CMPT 435 - Fall 2024 - Dr. Labouseur

Assignment Three: Spice Heist and Directed Graphs

Jayden Melendez
Jayden.Melendez1@Marist.edu

December 6th, 2024

Abstract

This report is the implementation and analysis of a greedy algorithm approach to solving a maximum value problem and a showcase of the Bellman-Ford algorithm. The greedy algorithm is illustrated through a story inspired by the Dune franchise, and the Bellman-Ford algorithm is used to highlight the features of a directed graph.

1 The Spice Heist on Arrakis

1.1 Create Spice Objects

```cpp
// Get the different spices
vector<Spice*> SpiceHeist::getSpices(string spiceStr) {
    // Get the spice information from the file
    vector<string> lines = parse.readHeistFile(SPICE_FILE);
    vector<Spice*> spices;
    // Iterate to find spice commands
    for (int i = 0; i < lines.size(); i++) {
        // If line is a spice command
        if (lines[i].find(spiceStr) != string::npos) {
            Spice *spice = newSpice(lines[i]);
            // Add the spice object to the vector of spices
            spices.push_back(spice);
        }
    }
    // Return the different spice objects
    return spices;
}
```

Listing 1: Implementation to get spice objects

This method reads the file containing spice information to keep track of what type and how much spice is on Arrakis. The spice types and quantities are then documented inside objects to conduct the greedy algorithm on. This is the implementation of the **newSpice()** method.

```cpp
// Create a new spice object
Spice* SpiceHeist::newSpice(string instructions) {
    // Character to split separated command data
    char delimeter = ';';
    // To store separated command data
    vector<string> splitInstructions = parse.splitString(instructions,
        delimeter);
    // Character to split within the command data
    delimeter = '=';
    // Individual spice characteristics
    vector<string> splitLineInstructions;
    vector<string> characteristics;
    string name, price, quantity, characteristic;
    // Iterate through each spice instruction
    for (int i = 0; i < splitInstructions.size(); i++) {
        // Ignore everything before an '='
        splitLineInstructions = parse.splitString(splitInstructions[i],
            delimeter);
        characteristic = splitLineInstructions[1];
        // Remove excess whitespace
        characteristic.erase(remove_if(characteristic.begin(),
        characteristic.end(), ::isspace), characteristic.end());
        // Add the characteristic to the vector
        characteristics.push_back(characteristic);
        // Release the string pointer
        characteristic.clear();
25     }
26     // Create a spice object
27     name = characteristics[0];
28     price = characteristics[1];
29     quantity = characteristics[2];
30     Spice *spice = new Spice(name, quantity, price);
31     // Return the object
32     return spice;
33 }
```

Listing 2: Make a new spice object from instructions

In this implementation of reading an instruction from the spice file, each line is parsed to separate the different characteristics of the spice. On line 14, each instruction line for different spices is then split into individual components. Each spice has a name (type), price, and quantity. All of these components are included in each **splitInstruction** line. The characteristics are then passed into the creation of a spice object. The **spice class** implementation is shown below:

```
#include "AssignmentFour/include/Spice.h"
#include <iostream>
#include <string>
using namespace std;

// Initialize spice name, quanity, and price
Spice::Spice(const string &nme, const string &quant, const string &prc) : name(nme), quantity(quant), price(prc), unitPrice(calculateUnitPrice()), originalQuantity(stoi(getQuantity())) {}

// Get the color of spice
string Spice::getName() {
    return name;
}

// Get the quantity of this spice
string Spice::getQuantity() {
    return quantity;
}

// Set the quantity to reset
void Spice::setQuantity(int quant) {
    quantity = to_string(quant);
}

// Get the price of this spice
string Spice::getPrice() {
    return price;
}

// Calculate the unit price of this spice
string Spice::calculateUnitPrice() {
    int quantity = stoi(this->getQuantity());
    int price = stoi(this->getPrice());
    float unitPrice = price / quantity;
    return to_string(unitPrice);
}

// Return the unit price
string Spice::getUnitPrice() {
    return unitPrice;
}

