Assignment Four

Alex Tocci

Mason.Tocci1@Marist.edu

December 4, 2023

1 Directed Graph

1.1 Edge Class

Listing 1: Edge Class

```
1 #include <iostream>
2 #include <fstream>
3 #include <string>
4 #include <list>
5 #include <vector>
6 #include <sstream>
7 #include <iomanip>
8 #include <limits>
9 #include <algorithm>
10
11 using namespace std;
12
13 class Vertex;
14
15 class Edge {
16 public:
17     Vertex* source;
18     Vertex* destination;
19     int weight;
20
21     // Constructor
22     Edge(Vertex* s, Vertex* d, int w) : source(s), destination(d), weight(w) {}
23 };
```

Lines 1-9: First we must include the necessary C++ libraries for our program to run.
Line 11: This allows us to use standard C++ functions without the prefix 'std::'.
Line 11: Forward declaration for the vertex class.

Lines 15-19: Define a C++ class called 'Edge' which contains three variables, 'Vertex* source', 'Vertex* destination', and 'int weight'. 'Vertex* source' stores the source/start vertex and 'Vertex* destination' stores the destination/end vertex. 'int weight' stores the weight of the edge.
Line 22: This is the Edge class constructor, which initializes the Edge object.

1.2 Vertex Class

Listing 2: Vertex Class

```
24 class Vertex {
25 public:
26     int vertexID;
27     int distance;
28     Vertex* predecessor;
29     vector<Edge*> outgoingEdges;
30 
31     // Constructor
32     Vertex(int id) : vertexID(id), distance(numeric_limits<int>::max()), predecessor(nullpt
33 
34     // Add an outgoing edge to the vertex
35     void addOutgoingEdge(Vertex* dest, int weight) {
36         outgoingEdges.push_back(new Edge(this, dest, weight));
37     }
38 };
```

Lines 24-32: Define a C++ class called 'Vertex' which contains four variables, 'int vertexID', 'int distance', 'Vertex* predecessor', and 'vector<Edge*> outgoingEdges'. 'int vertexID' stores the ID of the vertex. 'int distance' stores the current distance of the vertex from a source vertex. 'Vertex* predecessor' stores the predecessor of this vertex in the shortest path. 'vector<Edge*> outgoingEdges' stores pointers to Edge objects going out from this vertex.
Lines 35-38: This function adds an Edge to the 'outgoingEdges' vector.

1.3 Graph Class

1.3.1 Find Vertex and Find Edge Function

Listing 3: Find Vertex and Find Edge Function

```
39 class Graph {
40 public:
41     vector<Vertex*> vertices;
42 
43     Vertex* findVertex(int id) {
44         for (Vertex* vertex : vertices) {
45             if (vertex->vertexID == id) {
46                 return vertex;
47             }
48         }
49         return nullptr;
50     }
51 
52     Edge* findEdge(int sourceID, int destinationID) {
53         Vertex* source = findVertex(sourceID);
```

```
54          for (Edge* edge : source->outgoingEdges) {
55              if (edge->destination->vertexID == destinationID) {
56                  return edge;
57              }
58          }
59          return nullptr;
60      }
```

Line 41: Declare a vector which stores all the vertices in the graph.
Lines 43-50: The 'findVertex' function takes in an id and loops through all the vertices in the 'vertices' vector. If a vertex with a matching id is found, return a pointer to that vertex. If no match is found, return 'nullptr'.
Lines 52-60: The 'findEdge' function takes in a sourceID and a destinationID. First, it runs the 'findVertex' function to find the vertex with an id that matches the sourceID. Then, it loops through the 'outgoingEdges' vector looking for an edge with a matching destinationID. If a match is found, return a pointer to that edge. If no match is found, return 'nullptr'.

