex => {
        distances.set(vertex, Infinity);
        predecessors.set(vertex, null);
    });
    distances.set(source, 0);
    // Relax edges |V| - 1 times
    for (let i = 0; i < vertices.length - 1; i++) {
        edges.forEach(([from, to, weight]) => {
            const distFrom = distances.get(from)!;
            const distTo = distances.get(to)!;
            if (distFrom !== Infinity && distFrom + weight <
               distTo) {
            distances.set(to, distFrom + weight);
                predecessors.set(to, from); }
    }    }
```

```
    // Check for negative cycles
    let hasNegativeCycle = false;
    edges.forEach(([from, to, weight]) => {
        if (distances.get(from)! !== Infinity &&
        distances.get(from)! + weight < distances.get(to)!) {
            hasNegativeCycle = true;        }  });
```

This code begins by creating two Map structures: one for tracking the shortest distances to each vertex and another for storing the predecessor vertices that form the shortest paths. During initialization, all distances are set to Infinity except for the source vertex which is set to 0, and all predecessor pointers are set to null.

The second phase is the main relaxation phase, where the algorithm performs $|v|$ -1 iterations (V being the number of vertices) over all edges in the graph. Each iteration examines every edge and checks if taking that edge would result in a shorter path to its destination vertex. When a shorter path is found, both the distance to that vertex and its predecessor pointer are updated.

The final phase is the negative cycle detection, which makes one last pass through all edges in the graph. If during this pass the algorithm finds that any distance can still be improved it sets a hasNegativeCycle flag to true. This check works because after $|V|$-1 iterations, all legitimate shortest paths should have been found, so any further improvements indicate the presence of a negative cycle that could be traversed indefinitely to keep reducing the path length.

RESULTS

Graph #1

- $1 \rightarrow 2$: Cost = 2, Path: $1 \rightarrow 4 \rightarrow 3 \rightarrow 2$
- $1 \rightarrow 3$: Cost = 4, Path: $1 \rightarrow 4 \rightarrow 3$
- $1 \rightarrow 4$: Cost = 7, Path: $1 \rightarrow 4$
- $1 \rightarrow 5$: Cost = -2, Path: $1 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 5$

Graph #2

- $1 \rightarrow 2$: Cost = 0, Path: $1 \rightarrow 2$
- $1 \rightarrow 3$: Cost = 0, Path: $1 \rightarrow 2 \rightarrow 3$
- $1 \rightarrow 4$: Cost = 0, Path: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$
- $1 \rightarrow 5$: Cost = 0, Path: $1 \rightarrow 5$
- $1 \rightarrow 6$: Cost = 0, Path: $1 \rightarrow 6$
- $1 \rightarrow 7$: Cost = 0, Path: $1 \rightarrow 5 \rightarrow 7$

Graph #3

- $1 \rightarrow 2$: Cost = 1, Path: $1 \rightarrow 2$
- $1 \rightarrow 3$: Cost = 2, Path: $1 \rightarrow 2 \rightarrow 3$
- $1 \rightarrow 4$: Cost = 3, Path: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$
- $1 \rightarrow 5$: Cost = 1, Path: $1 \rightarrow 5$
- $1 \rightarrow 6$: Cost = 1, Path: $1 \rightarrow 6$
- $1 \rightarrow 7$: Cost = 2, Path: $1 \rightarrow 5 \rightarrow 7$

Graph #4

- $1 \rightarrow 2$: Cost = 2, Path: $1 \rightarrow 2$
- $1 \rightarrow 3$: Cost = 6, Path: $1 \rightarrow 2 \rightarrow 5 \rightarrow 3$
- $1 \rightarrow 4$: Cost = 7, Path: $1 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 4$
- $1 \rightarrow 5$: Cost = 1, Path: $1 \rightarrow 2 \rightarrow 5$
- $1 \rightarrow 6$: Cost = 3, Path: $1 \rightarrow 6$
- $1 \rightarrow 7$: Cost = 2, Path: $1 \rightarrow 2 \rightarrow 5 \rightarrow 7$

Part 2: Spice Heist

The knapsack problem is a optimization problem where the goal is to select items to maximize total value without exceeding a given capacity. For our spice heist we are using a fractional knapsack where we treat spices as items and each spice has a quantity (representing weight) and a unitPrice (value per unit). The knapsack has a fixed capacity that limits how much it can hold, and the objective is to fill it with the most valuable combination of spices.

We can implement this using this code.

```
export function solveFractionalKnapsack(spices: Spice[], capacity: number): Knapsack {
const knapsack: Knapsack = {
    capacity,
    contents: [],
    totalValue: 0
};
let remainingCapacity = capacity;
const spiceInventory = spices.map(s => ({...s})); // Create copy to track remaining quantities
for (const spice of spiceInventory) {
    if (remainingCapacity <= 0) break;
    const scoopsToTake = Math.min(spice.quantity, remainingCapacity);
    if (scoopsToTake > 0) {
        knapsack.contents.push({
            spiceName: spice.name,
            scoops: scoopsToTake
        });
        knapsack.totalValue += scoopsToTake * (spice.unitPrice || 0);
        remainingCapacity -= scoopsToTake;
    }
}
return knapsack;
}
```

This implementation works by taking in a list of spices with their quantities and unit prices, along with the knapsack's capacity then iterating through the spices, calculating how much to take based on the remaining capacity, and updating the total value. After that it outputs A Knapsack object containing the total value and the list of spices included.

RESULTS

After running the spice heist these were the results I got.

Knapsack of capacity 1 is worth 9 quatloos and contains 1 scoop of orange.

Knapsack of capacity 6 is worth 38 quatloos and contains 2 scoops of orange, 4 scoops of blue.

Knapsack of capacity 10 is worth 58 quatloos and contains 2 scoops of orange, 8 scoops of blue.

Knapsack of capacity 20 is worth 74 quatloos and contains 2 scoops of orange, 8 scoops of blue, 6 scoops of green, 4 scoops of red.

Knapsack of capacity 21 is worth 74 quatloos and contains 2 scoops of orange, 8 scoops of blue, 6 scoops of green, 4 scoops of red.

The data gathered looks correct for all 5 knapsacks except for knapsack 5 which contains the same data as knapsack 4 is what I would say if that wasn't because there's only 20 weight in total, meaning they have the same data because they're both large enough to collect all the spice.

The running time of this algorithm is $O(nlogn)$ where n is the number of spices because of its divide and conquer methodology like merge sort, when sorting the spices by unit price then choosing them.