```

```cpp
// Remove spice from Arrakis
void Spice::digSpice(string numOfSpice) {
    int newQuantity = stoi(quantity) - stoi(numOfSpice);
    quantity = to_string(newQuantity);
}

// Save the original quantity
int Spice::getOriginalQuantity() {
    return originalQuantity;
}
```

Listing 3: Spice class implementation

A spice object is used for the implementation of a greedy algorithm. The class from which all instances are derived is shown above. Each spice object has a name, price, and quantity associated with it. Each spice object also has an original quantity to keep track of the original amount found on Arrakis. This will be used to print the remaining spice on Arrakis in the output. The unit price is imperative for sorting spice objects for the greedy algorithm. The method is the price of the spice divided by the quantity of spice in the object.

In order to ensure that Spice objects are collected and put into Knapsack objects that will always yield the highest value, the spices need to be sorted by unit value. As stated previously, each Spice object has the method **getUnitPrice()** that calculates the unit price depending on the quantity of that type of Spice and the total cost of that quantity. In this implementation, merge sort was used because it is the most efficient and in previous testing has the smallest number of comparisons to sort a list of elements. The time complexity is $O(nlog_2(n))$ which is very efficient compared to most other sorting algorithms that commonly resort to $O(n^2)$ time complexity. This the code for merge sort adapated for the use in this spice heist:

```cpp
// Merge many arrays for Merge Sort
vector<Spice*> Sorting::mergeSpices(vector<Spice*> spices, vector<Spice*>
    leftElements, vector<Spice*> rightElements) {
    // Define helper variables for comparison
    int i = 0;
    int left = 0;
    int right = 0;
    // Compare left and right items for merging from greatest to least
    while (left < leftElements.size() && right < rightElements.size()) {
        // Merge together based on unit price
        if (leftElements[left]->getUnitPrice() > rightElements[right]->
            getUnitPrice()) {
            spices[i] = leftElements[left];
            left++;
        } else {
            spices[i] = rightElements[right];
            right++;
        }
        i++;
    }
    // Handle remaining elements from leftItems
    while (left < leftElements.size()) {
        spices[i] = leftElements[left];
        left++;
        i++;
    }
    // Handle remaining elements from rightItems
    while (right < rightElements.size()) {
        spices[i] = rightElements[right];
        right++;
        i++;
    }
31     return spices;  // Return the pointer to the merged array
32 }
33 
34 // Merge sort algorithm
35 vector<Spice*> Sorting::doMergeSort(vector<Spice*> spices) {
36     // Get the size of the current vector
37     int size = spices.size();
38     // Return the base case
39     if (size <= 1) {
40         return spices;
41     }
42     int middle = size / 2; // Determine the middle index
43     vector<Spice*> leftElements;  // Define the left array bound
44     vector<Spice*> rightElements; // Define the right array bound
45     // Populate left items from the left side of items
46     for (int i = 0; i < middle; i++) {
47         leftElements.push_back(spices[i]); // Add the items on the left to the
               left array
48     }
49     // Populate right items from the right side of items
50     for (int i = 0; i < size - middle; i++) {
51         rightElements.push_back(spices[i + middle]);
52     }
53     // Recursively call method until base case is reached
54     leftElements = doMergeSort(leftElements);  // Handle the left branch
55     rightElements = doMergeSort(rightElements);  // Handle the right branch
56     // Merge leftItems and rightItems until the full array is rebuilt sorted
57     vector<Spice*> mergedSpices = mergeSpices(spices, leftElements,
           rightElements);
58     return mergedSpices;
59 }
```

Listing 4: Sorting Spice objects

The merge sort implementation was explained in a previous write-up; however, there are a few key changes that should be highlighted. First off, the algorithm sorts Spice objects based on their attribute, **unitPrice** instead of a string. This can be found on line 2, where the method takes in only vectors that contain spice objects. Another key change was on line 10. The Spices need to be sorted from greatest unit price to least unit price because that is how the greedy algorithm ensures that the maximum value of spice enters a Knapsack with a limited capacity. In addition, the comparison in the merge is comparing the unitPrice attribute of the object instead of the ASCII values of strings. On line 35, the first part of the merge sort algorithm is defined as the method **doMergeSort()**. This implementation is a common version of the merge sort algorithm that recursively breaks a vector or array in half until there are many single-index vectors.

