Assignment Four - The Finale

Ethan Ondreicka

Ethan.Ondreicka1@Marist.edu

December 9, 2023

1 Part One: Implementing the Bellman-Ford dynamic programming algorithm for Single Source Shortest Path

1.1 Parsing the File

```cpp
// file reader for the graphs2.txt
void parseInputFile(const std::string& filename, Graph& graph) {
    std::ifstream inputFile(filename);
    if (!inputFile.is_open()) {
        std::cerr << "Error opening file." << std::endl;
        return;
    }

    std::string line;
    bool newGraph = true; // Flag to indicate a new graph
    while (std::getline(inputFile, line)) {
        if (line.empty() || line.substr(0, 2) == "--") {
            newGraph = true; // Indicates a new graph
            continue;
        }

        if (newGraph) {
            graph.clearVertices(); // Clear existing graph data for the new graph
            newGraph = false; // Reset flag
        }

        if (line.substr(0, 3) == "add") {
            std::istringstream iss(line);
            std::string command;
            iss >> command;

            if (command == "add") {
                std::string subcommand;
                iss >> subcommand;

                if (subcommand == "vertex") {
                    int vertexId;
                    iss >> vertexId;
                    graph.addVertex(vertexId);
```

```
                } else if (subcommand == "edge") {
                    int src, dest, weight;
                    char dash;
                    iss >> src >> dash >> dest >> weight;
                    graph.addEdge(src, dest, weight);
                }
            }
        }
    }

    inputFile.close();
}
```

Listing 1: Parsing through the input file

The function parses through a given input file to create and clear graphs.

1.2 Vertex Constructor

```
// contructs each vertex
struct Vertex {
    int id;
    bool processed;
    std::vector<Vertex*> neighbors;
    std::vector<int> weights;
    long long distance; //  store distance for Bellman-Ford
    Vertex* predecessor; // store predecessor for path reconstruction
    Vertex(int _id) : id(_id), processed(false), distance(std::numeric_limits<int>::max()),
    predecessor(nullptr) {}
};
```

Listing 2: Constructing the Vertex Object

This creates each new vertex as well as assigns the neighbors (edges) and their weights

1.3 The Bellman Ford Algorithm

```
    void bellmanFord(Vertex* source, int maxIterations) {
        initializeSingleSource(source);

        for (size_t i = 0; i < maxIterations; ++i) {
            for (std::unordered_map<int, Vertex*>::iterator it = vertices.begin(); it !=
    vertices.end(); ++it) {
                Vertex* u = it->second;
                for (size_t j = 0; j < u->neighbors.size(); ++j) {
                    relax(u, u->neighbors[j], u->weights[j]);
                }
            }
        }
    }
```

Listing 3: Bellman-Ford Algorithm

This function loops through all of the vertices in the graph specified by 'maxIterations' and attempts to relax the distance. This operation is $O(n)$.

2 Part Two: The Fractional Knapsack

2.1 Constructors

```
// constructs the knapsack object
struct Knapsack {
    int knapsackNumber;
    int capacity;
    double totalValue;
};

// construction of the spice object
struct Spice {
    std::string name;
    double total_price;
    int qty;
    double price_per_unit;
};
```

Listing 4: Spice and Knapsack Constructor

These constructors make the knapsack object as well as the spice object. The values for each of the attributes in the constructor are based on the input file.

2.2 Sorting

```
// Insertion Sort Function for spices based on price per unit in descending order
void insertionSort(std::vector<Spice>& spices) {
    for (size_t i = 1; i < spices.size(); ++i) {
        Spice valueToInsert = spices[i];
        int j = i - 1;
        while (j >= 0 && spices[j].price_per_unit < valueToInsert.price_per_unit) {
            spices[j + 1] = spices[j];
            j = j - 1;
        }
        spices[j + 1] = valueToInsert;
    }
}
```

Listing 5: Insertion Sort Algorithm

I chose to use the insertion sort algorithm to sort the spice prices (in descending order) before they fill the knapsacks. Because I decided to use the Insertion Sort Algorithm, the asymptotic running time for the partial knapsack problem is $O(n^2)$.

2.3 Filling the Knapsack!

```
int totalQuantity = 0; // Total available quantity of spices

    // Calculate the total quantity of available spices
    for (size_t i = 0; i < spices.size(); ++i) {
        const Spice& spice = spices[i];
        totalQuantity += spice.qty;
    }

    for (size_t i = 0; i < knapsacks.size(); ++i) {
        Knapsack& knapsack = knapsacks[i];
        std::cout << "Knapsack of capacity " << knapsack.capacity << " is worth ";

        int remainingCapacity = knapsack.capacity;
        std::vector<std::string> contents;
        std::vector<Spice> availableSpices = spices; // Create a copy for this knapsack

        // Loop through le spices and fill the current knapsack
        for (size_t j = 0; j < availableSpices.size(); ++j) {
            Spice& spice = availableSpices[j];

            while (remainingCapacity > 0 && spice.qty > 0) {
                if (spice.qty <= remainingCapacity) {
                    contents.push_back(std::to_string(spice.qty) + " scoops of " + spice.
    name);
                    remainingCapacity -= spice.qty;
                    knapsack.totalValue += spice.total_price;
                    spice.qty = 0; // All scoops used for this knapsack
                } else {
                    contents.push_back(std::to_string(remainingCapacity) + " scoops of " +
    spice.name);
                    spice.qty -= remainingCapacity;
                    knapsack.totalValue += remainingCapacity * spice.price_per_unit;
                    remainingCapacity = 0; // Knapsack filled to capacity for this knapsack
                }
            }
        }
}
```

Listing 6: Code to fill the knapsack with spices

This first for loop from lines 4-7 show the calculations to see how much spices there to fill your knapsack. After that, the bigger for loop from lines 9-35 actually fill the knapsack using a greedy solution.