1.3.2 Add Vertex, Add Edge, and Print Edges Function

Listing 4: Add Vertex, Add Edge, and Print Edges Function

```
61  // Add a new vertex to the graph
62  void addVertex(int id) {
63      vertices.push_back(new Vertex(id));
64  }
65
66  // Add a directed edge with weight between two vertices
67  void addDirectedEdge(Vertex* from, Vertex* to, int weight) {
68      from->addOutgoingEdge(to, weight);
69  }
70
71  void printAllEdges() {
72      cout << "All Edges and Their Weights:\n";
73      for (Vertex* vertex : vertices) {
74          for (Edge* edge : vertex->outgoingEdges) {
75              cout << edge->source->vertexID << " -> " << edge->destination->vertexID << " 
76          }
77      }
78      cout << endl;
79  }
```

Lines 62-64: The 'addVertex' function creates a new vertex using the id passed in and adds it to the 'vertices' vector.
Lines 67-69: The 'addDirectedEdge' function takes in two pointers to vertex objects; from is the source/start vertex and to is the destination/end vertex. It also takes in the weight. Then it calls the 'addOutgoingEdge' function of the 'from' vertex, passing in the 'to' vertex and the weight.
Lines 71-79: The 'printAllEdges' function loops through all the vertices in the 'vertices' vector, and loops through all the edges in each vertex's 'outgoingEdges' vector, printing the sourceID, destinationID, and weight.

1.3.3 Bellman-Ford Functions

Listing 5: Bellman-Ford Functions

```
80  // Initialize single source function
81  void initializeSingleSource(Vertex* source) {
82      for (Vertex* vertex : vertices) {
83          vertex->distance = numeric_limits<int>::max();
84          vertex->predecessor = nullptr;
85      }
86      source->distance = 0;
87  }
88
89  // Relax function
90  void relax(Vertex* u, Vertex* v, int weight) {
91      if (v->distance > u->distance + weight) {
92          v->distance = u->distance + weight;
93          v->predecessor = u;
94      }
95  }
96
97  // Bellman-Ford algorithm
98  bool bellmanFord(Vertex* source) {
99      initializeSingleSource(source);
100
101     for (size_t i = 1; i <= vertices.size() - 1; ++i) {
102         for (Vertex* vertex : vertices) {
103             for (Edge* edge : vertex->outgoingEdges) {
104                 relax(edge->source, edge->destination, edge->weight);
105             }
106         }
107     }
108
109     // Print the shortest paths
110     cout << "Shortest paths from vertex " << source->vertexID << ":\n";
111     for (Vertex* vertex : vertices) {
112         if (vertex->distance == numeric_limits<int>::max()) {
113             cout << "Vertex " << vertex->vertexID << ": No path" << endl;
114         }
115         else {
116             printPath(source, vertex);
117             cout << endl;
118         }
119     }
120
121     return true;
122 }
```

Lines 81-87: The 'initializeSingleSource' function takes in a source vertex. Then it loops through all the vertices in the 'vertices' vector, setting their distance to 'numeric-limits<int>::max()' (representing an infinite distance) and predecessor to 'nullptr'. Then, it sets the distance of the source vertex to 0.
Lines 90-95: The 'relax' function takes in two pointers to vertex objects, 'u' and 'v'. It also takes in a weight. If the distance to 'v' through 'u' is shorter than the current known distance to 'v', it updates the distance of 'v' to the new shorter distance and sets 'u' as the predecessor of 'v'.
Lines 98-99: The 'bellmanFord' function takes in a source vertex and finds the shortest path to all other

vertices in the graph. First, it calls the 'initializeSingleSource' function to initialize the distances and predecessors.
Lines 101-107: Then, it loops 'size-1' times where size is the number of vertices in the 'vertices' vector. In each loop, it loops through all edges in the graph and relaxes them using the 'relax' function.
Lines 110-122: Then, it loops through all the vertices in the 'vertices' vector and calls the 'printPath' function, passing in the source vertex as a source/start and the current vertex as a destination/end.