1.2 Creating Knapsack Objects

In this implementation of a greedy algorithm, a user needs to have a knapsack to carry a certain number of Spice. The knapsack has a maximum capacity that it cannot exceed. Therefore, the most useful way to represent a knapsack is to create a class that can be used to create knapsacks with different quantity maximums. This first code snippet illustrates where the file is read and parsed to extract instructions to create each knapsack object.

```
// Get the different knapsacks
vector<Knapsack*> SpiceHeist::getKnapsacks(string knapsackStr) {
    // Get the spice information from the file
    vector<string> lines = parse.readHeistFile(SPICE_FILE);
    vector<Knapsack*> knapsacks;
```

```
 6      // Iterate to find knapsack commands
 7      for (int i = 0; i < lines.size(); i++) {
 8          // If the line is a knapsack command
 9          if (lines[i].find(knapsackStr) != string::npos) {
10             Knapsack *knapsack = newKnapsack(lines[i]);
11             // Add the knapsack objects to the vector of knapsacks
12             knapsacks.push_back(knapsack);
13          }
14      }
15      // Return the knapsack objects
16      return knapsacks;
17  }
```

Listing 5: Reading file and getting knapsacks

The method, **getKnapsacks()** reads a file and looks for *knapsack* keywords in a text file line to denote that the instructions are for the creation of a knapsack. The value is passed in as a parameter; however, the string value is stored in a separate file of constants that can be easily modified or accessed across multiple areas in this project. The text file is split into lines and stored in a vector with only the relevant spice and knapsack instructions. The for loop on line 7 iterates through each line looking for knapsack instructions that have the *knapsack* substring. Each knapsack instruction is used to create a knapsack object in the method **newKnapsack()**. Each knapsack returned in this method is added to a knapsacks vector called **knapsacks** initialized on line 5. The knapsacks are then returned to be used for the Spice Heist greedy algorithm. This next code snippet illustrates how a knapsack object is created from the characteristics found in each *knapsack* instruction in the lines vector:

```
 1  // Create new knapsack objects from instruction file
 2  Knapsack* SpiceHeist::newKnapsack(string instructions) {
 3      // Define what to split on
 4      char delimeter = '=';
 5      // Divide the string on '='
 6      vector<string> splitInstructions = parse.splitString(instructions,
           delimeter);
 7      // Split on ';'
 8      delimeter = ';';
 9      splitInstructions = parse.splitString(splitInstructions[1], delimeter);
10      // Hold capacity of knapsack
11      string capacity;
12      for (int i = 0; i < splitInstructions.size(); i++) {
13          // Get the capacity value
14          capacity = splitInstructions[i];
15          // Remove whitespace
16          capacity.erase(remove_if(capacity.begin(),
17          capacity.end(), ::isspace), capacity.end());
18      }
19      Knapsack *knapsack = new Knapsack(capacity);
20      // Return the knapsack object
21      return knapsack;
22  }
```

Listing 6: Creating knapsacks from instructions

In this code snippet above, knapsack objects are created. Each knapsack instruction line is passed in as a parameter on line 2 in the method newKnapsack(). The method returns each new **Knapsack** created on line 21. A delimiter is used to trim out the unnecessary components of the line that are not useful for creating a spice object. The delimiter for the first split is an =. Each knapsack line from the text file looks like this:

```
"knapsack capacity =  1;"
```

As you can see, if the line is split on the =, there will be two components to the vector returned. On line 6, the splitString() method from the **ParseHeistFile** class denoted by **parse** takes in the delimiter and splits the string. In the case of the above, a **splitInstructions** string vector will be returned with its contents being

```
[ "knapsack capacity", "1;" ]
```

The method then ignores the first element in the vector because it is no longer useful since it has become clear that there is the intention to create the knapsack object. The knapsack object takes in a quantity parameter. Therefore, on line 9, the second element in the **splitInstructions** vector needs to split to isolate the quantity number. In this case, the quantity is 1 because splitting on a **;** as the second delimeter will leave the reassigned splitInstructions vector on line 9 with one element. This is what the splitInstructions vector now contains in this example:

```
[ "1" ]
```

Any extra whitespace is removed from any element in the split instructions vector. In this assignment, there should always be one element in the splitInstructions vector up to this point. On line 14, the quantity from the splitInstructions vector is now assigned to capacity to represent the parameter that will be passed on as the intended quantity of the next knapsack object. The knapsack object is created on line 19 and returned on line 21 for its use in the greedy algorithm. The implementation of the **Knapsack** class is shown below:

```cpp
#include "AssignmentFour/include/Knapsack.h"
#include "AssignmentFour/include/Spice.h"
#include <string>
using namespace std;

// Constructor
Knapsack::Knapsack(const string &cap) : capacity(cap), spices() {}

// Return the potential capacity of a knapsack
string Knapsack::getCapacity() {
    return capacity;
}

// Return the spices within a knapsack
vector<Spice*> Knapsack::getContents(){
    return spices;
}

// Add a spice to the knapsack
void Knapsack::addSpice(Spice* spice) {
    spices.push_back(spice);
}

// Calculate the total price of the knapsack
string Knapsack::getTotalPrice() {
    int total = 0;
    for (int i = 0; i < spices.size(); i++) {
        total = total + (stoi(spices[i]->getUnitPrice()) * stoi(spices[i]->getQuantity()));
    }
30     return to_string(total);
31 }
```

Listing 7: Knapsack class

As previously mentioned, a Knapsack object represents a knapsack that a user will use to collect spice from Arrakis. The capacity of the knapsack is defined and immutable when it's created. In *line 10*, the **getCapacity()** method returns the initial capacity of the knapsack, which will be used to determine how much spice can fit in the knapsack during the implementation of the greedy algorithm later in the workflow. The **getContents()** method, on *line 15*, returns the spices currently in the knapsack and will be used to display the results of the heist to the user. The **addSpice()** method, on *line 20*, adds a spice object to the private Spices vector in a Knapsack object. This method will be used for printing and to determine when the knapsack is full. On *line 25*, the **getTotalPrice()** method calculates the knapsack's total value at the end of the algorithm. This is determined by iterative addition to the total variable defined on *line 26*. During each iteration of the spices vector in the knapsack, the next Spice object's unit price is calculated, multiplied by the amount of that type of spice in the knapsack, and added to the total. After all spices have calculated their value, their total value is added to the total variable, and the knapsack's total value is returned.

2 Directed/Weighted Graphs

Directed and Weighted Graphs have vertices like normal graphs but also have edges that move in one direction and have a weight. A weight is another word for the cost to travel from one vertex to the next. This allows for some very cool algorithms that be created using these constructs. In this implementation, a directed graph is broken up into many different classes that represent its individual components. The first being the class that represents the class itself.

2.1 Creating a Graph

```
// Build graphs of vertices and edges
vector<DirectedGraph*> BuildDirectedGraph::buildGraph(vector<string>
    instructions) {
    // To build a graph
    DirectedGraph* graph;
    vector<DirectedGraph*> graphs;
    // To build a vertex
    vector<string> vertexInfo;
    // To build an edge
    vector<DirectedGraphVertex*> vertices;
    DirectedGraphVertex* fromVertex;
    DirectedGraphVertex* toVertex;
    vector<string> edgeInfo;
    int weight;
    // Iterate through all instructions
    for (int i = 0; i < instructions.size(); i++) {
        // Make graphs, vertices, and edges based on the text file
        if (instructions[i].find(NEW_GRAPH_SUBSTRING) != string::npos) {
            graph = newGraph();
            graphs.push_back(graph);
        } else if (instructions[i].find(ADD_VERTEX_SUBSTRING) != string::npos)
            {
            vertexInfo = parse.parseGraphInstruction(instructions[i]);
            newVertex(graph, vertexInfo[0]);
        } else if (instructions[i].find(ADD_EDGE_SUBSTRING) != string::npos) {
            edgeInfo = parse.parseGraphInstruction(instructions[i]);
            vertices = graph->getVertices();
26             // Get the 'from' vertex
27             fromVertex = doSearch.doVertexSearch(edgeInfo[0], vertices);
28             // Get the 'to' vertex
29             toVertex = doSearch.doVertexSearch(edgeInfo[1], vertices);
30             // Get the weight of the edge
31             weight = std::stoi(edgeInfo[2]);
32             newEdge(fromVertex, toVertex, weight);
33         }
34     }
35     return graphs;
36 }
```