1.3.4 Print Shortest Path

Listing 6: Print Shortest Path

```
123 private:
124     // Print the path from source to destination
125     void printPath(Vertex* source, Vertex* destination) {
126         list<int> path;
127         while (destination != nullptr) {
128             path.push_front(destination->vertexID);
129             destination = destination->predecessor;
130         }
131 
132         auto it = path.begin();
133         int prevVertexID = *it;
134         int totalCost = 0;
135         ++it;
136         for (; it != path.end(); ++it) {
137             Vertex* currentVertex = findVertex(*it);
138             totalCost += findEdge(prevVertexID, *it)->weight;
139             prevVertexID = *it;
140         }
141 
142         // Print the starting and ending vertices
143         cout << source->vertexID << " --> " << *path.rbegin() << " cost is " << totalCost <
144 
145         // Print the path
146         for (it = path.begin(); it != path.end(); ++it) {
147             cout << *it;
148             if (next(it) != path.end()) {
149                 cout << " --> ";
150             }
151         }
152     }
153 };
```

Line 125: The 'printPath' function prints the shortest path from a source vertex to a destination vertex. It takes in a source and destination vertex.
Line 126: Declare A linked list 'path' to store the vertices along the path.
Lines 127-130: This loop traverses the predecessor pointers from the destination vertex back to the source vertex, pushing each vertex's ID onto the 'path' list.
Line 132: Iterator for looping through the 'path' list.
Line 133: The ID of the previous vertex in the path, initialized with the first vertex in the path..
Line 134: Variable to keep track of the total cost of the path.
Line 135: Increment the iterator.

Lines 136-140: Loop through the vertices in the path and calculate the total cost of the path by adding the weights of the edges.
Line 143: Print the start and end vertex.
Lines 146-153: This loop prints the vertices in the path, separated by ' −> '.

2 Knapsack

2.1 Spice Struct

Listing 7: Spice Struct

```
154 struct Spice {
155     string name;
156     double total_price;
157     int qty;
158     double unit_price;
159
160     Spice(string n = "", double tp = 0.0, int q = 0) : name(n), total_price(tp), qty(q) {
161         unit_price = total_price / qty;
162     }
163 };
```

Lines 154-158: Define a C++ struct called 'Spice' which contains four variables, 'string name', 'double total-price', 'int qty', and 'double unit-price'. 'string name' stores the name of the spice, 'double total-price' stores the total price of the spice, 'int qty' stores the number of scoops is available for a spice, and 'double unit-price' stores the unit price.
Lines 160-163: This is the Spice struct constructor, which initializes the Spice object. Unit price is calculated using 'total-price / qty'.

2.2 Knapsack Struct

Listing 8: Knapsack Struct

```
164 struct Knapsack {
165     int capacity;
166     double worth;
167     vector<pair<string, double>> scoops;
168 };
```

Lines 164-168: Define a C++ struct called 'Knapsack' which contains three variables, 'int capacity', 'double worth', and 'vector<pair<string, double» scoops'. 'int capacity' stores the number of scoops that the knapsack can hold, 'double worth' stores the worth or value of the items in the knapsack, and 'vector<pair<string, double» scoops' stores a vector of pairs, where each pair represents a "scoop" placed in the knapsack.

2.3 Fractional Knapsack Functions

Listing 9: Fractional Knapsack Functions

```
169 bool compareSpicesByUnitPrice(const Spice& a, const Spice& b) {
170     return a.unit_price > b.unit_price;
171 }
172
173 void fractionalKnapsack(vector<Spice>& spices, vector<Knapsack>& knapsacks) {
174     // Sort spices by unit price in descending order
175     sort(spices.begin(), spices.end(), compareSpicesByUnitPrice);
176
177     // Fill knapsacks using fractional knapsack algorithm
```

```
178     for (auto& knapsack : knapsacks) {
179         int originalCapacity = knapsack.capacity;  // Store the original capacity
180 
181         for (auto& spice : spices) {
182             if (spice.qty > 0) {
183                 double quantity = min(static_cast<double>(knapsack.capacity), static_cast<d
184                 knapsack.scoops.push_back({ spice.name, quantity });
185                 knapsack.worth += quantity * spice.unit_price;
186                 spice.qty -= static_cast<int>(quantity);
187                 knapsack.capacity -= static_cast<int>(quantity);
188 
189                 if (knapsack.capacity == 0) {
190                     // Knapsack is full, move to the next one
191                     break;
192                 }
193             }
194         }
195 
196         // Reset knapsack capacity to its original value
197         knapsack.capacity = originalCapacity;
198 
199         // Reset spice quantities and unit prices after filling knapsack
200         for (auto& spice : spices) {
201             spice.qty = spice.total_price / spice.unit_price;
202 
203             // Recalculate unit price after reset
204             spice.unit_price = spice.total_price / spice.qty;
205         }
206     }
207 }
```