Listing 8: Building a graph

This method takes in the instructions for building a graph as a parameter. The instructions are already passed in parsed and usable for this method. The for loop on line 15 looks for graph, vertex, and edge substrings. These substrings denote what the instructions is for. It is imperative that the graph instructions comes before all of the vertex instructions and the edge instructions come after the vertex. When its substring is found, the appropriate helper method is called to build the different objects that will be highlighted later in this section. The condition for an edge instruction is a little more complex because on *line 27 and 29* the from (outgoing) and to (destination) vertex need to be defined.

At this point, the graph has all of the vertices it will have based on the order of instructions. Therefore, a simple search algorithm can be used to find the vertex pointers that have an **id** that matches the **id** value from the file. To start, this is the implementation for **newGraph()**:

```
1 // Create a new graph instance
2 DirectedGraph* BuildDirectedGraph::newGraph() {
3     DirectedGraph *graph = new DirectedGraph();
4     return graph;
5 }
```

Listing 9: Create a graph

This is simply creating a new graph object that needs to be ready to add vertices. This is the implementation of **newVertex()**:

```
1 // Create a new vertex instance in a graph
2 DirectedGraphVertex* BuildDirectedGraph::newVertex(DirectedGraph* graph, string
      id) {
3     DirectedGraphVertex *vertex = new DirectedGraphVertex(id);
4     graph->addVertex(vertex);
5     return vertex;
6 }
```

Listing 10: Add a vertex to a graph

A vertex object is created and added to the graph's vertices vector on line 4. This is the instantiation of creating **newEdge()**:

```
1 // Create a new edge instance between two vertices
2 void BuildDirectedGraph::newEdge(DirectedGraphVertex* fromVertex,
     DirectedGraphVertex* toVertex, int weight) {
3     DirectedGraphEdge *edge = new DirectedGraphEdge(toVertex, weight);
4     fromVertex->addNeighbor(edge);
5 }
```

Listing 11: Add a one way edge from one vertex to another in a directed graph

2.2 Directed/Weighted Graph Class

```
 1 #include "AssignmentFour/include/DirectedGraph.h"
 2 #include "AssignmentFour/include/DirectedGraphVertex.h"
 3 #include <vector>
 4 using namespace std;
 5 
 6 // Start with an empty directed graph
 7 DirectedGraph::DirectedGraph() : vertices() {}
 8 
 9 // Return the vertices
10 vector<DirectedGraphVertex*> DirectedGraph::getVertices() {
11     return vertices;
12 }
13 
14 // Add a vertex
15 void DirectedGraph::addVertex(DirectedGraphVertex* vertex) {
16     this->vertices.push_back(vertex);
17 }
```

Listing 12: Directed Graph class

Each **DirectedGraph** object has a vector of vertices. The vector of vertices contains vertex objects that will be elaborated on in the next section. On line 10, the vertices in the graph can be returned in order with the method **getVertices()**to manipulate the graph and its behavior in different algorithms. The **addVertex()** method on line 15 adds a vertex pointer to the private vertices vector in each DirectedGraph instance.

2.3 Directed/Weighted Graph Vertex Class

Vertices in the graph are very powerful. They hold the information of the graph and can be uniquely different from its neighbors and far away vertices.

```
#include "AssignmentFour/include/DirectedGraphVertex.h"
#include <vector>
#include <string>
#include <iostream>
using namespace std;

DirectedGraphVertex::DirectedGraphVertex(const std::string& identifier) : id(
    identifier), weight(), predecessor(), neighbors() {}

// Return the neighbors of a vertex
vector<DirectedGraphEdge*> DirectedGraphVertex::getNeighbors() {
    return this->neighbors;
}

// Add a neighbor to the vertex
void DirectedGraphVertex::addNeighbor(DirectedGraphEdge* edge) {
    this->neighbors.push_back(edge);
}

// Return the vertex predecessor
DirectedGraphVertex* DirectedGraphVertex::getPredecessor() {
    return this->predecessor;
}

// Update the vertex predecessor
void DirectedGraphVertex::setPredecessor(DirectedGraphVertex* predecessor) {
    this->predecessor = predecessor;
}
```

```
28
29 // Return the weight of the shortest path to the vertex
30 long DirectedGraphVertex::getWeight() {
31     return weight;
32 }
33
34 // Change the weight of the shortest path
35 void DirectedGraphVertex::setWeight(long weight) {
36     this->weight = weight;
37 }
38
39 // Return the id of the vertex
40 string DirectedGraphVertex::getId() {
41     return this->id;
42 }
```

Listing 13: Directed Graph Vertex class

Every vertex object in a directed graph must have a unique identifier. Without one, it would be challenging to traverse graphs, as there would be no way to visualize its structure. The optional properties of a vertex object include **weight**, **predecessor**, and **neighbors**. The weight and predecessor properties will be discussed in the Bellman-Ford section. The neighbors vector contains other vertices connected to the current vertex object via edge objects. Edge objects will be explained in the next section. For now, they represent one-way connections to another vertex.

On line 10, the **getNeighbors()** method returns the current edges connected to the vertex object using its vertex pointer. On line 15, the **addNeighbor()** method takes an edge object containing the target vertex, which the current vertex object will then connect to. The **getPredecessor()** and **setPredecessor()** methods on lines 20 and 25 are also used in the Bellman-Ford algorithm to keep track of the path to the start vertex. The **getWeight()** and **setWeight()** methods are used for comparisons in the Bellman-Ford algorithm. Finally, the **getId()** method returns the id of the current vertex object from the **id** property.

2.4 Directed/Weighted Edge Vertex Class

```
1  #include "AssignmentFour/include/DirectedGraphEdge.h"
2  #include "AssignmentFour/include/DirectedGraphVertex.h"
3  using namespace std;
4
5  DirectedGraphEdge::DirectedGraphEdge(DirectedGraphVertex *vertex, const int &
       edgeWeight) : vertex(vertex), edgeWeight(edgeWeight){}
6
7  // Return the next vertex pointer
8  DirectedGraphVertex* DirectedGraphEdge::getVertex() {
9      return this->vertex;
10 }
11
12 // Return the weight of the edge
13 int DirectedGraphEdge::getWeight() {
14     return this->edgeWeight;
15 }
```

Listing 14: Directed Graph Vertex class

The edge class is a simple class that represents the direct one-way connection between one vertex to another in a directed graph. Each object requires that a target vertex is specified and an edge weight is specified. This can be found on *line 5 and 6*. The **getVertex()** method returns the target vertex. The **getWeight()** method returns the weight of the edge from the source vertex to the target vertex. Each instance of an edge object is stored in the neighbors vector of a vertex object. This ensures that there is no confusion of what the target vertex is and what the source vertex is.

3 Bellman-Ford Algorithm

The Bellman-Ford Algorithm is a process that finds the shortest path to all vertices in a graph from a single vertex. This is a complex endeavor that involves traversing the graph multiple times from each vertex and checking its neighbors at each iteration. This is necessary because the edges between vertices have weights, and the **weight** properties in each vertex object represent the current minimum cost to reach that vertex. Therefore, the algorithm seeks the shortest path with the lowest cost to reach that vertex from a starting vertex.