Lines 169-171: The 'compareSpiceByUnitPrice' takes in two spice objects, 'a' and 'b'. If 'a' has a greater unit price than 'b', the function will return 'true'.
Line 173: The 'fractionalKnapsack' function takes in two vectors, the 'spices' vector which contains all the spice objects and the 'knapsacks' vector which takes in all the knapsack objects.
Line 175: Sort spices by unit price in descending order.
Lines 178-194: Loop through each knapsack in the 'knapsacks' vector. For each knapsack, loop through each spice in the sorted 'spices' vector. If the quantity of the spice is greater than 0, calculate how many scoops the knapsack can take based on its capacity. Add the quantity of the spice to the knapsacks 'scoops' vector, update the spice quantity, and the knapsack worth. If the knapsack becomes full, the loop breaks.
Line 197: Reset the knapsack capacity for printing purposes.
Lines 200-207: Reset the quantities and unit price for the remaining knapsacks.

3 Main Function

3.1 Read 'graphs2.txt' and Call The Graph Functions

Listing 10: Read 'graphs2.txt' and Call The Graph Functions

```cpp
int main() {
    Graph graph;

    // Open the file
    ifstream graphFile("graphs2.txt");

    // Handle failure
    if (!graphFile) {
        cerr << "Failed to open graphs1.txt" << endl;
        return 1;
    }

    // Create a pointer to a Graph to hold the current graph
    Graph* currentGraph = nullptr;

    // Process each line in the file
    string line;
    while (getline(graphFile, line)) {
        istringstream iss(line);
        string command;
        iss >> command;

        if (command == "new" && iss >> command && command == "graph") {
            // "new graph" command
            if (currentGraph) {
                // Print all edges before running Bellman-Ford
                currentGraph->printAllEdges();
                // Run Bellman-Ford algorithm
                currentGraph->bellmanFord(currentGraph->vertices[0]);
                delete currentGraph;
            }
            currentGraph = new Graph;
            cout << "\nCreated a new graph.\n";
        }
        else if (command == "add") {
            string subcommand;
            iss >> subcommand;

            if (subcommand == "vertex") {
                // "add vertex" command
                int id;
                if (iss >> id) {
                    currentGraph->addVertex(id);
                }
            }
            else if (subcommand == "edge") {
                // "add edge" command for directed graph with weight
```

```
255                 int fromID, toID, weight;
256                 char arrow;
257                 if (iss >> fromID >> arrow >> toID >> weight && arrow == '-') {
258                     Vertex* fromVertex = currentGraph->findVertex(fromID);
259                     Vertex* toVertex = currentGraph->findVertex(toID);
260                     currentGraph->addDirectedEdge(fromVertex, toVertex, weight);
261                 }
262             }
263         }
264     }
265
266     // Print and delete the final graph
267     if (currentGraph) {
268         // Print all edges before running Bellman-Ford
269         currentGraph->printAllEdges();
270         // Run Bellman-Ford algorithm
271         currentGraph->bellmanFord(currentGraph->vertices[0]);
272         delete currentGraph;
273         cout << endl;
274     }
275
276     // Close the file
277     graphFile.close();
```

Line 212: Open the file 'graphs2.txt' for reading.
Lines 215-218: Check if the file was successfully opened. If the file cannot be opened, print an error message and exit the program.
Line 221: Initialize a pointer to a graph 'currentGraph' and set it to 'nullptr'.
Line 225: Loop through and read the file line by line.
Lines 230-241: If the line reads 'new graph', check if a graph already exists. If one does, then print all the edges and run the 'bellmanFord' function for that graph, and then delete it. Finally, create a new graph.
Lines 242-252: If the line reads 'add vertex', run the 'addVertex' function, passing in the vertexID that is on the line.
Lines 253-262: If the line reads 'add edge', run the 'addDirectedEdge' function, passing in the two vertices that are on the line.
Lines 267-274: Print all the edges and run the 'bellmanFord' function for the final graph and delete it.
Line 277: Close the file.