3.1 Reset the Vertex Weights

```
1  // Reset the weights of the vertices in all graphs for the bellman-algorithm
2  void Search::doWeightReset(vector<DirectedGraph*> graphs) {
3      for (int i = 0; i < graphs.size(); i++) {
4          vector<DirectedGraphVertex*> vertices = graphs[i]->getVertices();
5          for (int j = 0; j < vertices.size(); j++) {
6              // Initialize the weights of the vertices
7              if (vertices[j]->getId() == BF_START_VERTEX) {
8                  vertices[j]->setWeight(START_VERTEX_WEIGHT);
9              } else {
10                 vertices[j]->setWeight(ABSURDLY_LARGE_VALUE);
11             }
12         }
13      }
14 }
```

Listing 15: Reset vertex weights for Bellman-Ford algorithm

This weight reset method is responsible for ensuring that all of the vertices in the graph are ready for the Bellman-Ford algorithm. The starting vertex needs to be of weight 0 so the algorithm doesn't constantly loop to infinity. On line 5 we define the vertices in the graph to its own vector of vertices. The conditional inside the loop then checks the id of each vertex. If the vertex id matches the **BF START VERTEX** constant, its weight is set to 0. All other vertices is set to infinity or the value of **ABSURDLY LARGE VALUE.**

3.2 The Algorithm

```
1  // Do a Bellman-Ford algorithm to find all of the shortest paths in a directed
       graph
2  bool Search::findShortestPath(vector<DirectedGraph*> graphs) {
3      bool isUsable = true;
4      // Set the initial weights to all vertices in all graphs
5      doWeightReset(graphs);
6      // Iterate through the graphs
7      for (int i = 0; i < graphs.size(); i++) {
8          vector<DirectedGraphVertex*> vertices = graphs[i]->getVertices();
9          // Iterate to cover maximum path lengths
10         for (int l = 0; l < vertices.size() - 1; l++) {
11             // Iterate vertices to discover updated shortest paths
12             for (int j = 0; j < vertices.size(); j++) {
13                 DirectedGraphVertex* vertex = vertices[j];
14                 vector<DirectedGraphEdge*> edges = vertex->getNeighbors();
15                 // Iterate through the current vertex neighbors
16                 for (int k = 0; k < edges.size(); k++) {
17                     DirectedGraphEdge* edge = edges[k];
18                     relaxEdge(vertex, edge);
19                 }
20             }
21         }
22         // Ensure there are negative loops between vertices
23         for (int j = 0; j < vertices.size(); j++) {
24             DirectedGraphVertex* vertex = vertices[j];
25             vector<DirectedGraphEdge*> edges = vertices[j]->getNeighbors();
26             for (int k = 0; k < edges.size(); k++) {
27                 DirectedGraphEdge* edge = edges[k];
28                 long currentVertexWeight = edge->getVertex()->getWeight();
29                 long outgoingVertexWeight = vertex->getWeight() + edge->
                       getWeight();
30                 // There are remaining edges that will 'never' have a shortest
                       path
31                 if ((currentVertexWeight) > outgoingVertexWeight) {
32                     isUsable = false;
33                 }
34             }
35         }
36 
37     }
38     print.printBellmanFord(graphs);
39     return isUsable;
40 }
```

Listing 16: Find lowest cost to each vertex

This method is the meat and potatoes of the Bellman-Ford algorithm. The method **findShortestPath()**, takes in a vector of graphs to do the algorithm on. After the graphs have been reset on line 5, it loops through each of the graphs and then loops through each graphs edges. This is represented by **vertices.size() - 1**. This is to ensure that the algorithm completes the correct number of times and the lowest path costs to all vertices are found. Then, we loop through each vertex in the graph and begin checking its neighbors on line 14. The **relaxEdge()** is called to lower the weight to get to a vertex if the current vertex is a lower value than one of its neighbors. This represents that there is a new lower cost path found. The next set of nested loops iterates through all vertices and their edges once more to verify the absence of any negative weight cycles. A negative weight cycle can be detected if a shorter path is discovered after the initial phase of the algorithm has been completed. This condition is identified on *line 31*. The current neighbor being examined has a higher weight than the outgoing vertex, indicating the existence of a shorter path. However, since this condition was found within this section of the algorithm, a negative weight cycle that will never terminate is identified. The paths are then printed on line 38 using a method from the PrintDirectedGraph class denoted by the print variable.

3.3 Relaxing an Edge

```
void Search::relaxEdge(DirectedGraphVertex* vertex, DirectedGraphEdge* edge) {
    long currentVertexWeight = edge->getVertex()->getWeight();
    long outgoingVertexWeight = vertex->getWeight() + edge->getWeight();
    // There is a new shortest path to the outgoing vertex
    if (currentVertexWeight > outgoingVertexWeight) {
        edge->getVertex()->setWeight(outgoingVertexWeight);
        edge->getVertex()->setPredecessor(vertex);
    }
  }
```

Listing 17: Relax an Edge

This method compares the outgoing vertex weight (source) with the destination (current) vertex weight. This declarations are derived from the vertex class and edge class respectively. The conditional is found on line 5. If the current vertex weight is greater than the outgoing vertex weight plus the weight of the edge, there is a shorter path if the path is through the outgoing vertex. Therefore, the weight is adjusted for the current vertex to reflect the original outgoing vertex weight plus the weight of the edge. In addition, the current vertex sets the outgoing vertex as its new predecessor. In other words, there is a pointer set to the previous vertex.

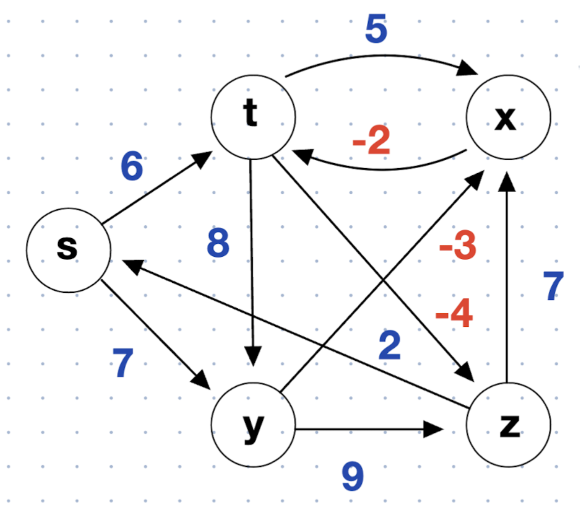

Figure 1: A directed graph

The visualization above represents an example directed graph. The circles represent vertex objects, the arrows and vertices at the ends of each arrow represent edge objects, and the numbers on the edges indicate their weights. An arrow pointing away from a vertex indicates that the edge is only one-way.

4 Conclusion

This assignment, the culmination of learning the fundamentals of data structures and algorithms in computer science and mathematics, has not only honed my programming skills for efficiency but also introduced me to a completely new language in less than four months. With each project, my syntax and structure improved, culminating in a profound understanding of the inner workings of programming in C++.

I firmly believe that this assignment has equipped me with the skills to program numerous additional algorithms into my repository in the future. My proficiency in C++, coupled with my problem-solving abilities to break down complex problems into manageable components, has been instrumental in this development. Until now, I lacked the necessary knowledge to analyze, enhance, and create future algorithms for my professional career. However, this submission demonstrates that I have acquired the requisite expertise. Notably, I efficiently completed the assignment well before the deadline. While I acknowledge that further improvements can be made, the final outcome meets the assignment's requirements.

With this assignment, I continued to leverage AI to enhance my productivity. However, this assignment was the first where I felt confident in my ability to debug my code without seeking assistance from LLM's. I noticed a significant improvement in my writing skills by utilizing the proofreading features introduced by Apple Intelligence recently. I can highlight text I've written, and the tool can either proofread or rewrite it using the words I've already typed.

I'm thrilled to delve deeper into the algorithms bible and implement new features into this code. I envision this repository becoming a valuable resource for my professional, personal, and academic endeavors in future projects and research. I appreciate the clear instructions and the ability to extract crucial details that aided me in translating language into code. While I've reached the end of this assignment, I recognize that this is merely the beginning of my journey. I'm eager to apply the skills I've acquired to enhance my code and utilize them for future projects I've envisioned.