3.2 Read 'spice.txt' and Add Data To 'spices' and 'knapsacks' Vector

Listing 11: Read 'spice.txt' and Add Data To 'spices' and 'knapsacks' Vector

```
    ifstream knapsackFile("spice.txt");

    if (!knapsackFile.is_open()) {
        cerr << "Failed to open spice.txt" << endl;
        return 1;
    }

    vector<Spice> spices;
    vector<Knapsack> knapsacks;
```

```
287
288     while (getline(knapsackFile, line)) {
289         if (line.find("spice name") != string::npos) {
290             size_t pos;
291
292             Spice spice;
293
294             pos = line.find("= ");
295             spice.name = line.substr(pos + 2, line.find(";") - pos - 2);
296
297             pos = line.find("total_price = ");
298             spice.total_price = stod(line.substr(pos + 14, line.find(";") - pos - 14));
299
300             pos = line.find("qty = ");
301             spice.qty = stoi(line.substr(pos + 6, line.find(";") - pos - 6));
302
303             spice.unit_price = spice.total_price / spice.qty;
304
305             spices.push_back(spice);
306         }
307         else if (line.find("knapsack capacity") != string::npos) {
308             Knapsack knapsack;
309             size_t pos = line.find("= ");
310             knapsack.capacity = stoi(line.substr(pos + 2, line.find(";") - pos - 2));
311             knapsack.worth = 0.0;
312
313             knapsacks.push_back(knapsack);
314         }
315     }
316
317     knapsackFile.close();
```

Line 278: Open the file 'spice.txt' for reading.
Lines 280-283: Check if the file was successfully opened. If the file cannot be opened, print an error message and exit the program.
Lines 285-286: Initialize vectors to hold the spices and the knapsacks.
Line 288: Loop through and read the file line by line.
Lines 289-306: If a line reads 'spice name' create an instance of a spice and extract the name, quantity, and total price from the line. Calculate unit price with 'spice.total price / spice.qty'. Add the spice to the 'spices' vector.
Lines 307-315: If a line reads 'knapsack capacity' create an instance of a knapsack and extract the capacity from the line. Set the worth of the knapsack to 0. Add the knapsack to the 'knapsacks' vector.
Line 317: Close the file.

3.3 Run Fractional Knapsack and Output Results

Listing 12: Run Fractional Knapsack and Output Results

```
    // Apply fractional knapsack algorithm
    fractionalKnapsack(spices, knapsacks);

    // Output the results
```

```
322     for (const auto& knapsack : knapsacks) {
323         cout << "Knapsack of capacity " << knapsack.capacity << " is worth " << knapsack.wo
324         for (const auto& scoop : knapsack.scoops) {
325             cout << scoop.second << " scoops of " << scoop.first << ", ";
326         }
327         cout << endl;
328     }
329 
330     return 0;
331 }
```

Line 319: Call the 'fractionalKnapsack' function, passing in the 'spices' and 'knapsacks' vectors.
Line 322-331: Loop through each knapsack in the 'knapsacks' vector, printing the worth. Loop through each scoop in the knapsack's 'scoops' vector, printing the spice name and the number of scoops.

4 Analysis

4.1 SSSP

The 'bellmanFord' function has a worst case time complexity of $O(V * E)$, where V is the number of vertices and E is the number of edges. This is because a nested for loop that runs $(V - 1)$ times, loops through all vertices and edges. This results in a time complexity of $O((V - 1) * V * E)$, which can be simplified to $O(V * E)$. The 'initializeSingleSource' and 'printPath' contribute linearly to the complexity, which is why they are not a factor.

4.2 Fractional Knapsack

The 'fractionalKnapsack' function has a worst case time complexity of $O(SlogS + K * S)$, where S is the number of spices and K is the number of knapsacks. This is because the spices are, first, sorted using 'std::sort' which has a time complexity of $O(SlogS)$. Then a nested for loop, loops through each knapsack and spice. In the worst case, each spice is considered for each knapsack and results in a $O(K * S)$ run time. Add the two run times together to get a total asymptotic run time of $O(SlogS + K * S)$.